Doctoral Thesis

# Design and Evaluation of Low-Latency Communication Middleware on High Performance Computing Systems

*Roberto Rey Expósito*

*2014*

UNIVERSIDADE DA CORUÑA

# Design and Evaluation of Low-Latency Communication Middleware on High Performance Computing Systems

Roberto Rey Expósito

Doctoral Thesis

July 2014

PhD Advisors:

Guillermo López Taboada
Juan Touriño Domínguez

PhD Program in Information Technology Research

UNIVERSIDADE DA CORUÑA

| | |
|---|---|
| Dr. Guillermo López Taboada | Dr. Juan Touriño Domínguez |
| Profesor Contratado Doctor | Catedrático de Universidad |
| Dpto. de Electrónica y Sistemas | Dpto. de Electrónica y Sistemas |
| Universidade da Coruña | Universidade da Coruña |

CERTIFICAN

Que la memoria titulada "*Design and Evaluation of Low-Latency Communication Middleware on High Performance Computing Systems*" ha sido realizada por D. Roberto Rey Expósito bajo nuestra dirección en el Departamento de Electrónica y Sistemas de la Universidade da Coruña, y concluye la Tesis Doctoral que presenta para optar al grado de Doctor en Ingeniería Informática con la Mención de Doctor Internacional.

En A Coruña, a 29 de Julio de 2014

| | |
|---|---|
| Fdo.: Guillermo López Taboada | Fdo.: Juan Touriño Domínguez |
| Director de la Tesis Doctoral | Director de la Tesis Doctoral |

Fdo.: Roberto Rey Expósito
Autor de la Tesis Doctoral

*A Alba*
*por tanto*

*A mis padres, familia y amigos*
*por soportarme*

# Acknowledgments

First of all, I would especially like to acknowledge my PhD advisors Guillermo and Juan for their valuable help and guidance during all these years. It has been an honor being their PhD student. Next, I will always be grateful to Alba, who has always been there for me, for her unconditional love and care. There are just no words to describe what her presence means to me. Of course, I cannot forget my parents, who have taught me about hard work, self-respect, persistence and how to be independent. I am also greatly indebted to Gore and Luci for their constant affection, as well as to all my loving family and closest friends, especially Gaby and Nando, for all the good times. Huge thanks also to all my past and present colleagues in the GAC group for their kindness, both the ones from the FIC and those from the CITIC. I would like to mention Iván, Sabela, Andión, Toño, Juan, Jacobo, Moisés, Jorge, Darriba, Andrés, Dyer, Óscar, Dani, Santorum and Sasi.

I would like to sincerely thank Dr. Heinz Kredel and his group for hosting me during my three-month research visit to the Rechenzentrum Universität Mannheim (Germany). Moreover, I am grateful to the following institutions for providing access to the clusters and supercomputers that have been used as testbeds for this work: CESGA (Galicia Supercomputing Center), BSC (Barcelona Supercomputing Center), HLRS (High Performance Computing Center Stuttgart), the Vrije University Amsterdam and the Phylogenomics Group of the University of Vigo.

Finally, I want to acknowledge the following institutions and projects for funding this work: the Computer Architecture Group, the Department of Electronics and Systems and the University of A Coruña for the human and material support, and for funding my attendance at some conferences; the Galician Government (Xunta de Galicia) for the Consolidation Program of Competitive Research Groups (Computer

Architecture Group, refs. GRC2013/055 and 2010/6) and the Galician Network of High Performance Computing (ref. 2010/53); the Open European Network for High Performance Computing on Complex Environments (ComplexHPC, COST Action ref. IC0805) for providing financial support to attend the ComplexHPC Spring School; the Barrié Foundation for the project HPC4HPT; Amazon Web Services (AWS) for providing access to the Amazon EC2 cloud through a research grant; INDITEX S.A. and the University of A Coruña for a joint grant to make a research visit to the Rechenzentrum Universität Mannheim; the Ministry of Economy and Competitiviness and the Ministry of Education, Culture and Sport of Spain for the project TIN2010-16735 and the FPU grant AP2010-4348.

*Roberto Rey Expósito*

*Life, Liberty and the pursuit of Happiness*

*(La vida, la libertad y la búsqueda de la felicidad)*

*United States Declaration of Independence*

# Resumo

O interese en Java para computación paralela está motivado polas súas interesantes características, tales como o seu apoio multithread, portabilidade, facilidade de aprendizaxe, alta produtividade e o aumento significativo no seu rendemento computacional. No entanto, as aplicacións paralelas en Java carecen xeralmente de mecanismos de comunicación eficientes, os cales adoitan usar protocolos baseados en sockets que son incapaces de obter o máximo proveito das redes de baixa latencia, obstaculizando a adopción de Java na computación de altas prestacións (*High Performance Computing*, HPC). Esta Tese de Doutoramento presenta o deseño, implementación e avaliación de solucións de comunicación en Java que superan esta limitación. En consecuencia, desenvolvéronse múltiples dispositivos de comunicación a baixo nivel para paso de mensaxes en Java (*Message-Passing in Java*, MPJ) que aproveitan ao máximo o hardware de rede subxacente mediante operacións de acceso directo a memoria remota que proporcionan comunicacións de baixa latencia. Tamén se inclúe unha biblioteca de paso de mensaxes en Java totalmente funcional, FastMPJ, na cal foron integrados os dispositivos de comunicación. A avaliación experimental amosou que as primitivas de comunicación de FastMPJ son competitivas en comparación con bibliotecas nativas, aumentando significativamente a escalabilidade de aplicacións MPJ. Por outra banda, esta Tese analiza o potencial da computación na nube (*cloud computing*) para HPC, onde o modelo de distribución de infraestrutura como servizo (*Infrastructure as a Service*, IaaS) xorde como unha alternativa viable aos sistemas HPC tradicionais. A ampla avaliación do rendemento de recursos cloud específicos para HPC do proveedor líder, Amazon EC2, puxo de manifesto o impacto significativo que a virtualización impón na rede, impedindo mover as aplicacións intensivas en comunicacións á nube. A clave atópase no soporte de virtualización apropiado, como o acceso directo ao hardware de rede, xunto coas directrices para a optimización do rendemento suxeridas nesta Tese.

# Resumen

El interés en Java para computación paralela está motivado por sus interesantes características, tales como su soporte multithread, portabilidad, facilidad de aprendizaje, alta productividad y el aumento significativo en su rendimiento computacional. No obstante, las aplicaciones paralelas en Java carecen generalmente de mecanismos de comunicación eficientes, los cuales utilizan a menudo protocolos basados en sockets incapaces de obtener el máximo provecho de las redes de baja latencia, obstaculizando la adopción de Java en computación de altas prestaciones (*High Performance Computing*, HPC). Esta Tesis Doctoral presenta el diseño, implementación y evaluación de soluciones de comunicación en Java que superan esta limitación. En consecuencia, se desarrollaron múltiples dispositivos de comunicación a bajo nivel para paso de mensajes en Java (*Message-Passing in Java*, MPJ) que aprovechan al máximo el hardware de red subyacente mediante operaciones de acceso directo a memoria remota que proporcionan comunicaciones de baja latencia. También se incluye una biblioteca de paso de mensajes en Java totalmente funcional, FastMPJ, en la cual se integraron los dispositivos de comunicación. La evaluación experimental ha mostrado que las primitivas de comunicación de FastMPJ son competitivas en comparación con bibliotecas nativas, aumentando significativamente la escalabilidad de aplicaciones MPJ. Por otro lado, esta Tesis analiza el potencial de la computación en la nube (*cloud computing*) para HPC, donde el modelo de distribución de infraestructura como servicio (*Infrastructure as a Service*, IaaS) emerge como una alternativa viable a los sistemas HPC tradicionales. La evaluación del rendimiento de recursos cloud específicos para HPC del proveedor líder, Amazon EC2, ha puesto de manifiesto el impacto significativo que la virtualización impone en la red, impidiendo mover las aplicaciones intensivas en comunicaciones a la nube. La clave reside en un soporte de virtualización apropiado, como el acceso directo al hardware de red, junto con las directrices para la optimización del rendimiento sugeridas en esta Tesis.

# Abstract

The use of Java for parallel computing is becoming more promising owing to its appealing features, particularly its multithreading support, portability, easy-to-learn properties, high programming productivity and the noticeable improvement in its computational performance. However, parallel Java applications generally suffer from inefficient communication middleware, most of which use socket-based protocols that are unable to take full advantage of high-speed networks, hindering the adoption of Java in the High Performance Computing (HPC) area. This PhD Thesis presents the design, development and evaluation of scalable Java communication solutions that overcome these constraints. Hence, we have implemented several low-level message-passing devices that fully exploit the underlying network hardware while taking advantage of Remote Direct Memory Access (RDMA) operations to provide low-latency communications. Moreover, we have developed a production-quality Java message-passing middleware, FastMPJ, in which the devices have been integrated seamlessly, thus allowing the productive development of Message-Passing in Java (MPJ) applications. The performance evaluation has shown that FastMPJ communication primitives are competitive with native message-passing libraries, improving significantly the scalability of MPJ applications. Furthermore, this Thesis has analyzed the potential of cloud computing towards spreading the outreach of HPC, where Infrastructure as a Service (IaaS) offerings have emerged as a feasible alternative to traditional HPC systems. Several cloud resources from the leading IaaS provider, Amazon EC2, which specifically target HPC workloads, have been thoroughly assessed. The experimental results have shown the significant impact that virtualized environments still have on network performance, which hampers porting communication-intensive codes to the cloud. The key is the availability of the proper virtualization support, such as the direct access to the network hardware, along with the guidelines for performance optimization suggested in this Thesis.

# Preface

The performance and scalability of inter-node communications are key aspects for running High Performance Computing (HPC) applications efficiently on parallel architectures. In fact, current HPC systems, from multi-core clusters to large supercomputers, are aggregating a significant number of cores interconnected via a high-speed network such as InfiniBand, RoCE or high-speed Ethernet (10/40 Gigabit). Furthermore, the advent of cloud computing [15, 72] has generated considerable interest in the HPC community due to the high availability of computational resources at large scale. Thus, public Infrastructure as a Service (IaaS) providers (e.g., Amazon) are increasingly offering virtualized HPC resources allowing end users to set up virtual clusters to exploit supercomputing-level power. In this context, cloud computing platforms have become an interesting alternative to deploy an HPC system without any knowledge of the underlying infrastructure.

The continuously increasing number of cores available in the current multi- and many-core era underscores the need for scalable parallel solutions, where the efficiency of the underlying communication middleware is fundamental. In this context, it is key to fully harness the power of the likely abundant processing resources from HPC systems while taking advantage of the interesting features of high-speed networks, such as Remote Direct Memory Access (RDMA) operations, with still easy-to-use programming models. The Message-Passing Interface (MPI) [61] remains as the de-facto standard in the area of parallel computing owing to its flexibility and portability, being able to achieve high performance in very different systems. Thus, MPI is still the most widely extended programming model for writing parallel applications on HPC systems, traditionally using natively compiled languages (e.g., C/C++, Fortran).

Java is currently among the preferred programming languages in web-based and distributed computing environments, and it has become a valuable alternative for parallel computing [73, 79]. The interest in Java for HPC is based on its appealing characteristics that can be of special benefit for parallel programming such as built-in networking and multithreading support, object orientation, automatic memory management, portability, easy-to-learn properties and thus high programming productivity. Moreover, the significant improvement in its computational performance has narrowed the performance gap between Java and natively compiled languages, thanks to the use of efficient Just-in-Time (JIT) compilers. However, although this performance gap is usually small for sequential applications, it can be particularly large for parallel applications when depending on communications performance. The main reason is that current parallel Java applications generally suffer from inefficient communication middleware that do not take full advantage of high-speed networks [44]. This lack of efficient communication support in current Message-Passing in Java (MPJ) [18] implementations usually results in lower performance than MPI, which has hindered the use of Java in this area.

The present PhD Thesis, "Design and Evaluation of Low-Latency Communication Middleware on High Performance Computing Systems", has been structured to accomplish a twofold purpose. On the one hand, it focuses on providing a more efficient Java communication middleware for parallel computing that overcomes the performance constraints discussed above by fully exploiting the underlying networking hardware, enabling low-latency and high-bandwidth inter-node communications for parallel Java applications. This goal has been tackled specifically in the first part of the Thesis, particularly for clusters and supercomputers as they are currently the most widespread HPC deployments. On the other hand, the second part focuses on the use of public cloud infrastructures for HPC and scientific computing. Hence, the scalability of the Java communication middleware developed in the previous part has been analyzed on the most popular IaaS cloud provider: Amazon EC2 [2]. Additionally, a comprehensive feasibility study of using Amazon EC2 resources for HPC has been carried out. This extensive review ranges from the identification of the main causes of communication inefficiency on a cloud computing environment up to the proposal of some techniques for reducing their impact on the scalability of communication-intensive HPC codes, both for Java and natively compiled languages. This study also considers other important aspects of Amazon EC2 to be a viable alternative in the HPC area, such as providing high-performance file I/O through Solid State Drive (SSD) disks, the performance characterization of parallel/distributed file systems for data-intensive applications (e.g., Big Data workloads) or the use of coprocessors, such as Graphics Processing Units (GPU), as many-core accelerators [56].

## Work Methodology

The Thesis methodology has followed a classical approach in research and engineering: analysis, design, implementation and evaluation. Thus, the work of this Thesis has started with the feasibility analysis of providing efficient Java communications for HPC. For this task, a proof-of-concept low-level communication device was implemented and integrated in a production Java communication middleware. This communication device provides native support for an InfiniBand network, selected for being widely used in current HPC systems. The feedback obtained from this initial task was used to define and structure the subsequent developments, where one of the most relevant contributions was the implementation of FastMPJ, a production-quality Java message-passing middleware. The communication support of this modular middleware was built upon several low-level communication devices that take full advantage of high-speed networks. Therefore, the communication device for InfiniBand was integrated in FastMPJ, while additional low-level communication devices were implemented to extend its high-speed network support. An important task was the evaluation of this middleware on representative clusters and supercomputers, as well as the analysis of the state of the art regarding the use of Java for HPC. Next, this Java communication middleware was evaluated on a public cloud computing infrastructure, Amazon EC2, using HPC-aimed resources, both in terms of performance and cost. As the popularity of public clouds has increased significantly in the last years, another task was to evaluate Amazon EC2 resources for running HPC applications. Amazon EC2 was selected as it is currently the most popular IaaS provider, offering several cloud resources that are intended to be well suited for HPC: cluster instances equipped with powerful multi-core CPUs, high-speed networks (10 Gigabit Ethernet), multi-GPU cluster instances and instances that provide SSD-based disks as local storage.

## Structure of the Thesis

In accordance with the current regulations of the University of A Coruña, the PhD dissertation has been structured as a compilation Thesis (i.e., based on merging research articles). Particularly, this research work comprises nine JCR-indexed journal papers which have been organized into two different parts. The Thesis begins with the *Introduction* chapter, intended to give the reader a general overview of all the research carried out in these papers. First, this chapter introduces the scope and main motivations of the Thesis in order to delimit its context and provides a clear description of the main objectives to be achieved. Next, it includes an overall discussion of the main research results of each part in order to provide consistency and coherence to the different works.

Hereafter, the Thesis is divided into two parts. In the first part (*Design of Low-Latency Java Communication Middleware on High-Speed Networks*), we present the design, implementation and evaluation of efficient Java communication middleware for parallel computing. The journal papers derived from the first part, each one presented as a separate chapter (Chapters 2-5), are the following:

- R. R. Expósito, G. L. Taboada, J. Touriño, and R. Doallo. Design of scalable Java message-passing communications over InfiniBand. *Journal of Supercomputing*, 61(1):141–165, 2012.
- R. R. Expósito, S. Ramos, G. L. Taboada, J. Touriño, and R. Doallo. FastMPJ: a scalable and efficient Java message-passing library. *Cluster Computing*, 2014. (in press, http://dx.doi.org/10.1007/s10586-014-0345-4).
- R. R. Expósito, G. L. Taboada, S. Ramos, J. Touriño, and R. Doallo. Low-latency Java communication devices on RDMA-enabled networks. 2014. (Submitted for journal publication).
- G. L. Taboada, S. Ramos, R. R. Expósito, J. Touriño, and R. Doallo. Java in the high performance computing arena: research, practice and experience. *Science of Computer Programming*, 78(5):425–444, 2013.

The second part of the Thesis (*Evaluation of Communication Middleware for HPC on a Public Cloud Infrastructure*) presents a detailed feasibility study of the use of a public cloud computing platform, Amazon EC2, for running HPC applications. The journal papers derived from the second part, each one also presented as a separate chapter (Chapters 6-10), are the following:

- R. R. Expósito, G. L. Taboada, S. Ramos, J. Touriño, and R. Doallo. Evaluation of messaging middleware for high-performance cloud computing. *Personal and Ubiquitous Computing*, 17(8):1709–1719, 2013.
- R. R. Expósito, G. L. Taboada, S. Ramos, J. Touriño, and R. Doallo. Performance analysis of HPC applications in the cloud. *Future Generation Computer Systems*, 29(1):218–229, 2013.
- R. R. Expósito, G. L. Taboada, S. Ramos, J. González-Domínguez, J. Touriño, and R. Doallo. Analysis of I/O performance on an Amazon EC2 cluster compute and high I/O platform. *Journal of Grid Computing*, 11(4):613–631, 2013.
- R. R. Expósito, G. L. Taboada, S. Ramos, J. Touriño, and R. Doallo. Performance evaluation of data-intensive computing applications on a public IaaS cloud. 2014. (Submitted for journal publication).
- R. R. Expósito, G. L. Taboada, S. Ramos, J. Touriño, and R. Doallo. General-purpose computation on GPUs for high performance cloud computing. *Concurrency and Computation: Practice and Experience*, 25(12):1628–1642, 2013.

Finally, the *Conclusions and Future Work* chapter summarizes the main contributions of the Thesis and outlines the main research lines that can be derived from this work.

## Funding and Technical Means

The necessary means to carry out this Thesis have been the following:

- Working material, human and financial support primarily by the Computer Architecture Group of the University of A Coruña, along with a Research

Fellowship funded by the Ministry of Education, Culture and Sport of Spain (FPU grant AP2010-4348).

- Access to bibliographical material through the library of the University of A Coruña.

- Additional funding through the following research projects:

  - European funding: project "Open European Network for High Performance Computing on Complex Environments" (ComplexHPC, COST Action ref. IC0805).
  - State funding by the Ministry of Economy and Competitiviness of Spain through the project "Architectures, Systems and Tools for High Performance Computing" (TIN2010-16735).
  - Regional funding by the Galician Government (Xunta de Galicia) under the Consolidation Program of Competitive Research Groups (Computer Architecture Group, refs. GRC2013/055 and 2010/6) and Galician Network of High Performance Computing (ref. 2010/53).
  - Private funding: projects "High Performance Computing for High Performance Trading (HPC4HPT)" funded by the Barrié Foundation and "F-MPJ-Cloud (Fast Message-Passing in Java on the Cloud)" funded by a research grant of Amazon Web Services (AWS) LLC.

- Access to clusters, supercomputers and cloud computing platforms:

  - *Pluton* cluster (Computer Architecture Group, University of A Coruña, Spain). Initially, 16 nodes with 2 Intel Xeon quad-core Nehalem-EP processors and up to 16 GB of memory, all nodes interconnected via InfiniBand DDR and 2 of them via 10 Gigabit Ethernet. Additionally, two nodes with one Intel Xeon quad-core Sandy Bridge-EP processor and 32 GB of memory, interconnected via InfiniBand FDR, RoCE and iWARP, and four nodes with one Intel Xeon hexa-core Westmere-EP processor, 12 GB of memory and 2 GPUs NVIDIA Tesla "Fermi" 2050 per node, interconnected via InfiniBand QDR. Later, 16 nodes have been added, each of them with 2 Intel Xeon octa-core Sandy Bridge-EP processors,

64 GB of memory and 2 GPUs NVIDIA Tesla "Kepler" K20m per node, interconnected via InfiniBand FDR.

- *DAS-4* cluster (Advanced School for Computing and Imaging, ASCI, Vrije University Amsterdam, the Netherlands). For the experiments of the Thesis, 64 nodes have been used, each of them with 2 Intel Xeon quad-core Westmere-EP processors and 24 GB of memory, interconnected via InfiniBand QDR. Furthermore, we have used one fat node with 4 AMD Opteron twelve-core Magny-Cours processors and 128 GB of memory.
- *XB5* cluster (Phylogenomics Group, University of Vigo, Spain). One fat node with 4 Intel Xeon ten-core Westmere-EX processors and 512 GB of memory has been used.
- *Finis Terrae* supercomputer (Galicia Supercomputing Center, CESGA, Spain): 144 nodes with 8 Intel Itanium-2 dual-core Montvale processors and 128 GB of memory, interconnected via InfiniBand DDR. Additionally, we have used one Superdome system with 64 Intel Itanium-2 dual-core Montvale processors and 1 TB of memory.
- *MareNostrum* supercomputer (Barcelona Supercomputing Center, BSC, Spain): 2560 nodes with 2 IBM PowerPC dual-core 970MP processors and 8 GB of memory, interconnected via Myrinet 2000.
- *Hermit* supercomputer (High Performance Computing Center Stuttgart, HLRS, Germany): 3552 nodes each of them with 2 AMD Opteron 16-core Interlagos processors and up to 64 GB of memory, interconnected via the custom Cray Gemini interconnect with a 3D torus topology.
- *Amazon EC2* IaaS cloud platform (Amazon Web Services, AWS). Several instance types have been used: (1) CC1, 2 Intel Xeon quad-core Nehalem-EP processors, 23 GB of memory and 2 local storage disks per instance; (2) CC2, 2 Intel Xeon octa-core Sandy Bridge-EP processors, 60.5 GB of memory and 4 local storage disks per instance; (3) CG1, 2 Intel Xeon quad-core Nehalem-EP processors, 22 GB of memory, 2 GPUs NVIDIA Tesla "Fermi" 2050 and 2 local storage disks per instance; (4) HI1, 2 Intel Xeon quad-core Westmere-EP processors, 60.5 GB of memory and 2 SSD-based local storage disks per instance; (5) CR1, 2 Intel Xeon octa-core Sandy Bridge-EP processors, 244 GB of memory and 2 SSD-based

local storage disks per instance; and (6) HS1, 1 Intel Xeon octa-core Sandy Bridge-EP processor, 117 GB of memory and 24 local storage disks per instance. All these instance types are interconnected via 10 Gigabit Ethernet.

- A three-month research visit to the Rechenzentrum Universität Mannheim, Germany, which has allowed the access to the *Hermit* supercomputer installed at the High Performance Computing Center Stuttgart (HLRS) for implementing the corresponding FastMPJ commmunication device for the network support of the Cray XE/XK/XC family of supercomputers. This research visit was funded by INDITEX S.A. in collaboration with the University of A Coruña through a competitive INDITEX-UDC grant obtained in 2013.

## Main Contributions

The main contributions related to the first part of the Thesis are:

1. Design and implementation of the `ibvdev` low-level communication library, which is a Java message-passing device implemented on top of the Verbs interface that provides scalable communications on InfiniBand systems [38].

2. Design, implementation and evaluation of an efficient Java message-passing library: FastMPJ. This middleware not only benefits from the integration of `ibvdev`, but also from two new low-level communication devices: (1) `mxdev`, implemented on top of MX/Open-MX for communication on Myrinet and generic Ethernet hardware; and (2) `psmdev`, implemented on top of InfiniPath PSM for the support of the InfiniPath family of InfiniBand adapters from Intel/QLogic [31].

3. Design, implementation and optimization of Java communication devices on RDMA-enabled networks. This work includes the design and implementation of two new communication devices, `ugnidev` and `mxmdev`, which have been integrated into FastMPJ. The former device is intended to provide efficient support for the RDMA networks used by Cray supercomputers. The latter

includes support for the recently released messaging library developed by Mellanox for its RDMA adapters. Furthermore, an enhanced version of `ibvdev`, which includes additional support for RDMA networks along with an optimized short-message communication protocol, has been provided [36].

4. An up-to-date review of Java for HPC, which includes an extensive performance evaluation that focuses on message passing due to its scalability and extended use in HPC. Thus, FastMPJ is evaluated comparatively with representative MPI and MPJ middleware. This review also analyzes the current state of Java for HPC both for shared and distributed memory programming, showing an important number of past and present projects which are the result of the sustained interest in the use of Java in this area [79].

The main contributions related to the second part of the Thesis are:

1. A detailed study of the impact of the virtualized network overhead on the scalability of HPC applications on the leading public IaaS cloud: Amazon EC2. This work compares the first generation of the HPC-aimed family of EC2 cluster instances, whose access to its high-speed network (i.e., 10 Gigabit Ethernet) is paravirtualized, with the performance of a similar private cloud testbed that supports the direct access to the same EC2 networking technology through the PCI passthrough technique and the same testbed running on a non-virtualized environment [33].

2. A thorough analysis of the main performance bottlenecks in HPC application scalability on Amazon EC2 along with the proposal and evaluation of some techniques to reduce the impact of the virtualized network overhead. This work gives more insight into the performance of running HPC applications on Amazon EC2, using an important number of cores (up to 512) and comparing the first and second HPC-aimed generations of cluster instances in terms of single instance performance, scalability and cost-efficiency, as well as taking into account hybrid programming models (e.g., MPI+OpenMP) [35].

3. A comprehensive evaluation of the I/O storage subsystem on Amazon EC2. This work evaluates both generations of cluster instances along with storage-optimized instances, which provide SSD-based disks, in order to determine

their suitability for I/O-intensive applications. The evaluation has been carried out at different layers, ranging from the cloud low-level storage devices (e.g., ephemeral disks) and I/O interfaces (e.g., MPI-IO, HDF5) up to the application level, including also an analysis in terms of cost. Furthermore, the Network File System (NFS) performance has been characterized, showing the impact of the main NFS configuration parameters on a virtualized cloud environment [32].

4. A performance analysis of running data-intensive computing applications on Amazon EC2. This work analyzes the full I/O software stack for data-intensive computing, both for HPC and Big Data workloads. More specifically, the performance and cost-efficiency of four instance types that provide 10 Gigabit Ethernet has been characterized at multiple layers, using a representative suite of benchmarking tools (e.g., IOzone, IOR, Intel Hibench), parallel/distributed file systems (OrangeFS, HDFS), distributed computing frameworks (Apache Hadoop), I/O-intensive parallel codes for HPC (e.g., FLASH-IO) and MapReduce-based workloads for Big Data processing (e.g., Sort, PageRank) [37].

5. A feasibility study of using heterogeneous architectures with many-core accelerators on Amazon EC2. Thus, the GPU family of EC2 cluster instances has been assessed using representative Compute Unified Device Architecture (CUDA) and Open Computing Language (OpenCL) synthetic kernels and benchmarks, and their results have been compared with a non-virtualized GPU environment in order to analyze the virtualization overhead for GPGPU. Furthermore, hybrid parallel codes (e.g., MPI+CUDA) have been also taken into account at the application level, using two real-world applications and the High Performance Linpack (HPL) benchmark [34].

# Contents

# List of Tables

# List of Figures

# Chapter 1

# Introduction

This introductory chapter is intended to give the reader a brief summary of the whole research carried out in this Thesis. The structure of this chapter is as follows: Section 1.1 introduces the scope and main motivations of the Thesis in order to delimit its context. A clear description of the main objectives that must be achieved is included in Section 1.2. Finally, Section 1.3 provides an overall discussion of the main research results in order to provide consistency and coherence to the different journal papers that compose this work.

## 1.1. Scope and Motivation

High Performance Computing (HPC) has become increasingly important over the last decades as an essential tool for scientific and industry research. In fact, HPC is currently one of the leading edge disciplines in Information Technology (IT) with a wide range of demanding applications in economy, science, and engineering. These applications usually involve the construction of mathematical models and numerical solution techniques that often require a huge number of computing resources to perform large scale experiments or to cut down the computational complexity into a reasonable time frame. These computational needs have been tipically addressed by supercomputers installed at national laboratories, big research groups or large companies. Despite the significance of HPC towards scientific research and industrial growth, only the largest research projects are able to afford expensive supercomputers. Moreover, the access to these systems is usually highly restricted and can only be obtained by an allocation through competitive research grant proposals. Hence, for small or medium-scale users with emerging, but limited, HPC demands, the conventional option has been to set up their own in-house HPC cluster.

In order to meet the computational power demands of HPC applications, current deployments are increasing significantly the number of cores installed, interconnected via high-speed networks. This trend in hardware evolution increases the complexity of the system, which demands scalable communication middleware and programming languages with high productivity in parallel code development. For performance reasons, those languages compiled to the native code of each platform (e.g., C/C++, Fortran) have been traditionally the most widely used for writing parallel applications. However, the interest in Java for parallel programming has been rising during the last years thanks to the noticeable improvement in its computational performance and its inherent productivity features. Nevertheless, even though Java is usually the preferred programming language in web-based and distributed computing environments, its presence in the HPC area is still reduced. The main reason that has hindered the adoption of Java is the lack of efficient Java communication middleware, most of which use socket-based protocols that are unable to take full advantage of high-speed networks.

Moreover, setting up a dedicated infrastructure for HPC is a complex endeavor that requires a long lead time, high capital expenditure, and large operational costs. These entry barriers have restricted HPC to a small number of significant users. A cheap, fast, and effective alternative could tremendously spread the outreach of HPC to new users. Recently, the emergence of cloud computing brings new possibilities for constructing and using HPC platforms. Public clouds (e.g., Amazon EC2 [2]) have emerged as a promising alternative to clusters and supercomputers, in which resources are no longer hosted by the researcher's computational facilities, but leased from big data centers only when needed. This is especially interesting for users who cannot afford their own HPC infrastructure due to the limitations mentioned earlier. In fact, cloud computing can provide HPC users with infrastructures at cheaper price compared to dedicated infrastructures, since they benefit from economy of scale and also multiple users sharing resources, which results in improved utilization.

In this context, the present PhD Thesis is composed of two parts in order to accomplish a twofold purpose. On the one hand, the first part of the Thesis (*Design of Low-Latency Java Communication Middleware on High-Speed Networks*) arises from the main goal of improving Java communications performance on traditional HPC systems (i.e., clusters and supercomputers) in order to increase the adoption of this language for parallel computing. The availability of an efficient Java communication middleware would allow parallel programmers to benefit from its appealing features at a reasonable overhead, thus increasing their programming productivity. On the other hand, the second part (*Evaluation of Communication Middleware for HPC on a Public Cloud Infrastructure*) focuses on the feasibility of using a public cloud platform, Amazon EC2, for HPC and scientific computing. This study includes not only the evaluation of our Java communication middleware in the cloud, but also provides an extensive review that considers other important aspects of the Amazon cloud to be a viable alternative in the HPC area. The specific motivations of each part of the Thesis are discussed next.

### 1.1.1. Java Communications for HPC

The growing computational power requirements of HPC applications have been traditionally addressed by using dedicated HPC infrastructures such as multi-core clusters or custom supercomputers, which are currently the most widely extended parallel architectures in the TOP500 list [89]. Most of these systems are increasingly aggregating a significant number of cores interconnected via high-speed networks. This current trend to increase the number of cores per processor underscores the importance of efficient and scalable communication middleware along with highly productive programming languages, in order to fully exploit the features of the abundant processing resources and the underlying networking hardware. Current networking technologies typically rely on scalable topologies and advanced network adapters that allow for low-latency and high-bandwidth communications, usually providing interesting features such as Remote Direct Memory Access (RDMA) operations that enable zero-copy and kernel-bypass facilities. Examples of popular high-speed networks widely extended in HPC environments are InfiniBand (IB) [47], Myrinet, high-speed Ethernet (10/40 Gigabit), RDMA over Converged Ethernet (RoCE) [46], Internet Wide Area RDMA Protocol (iWARP) [70] or the Gemini interconnect [1].

The performance of inter-node communications on top of high-speed networks can have a dramatic impact on the overall scalability of communication-intensive parallel applications. Hence, this scenario requires scalable parallel solutions, where communication efficiency is fundamental. This efficiency not only depends heavily on the use of high-speed networks [59], but also on the communication middleware [91]. Therefore, it is fundamental to fully harness the power of the likely abundant processing resources while taking full advantage of high-speed networks with still easy-to-use programming models. The Message-Passing Interface (MPI) [61] is the de-facto standard in the area of parallel computing owing to its flexibility and portability, being able to achieve high performance in very different systems. MPI has been the preferred choice for writing parallel applications on distributed memory systems for a long time, traditionally using natively compiled languages such as C/C++ or Fortran. With the advent of the many-core era, MPI remains as the most widely extended programming model for communication between nodes within parallel applications, but it is usually combined with other popular programming models, such as Compute Unified Device Architecture (CUDA) [64] or Open Computing Language (OpenCL) [77], in order to enable the use of the increasingly widespread coprocessors, such as Graphics Processing Units (GPUs), as many-core HPC accelerators [56].

Java is a highly portable and flexible programming language that enjoys a dominant position in a wide diversity of computing environments. In fact, Java has become the leading programming language both in academia and industry due to its appealing features that make it a highly productive language. Although some early works have identified its potential for scientific computing soon after its release [27, 40, 88], Java was severely criticized for its poor computational performance in its beginnings [12]. However, the performance gap between Java and natively compiled languages has been narrowing for the last years. The Java Virtual Machine (JVM) is currently equipped with efficient Just-in-Time (JIT) compilers that can obtain near-native performance from the platform-independent bytecode [78]. In fact, the JVM identifies sections of the code frequently executed and converts them to native machine code instead of interpreting the bytecode. Therefore, Java is currently gaining popularity in other domains which usually make use of HPC infrastructures, such as the area of parallel computing [73, 79] and Big Data analytics, where the Java-based Hadoop distributed computing framework [86] is among the preferred choices for the development of applications that follow the MapReduce programming model [25]. Additionally, Java provides some appealing properties that can be of special benefit for parallel programming, especially its built-in networking and multithreading support in the core of the language. These interesting features along with the continuous increase in the performance of the JVM and its other traditional advantages for general programming, such as object orientation, automatic memory management, type-safety, platform independence, portability, security, easy-to-learn properties and thus a higher productivity in code development, have turned Java into an interesting alternative for HPC.

Nevertheless, although the performance gap between Java and natively compiled languages is usually relatively small for sequential applications, it can be particularly large for parallel applications when depending on communications performance. The related literature points out the inefficiency of Java communications as the main cause for its usually low parallel performance. The main reason is that current parallel Java applications generally suffer from inefficient communication middleware, most of them based on protocols with high communication overhead (e.g., sockets-based protocols) that are unable to take full advantage of high-speed networks [44]. This lack of efficient communication support in current Message-Passing in Java (MPJ) [18] implementations usually results in lower performance than native MPI libraries, which has been the main obstacle for the embracement of Java in HPC. Thus, the adoption of Java as an alternative language on these systems heavily depends on the availability of efficient communication middleware in order to benefit from its appealing features at a reasonable overhead.

Therefore, the main motivation of the first part of the Thesis is to improve the efficiency of Java communications for HPC, providing a communication middleware that takes full advantage of high-speed networks in order to increase the performance of parallel Java applications. This communication middleware would definitely contribute to increase the benefits of the adoption of Java for HPC, in order to achieve higher parallel programming productivity.

### 1.1.2. HPC in the Cloud

Cloud computing [15, 72], the current emerging trend in delivering IT services, is a relatively recent Internet-based computing model which is gaining significant acceptance in many areas and IT organizations as an elastic, flexible, and variable-cost way to deploy their service platforms using outsourced resources. These resources can be rapidly provisioned and released with minimal management effort. Public cloud providers offer access to external users who are typically billed by consumption using the pay-as-you-go pricing model. More specifically, Infrastructure as a Service (IaaS) is a type of cloud service which dinamically provides, by means of virtualization technologies, on-demand and self-service access to elastic computational resources (e.g., CPU, memory, networking and storage). Public IaaS cloud providers typically make huge investments in data centers, and then rent it out, allowing consumers to avoid heavy capital investments and obtain both cost-effective and energy-efficient solutions [71]. Thus, organizations are no longer forced to invest in additional computational resources, since they can just leverage the infrastructure offered by the IaaS provider. Customers can derive significant economies of use by leveraging the pay-per-use model, instead of upgrading their infrastructure, dimensioned to handle peak requests.

Traditional HPC systems are typically managed and operated by individual organizations in private. However, computing demand is fluctuating, resulting in periods where dedicated resources are either underutilized or overloaded. The advantages of the pay-as-you-go model, elasticity, flexibility, customization, security, migration and resource control offered by virtualization make cloud computing an attractive option for meeting the needs of HPC applications. IaaS cloud providers can be well-suited for handling the increasing processing requirements of resource-intensive applications due to the high availability of on-demand computational resources at large scale. In fact, in the last years the use of cloud infrastructures for HPC has generated a strong interest in the scientific and research community as an emerging and feasible alternative to traditional HPC systems [29, 49, 63, 90, 92]. The cloud-based approach offers a lot of apparent benefits and interesting perspectives, promising cost savings and more flexibility. This model provides a powerful abstraction that easily allows end users to set up elastic virtual clusters to exploit supercomputing-level power without any knowledge of the underlying infrastructure. The provisioning of these virtual clusters as on-demand pay-as-you-go resources avoids initial cost for physically owned hardware and allows great flexibility and scalability for customers and their applications. Moreover, cloud-based virtual clusters allow scientists to securely gain administrative privileges within the guest operating system and completely customize their execution environment, thus providing the perfect setup for their experiments. Furthermore, public IaaS resources usually may on average be better utilized than private computing resources within laboratories or companies, which increases efficiency in terms of cost and energy consumption.

Despite the advantages of cloud for HPC, it still remains unclear whether and when clouds can become a feasible substitute or complement to clusters and supercomputers. There is a mismatch between the requirements and goals of HPC and the characteristics and goals of current cloud environments. HPC is performance-oriented, whereas clouds are usually cost and resource-utilization oriented. Furthermore, clouds have traditionally been designed to run business and web applications. The interest in the use of public clouds for HPC increases as their availability, computational power, price and performance improves. However, previous early studies [26, 29, 30, 49, 63, 67, 90, 92] have evaluated public clouds and the main conclusion was that clouds were not designed for running tightly-coupled HPC workloads, such as MPI applications. The main reasons are processor sharing and the poor network performance, mainly caused by the use of commodity interconnection technologies (e.g., Gigabit Ethernet) and the virtualization network overhead, that limit severely the scalability of HPC applications in public clouds. To overcome these constraints, IaaS providers have started to offer some HPC-aimed cloud resources in the last years. These recent efforts towards HPC-optimized clouds point to a promising direction to overcome some of the fundamental performance bottlenecks.

Amazon Web Services (AWS) is the leading commercial public cloud provider in terms of services and number of users, whose Elastic Compute Cloud (EC2) service [2] is nowadays among the most used and largest IaaS cloud platform. Amazon EC2 offers several cloud resources which specifically target HPC environments [3]. These resources are composed of several instance types or Virtual Machines (VM) which are intended to be well suited for HPC workloads and other demanding network-bound applications by offering dedicated physical node allocation (a single VM per node), powerful and up-to-date multi-core CPU and GPU resources, improved network performance via high-speed Ethernet (10 Gbps) and enhanced storage performance providing high-performance Solid State Drive (SSD) disks. Using these instance types customers can expedite their HPC workloads on elastic resources as needed, adding and removing compute resources to meet the size and time requirements for their specific workloads.

Therefore, the IaaS model in clouds can enable HPC to reach out to a wider scientific and industrial community. Whether and how this potential can be realized in practice is a research question that we aim to answer in the second part of this Thesis. Hence, the main motivation of this part is to examine the feasibility of using a public cloud infrastructure for HPC and scientific computing. The Amazon EC2 cloud has been selected as a representative public IaaS provider because it is currently the most popular one, offering several HPC-aimed resources which are suitable for this assessment study.

## 1.2. Objectives of the Thesis

Taking into account the particular motivations for the Thesis that have been discussed above, the main objectives for each part of this work have been defined. Thus, among the specific goals of the first part of the Thesis are the design and implementation of middleware to improve Java communication efficiency on top of high-speed networks, and the evaluation of the current state of Java for HPC, particularly for cluster and supercomputer systems. To achieve this, the design and development of low-level communication middleware that increases Java performance on current HPC systems has been carried out. Hence, the Thesis presents the design, implementation and evaluation of several low-level Java communication devices optimized for the following high-speed networks: InfiniBand, RoCE, iWARP, Myrinet, high-speed Ethernet and Cray Gemini/Aries (Chapters 2 and 4). These communication devices provide a high-level message-passing API that follows the MPJ specification, based on the MPI standard widely used in HPC. Moreover, the Thesis presents an efficient Java message-passing library, FastMPJ (Chapter 3), in which all these communication devices haven been successfully integrated. Furthermore, the Thesis provides an up-to-date review of Java for HPC (Chapter 5), which includes an extensive evaluation of the performance of current projects.

The main goals of the second part of the Thesis include the evaluation of the Java message-passing middleware, developed in the first part, on the Amazon EC2 IaaS platform using HPC-aimed cloud resources. This assessment has been conducted both in terms of performance (Chapters 6 and 7) and cost (Chapter 7), using both Java and natively compiled codes. Moreover, it is fundamental to measure not only the raw performance, but also the isolated impact of the virtualization overhead on the scalability of communication-intensive HPC applications (Chapter 6). In addition, some techniques to reduce this overhead have been assessed, including the impact of the use of different process mappings and several levels of parallelism (Chapter 7). Next, in order to provide a thorough analysis of the state of the art regarding the use of the Amazon EC2 cloud for HPC, other additional aspects of this cloud to be a viable alternative in the HPC area have to be considered. Hence, further assessments of Amazon EC2 have been carried out: the evaluation of the I/O storage subsystem (Chapter 8), the performance characterization of parallel/distributed file systems for data-intensive applications (Chapter 9), and the feasibility of using GPUs as many-core HPC accelerators (Chapter 10).

Finally, all these efforts have served to accomplish the twofold goal of this Thesis: improving Java communication efficiency for HPC applications on top of high-speed newtworks, and providing a feasibility study of the use of a public cloud computing infrastructure for HPC.

## 1.3. Research and Discussion

This section presents an overview of the whole research carried out in the Thesis. Basically, we describe how the main goals defined in the previous section have been accomplished by the different journal articles that make up this work, providing an overall discussion of the main research results that have been achieved. The particular research concerning each of the two parts is discussed separately next.

### 1.3.1. Part I: Design of Low-Latency Java Communication Middleware on High-Speed Networks

The first part of the Thesis follows a bottom-up approach. First, it is targeted at the design and development of low-level Java communication devices on high-speed networks. These devices provide efficient point-to-point communication primitives conforming to a low-level message-passing API, which is not aware of higher level MPI abstractions (e.g., communicators). Next, an MPJ library (FastMPJ) is implemented on top of the previously developed communication middleware, taking advantage of the more efficient high-speed network support. Additionally, an extensive evaluation of the FastMPJ performance has been provided, including an up-to-date review of Java for HPC. A brief summary of this research is next presented.

#### Design of Scalable Java Communications over InfiniBand (Chapter 2)

The first task of the Thesis has been the design and implementation of `ibvdev` [38], a low-level Java message-passing communication device over InfiniBand (IB). The efficient support of this high-speed network in Java is of great interest, as it is currently the most widely adopted RDMA networking technology in the TOP500 list [89]. Furthermore, the use of Java sockets on top of the IP over IB emulation protocol (IPoIB) [48] has shown quite poor performance in the related literature [44]. These facts have motivated this task as our first main goal, in order to explore the feasibility of implementing efficient Java communications over native IB in a production and already existing MPJ library. The MPJ Express (MPJE) middleware [9], which implements the mpiJava 1.2 API [17] (the most widely extended MPI-like Java bindings), was selected as it is the most active project in terms of adoption by the HPC community, presence on academia and production environments, and available documentation.

The `ibvdev` device transparently provides Java message-passing applications with efficient communications over IB thanks to its direct support of the IB Verbs (IBV) API and the scalable implementation of a lightweight communication protocol that is able to take advantage of RDMA operations. This device conforms with the `xdev` message-passing API [8], which is provided by MPJ Express to support a new transport protocol. More specifically, this device implements an eager and a rendezvous communication protocol (see Figure 1.1) through the Java Native Interface (JNI). These protocols rely on the Reliable Connection (RC) transport service defined in the IBV specification, which provides reliability, delivery order and data loss and error detection. On the one hand, the eager protocol of `ibvdev`, which is used for short messages to achieve the lowest latency, is illustrated in Figure 1.2. It has been implemented using a copy in/out protocol, as the overhead of data copies is small for short messages. Moreover, the buffer registration and unregistration overhead is avoided by using a shared pool of pre-registered, fixed size buffers for communication. Hence, for sending an eager data message, the user data is copied to one of the available buffers from the pool (step 1 of the figure) and then sent out from this buffer to the send queue. At the receiving side, a number of buffers from the pool are pre-posted (step 2). After the message is received, the message payload is copied to the user destination buffer (step 3).

On the other hand, the rendezvous protocol, used for large messages, negotiates the buffer availability at the receiving side before the message is actually transferred. A zero-copy protocol has been implemented by using the RDMA Write operation. In this implementation (see right part of Figure 1.1), the application buffers are registered on-the-fly and the buffer addresses are exchanged via control messages. The sending process first sends a control message to the receiver (RNDZ_START). The receiver replies to the sender using another control message (RNDZ_REPLY). This

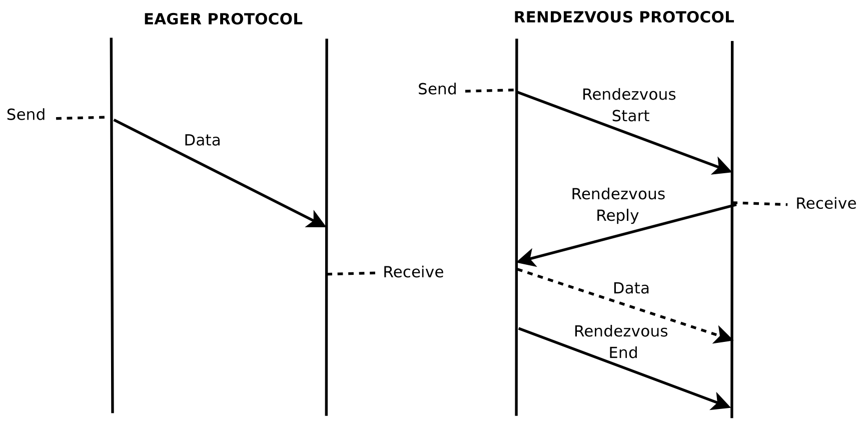


Figure 1.1: Eager and rendezvous protocols in `ibvdev`

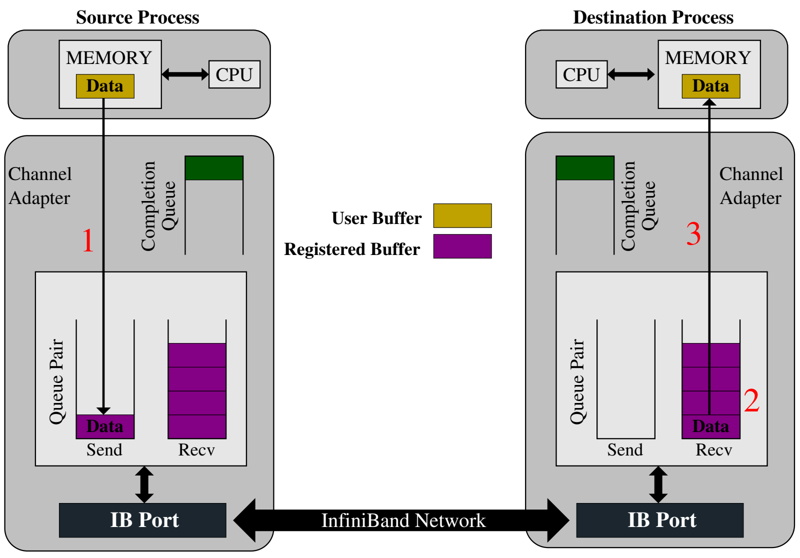


Figure 1.2: Eager protocol implementation in `ibvdev`

reply message contains the information of the receive application buffer along with the required remote key to access that memory region. The sending process then sends the large message directly to the receive application buffer by using RDMA Write. Finally, the sending process issues another control message (RNDZ_END) which indicates to the receiver that the message has been placed in the application buffer. Additionally, this protocol takes advantage of a cache of registered buffers that can reduce the buffer registration and unregistration overhead. The effectiveness of this cache depends heavily on the reuse rate of the buffers.

The experimental evaluation of the `ibvdev` device on a multi-core IB cluster has shown significant point-to-point performance benefits (see Figure 1.3), obtaining up to 85% start-up latency reduction and twice the bandwidth compared to previous Java sockets-based middleware using the IPoIB protocol (i.e., `niodev`). Furthermore, the analysis of the impact of the use of `ibvdev` on representative message-passing kernels and applications has shown significant performance gains, as shown in Figure 1.4, which presents the FT kernel of the NAS Parallel Benchmarks (NPB) [6] as example. Therefore, the efficiency of `ibvdev`, which is even competitive with native MPI libraries (Open MPI and MVAPICH) in terms of point-to-point performance, increases the scalability of communication-intensive parallel Java applications, which helps to bridge the performance gap between MPJ and MPI.

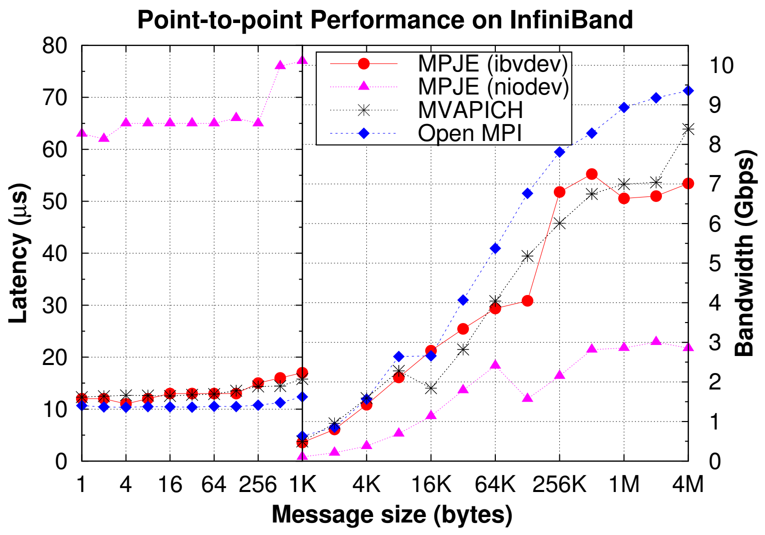
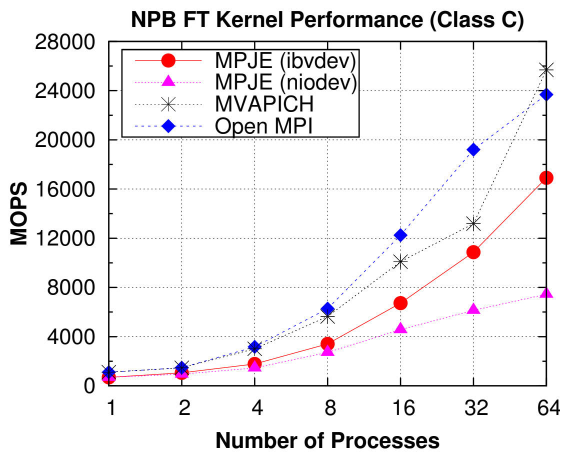


Figure 1.3: Message-passing point-to-point performance on IB



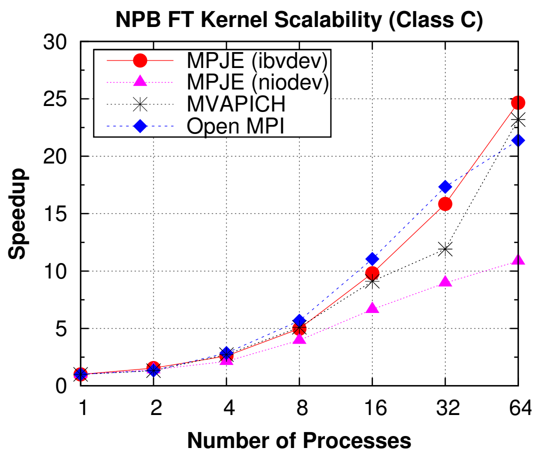


Figure 1.4: Performance and scalability of the NPB FT kernel on IB

**FastMPJ: a Scalable Java Message-Passing Library (Chapter 3)**

The proof-of-concept implementation of the `ibvdev` device in the MPJ Express library showed that it was feasible to obtain efficient Java communications over IB. However, the overall design of MPJ Express relies on a buffering layer [10] that is only able to transfer the custom `xdev` buffer objects. In fact, this layer adds a noticeable copying overhead that significantly limits performance and scalability of communications, as shown in previous works [84]. Furthermore, MPJ Express includes poorly scalable collective algorithms and its bootstrapping mechanism typically exhibits some issues in specific environments. As a consequence of these constraints, the `ibvdev` device was reimplemented to conform with the `xxdev` API and then adapted for its integration into the F-MPJ library [83] in order to improve its performance and scalability.

F-MPJ is our Java message-passing implementation of the mpiJava 1.2 API, presented as a proof of concept in [83]. The communication support of F-MPJ is implemented on top of the `xxdev` device layer, which has been designed as a simple and pluggable architecture of low-level communication devices. This layer supports the direct communication of any serializable Java object without data buffering, whereas `xdev`, the API that `xxdev` is extending, does not support this direct communication, as mentioned before. The avoidance of this intermediate data buffering overhead is the main benefit of the `xxdev` device layer with respect to its predecessor. The `xxdev` API (see Listing 1.1) is composed of basic operations such as point-to-point communications, both blocking (send and recv) and nonblocking (isend and irecv). It also includes synchronous communications (ssend and issend) and methods to check incoming messages without actually receiving them (probe and iprobe). Additionally, F-MPJ includes a scalable collective library with more than 60 topology-aware algorithms, which are implemented on top of point-to-point communications [80].

Nevertheless, F-MPJ presents several important constraints as it was intended as a research-oriented and proof-of-concept implementation. In fact, it only implements a small subset of the communication-related API, which prevents its use in most real-world applications (e.g., it does not support inter- and intra-communicators, virtual topologies and group operations). Additionally, it includes a basic bootstrap-

```
public abstract class Device
{
  public static Device newInstance(String device);
  abstract ProcessID[] init(String[] args);
  abstract ProcessID id();
  abstract void finish();

  abstract void send(Object buf, PID dst, int tag, int context);
  abstract Status recv(Objecct buf, PID src, int tag, int context);

  abstract Request isend(Object buf, PID dst, int tag, int context);
  abstract Request irecv(Object buf, PID src, int tag, int context, Status stts);

  abstract Request issend(Object buf, PID dst, int tag, int context);
  abstract void ssend(Object buf, PID src, int tag, int context);

  abstract Status iprobe(PID src, int tag, int context);
  abstract Status probe(PID src, int tag, int context);
}
```

Listing 1.1: API of the xxdev.Device class

ping mechanism based on system-dependent scripts, unable to provide portability on different platforms. Furthermore, it only includes one communication device implemented on top of Java IO sockets (`iodev`), which severely limits its overall scalability. Although the use of high-performance socket implementations, such as the Java Fast Sockets (JFS) project [82], can improve performance on shared memory and high-speed networks, the use of the sockets API still represents an important source of overhead and lack of scalability in Java communications, especially in the presence of high-speed networks [44]. Therefore, in addition to the reimplementation of `ibvdev`, our main goal has been to overcome the current limitations of F-MPJ in order to provide a production-quality implementation of the full MPJ specification, which is now known as FastMPJ [31].

In addition to the integration of `ibvdev` in FastMPJ, two new communication devices, `mxdev` and `psmdev`, have been included in order to improve its high-speed network support. On the one hand, `mxdev` implements the `xxdev` API on top of the Myrinet Express (MX) library [62], which runs natively on Myrinet networks. More recently, the MX API has also been supported in high-speed Ethernet networks (10/40 Gigabit Ethernet), both on Myricom specialized NICs and on any generic Ethernet hardware through the Open-MX open-source project [42]. Hence, the TCP/IP stack can be replaced by `mxdev` transfers over Ethernet networks providing higher performance than using standard Java sockets. On the other hand, `psmdev`

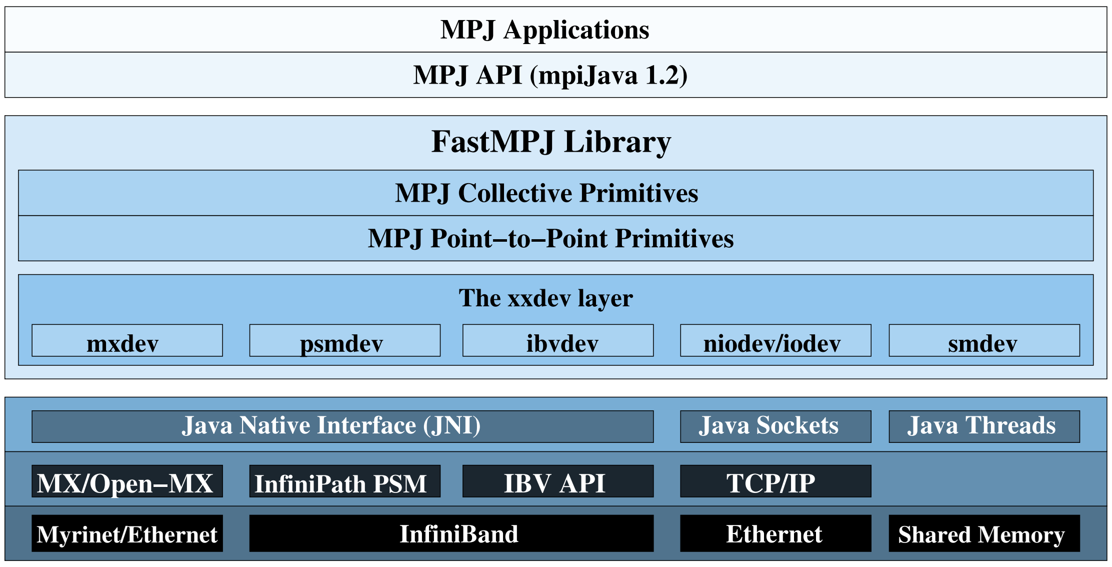


Figure 1.5: Overview of the FastMPJ layered design and `xxdev` devices

provides native support for the InfiniPath family of Intel/QLogic IB adapters over the Performance Scaled Messaging (PSM) interface. PSM is a low-level user-space messaging library which implements an intra-node shared memory and an inter-node communication protocol, which are completely transparent to the application. Although the Intel/QLogic IB adapters are also supported by the `ibvdev` device, `psmdev` usually achieves significantly higher performance than `ibvdev`, as PSM has been specifically designed by Intel/QLogic for its hardware. Furthermore, the implementation of a new device based on Java NIO sockets (`niodev`), which includes more scalable non-blocking communication support than `iodev` by providing `select()`-like functionality, has also been included. Nevertheless, these socket-based devices are only provided for portability reasons, as they rely on the ubiquitous TCP/IP stack, which introduces high communication overhead and limited scalability for communication-intensive applications. Figure 1.5 presents a high-level overview of the FastMPJ layered design and its `xxdev` communication devices.

Furthermore, FastMPJ has been evaluated comparatively with native MPI libraries on five representative testbeds: two IB multi-core clusters, one Myrinet supercomputer, and two shared memory systems using both Intel- and AMD-based processors. The comprehensive performance evaluation has revealed that FastMPJ communication primitives are quite competitive with their MPI counterparts, both

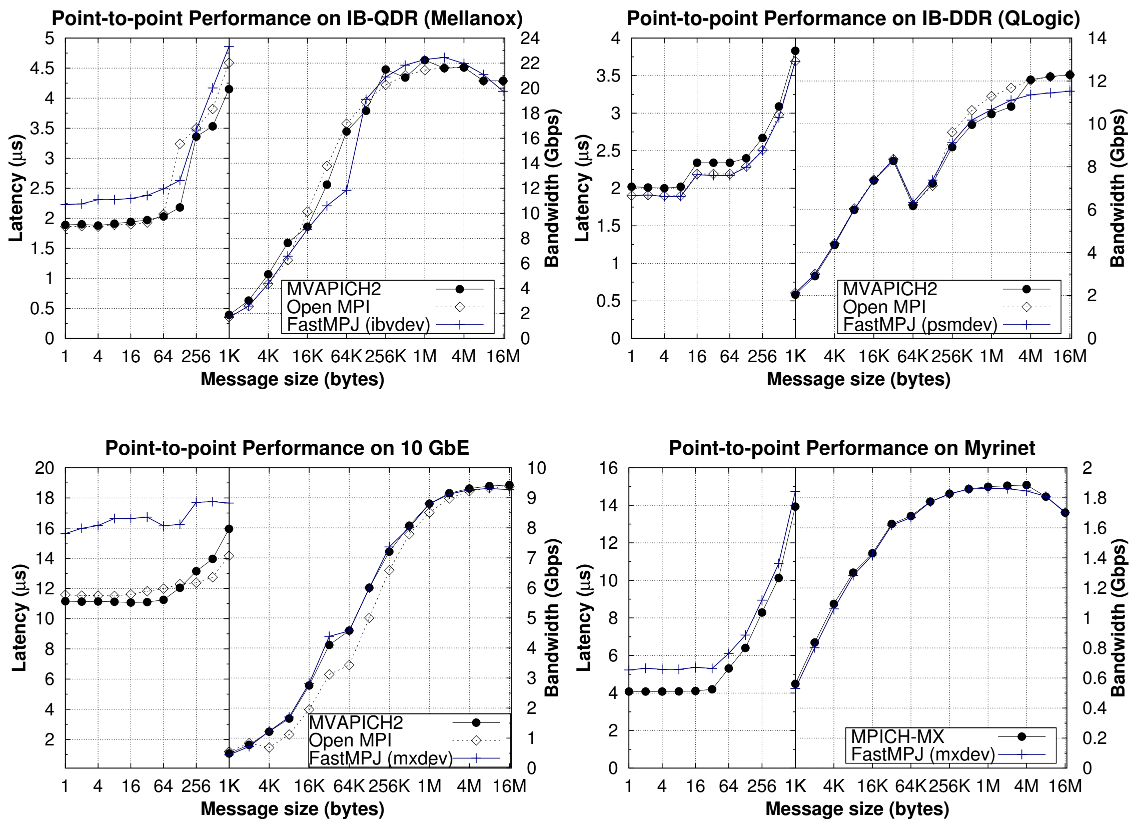


Figure 1.6: Point-to-point performance on IB, 10 Gigabit Ethernet (10 GbE) and Myrinet

in terms of point-to-point and collective operations performance (see Figures 1.6 and 1.7). Hence, the use of the FastMPJ library in communication-intensive HPC codes allows Java to benefit from a more efficient communication support, taking advantage of the use of a high number of cores and improving significantly the performance and scalability of parallel Java applications (see results for the NPB CG and MG kernels in Figure 1.8). In fact, the development of this efficient Java communication middleware is definitely bridging the gap between Java and native languages in HPC applications.

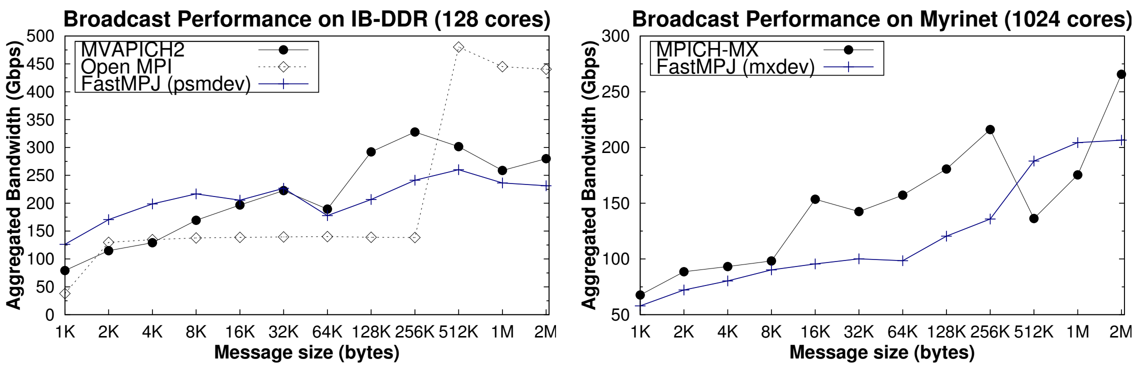


Figure 1.7: Broadcast performance on IB and Myrinet

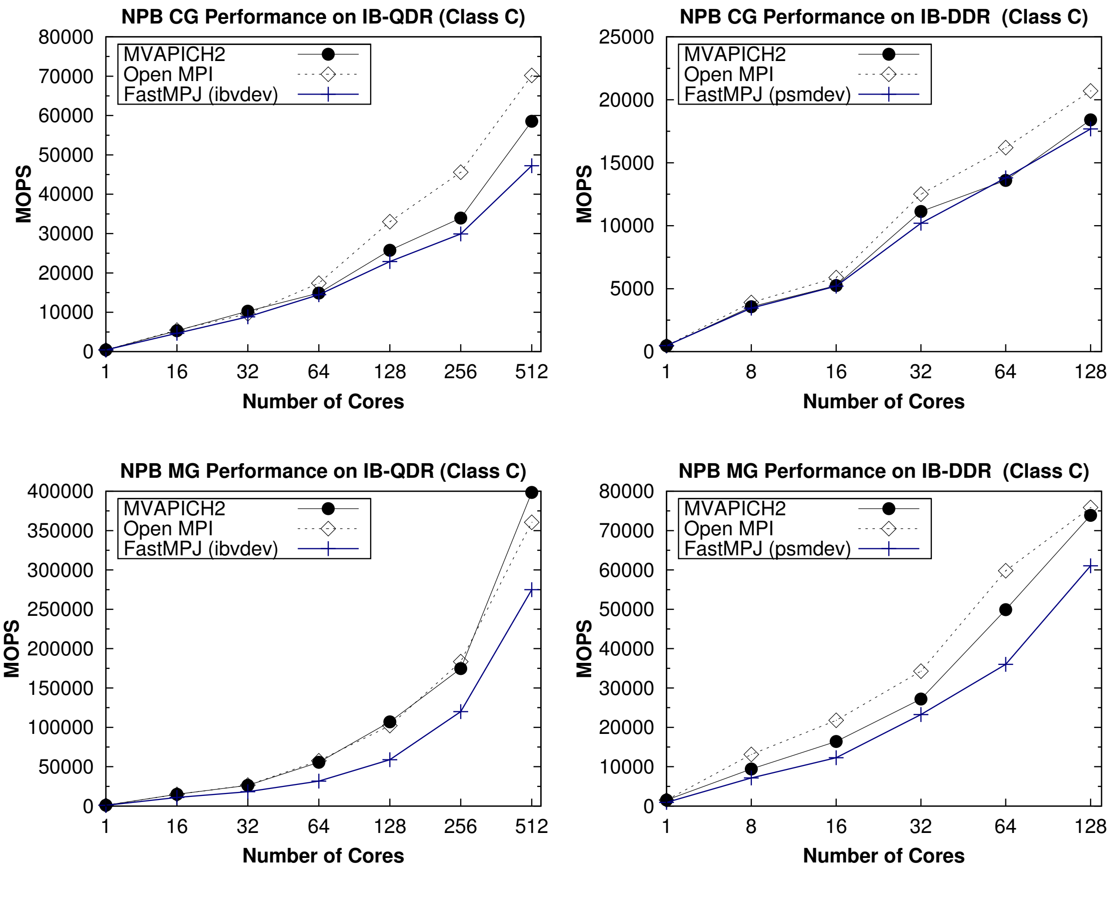


Figure 1.8: Performance of NPB CG and MG kernels on IB

### Java Communication Devices on RDMA-enabled Networks (Chapter 4)

RDMA is a well-known mechanism that enables zero-copy and kernel-bypass features, providing low-latency and high-bandwidth communications with low CPU utilization. In fact, RDMA has become a key capability of current high-speed networks to provide scalable inter-node communications for HPC applications. This work presents new research results on improving the RDMA network support in FastMPJ [36], our Java message-passing implementation. Thus, it focuses on providing efficient low-level communication devices that fully exploit the underlying RDMA hardware, enabling low-latency inter-node communications for Java message-passing applications. More specifically, it presents two new `xxdev` communication devices, `ugnidev` and `mxmdev`, implemented on top of the user-level Generic Network Interface (uGNI) [22] and MellanoX Messaging (MXM) [60] communication libraries, respectively. The former device is intended to provide efficient support for the Gemini/Aries RDMA networks used by the Cray XE/XK/XC family of supercomputers. The latter includes support for the MXM library, which has been developed by Mellanox for its RDMA adapters. Furthermore, an enhanced version of the `ibvdev` device, which extends its current RDMA support to RoCE and iWARP networking hardware and introduces an optimized short-message communication protocol that takes advantage of the inline feature, is also included. These `xxdev` devices (highlighted in italics and red in Figure 1.9) have been integrated transparently into FastMPJ thanks to its modular structure, allowing current MPJ applications to benefit transparently from a more efficient support of RDMA networks (depicted by red arrows at the hardware level).

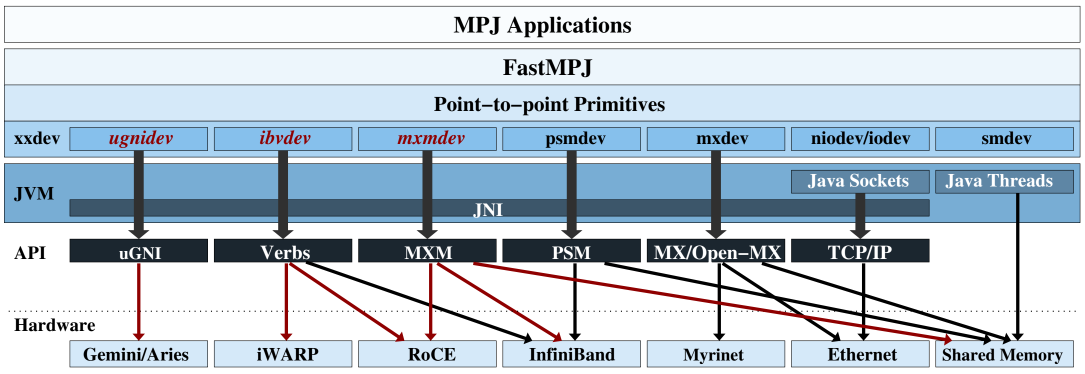
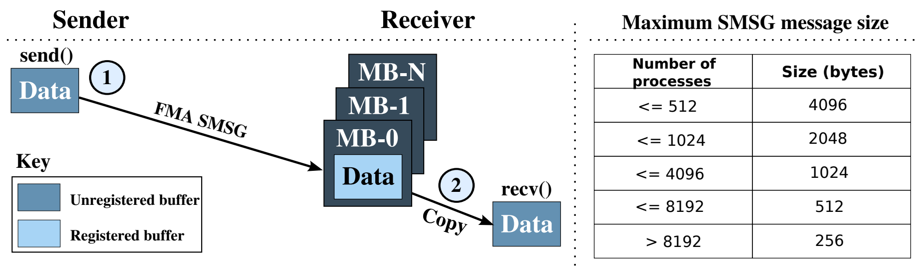


Figure 1.9: Overview of the FastMPJ communication devices



| Number of processes | Size (bytes) |
|---|---|
| <= 512 | 4096 |
| <= 1024 | 2048 |
| <= 4096 | 1024 |
| <= 8192 | 512 |
| > 8192 | 256 |

Figure 1.10: First path of the eager protocol in `ugnidev`

These devices have considered several communication protocols in order to provide scalable support for RDMA networks, enabling 1-$\mu$s start-up latencies and up to 49 Gbps bandwidth for Java message-passing applications, as will be shown in Figure 1.11. As a representative example of these protocols, the first path of the eager protocol in `ugnidev` is briefly described next. In this path (see Figure 1.10), which is implemented using the Fast Memory Access Short Messaging (FMA SMSG) facility provided by the Cray Gemini/Aries hardware, each process creates and registers with the network adapter per-process destination buffers called mailboxes (MB in the figure). During a message transfer, the sender directly writes data to its designated mailbox at the receiving side (step 1 in Figure 1.10). Next, the received data is copied out from the mailbox to the application buffer provided by the user (step 2). SMSG handles the delivery to the remote mailbox and raises both a local and a remote completion queue event on the sending and receiving sides, respectively, upon successful delivery. SMSG transactions are a special class of RDMA PUT operations which require remote buffer memory registration, but not local memory registration, which allows to send the data directly from the application buffer. However, using the SMSG facility requires a significant amount of registered memory resources which scale linearly with the number of processes in the job. To alleviate this problem, SMSG is only used for communications up to a certain small message size, which is a configurable runtime option. By default, the maximum message size that can be sent using SMSG varies with the job size, with smaller mailboxes being used as the job size increases, in order to decrease the amount of memory used for SMSG mailboxes for larger jobs (see table in Figure 1.10). Above this message size, `ugnidev` switches to a second alternative path. More details about the implementation of the `ugnidev` device, as well as about the rest of devices presented in this work, are described in Chapter 4.

In order to evaluate the benefits of these devices, their performance has been analyzed comparatively with other Java communication middleware on representative RDMA networks (IB, RoCE, iWARP, Cray Gemini) and parallel systems (a multi-core InfiniBand cluster and a TOP500 Cray supercomputer). The analysis of the results has demonstrated experimental evidence of significant point-to-point performance improvements when using the developed devices in FastMPJ (see results for RoCE and Gemini networks in Figure 1.11). In fact, the scalability of parallel Java codes can benefit transparently from this efficient support on RDMA networks (see results for the NPB FT kernel in Figure 1.12), allowing to obtain up to 24% and 40% improvement in application-level performance on 256 and 4096 cores of a multi-core IB cluster and a Cray XE6 supercomputer, respectively (see Figure 1.13).

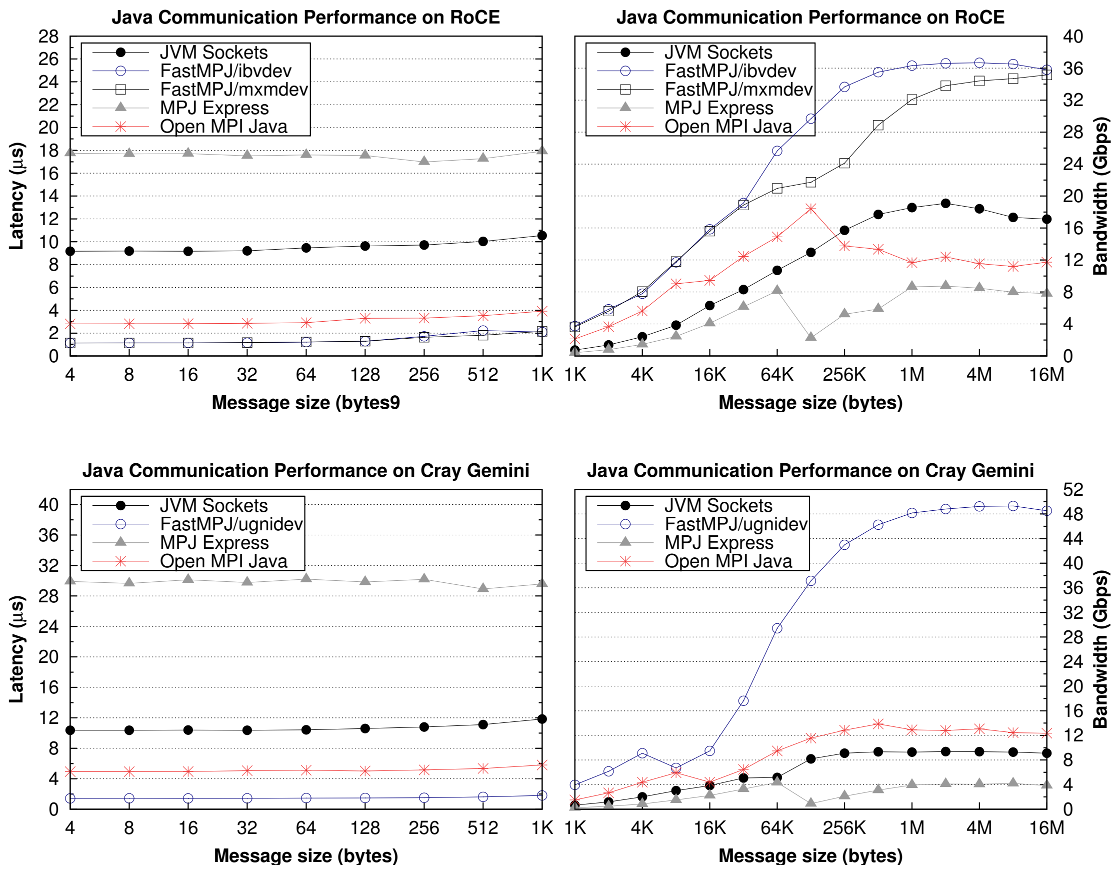


Figure 1.11: Point-to-point performance on RoCE and Cray Gemini

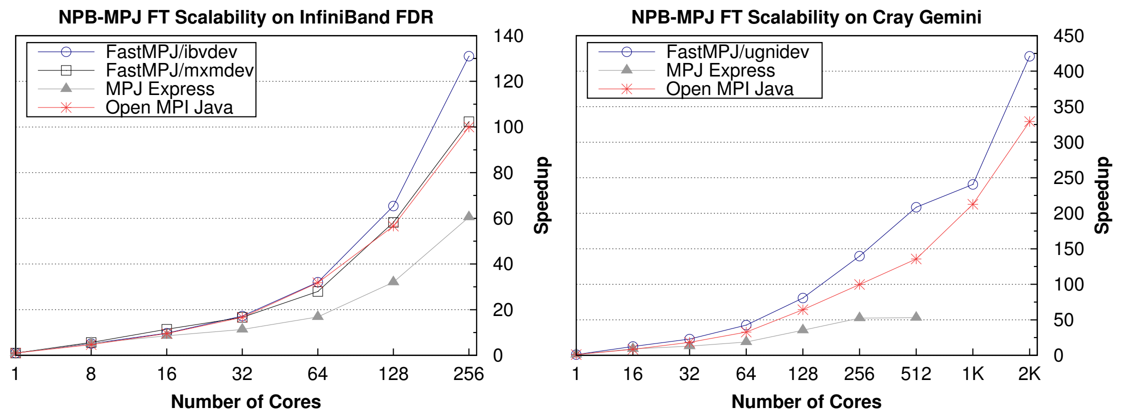


Figure 1.12: Scalability of the NPB FT kernel on IB and Cray Gemini

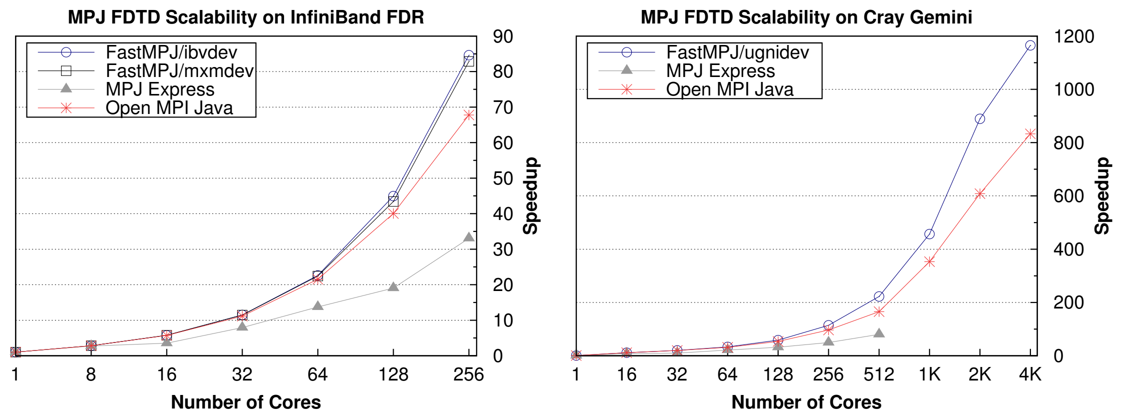


Figure 1.13: Scalability of the FDTD parallel application on IB and Cray Gemini

**Java in the HPC Arena: Research, Practice and Experience (Chapter 5)**

The final task of the first part of the Thesis has been to provide an up-to-date review of Java for HPC, both for shared and distributed memory programming [79]. This work shows an important number of past and present research projects which are the result of the sustained interest in the use of Java for HPC. Despite the reported advances in the efficiency of Java communications shown by our previous results, the use of Java in HPC is also being delayed by the lack of analysis of the existing programming options in Java for HPC and thorough and up-to-date evaluations of their performance, as well as the unawareness on current research projects in this field, whose solutions are needed in order to boost the embracement of Java in HPC.

As a consequence, this work first analyzes the existing programming options in Java for HPC, which allow the development of both high-level libraries and parallel Java applications. These options are usually classified into: (1) shared memory programming, such as the use of Java threads, OpenMP-like implementations (e.g., JOMP [55]) and Partitioned Global Address Space (PGAS) projects (e.g., Titanium [96]). The OpenMP-like approach has several advantages over the use of Java threads, such as the higher-level programming model with a code much closer to the sequential version and the exploitation of the familiarity with OpenMP, thus increasing programmability. However, current OpenMP-like implementations are still preliminary works and lack efficiency and portability. Regarding Titanium, [24] reports that it outperforms Fortran MPI code, but it also presents several limitations, such as the avoidance of the use of Java threads and the lack of portability as it relies on Titanium and C compilers. (2) Java sockets (IO/NIO), which usually lack efficient high-speed networks support and so Java has to resort to inefficient TCP/IP emulations for full networking support (e.g., IPoIB), and high-performance socket implementations (e.g., Ibis [65], JFS [82]). (3) Java Remote Method Invocation (RMI), which allows an object running in one JVM to invoke methods on an object running in another JVM, providing Java with remote communication between programs equivalent to Remote Procedure Calls (RPCs). The main advantage of this approach is its simplicity, although the main drawback is the poor performance shown by the RMI protocol, whose optimization has been the goal of several projects (e.g., ProActive [5], Manta [57], KaRMI [68], Opt RMI [81]). And

(4) efficient MPJ middleware (e.g., mpiJava [7], MPJ Express [9], MPJ/Ibis [13], FastMPJ [31]). Although the MPI standard declaration is limited to C and Fortran languages, there have been a number of standardization efforts made towards introducing an MPI-like Java binding. The two main APIs are the mpiJava 1.2 API [17], which has been proposed by the mpiJava [7] developers and tries to adhere to the MPI C++ interface defined in the MPI standard version 2.0, but restricted to the support of the MPI 1.1 subset; and the JGF MPJ API [18], which is the proposal of the Java Grande Forum (JGF) [50] to standardize the MPI-like Java API. The main differences between both APIs lie on naming conventions of variables and methods.

Next, current research efforts in Java for HPC are described, with special emphasis on providing scalable communication middleware. These projects can be mainly classified into: (1) Design and implementation of low-level Java message-passing devices, such as the ones presented in this Thesis [36, 38]. Message-passing libraries usually support new transport protocols through the use of pluggable low-level communication devices, such as Abstract Device Interface (ADI) in MPICH, Byte Transfer Layer (BTL) in Open MPI, `xdev` [8] in MPJ Express and `xxdev` [83] in FastMPJ. These communication devices abstract the particular operation of a communication protocol, such as IBV, MX, TCP or shared memory, conforming to an API on top of which the message-passing library implements its communications. (2) Improvement of the scalability of Java message-passing collective primitives and automatic selection of MPJ collective algorithms [80]. (3) Implementation and evaluation of MPJ benchmarks, such as the JGF benchmarking suite [14] and the implementation of the NPB kernels for MPJ (NPB-MPJ) [58]. (4) Language extensions in Java for parallel programming paradigms (Habanero Java [19] and X10 [20] projects). And (5) Java libraries to support data parallelism on massively parallel architectures such as GPUs, both for CUDA (e.g., jCuda [51], JCUDA [95]) and OpenCL (e.g., JOCL [53]) programming models. These ongoing projects are providing Java with several evaluations of its suitability for HPC, as well as solutions for increasing its performance and scalability in HPC systems with high-speed networks and many-core accelerators such as GPUs. This review has pointed out that the significant interest in Java for HPC has led to the development of numerous projects, although usually quite modest, which may have prevented a higher development of Java in this field.

Finally, this work includes an extensive performance evaluation that focuses on message passing due to its scalability and extended use in HPC. Hence, the performance of representative MPJ and MPI libraries has been assessed on two shared memory environments and two multi-core InfiniBand systems: an x86_64-based cluster and an Itanium supercomputer (Finis Terrae). This comparative evaluation consists of a micro-benchmarking of point-to-point and collectives primitives, as well as a kernel/application benchmarking using representative parallel codes, both NPB kernels and the Gadget-2 application [73, 76], in order to analyze the impact of the use of MPI/MPJ libraries on their overall performance. The main conclusion of the analysis of these results is that Java can achieve comparable performance to natively compiled languages, both for sequential and parallel applications (see Figure 1.14 for Gadget results), being an alternative for HPC programming. In fact, the recent advances in the efficient support of Java communications on high-speed networks, such as the ones presented in this Thesis, are bridging the gap between Java and more traditional HPC languages, although there is still room for improvement.

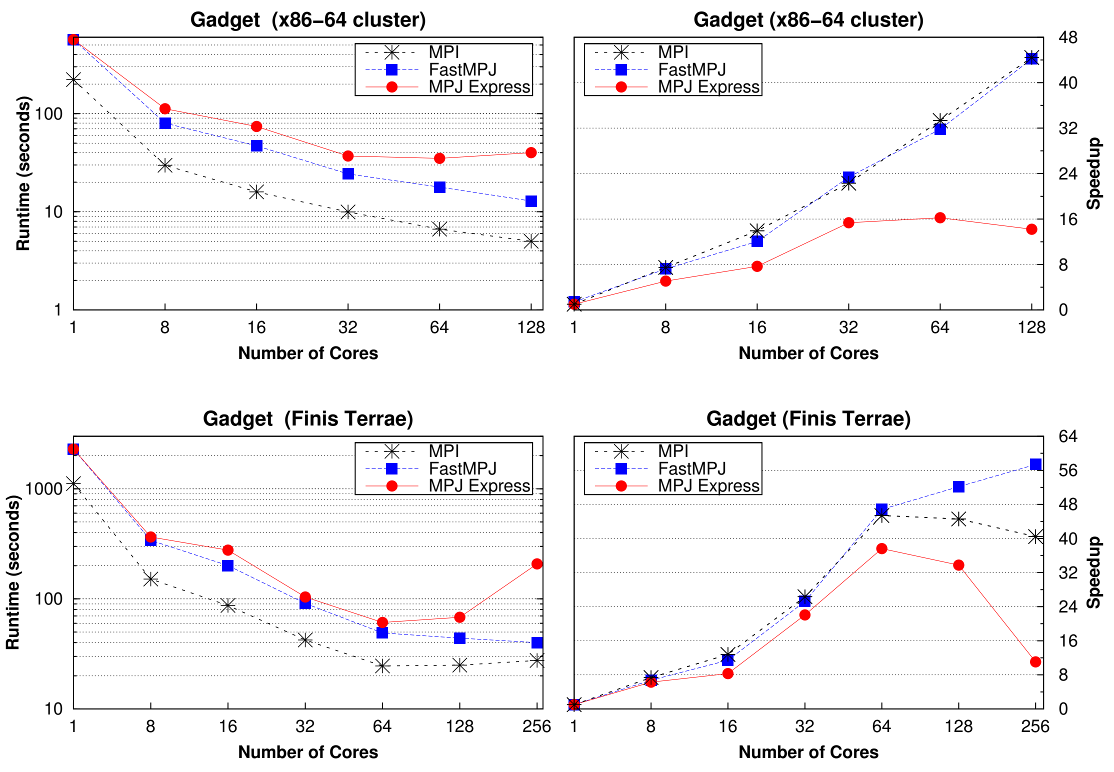


Figure 1.14: Runtime and scalability of the Gadget parallel application

### 1.3.2. Part II: Evaluation of Communication Middleware for HPC on a Public Cloud Infrastructure

The second part of the Thesis mainly follows an assessment approach. First, the performance of representative HPC applications has been evaluated on the Amazon EC2 cloud using the first generation of cluster instances, both for Java and native codes. The main performance bottlenecks when running the same benchmarks with and without different network virtualization technologies on a similar testbed have also been analyzed. Next, a larger assessment study provides more insight into the performance of running HPC applications on Amazon EC2, using a higher number of cores, comparing different cluster instance types and including an analysis in terms of cost. Moreover, several approaches to reduce the impact of the virtualization overhead have been explored (e.g., using hybrid MPI+OpenMP codes). Finally, this part concludes with further works that focus on other important aspects of the Amazon EC2 platform for HPC: the I/O storage subsystem, the performance characterization of parallel/distributed file systems for data-intensive applications and the feasibility of using GPUs as many-core accelerators. A brief summary of this research is presented below.

#### Evaluation of Messaging Middleware for Cloud Computing (Chapter 6)

This work presents an evaluation of representative native and Java message-passing middleware on the Amazon EC2 cloud infrastructure using the first generation of the HPC-aimed family of cluster instances (CC1) in order to assess their suitability for HPC applications [33]. These instances provide two powerful quad-core processors (i.e., up to 8 cores per instance) and they are interconnected via a high-speed network (10 Gigabit Ethernet), which are the differential characteristics of this resource with respect to previous non-cluster instances. According to Amazon, this instance type has been specifically designed for HPC applications and other demanding latency-bound applications.

Xen [11] is the Virtual Machine (VM) monitor or hypervisor used by Amazon EC2 among other cloud providers. The EC2 cluster instances use Xen Hardware Virtual Machine (HVM) virtualization with special ParaVirtual (PV) device drivers to bypass the emulation for disk and network, which means that the access to the 10 Gigabit Ethernet network is paravirtualized (i.e., not emulated). However, a direct access to the network hardware is possible using the Xen PCI passthrough technique [94], which provides an isolation of devices to a given guest operating system so the device can be used exclusively by that guest, which eventually achieves near-native performance.

The evaluation carried out in this work consists of a micro-benchmarking of point-to-point data transfers, both inter-VM (through 10 Gigabit Ethernet) and intra-VM (shared memory). Moreover, the scalability of representative parallel codes, the NPB kernels, has been assessed using up to 16 CC1 instances (i.e., 128 cores). Furthermore, in order to analyze the impact of the paravirtualized access to the network on Amazon EC2, a private cloud testbed (CAG, which stands for Computer Architecture Group) with similar hardware has been set up. This testbed also uses Xen as hypervisor, but it has been configured to support the direct access to the same EC2 networking technology using the Xen PCI passthrough technique. The motivation behind enabling PCI passthrough in this testbed is to analyze its impact on the efficient support of high-speed networks in virtualized environments. Additionally, the selected benchmarks have been executed on the same private testbed running a non-virtualized environment, thus obtaining the native performance of the system. The evaluated message-passing libraries have been Open MPI and MPICH2 for native codes, and FastMPJ as Java counterpart.

As main conclusion, the analysis of the performance results has shown the significant impact that virtualized environments still have on communications performance, especially on Amazon EC2, even though cluster instances feature a high-speed network. Figure 1.15 shows point-to-point latencies and bandwidths results on Amazon EC2, CAG and CAG native testbeds. It can be observed that Amazon EC2 (top graph) presents quite poor results, caused by the paravirtualized access to the network. The results on the private CAG testbed confirm that this communication overhead can be significantly alleviated through the use of PCI passthrough (see the middle graph in Figure 1.15), showing an overhead reduction for short messages of up to 50%. Moreover, the impact of the virtualization overhead can be measured using the native performance of the CAG testbed, shown by the bottom graph. This impact is noticeable for short messages but relatively small for long messages. Re-

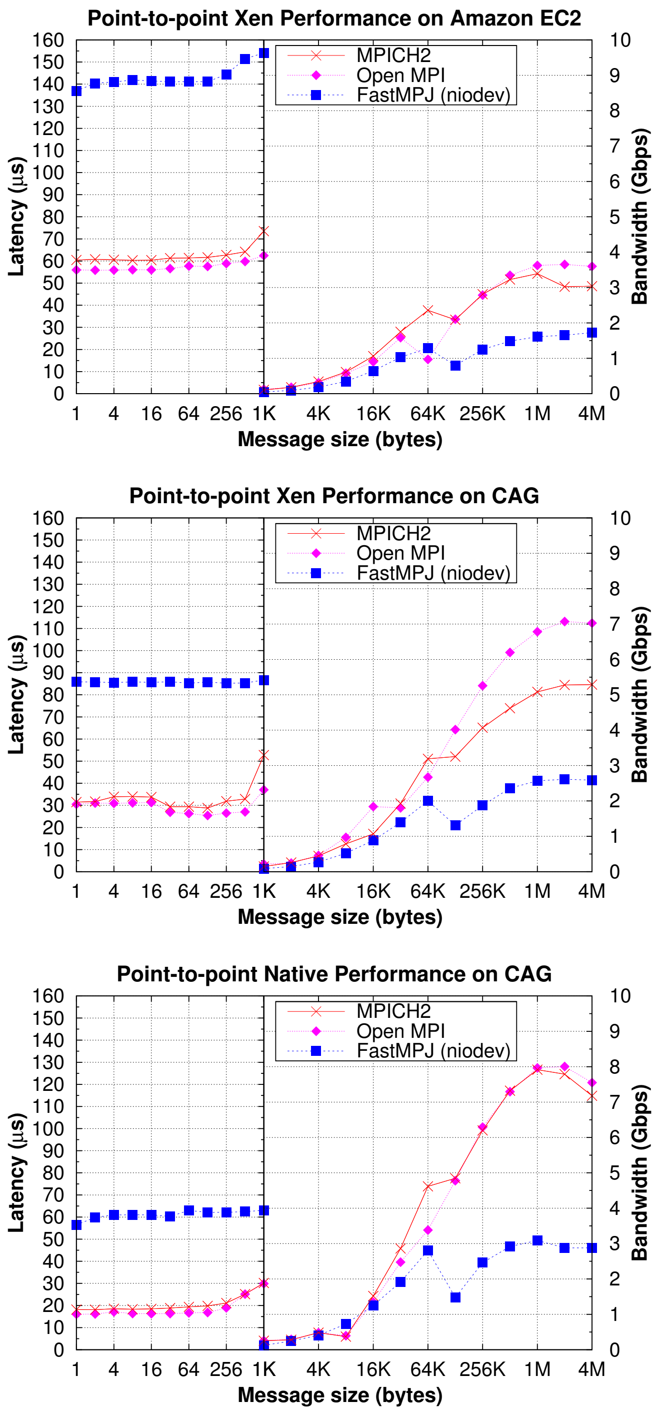


Figure 1.15: Message-passing point-to-point performance on 10 Gigabit Ethernet

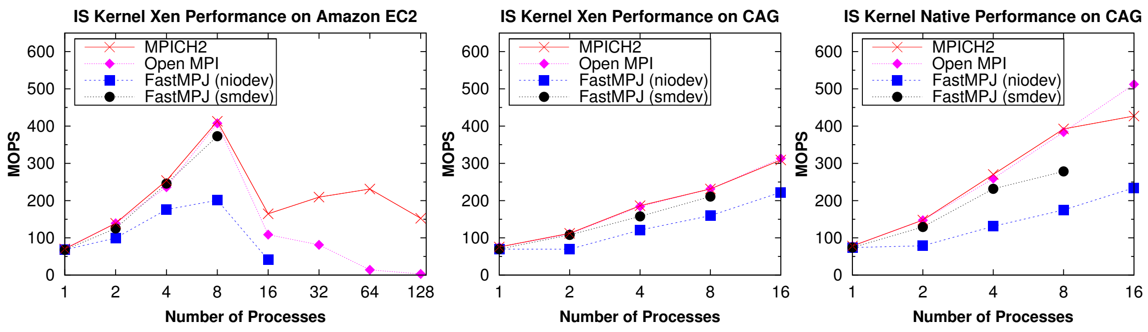


Figure 1.16: Performance of the NPB IS kernel on 10 Gigabit Ethernet

garding FastMPJ results, the `niodev` device presents poorer performance than MPI due to the high overhead of the operation of its NIO sockets implementation, which involves a significant performance penalty. In this scenario, the `mxdev` device would have improved these results avoiding the TCP/IP stack through Open-MX (also available for MPI libraries), but this library did not work on any of the virtualized environments under evaluation.

The impact of the virtualized network overhead on the scalability of HPC applications has been analyzed using the NPB suite. As a representative example, Figure 1.16 shows results for the IS kernel on Amazon EC2, CAG and CAG native testbeds. IS is a communication-intensive code whose scalability is highly dependent on communications performance. As can be observed, this kernel is not able to scale when using more than one node on Amazon EC2, thus when network activity is involved, which occurs for 16 or more processes. This statement has been proved by analyzing the IS results on CAG, where MPI and FastMPJ take advantage of the use of the network for 16 processes (2 nodes), both for Xen and native scenarios. Hence, these results have shown noticeable performance increases when supporting the direct access to the underlying network through the Xen PCI passthrough technique, reducing the communication processing overhead. This fact demands more efficient support in the virtualization layer on Amazon EC2, as these cluster instances rely on a paravirtualized access to the network that significantly limits its performance and scalability for communication-intensive HPC codes.

**Performance Analysis of HPC Applications in the Cloud (Chapter 7)**

This work gives more insight into the performance of running HPC applications on the Amazon EC2 cloud [35]. The main contributions with respect to the previous work are: (1) it compares the first generation (CC1) and the more recent second generation (CC2) of cluster instances both in terms of single instance performance, scalability and cost-efficiency of its use; (2) it assesses the scalability of the NPB codes using an important number of cores, up to 512, significantly higher than previous related works, and running heavier workloads (NPB Class C); and (3) it also explores alternatives to reduce the impact of the network virtualization overhead, such as reducing the number of processes per instance or using the combination of message passing with multithreading (e.g., hybrid MPI+OpenMP codes). The second generation of cluster instances is a resource that provides improved CPU power (88 ECUs) with two octa-core processors (i.e., up to 16 cores per instance), a more modern microarchitecture based on Sandy Bridge and higher memory bandwidth and capacity than the first generation. However, these resources also rely on a paravirtualized access to the 10 Gigabit Ethernet network (see Table 1.1 for more details on CC1 and CC2 instances).

The main conclusions of this work, which has evaluated the use of up to 64 CC1 and 32 CC2 instances, are the following: (1) it has revealed that CC2 instances, which provide more computational power and slightly higher performance for point-to-point communications (see Figure 1.17), present poorer scalability than CC1

| | **CC1** | **CC2** |
|---|---|---|
| **CPU** | 2 × Intel Xeon X5570 Nehalem-EP @2.93 GHz | 2 × Intel Xeon E5-2670 Sandy Bridge-EP @2.60GHz |
| **ECUs** | 33.5 | 88 |
| **#Cores** | 8 (16 with HyperThreading) | 16 (32 with HyperThreading) |
| **Memory** | 23 GB DDR3-1066 | 60.5 GB DDR3-1600 |
| **Storage** | 1690 GB (2 × HDD) | 3370 GB (4 × HDD) |
| **API name** | cc1.4xlarge | cc2.8xlarge |
| **Price (Linux)** | $1.30 per hour | $2.40 per hour |
| **Interconnect** | 10 Gigabit Ethernet (Full-bisection bandwidth) | |
| **Virtualization** | Xen HVM 64-bit platform (PV drivers for I/O) | |

Table 1.1: Description of the EC2 cluster compute instances: CC1 and CC2

instances for collective-based communication-intensive applications (see top graphs in Figure 1.18), while obtaining better performance for other codes (see bottom graphs); (2) the use of CC1 instances is generally more cost-effective than relying on CC2 instances, and therefore it is worth recommending CC2 only for applications with high memory requirements that cannot be executed on CC1 (see some productivity results in Figure 1.19); (3) it is possible to achieve higher scalability for some codes running only a single process per instance, thus reducing the communications performance penalty in the access to the network, in exchange for higher costs (see Figure 1.20); (4) finally, we proposed the use of multiple levels of parallelism, combining message passing with multithreading, as the most scalable option for running HPC applications on Amazon EC2 (see Figure 1.21, where FastMPJ results are not shown due to the lack of an implementation in Java of the hybrid code suite NPB-MZ [52]).

Note that FastMPJ results in this work have been obtained using an experimental communication device (`mpidev`) that allows to take advantage of native MPI libraries by implementing the `xxdev` point-to-point primitives on top of the MPI counterparts. As a consequence, FastMPJ significantly outperforms the `niodev` results shown in the previous work [33] when using CC1 instances, even obtaining quite competitive performance compared to MPI.

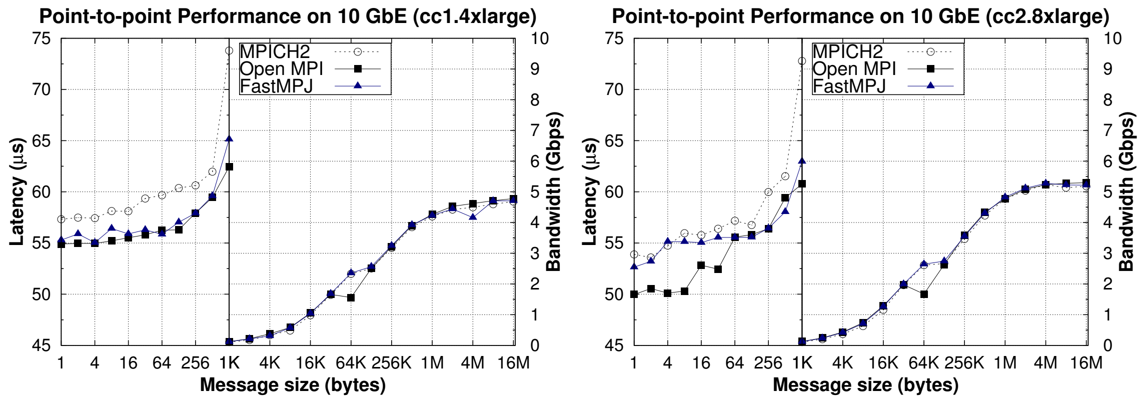


Figure 1.17: Point-to-point performance on Amazon EC2 cluster instances over 10 Gigabit Ethernet

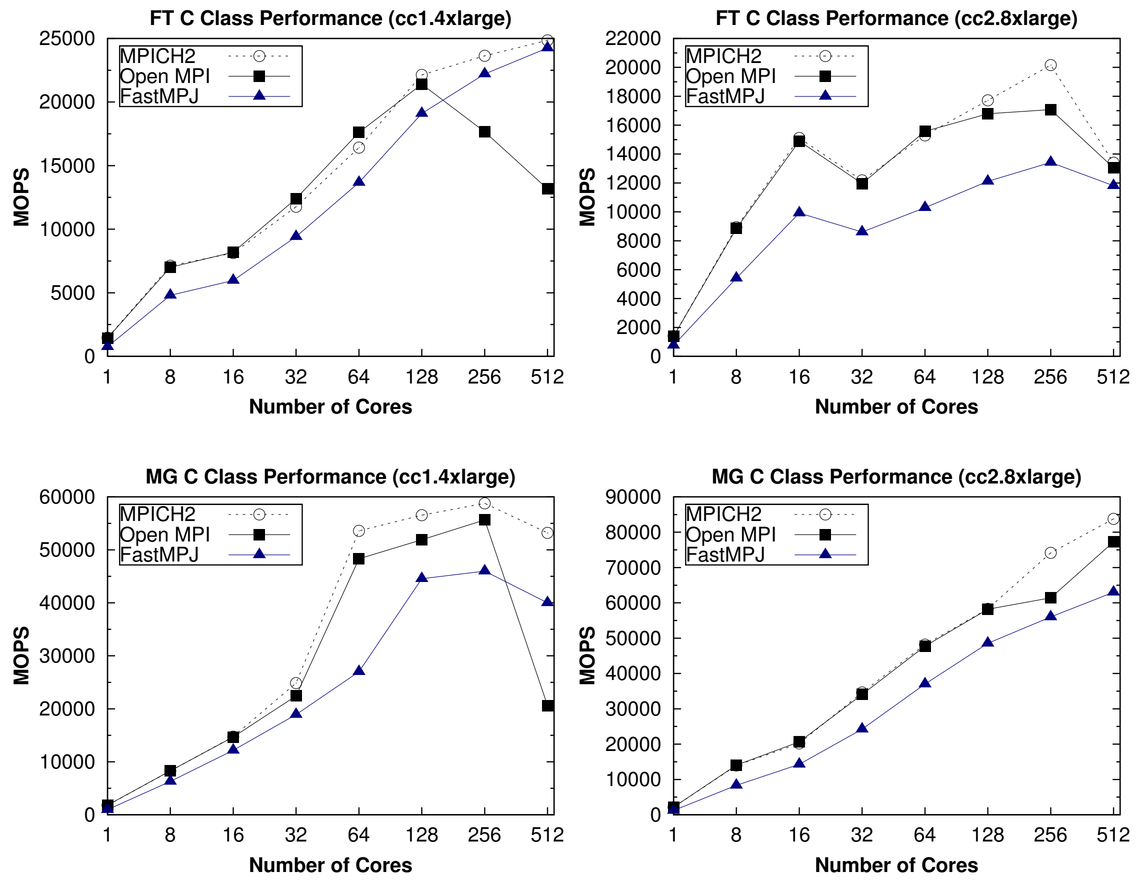


Figure 1.18: Performance of the NPB FT and MG kernels on Amazon EC2

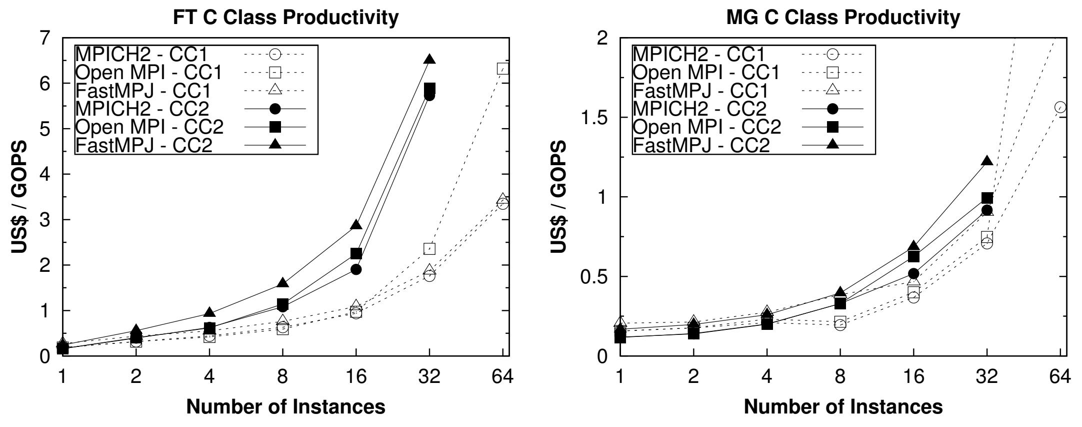


Figure 1.19: Productivity of NPB FT and MG kernels on Amazon EC2 (GOPS = GigaOperations Per Second)

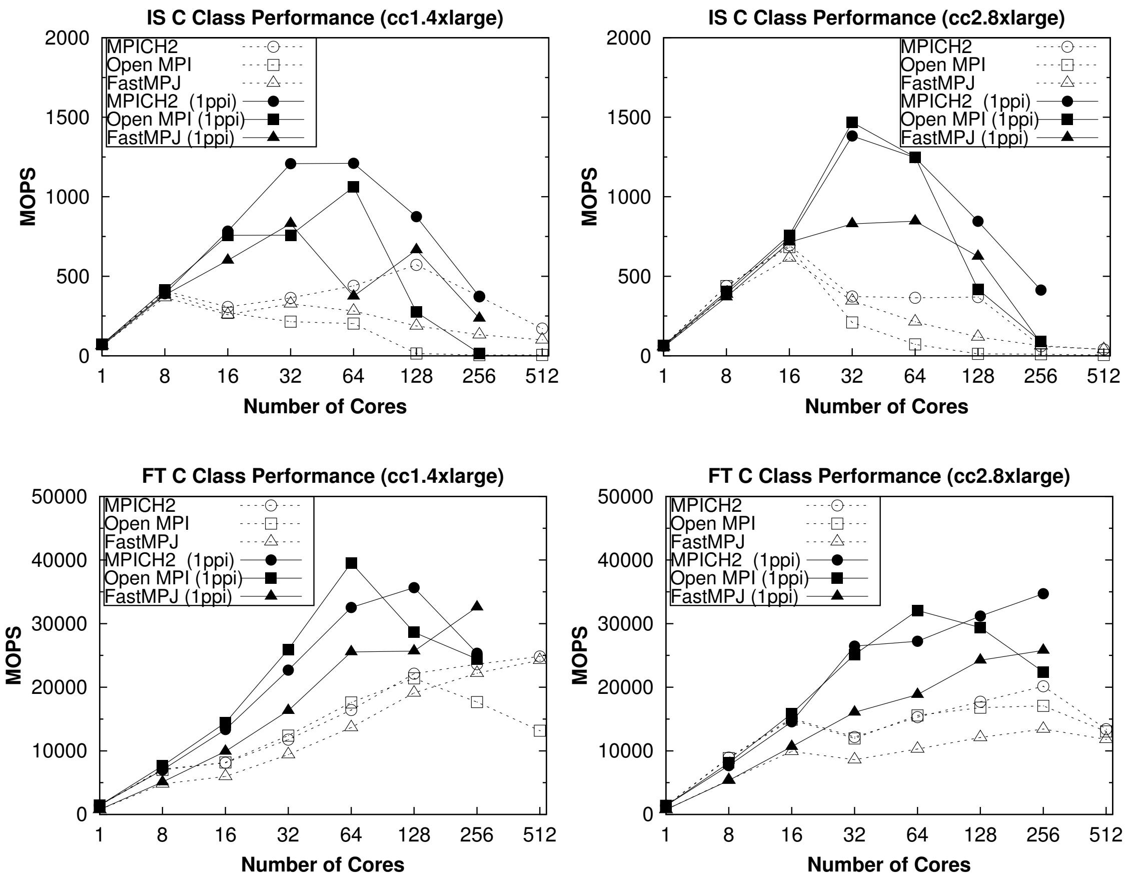


Figure 1.20: Performance of NPB IS and FT kernels on Amazon EC2 (ppi = processes per instance)

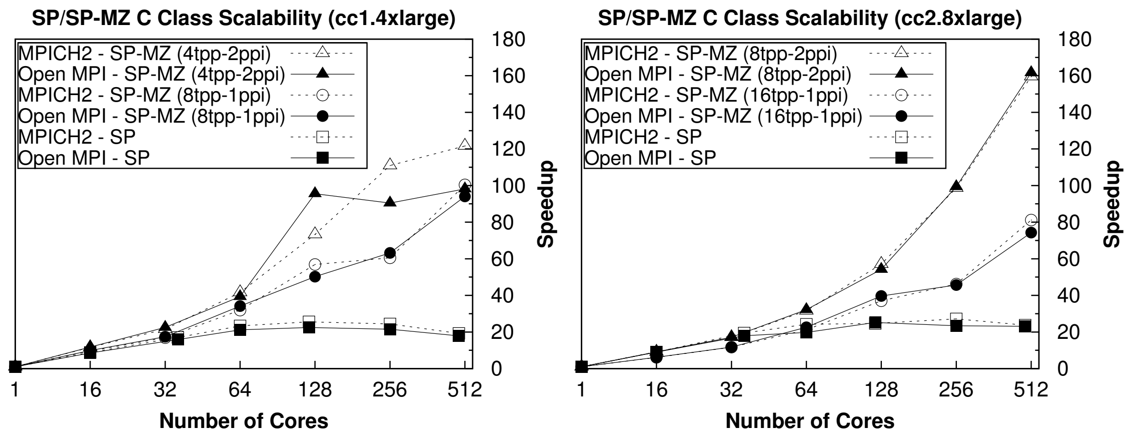


Figure 1.21: Scalability of the NPB/NPB-MZ SP kernel on Amazon EC2 (tpp = threads per process, ppi = processes per instance)

**Analysis of I/O Performance on the Amazon EC2 Cloud (Chapter 8)**

In the current era of Big Data, many scientific computing workloads are generating very large data sets that usually require a high number of computing resources to perform large-scale experiments into a reasonable time frame. In this scenario, scientific applications can be sensitive to CPU power, memory bandwidth/capacity, network bandwidth/latency as well as the performance of the I/O storage subsystem. This work presents a thorough evaluation of the I/O storage subsystem on the Amazon EC2 cloud to determine its suitability for I/O-intensive applications [32]. Generally, the instance types available in Amazon EC2 can access three types of storage: (1) the local block storage, known as ephemeral disk, where user data are lost once the instances are released (non-persistent storage); (2) off-instance Elastic Block Store (EBS), which are remote volumes accessible through the network that can be attached to an EC2 instance as block storage devices, and whose content is persistent; and (3) Simple Storage Service (S3), which is a distributed object storage system accessed through a web service that supports several interfaces such as Simple Object Access Protocol (SOAP). The ephemeral and EBS storage devices have different usage and capacity constraints. On the one hand, a CC1 instance can only mount up to two ephemeral disks of approximately 845 GB each one, whereas a CC2 instance can mount up to four disks of the aforementioned size. On the other hand, the number of EBS volumes attached to instances can be almost unlimited, and the size of a single volume can range from 1 GB to 1 TB. The first generation of storage-optimized instances (HI1) is a resource intended to provide very high storage I/O performance. In fact, the two ephemeral devices per HI1 instance are backed by SSD disks, which is the main differential characteristic of this resource. Additionally, HI1 provides with high levels of computational power (similar to CC1), a significant amount of memory (the same as CC2) and 10 Gigabit Ethernet network (as CC1 and CC2), as shown in Table 1.2.

The evaluation has been carried out at different levels (i.e., local/distributed file system, I/O interface and application levels) using representative benchmarks in order to evaluate the available low-level cloud storage devices: ephemeral disks and EBS volumes. The evaluation of S3 has not been considered since, unlike ephemeral and EBS devices, it lacks general file system interfaces required by scientific workloads so that the use of S3 is not transparent to the applications, and also due to

| | **HI1** |
|---|---|
| **CPU** | 2 × Intel Xeon E5620 Westmere-EP @2.40 GHz |
| **ECUs** | 35 |
| **#Cores** | 8 (16 with HyperThreading) |
| **Memory** | 60.5 GB DDR3-1066 |
| **Storage** | 2 TB (2 × SSD) |
| **API name** | hi1.4xlarge |
| **Price (Linux)** | $3.10 per hour |
| **Interconnect** | 10 Gigabit Ethernet (Full-bisection bandwidth) |
| **Virtualization** | Xen HVM 64-bit platform (PV drivers for I/O) |

Table 1.2: Description of the EC2 storage-optimized instances: HI1

the poor performance shown by previous works [54]. In addition, the performance of a representative distributed file system, NFS version 3, has been evaluated. NFS was selected as it is probably the most commonly used network file system, and it remains as the most popular choice for small and medium-scale clusters, which are the ones that are worth running in the cloud due to the poor network performance shown by our previous works. The evaluated instance types are HI1 and the HPC-aimed cluster instances (CC1 and CC2). Furthermore, widely extended I/O interfaces (POSIX, HDF5 [87] and MPI-IO [85]), which are directly implemented on top of file systems, have also been assessed as scientific workloads usually rely on them to perform I/O. Finally, the scalability of a representative parallel I/O code implemented on top of MPI-IO, the BT-IO kernel [93] from the NPB suite, has been analyzed at the application level both in terms of performance and cost metrics. The cost analysis has considered the three different purchasing options offered by Amazon EC2: (1) on-demand instances, which allow to access immediately computation power by paying a fixed hourly rate; (2) spot instances from the spot market, which allow customers to bid on unused Amazon EC2 capacity and run those instances for as long as their bid exceeds the current spot price (which changes periodically based on supply and demand); and (3) reserved instances for one- or three-year terms, which allow to receive a significant discount on the hourly charge. There are three reserved instance types: light, medium and heavy, that enable to balance the amount paid upfront with the effective hourly price.

The analysis of the performance results have shown that the available instance types and storage devices can present significant performance differences. In fact,

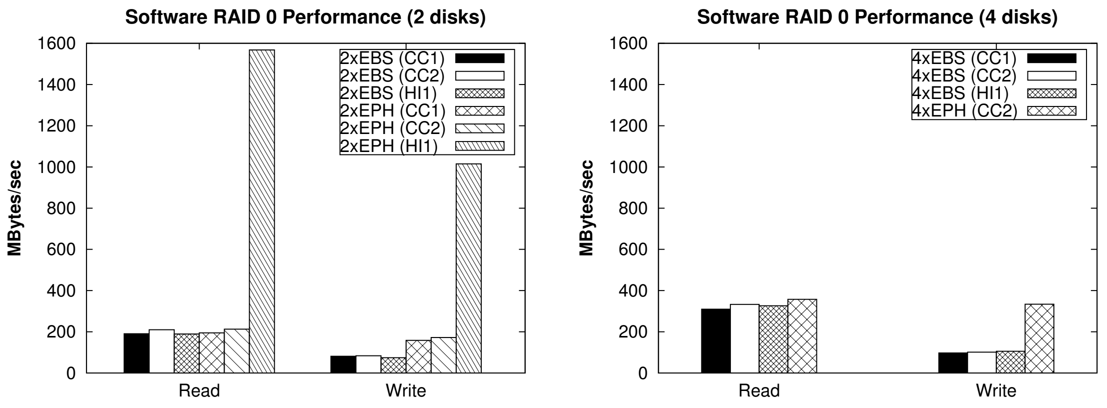


Figure 1.22: Software RAID 0 performance on CC1, CC2 and HI1 instances

this work has revealed that the use of ephemeral disks (labeled as "EPH" in the graphs) can provide better write performance than EBS volumes for CC1 and CC2, especially when software RAID is used (see Figure 1.22), thanks to the avoidance of additional network accesses to EBS. Obviously, the ephemeral SSD disks of the HI1 instances provide the best performance at the local file system level. Moreover, the analysis at the distributed file system level has shown that HI1 instances can provide significantly better NFS write performance than any other instance type, although the overall performance is ultimately limited by the poor network throughput (see Figure 1.23). Finally, the analysis of the performance/cost ratio of the BT-IO parallel application has shown that, although the use of the HI1 instance type provides slightly higher raw performance in terms of aggregated bandwidth (see Figure 1.24), it may not be the best choice when taking into account the incurred costs (see productivity results in Figure 1.25), especially when using reserved instances (see the right graph, labeled as "R"), which usually represents the lowest price that can be obtained for a particular instance type.

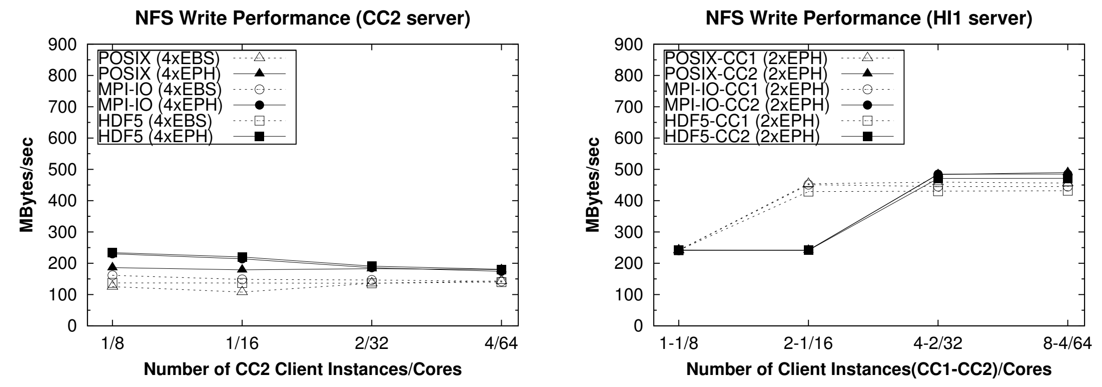


Figure 1.23: NFS write performance using CC2 and HI1 as server

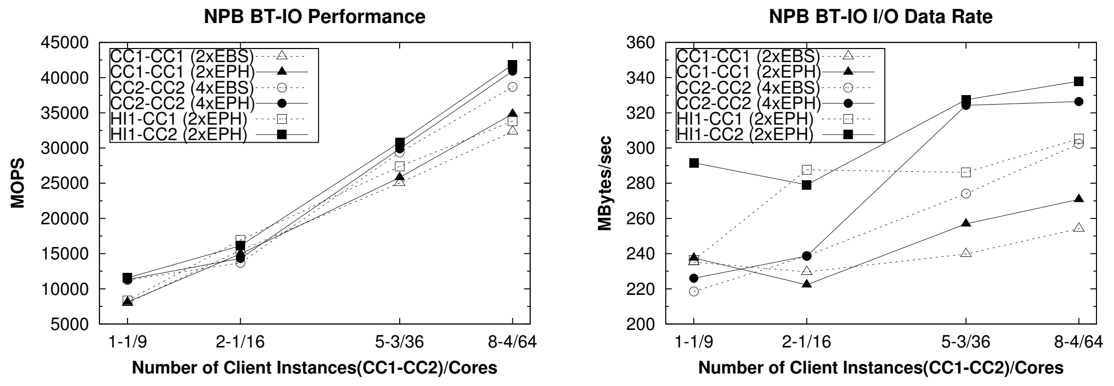


Figure 1.24: Performance of the NPB BT-IO kernel on Amazon EC2

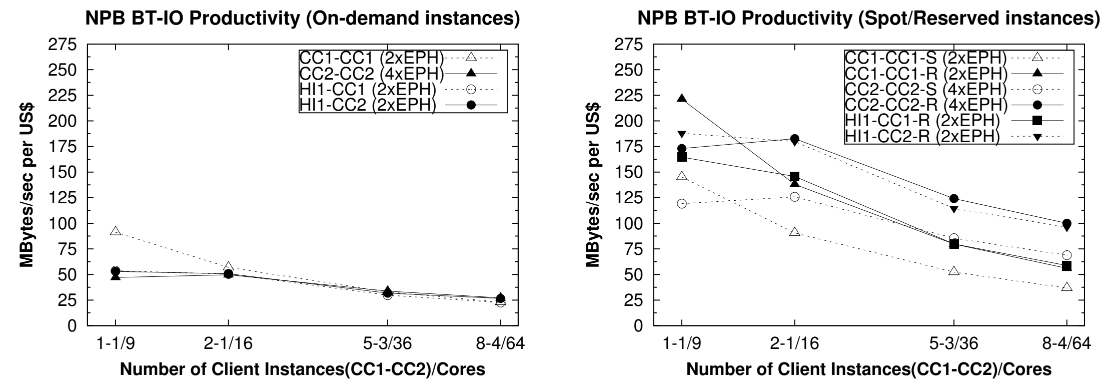


Figure 1.25: Productivity of the NPB BT-IO kernel using on-demand, spot (S) and reserved (R) EC2 instances

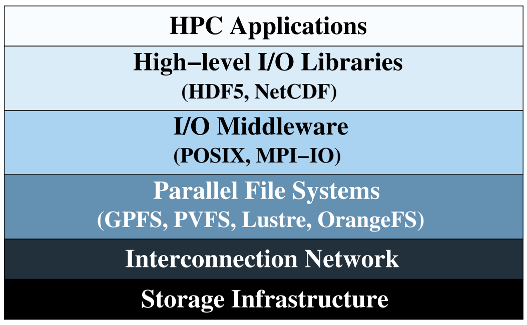
### Performance Evaluation of Data-Intensive Applications on a Public Cloud (Chapter 9)

Data-intensive computing [43] applications usually require a high number of computational resources together with the availability of a high-performance cluster file system for scalable performance. This work evaluates the full I/O software stack of data-intensive computing for running HPC and Big Data workloads on Amazon EC2 [37]. More specifically, the performance and cost-efficiency of four instance types with 10 Gigabit Ethernet has been characterized at several layers, ranging from low-level storage devices and cluster file systems up to the application level using representative data-intensive parallel codes and MapReduce-based workloads. The main contribution of this work with respect to the previous one presented in [32] is the evaluation of the more recent second generation of storage-optimized instances (HS1), whose differential characteristic is the provision of up to 24 ephemeral disks as local storage for very high storage density and high sequential read and write performance. In addition, the CR1 instance type has been evaluated, which has exactly the same capabilities as CC2 instances in terms of computational power, but providing four times more memory and two SSD-based ephemeral devices. More details on these instances are shown in Table 1.3. Other evaluated instance types are the first generation of storage-optimized instances (HI1, see Table 1.2) and CC2 instances (see Table 1.1).

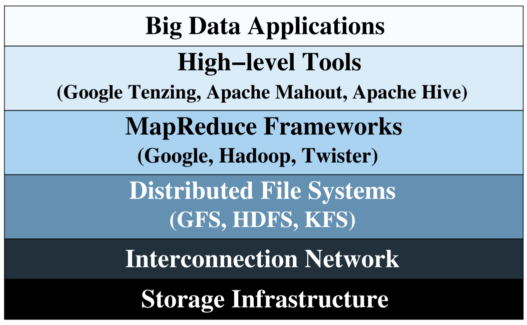
The evaluation has been conducted using representative benchmarks and appli-

| | **HS1** | **CR1** |
|---|---|---|
| **CPU** | 1 × Intel Xeon E5-2650 Sandy Bridge-EP @2 GHz | 2 × Intel Xeon E5-2670 Sandy Bridge-EP @2.60GHz |
| **ECUs** | 33.5 | 88 |
| **#Cores** | 8 (16 with HyperThreading) | 16 (32 with HyperThreading) |
| **Memory** | 117 GB DDR3-1600 | 244 GB DDR3-1600 |
| **Storage** | 48 TB (24 × HDD) | 240 GB (2 × SDD) |
| **API name** | hs1.8xlarge | cr1.8xlarge |
| **Price (Linux)** | $4.60 per hour | $3.50 per hour |
| **Interconnect** | 10 Gigabit Ethernet (Full-bisection bandwidth) | |
| **Virtualization** | Xen HVM 64-bit platform (PV drivers for I/O) | |

Table 1.3: Description of the HS1 and CR1 instance types



(a) I/O software stack for HPC applications



(b) Big Data analysis on top of MapReduce computing frameworks

Figure 1.26: I/O software stacks for data-intensive computing applications

cations at the different levels depicted in Figure 1.26. One of the key contributions of this work is the performance characterization at the cluster file system level. Regarding the HPC stack, the OrangeFS parallel file system [66], which is a relatively recent branch of the production-quality and widely extended Parallel Virtual File System (PVFS) [16], has been evaluated using the IOR benchmark [74] and the MPI-IO interface as representative I/O middleware. Regarding the Big Data software stack, the Intel HiBench suite [45] has been used for the evaluation of Apache Hadoop, selected as the most representative MapReduce computing framework. The Hadoop Distributed File System (HFDS) [75] has been evaluated using the Enhanced DFSIO benchmark included in the HiBench suite.

Figure 1.27 presents the aggregated bandwidth of OrangeFS for the write operation. These results have been obtained using a baseline cluster configuration that consists of 4 instances acting as I/O servers and multiple instances acting as clients. In these experiments, the number of physical compute cores in the cluster has been set to 128, as shown in Table 1.4. Hence, each client instance runs 8 (on HI1) or 16 (on CC2) parallel processes writing collectively a single shared file of 32 GB under different block sizes. As can be observed, the use of the largest block size is key to achieve high-performance parallel I/O, mainly using HI1 instances. In fact, the HI1-HI1 configuration achieves the best results, around 1600 MB/s, whereas the use of CC2 clients, both for HI1 and HS1 as servers, shows poor results. The CC2-CC2 configuration obtains around 800 MB/s, even outperforming HS1-CC2. The poor performance of HI1-CC2 and HS1-CC2 configurations, whose local storage devices

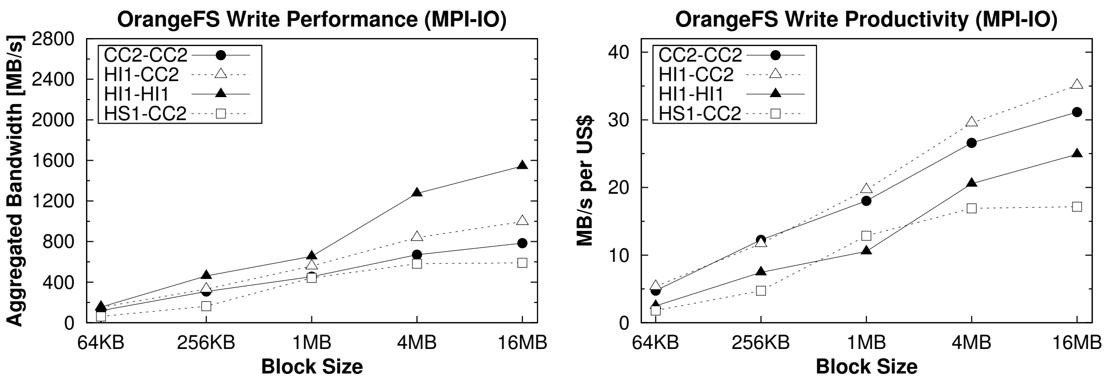


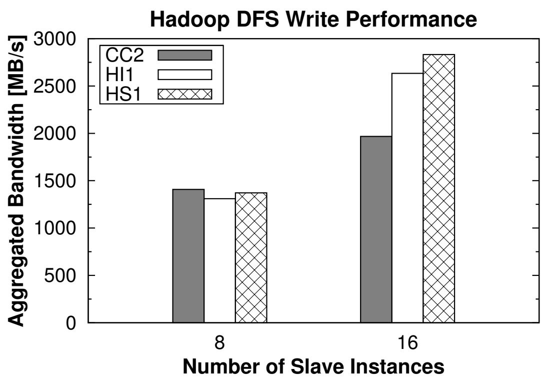
Figure 1.27: OrangeFS results using 4 I/O servers and 128 cores

| HPC Cluster | #I/O Servers | #Clients | #Compute Cores | Hourly Cost |
|---|---|---|---|---|
| **CC2-CC2** | 4 × CC2 | 8 × CC2 | 128 | $24 |
| **HI1-CC2** | 4 × HI1 | 8 × CC2 | 128 | $28.4 |
| **HI1-HI1** | 4 × HI1 | 16 × HI1 | 128 | $62 |
| **HS1-CC2** | 4 × HS1 | 8 × CC2 | 128 | $34.4 |

Table 1.4: Hourly cost of the EC2-based HPC clusters

are the fastest, is due to the impossibility of locating different instance types in the same placement group, so when using CC2 clients with HI1 and HS1 servers the network performance drops severely. Taking costs into account, the HI1-CC2 option becomes the best configuration from 256 KB on, due to the use of client instances cheaper than the HI1-HI1 configuration. Even CC2-CC2 seems to be a good choice instead of HI1-HI1, as it is the cheapest cluster under evaluation.

Figure 1.28 shows the aggregated bandwidth of HDFS for the write operation using a baseline cluster that consists of one instance acting as master node (running JobTracker/NameNode) and multiple instances acting as slave nodes (running Task-Trackers/DataNodes), all connected to the 10 Gigabit Ethernet network. In these experiments, two different cluster sizes have been evaluated using 8 and 16 slave instances (see Table 1.5), together with the master node of the same instance type, and thereby all instances located in the same placement group. The use of 8-slave clusters shows similar HDFS bandwidths (around 1400 MB/s) and thus the CC2-based cluster is the most cost-effective in terms of productivity. However, HI1 and HS1 instances obtain 34% and 44% more aggregated bandwidth, respectively, than



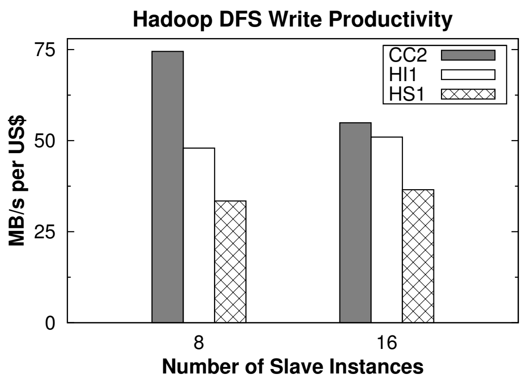


Figure 1.28: HDFS results using 8 and 16 slave instances

| Hadoop Cluster | #M/R tasks per slave (8-16 slaves) | Hourly Cost (8-16 slaves) |
|---|---|---|
| **CC2-based** | 12/4 (96/32 - 192/64) | \$18 - \$34 |
| **HI1-based** | 6/2 (48/16 - 96/32) | \$27.9 - \$52.7 |
| **HS1-based** | 6/2 (48/16 - 96/32) | \$41.4 - \$78.2 |

Table 1.5: Number of (M)ap/(R)educe tasks and hourly cost of the Hadoop clusters

CC2 instances when using 16-slave clusters. Nevertheless, CC2 remains as the most productive choice, although followed closely by HI1, whereas the HS1-based cluster, which is the most expensive, remains as the least competitive. Note that while the storage-optimized instances have slightly increased their productivity when doubling the number of slaves, CC2 has decreased 27%.

Finally, the performance and cost of two I/O-intensive HPC applications (BT-IO [93] and FLASH-IO [39, 41]) and four MapReduce workloads selected from the HiBench suite (Sort, WordCount, Mahout PageRank and Hive Aggregation) has been analyzed at the application level. As a representative example of the Big Data software stack evaluation, Figure 1.29 shows the results for the Hadoop Sort. The use of 8-slave clusters shows that CC2 is able to outperform HI1 and HS1 configurations by 20%, thanks to the higher map/reduce capacity of the CC2 cluster due to the availability of more CPU resources (see Table 1.5), which seem to be of great importance, especially during data compression. However, CC2 only reduces its execution time by 22% when doubling the number of slaves, whereas HI1 and HS1 reduce it by 40% and 42%, respectively. This allows storage-optimized instances to

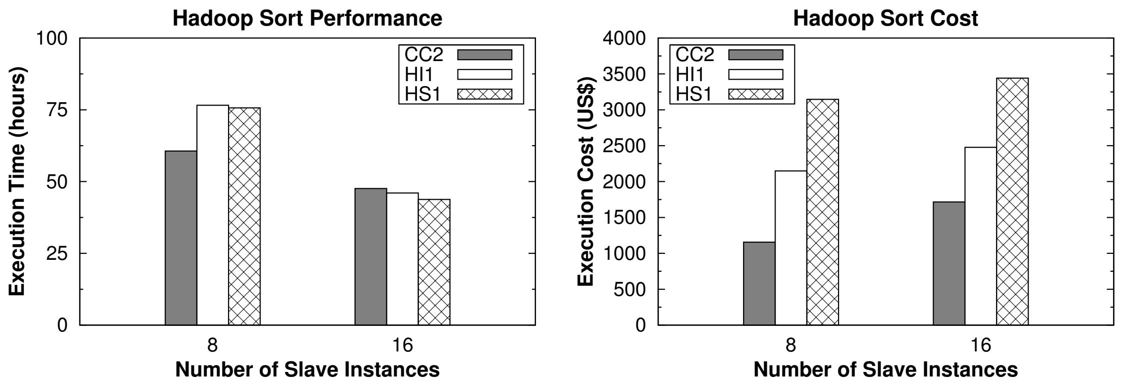


Figure 1.29: Hadoop Sort performance and execution cost

slightly outperform CC2 when using 16 slaves, due to their higher HDFS bandwidth, even taking into account that they have half the map/reduce capacity of CC2. Nevertheless, the cost of the CC2 cluster continues to be the lowest, a pattern that is maintained in the remaining workloads. If the comparison is done using the same map/reduce capacity, the 16-slave clusters using HI1 and HS1 instances outperform the 8-slave CC2 cluster by 24% and 27%, respectively (see Table 1.5 and left graph in Figure 1.29). However, these improvements are not enough to turn optimized-storage instances into interesting options when considering the associated costs.

The main results of this work have shown that data-intensive applications can benefit from tailored Amazon EC2-based virtual clusters, enabling users to obtain the highest performance and cost-effectiveness in the cloud. The analysis of the experimental results points out that the unique configurability and flexibility advantage offered by Amazon EC2, almost impossible to achieve in traditional platforms, can benefit significantly data-intensive applications, and it is critical for increasing performance and/or reduce costs. In fact, this work has revealed that the suitability of using EC2 resources for running data-intensive applications is highly workload-dependent. Furthermore, the most suitable configuration for a given application heavily depends on whether the main aim is to obtain the maximum performance or, instead, minimize the cost (or maximize the productivity). Therefore, a key contribution of this work is an in-depth study that provides guidelines for scientists and researchers to increase significantly the performance (or reduce the cost) of their data-intensive applications in the Amazon EC2 cloud.

### General-Purpose Computation on GPUs for Cloud Computing (Chapter 10)

In the last years, General-Purpose computation on GPUs (GPGPU) has gained much attention from the HPC and scientific community due to its massive parallel processing power, thus becoming an important programming model in HPC. The massively parallel GPU architecture, together with its high floating-point performance and memory bandwidth is well suited for many workloads, even outperforming multi-core processors. In fact, GPU clusters [56], which are usually programmed using a hybrid parallel paradigm (MPI+CUDA/OpenCL), are currently gaining high popularity. The Amazon EC2 cloud infrastructure provides a GPU family of cluster instances (CG1), which are also equipped with a 10 Gigabit Ethernet network. Instances of this family provide exactly the same hardware capabilities as the CC1 instance type in terms of computational power, memory capacity, storage devices and network performance. The differential feature of the CG1 instances is the provision of two NVIDIA Tesla GPUs per instance, which are intended for GPGPU codes (see description of CG1 in Table 1.6). Note that these instances rely on Xen PCI passthrough for accessing directly the GPU, while the access to the network is also paravirtualized as for previous cluster instance types.

This work has evaluated GPGPU on Amazon EC2 using a cluster of up to 32 CG1 instances [34] (i.e., up to 64 GPUs and 256 cores have been used), whose main

| | **CG1** |
|---|---|
| **CPU** | 2 × Intel Xeon X5570 Nehalem-EP @2.93 GHz (46.88 GFLOPS DP each CPU) |
| **ECUs** | 33.5 |
| **#Cores** | 8 (16 with HyperThreading) |
| **Memory** | 22 GB DDR3-1066 |
| **GPUs** | 2 × NVIDIA Tesla “Fermi” M2050 (515 GFLOPS DP each GPU) |
| **Storage** | 1690 GB (2 × HDD) |
| **API name** | cg1.4xlarge |
| **Price (Linux)** | $2.10 per hour |
| **Interconnect** | 10 Gigabit Ethernet (Full-bisection bandwidth) |
| **Virtualization** | Xen HVM 64-bit platform (PV drivers for I/O) |

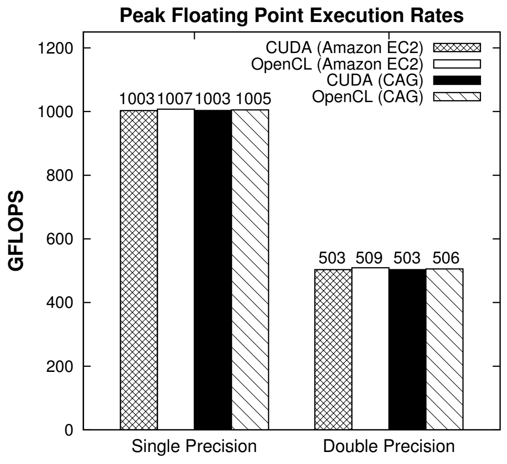
Table 1.6: Description of the EC2 cluster GPU instances: CG1

| **Number of CG1 instances** | 32 |
|---|---|
| **Interconnect** | 10 Gigabit Ethernet |
| **Total ECUs** | 1072 |
| **Total CPU cores** | 256 (3 TFLOPS DP) |
| **Total GPUs** | 64 (32.96 TFLOPS DP) |
| **Total FLOPS** | 35.96 TFLOPS DP |

Table 1.7: Characteristics of the CG1-based cluster

characteristics are presented in Table 1.7. More especifically, a CG1 instance has 8 cores, each of them capable of executing 4 floating-point operations per clock cycle in double precision (DP), hence 46.88 GFLOPS per processor, 93.76 GFLOPS per node and 3 TFLOPS in the 32-node (256-core) cluster. Moreover, each GPU has 3 GB of memory and a peak performance of 515 GFLOPS, hence 32.96 TFLOPS in the cluster. Aggregating CPU and GPU theoretical peak performances the entire cluster provides 35.96 TFLOPS. The evaluation has been conducted using representative GPGPU benchmarks and applications: 12 synthetic kernels using CUDA and OpenCL codes, which have been selected from two representative benchmark suites (SHOC [23] and Rodinia [21]), two real-world distributed applications that take advantage of GPUs (NAMD [69] and MC-GPU [4]) and the High-Performance Linpack (HPL) [28], which is the reference benchmark for the TOP500 list [89]. Additionally, the synthetic kernels have been executed on a single GPU of a non-virtualized testbed (named CAG) with the same GPU model in order to assess the overhead of the virtualization layer.

The analysis of the results has shown that GPGPU is a viable option for HPC in the cloud despite the significant impact that virtualized environments still have on network overhead, which still hampers the adoption of GPGPU communication-intensive applications. In fact, the peak floating-point benchmark results (see left graph in Figure 1.30) are very close to the theoretical peak performance on this GPU model (515 GFLOPS), showing insignificant differences between Amazon and the non-virtualized testbed (CAG) both for single and double precision tests. Hence, computationally intensive applications with algorithms that can be efficiently exploited by the GPU (e.g., MC-GPU) can take full advantage of their execution on CG1 instances, as shown in the left graph in Figure 1.32. These applications can benefit from the efficient access to the GPU accelerators without any significant performance penalty, except when accessing memory intensively, where a small penalty



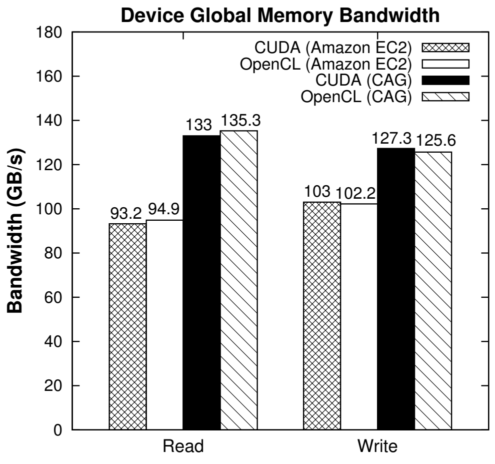
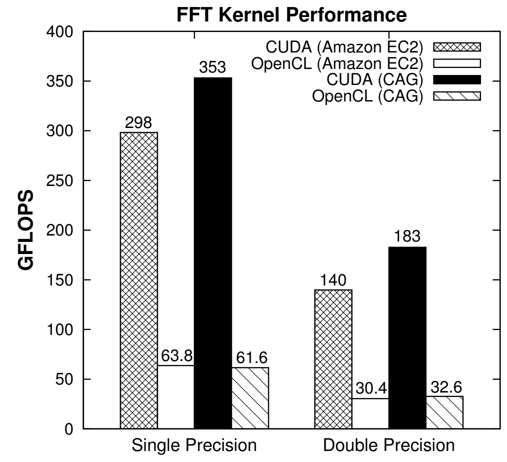


Figure 1.30: Performance of low-level GPU benchmarks on Amazon EC2 and CAG



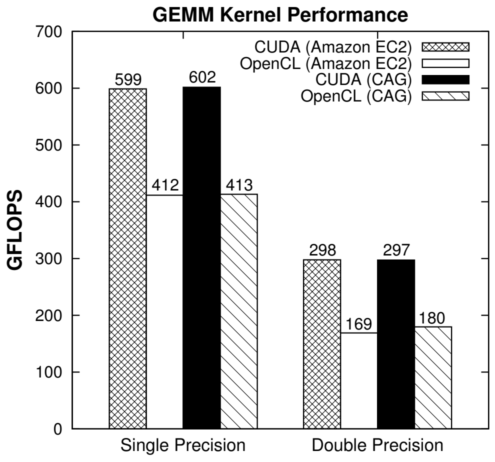


Figure 1.31: Performance of FFT and GEMM kernels on Amazon EC2 and CAG

can be observed as CG1 instances have the Error Correcting Code (ECC) memory protection enabled, which slightly limits the memory access bandwidth (see right graph in Figure 1.30). This causes that memory-bound kernels (e.g., FFT) achieve better performance on the CAG testbed, which has the ECC memory protection deactivated, as shown in Figure 1.31 (left graph), while compute-bound kernels (e.g., see GEMM in the right graph) obtain similar performance on both testbeds (note that the FFT implementation in OpenCL is less optimized than the CUDA counterpart for the Tesla "Fermi" architecture).

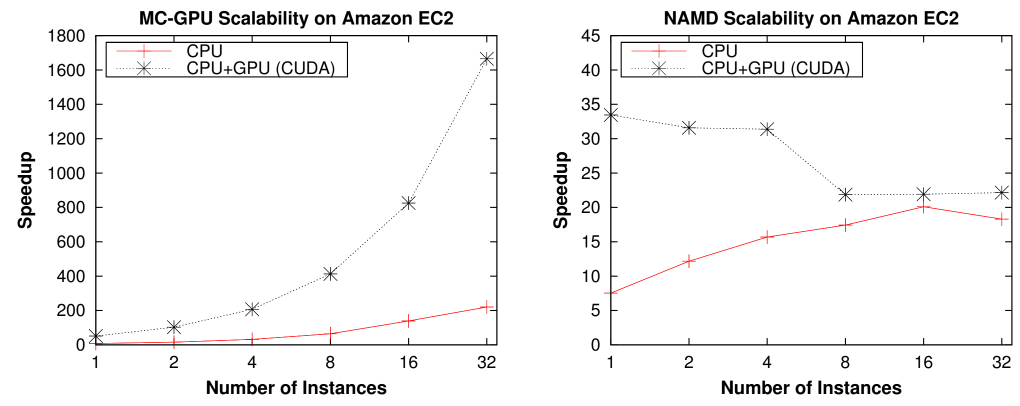


Figure 1.32: MC-GPU and NAMD results on CG1 instances

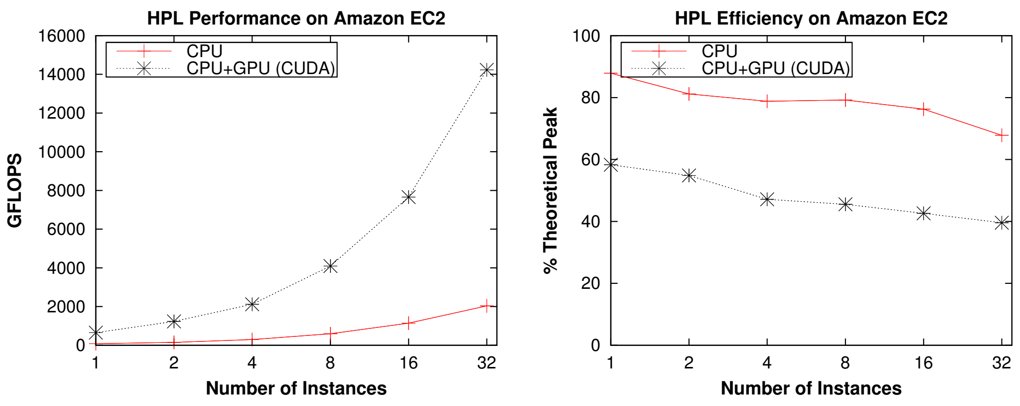


Figure 1.33: HPL benchmark results on CG1 instances

Nevertheless, communication-intensive codes (e.g., NAMD) still suffer from the overhead of the paravirtualized network access, which can reduce scalability significantly. In fact, this performance bottleneck is especially important for the CPU+GPU implementation, as it computes faster than the CPU version and therefore its communication requirements are even higher. Hence, the CPU+GPU version of NAMD is not able to scale when using more than one CG1 instance, while the CPU version obtains moderate scalability using up to 16 instances (see right graph in Figure 1.32). This overhead has also limited the HPL performance results, especially when using 32 CG1 instances as the efficiency drops below 40% for the CPU+GPU benchmark (14.23 out of 35.96 peak TFLOPS, see Figure 1.33). Therefore, a direct access to the network hardware is key to reduce the communication overhead in virtualized environments and make on-demand HPC in the cloud a widespread option.

# Part I

# Design of Low-Latency Java Communication Middleware on High-Speed Networks

# Chapter 2

# Design of Scalable Java Communications over InfiniBand

The content of this chapter corresponds to the following journal paper:

**- Title:** Design of scalable Java message-passing communications over InfiniBand

**- Authors:** Roberto R. Expósito, Guillermo L. Taboada, Juan Touriño, Ramón Doallo

**- Journal:** Journal of Supercomputing

**- Editorial:** Springer Science+Business Media

**- ISSN:** 0920-8542

**- Year:** 2012

**- Volume(Number):Pages:** 61(1):141–165

**- DOI:** 10.1007/s11227-011-0654-9

The final publication is available at `http://link.springer.com/article/10.1007%2Fs11227-011-0654-9`. A copy of the accepted paper has been included next.

# Design of scalable Java message-passing communications over InfiniBand

**Roberto R. Expósito · Guillermo L. Taboada · Juan Touriño · Ramón Doallo**

**Abstract** This paper presents `ibvdev` a scalable and efficient low-level Java message-passing communication device over InfiniBand. The continuous increase in the number of cores per processor underscores the need for efficient communication support for parallel solutions. Moreover, current system deployments are aggregating a significant number of cores through advanced network technologies, such as InfiniBand, increasing the complexity of communication protocols, especially when dealing with hybrid shared/distributed memory architectures such as clusters. Here, Java represents an attractive choice for the development of communication middleware for these systems, as it provides built-in networking and multithreading support. As the gap between Java and compiled languages performance has been narrowing for the last years, Java is an emerging option for High Performance Computing (HPC).

The developed communication middleware `ibvdev` increases Java applications performance on clusters of multicore processors interconnected via InfiniBand through: (1) providing Java with direct access to InfiniBand using InfiniBand Verbs API, somewhat restricted so far to MPI libraries; (2) implementing an efficient and scalable communication protocol which obtains start-up latencies and bandwidths similar to MPI performance results; and (3) allowing its integration in any Java parallel and distributed application. In fact, it has been successfully integrated in the Java messaging library MPJ Express.

R.R. Expósito · G.L. Taboada (✉) · J. Touriño · R. Doallo
Computer Architecture Group, Dept. of Electronics and Systems, University of A Coruña, A Coruña, Spain
e-mail: taboada@udc.es

R.R. Expósito
e-mail: rreye@udc.es

J. Touriño
e-mail: juan@udc.es

R. Doallo
e-mail: doallo@udc.es

The experimental evaluation of this middleware on an InfiniBand cluster of multicore processors has shown significant point-to-point performance benefits, up to 85% start-up latency reduction and twice the bandwidth compared to previous Java middleware on InfiniBand. Additionally, the impact of `ibvdev` on message-passing collective operations is significant, achieving up to one order of magnitude performance increases compared to previous Java solutions, especially when combined with multithreading. Finally, the efficiency of this middleware, which is even competitive with MPI in terms of performance, increments the scalability of communications intensive Java HPC applications.

**Keywords** Message-Passing in Java (MPJ) · InfiniBand · Multicore architectures · High performance computing · Remote Direct Memory Access (RDMA) · Performance evaluation

## 1 Introduction

Java is the leading programming language both in academia and industry environments, and it is an emerging alternative for High Performance Computing (HPC) [1] due to its appealing characteristics: built-in networking and multithreading support, object orientation, automatic memory management, platform independence, portability, security, an extensive API, and a wide community of developers. Furthermore, in the era of multicore processors, the use of Java threads is considered a feasible option to harness the performance of these processors.

Java initially was severely criticized for its poor computational performance [2], but the performance gap between Java and native (compiled) languages like C or Fortran has been narrowing for the last years. The main reason is that the Java Virtual Machine (JVM), which executes Java applications, is now equipped with Just-in-Time (JIT) compilers that obtain native performance from Java bytecode. Nevertheless, the tremendous improvement in its computational performance is not enough for Java to be a successful language in the area of parallel computing, as the performance of the communications is also essential to achieve high scalability in Java for HPC.

Message-passing is the most widely used parallel programming paradigm as it is highly portable, scalable, and usually provides good performance. It is the preferred choice for parallel programming distributed memory systems such as multicore clusters, currently the most popular system deployments due to their scalability, flexibility, and interesting cost/performance ratio. Here, Java represents an attractive alternative to languages traditionally used in HPC, such as C or Fortran, for the development of applications for these systems as it provides built-in networking and multithreading support, key features for taking full advantage of hybrid shared/distributed memory architectures. Thus, Java can use threads in shared memory (intranode) and its networking support for distributed memory (internode) communications.

The increasing number of cores per system demands efficient and scalable message-passing communication middleware. However, up to now Message-Passing in Java (MPJ) implementations have been focused on providing portable communication devices, rather than concentrate on developing efficient low-level communication devices on high-speed networks. The lack of efficient support for high-speed networks in Java, due to its inability to control the underlying specialized hardware, results in lower performance than MPI, especially for short messages. This paper presents a scalable and efficient Java low-level message-passing communication device, `ibvdev`, aiming to its integration in MPJ implementations in order to provide higher performance on InfiniBand multicore clusters. In fact, it has been already integrated successfully in the MPJ library MPJ Express [3] (http://mpj-express.org).

The structure of this paper is as follows: Sect. 2 presents InfiniBand background information. Section 3 introduces the related work. Section 4 describes the design and implementation of the efficient `ibvdev` middleware, covering in detail the operation of the communication algorithms that provide the highest performance over InfiniBand. Section 5 shows the performance results of the implemented library on an InfiniBand multicore cluster. The evaluation consists of a micro-benchmarking of point-to-point and collectives primitives, as well as a kernel/application benchmarking in order to analyze the impact of the use of the library on their overall performance. Section 6 summarizes our concluding remarks.

## 2 Java communications over InfiniBand

### 2.1 InfiniBand architecture

The InfiniBand Architecture (IBA) [4] defines a System Area Network (SAN) for interconnecting processing nodes and I/O nodes. In an InfiniBand network, processing nodes and I/O nodes are connected to the fabric by Channel Adapters (CA). Channel Adapters usually have programmable DMA engines with protection features. There are two kinds of channel adapters: Host Channel Adapter (HCA) and Target Channel Adapter (TCA). HCAs sit on processing nodes and TCAs connect I/O nodes to the fabric.

The InfiniBand communication stack consists of different layers. The interface presented by channel adapters to consumers belongs to the transport layer. A queue-based model is used in this interface. A Queue Pair (QP) in InfiniBand Architecture consists of two queues: a send queue and a receive queue. The send queue holds instructions to transmit data and the receive queue holds instructions that describe where received data has to be placed. Communication operations are described in Work Queue Requests (WQR), or descriptors, and submitted to the work queue. Once submitted, a Work Queue Request becomes a Work Queue Element (WQE). WQEs are executed by Channel Adapters. The completion of work queue elements is reported through Completion Queues (CQs). Once a work queue element is finished, a completion entry is placed in the associated completion queue. Applications can check the completion queue to see if any work queue request has been finished.

#### *2.1.1 Channel and memory semantics*

InfiniBand Architecture supports both channel and memory semantics. In channel semantics, send/receive operations are used for communication. To receive a message, the programmer posts a receive descriptor which describes where the message should be put at the receiver side. At the sender side, the programmer initiates the send operation by posting a send descriptor. The send descriptor describes where the source data is but does not specify the destination address at the receiver side. When the message arrives at the receiver side, the hardware uses the information in the receive descriptor to put data in the destination buffer. Multiple send and receive descriptors can be posted and they are consumed in FIFO order. The completion of descriptors are reported through CQs.

In memory semantics, Remote Direct Memory Access (RDMA) write and RDMA read operations are used instead of send and receive operations. These operations are one-sided and do not incur software overhead at the other side. The sender initiates RDMA operations by posting RDMA descriptors. A RDMA descriptor contains both the local data source address and the remote data destination address. At the sender side, the completion of a RDMA operation can be reported through CQs. The operation is transparent to the software layer at the receiver side.

Both communication semantics require communication memory to be registered with InfiniBand hardware and pinned in memory. The registration operation involves informing the network-interface of the virtual to physical address translation of the communication memory. The pinning operation requires the operating system to mark the pages corresponding to the communication memory as non-swappable. Thus, communication memory stays locked in physical memory, and the network-interface can access it as desired.

### *2.1.2 Transport services*

There are five transport modes defined by the InfiniBand specification: Reliable Connection (RC), eXtended Reliable Connection (XRC), Reliable Datagram (RD), Unreliable Connection (UC), and Unreliable Datagram (UD). All transports provide a checksum verification.

Reliable Connection (RC) is the most popular transport service for implementing MPI over InfiniBand. As a connection-oriented service, a QP with RC transport must be dedicated to communicating with only one other QP. A process that communicates with $N$ other peers must have at least $N$ QPs created. The RC transport provides almost all the features available in InfiniBand, most notably reliable send/receive, RDMA and atomic operations.

RC transport makes no distinction between connecting a process (generally one per core for MPI) and connecting a node. Thus, the associated resource consumption increased directly in relation to the number of cores in the system. To address this problem eXtended Reliable Connection (XRC) was introduced. Instead of having a per-process cost, XRC was designed to allow a single connection from one process to an entire node. XRC provides the services of the RC transport, but defines a very different connection model and method for determining data placement on the receiver in channel semantics. When using the RC transport, the connection model is purely based on processes. By contrast, XRC allows connection optimization based on the location of a process. The node of the peer to connect to is now taken into account, so instead of requiring a new QP for each process, now each process only needs to have

**Table 1** Operations available for each transport service

| Operation | RC | XRC | UC | RD | UD |
|---|---|---|---|---|---|
| Send (with immediate) | X | X | X | X | X |
| Receive | X | X | X | X | X |
| RDMA write (with immediate) | X | X | X | X | |
| RDMA read | X | X | | X | |
| Atomic | X | X | | X | |

one QP per node to be fully connected. This reduces the number of QPs required by a factor of the number of cores per node.

Unreliable Connection (UC) provides a connection-oriented service with no guarantees of ordering or reliability. It supports RDMA write capabilities and sending messages larger than the Maximum Transmission Unit (MTU) size. Being connection-oriented in nature, every communicating peer requires a separate QP. In regard to resources required, it is identical to RC, while no providing reliable service. Thus, it appears unattractive for implementing MPI over this transport.

Unreliable Datagram (UD) is a connection-less and unreliable transport, the most basic transport specified for InfiniBand. As a connection-less transport, a single UD QP can communicate with any number of other UD QPs. However, the UD transport has a number of limitations. The UD transport does not provide any reliability: lost packets are not reported and the arrival order is not guaranteed. However, this can be solved relying on Reliable Datagram (RD). Moreover, UD transport does not enable RDMA. All communication must be performed using channel semantics, i.e., send/receive.

Table 1 shows the available operations for each transport service, since not all transport services support all operations, which has to be taken into account for a message-passing middleware implementation.

### *2.1.3 Shared receive queues*

Shared Receive Queues (SRQs) were introduced in the InfiniBand 1.2 specification to address scalability issues with InfiniBand memory usage. In order to receive a message on a QP, a receive buffer must be posted in the Receive Queue (RQ) of that QP. To achieve high-performance, MPI implementations prepost buffers to the RQ to accommodate unexpected messages. When using the RC transport of InfiniBand, one QP is required per communicating peer. However, this task of preposting receives on each QP can have very high memory requirements for communication buffers. Recognizing that such buffers could be pooled, SRQ support was added so instead of connecting a QP to a dedicated RQ, buffers could be shared across QPs. In this method, a smaller pool can be allocated and then refilled on demand instead of preposting on each connection.

## 2.2 Message-passing communication devices

Message-passing libraries usually support new transport protocols through the use of pluggable low-level communication devices, such as Abstract Device Interface (ADI)

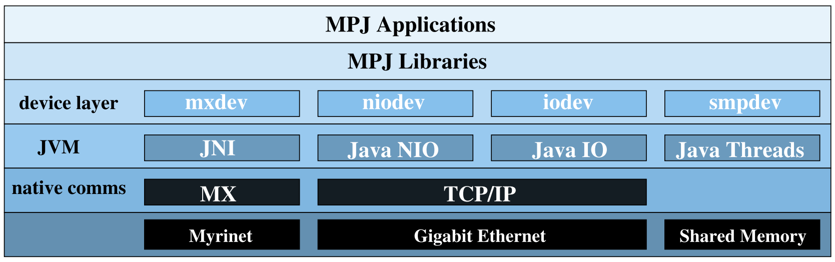


**Fig. 1** Communications support of MPJ applications

in MPICH, Byte Transfer Layer (BTL) in OpenMPI, and `xdev` [5] in MPJ Express. These communication devices abstract the particular operation of a communication protocol, such Myrinet eXpress (MX), uDAPL (user Direct Access Programming Library), InfiniBand Verbs (IBV), Shared Memory, or SCTP (Stream Control Transmission Protocol), conforming to an API on top of which the message-passing library implements its communications.

Figure 1 presents an overview of the communications support of MPJ applications on the high-speed Myrinet network, on Gigabit Ethernet, and on shared memory. From top to bottom, MPJ applications rely on MPJ libraries, whose communication support is implemented in the device layer. Current Java communication devices are implemented either on JVM threads (`smpdev`, a multithreading device), on sockets over the TCP/IP stack (`niodev` on Java NIO sockets and `iodev` on Java IO sockets), or on native communication layers such as Myrinet eXpress (`mxdev`, a device on MX).

Regarding InfiniBand, up to now no direct support was made available for MPJ applications to fully exploit the communication capability of InfiniBand networks. This lack of direct InfiniBand support in Java requires the use of upper layer protocols such as IPoIB [6] (IP over InfiniBand) TCP emulation, as shown in Fig. 2, or SDP (Sockets Direct Protocol), the high performance native sockets library on InfiniBand. However, the use of IPoIB, the only communication library that fully supports Java over InfiniBand, shows quite poor performance [7]. Moreover, when relying on SDP the performance generally improves, but this is not always possible. Regarding MPI libraries, their direct InfiniBand support has been implemented some years ago on top of InfiniBand Verbs (IBV) API (see Fig. 2), achieving very high performance results. Therefore, our objective is the implementation of the direct InfiniBand support in Java on IBV through the development of a low-level Java communication device that can take advantage of InfiniBand RDMA transfers, thus outperforming significantly previous Java support on InfiniBand.

## 3 Related work

Current research on efficient Java communication libraries over InfiniBand is, to our knowledge, restricted to Jackal, Aldeia, Java Fast Sockets (JFS), Jdib, and uStream projects, next presented. Jackal [8] is a Java DSM (Distributed Shared Memory) middleware for clusters with InfiniBand Verbs support, embracing also RDMA transfers,

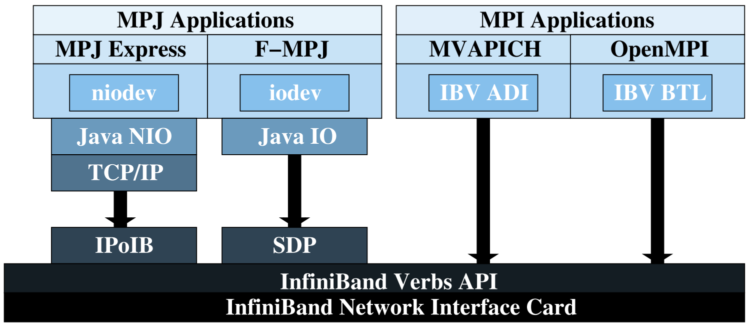


**Fig. 2** MPI/MPJ applications support on InfiniBand

but it does not provide any API to Java developers as it only implements data transfers specifically for Jackal. Aldeia [9] is a proposal of an asynchronous sockets communication layer over InfiniBand whose preliminary results were encouraging, but requires an extra-copy, which incurs an important overhead to provide asynchronous write operations, whereas the read method is synchronous.

JFS [10] is our high performance Java socket implementation for efficient shared memory and high-speed networks support. JFS relies on SDP (see Fig. 2) to support Java communication over InfiniBand. Moreover, JFS avoids the need for primitive data type array serialization and reduces buffering and unnecessary copies. Nevertheless, the use of the sockets API is a significant drawback to support efficient message-passing communications.

Jdib [11, 12] (Java Direct InfiniBand) is a Java encapsulation of IBV API which maximizes Java communication performance using directly, through Java Native Interface (JNI), the InfiniBand RDMA mechanism. The main contribution of Jdib is its direct access to RDMA, providing to performance-concerned developers, for the first time, a Java RDMA API. Thus, Jdib significantly outperforms its alternatives, currently limited to IPoIB- and SDP-based solutions. The main drawbacks of Jdib are its low-level API and the JNI overhead incurred for each Jdib operation.

uStream [13] is a user-level stream protocol implemented on top of IBV that provides a higher level API than Jdib. In fact, uStream abstracts developers from the most tedious operations in Jdib, such as the buffer management, synchronization and the use of the IBV API, while fully exploiting InfiniBand RDMA performance. Therefore, uStream is much more effective and easier to use than Jdib for building parallel and distributed applications.

## 4 ibvdev: efficient Java communications over InfiniBand

This section presents the design and implementation of the `ibvdev` communication device, the Java message-passing middleware over InfiniBand developed in this paper. Unlike VIA [14, 15], InfiniBand architecture does not specify an API. Instead, it defines the functionality provided by HCAs to operating systems in terms of Verbs (a “verb” is a semantic description of a function that must be provided). The Verbs interface specifies such functionality as transport resource management, multicast, work request processing, and event handling. The most important implementation used today of Verbs interface is the IBV API provided by the OFED (OpenFabrics Enterprise Distribution) driver distributed by the OpenFabrics Alliance [16]. IBV is also the lowest level InfiniBand networking API for applications, available only in C language. Therefore, any Java communication support on IBV must resort to JNI in order to access IBV API and obtain the best possible performance, the target of the communication middleware developed, `ibvdev`.

### 4.1 Message-passing in Java libraries

There have been several efforts [1] over the last decade to develop a Java message-passing system since its introduction [17]. Most of these projects were prototype implementations, without maintaining. Currently, the most relevant ones in terms of uptake by the HPC community are mpiJava [18], MPJ Express [3], MPJ/Ibis [19] and F-MPJ [20].

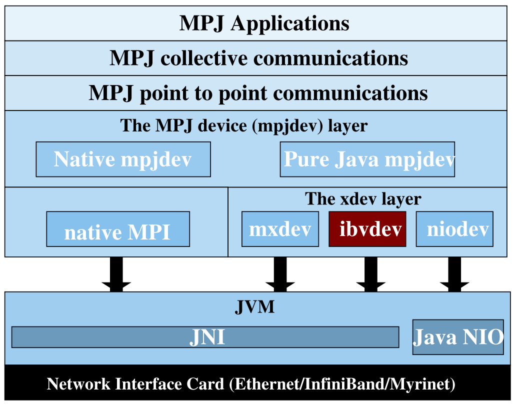
mpiJava [18] is a Java messaging system that uses JNI to interact with the underlying native MPI library. This project has been perhaps the most successful Java HPC messaging system, in terms of uptake by the community. However, although its performance is usually high, mpiJava currently only supports some native MPI implementations, as wrapping a wide number of functions and heterogeneous runtime environments entails an important maintaining effort. Additionally, this implementation presents instability problems, derived from the native code wrapping (all MPJ methods are wrapped), and has thread safety issues in the wrapper layer, being unable to take advantage of multicore systems through multithreading, even if the underlying MPI library is thread safe.

MPJ Express is an MPJ implementation of the mpiJava 1.2 API [17] specification. MPJ Express is thread-safe and presents a modular design which includes a pluggable architecture of communication devices that allows to combine the portability of the "pure" Java New I/O package (Java NIO) communications (`niodev` device) with the high performance Myrinet support (through the native Myrinet eXpress communication library in the `mxdev` device).

MPJ/Ibis [19] is an implementation of the JGF MPJ API [21] specification on top of Ibis [22]. The design philosophy of Ibis is similar to MPJ Express; it is possible to use 100% pure Java communication or use special HPC hardware like Myrinet. There are two pure Java devices in Ibis. The first called `TCPIbis` provides communication using the traditional `java.io` package. The second called `NIOIbis` uses the Java NIO package. Although `TCPIbis` and `NIOIbis` provide blocking and nonblocking communication at the device level, the higher-levels only use blocking versions of these methods. Nevertheless, MPJ/Ibis does not provide a multithreaded communication device, unlike MPJ Express, key to harness the performance of multicore processors.

F-MPJ [20] is our message-passing communication middleware that provides shared memory and high-speed networks (e.g., InfiniBand, Myrinet, and SCI) communication support through the use of JFS. However, the use of Java IO sockets in its communication device `iodev` limits scalability as the progress engine of F-MPJ has to check every connection for incoming messages, unlike Java NIO sockets whose support is already implemented in the `select` method.

**Fig. 3** Overview of the MPJ Express design including `ibvdev`


MPJ Applications
MPJ collective communications
MPJ point to point communications
The MPJ device (mpjdev) layer
Native mpjdev
Pure Java mpjdev
The xdev layer
native MPI
mxdev
ibvdev
niodev
JVM
JNI
Java NIO
Network Interface Card (Ethernet/InfiniBand/Myrinet)

MPJ Express project is currently the most active project in terms of adoption by the HPC community, presence on academia and production environments, and available documentation. This project is also stable and publicly available along with its source code at http://mpj-express.org. Therefore, MPJ Express has been selected for the integration of the `ibvdev` middleware in a production MPJ library.

## 4.2 MPJ Express communication devices design

MPJ Express has a layered design that enables its incremental development and provides the capability to update and swap layers in or out as required. Thus, at runtime end users can opt to use a high performance proprietary network device, or choose a pure Java device, based either on sockets or threads, for portability.

Figure 3 illustrates an overview of the MPJ Express design and the different levels of the software. From top to bottom, it can be seen that a message-passing application in Java (MPJ application) calls MPJ Express point-to-point and collective primitives. These primitives implement the MPJ communications API on top of the `xdev` layer, which has been designed as a pluggable architecture and provides a simple but powerful API. This design facilitates the development of new communication devices in order to provide custom implementations on top of specific native libraries and HPC hardware. Thus, `xdev` is portable as it presents a single API and provides efficient communication on different system configurations.

Figure 3 also shows the three implementations of the `xdev` API for networked communication: `niodev` on Java NIO, and hence TCP/IP, and `mxdev` on Myrinet, as well as the developed `xdev` middleware for direct InfiniBand support, `ibvdev` (depicted in red).

### *4.2.1* `xdev` *API design*

The `xdev` API, presented in Listing 1, has been designed with the goal of being simple and small, providing only basic communication methods, in order to ease the development of `xdev` devices. An `xdev` communication device is similar to the MPI communicator class, but with reduced functionality. The `init` method starts the communication device operation. The id method returns the identification (ProcessID) of the device. The finish method is the last method to be called and completes the device operation.

The xdev communication primitives only include point-to-point communication, both blocking (send and recv, like MPI_Send and MPI_Recv) and nonblocking (isend and irecv, like MPI_Isend and MPI_Irecv). Synchronous communications are also embraced (ssend and issend). These communication methods use PID (ProcessID) objects instead of using ranks as arguments to send and receive primitives. In fact, the xdev layer is focused on providing basic communication methods and it does not deal with high level message-passing abstractions such as groups and communicators. Therefore, a PID object unequivocally identifies a device object.

```
public abstract class Device {
public static Device newInstance(String dev);
ProcessID[] init(String[] args);
ProcessID id();
void finish();

Request isend(Buffer buf, PID dest, int tag, int cntx);
void send(Buffer buf, PID dest, int tag, int cntx);
Request issend(Buffer buf, PID dest, int tag, int cntx);
void ssend(Buffer buf, PID dest, int tag, int cntx);
Status recv(Buffer buf, PID src, int tag, int cntx);
Request irecv(Buffer buf, PID src, int tag, int cntx, Status s);
Status probe(PID src, int tag, int cntx);
Status iprobe(PID src, int tag, int cntx);
Request peek();
}
```

**Listing 1** API of the xdev.Device class

## 4.3 Communication device design

Figure 4 presents the overall design of the communication middleware, which consists of three distinct parts. The first is the definition of a new device, ibvdev, in the xdev layer of MPJ Express (1 in Fig. 4). The analysis of the other high-speed network support in MPJ Express, the implementation of the mxdev device, reveals that it also uses native code via JNI to rely on the MX library, thus posing similar design issues as ibvdev. The MX library [23] provides a set of primitives similar to those needed to implement xdev interface, so there are a number of functions, such as mx_isend, mx_issend, mx_irecv, and mx_wait, that are used in the JNI layer. Therefore, mxdev acts as a Java wrapper layer to MX library, so that the implementation of a method in xdev generally delegates directly in a native method that performs the requested operation in MX library. Nevertheless, the design of mxdev is not directly applicable to ibvdev since InfiniBand lacks an MX-style library that implements the functionality and operations that must be implemented in xdev. The available communication layer for ibvdev is the IBV API, which offers low-level methods for the management of the HCA InfiniBand card.

Therefore, an MX-like library has been defined in order to provide ibvdev with a set of communication primitives with message-passing semantics on InfiniBand,

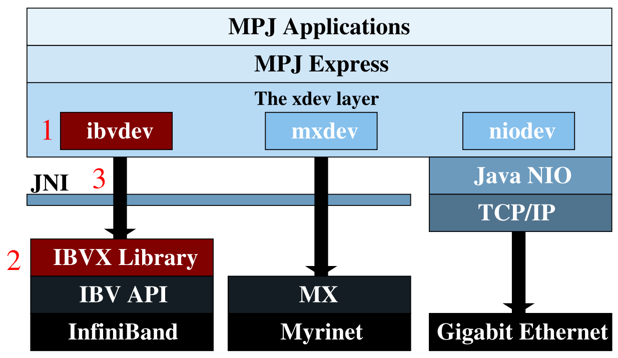


**Fig. 4** Overall design of the communication library

to ease the development of the `xdev` communication device. This library has been denominated IBV eXpress (IBVX) (2 in Fig. 4). With this design, a native communication library has been implemented on top of IBV to provide basic message-passing communication primitives to higher level layers (either Java or non Java). Thus, the new communication device `ibvdev` can rely on IBVX through JNI. The design of this layer allows the access to IBVX from MPJ Express through its `ibvdev` device (3 in Fig. 4).

### *4.3.1 IBV eXpress library design*

The IBVX library is a scalable and high performance low-level C message-passing middleware for communication on InfiniBand systems. It has been designed using the same approach as `xdev` communication devices. In fact, there is a mapping of `xdev` methods to IBVX functions, except for methods `id`, used for process identification, and `getSendOverhead` and `getRecvOverhead`, which are available only at the Java level as give information about the buffer handling. The IBVX API is presented in Listing 2. Like `xdev` API, IBVX includes only point-to-point communication, both blocking and nonblocking, and also synchronous communication support. In order to support nonblocking operations, IBVX implements `IBV_Wait` and `IBV_Test` functions, which handle nonblocking operation requests.

```
IBV_Init(char **pNames, int *pList, int nProcs, int rank, int psl);
IBV_Finalize();
IBV_Isend(void *buf, int size, int dst, int tag, int ctx, Request *r);
IBV_Issend(void *buf, int size, int dst, int tag, int ctx, Request *r);
IBV_Irecv(void *buf, int size, int src, int tag, int ctx, Request *r);
IBV_Send(void *buf, int size, int dst, int tag, int ctx);
IBV_Ssend(void *buf, int size, int dst, int tag, int ctx);
IBV_Recv(void *buf, int size, int src, int tag, int ctx, Status *s);
IBV_Wait(Request *request, Status *status);
IBV_Test(Request *request, Status *status);
IBV_Iprobe(int src, int tag, int context, Status *status);
IBV_Probe(int src, int tag, int context, Status *status);
Request* IBV_Peek();
```

**Listing 2** Public interface of the IBV eXpress library

### *4.3.2* `ibvdev` *JNI layer design*

The design of the JNI layer of `ibvdev` is quite straightforward as it acts as a thin wrapper over IBVX. Thus, each native method of `ibvdev` delegates on a native IBVX function through JNI, implementing a series of three steps: (1) get Java objects associated parameters required for calling the corresponding library function in IBVX; (2) call IBVX function; and (3) save the results in the appropriate attributes of the Java objects involved in the communication. As general rules in the implementation of the JNI layer, it has been extensively used the caching of object references, thus minimizing the overhead associated with the JNI calls.

## 4.4 IBV eXpress library implementation

IBVX library implements nonblocking low-level communication primitives (see Listing 2) on top of IBV API. The first decision is the transport service used to create the queue pairs. Not all transports services support RDMA operations (see Table 1), whose support is desirable, so these transport services (UC and UD) are discarded.

Moreover, for RD and XRC transport services is not applicable the InfiniBand end-to-end flow control and this requires the development of a specific flow control software layer, which can add significant overhead if the implementation is not efficient. Therefore, the RC transport service has been selected as it provides reliability, delivery order, data loss detection, and error detection.

IBVX implements all communication operations as nonblocking communication primitives. Then blocking communication support is implemented as a nonblocking primitive followed by an `IBV_Wait` call. Therefore, the basic set of functions implemented consists of `IBV_Init`, `IBV_Finalize,` and nonblocking communication functions (`IBV_Isend`, `IBV_Issend`, `IBV_Irecv`), and the function that checks the completion of a nonblocking operation (`IBV_Test`). Thus, the operation that waits for the completion of a nonblocking operation (`IBV_Wait`) has been implemented following a strategy of polling (busy loop) as a continuous loop calling `IBV_Test` until the test is positive (thus minimizing latency). Blocking communication functions (`IBV_Send`, `IBV_Ssend`, and `IBV_Recv`) have been implemented by a call to its corresponding nonblocking function followed by an `IBV_Wait` call. Moreover, the probe operation, which checks for incoming messages without actual receipt of any of them, has been also implemented in the nonblocking version `IBV_Iprobe`, whereas the blocking version (`IBV_Probe`) relies on the nonblocking operation completion.

### *4.4.1 IBV eXpress communication protocols*

Message-passing libraries usually implement two different communication protocols:

1. *Eager protocol*: the sender process eagerly sends the entire message to the receiver. In order to achieve this, the receiver needs to provide a sufficient number of buffers to handle incoming messages. This protocol has minimal startup over-

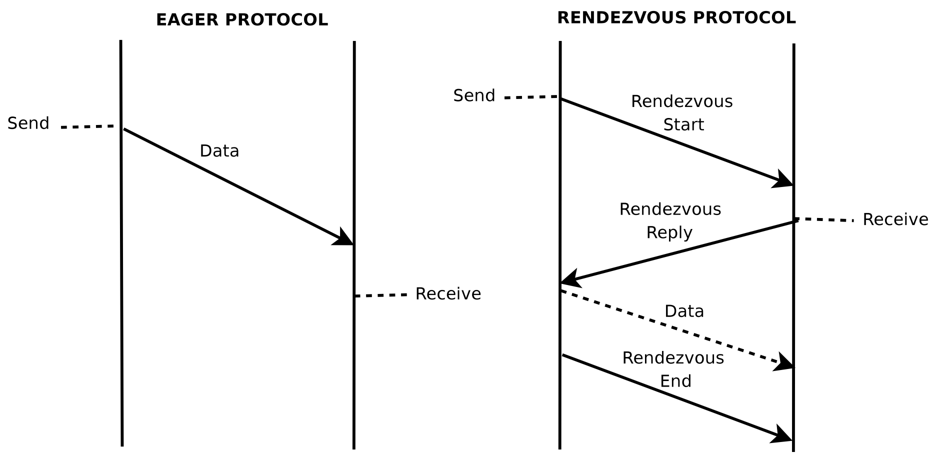
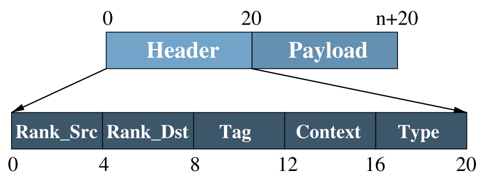


**Fig. 5** MPI eager and rendezvous protocols

**Fig. 6** Message format in IBVX library



heads and is used to implement low latency message-passing communication for smaller messages (typically $<$ 128 KB, configurable threshold).

2. *Rendezvous protocol*: this protocol negotiates (via control messages) the buffer availability at the receiver side before the message is actually transferred. This protocol is used for transferring large messages (typically $>$ 128 KB), whenever the sender is not sure whether the receiver actually has enough buffer space to hold the entire message.

Figure 5 presents graphically the operation of eager and rendezvous protocols.

### *4.4.2 Message format*

The presence of control messages in the operation of the rendezvous protocol and the need for a receiving process to unequivocally distinguish a message, has forced the introduction of a message header before the actual data payload. Thus, a message is defined as the union of a header of 20 bytes (starting from the beginning), which is followed by the data payload, as shown in Fig. 6.

The header consists of 5 fields of 4 bytes each representing in this order: the process rank that sends the message, the destination process rank, the tag or label of the message, the context to which it belongs, and the type of message. All header fields are 4-byte integers, for all types of messages.

### *4.4.3 Eager protocol*

The overhead of data copies is small for short messages, such as eager protocol transfers and control messages, which are eagerly push through the network to achieve the lowest latency. This operation matches with the semantic of InfiniBand send/receive communication.

In `IBV_Init` a reliable connection is set up between every two processes. For a single process, the send and receive queues of all connections are associated with a single CQ (Completion Queue). Through this CQ, the completion of all send and RDMA operations can be detected at the sender side. The completion of receive operations (or arrival of incoming messages) can also be detected through the CQ (see Fig. 7).

The InfiniBand Architecture requires the pinning of buffers previous to the communication, thus they must be registered with the hardware. In the eager protocol implementation (shown in Fig. 7), the buffer pinning and unpinning overhead is avoided by using a pool of prepinned, fixed size buffers for communication. For sending an eager data message, the data is copied to one of the buffers first and sent out from this buffer to the send queue (1 in Fig. 7). At the receiver side, a number of buffers from the pool are preposted (2 in Fig. 7). After the message is received, the payload is copied to the destination buffer (3 in Fig. 7). The communication of control messages also uses this buffer pool as they are actually sent using the eager protocol.

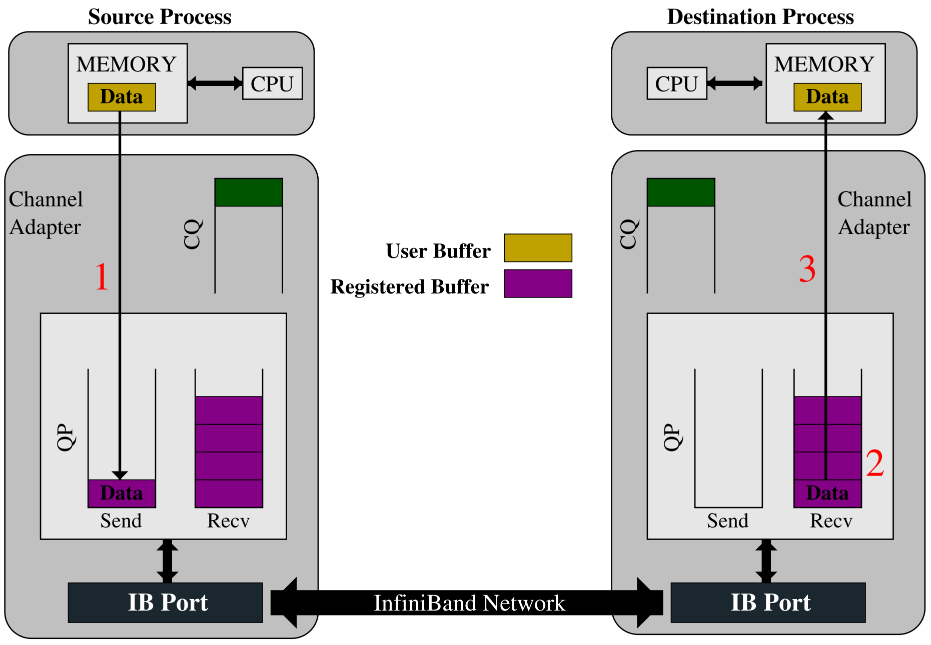


**Fig. 7** Eager protocol implementation in IBVX

### *4.4.4 Rendezvous protocol*

When transferring large messages it is extremely beneficial to avoid extra data copies. A zero-copy rendezvous protocol implementation can be achieved by using RDMA operations. The rendezvous protocol negotiates the buffer availability at the receiver side. However, the actual data can be transferred either by using RDMA Write or RDMA Read. RDMA Write-based approaches can totally eliminate intermediate copies and efficiently transfer large messages. RDMA Read-based approaches can enable both zero copy and computation and communication overlap. Similar approaches have been widely used for implementing MPI communications over different interconnects [24, 25].

The RDMA Write-based protocol is illustrated in Fig. 8 (right). In this implementation, the buffers are pinned down in memory and the buffer addresses are exchanged via control messages. The sending process first sends a control message to the receiver (RNDZ_START). The receiver replies to the sender using another control message (RNDZ_REPLY). This reply message contains the receiving application's buffer information along with the remote key to access that memory region. The sending process then sends the large message directly to the receiver's application buffer by using RDMA Write (DATA). Finally, the sending process issues another control message (RNDZ_END) which indicates to the receiver that the message has been placed in the application buffer.

IBVX uses a *progress engine* to discover incoming messages and to make progress on outstanding sends. As can be seen in Fig. 8, the RDMA Write-based rendezvous protocol generates multiple control messages which have to be discovered by the *progress engine*. Since the *progress engine* operation is based on polling, it requires a call to the IBVX library.

RDMA Read operation presents a small number of control messages and thus a reduced set of I/O bus transactions. In addition, since the receiver can progress

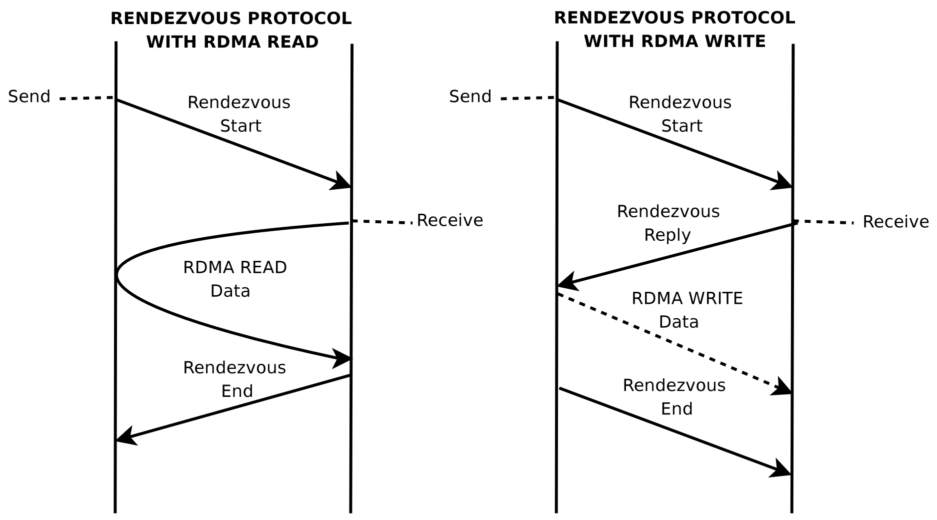


**Fig. 8** Rendezvous protocol alternatives

independently of the sender (once the RNDZ_START message is sent), the sender does not need to call any IBVX progress, the data transfer proceeds with RDMA Read without direct control of the sender.

The rendezvous protocol over RDMA Read is also illustrated in Fig. 8 (left). Here, the sending process begins with the RNDZ_START message, which has embedded the virtual address and memory handle information of the message buffer to be sent. Thus, upon the receipt of this RNDZ_START message all the information about the application buffer is available to the receiving process, and no RNDZ_REPLY message needs to be sent any more. Upon its discovery, the receiving process issues the DATA message over RDMA Read. When the operation has been completed, it informs the sending process by a RNDZ_END message. This approach, although simple, poses several design challenges that have to be addressed before directly utilize RDMA Read:

– Limited Outstanding RDMA Reads: The number of outstanding RDMA Reads on any QP is a fixed number (typically 8 or 16), decided during the QP creation.
– Issuing RNDZ_END Message: According to InfiniBand specification [4], Send or RDMA Write transactions are not guaranteed to finish in order with outstanding RDMA Reads.

For these reasons, the rendezvous protocol has been implemented with RDMA Write operation, in order to benefit from a more productive development.

### *4.4.5 Cache of registered buffers*

In rendezvous protocol, data buffers are pinned on-the-fly. However, the buffer pinning and unpinning overhead can be reduced by using the pin-down cache technique [26]. The idea is to maintain a cache of registered buffers. When a buffer is first registered, it is put into the cache. When the buffer is unregistered, the actual unregister operation is not carried out and the buffer stays in the cache. Thus, the next time when the buffer needs to be registered, we do not need to do anything because it is already in the cache. The effectiveness of pin-down cache depends on how often the application reuses its buffers. If the reuse rate is high, most of the buffer registration and deregistration operations can be avoided.

## 4.5 JNI layer implementation details

The JNI layer is a wrapper for IBVX library, in order to make it accessible from Java. Therefore, it implements the functions that the `javah` utility generated in terms of native operations contained in communication device Java classes. The development of this layer must take into account the design of the MPJ Express buffering layer [27]. The use of this buffering layer incurs a copying overhead that can be significant for large messages, and is considered a performance bottleneck for MPJ Express [28], so the handling of this layer has to be implemented efficiently.

The core class of the buffering layer used for packing and unpacking data is `mpjbuf.Buffer`. This class provides two storage options: static and dynamic. Implementations of static storage use the interface `mpjbuf.RawBuffer`. It is possible to have alternative implementations of the static section depending on the actual

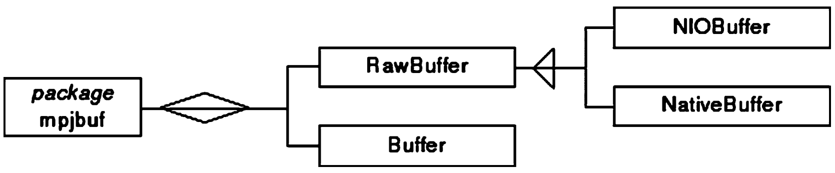


**Fig. 9** Primary buffering classes in mpjbuf

raw storage medium. In addition, it also contains an attribute of type `byte[]` that represents the dynamic section of the message. Figure 9 shows two implementations of the `mpjbuf.RawBuffer` interface. The first, `mpjbuf.NIOBuffer` is an implementation based on `ByteBuffers`. The second, `mpjbuf.NativeBuffer` is an implementation for the native MPI device, which allocates memory in the native C code. Figure 9 shows the primary buffering classes in the `mpjbuf` API.

Regarding `mpjbuf.Buffer` class design, it is necessary to handle at the JNI layer a second call to the IBVX library when communicating a buffer with data in the two sections (static and dynamic). To support this operation efficiently, the first 4 bytes of the static buffer indicate the size of the dynamic part of the buffer. Thus, the overhead of this protocol in terms of buffering space, returned by `getSendOverhead` and `getRecvOverhead` methods, is 4 bytes. These methods, implemented for every MPJ Express communication device, are used to express the extra space needed in the static buffer to implement the buffering layer support, and they are profusely used when handling the buffer contents.

## 5 Performance evaluation

This section presents a performance evaluation of the developed communication device `ibvdev`, compared to native MPI libraries (MVAPICH and OpenMPI) and the MPJ Express communications devices `niodev` over InfiniBand (using IPoIB) and `smpdev` for shared memory communication. This evaluation consists of a microbenchmarking of point-to-point data transfers (Sect. 5.2) and collective communications (Sect. 5.3), as well as an analysis of the impact on the overall performance of the use of the developed library on several representative MPJ codes (Sect. 5.4).

### 5.1 Experimental configuration

The evaluation of `ibvdev` has been carried out in a cluster which consists of 8 nodes, each of them with 8 GB of RAM and 2 Intel Xeon E5520 quad-core Nehalem processors. Although each node has 8 cores, the HyperThreading (HT) is enabled so it is possible to run 16 processes per node concurrently. The interconnection networks are InfiniBand (16 Gbps of maximum theoretical bandwidth), with OFED driver 1.5, and Gigabit Ethernet (1 Gbps). The OS is Linux CentOS 5.3 with kernel 2.6.18 and the JVM is Sun JDK 1.6.0_13. The evaluated MPJ implementation is MPJ Express [29] version 0.36 (labeled MPJE in graphs) and the evaluated MPI implementations are MVAPICH [25] v1.2.0 and OpenMPI [24] v1.3.3. The PSL (Protocol Switch Limit)

MPJ Express attribute, the threshold between eager and rendezvous send protocols, has been set to 128 KB message size for all the benchmarks. F-MPJ and MPJ/Ibis results are not shown for clarity purposes, apart from the fact that `ibvdev` is only integrated in MPJ Express. However, as they are sockets-based implementations, their performance is similar to `niodev` results.

## 5.2 Point-to-point micro-benchmarking

In order to micro-benchmark MPJ point-to-point and collectives primitives performance our own micro-benchmark suite [30], similar to Intel MPI Benchmarks used for MPI libraries, has been used due to the lack of suitable micro-benchmarks for MPJ evaluation. Here, the results shown are the half of the round-trip time of a pingpong test or its corresponding bandwidth. The transferred data are byte arrays, avoiding the serialization overhead that would distort the analysis of the results.

Figures 10 and 11 show point-to-point latencies (for short messages) and bandwidths (for long messages) on InfiniBand and shared memory, respectively. The `ibvdev` middleware obtains significant point-to-point performance benefits, thus obtaining 11 μs start-up latency and up to 7.2 Gbps bandwidth. The threshold between eager and rendezvous send protocols can be observed in the bandwidth graph at 128 KB, which confirms the efficiency of the implementation of the zero-copy rendezvous protocol with RDMA Write for `ibvdev`. These results outperform significantly `niodev` over InfiniBand, limited to 65 μs start-up latency and below 3 Gbps bandwidth.

Compared to native MPI libraries, `ibvdev` obtains a similar bandwidth than MVAPICH (7 Gbps) in this testbed, surpassing it even at several points (e.g., 32 KB, 256 KB, and 512 KB message sizes). Nevertheless, OpenMPI shows the best performance from 32 KB message size, obtaining up to 9.2 Gbps bandwidth. As for latency, `ibvdev` obtains better results than MVAPICH (13 μs) and only slightly worse than OpenMPI (10 μs), again the best performer.

Regarding shared memory communication performance, `ibvdev` obtains much better start-up latency, 6 μs, than the multithreading `smpdev` middleware, which achieves 17 μs, which means that `ibvdev` has implemented a highly efficient communication protocol and that `smpdev` presents poor start-up latency, caused by an excess of synchronizations. The native MPI libraries are again the best performers obtaining 0.5 μs an 1 μs for MVAPICH and OpenMPI, respectively, due to their efficient communications support on shared memory. Regarding bandwidth, MPJ devices are far from native MPI libraries, obtaining worse performance (15.3 Gbps and 22 Gbps for `ibvdev` and `smpdev`, respectively).

## 5.3 Collective primitives micro-benchmarking

Figure 12 presents the aggregated bandwidth for representative MPJ data movement operations (broadcast and allgather), and computational operations (reduce and allreduce double precision sum operations) with 128 processes. The aggregated bandwidth metric has been selected as it takes into account the global amount of data transferred. The `niodev` allgather results could not be taken due to flaws in the implementation that hanged its operation. In addition to `ibvdev`, `niodev`, and MPI

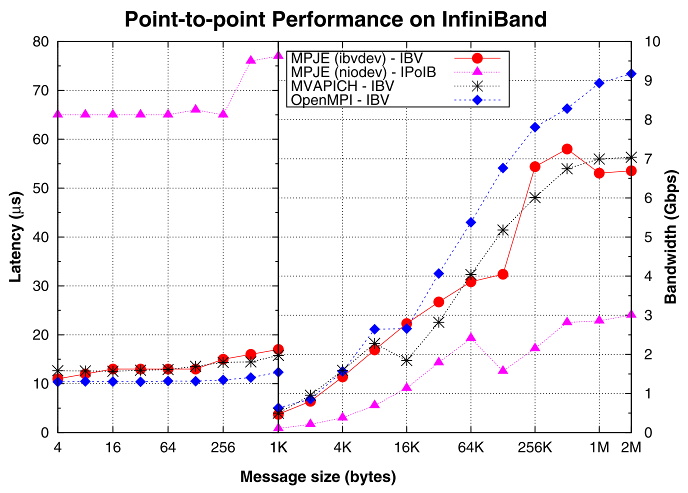


**Fig. 10** Message-passing point-to-point performance on InfiniBand

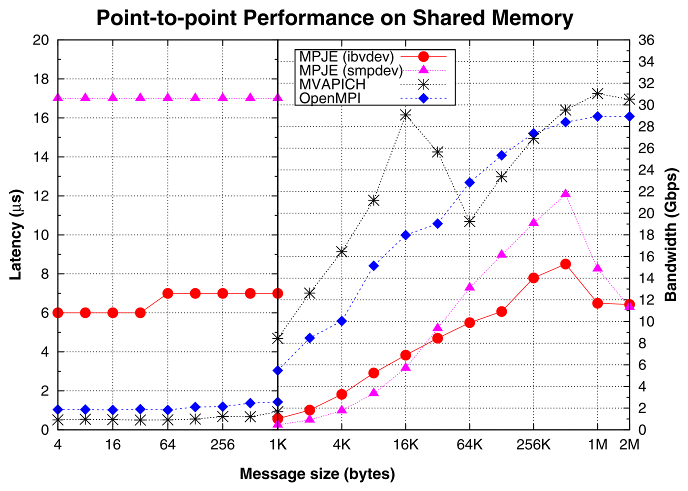


**Fig. 11** Message-passing point-to-point performance on shared memory

communications it has been evaluated the performance of multithreaded versions of the MPJ collective operations, running only one process per node, and 16 threads within each process. Thus, instead of running 128 processes on the cluster, only 8 processes are being used, taking advantage of intranode communications through multithreading. This hybrid support of network and multithreading communications is one of the main advantages of Java middleware for scalable and efficient communication on clusters of multicore processors.

The results confirm that `ibvdev` outperforms significantly `niodev`, achieving up to one order of magnitude higher performance, although generally the performance benefit is 2 or 3 times better. Moreover, both `ibvdev` and `niodev` take advantage of the multithreaded collectives. With respect to the MPI libraries, `ibvdev`

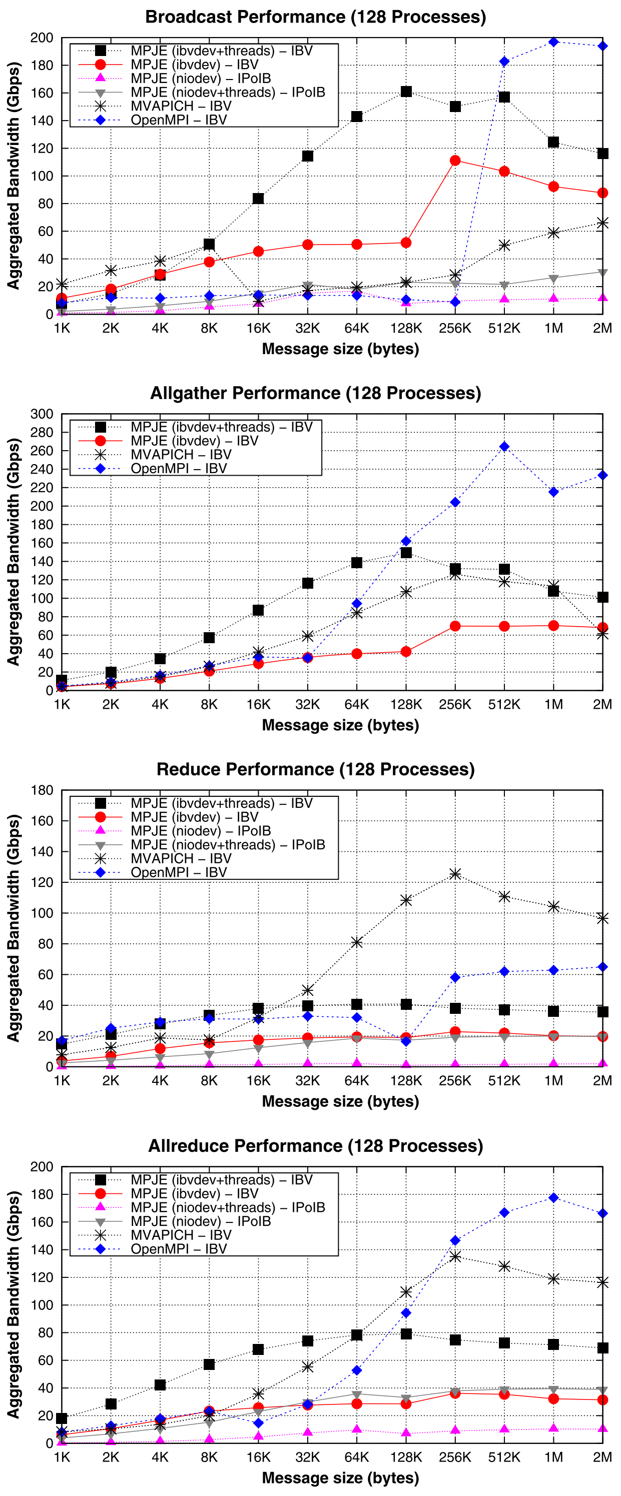


**Fig. 12** Message-passing collective primitives performance

achieves better performance than MPI collectives for short messages, up to 16–256 KB, thanks to the exploitation of multithreading in collectives implementation and the use of a high PSL (128 KB), whereas MPI libraries use smaller PSL (8 KB). However, for longer messages the MPI collectives achieve much better performance due to the use of better collective algorithms, and the use of pipelined transfers.

## 5.4 Kernel/application performance analysis

The impact of `ibvdev` on the scalability of Java parallel codes has been analyzed using the NAS Parallel Benchmarks (NPB) implementation for MPJ (NPB-MPJ) [31], selected as the NPB are probably the benchmarks most commonly used in the evaluation of languages, libraries, and middleware for HPC. In fact, there are implementations of the NPB for MPI, OpenMP, and hybrid MPI/OpenMP.

Four representative NPB codes have been evaluated: CG (Conjugate Gradient), FT (Fourier Transform), IS (Integer Sort), and MG (Multi-Grid). Moreover, the jGadget [32] cosmology simulation application has also been analyzed. These MPJ codes have been selected as they show very poor scalability with MPJ Express over InfiniBand. Hence, these are target codes for the evaluation of the impact on performance of the use of `ibvdev` in MPJ Express. The results have been obtained using up to 64 processes instead of 128, due to memory constraints on the cluster.

Figure 13 shows the NPB-MPJ CG, IS, FT, and MG results, respectively, for the Class C workload in terms of MOPS (Millions of Operations Per Second) (left) and its corresponding scalability, in terms of speedup (right). For CG kernel, `ibvdev` doubles the performance of the `niodev` device over InfiniBand, with almost 9000 MOPS compared to less than 4000 MOPS on 64 processes. With respect to IS kernel, the results for `niodev` over InfiniBand show a significant slowdown with 64 processes, not taking advantage of the use of 64 processes, while `ibvdev` keeps on scaling and gets up to 650 MOPS, significantly outperforming the `niodev` results. Regarding FT, `ibvdev` also doubles the performance of the `niodev` device over InfiniBand, with around 17000 MOPS compared to less than 8000 MOPS. Finally, the impact of `ibvdev` on MG is smaller than for the remaining codes as this NPB is less communication intensive, as obtains relatively good speedups, even with `niodev` (speedup of 30 with 64 processes).

The performance comparison of `ibvdev` against MPI libraries has two different analyses, depending on the metric used. If we take into account the MOPS achieved, MPI benchmarks obtain always the best performance, around a 50% higher than `ibvdev` results. The poorer performance of NPB-MPJ can be attributed to the lower performance of the JVM compared to native compilers. However, if we have a look at the speedups, `ibvdev` outperforms MPI for FT and MG, while obtains slightly lower scalability for CG and IS, which suggests that `ibvdev` implements a highly efficient communication support, even comparable to MPI libraries, and that the use of efficient communication libraries can bridge the gap between Java and natively compiled languages provided that an efficient communication support is made available.

The jGadget application is the MPJ implementation of Gadget [33], a popular cosmology simulation code initially implemented in C and parallelized using MPI that is used to study a large variety of problems like colliding and merging galaxies or the

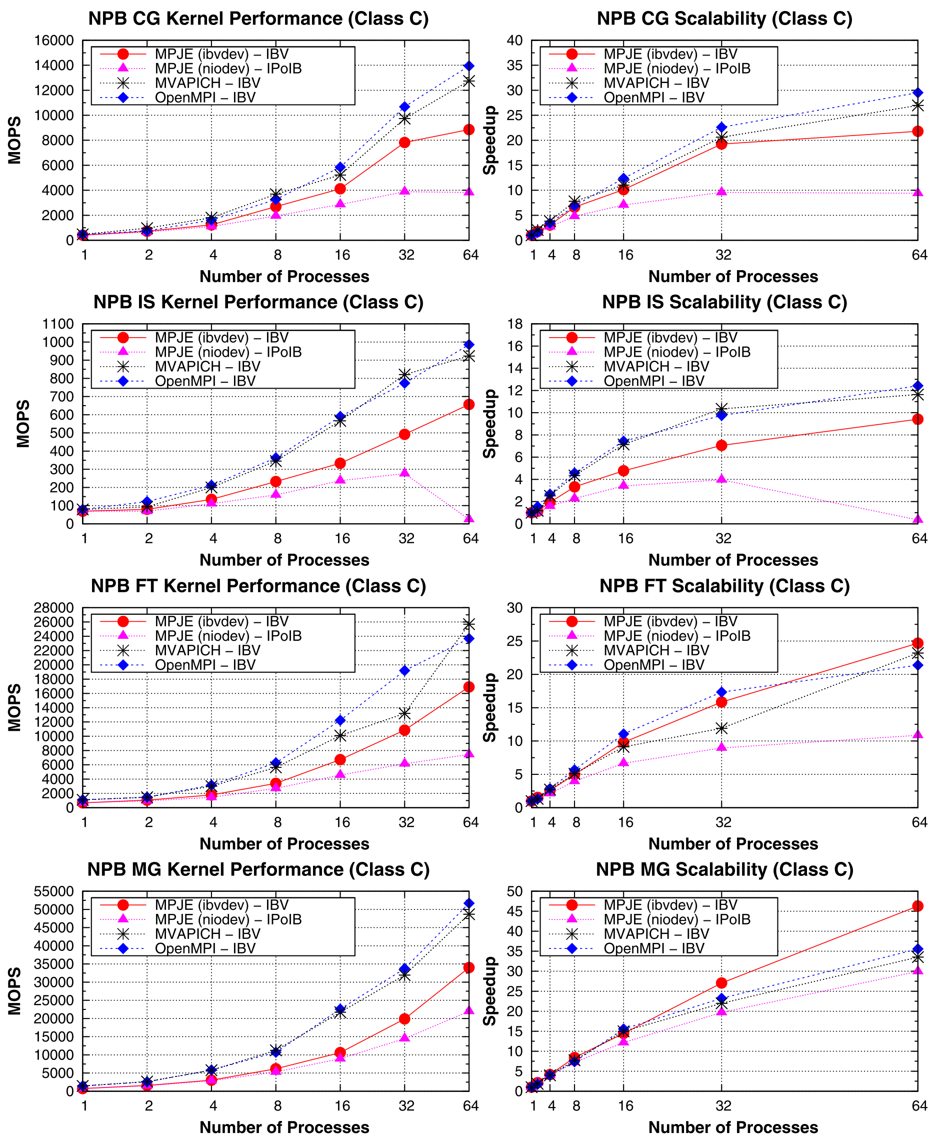
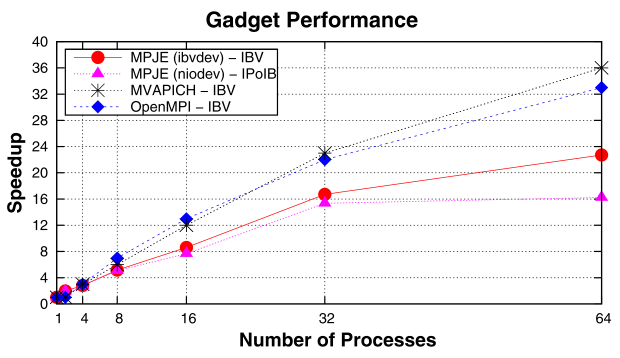


**Fig. 13** Performance and scalability of NPB-MPI/MPJ codes

formation of large-scale structures. This application has been selected for the performance evaluation of `ibvdev`, measuring its performance using up to 64 processes instead of 128, due to memory constraints on the cluster (each Java process is using its own JVM).

Figure 14 presents the performance results of jGadget running a two million particles cluster formation simulation. As jGadget is a communication-intensive application, with important collective operations overhead, only modest speedups are obtained. Here, `ibvdev` can take advantage of the use of 64 processes (speedup above 22), whereas `niodev` over IPoIB remains with a speedup of 16. Regarding

**Fig. 14** Scalability of MPI/MPJ Gadget



MPI results, OpenMPI and MVAPICH achieve around 45% higher speedup than `ibvdev` on 64 processes, which suggests that this middleware is bridging the gap between Java and natively compiled applications in HPC.

## 6 Conclusions

This paper has presented `ibvdev`, a scalable and efficient low-level Java message-passing device for communication on InfiniBand systems. The increase in the number of cores per system demands languages with built-in multithreading and networking support, such as Java, as well as scalable and efficient communication middleware that can take advantage of multicore systems. The developed device transparently provides Java message-passing applications with efficient performance on InfiniBand thanks to its direct support on IBV and the efficient and scalable implementation of a lightweight communication protocol which is able to take advantage of RDMA over InfiniBand.

The performance evaluation of `ibvdev` on an InfiniBand multicore cluster has shown that this middleware obtains start-up latencies and bandwidths similar to MPI performance results, obtaining in fact up to 85% start-up latency reduction and twice the bandwidth compared to previous Java middleware on InfiniBand. Additionally, the impact of `ibvdev` on message-passing collective operations is significant, achieving up to one order of magnitude performance increases compared to previous Java solutions, especially when taking advantage of shared memory intra-process (multithreading) communication. The analysis of the impact of the use of `ibvdev` on MPJ applications shows a significant performance increase compared to sockets-based middleware (`niodev`), which helps to bridge the gap between Java and natively compiled codes in HPC. To sum up, the efficiency of this middleware, which is even competitive with MPI point-to-point transfers, increments the scalability of communications intensive Java applications, especially in combination with the native multithreading support of Java.

**Acknowledgements** This work was funded by the Ministry of Science and Innovation of Spain under Project TIN2010-16735, and by the Xunta de Galicia under the Consolidation Program of Competitive Research Groups and Galician Network of High Performance Computing.

## References

1. Taboada GL, Ramos S, Expósito RR, Touriño J, Doallo R (2011, in press) Java in the high performance computing arena: research, practice and experience. Sci Comput Program. doi:10.1016/j.scico.2011.06.002
2. Blount B, Chatterjee S (1999) An evaluation of Java for numerical computing. Sci Program 7(2):97–110
3. Shafi A, Carpenter B, Baker M (2009) Nested parallelism for multi-core HPC systems using Java. J Parallel Distrib Comput 69(6):532–545
4. InfiniBand Trade Association (2004) Infiniband architecture specification volume 1, release 1.2.1. http://www.infinibandta.org/ [Last visited: April 2011]
5. Baker M, Carpenter B, Shafi A (2005) A pluggable architecture for high-performance Java messaging. IEEE Distrib Syst Online 6(10):1–4
6. IETF Draft. IP over IB. http://www.ietf.org/old/2009/ids.by.wg/ipoib.html [Last visited: April 2011]
7. Hongwei Z, Wan H, Jizhong H, Jin H, Lisheng Z (2007) A performance study of Java communication stacks over InfiniBand and Gigabit Ethernet. In: Proc 4th IFIP intl conf network and parallel computing (NPC'07), Dalian, China, pp 602–607
8. Veldema R, Hofman RFH, Bhoedjang R, Bal HE (2003) Run-time optimizations for a Java DSM implementation. Concurr Comput 15(3–5):299–316
9. Righi RDR, Navaux POA, Cera MC, Pasin M (2005) Asynchronous communication in Java over InfiniBand and DECK. In: Proc 17th intl symp on computer architecture and high performance computing (SBAC-PAD'05), Rio de Janeiro, Brazil, pp 176–183
10. Taboada GL, Touriño J, Doallo R (2008) Java Fast Sockets: enabling high-speed Java communications on high performance clusters. Comput Commun 31(17):4049–4059
11. Wan H, Hongwei Z, Jin H, Jizhong H, Lisheng Z (2007) Jdib: Java applications interface to unshackle the communication capabilities of InfiniBand networks. In: Proc 4th IFIP intl conf network and parallel computing (NPC'07), Dalian, China, pp 596–601
12. Wan H, Jizhong H, Jin H, Lisheng Z, Yao L (2008) Enabling RDMA capability of InfiniBand network for Java applications. In: Proc 4th IFIP intl conf on networking, architecture, and storage (NAS'08), Chongqing, China, pp 187–188
13. Yao L, Jizhong H, Jinjun G, Xubin H (2009) uStream: a user-level stream protocol over InfiniBand. In: Proc 15th intl conf on parallel and distributed systems (ICPADS'09), Shenzhen, China, pp 65–71
14. Dunning D, Regnier G, McAlpine G, Cameron D, Shubert B, Berry F, Merritt AM, Gronke E, Dodd C (1998) The Virtual Interface Architecture. IEEE MICRO 18(2):66–76
15. Compaq, Intel and Microsoft Corporations (1997) The Virtual Interface Architecture specification, version 1.0
16. OpenFabrics Alliance Website. http://www.openfabrics.org/ [Last visited: April 2011]
17. Carpenter B, Fox G, Ko S-H, Lim S mpiJava 1.2: API specification. http://www.hpjava.org/reports/mpiJava-spec/mpiJava-spec/mpiJava-spec.html [Last visited: April 2011]
18. The mpiJava project. http://www.hpjava.org/mpiJava.html [Last visited: April 2011]
19. Bornemann M, van Nieuwpoort RV, Kielmann T (2005) MPJ/Ibis: a flexible and efficient message passing platform for Java. In: Proc. 12th European PVM/MPI users' group meeting (EuroPVM/MPI'05), Sorrento, Italy, pp 217–224
20. Taboada GL, Touriño J, Doallo R (2011) F-MPJ: scalable Java message-passing communications on parallel systems. J Supercomput. doi:10.1007/s11227-009-0270-0
21. Java Grande Forum. http://www.javagrande.org [Last visited: April 2011]
22. van Nieuwpoort RV, Maassen J, Wrzesinska G, Hofman R, Jacobs C, Kielmann T, Bal HE (2005) Ibis: a flexible and efficient Java-based grid programming environment. Concurr Comput 17(7–8):1079–1107
23. MX User's Guide. http://www.myri.com/scs/MX/doc/mx.pdf [Last visited: April 2011]
24. Open Source High Performance MPI Library. http://www.open-mpi.org/ [Last visited: April 2011]
25. MPI over InfiniBand, 10GigE/iWARP and RDMAoE. http://mvapich.cse.ohio-state.edu/ [Last visited: April 2011]
26. Tezuka H, O'Carroll F, Hori A, Ishikaw Y (1998) Pin-down cache: a virtual memory management technique for zero-copy communication. In: Proc 12th intl parallel processing symposium/9th symposium on parallel and distributed processing (IPPS/SPDP'98), Orlando, FL, USA, pp 308–314
27. Baker M, Carpenter B, Shafi A (2007) A buffering layer to support derived types and proprietary networks for Java HPC. Scalable Comput Pract Experience 8(4):343–358

28. Taboada GL, Touriño J, Doallo R, Shafi A, Baker M, Carpenter B (2011, in press) Device level communication libraries for high-performance computing in Java. Concurr Comput. doi:110.1002/cpe.1777
29. MPJ Express Website. http://mpj-express.org/ [Last visited: April 2011]
30. Taboada GL, Touriño J, Doallo R (2003) Performance analysis of Java message-passing libraries on Fast Ethernet, Myrinet and SCI clusters. In: Proc 5th IEEE intl conf on cluster computing (CLUSTER'03), Hong Kong, China, pp 118–126
31. Mallón DA, Taboada GL, Touriño J, Doallo R (2009) NPB-MPJ: NAS parallel benchmarks implementation for message-passing in Java. In: Proc 17th Euromicro intl conf on parallel, distributed, and network-based processing (PDP'09), Weimar, Germany, pp 181–190
32. Baker M, Carpenter B, Shafi A (2006) MPJ Express meets Gadget: towards a Java code for cosmological simulations. In: 13th European PVM/MPI users' group meeting (EuroPVM/MPI'06), Bonn, Germany, pp 358–365
33. Springel V (2005) The cosmological simulation code GADGET-2. Mon Not R Astron Soc 364(4):1105–1134

# Chapter 3

# FastMPJ: a Scalable Java Message-Passing Library

The content of this chapter corresponds to the following journal paper:

**- Title:** FastMPJ: a scalable and efficient Java message-passing library

**- Authors:** Roberto R. Expósito, Sabela Ramos, Guillermo L. Taboada, Juan Touriño, Ramón Doallo

**- Journal:** Cluster Computing

**- Editorial:** Springer Science+Business Media

**- ISSN:** 1386-7857

**- Year:** 2014

**- Volume(Number):Pages:** In press

**- DOI:** 10.1007/s10586-014-0345-4

The final publication is available at http://link.springer.com/article/10.1007%2Fs10586-014-0345-4. A copy of the accepted paper has been included next.

## FastMPJ: a scalable and efficient Java message-passing library

**Roberto R. Expósito · Sabela Ramos · Guillermo L. Taboada · Juan Touriño · Ramón Doallo**

**Abstract** The performance and scalability of communications are key for high performance computing (HPC) applications in the current multi-core era. Despite the significant benefits (e.g., productivity, portability, multithreading) of Java for parallel programming, its poor communications support has hindered its adoption in the HPC community. This paper presents FastMPJ, an efficient message-passing in Java (MPJ) library, boosting Java for HPC by: (1) providing high-performance shared memory communications using Java threads; (2) taking full advantage of high-speed cluster networks (e.g., InfiniBand) to provide low-latency and high bandwidth communications; (3) including a scalable collective library with topology aware primitives, automatically selected at runtime; (4) avoiding Java data buffering overheads through zero-copy protocols; and (5) implementing the most widely extended MPI-like Java bindings for a highly productive development. The comprehensive performance evaluation on representative testbeds (InfiniBand, 10 Gigabit Ethernet, Myrinet, and shared memory systems) has shown that FastMPJ communication primitives rival native MPI implementations, significantly improving the efficiency and scalability of Java HPC parallel applications.

R. R. Expósito (✉) · S. Ramos · G. L. Taboada · J. Touriño · R. Doallo
Computer Architecture Group, Department of Electronics and Systems, University of A Coruña, La Coruña, Spain
e-mail: rreye@udc.es

S. Ramos
e-mail: sramos@udc.es

G. L. Taboada
e-mail: taboada@udc.es

J. Touriño
e-mail: juan@udc.es

R. Doallo
e-mail: doallo@udc.es

**Keywords** High performance computing (HPC) · Parallel computing · Message-Passing in Java (MPJ) · Communication middleware · High-speed networks · Performance evaluation

### 1 Introduction

Java is currently among the preferred programming languages in web-based and distributed computing environments, and is an attractive option for high performance computing (HPC) [1]. Java provides some interesting characteristics of special benefit for parallel programming: built-in multithreading and networking support in the core of the language, in addition to its other traditional advantages for general programming such as object orientation, automatic memory management, portability, easy-to-learn properties, an extensive API and a wide community of developers.

Although Java was severely criticized for its poor computational performance in its beginnings [2], the performance gap between Java and natively compiled languages (e.g., C/C++, Fortran) has been narrowing for the last years [3,1]. The Java Virtual Machine (JVM), which executes Java applications, is now equipped with just-in-time (JIT) compilers that can obtain native performance from Java bytecode [4]. Nevertheless, the significant improvement in its computational performance is not enough to be a successful language in the area of parallel computing, as the performance of the communications is also essential to achieve high scalability in Java for HPC, especially in the current multi-core era.

Message-passing interface (MPI) [5] is the most widely used parallel programming paradigm and it is highly portable, scalable and provides good performance. It is the preferred choice for writing parallel applications on distributed memory systems such as multi-core clusters, currently the most popular system deployments thanks to their interesting cost/performance ratio. Here, Java represents an attractive alternative to natively compiled languages traditionally used in HPC, for the development of applications for these systems as it provides built-in networking and multithreading support, key features for taking full advantage of hybrid shared/distributed memory architectures. Thus, Java can resort to threads in shared memory (intra-node) and to its networking support for distributed memory (inter-node) communications.

The increasing number of cores per system demands efficient and scalable message-passing communication middleware in order to meet the ever growing computational power needs. Moreover, current system deployments are aggregating a significant number of cores through advanced high-speed cluster networks such as InfiniBand (IB) [6], which usually provide interesting features such as remote direct memory access (RDMA) support, increasing the complexity of communication protocols. However, up to now message-passing in Java (MPJ) [7] implementations have been focused on providing new functionalities, rather than concentrate on developing efficient communications on high-speed networks and shared memory systems. This lack of efficient communication support in Java, especially in the presence of high-speed cluster networks, results in lower performance than native MPI implementations. Thus, the adoption of Java as a mainstream language on these systems heavily depends on the availability of efficient communication middleware in order to benefit from its appealing features at a reasonable overhead.

This paper presents FastMPJ, our efficient and scalable MPJ implementation for parallel computing, which addresses all these issues. Thus, FastMPJ provides high-performance shared memory communications, efficient support of high-speed networks, as well as a scalable collective library which includes topology aware primitives. The comprehensive performance evaluation has shown that FastMPJ is competitive with native MPI libraries, which increases the scalability of communication-intensive Java HPC parallel applications.

The structure of this paper is as follows: Sect. 2 presents background information about MPJ. Section 3 introduces the related work. Section 4 describes the overall design of FastMPJ. Section 5 details some aspects of the FastMPJ implementation, including point-to-point and collective communications support. Comprehensive benchmarking results from FastMPJ evaluation are shown in Sect. 6. Finally, Sect. 7 summarizes our concluding remarks.

## 2 Message-passing in Java

Soon after the introduction of Java, there have been several implementations of MPJ libraries. However, the MPI standard [5] defines bindings for C, C++ and Fortran programming languages. Therefore, as there are no bindings for the Java language in the standard, most of the initial MPJ projects have developed their own MPI-like bindings. In contrast, most recent projects generally adhere to one of the two major MPI-like Java bindings which have been proposed by the community: (1) the mpiJava 1.2 API [8], the most widely extended, which supports an MPI C++-like interface for the MPI 1.1 subset, and (2) the JGF MPJ API [7], which is the proposal of the Java Grande Forum (JGF) [9].

MPJ libraries are usually implemented in three ways: (1) using some high-level Java messaging API like Remote Method Invocation (RMI) to implement a "pure" Java message-passing system (i.e., 100 % Java code); (2) wrapping an underlying native MPI library through the Java Native Interface (JNI); or (3) following a hybrid layered design, which includes a pluggable architecture based on an idea of low-level communication devices. Thus, hybrid libraries provide Java-based implementations of the high-level features of MPI at the top levels of the software. Hence, they can offer a "pure" Java approach through the use of Java-based communication devices (e.g., via Java sockets), and additionally a higher performance approach through low-level native communication devices that use JNI to take advantage of specialized HPC hardware. Although most of the Java communication middleware is based on RMI, MPJ libraries looking for efficient communication have followed the latter two approaches.

Generally, applications implemented on top of Java messaging systems can have different requirements. Thus, for some applications the main concern could be portability, while for others could be high-performance communications. Each of the above solutions fit with specific situations, but can present associated trade-offs. On the one hand, the use of RMI ensures portability, but it may not provide an efficient solution, especially in the presence of high-speed HPC hardware. On the other hand, the wrapper-based approach presents some inherent portability and instability issues derived from the native code, as these implementations have to wrap all the methods of the MPJ API. Moreover, the support of multiple heterogeneous runtime platforms, MPI libraries and JVMs entails a significant maintenance effort, although usually in exchange for higher performance than RMI. However, the hybrid approach minimizes the JNI code to the bare minimum using low-level pluggable communication devices, being the only solution that can ensure both requirements. Nevertheless, most of the MPJ projects that conform with this hybrid design rely on Java sockets and inefficient TCP/IP emulations to support current HPC communication hardware (e.g., InfiniBand). Although the use of Java sockets usually outperforms RMI-based middleware, it requires an important programming effort. Furthermore, the use of the sockets API in a communication device still represents an important source of overhead and lack of scalability in Java communications, especially in the presence of high-speed networks [10].

## 3 Related work

Multiple MPI native implementations have been developed, improved and maintained over the last 15 years intended for cluster, grid and emerging cloud computing environments. Regarding MPJ libraries, there have been several efforts to develop a Java message-passing system for HPC since its introduction [1,11]. However, most of the developed projects over the last decade were prototype implementations, without maintenance. Currently, the most relevant ones in terms of uptake by the HPC community are mpiJava, MPJ Express, MPJ/Ibis, and FastMPJ, next presented.

mpiJava [12] is a Java message-passing system that consists of a collection of wrapper classes that use JNI to interact with an underlying native MPI library. This project implements the mpiJava 1.2 API and has been perhaps the most successful Java HPC messaging system, in terms of uptake by the community. However, mpiJava can incur a noticeable overhead, especially for large messages, and also presents some portability and instability issues. Thus, it only supports some native MPI implementations, as wrapping a wide number of methods and heterogeneous runtime platforms entails a significant maintenance effort, as mentioned before.

MPJ Express [13] is one of the projects that conforms with the aforementioned hybrid approach. This library implements the mpiJava 1.2 API and presents a modular design which includes a pluggable architecture of communication devices that allows to combine the portability of the "pure" Java New I/O (NIO) communications package together with the native Myrinet support through JNI. Additionally, it provides shared memory support using Java threads [14]. However, this project poses several important issues: (1) its overall design relies on a buffering layer [15] that significantly limits performance and scalability of communications; (2) it lacks efficient support for InfiniBand (IB), the most widely adopted networking technology in current HPC clusters; (3) it includes poorly scalable collective algorithms; and (4) its bootstrapping mechanism typically exhibits some issues in specific environments.

MPJ/Ibis [16] is another hybrid project that, in this case, conforms with the JGF MPJ API. Actually, this library is implemented on top of Ibis [17], a parallel and distributed Java computing framework. Thus, it can use either "pure" Java communications, based on Java sockets, or native communications on Myrinet. However, the Myrinet support is based on the GM library, an out-of-date low-level API which has been superseded by the Myrinet Express (MX) library [18]. Moreover, MPJ/Ibis also lacks efficient IB support, and additionally, does not provide efficient shared memory and collective communications. Furthermore, MPJ/Ibis does not fully implement some high-level features of MPI (e.g., inter-communicators and virtual topologies).

FastMPJ is our Java message-passing implementation of the mpiJava 1.2 API, which also presents a hybrid design approach. The initial prototype implementation was presented as a proof of concept in [19]. This prototype only implemented a small subset of the communications-related API. Furthermore, it only included one communication device implemented on top of Java IO sockets, which severely limited its overall scalability and performance. Although the use of high-performance socket implementations, such as the Java Fast Sockets (JFS) project [20], can improve performance on shared memory and high-speed networks, the use of sockets in a communication device can not provide an efficient and scalable solution, as mentioned in the previous section.

Currently, FastMPJ has overcome these limitations by: (1) implementing the remaining of the mpiJava 1.2 API (e.g., virtual topologies, inter-communicators and groups operations are currently available), except part of the derived data types (e.g., Vector, Struct) since Java can provide any user-defined structure natively, by using objects, which fits more straightforwardly into an object-oriented programming model; (2) providing high-performance shared memory support; (3) efficiently supporting high-speed cluster networks, especially IB; and (4) implementing a user friendly and scalable bootstrapping mechanism to start the Java parallel processes. The overcoming of the previous limitations of FastMPJ, together with the implementation of an efficient communications support which provides similar performance to native MPI libraries, are the main contributions of this paper.

Additionally, some previous works have already evaluated the aforementioned MPJ libraries [19,21]. As main conclusions, these studies have assessed that FastMPJ is the best performer among them, overcoming some of the previous performance limitations such as the high buffering penalty and the JNI overhead. Moreover, most of the MPJ projects, especially mpiJava and MPJ/Ibis, are currently outdated and without active development. Due to these drawbacks, mainly low performance and lack of up-to-date development, the performance evaluation carried out in Sect. 6 only considers the comparison of FastMPJ against native MPI libraries, for clarity purposes.

Finally, there have also been some additional works that focused on other important aspects of Java to be a successful option in HPC, such as providing high-performance file I/O [22,23].

## 4 FastMPJ design

Figure 1 presents an overview of the FastMPJ layered design and the different levels of the software. The MPJ commu-

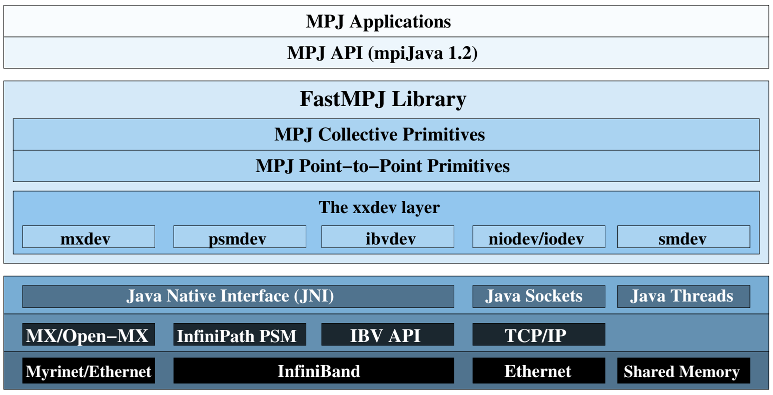


**Fig. 1** Overview of the FastMPJ layered design

nications API, which includes both collective and point-to-point primitives, is implemented on top of the `xxdev` device layer. The device layer has been designed as a simple and pluggable architecture of low-level communication devices. Moreover, this layer supports the direct communication of any serializable Java object without data buffering, whereas `xdev` [24], the API that `xxdev` is extending, does not support this direct communication. Thus, the `xdev` API, which is used internally by the MPJ Express library, relies on a buffering layer [15] which is only able to transfer the custom `xdev` buffer objects. This fact adds a noticeable copying overhead [1], especially for large messages, which prevents MPJ Express to implement zero-copy protocols. The avoidance of this intermediate data buffering overhead on the critical path of communications is the main benefit of the `xxdev` device layer with respect to its predecessor. Thus, this fact allows `xxdev` communication devices to implement zero-copy protocols when communicating primitive data types using, for instance, RDMA-capable high-speed cluster networks. Additional benefits of this API are its flexibility, portability and modularity thanks to its encapsulated design.

In more detail, the `xxdev` layer provides a Java low-level message-passing API (see Listing 1) with basic operations such as point-to-point blocking (`send` and `recv`) and non-blocking (`isend` and `irecv`) communication methods. Moreover, it also includes synchronous communications (`ssend` and `issend`) and functions to check incoming messages without actually receiving them (`probe` and `iprobe`). Thus, an `xxdev` device is similar to an MPI communicator, but with reduced functionality. This simple design eases significantly the development of `xxdev` communications devices in order to provide custom support of high-speed cluster networks (e.g., High-speed Ethernet and IB) and shared memory systems, while leveraging other infrastructure provided by the upper levels of FastMPJ, such as the runtime system and the layer that provides the full MPJ semantics (e.g., virtual topologies, intercommunicators). With this modular design FastMPJ enables its incremental development and provides the capability to update and swap layers in or out as required. Thus, end users can opt at runtime to use a high-performance native network device, or choose a "pure" Java device, based either on sockets or threads, for portability.

## 5 FastMPJ implementation

FastMPJ communication support relies on the efficient implementation of low-level `xxdev` devices on top of specific native libraries and HPC communication hardware. Currently, FastMPJ includes three communication devices that support high-speed cluster networks: (1) `mxdev`, for Myrinet and High-speed Ethernet; (2) `psmdev`, for Intel/Q-Logic InfiniBand adapters; and (3) `ibvdev`, for InfiniBand adapters in general terms. These devices are implemented on top of MX/Open-MX, InfiniPath PSM and IB Verbs (IBV) native libraries, respectively (see Fig. 1). Although these underlying native libraries have been initially designed for inter-node network-based communication, in the particular case of MX/Open-MX and PSM also provide efficient intra-node shared memory communication, usually implemented through some Inter-Process Communication (IPC) mechanism. Thus, this fact allows FastMPJ to take full advantage of hybrid shared/distributed memory architectures, such as clusters of multi-core nodes, except for the `ibvdev` device, as IBV does not support shared memory. Additionally, the TCP/IP stack (`niodev` and `iodev`) and high-performance shared memory systems (`smdev`) are also supported through "pure" Java communication devices, which ensures portability.

The user-level methods of the MPJ API related to the collective and point-to-point communication layers are implemented on top of these `xxdev` communication devices. This may involve some native code depending on the underlying device being used (e.g., `ibvdev` and `psmdev` for IB

```
public abstract class Device
{
  public static Device newInstance(String device);
  abstract ProcessID[] init(String[] args);
  abstract ProcessID id();
  abstract void finish();

  abstract Request isend(Object msg,PID dst,int tag,int context);
  abstract Request irecv(Object msg,PID src,int tag,int context,Status status);
  abstract void send(Object msg,PID dst,int tag,int context);
  abstract Status recv(Objecct msg,PID src,int tag,int context);
  abstract Request issend(Object msg,PID dst,int tag,int context);
  abstract void ssend(Object msg,PID src,int tag,int context);

  abstract Status iprobe(PID src,int tag,int context);
  abstract Status probe(PID src,int tag,int context);
}
```

**Listing 1** xxdev API

support). The rest of the high-level abstractions of the MPJ API (e.g., virtual topologies, intra- and inter-communicators, groups operations) is implemented in "pure" Java code (i.e., 100 % Java). Hence, this implementation can ensure both portability and/or high-performance requirements of Java message-passing applications, while avoiding some of the associated problems of the wrapper-based approach through JNI, as mentioned in Sect. 2 (e.g., instability and portability issues, high maintenance effort). These issues are derived from the amount of native code that is involved using a wrapper-based implementation (note that all the methods of the MPJ API have to be wrapped). However, FastMPJ can minimize to the bare minimum the amount of JNI code needed to support a specific network device, as the `xxdev` devices only have to implement a very small number of methods (see Listing 1). In the next sections, the implementation of the various MPI features in FastMPJ will be discussed.

## 5.1 High-speed networks support

FastMPJ provides efficient support for high-speed cluster networks through `mxdev`, `ibvdev` and `psmdev` communications devices, next presented.

### *5.1.1 Myrinet/high-speed Ethernet*

The `mxdev` device implements the `xxdev` API on top of the Myrinet Express (MX) library [18], which runs natively on Myrinet networks. More recently, the MX API has also been supported in high-speed Ethernet networks (10/40 Gigabit Ethernet), both on Myricom specialized NICs and on any generic Ethernet hardware through the Open-MX [25] open-source project. Thus, the TCP/IP stack can be replaced by `mxdev` transfers over Ethernet networks providing higher performance than using standard Java sockets. Moreover, the `mxdev` device can also take advantage of the efficient intra-node shared memory communication protocol implemented by MX/Open-MX [26] to improve the performance of networked applications in multi-core systems.

In MX messages are exchanged among endpoints, which are software representations of Myrinet/Ethernet NICs. Every message operation, either sending or receiving, starts with a non-blocking communication request (e.g., mx_isend), which is queued by MX, returning the control to `mxdev`. Then, the `mxdev` device is responsible for checking the successful completion of the communication operation. The message matching mechanism at the receiver side is based on a 64-bit matching field, specified by both communication peers, in order to deliver incoming messages to the right receive requests.

The MX API is only available in C, thus the `mxdev` device implements `xxdev` methods calling MX functions through JNI. Moreover, as MX already provides a low-level messaging API which closely matches the `xxdev` layer, `mxdev` deals with the Java objects marshaling and communication, the JNI transfers and the MX parameters handling. Therefore, FastMPJ with `mxdev` provides the user with a higher level messaging API than MX, also freeing Java developers from the implementation of JNI calls, which benefits programmability without trading off much performance.

### *5.1.2 InfiniBand*

The native and efficient InfiniBand (IB) support is also included in FastMPJ with `ibvdev` and `psmdev` devices. On the one hand, the `ibvdev` device directly implements its own communication protocols through JNI on top of the IBV API, which is part of the OpenFabrics Enterprise Distribution (OFED [27]), an open-source software for RDMA and kernel bypass applications. The native support of the IBV API in

Java is somewhat restricted so far to native MPI libraries, as previous MPJ libraries relied on the TCP/IP emulation over IB protocol (IPoIB) [28], which provides significantly poorer performance, especially for short messages [10].

A previous implementation of the `ibvdev` device was firstly integrated into the MPJ Express library [29] as a proof of concept, but only for internal testing purposes as it was never part of the official release. Although it was able to provide higher performance than using the IPoIB protocol, the buffering layer in MPJ Express significantly limited its performance and scalability. Therefore, the `ibvdev` device had to be reimplemented to conform with the `xxdev` API and then adapted for its integration into the FastMPJ library in order to improve its performance. Thus, FastMPJ achieves start-up latencies and bandwidths similar to native MPI performance results on IB networks thanks to the efficient, lightweight and scalable communication protocol implemented in `ibvdev`, which includes a zero-copy mechanism for large messages using the RDMA-write operation.

On the other hand, another original contribution of this paper is the introduction of the `psmdev` device, which provides for the first time in Java native support for the InfiniPath family of Intel/QLogic IB adapters over the Performance Scaled Messaging (PSM) interface. PSM is a low-level user-space messaging library which implements an intra-node shared memory and inter-node communication protocol, which are completely transparent to the application.

In order to establish the initial connections between endpoints, the `psmdev` device has to rely on an out-of-band mechanism, which has been implemented with TCP sockets, to distribute the endpoint identifiers. After initializing endpoints, a Matched Queue (MQ) interface is created and can be used to send and receive messages. The MQ interface semantics are consistent with those defined by the MPI 1.2 standard for message-passing between two processes. Thus, incoming messages are stored according to their tags to preposted receive buffers. The PSM API is only available in C; thus, following a similar approach to `mxdev`, the `psmdev` device also implements `xxdev` methods calling PSM functions through JNI dealing with the Java objects marshaling and communication, the JNI transfers and the PSM parameters handling. Although the Intel/QLogic adapters are also supported by the `ibvdev` device through the IBV API, `psmdev` usually achieves significantly higher performance than using `ibvdev`, as PSM is specifically designed and highly tuned by Intel/QLogic for its own IB adapters.

## 5.2 Socket-based communications devices

Initially, FastMPJ included only one communication device implemented on top of Java IO sockets (`iodev`), which turned out to be the limiting factor in performance and scalability, especially for non-blocking communication. This fact has motivated the implementation of a new communication device based on Java NIO sockets (`niodev`), which include more scalable non-blocking communication support by providing `select()` like functionality. Additionally, a new socket-based device (`sctpdev`) implemented on top of Stream Control Transmission Protocol (SCTP) sockets is currently work in progress.

Nevertheless, these “pure” Java communication devices are only provided for portability reasons, as they rely on the ubiquitous TCP/IP stack, which introduces high communication overhead and limited scalability for communication-intensive HPC applications.

## 5.3 Shared memory communications

FastMPJ includes a “pure” Java thread-based communication device (`smdev`) that efficiently supports shared memory intra-node communication [30], thus being able to exploit the underlying multi-core architecture replacing inter-process and network-based communications by Java threads and shared memory intra-process transfers.

In this thread-based device, there is a single JVM instance and each MPJ rank in the parallel application (i.e., each Java process in the case of using a network-based communication device) is represented by a Java thread. Consequently, message-passing communication between these threads is achieved using shared data structures. Therefore, the FastMPJ runtime must create a single JVM with as many Java threads as the number of ranks exist in the global communicator (i.e., MPI.COMM_WORLD), which depends on an input parameter that is specified by the user when starting the MPJ application.

An obvious advantage of this approach, especially in the context of Java, is that an application does not compromise portability. Moreover, the use of a single JVM can take advantage of lower memory consumption and garbage collection overhead. Furthermore, while multithreading programming allows to exploit shared memory intra-process transfers, it usually increases the development complexity due to the need for thread control and management, task scheduling, synchronization, and maintenance of shared data structures. Thus, using the `smdev` device, the developer does not have to deal with the issues of the multithreading programming, as `smdev` offers a high level of abstraction that supports handling threads as message-passing processes, providing similar or even higher performance than native MPI implementations.

### *5.3.1 Class loading*

The use of threads in the `smdev` device requires the isolation of the namespace for each thread, configuring a distributed memory space in which they can exchange messages through shared memory references. While processes from different JVMs are completely independent entities, threads within a JVM are instances of the same application class, sharing all static variables. Thus, this device creates each running thread with its custom class loader. Therefore, all the non-shared classes within a thread can be directly isolated in its own namespace in order to behave like independent processes. Nevertheless, communication through shared memory transfers requires the access to several shared classes within the device. When the system loader does not find a class, the custom class loader is used, following the JVM class loader hierarchy. This mechanism implies that the system class loader loads every reachable class that, in consequence, is shared by all threads. Thus, its classpath has to be bounded in such a way that it only has access to shared packages that contain the implementation of shared memory transfers among threads. Consequently, communications are delegated to a shared class which allocates and manages shared message queues (a pair of queues per thread) in order to implement the data transfers as regular data copies between threads, thus providing a highly efficient zero-copy protocol.

Finally, the use of a pair of queues per thread enables `smdev` to include fine-grained synchronizations, combining busy waits and locks, thus reducing contention in the access to the shared structures. As an example, MPJ Express shared memory support [14] uses a global pair of queues with class lock-based synchronization, which can result in a very inefficient approach in applications that involve a high number of threads.

## 5.4 Scalable collective communications

The MPI specification defines collective communication operations as a convenience to application developers, which can save significant time in the development of parallel applications. FastMPJ provides a scalable and efficient collective communication library for parallel computing on multi-core architectures. This library includes topology aware primitives which are implemented on top of point-to-point communications, taking advantage of communications overlapping and obtaining significant performance benefits in collective-based communication-intensive MPJ applications. The library implements up to six algorithms per collective primitive, whereas previous MPJ libraries are usually restricted to one algorithm. Furthermore, the algorithms can be selected automatically at runtime, depending on the number of cores and the message length involved in the collective operation.

The collective algorithms present in the FastMPJ collective library can be classified in six types, namely Flat Tree (FT) or linear, Minimum-Spanning Tree (MST), Binomial Tree (BT), Four-ary Tree (FaT), Bucket (BKT) or cyclic, and BiDirectional Exchange (BDE) or recursive doubling, which have been extensively described in the literature [31].

## 5.5 Runtime system

Although the runtime system is not part of the MPI specification, it is an essential element which allows to execute processes across various platforms. Thus, the FastMPJ runtime system is in charge of starting the parallel Java processes across multiple machines, supporting several OSs either UNIX-based (e.g., GNU/Linux, MAC OS X) or Microsoft Windows-based (XP/Vista/7/8). In addition, the runtime does not assume a shared file system and it allows to run MPJ applications using both class and JAR file formats.

This fully portable runtime system mainly consists of two modules: (1) an fmpjd module (Java daemon listening on a configurable TCP port) which executes on compute nodes and listens for requests to start Java processes in a new JVM; and (2) an fmpjrun module, client of the Java daemons. In UNIX-based OSs, a set of Java daemons can be started/stopped over the network using SSH within the fmpjrun application, as the OS is automatically detected by FastMPJ. Moreover, the runtime system is also compatible with traditional job schedulers such as SGE/OGE, SLURM, LSF and PBS. Additionally, other modules are provided to start, stop and trace the status (running/not running) of the daemons. However, on Windows platforms, the daemons either need to be: (1) manually started, (2) configured to start automatically on OS startup, or (3) installed as a native service, as SSH utilities are not usually available in these platforms.

The FastMPJ runtime efficiently supports the handling of a high number of machines and processes. For instance, a 1024-core "Hello World" MPJ program can be executed in less than 35 s, including the time needed for starting the Java daemons and the initialization of the parallel environment. Figure 2 compares FastMPJ against MPICH-MX, which was configured with the SLURM PMI process launcher, running a "Hello World" example application on the MareNostrum

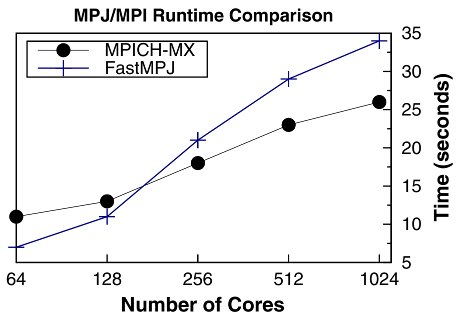


**Fig. 2** MPJ/MPI deployment and initialization

testbed (see Table 1 in Sect. 6.1 for more details on this testbed).

## 6 Performance evaluation

This section presents a comprehensive performance evaluation of the FastMPJ library compared to representative native MPI libraries: Open MPI [32], MVAPICH2 [33] and MPICH-MX [34], from point-to-point and collective message-passing primitives to the assessment of their impact on the scalability of representative parallel codes, using the NASA Advanced Supercomputing (NAS) Parallel Benchmarks suite (NPB) [35,36]. The NPB parallel codes have been selected as it is the benchmarking suite most commonly used in the evaluation of languages, libraries and middleware for HPC.

As mentioned in Sect. 3, previous works [19,21] have already characterized the performance of the other popular MPJ implementations (mpiJava, MPJ/Ibis and MPJ Express) against native MPI libraries, so for clarity purposes these MPJ implementations have not been re-evaluated. In fact, these libraries obtained poor performance, as shown in the references, and they have not been updated since their last evaluations.

### 6.1 Experimental configuration

Table 1 shows the main characteristics of the five representative systems used in the performance evaluation. Both FastMPJ and native MPI libraries have been configured with the most efficient settings and communication device for each testbed (e.g., using only the shared memory device in shared memory systems).

Regarding distributed memory systems, the first testbed (from now on IB-QDR) is a multi-core cluster [37] that consists of 64 nodes, each of them with 24 GB of memory and 2 Intel Xeon quad-core Westmere-EP processors (hence 8 cores per node) interconnected via IB QDR (Mellanox-based NICs). The performance results for the collective primitives micro-benchmarking and the NPB kernels evaluation on this system have been obtained using 8 processes per node (hence 512 cores in total). The second system (from now on IB-DDR) is a multi-core cluster that consists of 16 nodes, each of them with 16 GB of memory and 2 Intel Xeon quad-core Nehalem-EP processors (hence 8 cores per node) interconnected via IB DDR (QLogic-based NICs). Additionally, two nodes have also one 10 Gigabit Ethernet (GbE) Intel NIC. Performance results on this testbed have also been obtained using 8 processes per node (hence 128 cores in total). The third system is the MareNostrum supercomputer [38] (from now on MN), which was ranked #465 in the TOP500 [39] list (June 2012). This supercomputer consists of 2,560 nodes, each of them with 8 GB of memory and 2 PowerPC dual-

**Table 1** Description of the systems used in the performance evaluation

| | #nodes | CPU | Memory | #cores | NIC (Driver) | Network | OS (Kernel) | MPI libraries | JVM |
|---|---|---|---|---|---|---|---|---|---|
| **IB-QDR** | 64 | 2 x 4-core Intel Xeon E5620 | 24 GB | 512 | Mellanox MT26428 (OFED 1.3) | IB QDR (32 Gbps) | CentOS (2.6.32) | Open MPI 1.4.4 MVAPICH2 1.6 | OracleJDK 1.6.0_27 |
| **IB-DDR** | 16 | 2 x 4-core Intel Xeon E5520 | 16 GB | 128 | QLogic QLE7240 (OFED 1.5) Intel 82598EB (Open-MX 1.5.1) | IB DDR (16 Gbps) 10 GbE (10 Gbps) | CentOS (2.6.18) | Open MPI 1.4.5 MVAPICH2 1.7 | OracleJDK 1.6.0_23 |
| **MN** | 2560 | 2 x 2-core IBM PowerPC 970MP | 8 GB | 10240 | Myrinet 2000 (MX 1.2.7) | Myrinet (2 Gbps) | Suse (2.6.16) | MPICH-MX 1.2.7 | IBM 1.7.0 |
| **Intel-SHM** | 1 | 4 x 10-core Intel Xeon E7 4850 | 512 GB | 40 | – | – | Ubuntu (3.2.0) | Open MPI 1.4.5 MVAPICH2 1.7 | OpenJDK 1.6.0_23 |
| **AMD-SHM** | 1 | 4 x 12-core AMD Opteron 6172 | 128 GB | 48 | – | – | CentOS (2.6.32) | Open MPI 1.4.4 MVAPICH2 1.6 | OracleJDK 1.6.0_23 |

core processors (hence 4 cores per node) interconnected via Myrinet 2000. General user accounts on this supercomputer are limited to use up to 1,024 cores. Thus, performance results on this system have been obtained using 256 nodes and 4 processes per node (hence 1,024 cores in total).

Regarding shared memory systems, the Intel-SHM testbed has 4 Intel Xeon ten-core Westmere-EX processors (hence 40 cores) and 512 GB of memory, whereas the AMD-SHM testbed provides with 4 AMD Opteron twelve-core Magny-Cours processors (hence 48 cores) and 128 GB of memory. The NPB performance results on these systems have been executed using 1, 2, 4, 8, 16 and 32 cores. Thus, the maximum number of available cores in each shared memory system could not be used, as the selected NBP kernels only work for a number of cores which is a power of two.

The evaluation of message-passing communication primitives (Sects. 6.2 and 6.3) has been carried out using a representative micro-benchmarking suite, the Intel MPI Benchmarks (IMB) [40], and our own MPJ counterpart, which adheres to the IMB measurement methodology. The transferred data are byte arrays, avoiding the Java serialization overhead that would distort the analysis of the results, in order to present a fair comparison with MPI. In addition, these benchmark suites have been used without cache invalidaton, as it is more representative of a real scenario, where data to be transmitted is generally in cache.

Finally, the evaluation of representative message-passing parallel codes (Sect. 6.4) has used the MPI and OpenMP implementations of the NPB suite (NPB-MPI/NPB-OMP version 3.3) together with its MPJ counterpart (NPB-MPJ) [41]. Four representative NPB kernels have been evaluated: Conjugate Gradient (CG), Fourier Transform (FT), Integer Sort (IS) and Multi-Grid (MG), selected as they present medium to high communication intensiveness. The performance of two different common scaling metrics has been analyzed: (1) strong scaling (i.e., fix the problem size and vary the number of cores); and (2) weak scaling (i.e., vary the problem size linearly with the number of cores).

## 6.2 Point-to-point micro-benchmarking

Figure 3 presents point-to-point performance results obtained on IB (top graphs), 10 GbE and Myrinet (middle graphs), and on shared memory systems (bottom graphs). The metric shown is the half of the round-trip time of a pingpong test for messages up to 1 KB (left part of the graphs), and the bandwidth for messages larger than 1 KB (right part).

On the IB-QDR testbed (top left graph), FastMPJ `ibvdev` device obtains 2.2 µs start-up latency, quite close to MPI results (around 1.9 µs). Regarding bandwidth results, `ibvdev` bandwidth is slightly lower than the MPI performance up to 64-KB messages. From this point, `ibvdev` changes to an RDMA Write-based zero-copy protocol which is able to obtain similar bandwidths (up to 22.5 Gbps) to MPI libraries for large messages. On the IB-DDR testbed (top right graph), the `psmdev` device and Open MPI obtain the lowest start-up latency, around 1.9 µs, slightly outperforming MVAPICH2 (2 µs). The observed bandwidths are identical up to 128 KB, when MVAPICH2 gets slightly better results than Open MPI and FastMPJ in the message range (256 KB–2 MB). For messages $\geq$2 MB, `psmdev` obtains up to 11.5 Gbps whereas MPI libraries only achieve a 6 % more bandwidth, around 12.2 Gbps. These results confirm that `ibvdev` and `psmdev` devices implement highly efficient and lightweight communication protocols, which allows Java applications to take full advantage of the low-latency and high throughput provided by IB.

Regarding the 10 GbE testbed (middle left graph), `mxdev` gets start-up latencies as low as 15.6 µs, quite competitive compared to MPI libraries which obtain 11.2 and 11.5 µs for MVAPICH2 and Open MPI, respectively. Fortunately, this small gap disappears from 1 KB, when `mxdev` and MVAPICH2 achieve identical bandwidths, whereas Open MPI results are the worst up to 2 MB. From this point, the network turns out to be the main performance bottleneck, as the maximum bandwidth achieved is around 9.4 Gbps for all evaluated libraries, quite close to the 10 Gbps limit for this networking technology. Here, the avoidance of the TCP/IP protocol is key for FastMPJ to obtain competitive results compared to MPI, especially for short messages, as the use of a socket-based device (`iodev` or `niodev`) would incur a significant overhead due to the poor performance of Java sockets. The results on the MN supercomputer over a Myrinet network (middle right graph) show that `mxdev` start-up latency gets even closer to MPI results, obtaining 5.2 and 4.1 µs, respectively. Their observed bandwidths are quite similar from 1 KB, suffering the 2 Gbps limit for this networking technology.

Regarding shared memory systems, the performance results of the `smdev` device on the Intel-SHM testbed (bottom left graph) show even below 1 µs start-up latencies, but approximately twice the latency obtained by MPI libraries (around 0.42–0.48 µs). However, for message sizes >2 KB, the zero-copy thread-based intra-process protocol implemented by `smdev`, which allows direct data transfers between Java threads, clearly outperforms MPI libraries. Here, MPI libraries usually implement one-copy protocols since data transfers are IPCs through an intermediate shared memory structure, using IPC resources, which requires at least two data transfers. However, the direct communication in `smdev` does not show significant benefits in the latency of very short messages, as MPI libraries achieve lower start-up latencies for message sizes <2 KB. Thus, the thread synchronization overhead for `smdev`, which combines busy waits and locks, seems to be higher than the process synchronization overhead for MPI libraries, which usually use only lock-

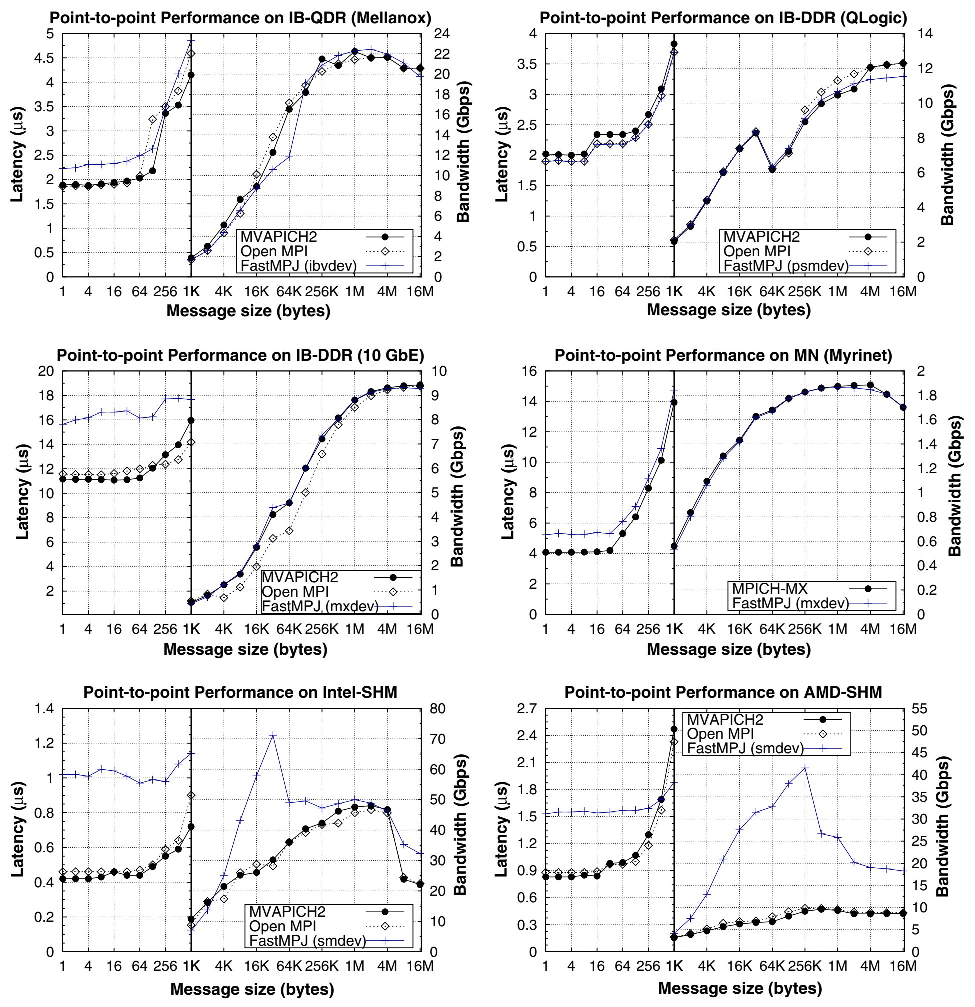


**Fig. 3** Point-to-point performance on InfiniBand QDR and DDR, 10 Gigabit Ethernet, Myrinet and shared memory

free algorithms. In addition, the high start-up latency overhead imposed by the JVM in the initialization of the copy is higher than the cost of the IPC extra copy performed by MPI when transferring short messages. As the overhead per byte transferred in MPI, which uses two data transfers, is higher than the combined overhead for `smdev` (thread synchronization plus JVM start-up latency), the consequence is that up to a certain threshold point (message size <2 KB), short messages have less overhead for MPI, whereas FastMPJ is the best performer for medium and large messages due to the avoidance of extra copies in `smdev`. Moreover, the `smdev` device obtains the highest performance (up to 71.2 Gbps) especially when messages are around the L1 cache size (32 KB). When the message does not fit in the L2 cache (256 KB), the performance gap between `smdev` and MPI reduces, which evidences the impact of the memory hierarchy on shared memory performance, as no cache invalidation is performed in this test, as mentioned before.

The performance results on the AMD-SHM testbed (bottom right graph) show a similar pattern. Thus, MPI obtains lower start-up latencies than `smdev`, 0.88 and 1.53 $\mu$s, respectively, but relatively high compared to the Intel-SHM ones owing to the lower computational power of the AMD processor core. Regarding large message performance, `smdev` again clearly outperforms MPI libraries, obtaining up to 41.6 Gbps whereas MPI does not even reach 10 Gbps. This poor performance is explained by the low memory access throughput and the high copy penalty in this system. In addition, the peak bandwidth for `smdev` now is obtained for 256 KB (the L2 cache size in this system), not taking advantage of the messages fitting in the L1 cache (64 KB), while in the Intel testbed the peak was for 32 KB (the L1 cache size for this system).

The observed point-to-point communication efficiency of `xxdev` devices allows FastMPJ to provide low-latency and high-bandwidth communications for MPJ parallel applications, both on high-speed networks and high-performance shared memory systems. Furthermore, the obtained results are quite close to native MPI results, even outperforming them in some scenarios (e.g., large message performance in shared memory).

## 6.3 Collective primitives micro-benchmarking

Figure 4 presents the aggregated bandwidth for the broadcast primitive, a representative data movement operation, on the IB-QDR, IB-DDR, MN and AMD-SHM testbeds using all the available cores in each system. The aggregated bandwidth metric has been selected as it takes into account the global amount of data transferred (i.e., $message\ size * number\ of\ processes$).

On the IB-QDR testbed (top left graph), the `ibvdev` device obtains higher bandwidth than MVAPICH2 in the message range (2–256 KB). However, Open MPI is the best performer, especially from 256 KB on. From this point, Open MPI dramatically increases its performance, which suggests that it switchs to a highly efficient algorithm for large messages (the same behaviour has been observed in the remaining scenarios where Open MPI is also evaluated). The IB-DDR testbed results (top right graph) show that `psmdev` is the best performer up to 64-KB messages, from then MVAPICH2 performs slightly better up to 256 KB, but then Open MPI becomes again the best large message performer. Regarding the MN supercomputer (bottom left graph), `mxdev` results are worse than the MPICH-MX ones up to 256 KB, but it shows quite competitive performance and scalability from this point on. Finally, on the AMD-SHM testbed (bottom right graph), `smdev` generally outperforms MVAPICH2 from 2-KB messages and shows results quite close to Open MPI up to 256 KB, although Open MPI benefits, once again, from its better large message performance.

The presented results show that FastMPJ is generally able to obtain performance results for the broadcast operation similar to MPI libraries, even outperforming them in some message ranges. This supports the fact that the MST-based algorithm implemented in the FastMPJ collective library is very efficient (e.g., clearly outperforms MVAPICH on the AMD-SHM testbed) and highly scalable (e.g., large message performance using 1,024 cores on the MN supercomputer). Therefore, these results confirm that FastMPJ is bridging the gap between MPJ and MPI collectives performance. Nevertheless, there is still potential room for improvement, especially

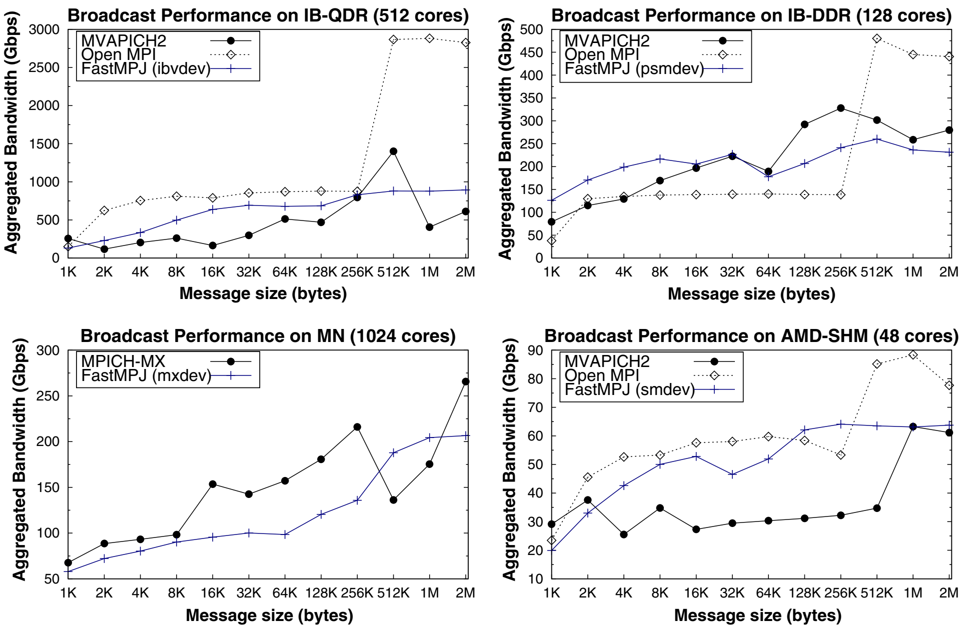


**Fig. 4** Broadcast performance on InfiniBand QDR and DDR, Myrinet and shared memory

for large message bandwidth, which means that enhanced collective algorithms and techniques need to be explored in order to achieve the high performance shown by Open MPI.

## 6.4 HPC Kernel performance analysis

The performance analysis of representative HPC kernels has been carried out using both strong (Sect. 6.4.1) and weak (Sect. 6.4.2) scaling models. The metrics considered for this evaluation using the NPB suite are Millions of Operations Per Second (MOPS), which measures the operations performed in the benchmark (and which differs from the CPU operations issued), and their corresponding speedups and efficiencies for the strong and weak scaling models, respectively.

### *6.4.1 Strong scaling*

In this first set of experiments, the problem size is fixed using the NPB class C workload while the number of cores is increased, hence applying a strong scaling model. These experiments have been conducted on the IB-QDR and IB-DDR testbeds, selected as they are the most representative distributed memory systems under evaluation. Thus, both multi-core clusters provide an IB interconnection network from the major current vendors (Mellanox and Intel/Q-Logic, respectively). Furthermore, in recent years, IB has become the most widely adopted networking technology in the TOP500 list. Additionally, both shared memory testbeds (Intel-SHM and AMD-SHM) have also been included in this analysis, as they provide with representative Intel- and AMD-based processors, respectively.

Figure 5 shows the NPB kernels performance on the IB-QDR testbed in terms of MOPS (left graphs) and its corresponding speedups (right graphs) using up to 512 cores. Regarding CG, FastMPJ and MPI show very similar results using up to 64 cores, as the scalability of this kernel is strongly based on point-to-point data transfers where FastMPJ and MPI achieve comparable performance, as has been observed before in the point-to-point micro-benchmarking. From 64 cores, `ibvdev` starts to suffer the current limitation of not being able to take advantage of intra-node communications, which seems to aggravate when the number of cores increases, as more communications have to be performed accessing the NIC instead of using a shared memory approach. This fact allows MPI libraries to obtain the highest performance and speedup from 128 cores on, but FastMPJ results remain competitive at least compared to MVAPICH2. FT results show that, while FastMPJ performance is on average 25 % lower than MPI, the reported speedups are quite similar. In this case, FastMPJ performance is limited by its poor performance on a single core, as this kernel presents the largest performance gap between Java and native implementations (approximately 35 % less performance). In addition, the FT kernel makes an intensive use of Alltoall collective operations, which has not prevented FastMPJ scalability. The performance and scalability of FastMPJ for the IS kernel is quite similar to Open MPI, although the maximum observed speedups are significantly low (below 60 on 256 cores for MVAPICH2). The implementation of this kernel relies heavily on Alltoall and Allreduce primitives, whose overhead is the main performance penalty, especially when using more than 256 cores on this testbed (all evaluated middleware drops in performance from this point). Finally, the MG kernel is the least communication-intensive code under evaluation; it shows relatively high speedups (above 300 on 512 cores) both for FastMPJ and MPI.

Figure 6 shows the NPB kernels performance on the IB-DDR testbed using up to 128 cores. CG results on this system show that FastMPJ is able to match the performance and speedup of MVAPICH2. In this scenario, all the middleware relies on the same underlying low-level communication subsystem (the PSM library). Thus, PSM implements the communication protocols and ultimately determines the point-to-point performance both for inter-node and intra-node communications, which prevents MPI libraries to use their own shared memory protocol (PSM already provides efficient shared memory support). Regarding the FT kernel, FastMPJ obtains the highest speedup when using 128 cores, although its performance is around 30 % lower than MVAPICH2 due to the poor Java serial performance, as mentioned before. The IS kernel shows again the poorest scalability (below 30 on 128 cores), where FastMPJ is able to achieve the same performance as MPI libraries using up to 64 cores. For MG, FastMPJ shows again the highest speedups, especially on 128 cores, motivated by the different serial runtime of the native and Java implementation (30 % gap in this testbed). This also causes that FastMPJ obtains lower performance than MPI on 128 cores (around 20 %).

Regarding shared memory systems, Figs. 7 and 8 show the NPB kernels performance on the Intel-SHM and AMD-SHM testbeds, respectively, using up to 32 cores. The comparison on this scenario also includes the results from the OpenMP implementation of the NPB kernels. On the one hand, Intel-SHM results show that OpenMP is generally the best performer, both in terms of MOPS and scalability, except for the MG kernel where FastMPJ obtains the highest speedup. In addition, FastMPJ is able to achieve better performance than MPI for the CG kernel, taking advantage of the higher bandwidth obtained by `smdev`, whereas for the remaining kernels FastMPJ shows competitive results compared to MPI using up to 16 cores. On the other hand, results on the AMD-SHM testbed show that: (1) FastMPJ is able to outperform all the middleware for the CG kernel using up to 16 cores; (2) it obtains similar results as Open MPI for FT; and (3) it outperforms OpenMP and gets comparable performance to MVAPICH2 for the IS and MG kernels, using up to 16

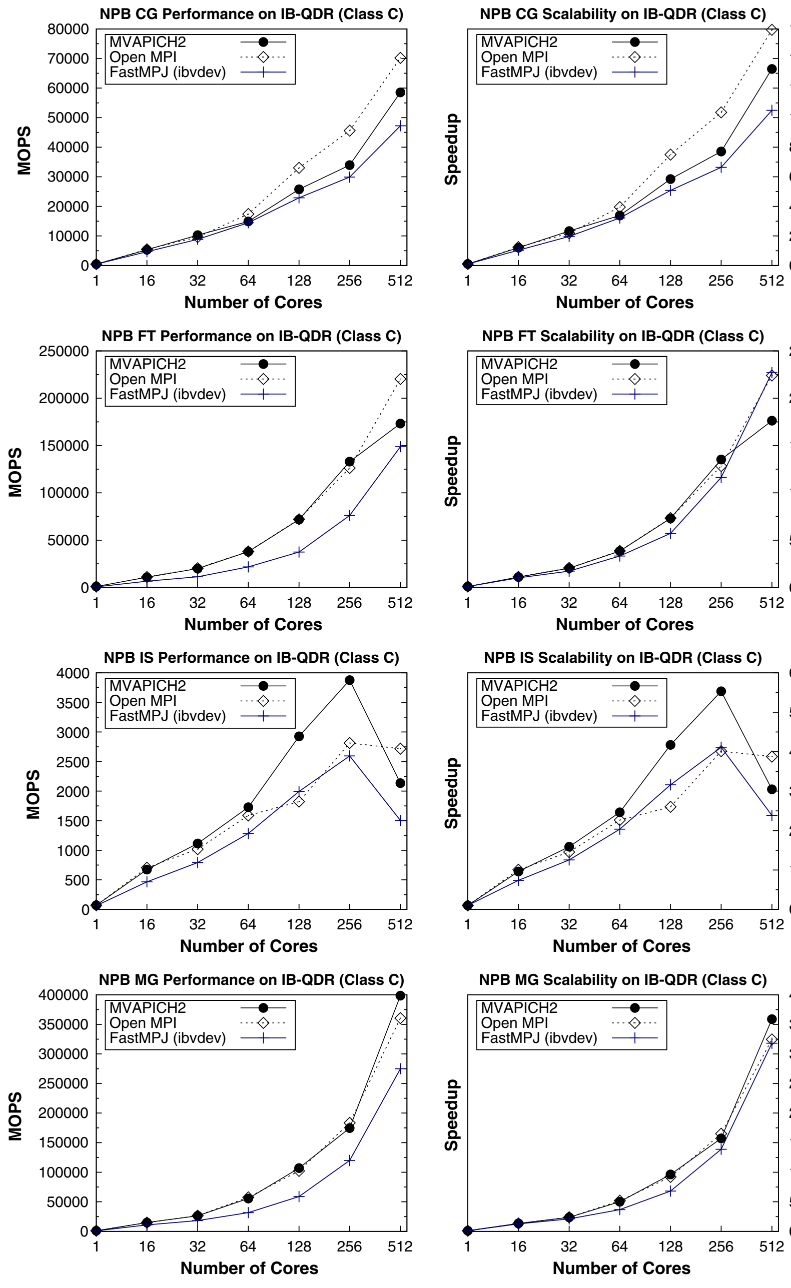


**Fig. 5** NPB kernel results on the IB-QDR testbed (strong scaling)

cores. However, the AMD system generally obtains lower performance than the Intel system for all the evaluated middleware, due to its lower computational power per core and poorer memory access throughput, which limits the obtained speedups.

### *6.4.2 Weak scaling*

In the case of weak scaling, the problem size increases with the number of cores so that the workload per core remains constant. In our experiments, the NPB Class C are solved

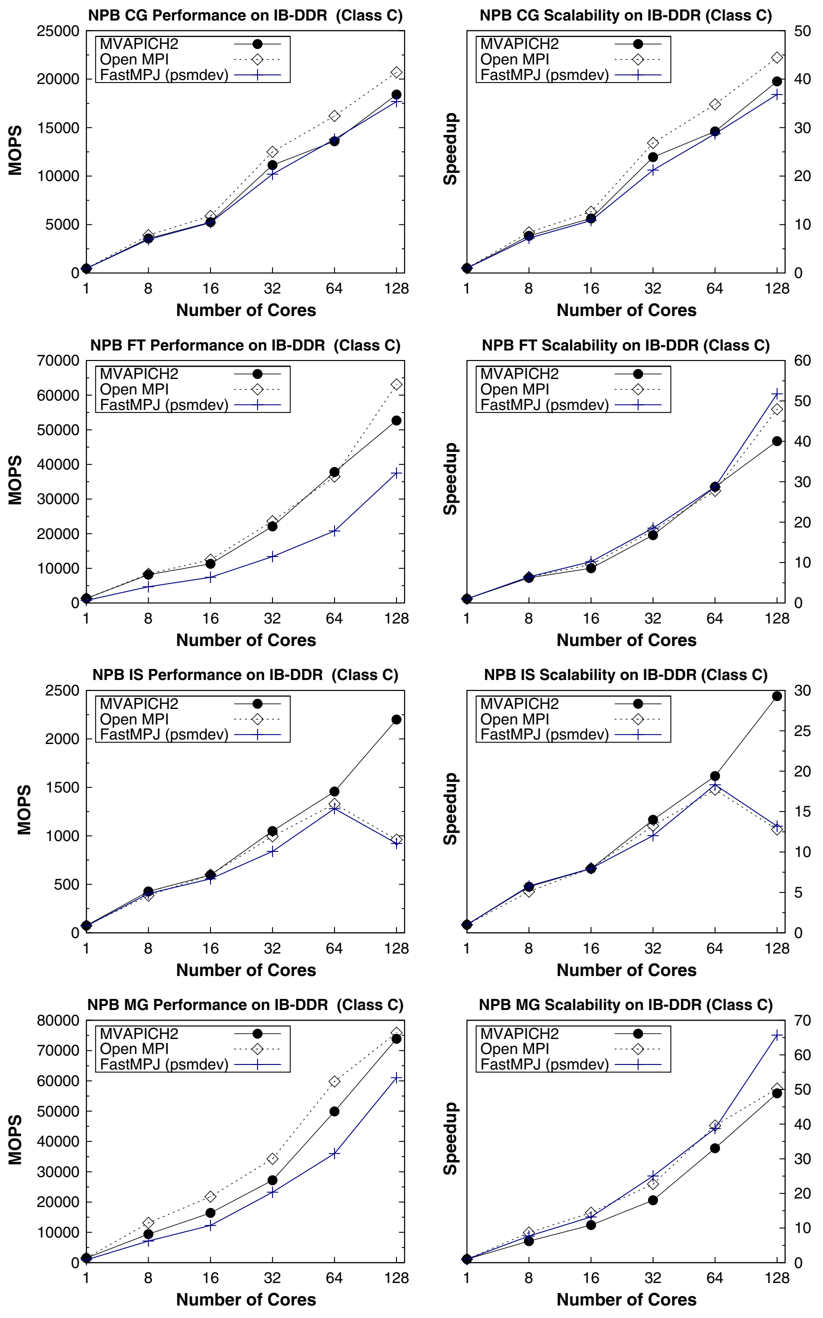


**Fig. 6** NPB kernel results on the IB-DDR testbed (strong scaling)

using a quarter of the number of available cores. Maintaining a fixed workload per core, results are reported from a workload of Class C divided by 8 up to 4 times Class C. Thus, the problem size is scaled lineraly with the core count, as will be shown in the X-axis of the graphs (see Figs. 9, 10). This set of experiments has been conducted on the IB-QDR and Intel-SHM testbeds, selected as representative distributed and shared memory systems, respectively, which,

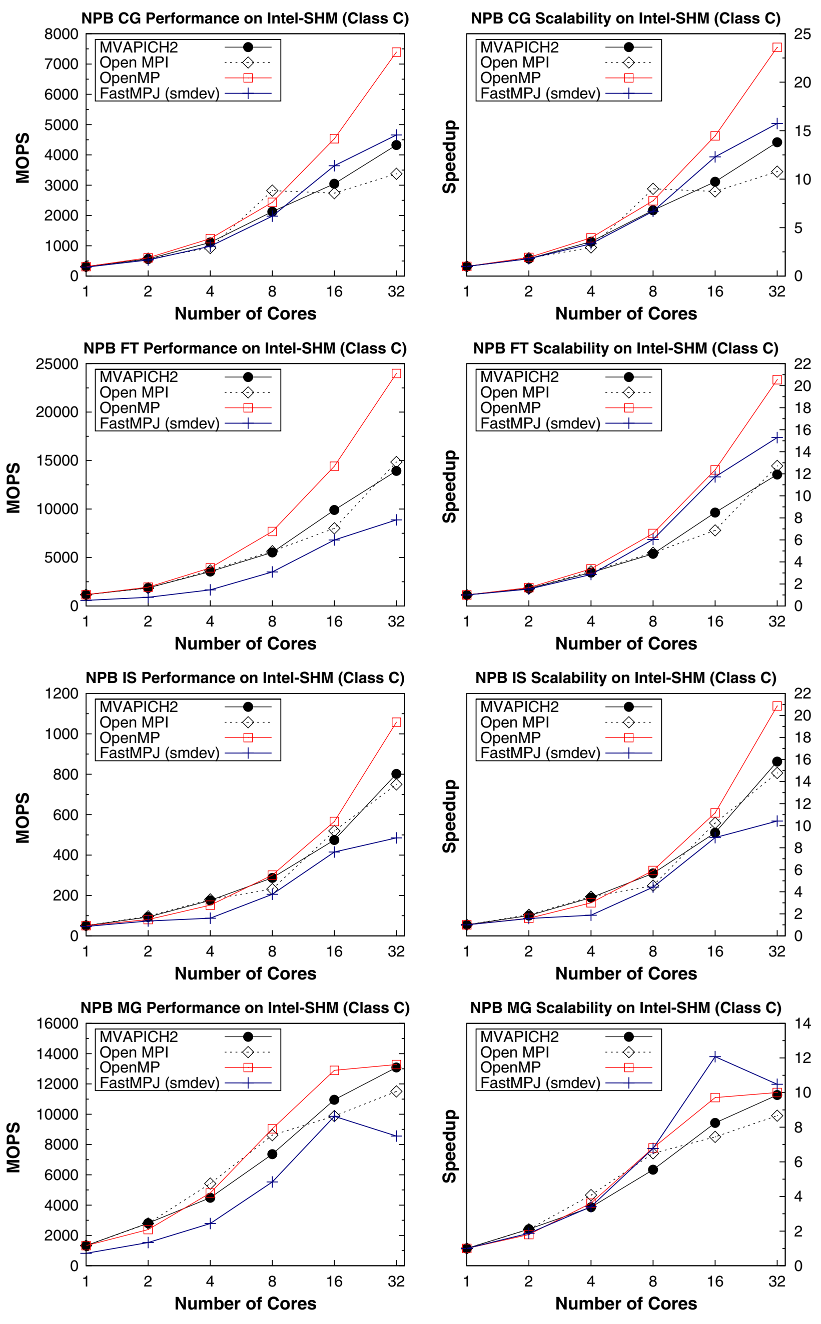


**Fig. 7** NPB kernel results on the Intel-SHM testbed (strong scaling)

according to the previous strong scaling evaluation, have shown the best performance results. In addition, as the NPB weak scaling results were, in general, quite similar to the previous strong scaling counterparts, both in terms of MOPS and speedups, only results for CG and FT kernels are shown for clarity purposes.

NPB weak scaling results are shown in MOPS (as in the case of strong scaling) together with their corresponding scal-

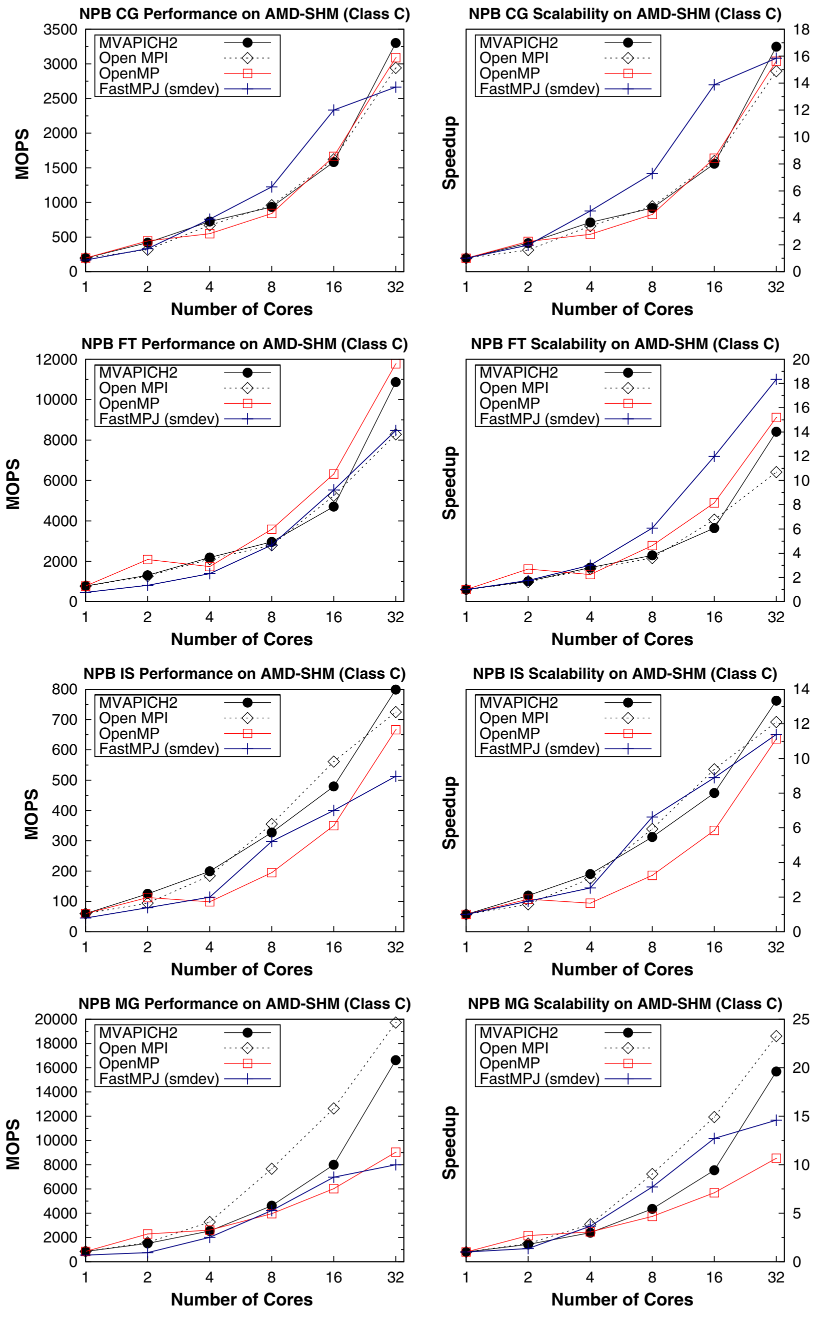


**Fig. 8** NPB kernel results on the AMD-SHM testbed (strong scaling)

ing efficiencies, instead of speedups. Note that the scaling efficiency metric has not been calculated as a percentage of the linear speedup, because usually can not be achieved. Instead, an upper bound on performance has been estimated for each core count using the serial code with the corresponding problem size. Thus, running multiple serial processes concurrently (as many processes per node as the number of cores under evaluation) takes into account the overhead asso-

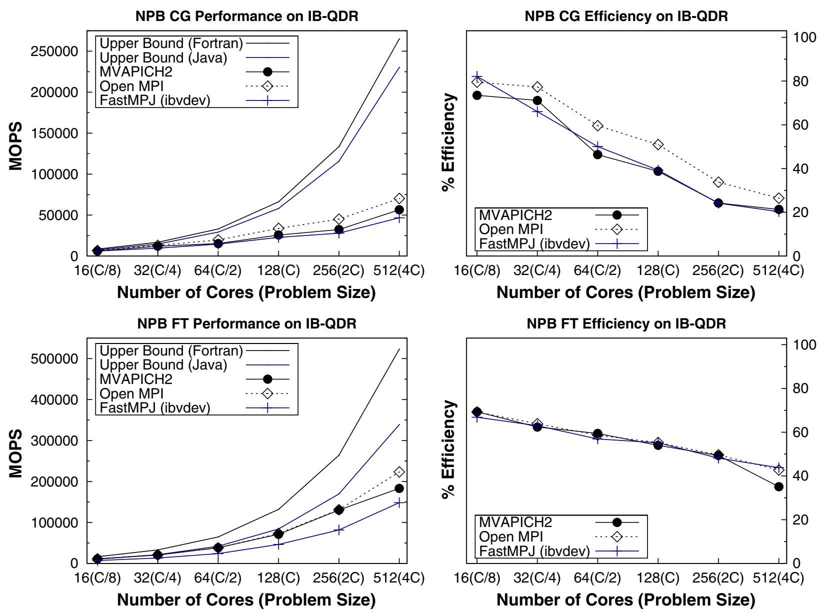


**Fig. 9** NPB kernel results on the IB-QDR testbed (weak scaling)

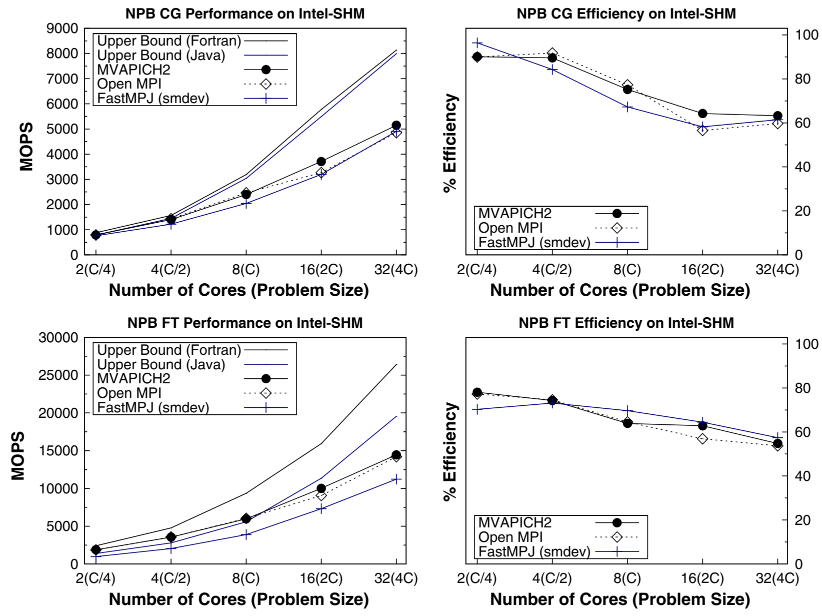


**Fig. 10** NPB kernel results on the Intel-SHM testbed (weak scaling)

ciated with several processes accessing some shared levels of cache and memory, which prevents obtaining the linear speedup. As an example, the upper bound performance for the FT kernel has achieved a speedup of 458 on 512 cores on IB-QDR, and 21 on 32 cores on Intel-SHM. Additionally, as there is no IPC involved in the estimation of this value, it also represents an upper bound on performance if it were possible to perform zero-latency communications. Therefore, the efficiency of the corresponding parallel code calculated as a percentage of this estimated upper bound value can serve as a reliable metric to measure the communication efficiency of message-passing libraries. As there are no explicit communication routines in the OpenMP standard, NPB-OMP results are not shown in the Intel-SHM testbed.

NPB results on the IB-QDR and Intel-SHM testbeds are presented in Figs. 9 and 10, respectively. On the one hand, the CG kernel shows that Java can obtain an upper bound performance quite similar to Fortran when no communication is involved, especially in the Intel-SHM testbed. In this scenario, FastMPJ is able to almost match the performance of at least one of the MPI libraries on both testbeds (MVAPICH2 on IB-QDR and OpenMPI on Intel-SHM). Consequently, the communication efficiency of FastMPJ is in tune with MPI libraries, as shown in the right graphs, especially for the higher core counts. On the other hand, the FT kernel results show that in this case the upper bound performance for Java is limited by its poor performance on a single core, which is on average around 60 % of Fortran's performance. The main performance penalty is the lack of a high-performance numerical library in Java that would implement the Fourier transform, which is the most computationally intensive part of this kernel. However, while FastMPJ performance is on average 20 % lower than MPI for the higher core counts, the reported efficiencies are quite similar. Thus, this fact confirms that the underlying communication support implemented by FastMPJ is able to achieve comparable performance to MPI.

To sum up, the NPB results using both scaling metrics have shown that FastMPJ is able to rival native MPI performance and scalability, even outperforming MPI in some scenarios (e.g., CG kernel on IB-DDR and shared memory systems). This allows Java to take advantage of the use of a high number of cores, especially on shared memory and hybrid shared/distributed memory architectures, widely extended nowadays.

## 7 Conclusions

The continuous increase in the number of cores per system underscores the need for scalable parallel solutions both in shared and distributed memory architectures, where the efficiency of the underlying communication middleware is fundamental. In fact, the scalability of Java message-passing parallel applications depends heavily on the communications performance. However, current Java communication middleware lacks efficient communication support, especially in the presence of high-speed cluster networks and shared memory systems.

This paper has presented FastMPJ, a scalable and efficient Java message-passing library for parallel computing, which overcomes these performance constraints by: (1) providing thread-based high-performance shared memory communications which obtains sub-microsecond start-up latencies and up to 71.2 Gbps bandwidth; (2) enabling low-latency (less than 2 $\mu$s) and high bandwidth communications (higher than 22 Gbps) on RDMA-capable high-speed cluster networks (e.g., InfiniBand); (3) including a scalable collective library with more than 60 topology aware algorithms, which are automatically selected at runtime; (4) avoiding Java data buffering overheads through efficient zero-copy protocols; and (5) implementing the mpiJava 1.2 API, the most widely extended MPI-like Java bindings, for a highly productive development of MPJ parallel applications.

FastMPJ has been evaluated comparatively with native MPI libraries on five representative testbeds: two InfiniBand multi-core clusters, one Myrinet supercomputer, and two shared memory systems using both Intel- and AMD-based processors. The comprehensive performance evaluation has revealed that FastMPJ communication primitives are quite competitive with MPI results, both in terms of point-to-point and collective operations performance. Thus, the use of our message-passing library in communication-intensive HPC codes allows Java to benefit from a more efficient communication support, taking advantage of the use of a high number of cores and improving significantly the performance and scalability of Java parallel applications. In fact, the development of this efficient Java communication middleware is definitely bridging the gap between Java and native languages in HPC applications. Further information of this project is available at http://torusware.com.

**Acknowledgments** This work has been funded by the Ministry of Education of Spain (FPU Grant AP2010-4348), the Ministry of Economy and Competitiveness (project TIN2010-16735) and the Galician Government (projects CN2012/211 and GRC2013/055), partially supported by FEDER funds. We thankfully acknowledge the computer resources, technical expertise and assistance provided by the Barcelona Supercomputing Center. We also gratefully thank the Advanced School for Computing and Imaging (ASCI) and the Vrije University Amsterdam for providing access to the DAS-4 cluster.

## References

1. Taboada, G.L., Ramos, S., Expósito, R.R., Touriño, J., Doallo, R.: Java in the high performance computing arena: research, practice and experience. Sci. Comput. Program. **78**(5), 425–444 (2013)
2. Blount, B., Chatterjee, S.: An evaluation of Java for numerical computing. Sci. Program. **7**(2), 97–110 (1999)
3. Shafi, A., Carpenter, B., Baker, M., Hussain, A.: A comparative study of Java and C performance in two large-scale parallel applications. Concurr. Comput. Pract. Exp. **21**(15), 1882–1906 (2009)
4. Suganuma, T., Ogasawara, T., Takeuchi, M., Yasue, T., Kawahito, M., Ishizaki, K., Komatsu, H., Nakatani, T.: Overview of the IBM Java just-in-time compiler. IBM Syst. J. **39**(1), 175–193 (2000)
5. Message Passing Interface Forum: MPI: a message passing interface standard. http://www.mcs.anl.gov/research/projects/mpi/ (1995). Accessed Oct 2013
6. IBTA: The InfiniBand Trade Association. http://www.infinibandta.org/. Accessed Oct 2013
7. Carpenter, B., Getov, V., Judd, G., Skjellum, A., Fox, G.: MPJ: MPI-like message passing for Java. Concurr. Comput.: Pract. Exp. **12**(11), 1019–1038 (2000)

8. Carpenter, B., Fox, G., Ko, S.H., Lim, S.: mpiJava 1.2: API specification. http://www.hpjava.org/reports/mpiJava-spec/mpiJava-spec/mpiJava-spec.html. Accessed Oct 2013
9. Java Grande Forum. http://www.javagrande.org. Accessed Oct 2013
10. Hongwei, Z., Wan, H., Jizhong, H., Jin, H., Lisheng, Z.: A performance study of Java communication stacks over InfiniBand and Gigabit Ethernet. In: Proceedings of the 4th IFIP International Conference on Network and Parallel Computing—Workshops (NPC'07), pp. 602–607. Dalian, China (2007)
11. Thiruvathukal, G.K., Dickens, P.M., Bhatti, S.: Java on networks of workstations (JavaNOW): a parallel computing framework inspired by Linda and the message passing interface (MPI). Concurr. Comput.: Pract. Exp. **12**(11), 1093–1116 (2000)
12. Baker, M., Carpenter, B., Fox, G., Ko, S.H., Lim, S.: mpiJava: an object-oriented Java interface to MPI. In: Proceedings of the 1st International Workshop on Java for Parallel and Distributed Computing (IWJPDC'99), pp. 748–762. San Juan, Puerto Rico (1999)
13. Baker, M., Carpenter, B., Shafi, A.: MPJ express: towards thread safe Java HPC. In: Proceedings of 8th IEEE International Conference on Cluster Computing (CLUSTER'06), pp. 1–10. Barcelona, Spain (2006)
14. Shafi, A., Manzoor, J., Hameed, K., Carpenter, B., Baker, M.: Multicore-enabling the MPJ express messaging library. In: Proceedings of 8th International Conference on the Principles and Practice of Programming in Java (PPPJ'10), pp. 49–58. Vienna, Austria (2010)
15. Baker, M., Carpenter, B., Shafi, A.: A buffering layer to support derived types and proprietary networks for Java HPC. Scalable Comput. Pract. Exp. **8**(4), 343–358 (2007)
16. Bornemann, M., van Nieuwpoort, R.V., Kielmann, T.: MPJ/Ibis: a flexible and efficient message passing platform for Java. In: Proceedings of 12th European PVM/MPI Users' Group Meeting (EuroPVM/MPI'05), pp. 217–224. Sorrento, Italy (2005)
17. van Nieuwpoort, R.V., Maassen, J., Wrzesinska, G., Hofman, R., Jacobs, C., Kielmann, T., Bal, H.E.: Ibis: a flexible and efficient Java-based grid programming environment. Concurr. Comput.: Pract. Exp. **17**(7–8), 1079–1107 (2005)
18. Myrinet Express (MX): A High Performance, Low-level, Message-Passing Interface for Myrinet, version 1.2, (2006)
19. Taboada, G.L., Touriño, J., Doallo, R.: F-MPJ: scalable Java message-passing communications on parallel systems. J. Supercomput. **60**(1), 117–140 (2012)
20. Taboada, G.L., Touriño, J., Doallo, R.: Java fast sockets: enabling high-speed Java communications on high performance clusters. Comput. Commun. **31**(17), 4049–4059 (2008)
21. Taboada, G.L., Touriño, J., Doallo, R., Shafi, A., Baker, M., Carpenter, B.: Device level communication libraries for high-performance computing in Java. Concurr. Comput.: Pract. Exp. **23**(18), 2382–2403 (2011)
22. Bonachea, D., Dickens, P.M., Thakur, R.: High-performance file I/O in Java: existing approaches and bulk I/O extensions. Concurr. Comput.: Pract. Exp. **13**(8–9), 713–736 (2001)
23. Dickens, P.M., Thakur, R.: An evaluation of Java's I/O capabilities for high-performance computing. In: Proceedings of 1st ACM Java Grande Conference (JAVA'00), pp. 26–35. San Francisco, CA, USA (2000)
24. Baker, M., Carpenter, B., Shafi, A.: A pluggable architecture for high-performance Java messaging. IEEE Distrib. Syst. Online **6**(10), 1–4 (2005)
25. Goglin, B.: High-performance message passing over generic Ethernet hardware with Open-MX. Parallel Comput. **37**(2), 85–100 (2011)
26. Goglin, B.: High throughput intra-node MPI communication with Open-MX. In: Proceedings of 17th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP'09), pp. 173–180. Weimar, Germany (2009)
27. OpenFabrics Alliance: http://www.openfabrics.org/. Accessed Oct 2013
28. IETF RFC 4392: IP over InfiniBand (IPoIB) Architecture. http://www.ietf.org/rfc/rfc4392.txt.pdf. Accessed Oct 2013
29. Expósito, R.R., Taboada, G.L., Touriño, J., Doallo, R.: Design of scalable Java message-passing communications over InfiniBand. J. Supercomput. **61**(1), 141–165 (2012)
30. Ramos, S., Taboada, G.L., Expósito, R.R., Touriño, J., Doallo, R.: Design of scalable Java communication middleware for multi-core systems. Comput. J. **56**(2), 214–228 (2013)
31. Chan, E., Heimlich, M., Purkayastha, A., van de Geijn, R.A.: Collective communication: theory, practice, and experience. Concurr. Comput.: Pract. Exp. **19**(13), 1749–1783 (2007)
32. Open MPI: Open Source High Performance Computing. http://www.open-mpi.org/. Accessed Oct 2013
33. MVAPICH: MPI over InfiniBand, 10GigE/iWARP and RoCE. http://mvapich.cse.ohio-state.edu/. Accessed Oct 2013
34. Portable MPI Model Implementation over MX. https://www.myricom.com/support/downloads/mx/mpich-mx.html. Accessed Oct 2013
35. NAS Parallel Benchmarks. http://www.nas.nasa.gov/publications/npb.html. Accessed Oct 2013
36. Bailey, D.H., et al.: The NAS parallel benchmarks. Int. J. High Perform. Comput. Appl. **5**(3), 63–73 (1991)
37. Advanced School for Computing and Imaging (ASCI): Distributed ASCI Supercomputer, Version 4 (DAS-4). http://www.cs.vu.nl/das4/. Accessed Oct 2013
38. MareNostrum supercomputer in TOP500 List. http://www.top500.org/system/8242. Accessed Oct 2013
39. TOP500 Org.: Top 500 Supercomputer Sites. http://www.top500.org/ Accessed Oct 2013
40. Saini, S., et al.: Performance evaluation of supercomputers using HPCC and IMB benchmarks. J. Comput. Syst. Sci. **74**(6), 965–982 (2008)
41. Mallón, D.A., Taboada, G.L., Touriño, J., Doallo, R.: NPB-MPJ: NAS parallel benchmarks implementation for message-passing in Java. In: Proceedings of 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP'09), pp. 181–190. Weimar, Germany (2009)

**Roberto R. Expósito** received the B.S. (2010) and M.S. (2011) degrees in Computer Science from the University of A Coruña, Spain. Currently he is a Ph.D. student in the Department of Electronics and Systems at the University of A Coruña. His research interests are in the area of High Performance Computing (HPC), focused on message-passing communications on high-speed cluster networks and cluster/cloud computing.

**Sabela Ramos** received the B.S. (2009), M.S. (2010) and Ph.D. (2013) degrees in Computer Science from the University of A Coruña, Spain. Currently she is a Teaching Assistant in the Department of Electronics and Systems at the University of A Coruña. Her research interests are in the area of High Performance Computing (HPC), focused on message-passing communications on multicore architectures and cluster/-cloud computing. Her homepage is http://gac.udc.es/~sramos.

**Guillermo L. Taboada** received the B.S. (2002), M.S. (2004) and Ph.D. (2009) degrees in Computer Science from the University of A Coruña, Spain. Currently he is an Associate Professor in the Department of Electronics and Systems at the University of A Coruña. His main research interest is in the area of High Performance Computing (HPC), focused on high-speed networks, programming languages for HPC, cluster/cloud computing and, in general, middleware for HPC. He is coauthor of more than 25 technical papers on these topics. His homepage is http://gac.udc.es/~gltaboada.

**Juan Touriño** received the B.S. (1993), M.S. (1993) and Ph.D. (1998) degrees in Computer Science from the University of A Coruña, Spain. In 1993 he joined the Department of Electronics and Systems at the University of A Coruña, where he is currently a Full Professor of computer engineering. He has extensively published in the areas of compilers and programming languages for High Performance Computing (HPC), and parallel and distributed computing. He is coauthor of more than 120 technical papers on these topics. His homepage is http://gac.udc.es/~juan.

**Ramón Doallo** received the B.S. (1987), M.S. (1987) and Ph.D. (1992) degrees in Physics from the University of Santiago de Compostela, Spain. In 1990 he joined the Department of Electronics and Systems at the University of A Coruña, Spain, where he became a Full Professor in 1999. He has extensively published in the areas of computer architecture, and parallel and distributed computing. He is coauthor of more than 140 technical papers on these topics. His homepage is http://gac.udc.es/~doallo.

# Chapter 4

# Java Communication Devices on RDMA-enabled Networks

The content of this chapter corresponds to the following paper:

- **Title:** Low-latency Java communication devices on RDMA-enabled networks

- **Authors:** Roberto R. Expósito, Guillermo L. Taboada, Sabela Ramos, Juan Touriño, Ramón Doallo

- **Journal:** Submitted for journal publication

- **Year:** 2014

A copy of the submitted paper has been included next.

# Low-latency Java communication devices on RDMA-enabled networks

Roberto R. Expósito[*,†], Guillermo L. Taboada, Sabela Ramos, Juan Touriño and Ramón Doallo

*Computer Architecture Group, Department of Electronics and Systems, University of A Coruña, Spain*

## SUMMARY

Providing high-performance inter-node communication is a key capability for running High Performance Computing (HPC) applications efficiently on parallel architectures. In fact, current systems deployments are aggregating a significant number of cores interconnected via advanced networking hardware with Remote Direct Memory Access (RDMA) mechanisms, that enable zero-copy and kernel-bypass features. The use of Java for parallel programming is becoming more promising thanks to some useful characteristics of this language, particularly its built-in multithreading support, portability, easy-to-learn properties and high productivity, along with the continuous increase in the performance of the Java Virtual Machine (JVM). However, current parallel Java applications usually suffer from inefficient communication middleware, mainly based on protocols with high communication overhead (e.g., sockets-based) that do not take full advantage of RDMA-enabled networks. This paper presents efficient low-level Java communication devices that overcome these constraints by fully exploiting the underlying RDMA hardware, providing low-latency and high-bandwidth communications for parallel Java applications. The performance evaluation conducted on representative RDMA networks and parallel systems has shown significant point-to-point performance increases compared to previous Java communication middleware, allowing to obtain up to 40% improvement in application-level performance on 4096 cores of a Cray XE6 supercomputer.

KEY WORDS: Parallel systems; Remote Direct Memory Access (RDMA); RDMA-enabled networks; Java communication middleware; Message-Passing in Java (MPJ)

## 1. INTRODUCTION

Java is a highly portable and flexible programming language, enjoying a dominant position in a wide diversity of computing environments. Some of the interesting features of Java are its built-in multithreading support in the core of the language, object orientation, automatic memory management, type-safety, platform independence, portability, easy-to-learn properties and thus higher productivity. Furthermore, Java has become the leading programming language both in academia and industry.

The Java Virtual Machine (JVM) is currently equipped with efficient Just-in-Time (JIT) compilers that can obtain near-native performance from the platform independent bytecode [1]. In fact, the JVM identifies sections of the code frequently executed and converts them to native machine code instead of interpreting the bytecode. This significant improvement in its computational performance has narrowed the performance gap between Java and natively compiled languages (e.g., C/C++, Fortran). Thus, Java is currently gaining popularity in other domains which usually make use of

[*]Correspondence to: Roberto R. Expósito, Department of Electronics and Systems, University of A Coruña, Campus de Elviña s/n, 15071, A Coruña, Spain
[†]E-mail: rreye@udc.es

High Performance Computing (HPC) infrastructures, such as the area of parallel computing [2, 3] or in Big Data analytics, where the Java-based Hadoop distributed computing framework [4] is among the preferred choices for the development of applications that follow the MapReduce programming model [5].

With the continuously increasing number of cores in current HPC systems to meet the ever growing computational power needs, it is vitally important for communication middleware to provide efficient inter-node communications on top of high-performance interconnects. Modern networking hardware provides Remote Direct Memory Access (RDMA) capabilities that enable zero-copy and kernel-bypass features, key mechanisms for obtaining scalable application performance. However, it is usually difficult to program directly with RDMA hardware. In this context, it is fundamental to fully harness the power of the likely abundant processing resources and take advantage of the interesting features of RDMA networks with still ease-to-use programming models. The Message-Passing Interface (MPI) [6] remains as the de-facto standard in the area of parallel computing, being the most widely extended programming model for writing C/C++ and Fortran parallel applications, but remains out of the scope of Java. The main reason is that current parallel Java applications usually suffer from inefficient communication middleware, mainly based on protocols with high overhead (e.g., sockets-based) that do not take full advantage of RDMA-enabled networks [7]. The lack of efficient RDMA hardware support in current Message-Passing in Java (MPJ) [8] implementations usually results in lower performance than natively compiled codes, which has prevented the use of Java in this area. Thus, the adoption of Java as a mainstream language on these systems heavily depends on the availability of efficient communication middleware in order to benefit from its appealing features at a reasonable overhead.

This paper focuses on providing efficient low-level communication devices that overcome these constraints by fully exploiting the underlying RDMA hardware, enabling low-latency and high-bandwidth communications for Java message-passing applications. The performance evaluation conducted on representative RDMA networks and parallel systems has shown significant point-to-point performance improvements compared to previous Java message-passing middleware, in addition to higher scalability for communication-intensive HPC codes. These communication devices have been integrated seamlessly in the FastMPJ middleware [9], our Java message-passing implementation, in order to make them available for current MPJ applications. Therefore, this paper presents our research results on improving the RDMA network support in FastMPJ, which would definitely contribute to increase the use of Java in parallel computing. More specifically, the main contributions of this paper are:

- The design and implementation of two new low-level communication devices, `ugnidev` and `mxmdev`. The former device is intended to provide efficient support for the RDMA networks used by the Cray XE/XK/XC family of supercomputers. The latter includes support for the recently released messaging library developed by Mellanox for its RDMA adapters.
- An enhanced version of the `ibvdev` communication device for InfiniBand systems [10], which now includes new support for RDMA networks along with an optimized communication protocol to improve short-message performance.
- The transparent integration of these communication devices in the FastMPJ middleware in order to allow existing MPJ applications to take full advantage of them without any source code modification. Additionally, this also makes it possible their performance evaluation at the MPJ level.
- An experimental comparison of representative MPJ middleware, which includes a micro-benchmarking of point-to-point primitives on several RDMA networks, and an application-level performance analysis conducted on two parallel systems: a multi-core InfiniBand cluster and a large Cray XE6 supercomputer.

The remainder of this paper is organized as follows. Section 2 presents background information about RDMA networks and their software support. Section 3 introduces the related work. Section 4 presents the overall design of `xxdev`, the low-level communication device layer included in FastMPJ. This is followed by Sections 5, 6 and 7, which describe the design and implementation of the new `xxdev` communication devices presented in this paper: `ugnidev`, `ibvdev` and `mxmdev`, respectively. Section 8 shows the performance results of the developed devices gathered from a micro-benchmarking of point-to-point primitives on several RDMA networks. Next, this section analyzes the impact of their use on the overall performance of representative Java HPC codes. Finally, our concluding remarks are summarized in Section 9.

## 2. OVERVIEW OF RDMA-ENABLED NETWORKS

Most high-performance clusters and custom supercomputers are deployed with high-speed interconnects. These networking technologies typically rely on scalable topologies and advanced network adapters that provide RDMA-capable specialized hardware to enable zero-copy and kernel-bypass facilities. Some of the main benefits of using RDMA hardware are low-latency and high-bandwidth inter-node communication with low CPU overhead.

In recent years, the InfiniBand (IB) architecture [11] has become the most widely extended RDMA networking technology in the TOP500 list [12], especially for multi-core clusters. In addition, two other popular RDMA implementations, the Internet Wide Area RDMA Protocol (iWARP) [13] and RDMA over Converged Ethernet (RoCE) [14], have also been proposed to extend the advantages of RDMA technologies to ubiquitous Internet Protocol (IP)/Ethernet-based networks. On the one hand, iWARP defines how to perform RDMA over a connection-oriented transport such as the Transmission Control Protocol (TCP). Thus, iWARP includes a TCP Offload Engine (TOE) to offload the whole TCP/IP stack onto the hardware, while the Direct Data Placement (DDP) protocol [15] implements the zero-copy and kernel-bypass mechanisms. On the other hand, RoCE takes advantage of the more recent enhancements to the Ethernet link layer. The IEEE Converged Enhanced Ethernet (CEE) is a set of standards, defined by the Data Center Bridging (DCB) task group [16] within IEEE 802.1, which are intented to make Ethernet reliable and lossless (like IB). This allows the IB transport protocol to be layered directly over the Ethernet link layer. Hence, RoCE utilizes the same transport and network layers from the IB stack and swaps the link layer for Ethernet, providing IB-like performance and efficiency to ubiquitous Ethernet infrastructures. Compared to iWARP, RoCE is a more natural extension of message-based transfers, and therefore usually offers better efficiency than iWARP. However, one disadvantage of RoCE is that it does not operate with standard Ethernet switches, as it requires DCB-capable ones.

Although the current market is dominated by clusters, many of the most powerful computing installations are custom supercomputers [12] that usually rely on specifically designed Operating Systems (OS) and proprietary RDMA-enabled interconnects. Some examples are the IBM Blue Gene/Q (BG/Q) and the Cray XE/XK/XC family of supercomputers. On the one hand, the compute nodes of the IBM BG/Q line are interconnected via a custom 5D torus network [17]. On the other hand, Cray XE/XK architectures include the Gemini interconnect [18] based on a 3D torus topology, while the XC systems provide the Aries interconnect that uses a novel network topology called Dragonfly [19].

### *2.1. Software support*

The IB architecture has no standard Application Programming Interface (API) within the specification, as it only defines the functionality provided by the RDMA adapter in terms of an abstract and low-level interface called Verbs[†]. The de-facto standard has been the implementation of the Verbs interface developed by the OpenFabrics Alliance (OFA) [20], which includes both user- and kernel-level APIs. This open-source software stack has been adopted by most vendors and is released as the OpenFabrics Enterprise Distribution (OFED). As a software stack, OFED spans both the OS kernel, providing hardware-specific drivers, and the user space, implementing the Verbs interface. Although OFED was originally developed to work over IB networks, currently

[†]A *verb* is a semantic description of a function that must be provided.

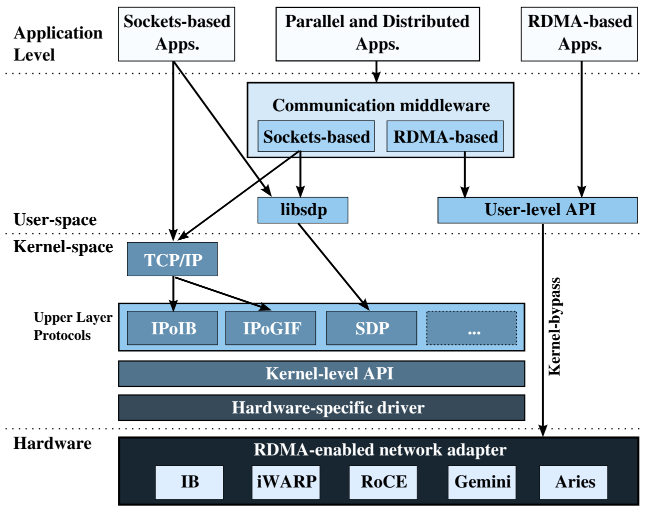


Figure 1. Overview of the RDMA software stack

also includes support for iWARP and RoCE. Hence, it offers a uniform and transport-independent low-level API for the development of RDMA and kernel-bypass applications on IB, iWARP and RoCE interconnects. In addition to the OFED stack, some vendors provide additional user-space libraries that are specifically designed for their RDMA hardware. Examples of these libraries are the Performance Scaled Messaging (PSM) and MellanoX Messaging (MXM), which are currently available for Intel/QLogic and Mellanox adapters, respectively. These libraries can offer a higher level API than Verbs, usually also matching some of the needs of upper level communication middleware (e.g., message-passing libraries). Regarding supercomputer systems, vendors provide a specific interface to their custom interconnects intended to be used for user-space communication. These interfaces are usually low-level APIs that directly expose the RDMA capabilities of the hardware (like Verbs), on top of which the communication middleware and applications can be implemented. For instance, IBM includes the System's Programming Interface (SPI) to program the torus-based interconnect of the BG/Q system, while Cray provides two different interfaces for implementing communication libraries targeted for Gemini/Aries interconnects: Generic Network Interface (GNI) and Distributed Memory Application (DMAPP). It is worth noting that all these programming interfaces are only available in C and therefore any communication support from Java must resort to the Java Native Interface (JNI).

Finally, existing sockets-based middleware and applications are usually able to run over RDMA networks without rewriting, using additional extensions known as Upper Layer Protocols (ULP). Examples of ULPs are the IP emulation over IB (IPoIB) [21] and the IP over Gemini Fabric (IPoGIF) modules. However, these ULPs are unable to take full advantage of the RDMA hardware, introducing additional TCP/IP processing overhead and performance penalties (e.g., multiple data copies, high CPU utilization) compared to native RDMA interfaces. In order to overcome these issues, some high-performance sockets implementations are available as additional ULPs. For instance, the Sockets Direct Protocol (SDP) [22] provides a user-space preloadable library and kernel module that bypasses the TCP/IP stack to take advantage of the IB/iWARP/RoCE hardware features. However, SDP has limited utility as only applications relying on the TCP/IP sockets API can use it, and other IP stack uses or TCP layer modifications (e.g., IPSec, UDP) cannot benefit from it. In addition, because of the restrictions of the socket interface, SDP can not provide the low latencies of native RDMA. Furthermore, OpenFabrics has recently ended the support for SDP and now is considered deprecated. Figure 1 provides a graphical overview of the described RDMA software support.

## 3. RELATED WORK

There have been several early works about Java for HPC soon after its release that have identified its potential for scientific computing [23, 24]. Moreover, some projects have been focused particularly on Java communication efficiency. These related works can be classified into: (1) Java over the Virtual Interface Architecture (VIA) [25]; (2) Java sockets implementations; (3) Java Remote Method Invocation (RMI) protocol optimizations; (4) Java Distributed Shared Memory (DSM) projects; (5) low-level Java libraries on RDMA networks; and (6) efficient MPJ middleware.

Javia [26] and Jaguar [27] provide access to high-speed cluster networks through VIA. The VIA architecture is one of the several approaches for user-level networking developed in the 90s, which has served as basis for IB. More specifically, Javia reduces data copying using native buffers, and Jaguar acts as a replacement of the JNI layer in the JVM, providing an API to access VIA. Their main drawbacks are the use of particular APIs, the need of modified Java compilers that ties the implementation to a certain JVM, and the lack of non-VIA communication support. Additionally, Javia exposes programmers to buffer management and uses a specific garbage collector.

The socket API is widely extended and can be considered the standard low-level communication layer. Thus, sockets have been the choice for implementing in Java the lowest level of network communication. However, Java sockets lack efficient high-speed network support and HPC tailoring, so they have to resort to inefficient TCP/IP emulations (e.g., IPoIB) for full networking support [7]. Ibis sockets partly solve these issues adding Myrinet support and being the base of Ibis [28], a parallel and distributed Java computing framework. However, Ibis lacks support for current RDMA networks, and its implementation on top of JVM sockets limits the performance benefits to serialization improvements. Aldeia [29] is a proposal of an asynchronous sockets communication layer over IB whose preliminary results were encouraging, but requires an extra-copy to provide asynchronous write operations, which incurs an important overhead, whereas the read method is synchronous. Java Fast Sockets (JFS) [30] is our high-performance Java sockets implementation that relies on SDP (see Figure 1) to support Java communications over IB. JFS avoids the need for primitive data type array serialization and reduces buffering and unnecessary copies. Nevertheless, the use of the socket API still represents an important source of overhead and lack of scalability in Java communications, especially in the presence of high-speed networks [7].

Other related work about performance optimization of Java communications included many efforts in RMI, which is a common communication facility for Java applications. ProActive [31] is a fully portable "pure" Java (i.e., 100% Java) RMI-based middleware for parallel, multithreaded and distributed computing. Nevertheless, the use of RMI as its default transport layer adds significant overhead to the operation of this middleware. Therefore, the optimization of the RMI protocol has been the goal of several projects, such as KaRMI [32], Manta [33], Ibis RMI [28] and Opt RMI [34]. However, the use of non-standard APIs, the lack of portability and the insufficient overhead reductions, still significantly larger than socket latencies, have restricted their applicability. Therefore, although Java communication middleware used to be based on RMI, current middleware use sockets due to their lower overhead.

Java DSM projects are usually based on sockets and thus benefit from socket optimizations, but their performance on top of high-speed networks still suffers from significant communication overheads. In order to reduce their impact, two DSM projects have implemented their communications relying on low-level libraries: CoJVM [35] uses VIA, whereas Jackal [36] includes RDMA support through the Verbs API [37]. Nevertheless, these projects share unsuitable characteristics such as the use of modified JVMs, the need of source code modification and limited interoperability and portability (e.g., Jackal is a Java-to-native compiler that does not provide any API to Java developers, implementing data transfers specifically for Jackal).

Other approaches are low-level Java libraries restricted to specific RDMA networks. For instance, Jdib [38, 39] is a Java encapsulation of the Verbs API through JNI, which increases Java communication performance using directly RDMA mechanisms. The main drawbacks of Jdib are its low-level API (like Verbs) and the JNI call overhead incurred for each Jdib operation (i.e., each function of the Verbs interface has to be wrapped through JNI). jVerbs [40] is a networking API and library for the JVM that offers RDMA semantics and exports the Verbs interface to Java. jVerbs maps the RDMA hardware resources directly into the JVM, allowing Java applications to transfer data without OS involvement. Although jVerbs is able to achieve almost bare-metal performance, its low-level API demands a high programming effort (as with Jdib). Additionally, jVerbs requires specific user drivers for each supported RDMA adapter, as the access to hardware resources in the data path is device specific. Currently, it only supports some models and vendors (e.g., Mellanox ConnectX-2).

Regarding MPJ libraries, there have been several efforts to develop a message-passing framework since the inception of Java. Although the current MPI standard declaration is limited to C and Fortran languages, there have been a number of standardization efforts made towards introducing an MPI-like Java binding. The most widely extended API set has been proposed by the mpiJava [41] developers, known as the mpiJava 1.2 API [42]. Currently, the most relevant implementations of this API are Open MPI Java, MPJ Express and FastMPJ, next presented.

mpiJava [41] consists of a collection of wrapper classes that use JNI to interact with an underlying native MPI library. However, mpiJava can incur a noticeable JNI overhead [43] and presents some inherent portability and interoperability issues derived from the amount of native code that is involved in a wrapper-based implementation (all the methods of the MPJ API have to be wrapped). Moreover, the support of multiple heterogeneous runtime platforms, MPI libraries and JVMs entails a significant maintenance effort, which has led to a non-active development and maintenance for years. More recently, Open MPI [44] has revamped this project and included Java support in the developer code trunk. The Open MPI Java interface is based on the original mpiJava code and integrated as a set of bindings on top of the Open MPI core [45].

MPJ Express [46] presents a modular design which includes a pluggable architecture of low-level communication devices that allows to combine the portability of the "pure" Java shared memory device (`smpdev`) and New I/O (NIO) sockets communications (`niodev`), along with the native Myrinet support (`mxdev`) through JNI, implemented on top of the Myrinet eXpress (MX) library [47]. Additionally, the recently included `hybrid` device allows to use simultaneously the `niodev` and `smpdev` devices for inter- and intra-node communications, respectively. However, the overall design of MPJ Express relies on an internal buffering layer that significantly limits performance and scalability [43]. Moreover, it lacks efficient support for current RDMA networks and includes poorly scalable collective algorithms.

Finally, FastMPJ [9] is our Java message-passing implementation that includes a layered design approach similar to MPJ Express, but avoiding its data buffering overhead by supporting direct communication of any serializable Java object. In addition, FastMPJ includes a scalable collective library which implements up to six algorithms per collective primitive. More details about FastMPJ design and communications support are next presented in Section 4.

## 4. OVERVIEW OF THE FASTMPJ COMMUNICATION DEVICE LAYER

Figure 2 presents a high-level overview of the FastMPJ design, whose point-to-point communication support relies on the `xxdev` device layer for interaction with the underlying hardware. This layer is designed as a simple and pluggable architecture of low-level communication devices that enables its incremental development. Additionally, it also eases the development of new `xxdev` devices reducing their implementation effort, and minimizing the amount of native code needed to support a specific network through JNI, as only a very small number of methods must be implemented. Hence, it allows to combine the portability of "pure" Java communication devices with high-performance network support wrapping native communication libraries through JNI. These `xxdev` devices abstract the particular operation of a communication protocol conforming to an API on top of which FastMPJ implements its communications. Therefore, FastMPJ communication devices must conform with the API provided by the abstract class `xxdev.Device`. The low-level `xxdev` API only provides basic point-to-point communication methods and is not aware of higher level MPI abstractions like communicators. Thus, it is composed of basic message-passing operations such as point-to-point blocking and non-blocking communication methods, including also synchronous

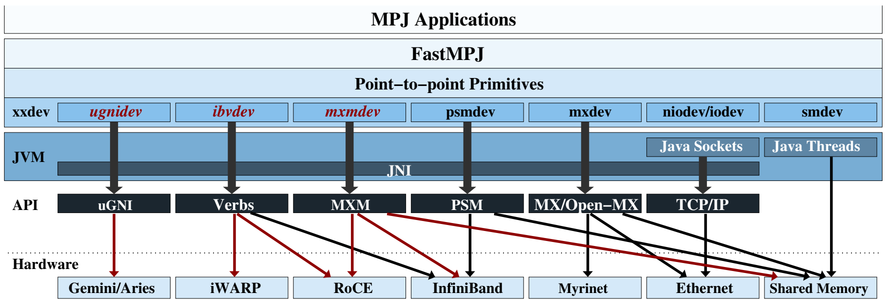


Figure 2. Overview of the FastMPJ communication devices

communications. The use of pluggable low-level devices for implementing the communication support is widely extended in native message-passing libraries, such as the Byte Transfer Layer (BTL) and Matching Transport Layer (MTL), both included in Open MPI [44].

Currently, FastMPJ includes three `xxdev` devices that support RDMA networks (see Figure 2): (1) `mxdev`, for Myrinet adapters and additionally for generic Ethernet hardware; (2) `psmdev`, for the InfiniPath family of IB adapters from Intel/QLogic; and (3) `ibvdev`, for IB adapters in general terms. These devices are implemented on top of MX/Open-MX, InfiniPath PSM and Verbs native communication layers, respectively. Furthermore, the TCP/IP stack support is included through Java NIO (`niodev`) and IO (`iodev`) sockets, whereas high-performance shared memory systems can benefit from the thread-based device (`smdev`).

As mentioned before, this paper presents two new `xxdev` communication devices, `ugnidev` and `mxmdev`, implemented on top of the user-level GNI (uGNI) and MXM native communication layers, respectively. The `mxmdev` device also includes efficient intra-node shared memory communication provided by MXM. An enhanced version of the `ibvdev` device, which extends its current support to RoCE and iWARP networking hardware and introduces an optimized short-message communication protocol, is also included. These communication devices (highlighted in italics and red in Figure 2) have been integrated transparently into FastMPJ thanks to its modular structure. Therefore, the developed devices allow current MPJ applications to benefit transparently from a more efficient support of RDMA networks (depicted by red arrows at the hardware level).

## 5. SCALABLE COMMUNICATIONS ON CRAY SUPERCOMPUTERS: `UGNIDEV`

The Cray XE/XK/XC family is nowadays an important class of custom supercomputers for running highly computationally intensive applications, with several systems ranked in the TOP500 list [12]. A critical component in realizing this level of performance is the underlying network infrastructure. As mentioned in Section 2, the Cray XE/XK architectures include the Gemini interconnect, whereas the newer XC systems are equipped with the Aries interconnect, both providing RDMA capabilities. Cray provides two low-level interfaces for implementing communication libraries targeted for these interconnects: Generic Network Interface (GNI) and Distributed Memory Application (DMAPP). In particular, the GNI API is mainly designed for applications whose communication patterns are message-passing in nature, while the DMAPP interface is geared towards Partitioned Global Address Space (PGAS) languages. Therefore, GNI would be the preferred interface on top of which a message-passing communication device as `ugnidev` should be implemented.

### *5.1. GNI API overview*

The GNI interface exposes a low-level API that is primarily intended for: (1) kernel-space communication through a Linux device driver and the kernel-level GNI (kGNI) implementation; and (2) direct user-space communication through the user-level GNI (uGNI) library, where the driver is used to establish communication domains and handle errors, but can be bypassed for data transfer. Hence, the `ugnidev` device has been layered over the uGNI API, which provides two hardware mechanisms for initiating RDMA transactions using either Fast Memory Access (FMA) or Block Transfer Engine (BTE).

On the one hand, the FMA hardware provides in-order RDMA as a low-overhead, kernel-bypass pathway for injecting messages into the network, achieving the lowest latencies and highest message rates for short messages. Several forms of FMA transactions are available:

- FMA Short Messaging (SMSG) and FMA Shared Message Queue (MSGQ) provide a reliable messaging protocol with send/receive semantics that can be used for short point-to-point messages. These facilities are implemented using a specialized RDMA PUT operation with remote notification.
- FMA Distributed Memory (FMA DM) is used to execute RDMA PUT, GET, and Atomic Memory Operations (AMOs), moving user data between local and remote memory.

On the other hand, the BTE hardware offloads the work of moving bulk data from the host processor to the network adapter, also providing RDMA PUT and GET operations. The BTE functionality is intended primarily for large asynchronous data transfers between nodes. More time is required to set up data for a transfer than for FMA, but once initiated, there is no further involvement by the CPU. However, the FMA hardware can give better results than BTE for medium size RDMA operations (2-8 KB), whereas BTE transactions can achieve the best computation-communication overlap because the responsibility of the transaction is completely offloaded to the network adapter, providing an essential component for realizing independent progress of messages. To achieve maximum performance, it is important to properly combine FMA and BTE mechanisms in the `ugnidev` implementation.

The memory allocated by an application must be registered with the network adapter before it can be given to a peer as a destination buffer or used as a source buffer for most uGNI transactions. Thus, in order to directly access a memory region on a remote node, the region must have been previously registered at that node. uGNI provides memory registration interfaces for the applications that allow to specify access permissions and memory ordering requirements. uGNI returns an opaque Memory Handle (MH) structure upon successful invocation of one of the memory registration functions. The MH can then be used for FMA/BTE RDMA transactions and SMSG/MSGQ messaging protocols. The registration and unregistration operations can be very expensive, which is an important performance factor that must be taken into account in the implementation of the `ugnidev` communication protocols.

Finally, uGNI also provides Completion Queues (CQ) management, as a lightweight event notification mechanism for applications. For example, an application may use the CQ to track the progress of local FMA/BTE transactions, or to notify a remote node that data have been delivered to its memory. An application can check for presence of CQ Events (CQE) on a CQ in either polling or blocking mode. A CQE includes application-specific data, information about what type of transaction is associated with the CQE, and whether the transaction associated with the CQE was successfully completed or not. More specific details of the uGNI API can be found in [48].

### *5.2. FastMPJ support for Cray ALPS*

Current Cray systems utilize the Cray Linux Environment (CLE), which is a suite of HPC tools that includes a Linux-based OS designed to run large applications and scale efficiently to a high number of cores. Hence, compute nodes run a lightweight Linux called Compute Node Linux (CNL) which ensures that OS services do not interfere with application scalability. Two separate execution environments for running jobs on the compute nodes of a Cray machine are currently available: Extreme Scalability Mode (ESM) and Cluster Compatibility Mode (CCM).

On the one hand, ESM is the high-performance and native execution environment specifically designed to run large applications at scale, which dedicates compute nodes for each user job and sets up the appropriate parallel environment automatically. This mode is required in order to access the underlying interconnect via the native uGNI API, thus allowing to obtain the highest network performance. However, ESM does not provide the full set of Linux services (e.g., ssh) needed to run standard cluster-based applications, which requires the implementation of specific support for this mode, as will be shown below. On the other hand, the CCM execution environment allows standard applications to run without modifications. Thus, users can request the CNL on compute nodes to be configured with CCM through the use of a special queue at job submission. This mode comes with a standardized communication layer (e.g., TCP/IP) and emulates a Linux-based cluster which provides the services needed to run most cluster-based third-party applications on Cray machines. However, this feature is generally site dependent and may not be available. In addition, it poses important constraints such as that the number of cores that can be used under this mode is usually very limited and there is no support for core specialization. Furthermore, the uGNI API can not be used to access directly the underlying interconnect, which prevents the implementation of `ugnidev`. Therefore, a mandatory prerequisite for this device is the implementation of the ESM mode support in FastMPJ, which basically involves modifying the FastMPJ runtime to work in conjunction with the specific Cray scheduler, as described next.

The Application Level Placement Scheduler (ALPS) [49] is the Cray supported mechanism for placing and launching applications under the ESM mode. More specifically, “aprun” is the user command that must be used to launch a parallel application to a set of compute nodes reserved through ALPS. The FastMPJ support for Cray ALPS mainly consists of two distinct parts. The first one is the “alps-spawner” utility, a small C program ($< 400$ source lines) intended to be launched with the “aprun” command that acts as a bridge between ALPS and FastMPJ. This utility uses the C-based implementation of the Process Management Interface (PMI) [50], which is provided by Cray to interact with ALPS. The PMI library allows to obtain the necessary data from ALPS to properly set up the parallel environment of FastMPJ (e.g., rank of each process in the application). After setting this information via environment variables, “alps-spawner” executes a new JVM using the *execvp()* function. Each JVM represents one of the Java processes of the MPJ application running a specific Java class of the FastMPJ runtime. This Java class, which is the second part of the implemented support, initializes the FastMPJ runtime with the information gathered from the environment and then invokes the main method of the MPJ application using the Java reflection facility. The MPJ application to be executed is one of the input parameters that are accepted by the “alps-spawner” utility, which can be specified using both class and JAR file formats. Once the main method is running, the application will call at some point the `Init` method of the MPJ API in order to initialize the FastMPJ execution environment, and hence the `ugnidev` device initialization takes place.

### *5.3. Initialization*

Since the uGNI interface allows for user-space RDMA communication, there is a hardware protection mechanism to validate all RDMA requests generated by the applications. To utilize this mechanism, uGNI provides applications with a Communication Domain (CDM), which is essentially a software construct that must be attached to a network adapter in order to enable data transfers. Hence, processes must use a previously agreed upon protection tag (ptag) to define and join a CDM. For user-space applications, ALPS supplies a ptag value for each job together with the network adapter that the processes on the local node can use. This information is available in the `ugnidev` device initialization as part of the procedure described in the previous section, in which the required data is first obtained from ALPS/PMI and then is set up by the FastMPJ runtime.

Therefore, `ugnidev` first creates a CDM using the ptag value provided by the FastMPJ runtime, and then attaches the CDM to the available network adapter. All the processes of the job must sign on to the CDM, as any attempt to communicate with a process outside of the CDM generates an error. In addition, each process must supply a 32-bit instance identifier which is unique within the CDM. The rank of the process within the global MPJ communicator (i.e., MPI.COMM_WORLD) is used for this purpose. After this step, `ugnidev` is able to create the CQs and register memory with the CDM. Having completed this sequence of steps, all processes can initiate communications. These operations are all asynchronous, with CQEs being generated when an operation or sequence of operations has been completed.

### *5.4. Communication protocols*

The `ugnidev` device implements all communication operations as non-blocking primitives through native methods in JNI. Blocking communication support is therefore implemented as a non-blocking primitive followed by a *wait*-like call. A message in `ugnidev` consists of a header plus user data (or payload). The message header includes the source identifier, the message size, the message tag and control information (e.g., message type).

As mentioned in Section 5.1, two mechanisms are provided to transfer data using uGNI: FMA and BTE. It is clear that efficiently transferring message data requires to select the best mechanism based on the message size and the overhead associated with each one. Thus, the `ugnidev` device implements two different communication protocols, which are widely extended in message-passing libraries:

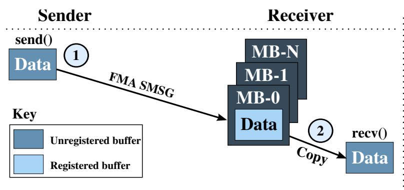
1. Eager protocol: the sending process eagerly sends the entire message to the receiver, on the assumption that the receiver has available storage space. This protocol is used to implement low-latency message-passing communications for short messages (see Section 5.5).
2. Rendezvous protocol: this protocol negotiates, via special control messages, the buffer availability at the receiving side before the message is actually transferred. This protocol is used for transferring long messages, whenever the sender is not sure whether the receiver actually has enough buffer space to hold the entire message (see Section 5.6).

The maximum message size that can be sent using the eager protocol is a configurable runtime option of `ugnidev` that serves as a threshold for switching from one protocol to another. By default, the value of this threshold is set to 16 KB. The benefits of these protocols on the performance of MPJ applications can be significant. On the one hand, the eager protocol reduces the start-up latency, allowing Java applications with intensive short-message communications to increase their scalability. On the other hand, the rendezvous protocol maximizes communication bandwidth, thus reducing the overhead of message buffering and network contention.

### *5.5. Eager protocol*

The eager protocol of `ugnidev` has been implemented using two different paths depending on the message size. The first path uses the FMA SMSG facility, as it provides the highest performance in terms of latency and short-message rates, but comes at the expense of memory usage. Although the FMA MSGQ messaging protocol can be more scalable in terms of memory usage, it was discarded because provides lower performance than SMSG, particularly in terms of short-message rate. Additionally, the maximum message size that can be sent using MSGQ is limited to 128 bytes. In theory, SMSG can be used to deliver messages up to 64 KB, but owing to memory footprint constraints and performance considerations, the practical upper limit is usually lower.

Figure 3 shows the operation of the FMA SMSG path. In this path, each process creates and registers with the network adapter per-process destination buffers called mailboxes (MB in the figure). During a message transfer, the sender directly writes data to its designated mailbox at the receiving side (step 1 in Figure 3). Next, the received data is copied out from the mailbox to the application buffer provided by the user (step 2). SMSG handles the delivery to the remote mailbox and raises both a local and remote CQE on the sending and receiving sides, respectively, upon successful delivery. SMSG transactions are a special class of RDMA PUT operations which require remote buffer memory registration, but not local memory registration, which allows to send the data directly from the application buffer. Additionally, SMSG allows to specify a header separately from the message payload to be sent. Every send request of `ugnidev` has been defined with a small buffer (16 bytes) that contains the message header, which is not shown in the figure for clarity purposes. However, using the SMSG protocol requires a significant amount of registered memory resources which scale linearly with the number of processes in the job. To alleviate this problem, SMSG is only used for communications up to a certain small message size, which is a configurable runtime option. By default, the maximum message size that can be sent using SMSG varies with the job size, with smaller mailboxes being used as the job size increases, in order to decrease the



**Maximum SMSG message size**

| Number of processes | Size (bytes) |
|---|---|
| <= 512 | 4096 |
| <= 1024 | 2048 |
| <= 4096 | 1024 |
| <= 8192 | 512 |
| > 8192 | 256 |

Figure 3. First path of the eager protocol in `ugnidev`

amount of memory used for SMSG mailboxes for larger jobs (see table in Figure 3). Above this size, but below the rendezvous limit (16 KB by default), `ugnidev` switches to the second path that is implemented using both FMA DM and BTE mechanisms, which require the memory addresses and handles of the send and receive buffers. Therefore, this path uses a small shared pool of pre-registered buffers as opposed to the per-process mailboxes of the FMA SMSG path. Each buffer is large enough to contain an entire eager message. These buffers are used in a copy in/out fashion (from/to application buffer), as the overhead of data copies is small for short messages. Since the entire pool is pre-registered during the initialization of the `ugnidev` device, there is no additional registration overhead for each message.

Figure 4 illustrates the operation of the second path. As can be seen in the left part of the figure, the sending process reserves one buffer from the pool and copies the user data in it (step 1). Next, a control message (EAGER_GET_INIT) that includes buffer information is sent to the receiver through the FMA SMSG path (step 2). All control messages of `ugnidev` are small enough to be sent using the SMSG path. Once the receiving side has processed the control message, a buffer is reserved from the pool and, based on the message size, either FMA DM or BTE is used to initiate an RDMA GET of the message data from the sender's memory (step 3). Once the receiving process completes the GET operation, it sends an EAGER_GET_END message to the sender to complete the message transfer (step 4). Upon receipt of this message, the sender marks the message as complete and puts the buffer back to the pool. The receiver will copy the data out from the buffer in the pool to the application buffer when a *recv* operation matches the corresponding *send* (step 5). The choice between using FMA DM or BTE is also configurable via a runtime option. By default, messages up to 2 KB are sent using the FMA DM hardware, while BTE is more suitable for longer transfers, as mentioned in Section 5.1.

However, current Gemini/Aries network adapters impose some buffer size or alignment restrictions for GET operations. Specifically, data transfers using GET require that the data buffer at both sides be a multiple of 4 and 4-byte aligned. When buffer size and alignment restrictions are not met, `ugnidev` uses a PUT-based eager protocol (see right part of Figure 4). If the alignment violation occurs at the sending side, an EAGER_PUT_INIT message is used after step 1 to express the intent to send an eager message using this protocol (step 2). When the receiver has processed this message and has taken a buffer from the pool, it replies to the sender with an EAGER_PUT_RTR message to express that is ready to receive the data (step 3). Upon receiving this message, the sender uses the buffer information that is included in the control message to send the data using an RDMA PUT operation (step 4). If the alignment violation occurs at the receiving side, the receiver sends an EAGER_PUT_RTR message to the sender in response to the initial EAGER_GET_INIT, including information of the receive buffer. Next, the sender proceeds as before and uses RDMA PUT to send the data. Once the PUT operation is complete, the sender sends an EAGER_PUT_END message in both cases to indicate the completion of the message transfer at the receiving side (step 5). Thus, the receiver is ready to copy the data out from the buffer in the pool to the application buffer if the corresponding *recv* operation has been called (step 6).

One clear advantage of the GET-based eager protocol over the PUT-based is that the latter requires one extra control message, which increases the protocol overhead. Additionally, the GET-based protocol can offer better computation-communication overlap since the receiver can progress independently of the sender once the EAGER_GET_INIT message is sent. In order to achieve the lowest latency for short messages, the GET-based protocol is always used when possible, whereas

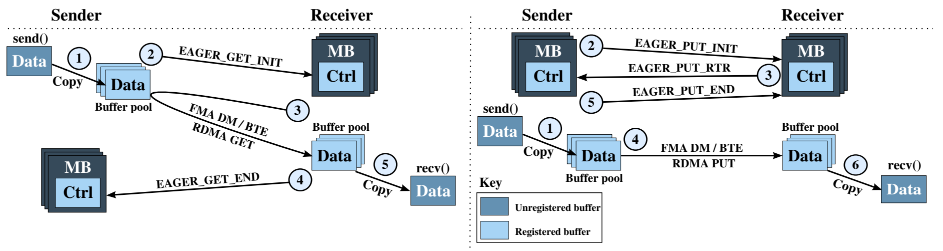


Figure 4. Second path of the eager protocol in `ugnidev`

the PUT-based path only acts as a fallback protocol due to the alignment restrictions of GET operations.

### 5.6. *Rendezvous protocol*

The rendezvous protocol is used for delivering messages exceeding the eager message size threshold. When transferring long messages it is extremely beneficial to avoid extra data copies through a zero-copy protocol. The zero-copy protocol of `ugnidev` first negotiates the buffer availability at the receiving side using control messages. Thus, the application buffers are registered on-the-fly and the buffer addresses are exchanged via control messages. However, the actual data can be transferred by using either RDMA GET or PUT. The efficiency of the RDMA GET operation in Gemini/Aries is sensitive to the alignment of the send and receive buffers, and better performance is obtained when these buffers start at the same relative offset into a cache line. However, RDMA PUT operations are much less sensitive to alignment and thus usually provide higher bandwidth than RDMA GET, especially for the long messages used in the rendezvous protocol. Hence, `ugnidev` employs a GET-based path up to a certain message size in order to benefit from its better computation-communication overlap capabilities, and a PUT-based path for longer transfers. The threshold for switching from using RDMA GET to PUT is also a configurable runtime option, set to 64 KB by default. Additionally, the PUT-based path must also be used when buffer size and alignment restrictions of GET operations are not met, as occurred in the eager protocol.

In the GET-based path (left part of Figure 5), when a sending process is ready to send a long message, it first registers the application buffer (step 1) and then sends a RNDZV_GET_INIT message to the receiving process (step 2). This control message, in addition to expressing the intent to send the message, also provides the receiver with information of the send application buffer for performing an RDMA GET operation. Once the receiver is prepared to receive the message (i.e., the corresponding *recv* operation has been issued and the receive buffer has been registered in step 3), an RDMA GET operation is initiated to access the message data directly from the send buffer (step 4). As occurred in the eager protocol, this GET operation can be done using either FMA DM or BTE, depending on the value of the corresponding threshold. However, with the default settings all the rendezvous transactions will use the BTE hardware, as it is the highest performance option for long messages. Next, a RNDZV_GET_END message is sent to the sending process once the GET operation has finished at the receiving side (step 5). Finally, buffers are unregistered at both sides (step 6).

The PUT-based path (right part of Figure 5) is implemented as a seven-step protocol which starts when the sending process sends a RNDZV_PUT_INIT message to the receiver after the registration of the send buffer (steps 1 and 2). Once the receiver is prepared to receive the message, it registers the application buffer (step 3) and replies with a RNDZV_PUT_RTR message to express that is ready to receive the data (step 4). This reply message contains the information of the receive application buffer to access that memory region. Upon receiving this control message, the sender directly writes data to the target receive buffer by using a PUT operation (step 5). After this operation is finished, a RNDZV_PUT_END message is sent to indicate the completion of the message transfer at the receiving side (step 6), and finally buffers can be unregistered (step 7).

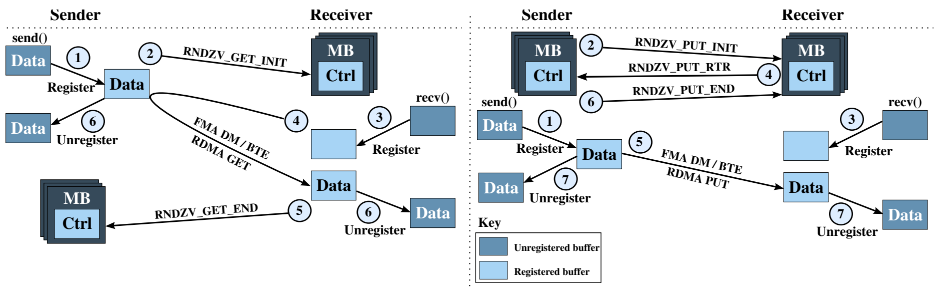


Figure 5. Rendezvous protocol implementation in ugnidev

### *5.7. Registration cache*

When using the rendezvous protocol, application buffers are registered/unregistered on-the-fly causing a performance penalty, especially for very long-message transfers. However, the overhead of the memory registration/unregistration can be hidden or at least reduced by using the pin-down cache technique [51]. The idea is to maintain a cache of registered buffers; thus, when a buffer is first registered it is put into the cache, and when the buffer is unregistered the actual unregister operation is not carried out and the buffer stays in the cache. Hence, the next time the buffer needs to be registered, no operation is performed because it is already in the cache. The effectiveness of this technique heavily depends on how often the application reuses its buffers. If the reuse rate is high, most of the buffer registration and unregistration operations can be avoided. By default, the `ugnidev` device uses the registration cache, which can be disabled via a configurable runtime option.

## 6. EFFICIENT SUPPORT FOR RDMA ADAPTERS BASED ON VERBS: IBVDEV

The `ibvdev` device is a low-level message-passing device for communication on InfiniBand (IB) systems. This device directly implements its communication protocols on top of the Verbs interface through JNI. An initial proof of concept implementation of `ibvdev` was first integrated into the MPJ Express library [10] for internal testing purposes, as it was never part of the official release. Although it was able to provide higher performance than using the IPoIB protocol, the buffering layer in MPJ Express significantly limited its performance and scalability. Next, the `ibvdev` device was reimplemented to conform with the `xxdev` API and adapted for its integration into the FastMPJ middleware in order to improve its performance.

However, the `ibvdev` device still presents two important limitations: (1) it does not include support for the RDMA Communication Manager (RDMA CM), relying instead on TCP sockets to exchange the necessary information for establishing the initial connections between processes during the initialization method. This causes `ibvdev` to only work on IB adapters, thus not supporting the remaining RDMA-compliant adapters based on the Verbs interface: iWARP and RoCE; and (2) it does not take advantage of the inline feature that is provided by some RDMA adapters to improve the latency of short messages. Currently, `ibvdev` has overcome these constraints by establishing initial connections through RDMA CM and implementing a more efficient eager protocol that uses the inline feature. The new connection setup using RDMA CM allows `ibvdev` to support iWARP and RoCE networks while avoiding any TCP processing overhead during the initialization method. These new features in `ibvdev` will be discussed in the next sections.

### *6.1. Eager protocol optimization*

The `ibvdev` device implements both the eager and rendezvous protocols relying on the Reliable Connection (RC) transport service defined in Verbs, which provides reliability, delivery order and

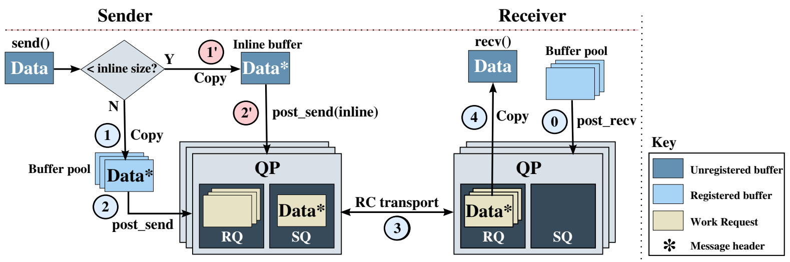


Figure 6. Eager protocol implementation in `ibvdev`

data loss and error detection. The eager protocol of `ibvdev` is illustrated in Figure 6. In the original implementation, the buffer registration/unregistration overhead is avoided by using a shared pool of pre-registered, fixed size buffers for communication. For sending an eager message, the user data along with the message header are first copied to one of the available buffers from the pool (step 1 of the figure). Next, it is sent out from this buffer to the Send Queue (SQ) of the corresponding Queue Pair (QP). This is done by using the *ibv_post_send()* function (step 2), which posts a Work Request (WR) to the SQ. At the receiving side, a number of buffers from the pool are pre-posted in the Receive Queue (RQ) using *ibv_post_recv()* (step 0). This function, which posts a WR to the RQ, is the receiving counterpart of *ibv_post_send()*. Once the message is received through the network (step 3), the message payload is copied out to the user destination buffer (step 4) and the receive buffer is returned back to the pool.

However, this implementation does not take advantage of sending data as inline, a feature that is supported by some modern RDMA adapters. Using this feature, the memory buffer that holds the message is placed inline in the WR posted to the SQ. This means that the CPU (i.e., not the RDMA adapter) will read the data from the buffer. Therefore, the data is transferred to the adapter at the same time that *ibv_post_send()* transfers the WR. The main benefit is that sending short messages as inline provides lower latency since it eliminates the need of the RDMA device to perform an extra read over the PCIe bus in order to read the message payload. In addition, the memory buffer used for communication at the sending side does not have to be registered with the RDMA adapter.

The inline feature is an implementation extension not defined in the RDMA specification. Hence, there is not any defined *verb* that specifies the maximum message size that can be sent inline in the SQ of a QP. In some RDMA adapters with this feature, creating a QP will set the value of the *max_inline_data* attribute to the message size that can be sent as inline (usually less than 1 KB). In other adapters, the message size to be sent inline must be explicitly specified before the creation of a QP. In the latter case, the maximum value supported by the RDMA adapter is calculated during the initialization method of `ibvdev` following an iterative approach, which first creates a dummy QP specifying a high initial value, and then continues to decrease if the QP creation fails. When the QP creation is successful, the inline size of the dummy QP is used to create all the QPs needed for establishing the connections between processes.

In the original implementation of `ibvdev`, when a WR is posted to the SQ, the buffer that holds the message can not be modified since it is not possible to know when the RDMA adapter will stop reading from it. That is to say, the WR is considered outstanding until a completion event is raised, which means that the buffer can now be reused. However, when using inline data the buffer can be reused immediately after *ibv_post_send()* is finished, since the data has been already transferred to the RDMA adapter. This allows to have a single dedicated buffer to send inline data to all processes. Therefore, the pool of pre-registered buffers can be bypassed when using the new implemented path: if the message is small enough to be sent inline, the message header and payload are now copied to a dedicated buffer (step 1’ in Figure 6) and then sent out from this buffer to the SQ using *ibv_post_send()* with the appropriate flags (step 2’). As mentioned before, this path reduces the latency of short messages, between 15-20% according to our tests, although the actual latency reduction heavily depends on the underlying RDMA adapter and CPU used. Additionally, it allows more buffers to be available to send messages through the original path if the message size is above the inline value, but below the rendezvous limit. Furthermore, all control messages of the rendezvous protocol can take advantage of this optimization, as they are small enough to be sent inline. This also contributes to increase the number of outstanding WRs that can be posted to the SQ at a time, which improves the overall efficiency of the RDMA adapter while memory consumption remains almost the same (only one additional buffer is needed). Note that this optimized path is only relevant at the sending side, as the receiver is not aware of the fact that a WR is sent inline.

### *6.2. RDMA CM-based connection setup*

The basic communication in `ibvdev` is achieved over connected QPs using the RC transport service. In the initialization method, an RC-based QP connection is established between every two processes (see Figure 6). To enable data transfers, each QP needs to be set up and must be transitioned through an incremental sequence of states. In order to transition into the final connected state, some information from the remote process is required: (1) the number of the remote QP to connect with (this value is returned at QP creation); and (2) the Local IDentifier (LID) of the remote process, which is a unique 16-bit address assigned to end nodes by the subnet manager. This information needs to be exchanged through some out-of-band mechanism. As a first step, the original initialization method of `ibvdev` uses sockets to set up a TCP connection between every two processes. Second, the necessary information is exchanged through TCP sockets. Third, the QPs are transitioned and connected to each other. Finally, the TCP connection is closed. The described connection setup works perfectly on IB adapters, which was the main goal of the original implementation of the device. However, it poses an important drawback: the iWARP protocol requires RDMA CM to establish connections, which prevents `ibvdev` from working on iWARP adapters. Another drawback is the additional TCP connection that is established in advance to initialize the device, which can add a noticeable delay and TCP processing overhead on IB adapters when using a high number of processes. These issues have been overcome by implementing an alternative connection method using RDMA CM.

RDMA CM is an abstraction layer for connection management defined by the OpenFabrics Alliance (OFA) [20], designed to establish connections between the QPs of a pair of processes. In fact, it is an event-driven connection manager based on a high-level IP address/port number abstraction that can set up connections over the multiple RDMA networks supported by Verbs, but is only mandatory for iWARP. The main responsibilities of RDMA CM include exchanging necessary connection information and transitioning the QPs through their states into the connected state, thus avoiding the additional TCP connection of `ibvdev`. It is to be noted that RDMA CM sets up the connections in a traditional client-server mechanism. Therefore, the API is based on sockets, but adapted for QP-based semantics: communication is over a specific RDMA device, and data transfers are message-based. RDMA CM provides only the communication management functionality (i.e., connection setup and teardown), and works in conjunction with the Verbs interface for data transfers.

The initialization method of `ibvdev` has been modified to use the RDMA CM manager in order to automate and simplify the connection setup. As mentioned before, RDMA CM uses the traditional TCP style, client-server mechanism to set up connections. Due to this, all the process pairs need to be separated into client-server pairs before any setting up of connections. For every pair of processes, the process with the lower rank takes the role of server (passive side or responder), and the process with the higher rank takes the role of client (active side or initiator). The main steps to complete the connection setup using RDMA CM are shown in Figure 7, as follows. (1) Each process identifies the port and IP address based on the RDMA adapter to use. This is accomplished via the information provided by the FastMPJ runtime to the `ibvdev` device. (2) Both sides allocate a communication identifier via *rdma_create_id()*, which is conceptually equivalent to a socket for RDMA communication. (3) The server must bind the RDMA identifier to a source address and listen for incoming connection requests. In the case of a client, it first resolves the server address and then allocates a new RDMA connection (i.e., a QP) via the *rdma_resolve_addr()* and *rdma_create_qp()* functions, respectively. (4) The client sends a connection request to the server using *rdma_connect()* after having resolved the destination route. (5) When the request is received

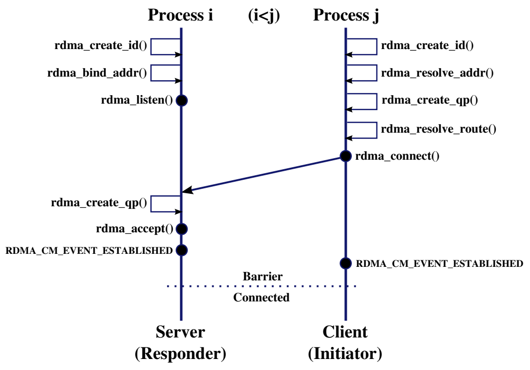


Figure 7. RDMA CM-based connection setup in `ibvdev`

at the server side, the responder then allocates a new RDMA connection and uses *rdma_accept()* to confirm the connection to the client. (6) The connections are established internally by RDMA CM, exchanging the necessary information and transitioning the corresponding QPs through their sequence of states. (7) The final transition into the connected state is detected via an event at both sides, which completes the establishment of the RDMA connection. (8) At this point, the processes synchronize with a barrier to make sure that all the peer processes are ready for communication. These steps are repeated for the setup of each of the QPs between a pair of processes. The overall procedure can be done concurrently due to the event-driven nature of the connection manager.

As mentioned before, the RDMA CM-based connection setup allows `ibvdev` to provide support for iWARP adapters while leveraging the existing communication protocols of the device. Additionally, as RDMA CM is also valid for RoCE adapters, `ibvdev` now supports all RDMA-compliant adapters based on Verbs: IB, iWARP and RoCE. The original TCP-based connection setup is still interesting to be supported as it serves as a fallback option in case of any issue with RDMA CM or even if it is not available in the system. Although the TCP-based approach can not work on iWARP since RDMA CM is mandatory for this network, its support for RoCE has also been implemented, as described next.

The OFA specifies that Verbs applications which run over IB/iWARP should work on RoCE as long as the Global Routing Header (GRH) information is provisioned when creating Address Handles (AH). The GRH is required for routing between subnets and is optional within IB/iWARP subnets. However, RoCE encapsulates the IB transport and GRH headers in Ethernet packets bearing a dedicated *ether* type. In this case, the GRH is used for routing inside the subnet and therefore is mandatory. The GRH information can be provisioned in the AH of a QP when using the RC transport. The AH describes the path to the remote QP and is needed to make the transition from the initial state to the ready-to-receive state. This is the reason why using RDMA CM works seamlessly on RoCE without any change (QPs are transitioned and set up automatically). However, using the original TCP-based method the GRH information must be specified manually using the Global IDentifier (GID) of the remote process, which is a unique 128-bit address used to identify a port on a network adapter that is assigned by the subnet manager. Hence, this method has been modified as follows. First, each process has to query its GID via *ibv_query_gid()*. Next, this value needs to be exchanged with the remote process along with the previously required information (LID and QP number). Once the TCP communication phase has been completed, the required GRH information for RoCE can be provisioned in the AH of each QP using the remote GID value. Finally, each QP can be transitioned through the required sequence of states as occurred in the original TCP-based implementation.

To sum up, the `ibvdev` device currently provides full support for IB, iWARP and RoCE through the new RDMA CM-based connection setup and, as a fallback option, IB and RoCE are also supported using the TCP-based approach.

## 7. A SPECIFIC DEVICE FOR MELLANOX RDMA ADAPTERS: `MXMDEV`

Another contribution of this paper is the introduction of the `mxmdev` device, which provides native support for the networking infrastructure provided by Mellanox RDMA hardware over the MellanoX Messaging (MXM) accelerator. MXM is a user-space messaging library that implements intra-node shared memory and inter-node communication protocols, which are completely transparent to the application. It includes a variety of enhancements that take advantage of Mellanox IB/RoCE adapters including proper management of resources and memory structures, efficient memory registration, handling of transport services and a tag matching mechanism at the receiving side. Hence, many of the low-level network features are built-in in MXM, which allows developers to work at a higher level and the main effort to be spent on the overall application development.

Therefore, the most important benefit of MXM is that it provides the developer with a higher level API than Verbs, based on a set of communication primitives with messaging semantics that eases the development of applications on top of the Mellanox RDMA hardware. However, MXM is not primarily intended for use by end-user applications. Instead, portable communication middleware (e.g., message-passing libraries) usually provide specific support for MXM, which allows the user to benefit from a higher level of abstraction without source code modifications. This fact has motivated the implementation of `mxmdev`, a new `xxdev` device layered on top of the MXM library. FastMPJ with `mxmdev` provides the programmer with all the high-level features of the MPJ layer (e.g., collective communications, virtual topologies, intra- and inter-communicators) while taking advantage of the infrastructure provided by FastMPJ, such as the runtime system. Additionally, it frees Java developers from the implementation of JNI calls, which is usually a cumbersome and time consuming development task. Hence, the `mxmdev` device allows the developer to benefit from the MPJ programmability, which greatly enhances productivity without trading off much performance.

### *7.1. Connection setup*

The MXM library is initialized using the *mxm_init()* method. Next, the connection setup must be carried out in order to enable communications. In MXM, messages are exchanged between endpoints, which are software representations of the Mellanox IB/RoCE adapters. At present, MXM does not include any communication manager to ease the connection setup. Thereby, in order to establish the initial connections between endpoints, the `mxmdev` device has to rely on an out-of-band mechanism to distribute the endpoint addresses between all the processes. Hence, each process first creates and sets up an endpoint using the *mxm_ep_create()* function. After initializing endpoints, a Matched Queue (MQ) interface is created via *mxm_mq_create()*. Basically, an MQ is a specific context of sending and receiving messages which maintains ordering between requests. It exposes a simplified messaging interface that resembles an MPI communicator, but supporting only basic point-to-point communications. Next, the endpoint addresses are exchanged between all processes relying on TCP sockets, selected as the ubiquitous out-of-band mechanism. Finally, the *mxm_ep_connect()* function must be used to establish the endpoint connections with the information gathered from the TCP communication phase, thus enabling data transfers.

### *7.2. Basic communication operation*

The MXM library provides a C-based API which includes a small set of point-to-point communication primitives similar to those needed to implement the `xxdev` interface (see Section 4). Thus, `mxmdev` acts as a thin wrapper over the MXM library, so that the implementation of a method in `xxdev` generally delegates directly in a native method that performs the requested operation in MXM through JNI. Therefore, `mxmdev` deals with the marshaling and communication of Java objects, the JNI transfers and the handling of MXM parameters, by implementing a series of three steps: (1) get the associated parameters of the Java objects that are required for calling the corresponding function in MXM; (2) call the MXM function; and (3) save the results in the appropriate attributes of the Java objects involved in the communication. As a general rule, the caching of object references has been extensively used in the implementation of the JNI layer, thus minimizing the overhead associated with the JNI calls.

Every message operation in MXM, either sending or receiving, starts with a non-blocking communication request (e.g., *mxm_req_send()*). This request is queued by MXM, returning the control to `mxmdev`. Next, the `mxmdev` device is responsible for checking the successful completion of the communication operation using one of the supported mechanisms in MXM (e.g., *mxm_wait(), mxm_req_test()*). The MXM tag matching mechanism at the receiving side is based on a 32-bit value (*mxm_tag_t)*, which must be specified by both communication peers in order to deliver incoming messages to the right receive requests. The tag value specified by the programmer at the corresponding MPJ-level method (e.g., *MPI.COMM_WORLD.Send()*) is used for this purpose. Hence, incoming MXM messages are stored according to their MPJ tags to pre-posted receive buffers. In this case note that, unlike the `ugnidev` and `ibvdev` devices, the underlying communication protocols are implemented internally by MXM. Currently, MXM includes both intra-node (via shared memory) and inter-node communication protocols, which allows MPJ applications to take full advantage of hybrid shared/distributed memory architectures, widely extended nowadays.

## 8. PERFORMANCE EVALUATION

This section presents a performance evaluation of the FastMPJ communication devices presented in this paper: `ugnidev`, `ibvdev` and `mxmdev`. The experimental results have been obtained at the MPJ API level in order to analyze the impact of their use on the overall middleware performance. Hence, FastMPJ (labeled as FMPJ in the graphs) has been evaluated comparatively with representative MPJ middleware: Open MPI Java [45] and MPJ Express [46]. First, this section includes a micro-benchmarking of point-to-point communication primitives on several RDMA networks (Section 8.1). Next, the impact of the communication devices on the overall application performance of representative parallel Java codes is analyzed (Section 8.2).

### *8.1. Micro-benchmarking of point-to-point primitives*

The goal of this micro-benchmarking is the comparative performance evaluation of Java point-to-point communications between two nodes across different RDMA networks (i.e., inter-node latency and bandwidth). This evaluation has been carried out using our own MPJ micro-benchmark suite. This suite is the MPJ counterpart of the Intel MPI Benchmarks (IMB) [52], widely used for MPI libraries, and tries to adhere to its measurement methodology. Here, the metric shown is the half of the round-trip time of a ping-pong test for short messages (up to 1 KB), and the corresponding bandwidth for longer messages (up to 16 MB). In order to obtain optimized JIT compiled bytecode results, 20,000 warm-up iterations have been executed before the actual measurements. The results shown are the average of 10,000 iterations, although the observed standard deviations were not significant. The transferred data are byte arrays, avoiding the Java serialization overhead that would distort the analysis of the results.

As most Java communication middleware (e.g., RMI) is based on sockets, the standard Java socket implementation of the JVM has also been evaluated for comparison purposes (labeled as JVM Sockets in the graphs). The NetPIPE benchmark suite [53] has been selected since its Java socket implementation performs a ping-pong test similar to the one used for the MPJ point-to-point benchmarking, and so the results are directly comparable.

*8.1.1. Experimental configuration.* Two different systems have been used in the evaluation of point-to-point primitives. The first testbed consists of two nodes, each of them with one Intel Xeon E5-2643 quad-core Sandy Bridge-EP processor at 3.3 GHz and 32 GB of memory. These nodes have been used to evaluate three different RDMA networks: IB (Mellanox MT27500 4x FDR, 56 Gbps), RoCE (Mellanox MT27500, 40 Gbps) and iWARP (Intel NetEffect NE020, 10 Gbps). Hence, this testbed allows the evaluation of the `ibvdev` device on IB, RoCE and iWARP, while `mxmdev` can be assessed on IB and RoCE. Regarding software configuration, the OS is Linux CentOS 6.4 with kernel 2.6.32-358 and the JVM is OpenJDK 1.7.0_25. Finally, the native communication layers are OFED driver 3.5-2 and MXM version 1.5.

The second testbed is the Hermit supercomputer installed at the High Performance Computing Center Sttutgart (HLRS), ranked #39 in the November 2013 TOP500 list [54]. This system is a petaflop Cray XE6 supercomputer with 113,664 cores and 126 TB of memory. More specifically, Hermit consists of 3552 compute nodes, each of them with 2 AMD Opteron 6276 16-core Interlagos processors at 2.3 GHz and 32/64 GB of memory. The nodes are connected via the custom Gemini interconnect [18], which allows the evaluation of the `ugnidev` device. This network has a 3D torus topology built from Gemini Application-Specific Integrated Circuits (ASICs) that provide 2 network adapters and a 48-port router. Hence, each ASIC connects two nodes to the network. In the ping-pong test, two adjacent nodes (i.e., connected to the same ASIC) have been used in order to report the lowest latencies and highest bandwidths for inter-node communications (i.e., results are shown using the minimum hop network count). Regarding software settings, this system runs CLE version 4.1.UP01 with kernel 2.6.32.59, which is an OS based on SUSE Linux Enterprise Server. The JVM is Oracle JDK 1.7.0_45 and the uGNI library version is 4.0-1. This supercomputer is also one of the systems selected for the analysis of performance scalability of parallel Java applications (shown in Section 8.2).

Regarding the Java middleware under comparison, MPJ Express has been evaluated using the NIO-socket device (`niodev`) over the corresponding TCP/IP transport layer on each testbed (e.g., IPoIB, IPoGIF). Furthermore, its results on the Cray supercomputer have been obtained under the CCM execution environment, as MPJ Express lacks native support for Cray ALPS in order to run under the ESM mode (see Section 5.2). Open MPI Java has been configured with the `openib` BTL, which is implemented over Verbs on IB, RoCE and iWARP networks. The results using the `mxm` MTL, implemented over MXM, are not shown for clarity purposes, as we have checked that the `openib` BTL generally obtains better performance than the `mxm` MTL. Unlike MPJ Express, Open MPI Java can benefit from its specific support for the Cray machine, which allows to use the ESM mode and the uGNI library via the `ugni` BTL. Finally, the evaluation of Java sockets has also been assessed at the corresponding TCP/IP layer, as occurred with MPJ Express, due to the lack of RDMA network support in the JVM.

*8.1.2. Analysis of the results.* Figure 8 shows point-to-point latencies and bandwidths on IB, RoCE, iWARP and Gemini networks. The latency graphs (at the left) serve to compare short-message performance (up to 1 KB), whereas the bandwidth graphs (at the right) show long-message performance (up to 16 MB).

The performance results on IB reveal that both FastMPJ devices (i.e., `ibvdev` and `mxmdev`) obtain the lowest start-up latencies, around 1 $\mu$s, as compared to 2.7 $\mu$s for Open MPI Java, showing an overhead reduction of approximately 62%. JVM sockets show significantly poorer latencies than FastMPJ and Open MPI Java, around 10 $\mu$s, as they rely on the IPoIB layer. MPJ Express presents the poorest performance (18 $\mu$s), incurring a significant overhead of 8 $\mu$s over the JVM performance. Regarding bandwidth, the FastMPJ `ibvdev` device obtains the best performance with up to 47 Gbps, whereas the maximum bandwidth for `mxmdev` is around 43 Gbps. FastMPJ clearly outperforms the other middleware for long messages, achieving up to 5 times higher performance than MPJ Express (8.8 Gbps), which suffers significantly from the lack of specialized support on IB and from the high overhead of its buffering layer. In fact, JVM sockets show approximately twice the bandwidth (16.1 Gbps) obtained by MPJ Express, whereas Open MPI Java incurs a noticeable wrapping overhead from 256 KB, showing a long-message bandwidth similar to JVM sockets.

The analysis of the performance results on RoCE shows a very similar pattern. On the one hand, FastMPJ devices obtain slightly higher latencies than on IB, around 1.15 $\mu$s, outperforming Open MPI Java (2.8 $\mu$s), while the sockets-based approaches (i.e., JVM and MPJ Express) do not show any significant difference compared to IB results. On the other hand, the maximum bandwidths obtained by `ibvdev` and `mxmdev` are 36.6 and 35.7 Gbps, respectively, up to 2 times higher performance than JVM sockets (19.1 Gbps) and quite close to the 40 Gbps limit for this networking

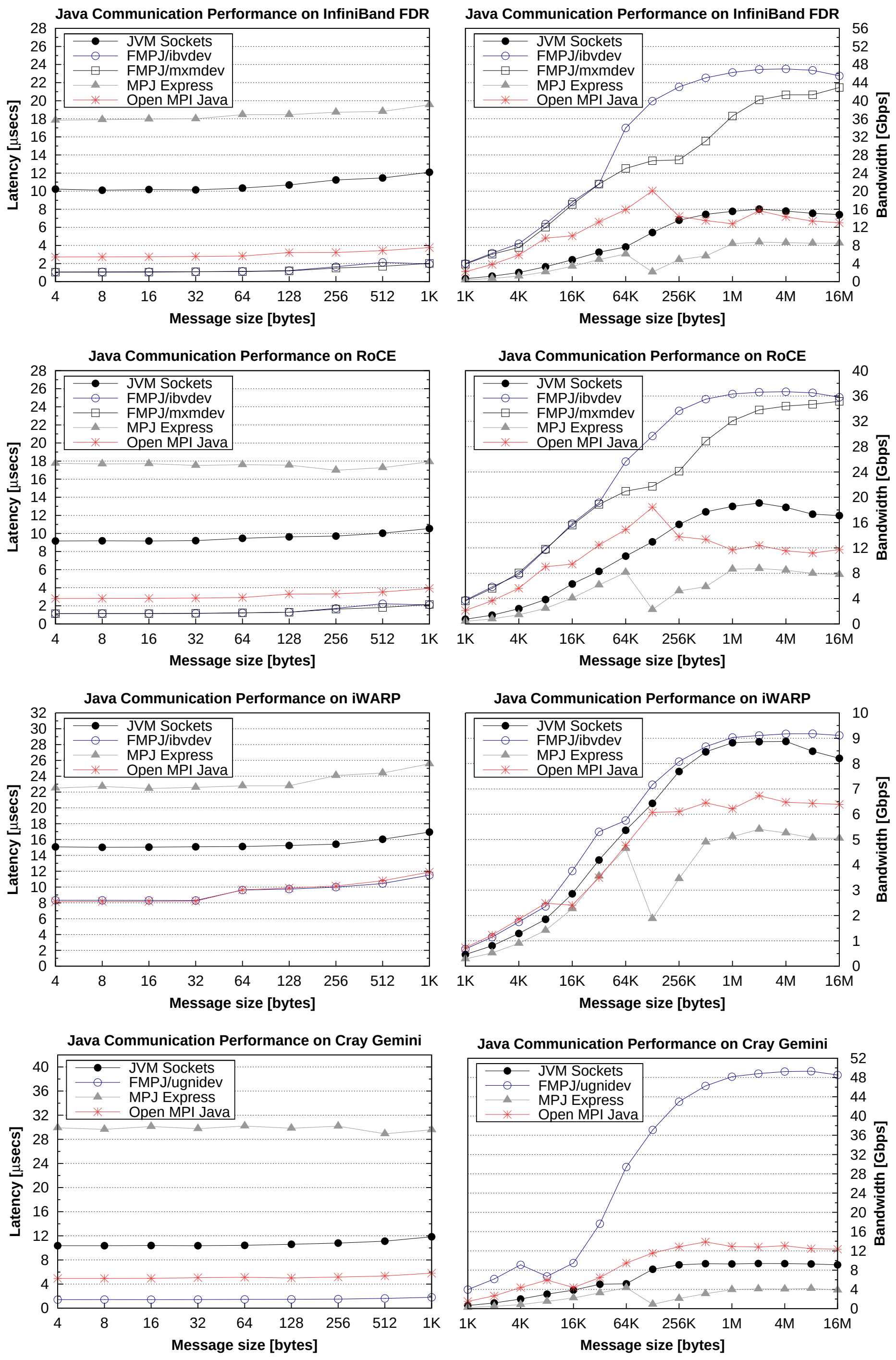


Figure 8. Performance of Java point-to-point communications on RDMA-enabled networks

technology. In fact, these results confirm that RoCE is able to provide IB-like latencies on the Ethernet infrastructure, while the maximum bandwidth of the RoCE adapters is increasing and approaching IB. Both Open MPI Java and MPJ Express incur once again a high wrapping overhead and buffering penalty, respectively, which penalizes especially long-message performance.

The start-up latencies obtained on iWARP are relatively high, at least compared with those on IB and RoCE. This fact suggests that the TCP/IP processing overhead seems to be the main performance bottleneck for short-message performance, even though it is offloaded onto the iWARP hardware. In fact, FastMPJ using `ibvdev` and Open MPI Java achieve similar latencies, around 8 $\mu$s, while the sockets-based approaches obtain 15 and 24 $\mu$s for JVM and MPJ Express, respectively. Regarding bandwidths, FastMPJ presents the best performance, achieving up to 9.2 Gbps, although followed closely by JVM sockets (8.8 Gbps). Here, the iWARP network, with a theoretical maximum bandwidth of 10 Gbps, turns out to be the main performance bottleneck for long-message performance on this scenario.

Finally, the performance results on the Cray Gemini interconnect show that the start-up latencies of the `ugnidev` device are around 1.45 $\mu$s, followed by Open MPI Java (4.9 $\mu$s) and JVM sockets (10.4 $\mu$s). This means that FastMPJ is able to provide a reduction of the communication overhead for short messages of up to 70% compared to Open MPI Java. MPJ Express, which obtains 30.5 $\mu$s, imposes an overhead on the start-up latency of around 20 $\mu$s over the JVM performance, a value even higher than on the previous scenarios. It can be observed that the performance increase of `ugnidev` for long-message bandwidth is up to 275% with respect to the second-best option, a result obtained for 4 MB messages, where FastMPJ achieves 49.2 Gbps and Open MPI Java is limited to around 13.1 Gbps. Furthermore, both sockets-based approaches present the poorest bandwidth results, up to 9.3 and 4.4 Gbps for JVM sockets and MPJ Express, respectively. Noted that, as mentioned before, both approaches must resort to the CCM mode in order to be executed on the Cray machine.

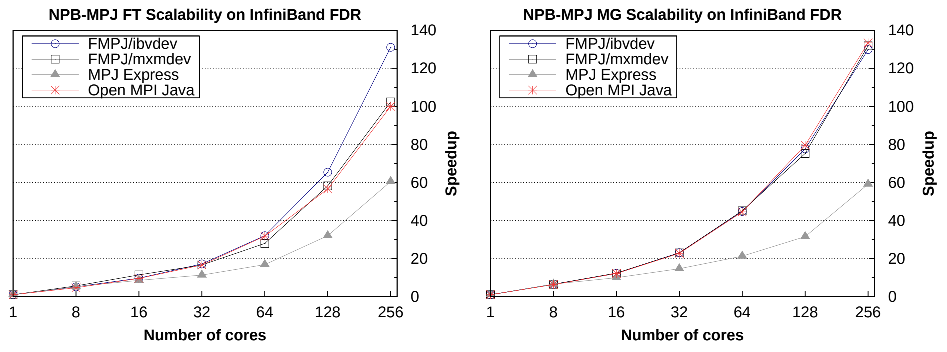
The micro-benchmarking of point-to-point communications has shown significant performance improvements using the communication devices presented in this paper. However, their usefulness depends on their impact on the overall application performance, as will be analyzed next.

## *8.2. Performance analysis of parallel Java codes*

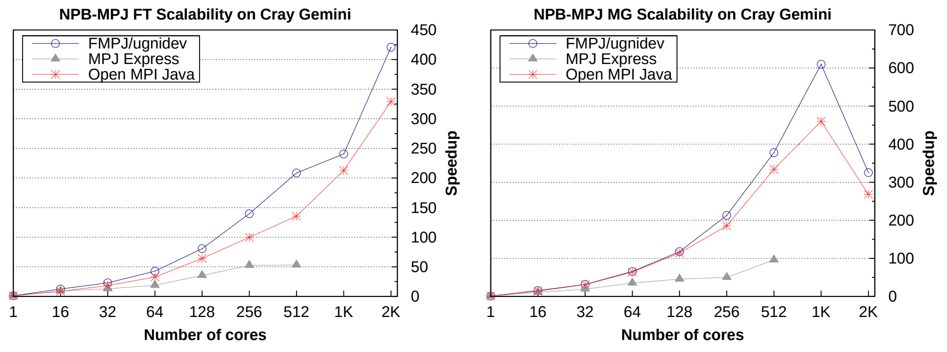
This section presents the analysis of the performance scalability of representative Java HPC kernels and applications. On the one hand, two kernels from the NAS Parallel Benchmarks (NPB) [55] implementation for MPJ (NPB-MPJ) [56] have been evaluated (Section 8.2.2): FT (Fourier Transform), and MG (Multi-Grid). On the other hand, the scalability of an MPJ version of the Finite-Difference Time-Domain (FDTD) method [57], which is a widely used numerical technique in computational electromagnetics, has been analyzed at the application level (Section 8.2.3). The selection of these parallel codes has been motivated by their high communication intensiveness, which allows the assessment of the impact of the developed communication devices on their scalability.

*8.2.1. Experimental configuration.* The experimental results have been conducted on two systems. The first testbed is Pluton, a 16-node multi-core cluster. Each node has 2 Intel Xeon E5-2660 octa-core Sandy Bridge-EP processors at 2.2 GHz (hence 16 cores per node) and 64 GB of memory. The performance results have been obtained using 16 processes per node (i.e., 256 cores in total), since we have checked that the use of 32 processes per node when resorting to Hyperthreading does not provide any performance benefit for the evaluated codes. The nodes of Pluton are interconnected via IB (Mellanox MT27500 4x FDR, 56 Gbps), which allows the assessment of the `ibvdev` and `mxmdev` devices. Regarding software configuration, the OS is Linux CentOS 6.4 with kernel 2.6.32-358 and the JVM is OpenJDK 1.7.0_25. Finally, the native communication layers are OFED driver 3.5-1 and MXM version 2.0.

The second testbed is Hermit, the Cray XE6 supercomputer described in Section 8.1.1. The AMD processor of Hermit has a quite complex architecture that provides up to 16 integer cores and 8 256-bit Floating Point Units (FPUs) per chip. A dual-processor node can provide up to 32 integer cores that access the half of the FPU executing 128-bit instructions, or 16 integer cores accessing the entire



(a) NPB-MPJ class C results on the multi-core cluster (Pluton)



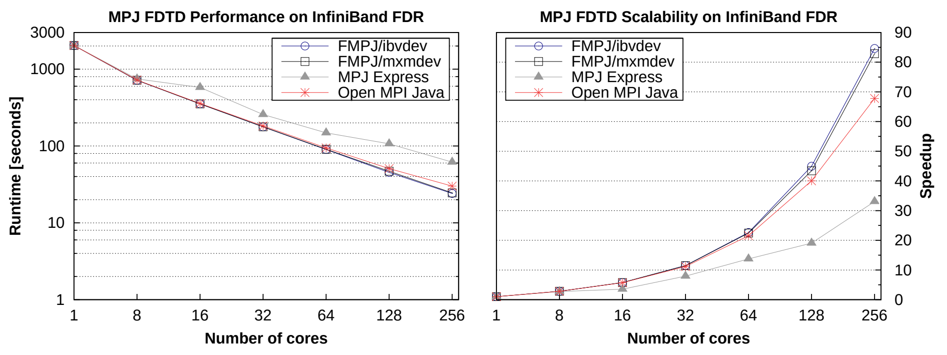
(b) NPB-MPJ class C results on the Cray XE6 supercomputer (Hermit)

Figure 9. Scalability of Java HPC kernels

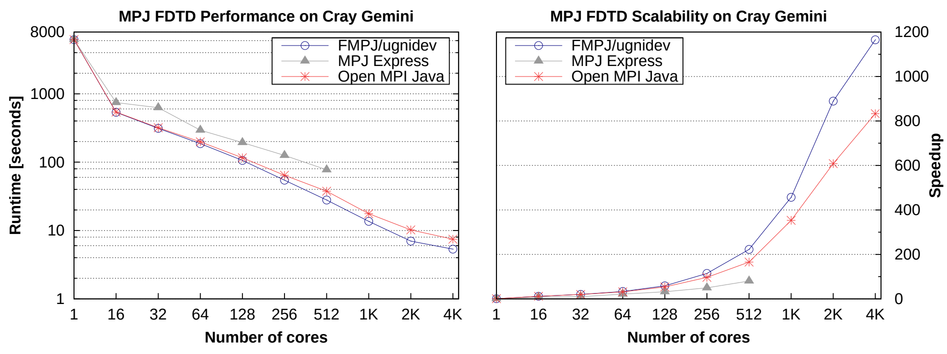
FPU with 256-bit instructions. This is because each FPU is shared between the two integer cores of a Bulldozer module, which is the building block of this architecture. Therefore, the results are shown using 16 processes per node (i.e., one process per Bulldozer module) in order to maximize the FPU performance on this system. We have experimentally checked that this configuration obtains the best performance for the evaluated codes, which carry out extensive double-precision floating-point operations, and thus the results using 32 processes per node are not shown for clarity purposes. Moreover, the reported results for a given application and core count were obtained within a single resource allocation to minimize timing differences due to node placement.

Regarding the Java middleware, the MPJ Express results have been obtained on both systems using the `hybrid` device, which allows to combine simultaneously the NIO-socket device (`niodev`) for inter-node communications and the multithreaded device (`smpdev`) for intra-node communications. Regarding Open MPI Java, it has been configured on Pluton with the `openib` BTL for inter-node communications and the `sm` BTL for intra-node communications, whereas the `ugni` and `vader` BTLs have been used on Hermit for inter- and intra-node communications, respectively.

*8.2.2. MPJ Kernel Performance Analysis.* Figure 9 presents the performance results in terms of speedups for the NPB-MPJ FT and MG kernels (with class C workload) on Pluton and Hermit using up to 256 and 2048 cores, respectively. Regarding the results on Pluton (see Figure 9(a)), FastMPJ using `ibvdev` shows the highest speedups for the FT kernel from 32 cores on, obtaining a speedup above 130 on 256 cores and outperforming Open MPI Java up to 30%. Regarding FastMPJ with the `mxmdev` device, it achieves very similar speedups to Open MPI Java: 102 and 99 on 256 cores, respectively. MPJ Express presents poor scalability from 32 cores, as a direct consequence of



(a) MPJ FDTD results on the multi-core cluster (Pluton)



(b) MPJ FDTD results on the Cray XE6 supercomputer (Hermit)

Figure 10. Scalability of the FDTD parallel Java application

its current limitations (e.g., the use of sockets and IPoIB in its communication layer, among others). The reported speedups for MG are quite similar for FastMPJ and Open MPI Java, around 133 on 256 cores. This fact suggests that the memory access bandwidth turns out to be the main performance bottleneck for this kernel on Pluton. MPJ Express shows again the poorest speedups, below 60 on 256 cores (around 2 times lower than FastMPJ).

The analysis of the results on Hermit (see Figure 9(b)) shows that the use of the `ugnidev` device allows FastMPJ to become the best performer for both kernels. Regarding FT results, it can be observed that FastMPJ outperforms Open MPI Java especially from 256 cores on, obtaining a speedup increase of up to 28% on 2048 cores. The peak speedup values for the MG kernel are achieved on 1024 cores both for FastMPJ and Open MPI Java, where `ugnidev` provides FastMPJ with a speedup increase of 32%. As can be observed, the scalability of the MG kernel degrades from 2048 cores. The results for MPJ Express are shown only up to 512 cores due to the limited number of cores that can be used under the CCM mode on Hermit. Nevertheless, this limitation is irrelevant due to the low scalability obtained by MPJ Express, around 4 times lower than FastMPJ on 512 cores.

*8.2.3. Performance Analysis of the MPJ FDTD application.* Figure 10 shows the runtime and speedup results for the MPJ FDTD application on Pluton (Figure 10(a)) and Hermit (Figure 10(b)) using up to 256 and 4096 cores, respectively. This application simulates a Ricker wavelet propagating in free space surrounded by perfectly electrically conducting walls that reflect impinging electromagnetic waves. The parallel code is based on a domain decomposition approach that divides the workload equally among the cores, requiring frequent data transfers between processes (mainly point-to-point communications) during the entire simulation. The results are shown for a simulation of 2,500 time steps using a 16384x8192 grid.

According to the performance results, FastMPJ achieves the highest speedups for the FDTD code, as shown in the right graphs, especially when using a high number of cores. In particular, the performance improvements compared to Open MPI Java when using the maximum number of cores are 24% on Pluton and 40% on Hermit. Note also that the `ibvdev` and `mxmdev` devices achieve very similar scalability results on Pluton. Compared to MPJ Express, FastMPJ increases speedup up to 151% on Pluton (for 256 cores) and up to 176% on Hermit (for 512 cores). Therefore, these results reinforce that the use of the developed low-level communication devices can improve transparently the scalability of parallel Java applications.

## 9. CONCLUSIONS

RDMA is a well-known mechanism that enables zero-copy and kernel-bypass features, providing low-latency and high-bandwidth communications with low CPU utilization. However, RDMA-enabled networks also pose some important challenges (e.g., high programming effort) that require appropriate middleware support for the development of scalable parallel applications with underlying hardware transparency. In order to take full advantage of the abundant hardware resources due to the current trend of increasing the number of cores, applications have to resort to efficient middleware. Nevertheless, current Java communication middleware is usually based on protocols with high communication overhead (e.g., sockets-based protocols) which do not provide scalable communications on RDMA networks.

This paper has described in detail the implementation of several low-latency communication devices, which have been successfully integrated in our Java message-passing implementation, FastMPJ. These devices have considered several communication protocols in order to provide scalable support for RDMA networks, enabling 1-$\mu$s start-up latencies and up to 49 Gbps bandwidth for Java message-passing applications, thanks to the efficient exploitation of the underlying RDMA hardware. In order to evaluate their benefits, the performance of these devices has been analyzed comparatively with other Java communication middleware on representative RDMA networks (IB, RoCE, iWARP, Cray Gemini) and parallel systems (a multi-core InfiniBand cluster and a TOP500 Cray supercomputer). The analysis of the results has demonstrated experimental evidence of significant performance improvements when using the developed devices in FastMPJ. In fact, the scalability of parallel Java codes can benefit transparently from this efficient support on RDMA networks, reducing the latency by up to 70% and 90%, and increasing the bandwidth by up to 3.6 and 5.1 times compared to Open MPI Java and JVM sockets, respectively. Therefore, the reported advances in Java communication efficiency can contribute to increase the benefits of the adoption of Java for parallel computing, in order to achieve higher programming productivity.

ACKNOWLEDGEMENT

This work was funded by the Ministry of Education of Spain (FPU grant AP2010-4348), the Ministry of Economy and Competitiviness (project TIN2010-16735) and the Galician Government (project GRC2013/055), partially supported by FEDER funds of the EU. The authors thankfully acknowledge the High Performance Computing Center Stuttgart (HLRS) for providing access to the Cray XE6 supercomputer (Hermit) through a research visit to the Rechenzentrum Universität Mannheim (Germany) funded by an INDITEX-UDC grant.

## REFERENCES

1. Suganuma T, Ogasawara T, Takeuchi M, Yasue T, Kawahito M, Ishizaki K, Komatsu H, Nakatani T. Overview of the IBM Java just-in-time compiler. *IBM Systems Journal* 2000; **39**(1):175–193.
2. Shafi A, Carpenter B, Baker M, Hussain A. A comparative study of Java and C performance in two large-scale parallel applications. *Concurrency and Computation: Practice and Experience* 2009; **21**(15):1882–1906.

3. Taboada GL, Ramos S, Expósito RR, Touriño J, Doallo R. Java in the high performance computing arena: research, practice and experience. *Science of Computer Programming* 2013; **78**(5):425–444.
4. Apache Hadoop. `http://hadoop.apache.org/`. [Last visited: April 2014].
5. Dean J, Ghemawat S. MapReduce: simplified data processing on large clusters. *Communications of the ACM* 2008; **51**(1):107–113.
6. Message Passing Interface Forum. MPI: a Message Passing Interface standard. `http://www.mcs.anl.gov/research/projects/mpi/` 1995. [Last visited: April 2014].
7. Hongwei Z, Wan H, Jizhong H, Jin H, Lisheng Z. A performance study of Java communication stacks over InfiniBand and Gigabit Ethernet. *Proc. of 4th IFIP International Conference on Network and Parallel Computing Workshops (NPC'07)*, Dalian, China, 2007; 602–607.
8. Carpenter B, Getov V, Judd G, Skjellum A, Fox G. MPJ: MPI-like message passing for Java. *Concurrency and Computation: Practice and Experience* 2000; **12**(11):1019–1038.
9. Expósito RR, Ramos S, Taboada GL, Touriño J, Doallo R. FastMPJ: a scalable and efficient Java message-passing library. *Cluster Computing* 2014; (in press, `http://dx.doi.org/10.1007/s10586-014-0345-4`).
10. Expósito RR, Taboada GL, Touriño J, Doallo R. Design of scalable Java message-passing communications over InfiniBand. *Journal of Supercomputing* 2012; **61**(1):141–165.
11. IBTA. InfiniBand trade association. `http://www.infinibandta.org/`. [Last visited: April 2014].
12. TOP500 Org. Top 500 supercomputer sites. `http://www.top500.org/`. [Last visited: April 2014].
13. RDMA consortium. Architectural specifications for RDMA over TCP/IP. `http://www.rdmaconsortium.org/`. [Last visited: April 2014].
14. IBTA. InfiniBand architecture specification release 1.2.1 annex A16: RDMA over Converged Ethernet (RoCE).
15. IETF RFC 5041. Direct data placement over reliable transports. `http://www.ietf.org/rfc/rfc5041`. [Last visited: April 2014].
16. IEEE 8021. Data Center Bridging (DCB) task group. `http://www.ieee802.org/1/pages/dcbridges.html`. [Last visited: April 2014].
17. Chen D et al. The IBM Blue Gene/Q interconnection network and message unit. *Proc. of 23rd ACM/IEEE Supercomputing Conference (SC'11)*, Seattle, WA, USA, 2011; 26:1–26:10.
18. Alverson R, Roweth D, Kaplan L. The Gemini system interconnect. *Proc. of 18th IEEE Annual Symposium on High-Performance Interconnects (HOTI'10)*, Google Campus, Mountain View, CA, USA, 2010; 83–87.
19. Kim J, Dally W, Scott S, Abts D. Technology-driven, highly-scalable Dragonfly topology. *Proc. of 35th Annual International Symposium on Computer Architecture (ISCA'08)*, Beijing, China, 2008; 77–88.
20. The OpenFabrics Alliance (OFA). `http://www.openfabrics.org/`. [Last visited: April 2014].
21. IETF RFC 4392. IP over InfiniBand (IPoIB) architecture. `http://www.ietf.org/rfc/rfc4392`. [Last visited: April 2014].
22. Goldenberg D, Kagan M, Ravid R, Tsirkin M. Zero copy sockets direct protocol over InfiniBand - Preliminary implementation and performance analysis. *Proc. of 13th IEEE Annual Symposium on High-Performance Interconnects (HOTI'05)*, Stanford University, CA, USA, 2005; 128–137.
23. Fox G, Furmanski W. Java for parallel computing and as a general language for scientific and engineering simulation and modeling. *Concurrency: Practice and Experience* 1997; **9**(6):415–425.
24. Thiruvathukal GK, Dickens PM, Bhatti S. Java on networks of workstations (JavaNOW): a parallel computing framework inspired by Linda and the Message Passing Interface (MPI). *Concurrency: Practice and Experience* 2000; **12**(11):1093–1116.
25. Dunning D et al. The virtual interface architecture. *IEEE Micro* 1998; **18**(2):66–76.
26. Chang CC, von Eicken T. Javia: a Java interface to the virtual interface architecture. *Concurrency: Practice and Experience* 2000; **12**(7):573–593.
27. Welsh M, Culler D. Jaguar: enabling efficient communication and I/O in Java. *Concurrency: Practice and Experience* 2000; **12**(7):519–538.
28. van Nieuwpoort RV, Maassen J, Wrzesinska G, Hofman R, Jacobs C, Kielmann T, Bal HE. Ibis: a flexible and efficient Java-based grid programming environment. *Concurrency and Computation: Practice and Experience* 2005; **17**(7-8):1079–1107.
29. Righi RR, Navaux POA, Cera MC, Pasin M. Asynchronous communication in Java over InfiniBand and DECK. *Proc. of 17th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD'05)*, Rio de Janeiro, Brazil, 2005; 176–183.
30. Taboada GL, Touriño J, Doallo R. Java Fast Sockets: enabling high-speed Java communications on high performance clusters. *Computer Communications* 2008; **31**(17):4049–4059.
31. Baduel L, Baude F, Caromel D. Object-oriented SPMD. *Proc. of 5th IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID'05)*, Cardiff, Wales, UK, 2005; 824–831.
32. Philippsen M, Haumacher B, Nester C. More efficient serialization and RMI for Java. *Concurrency: Practice and Experience* 1999; **12**(7):495–518.
33. Maassen J, Van Nieuwpoort R, Veldema R, Bal H, Kielmann T, Jacobs C, Hofman R. Efficient Java RMI for parallel programming. *ACM Transactions on Programming Languages and Systems* 2001; **23**(6):747–775.
34. Taboada GL, Teijeiro C, Touriño J. High performance Java remote method invocation for parallel computing on clusters. *Proc. of 12th IEEE Symposium on Computers and Communications (ISCC'07)*, Aveiro, Portugal, 2007; 233–239.
35. Lobosco M, Silva A, Loques O, de Amorim CL. A new distributed JVM for cluster computing. *Proc. of 9th European Conference on Parallel Processing (Euro-Par'03)*. Klagenfurt, Austria, 2003; 1207–1215.
36. Veldema R, Hofman RF, Bhoedjang RA, Bal HE. Run-time optimizations for a Java DSM implementation. *Concurrency and Computation: Practice and Experience* 2003; **15**(3-5):299–316.
37. Veldema R, Philippsen M. Evaluation of RDMA opportunities in an object-oriented DSM. *Proc. of 20th International Workshop on Languages and Compilers for Parallel Computing (LCPC'07)*. Urbana, IL, USA, 2007; 217–231.

38. Huang W, Zhang H, He J, Han J, Zhang L. Jdib: Java applications interface to unshackle the communication capabilities of InfiniBand networks. *Proc. of 4th IFIP International Conference on Network and Parallel Computing Workshops (NPC'07)*, Dalian, China, 2007; 596–601.
39. Huang W, Han J, He J, Zhang L, Lin Y. Enabling RDMA capability of InfiniBand network for Java applications. *Proc. of 4th IEEE International Conference on Networking, Architecture, and Storage (NAS'08)*, Chongqing, China, 2008; 187–188.
40. Stuedi P, Metzler B, Trivedi A. jVerbs: ultra-low latency for data center applications. *Proc. of 4th ACM Annual Symposium on Cloud Computing (SOCC'13)*, Santa Clara, CA, USA, 2013; 10:1–10:14.
41. Baker M, Carpenter B, Fox G, Ko SH, Lim S. mpiJava: an object-oriented Java interface to MPI. *Proc. of 1st International Workshop on Java for Parallel and Distributed Computing (IWJPDC'99)*, San Juan, Puerto Rico, 1999; 748–762.
42. Carpenter B, Fox G, Ko SH, Lim S. mpiJava 1.2: API specification. `http://www.hpjava.org/reports/mpiJava-spec/mpiJava-spec/mpiJava-spec.html`. [Last visited: April 2014].
43. Taboada GL, Touriño J, Doallo R, Shafi A, Baker M, Carpenter B. Device level communication libraries for high-performance computing in Java. *Concurrency and Computation: Practice and Experience* 2011; **23**(18):2382–2403.
44. Gabriel E et al. Open MPI: goals, concept, and design of a next generation MPI implementation. *Proc. of 11th European PVM/MPI Users' Group Meeting (EuroPVM/MPI'04)*, Budapest, Hungary, 2004; 97–104.
45. Cheptsov A. Promoting data-centric supercomputing to the WWW world: Open MPI's Java bindings. *Proc. of 1st International Workshop on Engineering Object-Oriented Parallel Software (EOOPS'13)*, Barcelona, Spain, 2013; 1417–1422.
46. Baker M, Carpenter B, Shafi A. MPJ Express: towards thread safe Java HPC. *Proc. of 8th IEEE International Conference on Cluster Computing (CLUSTER'06)*, Barcelona, Spain, 2006; 1–10.
47. Myrinet eXpress (MX). A high performance, low-level, message-passing interface for Myrinet. Version 1.2. October 2006. `http://www.myricom.com/scs/MX/doc/mx.pdf`. [Last visited: April 2014].
48. Cray Inc. Using the GNI and DMAPP APIs. September 2013. `http://docs.cray.com/books/S-2446-51/S-2446-51.pdf`. [Last visited: April 2014].
49. Karo M, Lagerström R, Kohnke M, Albing C. The application level placement scheduler. *Proc. of 48th Cray User Group (CUG'06)*, Lugano, Switzerland, 2006; 1–7.
50. Balaji P et al. PMI: a scalable parallel process-management interface for extreme-scale systems. *Proc. of 17th European MPI Users' Group Meeting (EuroMPI'10)*, Stuttgart, Germany, 2010; 31–41.
51. Tezuka H, O'Carroll F, Hori A, Ishikawa Y. Pin-down cache: a virtual memory management technique for zero-copy communication. *Proc. of 12th International Parallel Processing Symposium (IPPS'98)*, Orlando, FL, USA, 1998; 308–314.
52. Saini S et al. Performance evaluation of supercomputers using HPCC and IMB benchmarks. *Journal of Computer and System Sciences* 2008; **74**(6):965–982.
53. NetPIPE. A network protocol independent performance evaluator. `http://www.scl.ameslab.gov/netpipe/`. [Last visited: April 2014].
54. HLRS' Hermit supercomputer in the TOP500 list. `http://www.top500.org/system/177473`. [Last visited: April 2014].
55. Bailey DH et al. The NAS parallel benchmarks. *International Journal of High Performance Computing Applications* 1991; **5**(3):63–73.
56. Mallón DA, Taboada GL, Touriño J, Doallo R. NPB-MPJ: NAS parallel benchmarks implementation for message-passing in Java. *Proc. of 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP'09)*, Weimar, Germany, 2009; 181–190.
57. Shafi A, Hussain A, Raza J. A parallel implementation of the finite-domain time-difference algorithm using MPJ Express. *Proc. of 10th International Workshop on Java and Components for Parallelism, Distribution and Concurrency (IWJPDC'08)*, Miami, FL, USA, 2008; 1–6.

# Chapter 5

# Java in the HPC Arena: Research, Practice and Experience

The content of this chapter corresponds to the following journal paper:

**- Title:** Java in the high performance computing arena: research, practice and experience

**- Authors:** Guillermo L. Taboada, Sabela Ramos, Roberto R. Expósito, Juan Touriño, Ramón Doallo

**- Journal:** Science of Computer Programming

**- Editorial:** Elsevier B.V.

**- ISSN:** 0167-6423

**- Year:** 2013

**- Volume(Number):Pages:** 78(5):425–444

**- DOI:** 10.1016/j.scico.2011.06.002

The final publication is available at http://www.sciencedirect.com/science/article/pii/S0167642311001420. A copy of the accepted paper has been included next.

# Java in the High Performance Computing arena: Research, practice and experience

Guillermo L. Taboada [*], Sabela Ramos, Roberto R. Expósito, Juan Touriño, Ramón Doallo

*Computer Architecture Group, University of A Coruña, A Coruña, Spain*

A R T I C L E I N F O

*Article history:*

*Keywords:*
Java
High Performance Computing
Performance evaluation
Multi-core architectures
Message-passing
Threads
Cluster
InfiniBand

A B S T R A C T

The rising interest in Java for High Performance Computing (HPC) is based on the appealing features of this language for programming multi-core cluster architectures, particularly the built-in networking and multithreading support, and the continuous increase in Java Virtual Machine (JVM) performance. However, its adoption in this area is being delayed by the lack of analysis of the existing programming options in Java for HPC and thorough and up-to-date evaluations of their performance, as well as the unawareness on current research projects in this field, whose solutions are needed in order to boost the embracement of Java in HPC.

This paper analyzes the current state of Java for HPC, both for shared and distributed memory programming, presents related research projects, and finally, evaluates the performance of current Java HPC solutions and research developments on two shared memory environments and two InfiniBand multi-core clusters. The main conclusions are that: (1) the significant interest in Java for HPC has led to the development of numerous projects, although usually quite modest, which may have prevented a higher development of Java in this field; (2) Java can achieve almost similar performance to natively compiled languages, both for sequential and parallel applications, being an alternative for HPC programming; (3) the recent advances in the efficient support of Java communications on shared memory and low-latency networks are bridging the gap between Java and natively compiled applications in HPC. Thus, the good prospects of Java in this area are attracting the attention of both industry and academia, which can take significant advantage of Java adoption in HPC.

## 1. Introduction

Java has become a leading programming language soon after its release, especially in web-based and distributed computing environments, and it is an emerging option for High Performance Computing (HPC) [1,2]. The increasing interest in Java for parallel computing is based on its appealing characteristics: built-in networking and multithreading support, object orientation, platform independence, portability, type safety, security, it has an extensive API and a wide community of developers, and finally, it is the main training language for computer science students. Moreover, performance is no longer an obstacle. The performance gap between Java and native languages (e.g., C and Fortran) has been narrowing for the past years, thanks to the Just-in-Time (JIT) compiler of the Java Virtual Machine (JVM) that obtains native performance from Java bytecode. However, the use of Java in HPC is being delayed by the lack of analysis of the existing programming options in

[*] Corresponding author.
*E-mail addresses:* taboada@udc.es (G.L. Taboada), sramos@udc.es (S. Ramos), rreye@udc.es (R.R. Expósito), juan@udc.es (J. Touriño), doallo@udc.es (R. Doallo).

this area and thorough and up-to-date evaluations of their performance, as well as the unawareness on current research projects in Java for HPC, whose solutions are needed in order to boost its adoption.

Regarding HPC platforms, new deployments are increasing significantly the number of cores installed in order to meet the ever growing computational power demand. This current trend to multi-core clusters underscores the importance of parallelism and multithreading capabilities [3]. In this scenario Java represents an attractive choice for the development of parallel applications as it is a multithreaded language and provides built-in networking support, key features for taking full advantage of hybrid shared/distributed memory architectures. Thus, Java can use threads in shared memory (intra-node) and its networking support for distributed memory (inter-node) communication. Nevertheless, although the performance gap between Java and native languages is usually small for sequential applications, it can be particularly high for parallel applications when depending on inefficient communication libraries, which has hindered Java adoption for HPC. Therefore, current research efforts are focused on providing scalable Java communication middleware, especially on high-speed networks commonly used in HPC systems, such as InfiniBand or Myrinet.

The remainder of this paper is organized as follows. Section 2 analyzes the existing programming options in Java for HPC. Section 3 describes current research efforts in this area, with special emphasis on providing scalable communication middleware for HPC. A comprehensive performance evaluation of representative solutions in Java for HPC is presented in Section 4. Finally, Section 5 summarizes our concluding remarks.

## 2. Java for High Performance Computing

This section analyzes the existing programming options in Java for HPC, which can be classified into: (1) shared memory programming; (2) Java sockets; (3) Remote Method Invocation (RMI); and (4) message-passing in Java. These programming options allow the development of both high-level libraries and Java parallel applications.

### *2.1. Java shared memory programming*

There are several options for shared memory programming in Java for HPC, such as the use of Java threads, OpenMP-like implementations, and Titanium.

As Java has built-in multithreading support, the use of Java threads for parallel programming is quite extended due to its high performance, although it is a rather low-level option for HPC (work parallelization and shared data access synchronization are usually hard to implement). Moreover, this option is limited to shared memory systems, which provide less scalability than distributed memory machines. Nevertheless, its combination with distributed memory programming models can overcome this restriction. Finally, in order to partially relieve programmers from the low-level details of threads programming, Java has incorporated from the 1.5 specification the concurrency utilities, such as thread pools, tasks, blocking queues, and low-level high performance primitives for advanced concurrent programming like CyclicBarrier.

The project Parallel Java (PJ) [4] has implemented several high-level abstractions over these concurrency utilities, such as ParallelRegion (code to be executed in parallel), ParallelTeam (group of threads that execute a ParallelRegion) and ParallelForLoop (work parallelization among threads), allowing an easy thread-base shared memory programming. Moreover, PJ also implements the message-passing paradigm as it is intended for programming hybrid shared/distributed memory systems such as multi-core clusters.

There are two main OpenMP-like implementations in Java, JOMP [5] and JaMP [6]. JOMP consists of a compiler (written in Java, and built using the JavaCC tool) and a runtime library. The compiler translates Java source code with OpenMP-like directives to Java source code with calls to the runtime library, which in turn uses Java threads to implement parallelism. The whole system is "pure" Java (100% Java), and thus can be run on any JVM. Although the development of this implementation stopped in 2000, it has been used recently to provide nested parallelism on multi-core HPC systems [7]. Nevertheless, JOMP had to be optimized with some of the utilities of the concurrency framework, such as the replacement of the busy-wait implementation of the JOMP barrier by the more efficient java.util.concurrent.CyclicBarrier. The experimental evaluation of the hybrid Java message-passing + JOMP configuration (being the message-passing library thread-safe) showed up to 3 times higher performance than the equivalent pure message-passing scenario. Although JOMP scalability is limited to shared memory systems, its combination with distributed memory communication libraries (e.g., message-passing libraries) can overcome this issue. JaMP is the Java OpenMP-like implementation for Jackal [8], a software-based Java Distributed Shared Memory (DSM) implementation. Thus, this project is limited to this environment. JaMP has followed the JOMP approach, but taking advantage of the concurrency utilities, such as tasks, as it is a more recent project.

The OpenMP-like approach has several advantages over the use of Java threads, such as the higher-level programming model with a code much closer to the sequential version and the exploitation of the familiarity with OpenMP, thus increasing programmability. However, current OpenMP-like implementations are still preliminary works and lack efficiency (busy-wait JOMP barrier) and portability (JaMP).

Titanium [9] is an explicitly parallel dialect of Java developed at UC Berkeley which provides the Partitioned Global Address Space (PGAS) programming model, like UPC and Co-array Fortran, thus achieving higher programmability. Besides the features of Java, Titanium adds flexible and efficient multi-dimensional arrays and an explicitly parallel SPMD control model with lightweight synchronization. Moreover, it has been reported that it outperforms Fortran MPI code [10], thanks to its source-to-source compilation to C code and the use of native libraries, such as numerical and high-speed network communication libraries. However, Titanium presents several limitations, such as the avoidance of the use of Java threads and the lack of portability as it relies on Titanium and C compilers.

### *2.2. Java sockets*

Sockets are a low-level programming interface for network communication, which allows sending streams of data between applications. The socket API is widely extended and can be considered the standard low-level communication layer as there are socket implementations on almost every network protocol. Thus, sockets have been the choice for implementing in Java the lowest level of network communication. However, Java sockets usually lack efficient high-speed networks support [11], so it has to resort to inefficient TCP/IP emulations for full networking support. Examples of TCP/IP emulations are IP over InfiniBand (IPoIB), IPoMX on top of the Myrinet low-level library MX (Myrinet eXpress), and SCIP on SCI.

Java has two main sockets implementations, the widely extended Java IO sockets, and Java NIO (New I/O) sockets which provide scalable non-blocking communication support. However, both implementations do not provide high-speed network support nor HPC tailoring. Ibis sockets partly solve these issues adding Myrinet support and being the base of Ibis [12], a parallel and distributed Java computing framework. However, their implementation on top of the JVM sockets library limits their performance benefits.

Java Fast Sockets (JFS) [11] is our high performance Java socket implementation for HPC. As JVM IO/NIO sockets do not provide high-speed network support nor HPC tailoring, JFS overcomes these constraints by: (1) reimplementing the protocol for boosting shared memory (intra-node) communication; (2) supporting high performance native sockets communication over SCI Sockets, Sockets-MX, and Socket Direct Protocol (SDP), on SCI, Myrinet and InfiniBand, respectively; (3) avoiding the need of primitive data type array serialization; and (4) reducing buffering and unnecessary copies. Thus, JFS is able to reduce significantly JVM sockets communication overhead. Furthermore, its interoperability and user and application transparency through reflection allow for its immediate applicability on a wide range of parallel and distributed target applications.

### *2.3. Java Remote Method Invocation*

The Java Remote Method Invocation (RMI) protocol allows an object running in one JVM to invoke methods on an object running in another JVM, providing Java with remote communication between programs equivalent to Remote Procedure Calls (RPCs). The main advantage of this approach is its simplicity, although the main drawback is the poor performance shown by the RMI protocol.

ProActive [13] is an RMI-based middleware for parallel, multithreaded and distributed computing focused on Grid applications. ProActive is a fully portable "pure" Java (100% Java) middleware whose programming model is based on a Meta-Object protocol. With a reduced set of simple primitives, this middleware simplifies the programming of Grid computing applications: distributed on Local Area Network (LAN), on clusters of workstations, or for the Grid. Moreover, ProActive supports fault-tolerance, load-balancing, mobility, and security. Nevertheless, the use of RMI as its default transport layer adds significant overhead to the operation of this middleware.

The optimization of the RMI protocol has been the goal of several projects, such as KaRMI [14], RMIX [15], Manta [16], Ibis RMI [12], and Opt RMI [17]. However, the use of non-standard APIs, the lack of portability, and the insufficient overhead reductions, still significantly larger than socket latencies, have restricted their applicability. Therefore, although Java communication middleware (e.g., message-passing libraries) used to be based on RMI, current Java communication libraries use sockets due to their lower overhead. In this case, the higher programming effort required by the lower-level API allows for higher throughput, key in HPC.

### *2.4. Message-passing in Java*

Message-passing is the most widely used parallel programming paradigm as it is highly portable, scalable and usually provides good performance. It is the preferred choice for parallel programming distributed memory systems such as clusters, which can provide higher computational power than shared memory systems. Regarding the languages compiled to native code (e.g., C and Fortran), MPI is the standard interface for message-passing libraries.

Soon after the introduction of Java, there have been several implementations of Java message-passing libraries (eleven projects are cited in [18]). However, most of them have developed their own MPI-like binding for the Java language. The two main proposed APIs are the mpiJava 1.2 API [19], which tries to adhere to the MPI C++ interface defined in the MPI standard version 2.0, but restricted to the support of the MPI 1.1 subset, and the JGF MPJ (message-passing interface for Java) API [20], which is the proposal of the Java Grande Forum (JGF) [21] to standardize the MPI-like Java API. The main differences among these two APIs lie on naming conventions of variables and methods.

The message-passing in Java (MPJ) libraries can be implemented: (1) using Java RMI; (2) wrapping an underlying native messaging library like MPI through Java Native Interface (JNI); or (3) using Java sockets. Each solution fits with specific situations, but presents associated trade-offs. The use of Java RMI, a "pure" Java (100% Java) approach, as base for MPJ libraries, ensures portability, but it might not be the most efficient solution, especially in the presence of high-speed

**Table 1**
Java message-passing projects overview.

| | Pure Java Impl. | Socket impl. | | High-speed network support | | | API | | |
|---|---|---|---|---|---|---|---|---|---|
| | | Java IO | Java NIO | Myrinet | InfiniBand | SCI | mpiJava 1.2 | JGF MPJ | Other APIs |
| MPJava [23] | ✓ | | ✓ | | | | | | ✓ |
| Jcluster [24] | ✓ | ✓ | | | | | | | ✓ |
| Parallel Java [4] | ✓ | ✓ | | | | | | | ✓ |
| mpiJava [22] | | | | ✓ | ✓ | ✓ | ✓ | | |
| P2P-MPI [25] | ✓ | ✓ | ✓ | | | | ✓ | | |
| MPJ Express [7] | ✓ | | ✓ | ✓ | | | ✓ | | |
| MPJ/Ibis [26] | ✓ | ✓ | | ✓ | | | | ✓ | |
| JMPI [27] | ✓ | ✓ | | | | | | | ✓ |
| F-MPJ [28] | ✓ | ✓ | | ✓ | ✓ | ✓ | ✓ | | |

communication hardware. The use of JNI has portability problems, although usually in exchange for higher performance. The use of a low-level API, Java sockets, requires an important programming effort, especially in order to provide scalable solutions, but it significantly outperforms RMI-based communication libraries. Although most of the Java communication middleware is based on RMI, MPJ libraries looking for efficient communication have followed the latter two approaches.

The mpiJava library [22] consists of a collection of wrapper classes that call a native MPI implementation (e.g., MPICH2 or OpenMPI) through JNI. This wrapper-based approach provides efficient communication relying on native libraries, adding a reduced JNI overhead. However, although its performance is usually high, mpiJava currently only supports some native MPI implementations, as wrapping a wide number of functions and heterogeneous runtime environments entails an important maintaining effort. Additionally, this implementation presents instability problems, derived from the native code wrapping, and it is not thread-safe, being unable to take advantage of multi-core systems through multithreading.

As a result of these drawbacks, the mpiJava maintenance has been superseded by the development of MPJ Express [7], a "pure" Java message-passing implementation of the mpiJava 1.2 API specification. MPJ Express is thread-safe and presents a modular design which includes a pluggable architecture of communication devices that allows to combine the portability of the "pure" Java shared memory (smpdev device) and New I/O package (Java NIO) communications (niodev device) with the high performance Myrinet support (through the native Myrinet eXpress (MX) communication library in mxdev device).

Currently, MPJ Express is the most active project in terms of uptake by the HPC community, presence on academia and production environments, and available documentation. This project is also stable and publicly available along with its source code.

In order to update the compilation of Java message-passing implementations presented in [18], this paper presents the projects developed since 2003, in chronological order:

- MPJava [23] is the first Java message-passing library implemented on Java NIO sockets, taking advantage of their scalability and high performance communications.
- Jcluster [24] is a message-passing library which provides both PVM-like and MPI-like APIs and is focused on automatic task load balance across large-scale heterogeneous clusters. However, its communications are based on UDP and it lacks high-speed networks support.
- Parallel Java (PJ) [4] is a "pure" Java parallel programming middleware that supports both shared memory programming (see Section 2.1) and an MPI-like message-passing paradigm, allowing applications to take advantage of hybrid shared/distributed memory architectures. However, the use of its own API makes its adoption difficult.
- P2P-MPI [25] is a peer-to-peer framework for the execution of MPJ applications on the Grid. Among its features are: (1) self-configuration of peers (through JXTA peer-to-peer technology); (2) fault-tolerance, based on process replication; (3) a data management protocol for file transfers on the Grid; and (4) an MPJ implementation that can use either Java NIO or Java IO sockets for communications, although it lacks high-speed networks support. In fact, this project is tailored to grid computing systems, disregarding the performance aspects.
- MPJ/Ibis [26] is the only JGF MPJ API implementation up to now. This library can use either "pure" Java communications, or native communications on Myrinet. Moreover, there are two low-level communication devices available in Ibis for MPJ/Ibis communications: TCPIbis, based on Java IO sockets (TCP), and NIOIbis, which provides blocking and non-blocking communication through Java NIO sockets. Nevertheless, MPJ/Ibis is not thread-safe, and its Myrinet support is based on the GM library, which shows poorer performance than the MX library.
- JMPI [27] is an implementation which can use either Java RMI or Java sockets for communications. However, the reported performance is quite low (it only scales up to two nodes).
- Fast MPJ (F-MPJ) [28] is our Java message-passing implementation which provides high-speed networks support, both direct and through Java Fast Sockets (see Section 3.1). F-MPJ implements the mpiJava 1.2 API, the most widely extended, and includes a scalable MPJ collectives library [29].

Table 1 serves as a summary of the Java message-passing projects discussed in this section.

## 3. Java for HPC: current research

This section describes current research efforts in Java for HPC, which can be classified into: (1) design and implementation of low-level Java message-passing devices; (2) improvement of the scalability of Java message-passing collective primitives; (3) automatic selection of MPJ collective algorithms; (4) implementation and evaluation of efficient MPJ benchmarks; (5) language extensions in Java for parallel programming paradigms; and (6) Java libraries to support data parallelism. These ongoing projects are providing Java with several evaluations of their suitability for HPC, as well as solutions for increasing their performance and scalability in HPC systems with high-speed networks and hardware accelerators such as Graphics Processing Units (GPUs).

### *3.1. Low-level Java message-passing communication devices*

The use of pluggable low-level communication devices for high performance communication support is widely extended in native message-passing libraries. Both MPICH2 and OpenMPI include several devices on Myrinet, InfiniBand and shared memory. Regarding MPJ libraries, in MPJ Express the low-level xdev layer [7] provides communication devices for different interconnection technologies. The three implementations of the xdev API currently available are niodev (over Java NIO sockets), mxdev (over Myrinet MX), and smpdev (shared memory communication), which has been introduced recently [30]. This latter communication device has two implementations, one thread-based (pure Java) and the other based on native IPC resources.

F-MPJ communication devices conform with the xxdev API [28], which supports the direct communication of any serializable object without data buffering, whereas xdev, the API that xxdev is extending, does not support this direct communication, relying on a buffering layer (mpjbuf layer). Additional benefits of the use of this API are its flexibility, portability and modularity thanks to its encapsulated design.

The xxdev API (see Listing 1) has been designed with the goal of being simple and small, providing only basic communication methods in order to ease the development of xxdev devices. In fact, this API is composed of 13 simple methods, which implement basic message-passing operations, such as point-to-point communication, both blocking (send and recv, like MPI_Send and MPI_Recv) and non-blocking (isend and irecv, like MPI_Isend and MPI_Irecv). Moreover, synchronous communications are also embraced (ssend and issend). However, these communication methods use ProcessID objects instead of using ranks as arguments to send and receive primitives. In fact, the xxdev layer is focused on providing basic communication methods and it does not deal with high-level message-passing abstractions such as groups and communicators. Therefore, a ProcessID object unequivocally identifies a device object.

**Listing 1**
API of the xxdev.Device class

```
public class Device {
  static public Device newInstance(String deviceImplementation);
  ProcessID[] init(String[] args);
  ProcessID id();
  void finish();

  Request isend(Object message, ProcessID dstID, int tag, int context);
  Request irecv(Object message, ProcessID srcID, int tag, int context, Status status);
  void send(Object message, ProcessID  dstID, int tag, int context);
  Status recv(Objecct message, ProcessID srcID, int tag, int context);
  Request issend(Object message, ProcessID dstID, int tag, int context);
  void ssend(Object message, ProcessID srcID, int tag, int context);

  Status iprobe(ProcessID srcID, int tag, int context);
  Status probe(ProcessID srcID, int tag, int context);
  Request peek();
}
```

Fig. 1 presents an overview of the F-MPJ communication devices on shared memory and cluster networks. From top to bottom, the communication support of MPJ applications run with F-MPJ is implemented in the device layer. Current F-MPJ communication devices are implemented either on JVM threads (smpdev, a thread-based device), on sockets over the TCP/IP stack (iodev on Java IO sockets), or on native communication layers such as Myrinet eXpress (mxdev) and InfiniBand Verbs (IBV) (ibvdev), which are accessed through JNI.

The initial implementation of F-MPJ included only one communication device, iodev, implemented on top of Java IO sockets, which therefore can rely on top of JFS and hence obtain high performance on shared memory and Gigabit Ethernet, SCI, Myrinet, and InfiniBand networks. However, the use of sockets in a communication device, despite the high performance provided by JFS, still represents an important source of overhead in Java communications. Thus, F-MPJ is including the direct support of communications on high performance native communication layers, such as MX and IBV.

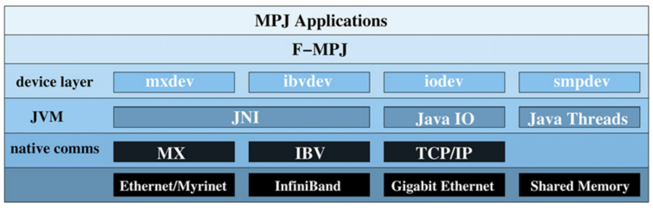


**Fig. 1.** F-MPJ communication devices on shared memory and cluster networks.

The mxdev device implements the xxdev API on MX, which runs natively on Myrinet and high-speed Ethernet networks, such as 10 Gigabit Ethernet, relying on MXoE (MX over Ethernet) stack. As MX already provides a low-level messaging API, mxdev deals with the Java Objects marshaling and communication, the JNI transfers and the MX parameters handling. The ibvdev device implements the xxdev API on IBV, the low-level InfiniBand communication driver, in order to take full advantage of the InfiniBand network. Unlike mxdev, ibvdev has to implement its own communication protocols, as IBV API is quite close to the InfiniBand Network Interface Card (NIC) operation. Thus, this communication device has implemented two communication protocols, eager and rendezvous, on RDMA (Remote Direct Memory Access) Write/Send operations. This direct access of Java to InfiniBand network was somewhat restricted so far to MPI libraries. Like mxdev, this device has to deal with the Java Objects communication and the JNI transfers, and additionally with the communication protocols operation. Finally, both mxdev and ibvdev, although they have been primarily designed for network communication, support shared memory intra-node communication. However, smpdev device is the thread-based communication device that should support more efficiently shared memory transfers. This device isolates a naming space for each running thread (relying on custom class loaders) and allocates shared message queues in order to implement the communications as regular data copies between threads.

### *3.2. MPJ collectives scalability*

MPJ application developers use collective primitives for performing standard data movements (e.g., Broadcast, Scatter, Gather and Alltoall or total exchange) and basic computations among several processes (reductions). This greatly simplifies code development, enhancing programmers productivity together with MPJ programmability. Moreover, it relieves developers from communication optimization. Thus, collective algorithms, which generally consist of multiple point-to-point communications, must provide scalable performance, usually through overlapping communications in order to maximize the number of operations carried out in parallel. An unscalable algorithm can easily waste the performance provided by an efficient communication middleware.

The design, implementation and runtime selection of efficient collective communication operations have been extensively discussed in the context of native message-passing libraries [31–34], while there is little discussion in MPJ, except for F-MPJ, which provides a scalable and efficient MPJ collective communication library [29] for parallel computing on multi-core architectures. This library provides multi-core aware primitives, implements several algorithms per collective operation, and explores thread-based communications, obtaining significant performance benefits in communication-intensive MPJ applications.

The collective algorithms present in MPJ libraries can be classified in six types, namely Flat Tree (FT) or linear, Minimum-Spanning Tree (MST), Binomial Tree (BT), Four-ary Tree (FaT), Bucket (BKT) or cyclic, and BiDirectional Exchange (BDE) or recursive doubling, which are extensively described in [32]. Table 2 presents a complete list of the collective algorithms used in MPJ Express and F-MPJ (the prefix “b” means that only blocking point-to-point communication is used, whereas “nb” refers to the use of non-blocking primitives). It can be seen that F-MPJ implements up to six algorithms per collective primitive, allowing their selection at runtime, as well as it takes more advantage of communications overlapping, achieving higher performance scalability. Regarding the memory requirements of the collective primitives, some algorithms require more memory than others (e.g., the MST algorithm for the Scatter and Gather demands more memory than the FT algorithm). Thus, when experiencing memory limitations the algorithms with less memory requirements must be selected in order to overcome the limitation.

### *3.3. Automatic selection of MPJ collective algorithms*

The F-MPJ collectives library allows the runtime selection of the collective algorithm that provides the highest performance in a given multi-core system, among the several algorithms available, based on the message size and the number of processes. The definition of a threshold for each of these two parameters allows the selection of up to four algorithms per collective primitive. Moreover, these thresholds can be configured for a particular system by means of an autotuning process, which obtains an optimal selection of algorithms, based on the particular performance results on a specific system and taking into account the particularities of the Java execution model.

The information of the selected algorithms is stored in a configuration file that, if available in the system, is loaded at MPJ initialization, otherwise the default algorithms are selected, thus implementing a portable and user transparent approach.

**Table 2**
Algorithms implemented in MPJ collectives libraries.

| Primitive | MPJ Express collectives library | F-MPJ collectives library |
|---|---|---|
| Barrier | Gather+Bcast | nbFTGather+bFaTBcast, Gather+Bcast, BT |
| Bcast | bFaTBcast | bFT, nbFT, bFaTBcast, MST |
| Scatter | nbFT | nbFT, MST |
| Scatterv | nbFT | nbFT, MST |
| Gather | nbFT | bFT, nbFT, nb1FT, MST |
| Gatherv | nbFT | bFT, nbFT, nb1FT, MST |
| Allgather | nbFT, BT | nbFT, BT, nbBDE, bBKT, nbBKT, Gather+Bcast |
| Allgatherv | nbFT, BT | nbFT, BT, nbBDE, bBKT, nbBKT, Gather+Bcast |
| Alltoall | nbFT | bFT, nbFT, nb1FT, nb2FT |
| Alltoallv | nbFT | bFT, nbFT, nb1FT, nb2FT |
| Reduce | bFT | bFT, nbFT, MST |
| Allreduce | nbFT, BT | nbFT, BT, bBDE, nbBDE, Reduce+Bcast |
| Reduce-Scatter | Reduce+Scatterv | bBDE, nbBDE, bBKT, nbBKT, Reduce+Scatterv |
| Scan | nbFT | nbFT, linear |

**Table 3**
Example of configuration file for the selection of collective algorithms.

| Primitive | Short message/ small number of processes | Short message/ large number of processes | Long message/ small number of processes | Long message/ large number of processes |
|---|---|---|---|---|
| Barrier | nbFTGather+bFatBcast | nbFTGather+bFatBcast | Gather+Bcast | Gather+Bcast |
| Bcast | nbFT | MST | MST | MST |
| Scatter | nbFT | nbFT | nbFT | nbFT |
| Gather | nbFT | nbFT | MST | MST |
| Allgather | Gather+Bcast | Gather+Bcast | Gather+Bcast | Gather+Bcast |
| Alltoall | nb2FT | nb2FT | nb2FT | nb2FT |
| Reduce | nbFT | nbFT | MST | MST |
| Allreduce | Reduce+Bcast | Reduce+Bcast | Reduce+Bcast | Reduce+Bcast |
| Reduce-Scatter | bFTReduce+nbFTScatterv | bFTReduce+nbFTScatterv | BDE | BDE |
| Scan | Linear | Linear | Linear | Linear |

The autotuning process consists of the execution of our own MPJ collectives micro-benchmark suite [18], the gathering of their experimental results, and finally the generation of the configuration file that contains the algorithms that maximize performance. The performance results have been obtained on a number of processes power of two, up to the total number of cores of the system, and for message sizes power of two. The parameter thresholds, which are independently configured for each collective, are those that maximize the performance measured by the micro-benchmark suite. Moreover, this autotuning process is required to be executed only once per system configuration in order to generate the configuration file. After that MPJ applications would take advantage of this information.

Table 3 presents the information contained in the optimum configuration file for the x86-64 multi-core cluster used in the experimental evaluation presented in this paper (Section 4). The thresholds between short and long messages, and between small and large number of processes are specific for each collective, although in the evaluated testbeds their values are generally 32 Kbytes and 16 processes, respectively.

### 3.4. Implementation and evaluation of efficient HPC benchmarks

Java lacks efficient HPC benchmarking suites for characterizing its performance, although the development of efficient Java benchmarks and the assessment of their performance is highly important. The JGF benchmark suite [35], the most widely used Java HPC benchmarking suite, presents quite inefficient codes, as well as it does not provide the native language counterparts of the Java parallel codes, preventing their comparative evaluation. Therefore, we have implemented the NAS Parallel Benchmarks (NPB) suite for MPJ (NPB-MPJ) [36], selected as this suite is the most extended in HPC evaluations, with implementations for MPI (NPB-MPI), OpenMP (NPB-OMP), Java threads (NPB-JAV) and ProActive (NPB-PA).

NPB-MPJ allows, as main contributions: (1) the comparative evaluation of MPJ libraries; (2) the analysis of MPJ performance against other Java parallel approaches (e.g., Java threads); (3) the assessment of MPJ versus native MPI scalability; (4) the study of the impact on performance of the optimization techniques used in NPB-MPJ, from which Java HPC applications can potentially benefit. The description of the NPB-MPJ benchmarks implemented is next shown in Table 4.

In order to maximize NPB-MPJ performance, the "plain objects" design has been chosen as it reduces the overhead of the "pure" object-oriented design (up to 95% overhead reduction). Thus, each benchmark uses only one object instead of defining an object per each element of the problem domain. Thus, complex numbers are implemented as two-element arrays instead of complex numbers objects.

The inefficient multi-dimensional array support in Java (an $n$-dimensional array is defined as an array of $n-1$-dimensional arrays, so data is not guaranteed to be contiguous in memory) imposed a significant performance penalty in NPB-MPJ, which handles arrays of up to five dimensions. This overhead was reduced through the array flattening optimization, which consists

**Table 4**
NPB-MPJ benchmarks description.

| Name | Operation | Communicat. intensiveness | Kernel | Applic. |
|---|---|---|---|---|
| CG | Conjugate Gradient | Medium | ✓ | |
| EP | Embarrassingly Parallel | Low | ✓ | |
| FT | Fourier Transformation | High | ✓ | |
| IS | Integer Sort | High | ✓ | |
| MG | Multi-Grid | High | ✓ | |
| SP | Scalar Pentadiagonal | Low | | ✓ |

of the mapping of a multi-dimensional array in a one-dimensional array. Thus, adjacent elements in the C/Fortran versions are also contiguous in Java, allowing the data locality exploitation.

Finally, the implementation of the NPB-MPJ takes advantage of the JVM JIT (Just-in-Time) compiler-based optimizations. The JIT compilation of the bytecode (or even its recompilation in order to apply further optimizations) is reserved to heavily used methods, as it is an expensive operation that increases significantly the runtime. Thus, the NPB-MPJ codes have been refactored toward simpler and independent methods, such as methods for mapping elements from multi-dimensional to one-dimensional arrays, and complex number operations. As these methods are invoked more frequently, the JVM gathers more runtime information about them, allowing a more effective optimization of the target bytecode.

The performance of NPB-MPJ significantly improved using these techniques, achieving up to 2800% throughput increase (on SP benchmark). Furthermore, we believe that other Java HPC codes can potentially benefit from these optimization techniques.

### 3.5. *Language extensions in Java for parallel programming paradigms*

Regarding language extensions in Java to support various parallel programming paradigms, X10 and Habanero Java deserve to be mentioned. X10 [37,38] is an emerging Java-based programming language developed in the DARPA program on High Productivity Computer Systems (HPCS). Moreover, it is an APGAS (Asynchronous Partitioned Global Address Space) language implementation focused on programmability which supports locality exploitation, lightweight synchronization, and productive parallel programming. Additionally, an ongoing project based on X10 is Habanero Java [39], focused on supporting productive parallel programming on extreme scale homogeneous and heterogeneous multi-core platforms. It allows to take advantage of X10 features in shared memory systems together with the Java Concurrency framework. Both X10 and Habanero Java applications can be compiled with C++ or Java backends, although looking for performance the use of the C++ one is recommended. Nevertheless, these are still experimental projects with limited performance, especially for X10 arrays handling, although X10 has been reported to rival Java threads performance on shared memory [40].

### 3.6. *Java libraries to support data parallelism*

There are several ongoing efforts in the support in Java of data parallelism using hardware accelerators, such as GPUs, once they have emerged as a viable alternative for significantly improving the performance of appropriate applications. On the one hand this support can be implemented in the compiler, at language level such as for JCUDA [41]. On the other hand, the interface to these accelerators can be library-based, such as the following Java bindings of CUDA: jcuda.org [42], jCUDA [43], JaCuda [44], Jacuzzi [45], and java-gpu [46].

Furthermore, the bindings are not restricted to CUDA as there are several Java bindings for OpenCL: jocl.org [47], JavaCL [48], and JogAmp [49].

This important number of projects is an example of the interest of the research community in supporting data parallelism in Java, although their efficiency is lower than using directly CUDA/OpenCL due to the overhead associated to the Java data movements to and from the GPU, the support of the execution of user-written CUDA code from Java programs and the automatic support for data transfer of primitives and multi-dimensional arrays of primitives. An additional project that targets these sources of inefficiency is JCudaMP [50], an OpenMP framework that exploits more efficiently GPUs. Finally, another approach for Java performance optimization on GPUs is the direct generation of GPU-executable code (without JNI access to CUDA/OpenCL) by a research Java compiler, Jikes, which is able to automatically parallelize loops [51].

## 4. Performance evaluation

This paper presents an up-to-date comparative performance evaluation of representative MPJ libraries, F-MPJ and MPJ Express, on two shared memory environments and two InfiniBand multi-core clusters. First, the performance of point-to-point MPJ primitives on InfiniBand, 10 Gigabit Ethernet and shared memory is presented. Next, this section evaluates the results gathered from a micro-benchmarking of MPJ collective primitives. Finally, the impact of MPJ libraries on the scalability of representative parallel codes, both NPB-MPJ kernels and the Gadget2 application [52], has been assessed comparatively with MPI, Java threads and OpenMP performance results.

### *4.1. Experimental configuration*

Two systems have been used in this performance evaluation, a multi-core x86-64 Infiniband cluster and the Finis Terrae supercomputer [53]. The first system (from now on x86-64 cluster) is a 16-node cluster with 16 Gbytes of memory and 2 x86-64 Xeon E5620 quad-core Nehalem-based "Gulftown" processors at 2.40 GHz per node (hence 128 physical cores in the cluster). The interconnection network is InfiniBand (QLogic IBA7220 4x DDR, 16 Gbps), although 2 of the nodes have additionally a 10 Gigabit Ethernet NIC (Intel PRO/10GbE NIC). As each node has 8 physical cores, and 16 logical cores when hyperthreading is enabled, shared memory performance has been also evaluated on one node of the cluster, using up to 16 processes/threads. The performance results on this system have been obtained using one core per node, except for 32, 64 and 128 processes, for which 2, 4 and 8 cores per node, respectively, have been used.

The OS is Linux CentOS 5.3, the C/Fortran compilers are the Intel compiler (used with -fast flag) version 11.1.073 and the GNU compiler (used with -O3 flag) version 4.1.2, both with OpenMP support, the native communication libraries are OFED (OpenFabrics Enterprise Distribution) 1.5 and Open-MX 1.3.4, for InfiniBand and 10 Gigabit Ethernet, respectively, and the JVM is Oracle JDK 1.6.0_23. Finally, the evaluated message-passing libraries are F-MPJ with JFS 0.3.1, MPJ Express 0.35, and OpenMPI 1.4.1.

The second system used is the Finis Terrae supercomputer (14 TFlops), an InfiniBand cluster which consists of 142 HP Integrity rx7640 nodes, each of them with 16 Montvale Itanium2 (IA64) cores at 1.6 GHz and 128 Gbytes of memory. The InfiniBand NIC is a 4X DDR Mellanox MT25208 (16 Gbps). Additionally an HP Integrity Superdome system with 64 Montvale Itanium 2 dual-core processors (total 128 cores) at 1.6 GHz and 1 TB of memory has also been used for the shared memory evaluation. The OS of the Finis Terrae is SUSE Linux Enterprise Server 10 with Intel compiler 10.1.074 (used with the -fast flag) and GNU compiler (used with the -O3 flag) version 4.1.2. Regarding native message-passing libraries, HP-MPI 2.2.5.1 has been selected as it achieves the highest performance on InfiniBand and shared memory on the Finis Terrae. The InfiniBand drivers are OFED version 1.4. The JVM is Oracle JDK 1.6.0_20 for IA64. The poor performance of Java on IA64 architectures, due to the lack of mature support for this processor in the Java Just-In-Time compiler, has motivated the selection of this system only for the analysis of the performance scalability of MPJ applications, due to its high number of cores. The performance results on this system have been obtained using 8 cores per node, the recommended configuration for maximizing performance. In fact, the use of a higher number of cores per node increases significantly network contention and memory access bottlenecks.

Regarding the benchmarks, Intel MPI Benchmarks (IMB, formerly Pallas) and our own MPJ micro-benchmark suite, which tries to adhere to IMB measurement methodology, have been used for the message-passing primitives evaluation. Moreover, the NPB-MPI/NPB-OMP version 3.3 and the NPB-JAV version 3.0 have been used together with our own NPB-MPJ implementation [36]. The metrics that have been considered for the NPB evaluation are the speedup and MOPS (Millions of Operations Per Second), which measures the operations performed in the benchmark, that differ from the CPU operations issued. Moreover, NPB Class C workloads have been selected as they are the largest workloads that can be executed in a single node, which imposes the restriction of using workloads with memory requirements below 16 Gbytes (the amount of memory available in a node of the x86-64 cluster).

### *4.2. Performance evaluation methodology*

All performance results presented in this paper are the median of 5 measurements in case of the kernels and applications and the median of up to the 1000 samples measured for the collective operations. The selection of the most appropriate performance evaluation methodology in Java has been thoroughly addressed in [54], concluding that the median is considered one of the best measures as its accuracy seems to improve with the number of measurements, which is in tune with the results reported in this paper.

Regarding the influence of JIT compilation in HPC performance results, the use of long-running codes (with runtimes of several hours and days) generally involves the use of a high percentage of JIT compiled code, which eventually improves performance. Moreover, the JVM execution mode selected for the performance evaluation is the default one (*mixed mode*) which compiles dynamically at runtime, based on profiling information, the bytecode of costly methods to native code, while interprets inexpensive pieces of code without incurring in runtime compilation overheads. Thus, this mode is able to provide higher performance than the use of the interpreted and even the compiled (an initial static compilation) execution modes. In fact, we have experimentally assessed the higher performance of the use of the mixed mode for the evaluated codes, whose percentage of runtime of natively compiled code is generally higher than 95% (hence, less than 5% of the runtime is generally devoted to interpreted code).

Furthermore, the non-determinism of JVM executions leads to oscillations in the time measures of Java applications. The main sources of variation are the JIT compilation and optimization in the JVM driven by a timer-based method sampling, thread scheduling, and garbage collection. However, the exclusive access to HPC resources and the characteristics of HPC applications (e.g., numerical intensive computation and a restricted use of object-oriented features such as extensions and handling numerous objects) limit the variations in the experimental results of Java. In order to assess the variability of representative Java codes in HPC, the NPB kernels evaluated in this paper (CG, FT, IS and MG with Class C problem size) have been executed 40 times, both using F-MPJ and MPI, on 64 and 128 cores of the x86-64 cluster. Regarding message-passing primitives, both point-to-point and collectives include calls to native methods, which provide efficient communications on

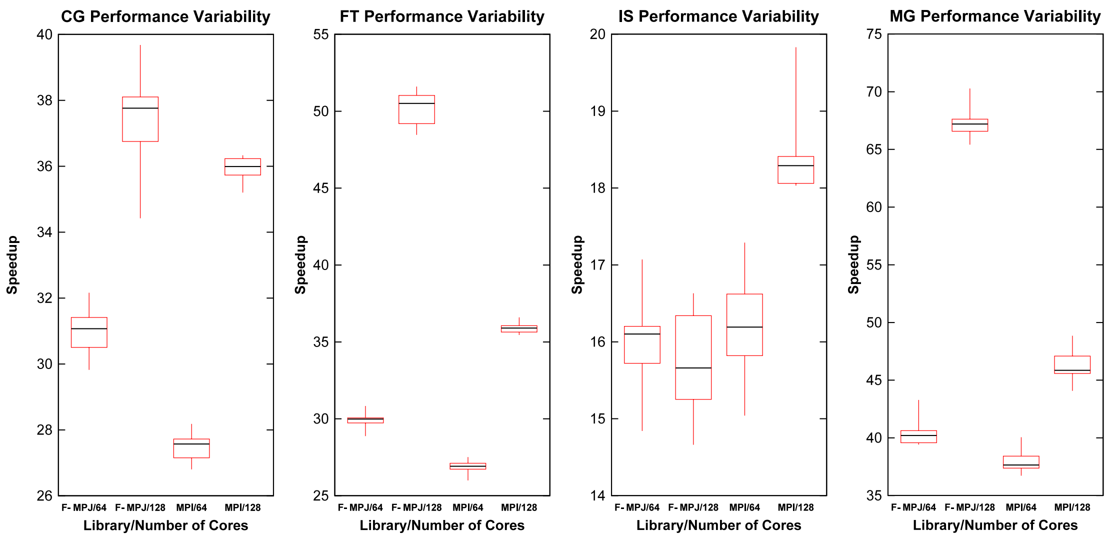


**Fig. 2.** NPB performance variability on the x86-64 cluster.

high-speed networks, thus obtaining performance results close to the theoretical limits of the network hardware. Moreover, their performance measures, when relying on native methods, provide results with little variation among iterations. Only message-passing transfers on shared memory present a high variability due to the scheduling of the threads on different cores within a node. In this scenario the performance results depend significantly on the scheduling of the threads on cores that belong to the same processor and that even can share some cache levels. Nevertheless, due to space restrictions a detailed analysis of the impact of thread scheduling on Java communications performance cannot be included in this paper. Thus, only the NPB kernels have been selected for the analysis of the performance variability of Java in HPC due to their balance in the combination of computation and communication as well as for their representativeness in HPC evaluation.

Fig. 2 presents speedup graphs with box and whisker diagrams for the evaluated benchmarks, showing the measure of the minimum sample, the lower quartile (Q1), the median (Q2), upper quartile (Q3), and the maximum sample. The selected metric, speedup, has been selected for clarity purposes, as it allows a straightforward analysis of F-MPJ and MPI results, especially for the comparison of their range of values, which lie closer using speedups than other metrics such as execution times.

The analysis of the variability of the performance of these NPB kernels shows that F-MPJ results present similar variability as MPI codes, although for CG and FT on 128 cores the NPB-MPJ measures present higher variations than their natively compiled counterparts (MPI kernels). However, even in this scenario the variability of the Java codes is less than 10% of the speedup value (the measured speedups fall in the range of 90% and 110% of the median value), whereas the average variation is less than 5% of the speedup value. Furthermore, there is no clear evidence of the increase of the variability with the number of cores, except for NPB-MPJ CG and FT.

### 4.3. *Experimental performance results on one core*

Fig. 3 shows a performance comparison of several NPB implementations on one core from the x86-64 cluster (left graph) and on one core from the Finis Terrae (right graph). The results are shown in terms of speedup relative to the MPI library (using the GNU C/Fortran compiler), Runtime(*NPB-MPI benchmark*) / Runtime(*NPB benchmark*). Thus, a value higher than 1 means than the evaluated benchmark achieves higher performance (shorter runtime) than the NPB-MPI benchmark, whereas a value lower than 1 means than the evaluated code shows poorer performance (longer runtime) than the NPB-MPI benchmark. The NPB implementations and NPB kernels evaluated are those that will be next used in this section for the performance analysis of Java kernels (Section 4.6.1). Moreover, only F-MPJ results are shown for NPB-MPJ performance for clarity purposes, as other MPJ libraries (e.g., MPJ Express) obtain quite similar results on one core.

The differences in performance that can be noted in the graphs are explained by the different implementations of the NPB benchmarks, the use of Java or native code (C/Fortran), and for native code the compiler being used (Intel or GNU compiler). Regarding Java performance, as the JVM used in this performance evaluation, the Oracle JVM for Linux, has been built with the GNU compiler, Java performance is limited by the throughput achieved with this compiler. Thus, Java codes (MPJ and Threads) cannot generally outperform their equivalent GNU-built benchmarks. This fact is of special relevance on the Finis Terrae, where the GNU compiler is not able to take advantage of the Montvale Itanium2 (IA64) processor, whereas the Intel compiler does. As a consequence of this, the performance of Java kernels on the Finis Terrae is significantly lower, even an

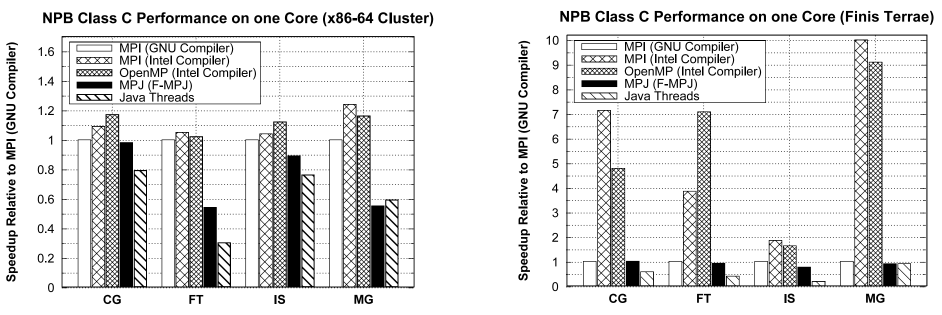


**Fig. 3.** NPB relative performance on one core.

order of magnitude lower, than the performance of the kernels built with the Intel compiler. The performance of Java kernels on the x86-64 cluster is close to the natively compiled kernels for CG and IS, whereas for FT and MG Java performance is approximately 55% of the performance of MPI kernels built with the GNU compiler.

This analysis of the performance of Java and natively compiled codes on the x86-64 cluster and the Finis Terrae has also verified that the use of the Intel compiler shows better performance results than the use of the GNU compiler, especially on the Finis Terrae. Thus, from now on only the Intel compiler has been used in the performance evaluation included in this paper, although a fair comparison with Java would have considered the GNU compiler (both Oracle JVM and the GNU compiler are freely available software). However, the use of the compiler provided by the processor vendor is the most generally adopted solution in HPC. Furthermore, a wider availability of JVMs built with commercial compilers would improve this scenario, especially on Itanium platforms.

### *4.4. Message-passing point-to-point micro-benchmarking*

The performance of message-passing point-to-point primitives has been measured on the x86-64 cluster using our own MPJ micro-benchmark suite and IMB. Regarding Finis Terrae, its results are not considered for clarity purposes, as well as due to the poor performance of Java on this system. Moreover, Finis Terrae communication mechanisms, InfiniBand and shared memory, are already covered in the x86-64 cluster evaluation.

Fig. 4 presents message-passing point-to-point latencies (for short messages) and bandwidths (for long messages) on InfiniBand (top graph), 10 Gigabit Ethernet (middle graph) and shared memory (bottom graph). Here, the results shown are the half of the round-trip time of a pingpong test or its corresponding bandwidth.

On the one hand these results show that F-MPJ is quite close to MPI performance, which means that F-MPJ is able to take advantage of the low latency and high throughput provided by shared memory and these high-speed networks. In fact, F-MPJ obtains start-up latencies as low as 2 $\mu$s on shared memory, 10 $\mu$s on InfiniBand and 12 $\mu$s on 10 Gigabit Ethernet. Regarding throughput, F-MPJ significantly outperforms MPI for 4 Kbytes and larger messages on shared memory when using smpdev communication device, achieving up to 51 Gbps thanks to the exploitation of the thread-based intra-process communication mechanism, whereas the inter-process communication protocols implemented in MPI and the F-MPJ network-based communication devices (ibvdev and mxdev) are limited to less than 31 Gbps.

On the other hand, MPJ Express point-to-point performance suffers from the lack of specialized support on InfiniBand, having to rely on NIO sockets over IP emulation IPoIB, and the use of a buffering layer, which adds noticeable overhead for long messages. Moreover, the communication protocols implemented in this library show a significant start-up latency. In fact, MPJ Express and F-MPJ rely on the same communication layer on shared memory (intra-process transfers) and 10 Gigabit Ethernet (Open-MX library), but MPJ Express adds an additional overhead of 8 $\mu$s and 11 $\mu$s, respectively, over F-MPJ.

### *4.5. Message-passing collective primitives micro-benchmarking*

Fig. 5 presents the performance of representative message-passing data movement operations (Broadcast and Allgather), and computational operations (Reduce and Allreduce double precision sum operations), as well as their associated scalability using a representative message size (32 Kbytes). The results, obtained using 128 processes on the x86-64 cluster, are represented using aggregated bandwidth metric as this metric takes into account the global amount of data transferred, generally *message size* $*$ *number of processes*.

The original MPJ Express collective primitives use the algorithms listed in Table 2 (column MPJ Express), whereas F-MPJ collectives library uses the algorithms that maximize the performance on this cluster according to the automatic performance tunning process. The selected algorithms are presented in Table 5, which extracts from the configuration file the most relevant information about the evaluated primitives.

The results confirm that F-MPJ is bridging the gap between MPJ and MPI collectives performance, but there is still room for improvement, especially when using several processes per node as F-MPJ collectives are not taking full advantage of the

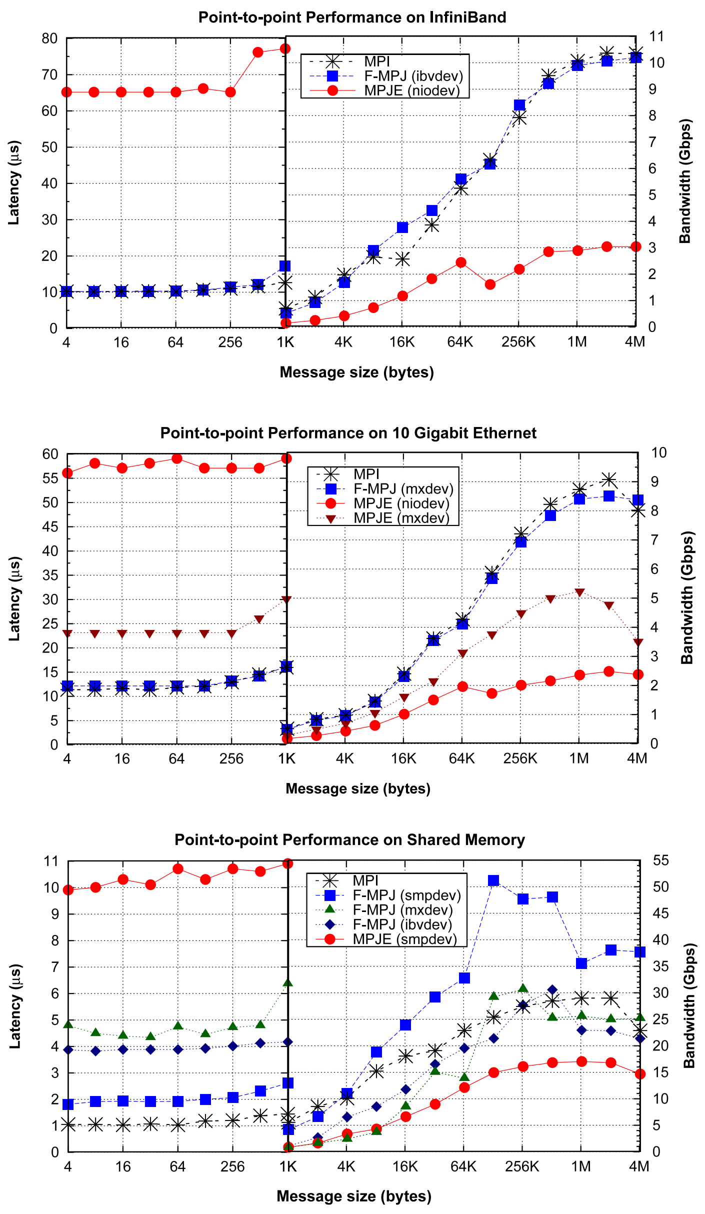


**Fig. 4.** Message-passing point-to-point performance on InfiniBand, 10 Gigabit Ethernet and shared memory.

**Table 5**
Algorithms that maximize performance on the x86-64 cluster.

| Primitive | Short message/small number of processes | Short message/large number of processes | Long message/small number of processes | Long message/large number of processes |
|---|---|---|---|---|
| Bcast | nbFT | MST | MST | MST |
| Allgather | nbFTGather+nbFTBcast | nbFTGather+MSTBcast | MSTGather+MSTBcast | MSTGather+MSTBcast |
| Reduce | bFT | bFT | MST | MST |
| Allreduce | bFTReduce+nbFTBcast | bFTReduce+MSTBcast | MSTReduce+MSTBcast | MSTReduce+MSTBcast |

cores available within each node. The scalability graphs (right graphs) confirm this analysis, especially for the Broadcast and the Reduce operations.

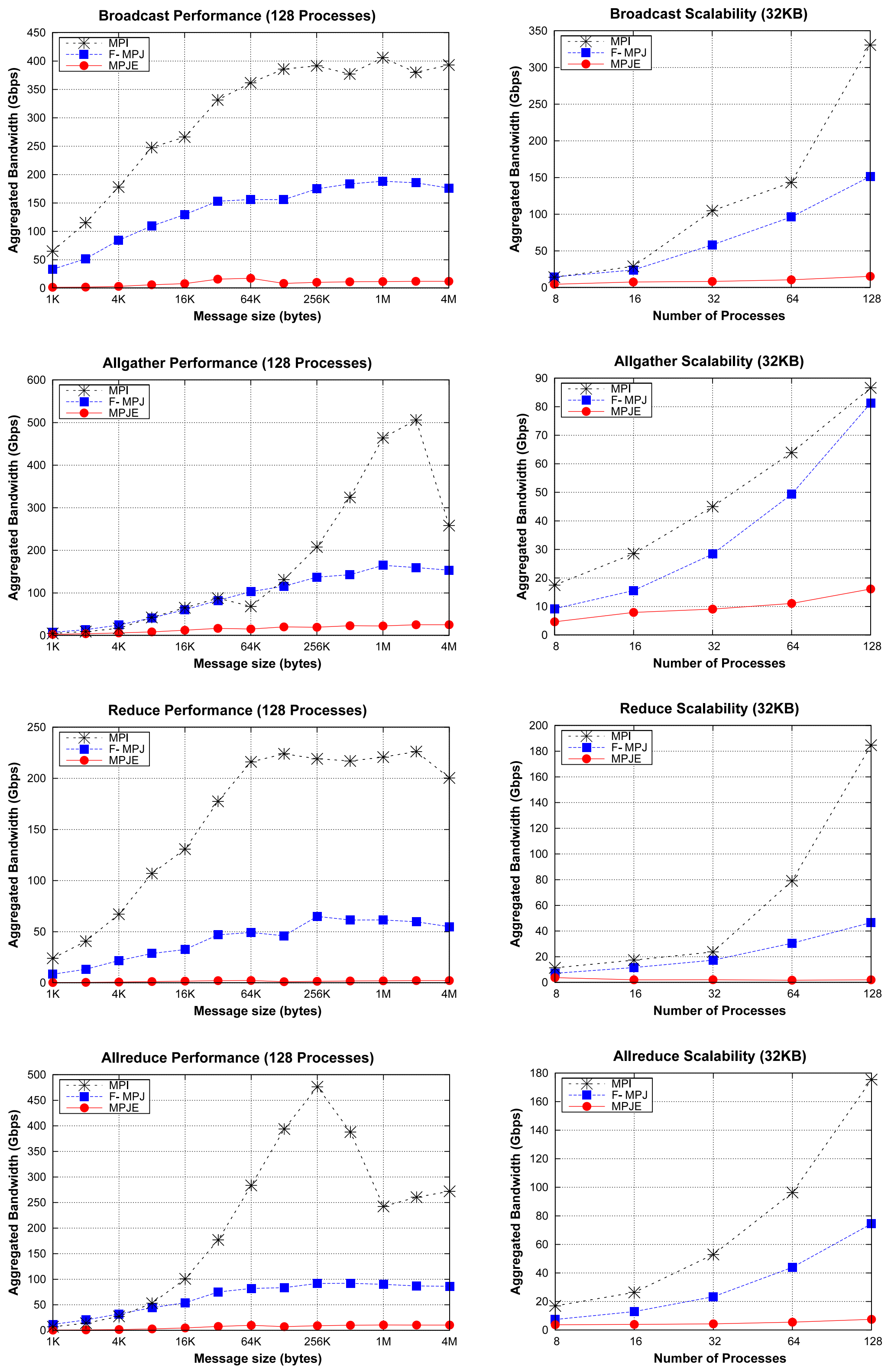


**Fig. 5.** Collective primitives performance on the InfiniBand multi-core cluster.

### 4.6. Java HPC kernel/application performance analysis

The scalability of Java for HPC has been analyzed using the NAS Parallel Benchmarks (NPB) implementation for MPJ (NPB-MPJ) [36]. The selection of the NPB has been motivated by its widespread adoption in the evaluation of languages, libraries and middleware for HPC. In fact, there are implementations of this benchmarking suite for MPI (NPB-MPI), Java Threads (NPB-JAV), OpenMP (NPB-OMP) and hybrid MPI/OpenMP (NPB-MZ). Four representative NPB codes, those with medium/high communication intensiveness (see Table 4), have been evaluated: CG (Conjugate Gradient), FT (Fourier Transform), IS (Integer Sort) and MG (Multi-Grid). Furthermore, the jGadget [55] cosmology simulation application has also been analyzed.

These MPJ codes have been selected for showing poor scalability in the related literature [1,52]. Hence, these are target codes for the analysis of the scalability of current MPJ libraries, which have been evaluated using up to 128 processes on the x86-64 cluster, and up to 256 processes on the Finis Terrae.

#### 4.6.1. Java NAS parallel benchmarks performance analysis

Figs. 6 and 7 present the NPB CG, IS, FT and MG kernel results on the x86-64 cluster and Finis Terrae, respectively, for the Class C workload in terms of MOPS (Millions of Operations Per Second) (left graphs) and their corresponding scalability, in terms of speedup (right graphs). These four kernels (CG, IS, FT and MG) have been selected as they present medium or high communication intensiveness (see Table 4). The two remaining kernels, EP and SP, were discarded due to their low communication intensiveness (see Table 4) so their results show high scalability, having limited abilities to assess the impact of multithreading and MPJ libraries on the scalability of parallel codes. The NPB implementations used are NPB-MPI and NPB-MPJ for the message-passing scalability evaluation on distributed memory and NPB-OMP and NPB-JAV for the evaluation of shared memory performance.

Although the configuration of the shared and the distributed memory scenarios are different, they share essential features such as the processor and the architecture of the system, so their results are shown together in order to ease their comparison. Thus, Fig. 6 presents NPB results of shared and distributed memory implementations measured in the x86-64 cluster. The selected NPB kernels (CG, IS, FT and MG) are implemented in the four NPB implementations evaluated, in fact the lack of some of these kernels has prevented the use of additional benchmark suites, such as the hybrid MPI/OpenMP NPB Multi-Zone (NPB-MZ), which does not implement any of these kernels.

NPB-MPI results have been obtained using the MPI library that achieves the highest performance on each system, OpenMPI on the x86-64 cluster and HP-MPI on the Finis Terrae, in both cases in combination with the Intel C/Fortran compiler. Regarding NPB-MPJ, both F-MPJ and MPJ Express have been benchmarked using the communication device that shows the best performance on InfiniBand, the interconnection network of both systems. Thus, F-MPJ has been run using its ibvdev device whereas MPJ Express relies on niodev over the IP emulation IPoIB. NPB-OMP benchmarks have been compiled with the OpenMP support included in the Intel C/Fortran compiler. Finally, NPB-JAV codes only require a standard JVM for running.

The analysis of the x86-64 cluster results (Fig. 6) first reveals that F-MPJ achieves similar performance to OpenMPI for CG when using 32 and higher number of cores, showing higher speedups than the MPI library in this case. As this kernel only includes point-to-point communication primitives, F-MPJ takes advantage of obtaining similar point-to-point performance to MPI. However, MPJ Express and the Java threads implementations present poor scalability from 8 cores. On the one hand, the poor speedups of MPJ Express are direct consequence of the use of sockets and IPoIB in its communication layer. On the other hand, the poor performance of the NPB-JAV kernels is motivated by their inefficient implementation. In fact, the evaluated codes obtain lower performance on a single core than the MPI, OpenMP and MPJ kernels, except for NPB-JAV MG, which outperforms NPB-MPJ MG (see in Section 4.3 the left graph in Fig. 3). The reduced performance of NPB-JAV kernels on a single core, which can incur up to 50% performance overhead compared to NPB-MPJ codes, determines the lower overall performance in terms of MOPS.

Additionally, the NPB shared memory implementations, using OpenMP and Java Threads, present poorer scalability on the x86_64 cluster than distributed memory (message-passing) implementations, except for NPB-OMP IS. The main reason behind this behavior is the memory access overhead when running 8 and even 16 threads on 8 physical cores, which thanks to hyperthreading are able to run up to 16 threads simultaneously. Thus, the main performance bottleneck for these shared memory implementations is the access to memory, which limits their scalability and prevents taking advantage of enabling hyperthreading.

Regarding FT results, although F-MPJ scalability is higher than MPI (F-MPJ speedup is about 50 on 128 cores whereas the MPI one is below 36), this is not enough for achieving similar performance in terms of MOPS. In this case MPJ performance is limited by its poor performance on one core, which is 54% of the MPI performance (see in Section 4.3 the left graph in Fig. 3). Moreover, the scalability of this kernel relies on the performance of the Alltoall collective, which has not prevented F-MPJ scalability. As for CG, MPJ Express and the shared memory NPB codes show poor performance, although NPB-JAV FT presents a slightly performance benefit when resorting to hyperthreading, probably due to its poor performance on one core, which is below 30% of the NPB-MPI FT result. In fact, a longer runtime reduces the impact of communications and memory bottlenecks in the scalability of parallel codes.

The significant communication intensiveness of IS, the highest among the evaluated kernels, reduces the observed speedups, which are below 20 on 128 cores. On the one hand, the message-passing implementations of this kernel rely

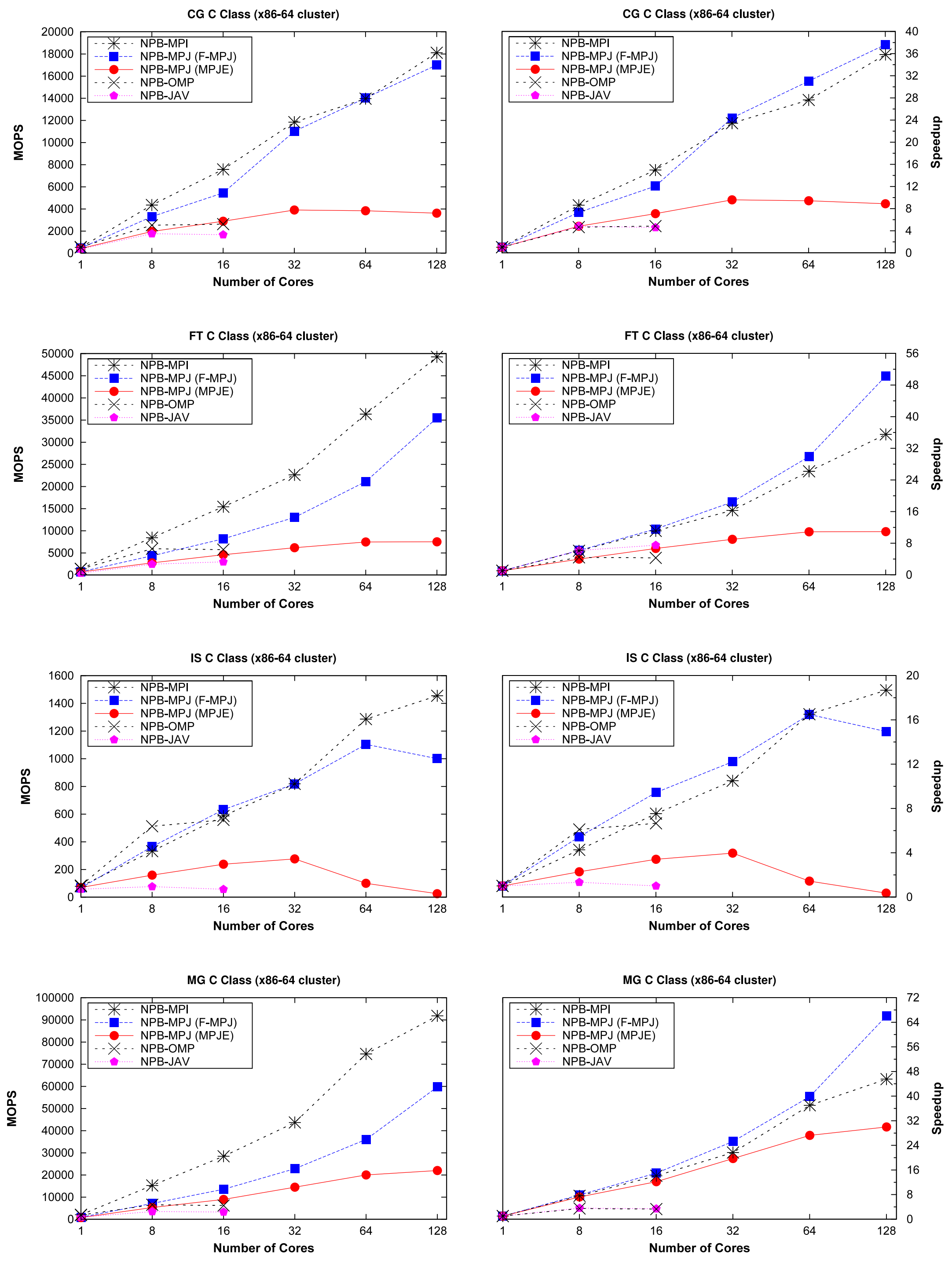


**Fig. 6.** NPB Class C results on the x86-64 cluster.

heavily on Alltoall and Allreduce primitives, whose overhead is the main performance penalty. In fact, F-MPJ scalability drops from 64 cores (MPJ Express from 32 cores), whereas MPI shows poor scalability from 64 cores (the performance comparison between 64 and 128 cores shows that the use of the additional 64 cores only increases the speedup in 3 units, from 16 to 19). On the other hand, OpenMP IS obtains the best results on 8 cores, showing a high parallel efficiency, and even takes advantage of the use of hyperthreading. However, the implementation of IS using Java threads shows very poor scalability, with speedups below 2.

The highest MG performance in terms of MOPS has been obtained with MPI, followed at a significant distance by F-MPJ although this Java library shows higher speedups, especially on 128 cores. The reason, as for FT, is that MPJ performance is limited by its poor performance on one core, which is 55% of the MPI performance (see in Section 4.3 the left graph in Fig. 3). The longer MPJ runtime contributes to achieve high speedups in MG, trading off the bottleneck that represents the extensive use by this kernel of Allreduce, a collective whose performance is lower for MPJ than for MPI. In fact, the message-passing implementations of this kernel, both MPI and MPJ, present relatively good scalability, even for MPJ Express which achieves speedups around 30 on 64 and 128 cores. Nevertheless, the shared memory codes show little speedups, below 4 on 8 cores.

Fig. 7 shows the Finis Terrae results, where the message-passing kernel implementations, NPB-MPI and NPB-MPJ, have been run on the rx7640 nodes of this supercomputer, using 8 cores per node and up to 32 nodes (hence up to 256 cores), whereas the shared memory results (NPB-OMP and NPB-JAV) have been obtained from the HP Integrity Superdome using up to 128 cores. Although the results have been obtained using two different hardware configurations, both subsystems share the same features but the memory architecture, which is distributed in rx7640 nodes and shared in the Integrity Superdome, as presented in Section 4.1.

The analysis of the Finis Terrae results (Fig. 7) shows that the best performer is OpenMP, showing significantly higher MOPS than the other implementations, except for MG where it is outperformed by MPI. Nevertheless, OpenMP suffers scalability losses from 64 cores due to the access to remote cells and the relative poor bidirectional traffic performance in the cell controller (the Integrity Superdome is a ccNUMA system which consists of 16 cells, each one with 4 dual-core processors and 64 Gbytes memory, interconnected through a crossbar network) [56]. The high performance of OpenMP contrasts with the poor results in terms of MOPS of NPB-JAV, although this is motivated by its poor performance on one core, which is usually an order of magnitude lower than MPI (Intel Compiler) performance (see in Section 4.3 the right graph in Fig. 3). Although this poor runtime favors the obtaining of high scalability, in fact NPB-JAV obtains speedups above 30 for CG and FT, this is not enough to bridge the gap with OpenMP results as NPB-OMP codes achieves even higher speedups, except for FT. Furthermore, NPB-JAV results are significantly poorer than those of NPB-MPJ (around 2–3 times lower), except for MG, which confirms the inefficiency of this Java threads implementation.

The performance results of the message-passing codes, NPB-MPI and NPB-MPJ, are between NPB-OMP kernels and the shared memory implementations, except for NPB-MPI MG, which is the best performer for MG kernel. Nevertheless, there are significant differences among the libraries been used. Thus, MPJ Express presents modest speedups, below 30, due to the use of a sockets-based (niodev) communication device over the IP emulation IPoIB. This limitation is overcome in F-MPJ, relying more directly on IBV. Thus, F-MPJ is able to achieve the highest speedups, motivated in part by the longer runtimes on one core (see in Section 4.3 the right graph in Fig. 3) which favor this scalability (a heavy workload reduces the impact of communications on the overall performance scalability). The high speedups of F-MPJ, which are significantly higher than those of MPI (e.g., up to 7 times higher in CG), allow F-MPJ to bridge the gap between Java and natively compiled languages in HPC. In fact, F-MPJ performance results for CG and FT on 256 are close to those of MPI, although their performance on one core is around 7 and 4 times lower than MPI results for CG and FT, respectively.

The analysis of these NPB experimental results show that the performance of MPJ libraries heavily depends on their InfiniBand support. Thus, F-MPJ, which relies directly on IBV, outperforms significantly MPJ Express, whose socket-based communication device runs on IPoIB, obtaining relatively low performance, especially in terms of start-up latency. Furthermore, NPB-MPJ kernels have revealed to be the most efficient Java implementation, significantly outperforming Java threads implementations, both in terms of performance on one core and scalability. Moreover, the comparative evaluation of NPB-MPJ and NPB-MPI results reveals that efficient MPJ libraries can help to bridge the gap between Java and native code performance in HPC. Finally, the evaluated libraries have shown higher speedups on Finis Terrae than on the x86-64 cluster. The reason behind this behavior is that the obtaining of poorer performance on one core allows for higher scalability given the same interconnection technology (both systems use 16 Gbps InfiniBand DDR networks). Thus, NPB-MPJ kernels on the Finis Terrae, showing some of the poorest performance on one core, are able to achieve speedups of up to 175 on 256 cores, whereas NPB-MPI scalability on the x86-64 cluster is always below a speedup of 50. Nevertheless, NPB-MPI on the x86-64 cluster shows the highest performance in terms of MOPS, outperforming NPB-MPI results on the Finis Terrae, which has double the number of available cores (256 cores available on the Finis Terrae vs. 128 cores available on the x86-64 cluster).

#### *4.6.2. Performance analysis of the jGadget application*

The jGadget [55] application is the MPJ implementation of Gadget [57], a popular cosmology simulation code initially implemented in C and parallelized using MPI that is used to study a large variety of problems like colliding and merging galaxies or the formation of large-scale structures. The parallelization strategy, both with MPI and MPJ, is an irregular and dynamically adjusted domain decomposition, with copious communication between processes. jGadget has been selected as representative Java HPC application as its performance has been previously analyzed [52] for their Java (MPJ) and C (MPI) implementations, as well as for its communication intensiveness and its popularity.

Fig. 8 presents jGadget and Gadget performance results on the x86-64 cluster and the Finis Terrae for a galaxy cluster formation simulation with 2 million particles in the system (simulation available within the examples of Gadget software

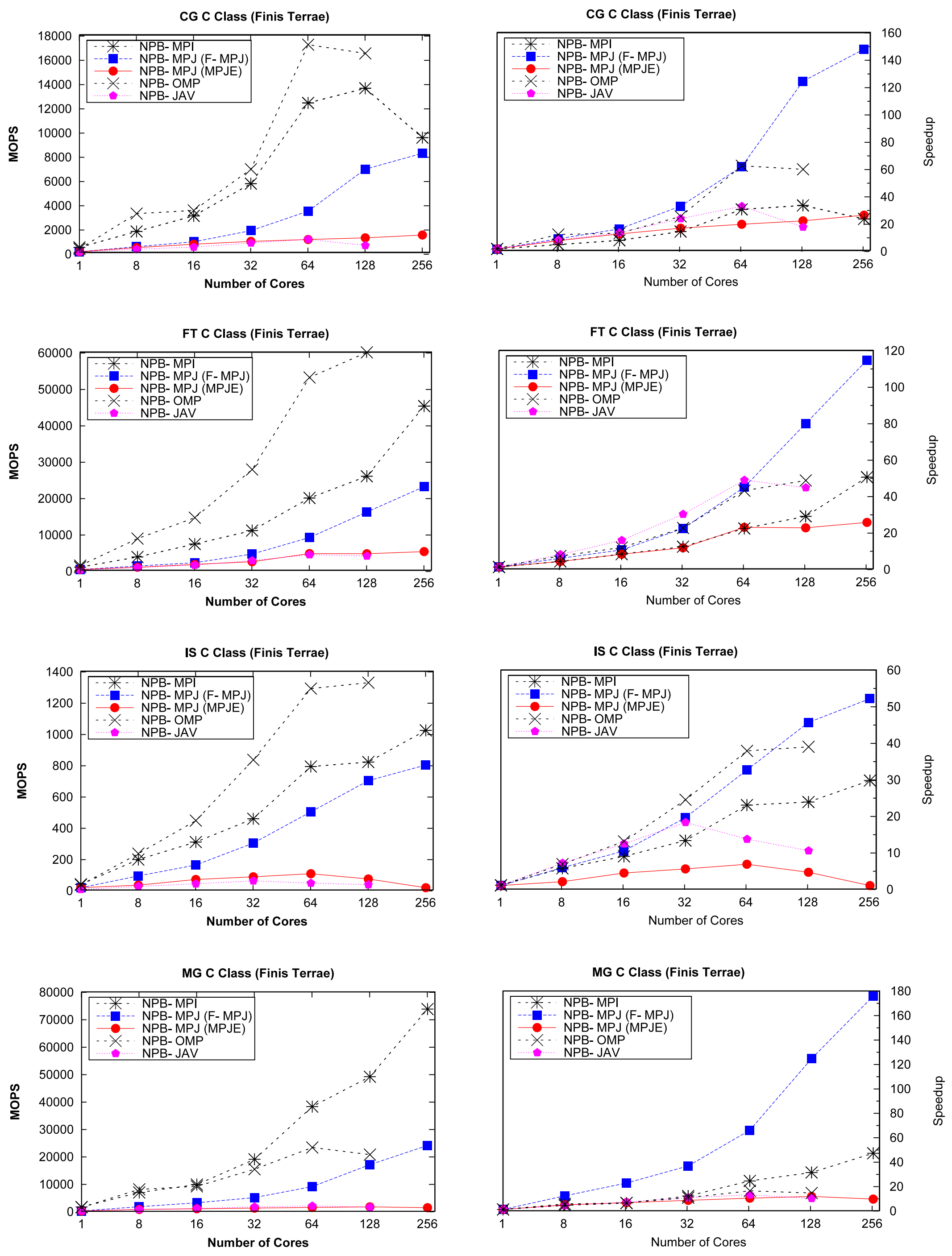


**Fig. 7.** NPB Class C results on Finis Terrae.

bundle). As Gadget is a communication-intensive application, with significant collective operations overhead, its scalability is modest, obtaining speedups of up to 48 on 128 cores of the x86-64 cluster and speedups of up to 57 on 256 cores of the Finis Terrae. Here F-MPJ achieves generally the highest speedups, followed closely by MPI, except from 64 cores on the

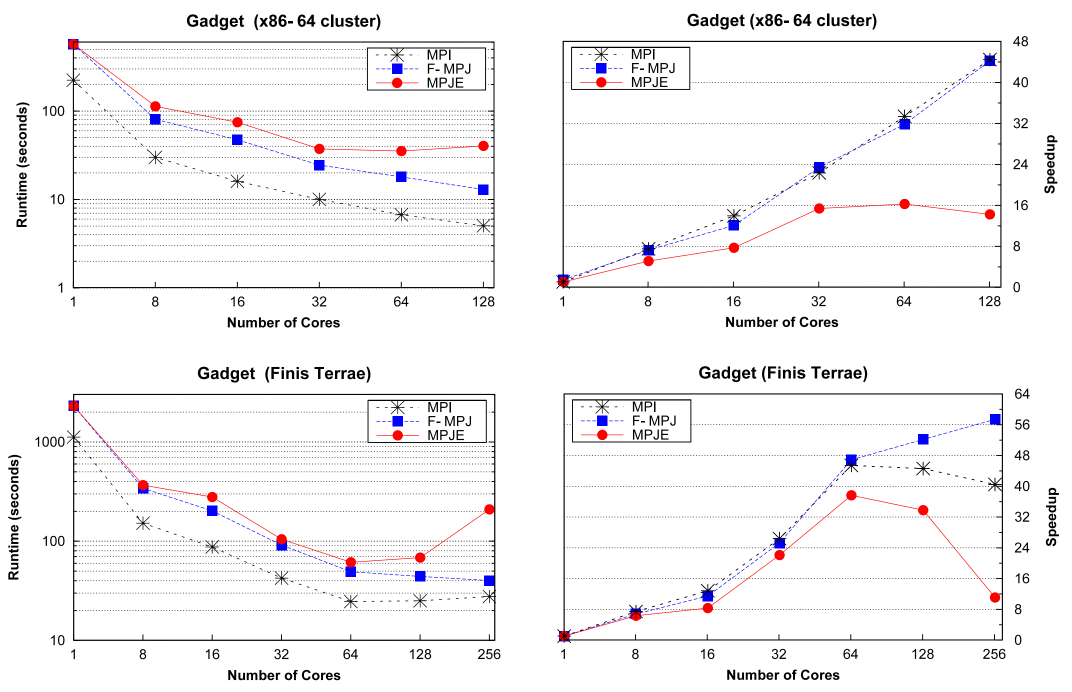


**Fig. 8.** Gadget runtime and scalability on the x86-64 cluster and the Finis Terrae supercomputer.

Finis Terrae where MPI loses performance. This slowdown is shared with MPJ Express, which shows its highest performance on 64 cores for both systems. Nevertheless, MPJ Express speedups on the Finis Terrae are much higher (up to 37) than on the x86-64 cluster (only up to 16), something motivated by the different runtime of the application on the x86-64 cluster and the Finis Terrae. In fact, MPI Gadget presents numerous library dependences, such as FFTW-MPI, Hierarchical Data Format (HDF) support, and the numerical GNU Scientific Library (GSL), which are not fully optimized for this system, thus increasing significantly its runtime. An example of inefficiency is that GSL shows poor performance on the Finis Terrae. Here the use of Intel Math Kernel Library (MKL) would show higher performance but the support for this numerical library is not implemented in Gadget. As a consequence of this jGadget performs better, compared in relative terms with MPI, on the Finis Terrae (only 2 times slower than MPI) than on the x86-64 cluster (3 times slower than MPI), although the performance of Java on IA64 architectures is quite poor.

Moreover, the performance gap between Gadget and jGadget is motivated by the poor performance of the numerical methods included in jGadget, which consist of a translation of the GSL functions invoked in the Gadget source code, without relying on external numerical libraries. The use of an efficient Java numerical library [58], comparable in performance to Fortran numerical codes, would have improved the performance of jGadget. The development of such a library is still an ongoing effort, although it started a decade ago when it was demonstrated that Java was able to compete with Fortran in high performance numerical computing [59,60]. In the last years a few projects are being actively developed [61], such as Universal Java Matrix Package (UJMP) [62], Efficient Java Matrix Library (EJML) [63], Matrix Toolkit Java (MTJ) [64] and jblas [65], which are replacing more traditional frameworks such as JAMA [66]. Furthermore, a recent evaluation of Java for numerical computing [67] has shown that the performance of Java applications can be significantly enhanced by delegating numerically intensive tasks to native libraries (e.g., Intel MKL) which supports the development of efficient high performance numerical applications in Java.

## 5. Conclusions

This paper has analyzed the current state of Java for HPC, both for shared and distributed memory programming, showing an important number of past and present projects which are the result of the sustained interest in the use of Java for HPC. Nevertheless, most of these projects are restricted to experimental environments, which prevents their general adoption in this field. However, the analysis of the existing programming options and available libraries in Java for HPC, together with the presentation in this paper of our current research efforts in the improvement of the scalability of our Java message-passing library, F-MPJ, would definitively contribute to boost the embracement of Java in HPC.

Additionally, Java lacks thorough and up-to-date evaluations of their performance in HPC. In order to overcome this issue this paper presents the performance evaluation of current Java HPC solutions and research developments on two shared memory environments and two InfiniBand multi-core clusters. The main conclusion of the analysis of these results is that Java can achieve almost similar performance to natively compiled languages, both for sequential and parallel applications, being an alternative for HPC programming. In fact, the performance overhead that Java may impose is a reasonable trade-off for the appealing features that this language provides for parallel programming multi-core architectures. Furthermore, the recent advances in the efficient support of Java communications on shared memory and low-latency networks are bridging the performance gap between Java and more traditional HPC languages.

Finally, the active research efforts in this area are expected to bring in the next future new developments that will continue rising the interest of both industry and academia and increasing the benefits of the adoption of Java for HPC.

## Acknowledgements

This work was funded by the Ministry of Science and Innovation of Spain under Project TIN2010-16735 and an FPU grant AP2009-2112. We gratefully thank CESGA (Galicia Supercomputing Center, Santiago de Compostela, Spain) for providing access to the Finis Terrae supercomputer.

## References

[1] G.L. Taboada, J. Touriño, R. Doallo, Java for high performance computing: assessment of current research and practice, in: Proc. 7th Intl. Conference on the Principles and Practice of Programming in Java, PPPJ'09, Calgary, Alberta, Canada, 2009, pp. 30–39.
[2] B. Amedro, D. Caromel, F. Huet, V. Bodnartchouk, C. Delbé, G.L. Taboada, ProActive: using a Java middleware for HPC design, implementation and benchmarks, International Journal of Computers and Communications 3 (3) (2009) 49–57.
[3] J.J. Dongarra, D. Gannon, G. Fox, K. Kennedy, The impact of multicore on computational science software, CTWatch Quarterly 3 (1) (2007) 1–10.
[4] A. Kaminsky, Parallel Java: a unified API for shared memory and cluster parallel programming in 100% Java, in: Proc. 9th Intl. Workshop on Java and Components for Parallelism, Distribution and Concurrency, IWJacPDC'07, Long Beach, CA, USA, 2007, p. 196a (8 pages).
[5] M.E. Kambites, J. Obdrzálek, J.M. Bull, An OpenMP-like interface for parallel programming in Java, Concurrency and Computation: Practice and Experience 13 (8–9) (2001) 793–814.
[6] M. Klemm, M. Bezold, R. Veldema, M. Philippsen, JaMP: an implementation of OpenMP for a Java DSM, Concurrency and Computation: Practice and Experience 19 (18) (2007) 2333–2352.
[7] A. Shafi, B. Carpenter, M. Baker, Nested parallelism for multi-core HPC systems using Java, Journal of Parallel and Distributed Computing 69 (6) (2009) 532–545.
[8] R. Veldema, R.F.H. Hofman, R. Bhoedjang, H.E. Bal, Run-time optimizations for a Java DSM implementation, Concurrency and Computation: Practice and Experience 15 (3–5) (2003) 299–316.
[9] K.A. Yelick, et al., Titanium: a high-performance Java dialect, Concurrency — Practice and Experience 10 (11–13) (1998) 825–836.
[10] K. Datta, D. Bonachea, K.A. Yelick, Titanium performance and potential: an NPB experimental study, in: Proc. 18th Intl. Workshop on Languages and Compilers for Parallel Computing, LCPC'05, in: LNCS, vol. 4339, Hawthorne, NY, USA, 2005, pp. 200–214.
[11] G.L. Taboada, J. Touriño, R. Doallo, Java Fast Sockets: enabling high-speed Java communications on high performance clusters, Computer Communications 31 (17) (2008) 4049–4059.
[12] R.V.v. Nieuwpoort, J. Maassen, G. Wrzesinska, R. Hofman, C. Jacobs, T. Kielmann, H.E. Bal, Ibis: a flexible and efficient Java-based Grid programming environment, Concurrency and Computation: Practice and Experience 17 (7–8) (2005) 1079–1107.
[13] L. Baduel, F. Baude, D. Caromel, Object-oriented SPMD, in: Proc. 5th IEEE Intl. Symposium on Cluster Computing and the Grid, CCGrid'05, Cardiff, UK, 2005, pp. 824–831.
[14] M. Philippsen, B. Haumacher, C. Nester, More efficient serialization and RMI for Java, Concurrency: Practice and Experience 12 (7) (2000) 495–518.
[15] D. Kurzyniec, T. Wrzosek, V. Sunderam, A. Slominski, RMIX: a multiprotocol RMI framework for Java, in: Proc. 5th Intl. Workshop on Java for Parallel and Distributed Computing, IWJPDC'03, Nice, France, 2003, p. 140 (7 pages).
[16] J. Maassen, R.V.v. Nieuwpoort, R. Veldema, H.E. Bal, T. Kielmann, C. Jacobs, R. Hofman, Efficient Java RMI for parallel programming, ACM Transactions on Programming Languages and Systems 23 (6) (2001) 747–775.
[17] G.L. Taboada, C. Teijeiro, J. Touriño, High performance Java remote method invocation for parallel computing on clusters, in: Proc. 12th IEEE Symposium on Computers and Communications, ISCC'07, Aveiro, Portugal, 2007, pp. 233–239.
[18] G.L. Taboada, J. Touriño, R. Doallo, Performance analysis of Java message-passing libraries on Fast Ethernet, Myrinet and SCI clusters, in: Proc. 5th IEEE Intl. Conf. on Cluster Computing, CLUSTER'03, Hong Kong, China, 2003, pp. 118–126.
[19] B. Carpenter, G. Fox, S.-H. Ko, S. Lim, mpiJava 1.2: API Specification, http://www.hpjava.org/reports/mpiJava-spec/mpiJava-spec/mpiJava-spec.html [Last visited: May 2011].
[20] B. Carpenter, V. Getov, G. Judd, A. Skjellum, G. Fox, MPJ: MPI-like message passing for Java, Concurrency: Practice and Experience 12 (11) (2000) 1019–1038.
[21] Java grande forum, http://www.javagrande.org, [Last visited: May 2011].
[22] M. Baker, B. Carpenter, G. Fox, S. Ko, S. Lim, mpiJava: an object-oriented Java interface to MPI, in: Proc. 1st Intl. Workshop on Java for Parallel and Distributed Computing, IWJPDC'99, in: LNCS, vol. 1586, San Juan, Puerto Rico, 1999, pp. 748–762.
[23] B. Pugh, J. Spacco, MPJava: high-performance message passing in Java using Java.nio, in: Proc. 16th Intl. Workshop on Languages and Compilers for Parallel Computing, LCPC'03, in: LNCS, vol. 2958, College Station, TX, USA, 2003, pp. 323–339.
[24] B.-Y. Zhang, G.-W. Yang, W.-M. Zheng, Jcluster: an efficient Java parallel environment on a large-scale heterogeneous cluster, Concurrency and Computation: Practice and Experience 18 (12) (2006) 1541–1557.
[25] S. Genaud, C. Rattanapoka, P2P-MPI: a peer-to-peer framework for robust execution of message passing parallel programs, Journal of Grid Computing 5 (1) (2007) 27–42.
[26] M. Bornemann, R.V.v. Nieuwpoort, T. Kielmann, MPJ/Ibis: a flexible and efficient message passing platform for Java, in: Proc. 12th European PVM/MPI Users' Group Meeting, EuroPVM/MPI'05, Sorrento, Italy, LNCS, vol. 3666, 2005, pp. 217–224.
[27] S. Bang, J. Ahn, Implementation and performance evaluation of socket and RMI based Java message passing systems, in: Proc. 5th ACIS Intl. Conf. on Software Engineering Research, Management and Applications, SERA'07, Busan, Korea, 2007, pp. 153–159.
[28] G.L. Taboada, J. Touriño, R. Doallo, F-MPJ: scalable Java message-passing communications on parallel systems, Journal of Supercomputing (2011) doi:10.1007/s11227-009-0270-0.
[29] G.L. Taboada, S. Ramos, J. Touriño, R. Doallo, Design of efficient Java message-passing collectives on multi-core clusters, Journal of Supercomputing 55 (2) (2011) 126–154.
[30] A. Shafi, J. Manzoor, K. Hameed, B. Carpenter, M. Baker, Multicore-enabling the MPJ Express messaging library, in: Proc. 8th Intl. Conference on the Principles and Practice of Programming in Java, PPPJ'10, Vienna, Austria, 2010, pp. 49–58.

[31] L.A. Barchet-Estefanel, G. Mounié, Fast tuning of intra-cluster collective communications, in: Proc. 11th European PVM/MPI Users' Group Meeting, EuroPVM/MPI'04, Budapest, Hungary, LNCS, vol. 3241, 2004, pp. 28–35.
[32] E. Chan, M. Heimlich, A. Purkayastha, R.A. van de Geijn, Collective communication: theory, practice, and experience, Concurrency and Computation: Practice and Experience 19 (13) (2007) 1749–1783.
[33] R. Thakur, R. Rabenseifner, W. Gropp, Optimization of collective communication operations in MPICH, International Journal of High Performance Computing Applications 19 (1) (2005) 49–66.
[34] S.S. Vadhiyar, G.E. Fagg, J.J. Dongarra, Towards an accurate model for collective communications, International Journal of High Performance Computing Applications 18 (1) (2004) 159–167.
[35] J.M. Bull, L.A. Smith, M.D. Westhead, D.S. Henty, R.A. Davey, A benchmark suite for high performance Java, Concurrency: Practice and Experience 12 (6) (2000) 375–388.
[36] D.A. Mallón, G.L. Taboada, J. Touriño, R. Doallo, NPB-MPJ: NAS parallel benchmarks implementation for message-passing in Java, in: Proc. 17th Euromicro Intl. Conf. on Parallel, Distributed, and Network-Based Processing (PDP'09), Weimar, Germany, 2009, pp. 181–190.
[37] P. Charles, C. Grothoff, V.A. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, V. Sarkar, X10: an object-oriented approach to non-uniform cluster computing, in: Proc. 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA'05, San Diego, CA, USA, 2005, pp. 519–538.
[38] X10: Performance and productivity at scale, http://x10plus.cloudaccess.net/ [Last visited: May 2011].
[39] Habanero Java, http://habanero.rice.edu/hj.html [Last visited: May 2011].
[40] J. Shirako, H. Kasahara, V. Sarkar, Language extensions in support of compiler parallelization, in: Proc. 20th Intl. Workshop on Languages and Compilers for Parallel Computing, LCPC'07, Urbana, IL, USA, 2007, pp. 78–94.
[41] Y. Yan, M. Grossman, V. Sarkar, JCUDA: a programmer-friendly interface for accelerating Java programs with CUDA, in: Proc. 15th Intl. European Conference on Parallel and Distributed Computing, Euro-Par'09, Delft, The Netherlands, 2009, pp. 887–899.
[42] jcuda.org, http://jcuda.org [Last visited: May 2011].
[43] jCUDA, http://hoopoe-cloud.com/Solutions/jCUDA/Default.aspx [Last visited: May 2011].
[44] JaCuda, http://jacuda.sourceforge.net [Last visited: May 2011].
[45] Jacuzzi, http://sourceforge.net/apps/wordpress/jacuzzi [Last visited: May 2011].
[46] java-gpu, http://code.google.com/p/java-gpu [Last visited: May 2011].
[47] jocl.org, http://jocl.org [Last visited: May 2011].
[48] JavaCL, http://code.google.com/p/javacl [Last visited: May 2011].
[49] JogAmp, http://jogamp.org [Last visited: May 2011].
[50] G. Dotzler, R. Veldema, M. Klemm, JCudaMP: OpenMP/Java on CUDA, in: Proc. 3rd Intl. Workshop on Multicore Software Engineering, IWMSE'10, Cape Town, South Africa, 2010, pp. 10–17.
[51] A. Leung, O. Lhoták, G. Lashari, Parallel execution of Java loops on graphics processing units, Science of Computer Programming (2011) doi:10.1016/j.scico.2011.06.004.
[52] A. Shafi, B. Carpenter, M. Baker, A. Hussain, A comparative study of Java and C performance in two large-scale parallel applications, Concurrency and Computation: Practice and Experience 21 (15) (2009) 1882–1906.
[53] Finis Terrae Supercomputer, Galicia Supercomputing Center, CESGA, http://www.top500.org/system/9156 [Last visited: May 2011].
[54] A. Georges, D. Buytaert, L. Eeckhout, Statistically rigorous Java performance evaluation, in: Proc. 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA'07, Montreal, Quebec, Canada, 2007, pp. 57–76.
[55] M. Baker, B. Carpenter, A. Shafi, MPJ Express meets Gadget: towards a Java code for cosmological simulations, in: Proc. 13th European PVM/MPI Users' Group Meeting, EuroPVM/MPI'06, Bonn, Germany, 2006, pp. 358–365.
[56] D.A. Mallón, G. Taboada, C. Teijeiro, J. Touriño, B. Fraguela, A. Gómez, R. Doallo, J. Mouriño, Performance evaluation of MPI, UPC and OpenMP on multicore architectures, in: Proc. 16th European PVM/MPI Users' Group Meeting, EuroPVM/MPI'09, Espoo, Finland, 2009, pp. 174–184.
[57] V. Springel, The cosmological simulation code GADGET-2, Monthly Notices of the Royal Astronomical Society 364 (4) (2005) 1105–1134.
[58] JavaGrande JavaNumerics, http://math.nist.gov/javanumerics/ [Last visited: May 2011].
[59] R.F. Boisvert, J.J. Dongarra, R. Pozo, K.A. Remington, G.W. Stewart, Developing numerical libraries in Java, Concurrency: Practice and Experience 10 (11–13) (1998) 1117–1129.
[60] J.E. Moreira, S.P. Midkiff, M. Gupta, P.V. Artigas, M. Snir, R.D. Lawrence, Java programming for high-performance numerical computing, IBM Systems Journal 39 (1) (2000) 21–56.
[61] H. Arndt, M. Bundschus, A. Naegele, Towards a next-generation matrix library for Java, in: Proc. 33rd Annual IEEE Intl. Computer Software and Applications Conference, COMPSAC'09, Seattle, WA, USA, 2009, pp. 460–467.
[62] Universal Java Matrix Package (UJMP), http://www.ujmp.org [Last visited: May 2011].
[63] Efficient Java Matrix Library (EJML), http://code.google.com/p/efficient-java-matrix-library/ [Last visited: May 2011].
[64] Matrix Toolkits Java (MTJ), http://code.google.com/p/matrix-toolkits-java/ [Last visited: May 2011].
[65] Linear Algebra for Java (jblas), http://jblas.org/ [Last visited: May 2011].
[66] JAMA: A Java Matrix Package, http://math.nist.gov/javanumerics/jama [Last visited: May 2011].
[67] M. Baitsch, N. Li, D. Hartmann, A toolkit for efficient numerical applications in Java, Advances in Engineering Software 41 (1) (2010) 75–83.

# Part II

# Evaluation of Communication Middleware for HPC on a Public Cloud Infrastructure

# Chapter 6

# Evaluation of Messaging Middleware for Cloud Computing

The content of this chapter corresponds to the following journal paper:

**- Title:** Evaluation of messaging middleware for high-performance cloud computing

**- Authors:** Roberto R. Expósito, Guillermo L. Taboada, Sabela Ramos, Juan Touriño, Ramón Doallo

**- Journal:** Personal and Ubiquitous Computing

**- Editorial:** Springer Science+Business Media

**- ISSN:** 1617-4909

**- Year:** 2013

**- Volume(Number):Pages:** 17(8):1709–1719

**- DOI:** 10.1007/s00779-012-0605-3

The final publication is available at http://link.springer.com/article/10.1007%2Fs00779-012-0605-3. A copy of the accepted paper has been included next.

# Evaluation of messaging middleware for high-performance cloud computing

**Roberto R. Expósito · Guillermo L. Taboada · Sabela Ramos · Juan Touriño · Ramón Doallo**

**Abstract** Cloud computing is posing several challenges, such as security, fault tolerance, access interface singularity, and network constraints, both in terms of latency and bandwidth. In this scenario, the performance of communications depends both on the network fabric and its efficient support in virtualized environments, which ultimately determines the overall system performance. To solve the current network constraints in cloud services, their providers are deploying high-speed networks, such as 10 Gigabit Ethernet. This paper presents an evaluation of high-performance computing message-passing middleware on a cloud computing infrastructure, Amazon EC2 cluster compute instances, equipped with 10 Gigabit Ethernet. The analysis of the experimental results, confronted with a similar testbed, has shown the significant impact that virtualized environments still have on communication performance, which demands more efficient communication middleware support to get over the current cloud network limitations.

R. R. Expósito (✉) · G. L. Taboada · S. Ramos · J. Touriño · R. Doallo
Department of Electronics and Systems,
University of A Coruña, A Coruña, Spain
e-mail: rreye@udc.es

G. L. Taboada
e-mail: taboada@udc.es

S. Ramos
e-mail: sramos@udc.es

J. Touriño
e-mail: juan@udc.es

R. Doallo
e-mail: doallo@udc.es

**Keywords** Cloud computing · High-performance computing · Virtualization · 10 Gigabit ethernet · Message-passing middleware · Performance evaluation

## 1 Introduction

Cloud computing is a model that enables convenient, on demand and self-service access to a shared pool of highly scalable, abstracted infrastructure that hosts applications, which are billed by consumption. This computing paradigm is changing rapidly the way enterprise computing is provisioned and managed, thanks to the commoditization of computing resources (e.g., networks, servers, storage and applications) which provide cost-effective solutions and efficient server virtualization [5]. Moreover, cloud computing technologies can be useful in a wide range of applications such as email, file storage or document clustering [39], among other domains [20]. However, this model is posing several challenges, such as security [35], heterogeneity of the cloud management frameworks and handling network constraints, both in terms of latency (the cost of sending a message with minimal size through the network) and bandwidth, that limit the scalability of the cloud resources.

In cloud computing, the performance of communications depends both on the network fabric and its efficient support in cloud middleware, which ultimately determines the overall system performance. Cloud infrastructures typically relied on a virtualized access to Gigabit Ethernet and the use of TCP/IP stack, a combination that provides poor performance [33], especially when the underlying infrastructure consists of systems with an increasing number of cores per processor due to the poor ratio between CPU power and network performance. To solve these network constraints in cloud services, their providers are deploying high-speed networks (e.g., 10 Gigabit Ethernet and InfiniBand).

High-speed networks have been traditionally deployed in high-performance computing (HPC) environments, where message-passing middleware is the preferred choice for supporting communications across distributed memory systems. MPI [1] is the standard interface for programming message-passing applications in languages compiled to native code (e.g., C/C++ and Fortran), whereas MPJ [7, 31] is the messaging middleware for Java applications. These HPC applications, whose scalability depends on low-latency communications [37], generally achieve good performance on clusters with high-speed networks, whereas they suffer significant performance bottlenecks on virtualized environments, especially networking overheads, at public cloud infrastructures.

Amazon Elastic Compute Cloud (Amazon EC2) [21] provides users with access to on demand computational resources to run their applications. EC2 allows scalable deployment of applications by providing a Web service through which a user can boot an Amazon Machine Image (AMI) to create a custom Virtual Machine (a VM or "instance"). Cluster compute instances [22], a resource introduced in July 2010, provide the most powerful CPU resources with increased network performance, which is intended to be well suited for HPC applications and other demanding network-bound applications. This is particularly valuable for those applications that rely on messaging middleware like MPI for tightly coupled inter-node communication.

This paper presents an evaluation of HPC message-passing middleware on a cloud computing infrastructure, Amazon EC2 cluster compute instances, with a high-speed network, 10 Gigabit Ethernet, in order to assess their suitability for HPC. Nevertheless, the experimental results have shown several performance penalties, such as the lack of efficient network virtualization support and a proper VM-aware middleware adapted to cloud environments.

The structure of this paper is as follows: Sect. 2 introduces the related work. Section 3 describes the network support in virtualized environments, the building block in cloud infrastructures. Section 4 introduces the message-passing middleware considered in this work. Section 5 analyzes the performance results of the selected middleware on HPC cluster instances of Amazon EC2 cloud, compared to our private cloud testbed with a similar configuration. These results have been obtained from a micro-benchmarking of point-to-point primitives, as well as an application benchmarking in order to analyze the scalability of HPC applications on cloud services. Section 6 summarizes our concluding remarks and future work.

## 2 Related work

Typically, computationally intensive codes present little overhead when running on virtualized environments, whereas I/O bound applications, especially network intensive ones, suffer significant performance losses [14]. Thus, message-passing applications whose scalability heavily depends on start-up latency performance were initially highly inefficient in such virtualized environments.

There have been many studies of virtualization techniques in the literature, including performance enhancements focused on reducing this I/O overhead. Paravirtualization [36] was introduced in order to reduce the performance overhead associated with emulated I/O access to virtual devices. Liu et al. [19] describe the I/O bypass of the virtual machine monitor (VMM) or hypervisor, using InfiniBand architecture, extending the OS-bypass mechanism to high-speed interconnects. The use of this technique has shown that Xen hypervisor [8] is capable of near-native bandwidth and latency, although it has not been officially integrated in Xen so far. Nanos et al. [27] developed Myrixen, a thin split driver layer on top of the Myrinet Express (MX) driver to support message-passing in Xen VMs over the wire protocols in Myri-10G infrastructures.

A VM-aware MPI library was implemented in [15], reducing the communication overhead for HPC applications by supporting shared memory transfers among VMs in the same physical host. Mansley et al. [24] developed a direct data path between the VM and the network using Solarflare Ethernet NICs, but its operation is restricted to these devices. Raj et al. [28] describe network processor-based self via specialized network interface cards to minimize network overhead.

Hybrid computing concept was studied in [25]. They have examined how cloud computing can be best combined with traditional HPC approaches, proposing a hybrid infrastructure for the predictable execution of complex scientific workloads across a hierarchy of internal and external resources. They presented the `Elastic Cluster` as a unified model of managed HPC and cloud resources.

Additionally, some works have already evaluated the performance of cloud computing services, such as Amazon EC2. Wang et al. [34] present a quantitative study of the end-to-end networking performance among Amazon EC2 medium instances, and they observed unstable TCP/UDP throughput caused by virtualization and processor sharing. Walker [33] evaluates the performance of Amazon EC2 for high-performance scientific applications, reporting that Amazon has much worse performance than traditional HPC clusters. Walker used only up to 32 cores from Amazon EC2 high-CPU extra large instances, the most powerful CPU instances in 2008, with a standard Gigabit Ethernet interconnection network. His main conclusion was that the cloud computing service was not mature for HPC at that moment.

The suitability for HPC of several virtualization technologies was evaluated in [29], showing that operating system virtualization was the only solution that offers near-native CPU and I/O performance. They included in their testbed four Amazon EC2 cluster compute instances, interconnected via 10 Gigabit Ethernet, although they focused more on the overall performance of the VM instead of the scalability of HPC applications.

## 3 Network support in virtualized environments

The basic building blocks of the system (i.e., CPUs, memory and I/O devices) in virtualized environments are multiplexed by the virtual machine monitor (VMM) or hypervisor. Xen [8] is a popular high-performance VMM, used by Amazon EC2 among other cloud providers. Xen systems have a structure with the Xen hypervisor as the lowest and most privileged layer. Above this layer come one or more guest operating systems, which the hypervisor schedules across the physical CPUs. The first guest operating system, called in Xen terminology domain 0 (dom0), boots automatically when the hypervisor boots and receives special management privileges and direct access to all physical hardware by default. The system administrator can log into dom0 in order to manage any further guest operating systems, called domain U (domU) in Xen terminology.

Xen supports two virtualization technologies:

- *Full Virtualization (HVM):* This type of virtualization allows the virtualization of proprietary operating systems, since the guest system's kernel does not require modification, but guests require CPU virtualization extensions from the host CPU (Intel VT [16], AMD-V [4]). In order to boost performance, fully virtualized HVM guests can use special paravirtual device drivers to bypass the emulation for disk and network I/O. Amazon EC2 cluster compute instances use this Xen virtualization technology.
- *Paravirtualization (PV):* This technique requires changes to the virtualized operating system to be hypervisor-aware. This allows the VM to coordinate with the hypervisor, reducing the use of privileged instructions that are typically responsible for the major performance penalties in full virtualization. For this reason, PV guests usually outperform HVM guests. Paravirtualization does not require virtualization extensions from the host CPU.

VMs in Xen usually do not have direct access to network hardware, except using PCI passthrough technique (see next paragraph) or with third-party specific support (like [19] for InfiniBand and [27] for Myrinet). Since most existing device drivers assume a complete control of the device, there cannot be multiple instantiations of such drivers in different guests. To ensure manageability and safe access, Xen follows a split driver model [11]. Domain 0 is a privileged guest that accesses I/O devices directly and provides the VMs abstractions to interface with the hardware. In fact, dom0 hosts a backend driver that communicates with the native driver and the device. Guest VM kernels host a frontend driver, exposing a generic API to guest users. Guest VMs need to pass the I/O requests to the driver domain to access the devices, and this control transfer between domains requires involvement of the VMM. Therefore, Xen networking is completely virtualized. A series of virtual Ethernet devices are created on the host system which ultimately function as the endpoints of network interfaces in the guests. The guest sees its endpoints as standard Ethernet devices, and bridging is used on the host to allow all guests to appear as individual servers.

PCI passthrough [17] is a technique that provides an isolation of devices to a given guest operating system so the device can be used exclusively by that guest, which eventually achieves near-native performance. Thus, this approach benefits network-bounded applications (e.g., HPC applications) that have not adopted virtualization because of contention and performance degradation through the hypervisor (to a driver in the hypervisor or through the hypervisor to a user space emulation). However, assigning devices to specific guests is also useful when those devices cannot be shared. For example, if a system included multiple video adapters, those adapters could be passed through to unique guest domains.

Both Intel and AMD provide support for PCI passthrough in their more recent processor architectures (in addition to new instructions that assist the hypervisor). Intel calls its option Virtualization Technology for Directed I/O (VT-d [2]), while AMD refers to I/O Memory Management Unit (IOMMU [3]). For each case, the new CPUs provide the means to map PCI physical addresses to guest virtual addresses. When this mapping occurs, the hardware takes care of access (and protection), and the guest operating system can use the device as if it were a non-virtualized system. In addition to this mapping of virtual guest addresses to physical memory, isolation is provided in such a way that other guests (or the hypervisor) are precluded from accessing it.

Xen supports PCI passthrough [38] for PV or HVM guests, but dom0 operating system must support it, typically available as a kernel build-time option. For PV guests, Xen does not require any special hardware support, but PV domU kernel must support the Xen PCI frontend driver for PCI passthrough in order to work. Hiding the devices from the dom0 VM is also required, which can be done with Xen using pciback driver. For HVM guests, hardware support (Intel VT-d or AMD IOMMU) is required as well as pciback driver on dom0 kernel. However, domU kernel does not need any special feature, so PCI passthrough with proprietary operating systems is also possible with Xen using HVM guests.

## 4 Messaging middleware for high-performance cloud computing

Two widely extended HPC messaging middleware, Open MPI [12] and MPICH2 [26], were selected for the performance evaluation of native codes (C/C++ and Fortran) carried out on Amazon EC2. In addition, FastMPJ [32] was also selected as representative Java messaging middleware.

Open MPI is an open source MPI-2 implementation developed and maintained by a consortium of academic, research and industry partners. Open MPI's Modular Component Architecture (MCA) allows for developers to implement extensions and features within self-contained components. The Byte Transfer Layer (BTL) framework is designed to provide a consistent interface to different networks for basic data movement primitives between peers. This favors the quick and efficient support of emerging network technologies. Therefore, adding native support for a new network interconnect is straightforward; thus, Open MPI now includes several BTL implementations for TCP, Myrinet, InfiniBand and shared memory support, among other (see Fig. 1).

MPICH2 is a high-performance and open source implementation of the MPI standard (both MPI-1 and MPI-2). Its goal is to provide an MPI implementation that efficiently supports different computation and communication platforms including commodity clusters, high-speed networks, and proprietary high-end computing systems. The ADI-3 (Abstract Device Interface) layer is a full featured low-level interface used in the MPICH2 implementation to provide a portability layer that allows access to many features of a wide range of communication systems. It is responsible for both the point-to-point and one-sided communications. The ADI-3 layer can be implemented on top of the CH3 device, which only requires the implementation of a dozen functions but provides many of the performance advantages of the full ADI-3 interface. In order to support a new platform in MPICH2, only the CH3 channel has to be implemented. Several CH3 channels already offer support for TCP, Myrinet, InfiniBand and shared memory (see Fig. 2).

Nemesis [9] is a new generic communication subsystem designed and implemented to be scalable and efficient both in the intra-node communication context using shared memory and in the inter-node communication case using high-performance networks. Nemesis has been integrated

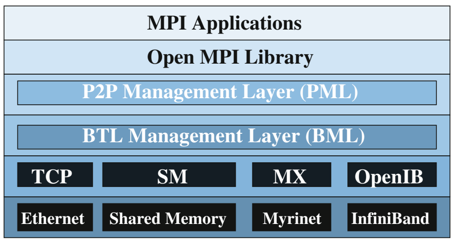


**Fig. 1** Open MPI software layers

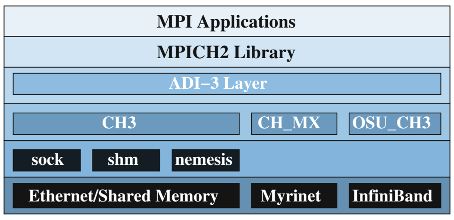


**Fig. 2** MPICH2 software layers

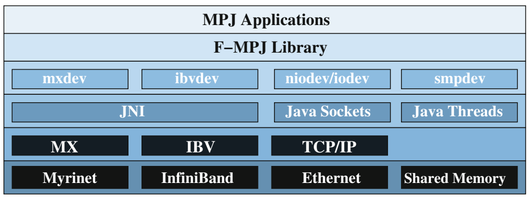


**Fig. 3** FastMPJ communications support overview

in MPICH2 as a CH3 channel and delivers better performance than other dedicated communication channels.

FastMPJ is a Java message-passing implementation which provides shared memory and high-speed networks support on InfiniBand and Myrinet. FastMPJ implements the mpiJava 1.2 API [10], the most widely extended MPJ API, and includes a scalable MPJ collectives library [30]. Figure 3 presents an overview of the FastMPJ communication devices on shared memory and high-speed cluster networks. From top to bottom, the communication support of MPJ applications with FastMPJ is implemented in the device layer. Current FastMPJ communication devices are implemented on JVM threads (`smpdev`, a thread-based device), on sockets over the TCP/IP stack (`iodev` on Java IO sockets and `niodev` on Java NIO sockets), on Myrinet and InfiniBand.

## 5 Performance evaluation

This section presents a performance evaluation of native (C/C++ and Fortran) and Java message-passing middleware for HPC on a cloud computing infrastructure, Amazon EC2 cluster compute instances, whose access to the high-speed network, 10 Gigabit Ethernet, is virtualized. In order to analyze the impact of the cloud network overhead in representative HPC codes, a testbed with similar hardware has been set up. This evaluation consists of a micro-benchmarking of point-to-point data transfers, both inter-VM (through 10 Gigabit Ethernet) and intra-VM (shared memory), at the message-passing library level and its underlying layer, TCP/IP. Then, the significant impact of virtualized communication overhead on the scalability of representative parallel codes, NAS Parallel Benchmarks (NPB) kernels [6], has been assessed. These results indicate that more efficient communication middleware support is required to get over the current cloud network limitations.

## 5.1 Experimental configuration

The evaluation has been conducted on sixteen cluster compute instances of the Amazon EC2 cloud [21], which have been allocated simultaneously in order to obtain nearby instances, and two nodes from our private cloud infrastructure (CAG testbed). Performance results using up to 8 processes have been obtained in a single node, whereas *processes*/8 nodes have been used in the remaining scenarios.

The Amazon EC2 cluster compute instances are a resource introduced in July 2010 with 23 GB of memory and 33.5 EC2 Compute Units (according to Amazon, one EC2 Compute Unit provides the equivalent CPU capacity of a 1.0–1.2 GHz 2007 Opteron or 2007 Xeon processor), running a Linux OS. For these instances, Xen HVM guests, the provider details the specific processor architecture (see Table 1), two Intel Xeon X5570 (2.93 GHz) quad-core Nehalem processors, hence 8 cores per instance, to allow the performance tuning of applications on this processor. These systems are interconnected via 10 Gigabit Ethernet, which is the differential characteristic of this resource. In fact, this EC2 instance type has been specifically designed for HPC applications and other demanding latency-bound applications. However, unfortunately, the network interface card is not available via PCI passthrough in these instances, so its access is virtualized. The Java Virtual Machine (JVM) used is OpenJDK 1.6.0_20 (Amazon Linux 2.6.35 Cluster virtual machine). There is no control on the location of the requested resources as the provider does not support yet the allocation of several instances connected to the same physical switch.

Regarding CAG testbed (see Table 1 again), each node has two Intel Xeon E5620 (2.40 GHz) quad-core processors (thus, 8 cores per node) and 16 GB of memory, interconnected via 10 Gigabit Ethernet (Intel PRO/10GbE NIC). These machines run Xen PV guests whose OS is Linux Debian with kernel 2.6.35, and the JVM is Sun JDK 1.6.0_20. These VMs access directly the network interface card through PCI passthrough, using this approach in 10 Gigabit Ethernet for the first time to the best of our knowledge. The motivation behind enabling PCI passthrough in this cloud is to analyze its impact on the efficient support of high-speed networks in virtualized environments.

The evaluated message-passing libraries are Open MPI 1.4.3 and MPICH2 1.4, used with GNU C/Fortran compiler 4.1.2 and Intel C/Fortran compiler 12.1 (both compilers with -O3 flag), as well as FastMPJ 0.1 (labeled F-MPJ in graphs). The performance differences observed between GNU and Intel compilers were reduced, below 4 %, when no network communications are involved, and completely negligible when network traffic is considered due to the virtualized network I/O overhead. As some works have already stated that GNU C compiler is generally the most efficient and reliable under Linux OS [18], only GNU compiler results are shown for clarity purposes. The point-to-point micro-benchmarking results have been obtained with Intel MPI Benchmarks suite (IMB, formerly Pallas) and its MPJ counterpart communicating byte arrays (hence,

**Table 1** Description of the specific hardware details of the two clouds used

| | Amazon EC2 | CAG |
|---|---|---|
| CPU | 2 × Intel Xeon X5570 Nehalem @2.93 GHz | 2 × Intel Xeon E5620 Westmere @2.40 GHz |
| #Cores | 8 (16 with HT) | 8 (16 with HT) |
| Memory | 23 GB DDR3-1333 MHz | 16 GB DDR3-1066 MHz |
| Memory bandwidth | 32 GB/s | 25.6 GB/s |
| #Memory channels | 3 | 3 |
| QPI speed | 6.4 GT/s | 5.86 GT/s |
| #QPI links | 2 | 2 |
| L3 cache size | 8 MB | 12 MB |
| Interconnect | 10 Gigabit Ethernet | 10 Gigabit Ethernet |
| Virtualization | Xen HVM | Xen PV |

with no serialization overhead). The NPB implementations are the official NPB-MPI version 3.3 and the NPB-MPJ implementation [23]. The metric considered for the evaluation of the NPB kernels is MOPS (Millions of Operations Per Second), which measures the operations performed in the benchmark, that differs from the CPU operations issued. Moreover, NPB Class B workloads have been selected as they are the largest workloads that can be executed in a single Xen PV VM in CAG testbed, due to a Xen bug that limits the amount of memory to 3 GB when using PCI passthrough.

Finally, the performance results presented in this paper are the mean of several measurements, generally 1,000 iterations in ping-pong benchmarks and 5 measurements for NPB kernels. The results show some variability due to the scheduling of the processes/threads on different cores within a node (the pinning of threads to specific cores has not been considered in this work).

## 5.2 Point-to-point micro-benchmarking

Figures 4 and 5 show point-to-point latencies (for short messages) and bandwidths (for long messages) of message-passing transfers using the evaluated message-passing middleware on 10 Gigabit Ethernet and shared memory, respectively. Here, the results shown are the half of the round-trip time of a ping-pong test or its corresponding bandwidth. Each figure presents the performance results of Amazon EC2 HVM and CAG native and CAG PV testbeds.

MPI (MPICH2 and Open MPI) obtains 55–60 µs start-up latency and up to 3.7 Gbps bandwidth for Xen HVM point-to-point communication on Amazon EC2 over 10 Gigabit Ethernet (top graph in Fig. 4). Here, both MPI libraries rely on TCP sockets, which, according to the raw TCP communication test from the Hpcbench suite [13], show poor start-up latency, around 52 µs, similar to MPI start-up latency. However, TCP sockets obtain higher bandwidth than MPI, up to 5.5 Gbps. These results, both MPI and Hpcbench, are quite poor, caused by the overhead in the virtualized access of Xen HVM to the NIC. However, the communication protocol also presents a significant influence as MPI is not able to achieve as much bandwidth as Hpcbench. This assessment is confirmed by the Java results, which show even poorer performance than MPI, with about 140 µs start-up latency and below 2 Gbps bandwidth for FastMPJ using niodev. Here, Java sockets operation suffers a significant performance penalty.

The communication overhead caused by the virtualized access of Xen HVM to the NIC can be significantly alleviated through the use of Xen PCI passthrough (middle graph in Fig. 4). Thus, MPI and Hpcbench achieve start-up latencies around 28–35 µs and bandwidths up to 7.2 Gbps.

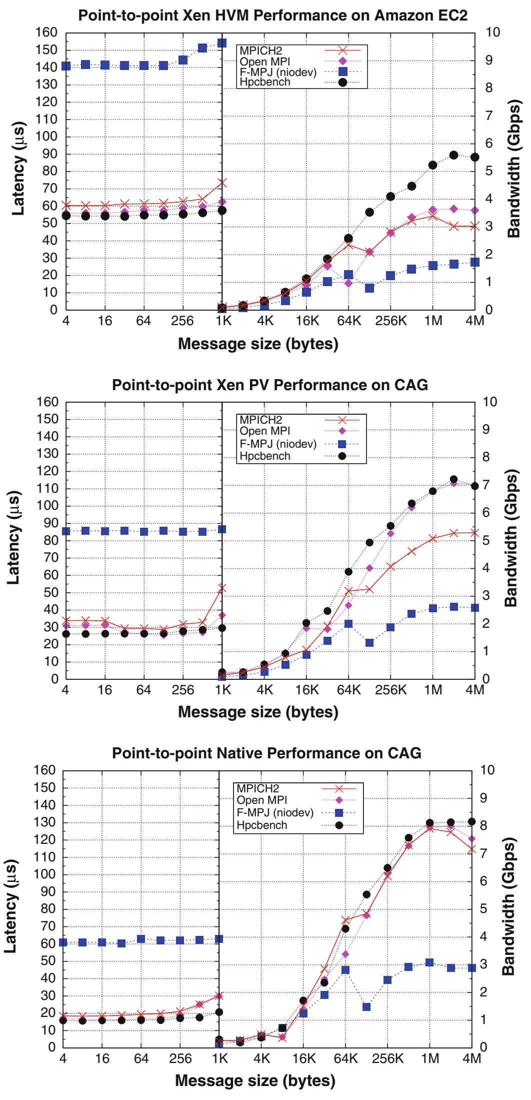


**Fig. 4** Point-to-point communication performance on the analyzed testbeds over 10 Gigabit Ethernet

It is noticeable that Hpcbench and Open MPI obtain quite similar results on this scenario, which suggests that the overhead incurred by network virtualization on Xen HVM scenario without PCI passthrough limits long message performance. Java performance on Xen PV also outperforms its results on Xen HVM, although the performance is still far from the MPI results.

In order to assess the impact of Xen PV virtualization overhead on the previous results, the performance of the communications has been measured on the CAG testbed

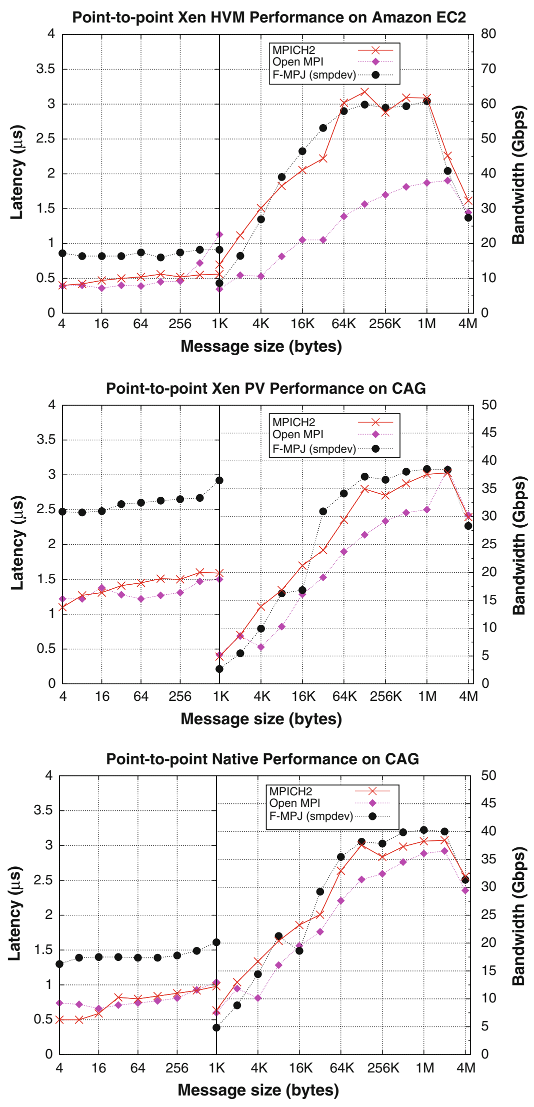


**Fig. 5** Point-to-point shared memory communication performance on the analyzed testbeds

running a non-virtualized environment, thus obtaining the native performance of the system (bottom graph in Fig. 4). The main conclusions that can be derived from these results are that Xen PV incurs an overhead of around 11 µs in start-up latency (25 µs overhead in the case of FastMPJ), whereas native long message performance achieves up to 8.2 Gbps for Hpcbench and almost 8 Gbps for MPI, which are reduced in Xen PV down to 5.3 Gbps for MPICH2 and 7.2 Gbps for Hpcbench. These results are still a little far from the theoretical performance of 10 Gigabit Ethernet, which suggests that TCP processing is the main performance bottleneck, for both short and long message performance. Regarding Java results on the non-virtualized scenario, long message performance is similar to Xen PV results, only showing a small 10 % improvement due to the high overhead of the operation of its NIO sockets implementation.

As the access to multi-core systems is a popular option in cloud computing, in fact each Amazon EC2 cluster computing instance provides two quad-core processors, the performance of communications on such shared memory systems has also been considered. Here, communications are done generally within a single VM, without accessing the network hardware or their communication support (i.e., the communication protocol is not assisted by the NIC). In fact, the use of several VMs per compute node is inefficient, especially in terms of start-up latency, as the data transfers must pass through the hypervisor and the domain0 VM.

The shared memory performance results of message-passing middleware on Amazon EC2 cluster compute instances (top graph in Fig. 5) are significantly superior than on 10 Gigabit Ethernet. Thus, MPI shows start-up latencies below 0.5 µs, thanks to the use of Nemesis in MPICH2 and the shared memory BTL in Open MPI, whereas Java shows latencies below 1 µs. Regarding long message bandwidth, both FastMPJ (smpdev) and MPICH2, which relies on Nemesis, achieve up to 60 Gbps due to their efficient exploitation of multithreading, whereas Open MPI gets up to 38 Gbps.

These high-performance results confirm that Xen obtains close to native performance results for CPU and memory intensive operations, when no I/O activity is involved. In order to prove this statement, the performance results of our CAG testbed with the Xen PV and native configurations (middle and bottom graphs in Fig. 5) have been analyzed. These results show very low start-up latencies, below 1 µs for MPI and 1.5 µs for FastMPJ in the native scenario, which are slightly increased in approximately 0.5 µs for MPI and 1 µs for FastMPJ due to the Xen PV overhead. Moreover, the performance for long messages is similar for the three evaluated middleware, which suggests that the main memory subsystem is the main performance bottleneck, not the communication protocol. The drop in performance for large messages is due to the effect of cache size, as the storage needs exceed the L3 cache size (L3 cache size per processor is 8 MB in EC2 and 12 MB in CAG and is shared by all available cores for both cases). Thus, as the two processes involved in this micro-benchmark are scheduled in the same processor, the performance drops when the message size is equals or higher than a quarter of the L3 cache size due to the one-copy protocol implemented for large messages by

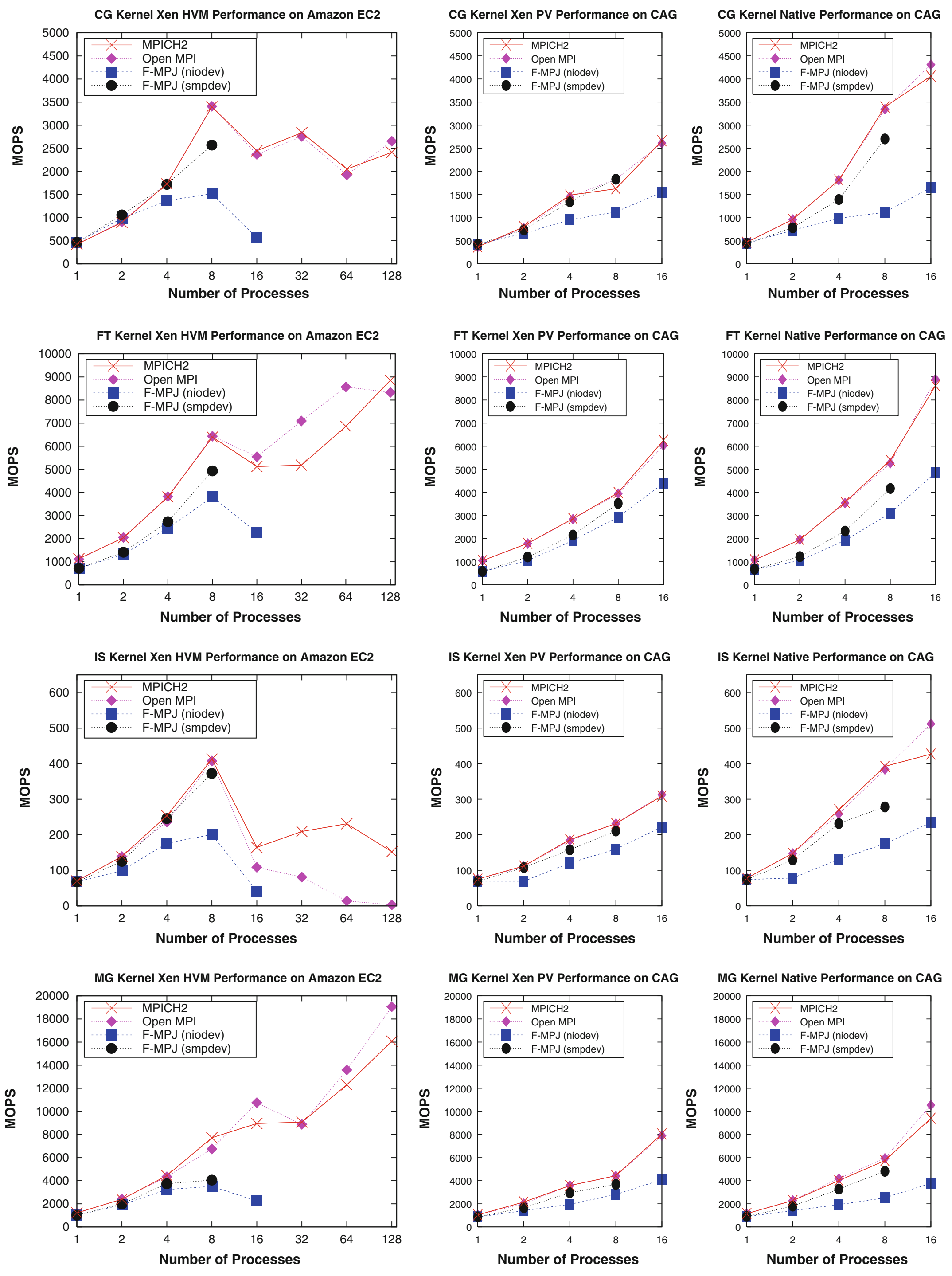


**Fig. 6** NPB performance on Amazon EC2 HVM, CAG PV and CAG native testbeds

these middleware. These results confirm the high efficiency of shared memory message-passing communication on virtualized cloud environments, especially on Amazon EC2 cluster compute instances, which is able to obtain two times better start-up latency and around 40 % higher bandwidth than in CAG mainly thanks to its higher computational power and the higher performance of the memory.

### 5.3 Impact of virtualized networking on applications scalability

The impact of the virtualized network overhead on the scalability of HPC applications has been analyzed using the MPI and MPJ implementations of the NPB, selected as they are probably the benchmarks most commonly used in the evaluation of languages, libraries and middleware for HPC. In fact, there are additional implementations of the NPB for Grid middleware, Java Threads, OpenMP and hybrid MPI/OpenMP.

Four representative NPB codes have been evaluated: CG (Conjugate Gradient), FT (Fourier Transform), IS (Integer Sort) and MG (Multi-Grid). Moreover, Class B workloads have been selected due to the technical limitation of Xen PV as well as they present limited scalability, with performance results highly influenced by the efficiency of the communication middleware, which favor the comparison of the evaluated middleware.

Figure 6 presents the performance, in terms of Millions of Operations Per Second (MOPS), of CG, FT, IS and MG using up to 128 processes on Amazon EC2 and using up to 16 processes on CAG. Regarding CG kernel, which is a communication-intensive code that only includes point-to-point communication primitives, the evaluated middleware is not able to scale when using more than one node on Amazon EC2, due to their poor communications performance on 10 Gigabit Ethernet as it has shown a very high start-up latency. This statement has been proved analyzing CG results on CAG, where MPI and FastMPJ take advantage of the use of 16 processes (2 nodes), both for Xen PV and native scenarios. However, the higher performance of shared memory data transfers on Amazon EC2 allows their cluster compute instances to obtain the best performance results on 8 processes, whereas the CAG testbed shows a small performance penalty due to the higher start-up latency of Xen PV than the native scenario for shared memory communications. This behavior has also been appreciated in the remaining NPB kernels under evaluation. FastMPJ results on 32 and further number of processes are not shown for CG and the other kernels due to the poor performance they achieve.

Regarding FT results, Amazon EC2 suffers significantly the network overhead due to the extensive use of Alltoall primitives, showing similar results for one and four/eight nodes. Only using 16 nodes, this kernel clearly outperforms the results obtained in a single node. Nevertheless, native CAG testbed using 16 processes (2 nodes) outperforms Amazon EC2 results on 128 processes (16 nodes) due to its higher network performance that allows this kernel to scale. IS kernel is a communication-intensive code whose scalability is highly dependent on Allreduce and point-to-point communication latency. Thus, native CAG is able to outperform Amazon EC2, which is able to scale only within a single node using up to the all available processes (8) thanks to its good shared memory performance and the avoidance of network traffic. Finally, MG is a less communication-intensive kernel that is able to scale on Amazon EC2. Nevertheless, CAG results are quite competitive, achieving around 10,000 MOPS with only 2 nodes (16 processes). Regarding FastMPJ results, the shared memory device (smpdev) generally shows the best performance although it is limited to shared memory systems.

The performance evaluation presented in this section has shown that communication bound applications would greatly benefit from the direct access to the NIC in virtualized environments. This is especially true for applications sensitive to network start-up latency that, therefore, can take advantage from the flexibility, elasticity, and economy of cloud services provided that an efficient communication middleware for virtualized cloud environments would be made available.

## 6 Conclusions

The scalability of HPC applications on cloud infrastructures relies heavily on the performance of communications, which depends both on the network fabric and its efficient support in cloud middleware. To solve the current latency and network limitations in cloud services, their providers are deploying high-speed networks (10 Gigabit Ethernet and InfiniBand), although without the proper middleware support as they rely on TCP/IP stack and a virtualized access to the NIC.

This paper has presented an evaluation of HPC message-passing middleware on a cloud computing infrastructure, Amazon EC2 cluster compute instances, equipped with 10 Gigabit Ethernet. The analysis of the performance results obtained, confronted with the experimental results measured in a similar testbed, a private cloud infrastructure, has shown the significant impact that virtualized environments still have on communications performance. This fact demands more efficient communication middleware support to get over the current cloud network limitations, such as TCP/IP stack replacement on high-speed Ethernet networks.

The analysis of the measured performance results has shown significant scalability increases when supporting the direct access to the underlying NIC through PCI passthrough, reducing the communication processing overhead and the associated data copies. In fact, thanks to this technique, our experimental two-node cloud testbed is able to outperform the results obtained on sixteen Amazon EC2 cluster compute instances.

**Acknowledgments** This work was funded by the Ministry of Science and Innovation of Spain under Project TIN2010-16735, by the Galician Government (Consolidation of Competitive Research Groups, Xunta de Galicia ref. 2010/6), by an FPU Grant AP2010-4348 and by an Amazon Web Services (AWS) LLC research grant.

## References

1. A Message Passing Interface Standard (MPI) http://www.mcs.anl.gov/research/projects/mpi/. Accessed July 2012
2. Abramson D et al (2006) Intel virtualization technology for directed I/O. Intel Technol J 10(3):179–192
3. Advanced Micro Devices (AMD) (2009) I/O Virtualization Technology (IOMMU). http://support.amd.com/us/Processor_TechDocs/34434-IOMMU-Rev_1.26_2-11-09.pdf. Accessed July 2012
4. Advanced Micro Devices (AMD) Virtualization Technology (AMD-V). http://sites.amd.com/us/business/it-solutions/virtualization/Pages/amd-v.aspx. Accessed July 2012
5. Baek SJ, Park SM, Yang SH, Song EH, Jeong YS (2010) Efficient server virtualization using grid service infrastructure. J Inf Process Syst 6(4):553–562
6. Bailey DH et al (1991) The NAS parallel benchmarks. Int J High Perform Comput Appl 5(3):63–73
7. Baker M, Carpenter B (2000) MPJ: a proposed Java message passing API and environment for high performance computing. In: Proceedings of 15th IPDPS workshops on parallel and distributed processing (IPDPS'00), Cancun, LNCS, vol 1800, pp 552–559
8. Barham P, Dragovic B, Fraser K, Hand S, Harris TL, Ho A, Neugebauer R, Pratt I, Warfield A (2003) Xen and the art of virtualization. In: Proceedings of 19th ACM symposium on operating systems principles (SOSP'03), Bolton Landing (Lake George), pp 164–177
9. Buntinas D, Mercier G, Gropp W (2006) Design and evaluation of nemesis, a scalable, low-latency, message-passing communication subsystem. In: Proceedings of 6th IEEE international symposium on cluster computing and the grid (CCGRID'06), Singapore, pp 521–530
10. Carpenter B, Fox G, Ko S, Lim S (2002) mpiJava 1.2: API specification. http://www.hpjava.org/reports/mpiJava-spec/mpiJava-spec/mpiJava-spec.html. Accessed July 2012
11. Fraser K, Hand S, Neugebauer R, Pratt I, Warfield A, Williamson M (2004) Safe hardware access with the Xen virtual machine monitor. In: Proceedings of 1st workshop on operating system and architectural support for the on demand IT infrastructure (OASIS'04), Boston
12. Gabriel E et al (2004) Open MPI: goals, concept, and design of a next generation MPI implementation. In: Proceedings of 11th European PVM/MPI users' group meeting, Budapest, pp 97–104
13. Huang B, Bauer M, Katchabaw M (2005) Hpcbench—a Linux-based network benchmark for high performance networks. In: Proceedings of 19th international symposium on high performance computing systems and applications (HPCS'05), Guelph, pp 65–71
14. Huang W, Liu J, Abali B, Panda DK (2006) A case for high performance computing with virtual machines. In: Proceedings of 20th international conference on supercomputing (ICS'06), Cairns, pp 125–134
15. Huang W, Koop MJ, Gao Q, Panda DK (2007) Virtual machine aware communication libraries for high performance computing. In: Proceedings of ACM/IEEE conference on supercomputing (SC'07), Reno, pp 1–12
16. Intel Corporation (2006) Virtualization technology (Intel VT). http://www.intel.com/technology/virtualization/technology.htm?iid=tech_vt+tech. Accessed July 2012
17. Jones T (2009) Linux virtualization and PCI passthrough. http://www.ibm.com/developerworks/linux/library/l-pci-passthrough/. Accessed July 2012
18. Karna AK, Zou H (2010) Cross comparison on C compilers reliability impact. J Convergence 1(1):65–74
19. Liu J, Huang W, Abali B, Panda DK (2006) High performance VMM-bypass I/O in virtual machines. In: Proceedings of USENIX'06 annual technical conference, Boston, pp 29–42
20. Liu ST, Chen YM (2011) Retrospective detection of malware attacks by cloud computing. Int J Inf Technol Commun Convergence 1(3):280–296
21. Amazon Web Services LLC (AWS LLC) Amazon elastic compute cloud (Amazon EC2). http://aws.amazon.com/ec2. Accessed July 2012
22. Amazon Web Services LLC (AWS LLC) High performance computing using Amazon EC2. http://aws.amazon.com/ec2/hpc-applications/. Accessed July 2012
23. Mallón DA, Taboada GL, Touriño J, Doallo R (2009) NPB-MPJ: NAS parallel benchmarks implementation for message-passing in Java. In: Proceedings of 17th Euromicro international conference on parallel, distributed, and network-based processing (PDP'09), Weimar, pp 181–190
24. Mansley K, Law G, Riddoch D, Barzini G, Turton N, Pope S (2007) Getting 10 Gb/s from Xen: safe and fast device access from unprivileged domains. In: Proceedings of workshop on virtualization/Xen in high-performance cluster and grid computing (VHPC'07), Rennes, pp 224–233
25. Mateescu G, Gentzsch W, Ribbens CJ (2011) Hybrid computing-where HPC meets grid and cloud computing. Future Gener Comput Syst 27(5):440–453
26. MPICH2 (2005) High-performance and widely portable MPI. http://www.mcs.anl.gov/research/projects/mpich2/. Accessed July 2012
27. Nanos A, Koziris N (2009) MyriXen: message passing in Xen virtual machines over Myrinet and Ethernet. In: Proceedings of 4th workshop on virtualization in high-performance cloud computing (VHPC'09), Delft, pp 395–403
28. Raj H, Schwan K (2007) High performance and scalable I/O virtualization via self-virtualized devices. In: Proceedings of 16th international symposium on high performance distributed computing (HPDC'07), Monterey, pp 179–188
29. Regola N, Ducom JC (2010) Recommendations for virtualization technologies in high performance computing. In: Proceedings of 2nd international conference on cloud computing technology and science (CloudCom'10), Indianapolis, pp 409–416
30. Taboada GL, Ramos S, Touriño J, Doallo R (2011) Design of efficient java message-passing collectives on multi-core clusters. J Supercomput 55(2):126–154
31. Taboada GL, Ramos S, Expósito RR, Touriño J, Doallo R (2012) Java in the high performance computing arena: research, practice and experience. Sci Comput Program (in press). doi:10.1016/j.scico.2011.06.002

32. Taboada GL, Touriño J, Doallo R (2012) F-MPJ: scalable Java message-passing communications on parallel systems. J Supercomput 60(1):117–140
33. Walker E (2008) Benchmarking Amazon EC2 for high-performance scientific computing. LOGIN: USENIX Mag 33(5):18–23
34. Wang G, Ng TSE (2010) The impact of virtualization on network performance of Amazon EC2 data center. In: Proceedings of 29th conference on information communications (INFOCOM'10), San Diego, pp 1163–1171
35. Wang X, Sang Y, Liu Y, Luo Y (2011) Considerations on security and trust measurement for virtualized enviroment. J Convergence 2(2):19–24
36. Whitaker A, Shaw M, Gribble SD (2002) Denali: lightweight virtual machines for distributed and networked applications. Technical Report 02-02-01, University of Washington, USA
37. Won C, Lee B, Park K, Kim MJ (2008) Eager data transfer mechanism for reducing communication latency in user-level network protocols. J Inf Process Syst 4(4):133–144
38. Xen Org (2005) Xen PCI passthrough. http://wiki.xensource.com/xenwiki/XenPCIpassthrough. Accessed July 2012
39. Ye Y, Li X, Wu B, Li Y (2011) A comparative study of feature weighting methods for document co-clustering. Int J Inf Technol Commun Convergence 1(2):206–220

# Chapter 7

# Performance Analysis of HPC Applications in the Cloud

The content of this chapter corresponds to the following journal paper:

- **Title:** Performance analysis of HPC applications in the cloud

- **Authors:** Roberto R. Expósito, Guillermo L. Taboada, Sabela Ramos, Juan Touriño, Ramón Doallo

- **Journal:** Future Generation Computer Systems

- **Editorial:** Elsevier B.V.

- **ISSN:** 0167-739X

- **Year:** 2013

- **Volume(Number):Pages:** 29(1):218–229

- **DOI:** 10.1016/j.future.2012.06.009

The final publication is available at http://www.sciencedirect.com/science/article/pii/S0167739X12001458. A copy of the accepted paper has been included next.

# Performance analysis of HPC applications in the cloud

Roberto R. Expósito*, Guillermo L. Taboada, Sabela Ramos, Juan Touriño, Ramón Doallo

*Computer Architecture Group, University of A Coruña, A Coruña, Spain*

A R T I C L E  I N F O

*Article history:*

*Keywords:*
Cloud computing
High Performance Computing
Amazon EC2 Cluster Compute platform
MPI
OpenMP

A B S T R A C T

The scalability of High Performance Computing (HPC) applications depends heavily on the efficient support of network communications in virtualized environments. However, Infrastructure as a Service (IaaS) providers are more focused on deploying systems with higher computational power interconnected via high-speed networks rather than improving the scalability of the communication middleware. This paper analyzes the main performance bottlenecks in HPC application scalability on the Amazon EC2 Cluster Compute platform: (1) evaluating the communication performance on shared memory and a virtualized 10 Gigabit Ethernet network; (2) assessing the scalability of representative HPC codes, the NAS Parallel Benchmarks, using an important number of cores, up to 512; (3) analyzing the new cluster instances (CC2), both in terms of single instance performance, scalability and cost-efficiency of its use; (4) suggesting techniques for reducing the impact of the virtualization overhead in the scalability of communication-intensive HPC codes, such as the direct access of the Virtual Machine to the network and reducing the number of processes per instance; and (5) proposing the combination of message-passing with multithreading as the most scalable and cost-effective option for running HPC applications on the Amazon EC2 Cluster Compute platform.

## 1. Introduction

Cloud computing [1,2] is an Internet-based computing model which has gained significant popularity in the past several years as it provides on-demand network access to a shared pool of configurable and often virtualized computing resources typically billed on a pay-as-you-use basis.

Virtualization is a mechanism to abstract the hardware and system resources from a given operating system, and it is one of the most important technologies that make the cloud computing paradigm possible. Infrastructure-as-a-Service (IaaS) is a type of cloud service which easily enables users to set up a virtual cluster providing cost-effective solutions. Many research efforts have been done in the last years to reduce the overhead imposed by virtualized environments, and due to this, cloud computing is becoming an attractive option for High Performance Computing (HPC).

Amazon Web Services (AWS) is an IaaS provider whose Elastic Compute Cloud (EC2) [3] is nowadays among the most used and largest public clouds platforms. Some early studies [4–6] have evaluated public clouds for HPC since 2008 and the main conclusion was that clouds at that time were not designed for running tightly coupled HPC applications. The main reasons were the poor network performance caused by the virtualization I/O overhead, the use of commodity interconnection technologies (e.g., Gigabit Ethernet) and processor sharing, that limit severely the scalability of HPC applications in public clouds. To overcome these constraints Amazon released the Cluster Compute Quadruple Extra Large instance (cc1.4xlarge, abbreviated as CC1) in July 2010, a resource that provides powerful CPU resources (two quad-core processors) and dedicated physical node allocation, together with a full-bisection high-speed network (10 Gigabit Ethernet). The availability of a high-speed network is key for the scalability of HPC applications. However, virtualized network resources lack efficient communication support, which prevents HPC codes to take advantage of these high performance networks. More recently, in November 2011, Amazon released a new type of instance suitable for HPC, the Cluster Compute Eight Extra Large instance (cc2.8xlarge, abbreviated as CC2), with improved CPU power as it has two octa-core processors. Both resources (CC1 and CC2 instances) are intended to be well suited for HPC applications and other demanding network-bound applications [7].

This paper evaluates CC1 and CC2 instances for High Performance Cloud Computing on Amazon EC2 cloud infrastructure, the largest public cloud in production, using up to 512 cores, hence 64 CC1 instances and 32 CC2 instances were used. In this evaluation, the scalability of representative parallel HPC codes using the NAS Parallel Benchmarks (NPB) suite [8] is analyzed. Moreover, as the

* Correspondence to: Computer Architecture Group, University of A Coruña, Campus de Elviña s/n, 15071 A Coruña, Spain. Tel.: +34 981167000; fax: +34 981167160.

*E-mail addresses:* rreye@udc.es (R.R. Expósito), taboada@udc.es (G.L. Taboada), sramos@udc.es (S. Ramos), juan@udc.es (J. Touriño), doallo@udc.es (R. Doallo).

NPB original suite only contains pure MPI kernels, the NPB Multi-Zone (NPB-MZ) suite [9], which contains hybrid MPI+OpenMP codes, have also been assessed since the use of threads can alleviate the network bottleneck in the cloud.

The structure of this paper is as follows: Section 2 presents the related work. Section 3 describes the I/O support in virtualized environments, and particularly explains the network support in Amazon EC2 CC platform. Section 4 introduces the experimental configuration, both hardware and software, and the methodology of the evaluation conducted in this work. Section 5 analyzes the performance results of the selected message-passing middleware on Amazon EC2 CC instances. These results have been obtained from a micro-benchmarking of point-to-point primitives, as well as an application benchmarking using representative HPC codes in order to analyze the scalability of HPC applications. Section 6 summarizes our concluding remarks.

## 2. Related work

There are several feasibility studies of the use of public clouds for HPC, all concluding that the network communication overhead is the main performance bottleneck, limiting severely applications scalability. There are also many works that tackle the reduction of this network overhead.

Currently virtualization of CPU and memory resources presents very low overhead, close to native (bare-metal) performance on x86 architectures [10]. The key for reducing the virtualization overhead imposed by hypervisors, also known as Virtual Machine Monitors (VMM), are software techniques such as ParaVirtualization (PV) [11] or Hardware-assisted Virtualization (HVM) [12,13]. However, the efficient virtualization support of I/O devices is still work in progress, especially for network I/O, which turns out to be a major performance penalty [14].

### *2.1. I/O virtualization*

Previous works on I/O virtualization can be classified as either software- or hardware-based approaches. In software-based approaches devices cannot be accessed directly by the guest VMs and every I/O operation is virtualized. A representative project is the driver domain model [15], where only an Isolated Device Domain (IDD) has access to the hardware and runs native device drivers. The rest of guest VMs pass the I/O requests to the IDD through special VMM-aware or paravirtual drivers. This technique, implemented by all modern hypervisors, has two main drawbacks: (1) the still poor performance, despite its continuous improvements [16,17], and (2) the need for guest VM modification, unfeasible in some systems.

Hardware-based I/O virtualization achieves higher performance supporting the direct device access (also known as "PCI passthrough access" or "direct device assignment") from a guest VM. Initially the use of self-virtualized adapters (smart network devices) improved I/O performance by offloading some virtualization functionality onto the device [18,19]. However, the requirements of non-standard hardware support and custom drivers hindered the adoption of this early ad-hoc hardware-approach. A more recent work by Yassour et al. [20] provides almost native network performance in Linux/KVM environments, but its implementation of the PCI passthrough technique does not support live migration and requires that the VM has exclusive access to a device.

Another hardware-based approach is SR/MR-IOV [21], a standard that allows a physical device to present itself to the system as multiple virtual devices, exporting to VMs part of the capabilities of smart network devices. Thus, VMs have a direct network path bypassing the hypervisor and the privileged guest. However, the access from multiple VMs to the physical device has to be multiplexed by the network interface firmware. Although this approach achieves reasonably good performance [22], a new hardware support has to be incorporated to PCI devices. Nevertheless, the direct device access cannot provide bare-metal performance, according to Gordon et al. [23], due to the host involvement, as it intercepts all interrupts inducing multiple unwarranted guest/host context switches. In order to eliminate the overhead caused by these unwarranted exits Gordon et al. proposed ELI, a software-only approach for handling interrupts directly and securely within guest VMs.

### *2.2. HPC in the cloud*

The interest in the use of public clouds for HPC increases as their availability, computational power and performance improves, which has motivated lately multiple works about adopting cloud computing for HPC applications. Among them, the feasibility of HPC on Amazon EC2 public cloud infrastructure is the most common approach. Regarding applications, the execution of scientific workflows [24–27] has obtained significant success, whereas many early studies [4–6,28,29] have assessed that public clouds have not been designed for running tightly coupled MPI applications, primarily due to their poor network performance, processor sharing and the use of commodity interconnection technologies. In order to overcome this performance bottleneck, Amazon EC2 introduced their Cluster Compute (CC) instances.

Amazon EC2 CC1 instances, the first version of CC instance type, have been evaluated in some recent related work. Thus, Carlyle et al. [30] compared, from an economic point of view, the benefits in academia of operating a community cluster program versus the provision of Amazon EC2 CC1 instances. Regola and Ducom [31] analyzed the suitability of several virtualization technologies for HPC, among them 4 CC1 instances. Nevertheless, their work is more focused on the overall performance of the evaluated hypervisors instead of the virtualized network performance and the scalability of communications. Ramakrishnan et al. [32] stated that virtualized network is the main performance bottleneck, after analyzing the communication overhead of a number of different interconnect technologies, including 10 Gigabit Ethernet. Furthermore, Zhai et al. [33] conducted a comprehensive evaluation of MPI applications on 16 CC1 instances, revealing a significant performance increase compared to previous evaluations on standard and High-CPU EC2 instances. Finally, Mauch et al. [34] give an overview on the current state of HPC IaaS offerings and present an approach to use InfiniBand in a private virtualized environment. They present HPL benchmark results for both CC1 and CC2 instance types, but using only one instance of each type. In addition, they do not study CC1/CC2 performance and scalability using representative HPC applications and a high number of cores.

The review of the related works on evaluating Amazon EC2 for HPC has revealed the lack of suitable assessments of the performance of the new CC2 instances. This paper addresses this lack evaluating thoroughly the performance of CC2 instances, as well as comparing these instances against the previous CC1 instances. Moreover, previous works were limited to the evaluation of MPI codes using up to 16 CC1 instances (128 cores). Additional contributions of this paper are the performance evaluation using a significantly higher number of cores (up to 512) both in terms of single instance performance, scalability and cost-efficiency of its use as well as taking into account hybrid programming models such as MPI+OpenMP.

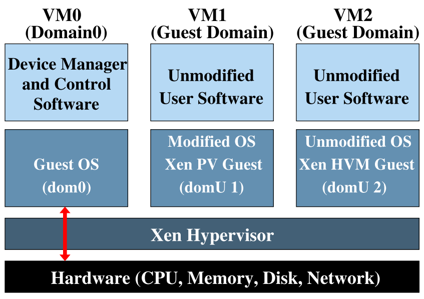

VM0
(Domain0)
VM1
(Guest Domain)
VM2
(Guest Domain)
Device Manager and Control Software
Unmodified User Software
Unmodified User Software
Guest OS (dom0)
Modified OS Xen PV Guest (domU 1)
Unmodified OS Xen HVM Guest (domU 2)
Xen Hypervisor
Hardware (CPU, Memory, Disk, Network)

**Fig. 1.** Xen architecture overview.

## 3. Virtualization support on Amazon EC2

Xen [10] is a high performance hypervisor or VMM quite popular among cloud providers, and it is used by all current Amazon EC2 instances. Xen architecture (Fig. 1) has the hypervisor as the lowest and most privileged layer and above it comes one or more guest operating systems, which the hypervisor schedules across the physical CPUs. The first guest OS, called domain 0 (dom0), boots automatically when the hypervisor boots and receives special management privileges and exclusive direct access to all physical hardware. This dom0 OS is used to manage any further guest OS, called domain U (domU), and the virtualization technologies supported for creating these domU guests are full virtualization assisted with hardware support (HVM) and ParaVirtualization (PV). I/O virtualization is usually implemented in one of three ways (Fig. 2): device emulation, using paravirtualized I/O drivers and giving a VM a direct device access (“PCI passthrough access” or “direct device assignment”).

In Xen VMM, device emulation is available for domU HVM guests by default, where dom0 emulates a real I/O device for which the guest already has a driver. The dom0 has to trap all device accesses from domU and converts them to operations on a real and possibly different device (Fig. 2(a)), requiring many context switches between domains and therefore offering low performance. In the PV approach (Fig. 2(b)), available for PV guests as well as HVM guests, domU guests use special VMM-aware drivers where the I/O code is optimized for VM execution. This code is manually pre-processed or re-coded to remove privileged instructions that are substituted by VM application programming interfaces, or “hypercalls”, in their place. Therefore, performance is improved but it is still far from native, besides the fact that it requires changes in the device drivers used by domU guests.

With direct device access (Fig. 2(c)), also available for both types of guests, the domU guest “sees” a real device and interacts with it directly, without software intermediaries, improving the performance since no dom0 involvement is required. Moreover, domU guests can use any device for which they have a driver, as no modifications are necessary in the native device drivers used by them. For PV guests, Xen does not require any special hardware support, but domU kernel must support the PCI frontend driver (pcifront) in order to work. Hiding the devices from the dom0 is also required, which can be done using the pciback driver. However, an I/O Memory Management Unit (IOMMU) in hardware is required for HVM guests as well as a pciback driver on dom0 kernel. Examples of IOMMUs are Intel’s VT-d [35] or AMD’s IOMMU [36]. This technique, the only one which can provide near bare-metal performance, also has limitations: it is not fully compatible with live migration and it requires dedication of a device to a domU guest (the latter can be solved with PCI standard SR/MR-IOV [21] devices).

### 3.1. *Amazon EC2 cluster compute platform*

The Amazon EC2 Cluster Compute Quadruple Extra Large instances (abbreviated as CC1) and Cluster Compute Eight Extra Large instances (CC2) are resources with 23 and 60.5 GB of memory and 33.5 and 88 EC2 Compute Units (ECUs) for CC1 and CC2, respectively. According to Amazon one ECU provides the equivalent CPU capacity of a 1.0–1.2 GHz 2007 Opteron or 2007 Xeon processor.

For these instances, the provider details the specific processor architecture: two Intel Xeon X5570 quad-core Nehalem processors for CC1, hence 8 cores per CC1 instance, and two Intel Xeon E5-2670 octa-core Sandy Bridge processors for CC2, hence 16 cores per CC2 instance. These systems are interconnected via a high-speed network (10 Gigabit Ethernet), which is the differential characteristic of these resources. In fact, these EC2 instance types have been specifically designed for HPC applications and other demanding latency-bound applications.

Both versions of CC instances, whose main characteristics are presented in Table 1, use Xen HVM virtualization technology, whereas the rest of Amazon EC2 instance types are Xen PV guests. Moreover, instead of using an I/O device emulation for the Network Interface Card (NIC) which is configured by default in HVM guests, these cluster instances have installed paravirtual drivers for improving network and disk performance. Therefore, the access to the NIC in Amazon EC2 instances is paravirtualized, so a direct access is not available which causes a significant performance penalty as will be shown later.

## 4. Experimental configuration and evaluation methodology

The performance evaluation has been conducted on 64 CC1 and 32 CC2 instances of the Amazon EC2 cloud [3]. These resources have been allocated simultaneously and in the same placement group in order to obtain nearby instances, and thus obtaining the

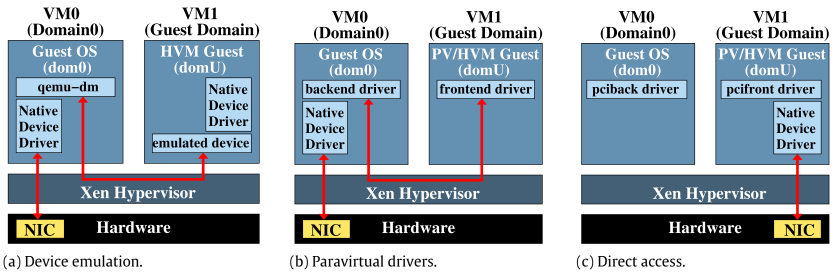


**Fig. 2.** Network virtualization support in Xen.

**Table 1**
Description of the Amazon EC2 cluster compute quadruple and eight extra large instances.

| | Quadruple extra large | Eight extra large |
|---|---|---|
| CPU | 2 × Intel Xeon X5570 Nehalem @2.93 GHz | 2 × Intel Xeon E5-2670 Sandy Bridge @2.60 GHz |
| ECUs | 33.5 | 88 |
| #Cores | 8 (16 with HT) | 16 (32 with HT) |
| Memory | 23 GB DDR3 | 60.5 GB DDR3 |
| Storage | 1690 GB | 3370 GB |
| API name | cc1.4xlarge | cc2.8xlarge |
| Price (Linux) | $1.30 per hour | $2.40 per hour |
| Interconnect | 10 Gigabit Ethernet (full-bisection bandwidth with placement groups) | |
| Virtualization | Xen HVM 64-bit platform (PV drivers for I/O) | |

benefits from the full-bisection high bandwidth network provided by Amazon for these instance types.

Regarding the software, two widely extended HPC messaging middleware, OpenMPI [37] 1.4.4 and MPICH2 [38] 1.4.1, were selected for the performance evaluation of native codes (C/C++ and Fortran). In addition, FastMPJ [39] was also selected as representative Message-Passing in Java (MPJ) middleware [40]. In all cases, the most efficient communication mechanism available for network transfers and shared memory scenarios was selected.

The evaluation consists of a micro-benchmarking of point-to-point data transfers, both inter-VM (through 10 Gigabit Ethernet) and intra-VM (shared memory), at the message-passing library level. The point-to-point micro-benchmarking results have been obtained with the Intel MPI Benchmarks suite (IMB) and its MPJ counterpart communicating byte arrays (hence, with no serialization overhead). Then, the impact of paravirtualized network on the scalability of representative parallel codes, NAS Parallel Benchmarks (NPB) kernels [8], has been assessed using the official NPB-MPI version 3.3 and the NPB-MPJ implementation [41]. In order to determine whether a hybrid parallel programming model could overcome the current Amazon EC2 network limitations, the NPB Multi-Zone (NPB-MZ) suite [9] version 3.3.1 for hybrid MPI+OpenMP codes has also been analyzed. The metrics considered for the evaluation of the NPB kernels are MOPS (Millions of Operations Per Second), which measures the operations performed in the benchmark (that differ from the CPU operations issued), and their corresponding speedups. Moreover, NPB Class C workloads have been selected because their performance is highly influenced by the efficiency of the communication middleware and the support of the underlying network, as well as they are the largest workloads that can be executed in a single CC1 instance.

Both the GNU 4.4.4 and the Intel 12.1 compilers have been considered for the NPB kernels, and the reported results in the next graphs have been obtained from binaries compiled with the best compiler in each case. Regarding the Java Virtual Machine (JVM), the version used was the OpenJDK Runtime Environment 1.6.0_20. Finally, the performance results presented in this paper are the mean of several measurements, generally 10,000 iterations in ping-pong benchmarks and 5 measurements for NPB kernels.

## 5. Assessment of Amazon EC2 cluster compute platform for HPC

This section presents an analysis of the performance of point-to-point communications and the scalability of HPC codes on the Amazon EC2 Cluster Compute platform designed for HPC, using the selected micro-benchmarks and representative kernels described in the previous section.

### 5.1. Inter-VM point-to-point micro-benchmarking

Fig. 3 shows point-to-point latencies (for short messages) and bandwidths (for long messages) of message-passing transfers using the selected message-passing middleware, MPICH2, OpenMPI and FastMPJ, on Amazon EC2 CC1 (left graph) and CC2 (right graph) instances in an inter-VM scenario, where communications are performed through a 10 Gigabit Ethernet network link. The results shown are the half of the round-trip time of a ping-pong test and its corresponding bandwidth.

On CC1 instances (left graph) OpenMPI and FastMPJ obtains the lowest start-up latency (55 μs), slightly better than MPICH2 latencies (around 57 μs), but quite high compared to the usual 10 Gigabit Ethernet start-up latencies on bare-metal, which can be as low as 10 μs. Regarding bandwidth the three libraries show similar performance on CC1 instances, up to 4.9 Gbps bandwidth, quite far from the theoretical 10 Gbps bandwidth provided by the interconnection technology and the 9 Gbps that message-passing libraries can achieve without virtualization overhead.

The results on CC2 instances (right graph) show slightly better performance (10% in the best case -OpenMPI-) than using CC1 instances, which can be motivated by the higher performance (about 31%) of the CC2 processor core (5.5 ECUs) compared to the computational power of the CC1 processor core (4.2 ECUs). Thus, these libraries obtain start-up latencies around 50–54 μs, whereas observed bandwidths are up to 5.4 Gbps, also suffering from poor network virtualization support. Despite these minimum improvements the overhead in the paravirtualized access of VMs to the 10 Gigabit Ethernet NIC still represents the main performance bottleneck.

### 5.2. Intra-VM point-to-point micro-benchmarking

Fig. 4 shows point-to-point performance of message-passing transfers in the intra-VM scenario, where data transfers are implemented on shared memory (hence, without accessing the network hardware). Thus, the observed shared memory performance results are significantly better than the inter-VM scenario.

Performance results on CC1 instances (left graph) present very low start-up latencies, 0.3 μs for OpenMPI and 0.4 μs for MPICH2 and FastMPJ, and similar high bandwidths, up to 60.9 Gbps for OpenMPI and 62.5 Gbps for MPICH2 and FastMPJ, which confirms the efficient virtualization support in the Xen hypervisor for CPU- and memory-intensive operations when no I/O network activity is involved.

Regarding CC2 instances (right graph), start-up latencies are slightly higher than on CC1: OpenMPI also obtains the lowest values, 0.35 μs, whereas MPICH2 and FastMPJ obtain around 0.47 μs, probably due to the lower clock frequency of the processor in CC2 instances (2.60 GHz) than in CC1 (2.93 GHz). The three evaluated libraries show again similar long message performance results, which suggests that their communication protocols are close to the maximum effective memory bandwidth. The measured shared memory bandwidths are slightly higher in CC1, particularly in the range 1 kB–1 MB, due to its higher clock frequency. However, from 2 MB CC2 shared memory performance is higher as its L3 cache size (20 MB shared by 8 cores) is larger than CC1 L3 cache size

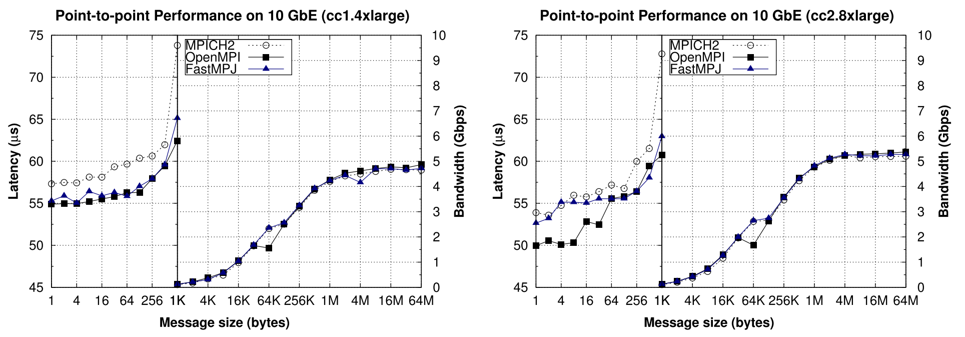


**Fig. 3.** Point-to-point communication performance on Amazon EC2 CC instances over 10 Gigabit Ethernet.

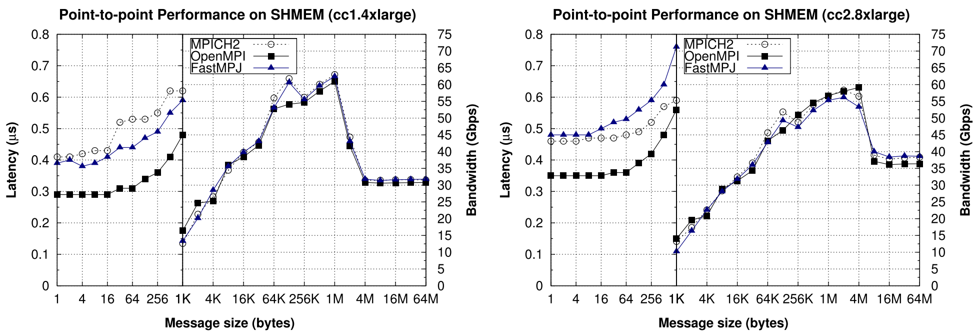


**Fig. 4.** Point-to-point shared memory communication performance on Amazon EC2 CC instances.

(8 MB shared by 4 cores). In fact, CC1 performance results fall from 2 MB on as the messages and the intermediate shared memory buffer (used internally by message-passing libraries) exceed the L3 cache region that can be addressed by a single process (4 MB). Moreover, as the message size increases the percentage of the message that fits in L3 cache reduces and as a direct consequence its performance falls. It has to be taken into account that the OS generally schedules the two communicating processes on the same processor, thus reducing the L3 cache region available for each one. However, this scheduling strategy improves the overall communication performance, especially the start-up latency.

This significant influence of the cache hierarchy on the performance of shared memory communications on both CC1 and CC2 instances confirms the low virtualization overhead experienced, thus providing highly efficient shared memory message-passing communication on virtualized cloud environments.

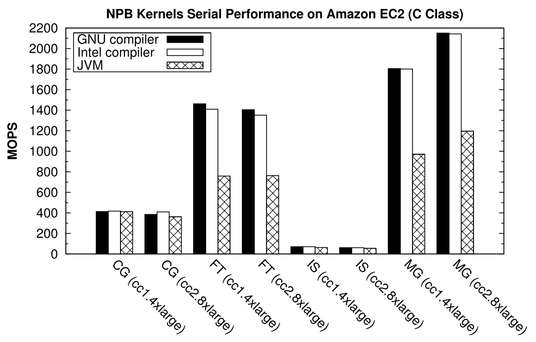


**Fig. 5.** NPB kernels serial performance.

### 5.3. *HPC kernels performance analysis*

The impact of the paravirtualized network overhead on the scalability of representative HPC codes has been analyzed using the MPI and MPJ implementations of the NPB, selected as it is probably the benchmarking suite most commonly used in the evaluation of languages, libraries and middleware for HPC. Four representative NPB kernels, the most communication-intensive codes of the suite, have been evaluated: CG (Conjugate Gradient), FT (Fourier Transform), IS (Integer Sort) and MG (Multi-Grid). As mentioned in Section 4, NPB Class C workloads have been used.

The native (C/Fortran) implementations of the kernels have been compiled using the GNU and Intel compilers (both with -O3 flag), showing their performance for the serial kernels, including also Java results, in Fig. 5. As it can be seen, the impact of the compiler on the performance of these codes is almost negligible. Thus, for clarity purposes the next graphs only show results obtained with the best performer compiler for each kernel.

Figs. 6 and 7 present the performance of CG, FT, IS and MG using up to 512 cores on Amazon EC2 CC platform (hence, using 64 CC1 and 32 CC2 instances). The performance metrics reported are MOPS (left graphs), and their corresponding speedups (right graphs). The number of CC instances used in the performance evaluation is the number of cores used divided by the number of cores per instance type (8 cores for CC1 and 16 cores for CC2).

The analysis of the NPB kernels performance shows that the evaluated libraries obtain good results when running entirely on shared memory (on a single VM) using up to 8 and 16 cores in CC1 and CC2 instances, respectively, due to the higher performance and

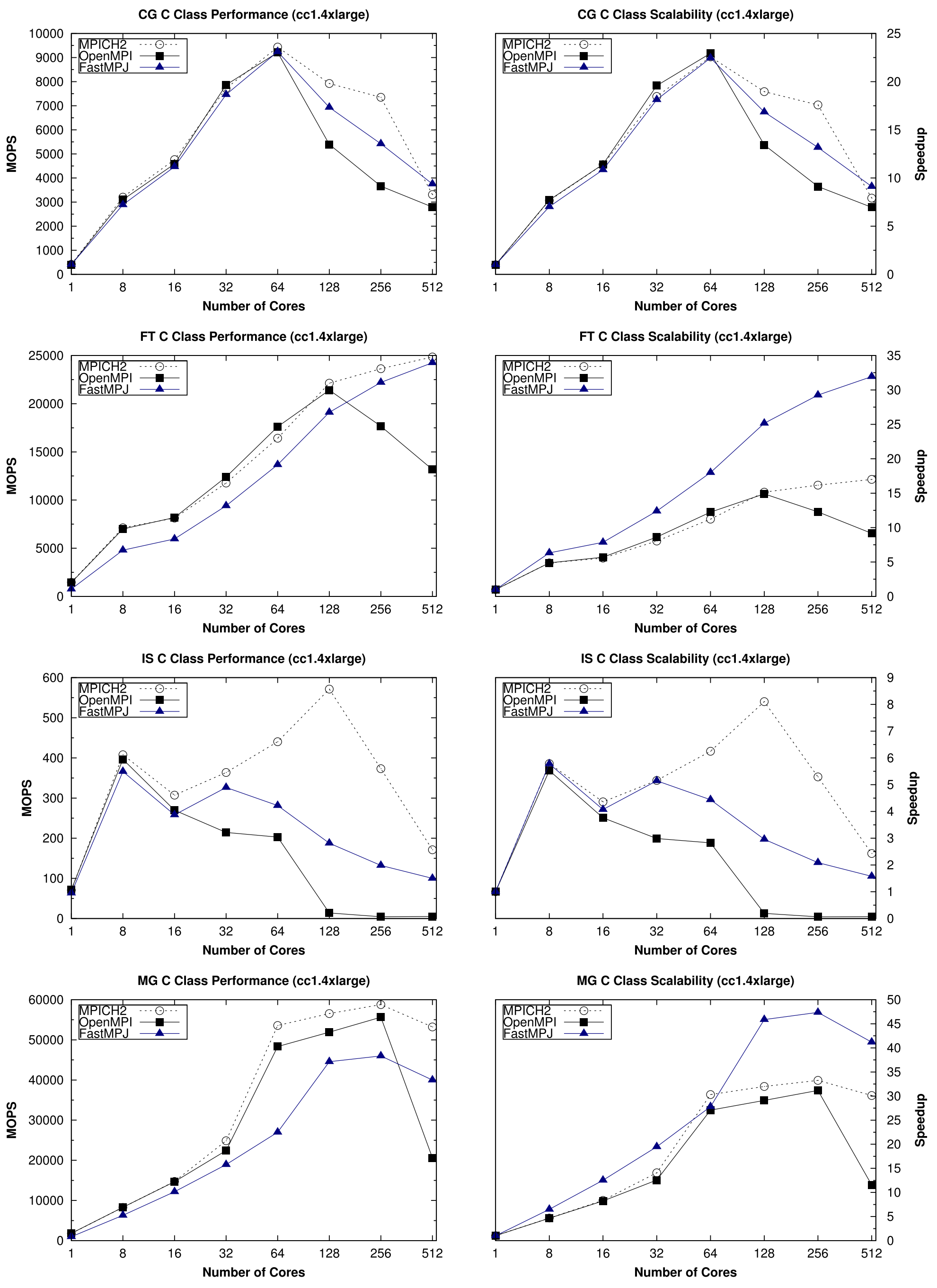


**Fig. 6.** NPB kernels performance and scalability on Amazon EC2 CC1 instances.

scalability of intra-VM shared memory communications. However, when using more than one VM, the evaluated kernels scale poorly, experiencing important performance penalties due to the network virtualization overhead. The poorest scalability has been obtained by FT kernel on CC2 instances, CG on CC1, and IS both on CC1 and CC2.

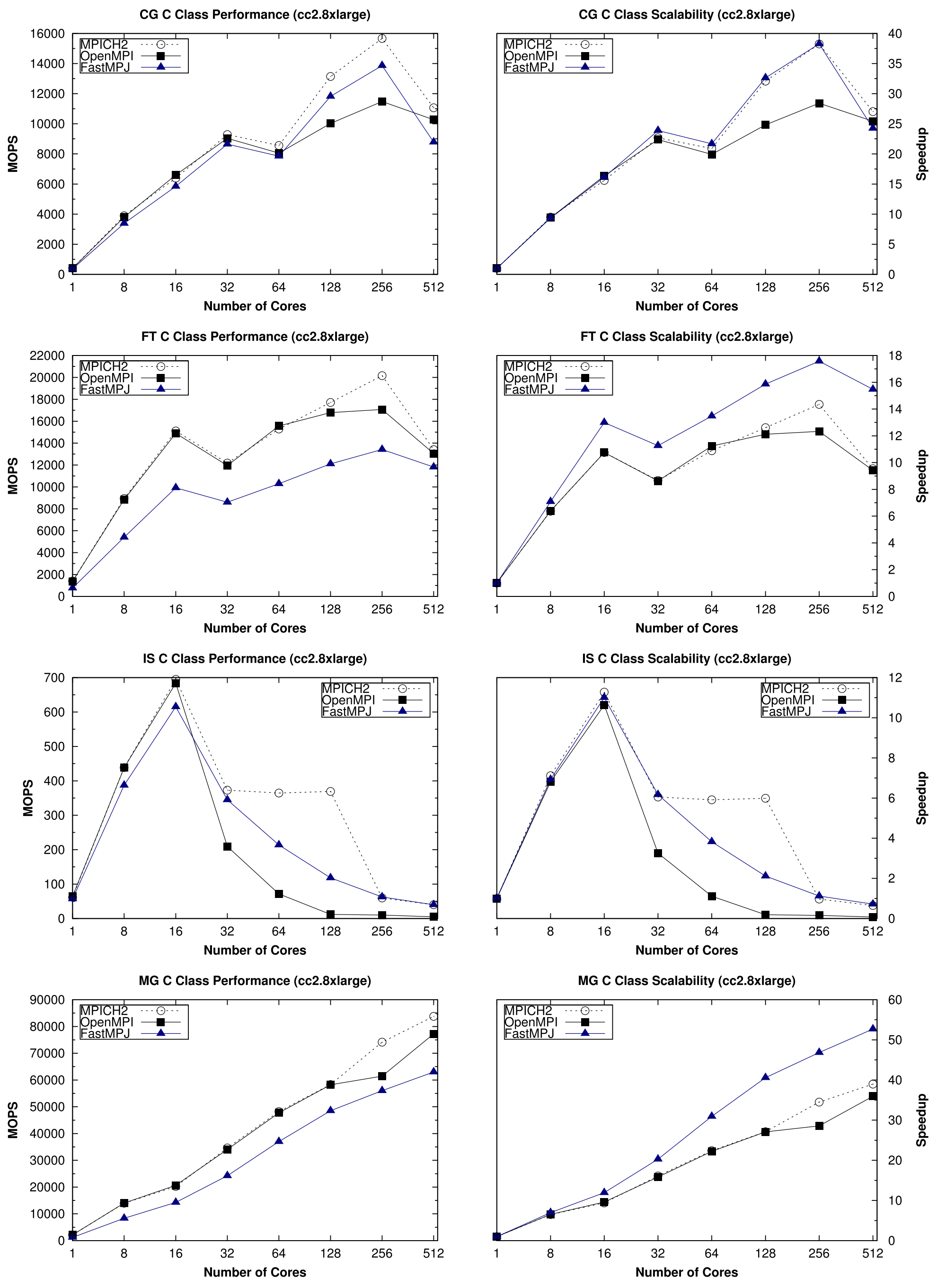


**Fig. 7.** NPB kernels performance and scalability on Amazon EC2 CC2 instances.

CG kernel, characterized by multiple point-to-point data movements, achieves on CC1 its highest speedup value of 22 using 64 cores, dropping dramatically its performance from that point on as it has to rely on inter-VM communications, where the network virtualization overhead represents the main performance bottleneck. CG obtains higher performance on CC2 instances, a speedup of 40 on 256 cores, although its parallel efficiency is very poor in this case, below 16%. FT kernel achieves a limited

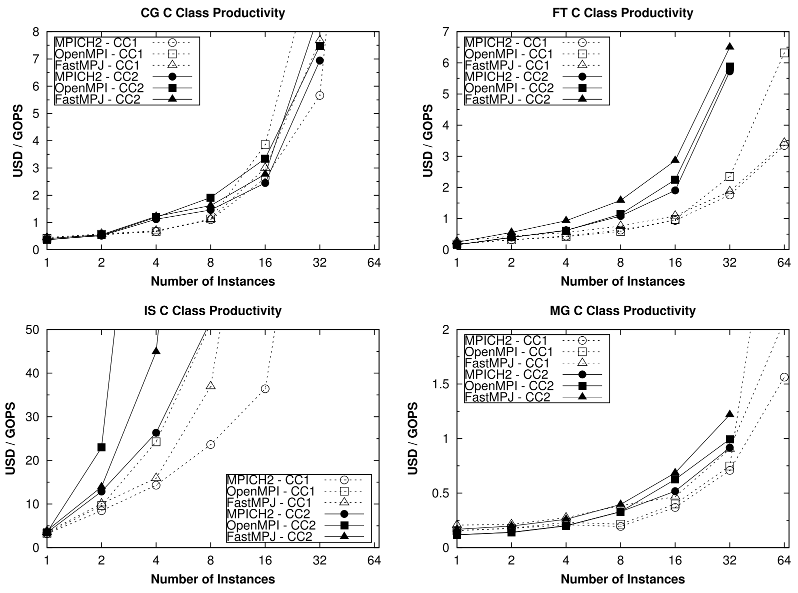


**Fig. 8.** NPB kernels productivity on Amazon EC2 CC instances.

scalability on CC1 whereas it does not scale at all on CC2. In fact, FT shows similar results on CC2 using a single instance (16 cores) and 32 instances (512 cores). The reason for this behavior is that FT uses extensively Alltoall primitives which access massively the interconnection network, increasing significantly the network overhead imposed by the hypervisor by having 8/16 processes on each instance accessing the paravirtualized NIC.

IS kernel is a communication-intensive code whose scalability greatly depends on the performance of the Alltoall(v)/Allreduce primitives and point-to-point communication start-up latency. Thus, this code only scales when using a single VM thanks to the high performance of intra-VM transfers, whereas it suffers a significant slowdown when using more than one VM, both for CC1 and CC2 instance types. Finally, MG kernel is the less communication-intensive code under evaluation and for this reason it presents the highest scalability on Amazon EC2, especially on CC2 instances, achieving a maximum speedup value of 53 for FastMPJ.

The analysis of scalability has revealed an important issue: the high start-up latencies and limited bandwidths imposed by the paravirtualized access to the NIC limit severely the scalability of communication-intensive codes. The presence of processors with higher computational power in the new CC2 instances partly alleviates this issue for some codes but also aggravates the situation when communication-intensive collective operations (e.g., Alltoall or Allreduce) are involved. Thus, a more efficient I/O network virtualization support, such as a direct access to the NIC in virtualized environments, is needed together with techniques that reduce the number of communicating processes accessing simultaneously through the paravirtualized NIC.

### *5.4. HPC kernels cost analysis*

The use of a public cloud infrastructure like Amazon involves a series of associated costs that have to be taken into account. In order to ease the analysis of the cost of Amazon EC2 resources for HPC we present in Fig. 8 the productivity in terms of USD per GOPS (Giga Operations Per Second) of the already evaluated NPB codes. This proposed metric is independent of the number of cores being used. Moreover, these kernels ran on-demand CC1 and CC2 instances. The use of spot instances from the spot market [42] enables bidding for unused Amazon EC2 capacity, which can significantly reduce costs as they provide the same performance as on-demand instances but at lower cost. However, the spot price fluctuates periodically depending on the supply and the demand for spot instance capacity. Moreover, the provision of the requested number of instances, especially when it is a high number, something that it can be expected for an HPC application, is not fully guaranteed unless the resources are invoked as on-demand instances.

CC2 instances provide twice the number of physical cores and 2.6 times more memory than CC1. Additionally, CC2 instances are based on Sandy Bridge, a more modern microarchitecture than Nehalem, the technology behind CC1, which results in more performance per core. However, CC2 does not provide communication-intensive HPC applications with a performance increase proportional to its higher features due to the network overhead which severely limits the scalability of these applications. If we take into account the cost of each instance (as of May 2012), $1.30 for CC1 and $2.40 for CC2, it can be easily concluded with the aid of Fig. 8 that the use of CC2 instances is generally more expensive than the use of CC1 instances and therefore it is worth recommending CC2 only for applications with high memory requirements that cannot be executed on CC1.

### *5.5. Impact of process allocation strategies*

It seems intuitive that after allocating a certain number of EC2 instances the best option would be running as many processes per instance as the cores the instance has, thus fully using the paid resources. However, in terms of performance, this option is not going to be always the best.

Fig. 9 presents performance results of the NPB kernels with CC1 instances (left graphs) and CC2 ones (right graphs) using only one process per instance (labeled as "1 ppi") until it reaches the maximum number of available instances, and posteriorly two, four, and finally eight processes per instance. This scheduling strategy,

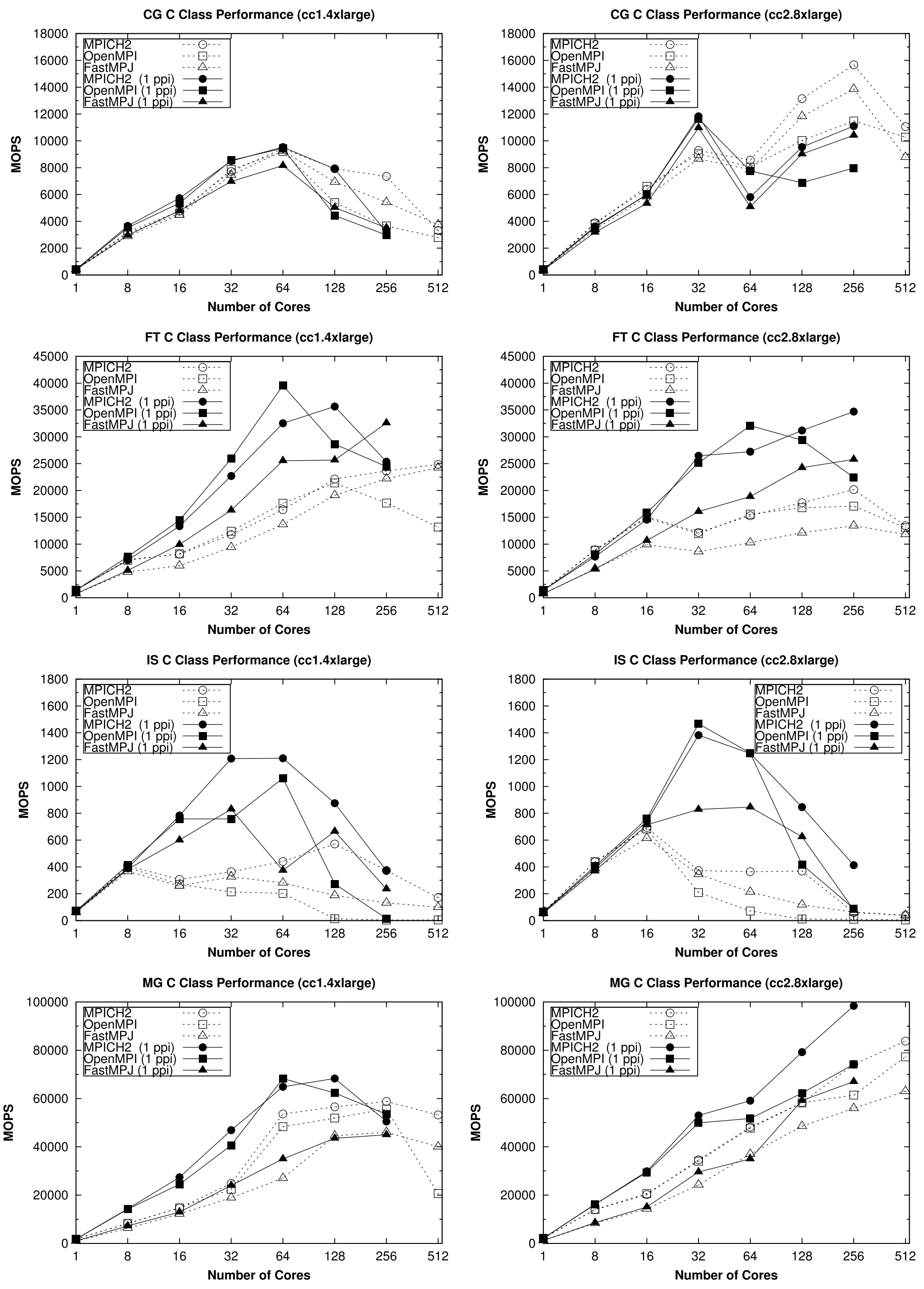


**Fig. 9.** NPB kernels performance on Amazon EC2 CC instances (ppi = processes per instance).

which is significantly much more expensive than the previous one, is able to obtain higher performance in the evaluated kernels, except for CG where this strategy seems to perform similarly (for CC1) or even slightly worse (for CC2). For the other kernels this scheduling achieves up to 7 times better results for IS on CC2 instances (OpenMPI), 3.3 times on CC1 instances (MPICH2) or 2.3 for FT on CC1 instances (OpenMPI). Here FT and IS take full advantage of this approach as the use of less processes per

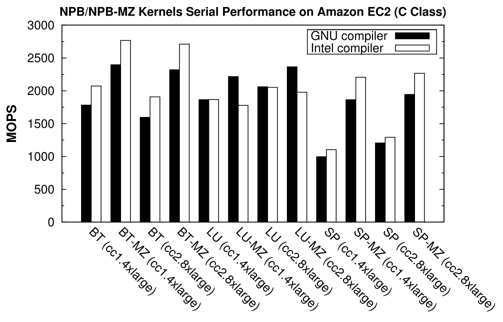


**Fig. 10.** NPB/NPB-MZ kernels serial performance.

instance alleviates the overhead introduced by the hypervisor in collective communications, frequently used in these kernels. This strategy can be very useful when there is a strong constraint about the execution time of a particular application, such as meeting a deadline, in exchange for much higher costs.

### 5.6. *Using several levels of parallelism*

The popularity of hybrid shared/distributed memory architectures, such as clusters of multi-core processors, is currently leading to the support of several levels of parallelism in many prominent scientific problems. The NPB suite is not an exception, an existing NPB Multi-Zone (NPB-MZ) implementation takes advantage of two-level parallelism through hybrid MPI+OpenMP codes. The NPB-MZ contains three applications, the Lower–Upper Symmetric Gauss–Seidel (LU, but limited to 16 MPI processes in the MZ version), Scalar Penta-diagonal (SP), and Block Tri-diagonal (BT) applications, whose serial performance is shown in Fig. 10. FastMPJ results are not shown due to the lack of an NPB-MZ implementation in Java.

Fig. 11 shows performance results, in terms of speedups, of the hybrid NPB-MZ Class C workloads, together with their NPB counterparts. The NPB-MZ codes have been executed with the

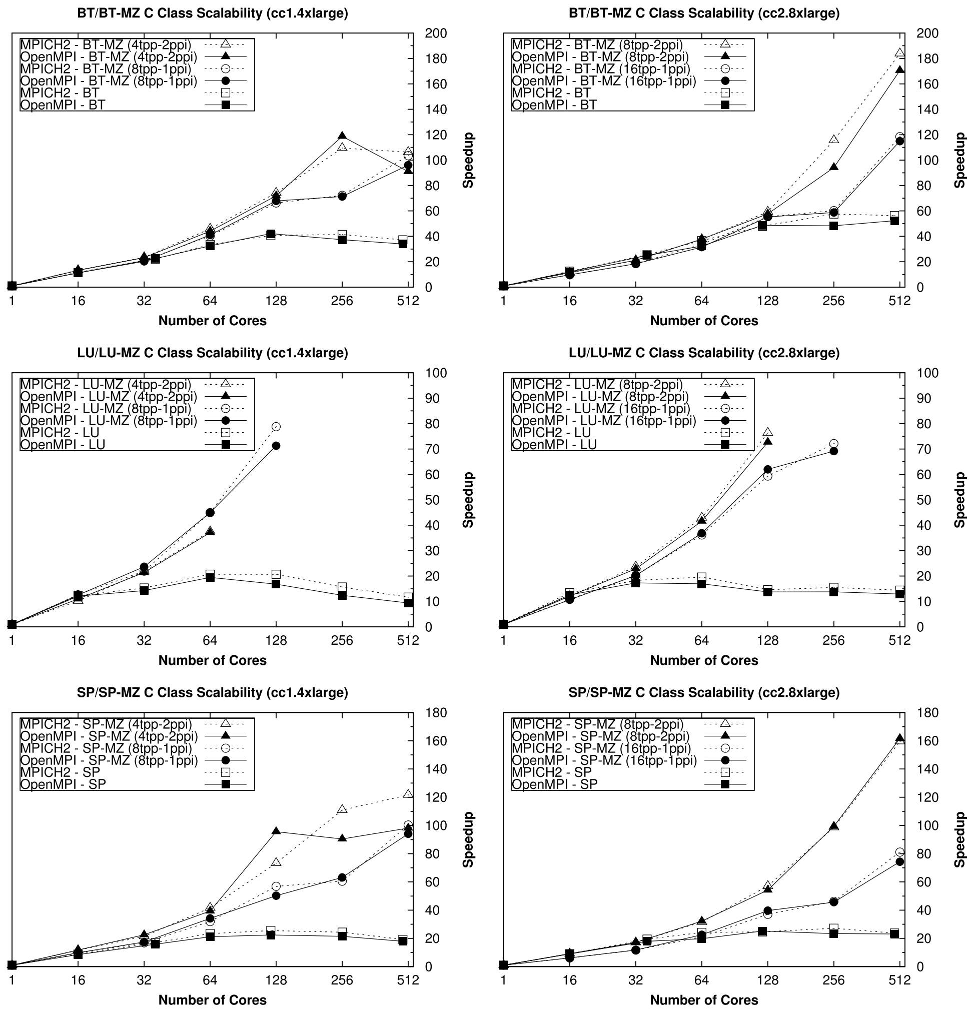


**Fig. 11.** NPB/NPB-MZ kernels scalability (tpp = threads per process — ppi = processes per instance).

following two configurations: (1) a single MPI process per instance with as many OpenMP threads as cores of the instance (8 and 16 for CC1 and CC2, respectively); and (2) two MPI processes per instance with as many OpenMP threads as cores of each processor (4 and 8 for CC1 and CC2, respectively). Additional configurations have also been evaluated, such as the use of 4 MPI processes per instance and 2 and 4 threads for CC1 and CC2, respectively, but the results obtained were worse.

These results have revealed some important facts: (1) although these codes (BT, LU and SP) are not as communication-intensive as the previously evaluated kernels (CG, FT, IS and MG), the pure MPI versions present poor scalability on CC instances (the maximum speedup achieved is 60 for BT on CC2 using 512 cores); (2) NPB-MZ kernels can overcome this issue and outperform significantly NPB kernels (the maximum reported speedup is 185 on 512 cores for BT-MZ on CC2). Thus, as the message transfers through the network are reduced to only one (or two) MPI processes communicating per instance, the overhead caused by the paravirtualized access to the NIC is drastically decreased; (3) BT and SP obtain the best performance using two MPI processes per instance, which suggests that the right balance between MPI processes and OpenMP threads has to be found for each particular application; and (4) CC2 significantly outperforms CC1 for BT-MZ and SP-MZ in the best configuration (for LU-MZ similar results are obtained) as their overall performance does not rely heavily on the communication performance. Unlike NPB kernels, BT-MZ and SP-MZ present a better cost/performance ratio on CC2 than on CC1.

## 6. Conclusions

The scalability of HPC applications on public cloud infrastructures relies heavily on the performance of communications, which depends both on the network fabric and its efficient support in the virtualization layer. Amazon EC2 Cluster Compute (CC) platform provides powerful HPC resources with access to a high-speed network (10 Gigabit Ethernet), although without a proper I/O virtualization support as these resources rely on a paravirtualized access to the NIC.

The contributions of this paper are: (1) it has evaluated the performance of communications on Amazon EC2 CC platform, both 10 Gigabit Ethernet and shared memory transfers for CC1 and CC2 instances; (2) it has assessed the scalability of representative message-passing codes (NPB) using up to 512 cores; (3) it has revealed that the new CC2 instances, despite providing more computational power and slightly better point-to-point communication performance, present poorer scalability than CC1 instances for collective-based communication-intensive applications; (4) the use of CC1 instances is generally more cost-effective than relying on CC2 instances; (5) it is possible to achieve higher scalability running only a single process per instance, thus reducing the communications performance penalty in the access to the network; (6) finally, the use of multiple levels of parallelism has been proposed, combining message-passing with multithreading, as the most scalable and cost-effective option for running HPC applications on the Amazon EC2 CC platform.

## Acknowledgment

This work was funded by the Ministry of Science and Innovation of Spain under Project TIN2010-16735, an FPU Grant AP2010-4348, and by an Amazon Web Services (AWS) LLC research grant.

## References

[1] R. Buyya, C.S. Yeo, S. Venugopal, J. Broberg, I. Brandic, Cloud computing and emerging IT platforms: vision, hype, and reality for delivering computing as the 5th utility, Future Generation Computer Systems 25 (6) (2009) 599–616.
[2] L. Rodero-Merino, L.M. Vaquero, V. Gil, F. Galán, J. Fontán, R.S. Montero, I.M. Llorente, From infrastructure delivery to service management in clouds, Future Generation Computer Systems 26 (8) (2010) 1226–1240.
[3] Amazon Web Services LLC, Amazon elastic compute cloud (Amazon EC2), last visited: May 2012. http://aws.amazon.com/ec2.
[4] C. Evangelinos, C.N. Hill, Cloud computing for parallel scientific HPC applications: feasibility of running coupled atmosphere-ocean climate models on Amazon's EC2, in: Proc. 1st Workshop on Cloud Computing and Its Applications, CCA'08, Chicago, IL, USA, 2008, pp. 1–6.
[5] E. Walker, Benchmarking Amazon EC2 for high-performance scientific computing, ;login: The usenix journal 33 (5) (2008) 18–23.
[6] J. Ekanayake, G.C. Fox, High performance parallel computing with clouds and cloud technologies, in: Proc. 1st Intl. Conference on Cloud Computing, CLOUDCOMP'09, Munich, Germany, 2009, pp. 20–38.
[7] Amazon Web Services LLC, High performance computing using amazon EC2, last visited: May 2012. http://aws.amazon.com/ec2/hpc-applications/.
[8] D.H. Bailey, et al., The NAS parallel benchmarks, International Journal of High Performance Computing Applications 5 (3) (1991) 63–73.
[9] H. Jin, R.F. Van der Wijngaart, Performance characteristics of the multi-zone NAS parallel benchmarks, Journal of Parallel and Distributed Computing 66 (2006) 674–685.
[10] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, A. Warfield, Xen and the art of virtualization, in: Proc. 19th ACM Symposium on Operating Systems Principles, SOSP'03, Bolton Landing, NY, USA, 2003, pp. 164–177.
[11] A. Whitaker, M. Shaw, S.D. Gribble, Denali: lightweight virtual machines for distributed and networked applications, Technical Report 02-02-01, University of Washington, USA, 2002.
[12] D. Abramson, et al., Intel virtualization technology for directed I/O, Intel Technology Journal 10 (3) (2006) 179–192.
[13] AMD virtualization technology (AMD-V), last visited: May 2012. http://sites.amd.com/us/business/it-solutions/virtualization/Pages/amd-v.aspx.
[14] D. Ghoshal, R.S. Canon, L. Ramakrishnan, I/O performance of virtualized cloud environments, in: Proc. 2nd Intl. Workshop on Data Intensive Computing in the Clouds, DataCloud-SC'11, Seattle, WA, USA, 2011, pp. 71–80.
[15] K. Fraser, S. Hand, R. Neugebauer, I. Pratt, A. Warfield, M. Williamson, Safe hardware access with the xen virtual machine monitor, in: Proc. 1st Workshop on Operating System and Architectural Support for the On-Demand IT Infrastructure, OASIS'04, Boston, MA, USA, 2004.
[16] A. Menon, A.L. Cox, W. Zwaenepoel, Optimizing network virtualization in Xen, in: Proc. USENIX'06 Annual Technical Conference, Boston, MA, USA, 2006, p. 2.
[17] J.R. Santos, Y. Turner, G. Janakiraman, I. Pratt, Bridging the gap between software and hardware techniques for I/O virtualization, in: Proc. USENIX'08 Annual Technical Conference, Boston, MA, USA, 2008, pp. 29–42.
[18] H. Raj, K. Schwan, High performance and scalable I/O virtualization via self-virtualized devices, in: Proc. 16th IEEE Intl. Symposium on High-Performance Distributed Computing, HPDC'07, Monterey, CA, USA, 2007, pp. 179–188.
[19] K. Mansley, G. Law, D. Riddoch, G. Barzini, N. Turton, S. Pope, Getting 10 Gb/s from Xen: safe and fast device access from unprivileged domains, in: Proc. Workshop on Virtualization/Xen in High-Performance Cluster and Grid Computing, VHPC'07, Rennes, France, 2007, pp. 224–233.
[20] B.-A. Yassour, M. Ben-Yehuda, O. Wasserman, Direct device assignment for untrusted fully-virtualized virtual machines, Tech. Rep., 2008.
[21] PCI SIG. I/O virtualization, last visited: May 2012. http://www.pcisig.com/specifications/iov/.
[22] J. Liu, Evaluating standard-based self-virtualizing devices: a performance study on 10 GbE NICs with SR-IOV support, in: Proc. 24th IEEE Intl. Parallel and Distributed Processing Symposium, IPDPS'10, Atlanta, GA, USA, 2010, pp. 1–12.
[23] A. Gordon, N. Amit, N. Har'El, M. Ben-Yehuda, A. Landau, D. Tsafrir, A. Schuster, ELI: bare-metal performance for I/O virtualization, in: Proc. 17th Intl. Conference on Architectural Support for Programming Languages Operating Systems, ASPLOS'12, London, UK, 2012.
[24] C. Vecchiola, S. Pandey, R. Buyya, High-performance cloud computing: a view of scientific applications, in: Proc. 10th Intl. Symposium on Pervasive Systems, Algorithms, and Networks, ISPAN'09, Kaoshiung, Taiwan, ROC, 2009, pp. 4–16.
[25] G. Juve, E. Deelman, K. Vahi, G. Mehta, B. Berriman, B.P. Berman, P. Maechling, Data sharing options for scientific workflows on Amazon EC2, in: Proc. 22th ACM/IEEE Conference on Supercomputing, SC'10, New Orleans, LA, USA, 2010, pp. 1–9.
[26] A. Iosup, S. Ostermann, N. Yigitbasi, R. Prodan, T. Fahringer, D. Epema, Performance analysis of cloud computing services for many-tasks scientific computing, IEEE Transactions on Parallel and Distributed Systems 22 (2011) 931–945.
[27] S. Gogouvitis, K. Konstanteli, S. Waldschmidt, G. Kousiouris, G. Katsaros, A. Menychtas, D. Kyriazis, T. Varvarigou, Workflow management for soft real-time interactive applications in virtualized environments, Future Generation Computer Systems 28 (1) (2012) 193–209.
[28] J. Napper, P. Bientinesi, Can cloud computing reach the TOP500? in: Proc. Combined Workshops on UnConventional High Performance Computing Workshop Plus Memory Access Workshop, UCHPC-MAW'09, Ischia, Italy, 2009, pp. 17–20.
[29] K.R. Jackson, L. Ramakrishnan, K. Muriki, S. Canon, S. Cholia, J. Shalf, H.J. Wasserman, N.J. Wright, Performance analysis of high performance computing applications on the amazon web services cloud, in: Proc. 2nd IEEE Intl. Conference on Cloud Computing Technology and Science, CloudCom'10, Indianapolis, USA, 2010, pp. 159–168.
[30] A.G. Carlyle, S.L. Harrell, P.M. Smith, Cost-effective HPC: the community or the cloud? in: Proc. 2nd IEEE Intl. Conference on Cloud Computing Technology and Science, CloudCom'10, Indianapolis, USA, 2010, pp. 169–176.

[31] N. Regola, J.C. Ducom, Recommendations for virtualization technologies in high performance computing, in: Proc. 2nd IEEE Intl. Conference on Cloud Computing Technology and Science, CloudCom'10, Indianapolis, USA, 2010, pp. 409–416.
[32] L. Ramakrishnan, R.S. Canon, K. Muriki, I. Sakrejda, N.J. Wright, Evaluating interconnect and virtualization performance for high performance computing, in: Proc. 2nd Intl. Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems, PMBS'11, Seattle, WA, USA, 2011, pp. 1–2.
[33] Y. Zhai, M. Liu, J. Zhai, X. Ma, W. Chen, Cloud versus in-house cluster: evaluating amazon cluster compute instances for running MPI applications, in: Proc. 23th ACM/IEEE Conference on Supercomputing, SC'11, State of the Practice Reports, Seattle, WA, USA, 2011, pp. 1–10.
[34] V. Mauch, M. Kunze, M. Hillenbrand, High performance cloud computing, Future Generation Computer Systems (2012), http://dx.doi.org/10.1016/j.future.2012.03.011.
[35] G. Neiger, A. Santoni, F. Leung, D. Rodgers, R. Uhlig, Intel virtualization technology: hardware support for efficient processor virtualization, Intel Technology Journal 10 (3) (2006) 167–178.
[36] AMD I/O virtualization technology (IOMMU) specification, last visited: May 2012. http://support.amd.com/us/Processor_TechDocs/34434-IOMMU-Rev_1.26_2-11-09.pdf.
[37] Open MPI Website, last visited: May 2012. http://www.open-mpi.org/.
[38] MPICH2 Website, last visited: May 2012. http://www.mcs.anl.gov/research/projects/mpich2/.
[39] G.L. Taboada, J. Touriño, R. Doallo, F-MPJ: scalable java message-passing communications on parallel systems, Journal of Supercomputing 60 (1) (2012) 117–140.
[40] M. Baker, B. Carpenter, MPJ: a proposed Java message passing API and environment for high performance computing, in: Proc. 2nd International Workshop on Java for Parallel and Distributed Computing, JavaPDC'00, Cancun, Mexico, 2000, pp. 552–559.
[41] D.A. Mallón, G.L. Taboada, J. Touriño, R. Doallo, NPB-MPJ: NAS parallel benchmarks implementation for message-passing in java, in: Proc. 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing, PDP'09, Weimar, Germany, 2009, pp. 181–190.
[42] W. Voorsluys, S.K. Garg, R. Buyya, Provisioning spot market cloud resources to create cost-effective virtual clusters, in: Proc. 11th Intl. Conference on Algorithms and Architectures for Parallel Processing, ICA3PP'11, Melbourne, Australia, 2011, pp. 395–408.

**Roberto R. Expósito** received his B.S. (2010) and M.S. (2011) degrees in Computer Science from the University of A Coruña, Spain. Currently he is a Ph.D. student at the Department of Electronics and Systems at the University of A Coruña. His research interests are in the area of High Performance Computing (HPC), focused on message-passing communications on high-speed networks and cluster/grid/cloud computing.

**Guillermo L. Taboada** received his B.S. (2002), M.S. (2004) and Ph.D. (2009) degrees in Computer Science from the University of A Coruña, Spain. Currently he is an Associate Professor in the Department of Electronics and Systems at the University of A Coruña. His main research interest is in the area of High Performance Computing (HPC), focused on high-speed networks, programming languages for HPC, cluster/grid/cloud computing and, in general, middleware for HPC.

**Sabela Ramos** received her B.S. (2009) and M.S. (2010) degrees in Computer Science from the University of A Coruña, Spain. Currently she is a Ph.D. student at the Department of Electronics and Systems at the University of A Coruña. Her research interests are in the area of High Performance Computing (HPC), focused on message-passing communications on multicore architectures and cluster/grid/cloud computing.

**Juan Touriño** received his B.S. (1993), M.S. (1993) and Ph.D. (1998) degrees in Computer Science from the University of A Coruña, Spain. In 1993 he joined the Department of Electronics and Systems at the University of A Coruña, Spain, where he is currently a Full Professor of Computer Engineering and Director of the department. He has extensively published in the areas of compilers and programming languages for HPC, and parallel and distributed computing. He is coauthor of more than 120 technical papers on these topics.

**Ramón Doallo** received his B.S. (1987), M.S. (1987) and Ph.D. (1992) degrees in Physics from the University of Santiago de Compostela, Spain. In 1990 he joined the Department of Electronics and Systems at the University of A Coruña, Spain, where he became a Full Professor in 1999. He has extensively published in the areas of computer architecture, and parallel and distributed computing. He is coauthor of more than 140 technical papers on these topics.

# Chapter 8

# Analysis of I/O Performance on the Amazon EC2 Cloud

The content of this chapter corresponds to the following journal paper:

**- Title:** Analysis of I/O performance on an Amazon EC2 cluster compute and high I/O platform

**- Authors:** Roberto R. Expósito, Guillermo L. Taboada, Sabela Ramos, Jorge González-Domínguez, Juan Touriño, Ramón Doallo

**- Journal:** Journal of Grid Computing

**- Editorial:** Springer Science+Business Media

**- ISSN:** 1570-7873

**- Year:** 2013

**- Volume(Number):Pages:** 11(4):613–631

**- DOI:** 10.1007/s10723-013-9250-y

The final publication is available at http://link.springer.com/article/10.1007%2Fs10723-013-9250-y. A copy of the accepted paper has been included next.

# Analysis of I/O Performance on an Amazon EC2 Cluster Compute and High I/O Platform

**Roberto R. Expósito · Guillermo L. Taboada · Sabela Ramos · Jorge González-Domínguez · Juan Touriño · Ramón Doallo**

**Abstract** Cloud computing is currently being explored by the scientific community to assess its suitability for High Performance Computing (HPC) environments. In this novel paradigm, compute and storage resources, as well as applications, can be dynamically provisioned on a pay-per-use basis. This paper presents a thorough evaluation of the I/O storage subsystem using the Amazon EC2 Cluster Compute platform and the recent High I/O instance type, to determine its suitability for I/O-intensive applications. The evaluation has been carried out at different layers using representative benchmarks in order to evaluate the low-level cloud storage devices available in Amazon EC2, ephemeral disks and Elastic Block Store (EBS) volumes, both on local and distributed file systems. In addition, several I/O interfaces (POSIX, MPI-IO and HDF5) commonly used by scientific workloads have also been assessed. Furthermore, the scalability of a representative parallel I/O code has also been analyzed at the application level, taking into account both performance and cost metrics. The analysis of the experimental results has shown that available cloud storage devices can have different performance characteristics and usage constraints. Our comprehensive evaluation can help scientists to increase significantly (up to several times) the performance of I/O-intensive applications in Amazon EC2 cloud. An example of optimal configuration that can maximize I/O performance in this cloud is the use of a RAID 0 of 2 ephemeral disks, TCP with 9,000 bytes MTU, NFS async and MPI-IO on the High I/O instance type, which provides ephemeral disks backed by Solid State Drive (SSD) technology.

**Keywords** Cloud computing · Virtualization · I/O performance evaluation · Network File System (NFS) · MPI-IO · Solid State Drive (SSD)

R. R. Expósito (✉) · G. L. Taboada · S. Ramos · J. González-Domínguez · J. Touriño · R. Doallo
Department of Electronics and Systems, University of A Coruña, A Coruña, Spain
e-mail: rreye@udc.es

G. L. Taboada
e-mail: taboada@udc.es

S. Ramos
e-mail: sramos@udc.es

J. González-Domínguez
e-mail: jgonzalezd@udc.es

J. Touriño
e-mail: juan@udc.es

R. Doallo
e-mail: doallo@udc.es

## 1 Introduction

Data management is a critical component of many current scientific computing workloads, which are generating very large data sets, contributing significantly to the consolidation of the so-called current *big data* era. These applications often require a high number of computing resources to perform large-scale experiments into a reasonable time frame, and these needs have been typically addressed with dedicated High Performance Computing (HPC) infrastructures such as clusters or big supercomputers. In this scenario, scientific applications can be sensitive to CPU power, memory bandwidth/capacity, network bandwidth/latency as well as the performance of the I/O storage subsystem.

The cloud computing paradigm is a relatively recent computing model where dynamically scalable and often virtualized resources are provided as a service over the Internet. This novel paradigm has gained significant popularity in many areas, including the scientific community. The combination of this model together with the rich set of cloud infrastructure services can offer a feasible alternative to traditional servers and computing clusters, saving clients from the expense of building an in-house datacenter that is provisioned to support the highest predicted load. With cloud-based technologies, scientists can have easy access to large distributed infrastructures and completely customize their execution environment, thus providing the perfect setup for their experiments. In addition, the interest in the use of public clouds for HPC applications increases as their availability, computational power, price and performance improves.

Amazon Web Services (AWS) is nowadays the leading public Infrastructure-as-a-Service (IaaS) cloud provider in terms of number of users, allowing resources in their data centers to be rented on-demand through Elastic Compute Cloud (EC2) service [7]. By means of virtualization technologies, EC2 allows scalable deployment of applications by providing a web service through which a user can, among other tasks, boot straightforwardly an Amazon Machine Image (AMI) into a custom Virtual Machine (a VM or “instance”). This on-demand allocation of resources provides a new dimension for HPC due to the elastic capability of the cloud computing model, which additionally can provide both cost-effective and energy-efficient solutions [31].

Amazon EC2 offers a cloud infrastructure, the Cluster Compute (CC) platform, which specifically targets HPC environments [9]. The CC platform is a family of several instance types which are intended to be well suited for large-scale scientific experiments and HPC applications by offering physical node allocation (a single VM per node), powerful and up-to-date CPUs and GPUs, and an improved interconnection network (10 Gigabit Ethernet). Additionally, the High I/O instance type shares the same characteristics as the CC instances with enhanced storage performance providing Solid State Drives (SSD) disks. Using these instance types customers can expedite their HPC workloads on elastic resources as needed, adding and removing compute resources to meet the size and time requirements for their specific workloads. An example of the extent and magnitude of Amazon EC2 is the self-made cluster that, with only a small portion of its resources (about 1,000 CC instances), ranks #102 in the latest Top 500 list (November 2012) [1].

This paper evaluates the I/O storage subsystem on the Amazon EC2 CC platform to determine its suitability for scientific applications with high I/O performance requirements. Moreover, the evaluation includes, for the first time to the best of our knowledge, the High I/O instance type, which has been recently released in July 2012. This instance type is intended to provide very high instance storage I/O performance, as it is backed by SSD disks, which is the main differential characteristic of this resource, but it also provides high levels of CPU, memory and network performance as CC instances.

In this evaluation, experiments at different levels are conducted. Thus, several micro-benchmarks are used to evaluate different cloud low-level storage devices available in CC instances, ephemeral disks and Elastic Block Store (EBS) volumes [6], both at the local and distributed file system levels. In addition, common middleware libraries such as HDF5 [4] and MPI-IO [35], which are directly implemented on top of file systems, have also been assessed as scientific workloads usually rely on them to perform I/O. Finally, the scalability of a representative parallel I/O code implemented on top of MPI-IO, the BT-IO kernel [38] from the NAS Parallel Benchmarks (NPB) suite [24], has also been analyzed at the application level both in terms of performance and cost effectiveness.

The paper is organized as follows: Section 2 describes the related work. Section 3 presents an overview of the storage system of the Amazon EC2 public cloud. Section 4 introduces the experimental configuration, both hardware and software, and the methodology of the evaluation conducted in this work. Section 5 analyzes the I/O performance results of the selected benchmarks/kernels on Amazon EC2 CC and High I/O instances. Finally, our conclusions are presented in Section 6.

## 2 Related Work

In recent years there has been a spur of research activity in assessing the performance of virtualized resources and cloud computing environments [18, 25, 30, 40, 41]. The majority of recent studies have evaluated Amazon EC2 to examine the feasibility of using public clouds for high performance or scientific computing, but with different focuses.

Some previous works have shown that computationally-intensive codes present little overhead when running on virtualized environments, whereas communication-intensive applications tend to perform poorly [12, 13, 23, 27, 37], especially tightly-coupled parallel applications such as MPI [3] jobs. This is primarily due to the poor virtualized network performance, processor sharing among multiple users and the use of commodity interconnection technologies (Gigabit Ethernet). In order to overcome this performance bottleneck, Amazon EC2 offers the Cluster Compute (CC) platform, which introduces several HPC instance types, cc1.4xlarge and cc2.8xlarge, abbreviated as CC1 and CC2, respectively, in addition to the recent High I/O instance type (hi1.4xlarge, abbreviated as HI1) which provides SSD disks. Thus, Sun et al. [34] relied on 16 CC1 instances for running the Lattice Optimization and the HPL benchmark. The main conclusion derived from the results is that MPI codes, especially those which are network latency bound, continue to present poor scalability. Ramakrishnan et al. [29] stated that virtualized network is the main performance bottleneck on Amazon EC2 after analyzing the communication overhead on CC1 instances. Mauch et al. [21] presented an overview of the current state of HPC IaaS offerings and suggested how to use InfiniBand in a private virtualized environment, showing some HPL benchmark results using a single instance of CC1 and CC2 instance types. Finally, our previous work [14] has stated that CC1 and CC2 instances are able to achieve reasonable scalable performance in parallel applications, especially when hybrid shared/distributed memory programming paradigms, such as MPI+OpenMP, are used in order to minimize network communications.

However, most of the previous work is focused on computation and communication, whereas there are very little works that have investigated I/O and storage performance. Some of them analyzed the suitability of running scientific workflows in the cloud [19, 26, 36], showing that it can be a successful option as these workloads are loosely-coupled parallel applications. Thus, Juve et al. [19] studied the performance and cost of different storage options for scientific workflows on Amazon EC2, although regarding CC platform they only evaluated three workflows on the CC1 instance type. Vecchiola et al. [36] ran an fMRI brain imaging workflow on Amazon EC2 using the object-based Amazon Simple Storage Service (S3) [8] for storage, and analyzed the cost varying the number of nodes. In [5], Abe and Gibson provide S3-like storage access on top of PVFS [10] on an open-source cloud infrastructure. Palankar et al. [28] assessed the feasibility of using Amazon S3 for scientific Grid computing. Zhai et al. [42] conducted a comprehensive evaluation of MPI applications on Amazon EC2 CC platform, revealing a significant performance increase compared to previous evaluations on non-CC instances. They also reported some experimental results of storage performance, but limited to ephemeral devices (local disks) without RAID and using only CC1 instances. In [32], the storage and network performance of the Eucalyptus cloud computing framework is analyzed, confronted with some results from one large instance type of Amazon EC2. Ghoshal et al. [16] compared I/O performance on two cloud platforms, Amazon EC2 and Magellan, using the IOR benchmark on the CC1 instance type. Their study is limited to the file system level, so RAID configurations as well as the performance of I/O interfaces are not taken into account. Finally, Liu et al. [20] ran two parallel applications (BT-IO and POP) on CC1 instances using ephemeral disks, both for NFS and PVFS file systems. Their results show that cloud-based clusters enable users to build per-application parallel file systems, as a single parallel I/O solution can not satisfy the needs of all applications.

In addition, many current applications (e.g., data mining, social network analysis) demand distributed computing frameworks such as MapReduce [11] and iMapReduce [43] to process massive data sets. An attractive feature of these frameworks is that they support the analysis of petabytes of data with the help of cloud computing without any prior investment in infrastructure, which has popularized big data analysis. For instance, Gunarathne et al. [17] present a new MapReduce runtime for scientific applications built using the Microsoft Azure cloud infrastructure. In [39], Yang et al. proposed a regression model for predicting relative performance of workloads under different Hadoop configurations with 87 % accuracy.

In this paper we evaluate the I/O performance of an Amazon EC2 CC and High I/O platform. Thus, we evaluated CC1 and CC2 instance types together with the most recent HI1 instances, so they can be directly compared. Moreover, we analyze the performance of the different low-level storage devices available on these instances (EBS volumes and ephemeral disks) in addition to the use of software RAID. Moreover, our study is carried out at several layers (storage devices, file systems, I/O interfaces and applications) using representative benchmarks/applications for each layer. Finally, we also take into account the costs associated with the use of a public cloud infrastructure, presenting a cost analysis at the application level.

## 3 Overview of Amazon EC2 CC and High I/O Instances

Amazon EC2 offers the CC platform which currently provides two HPC instance types. The Cluster Compute Quadruple Extra Large instances (cc1.4xlarge, abbreviated as CC1) and Cluster Compute Eight Extra Large instances (cc2.8xlarge, abbreviated as CC2) are resources with 23 and 60.5 GBytes of memory and 33.5 and 88 EC2 Compute Units (ECUs) for CC1 and CC2, respectively. According to Amazon WS one ECU provides the equivalent CPU capacity of a 1.0–1.2 GHz 2007 Opteron or 2007 Xeon processor.

In addition to the CC instances, Amazon has recently launched (July 2012) the High I/O Quadruple Extra Large instances (hi1.4xlarge, abbreviated as HI1). These instances have two SSD disks as local block storage, which is the main differential characteristic of this resource, in order to provide very high instance storage I/O performance. Moreover, HI1 instances also have powerful CPUs (35 ECUs) and a significant amount of memory (60.5 GBytes). The very high demand for these instances has caused that Amazon currently limits their use to only two simultaneous HI1 instances per user.

Regarding the hardware characteristics of these instances (see Table 1), the provider details the specific processor: two Intel Xeon X5570 quad-core Nehalem processors for CC1, hence 8 cores per CC1 instance, two Intel Xeon E5-2670 octa-core Sandy Bridge processors for CC2, hence 16 cores per CC2 instance, and two Intel Xeon E5620 quad-core Westmere processors for HI1, hence 8 cores per HI1 instance. Each instance will be allocated to users in a dedicated manner (a single VM per physical node), unlike the allocation mode in most other EC2 instance types (multiple VMs per physical node). These instances are interconnected via a high-speed network (10 Gigabit Ethernet), which is also among the main differential characteristics of these resources. Moreover, these instances can be launched within a placement

**Table 1** Description of the Amazon EC2 CC1, CC2 and HI1 instance types

| | CC1 (cc1.4xlarge) | CC2 (cc2.8xlarge) | HI1 (hi1.4xlarge) |
|---|---|---|---|
| Release date | July 2010 | November 2011 | July 2012 |
| CPU | 2 × Intel Xeon X5570 Nehalem-EP @2.93 GHz | 2 × Intel Xeon E5 2670 Sandy Bridge-EP @2.60 GHz | 2 × Intel Xeon E5620 Westmere-EP @2.40 GHz |
| ECUs | 33.5 | 88 | 35 |
| #Cores | 8 | 16 | 8 |
| Memory | 23 GBytes DDR3 | 60.5 GBytes DDR3 | 60.5 GBytes DDR3 |
| Ephemeral storage | 1.7 TBytes (2 HDD) | 3.4 TBytes (4 HDD) | 2 TBytes (2 SSD) |
| API name | cc1.4xlarge | cc2.8xlarge | hi1.4xlarge |
| Price (Linux) | $1.30 per hour | $2.40 per hour | $3.10 per hour |
| Interconnect | 10 Gigabit ethernet (full bisection bandwidth) | | |
| Virtualization | Xen HVM 64-bit platform | | |

group to obtain low latency, full bisection 10 Gbps bandwidth between them, but with the important restriction that only instances of the same type can be included in the same group.

Related with CC instances is the Cluster GPU Quadruple Extra Large Instance (cg1.4xlarge, abbreviated as CG1). Instances of this family provide exactly the same hardware capabilities than CC1 in terms of CPU power, memory capacity, I/O storage and network performance. The differential feature of CG1 instances is the provision of two GPUs for General-Purpose GPU computing (GPGPU). As the main goal of this work is the evaluation of the I/O storage subsystem, which is the same in CC1 and CG1, the study of CG1 instances has not been considered.

### 3.1 Storage System Overview of Amazon EC2

The virtual machines available in Amazon EC2 provide several storage solutions with different levels of abstraction, performance and access interfaces. Generally, each instance can access three types of storage: (1) the local block storage, known as ephemeral disk, where user data are lost once the instances are released (non-persistent storage); (2) off-instance Elastic Block Store (EBS), which are remote volumes accessible through the network that can be attached to an EC2 instance as block storage devices, and whose content is persistent; and (3) Simple Storage Service (S3), which is a distributed object storage system, accessed through a web service that supports both SOAP and REST. We have not considered S3 in our evaluation since, unlike ephemeral and EBS devices, it lacks general file system interfaces required by scientific workloads so that the use of S3 is not transparent to the applications, and also due to the poor performance shown by previous recent works [19].

The ephemeral and EBS storage devices have different performance characteristics and usage constraints. On the one hand, a CC1 instance can only mount up to two ephemeral disks of approximately 845 GBytes each one, resulting in a total capacity of 1,690 GBytes (see Table 1), whereas a CC2 instance can mount up to four disks of the aforementioned size, 845 GBytes, which represents an overall capacity of 3,380 GBytes. The new HI1 instances, as mentioned before, provide two SSD disks of 1,024 GBytes each one as ephemeral storage, for a total of 2,048 GBytes. On the other hand, the number of EBS volumes attached to instances can be almost unlimited, and the size of a single volume can range from 1 GByte to 1 TByte.

## 4 Experimental Configuration and Evaluation Methodology

The I/O performance evaluation of the Amazon platform has been conducted on CC1, CC2 and HI1 instance types. This evaluation consists of a micro-benchmarking with IOzone benchmark [2] of a local file system (ext3) on ephemeral disks and EBS volumes, using a single storage device as well as multiple storage devices combined in a single software RAID 0 (data striping) array using the *mdadm* utility, as this RAID level can improve both write and read performance without losing overall capacity. The IOzone benchmark has also been used to evaluate the performance of a representative distributed file system, NFS version 3, selected as it is probably the most commonly used network file system. Additionally, it remains as the most popular choice for small and medium-scale clusters.

After characterizing NFS with the IOzone benchmark, the performance of several I/O interfaces commonly used in scientific applications has been analyzed using the IOR benchmark [33] with multiple NFS clients. Three I/O interfaces were tested: (1) POSIX, which is the IEEE Portable Operating System Interface for computing environments that defines a standard way for an application to obtain basic services from the operating system, the I/O API among them; (2) MPI-IO [35], which is a comprehensive API with many features intended specifically to provide a high performance, portable, and parallel I/O interface to MPI programs; and (3) HDF5 [4], which is a data model, library, and file format for storing and managing data that supports an unlimited variety of datatypes, and is designed for flexible and efficient I/O and for high volume and complex data. Additionally, the scalability of the parallel BT-IO kernel [38], which is implemented on top of MPI-IO, has also been analyzed at the application level both in terms of performance and cost metrics.

The pattern of the I/O operations performed by the benchmarks/applications previously selected is essentially sequential, both for write and read operations. Random accesses, widely used in some scenarios such as database environments, are rarely used in HPC applications and scientific workloads. Most parallel I/O in HPC involves large-scale data movements, such as checkpointing the state of a running application, which makes that I/O access is mainly dominated by sequential operations.

CC1 and CC2 resources have been allocated simultaneously in the same placement group in order to obtain nearby instances, thus being able to benefit from the low latency and full bisection bandwidth of the 10 Gigabit Ethernet network. However, Amazon's current restriction for HI1 instances (only two HI1 instances can run simultaneously) has severely determined the evaluation. As one HI1 instance is needed to run the NFS server, the other HI1 instance can be used for the NFS clients. While this configuration is enough for the characterization of NFS with IOzone (as only one NFS client is used), for the IOR and BT-IO benchmarks it would be necessary to launch 8 HI1 instances in order to run up to 64 NFS clients. Therefore, for the evaluation of HI1 with these benchmarks, we opted for the use of one HI1 instance for the NFS server and CC1 and CC2 instances for the NFS clients. This fact implies that the HI1 server and CC1/CC2 clients can not be allocated in the same placement group, because only instances of the same type can be included in it, which can cause a loss in network performance. In order to minimize it, the HI1 server was always executed in the same clients' availability zone. Thus, all the experiments were performed in the us-east-1 region (North Virginia), within the us-east-1d availability zone.

Regarding software settings, the Amazon Linux AMI 2012.03 was selected as it is a supported and maintained Linux image provided by AWS for its usage on Amazon EC2 CC instances. This AMI, which comes with kernel 3.2.18, was customized with the incorporation of the previously described benchmarks: IOzone version 3.405 and IOR version 2.10.3. In addition, the MPI implementation of the NPB suite version 3.3 was also installed for the BT-IO kernel evaluation. The metrics considered for the evaluation of the BT-IO kernel are MOPS (Millions of Operations Per Second), which measures the operations performed in the benchmark (that differ from the CPU operations issued), and its corresponding I/O aggregated bandwidth measured in MBytes/sec. Moreover, the BT-IO Class C workload has been selected because it is the largest workload that can be executed in a single CC1 instance. The GNU C/Fortran 4.4.6 compiler has been used with -O3 flag, whereas the message-passing library selected for IOR and BT-IO evaluation is Open MPI [15], version 1.4.5. Finally, the performance results presented in this paper are the mean of the five measurements performed for each evaluated configuration. Unlike non-dedicated instance types, the dedicated (one VM per physiscal node) CC and HI1 instances present reduced variability in the performance measurements, so the standard deviation is not significant.

## 5 Evaluation of I/O Performance on Amazon EC2

This section presents an analysis of the performance results of the I/O subsystem on a cloud computing infrastructure, Amazon EC2 Cluster Compute platform and High I/O instances, using the representative benchmarks described in the previous section.

### 5.1 Local File System Performance

Figure 1 presents the maximum bandwidth obtained (measured in MBytes per second) for sequential read and write operations using the IOzone benchmark on EBS volumes and ephemeral disks (labeled as "EPH" on the graphs) on a single device formatted with the ext3 file system for CC1, CC2 and HI1 instances. The Linux buffer cache was bypassed using direct I/O (O_DIRECT flag) in order to get the real performance of the underlying storage devices.

All the configurations perform very similarly for the read operation, achieving all of them at least 100 MBytes/sec. The exception here is the SSD (ephemeral) disk on the HI1 instance, which

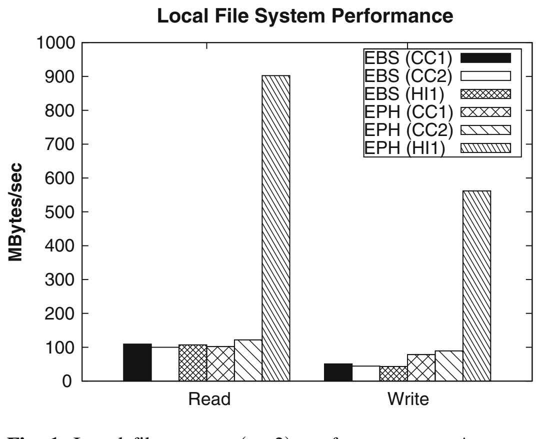


**Fig. 1** Local file system (ext3) performance on Amazon EC2 CC and high I/O instances

is clearly the best performer in this case as it is able to get 900 MBytes/sec, nine times better than the rest of configurations. Regarding the write operation, the results are more insightful as EBS volumes can obtain only a 40–50 % of the performance of ephemeral disks on CC1 and CC2 instances. For writing on EBS volumes, which requires network access, the maximum bandwidth is 50.8 MBytes/sec on CC1 and 44.5 MBytes/sec on CC2, whereas for ephemeral disks the maximum bandwidth is 78.2 and 89.4 MBytes/sec on CC1 and CC2, respectively. In addition, the HI1 instance type presents similar results than CC1 and CC2 instances when using EBS volumes. Although 10 Gigabit Ethernet is available for communication among CC instances, the interconnection technology to access EBS volumes is not known. However, HI1 obtains again the best performance using the SSD disk, which obtains 562 MBytes/sec, around six times higher than the best result for the ephemeral disk on CC2. These results show that the ephemeral disks on HI1 instances provide very high performance that seems not to be affected at all by the virtualization layer.

Figure 2 presents the maximum bandwidth obtained when 2 (left graph) or 4 devices (right graph) are combined into a single software RAID 0 array (chunk size was configured at 64 KBytes). For the 2-device array configurations and the read operation, the results in CC1 and CC2 with EBS volumes and ephemeral disks are again very similar as they achieve around 200 MBytes/sec, double the performance of a single device, which suggests that the use of software RAID and virtualization in this scenario is not harming performance. The HI1 instance type, which is obviously the best option again, gets up to 1567 MBytes/sec, 75 % improvement compared to a single SSD disk. For the write operation, the performance of ephemeral disks on CC1 an CC2 instances is again significantly better than the results obtained with EBS volumes, approximately doubling its performance, whereas the SSD array is able to achieve up to 1014 MBytes/sec, which represents around 80 % improvement.

Regarding 4-device array configurations (right graph), the option with ephemeral disks is only available for CC2 instances, as CC1 and HI1 are physically limited to 2 ephemeral disks by

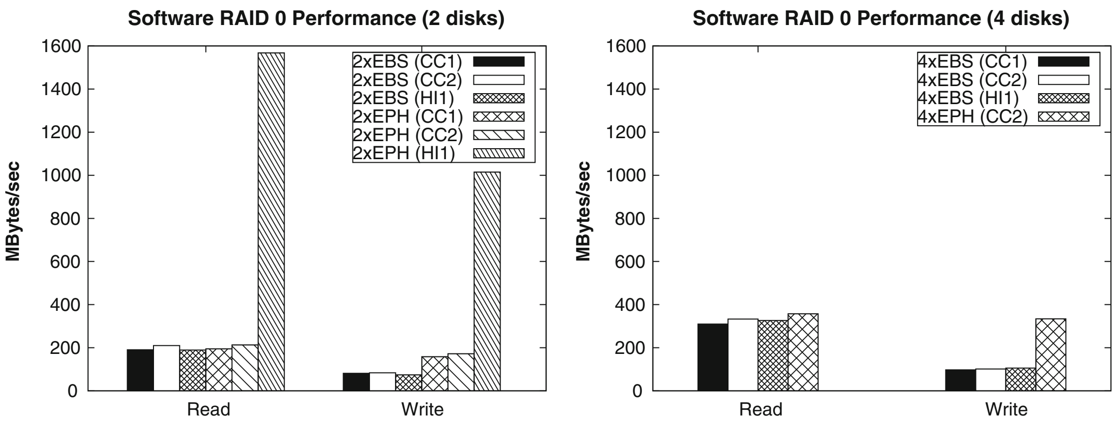


**Fig. 2** Software RAID 0 performance on Amazon EC2 CC and high I/O instances

Amazon, whereas the number of EBS volumes attached to them can be almost unlimited. For the read operation, 4 EBS volumes combined are able to get a 60 % improvement compared to the 2-device array (e.g., from 209 MBytes/sec to 333 MBytes/sec on CC2). However, the combination of 4 ephemeral disks on CC2 slightly outperforms EBS for reading, achieving a maximum of 358 MBytes/sec. For the write operation, although the use of two EBS volumes is able to double the performance of a single volume (83.7 vs 44.5 MBytes/sec on CC2, respectively), when 4 volumes are combined the maximum bandwidth obtained is only 101.3 MBytes/sec on CC2 and 105.3 MBytes/sec on HI1, showing poor scalability (this is the reason why the number of EBS volumes was limited to 4 in the evaluation). This poor result shows that the write performance on EBS is severely limited by the network performance. Furthermore, although the three instance types can not effectively take advantage of using 4 EBS volumes for the write operation, the combination of 4 ephemeral disks on CC2 is clearly the best performer, obtaining up to 334 MBytes/sec, showing almost a linear speedup (172 and 334 MBytes/sec for 2-device and 4-device arrays, respectively). Nevertheless, CC2 instances can not rival HI1 instances at all, as the 2-device SSD-based array on HI1 obtains more than 4 times higher read performance and up to three times more write performance than the 4-device array on CC2.

This evaluation has shown that EBS volumes suffer a significant performance penalty for write operations, and the use of software RAID can only help to partly alleviate this issue using up to 4 volumes. Therefore, the ephemeral disks, especially in RAID configuration, are the best option in a local file system scenario, as write operations are highly used in scientific applications. In fact, the SSD-based ephemeral disks of the new released HI1 instances clearly become the best choice, as they provide significantly higher performance than CC1 and CC2 ephemeral disks, both for read and write. Another advantage of using ephemeral disks is that the cost of their use is free, whereas the use of EBS volumes is charged by Amazon. However, if a particular application requires data persistence as a strong constraint, EBS volumes should be used for data safety reasons. In this scenario, a combination of both ephemeral and EBS volumes could be used.

## 5.2 Distributed File System Performance

Figures 3 and 4 present the results of the read and write performance of a distributed file system, NFS, with a base configuration of one instance running the NFS server and one client instance connecting to the server through the 10 Gigabit Ethernet network. The micro-benchmark selected is the IOzone benchmark using both EBS volumes and ephemeral disks as storage devices, and

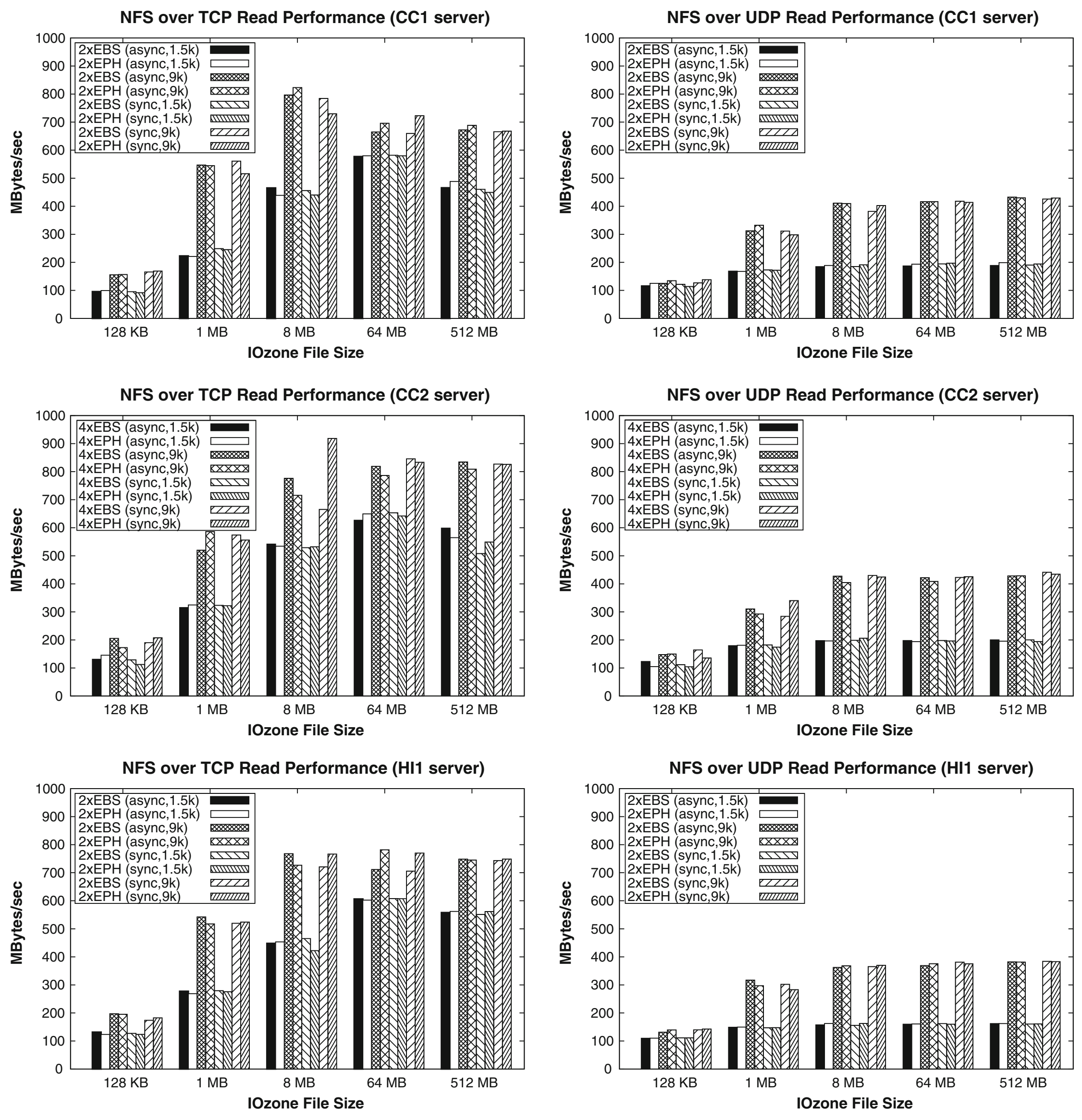


**Fig. 3** NFS read performance through 10 Gigabit Ethernet on Amazon EC2

using different file sizes for CC1, CC2 and HI1 instances. In these experiments, as only two instances are needed (one for the server and one for the client), two HI1 instances (the maximum that can be launched) allocated in the same placement group have been used, as for CC1 and CC2. For clarity purposes, the figures only present experimental results from RAID configurations as they provide better performance. In these experiments the NFS server buffer cache (or page cache) has not been bypassed in order to reflect the performance results of a typical NFS configuration, which generally takes advantage of this mechanism to achieve higher performance.

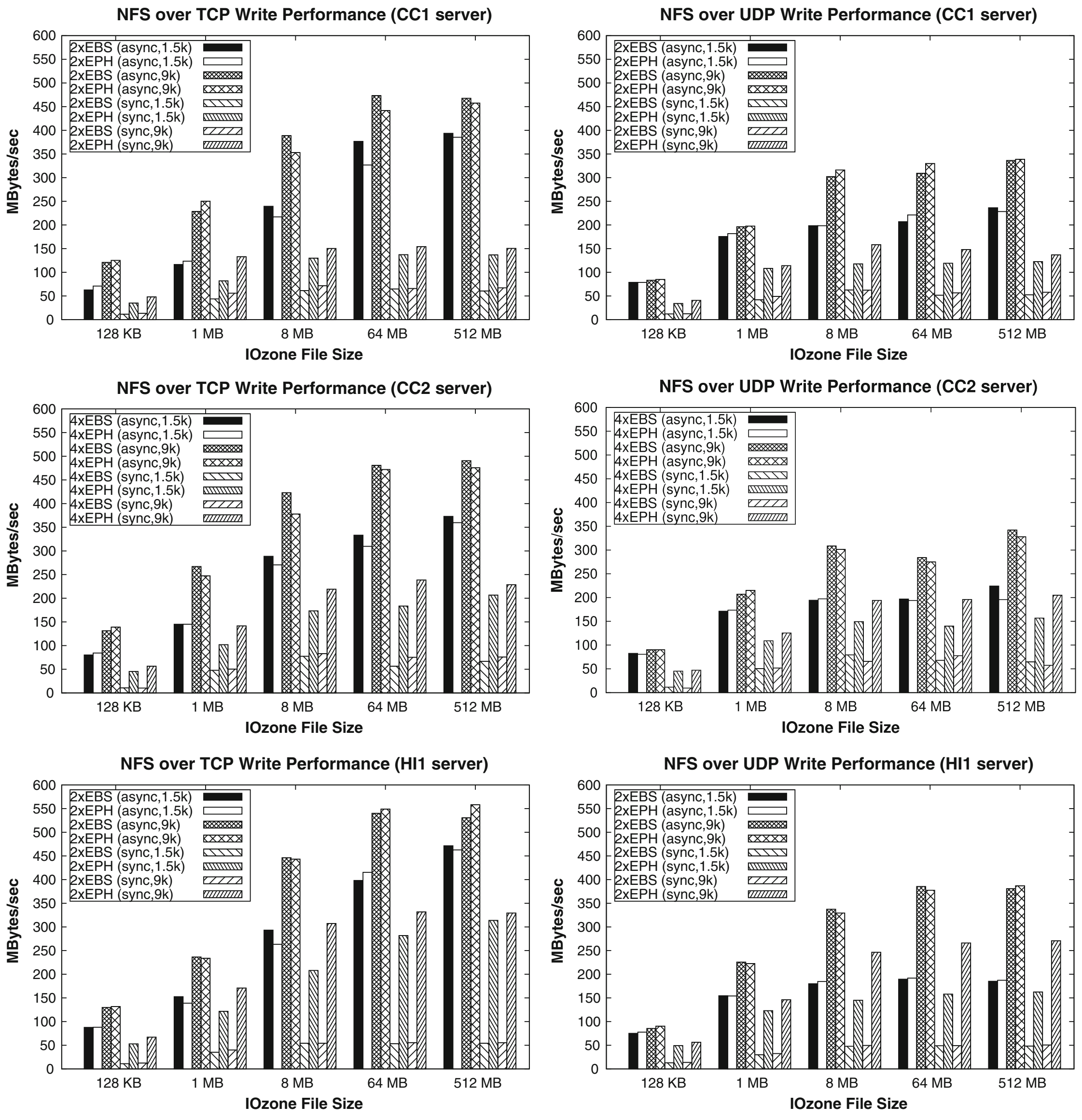


**Fig. 4** NFS write performance through 10 Gigabit Ethernet on Amazon EC2

Two NFS server configurations for the write operation have been used: (1) asynchronous (*async*) mode, which can provide high performance as it supports NFS calls to return the control to the client before the data has been flushed to disk; and (2) synchronous (*sync*) mode, where a block has to be actually written to disk before returning the control to the client, providing higher data safety in terms of data persistence when the NFS server crashes. In addition, the Amazon AMI for CC instances sets the Maximum Transmission Unit (MTU) of the network to 1,500 bytes by default. In order to assess the impact of the use of Jumbo frames (MTUs higher than the default 1,500 bytes) the tests have been repeated with the MTU configured at 9,000 bytes (maximum MTU value supported) both for server and client. Moreover, TCP and UDP transport protocols have been tested in order to characterize the impact of the selected protocol on the overall performance, which is highly important in virtualized environments where the network plays a key role. Both transport protocols have been configured with the maximum block size allowed for each one in this kernel version (1 MByte and 32 KBytes for TCP and UDP, respectively). These are the parameters that have generally shown a significant impact on performance. Finally, *noatime* and *nodiratime* mount options were enabled in the client as they can provide a small performance gain.

Figure 3 shows performance results on CC1, CC2 and HI1 instances (from top to bottom) which compare TCP and UDP protocols for the read operation under the different settings considered. TCP results (left graphs) significantly outperform UDP (right graphs) for all file sizes and configurations, especially from 8 MBytes on. This fact is due to the higher block size allowed in the TCP protocol, which benefits TCP especially when file sizes larger than 1 MByte are used. The performance comparison between EBS volumes and ephemeral disks for both protocols results in a tie, since data is actually read from the NFS server buffer cache that hides the underlying storage performance. In addition, the server *async* configuration is not able to achieve better performance as only write operations can take full advantage of this feature. However, the MTU parameter presents a huge impact on performance for both protocols; in fact in some cases the 9,000 byte MTU value allows for up to 165 % improvement over using the default MTU value of 1,500 bytes (e.g., see the 1 MByte file size for TCP protocol on CC1). The comparative analysis among different instance types shows that they achieve generally similar read performance, once again due to the operation of the NFS server cache.

Figure 4 compares TCP and UDP protocols for the write operation. In this case, these results have revealed some important facts: (1) TCP performance is again generally higher than UDP performance, except for the smallest file sizes, 128 KBytes and 1 MByte, where UDP slightly outperforms TCP when MTU is 1,500 bytes; (2) the *async* server configuration is able to provide significantly higher performance (up to 4 times better) than *sync* configuration for both protocols and all file sizes; (3) increasing the MTU value to 9,000 bytes provides better performance results for all configurations, especially for the *async* mode; (4) ephemeral disks show significantly better results than EBS volumes in the *sync* mode, confirming some of the conclusions derived from the local file system benchmarking. However, the *async* server mode allows to reduce the performance penalties of using EBS volumes enabling the overlapping of I/O, as control is returned to the client when data is written in the server buffer cache (the actual writing to disk is done asynchronously), which results in similar performance for EBS volumes and ephemeral disks in this scenario; (5) the *async* mode shows very similar results on CC1 and CC2 (for both protocols), whereas HI1 seems to perform slightly better than CC instances especially when using TCP and large file sizes; and (6) the *sync* mode allows to analyze more straightforwardly the underlying storage system performance. Thus, CC1 achieves up to 154 MBytes/sec on TCP with ephemeral disks, showing that performance in this case is being limited by the 2-device array. CC2, relying on an array of 4 ephemeral disks, outperforms CC1 with 238 MBytes/sec, whereas HI1 obtains up to 332 MBytes/sec thanks to the use of SSD disks. However, these results on CC2 and HI1 instances with *sync* mode reveal that the network becomes the main performance bottleneck, reducing significantly the maximum underlying disk performance obtained in the Section 5.1 (334 and 1014 MBytes/sec for CC2 and HI1, respectively).

#### *5.2.1 Multi-Client NFS Performance Using Different I/O Interfaces*

Figure 5 shows the aggregated read (left graphs) and write (right graphs) bandwidths, measured in MBytes/sec, obtained with the parallel I/O IOR benchmark in a configuration with one NFS server exporting multiple devices, either EBS volumes or ephemeral disks, and with multiple NFS clients which access the server through a 10 Gigabit Ethernet network. The experimental configuration

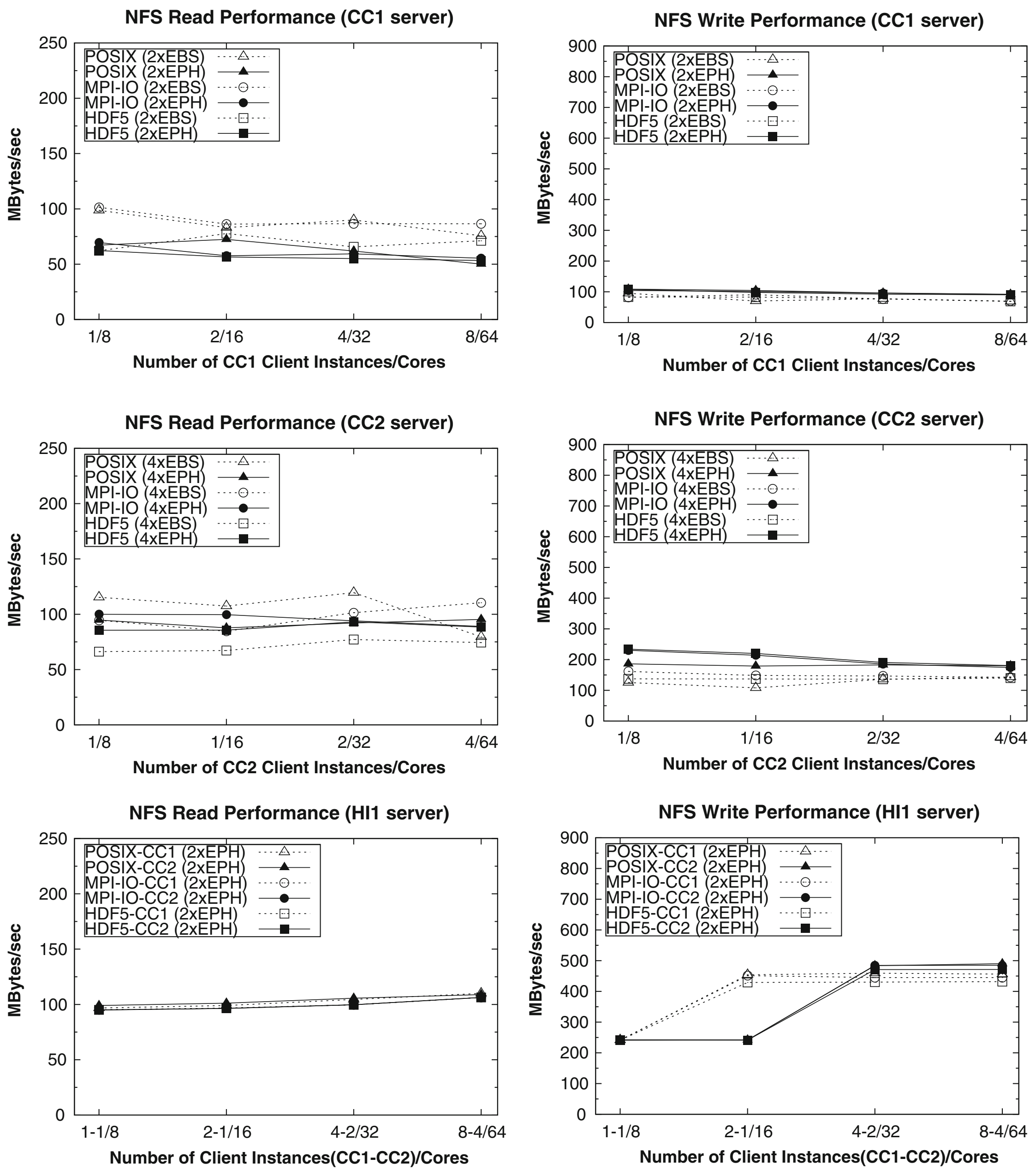


**Fig. 5** NFS performance using multiple clients on Amazon EC2 CC and high I/O instances

of this micro-benchmarking includes an optimal combination of values of NFS parameters, in order to provide the highest performance, that is to say the *async* mode for the NFS server as well as the use of the TCP protocol for NFS clients. The MTU value has also been configured at 9,000 bytes on all the machines involved, replacing the default value.

In these experiments, each of the client instances runs 8 (on CC1) or 16 (on CC2) parallel processes, reading and writing a single shared file collectively. For CC1 and CC2, both server and clients are instances of the same type. However, as mentioned in Section 4, the current restriction in the use of the HI1 instance type (only two instances can run simultaneously) limits the configuration of the HI1 testbed to the use of one HI1 instance for the NFS server and either CC1 or CC2 instance types for the NFS clients, which also implies that server and clients can not be allocated in the same placement group. EBS results on this HI1 testbed have been omitted for clarity purposes, since they are very similar to those of CC1 (for a 2-device array) and CC2 (for a 4-device array).

In order to obtain the underlying I/O throughput, the aggregated file size has been configured for each test to ensure that the NFS server memory is exhausted. This means that the aggregated file size under consideration is significantly larger than the available memory in the server, 23 and 60.5 GBytes for CC1 and CC2/HI1, respectively, reducing the impact of the usage of the NFS server buffer cache on the results. Finally, it has been set a high value (16 MBytes) for the transfer size parameter of IOR, which represents the amount of data transferred per process between memory and file for each I/O function call, in order to achieve the maximum possible aggregated bandwidth.

Performance results on CC1, CC2 and HI1 instances for the read operation are presented in the left graphs. As can be observed, CC2 obtains better results than CC1 for both storage devices and all interfaces (except for HDF5 on EBS which performs very similarly) due to the use of a 4-device array. On CC1 instances EBS volumes outperform ephemeral disks for the three evaluated interfaces, whereas this fact only occurs for MPI-IO on CC2. This result confirms that ephemeral disks scale better than EBS volumes when the number of devices combined in the RAID array increases, as seen in the local file system benchmarking results. The comparison between the interfaces shows that their performance is quite similar, although HDF5 results are slightly worse than the others for the read operation. On average, MPI-IO is the best performer for CC instances. Regarding the HI1 read graph, it shows almost similar results for the three interfaces evaluated on ephemeral disks both for CC1 and CC2 clients, and also very similar to CC2 results. This fact confirms that the network is the main performance bottleneck and thus the availability of a high performance storage device does not improve performance significantly.

The right graphs of Fig. 5 present the results for the write operation, where CC2 clearly outperforms CC1 instance type again, doubling the performance in some cases. Moreover, ephemeral disks are able to obtain better performance than EBS, even when the number of processes increases. Here, the additional network access incurred using EBS volumes seems to be the responsible for this performance penalty. Regarding HI1 instance type, the results show again that the network is clearly limiting its overall performance, achieving up to 456 and 490 MBytes/sec with 64 clients of CC1 and CC2 instance types, respectively. In addition, the performance difference between the I/O interfaces is almost null, and the difference between CC1 and CC2 clients is also negligible for 8, 32 and 64 clients. However, the use of a single CC2 instance to run 16 client processes (as it has 16 cores) obtains significantly lower (around half) performance than using two CC1 instances, where each one runs 8 client processes. This fact suggests that, once again, the performance bottleneck is in the network access, as the CC2 instance client has twice processes (16) accessing simultaneously the network card, thus dividing the available bandwidth per process and showing a poor ratio between network and CPU performance. For the remaining scenarios (8, 32 and 64 clients) the network link between the server (HI1) and the clients (CC1/CC2) remains as the limiting factor for the overall performance.

These results have revealed an important fact: the poor virtualized network performance clearly limits the new HI1 instances with SSD disks, especially for the read operation. Nevertheless, HI1 instances can provide better performance (up to twice higher) than CC instances for the write operation when the NFS server is configured in the *async* mode. Additionally, these results have confirmed the higher performance of the ephemeral disks for the write operation compared to EBS volumes, especially when using the CC2 and HI1 resources. However, for the read operation EBS volumes can achieve similar or even better performance than ephemeral disks. Therefore, the choice between storage devices, EBS or ephemeral, will depend on the I/O characteristics and requirements of each particular application.

### *5.2.2 The Effect of Caching and Transfer Size on NFS Performance*

In Section 5.2.1, the NFS server buffer cache was exhausted writing a shared file size which was significantly larger than the server memory in order to ensure that the performance of the underlying storage subsystem was actually being measured. This section presents the analysis of the effect of caching by writing a shared file which size is less than the server memory, and under different transfer sizes (from 16 KBytes to 16 MBytes), using MPI-IO as a representative I/O interface. The results for CC1, CC2 and HI1 instances using 64 clients are shown in Fig. 6, under the same NFS configuration than in the previous subsection but only including ephemeral disks that they provide the best write performance.

The left graph of Fig. 6 shows the performance results of writing a large file, twice larger than the available memory on the NFS server (64 GBytes for CC1 and 128 GBytes for CC2 and HI1). The results using a transfer size of 16 MBytes are the same as those shown in the previous subsection for MPI-IO (see Fig. 5). The right graph shows the performance of writing a file size that fits into the available memory of the server (16-GByte file size). Performance results have been obtained for 4 different configurations: (1) a CC1 server and 8 CC1 clients (CC1-CC1); (2) a CC2 server and 4 CC2 clients (CC2-CC2); (3) an HI1 server and 8 CC1 clients (HI1-CC1); and (4) an HI1 server and 4 CC2 clients (HI1-CC2). The limitation in the number of available HI1 instances (right now up to 2 per user) has prevented the evaluation of a scenario with both server and clients in HI1 instance. However, in this hypothetical configuration (HI1-HI1) the clients would benefit from being located in the same placement group, but they will not take advantage of the locally attached SSD disks and will suffer from the limited computational power of the HI1 systems (35 ECUs, similar to CC1 instances, but far from the 88 ECUs of CC2 instances).

The first conclusion that can be derived from this analysis is that the use of the largest transfer size (16 MBytes in this scenario) is key to achieve high parallel I/O performance, mainly in HI1 instances. The results in the left graph clearly show that the SSD disks on HI1 provide significantly better performance from 64 KBytes on. Once the server cache is exhausted, performance is deter-

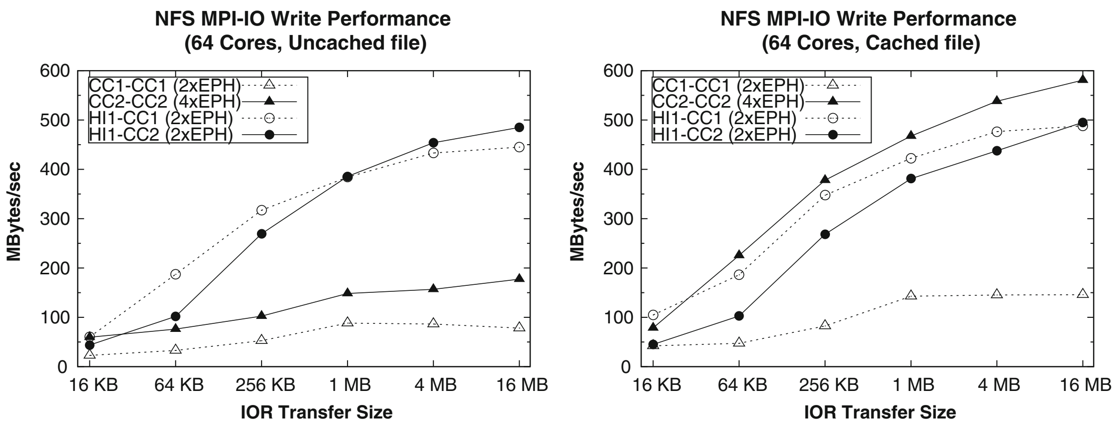


**Fig. 6** NFS performance depending on caching and transfer size

mined by the I/O subsystem, although for HI1 performance is ultimately determined by the network, as seen in Section 5.2.1.

The right graph shows significant performance increases for the CC2-CC2 configuration when the file is cached, whereas for the rest of configurations the results are only slightly better. The performance of the CC1-CC1 configuration is clearly limited by the poor performance of the underlying I/O subsystem, based on a 2-device array of ephemeral disks that, according to our previous local file system benchmarking, only provides up to 160 MBytes/sec (see Fig. 2 in Section 5.1). Regarding the HI1 server-based configurations (HI1-CC1 and HI1-CC2), whose 2-device array is able to provide above 1000 MBytes/sec, the network severely limits performance. This is mainly to the fact that the server is not in the same placement group of CC1/CC2 clients, which increases latency and reduces the full bandwidth available. Moreover, the CC2-CC2 configuration is now the best performer, even slightly better than the HI1 server-based configurations, due to a combination of two facts: (1) unlike the HI1 configurations, the allocation of CC2 server and clients in the same placement group enables to exploit the full network bandwidth; and (2) a CC2 instance provides higher memory write performance (up to 20 % more) than an HI1 instance, according to the STREAM benchmark [22], which clearly benefits the operation of the NFS buffer cache in a CC2 server.

## 5.3 I/O-Intensive Parallel Application Performance

The performance of a representative I/O-intensive parallel application, the BT-IO kernel from the NPB suite, has been analyzed. This code is the I/O-enabled version of the NPB BT benchmark, which solves Navier-Stokes equations in three spatial dimensions. As mentioned in Section 4, the NPB Class C workload has been selected, whereas the I/O size is the Full subtype. With these settings, all processes append data to a single file through 40 collective MPI-IO write operations, generating a total of 6.8 GBytes of output data, which are also read at the end of the execution. It has been used the default I/O frequency, which consists of appending data to the shared output file every 5 computation time steps. Finally, the BT-IO evaluation has been performed using the same NFS configuration as in Section 5.2, as it maximizes the NFS performance.

Figure 7 presents BT-IO performance using up to 64 clients. The performance metrics reported are the aggregated bandwidth measured in MBytes/sec (left graph) and MOPS (right graph). BT-IO requires that the number of client processes must be square numbers.

The aggregated bandwidth results confirm that ephemeral disks can provide better performance than EBS volumes. Thus, the CC1-CC1 configuration with ephemeral devices achieves up to 271 MBytes/sec (for 64 clients) whereas

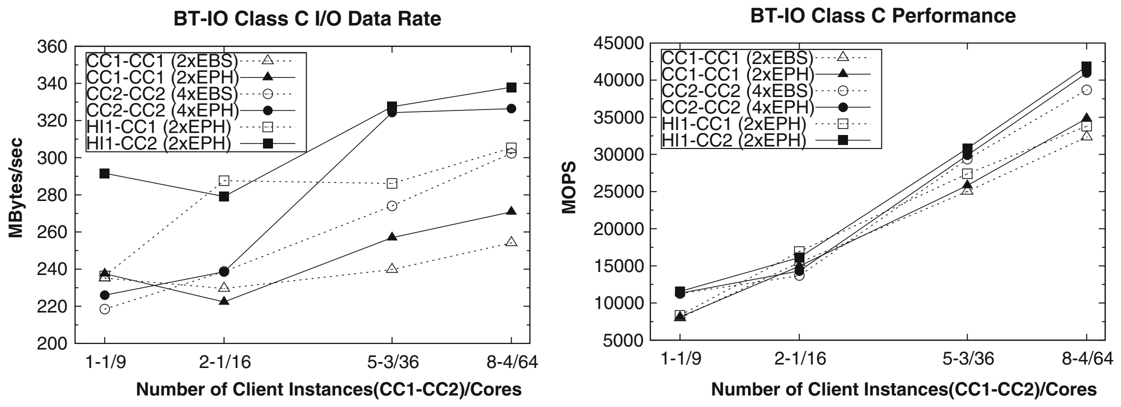


**Fig. 7** BT-IO performance on Amazon EC2 CC and high I/O instances

EBS volumes obtain up to 254 MBytes/sec. The numbers for the CC2-CC2 configuration are 327 and 302 MBytes/sec, respectively. Regarding the HI1 server-based testbeds, only ephemeral (SSD) devices have been considered, showing the best result for CC2 clients (e.g., 338 MBytes/sec for 64 clients). Here, the need of double the number of CC1 instances than CC2 ones (8 vs 4 when considering 64 clients) represents an important performance bottleneck for this code, as BT-IO is also a computation/communication-intensive code. Thus, the higher number of network communications required by CC1 are significantly affected by the poor virtualized network performance. This fact causes that the CC2-CC2 configuration using ephemeral devices outperforms HI1-CC1 for 36 and 64 clients.

These assessments are confirmed by the overall performance of the application in terms of MOPS (right graph). Thus, HI1-CC1 achieves similar results to CC1-CC1 (less than 35,000 MOPS), while using CC2 clients (either for CC2-CC2 or HI1-CC2) the measured performance is up to 20 % higher (e.g., 42,000 MOPS for HI1-CC2). Here, the differences in I/O performance are not translated into equivalent differences in MOPS, because I/O represents around 25–35 % of the total execution time of the application on these scenarios. The remaining execution time is spent in computation and MPI communications, increasing the communication overhead with the number of processes, which explains the impact of MPI communications on the overall performance as well as on I/O performance since disk writes are performed collectively.

Taking into account the different instance types, the use of a 4-device array allows CC2 to achieve higher performance than CC1 both for ephemeral and EBS devices, especially when considering the aggregated bandwidth, where CC2 servers obtain approximately 20 % more bandwidth than CC1 servers. As mentioned before, the HI1-CC1 configuration obtains poor results, whereas HI1-CC2 slightly outperforms the CC2-CC2 testbed on 64 clients (338 vs 327 MBytes/sec, respectively). The fact that all data are cached in the NFS server (in this scenario, 6.8 GBytes are written to disk) together with the poor performance of the virtualized network between the NFS server and its clients prevented the HI1 server-based configurations (in particular, HI1-CC2) to obtain better performance.

### *5.3.1 Cost Analysis of I/O-Intensive Parallel Codes*

Amazon EC2 offers different purchasing options: (1) on-demand instances, which allow to access immediately computation power by paying a fixed hourly rate; (2) spot instances from the spot market, which allow customers to bid on unused Amazon EC2 capacity and run those instances for as long as their bid exceeds the current spot price (which changes periodically based on supply and demand); and (3) reserved instances for one- or three-year terms, which allow to receive a significant discount on the hourly charge. There are three reserved instance types: light, medium and heavy, that enable to balance the amount payed upfront with the effective hourly price. Table 2 presents all the prices considered in the analysis.

In order to ease the analysis of the cost of Amazon EC2 resources, Fig. 8 presents the productivity of the previously evaluated BT-IO application in terms of aggregated bandwidth per USD$ . Only results for ephemeral disks are shown for clarity purposes as they provide better performance. Different purchasing options are compared: (1) using the price of on-demand instances (left graph); (2) using the average spot price in the July-September 2012 period (right graph, labeled as "S"), only for CC1 and CC2 instances as currently HI1 instances are not offered in the spot market; and (3) using the price calculated with heavy utilization reserved instances for a three-year term (right graph, labeled as "R"), which usually represents the lowest price that can be obtained for a particular instance type.

**Table 2** EC2 pricing for CC1, CC2 and HI1 (Linux/UNIX) instance types (us-east-1 region)

| Instance type | On-demand | Spot price | Reserved (3-year term) |
|---|---|---|---|
| CC1 | $1.30 | $0.818 | $0.537 |
| CC2 | $2.40 | $0.948 | $0.653 |
| HI1 | $3.10 | Not available | $0.899 |

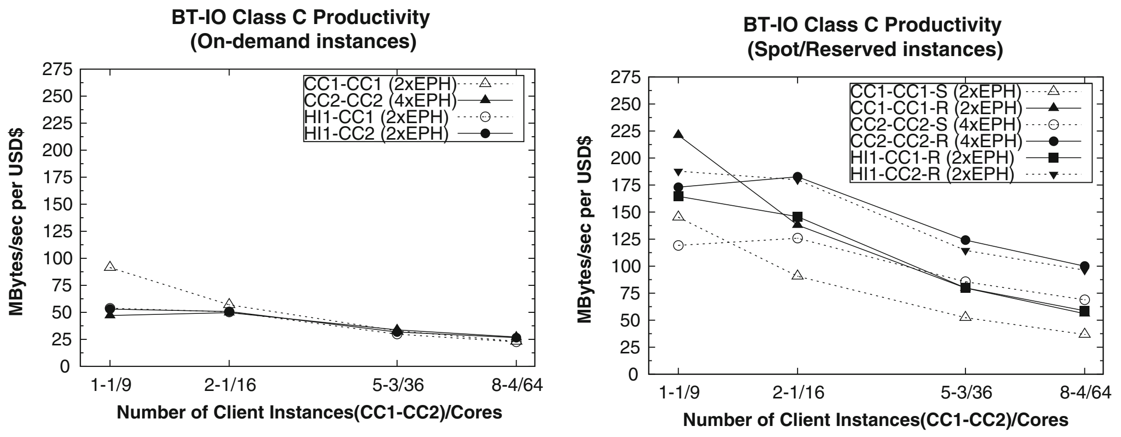


**Fig. 8** BT-IO productivity using on-demand, spot and reserved instances

On the one hand, the results using on-demand instances (left graph) show negligible differences between the different options except for the CC1-CC1 configuration with 9 clients, which is the most cost-effective option. On the other hand, the results using spot and reserved instances (right graph) present more differences among the evaluated configurations. First of all, the use of spot instances can provide significant cost improvements, up to 3 times higher performance/cost ratio than using on-demand instances (e.g., for CC2-CC2 with 64 clients), although here the main drawback of spot instances is that they can be shut down at any moment (when the spot price moves higher than the customer's maximum price). Regarding reserved instances, they allow for up to 25 % higher performance/cost ratio on average than using spot instances, being CC2-CC2 the most cost-effective option, slighly better than HI1-CC2. Furthermore, reserved instances will be always active for the availability zone specified at purchase time. Thus, reserved instances are the best choice in terms of performance/cost ratio.

## 6 Conclusions

Cloud computing is a model that enables on-demand and self-service access to a pool of highly scalable, abstracted infrastructure, platform and/or services, which are billed by consumption. This paradigm is currently being explored by the scientific community to assess its suitability for HPC applications. Among current cloud providers, Amazon WS is the leading commercial public cloud infrastructure provider.

This work has presented a comprehensive evaluation of the I/O storage subsystem on the Amazon EC2 Cluster Compute platform, a family of instance types which are intended to be well suited for HPC applications. Moreover, this work has included the evaluation of the new High I/O (HI1) instance type recently released by Amazon (July 2012), which provides SSD disks as ephemeral storage, thus highly oriented to be used in scalable cloud storage I/O systems. The performance evaluation was carried out at different layers and using several representative micro-benchmarks. Thus, the cloud low-level storage devices available in these instances (ephemeral disks and EBS volumes) have been evaluated both on local and distributed file systems, as well as the performance of several I/O interfaces commonly used in scientific applications (POSIX, MPI-IO and HDF5). Moreover, the scalability of an I/O-intensive code, the BT-IO application from the NPB suite, has also been analyzed at the application level, including an analysis in terms of cost.

Performance results have shown that the available cloud storage devices present significant performance differences. Thus, this paper has revealed that the use of ephemeral disks can provide more performance than EBS volumes for the write operation, especially when software RAID is used, thanks to the avoidance of additional network accesses to EBS, outside of the placement group, as EBS performance is deeply influenced by the network overhead and variability. In addition, this paper has characterized NFS performance on Amazon EC2, showing the impact of the main NFS configuration parameters on a virtualized cloud environment. Moreover, the analysis of the parallel I/O performance on Amazon EC2 has revealed that HI1 instances can provide significantly better write performance than any other instance type when writing very large files with large transfer sizes, although the overall performance is ultimately limited by the poor network throughput. Finally, the analysis of the performance/cost ratio of the BT-IO application has shown that, although the use of the HI1 instance type provides slightly better raw performance in terms of aggregated bandwidth, it may not be the best choice when taking into account the incurred costs.

**Acknowledgements** This work has been funded by the Ministry of Science and Innovation of Spain [Project TIN2010-16735 and an FPU Grant AP2010-4348], by the Galician Government [Program for the Consolidation of Competitive Research Groups, ref. 2010/6 and grant CN2011/007, partially supported by REDF funds] and by an Amazon Web Services (AWS) LLC research grant.

## References

1. Amazon Web Services in Top 500 list: http://www.top500.org/system/177457. Last visited: Nov 2012
2. IOzone Filesystem Benchmark: http://www.iozone.org/. Last visited: Nov 2012
3. MPI: A Message Passing Interface Standard: http://www.mcs.anl.gov/research/projects/mpi/. Last visited: Nov 2012
4. The HDF Group: http://www.hdfgroup.org/HDF5/. Last visited: Nov 2012
5. Abe, Y., Gibson, G.: pWalrus: towards better integration of parallel file systems into cloud storage. In: Workshop on Interfaces and Abstractions for Scientific Data Storage (IASDS'10), Heraklion, Crete, Greece, pp. 1–7 (2010)
6. Amazon Web Services LLC: Amazon Elastic Block Store (EBS). http://aws.amazon.com/ebs/. Last visited: Nov 2012
7. Amazon Web Services LLC: Amazon Elastic Compute Cloud (Amazon EC2). http://aws.amazon.com/ec2. Last visited: Nov 2012
8. Amazon Web Services LLC: Amazon Simple Storage Service (Amazon S3). http://aws.amazon.com/s3/. Last visited: Nov 2012
9. Amazon Web Services LLC: High Performance Computing Using Amazon EC2. http://aws.amazon.com/ec2/hpc-applications/. Last visited: Nov 2012
10. Carns, P., Ligon III, W., Ross, R., Thakur, R.: PVFS: a parallel virtual file system for linux clusters. In: Proc. 4th Annual Linux Showcase & Conference, Atlanta, GA, USA, pp. 317–328 (2000)
11. Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. Commun. ACM **51**(1), 107–113 (2008)
12. Deelman, E., Singh, G., Livny, M., Berriman, B., Good, J.: The cost of doing science on the cloud: the montage example. In: Proc. 20th ACM/IEEE Supercomputing Conference (SC'08), Austin, TX, USA, pp. 50:1–50:12 (2008)
13. Evangelinos, C., Hill, C.N.: Cloud computing for parallel scientific HPC applications: feasibility of running coupled atmosphere-ocean climate models on Amazon's EC2. In: Proc. 1st Workshop on Cloud Computing and Its Applications (CCA'08), Chicago, IL, USA, pp. 1–6 (2008)
14. Expósito, R.R., Taboada, G.L., Ramos, S., Touriño, J., Doallo, R.: Performance analysis of HPC applications in the cloud. Future Gener. Comput. Syst. **29**(1), 218–229 (2013)
15. Gabriel, E., et al.: Open MPI: goals, concept, and design of a next generation MPI implementation. In: Proc. 11th European PVM/MPI Users' Group Meeting (EuroPVM/MPI'04), Budapest, Hungary, pp. 97–104 (2004)
16. Ghoshal, D., Canon, R.S., Ramakrishnan, L.: I/O performance of virtualized cloud environments. In: Proc. 2nd International Workshop on Data Intensive Computing in the Clouds (DataCloud-SC'11), Seattle, WA, USA, pp. 71–80 (2011)
17. Gunarathne, T., Wu, T.L., Qiu, J., Fox, G.: MapReduce in the clouds for science. In: Proc. 2nd IEEE International Conference on Cloud Computing Technology and Science (CloudCom'10), Indianapolis, IN, USA, pp. 565–572 (2010)
18. Huang, W., Liu, J., Abali, B., Panda, D.K.: A case for high performance computing with virtual machines. In: Proc. 20th ACM International Conference on Supercomputing (ICS'06), Cairns, Australia, pp. 125–134 (2006)
19. Juve, G., Deelman, E., Berriman, G.B., Berman, B.P., Maechling, P.: An evaluation of the cost and performance of scientific workflows on Amazon EC2. J. Grid Comput. **10**(1), 5–21 (2012)
20. Liu, M., Zhai, J., Zhai, Y., Ma, X., Chen, W.: One optimized I/O configuration per HPC application: leveraging the configurability of cloud. In: Proc. 2nd ACM SIGOPS Asia-Pacific Workshop on Systems (APSys'11), Shanghai, China, pp. 1–5 (2011)

21. Mauch, V., Kunze, M., Hillenbrand, M.: High performance cloud computing. Future Gener. Comput. Syst. (2012) doi:10.1016/j.future.2012.03.011
22. McCalpin, J.D.: Memory bandwidth and machine balance in current high performance computers. In: IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter, pp. 19–25 (1995)
23. Napper, J., Bientinesi, P.: Can cloud computing reach the TOP500? In: Proc. Combined Workshops on UnConventional High Performance Computing Workshop Plus Memory Access Workshop (UCHPC-MAW'09), Ischia, Italy, pp. 17–20 (2009)
24. NASA: NAS Parallel Benchmarks. http://www.nas.nasa.gov/publications/npb.html. Last visited: Nov 2012
25. Nurmi, D., Wolski, R., Grzegorczyk, C., Obertelli, G., Soman, S., Youseff, L., Zagorodnov, D.: The eucalyptus open-source cloud-computing system. In: Proc. 9th IEEE International Symposium on Cluster Computing and the Grid (CCGRID'09), Shanghai, China, pp. 124–131 (2009)
26. de Oliveira, D., Ocaña, K.A.C.S., Baião, F.A., Mattoso, M.: A provenance-based adaptive scheduling heuristic for parallel scientific workflows in clouds. J. Grid Comput. **10**(3), 521–552 (2012)
27. Ostermann, S., Iosup, A., Yigitbasi, N., Prodan, R., Fahringer, T., Epema, D.: A performance analysis of EC2 cloud computing services for scientific computing. In: Proc. 1st International Conference on Cloud Computing (CLOUDCOMP'09), Munich, Germany, pp. 115–131 (2009)
28. Palankar, M.R., Iamnitchi, A., Ripeanu, M., Garfinkel, S.: Amazon S3 for science Grids: a viable solution? In: Proc. 1st International Workshop on Data-aware Distributed Computing (DADC'08), Boston, MA, USA, pp. 55–64 (2008)
29. Ramakrishnan, L., Canon, R.S., Muriki, K., Sakrejda, I., Wright, N.J.: Evaluating interconnect and virtualization performance for high performance computing. SIGMETRICS Perform. Eval. Rev. **40**(2), 55–60 (2012)
30. Regola, N., Ducom, J.C.: Recommendations for virtualization technologies in high performance computing. In: Proc. 2nd IEEE International Conference on Cloud Computing Technology and Science (CloudCom'10), Indianapolis, IN, USA, pp. 409–416 (2010)
31. Rodero, I., Viswanathan, H., Lee, E.K., Gamell, M., Pompili, D., Parashar, M.: Energy-efficient thermal-aware autonomic management of virtualized HPC cloud infrastructure. J. Grid Comput. **10**(3), 447–473 (2012)
32. Shafer, J.: I/O virtualization bottlenecks in cloud computing today. In: Proc. 2nd Workshop on I/O Virtualization (WIOV'10), Pittsburgh, PA, USA, p. 5 (7 p.) (2010)
33. Shan, H., Antypas, K., Shalf, J.: Characterizing and predicting the I/O performance of HPC applications using a parameterized synthetic benchmark. In: Proc. 20th ACM/IEEE Supercomputing Conference (SC'08), Austin, TX, USA, pp. 42:1–42:12 (2008)
34. Sun, C., Nishimura, H., James, S., Song, K., Muriki, K., Qin, Y.: HPC cloud applied to lattice optimization. In: Proc. 2nd International Particle Accelerator Conference (IPAC'11), San Sebastian, Spain, pp. 1767–1769 (2011)
35. Thakur, R., Gropp, W., Lusk, E.: On implementing MPI-IO portably and with high performance. In: Proc. 6th Workshop on I/O in Parallel and Distributed Systems (IOPADS '99), Atlanta, GA, USA, pp. 23–32 (1999)
36. Vecchiola, C., Pandey, S., Buyya, R.: High-performance cloud computing: a view of scientific applications. In: Proc. 10th International Symposium on Pervasive Systems, Algorithms, and Networks (ISPAN'09), Kaoshiung, Taiwan, pp. 4–16 (2009)
37. Walker, E.: Benchmarking Amazon EC2 for high-performance scientific computing. USENIX ;login: **33**(5), 18–23 (2008)
38. Wong, P., van der Wijngaart, R.: NAS parallel benchmarks I/O version 2.4. Tech. Rep. NAS-03-002, NASA Ames Research Center (2003)
39. Yang, H., Luan, Z., Li, W., Qian, D.: MapReduce workload modeling with statistical approach. J. Grid Comput. **10**(2), 279–310 (2012)
40. Youseff, L., Wolski, R., Gorda, B., Krintz, C.: Paravirtualization for HPC systems. In: Proc. International Workshop on XEN in HPC Cluster and Grid Computing Environments (XHPC'06), Sorrento, Italy, pp. 474–486 (2006)
41. Yu, W., Vetter, J.S.: Xen-based HPC: a parallel I/O perspective. In: Proc. 8th IEEE International Symposium on Cluster Computing and the Grid (CCGRID'08), Lyon, France, pp. 154–161 (2008)
42. Zhai, Y., Liu, M., Zhai, J., Ma, X., Chen, W.: Cloud versus in-house cluster: evaluating Amazon cluster compute instances for running MPI applications. In: Proc. 23rd ACM/IEEE Supercomputing Conference (SC'11, State of the Practice Reports), Seattle, WA, USA, pp. 11:1–11:10 (2011)
43. Zhang, Y., Gao, Q., Gao, L., Wang, C.: iMapReduce: a distributed computing framework for iterative computation. J. Grid Comput. **10**(1), 47–68 (2012)

# Chapter 9

# Performance Evaluation of Data-Intensive Applications on a Public Cloud

The content of this chapter corresponds to the following journal paper:

**- Title:** Performance evaluation of data-intensive computing applications on a public IaaS cloud

**- Authors:** Roberto R. Expósito, Guillermo L. Taboada, Sabela Ramos, Juan Touriño, Ramón Doallo

**- Journal:** Submitted for journal publication

**- Year:** 2014

A copy of the submitted paper has been included next.

# Performance evaluation of data-intensive computing applications on a public IaaS cloud

Roberto R. Expósito, Guillermo L. Taboada, Sabela Ramos, Juan Touriño and Ramón Doallo

*Computer Architecture Group, Department of Electronics and Systems, University of A Coruña, Campus de Elviña s/n, 15071 A Coruña, Spain*
*Email: {rreye,taboada,sramos,juan,doallo}@udc.es*

**The advent of cloud computing technologies, which dynamically provide on-demand access to computational resources over the Internet, is offering new possibilities to many scientists and researchers. Nowadays, Infrastructure as a Service (IaaS) cloud providers can offset the increasing processing requirements of data-intensive computing applications, becoming an emerging alternative to traditional servers and clusters. In this paper, a comprehensive study of the leading public IaaS cloud platform, Amazon EC2, has been conducted in order to assess its suitability for data-intensive computing. One of the key contributions of this work is the analysis of the storage-optimized family of EC2 instances. Furthermore, this study presents a detailed analysis of both performance and cost metrics. More specifically, multiple experiments have been carried out to analyze the full I/O software stack, ranging from the low-level storage devices and cluster file systems up to real-world applications using representative data-intensive parallel codes and MapReduce-based workloads. The analysis of the experimental results has shown that data-intensive applications can benefit from tailored EC2-based virtual clusters, enabling users to obtain the highest performance and cost-effectiveness in the cloud.**

*Keywords: Data-intensive computing; Cloud computing; Infrastructure as a Service (IaaS); Amazon EC2; Cluster file system; MapReduce*

## 1. INTRODUCTION

In recent years, the computational requirements for large-scale data-intensive computing [1] applications across distributed clusters or data centers have grown significantly in various disciplines including bioinformatics, astronomy or medical image analysis. In the current era of Big Data, characterized by the unprecedented volume of data, these applications are generating and analyzing large data sets, which usually require a high number of computational resources together with the availability of a high-performance cluster file system for scalable performance.

Cloud computing [2] is a relatively recent Internet-based computing model which is gaining significant acceptance in many areas and IT organizations as an elastic, flexible, and variable-cost way to deploy their service platforms using outsourced resources. These resources can be rapidly provisioned and released with minimal management effort. Public cloud providers offer access to external users who are typically billed by consumption using the pay-per-use pricing model.

Infrastructure as a Service (IaaS) is a type of cloud service which dynamically provides, by means of virtualization technologies, on-demand and self-service access to elastic computational resources (e.g., CPU, memory, networking and storage), offering a powerful abstraction that easily allows end users to set up virtual clusters to exploit supercomputing-level power without any knowledge of the underlying infrastructure. Public IaaS providers typically make huge investments in data centers and then rent them out, allowing consumers to avoid heavy capital investments and obtain both cost-effective and energy-efficient solutions. Hence, organizations are no longer required to invest in additional computational resources, since they can just leverage the infrastructure offered by the IaaS provider.

Most popular public cloud providers include Amazon Web Services (AWS) [3], Google Compute Engine (GCE) [4], Microsoft Azure [5] and Rackspace [6]. Nowadays, AWS remains as the top public cloud provider [7], offering the widest range of cloud-based services. In fact, the Elastic Compute Cloud (EC2) service [8] is among the most used and largest IaaS cloud platforms [9], which allows computational resources in Amazon's data centers to be easily rented on-demand. Moreover, Amazon EC2 offers several cloud resources which specifically target High Performance Computing (HPC) environments [10], composed of several virtual machines that are intended to be well suited for highly demanding workloads by offering powerful multi-core CPU resources, improved network performance via a high-speed interconnect (10 Gigabit Ethernet) and enhanced Input/Output (I/O) performance by providing Solid State Drive (SSD) disks.

In this context, the cloud computing paradigm has experienced tremendous growth in the last few years, particularly for general-purpose applications such as web servers or commercial web applications. Furthermore, it has also generated considerable interest both in the scientific community and industry. Thus, cloud computing is becoming an attractive option for distributed computing and HPC due to the high availability of computational resources at large scale. This fact has motivated multiple works that analyze the feasibility of using public clouds, especially Amazon EC2, instead of traditional clusters for running HPC applications [11, 12, 13, 14]. However, most of the previous works are focused mainly on computation- and communication-intensive HPC codes, especially tightly-coupled parallel applications using the Message-Passing Interface (MPI), whereas there are few works that have investigated cloud storage and I/O performance using data-intensive applications (e.g., MapReduce [15] workloads). In addition, previous evaluations have been carried out before Amazon introduced storage-optimized instances [16], which provide with direct-attached storage devices specifically optimized for applications with high disk I/O requirements.

This paper presents a comprehensive study of running data-intensive applications on the leading Amazon EC2 cloud, using storage-optimized instances and conducting a related analysis that takes into account both performance and cost metrics. Hence, multiple experiments have been performed at several layers using a suite of micro-benchmarks and applications to evaluate the full I/O software stack of data-intensive computing, ranging from the low-level storage devices and cluster file systems up to the application level using representative parallel codes implemented on top of common computing frameworks and I/O middleware (e.g., Apache Hadoop [17], MPI-IO [18]). The main experimental results indicate that the unique configurability advantage offered by public clouds, almost impossible to achieve in traditional platforms, can benefit significantly data-intensive applications. Thus, our main conclusions point out that current cloud-based virtual clusters enable end users to build up high-performance I/O systems when using the appropriate resources and configurations.

The rest of the paper is organized as follows: Section 2 describes the related work. Section 3 presents an overview of the software stack for data-intensive computing. Section 4 provides general background of the Amazon EC2 platform and the experimental configuration, briefly describing both the benchmarks and applications used as well as the evaluation methodology. Section 5 presents and analyzes the performance results obtained in the evaluation conducted in this work. Finally, Section 6 summarizes our concluding remarks.

## 2. RELATED WORK

Recently, there have been a series of research efforts assessing the suitability of using public cloud platforms for HPC and scientific computing. Although there are some works that have evaluated other public IaaS providers such as Microsoft Azure [19] and GoGrid [20], the vast majority of them have assessed the most popular IaaS platform: Amazon EC2 [11, 12, 13, 14, 21, 22, 23]. Most of these previous studies are mainly focused on computation and communication behavior, evaluating only tightly-coupled parallel codes, usually MPI-based applications, which are commonly used in HPC environments, hence characterizing only the performance of CPU and network. As a main conclusion of these works, it has been fairly well-established that communication-intensive MPI codes tend to perform poorly on Amazon EC2, primarily due to the low virtualized network performance.

However, few works have investigated I/O and storage performance on Amazon EC2. Some of them analyzed the suitability of running scientific workflows [24, 25, 26], showing that it can be a successful option as these are loosely-coupled parallel applications. Other works, next presented, have evaluated some I/O aspects of Amazon EC2, but they were carried out before the availability of storage-optimized instances. Evangelinos and Hill [27] reported some I/O storage results using an I/O-intensive workload, but limited to sequential runs on a basic Network File System (NFS) setting. The storage and network performance of the Eucalyptus cloud computing platform was analyzed in [28], confronted with some results from one general-purpose EC2 large instance. Ghoshal et al. [29] compared the I/O performance of Amazon EC2 confronted with Magellan, a private cloud platform, and an HPC cluster. Their results show that NFS performance in Amazon is many orders of magnitude worse than the parallel file system installed in the HPC cluster. Similar studies have been carried out in [30, 31, 32], which have evaluated some parallel codes for NFS and/or PVFS file systems, but only limited to MPI-based applications, as MapReduce-based workloads were not taken into account. Gunarathne et al. [33] presented a MapReduce framework designed on top of the Microsoft Azure cloud, which was evaluated against the Amazon Elastic MapReduce (EMR) service and an

EC2-based Hadoop cluster. More recently, the same authors have introduced a new runtime that supports iterative MapReduce computations [34]. Finally, some other works have evaluated MapReduce frameworks and applications on private cloud platforms such as Nimbus [35] and on traditional HPC clusters [36, 37].

## 3. OVERVIEW OF DATA-INTENSIVE COMPUTING APPLICATIONS

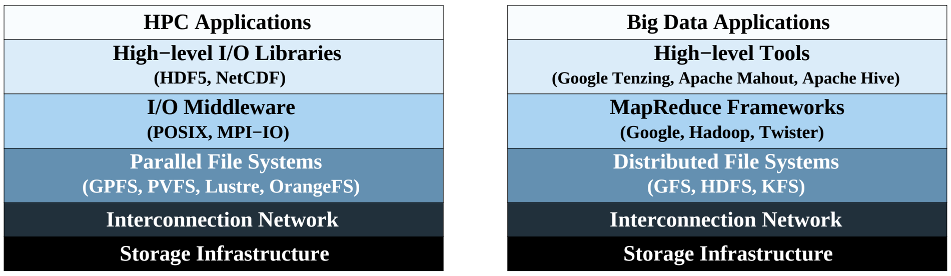
Most current data-intensive applications can be classified into one of the following categories: HPC and Big Data analysis. In both categories, applications are executed across distributed clusters or data centers using multiple compute nodes and handling massive amounts of data, in which the underlying cluster file system is a key component for providing scalable application performance.

On the one hand, HPC is traditionally defined by parallel scientific applications in the fields of science and engineering that rely on low-latency networks for message passing and cluster deployments that usually separate compute and storage nodes. HPC applications are typically large simulations that run for a long time and protect themselves from failures using fault tolerance methods such as checkpointing [38]. These methods involve large-scale data movements as the system state is periodically written to persistent storage in order to be able to restart the application in case of failure. Therefore, checkpointing can be particularity challenging when all processes in the parallel application write to the same checkpoint file at the same time, an I/O access pattern fairly known as N-1 writing [39]. Hence, these HPC applications are considered data-intensive, and they typically rely on a POSIX-based parallel file system for highly scalable and concurrent parallel I/O. In these systems, multiple dedicated storage nodes act as I/O servers to provide a UNIX file system API and expose a POSIX-compliant interface to applications to support a broad range of access patterns for many different workloads. In this scenario, the I/O access pattern is mainly dominated by sequential operations, being random accesses rare in data-intensive HPC applications [40, 41].

On the other hand, Big Data analysis generally refers to a heterogeneous class of business applications that operate on large amounts of unstructured and semi-structured data, usually implemented using the MapReduce programming model, which was first proposed by Google [15]. In fact, MapReduce has become the most popular computing framework for large-scale processing and analysis of vast data sets in clusters [42], mainly because of its simplicity and scalability. These data-intensive applications are designed to handle data more efficiently than a traditional structured query language to quickly extract value from the data. They include traditional batch-oriented jobs such as data mining, building search indices and log collection and analysis [43], as well as web search and advertisement selection [44]. One key aspect of these applications is that they are aware in advance of the workloads and I/O access patterns, typically relying on a custom, purpose-built distributed file system that is usually implemented from scratch. These file systems are specifically designed to support only one programming model (e.g., MapReduce) in order to provide high scalability with reliability by striping and replicating the data in large chunks across the locally attached storage of the cluster nodes. Hence, by exposing the data layout to the applications, the MapReduce task scheduler is able to co-locate a compute task on a node that stores the required input data [15], thereby relying on cluster deployments that co-locate compute and storage nodes on the same cluster node. However, these file systems feature a simplified design and implementation without providing a POSIX-compliant interface or consistency semantics. Consequently, they work well for MapReduce applications but cannot support traditional HPC applications without changes. Unlike distributed file systems, parallel file systems cannot exploit data locality as they do not generally expose the data layout to the applications, which usually results in a significant performance loss when running MapReduce applications. Hence, the developers of the most popular parallel file systems have demonstrated the feasibility of using them with the MapReduce paradigm by applying some minor modifications and providing simple file system extensions, obtaining comparable performance to distributed file systems [45, 46].

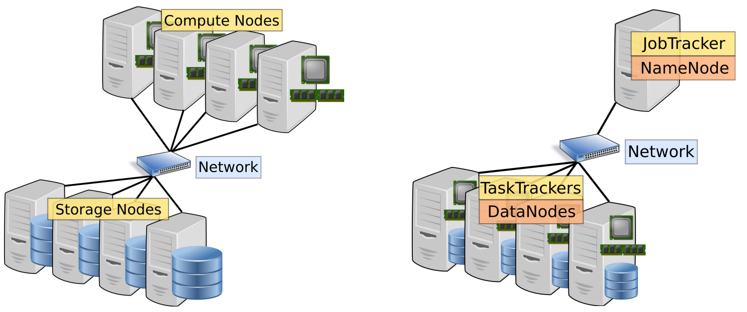
### 3.1. I/O Hardware and Software Support

Figure 1 shows the I/O software stacks more commonly available on current HPC and Big Data platforms that support data-intensive applications. On the one hand, HPC applications perform I/O at a specific level in the software stack depicted in Figure 1(a). In the upper levels, advanced data models such as HDF5 [47] and NetCDF [48] can provide certain advantages for particular applications. These high-level libraries usually feature a parallel version (Parallel HDF5 [49], Parallel NetCDF [50]) implemented on top of the MPI-IO [18] middleware in order to perform parallel I/O cooperatively among many processes. MPI-IO is specified in the MPI-2 standard and defines a comprehensive API, which is implemented on top of file systems, intended specifically to provide MPI programs with a high-performance, portable and parallel I/O interface. Towards the bottom of the stack, parallel file systems (e.g., GPFS [51], PVFS [52], Lustre [53], OrangeFS [54]) are used to transfer I/O requests and file data across the underlying interconnection network and storage devices. Figure 2(a) shows the most widely extended cluster architecture in HPC platforms, which usually separates compute and storage nodes.



(a) I/O software stack for HPC applications (b) Big Data analysis on top of MapReduce frameworks

**FIGURE 1.** I/O software stacks for data-intensive computing applications



(a) Traditional HPC cluster architecture (b) High-level overview of Hadoop and HDFS architecture

**FIGURE 2.** Hardware support for data-intensive computing applications

On the other hand, Big Data applications can either use a high-level tool or directly a distributed computing framework (e.g., Google MapReduce [15]) to perform their data analysis (see Figure 1(b)). Usually, the high-level tools (e.g., Google Tenzing [55], Apache Mahout [56]) are built on top of a computing framework to allow users write applications that take advantage of the MapReduce programming model without having to learn all its details. Among these frameworks, the Apache Hadoop project [17] has gained significant attention in the last years as a popular open-source Java-based implementation of the MapReduce paradigm derived from the Google's proprietary implementation. As mentioned before, these frameworks generally rely on custom distributed file systems (e.g., GFS [57], HDFS [58], KFS [59]) to transfer I/O requests across the interconnection network and the underlying storage infrastructure. Figure 2(b) shows an overview of a typical Hadoop cluster that stores the data in HDFS, using a master-slave architecture which co-locates compute and storage nodes on the same slave node. Using the Hadoop terminology, the TaskTracker and DataNode processes, which execute the tasks and store the data, respectively, run on the slave nodes. The master node runs the JobTracker and NameNode processes, acting as a single task manager and metadata server.

## 4. EXPERIMENTAL CONFIGURATION

In this section, the Amazon EC2 platform and the selected instance types are described along with brief descriptions of the representative benchmarks and applications used in the performance evaluation section.

### 4.1. Amazon EC2 Platform

The Amazon EC2 cloud service currently offers a rich variety of Xen-based virtual machine configurations called instance types [16], which are optimized to fit different use cases. Virtual machines of a given instance type comprise varying combinations of CPU, memory, storage and networking capacity, each with a different price point. One of the key contributions of this paper is the evaluation of the storage-optimized family of EC2 instances, which according to Amazon are specifically intented to be well suited for Hadoop, cluster file systems and NoSQL databases, among other uses.

The storage-optimized family includes two EC2 instance types: High I/O (hi1.4xlarge, hereafter HI1) and High Storage (hs1.8xlarge, hereafter HS1). These instances provide direct-attached storage devices (known as "ephemeral" or local disks) optimized for applications with specific disk I/O and capacity requirements. More specifically, HI1 provides 2 TB of instance storage capacity backed by 2 SSD-based disks, whereas HS1 provides 48 TB across 24 standard Hard Disk Drive (HDD) disks (see Table 1). Generally, EC2 instances can access two additional storage options: (1) off-instance Elastic Block Store (EBS), which are remote volumes accessible through the network that can be attached to an instance as block storage devices; and (2) Simple Storage Service (S3), which is a distributed key-value based object storage system, accessed through a web service interface. However, S3, unlike EBS and ephemeral devices, lacks general file system interfaces usually required by data-intensive applications. Hence, S3 has not been considered in our evaluation since its use is not transparent to applications, and because of its poor performance shown by previous works [24]. Moreover, ephemeral (local) disks perform better than EBS volumes according to [28, 29] due to the overhead caused by the additional network access incurred by EBS. This superior performance of ephemeral disks was assessed even before the availability of storage-optimized instances, which favor significantly the performance of local disks.

In addition, storage-optimized instances provide 8 physical cores that represent 35 EC2 Compute Units (ECUs[1]) of computational power, together with 60.5 and 117 GB of memory for HI1 and HS1, respectively. Moreover, these instances support cluster networking via a high-speed network (10 Gigabit Ethernet), which is also another differential characteristic of these resources. Hence, instances launched into a common placement group are placed in a logical cluster that provides low-latency, full-bisection 10 Gbps bandwidth connectivity between instances in the cluster. However, there is a current limitation that only instances of the same type can be included in a placement group (i.e., a placement group cannot combine HI1 and HS1 instances).

Two additional high-performance instance types which also provide the cluster networking feature together with the 10 Gigabit Ethernet network have been evaluated. On the one hand, the Cluster Compute instance type (cc2.8xlarge, hereafter CC2) is a compute-optimized instance with 16 physical cores, 60.5 GB of memory and 3.4 TB of instance storage capacity across 4 HDDs. On the other hand, the High Memory Cluster Compute instance type (cr1.8xlarge, herefater CR1) is a memory-optimized instance with 16 physical cores, 244 GB of memory and 240 GB of instance storage capacity across 2 SSDs. Note that CC2 and CR1 are among the EC2 instances with the most powerful computational resources (i.e., 88 ECUs). Table 1 summarizes the main characteristics of the selected instance types together with their hourly price for on-demand Linux usage in the North Virginia data center.

[1]According to Amazon one ECU provides the equivalent CPU capacity of a 1.0-1.2 GHz 2007 Opteron or Xeon processor.

### 4.2. Benchmarks and Applications

The evaluation of data-intensive applications on Amazon EC2 has been conducted using representative benchmarking tools and applications (see details in Table 2) ranging at the different levels shown in the software stacks of Figures 1(a) and 1(b). The first set of experiments consists of a single-instance micro-benchmarking of a local file system, *xfs*, selected as it provides interesting features (e.g., allocation groups, B-tree indexing, metadata journaling) as building blocks for scalable data-intensive infrastructures. This analysis has been carried out using the IOzone benchmark [60] on a single ephemeral disk as well as on multiple disks combined in a single software RAID array with the Linux *mdadm* utility.

After characterizing the performance of the underlying storage infrastructure, both software stacks have been analyzed at the cluster file system level. Regarding the HPC stack, the OrangeFS parallel file system [54] has been evaluated using the IOR benchmark [61] and the MPI-IO interface [18] as representative I/O middleware. OrangeFS is a relatively recent branch of the production-quality and widely extended Parallel Virtual File System (PVFS) [52], but unlike PVFS, under active development and introducing new features. OrangeFS has also been selected because it lacks, to the best of our knowledge, suitable assessments of its performance on a public cloud infrastructure. Regarding the Big Data software stack, the Intel HiBench suite [62] has been used for the evaluation of Apache Hadoop [17], selected as the most representative MapReduce computing framework. The HiBench suite consists of a set of Hadoop programs including both synthetic micro-benchmarks and real-world applications. Hadoop HDFS [58] has been evaluated at the distributed file system level using the Enhanced DFSIO benchmark included in HiBench.

Next, the performance of several data-intensive codes has been analyzed at the application level. On the one hand, the BT-IO kernel [63] and the FLASH-IO code [64], which are implemented on top of the MPI-IO and Parallel HDF5 (PHDF5) libraries, respectively, have been selected as representative I/O-intensive HPC applications. On the other hand, the Sort and WordCount workloads, also included in HiBench, have been selected as they are representative of two widespread kinds of MapReduce jobs: transforming data from one representation to another, and extracting a small amount of information from a large data set. Additionally, the PageRank and Aggregation workloads have been evaluated. They are based on high-level tools built on top of the Hadoop framework. PageRank is an implementation of the page-rank algorithm in Apache Mahout [56], an open-source machine learning library. Aggregation measures the performance of Apache Hive [65] through computing the sum of each group in Hive over a single read-only table.

**TABLE 1.** Description of the Amazon EC2 instances: CC2, HI1, HS1 and CR1

| | **CC2** | **HI1** | **HS1** | **CR1** |
|---|---|---|---|---|
| **Release Date** | November 2011 | July 2012 | December 2012 | January 2013 |
| **Instance Family** | Compute-optimized | Storage-optimized | Storage-optimized | Memory-optimized |
| **API name** | cc2.8xlarge | hi1.4xlarge | hs1.8xlarge | cr1.8xlarge |
| **Price (Linux)** | $2 per hour | $3.10 per hour | $4.60 per hour | $3.50 per hour |
| **CPU Model** | Intel Xeon E5-2670 Sandy Bridge | Intel Xeon E5620 Westmere | Intel Xeon E5-2650 Sandy Bridge | Intel Xeon E5-2670 Sandy Bridge |
| **#CPUs** | 2 (Eight-Core) | 2 (Quad-Core) | 1 (Eight-Core) | 2 (Eight-Core) |
| **CPU Speed (Turbo)** | 2.6 GHz (3.3 GHz) | 2.4 GHz (2.66 GHz) | 2 GHz (2.8 GHz) | 2.6 GHz (3.3 GHz) |
| **#Cores/Threads** | 16/32 (HT[2] enabled) | 8/16 (HT enabled) | 8/16 (HT enabled) | 16/32 (HT enabled) |
| **Amazon ECUs** | 88 | 35 | 35 | 88 |
| **ECUs per Core** | 5.5 | 4.4 | 4.4 | 5.5 |
| **ECUs per US$** | 44 | 11.29 | 7.61 | 25.14 |
| **L3 Cache size** | 20 MB | 12 MB | 20 MB | 20 MB |
| **Memory** | 60.5 GB | 60.5 GB | 117 GB | 244 GB |
| **#Memory Channels** | 4 (DDR3-1600) | 3 (DDR3-1066) | 4 (DDR3-1600) | 4 (DDR3-1600) |
| **Memory Bandwidth** | 51.2 GB/s | 25.6 GB/s | 51.2 GB/s | 51.2 GB/s |
| **#QPI Links** | 2 (4000 MHz) | 2 (2933 MHz) | 2 (4000 MHz) | 2 (4000 MHz) |
| **QPI Speed** | 8 GT/s | 5.86 GT/s | 8 GT/s | 8 GT/s |
| **Ephemeral Disks** | 4 × 845 GB (HDD) | 2 × 1 TB (SSD) | 24 × 2 TB (HDD) | 2 × 120 GB (SSD) |
| **Storage Capacity** | 3.4 TB | 2 TB | 48 TB | 240 GB |
| **Storage per US$** | 1.7 TB | 0.65 TB | 10.43 TB | 0.07 TB |
| **Interconnect** | 10 Gigabit Ethernet (Full-bisection bandwidth using Placement Groups) | | | |

[2]The Intel Hyper-Threading (HT) technology is enabled for all instance types. Hence, Amazon announces that these instances have a number of available virtual cores that takes into account hyper-threaded cores (i.e., Amazon states that CC2 instances provide 32 virtual cores). However, the performance increase of using this technology is highly workload-dependent, and may be even harmful in some scenarios. Particularly, we have not seen any improvements using all the available virtual cores in our experiments on Amazon EC2. Therefore, all the configurations shown in this paper have used only one virtual core per physical core.

### 4.3. Software Settings

The Linux distribution selected for the performance evaluation was Amazon 2012.09.1, as it is a supported and maintained Linux provided by Amazon for its specific usage on EC2 instances. This Linux flavour comes with kernel 3.2.38 and has been tailored for the performance evaluation with the incorporation of the previously described benchmarks: IOzone 3.414, IOR 2.10.3 and HiBench 2.2. For the parallel file system evaluation, OrangeFS version 2.8.7 was used, whereas the selected MPI library was MPICH [66] version 3.0.2, which includes MPI-IO support. In addition, the MPI implementation of the NASA Advanced Supercomputing (NAS) Parallel Benchmarks suite (NPB) [67] version 3.3 was installed for the BT-IO kernel evaluation, whereas the HDF5 data library version 1.8.10 was used for the FLASH-IO code. Regarding the Hadoop experiments, the versions used were Hadoop 1.0.4 (stable), Mahout 0.7 and Hive 0.9.0. The Java Virtual Machine (JVM) was OpenJDK version 1.7.0_19.

Finally, the performance results presented in this paper are averages of a minimum of five measurements for each evaluated configuration. Furthermore, all the experiments have been carried out in the US East EC2 region (i.e., *us-east-1*), corresponding to North Virginia, the main data center, which has the highest availability of the evaluated instance types. The selected availability zone was *us-east-1d*, where, according to previous works [84] and our own experience, there is usually the lowest variability.

## 5. EVALUATION OF AMAZON EC2 FOR DATA-INTENSIVE COMPUTING

This section presents an in-depth performance and cost analysis of data-intensive computing applications on the selected public IaaS cloud, Amazon EC2, using the representative micro-benchmarks, cluster file systems and kernels/applications described in the previous section.

### 5.1. Single-Instance Storage Micro-benchmarking

Figure 3 presents the sequential write and read bandwidth (left graphs) using the IOzone benchmark on a single ephemeral disk (two top rows of graphs) and on multiple ephemeral disks combined into a software RAID array (two bottom rows of graphs). The right graphs of the figure show the performance/cost ratio on these scenarios, a productivity metric defined as the bandwidth obtained per invoiced US$. As mentioned before, the underlying local file system is *xfs*, whereas the default chunk size[3] (512 KB) has been used in RAID configurations. Furthermore, the evaluation of CC2, CR1 and HI1 instances has only considered the RAID 0 level (data striping) owing to the low number of available ephemeral disks on these instances (i.e., 2 disks in CR1 and HI1 and 4 disks in CC2). However, HS1 provides up to 24 ephemeral disks and so the RAID 0 level might not be the most reliable solution. Hence, HS1 has been evaluated using two additional RAID levels: RAID 6 and 10. These experiments basically write and read a single data file of 8 GB on *xfs*, and the performance results are shown for a certain block size (i.e., the transfer size of each underlying I/O operation at the file system level) varying from 16 KB up to 16 MB, which is the maximum value supported by IOzone. Finally, the Linux buffer cache was bypassed using direct I/O (O_DIRECT flag) in order to get the real performance (without buffering that might distort the results) of the underlying storage devices.

The results using a single ephemeral disk show that the SSD-based device of the HI1 instance significantly outperforms the rest of instances, especially from 64 KB on, obtaining up to 3 times (for writing, achieving 545 MB/s) and up to 4 times (for reading, achieving 960 MB/s) higher performance and productivity. However, CR1, which also provides SSD-based disks, obtains poor results compared to HI1, and only slightly better than HS1 for the read operation. The CC2 instance type gets the worst results in terms of performance, but it can be a competitive option, at least compared to CR1 and HS1, when taking into account the incurred costs, especially for the write operation.

Regarding software RAID results, HI1 almost doubles the performance of a single ephemeral disk obtaining up to 1060 and 1620 MB/s for writing and reading, respectively. However, HS1 now obtains the maximum bandwidth results achieving up to 2200 MB/s for both operations when using the RAID 0 level and a block size larger than 1 MB, thus taking full advantage of the 24 available disks. In terms of write productivity, HS1 using RAID 0 and the largest block sizes ($\geq$ 4 MB) is again the best option, despite the fact that it is the most expensive instance ($4.6 per hour), whereas its read productivity is slightly lower than HI1. Nevertheless, the RAID 6 level imposes a high performance penalty for the write operation as only 200 MB/s are achieved. Therefore, the RAID 10 level is the midpoint between performance and reliability for HS1, obtaining write bandwidth results very similar to those of HI1, but up to 30% lower productivity. Finally, CC2 and CR1, which cannot rival previous instances for large block sizes, obtain very similar results, which allows CC2 to be more productive than CR1 due to its lower price ($2 vs $3.5 per hour).

As main conclusions, these results have revealed that: (1) HI1 instances clearly provide the best performance and productivity when using a single ephemeral disk. (2) Using RAID 0, the HS1 instance is clearly the best performer thanks to the availability of up to 24 ephemeral disks. However, if the storage reliability is

[3]Chunk size is defined as the smallest "atomic" amount of data that is written to the storage devices.

**TABLE 2.** Selected benchmarks, kernels and applications

| Name | Type | Description |
|---|---|---|
| IOzone [60] | I/O Benchmark | IOzone is a popular open-source file system benchmarking tool used in several works [68, 69, 70] that generates and measures a variety of file system operations. It has been ported to many machines and operating systems. |
| IOR [61] | Parallel I/O Benchmark | IOR is a widely extended benchmarking tool for evaluating parallel I/O performance, as done in [40, 41, 70, 71, 72, 73, 74, 75]. It is highly parameterized and allows to mimic a wide variety of I/O patterns, supporting different APIs (e.g., MPI-IO, HDF5). |
| Intel HiBench [62] | Benchmark / Application Suite | HiBench is a representative benchmark suite for Hadoop, used for instance in [76, 77, 78]. It consists of a set of typical Hadoop workloads (e.g., Sort, WordCount), HDFS micro-benchmarks (e.g., DFSIO) as well as web search (PageRank), machine learning (K-means) and data analytics (e.g., Hive queries such as Aggregation) real-world applications. |
| NPB BT-IO [63] | Parallel I/O Benchmark | The NPB suite [67] is the de-facto standard to evaluate the performance of parallel computers, as done in [70, 74, 79, 80, 81] for I/O. BT-IO extends the NPB BT kernel by adding support for periodic solution checkpointing using the MPI-IO interface. |
| FLASH-IO [64] | Parallel I/O Benchmark | FLASH-IO is a widely used benchmark, e.g. in [50, 75, 81, 82], that mimics the I/O pattern of FLASH [83], a parallel hydrodynamics application. It recreates the primary data structures in FLASH and produces a checkpoint file using the parallel HDF5 library. |

critical it will be more reasonable to use the RAID 10 level, which provides comparable performance to HI1 but lower productivity. (3) Although CC2 performance is usually among the worst ones, it achieves competitive productivity results using RAID 0 and a block size $\leq 1$ MB, as it is the cheapest instance under evaluation. And (4) the poor performance results obtained by CR1, together with its high price, low productivity and reduced storage capacity (240 GB) points out that it is not a good choice for data-intensive applications, which has led us to discard its evaluation in the remainder of the paper for clarity purposes.

### 5.2. Performance Analysis at the Cluster File System Level

Two representative file systems have been selected for this analysis: (1) OrangeFS, a parallel file system widely extended in HPC clusters (Section 5.2.1), and (2) HDFS, which is the distributed file system especifically designed for the Hadoop MapReduce computing framework (Section 5.2.2).

#### *5.2.1. Parallel File System Performance*

Figures 4 and 5 present the aggregated bandwidth of OrangeFS (top graphs) for write and read operations, respectively, using the MPI-IO interface. These results have been obtained using the IOR benchmark with a baseline cluster configuration that consists of 4 instances acting as I/O servers (i.e., storage nodes of instance type CC2, HI1 or HS1) and multiple instances acting as clients (i.e., compute nodes of instance type only CC2 or HI1), which access the server data through the 10 Gigabit Ethernet network. In these experiments, the number of cores in the cluster has been set to 128, which determines the number of compute nodes being used depending on the client instance type. Hence, each client instance runs 8 (on HI1) or 16 (on CC2) parallel processes (i.e., one MPI process per core), writing and reading collectively a single shared file of 32 GB under different block sizes (from 64 KB to 16 MB), thus simulating the aforementioned N-1 access pattern. For clarity purposes, the graphs only present experimental results using RAID configurations on the storage instances, as they maximize their performance. Furthermore, the usage of RAID in HPC environments is the common rule as it allows to increase performance and/or redundancy. Therefore, the RAID 0 level for CC2 and HI1 storage instances and RAID 10 for HS1 have been selected for this benchmarking.

As shown in Table 3, four different cluster configurations have been evaluated: (1) using 4 CC2 servers and 8 CC2 clients (labeled as CC2-CC2); (2) 4 HI1 servers and 8 CC2 clients (HI1-CC2); (3) 4 HI1 servers and 16 HI1 clients (HI1-HI1); and (4) 4 HS1 servers and 8 CC2 clients (HS1-CC2). Note that the HS1-HS1 configuration (i.e., 4 HS1 servers and 16 HS1 clients) has not been included for clarity purposes, as it provides similar performance than HI1-HI1 but incurring significantly higher costs. The use of CC2 instances as clients in the heterogeneous cluster deployments (i.e., HI1-CC2 and HS1-CC2) has been motivated for their higher number of cores and computational power compared to HI1 and HS1 instances: 16 vs 8 cores and 88 vs 35 ECUs. In addition, their lower price significantly reduces the overall cost of

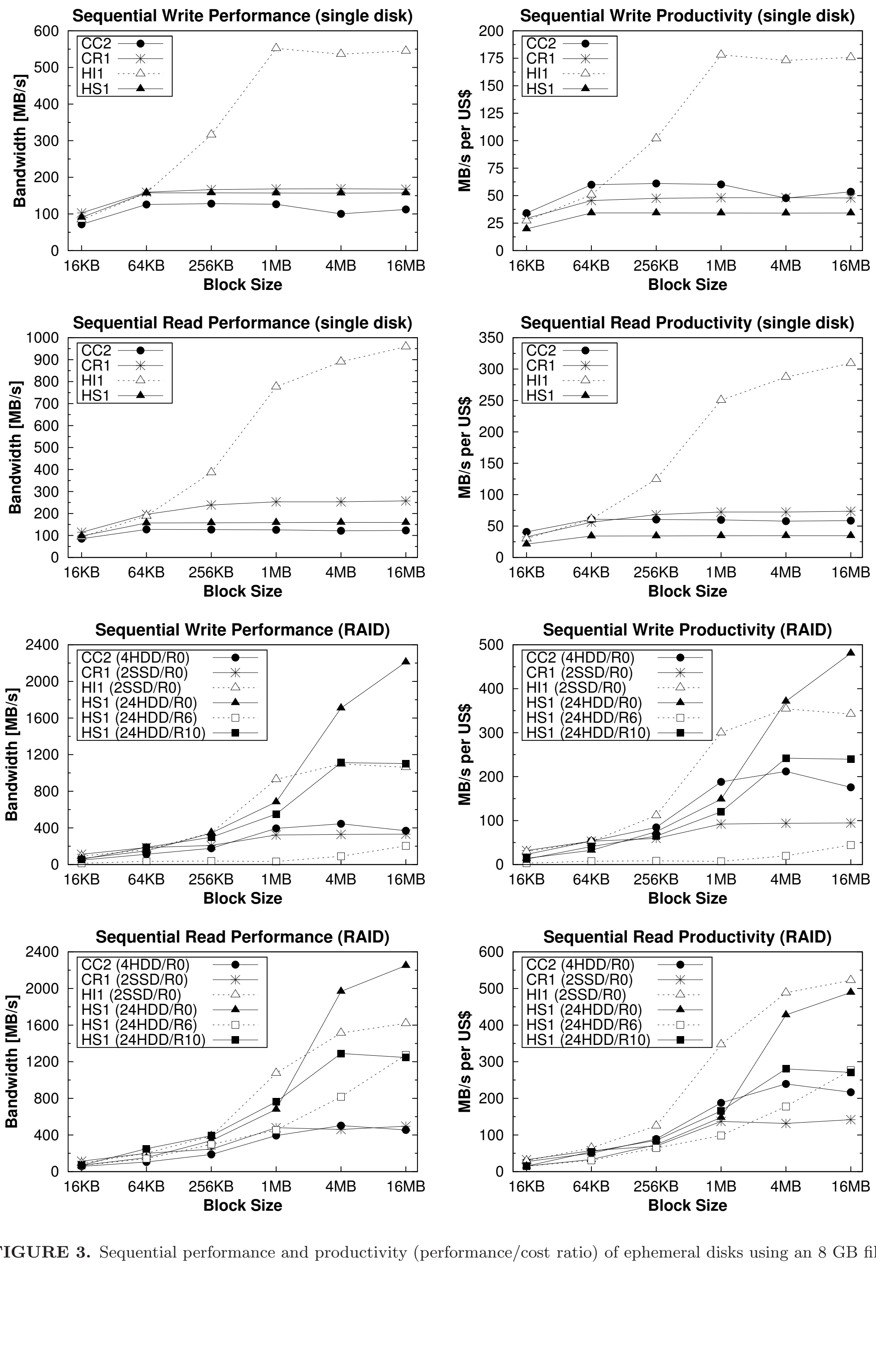


**FIGURE 3.** Sequential performance and productivity (performance/cost ratio) of ephemeral disks using an 8 GB file

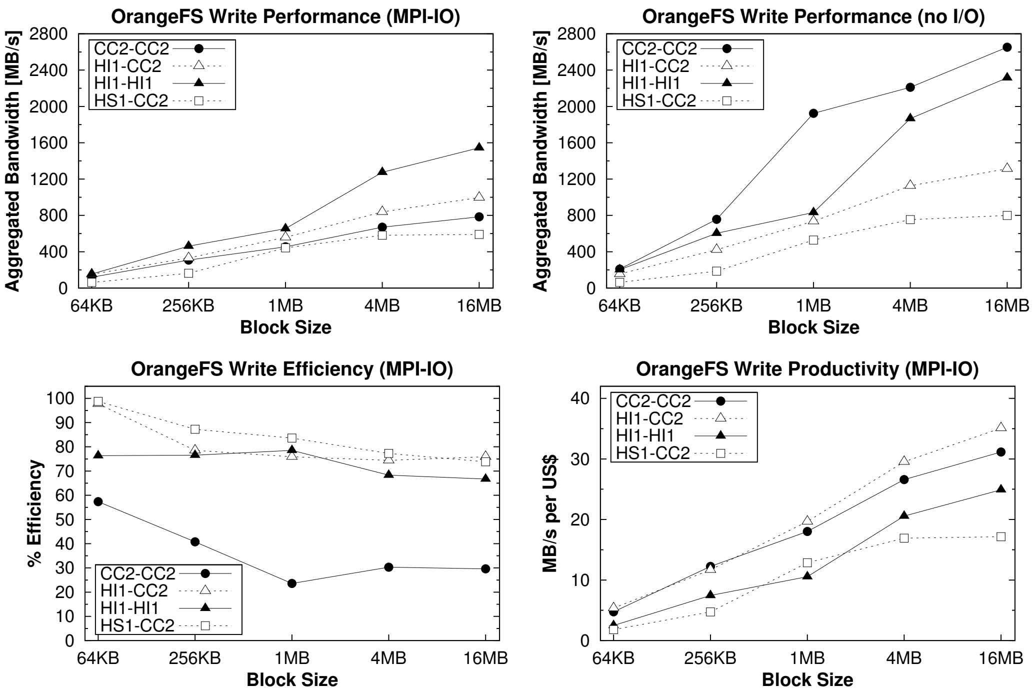


**FIGURE 4.** OrangeFS write performance, efficiency and productivity using 4 I/O servers and 128 cores

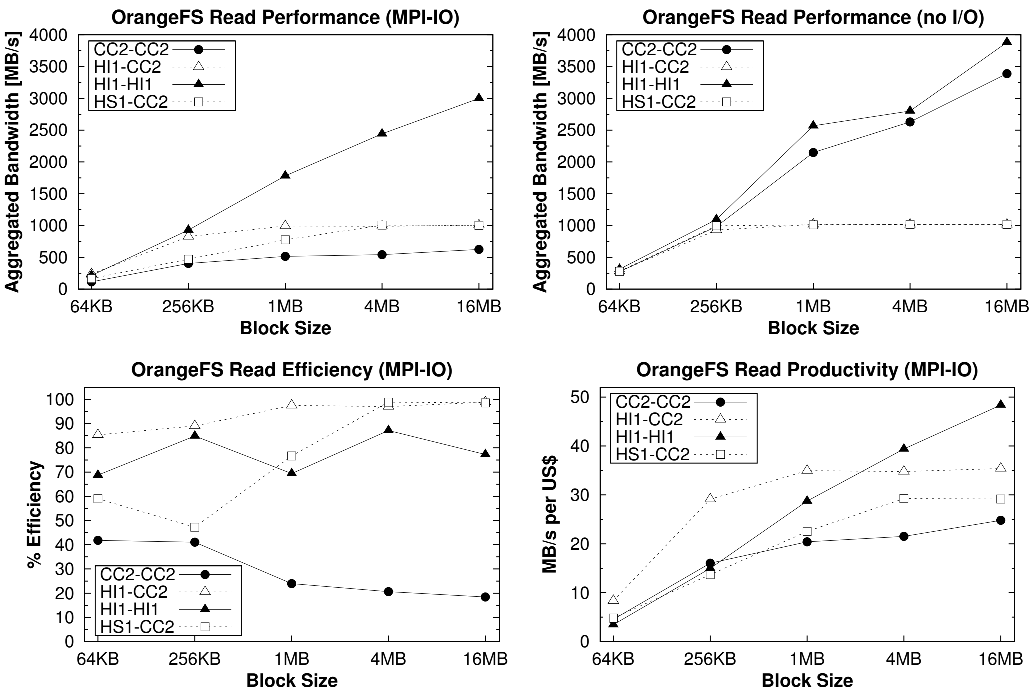


**FIGURE 5.** OrangeFS read performance, efficiency and productivity using 4 I/O servers and 128 cores

**TABLE 3.** Hourly cost using on-demand instances of the EC2-based HPC clusters

| HPC Cluster | #I/O Server Instances | #Client Instances | #Compute Cores | Hourly Cost |
|---|---|---|---|---|
| **CC2-CC2** | 4 × CC2 | 8 × CC2 | 128 | $24 |
| **HI1-CC2** | 4 × HI1 | 8 × CC2 | 128 | $28.4 |
| **HI1-HI1** | 4 × HI1 | 16 × HI1 | 128 | $62 |
| **HS1-CC2** | 4 × HS1 | 8 × CC2 | 128 | $34.4 |

the cluster deployment on Amazon EC2, which favors their popularity for HPC applications.

Furthermore, the use of storage-optimized instances as clients does not take advantage of the locally attached ephemeral disks. However, the heterogeneous cluster deployments (e.g., HI1-CC2) cannot benefit from being located in the same placement group, which can cause a loss in network performance. In order to analyze the impact of locating nodes outside the same placement group, the "null-aio" TroveMethod[4] available in OrangeFS has been used. This method consists of an implementation that does no disk I/O, and is useful to test the performance of the underlying interconnection network. Hence, the top right graphs in Figures 4 and 5 show the aggregated network bandwidth using the "null-aio" method (labeled as "no I/O"), whereas the top left graphs have been obtained with the default method ("alt-aio", which uses an asynchronous I/O implementation). The "null-aio" method also allows to analyze each cluster configuration in terms of the efficiency metric, shown in the bottom left graphs, which has been defined as the aggregated network bandwidth divided by the aggregated bandwidth when doing I/O. Finally, the bottom right graphs show the productivity results in terms of aggregated bandwidth per invoiced US$, taking into account the hourly cost of the different EC2-based HPC clusters (see the last column of Table 3).

Regarding write performance results (see the top left graph in Figure 4), it can be clearly observed that the use of the largest block size (16 MB) is key to achieve high-performance parallel I/O, mainly using HI1 instances. HI1 server-based configurations achieve the best results, around 1600 and 1000 MB/s for HI1-HI1 and HI1-CC2, respectively, whereas CC2-CC2 obtains only 800 MB/s, although it outperforms the HS1-CC2 configuration (600 MB/s). These results can be explained by the top right graph, where HI1-CC2 and HS1-CC2 obtain the poorest network bandwidths, which significantly limit their overall storage performance. As mentioned before, it is not possible to locate different instance types in the same placement group, so when using CC2 clients together with HI1 and HS1 servers the network performance drops severely. However, the HI1-CC2 cluster achieves slightly higher network bandwidth than HS1-CC2, which can suggest that the physical location of the HI1 and HS1 instances with respect to the clients (CC2) could be different inside the same EC2 region (*us-east-1* in these experiments). Here, the CC2-CC2 cluster achieves the highest network bandwidth (up to 2650 MB/s), but very poor efficiency (around 30%) when using the largest block sizes, as shown in the bottom left graph, because of its limited underlying storage performance. However, the remaining configurations achieve above 65%, being HS1-CC2 the most effective configuration. Taking costs into account (see the bottom right graph), HI1-CC2 becomes the best configuration from 256 KB on, owing to its relatively high performance and the use of client instances that are cheaper than the HI1-HI1 configuration. Even CC2-CC2 seems to be a good choice instead of HI1-HI1, as it is the cheapest cluster under evaluation. Finally, HS1-CC2 is obviously the worst option in terms of productivity due to its high price and low performance, especially when using block sizes $>$ 1 MB.

Regarding read performance (see the top left graph in Figure 5), HI1-HI1 is again the best performer, up to 3000 MB/s of aggregated bandwidth using the largest block size. In this case, HI1-CC2 and HS1-CC2 achieve similar results from 1 MB on, around 1000 MB/s, clearly limited by the interconnection network, as shown in the top right graph. Here, HS1-CC2 requires a large block size ($\geq$ 4 MB) to exploit the underlying network bandwidth, whereas HI1-CC2 does not. The CC2-CC2 configuration presents the worst performance with 625 MB/s, although it obtains up to 3400 MB/s of network bandwidth, showing again very poor efficiencies (below 25%, see the bottom left graph). HI1-HI1 achieves between 70 and 85% of the available network bandwidth, although the maximum efficiency is obtained by HI1-CC2 and HS1-CC2, especially when using block sizes $>$ 1 MB (nearly 98%). This fact allows HI1-CC2 to obtain the best productivity up to a block size of 1 MB (see the bottom right graph). From 4 MB on the highest-performance and most expensive HI1-HI1 configuration offsets its price. CC2-CC2 gets the worst productivity from 1 MB on, even below HS1-CC2.

To sum up, these results have shown that: (1) the HI1-HI1 configuration is the best performer for both write and read operations using any block size; (2) the overall performance of heterogeneous cluster deployments (i.e., HI1-CC2 and HS1-CC2) is severely limited by the network bandwidth, as their instances cannot be located in the same placement group; and (3) taking costs into account, the HI1-CC2 configuration

[4]The TroveMethod parameter specifies how both metadata and data are stored and managed by the OrangeFS servers.

achieves the best productivity, except when reading block sizes > 1 MB, a scenario where HI1-HI1 is the best configuration.

*5.2.2. Distributed File System Performance*

Figure 6 shows the aggregated bandwidth of HDFS (left graphs) for write and read operations using a baseline cluster that consists of one instance acting as master node (running JobTracker/NameNode) and multiple instances acting as slave nodes (running TaskTrackers/DataNodes), all connected to the 10 Gigabit Ethernet network. As mentioned in Section 4.2, these results have been obtained using the Enhanced DFSIO benchmark included in the Intel HiBench suite, writing and reading 512 files of 512 MB (i.e., 256 GB in total). In these experiments, two different cluster sizes have been evaluated using 8 and 16 slave instances, together with the master node of the same instance type. Therefore, all the instances are located in the same placement group, thus relying on homogeneous cluster deployments. In addition, the right graphs show the productivity results in terms of the aggregated bandwidth per US$, according to the hourly cost of the different EC2-based Hadoop clusters considered in this analysis (shown in the last column of Table 4).

Moreover, each DataNode instance (i.e., slave node) uses all the available locally attached ephemeral disks (e.g., 24 disks for HS1 instances) to store HDFS data. The ephemeral disks have been configured as a Just a Bunch Of Disks (JBOD) (i.e., each disk is accessed directly as an independent drive [5]). This is the most recommended setting for Hadoop clusters, as HDFS does not benefit from using RAID for the DataNode storage. The redundancy provided by RAID is not needed since HDFS handles it by replication between nodes. Furthermore, the RAID 0 level, which is commonly used to increase performance, turns out to be slower than the JBOD configuration, as HDFS uses round-robin allocation of blocks and data across all the available disks.

Regarding Hadoop settings, several configurations have been tested in order to choose the one with the best observed performance. However, the main goal of these experiments focuses on the comparison between different instance types for running Hadoop-based clusters on Amazon EC2 under common setting rules. Therefore, our setup is not intended as a guide to optimize the overall Hadoop performance, which is usually highly workload-dependent. Taking this into account, our main Hadoop settings are as follows: (1) the block size of HDFS is set to 128 MB. (2) The replication factor of HDFS is set to 2. (3) The I/O buffer size is set to 64 KB. (4) The number of map/reduce tasks that are allowed to run in each TaskTracker instance (i.e., slave node) is configured as shown in Table 4. This configuration follows the widespread rule saying that the number of map and reduce tasks should be set to the number of available physical cores in the TaskTracker instance, considering that the DataNode and TaskTracker processes would use 2 hyper-threaded cores. (5) The ratio of map/reduce tasks is 3:1. (6) The available memory to use while sorting files is set to 200 MB. And (7) the compression of the intermediate output data during the map phase is enabled using the *Snappy* codec [85]. This reduces the network overhead as the map output is compressed before being transferred to the reducers.

Concerning write performance results (see the top left graph in Figure 6), the use of 8-slave clusters shows similar HDFS bandwidths, around 1400 MB/s, and thus the CC2-based cluster is clearly the most cost-effective in terms of productivity, as shown in the top right graph. However, storage-optimized instances (i.e., HI1 and HS1) obtain 34% and 44% more aggregated bandwidth, respectively, than CC2 instances when using 16-slave clusters. Nevertheless, CC2 remains as the most productive choice, although followed closely by HI1, whereas the HS1-based cluster, which is the most expensive, remains as the least competitive. Note that while the storage-optimized instances have slightly increased their productivity when doubling the number of slaves, CC2 has decreased 27%. Regarding read results (see the bottom left graph), HS1 obtains the highest bandwidth both for 8 and 16 slaves, around 2920 and 5630 MB/s, respectively. These results are 9% and 70% higher than HI1 and CC2 bandwidths using 8 slaves and 7% and 80% higher using 16 slaves, respectively. Nevertheless, the high performance of the HS1 cluster is not enough to be the best option in terms of productivity due to its high price, even being outperformed by CC2 (see the bottom right graph), the worst cluster in terms of read performance. Here, HI1 is clearly the best choice when taking into account the incurred costs, achieving up to 13% higher productivity than CC2. In this case, all the configurations are able to maintain (or even to increase in the case of HI1) their productivity when doubling the number of slaves.

Another interesting comparison can be conducted taking into account the number of map and reduce tasks. Hence, an 8-slave CC2-based cluster provides the same capacity in terms of map/reduce tasks (i.e., 96/32) as a 16-slave cluster using storage-optimized instances, as can be seen in Table 4. This fact is due to the different number of physical cores provided by each instance type (16 vs 8 cores for CC2 and storage-optimized instances, respectively). For the write operation, the 16-slave HS1-based cluster doubles the performance of the 8-slave CC2-based cluster, but significantly incurring higher costs (the same would apply to HI1). For the read operation, the storage-optimized instances obtain up to 3 times higher bandwidth than CC2, which allows the HI1-based cluster to be the most productive in this case.

[5]We have specified a storage directory for each ephemeral disk using the *dfs.data.dir* property of Hadoop.

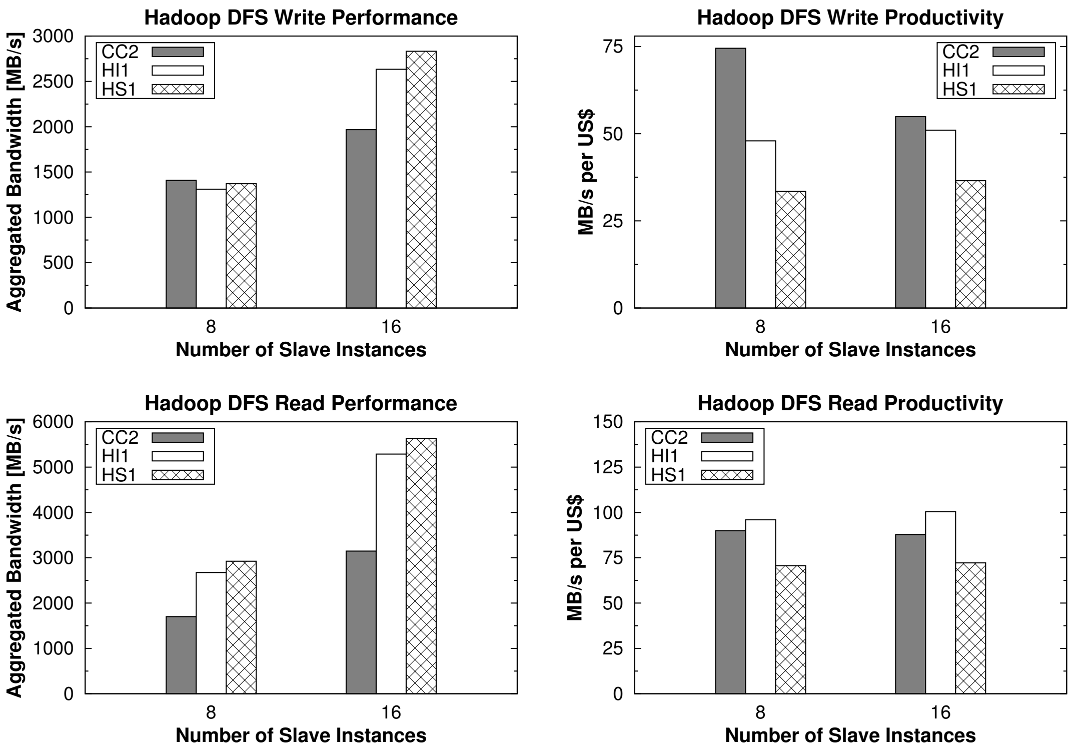


**FIGURE 6.** HDFS performance and productivity using 8 and 16 slave instances

**TABLE 4.** Number of map/reduce tasks and hourly cost using on-demand instances of the EC2-based Hadoop clusters

| Hadoop Cluster | #Map/Reduce tasks per slave node (8 - 16 slaves) | Hourly Cost (8 - 16 slaves) |
|---|---|---|
| **CC2-based** | 12/4 (96/32 - 192/64) | \$18 - \$34 |
| **HI1-based** | 6/2 (48/16 - 96/32) | \$27.9 - \$52.7 |
| **HS1-based** | 6/2 (48/16 - 96/32) | \$41.4 - \$78.2 |

The main conclusions that can be drawn from these results are: (1) the HS1-based cluster is the best option in terms of HDFS performance, especially when using 16 slaves, but followed closely by HI1; (2) the high price of the HS1 instance discourages its use, favoring HI1 instances as the preferred choice when costs are taken into account without trading off much performance; and (3) comparing clusters in terms of the same map/reduce capacity, the storage-optimized instances provide up to 2 and 3 times higher write and read bandwidths, respectively, than CC2 instances.

## 5.3. Data-intensive Parallel Application Performance

The performance of two I/O-intensive HPC applications (Section 5.3.1) and four real-world MapReduce workloads (Section 5.3.2) has been analyzed at the application level. As mentioned in Section 4.2, the BT-IO kernel from the NPB suite and the FLASH-IO code have been selected for the HPC software stack evaluation. In addition, the Sort, WordCount, Mahout PageRank and Hive Aggregation workloads have been selected for the Big Data counterpart. Finally, the performance variability of the evaluated workloads is briefly discussed in Section 5.3.3.

### *5.3.1. I/O-Intensive HPC Applications*

The NPB BT kernel solves systems of block-tridiagonal equations in parallel. BT-IO extends the BT kernel by adding support for periodic solution checkpointing using the MPI-IO interface. In these experiments, the NPB class C workload was selected, whereas the I/O size was the “full” subtype. With these settings, all the parallel processes append data to a single file through 40 collective MPI-IO write operations, resulting in an output data file sized around 6.8 GB.

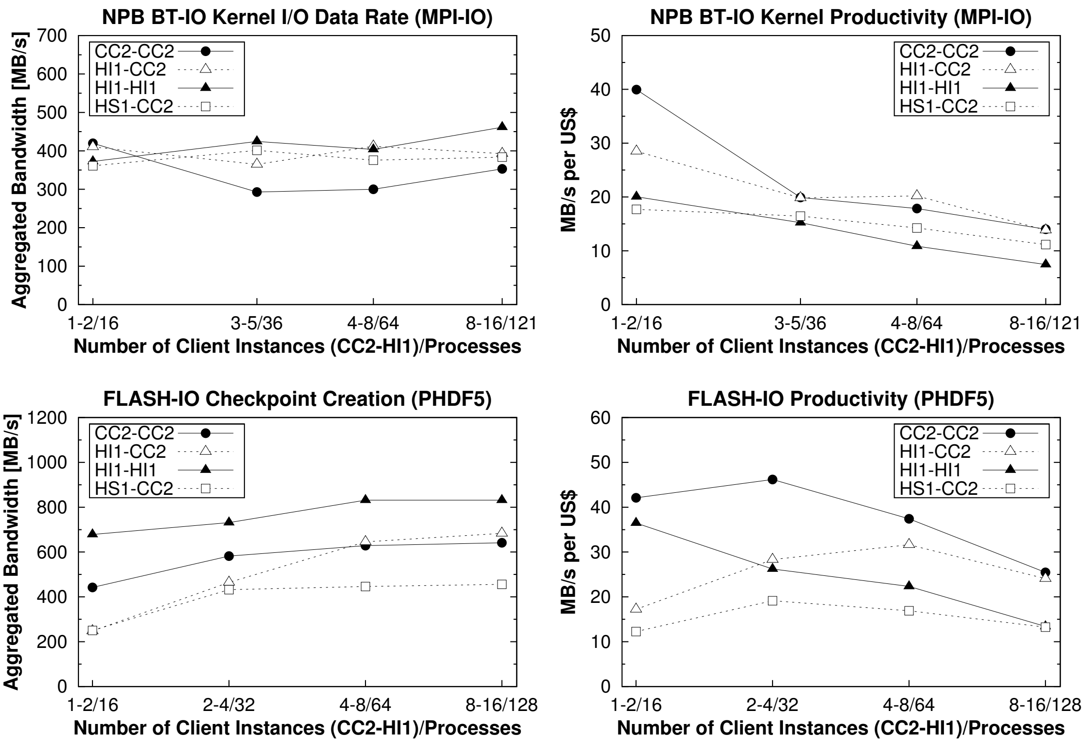


**FIGURE 7.** NPB BT-IO and FLASH-IO results using OrangeFS with 4 I/O servers

The default I/O frequency, which consists of appending data to the shared output file every 5 computation time steps, was used. Note that this kernel requires that the number of client processes be square numbers. The FLASH-IO kernel simulates the I/O pattern of FLASH [83], a parallel hydrodynamics application that simulates astrophysical thermonuclear flashes in two or three dimensions. The parallel I/O routines of the FLASH-IO code are identical to those used by FLASH, so their performance closely reflects the full parallel I/O performance of the application. This kernel recreates the primary data structures in the FLASH application and produces a checkpoint file sized around 7.6 GB (for 128 clients) using PHDF5, together with smaller plotfiles for visualization and analysis. Finally, the experiments have been conducted under the same cluster configurations previously used in the parallel file system evaluation (see Table 3 in Section 5.2.1).

Figure 7 presents NPB BT-IO (top graphs) and FLASH-IO (bottom graphs) performance and productivity results (left and right graphs, respectively). Regarding BT-IO results, HI1- and HS1-server based configurations achieve the highest bandwidths from 36 processes on, being the HI1-HI1 cluster the best performer in accordance with the previous parallel file system evaluation for the write operation (see the top left graph in Figure 4), obtaining up to 30% higher bandwidth than the CC2-CC2 cluster (460 MB/s vs 350 MB/s for 121 processes). Nevertheless, HI1-CC2 is the configuration with the highest productivity from 36 processes on, as happened before (see the bottom right graph in Figure 4). Here, the performance advantage obtained by HI1-HI1 is not enough to be an interesting cost-effective option, even being outperformed by HS1-CC2. Note that this code is also a computation- and communication-intensive application that performs lots of MPI communication calls, increasing the communication overhead with the number of processes. Hence, MPI communications across client instances share the interconnection network with MPI-IO communications (i.e., I/O operations) between client and server instances. This sharing reduces the available network bandwidth, which limits parallel I/O performance. This limitation becomes more noticeable for CC2-client based configurations, which run 16 parallel processes per instance, due to the poorer ratio between CPU power and network bandwidth. Furthermore, the computation phase of this kernel allows to overlap I/O with computation. This fact boosts the slowest storage configuration for writing, which was HS1-CC2 according to the parallel file system evaluation (see the top left graph in Figure 4), now obtaining more competitive performance results.

Regarding FLASH-IO results, HI1-HI1 shows again the best performance, obtaining more than 830 MB/s for 64 and 128 processes, up to 23% and 30% higher aggregated bandwidth than HI1-CC2 and CC2-CC2 clusters, respectively. The HS1-CC2 cluster gets stucked at 455 MB/s from 32 processes on, being the worst performer as occurred before in the parallel file system evaluation for the write operation. Thus, the underlying network bandwidth significantly limits its overall I/O performance. Note that the FLASH-IO code simply creates the checkpoint file using PHDF5, which is implemented on top of MPI-IO. This means that, unlike the BT-IO kernel, this code is neither computation-intensive nor makes an extensive use of MPI communication routines. Therefore, the FLASH-IO results are more in tune with the write performance at the parallel file system level than the BT-IO results, but using a high-level library (PHDF5) instead of using directly the MPI-IO interface. In terms of productivity, CC2-CC2 becomes the best option, but closely followed by the HI1-CC2 cluster when 64 or more processes are used. HI1-HI1 shows again poor productivity, especially from 32 processes on, which makes it a bad choice when costs are taken into account.

The main conclusions of this evaluation are: (1) in terms of performance, HI1-HI1 remains as the best option for I/O-intensive HPC applications, in accordance with the previous evaluation at the parallel file system level; and (2) in terms of productivity, it is generally advisable to use CC2 instances as clients in order to save costs when using a high number of cores, and combine them with HI1 or CC2 as storage servers to maximize performance.

### *5.3.2. MapReduce-based Workloads*

Figure 8 shows performance (left graphs) and cost results (right graphs) of the selected Hadoop applications from the Intel HiBench suite (see details in Section 4.2). In this case, these workloads do not report any aggregated bandwidth metric as output. Hence, the performance metric reported is the time (in hours) required to run 1,000 executions of each evaluated application. Consequently, the productivity metric reported is the total execution cost in US$, which represents the cost of having each Hadoop cluster running during the number of hours required to complete the 1,000 executions for each experiment. The experiments have been conducted under the same Hadoop settings and cluster configurations previously used in the distributed file system evaluation (see Table 4 in Section 5.2.2). The specific workloads for these applications have been configured as summarized in Table 5. This table shows the total data that are read and written from and to HDFS, which represents the job/map input and job/reduce output, respectively. The last column of the table shows the total data that are transferred during the shuffle phase (i.e., data from output mappers that are read by reducers). As mentioned in Section 5.2.2, these intermediate data are compressed before being transferred, thus reducing the network overhead.

Since the Sort workload transforms data from one representation to another, the job output has the same size as the job input (102 GB), as can be seen in Table 5. Although this workload is I/O bound in nature, it also presents moderate CPU utilization during the map and shuffle phases, especially due to the intermediate data compression, and low CPU utilization and heavy disk I/O during the reduce phase. In addition, considering the large amount of shuffle data (53 GB), even though compressed, it is expected to be the most network-intensive workload under evaluation. Results using 8-slave clusters show that CC2 is able to outperform HI1 and HS1 configurations by 20%, thanks to the higher map/reduce capacity of the CC2 cluster due to the availability of more CPU resources, which seem to be of great importance, especially during data compression. However, CC2 only reduces its execution time by 22% when doubling the number of slaves, whereas HI1 and HS1 reduce it by 40% and 42%, respectively, but far from linear scalability due to the intensive use of the network. This allows storage-optimized instances to slightly outperform CC2 when using 16 slaves, due to their higher HDFS bandwidth, even taking into account that they have half the map/reduce capacity of CC2 (see Table 4). Nevertheless, the cost of the CC2 cluster continues to be the lowest, a pattern that is maintained in the remaining workloads. If the comparison is done using the same map/reduce capacity, the 16-slave clusters using HI1 and HS1 instances outperform the 8-slave CC2 cluster by 24% and 27%, respectively. However, these performance improvements are not enough to turn optimized-storage instances into interesting options when considering the associated costs.

The shuffle data (12.7 MB) and job output (25.5 KB) of WordCount are much smaller than the job input (102 GB), as this application extracts a small amount of information from a large input data set. Consequently, the WordCount workload is mostly CPU bound, especially during the map phase, having very high CPU utilization and light disk/network I/O. Therefore, the CC2 instance type, which provides the largest CPU resources, shows the best performance and productivity results both for 8- and 16-slave clusters. For instance, the 8-slave CC2 cluster achieves up to 1.9 and 2.1 times higher performance than HI1 and HS1 clusters, respectively. These numbers are reduced to 1.7 and 1.8 times when 16-slave clusters are used. Therefore, CC2 obtains again lower execution time reduction (42%) when doubling the number of slaves compared to HI1 (46%) and HS1 (50%, i.e. linear scalability). In this case, the use of 16-slave storage-

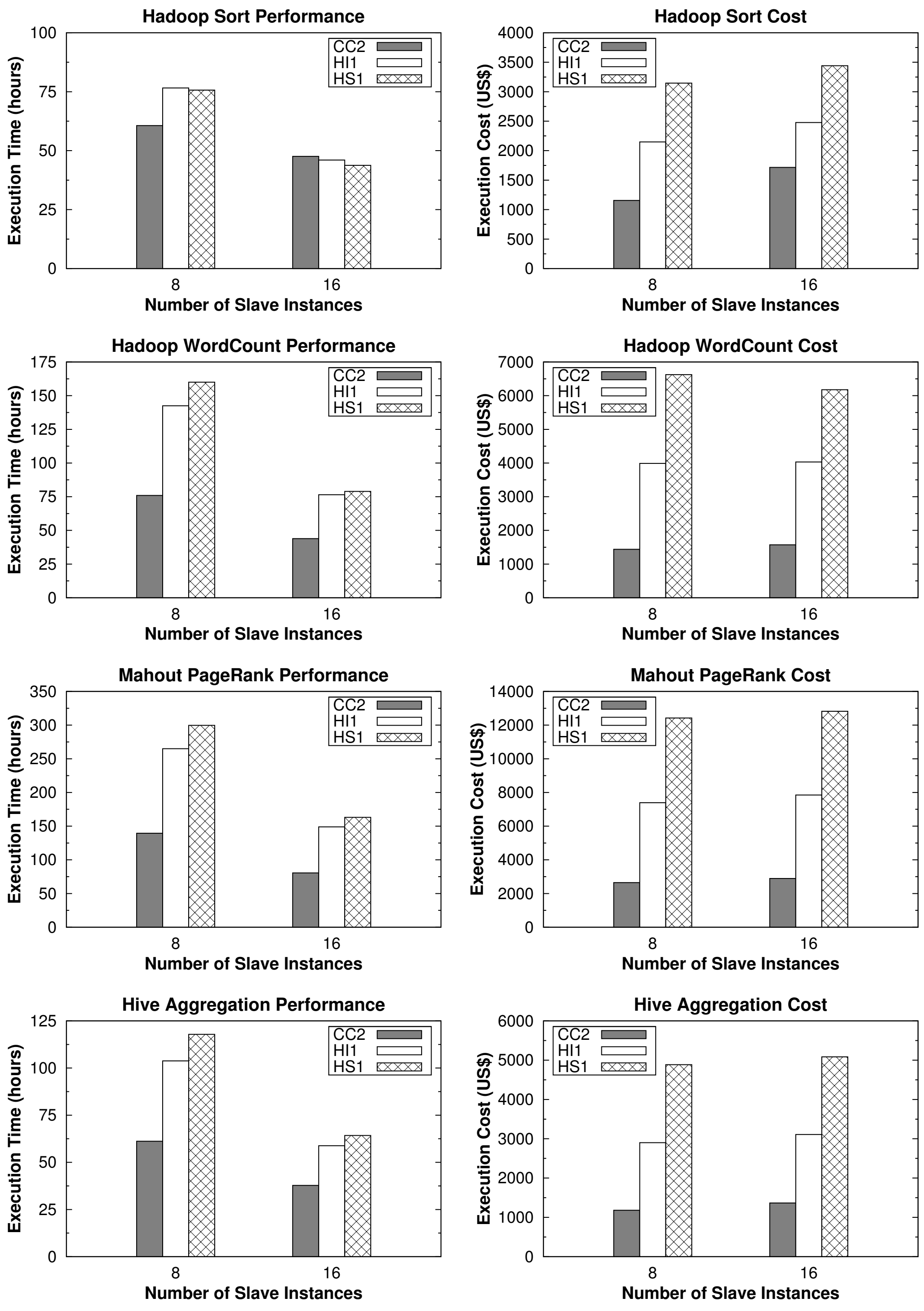


**FIGURE 8.** Hadoop performance and cost results using 8 and 16 slave instances

**TABLE 5.** Data size for map, reduce and shuffle phases

| Workload | Job/Map Input (HDFS Read) | Job/Reduce Output (HDFS Write) | Shuffle Data |
|---|---|---|---|
| **Sort** | 102 GB | 102 GB | 53 GB |
| **WordCount** | 102 GB | 25.5 KB | 12.7 MB |
| **PageRank** | 89.8 GB | 49.6 GB | 4.6 GB |
| **Aggregation** | 75 GB | 14.3 GB | 7.5 GB |

optimized clusters does not outperform the 8-slave CC2 cluster, as occurred for Sort. In fact, the performance of CC2 using 8 slaves is similar to that of HI1 and HS1 clusters using 16 slaves, but having 2.8 and 4.3 times lower costs, respectively.

The Mahout PageRank and Hive Aggregation workloads share similar characteristics. They are more CPU bound during the map phase and more disk I/O bound during the reduce phase, mostly due to the output to HDFS (49.6 GB and 14.3 GB, respectively). They also present low to medium network I/O, as the amount of shuffle data is relatively small (4.6 GB and 7.5 GB, respectively). Hence, both workloads show very similar results, also similar to those of WordCount. Consequently, the CC2-based cluster is again the best option, both in terms of performance and cost. For instance, CC2 obtains 2.1 times higher performance and 4.6 times lower cost than HS1 when using 8 slaves for the PageRank workload. Furthermore, it is not worth using 16-slave storage-optimized clusters, which provide the same map/reduce capacity than the 8-slave CC2 cluster, as similar or even lower performance is obtained but incurring significantly higher costs.

These results have shown that: (1) storage-optimized instances do not seem to be the most suitable choice for the evaluated MapReduce workloads. As explained before, the main reason is that these instances provide poorer CPU resources than CC2, both in terms of number of physical cores (8 vs 16) and computational power per core (4.4 vs 5.5 ECUs), as shown in Table 1. (2) This fact encourages the use of the CC2-based cluster which, using the same number of slaves, provides twice the map/reduce capacity of clusters based on storage-optimized instances. (3) The CC2-based cluster usually achieves significantly higher performance and lower cost, which offsets the fact that the CC2 instance type provides lower underlying I/O performance, as shown in Section 5.1. And (4) if the instance types are compared using the same map/reduce capacity (i.e., 8-slave CC2 cluster vs 16-slave HI1 and HS1 clusters), only Sort, the most I/O-intensive workload under evaluation, experiences some performance benefit when using storage-optimized instances, but incurring more costs.

### *5.3.3. Analysis of Performance Variability*

Performance unpredictability in the cloud can be an important issue for researchers because of repeatability of results. In this paper, the main guidelines and hints suggested by Schad et al. [84] have been followed in order to minimize the variance and maximize the repeatability of our experiments in Amazon EC2. Examples of these guidelines are always specifying one availability zone when launching the instances (*us-east-1d*, as mentioned in Section 4.3) and reporting the underlying system hardware of the evaluated instances (see Table 1). In this section, a brief analysis of the performance variability at the application level is presented, as it takes into account the likely variability of all the lower levels (i.e., storage infrastructure, interconnection network, cluster file system, I/O middleware and MapReduce frameworks, and high-level I/O libraries and tools), showing its impact on the performance of real applications.

Figure 9 presents the performance variability for the data-intensive applications evaluated in Sections 5.3.1 (left graph) and 5.3.2 (right graph). These graphs show the measure of the mean value and include error bars to indicate the measure of the minimum sample (bottom arrow) and the maximum sample (top arrow). The results are shown using the largest cluster configuration for each corresponding application (i.e., a 128-core cluster for I/O-intensive HPC applications and a 16-slave Hadoop cluster for MapReduce workloads).

As main conclusion, the evaluated MapReduce workloads generally present negligible variance in their performance (see the right graph), except for the Sort application executed on the CC2 cluster. This variability is significantly lower than the one observed for HPC applications, as shown in the left graph. This is motivated by the fact that these MapReduce workloads are more computationally intensive than the BT-IO and FLASH-IO applications, especially due to the intermediate data compression and map/reduce phases. In fact, as mentioned in Section 5.3.1, BT-IO also presents higher computation intensiveness than FLASH-IO and thus its variability is also lower. Among the evaluated instances, HI1 shows the highest variability, particularly for the BT-IO and FLASH-IO codes, which present a write-only access pattern. This fact can be due to the varying overhead of the background tasks associated with the SSD internal architecture. Hence, the inherent variable write performance of SSD disks, especially when using RAID, is due to the internal garbage collection process that can block incoming requests that are mapped to the flash chips currently performing erase operations [86].

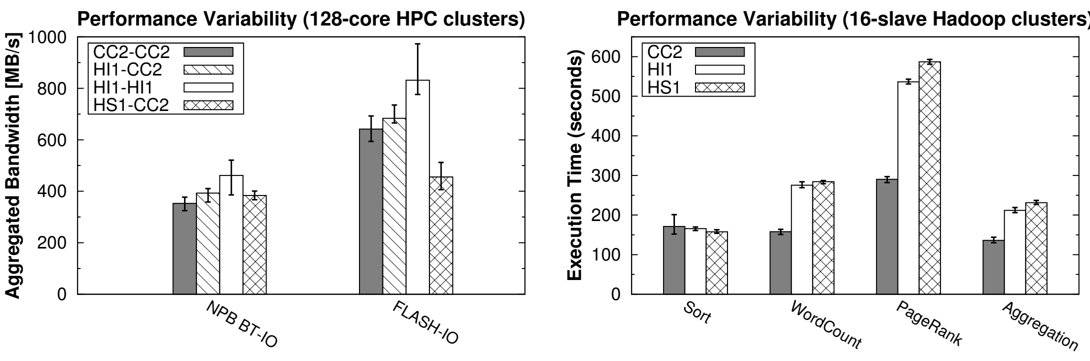


**FIGURE 9.** Performance variability of data-intensive computing applications

## 6. CONCLUSIONS

Cloud computing platforms are becoming widely available and are gaining significant popularity in many domains, as a convenient way to virtualize data centers and increase the flexibility in the use of computational resources. Amazon Web Services is the leading commercial public cloud provider, whose EC2 IaaS cloud service provides end users with reliable access to on-demand resources to run their applications. At the same time, scientific research is increasingly reliant on the processing of very large amounts of data. In fact, current data-intensive applications generally demand significant computational resources together with scalable cluster file systems. Public IaaS clouds can satisfy the increasing processing requirements of these applications while offering high flexibility and promising cost savings.

The main contributions of this paper are: (1) extensive assessment of the suitability of using the Amazon EC2 IaaS cloud platform for running data-intensive computing applications; (2) a thorough performance study of the storage-optimized family of EC2 instances, which provide direct-attached storage devices intended to be well suited for applications with specific disk I/O requirements; and (3) a detailed performance, cost and variability analysis of four EC2 instance types that provide 10 Gigabit Ethernet, conducting multiple experiments at several layers and using a representative suite of benchmarking tools (IOzone, IOR, Intel HiBench), cluster file systems (OrangeFS, HDFS), I/O middleware (MPI-IO, HDF5), distributed computing frameworks (Apache Hadoop), I/O-intensive parallel codes for HPC (NPB BT-IO and FLASH-IO) and MapReduce workloads for Big Data analysis (Sort, WordCount, PageRank, Aggregation).

The analysis of the experimental results points out that the unique configurability and flexibility advantage offered by Amazon EC2, almost impossible to achieve in traditional platforms, is critical for increasing performance and/or reduce costs. Hence, this paper has revealed that the suitability of using EC2 resources for running data-intensive applications is highly workload-dependent. Furthermore, the most suitable configuration for a given application heavily depends on whether the main aim is to obtain the maximum performance or, instead, minimize the cost (or maximize the productivity). Therefore, one of the key contributions of this work is an in-depth and exhaustive study that provides guidelines for scientists and researchers to increase significantly the performance (or reduce the cost) of their applications in Amazon EC2. Finally, our main outcomes indicate that current data-intensive applications can benefit from tailored EC2-based virtual clusters, enabling end users to obtain the highest performance and cost-effectiveness in the cloud.

## FUNDING

This work was supported by the Ministry of Economy and Competitiviness of Spain and ERDF funds of the European Union [Project TIN2013-42148-P]; by the Galician Government [Project GRC2013/055]; by the FPU Program of the Ministry of Education of Spain [ref. AP2010-4348]; and by an Amazon Web Services (AWS) Research Grant.

## REFERENCES

[1] Gorton, I., Greenfield, P., Szalay, A., and Williams, R. (2008) Data-intensive computing in the 21st century. *Computer*, **41**, 30–32.

[2] Buyya, R., Yeo, C. S., Venugopal, S., Broberg, J., and Brandic, I. (2009) Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility. *Future Generation Computer Systems*, **25**, 599–616.

[3] Amazon Web Services Inc. `http://aws.amazon.com/`. [Last visited: July 2014].

[4] Google Inc. Google Compute Engine. `https://cloud.google.com/products/compute-engine/`. [Last visited: July 2014].

[5] Microsoft Corporation. Microsoft Azure. `http://azure.microsoft.com/`. [Last visited: July 2014].

[6] Rackspace Inc. Rackspace Public Cloud. `http://www.rackspace.com/`. [Last visited: July 2014].

[7] Top 10 cloud platforms of 2013. `http://yourstory.com/2013/12/top-10-cloud-platforms-2013/`. [Last visited: July 2014].

[8] Amazon Web Services Inc. Amazon Elastic Compute Cloud (Amazon EC2). `http://aws.amazon.com/ec2`. [Last visited: July 2014].

[9] Top 20 cloud IaaS providers of 2014. `http://www.crn.com/slide-shows/cloud/240165705/the-20-coolest-cloud-infrastructure-iaas-vendors-of-the-2014-cloud-100.htm/pgno/0/1`. [Last visited: July 2014].

[10] Amazon Web Services Inc. High Performance Computing using Amazon EC2. `http://aws.amazon.com/ec2/hpc-applications/`. [Last visited: July 2014].

[11] Jackson, K. R., Ramakrishnan, L., Muriki, K., Canon, S., Cholia, S., Shalf, J., Wasserman, H. J., and Wright, N. J. (2010) Performance analysis of high performance computing applications on the Amazon web services cloud. *Proc. 2nd IEEE International Conference on Cloud Computing Technology and Science (CloudCom'10)*, Indianapolis, IN, USA, 30 November–3 December, pp. 159–168. IEEE Computer Society, Washington, DC, USA.

[12] Mauch, V., Kunze, M., and Hillenbrand, M. (2013) High performance cloud computing. *Future Generation Computer Systems*, **29**, 1408–1416.

[13] Napper, J. and Bientinesi, P. (2009) Can cloud computing reach the TOP500? *Proc. Combined Workshops on UnConventional High Performance Computing Workshop plus Memory Access Workshop (UCHPC-MAW'09)*, Ischia, Italy, 18–20 May, pp. 17–20. ACM, New York, NY, USA.

[14] Walker, E. (2008) Benchmarking Amazon EC2 for high-performance scientific computing. *USENIX;login:*, **33**, 18–23.

[15] Dean, J. and Ghemawat, S. (2008) MapReduce: Simplified data processing on large clusters. *Communications of the ACM*, **51**, 107–113.

[16] Amazon Web Services Inc. Amazon EC2 instance types. `http://aws.amazon.com/ec2/instance-types/`. [Last visited: July 2014].

[17] Apache Hadoop. `http://hadoop.apache.org/`. [Last visited: July 2014].

[18] Thakur, R., Gropp, W. D., and Lusk, E. (1999) On implementing MPI-IO portably and with high performance. *Proc. 6th Workshop on I/O in Parallel and Distributed Systems (IOPADS'99)*, Atlanta, GA, USA, 5 May, pp. 23–32. ACM, New York, NY, USA.

[19] Roloff, E., Birck, F., Diener, M., Carissimi, A., and Navaux, P. O. A. (2012) Evaluating high performance computing on the Windows Azure platform. *Proc. 5th IEEE International Conference on Cloud Computing (CLOUD'12)*, Honolulu, HI, USA, 24–29 June, pp. 803–810. IEEE Computer Society, Washington, DC, USA.

[20] He, Q., Zhou, S., Kobler, B., Duffy, D., and McGlynn, T. (2010) Case study for running HPC applications in public clouds. *Proc. 19th ACM International Symposium on High Performance Distributed Computing (HPDC'10)*, Chicago, IL, USA, 21–25 June, pp. 395–401. ACM, New York, NY, USA.

[21] Expósito, R. R., Taboada, G. L., Ramos, S., Touriño, J., and Doallo, R. (2013) Performance analysis of HPC applications in the cloud. *Future Generation Computer Systems*, **29**, 218–229.

[22] Mehrotra, P., Djomehri, J., Heistand, S., Hood, R., Jin, H., Lazanoff, A., Saini, S., and Biswas, R. (2013) Performance evaluation of Amazon elastic compute cloud for NASA high-performance computing applications. *Concurrency and Computation: Practice and Experience (in press)*, `http://dx.doi.org/10.1002/cpe.3029`.

[23] Ostermann, S., Iosup, A., Yigitbasi, N., Prodan, R., Fahringer, T., and Epema, D. (2009) A performance analysis of EC2 cloud computing services for scientific computing. *Proc. 1st International Conference on Cloud Computing (CLOUDCOMP'09)*, Munich, Germany, 19–21 October, pp. 115–131. Springer-Verlag, Berlin, Germany.

[24] Juve, G., Deelman, E., Berriman, G. B., Berman, B. P., and Maechling, P. (2012) An evaluation of the cost and performance of scientific workflows on Amazon EC2. *Journal of Grid Computing*, **10**, 5–21.

[25] Vecchiola, C., Pandey, S., and Buyya, R. (2009) High-performance cloud computing: A view of scientific applications. *Proc. 10th International Symposium on Pervasive Systems, Algorithms, and Networks (ISPAN'09)*, Kaoshiung, Taiwan, 14–16 December, pp. 4–16. IEEE Computer Society, Washington, DC, USA.

[26] Iosup, A., Ostermann, S., Yigitbasi, N., Prodan, R., Fahringer, T., and Epema, D. (2011) Performance analysis of cloud computing services for many-tasks scientific computing. *IEEE Transactions on Parallel and Distributed Systems*, **22**, 931–945.

[27] Evangelinos, C. and Hill, C. N. (2008) Cloud computing for parallel scientific HPC applications: Feasibility of running coupled atmosphere-ocean climate models on Amazon's EC2. *Proc. 1st Workshop on Cloud Computing and Its Applications (CCA'08)*, Chicago, IL, USA, 22–23 October, pp. 1–6. ACM, New York, NY, USA.

[28] Shafer, J. (2010) I/O virtualization bottlenecks in cloud computing today. *Proc. 2nd Workshop on I/O Virtualization (WIOV'10)*, Pittsburgh, PA, USA, March 13, pp. 5:1–5:7. USENIX Association, Berkeley, CA, USA.

[29] Ghoshal, D., Canon, R. S., and Ramakrishnan, L. (2011) I/O performance of virtualized cloud environments. *Proc. 2nd International Workshop on Data Intensive Computing in the Clouds (DataCloud-SC'11)*, Seattle, WA, USA, 12–18 November, pp. 71–80. ACM, New York, NY, USA.

[30] Zhai, Y., Liu, M., Zhai, J., Ma, X., and Chen, W. (2011) Cloud versus in-house cluster: Evaluating Amazon cluster compute instances for running MPI applications. *Proc. 23rd ACM/IEEE Supercomputing Conference (SC'11, State of the Practice Reports)*, Seattle, WA, USA, 12–18 November, pp. 11:1–11:10. ACM, New York, NY, USA.

[31] Expósito, R. R., Taboada, G. L., Ramos, S., González-Domínguez, J., Touriño, J., and Doallo, R. (2013)

Analysis of I/O performance on an Amazon EC2 cluster compute and high I/O platform. *Journal of Grid Computing*, **11**, 613–631.

[32] Liu, M., Zhai, J., Zhai, Y., Ma, X., and Chen, W. (2011) One optimized I/O configuration per HPC application: Leveraging the configurability of cloud. *Proc. 2nd ACM SIGOPS Asia-Pacific Workshop on Systems (APSys'11)*, Shanghai, China, 11–12 July, pp. 1–5. ACM, New York, NY, USA.

[33] Gunarathne, T., Wu, T.-L., Qiu, J., and Fox, G. (2010) MapReduce in the clouds for science. *Proc. 2nd IEEE International Conference on Cloud Computing Technology and Science (CloudCom'10)*, Indianapolis, IN, USA, 30 November–3 December, pp. 565–572. IEEE Computer Society, Washington, DC, USA.

[34] Gunarathne, T., Zhang, B., Wu, T.-L., and Qiu, J. (2013) Scalable parallel computing on clouds using Twister4Azure iterative MapReduce. *Future Generation Computer Systems*, **29**, 1035–1048.

[35] Moise, D. and Carpen-Amarie, A. (2012) MapReduce applications in the cloud: A cost evaluation of computation and storage. *Proc. 5th International Conference on Data Management in Cloud, Grid and P2P Systems (Globe 2012)*, Vienna, Austria, 5–6 September, pp. 37–48. Springer-Verlag, Berlin, Germany.

[36] Fadika, Z., Govindaraju, M., Canon, R., and Ramakrishnan, L. (2012) Evaluating Hadoop for data-intensive scientific operations. *Proc. 5th IEEE International Conference on Cloud Computing (CLOUD'12)*, Honolulu, HI, USA, 24–29 June, pp. 67–74. IEEE Computer Society, Washington, DC, USA.

[37] Zhang, C., De Sterck, H., Aboulnaga, A., Djambazian, H., and Sladek, R. (2009) Case study of scientific data processing on a cloud using Hadoop. *Proc. 23rd International Conference on High Performance Computing Systems and Applications (HPCS'09)*, Kingston, ON, Canada, 14–17 June, pp. 400–415. Springer-Verlag, Berlin, Germany.

[38] Elnozahy, E. N., Alvisi, L., Wang, Y.-M., and Johnson, D. B. (2002) A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys*, **34**, 375–408.

[39] Bent, J., Gibson, G., Grider, G., McClelland, B., Nowoczynski, P., Nunez, J., Polte, M., and Wingate, M. (2009) PLFS: A checkpoint filesystem for parallel applications. *Proc. 21st ACM/IEEE Supercomputing Conference (SC'09)*, Portland, OR, USA, 14–20 November, pp. 21:1–21:12. ACM, New York, NY, USA.

[40] Carns, P. H., Harms, K., Allcock, W., Bacon, C., Lang, S., Latham, R., and Ross, R. B. (2011) Understanding and improving computational science storage access through continuous characterization. *Proc. 27th IEEE Symposium on Mass Storage Systems and Technologies (MSST'11)*, Denver, CO, USA, 23–27 May, pp. 1–14. IEEE Computer Society, Washington, DC, USA.

[41] Shan, H., Antypas, K., and Shalf, J. (2008) Characterizing and predicting the I/O performance of HPC applications using a parameterized synthetic benchmark. *Proc. 20th ACM/IEEE Supercomputing Conference (SC'08)*, Austin, TX, USA, 15–21 November, pp. 42:1–42:12. IEEE Press, Piscataway, NJ, USA.

[42] Doulkeridis, C. and Norvag, K. (2014) A survey of large-scale analytical query processing in MapReduce. *The VLDB Journal*, **23**, 355–380.

[43] Thusoo, A., Shao, Z., Anthony, S., Borthakur, D., Jain, N., Sen Sarma, J., Murthy, R., and Liu, H. (2010) Data warehousing and analytics infrastructure at Facebook. *Proc. 2010 ACM SIGMOD International Conference on Management of Data (SIGMOD'10)*, Indianapolis, IN, USA, 6–10 June, pp. 1013–1020. ACM, New York, NY, USA.

[44] Borthakur, D. et al. (2011) Apache Hadoop goes realtime at Facebook. *Proc. 2011 ACM SIGMOD International Conference on Management of Data (SIGMOD'11)*, Athens, Greece, 12–16 June, pp. 1071–1080. ACM, New York, NY, USA.

[45] Ananthanarayanan, R., Gupta, K., Pandey, P., Pucha, H., Sarkar, P., Shah, M., and Tewari, R. (2009) Cloud analytics: Do we really need to reinvent the storage stack? *Proc. 1st Workshop on Hot Topics in Cloud Computing (HotCloud'09)*, San Diego, CA, USA, June 15, pp. 15:1–15:5. USENIX Association, Berkeley, CA, USA.

[46] Tantisiriroj, W., Son, S. W., Patil, S., Lang, S. J., Gibson, G., and Ross, R. B. (2011) On the duality of data-intensive file system design: Reconciling HDFS and PVFS. *Proc. 23rd ACM/IEEE Supercomputing Conference (SC'11)*, Seattle, WA, USA, 12–18 November, pp. 67:1–67:12. ACM, New York, NY, USA.

[47] HDF5 home page. `http://www.hdfgroup.org/HDF5/`. [Last visited: July 2014].

[48] Rew, R. and Davis, G. (1990) NetCDF: An interface for scientific data access. *IEEE Computer Graphics and Applications*, **10**, 76–82.

[49] Parallel HDF5 home page. `http://www.hdfgroup.org/HDF5/PHDF5/`. [Last visited: July 2014].

[50] Li, J., Liao, W.-K., Choudhary, A., Ross, R. B., Thakur, R., Gropp, W. D., Latham, R., Siegel, A., Gallagher, B., and Zingale, M. (2003) Parallel netCDF: A high-performance scientific I/O interface. *Proc. 15th ACM/IEEE Supercomputing Conference (SC'03)*, Phoenix, AZ, USA, 15–21 November, pp. 39:1–39:11. ACM, New York, NY, USA.

[51] Schmuck, F. and Haskin, R. (2002) GPFS: A shared-disk file system for large computing clusters. *Proc. 1st USENIX Conference on File and Storage Technologies (FAST'02)*, Monterey, CA, USA, 28–30 January, pp. 231–244. USENIX Association, Berkeley, CA, USA.

[52] Carns, P. H., Ligon III, W. B., Ross, R. B., and Thakur, R. (2000) PVFS: A parallel virtual file system for Linux clusters. *Proc. 4th Annual Linux Showcase & Conference (ALS'00)*, Atlanta, GA, USA, 10–14 October, pp. 317–328. USENIX Association, Berkeley, CA, USA.

[53] Lustre File System. `http://www.lustre.org/`. [Last visited: July 2014].

[54] Orange File System (OrangeFS). `http://www.orangefs.org/`. [Last visited: July 2014].

[55] Chattopadhyay, B., Lin, L., Liu, W., Mittal, S., Aragonda, P., Lychagina, V., Kwon, Y., and Wong, M. (2011) Tenzing a SQL implementation on the MapReduce framework. *Proc. VLDB Endowment (PVLDB)*, **4**, 1318–1327.

[56] Apache Mahout. http://mahout.apache.org/. [Last visited: July 2014].

[57] Ghemawat, S., Gobioff, H., and Leung, S.-T. (2003) The Google file system. *Proc. 19th ACM Symposium on Operating Systems Principles (SOSP'03)*, Bolton Landing, NY, USA, 19–22 October, pp. 29–43. ACM, New York, NY, USA.

[58] Shvachko, K., Kuang, H., Radia, S., and Chansler, R. (2010) The Hadoop distributed file system. *Proc. 26th IEEE Symposium on Mass Storage Systems and Technologies (MSST'10)*, Incline Village, NV, USA, 3–7 May, pp. 1–10. IEEE Computer Society, Washington, DC, USA.

[59] Kosmos Distributed Filesystem (KFS). https://code.google.com/p/kosmosfs/. [Last visited: July 2014].

[60] IOzone filesystem benchmark. http://www.iozone.org/. [Last visited: July 2014].

[61] Shan, H. and Shalf, J. (2007) Using IOR to analyze the I/O performance of HPC platforms. *Proc. 49th Cray User Group Conference (CUG'07)*, Seattle, WA, USA, 7–10 May. CUG, Oak Ridge, TN, USA.

[62] Huang, S., Huang, J., Dai, J., Xie, T., and Huang, B. (2010) The HiBench benchmark suite: Characterization of the MapReduce-based data analysis. *Proc. 2nd IEEE Workshop on Information & Software as Services (WISS'10)*, Long Beach, CA, USA, 1–6 March, pp. 41–51. IEEE Computer Society, Washington, DC, USA.

[63] Wong, P. and Van der Wijngaart, R. F. (2003) NAS parallel benchmarks I/O version 2.4. Technical Report NAS-03-002. NASA Ames Research Center, Moffett Field, CA, USA.

[64] FLASH I/O benchmark routine. http://www.ucolick.org/~zingale/flash_benchmark_io/. [Last visited: July 2014].

[65] Apache Hive. http://hive.apache.org/. [Last visited: July 2014].

[66] MPICH: High performance and widely portable implementation of the MPI standard. http://www.mpich.org/. [Last visited: July 2014].

[67] NASA. NASA Advanced Supercomputing (NAS) Parallel Benchmarks (NPB). http://www.nas.nasa.gov/publications/npb.html. [Last visited: July 2014].

[68] Polte, M., Simsa, J., and Gibson, G. (2008) Comparing performance of solid state devices and mechanical disks. *Proc. 3rd Petascale Data Storage Workshop (PDSW'08)*, Austin, TX, USA, November 17, pp. 1–7. IEEE Press, Piscataway, NJ, USA.

[69] Kasick, M. P., Gandhi, R., and Narasimhan, P. (2010) Behavior-based problem localization for parallel file systems. *Proc. 6th International Conference on Hot Topics in System Dependability (HotDep'10)*, Vancouver, BC, Canada, October 3, pp. 1–13. USENIX Association, Berkeley, CA, USA.

[70] Méndez, S., Rexachs, D., and Luque, E. (2012) Evaluating utilization of the I/O system on computer clusters. *Proc. 18th International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'12)*, Las Vegas, NV, USA, 16–19 July, pp. 366–372. UCMSS, San Diego, CA, USA.

[71] Yu, H., Sahoo, R. K., Howson, C., Almasi, G., Castanos, J. G., Gupta, M., Moreira, J. E., Parker, J. J., Engelsiepen, T. E., Ross, R. B., Thakur, R., Latham, R., and Gropp, W. D. (2006) High performance file I/O for the Blue Gene/L supercomputer. *Proc. 12th International Symposium on High-Performance Computer Architecture (HPCA-12)*, Austin, TX, USA, 11–15 February, pp. 187–196. IEEE Computer Society, Washington, DC, USA.

[72] Méndez, S., Rexachs, D., and Luque, E. (2011) Methodology for performance evaluation of the input/output system on computer clusters. *Proc. 13th IEEE International Conference on Cluster Computing (CLUSTER'11)*, Austin, TX, USA, 26–30 September, pp. 474–483. IEEE Computer Society, Washington, DC, USA.

[73] Saini, S., Talcott, D., Thakur, R., Adamidis, P., Rabenseifner, R., and Ciotti, R. (2007) Parallel I/O performance characterization of Columbia and NEC SX-8 superclusters. *Proc. 21st IEEE International Parallel and Distributed Processing Symposium (IPDPS'07)*, Long Beach, CA, USA, 26–30 March, pp. 1–10. IEEE Computer Society, Washington, DC, USA.

[74] Méndez, S., Rexachs, D., and Luque, E. (2012) Modeling parallel scientific applications through their input/output phases. *Proc. 4th Workshop on Interfaces and Architectures for Scientific Data Storage (IASDS'12)*, Beijing, China, September 28, pp. 7–15. IEEE Computer Society, Washington, DC, USA.

[75] Sigovan, C., Muelder, C., Ma, K.-L., Cope, J., Iskra, K., and Ross, R. B. (2013) A visual network analysis method for large-scale parallel I/O systems. *Proc. 27th IEEE International Parallel and Distributed Processing Symposium (IPDPS'13)*, Boston, MA, USA, 20–24 May, pp. 308–319. IEEE Computer Society, Washington, DC, USA.

[76] Islam, N. S., Lu, X., Rahman, M. W., and Panda, D. K. (2014) SOR-HDFS: A SEDA-based approach to maximize overlapping in RDMA-enhanced HDFS. *Proc. 23rd ACM International Symposium on High-Performance Parallel and Distributed Computing (HPDC'14)*, Vancouver, BC, Canada, 23–27 June, pp. 261–264. ACM, New York, NY, USA.

[77] Wu, D., Luo, W., Xie, W., Ji, X., He, J., and Wu, D. (2013) Understanding the impacts of solid-state storage on the Hadoop performance. *Proc. 1st International Conference on Advanced Cloud and Big Data (CBD'13)*, Nanjing, China, 13–15 December, pp. 125–130. IEEE Computer Society, Washington, DC, USA.

[78] Dimitrov, M., Kumar, K., Lu, P., Viswanathan, V., and Willhalm, T. (2013) Memory system characterization of big data workloads. *Proc. 1st Workshop on Big Data Benchmarks, Performance Optimization, and Emerging Hardware (BPOE'13)*, Silicon Valley, CA, USA, October 8, pp. 15–22. IEEE Computer Society, Washington, DC, USA.

[79] Worringen, J., Träff, J. L., and Ritzdorf, H. (2003) Fast parallel non-contiguous file access. *Proc. 15th ACM/IEEE Supercomputing Conference (SC'03)*, Phoenix, AZ, USA, 15–21 November, pp. 60:1–60:18. ACM, New York, NY, USA.

[80] Thakur, R., Gropp, W., and Lusk, E. (1999) Data sieving and collective I/O in ROMIO. *Proc. 7th Symposium on the Frontiers of Massively Parallel Computation (FRONTIERS'99)*, Annapolis, MD, USA, 21-25

February, pp. 182–189. IEEE Computer Society, Washington, DC, USA.

[81] Coloma, K., Choudhary, A., Liao, W.-K., Ward, L., Russell, E., and Pundit, N. (2004) Scalable high-level caching for parallel I/O. *Proc. 18th IEEE International Parallel and Distributed Processing Symposium (IPDPS'04)*, Santa Fe, NM, USA, 26–30 April, pp. 96b. IEEE Computer Society, Washington, DC, USA.

[82] Ching, A., Choudhary, A., Coloma, K., Liao, W.-K., Ross, R. B., and Gropp, W. D. (2003) Noncontiguous I/O accesses through MPI-IO. *Proc. 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID'03)*, Tokyo, Japan, 12–15 May, pp. 104–111. IEEE Computer Society, Washington, DC, USA.

[83] Fryxell, B., Olson, K., Ricker, P., Timmes, F. X., Zingale, M., Lamb, D. Q., MacNeice, P., Rosner, R., Truran, J. W., and Tufo, H. (2000) FLASH: An adaptive mesh hydrodynamics code for modeling astrophysical thermonuclear flashes. *Astrophysical Journal Supplement*, **131**, 273–334.

[84] Schad, J., Dittrich, J., and Quiané-Ruiz, J.-A. (2010) Runtime measurements in the cloud: Observing, analyzing, and reducing variance. *Proc. VLDB Endowment (PVLDB)*, **3**, 460–471.

[85] A Hadoop library of Snappy compression. `http://code.google.com/p/hadoop-snappy/`. [Last visited: July 2014].

[86] Kim, Y., Lee, J., Oral, S., Dillow, D. A., Wang, F., and Shipman, G. M. (2014) Coordinating garbage collection for arrays of solid-state drives. *IEEE Transactions on Computers*, **63**, 888–901.

# Chapter 10

# General-Purpose Computation on GPUs for Cloud Computing

The content of this chapter corresponds to the following journal paper:

**- Title:** General-purpose computation on GPUs for high performance cloud computing

**- Authors:** Roberto R. Expósito, Guillermo L. Taboada, Sabela Ramos, Juan Touriño, Ramón Doallo

**- Journal:** Concurrency and Computation: Practice and Experience

**- Editorial:** John Wiley & Sons, Ltd.

**- ISSN:** 1532-0634

**- Year:** 2013

**- Volume(Number):Pages:** 25(12):1628–1642

**- DOI:** 10.1002/cpe.2845

The final publication is available at http://onlinelibrary.wiley.com/doi/10.1002/cpe.2845/abstract. A copy of the accepted paper has been included next.

# General-purpose computation on GPUs for high performance cloud computing

Roberto R. Expósito[*,†], Guillermo L. Taboada, Sabela Ramos,
Juan Touriño and Ramón Doallo

*Computer Architecture Group, Department of Electronics and Systems, University of A Coruña, A Coruña, Spain*

## SUMMARY

Cloud computing is offering new approaches for High Performance Computing (HPC) as it provides dynamically scalable resources as a service over the Internet. In addition, General-Purpose computation on Graphical Processing Units (GPGPU) has gained much attention from scientific computing in multiple domains, thus becoming an important programming model in HPC. Compute Unified Device Architecture (CUDA) has been established as a popular programming model for GPGPUs, removing the need for using the graphics APIs for computing applications. Open Computing Language (OpenCL) is an emerging alternative not only for GPGPU but also for any parallel architecture. GPU clusters, usually programmed with a hybrid parallel paradigm mixing Message Passing Interface (MPI) with CUDA/OpenCL, are currently gaining high popularity. Therefore, cloud providers are deploying clusters with multiple GPUs per node and high-speed network interconnects in order to make them a feasible option for HPC as a Service (HPCaaS). This paper evaluates GPGPU for high performance cloud computing on a public cloud computing infrastructure, Amazon EC2 Cluster GPU Instances (CGI), equipped with NVIDIA Tesla GPUs and a 10 Gigabit Ethernet network. The analysis of the results, obtained using up to 64 GPUs and 256-processor cores, has shown that GPGPU is a viable option for high performance cloud computing despite the significant impact that virtualized environments still have on network overhead, which still hampers the adoption of GPGPU communication-intensive applications.

KEY WORDS: Cloud Computing; General-Purporse computation on GPU (GPGPU); High Performance Computing (HPC); 10 Gigabit Ethernet; CUDA; OpenCL; MPI

## 1. INTRODUCTION

Cloud computing [1] is an Internet-based computing model that enables convenient, on-demand network access to a shared pool of configurable and virtualized computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort. Public clouds offer access to external users who are typically billed on a pay-as-you-use basis. With the cloud, and the availability of multiple cloud service providers, organizations are no longer forced to invest in additional technology infrastructure. They can just leverage the infrastructure provided by the cloud service provider or move their own applications to this infrastructure. Customers can derive significant economies of use by leveraging the pay-by-use model, instead of upgrading their infrastructure, dimensioned to handle peak requests.

*Correspondence to: Roberto R. Expósito, Department of Electronics and Systems, University of A Coruña, Campus de Elviña s/n, 15071, A Coruña, Spain.

†E-mail: rreye@udc.es

Cloud computing has been driven from the start predominantly by the industry through Amazon, Google, and Microsoft, but because of its potential benefits, this model has also been adopted by academia, as it is well suited for handling peak demands in resource-intensive applications in science and engineering. Thus, cloud computing is an option for High Performance Computing (HPC), where the use of cloud infrastructures for HPC applications has generated considerable interest in the scientific community [2–6], which has coined the term HPC as a Service (HPCaaS), an extension of the provision of Infrastructure as a Service (IaaS).

HPC workloads typically require low latency and high bandwidth interprocessor communication to provide scalable performance. However, the widely extended use of commodity interconnect technologies (Ethernet and TCP/IP) and the overhead of the virtualized access to the network limit severely the scalability of HPC applications in public cloud infrastructures. To overcome these constraints, cloud infrastructure providers are increasingly deploying high-speed networks (e.g., 10 Gigabit Ethernet and InfiniBand), widely extended in HPC environments, where message-passing middleware is the preferred choice for communications. MPI [7] is the de facto standard in message-passing interface as it generally provides HPC applications with high scalability on clusters with high-speed networks.

The advent of many-core accelerators, such as Graphical Processing Units (GPUs), to HPC is already consolidated thanks to the outbreak of General-Purpose computing on GPUs (GPGPU) [8, 9]. The massively parallel GPU architecture, together with its high floating point performance and memory bandwidth, is well suited for many workloads in science and engineering, even outperforming multicore processors, which has motivated the incorporation of GPUs as HPC accelerators [10]. As a result, new parallel programming models such as Compute Unified Device Architecture (CUDA) [11] and Open Computing Language (OpenCL) [12] have emerged to expose the parallel capabilities of GPUs to GPGPU programmers in a productive way [13]. These models can be combined with well-established HPC programming models such as MPI [14].

Amazon Elastic Compute Cloud (Amazon EC2) [15] is an IaaS that provides users with access to on-demand computational resources to run their applications. Amazon EC2 supports the management of the resources through a Web service or an API that is able, among other tasks, to boot straightforwardly an Amazon Machine Image (AMI) into a custom virtual machine (a VM or 'instance'). Amazon EC2 Cluster Compute Instances (CCIs) [16], a resource available since July 2010, provide a significant CPU power (two quad-core processors), together with a high performance 10 Gigabit Ethernet network, thus targeting HPC applications. In November 2010, Amazon EC2 introduced Cluster GPU Instances (CGIs), which have the same configuration as CCI plus two NVIDIA Tesla GPUs. HPC applications running on CGI are expected to benefit significantly from the massive parallel processing power the GPUs provide, as well as from their full-bisection high bandwidth network (10 Gigabit Ethernet). This is particularly valuable for applications that rely on messaging middleware such as MPI for communications while taking advantage of GPGPU, such as hybrid MPI/CUDA or MPI/OpenCL codes. This paper evaluates GPGPU for high performance cloud computing on Amazon EC2 cloud infrastructure, the largest public cloud in production, by using up to 32 CGIs, demonstrating the viability of providing HPC as a service taking advantage of GPGPU in a cluster of CGIs.

The structure of this paper is as follows. Section 2 presents the related work. Section 3 introduces the experimental configuration, both hardware and software, and the methodology of the evaluation conducted in this work. Section 4 analyzes the performance results obtained by representative benchmarks on the experimental test bed on Amazon EC2 public cloud. Section 5 summarizes our concluding remarks.

## 2. RELATED WORK

There has been a spur of research activity in assessing the performance of virtualized resources in cloud computing environments [17–19]. The suitability of these resources for HPC increases as their performance improves, which has motivated lately several projects on the adoption of cloud computing for HPC applications. Among them, a popular topic is the feasibility of Amazon EC2 for HPC applications, which has been discussed in several papers since 2008, next presented.

Deelman *et al.* [20] explored the application of cloud computing for science, examining the trade-offs of different workflow execution modes and provisioning plans for cloud resources. In [2], Evangelinos and Hill analyzed the performance of HPC benchmarks on EC2 cloud infrastructure. These authors revealed an important drawback in network performance, as messaging middleware obtains latencies and bandwidths around one and two orders of magnitude inferior to supercomputers. In [3], Walker evaluated the performance of Amazon EC2 High-CPU instances for high performance scientific applications, reporting significantly lower performance than traditional HPC clusters. His work also presented the comparative performance evaluation between Amazon EC2 High-CPU instances and an InfiniBand cluster with similar hardware configuration, reporting 40–1000% runtime increase on EC2 resources.

Buyya *et al.* [4] discussed the potential opportunities of high performance scientific applications on public clouds through some practical examples on Amazon EC2, assessing that the trade-offs between cost and performance have to be considered. Ekanayake and Fox [21] compared applications with different communication and computation complexities and observed that latency-sensitive applications experience higher performance degradation than bandwidth-sensitive applications. Ostermann *et al.* [22] presented an evaluation of the feasibility of Amazon EC2 cloud computing services for scientific computing. They analyzed the performance of the Amazon EC2 standard and High-CPU instances using representative microbenchmarks and kernels. The main conclusion of their work is that Amazon EC2 required, in 2009, an order of magnitude higher performance in their instances to be feasible for use by the scientific community. Napper and Bientinesi [5] examined the performance of the Linpack benchmark on several EC2 instance types (standard and High-CPU instances). They concluded that clouds cannot compete with conventional HPC clusters (supercomputers and high-speed clusters) on the basis of the metric GFLOPS/$, because memory and network performance is insufficient to compete with existing scalable HPC systems.

Jackson *et al.* [6] performed a comprehensive evaluation, comparing conventional HPC platforms with Amazon EC2 standard instances, by using real applications representative of the workload at a typical supercomputing center. Their main conclusion is that the interconnection network of the evaluated instances (Gigabit Ethernet at that time) severely limits performance and causes significant runtime variability. Wang and Eugene [23] studied the impact of virtualization on network performance. They presented a quantitative study of the network performance among Amazon EC2 High-CPU instances, detecting unstable TCP/UDP throughput caused by virtualization and processor sharing. Rehr *et al.* [24] confirmed that Amazon EC2, using standard and High-CPU instance types, is a feasible platform for applications that do not demand high performance network. He *et al.* [25] extended previous research to three public clouds (including Amazon) by running a real application in addition to classical benchmarks to compare the cloud results with those from dedicated HPC systems. Their main conclusion is that public clouds are not designed for running scientific applications primarily because of their poor network performance. Iosup *et al.* [26] investigated the presence of a Many Task Computing (MTC) component in existing scientific computing workloads, and then they performed an empirical performance evaluation of this MTC component on four public computing clouds, including Amazon EC2 using standard and High-CPU instance types. Their main conclusion is that the computational performance of the evaluated clouds is low, which is insufficient for scientific computing at large, but cloud computing can be a good solution for instant and temporal resource needs.

The main drawback detected by most of these previous works since 2005 up to 2010, is the high network overhead due to the use of commodity interconnection technologies (e.g., Ethernet and TCP/IP) that are not suitable for HPC. In order to overcome this drawback, Amazon EC2 introduced 10 Gigabit Ethernet as interconnection technology together with the launch of its CCI and CGI instance types, in June and November 2010, respectively. These instances target HPC applications thanks to its high computational power and the adoption of a high performance network. The CCI instance type has been evaluated in recent related work. Thus, Regola and Ducom [27] evaluated the suitability of several virtualization technologies for HPC, showing that Operating System (OS) virtualization was the only solution that offers near native CPU and Input/Output (I/O) performance. They included in their test bed four Amazon EC2 CCIs, although they focused more on the overall performance of the several evaluated hypervisors instead of the network performance and the scalability of HPC applications. Carlyle *et al.* [28] reported that it is much more effective in academia operating a community cluster program than providing HPC resources with nodes using Amazon EC2 CCIs. Sun *et al.* [29] relied on Amazon EC2 CCIs for running the Lattice optimization and some additional performance benchmarks, concluding that these instances suffer from performance degradation for large-scale parallel MPI applications, especially network-bound or memory-bound applications. Ramakrishnan *et al.* [30] analyzed the performance of a number of different interconnect technologies, including 10 Gigabit Ethernet network from Amazon EC2 CCIs, showing that virtualization can have a significant impact on performance. Zhai *et al.* [31] conducted a comprehensive evaluation of MPI applications on Amazon EC2 CCIs, revealing a significant performance increase compared with previous evaluations on standard and High-CPU instances. However, the interconnection network overhead, especially the high startup latency (poor small message performance), remains as the main MPI scalability limitation.

Additionally, some works have evaluated the performance of other public cloud computing services, such as a seminal study at Amazon S3 by Palankar *et al.* [32], which also includes an evaluation of file transfer between Amazon EC2 and S3.

This exhaustive review of the related works on evaluating Amazon for HPC applications has revealed the lack, to the best of our knowledge in December 2011, of assessments of Amazon EC2 CGIs performance, as well as analyses on the viability of providing HPC as a service in a public cloud taking advantage of message-passing and GPGPU.

## 3. EXPERIMENTAL CONFIGURATION AND EVALUATION METHODOLOGY

The evaluation of GPGPU for high performance cloud computing has been carried out on a public cloud computing infrastructure, Amazon EC2, using the Cluster GPU Instances (CGIs), which target HPC applications. This section presents the description of the CGI-based platform used in the evaluation (Section 3.1), an overview of the virtualization technologies available in Amazon EC2 (Section 3.2), the description of the GPGPU codes used in the evaluation (Section 3.3), and finally, this section concludes with the methodology followed in this evaluation (Section 3.4).

### *3.1. Amazon EC2 CGI platform*

The evaluation of GPGPU on Amazon EC2 has been carried out on a cluster of 32 CGIs, whose main characteristics are presented in Tables I (CGI) and II (cluster of CGIs). A CGI node has eight cores, each of them capable of executing four floating point operations per clock cycle in double precision (DP), hence 46.88 Giga Floating Point Operations per Second (GFLOPS) per processor, 93.76 GFLOPS per node, and 3000 GFLOPS in the 32-node (256-core) cluster. Moreover, each GPU comes with 3 GB GDDR5 of memory and has a peak performance of 515 GFLOPS in DP,

Table I. Description of the Amazon EC2 Cluster GPU Instance.

| | |
|---|---|
| CPU | 2 × Intel(R) Xeon quad-core X5570 @2.93 GHz (46.88 GFLOPS DP each CPU) |
| EC2 compute units | 33.5 |
| GPU | 2 × NVIDIA Tesla ‘Fermi’ M2050 (515 GFLOPS DP each GPU) |
| Memory | 22 GB DDR3 |
| Storage | 1690 GB |
| Virtualization | Xen HVM 64-bit platform (PV drivers for I/O) |
| API name | cg1.4xlarge |

Table II. Characteristics of the CGI-based cluster.

| | |
|---|---|
| Number of CGI nodes | 32 |
| Interconnection network | 10 Gigabit Ethernet |
| Total EC2 compute units | 1072 |
| Total CPU cores | 256 (3 TFLOPS DP) |
| Total GPUs | 64 (32.96 TFLOPS DP) |
| Total FLOPS | 35.96 TFLOPS DP |

hence 1030 GFLOPS per node and 32,960 GFLOPS in the 32-node (64-GPU) cluster. Aggregating CPU and GPU theoretical peak performances, each node provides 1124 GFLOPS; hence, the entire cluster provides 35,960 GFLOPS.

The instances of this GPU cluster have been allocated simultaneously in a single placement group in order to obtain nearby instances, with full-bisection 10 Gbps bandwidth connectivity among them. The interconnection technology, 10 Gigabit Ethernet, is a differential characteristic of the Amazon EC2 cluster instances. Additionally, Amazon EC2 guarantees that the hardware infrastructure of the cluster instances, both CGI and CCI, is not shared with any other Amazon EC2 instances and at any given time each node runs a single VM.

The CentOS 5.5 GPU Hardware Virtual Machine (HVM) AMI (ami-aa30c7c3) is provided by Amazon Web Services (AWS) for preparing the software configuration of EC2 GPU-based clusters. However, this AMI comes with CUDA toolkit version 3.1, so it is required to be upgraded to CUDA version 4.0, mainly because it comes with some extra features such as the ability to share GPUs across multiple threads and the control from a single host thread of all GPUs in the system concurrently. The GNU C/Fortran 4.1.2 compiler was used with −O3 flag to compile all the benchmarks and applications, except NAMD, which is distributed in binary form. The Intel compiler version 12.1 and the Intel Math Kernel Library (MKL) 10.3 have been used for building the High Performance Linpack (HPL) benchmark. The message-passing library used for NAMD, MC-GPU, and HPL is OpenMPI [33], version 1.4.4.

### *3.2. Amazon EC2 virtualization technologies overview*

The Virtualization Machine Monitor (VMM), also called hypervisor, used by all Amazon EC2 instances is Xen [34], a high performance VMM quite popular among cloud providers. Xen systems have the hypervisor at the lowest and most privileged layer. The hypervisor schedules one or more guest OSs across the physical CPUs. The first guest OS, called in Xen terminology domain 0 (dom0), boots automatically when the hypervisor boots and receives special management privileges and direct access to all physical hardware by default. The system administrator can log into dom0 in order to manage any further guest OSs, known as domain U (domU) in Xen terminology.

Xen supports two virtualization technologies, full virtualization (HVM) and paravirtualization (PV). On one hand, HVM allows the virtualization of proprietary OSs, because the guest system's kernel does not require modification, but guests require CPU virtualization extensions from the host CPU (Intel VT [35] and AMD-V [36]). In order to boost performance, fully virtualized HVM guests can use special paravirtual device drivers to bypass the emulation for disk and network I/O. On the other hand, PV requires changes to the virtualized OS to be hypervisor aware. This allows the VM to coordinate with the hypervisor, reducing the use of privileged instructions that are typically responsible for the major performance penalties in full virtualization. This technology does not require virtualization extensions from the host CPU.

Amazon EC2 CCI and CGI use Xen HVM virtualization technology with paravirtual drivers for improving network performance, whereas the rest of Amazon EC2 instance types are Xen PV guests. Therefore, the access to the Network Interface Card (NIC) in Amazon EC2 instances is paravirtualized. However, a direct access to the underlying NIC (and other PCI cards) is also possible in virtualized environments using PCI passthrough [37]. This technique consists of isolating a device in order to be used exclusively by a guest OS, which eventually achieves near-native performance from the device. Xen supports PCI passthrough [38] for PV and HVM guests, but dom0 OS must support it, typically available as a kernel build-time option. Additionally, the latest processor architectures by Intel and AMD also support PCI passthrough (in addition to new instructions that assist the hypervisor). Intel calls its approach Virtualization Technology for Directed I/O (VT-d) [39], whereas AMD refers to it as I/O Memory Management Unit (IOMMU) [40]. In both cases, the CPU is able to map PCI physical addresses to guest virtual addresses and take care of access (and protection) to the mapped device, so that the guest OS can use and to take advantage of the device like in a nonvirtualized system. In addition to this mapping of virtual guest addresses to physical memory, isolation is provided in such a way that other guests (or even the hypervisor) are precluded from accessing it.

Amazon EC2 CGI relies on Xen PCI passthrough for accessing the GPUs using Intel VT-d technology, so domU OS and applications do direct I/O with the GPUs. Unfortunately, as we have mentioned previously, the NIC is not available via PCI passthrough, so the access to the network is virtualized in Amazon EC2 instances. This lack of efficient virtualized network support in Amazon EC2 explains why previous works using CCIs on Amazon EC2 concluded that the communications performance remains as the main issue for the scalability of MPI applications in the cloud.

### 3.3. *GPGPU kernels and applications*

The evaluation of GPGPU on Amazon EC2 has been carried out using representative GPGPU benchmarks and applications, several synthetic kernels, two real-world applications, and the widely used HPL implementation [41] of the Linpack [42] benchmark.

*3.3.1. Synthetic kernels.* They are code snippets that implement operations that can either take full advantage of the hardware (e.g., a bus speed characterization code) or provide with widely extended basic building blocks in HPC applications (e.g., a matrix multiplication kernel). The synthetic kernels used for the evaluation of GPGPU on Amazon EC2 have been selected from two representative benchmark suites, Scalable HeterOgeneus Computing (SHOC) [43] and Rodinia benchmark suite [44]. On one hand, the SHOC suite assesses low-level architectural features through microbenchmarking, as well as determines the computational performance of the system with the aid of application kernels. Table III presents the 10 SHOC synthetic kernels selected. Furthermore, these kernels have OpenCL and CUDA implementations, which allows the comparative analysis of their performance. On the other hand, the Rodinia suite targets the performance analysis of heterogeneous systems, providing application kernels implemented with OpenMP, OpenCL, and CUDA for both GPUs and multicore CPUs. Table III also includes two synthetic kernels from the Rodinia suite.

*3.3.2. Applications in science and engineering.* Two distributed real-world applications that support the use of multiple GPUs per node in CUDA, NAMD [45, 46] and MC-GPU [47, 48], have been selected. Both applications can be executed either by using only CPUs or by mixing CPUs and GPUs. On one hand, NAMD is a parallel molecular dynamics code, based on Charm++ parallel objects, designed for high performance simulation of large biomolecular systems. The NAMD suite includes ApoA1, one of the most used data sets for benchmarking NAMD, which models a bloodstream lipoprotein particle with 92K atoms of lipid, protein, and water on 500 steps, with 12A cutoff. NAMD is a communication-intensive iterative application. On the other hand, MC-GPU is an

Table III. Selected synthetic kernels.

| Kernel | Suite | Performance unit | Description |
|---|---|---|---|
| BusSpeedDownload | SHOC | GB/s | PCIe bus bandwidth (host to device) |
| BusSpeedReadback | SHOC | GB/s | PCIe bus bandwidth (device to host) |
| MaxFlops | SHOC | GFLOPS | Peak floating point operations per second |
| Device Memory | SHOC | GB/s | Device memory bandwidth |
| SPMV | SHOC | GFLOPS | Multiplication of sparse matrix and vector |
| GEMM | SHOC | GFLOPS | Matrix multiplication |
| FFT | SHOC | GFLOPS | Fast Fourier transform |
| MD | SHOC | GFLOPS | Molecular dynamics |
| Stencil2D | SHOC | Time(s) | A two-dimensional nine point stencil calculation |
| S3D | SHOC | GFLOPS | Computes the rate of various chemical reactions across a regular three-dimensional grid. |
| CFD | Rodinia | Time(s) | Grid finite volume solver for the three-dimensional Euler equations for compressible flow |
| Hotspot | Rodinia | Time(s) | Estimate processor temperature on the basis of an architectural floor plan and simulated power measurements |

X-ray transport simulation code that can generate clinically realistic radiographic projection images and computed tomography scans of the human anatomy. It uses an MPI library to address multiple nodes in parallel during the computed tomography simulations. It is a computation-intensive code with little communication.

*3.3.3. High Performance Linpack (HPL) benchmark.* The HPL benchmark [42] solves a random dense system of linear equations. The solution is computed by an LU decomposition with partial pivoting followed by backsubstitution. Dense linear algebra workloads are pervasive in scientific applications, especially in compute-intensive algorithms, so HPL provides a good upper bound on the expected performance of scientific workloads. In addition, TOP500 [49] list is based on HPL. The HPL implementation used in this work is the hybrid MPI/CUDA [50], not publicly available but provided to academia, research centers, and registered CUDA developers by NVIDIA.

### *3.4. Evaluation methodology*

The 15 codes selected for the GPGPU evaluation, the 12 synthetic kernels (10 from SHOC and 2 from Rodinia), the 2 applications (NAMD and MC-GPU), and the HPL have been initially executed both in a single CPU core and in a single GPU of an Amazon EC2 CGI VM. Additionally, the synthetic kernels have also been executed both in a single CPU core and in a single GPU of a non-virtualized node with the same CPU and GPU in order to assess the overhead of the virtualization on the CPU and GPU computational power. This physical node from our cluster has been denoted from now on as Computer Architecture Group (CAG) test bed.

Once the performance of a single core and GPU has been measured, the evaluated codes have been executed from 1 up to 32 CGI VMs, using all the available CPU cores (eight) and the available GPUs (two) per node. This is the most efficient configuration as the virtualized network imposes a significant overhead on communications performance, which motivates the minimization of the use of the network. Except special notice, all the codes use DP floating point arithmetic.

Furthermore, both the CPU and the GPU speedups have been computed taking as baseline the performance of a single CPU core. Thus, the performance of the GPUs is significantly higher than the performance of a single CPU core for the evaluated codes. For example, NAMD is able to achieve a speedup of 34 using the two GPUs of a VM, whereas the speedup achieved with the eight CPU cores is 8.

Finally, both the GNU and the Intel compilers have been considered for the CPU code used in this performance evaluation, showing little performance differences between them, never exceeding 4%. As the Intel compiler was generally the best performer, the reported results have been obtained from binaries compiled with this compiler.

## 4. ASSESSMENT OF GPGPU FOR HIGH PERFORMANCE CLOUD COMPUTING

This section assesses the performance of GPGPU for high performance cloud computing on a public cloud infrastructure, Amazon EC2, using the selected benchmarks/applications presented in the previous section. Two features of the CGI VMs used are key for the analysis of the obtained performance results, the high penalty of the virtualized access to the high-speed 10 Gigabit Ethernet network and the low overhead of the direct access to the GPUs through Xen PCI passthrough using Intel VT-d hardware.

### *4.1. Synthetic kernels' performance with CUDA and OpenCL*

Figures 1 and 2 present the performance results obtained by the selected synthetic kernels listed in Table III using their CUDA and OpenCL implementations on a single NVIDIA Tesla 'Fermi' M2050, both on Amazon EC2 and on CAG test beds.

Regarding Figure 1, analyzing the results from left to right and from top to bottom, the PCIe bus bandwidth benchmark shows similar results for CUDA and OpenCL, both from host to device as well as from device to host. Moreover, the results reported on Amazon EC2 are similar to those obtained on CAG.

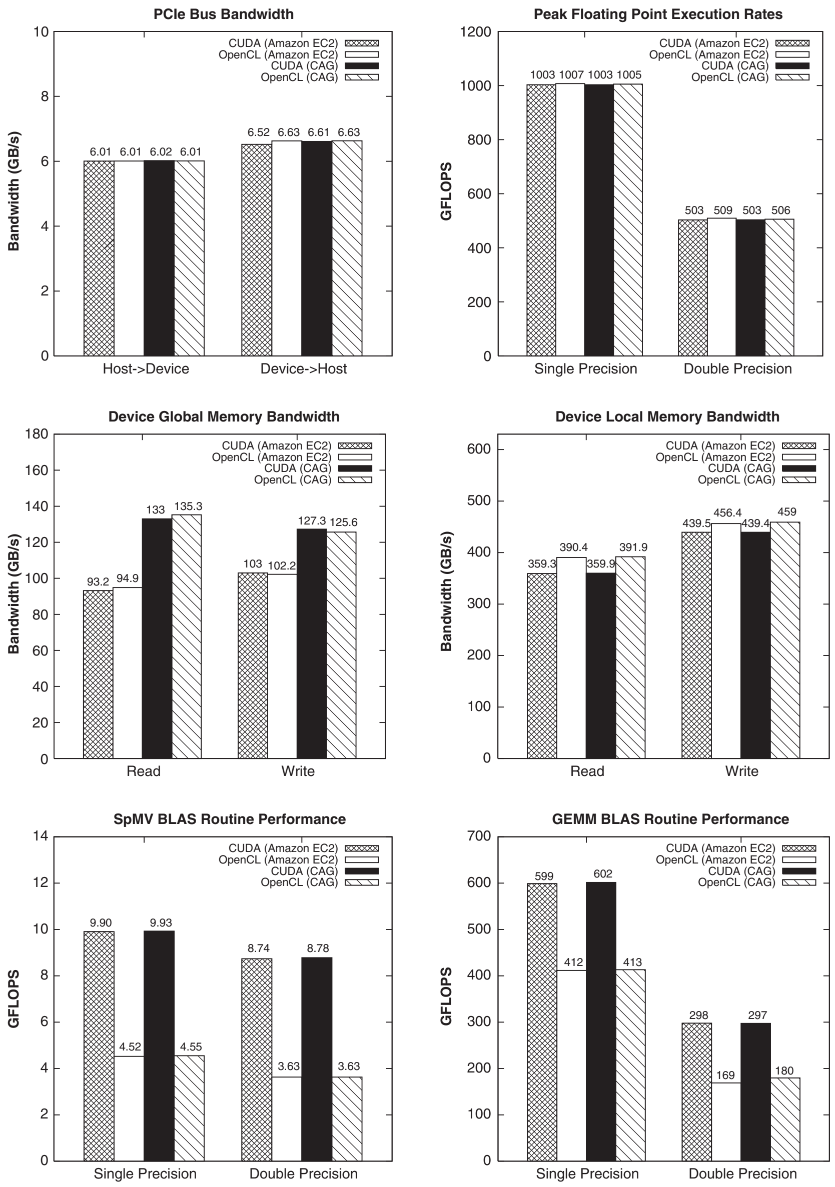


Figure 1. SHOC synthetic benchmarks' performance on Amazon EC2 and CAG test beds.

The peak floating point benchmark presents a similar behavior, with insignificant differences between Amazon and CAG for both single precision (SP) and DP tests. Here, CUDA and OpenCL also achieve a similar peak performance (to be more precise, OpenCL outperforms CUDA slightly), which supports the conclusion that OpenCL has the same potential as CUDA to take full advantage of the underlying hardware. In fact, the observed performance results are very close to the theoretical peak performance on its GPU (NVIDIA Tesla C2050), 1030 GFLOPS for SP and 515 GFLOPS for DP.

Next two graphs in Figure 1 present the device memory bandwidth tests, for both read and write operations. On one hand, the left graph shows the device global memory bandwidth, measured by accessing global memory in a coalesced manner, where CUDA and OpenCL obtain similar results.

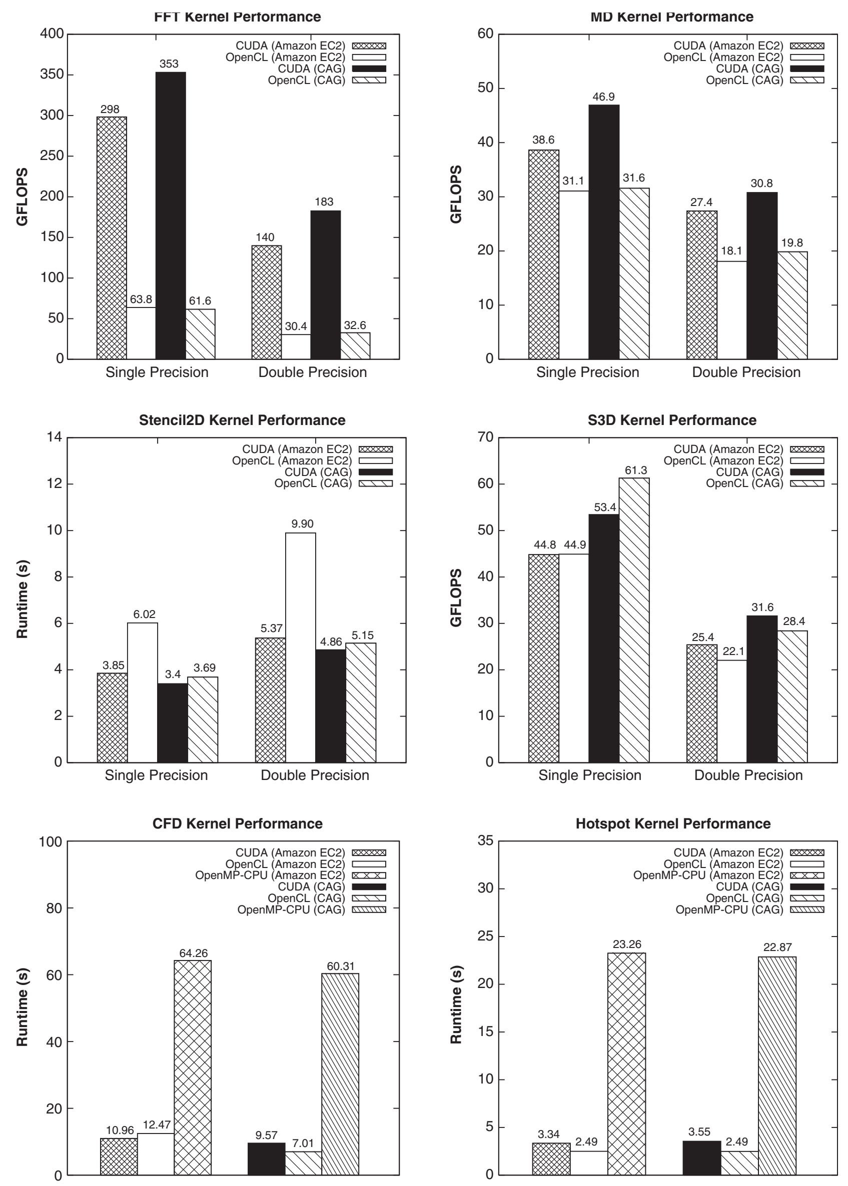


Figure 2. SHOC and Rodinia kernels' performance on Amazon EC2 and CAG test beds.

However, for these tests the difference between Amazon EC2 and CAG performance is significant, especially for read operations. The main reason for this performance gap is the Error-Correcting Code (ECC) memory error protection which is enabled in Amazon EC2, whereas it is disabled in CAG. With ECC memory error protection activated, a portion of the GPU memory is used for ECC bits, so the available user memory is reduced by 12.5% (3-GB total memory yields 2.625 GB of user available memory). The overhead associated with the handling of these ECC bits represents an important performance penalty. On the other hand, the right graph shows the device local memory bandwidth, where OpenCL outperforms CUDA, especially for read operations with up to 8% more bandwidth. In this case, there are no significant performance differences between Amazon EC2 and CAG test beds.

The last two graphs at the bottom in Figure 1 present the results for two widely used Basic Linear Algebra Subprograms (BLAS) subroutines: the level 2 operation Sparse Matrix–Vector Multiplication (SpMV) and the level 3 operation General Matrix Multiplication (GEMM), for both SP and DP. Regarding the test bed, these routines obtain practically the same performance results in Amazon and CAG. However, CUDA significantly outperforms OpenCL, even doubling its performance in some cases (SpMV in SP and DP).

Regarding Figure 2, analyzing the results from left to right and from top to bottom, CUDA outperforms significantly OpenCL for the Fast Fourier Transform (FFT) and Molecular Dynamics (MD) kernels. This confirms that the OpenCL SHOC kernels are less optimized for this GPU architecture (Tesla 'Fermi') than the CUDA kernels. Moreover, these CUDA codes achieve the highest performance on CAG, mainly because of its superior memory performance thanks to having deactivated the ECC memory correction.

Next graph in Figure 2 presents the Stencil2D kernel, whose OpenCL implementation shows quite poor performance on Amazon EC2 but has a runtime slightly higher than CUDA on CAG. Next graph corresponds to S3D kernel, where OpenCL is the best performer in CAG with SP, whereas OpenCL and CUDA results are similar in Amazon EC2.

The last two graphs at the bottom in Figure 2 present the results for the two selected kernels from Rodinia suite, the Computational Fluid Dynamics (CFD) and the Hotspot kernels. These kernels have an OpenMP version, in addition to the OpenCL and CUDA versions, which has been executed using the eight CPU cores available in the Amazon EC2 VM and in the CAG node. The CFD results show that the GPUs are able to speed up around 5.2–8.6 times the CPU performance, thanks to the implementation in CFD of an algorithm that is suitable for its execution on a GPU. The GPU runtime of the Hotspot kernel also outperforms the CPU runtime, achieving around 6.4–9.3 times higher performance. Finally, there are no significant differences in the performance of the CUDA and OpenCL versions.

### *4.2. Distributed CUDA applications scalability*

Figure 3 presents the performance of the ApoA1 benchmark executed with NAMD 2.8 and up to 32 Amazon EC2 CGIs. The left graph shows the achieved simulation rate measured in days/nanosecond, whereas the right graph presents their corresponding speedups. This benchmark has been executed using the two implementations of NAMD, the only-CPU version and the CPU+GPU (CUDA) implementation. Regarding the results obtained, using a single VM, the CPU+GPU version outperforms clearly the only-CPU version, achieving around four times higher performance. However, the CPU+GPU version cannot take advantage of running on two or more VMs; in fact, it obtains lower performance, especially using eight or more VMs than using a single VM. This behavior contrasts with the only-CPU version, which is able to scale moderately using up to 16 VMs, almost reaching the performance results achieved by the CPU+GPU version.

This poor scalability of the NAMD ApoA1 Benchmark on Amazon EC2 is explained by the moderately communication-intensive nature of this benchmark, which suffers a significant

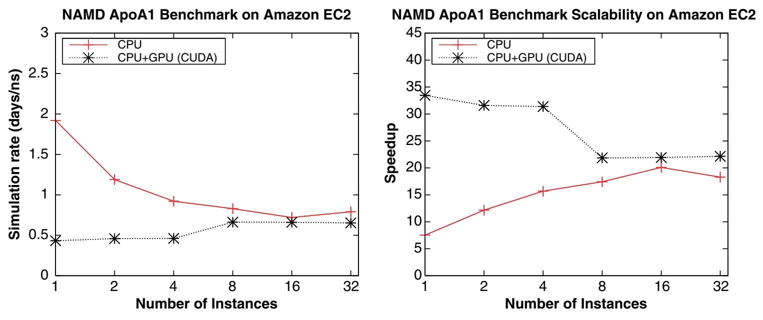


Figure 3. Performance of NAMD ApoA1 benchmark on Amazon EC2 CGIs.

performance penalty caused by the overhead of the virtualized access to the network. This performance bottleneck is especially important for the CPU+GPU implementation as it computes faster than the only-CPU version; therefore, its communication requirements are higher. In fact, NAMD developers recommend the use of low-latency interconnects (e.g., InfiniBand) for CUDA-accelerated NAMD executions across multiple nodes [46].

Figure 4 presents the performance of MC-GPU using up to 32 Amazon EC2 CGIs. The left graph shows the runtime measured in seconds, whereas the right graph depicts their corresponding speedups. This code is a massively MPI Monte Carlo simulation that has been executed with OpenMPI using only-CPU computation as well as CPU+GPU computation with CUDA. Unlike NAMD, this is a computation-intensive code with little communication, which eventually is able to achieve almost linear speedups using up to 32 instances. Thus, a speedup of almost 220 over 256 cores is achieved by the only-CPU version, whereas a speedup of around 1700, almost eight times the speedup of the only-CPU version, is achieved by the CPU+GPU version thanks to the use of 64 GPUs.

### *4.3. HPL benchmark performance*

Figure 5 depicts the performance achieved by the HPL benchmark using up to 32 Amazon EC2 CGIs. The left graph shows the GFLOPS obtained by the resolution of the dense system of linear equations, whereas the right graph presents their corresponding speedups. This MPI application has been executed with the OpenMPI implementation using only-CPU computation as well as using CPU+GPU computation with CUDA. The analysis of the breakdown of the runtime has revealed that most of the HPL runtime is spent in matrix–matrix multiplications in the update of trailing matrices. The bigger the problem size $N$ is, the more time is spent in this routine, so optimization of DGEMM is critical to achieve a high score. Thus, the performance of HPL depends on two main factors, basically not only on the GEMM subroutine performance in DP (DGEMM) but also on

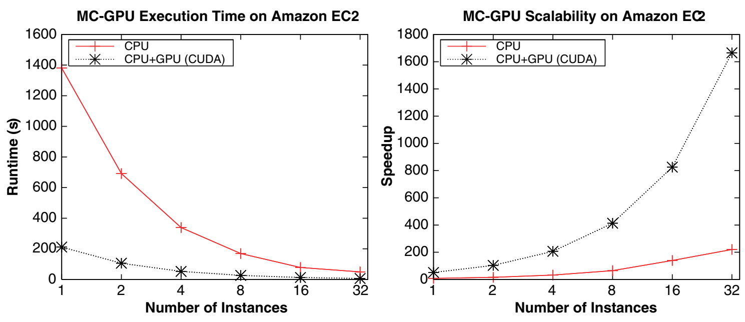


Figure 4. Performance of MC-GPU on Amazon EC2 CGIs.

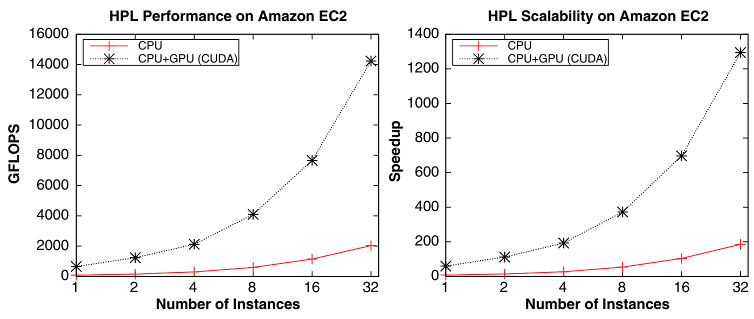


Figure 5. HPL (Linpack) performance on Amazon EC2 CGIs.

network performance, especially as the number of nodes increases. We ran many different configurations to find the best HPL settings (problem size, number of rows and columns, partitioning block size, panel factorization algorithm, and broadcast algorithm among others) and reported the peak for each number of instances used.

The measured HPL performance results show a moderate scalability limited severely by the poor virtualized network support in CGI VMs. Thus, the ratio between the achieved HPL performance and the theoretical peak performance decreases significantly as the number of instances increases, limiting the scalability of Amazon EC2 CGIs. Regarding the CPU+GPU (CUDA) implementation of HPL, with the use of one CGI VM, a 59% efficiency is obtained (655 GFLOPS out of 1124 peak GFLOPS are achieved), but when using 32 CGI VMs, the efficiency drops below 40% (14.23 TFLOPS out of 35.96 peak TFLOPS are achieved). Although 14.23 TFLOPS represents a speedup of 1290 with respect to the baseline HPL execution on a single CPU core, an efficiency of the CPU+GPU version below 40% with only 32 instances is quite poor.

The only-CPU version presents a similar scalability behavior although the efficiencies are higher. Thus, running HPL on one instance, the efficiency obtained is relatively 88% high (82.39 GFLOPS out of 93.76 peak GFLOPS are achieved), whereas the efficiency on 32 instances drops below 68% (2.035 TFLOPS out of 3 peak TFLOPS are achieved). The speedup obtained, 185 on 256 cores, represents a moderate scalability.

Figure 6 and Table IV show the achieved efficiency in terms of percentage of the theoretical peak performance in GFLOPS for each number of instances considered. To the best of our knowledge, this is the first time that HPL performance results on Amazon EC2 CGIs are reported, obtaining more than 14 TFLOPS. Amazon EC2 cluster instances have been reported to obtain 240 TFLOPS Rmax (354 TFLOPS Rpeak), ranked #42 in the last TOP500 list (November 2011), with 17,024 cores (1064 nodes, each with 16 cores), showing an efficiency of 67.80%, a performance that could only be obtained by running HPL on a nonvirtualized infrastructure. The previous appearance of Amazon EC2 on TOP500 list was on November 2010, with a cluster ranked #233, which obtained 41.8 TFLOPS Rmax (82.5 TFLOPS Rpeak) with 7040 cores (880 nodes, each comprising eight cores), reporting an efficiency of 51%, in tune with our measured results. As the last system (ranked

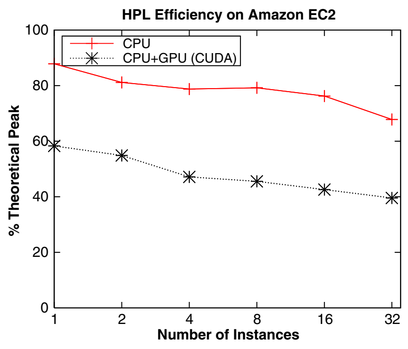


Figure 6. HPL (Linpack) efficiency as percentage of peak GFLOPS on Amazon EC2 CGIs.

Table IV. HPL (Linpack) efficiency on Amazon EC2 CGIs.

| # instances | # cores | # GPUs | CPU GFLOPS (Rpeak) | CPU GFLOPS (Rmax) | CPU+GPU TFLOPS (Rpeak) | CPU+GPU TFLOPS (Rmax) |
|---|---|---|---|---|---|---|
| 1 | 8 | 2 | 93.76 | 82.39 (87.87%) | 1.124 | 0.655 (59.27%) |
| 2 | 16 | 4 | 187.52 | 152.2 (81.16%) | 2.248 | 1.233 (54.85%) |
| 4 | 32 | 8 | 375.04 | 295.5 (78.71%) | 4.496 | 2.120 (47.15%) |
| 8 | 64 | 16 | 750.08 | 594.2 (79.21%) | 8.992 | 4.096 (45.55%) |
| 16 | 128 | 32 | 1500.16 | 1144 (76.26%) | 17.984 | 7.661 (42.60%) |
| 32 | 256 | 64 | 3000.32 | 2035 (67.83%) | 35.968 | 14.23 (39.66%) |

#500) in the current TOP500 list has 51 TFLOPS Rmax, it would be expected that a system with around 150 CGIs could have entered in the last TOP500 list. The cost of the access to such an infrastructure would be barely $300 per hour ($2.10 per CGI).

The performance evaluation presented in this section has shown that computationally intensive applications with algorithms that can be efficiently exploited in GPGPU can take full advantage of their execution in CGI VMs from Amazon EC2 cloud. In fact, the applications can benefit from the use of GPUs without any significant performance penalty, except when accessing memory intensively, where a small penalty can be observed as Amazon EC2 CGIs have ECC memory error protection enabled, which slightly limits the memory access bandwidth. Communication-intensive applications suffer from the overhead of the virtualized network access, which can reduce scalability significantly.

## 5. CONCLUSIONS

General-Purpose computation on GPU is attracting a considerable interest, especially in the scientific community, because of its massive parallel processing power. Another technology that is receiving an increasing attention is cloud computing, especially in enterprise environments, for which cloud services represent a flexible, reliable, powerful, convenient, and cost-effective alternative to owning and managing their own computing infrastructure. Regarding HPC area, cloud providers, such as Amazon public cloud, are already providing HPC resources, such as high-speed networks and GPUs.

This paper has evaluated GPGPU for high performance cloud computing on a public cloud computing infrastructure, using 32 Amazon EC2 Cluster GPU Instances, equipped with 10 Gigabit Ethernet and two NVIDIA Tesla GPUs each instance. The analysis of the performance results confirms that GPGPU is a feasible option in a public cloud because of its efficient access to the GPU accelerator in virtualized environments. However, our research has also detected that the virtualized network overhead limits severely the scalability of the applications, especially those sensitive to communication start-up latency. Therefore, more efficient communication middleware support is required to get over current cloud network limitations, with appropriate optimizations on communication libraries and OS virtualization layers. Thus, a direct access to the NIC through a device assignment to the VM is the key to reduce the network overhead in cloud environments and make HPC on demand as a widespread option for GPGPU.

ACKNOWLEDGEMENTS

This work was funded by the Ministry of Science and Innovation of Spain under Project TIN2010-16735 and an FPU Grant AP2010-4348 and by an Amazon Web Services (AWS) research grant.

REFERENCES

1. Buyya R, Yeo CS, Venugopal S, Broberg J, Brandic I. Cloud computing and emerging IT platforms: vision, hype, and reality for delivering computing as the 5th utility. *Future Generation Computer Systems* 2009; **25**(6):599–616.
2. Evangelinos C, Hill CN. Cloud Computing for parallel scientific HPC applications: feasibility of running coupled atmosphere-ocean climate models on Amazon's EC2. *Proceedings of 1st Workshop on Cloud Computing and its Applications (CCA'08)*, Chicago, IL, USA, 2008; 1–6.
3. Walker E. Benchmarking Amazon EC2 for high-performance scientific computing. *LOGIN: The USENIX Magazine* 2008; **33**(5):18–23.
4. Vecchiola C, Pandey S, Buyya R. High-performance cloud computing: a view of scientific applications. *Proceedings of 10th International Symposium on Pervasive Systems, Algorithms, and Networks (ISPAN'09)*, Kaoshiung, Taiwan, ROC, 2009; 4–16.
5. Napper J, Bientinesi P. Can cloud computing reach the TOP500? *Proceedings of Combined Workshops on UnConventional High Performance Computing Workshop Plus Memory Access Workshop (UCHPC-MAW'09)*, Ischia, Italy, 2009; 17–20.
6. Jackson KR, Ramakrishnan L, Muriki K, Canon S, Cholia S, Shalf J, Wasserman HJ, Wright NJ. Performance analysis of high performance computing applications on the Amazon Web services cloud. *Proceedings of 2nd IEEE International Conference on Cloud Computing Technology and Science (CloudCom'10)*, Indianapolis, USA, 2010; 159–168.

7. MPI: A Message Passing Interface Standard. (Available from: http://www.mcs.anl.gov/research/projects/mpi/ [Last visited: December 2011]).
8. Leist A, Playne DP, Hawick KA. Exploiting graphical processing units for data-parallel scientific applications. *Concurrency and Computation: Practice and Experience* 2009; **21**(18):2400–2437.
9. Pagès G, Wilbertz B. GPGPUs in computational finance: massive parallel computing for American style options. *Concurrency and Computation: Practice and Experience* 2011. DOI: 10.1002/cpe.1774. (In Press).
10. Fan Z, Qiu F, Kaufman A, Yoakum-Stover S. GPU cluster for high performance computing. *Proceedings of 16th ACM/IEEE Conference on Supercomputing (SC'04)*, Pittsburgh, PA, USA, 2004; 47 (12 pages).
11. Nickolls J, Buck I, Garland M, Skadron K. Scalable parallel programming with CUDA. *Queue* March 2008; **6**:40–53.
12. Stone JE, Gohara D, Guochun S. OpenCL: a parallel programming standard for heterogeneous computing systems. *Computing in Science and Engineering* 2010; **12**(3):66–73.
13. Malik M, Li T, Sharif U, Shahid R, El-Ghazawi T, Newby G. Productivity of GPUs under different programming paradigms. *Concurrency and Computation: Practice and Experience* 2011; **24**(2):179–191.
14. Karunadasa NP, Ranasinghe DN. Accelerating high performance applications with CUDA and MPI. *Proceedings of 4th International Conference on Industrial and Information Systems (ICIIS'09)*, Sri Lanka, India, 2009; 331–336.
15. Amazon Inc. Amazon Elastic Compute Cloud (Amazon EC2). (Available from: http://aws.amazon.com/ec2 [Last visited: December 2011]).
16. Amazon Inc. High performance computing using Amazon EC2. (Available from: http://aws.amazon.com/ec2/hpc-applications/ [Last visited: December 2011]).
17. Luszczek P, Meek E, Moore S, Terpstra D, Weaver VM, Dongarra JJ. Evaluation of the HPC challenge benchmarks in virtualized environments. *Proceedings of 6th Workshop on Virtualization in High-Performance Cloud Computing (VHPC'11)*, Bordeux, France, 2011; 1–10.
18. Younge AJ, Henschel R, Brown JT, von Laszewski G, Qiu J, Fox GC. Analysis of virtualization technologies for high performance computing environments. *Proceedings of 4th IEEE International Conference on Cloud Computing (CLOUD'11)*, Washington, DC, USA, 2011; 9–16.
19. Gavrilovska A, Kumar S, Raj H, Schwan K, Gupta V, Nathuji R, Niranjan R, Ranadive A, Saraiya P. High-Performance hypervisor architectures: virtualization in HPC systems. *Proceedings of 1st Workshop on System-level Virtualization for High Performance Computing (HPCVirt'07)*, Lisbon, Portugal, 2007; 1–8.
20. Deelman E, Singh G, Livny M, Berriman B, Good J. The cost of doing science on the cloud: the Montage example. *Proceedings of 20th ACM/IEEE Conference on Supercomputing (SC'08)*, Austin, Texas, 2008; 50 (12 pages).
21. Ekanayake J, Fox GC. High performance parallel computing with clouds and cloud technologies. *Proceedings of 1st International Conference on Cloud Computing (CloudComp'09)*, Munich, Germany, 2009; 20–38.
22. Ostermann S, Iosup A, Yigitbasi N, Prodan R, Fahringer T, Epema D. A performance analysis of EC2 cloud computing services for scientific computing. *Proceedings of 1st International Conference on Cloud Computing (CloudComp'09)*, Munich, Germany, 2009; 115–131.
23. Wang G, Eugene Ng TS. The impact of virtualization on network performance of Amazon EC2 data center. *Proceedings of 29th IEEE Conference on Computer Communications (INFOCOM'10)*, San Diego, CA, USA, 2010; 1163–1171.
24. Rehr JJ, Vila FD, Gardner JP, Svec L, Prange M. Scientific computing in the cloud. *Computing in Science and Engineering* 2010; **12**(3):34–43.
25. He Q, Zhou S, Kobler B, Duffy D, McGlynn T. Case study for running HPC applications in public clouds. *Proceedings of 19th ACM International Symposium on High Performance Distributed Computing (HPDC'10)*, Chicago, IL, USA, 2010; 395–401.
26. Iosup A, Ostermann S, Yigitbasi N, Prodan R, Fahringer T, Epema D. Performance analysis of cloud computing services for many-tasks scientific computing. *IEEE Transactions on Parallel and Distributed Systems* 2011; **22**:931–945.
27. Regola N, Ducom JC. Recommendations for virtualization technologies in high performance computing. *Proceedings of 2nd IEEE International Conference on Cloud Computing Technology and Science (CloudCom'10)*, Indianapolis, USA, 2010; 409–416.
28. Carlyle AG, Harrell SL, Smith PM. Cost-effective HPC: the community or the cloud? *Proceedings of 2nd IEEE International Conference on Cloud Computing Technology and Science (CloudCom'10)*, Indianapolis, USA, 2010; 169–176.
29. Sun C, Nishimura H, James S, Song K, Muriki K, Qin Y. HPC cloud applied to lattice optimization. *Proceedings of 2nd International Particle Accelerator Conference (IPAC'11)*, San Sebastian, Spain, 2011; 1767–1769.
30. Ramakrishnan L, Canon RS, Muriki K, Sakrejda I, Wright NJ. Evaluating interconnect and virtualization performance for high performance computing. *Proceedings of 2nd International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS'11)*, Seattle, WA, USA, 2011; 1–2.
31. Zhai Y, Liu M, Zhai J, Ma X, Chen W. Cloud versus in-house cluster: evaluating Amazon cluster compute instances for running MPI applications. *Proceedings of 23th ACM/IEEE Conference on Supercomputing (SC'11, State of the Practice Reports)*, Seattle, WA, USA, 2011; 11 (10 pages).
32. Palankar MR, Iamnitchi A, Ripeanu M, Garfinkel S. Amazon S3 for science grids: a viable solution? *Proceedings of 1st International Workshop on Data-aware Distributed Computing (DADC'08)*, Boston, MA, USA, 2008; 55–64.

33. Gabriel E, Fagg GE, Bosilca G, Angskun T, Dongarra JJ, Suyres JM, Sahay V, Kambadur P, Barrett B, Lumsdaine A. *et al.*. Open MPI: goals, concept, and design of a next generation MPI implementation. *Proceedings of 11th European PVM/MPI Users' Group Meeting (EuroPVM/MPI'04)*, Budapest, Hungary, 2004; 97–104.
34. Barham P, Dragovic B, Fraser K, Hand S, Harris T, Ho A, Neugebauer R, Pratt I, Warfield A. Xen and the art of virtualization. *Proceedings of 19th ACM Symposium on Operating Systems Principles (SOSP'03)*, Bolton Landing, NY, USA, 2003; 164–177.
35. Intel Virtualization Technology (Intel VT). (Available from: http://www.intel.com/technology/virtualization/technology.htm?iid=tech_vt+tech [Last visited: December 2011]).
36. AMD Virtualization Technology (AMD-V). (Available from: http://sites.amd.com/us/business/it-solutions/virtualization/Pages/amd-v.aspx [Last visited: December 2011]).
37. Jones T. Linux Virtualization and PCI Passthrough. (Available from: http://www.ibm.com/developerworks/linux/library/l-pci-passthrough/ [Last visited: December 2011]).
38. Xen PCI Passthrough. (Available from: http://wiki.xensource.com/xenwiki/XenPCIpassthrough [Last visited: December 2011]).
39. Abramson D, Jackson J, Muthrasanallur S, Neiger G, Regnier G, Sankaran R, Schoinas I, Uhlig R, Vembu B, Wiegert J. Intel Virtualization Technology for Directed I/O. *Intel Technology Journal* 2006; **10**(3):179–192.
40. AMD I/O Virtualization Technology (IOMMU) Specification. (Available from: http://support.amd.com/us/Processor_TechDocs/34434-IOMMU-Rev_1.26_2-11-09.pdf [Last visited: December 2011]).
41. Petitet A, Whaley RC, Dongarra JJ, Cleary A. HPL: a portable implementation of the high-performance Linpack benchmark for distributed-memory computers. (Available from: http://www.netlib.org/benchmark/hpl/ [Last visited: December 2011]).
42. Dongarra JJ, Luszczek P, Petitet A. The LINPACK benchmark: past, present and future. *Concurrency and Computation: Practice and Experience* 2003; **15**(9):803–820.
43. Danalis A, Marin G, McCurdy C, Meredith JS, Roth PC, Spafford K, Tipparaju V, Vetter JS. The Scalable Heterogeneous Computing (SHOC) benchmark suite. *Proceedings of 3rd Workshop on General-Purpose Computation on Graphics Processing Units (GPGPU-3)*, Pittsburgh, PA, USA, 2010; 63–74.
44. Che S, Boyer M, Meng J, Tarjan D, Sheaffer JW, Lee SH, Skadron K. Rodinia: a benchmark suite for heterogeneous computing. *Proceedings of IEEE International Symposium on Workload Characterization (IISWC'09)*, Austin, Texas, USA, 2009; 44–54.
45. Phillips JC, Braun R, Wang W, Gumbart J, Tajkhorshid E, Villa E, Chipot C, Skeel RD, Kalé L, Schulten K. Scalable molecular dynamics with NAMD. *Journal of Computational Chemistry* 2005; **26**(16):1781–1802.
46. NAMD: Scalable Molecular Dynamics. (Available from: http://www.ks.uiuc.edu/Research/namd/ [Last visited: December 2011]).
47. Badal A, Badano A. Accelerating Monte Carlo simulations of photon transport in a voxelized geometry using a massively parallel graphics processing unit. *Medical Physics* 2009; **36**(11):4878–4880.
48. MC-GPU: Monte Carlo simulation of x-ray transport in a GPU with CUDA. (Available from: http://code.google.com/p/mcgpu/ [Last visited: December 2011]).
49. TOP500 Org. Top 500 Supercomputer Sites. (Available from: http://www.top500.org/ [Last visited: December 2011]).
50. Fatica M. Accelerating Linpack with CUDA on heterogenous clusters. *Proceedings of 2nd Workshop on General Purpose Computation on Graphics Processing Units (GPGPU-2)*, Washington, DC, USA, 2009; 46–51.

# Conclusions and Future Work

Next, we summarize the main contributions of the work done in the Thesis and provide some insights on future research directions.

## Conclusions

This PhD Thesis, *"Design and Evaluation of Low-Latency Communication Middleware on High Performance Computing Systems"*, has been conducted to accomplish successfully a twofold purpose. On the one hand, it has presented the design, implementation and optimization of scalable Java communication solutions for HPC on top of high-speed networks. Hence, one of the main outcomes of the Thesis is the development and evaluation of Java message-passing communication middleware for parallel computing, named FastMPJ, which allows to increase significantly the performance of parallel Java applications on current clusters and supercomputers, the most widespread HPC deployments. FastMPJ implements the mpiJava 1.2 API, the most widely extended MPI-like Java bindings, for a highly productive development of parallel MPJ applications. This middleware integrates the collection of the low-level Java communication devices developed in this Thesis, thus enabling low-latency (1 $\mu$s) and high-bandwidth (up to 49 Gbps) point-to-point communciations for a more efficient high-speed network support in Java. Therefore, a key contribution of this Thesis is the design and implementation of the following communication devices: (1) `ibvdev`, for InfiniBand, RoCE and iWARP adapters in general terms; (2) `mxdev`, for Myrinet and both high-speed and generic Ethernet hardware; (3) `psmdev`, for Intel/QLogic-based InfiniBand adapters; (4) `mxmdev`, for Mellanox-based InfiniBand/RoCE adapters; (5) `ugnidev`, for Gemini/Aries adapters, used by current Cray supercomputers; and (6) `mpidev`, to be used on top of an MPI library. The integration of these devices into FastMPJ has allowed MPJ applications to take advantage of the use of a high number of cores (up to 4096) while exploiting efficiently the underlying RDMA hardware. In fact, the development of this middleware, which is even competitive with MPI point-to-point data transfers, is definitely bridging the gap between Java and natively compiled languages in HPC applications. Furthermore, the Thesis has also provided an up-to-date review of Java for HPC, which includes a thorough evaluation of the performance of current projects. As main conclusion, the reported advances in the efficiency of Java communications have shown that the use of Java in HPC is feasible, as it is able to achieve high-performance results. All these efforts have served to increase definitely the benefits of the adoption of Java for HPC, in order to achieve higher parallel programming productivity.

On the other hand, this Thesis has analyzed the feasibility of using a public cloud infrastructure for HPC and scientific computing. Hence, it has evaluated several HPC-aimed cloud resources offered by Amazon EC2, the leading commercial IaaS cloud provider, which specifically target HPC environments. Among these resources, cluster instances that provide powerful multi-core CPUs and are interconnected via a high-speed network (10 Gigabit Ethernet) have been assessed. Therefore, another key contribution of this Thesis is an extensive study of Amazon EC2 resources for HPC, which ranges from the identification of the main causes of communication inefficiency on this cloud computing environment up to the proposal and evaluation of techniques for reducing their impact on the scalability of communication-intensive HPC codes, both for Java and natively compiled languages. Furthermore, this study includes not only a comparative evaluation of our Java communication middleware in the cloud, but also considers other important aspects of Amazon EC2 to be a viable alternative in the HPC area. Therefore, further assessments that take into account the performance of the I/O storage subsystem, the characterization of parallel/distributed file systems for data-intensive computing (e.g., Big Data workloads) and the feasibility of using heterogeneous architectures with many-core GPU accelerators have been carried out. Through a performance and cost analysis of HPC applications and comparison on private virtualized and non-virtualized testbeds, we have shown that different applications exhibit different characteristics that make them more or less suitable to be run on a cloud environment. As main conclusion, this Thesis has shown the significant impact that virtualized environments still have on communications performance, which hampers the adoption of communication-intensive parallel applications in the cloud. Although it is possible to increase significantly their scalability following the guidelines for performance optimization suggested in this Thesis, such as reducing the number of processes per instance or using the combination of message passing with multithreading, the network remains as the major performance penalty, especially for GPGPU communication-intensive codes and I/O-intensive parallel applications. This fact demands appropriate optimizations on the virtualization layer, such as the direct access to the network hardware, to get over current cloud network limitations and enable efficient HPC in the cloud. Therefore, the release of enhanced cloud network resources that overcome these constraints is the way to go as current virtualized resources showed high scalability when the communications throughput was not the main performance bottleneck.

## Future Work

In terms of future research work, the development of a new FastMPJ communication device optimized for shared/distributed memory environments would be of great interest. Thus, in this hybrid device, intra-node communications would be performed through some shared memory operation, either an inter-process (e.g., mmap, SysV) or intra-process (e.g., Java threads) mechanism, whereas inter-node transfers would take advantage of the efficient communication devices presented in this Thesis. Regarding this point, FastMPJ devices could be further extended by including their own implementation of collective operations at the `xxdev` layer, even without relying on point-to-point primitives. The use of these algorithms would allow to perform only one call to the device per collective, thus reducing the calling overhead. Furthemore, the features of the underlying network could be exploited more efficiently by using hardware-based collective acceleration, which would offload the collective operations onto the fabric switches and adapters. The extension of the mpiJava 1.2 API to conform with the MPI 3.0 specification and its corresponding reference implementation in FastMPJ would also be interesting.

Regarding HPC in the cloud, further assessments of the new HPC-aimed resources recently released by Amazon (e.g., C3 cluster instances, which according to Amazon support enhanced networking for higher packet per second performance and lower latencies) are required in order to determine if the network bottleneck has been reduced. Another major issue that discourages HPC users to move their applications to the cloud are security concerns, such as lack of trust on the provider or loss of data control. Hence, future research is required in order to address this important topic and enable secure HPC in the cloud. Further benchmarking, both in terms of performance and cost, that compare traditional HPC systems and public clouds, also taking into account other IaaS providers (e.g., Microsoft Azure, Google Compute Engine), would help to determine more accurately the performance/cost gap among those platforms.

Finally, a future direction is to further evaluate and characterize Big Data applications, particularly those that use the MapReduce programming model [25], where the Java-based Hadoop project [86] has become the preferred choice for their development. Nowadays, these applications are increasingly generating data sets so large that the use of HPC infrastructures (i.e., clusters, supercomputers and clouds) for efficient Big Data processing is becoming more and more frequent. As the Hadoop framework relies on Java sockets to implement its communication support, it can not take full advantage of the underlying high-speed networks used in HPC platforms. Therefore, some of the ideas and projects discussed in this Thesis could be of great interest for the implementation of a Java-based Hadoop-like framework optimized for HPC environments.

# Bibliography

[1] R. Alverson, D. Roweth, and L. Kaplan. The Gemini system interconnect. In *Proceedings of the 18th IEEE Annual Symposium on High-Performance Interconnects (HOTI'10)*, pages 83–87, Google Campus, Mountain View, CA, USA, 2010. pages 3

[2] Amazon Web Services LLC. Amazon Elastic Compute Cloud (Amazon EC2). http://aws.amazon.com/ec2. [Last visited: July 2014]. pages xviii, 2, 7, 254

[3] Amazon Web Services LLC. High Performance Computing using Amazon EC2. http://aws.amazon.com/ec2/hpc-applications/. [Last visited: July 2014]. pages 7

[4] A. Badal and A. Badano. Accelerating Monte Carlo simulations of photon transport in a voxelized geometry using a massively parallel graphics processing unit. *Medical Physics*, 36(11):4878–4880, 2009. pages 44

[5] L. Baduel, F. Baude, and D. Caromel. Object-oriented SPMD. In *Proceedings of the 5th IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID'05)*, pages 824–831, Cardiff, Wales, UK, 2005. pages 23

[6] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS parallel benchmarks. *International Journal of High Performance Computing Applications*, 5(3):63–73, 1991. pages 12

[7] M. Baker, B. Carpenter, G. Fox, S. H. Ko, and S. Lim. mpiJava: an object-oriented Java interface to MPI. In *Proceedings of the 1st International Work-*

*shop on Java for Parallel and Distributed Computing (IWJPDC'99)*, pages 748–762, San Juan, Puerto Rico, 1999. pages 24

[8] M. Baker, B. Carpenter, and A. Shafi. A pluggable architecture for high-performance Java messaging. *IEEE Distributed Systems Online*, 6(10):1–4, 2005. pages 10, 24

[9] M. Baker, B. Carpenter, and A. Shafi. MPJ Express: towards thread safe Java HPC. In *Proceedings of the 8th IEEE International Conference on Cluster Computing (CLUSTER'06)*, pages 1–10, Barcelona, Spain, 2006. pages 10, 24

[10] M. Baker, B. Carpenter, and A. Shafi. A buffering layer to support derived types and proprietary networks for Java HPC. *Scalable Computing: Practice and Experience*, 8(4):343–358, 2007. pages 14

[11] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. L. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP'03)*, pages 164–177, Bolton Landing (Lake George), NY, USA, 2003. pages 26

[12] B. Blount and S. Chatterjee. An evaluation of Java for numerical computing. *Scientific Programming*, 7(2):97–110, 1999. pages 4

[13] M. Bornemann, R. V. v. Nieuwpoort, and T. Kielmann. MPJ/Ibis: a flexible and efficient message passing platform for Java. In *Proceedings of the 12th European PVM/MPI Users' Group Meeting (EuroPVM/MPI'05)*, pages 217–224, Sorrento, Italy, 2005. pages 24

[14] J. M. Bull, L. A. Smith, M. D. Westhead, D. S. Henty, and R. A. Davey. A benchmark suite for high performance Java. *Concurrency: Practice and Experience*, 12(6):375–388, 2000. pages 24

[15] R. Buyya, C. S. Yeo, S. Venugopal, J. Broberg, and I. Brandic. Cloud computing and emerging IT platforms: vision, hype, and reality for delivering computing as the 5th utility. *Future Generation Computer Systems*, 25(6):599–616, 2009. pages XVII, 6, 252

[16] P. H. Carns, W. B. Ligon III, R. B. Ross, and R. Thakur. PVFS: a parallel virtual file system for Linux clusters. In *Proceedings of the 4th Annual Linux*

*Showcase & Conference (ALS'00)*, pages 317–328, Atlanta, GA, USA, 2000. pages 39

[17] B. Carpenter, G. Fox, S. H. Ko, and S. Lim. mpiJava 1.2: API specification. http://www.hpjava.org/reports/mpiJava-spec/mpiJava-spec/mpiJava-spec.html. [Last visited: July 2014]. pages 10, 24

[18] B. Carpenter, V. Getov, G. Judd, A. Skjellum, and G. Fox. MPJ: MPI-like message passing for Java. *Concurrency: Practice and Experience*, 12(11):1019–1038, 2000. pages xviii, 5, 24, 253

[19] V. Cavé, J. Zhao, J. Shirako, and V. Sarkar. Habanero-Java: the new adventures of old X10. In *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java (PPPJ'11)*, pages 51–61, Kongens Lyngby, Denmark, 2011. pages 24

[20] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'05)*, pages 519–538, San Diego, CA, USA, 2005. pages 24

[21] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: a benchmark suite for heterogeneous computing. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC'09)*, pages 44–54, Austin, TX, USA, 2009. pages 44

[22] Cray Inc. Using the GNI and DMAPP APIs. September 2013. http://docs.cray.com/books/S-2446-51/S-2446-51.pdf. [Last visited: July 2014]. pages 19

[23] A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tipparaju, and J. S. Vetter. The Scalable Heterogeneous Computing (SHOC) benchmark suite. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units (GPGPU-3)*, pages 63–74, Pittsburgh, PA, USA, 2010. pages 44

[24] K. Datta, D. Bonachea, and K. Yelick. Titanium performance and potential: an NPB experimental study. In *Proceedings of the 18th International Workshop on Languages and Compilers for Parallel Computing (LCPC'05)*, pages 200–214, Hawthorne, NY, USA, 2005. pages 23

[25] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008. pages 5, 238, 263

[26] E. Deelman, G. Singh, M. Livny, B. Berriman, and J. Good. The cost of doing science on the cloud: the Montage example. In *Proceedings of the 20th ACM/IEEE Supercomputing Conference (SC'08)*, pages 50:1–50:12, Austin, TX, USA, 2008. pages 7

[27] P. M. Dickens and R. Thakur. An evaluation of Java's I/O capabilities for high-performance computing. In *Proceedings of the 1st ACM Java Grande Conference (JAVA'00)*, pages 26–35, San Francisco, CA, USA, 2000. pages 4

[28] J. J. Dongarra, P. Luszczek, and A. Petitet. The LINPACK benchmark: past, present and future. *Concurrency and Computation: Practice and Experience*, 15(9):803–820, 2003. pages 44

[29] J. Ekanayake and G. Fox. High performance parallel computing with clouds and cloud technologies. In *Proceedings of the 1st International Conference on Cloud Computing (CLOUDCOMP'09)*, pages 20–38, Munich, Germany, 2009. pages 6, 7

[30] C. Evangelinos and C. N. Hill. Cloud computing for parallel scientific HPC applications: feasibility of running coupled atmosphere-ocean climate models on Amazon's EC2. In *Proceedings of the 1st Workshop on Cloud Computing and its Applications (CCA'08)*, pages 1–6, Chicago, IL, USA, 2008. pages 7

[31] R. R. Expósito, S. Ramos, G. L. Taboada, J. Touriño, and R. Doallo. FastMPJ: a scalable and efficient Java message-passing library. *Cluster Computing*, 2014. (in press, http://dx.doi.org/10.1007/s10586-014-0345-4). pages xxiv, 15, 24, 263

[32] R. R. Expósito, G. L. Taboada, S. Ramos, J. González-Domínguez, J. Touriño, and R. Doallo. Analysis of I/O performance on an Amazon EC2 cluster compute

and high I/O platform. *Journal of Grid Computing*, 11(4):613–631, 2013. pages xxvi, 34, 38, 265

[33] R. R. Expósito, G. L. Taboada, S. Ramos, J. Touriño, and R. Doallo. Evaluation of messaging middleware for high-performance cloud computing. *Personal and Ubiquitous Computing*, 17(8):1709–1719, 2013. pages xxv, 26, 31, 264

[34] R. R. Expósito, G. L. Taboada, S. Ramos, J. Touriño, and R. Doallo. General-purpose computation on GPUs for high performance cloud computing. *Concurrency and Computation: Practice and Experience*, 25(12):1628–1642, 2013. pages xxvi, 43, 265

[35] R. R. Expósito, G. L. Taboada, S. Ramos, J. Touriño, and R. Doallo. Performance analysis of HPC applications in the cloud. *Future Generation Computer Systems*, 29(1):218–229, 2013. pages xxv, 30, 264

[36] R. R. Expósito, G. L. Taboada, S. Ramos, J. Touriño, and R. Doallo. Low-latency Java communication devices on RDMA-enabled networks. 2014. (Submitted for journal publication). pages xxv, 19, 24, 264

[37] R. R. Expósito, G. L. Taboada, S. Ramos, J. Touriño, and R. Doallo. Performance evaluation of data-intensive computing applications on a public IaaS cloud. 2014. (Submitted for journal publication). pages xxvi, 38, 265

[38] R. R. Expósito, G. L. Taboada, J. Touriño, and R. Doallo. Design of scalable Java message-passing communications over InfiniBand. *Journal of Supercomputing*, 61(1):141–165, 2012. pages xxiv, 10, 24, 263

[39] FLASH I/O benchmark routine. http://www.ucolick.org/~zingale/flash_benchmark_io/. [Last visited: July 2014]. pages 41

[40] G. Fox and W. Furmanski. Java for parallel computing and as a general language for scientific and engineering simulation and modeling. *Concurrency: Practice and Experience*, 9(6):415–425, 1997. pages 4

[41] B. Fryxell, K. Olson, P. Ricker, F. X. Timmes, M. Zingale, D. Q. Lamb, P. MacNeice, R. Rosner, J. W. Truran, and H. Tufo. FLASH: an adaptive mesh hydrodynamics code for modeling astrophysical thermonuclear flashes. *Astrophysical Journal Supplement*, 131:273–334, 2000. pages 41

[42] B. Goglin. High-performance message passing over generic Ethernet hardware with Open-MX. *Parallel Computing*, 37(2):85–100, 2011. pages 15

[43] I. Gorton, P. Greenfield, A. Szalay, and R. Williams. Data-intensive computing in the 21st century. *Computer*, 41(4):30–32, 2008. pages 38

[44] Z. Hongwei, H. Wan, H. Jizhong, H. Jin, and Z. Lisheng. A performance study of Java communication stacks over InfiniBand and Gigabit Ethernet. In *Proceedings of the 4th IFIP International Conference on Network and Parallel Computing Workshops (NPC'07)*, pages 602–607, Dalian, China, 2007. pages xviii, 5, 10, 15, 253

[45] S. Huang, J. Huang, J. Dai, T. Xie, and B. Huang. The HiBench benchmark suite: characterization of the MapReduce-based data analysis. In *Proceedings of the 2nd International Workshop on Information & Software as Services (WISS'10)*, pages 41–51, Long Beach, CA, USA, 2010. pages 39

[46] IBTA. InfiniBand architecture specification release 1.2.1 Annex A16: RDMA over Converged Ethernet (RoCE). https://cw.infinibandta.org/document/dl/7148. [Last visited: July 2014]. pages 3

[47] IBTA. InfiniBand Trade Association. http://www.infinibandta.org/. [Last visited: July 2014]. pages 3

[48] IETF RFC 4392. IP over InfiniBand (IPoIB) architecture. http://www.ietf.org/rfc/rfc4392. [Last visited: July 2014]. pages 10

[49] K. R. Jackson, L. Ramakrishnan, K. Muriki, S. Canon, S. Cholia, J. Shalf, H. J. Wasserman, and N. J. Wright. Performance analysis of high performance computing applications on the Amazon web services cloud. In *Proceedings of the 2nd IEEE International Conference on Cloud Computing Technology and Science (CloudCom'10)*, pages 159–168, Indianapolis, IN, USA, 2010. pages 6, 7

[50] Java Grande Forum. http://www.javagrande.org. [Last visited: July 2014]. pages 24

[51] jCuda. Java bindings for CUDA. http://www.jcuda.org/. [Last visited: July 2014]. pages 24

[52] H. Jin and R. F. Van der Wijngaart. Performance characteristics of the multi-zone NAS parallel benchmarks. *Journal of Parallel and Distributed Computing*, 66(5):674–685, 2006. pages 31

[53] JOCL. Java bindings for OpenCL. http://www.jocl.org/. [Last visited: July 2014]. pages 24

[54] G. Juve, E. Deelman, G. B. Berriman, B. P. Berman, and P. Maechling. An evaluation of the cost and performance of scientific workflows on Amazon EC2. *Journal of Grid Computing*, 10(1):5–21, 2012. pages 35

[55] M. E. Kambites, J. Obdrzálek, and J. M. Bull. An OpenMP-like interface for parallel programming in Java. *Concurrency: Practice and Experience*, 13(8–9):793–814, 2001. pages 23

[56] V. V. Kindratenko, J. J. Enos, G. Shi, M. T. Showerman, G. W. Arnold, J. E. Stone, J. C. Phillips, and W.-M. Hwu. GPU clusters for high-performance computing. In *Proceedings of the 11th IEEE International Conference on Cluster Computing (CLUSTER'09)*, pages 1–8, New Orleans, LA, USA, 2009. pages xix, 4, 43, 254

[57] J. Maassen, R. V. v. Nieuwpoort, R. Veldema, H. E. Bal, T. Kielmann, C. J. H. Jacobs, and R. F. H. Hofman. Efficient Java RMI for parallel programming. *ACM Transactions on Programming Languages and Systems*, 23(6):747–775, 2001. pages 23

[58] D. A. Mallón, G. L. Taboada, J. Touriño, and R. Doallo. NPB-MPJ: NAS parallel benchmarks implementation for message-passing in Java. In *Proceedings of the 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP'09)*, pages 181–190, Weimar, Germany, 2009. pages 24

[59] R. P. Martin, A. M. Vahdat, D. E. Culler, and T. E. Anderson. Effects of communication latency, overhead, and bandwidth in a cluster architecture. In *Proceedings of the 24th Annual International Symposium on Computer Architecture (ISCA'97)*, pages 85–97, Denver, CO, USA, 1997. pages 4

[60] Mellanox Technologies. MellanoX Messaging (MXM) Accelerator. http://www.mellanox.com/page/products_dyn?product_family=135&menu_section=73. [Last visited: July 2014]. pages 19

[61] Message Passing Interface Forum. MPI: a Message Passing Interface standard. http://www.mcs.anl.gov/research/projects/mpi/, 1995. [Last visited: July 2014]. pages XVII, 4, 253

[62] Myrinet eXpress (MX). A high performance, low-level, message-passing interface for Myrinet. Version 1.2. October 2006. http://www.myricom.com/scs/MX/doc/mx.pdf. [Last visited: July 2014]. pages 15

[63] J. Napper and P. Bientinesi. Can cloud computing reach the Top500? In *Proceedings of the Combined Workshops on UnConventional High Performance Computing Workshop plus Memory Access Workshop (UCHPC-MAW'09)*, pages 17–20, Ischia, Italy, 2009. pages 6, 7

[64] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable parallel programming with CUDA. *Queue*, 6(2):40–53, 2008. pages 4

[65] R. V. v. Nieuwpoort, J. Maassen, G. Wrzesińska, R. F. H. Hofman, C. J. H. Jacobs, T. Kielmann, and H. E. Bal. Ibis: a flexible and efficient Java-based grid programming environment. *Concurrency and Computation: Practice and Experience*, 17(7-8):1079–1107, 2005. pages 23

[66] Orange File System (OrangeFS). http://www.orangefs.org/. [Last visited: July 2014]. pages 39

[67] S. Ostermann, A. Iosup, N. Yigitbasi, R. Prodan, T. Fahringer, and D. Epema. A performance analysis of EC2 cloud computing services for scientific computing. In *Proceedings of the 1st International Conference on Cloud Computing (CLOUDCOMP'09)*, pages 115–131, Munich, Germany, 2009. pages 7

[68] M. Philippsen, B. Haumacher, and C. Nester. More efficient serialization and RMI for Java. *Concurrency: Practice and Experience*, 12(7):495–518, 1999. pages 23

[69] J. C. Phillips, R. Braun, W. Wang, J. Gumbart, E. Tajkhorshid, E. Villa, C. Chipot, R. D. Skeel, L. Kalé, and K. Schulten. Scalable molecular dynamics

with NAMD. *Journal of Computational Chemistry*, 26(16):1781–1802, 2005. pages 44

[70] RDMA consortium. Architectural specifications for RDMA over TCP/IP. http://www.rdmaconsortium.org/. [Last visited: July 2014]. pages 3

[71] I. Rodero, H. Viswanathan, E. K. Lee, M. Gamell, D. Pompili, and M. Parashar. Energy-efficient thermal-aware autonomic management of virtualized HPC cloud infrastructure. *Journal of Grid Computing*, 10(3):447–473, 2012. pages 6

[72] L. Rodero-Merino, L. M. Vaquero, V. Gil, F. Galán, J. Fontán, R. S. Montero, and I. M. Llorente. From infrastructure delivery to service management in clouds. *Future Generation Computer Systems*, 26(8):1226–1240, 2010. pages xvii, 6, 252

[73] A. Shafi, B. Carpenter, M. Baker, and A. Hussain. A comparative study of Java and C performance in two large-scale parallel applications. *Concurrency and Computation: Practice and Experience*, 21(15):1882–1906, 2009. pages xviii, 4, 25, 253

[74] H. Shan, K. Antypas, and J. Shalf. Characterizing and predicting the I/O performance of HPC applications using a parameterized synthetic benchmark. In *Proceedings of the 20th ACM/IEEE Supercomputing Conference (SC'08)*, pages 42:1–42:12, Austin, TX, USA, 2008. pages 39

[75] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The Hadoop distributed file system. In *Proceedings of the 26th IEEE Symposium on Mass Storage Systems and Technologies (MSST'10)*, pages 1–10, Incline Village, NV, USA, 2010. pages 39

[76] V. Springel. The cosmological simulation code GADGET-2. *Monthly Notices of the Royal Astronomical Society*, 364(4):1105–1134, 2005. pages 25

[77] J. E. Stone, D. Gohara, and S. Guochun. OpenCL: a parallel programming standard for heterogeneous computing systems. *Computing in Science and Engineering*, 12(3):66–73, 2010. pages 4

[78] T. Suganuma, T. Ogasawara, M. Takeuchi, T. Yasue, M. Kawahito, K. Ishizaki, H. Komatsu, and T. Nakatani. Overview of the IBM Java just-in-time compiler. *IBM Systems Journal*, 39(1):175–193, 2000. pages 4

[79] G. L. Taboada, S. Ramos, R. R. Expósito, J. Touriño, and R. Doallo. Java in the high performance computing arena: research, practice and experience. *Science of Computer Programming*, 78(5):425–444, 2013. pages xviii, xxv, 4, 23, 253, 264

[80] G. L. Taboada, S. Ramos, J. Touriño, and R. Doallo. Design of efficient Java message-passing collectives on multi-core clusters. *Journal of Supercomputing*, 55(2):126–154, 2011. pages 14, 24

[81] G. L. Taboada, C. Teijeiro, and J. Touriño. High performance Java remote method invocation for parallel computing on clusters. In *Proceedings of the 12th IEEE Symposium on Computers and Communications (ISCC'07)*, pages 233–239, Aveiro, Portugal, 2007. pages 23

[82] G. L. Taboada, J. Touriño, and R. Doallo. Java Fast Sockets: enabling high-speed Java communications on high performance clusters. *Computer Communications*, 31(17):4049–4059, 2008. pages 15, 23

[83] G. L. Taboada, J. Touriño, and R. Doallo. F-MPJ: scalable Java message-passing communications on parallel systems. *Journal of Supercomputing*, 60(1):117–140, 2012. pages 14, 24

[84] G. L. Taboada, J. Touriño, R. Doallo, A. Shafi, M. Baker, and B. Carpenter. Device level communication libraries for high-performance computing in Java. *Concurrency and Computation: Practice and Experience*, 23(18):2382–2403, 2011. pages 14

[85] R. Thakur, W. Gropp, and E. Lusk. On implementing MPI-IO portably and with high performance. In *Proceedings of the 6th Workshop on I/O in Parallel and Distributed Systems (IOPADS'99)*, pages 23–32, Atlanta, GA, USA, 1999. pages 35

[86] The Apache Software Foundation. Apache Hadoop. http://hadoop.apache.org/. [Last visited: July 2014]. pages 4, 238, 263

[87] The Hierarchical Data Format (HDF) Group. HDF5. http://www.hdfgroup.org/HDF5/. [Last visited: July 2014]. pages 35

[88] G. K. Thiruvathukal, P. M. Dickens, and S. Bhatti. Java on networks of workstations (JavaNOW): a parallel computing framework inspired by Linda and the Message Passing Interface (MPI). *Concurrency: Practice and Experience*, 12(11):1093–1116, 2000. pages 4

[89] TOP500 Org. Top 500 supercomputer sites. http://www.top500.org/. [Last visited: July 2014]. pages 3, 10, 44

[90] C. Vecchiola, S. Pandey, and R. Buyya. High-performance cloud computing: a view of scientific applications. In *Proceedings of the 10th International Symposium on Pervasive Systems, Algorithms, and Networks (ISPAN'09)*, pages 4–16, Kaoshiung, Taiwan, ROC, 2009. pages 6, 7

[91] K. Verstoep, R. A. F. Bhoedjang, T. Rühl, H. E. Bal, and R. F. H. Hofman. Cluster communication protocols for parallel-programming systems. *ACM Transactions on Computer Systems*, 22(3):281–325, 2004. pages 4

[92] E. Walker. Benchmarking Amazon EC2 for high-performance scientific computing. *USENIX ;login:*, 33(5):18–23, 2008. pages 6, 7

[93] P. Wong and R. F. Van der Wijngaart. NAS parallel benchmarks I/O version 2.4. Technical Report NAS-03-002, NASA Ames Research Center, Moffett Field, CA, USA, 2003. pages 35, 41

[94] Xen Org. Xen PCI passthrough. http://wiki.xenproject.org/wiki/XenPCIpassthrough, 2005. [Last visited: July 2014]. pages 27

[95] Y. Yan, M. Grossman, and V. Sarkar. JCUDA: a programmer-friendly interface for accelerating Java programs with CUDA. In *Proceedings of the 15th International European Conference on Parallel and Distributed Computing (Euro-Par'09)*, pages 887–899, Delft, the Netherlands, 2009. pages 24

[96] K. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gay, P. Colella, and A. Aiken. Titanium: a high-performance Java dialect. *Concurrency: Practice and Experience*, 10(11–13):825–836, 1998. pages 23

# Appendix A

# Summary in Spanish

Este resumen se compone de una introducción, que explica la motivación y contexto de la Tesis, seguida de una sección sobre su organización en partes y capítulos. Después, sigue una enumeración de los medios que han sido necesarios para llevarla a cabo, para finalizar con las conclusiones, trabajo futuro y las principales contribuciones recogidas en ella.

## Introducción

La computación de altas prestaciones (*High Performance Computing*, HPC) se ha convertido durante las últimas décadas en una disciplina cada vez más importante como herramienta esencial para la investigación científica y la industria. De hecho, HPC es actualmente una de las disciplinas de vanguardia en el ámbito de las tecnologías de la información, con una amplia gama de aplicaciones paralelas que tienen una necesidad de potencia de cómputo creciente en diversos campos de la economía, la ciencia y la ingeniería. Estas aplicaciones, por lo general, consisten en la construcción de modelos matemáticos y técnicas de solución numérica que a menudo requieren una gran cantidad de recursos computacionales para realizar experimentos a gran escala o reducir la complejidad computacional en un plazo de tiempo razonable. Estas necesidades computacionales han sido satisfechas típicamente por supercomputadores instalados en laboratorios nacionales, grandes grupos de investigación o grandes empresas. A pesar de la importancia del HPC para la investigación científica y el crecimiento industrial, sólo los proyectos de investigación más importantes son capaces de afrontar el coste de un supercomputador. Además, el acceso a estos sistemas es por lo general muy restringido y habitualmente se realiza a través de convocatorias competitivas. Por lo tanto, para usuarios con pequeñas o medianas demandas de HPC, la opción convencional ha sido la instalación de su propio clúster privado.

El rendimiento y eficiencia de las comunicaciones inter-nodo son aspectos clave para la ejecución escalable de aplicaciones paralelas del ámbito HPC. De hecho, las arquitecturas de computación actuales, desde clusters multinúcleo a grandes supercomputadores, están agregando un número significativo de núcleos de procesamiento interconectados mediante una red de altas prestaciones o de baja latencia como InfiniBand, RoCE o Ethernet de alta velocidad (10/40 Gigabit). Por otra parte, el auge de la computación en la nube, o cloud computing [15, 72], ha generado un considerable interés en la comunidad HPC debido a la alta disponibilidad y fácil acceso a recursos computacionales a gran escala. Por lo tanto, proveedores de cloud públicos que ofrecen la infraestructura como servicio (*Infrastructure as a Service*, IaaS), como por ejemplo Amazon, están desplegando cada vez más recursos virtualizados específicos para aplicaciones HPC, los cuales permiten a los usuarios crear un clúster virtual en la nube sin excesivo esfuerzo y obtener así una gran potencia computacional. En este contexto, las plataformas de computación en la nube se han convertido en una alternativa interesante para desplegar un sistema HPC en la nube sin ningún conocimiento de la infraestructura subyacente, especialmente para aquellos usuarios con una demanda limitada y/o esporádica de recursos computacionales, o bien aquellos que no pueden permitirse el coste de adquirir un clúster para HPC.

El aumento progresivo y significativo en el número de núcleos disponibles en los procesadores actuales enfatiza la necesidad de disponer de soluciones paralelas escalables en las que la eficiencia del middleware de comunicación subyacente es fundamental. En este contexto, es vital poder aprovechar al máximo la potencia de los abundantes recursos computacionales de los sistemas HPC actuales al mismo tiempo que se hace un uso eficiente de las interesantes características que ofrecen las redes de altas prestaciones, como las operaciones de acceso directo a memoria remota (*Remote Direct Memory Access*, RDMA), y para ello resulta esencial continuar utilizando modelos de programación de fácil aprendizaje y uso. La interfaz de paso de mensajes (*Message-Passing Interface*, MPI) [61] sigue siendo el estándar de facto en el área de la computación paralela debido a su flexibilidad y portabilidad, y por ser capaz de lograr un alto rendimiento en sistemas muy diferentes. Por lo tanto, MPI constituye el modelo de programación más ampliamente extendido para aplicaciones paralelas en sistemas HPC, usando tradicionalmente lenguajes compilados a código nativo de cada plataforma (p.e., C/C++, Fortran).

Java es actualmente uno de los lenguajes de programación más populares, tanto en el mundo de la web e Internet como en computación distribuida. Además, Java se ha convertido en una alternativa interesante para computación paralela [73, 79]. El interés en Java para HPC está motivado por sus atractivas características que pueden ser especialmente beneficiosas para la programación paralela, tales como su inherente soporte multithreading y para aplicaciones en red, orientación a objetos, gestión automática de la memoria, portabilidad, facilidad de aprendizaje y, por tanto, alta productividad. Además, la mejora significativa en su rendimiento computacional ha reducido la tradicional diferencia de rendimiento entre Java y los lenguajes compilados a código nativo de cada plataforma. Esto ha sido gracias al uso de eficientes compiladores *Just-In-Time* (JIT), la técnica de compilación de la máquina virtual de Java (*Java Virtual Machine*, JVM) que proporciona al bytecode de Java rendimientos comparables al del código nativo. Sin embargo, aunque esta diferencia de rendimiento es normalmente pequeña en códigos secuenciales, obteniendo rendimientos similares a los lenguajes compilados, en aplicaciones paralelas puede ser particularmente grande, ya que la escalabilidad de estas aplicaciones depende en gran medida de la eficiencia de las comunicaciones. La razón principal es que las aplicaciones paralelas en Java carecen generalmente de un middleware de comunicación eficiente, incapaz de obtener el máximo rendimiento en presencia de redes de altas prestaciones [44]. La inexistencia de un soporte de comunicación eficiente en las bibliotecas de paso de mensajes en Java (*Message-Passing in Java*, MPJ) [18] conlleva por lo general un rendimiento sensiblemente inferior al de las bibliotecas MPI, lo que ha obstaculizado enormemente el uso de Java en HPC.

La presente Tesis Doctoral, "*Design and Evaluation of Low-Latency Communication Middleware on High Performance Computing Systems*", nace del intento de abordar un doble propósito. Por un lado, se centra en impulsar la utilización de Java en HPC mediante el desarrollo de un middleware de comunicación eficiente para computación paralela que supere las limitaciones discutidas anteriormente. Así, este middleware debe sacar el máximo provecho posible del hardware de red subyacente para proporcionar a las aplicaciones paralelas en Java comunicaciones inter-nodo de baja latencia y gran ancho de banda. Este primer objetivo se ha abordado específicamente en la primera parte de la Tesis, en particular para sistemas clúster y supercomputadores, por ser las arquitecturas para HPC más extendidas en la actualidad. Por otro lado, el segundo objetivo principal de esta Tesis (recogido en la segunda parte de la misma) se centra en el uso de infraestructuras de nube pública para HPC y computación científica. Por lo tanto, la escalabilidad del middleware de comunicación en Java desarrollado en la primera parte se ha analizado en el proveedor público de IaaS más popular: Amazon EC2 [2]. Además, se ha llevado a cabo un estudio de viabilidad del uso de los recursos de Amazon EC2 específicos para aplicaciones HPC. Este extenso análisis abarca desde la identificación de las principales causas de ineficiencia en las comunicaciones en un entorno de computación en la nube hasta la propuesta y evaluación de técnicas para la reducción de su impacto en la escalabilidad de códigos HPC intensivos en comunicaciones, tanto para Java como para lenguajes compilados a código nativo. Además, este estudio también considera otros aspectos importantes de Amazon EC2 para poder convertirse en una alternativa viable en el ámbito del HPC, tales como proporcionar un alto rendimiento de E/S a través del uso de discos de estado sólido (*Solid State Drive*, SSD), la caracterización del rendimiento de sistemas de ficheros paralelos/distribuidos para códigos intensivos en datos (p.e., aplicaciones Big Data) o la utilización de coprocesadores, tales como las unidades de procesamiento gráfico (*Graphics Processing Units*, GPU), como aceleradores many-core en códigos HPC [56].

## Organización de la Tesis

De acuerdo con la normativa vigente de la Universidade da Coruña, la presente Tesis Doctoral se ha estructurado como una Tesis por compendio de publicaciones de investigación. Concretamente, se compone de 9 artículos publicados en revistas indexadas en el *Journal Citation Reports* (JCR), y que se han agrupado en dos partes diferenciadas. La Tesis comienza con un capítulo introductorio, destinado a ofrecer al lector un breve resumen de la investigación realizada en dichos artículos. En primer lugar, este capítulo presenta el alcance y la motivación de la Tesis y proporciona una descripción clara de los principales objetivos a alcanzar. Seguidamente, se incluye una discusión general de los principales resultados de investigación de cada una de las partes con el fin de proporcionar una visión vertebradora y coherente de los diferentes trabajos.

A continuación, la Tesis se divide en dos partes. En la primera parte (*"Design of Low-Latency Java Communication Middleware on High-Speed Networks"*) se presenta el diseño, implementación y evaluación de un middleware de comunicaciones eficiente en Java para computación paralela. Este middleware implementa el soporte de sus comunicaciones sobre varios dispositivos a bajo nivel para paso de mensajes en Java, los cuales son capaces de obtener un gran rendimiento sobre diferentes redes de altas prestaciones. Además, se ha realizado un análisis del estado del arte de Java para HPC, incluyendo una extensa evaluación del rendimiento del middleware de comunicaciones desarrollado. Los artículos de revista que se incluyen en la primera parte de la Tesis, cada uno de ellos presentado en un capítulo independiente (Capítulos 2-5), son:

- R. R. Expósito, G. L. Taboada, J. Touriño, and R. Doallo. Design of scalable Java message-passing communications over InfiniBand. *Journal of Supercomputing*, 61(1):141–165, 2012.

- R. R. Expósito, S. Ramos, G. L. Taboada, J. Touriño, and R. Doallo. FastMPJ: a scalable and efficient Java message-passing library. *Cluster Computing*, 2014. (en prensa, `http://dx.doi.org/10.1007/s10586-014-0345-4`).

- R. R. Expósito, G. L. Taboada, S. Ramos, J. Touriño, and R. Doallo. Low-latency Java communication devices on RDMA-enabled networks. 2014. (Enviado para publicar en revista).

- G. L. Taboada, S. Ramos, R. R. Expósito, J. Touriño, and R. Doallo. Java in the high performance computing arena: research, practice and experience. *Science of Computer Programming*, 78(5):425–444, 2013.

La segunda parte de la Tesis (*"Evaluation of Communication Middleware for HPC on a Public Cloud Infrastructure"*) presenta un estudio detallado de la viabilidad del uso de una plataforma pública de computación en la nube para la ejecución eficiente de aplicaciones HPC. Amazon EC2, el principal proveedor público de IaaS, permite a los usuarios configurar clusters virtuales bajo demanda a través de varios recursos cloud destinados específicamente para HPC: instancias de tipo clúster con potentes procesadores multinúcleo, redes de altas prestaciones (10 Gigabit Ethernet), instancias clúster multi-GPU e instancias que proporcionan discos basados en tecnología SSD. Los artículos de revista que se incluyen en la segunda parte, cada uno de ellos también presentado en un capítulo independiente (Capítulos 6-10), son:

- R. R. Expósito, G. L. Taboada, S. Ramos, J. Touriño, and R. Doallo. Evaluation of messaging middleware for high-performance cloud computing. *Personal and Ubiquitous Computing*, 17(8):1709–1719, 2013.
- R. R. Expósito, G. L. Taboada, S. Ramos, J. Touriño, and R. Doallo. Performance analysis of HPC applications in the cloud. *Future Generation Computer Systems*, 29(1):218–229, 2013.
- R. R. Expósito, G. L. Taboada, S. Ramos, J. González-Domínguez, J. Touriño, and R. Doallo. Analysis of I/O performance on an Amazon EC2 cluster compute and high I/O platform. *Journal of Grid Computing*, 11(4):613–631, 2013.
- R. R. Expósito, G. L. Taboada, S. Ramos, J. Touriño, and R. Doallo. Performance evaluation of data-intensive computing applications on a public IaaS cloud. 2014. (Enviado para publicar en revista).
- R. R. Expósito, G. L. Taboada, S. Ramos, J. Touriño, and R. Doallo. General-purpose computation on GPUs for high performance cloud computing. *Concurrency and Computation: Practice and Experience*, 25(12):1628–1642, 2013.

Finalmente, en el último capítulo, de conclusiones y trabajo futuro, se resumen las principales contribuciones de la Tesis y se esbozan las principales líneas de investigación que se pueden derivar de este trabajo.

# Medios

Los medios necesarios para realizar esta Tesis Doctoral han sido los siguientes:

- Material de trabajo, recursos humanos y financiación económica proporcionados fundamentalmente por el Grupo de Arquitectura de Computadores de la Universidade da Coruña, junto con el Ministerio de Educación, Cultura y Deporte (a través de la beca FPU AP2010-4348).
- Acceso a material bibliográfico, a través de la biblioteca de la Universidade da Coruña.
- Además, esta Tesis se ha financiado a través de los siguientes proyectos de investigación:
  - Con financiación europea a través del proyecto "Open European Network for High Performance Computing on Complex Environments" (ComplexHPC, Acción COST ref. IC0805).
  - Con financiación estatal por parte del Ministerio de Economía y Competitividad a través del proyecto "Arquitecturas, Sistemas y Herramientas para Computación de Altas Prestaciones" (TIN2010-16735).
  - Con financiación autonómica por parte de la Xunta de Galicia a través del Programa de Consolidación y Estructuración de Unidades de Investigación Competitivas, en la modalidad de Grupos de Referencia Competitiva (Grupo de Arquitectura de Computadores, refs. GRC2013/055 y 2010/6), y en la modalidad de Redes de Investigación (Red Gallega de Computación de Altas Prestaciones, ref. 2010/53).
  - Con financiación privada a través de los proyectos "High Performance Computing for High Performance Trading (HPC4HPT)" financiado por la Fundación Barrié, y "F-MPJ-Cloud (Fast Message-Passing in Java on the Cloud)" financiado por una Research Grant de Amazon Web Services (AWS) LLC.
- Acceso a diversos clusters, supercomputadores y plataformas de computación en la nube:

- Clúster *Pluton* (Grupo de Arquitectura de Computadores, Universidade da Coruña). Compuesto inicialmente por 16 nodos con 2 procesadores Intel Xeon Nehalem-EP de 4 núcleos y hasta 16 GB de RAM, interconectados mediante InfiniBand DDR, y adicionalmente dos nodos mediante 10 Gigabit Ethernet. Además, 2 nodos con procesador Intel Xeon Sandy Bridge-EP de 4 núcleos y 32 GB de RAM, interconectados mediante InfiniBand FDR, RoCE e iWARP, y 4 nodos con procesador Intel Xeon Westmere-EP de 6 núcleos, 12 GB de RAM y 2 GPUs NVIDIA Tesla "Fermi" 2050 por nodo, interconectados mediante InfiniBand QDR. Posteriormente se han incorporado 16 nodos con 2 procesadores Intel Xeon Sandy Bridge-EP de 8 núcleos, 64 GB de RAM y 2 GPUs NVIDIA Tesla "Kepler" K20m por nodo, interconectados mediante InfiniBand FDR.

- Clúster *DAS-4* (Advanced School for Computing and Imaging, ASCI, Vrije University Amsterdam, Holanda). Para los experimentos de la Tesis se utilizaron hasta 64 nodos con 2 procesadores Intel Xeon Westmere-EP de 4 núcleos y 24 GB de RAM, interconectados mediante InfiniBand QDR. Adicionalmente se utilizó un nodo de memoria compartida con 4 procesadores AMD Opteron Magny-Cours de 12 núcleos y 128 GB de RAM.

- Clúster *XB5* (Grupo de Filogenómica, Universidad de Vigo). Se utilizó un nodo de memoria compartida con 4 procesadores Intel Xeon Westmere-EX de 10 núcleos y 512 GB de RAM.

- Supercomputador *Finis Terrae* (Centro de Supercomputación de Galicia, CESGA): 144 nodos con 8 procesadores Intel Itanium-2 Montvale de 2 núcleos y 128 GB de RAM, interconectados mediante InfiniBand DDR. Adicionalmente se utilizó un sistema Superdome de memoria compartida con 64 procesadores Intel Itanium-2 Montvale de 2 núcleos y 1 TB de RAM.

- Supercomputador *MareNostrum* (Barcelona Supercomputing Center, BSC): 2560 nodos con 2 procesadores IBM PowerPC 970MP de 2 núcleos y 8 GB de RAM, interconectados mediante Myrinet 2000.

- Supercomputador *Hermit* (High Performance Computing Center Stuttgart, HLRS, Alemania): 3552 nodos con 2 procesadores AMD Opteron Interlagos de 16 núcleos y hasta 64 GB de RAM, interconectados mediante Gemini, una red propietaria de Cray con una topología de toro 3D.

- Plataforma de computación en la nube *Amazon EC2* (Amazon Web Services, AWS). Se han utilizado diversos tipos de instancias: (1) CC1, 2 procesadores Intel Xeon Nehalem-EP de 4 núcleos, 23 GB de RAM y 2 discos de almacenamiento local por instancia; (2) CC2, 2 procesadores Intel Xeon Sandy Bridge-EP de 8 núcleos, 60.5 GB de RAM y 4 discos por instancia; (3) CG1, 2 procesadores Intel Xeon Nehalem-EP de 4 núcleos, 22 GB de RAM, 2 GPUs NVIDIA Tesla "Fermi" 2050 y 2 discos por instancia; (4) HI1, 2 procesadores Intel Xeon Westmere-EP de 4 núcleos, 60.5 GB de RAM y 2 discos con tecnología SSD por instancia; (5) CR1, 2 procesadores Intel Xeon Sandy Bridge-EP de 8 núcleos, 244 GB de RAM y 2 discos con tecnología SSD por instancia; y (6) HS1, 1 procesador Intel Xeon Sandy Bridge-EP de 8 núcleos, 117 GB de RAM y 24 discos por instancia. Todas estas instancias están interconectadas mediante una red 10 Gigabit Ethernet.

- Estancia de investigación de 3 meses de duración en el Rechenzentrum Universität Mannheim, Alemania, la cual permitió el acceso al supercomputador *Hermit* instalado en el High Performance Computing Center Stuttgart (HLRS). En esta estancia se desarrolló el dispositivo de comunicaciones de FastMPJ para el soporte de red de la familia de supercomputadores Cray XE/XK/XC. La estancia fue financiada por la Universidade da Coruña junto con la empresa INDITEX S.A. mediante una beca obtenida en concurrencia competitiva en 2013.

## Conclusiones

Esta Tesis Doctoral, *"Design and Evaluation of Low-Latency Communication Middleware on High Performance Computing Systems"*, ha llevado a cabo con éxito un doble propósito. Por un lado, se ha presentado el diseño, implementación y optimización de soluciones de comunicación escalables en Java para un soporte eficiente de redes de altas prestaciones. Una de las principales contribuciones de la Tesis es el desarrollo y evaluación de un middleware de comunicación de paso de mensajes en Java para computación paralela, denominado FastMPJ, que permite incrementar significativamente el rendimiento de las aplicaciones paralelas en Java sobre clusters y supercomputadores, las arquitecturas para HPC más extendidas en la actualidad. FastMPJ implementa el API mpiJava 1.2, la especificación basada en el estándar MPI más extendida en Java, con el fin de permitir un desarrollo altamente productivo de aplicaciones paralelas MPJ. Este middleware integra la colección de dispositivos de comunicación a bajo nivel en Java desarrollados en esta Tesis, los cuales permiten obtener comunicaciones punto a punto de baja latencia (1 $\mu$s) y alto ancho de banda (hasta 49 Gbps), proporcionando así un soporte eficiente para redes de altas prestaciones en Java. Por lo tanto, contribuciones clave de esta Tesis son el diseño y la implementación de los siguientes dispositivos de comunicación: (1) `ibvdev`, para el soporte de adaptadores InfiniBand, RoCE e iWARP en general; (2) `mxdev`, para redes Myrinet y Ethernet de alta velocidad y también hardware Ethernet genérico; (3) `psmdev`, para adaptadores InfiniBand del fabricante Intel/QLogic; (4) `mxmdev`, para adaptadores InfiniBand/RoCE del fabricante Mellanox; (5) `ugnidev`, para las redes Gemini/Aries utilizadas en los supercomputadores Cray XE/XK/XC; y (6) `mpidev`, un dispositivo nativo para ser utilizado con una librería MPI. La integración de estos dispositivos en FastMPJ ha permitido a las aplicaciones MPJ hacer uso de un gran número de núcleos (hasta 4096) y aprovechar eficientemente el hardware RDMA subyacente. De hecho, el desarrollo de este middleware, que incluso es competitivo con bibliotecas MPI en términos de comunicaciones punto a punto, permite reducir la diferencia de rendimiento en aplicaciones HPC entre Java y los lenguajes compilados a código nativo. La Tesis también proporciona un análisis actualizado del estado del arte de Java para HPC, incluyendo una evaluación exhaustiva de proyectos actuales. Como principal conclusión, los avances obtenidos en la eficiencia de las comunicaciones de Java han servido para demostrar que el uso de Java en HPC es factible, ya que es capaz de obtener un alto rendimiento.

Por otro lado, esta Tesis también ha analizado la viabilidad de usar una infraestructura pública en la nube para la ejecución de aplicaciones HPC. Se han evaluado diversos recursos cloud específicamente diseñados para HPC del principal proveedor público de IaaS, Amazon EC2. Entre estos recursos, se han evaluado distintas instancias de tipo clúster que proporcionan potentes procesadores multinúcleo y están interconectadas mediante una red de altas prestaciones (10 Gigabit Ethernet). Por lo tanto, otra aportación fundamental de esta Tesis ha sido un extenso estudio de los recursos cloud de Amazon EC2 para HPC, que abarca desde la identificación de las principales causas de la ineficiencia en las comunicaciones en un entorno de computación en la nube hasta la propuesta y evaluación de técnicas para la reducción de su impacto en la escalabilidad de códigos HPC intensivos en comunicaciones, tanto para Java como para lenguajes compilados a código nativo. Este estudio no sólo incluye una evaluación comparativa en la nube del middleware de comunicación desarrollado en esta Tesis, sino que también considera otros aspectos importantes de Amazon EC2 para poder llegar a ser una alternativa viable en el ámbito del HPC. Por lo tanto, se han realizado evaluaciones adicionales teniendo en cuenta el rendimiento del subsistema de E/S, la caracterización del rendimiento de sistemas de ficheros paralelos/distribuidos para códigos intensivos en datos (p.e., aplicaciones Big Data) y la viabilidad de utilizar arquitecturas heterogéneas con aceleradores many-core como las GPUs. A través de un análisis en términos de rendimiento y coste de aplicaciones HPC y de la comparación con entornos privados virtualizados y no virtualizados, se ha demostrado que diferentes aplicaciones exhiben diferentes características que las pueden hacer más o menos adecuadas para ejecutarse en la nube. Como principal conclusión, esta Tesis demuestra el impacto significativo que los entornos virtualizados todavía imponen en el rendimiento de las comunicaciones, lo que dificulta la adopción de aplicaciones intensivas en comunicaciones en la nube. Aunque es posible aumentar de manera significativa su escalabilidad siguiendo las directrices para optimización del rendimiento que se sugieren en esta Tesis, tales como reducir el número de procesos por instancia o la combinación del paradigma de paso de mensajes con el uso de multithreading, la red de interconexión continúa siendo el principal cuello de botella en el rendimiento, especialmente para códigos GPGPU intensivos en comunicaciones y aplicaciones paralelas intensivas en E/S. Esta situación exige optimizaciones adecuadas en la capa de virtualización, como el acceso directo al hardware de red, para poder superar las limitaciones actuales.

## Trabajo Futuro

En cuanto al trabajo futuro, el desarrollo de un nuevo dispositivo de comunicación en Java específico para entornos de memoria compartida/distribuida sería de gran interés. Así, en este dispositivo híbrido, las comunicaciones intra-nodo se realizarían mediante algún mecanismo de memoria compartida, ya sea inter-proceso (p.e., mmap, SysV) o intra-proceso (p.e., threads de Java), mientras que las transferencias inter-nodo se aprovecharían de los dispositivos de comunicación eficientes que se han presentado en esta Tesis. Con respecto a este punto, los dispositivos de FastMPJ se podrían ampliar mediante la inclusión de su propia implementación de operaciones colectivas en la capa `xxdev`, incluso sin necesidad de hacer uso de comunicaciones punto a punto. El uso de estos algoritmos permitiría realizar una única llamada al dispositivo por cada operación colectiva, lo que reduciría la sobrecarga. Además, las características de la red subyacente podrían explotarse de manera más eficiente mediante la aceleración de colectivas basada en hardware, que descargaría dichas operaciones sobre los adaptadores y conmutadores de la red. Por otra parte, la extensión de la API mpiJava 1.2 para cumplir con la especificación de MPI 3.0 y su correspondiente implementación de referencia en FastMPJ también sería altamente interesante.

En cuanto al tema de HPC en la nube, serían necesarias evaluaciones adicionales de los nuevos recursos cloud para HPC recientemente ofertados por Amazon (p.e., las instancias clúster C3, las cuales, según Amazon, proporcionan mejoras en la red de interconexión para aumentar el número de paquetes por segundo y reducir la latencia), con el objetivo de determinar si se ha reducido el cuello de botella de la red. Otro de los principales problemas que desalientan habitualmente a los usuarios de HPC a mover sus aplicaciones a la nube son aquellos que tienen que ver con la seguridad, como pueden ser la falta de confianza en el proveedor o la pérdida de control de los datos. Por lo tanto, se necesitan nuevas investigaciones que aborden este importante tópico y que permitan así dotar de seguridad al HPC en la nube. Comparaciones adicionales, tanto en términos de rendimiento como de coste, entre sistemas HPC tradicionales y plataformas públicas en la nube, teniendo en cuenta también a otros proveedores de IaaS (p.e., Microsoft Azure, Google Compute Engine), ayudarían a determinar con mayor precisión la diferencia de rendimiento/coste entre dichos entornos.

Por último, otra línea futura sería profundizar en la evaluación y caracterización del rendimiento de aplicaciones en el ámbito del procesamiento Big Data, en particular aquellas que utilizan el modelo de programación MapReduce [25], donde el proyecto Apache Hadoop [86], basado en Java, se ha convertido en el entorno preferido para su desarrollo. Hoy en día estas aplicaciones están generando conjuntos de datos tan grandes que el uso de infraestructuras HPC, tanto clusters como supercomputadores y clouds, se está haciendo cada vez más frecuente para conseguir un procesamiento Big Data escalable. Como Hadoop implementa el soporte de sus comunicaciones mediante sockets de Java, no puede explotar el máximo rendimiento de las redes de interconexión utilizadas en HPC. Por lo tanto, algunas de las ideas y proyectos que se han discutido en esta Tesis pueden ser de gran interés para la implementación en Java de un entorno Hadoop optimizado para sistemas HPC.

## Principales Contribuciones

Las principales aportaciones de la primera parte de esta Tesis son:

1. El diseño e implementación de un dispositivo de comunicación a bajo nivel para paso de mensajes en Java, `ibvdev`, desarrollado sobre la interfaz Verbs y que proporciona comunicaciones escalables en sistemas interconectados mediante la red InfiniBand [38].

2. El diseño, implementación y evaluación de una biblioteca eficiente para paso de mensajes en Java: FastMPJ. Este middleware de comunicación en Java no sólo se beneficia de la integración de `ibvdev`, sino también de la implementación de dos nuevos dispositivos de comunicación a bajo nivel: (1) `mxdev`, implementado sobre MX/Open-MX para el soporte de redes Myrinet y hardware Ethernet genérico; y (2) `psmdev`, implementado sobre InfiniPath PSM para el soporte nativo de la familia de adaptadores InfiniBand del fabricante Intel/QLogic [31].

3. El diseño, implementación y optimización de dispositivos de comunicación en Java para el soporte de redes RDMA. Este trabajo incluye el diseño e implementación de dos nuevos dispositivos de comunicación a bajo nivel, `ugnidev` y `mxmdev`, que también se han integrado en FastMPJ. El primer dispositivo

proporciona un soporte eficiente de las redes RDMA utilizadas en los supercomputadores Cray actuales. El segundo incluye el soporte para la biblioteca de comunicaciones basada en paso de mensajes recientemente desarrollada por Mellanox para sus adaptadores RDMA. Además, se presenta una versión mejorada del dispositivo `ibvdev`, la cual soporta redes RDMA adicionales e incluye un protocolo optimizado para el envío de mensajes de pequeño tamaño [36].

4. Un análisis actual del estado del arte de Java para computación de altas prestaciones, incluyendo una extensa evaluación del rendimiento de soluciones de paso de mensajes debido a su escalabilidad y uso extendido en HPC. En consecuencia, FastMPJ ha sido evaluado de forma comparativa con bibiliotecas MPI y MPJ representativas. Este análisis también incluye una revisión de soluciones en Java para programación paralela tanto en memoria compartida como distribuida, mostrando un número importante de proyectos pasados y actuales que son el resultado del interés continuo en el uso de Java en HPC [79].

Las principales aportaciones de la segunda parte de la Tesis son:

1. Un estudio detallado del impacto de la sobrecarga de la virtualización de red en la escalabilidad de aplicaciones HPC en el proveedor de cloud público de IaaS más popular: Amazon EC2. Este trabajo compara la primera generación de instancias de EC2 de tipo clúster específicas para HPC, cuyo acceso a su red de altas prestaciones (10 Gigabit Ethernet) está paravirtualizado, con el rendimiento de un cloud privado similar que soporta el acceso directo a la misma tecnología de red utilizando la técnica *PCI passthrough* y con el mismo sistema ejecutándose de forma nativa en un entorno no virtualizado [33].

2. Un análisis exhaustivo de los principales cuellos de botella en la escalabilidad de aplicaciones HPC en Amazon EC2, junto con la propuesta y evaluación de técnicas para reducir el impacto de la sobrecarga que produce la virtualización de red. Este trabajo proporciona una visión más amplia sobre el rendimiento de las aplicaciones HPC en Amazon EC2, utilizando para ello un importante número de núcleos (hasta 512) y comparando la primera y segunda generación de instancias de tipo clúster para HPC, tanto en términos de rendimiento, escalabilidad y relación coste-eficiencia como teniendo en cuenta modelos de programación híbridos (p.e., MPI+OpenMP) [35].

3. Una evaluación del subsistema de E/S en Amazon EC2. En este trabajo se evalúan las dos generaciones de instancias clúster para HPC junto con instancias de almacenamiento optimizado que proporcionan discos basados en tecnología SSD, con el objetivo de determinar su idoneidad para aplicaciones intensivas en E/S. La evaluación se ha realizado a diferentes niveles, que abarcan desde los dispositivos de almacenamiento en la nube a bajo nivel (p.e., discos efímeros) e interfaces de E/S (p.e., MPI-IO, HDF5), hasta el nivel de aplicación, incluyendo también un análisis en términos de coste. Además, se ha caracterizado el rendimiento del sistema de archivos de red (*Network File System*, NFS), mostrando el impacto en el rendimiento de sus principales parámetros de configuración en un entorno cloud virtualizado [32].

4. Un análisis del rendimiento de aplicaciones intensivas en datos en Amazon EC2. Este trabajo analiza al detalle la pila software de E/S para códigos intensivos en datos, tanto en el ámbito HPC como para el procesamiento Big Data. Concretamente, se ha caracterizado el rendimiento y el coste de cuatro tipos de instancias que proporcionan una red 10 Gigabit Ethernet. Para ello se han realizado experimentos a diferentes niveles utilizando un conjunto representativo de benchmarks (p.e., IOR, Intel Hibench), sistemas de ficheros paralelos/distribuidos (OrangeFS, HDFS), entornos para computación distribuida (Apache Hadoop), aplicaciones HPC intensivas en E/S (p.e., FLASH-IO) y códigos representativos basados en el paradigma MapReduce para procesamiento Big Data (p.e., Sort, PageRank) [37].

5. Un estudio de viabilidad del uso de arquitecturas heterogéneas con aceleradores many-core en Amazon EC2. La familia de instancias clúster para GPGPU se ha evaluado mediante kernels sintéticos y benchmarks representativos, utilizando códigos basados tanto en CUDA (*Compute Unified Device Architecture*) como en OpenCL (*Open Computing Language*), y sus resultados se han comparado con un entorno GPU no virtualizado con el objetivo de analizar la sobrecarga de virtualización para códigos GPGPU. Por otra parte, a nivel de aplicación se han tenido en cuenta códigos paralelos híbridos (p.e., MPI+CUDA), utilizando para ello dos aplicaciones paralelas y el benchmark HPL (*High Performance Linpack*) [34].