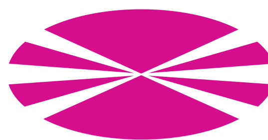
High Performance Java for Multi-core Systems

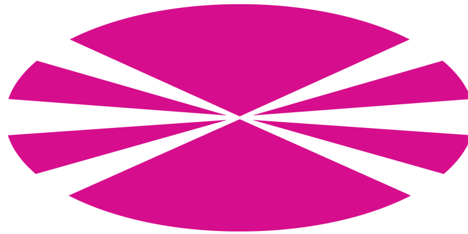
Sabela Ramos Garea



Department of Electronics and Systems
University of A Coruña, Spain

Department of Electronics and Systems

University of A Coruña, Spain



PhD Thesis

High Performance Java for Multi-core Systems

Sabela Ramos Garea

June 2013

PhD Advisors:
Guillermo López Taboada
Juan Touriño Domínguez

Dr. Guillermo López Taboada
Profesor Contratado Doutor
Dpto. de Electrónica e Sistemas
Universidade da Coruña

Dr. Juan Touriño Domínguez
Catedrático de Universidade
Dpto. de Electrónica e Sistemas
Universidade da Coruña

CERTIFICAN

Que a memoria titulada "*High Performance Java for Multi-core Systems*" foi realizada por Dna. Sabela Ramos Garea baixo a nosa dirección no Departamento de Electrónica e Sistemas da Universidade da Coruña e conclúe a Tese de Doutoramento que presenta para a obtención do título de Doutora pola Universidade da Coruña coa Mención de Doutor Internacional.

Na Coruña, o 25 de Xuño de 2013

Asdo.: Guillermo López Taboada
Director da Tese de Doutoramento

Asdo.: Juan Touriño Domínguez
Director da Tese de Doutoramento

Asdo.: Sabela Ramos Garea
Autora da Tese de Doutoramento

Resumo da Tese de Doutoramento

Introducción

O interese que a comunidade da computación de altas prestacións (High Performance Computing, HPC) vén demostrando na linguaxe Java, aumentou considerablemente nos últimos anos grazas á mellora experimentada en canto a rendemento e ás características que fan de Java unha linguaxe altamente productiva. Entre elas, cabe destacar a portabilidade e independenza da plataforma, a simplicidade, a robustez, a seguridade, a orientación a obxectos e contar cunha grande comunidade de desarrolladores, tanto no mundo académico coma no empresarial. Ademais da mellora de rendemento, hai dúas características cruciais que propiciaron a progresiva adopción de Java na computación de altas prestacións: o soporte para redes de interconexión e a incorporación de multithreading no núcleo da linguaxe. Estas dúas calidades fan de Java unha linguaxe idónea para a programación en contornos paralelos.

Dentro dos contornos paralelos actuais, os sistemas de memoria compartida volven a cobrar forza nun escenario que, ata hai pouco, estaba dominado por clusters de sistemas monoprocesador, que permitían construír contornos de computación de altas prestacións reducindo os costes. Non obstante, o incremento de rendemento dos procesadores empezou a acadar límites físicos, o que levou á aparición de problemas de disipación de calor e de alta ineficiencia enerxética. Para superar estes obstáculos, os fabricantes de hardware comezaron a centrarse na mellora do rendemento dos procesadores mediante o incremento do número de núcleos presentes en cada un deles, o que se coñece como procesadores multi-core ou multi-núcleo. De feito, na actualidade, procesadores de catro ou máis núcleos son comúns en ordenadores de

uso persoal.

O incremento progresivo no número de núcleos está conducindo tamén á xeralización de procesadores many-core para a computación de ámbito xeral. E neste eido volve a aparecer o problema do consumo enerxético, xa que ter un grande número de núcleos por procesador en situacións nas que só se precisa unha pequena parte, provoca unha grande ineficiencia enerxética. Para solventar este problema, aparecen os aceleradores de uso específico, como as tarxetas gráficas (Graphics Processing Unit, GPU), arquitecturas many-core que só se utilizan para tareas específicas de procesamento de gráficos mentres outro procesador (host) é o que se encarga da computación xeral. As características destes aceleradores e as súas elevadas capacidades de cómputo fixeron que fosen adoptadas para a aceleración de códigos vectoriais non necesariamente relacionados coa computación gráfica. Sen embargo, as dificultades de programación destes aceleradores favoreceron a aparición de aceleradores de arquitecturas x86 que soportan linguaxes e paradigmas tradicionais de programación, coma o coprocesador Xeon Phi comercializado por Intel.

O éxito das arquitecturas multi- e many-core indica a necesidade de ferramentas e librerías de programación que exploten o rendemento en memoria compartida, onde Java, co soporte para multithreading, presenta unha grande vantaxe. O principal problema é que a API de manexo de threads é complicada, sendo o usuario o que ten que lidiar coa creación/destrucción de threads e, o que é máis importante, coa posibilidade de aparición de race conditions e inconsistencias. Para solventar isto, Java inclúe unha librería de concorrencia na que se traballa con pools de threads e cun paradigma de programación orientado á descomposición da carga de traballo en tarefas (que pode ser recursiva utilizando a funcionalidade de fork/join). Este paradigma obliga a que os algoritmos paralelos non orientados a tarefas teñan que ser reescritos ou utilizar estas ferramentas de maneira non eficiente.

Esta Tese presenta unha análise detallada do estado da arte en canto á situación de Java para a programación de sistemas de memoria compartida, centrándose en solucións axeitadas para a computación de altas prestacións. Os principais obxectivos deste traballo son a análise do estado do soporte Java á computación de altas prestacións en memoria compartida e o desenvolvemento de middleware para mellorar tanto o rendemento como a productividade. En consecuencia, levouse a cabo un estudo do soporte software dispoñible para a programación multi-núcleo en Java

e realizouse o deseño, implementación e avaliación dun dispositivo de comunicacións de paso de mensaxes en Java optimizado para memoria compartida. Este dispositivo proporciona unha API de alto nivel que elimina a necesidade de manexar threads ou a descomposición en tarefas. Esta API segue a especificación Java de paso de mensaxes (Message Passing in Java, MPJ) baseada no estándar MPI, amplamente utilizado en computación de altas prestacións. Tamén se inclúe unha optimización de patróns de comunicacións entre procesos ou threads (operacións colectivas), tanto bloqueantes coma non bloqueantes, para contornos baseados en sistemas multinúcleo, alén dunha análise da adecuación e potencial das colectivas non bloqueantes en contornos de memoria compartida. Por outra banda, fíxose un estudo e avaliación de solucións dispoñibles para a explotación de sistemas many-core en aplicacións Java. A principal conclusión deste estudo é que o uso de Java en contornos many-core é productivo e pode proporcionar resultados de alto rendemento.

Metodoloxía de Traballo

A metodoloxía de investigación seguida na presente Tese de Doutoramento consistiu en:

- Definir a lista de obxectivos identificando as tarefas necesarias para acadalos, tendo en conta os traballos previos e os recursos dispoñibles.
- Determinar a secuencia de execución das tarefas aténdose ás restriccións que puidesen existir e buscando a orde máis axeitada.
- Establecer a duración das tarefas e a oportunidade de desenvolvemento nun momento determinado.
- Organizar os obxectivos e tarefas en bloques de certa entidade que definan etapas.
- Definir, para cada etapa, os fitos, ou metas a acadar en tempo definido, tendo en conta que cada etapa pode constar dunha ou varias metas.

Os obxectivos e tarefas foron definidos de maneira iterativa para poder aproveitar o coñecemento adquirido en etapas previas.

A continuación, enumérase a lista de obxectivos (***O***), agrupados en bloques (**B**), detallando as tarefas (T) que foron desenvolvidas na Tese para acadar cada un dos obxectivos.

B 1. Análise das capacidades da linguaxe Java para a programación de altas prestacións en memoria compartida.

O 1.1. Análise da programación en Java para memoria compartida.

T 1.1.1. Estudo da usabilidade e rendemento de Java para programación de altas prestacións.

T 1.1.2. Análise das caraterísticas internas de Java para programación paralela en memoria compartida.

T 1.1.3. Análise doutros modelos de programación paralela utilizados en Java actualmente.

T 1.1.4. Avaliación das necesidades de optimización do soporte Java para arquitecturas de memoria compartida.

O 1.2. Análise do estado actual de dispoñibilidade do soporte en Java para programación heteroxénea.

T 1.2.1. Búsqueda bibliográfica de solucións e proxectos existentes que den soporte á programación heteroxénea en Java.

T 1.2.2. Análise do soporte dispoñible e identificación de carencias.

B 2. Estudo das principais arquitecturas de memoria compartida dispoñibles.

O 2.1. Estudo e avaliación de arquitecturas de memoria compartida con soporte para a execución de instruccións x86.

T 2.1.1. Análise detallada de arquitecturas multi-core dispoñibles.

T 2.1.2. Análise detallada de arquitecturas many-core x86 dispoñibles.

O 2.2. Estudo e avaliación doutras arquitecturas de memoria compartida.

T 2.2.1. Análise detallada de unidades de procesamento gráfico (Graphics Processing Units, GPU) para programación de propósito xeral.

B 3. Análise, deseño e implementación dunha solución de paso de mensaxes en Java para memoria compartida.

O 3.1. Avaliación do estado da arte do paso de mensaxes en Java para memoria compartida.

T 3.1.1. Análise de proxectos de paso de mensaxes noutras linguaxes con soporte específico para memoria compartida.

T 3.1.2. Análise do soporte para paso de mensaxes en Java.

O 3.2. Deseño e implementación dunha solución de paso de mensaxes en Java para memoria compartida.

T 3.2.1. Deseño da solución tendo en conta as características específicas de Java e o seu soporte para programación paralela en memoria compartida.

T 3.2.2. Implementación do deseño de paso de mensaxes proposto.

T 3.2.3. Optimización da solución implementada facendo especial fincapé nos puntos de sincronización entre threads.

O 3.3. Avaliación da solución proposta.

T 3.3.1. Deseño do conxunto de probas a realizar e selección das librerías máis relevantes entre as atopadas no punto ***O 3.1*** para comparar coa solución proposta.

T 3.3.2. Análise e selección dos contornos de probas e do hardware dispoñible.

T 3.3.3. Realización de probas e análise de resultados coa consecuente extracción de conclusións e posible re-optimización.

B 4. Análise do rendemento e optimización de operacións colectivas para memoria compartida en Java.

O 4.1. Análise e implementación de operacións colectivas para contornos con procesadores multi-core.

T 4.1.1. Análise do estado da arte das operacións colectivas, tanto en Java coma en linguaxes nativas, e da súa adecuación a contornos multi-core.

T 4.1.2. Implementación de algoritmos de operacións colectivas optimizados para sistemas con nodos multi-core, tanto aislados (memoria compartida) coma conectados mediante redes de interconexión (memoria compartida-distribuída).

O 4.2. Análise do potencial das operacións colectivas non bloqueantes para memoria compartida en Java.

T 4.2.1. Estudo das implementacións existentes e da adecuación dunha implementación para memoria compartida utilizando Java.

T 4.2.2. Deseño e implementación dunha librería de colectivas non bloqueantes para Java optimizada para memoria compartida.

O 4.3. Avaliación e análise do rendemento das librerías de colectivas implementadas.

T 4.3.1. Deseño do conxunto de probas.

T 4.3.2. Análise e selección dos contornos de probas e do hardware dispoñible.

T 4.3.3. Realización de probas e análise de rendemento coa consecuente extracción de conclusións e posible re-optimización.

B 5. Análise da situación actual de Java para programación heteroxénea.

O 5.1. Avaliación da programación heteroxénea en Java utilizando coprocesadores con arquitectura x86.

T 5.1.1. Análise bibliográfica de solucións existentes.

T 5.1.2. Avaliación do rendemento das solucións identificadas a través do deseño dun conxunto de benchmarks.

O 5.2. Avaliación da programación heteroxénea en Java utilizando aceleradores gráficos.

T 5.2.1. Análise bibliográfica de solucións existentes para a programación de propósito xeral en GPUs e noutros aceleradores utilizando Java.

T 5.2.2. Avaliación dunha selección das solucións atopadas a través do deseño dun conxunto de benchmarks.

O 5.3. Análise da productividade das solucións para programación heteroxénea en Java.

T 5.3.1. Análise bibliográfica de métricas existentes para a avaliación da productividade.

T 5.3.2. Avaliación da productividade mediante o deseño dun conxunto de medidas.

B 6. Conclusións e análise das futuras liñas de traballo.

O 6.1. Exposición das principais leccións aprendidas.

T 6.1.1. Resumo do traballo feito, principais aportacións e conclusións.

T 6.1.2. Análise das posibles liñas de traballo futuro.

O 6.2. Elaboración da memoria final da Tese de Doutoramento.

T 6.2.1. Estructuración e organización dos informes do traballo realizado.

T 6.2.2. Redacción da memoria.

Medios

Para a elaboración desta Tese de Doutoramento utilizáronse os medios descritos a continuación:

- Material de traballo e financiamento económico proporcionados polo Grupo de Arquitectura de Computadores da Universidade da Coruña, o Ministerio de Educación (bolsa predoutoral FPU AP2009-2112) e a Universidade da Coruña (contrato de profesor axudante).
- Proxectos de investigación que financiaron esta Tese:
  - Con financiamento europeo: European Network of Excellence on High Performance and Embedded Architecture and Compilation HiPEAC-2 (7° PM, ICT-217068), HiPEAC-3 (7° PM, ICT-287759) e Open European Network for High Performance Computing on Complex Environments (ComplexHPC, COST Action ref. IC0805).
  - Con financiamento estatal: proxectos do Plan Nacional de I+D “Arquitecturas, sistemas y herramientas para computación de altas prestaciones” (TIN2010-16735) e “Soporte hardware y software para computación de altas prestaciones” (TIN2007-67537-C03-02).
  - Con financiamento autonómico: Programa de Consolidación e Estructuración de Unidades de Investigación Competitivas da Xunta de Galicia, na modalidade de Grupos de Referencia Competitiva (Grupo de Arquitectura de Computadores, refs. 2010/6 e 2006/3) e na modalidade de Redes

de Investigación (Rede Galega de Computación de Altas Prestacións, refs. 2010/53 e 2007/147).

- Con financiamento privado: proxecto High Performance Computing for High Performance Trading (HPC4HPT), financiado pola Fundación Barrié, e o proxecto FastMPJ Cloud, financiado por Amazon mediante unha AWS Research Grant.

- Clusters e supercomputadores utilizados (detállanse só os recursos utilizados de cada sistema):

  - Clúster *pluton* (Grupo de Arquitectura de Computadores da Universidade da Coruña), 8 nodos con 2 procesadores Intel Xeon E5520 de 4 núcleos a 2.27 GHz con ata 8 GB de RAM e rede de interconexión InfiniBand; e 16 nodos con 2 procesadores Intel Xeon E5-2660 de 8 núcleos a 2.20 GHz con 64 GB de RAM, GPUs NVIDIA K20m e coprocesadores Intel Xeon Phi 5110P.
  - Clúster *DAS-4* (Advanced School for Computing and Imaging, ASCI, Vrije University Amsterdam), formado por recursos de diversas institucións holandesas. Para a Tese, utilizouse un nodo con procesador AMD Magny-Cours de 48 núcleos e 128 GB de RAM, e ata 16 nodos con 2 procesadores Intel Xeon E5620 de 4 núcleos a 2.40 GHz con ata 24 GB de RAM e rede de interconexión InfiniBand.
  - Supercomputador *Finis Terrae* (Centro de Supercomputación de Galicia): 144 nodos con procesador Itanium2 Montvale de 16 núcleos a 1.6 GHz con 128 GB de RAM e rede de interconexión InfiniBand, e 1 nodo Superdome con procesador Itanium2 Montvale de 128 núcleos a 1.6 GHz con 1 TB de RAM.

- Estancia de tres meses na ETH Zürich no Scalable Parallel Computing Lab do profesor Torsten Hoefler, que favoreceu a colaboración no desenvolvemento da librería de colectivas non bloqueantes e o estudo de algoritmos de operacións colectivas para memoria compartida, así como a profundización na arquitectura do coprocesador Intel Xeon Phi. Esta estancia estivo financiada por unha Collaboration Grant da rede HiPEAC-3 no 2012.

Conclusións

Esta Tese de Doutoramento, *"High Performance Java for Multi-core Systems"*, presenta unha análise da adecuación de Java para a programación de altas prestacións en arquitecturas multi-núcleo, así como un estudo das principais arquitecturas de memoria compartida dispoñibles, proporcionando a implementación dun middleware de paso de mensaxes en Java para a programación paralela de sistemas multi-núcleo e unha librería de operacións colectivas optimizadas, tanto bloqueantes coma non bloqueantes. Ademais, realizouse unha análise detallada das posibilidades actuais de programación orientada a sistemas heteroxéneos utilizando Java.

Como principal conclusión extraída, Java permite obter un alto rendemento en sistemas de memoria compartida mediante o uso da API de multithreading. Pero, pese a elevada productividade que proporciona a linguaxe Java en xeral, a API de multithreading é complicada e propensa a erros, e o manexo da sincronización pode dar lugar a códigos ineficientes ou, o que é peor, inconsistentes. Por outra banda, aínda que existe a posibilidade de utilizar ferramentas de concorrencia de alto nivel incluídas na linguaxe, estas están orientadas a obter un elevado largo de banda e non baixa latencia, esixindo a restructuración de algoritmos en base a tarefas. A solución proposta nesta Tese pasa por aproveitar o soporte para multithreading, que permite explotar os sistemas multi-núcleo eficientemente, para desenvolver ferramentas e librerías de alto rendemento que proporcionen interfaces sinxelas manexando, de forma transparente ao usuario, a API de threads.

Ademais, as optimizacións desenvolvidas, tanto no middleware de paso de mensaxes coma na librería de operacións colectivas, mostran que á hora de intentar maximizar o rendemento das aplicacións Java non podemos centrarnos únicamente nas características do hardware. Tamén é necesario ter en conta as peculiaridades da JVM (Java Virtual Machine), xa que os costes de inicialización ou a falta de compilación poden provocar un aumento da latencia que non compense a mellora obtida mediante técnicas tradicionais de aproveitamento do hardware. Non obstante, a JVM presenta avances neste aspecto que permiten certa optimización do rendemento tendo en conta o hardware, como por exemplo o mapeo de threads da JVM a threads do sistema operativo, o que permite ter en conta a afinidade dos threads a núcleos específicos nun código Java.

Finalmente, aínda que os fabricantes non proporcionan soporte específico en Java para arquitecturas many-core, existen librerías que permiten programar en Java baseándose en soportes nativos que combinan a alta eficiencia coa productividade. Non obstante, existe unha diferencia de rendemento con respecto a solucións nativas debido á falla de soporte directo destas arquitecturas en Java.

En canto ás liñas de traballo futuro, en primeiro lugar estaría a análise dos beneficios e principais problemas do soporte Java directo para arquitecturas many-core. Xa que o soporte para GPUs depende principalmente dos fabricantes e dos responsables do desenvolvemento das JVMs, é de esperar que, xa que o Intel Xeon Phi presenta unha arquitectura x86, nun futuro cercano será posible executar máquinas virtuais Java neste coprocesador. Ademais, tanto as operacións colectivas coma o dispositivo de paso de mensaxes en Java para sistemas multi-núcleo poderían optimizarse incorporando unha maior caracterización hardware ou parametrización nos algoritmos e implementando un sistema de selección de algoritmos en tempo de execución para as operacións colectivas máis preciso e mediante o modelado das interaccións de threads. Outra liña interesante sería unha exploración da posibilidade de implementar un dispositivo de comunicacións híbrido de memoria compartida-distribuída. Aínda que esta posibilidade foi analizada en [109] a partir do soporte para comunicacións en memoria compartida incluido na librería de paso de mensaxes MPJ Express [118], esta análise estivo limitada polos problemas de eficiencia na sincronización de MPJ Express e pola falta de códigos adaptados á explotación do rendemento de arquitecturas clúster multi-núcleo (é dicir, minimizando as comunicacións inter-nodo).

Principais Contribucións

As principais contribucións desta Tese de Doutoramento son:

- Análise do estado da arte de Java para computación de altas prestacións [121].
- Implementación dunha solución eficiente e escalable de comunicacións en memoria compartida utilizando Java e proporcionando unha API de paso de mensaxes [108, 109].

- Optimización de operacións colectivas bloqueantes para memoria compartida e clusters de procesadores multi-núcleo [122].
- Implementación de colectivas non bloqueantes en Java, e estudo do rendemento en sistemas de memoria compartida [107].
- Estudo do rendemento e productividade das solucións dispoñibles para programación Java en contornos heteroxéneos utilizando arquitecturas many-core [25, 106].

Publications from the Thesis

Journal Papers (8)

- S. Ramos, G. L. Taboada, R. R. Expósito, and J. Touriño. Nonblocking Collectives for Scalable Java Communications, 2013 (Submitted for journal publication).

- R. R. Expósito, S. Ramos, G. L. Taboada, J. Touriño, and R. Doallo. FastMPJ: a Scalable and Efficient Java Message-Passing Library, 2013 (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*, 2013 (In press).

- 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*, 2013 (In press).

- S. Ramos, G. L. Taboada, R. R. Expósito, J. Touriño, and R. Doallo. Design of Scalable Java Communication Middleware for Multi-Core Systems. *The Computer Journal*, 56(2): 214–228, 2013.

- 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.

- 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.

- G. L. Taboada, S. Ramos, J. Touriño, and R. Doallo. Design of Efficient Java Message-passing Collectives on Multi-core Architectures. *Journal of Supercomputing*, 55(2): 126–154, 2011.

International Conferences (3)

- S. Ramos, T. Hoefler. Modeling Communication in Cache-Coherent SMP Systems - A Case-Study with Xeon Phi. In *Proc. 22nd ACM Intl. Symposium on High-Performance Parallel and Distributed Computing (HPDC'13)*, pages 97–108, New York City, NY, USA, 2013.

- J. Docampo, S. Ramos, G. L. Taboada, R. R. Expósito, J. Touriño, and R. Doallo. Evaluation of Java for General Purpose GPU Computing. In *Proc. Intl. Workshop on Engineering Object-Oriented Parallel Software (EOOPS'13)*, pages 1398–1404, Barcelona, Spain, 2013.

- S. Ramos, G. L. Taboada, J. Touriño, and R. Doallo. Scalable Java Communication Middleware for Hybrid Shared/Distributed Memory Architectures. In *Proc. 13th Intl. Conf. on High Performance Computing and Communications (HPCC'11)*, pages 221–228, Banff, Alberta, Canada, 2011.

National Conferences (3)

- J. Docampo, S. Ramos, G. L. Taboada, R. R. Expósito, J. Touriño, and R. Doallo. Evaluación de Java para Computación de Propósito General en GPU. In *Proc. XXIV Jornadas de Paralelismo*, Madrid, Spain, 2013.

- S. Ramos, G. L. Taboada, J. Touriño, and R. Doallo. Comunicaciones Escalables en Memoria Compartida para Paso de Mensajes en Java. *In Proc. XXII Jornadas de Paralelismo*, pages 439–444, La Laguna, Tenerife, Spain, 2011.

- S. Ramos, G. L. Taboada, J. Touriño, and R. Doallo. Biblioteca de Primitivas Colectivas de Paso de Mensajes para Java en Sistemas Multicore. *In Proc. XX Jornadas de Paralelismo*, pages 511–516, A Coruña, Spain, 2009.

Abstract

The interest in Java within the High Performance Computing (HPC) community has been rising during the last years thanks to its noticeable performance improvements and its productivity features. In a context where the trend to increase the number of cores per processor is leading to the generalization of many-core processors and accelerators, multithreading as an inherent feature of the language makes Java extremely interesting to exploit the performance provided by multi- and many-core architectures. This PhD Thesis presents a thorough analysis of the current state of the art regarding multi- and many-core programming in Java and provides the design, implementation and evaluation of several solutions to enable Java for the many-core era. To achieve this, a shared memory message-passing solution has been implemented to provide shared memory programming with the scalability of distributed memory paradigms, also with the benefits of a portable programming model that allows the developed codes to be run on distributed memory systems. Moreover, representative collective operations, involving computation and communication among different processes or threads, have been optimized, also introducing in Java new features for scalability from the MPI 3.0 specification, namely nonblocking collectives. Regarding the exploitation of many-core architectures, the lack of direct Java support forces to resort to wrappers or higher-level solutions to translate Java code into CUDA or OpenCL. The most relevant among these solutions have been evaluated and thoroughly analyzed in terms of performance and productivity. Guidelines for taking advantage of shared memory environments have been derived during the analysis and development of the proposed solutions, and the main conclusion is that the use of Java for shared memory programming on multi- and many-core systems is not only productive but also can provide high performance competitive results. However, in order to effectively take advantage of the underlying multi- and many-core architectures, the key is the availability of optimized middleware that abstracts multithreading details from the user, like the one proposed in this Thesis, and the optimization of common operations like collective communications.

Aos que o fixestes posible.

Acknowledgments

I would like to start saying thank you to Guillermo and Juan for their valuable help and support during the elaboration of this PhD, and, of course, to all my colleagues in the Computer Architecture Group, both the ones from the ground floor and those from the upper floor. I do not want to leave anybody, but I have to make a special mention to Jorge, Jose, Raquel, CH, Toño, Pablo, Iván, Jacobo, Moisés, Rober, Dani and Sasi.

Of course, I do not want to forget other people that have also been there supporting me during these years, especially my parents and my sister Elena, Juan, and all my closest friends.

I gratefully thank Dr. Torsten Hoefler for hosting me during my visit to the Scalable Parallel Computing Laboratory at the ETH Zürich, Switzerland, and the CSCS people for their efficient and useful support.

Moreover, I want to thank the Supercomputing Center of Galicia (CESGA), and the Advanced School for Computing and Imaging (ASCI) and Vrije University Amsterdam, for the access to their computational resources (Finis Terrae supercomputer and DAS-4 cluster, respectively).

Last, but not least, I want also to express my gratitude to the following institutions for providing financial support for this work: the Computer Architecture Group, the Department of Electronics and Systems and the University of A Coruña for the human and material support; the HiPEAC network (EU FP7 ICT-217068 and ICT-287759) and its collaboration grants program, 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 Ministry of Education and Culture and the Ministry of Economy and Competitiveness of Spain for the projects TIN2007-67537-C03-02 and TIN2010-16735, and the FPU Grant AP2009-2112.

Sabela Ramos Garea

It is not the task of the University to offer what society asks for,
but to give what society needs.

(Non é labor da Universidade ofrecer o que a sociedade pide,
senón o que a sociedade necesita)

Edsger W. Dijkstra.

Contents

Preface **1**

1. Java for High Performance Computing **5**

1.1. Parallel Programming in Java 6

1.1.1. High Performance Utilities in Java 8

1.2. Message Passing in Java (MPJ) 8

1.3. Heterogeneous Programming in Java 11

2. Shared Memory Architectures **15**

2.1. Multi-core Architectures 15

2.1.1. Thread Affinity Control in Java 19

2.2. Many-core Architectures 19

2.2.1. Many-core x86 Architectures: Intel Xeon Phi 20

2.2.2. Graphics Processing Units (GPUs) 22

2.3. Java Support for Shared Memory Architectures 25

3. A Shared Memory Communication Device for MPJ **27**

3.1. State of the Art of Message Passing for Shared Memory 28

3.2. Design and Implementation of the Shared Memory Device `smdev` . . . 29

3.2.1. Low-level Message-Passing API: `xxdev` 30

3.2.2. Class Loading in `smdev` 31

3.2.3. Message Queues 32

3.2.4. Message Transfer Protocols 33

3.2.5. Synchronization 39

3.2.6. Integration of `smdev` in FastMPJ 40

3.3. Performance Evaluation 40

3.3.1. Experimental Configuration 40

3.3.2. Point-to-point Micro-benchmarking 41

3.3.3. Impact of `smdev` on the Scalability of Parallel Codes 48

3.4. Main Contributions of the `smdev` Device 51

4. Efficient Support of Collective Communications in Java 53

4.1. Blocking Collectives for Multi-core Systems 54

4.1.1. State of the Art of MPJ Collectives 56

4.1.2. Multi-core Aware Collectives 58

4.1.3. Performance Evaluation 68

4.2. Nonblocking Collectives in Java 83

4.2.1. State of the Art of Nonblocking Collectives 84

4.2.2. Nonblocking MPJ Collectives 86

4.2.3. Performance Evaluation 93

4.3. Main Contributions of the MPJ Collectives Support 100

5. Java Heterogeneous Computing 103

5.1. General Purpose GPU Computing in Java 103

5.1.1. Experimental Configuration 104

5.1.2. Analysis of Experimental Results 105

5.2. Many-core x86-based Computing in Java 110

5.2.1. Experimental Configuration 110

5.2.2. Analysis of Experimental Results 111

5.3. Analysis of Productivity of Java Heterogeneous Computing Codes . . 115

5.3.1. Characterization of the Manual Effort 117

5.3.2. Characterization of the Conceptual Effort 118

5.4. Lessons Learned from Java Heterogeneous Computing 122

Conclusions and Future Work **123**

References **127**

List of Tables

1.1. Available solutions for GPGPU computing in Java 13

3.1. MOPS of NPB codes on a single core 49

4.1. Algorithms implemented in the MPJ collectives library 62

4.2. Algorithms that maximize collectives performance on the DAS-4 multi-core cluster 70

4.3. Latency of Barrier algorithms (time in μseconds) 83

5.1. Selected kernels for Java GPGPU performance analysis 105

5.2. MaxFlops performance on the NVIDIA K20 GPU (in GFLOPS) . . . 106

5.3. MaxFlops performance on the Intel Xeon Phi (in GFLOPS) 111

5.4. Manual effort for the development of the Aparapi and jCuda implementations of representative SHOC kernels 117

5.5. Conceptual effort in MaxFlops 120

5.6. Conceptual effort in matrix multiplication (GEMM) 120

5.7. Conceptual effort in Stencil2D 121

5.8. Conceptual effort in FFT 121

List of Figures

1.1. FastMPJ communication devices on shared memory and cluster networks . . . 11

2.1. Architecture of an Intel Xeon E5-2670 Sandy Bridge-based system . . 17

2.2. Architecture of a Magny-Cours AMD Opteron 6172-based system . . 18

2.3. Architecture of the Intel Xeon Phi coprocessor . . . 22

2.4. Architecture of the Kepler GK110 SMX, basic block of the NVIDIA K20 . . . 24

3.1. Java communications on a dual-core dual processor using distributed (left) and shared (right) memory-oriented middleware . . . 30

3.2. Communication protocols in `smdev` . . . 34

3.3. Send/Recv operations in `smdev` . . . 36

3.4. `smdev` performance on the Xeon E5 . . . 42

3.5. `smdev` performance on the Magny-Cours . . . 44

3.6. Message Passing in Java performance on the Xeon E5 . . . 45

3.7. Message Passing in Java performance on the Magny-Cours . . . 45

3.8. Sockets performance on the Xeon E5 . . . 46

3.9. Sockets performance on the Magny-Cours . . . 46

3.10. Memory performance on the Xeon E5 47
3.11. Memory performance on the Magny-Cours 48
3.12. NAS Parallel Benchmarks performance on the Xeon E5 50
3.13. NAS Parallel Benchmaks performance on the Magny-Cours 51

4.1. Overview of the Broadcast MST algorithm 60
4.2. Overview of the Gather MST algorithm 61
4.3. Overview of the Allgather BDE algorithm 61
4.4. Overview of the Allgather BKT algorithm 62
4.5. Threaded Broadcast 65
4.6. MPJ Broadcast and Allgather performance on the DAS-4 multi-core cluster 71
4.7. MPJ Reduce and Allreduce performance on the DAS-4 multi-core cluster 72
4.8. Scalability of MPJ collectives for a 1-KByte message 73
4.9. Scalability of MPJ collectives for a 32-KByte message 74
4.10. Scalability of MPJ collectives for a 1-MByte message 75
4.11. Scalability of NPB kernels on the DAS-4 multi-core cluster 77
4.12. Performance (in MOPS) of NPB kernels on the DAS-4 multi-core cluster 78
4.13. Broadcast performance on 8 and 16 cores (Xeon E5) 80
4.14. Broadcast performance on 8 and 48 cores (Magny-Cours) 81
4.15. Flat Tree-based Blocking and Nonblocking Broadcast implementation 87
4.16. MST-based Blocking and Nonblocking Broadcast Implementation . . 89
4.17. Shared memory Broadcast: Blocking vs. Nonblocking 94
4.18. MST Broadcast vs. Shared memory Nonblocking Broadcast 95

4.19. *nb*FT Broadcast vs. Shared memory Nonblocking Broadcast 95
4.20. Shared memory Scatter: Blocking vs. Nonblocking 96
4.21. *nb*FT Scatter vs. Shared memory Nonblocking Scatter 97
4.22. Performance comparison of an I/O-intensive MPJ application using Blocking ("block") and Nonblocking ("nbc") collectives 99

5.1. Matrix multiplication (GEMM) kernel performance on the NVIDIA K20 GPU 107
5.2. Stencil2D kernel performance on the NVIDIA K20 GPU 108
5.3. FFT kernel performance on the NVIDIA K20 GPU 109
5.4. Matrix multiplication (GEMM) kernel performance on the Intel Xeon Phi 112
5.5. Stencil2D kernel performance on the Intel Xeon Phi 113
5.6. FFT kernel performance on the Intel Xeon Phi 114

Preface

Introduction and Motivation

The Java language has become an interesting choice for High Performance Computing (HPC) due to its significant performance improvements and the features that make it a highly productive language. Among them, it is worth mentioning portability and platform independence, simplicity, robustness, security, object orientation and having a large development community both in industry and academia. Besides performance improvements, there are two crucial properties that favor the adoption of Java for HPC: the network and the multithreading support in the core of the language. These qualities make Java a perfect candidate for parallel programming.

Among current parallel environments, shared memory systems have become very popular due to the physical limits that performance scaling reached in uni-core processors, with dissipation issues and lack of energy efficiency. To overcome these drawbacks, hardware manufacturers increased performance by including more cores per processor, leading to the generalization of multi-core processors. In fact, processors with four or more cores are getting popular in commodity personal computers.

Furthermore, the increase in the number of cores is leading to the spread of many-core processors for general purpose computation. In this area, again, energy consumption emerges as an important concern because the large number of cores causes these processors to be inefficient when only a small amount of cores is needed. To deal with this issue, specific purpose accelerators, such as Graphics Processing Units (GPUs), appeared to offload specific parallel tasks while a host processor tackles general purpose computation. In the case of GPUs, their features and high compute capabilities made them also suitable for streaming programming,

not necessarily related to graphics processing. However, programming difficulties associated with the peculiarities of these accelerators favored the birth of x86-based coprocessors, such as the Intel Xeon Phi, that can be programmed with traditional paradigms and languages.

The success of multi- and many-core architectures points out the need for languages and tools to exploit shared memory performance, where Java, with its intrinsic multithreading support, has a tremendous advantage over other languages. The main drawback is that the threading API is difficult to manage, making the user responsible for dealing with the scheduling of threads, and race conditions and inconsistencies may arise. As a potential solution, Java provides a high-level concurrency library to work with thread pools and oriented to task programming or to a fork/join approach, with load balancing achieved by work-stealing techniques. These paradigms are throughput oriented and parallel algorithms have to be rewritten or programmed directly with the basic threading API.

This PhD Thesis presents a detailed analysis of the state of the art regarding the use of Java in shared memory systems programming, focusing on suitable HPC solutions. The main goals of this work are the analysis of the Java support for shared memory HPC programming and the development of middleware to improve performance and productivity. To achieve this, a thorough review of available libraries for multi-core programming has been carried out. As a result, the Thesis presents the design, implementation and evaluation of a Java message-passing communication device optimized for shared memory. This device provides a high-level message-passing API avoiding the need for managing threads or using a task-based approach. This API follows the Message Passing in Java (MPJ) specification, based on the MPI standard widely used in HPC. Moreover, the Thesis includes an optimization of communication operations among processes or threads (collective operations) for multi-core environments, also providing nonblocking collectives support and a thorough analysis regarding the feasibility of nonblocking collectives for shared memory environments. In addition, this Thesis presents an evaluation of currently available solutions to exploit many-core architectures using Java. The main conclusion is that the use of Java in shared memory environments is highly productive and can provide high performance results, but it is necessary to provide optimized middleware that abstracts multithreading details from the user, like the one proposed in the Thesis,

and the optimization of widely used operations like collective communications.

Main Contributions of the Thesis

The main original contributions derived from the Thesis are the following:

- Analysis of the state of the art regarding Java for High Performance Computing (HPC) [121].
- Implementation of an efficient and scalable communication solution for shared memory systems using Java and supporting a message-passing API [108, 109].
- Optimization of blocking collective operations for shared memory and multi-core clusters [122].
- Implementation of nonblocking message-passing collectives in Java and performance study on a shared memory system [107].
- Performance and productivity evaluation of available solutions for Java programming in heterogeneous environments with many-core architectures [25, 106].

Structure of the Thesis

The Thesis is organized in the following chapters:

- Chapter 1, Java for High Performance Computing, summarizes the state of the art regarding Java HPC programming for multi- and many-core systems. Different parallel programming frameworks and tools are studied, focusing on message passing, due to its scalability and extended use in HPC. Moreover, it provides an overview of current heterogeneous programming solutions for Java.

- Chapter 2, Shared Memory Architectures, analyzes some of the most popular shared memory architectures. First, it provides some insight on architectural aspects of two multi-core processors from the two main manufacturers, Intel and AMD. Then, it goes through similar aspects regarding many-core systems, analyzing the last NVIDIA GPU architecture and the recently released x86-based Intel Xeon Phi.

- Chapter 3, A Shared Memory Communication Device for Message Passing in Java, presents the design, implementation and performance evaluation of a shared memory communication device for message passing in Java. This device provides efficient communications relying on shared memory transfers among threads, instead of processes, using zero-copy protocols and minimizing synchronizations.

- Chapter 4, Efficient Support of Collective Communications in Java, includes a collectives library with blocking and nonblocking operations support and optimizations for multi-core systems. Blocking collectives are optimized not only for shared memory systems, but also for multi-core clusters, exploring the combination of process communication across the network and the use of threads within each node. Moreover, the nonblocking support assesses the feasibility of using nonblocking collectives in Java for shared memory programming.

- Chapter 5, Java Heterogeneous Computing, studies the currently available solutions to program many-core accelerators with Java. It provides a thorough analysis of efficiency and productivity of current projects that allow us to take advantage of these accelerators using Java.

- Finally, the Thesis presents the main conclusions, guidelines and future work derived from the research carried out.

Chapter 1

Java for High Performance Computing

Java is the leading programming language both in academia and industry environments. The success of Java is motivated by its appealing features such as built-in networking and multithreading support, automatic memory management, platform independence, portability, security, object orientation, an extensive API and a wide community of developers. Moreover, it is increasingly being adopted by the High Performance Computing (HPC) community [121] due to its improvements in performance, which makes it competitive in comparison with natively compiled languages like C/C++, enabling the use of Java in performance-bounded scenarios.

The physical limits in frequency scaling has favored the trend to increase the number of available cores per processor and the use of specific accelerators and co-processors to improve performance and energy efficiency. This leads to the need for scalable parallel programming paradigms to exploit the characteristics of the underlying hardware in order to reflect the performance improvements in real applications. Although this is not a new situation since clusters of processors are widely used to build more powerful computers that meet the needs of the HPC community, the particularities of multi-core processors have to be addressed not only in an isolated manner, but also to increase performance on clusters of multi-core processors (e.g., heterogeneity of performance for communications within cores on the same processor and on different processors).

This chapter presents an analysis of the state of the art of HPC programming in Java focusing on solutions for shared memory multi- and many-core systems.

1.1. Parallel Programming in Java

Current trends in hardware evolution make it unavoidable to any HPC language to provide parallel tools that allow programmers to exploit the underlying architectures. Multithreading support in the core of the Java language makes it inherently parallel which, combined with the built-in networking, turns Java into a more than suitable choice for the development of projects that aim to take advantage of parallel architectures [121].

This multithreading support allows Java to exploit shared memory architectures without having to resort to external projects or libraries. Nevertheless, its threading API generally requires low-level programming skills. The concurrency framework, included in the core of the language since Java 1.5, simplifies the management of threads hiding part of the complexity and providing a task-oriented programming paradigm based on thread pools. However, the task management targets the scheduling of a high number of tasks instead of reducing the task start-up time (the initialization overhead). Moreover, it is limited to the execution in parallel of individual tasks, so the developer has to resort to threads for high performance parallel codes, where threads cooperate to reduce the runtime of a workload. Java 1.7 extends the concurrency framework by including the fork/join utilities developed by Doug Lea [70] to favor parallel programming oriented to the fragmentation of a complex problem into recursive tasks following the divide-and-conquer strategy. The fork/join structure is based on work-stealing techniques that automatically balance the workload among the available threads.

However, codes developed using threads or tasks cannot run on distributed memory environments, as it happens for the traditional approach followed in compiled languages, such as C/C++, with the use of shared memory models like POSIX threads (Pthreads) or OpenMP directives. In order to overcome this limitation, natively compiled languages resort to several tools that execute multithreaded applications on distributed memory architectures but, up to now, either their imple-

mentation is based on software translations to message-passing models like MPI [11] or it relies on Distributed Shared Memory (DSM) systems [84]. Another option is the use of a hybrid shared/distributed memory programming model combining MPI for inter-node communications and a shared memory model to take advantage of intra-node parallelism [143]. Additionally, new programming paradigms such as PGAS (Partitioned Global Address Space) arise for programming hybrid shared/distributed memory systems, although generally their performance is lower than MPI [78]. In Java, the use of Java DSM implementations generally involves portability issues due to the need for modified Java Virtual Machines (JVMs). The Parallel Java project [58] provides several abstractions over the concurrency utilities, also implementing the message-passing paradigm for distributed memory but with its own interface instead of a standard one like MPI. There are also OpenMP-like Java implementations such as JOMP [16] and JaMP [63]. Both systems are "pure" Java and thread-based, but the second one also takes advantage of concurrency utilities overcoming some efficiency problems of JOMP. JaMP is part of Jackal [141], a software-based Java DSM implementation, and its main drawback is the lack of portability since it cannot run on standard JVMs.

Java communication middleware, such as Java Message Service (JMS) and Remote Method Invocation (RMI), always resort to JVM sockets, which currently have two implementations: the standard I/O sockets (the counterpart of the widely available POSIX sockets), and the New I/O (NIO) sockets, an implementation focused on the scalability of communications in servers introduced in Java 1.4. However, programming with sockets requires a significant effort due to their low-level API. Moreover, performance is generally limited as sockets rely on TCP/IP. In order to overcome these limitations, parallel programmers generally develop their codes using message-passing libraries, which provide a higher-level API, scalability and relatively good performance.

Another project, recently released, that provides high performance capabilities in Java is Disruptor [132], based on a cyclic queue (or ring buffer) and a consumer-provider programming paradigm. Disruptor aims to provide a low-latency and high-throughput solution for data interchange among threads, taking into account low-level features such as cache sizes and lock-free strategies.

1.1.1. High Performance Utilities in Java

There are also other tools that support HPC programming although they are not intrinsically parallel. The best example are mathematical libraries, very useful especially on scientific environments. Although it has been proven that Java is able to compete with Fortran in high performance numerical computing [12, 54, 85], the development of a competitive numerical Java library is still an ongoing effort. In fact, in [7] the authors evaluate Java for numerical computing showing that the performance of Java can be significantly enhanced by delegating numerically intensive tasks to native libraries such as Intel MKL. There are some active projects that tackle different numerical operations, such as the Universal Java Matrix Package (UJMP) [5, 140], the Efficient Java Matrix Library (EJML) [27], the Matrix Toolkits Java (MTJ) [81], the Java Algebra System (JAS) Project [64, 65] and jblas [74], which are replacing more traditional frameworks like JAMA [52].

Another useful tool for HPC is the data management support in terms of collections and optimizations of common operations like insertion, sort, deletion, etc. In this field, there are several projects that attempt to provide with high performance collections, such as the High Performance Primitive Collections for Java (HPPC) [47] or TROVE [137]. However, some of them have programmability or thread-safety limitations.

1.2. Message Passing in Java (MPJ)

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 in distributed memory systems such as clusters, and it is becoming popular also for shared memory architectures with a high number of cores or threads, due to its scalability, flexibility and interesting cost/performance ratio.

MPI is the standard message-passing interface for languages compiled to native code (e.g., C and Fortran). Regarding Java, there have been several implementations of message-passing libraries from its inception [121]. Although initially each project

developed its own MPI-like binding for the Java language, current projects generally adhere to one of the two main proposed APIs: (1) the mpiJava 1.2 API [20], which supports an MPI C++–like interface for the MPI 1.1 subset, and (2) the JGF MPJ API [8], which is the proposal of the Java Grande Forum (JGF) [53] to standardize the MPI-like Java API. The collective communication primitives are essential part of the different MPJ APIs, both in terms of number of methods and widespread use.

MPJ libraries can be implemented in two ways: (1) wrapping an underlying native messaging library like MPI through the Java Native Interface (JNI); or (2) using a "pure" Java (100% Java) approach, based on RMI or sockets. Each solution fits with specific situations, but presents associated trade-offs. On the one hand, the use of the pure Java approach ensures portability, but it might not be the most efficient solution, especially in the presence of high-speed communication hardware and when using RMI or JMS, as these technologies are oriented to distributed computing on loosely coupled peers and show high start-up latencies. On the other hand, the use of JNI has portability problems, although usually in exchange for higher performance.

The mpiJava library [9] is a wrapper implementation which provides efficient communication resorting to an underlying native MPI library, adding a reduced JNI overhead. However, despite its usually high performance, mpiJava currently only supports some native MPI implementations, as wrapping a wide number of functions (especially the collectives) and heterogeneous runtime environments entails an important maintenance 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 project maintenance has been superseded by the development of MPJ Express [116], 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 NIO communications (`niodev` device) with a high performance Myrinet support (through the native Myrinet eXpress –MX– communication library in the `mxdev` device), and a specific device for shared memory. Furthermore, this project is the most active in terms of adoption by the HPC community, presence in academia and production environ-

ments, and available documentation. The project is also stable and publicly available along with its source code at `http://sourceforge.net/projects/mpjexpress/`.

There are additional MPJ implementations, such as MPJ/Ibis [13], based on the JGF API. This library supports “pure” Java and native communications on Myrinet. In addition, they use two low-level communication devices based on Ibis: TCPIbis, based on Java IO sockets (TCP/IP), and NIOIbis, which provides blocking and nonblocking communications through Java NIO sockets. Nevertheless, it lacks thread safety and provides limited performance.

The increasing interest in Java is also evidenced by the recent efforts of Open MPI [128, 129], one of the main open source MPI projects, that has announced its Java interface motivated by a request of the Hadoop community. While it is still bounded to the MPI 1.2 API (using the mpiJava 1.2 API and the mpiJava library bindings), they plan to extend it to the full MPI 3.0 specification. Although this Java support could benefit from a large community of users and developers, and from the highly optimized MPI support of Open MPI, it is still a wrapper to a native implementation with no pure Java support, thus lacking portability and with instability issues that has put off its release until these issues are solved.

The most recent “pure” Java MPJ project is our FastMPJ library [29, 123], with a modular design that can be seen in Figure 1.1. It is similar to MPJ Express, which has a pluggable architecture of communication devices that allows to combine the portability of the “pure” Java communication devices with high performance network support wrapping native communication libraries through JNI. Figure 1.1 shows the communication support implemented in FastMPJ, either on JVM threads (`smdev`), sockets over the TCP/IP stack (`niodev` and `iodev`), or on native communication layers such as Open-MX (`mxdev`), InfiniPath PSM (`psmdev`) and InfiniBand Verbs (IBV) (`ibvdev`), which are accessed through JNI. The main advantages over MPJ Express are the more stable runtime framework and higher performance and scalability.

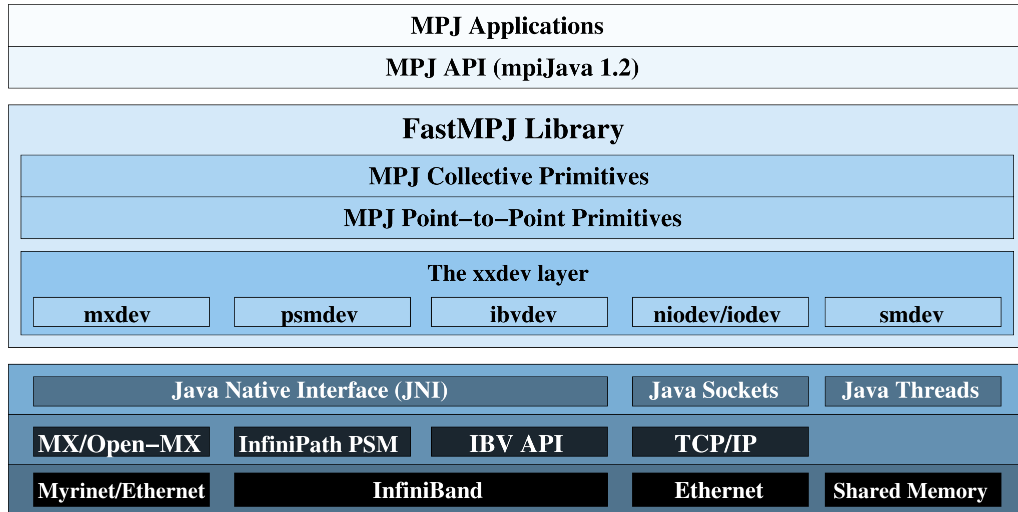


Figure 1.1: FastMPJ communication devices on shared memory and cluster networks

1.3. Heterogeneous Programming in Java

Besides multi-core processors, another successful trend in hardware development is the inclusion of coprocessors to increase the performance of specific regions of parallel codes. Most of them are composed by a large number of small and specialized cores, such as the Graphics Processing Units or GPUs. These accelerators provide high performance and energy efficiency since they are external units only used when necessary.

The most popular coprocessors are the GPUs, with a heavy and increasing presence in the Top 500 list of supercomputers [134]. The massively parallel architecture of the GPU, together with its floating point capability, has motivated the growth of GPGPU (General Purpose computing on GPU) [71], along with different programming models, such as Compute Unified Device Architecture (CUDA) [91] or Open Computing Language (OpenCL) [120], to enable the use of GPUs as many-core accelerators for non-graphics workloads. Hence, the adoption of GPUs as accelerators in HPC environments [34] is increasing, since many scientific applications present a huge degree of parallelism that can take advantage of GPU's features.

The recently released Intel Xeon Phi, from the Intel MIC family, aims to share this popularity by providing many-core coprocessors with a x86 architecture to en-

able the use of traditional programming languages and paradigms to exploit its performance. As the Xeon Phi accelerator has just been released, there is no Java support yet, although given its architecture it should be possible to have JVMs running in this system in the short term. Moreover, it supports traditional shared memory programming paradigms, including OpenCL, allowing it to be used as a mere coprocessor with OpenCL accessed via JNI.

Unfortunately, this situation is not exclusive of the Xeon Phi accelerator. GPGPU programming models are provided as libraries and intended to be used as C/C++ extensions, whereas languages like Java must resort to wrappers (via JNI) to be able to take advantage of GPUs as accelerators. This has motivated the growth of several projects that aim to ease Java GPGPU programming by providing frameworks to deal with C/C++ extensions. Moreover, there is an ongoing effort within the OpenJDK community, together with Oracle and AMD, to include GPGPU support directly in the JVM [96].

Among the Java GPGPU projects, we can distinguish two approaches: the ones that provide Java bindings to a lower-level language (CUDA or OpenCL), or those with a user-friendly API that abstracts GPU programming along with a runtime system which translates Java bytecode into CUDA or OpenCL in a transparent manner. While Java bindings are meant to provide better performance, the second approach makes it possible to find a trade-off between performance and productivity.

Table 1.1 summarizes the most relevant projects for GPGPU computing in Java classified by the underlying native library used. Among the CUDA-related projects, JCUDA [144] has its own interface to invoke certain CUDA functions and user developed kernels. Nevertheless, it is not included in the “User-friendly projects” group since it still requires low-level programming skills and certain knowledge of CUDA functions.

jCuda [55] is the most active Java GPGPU project. It provides a direct wrapper over CUDA 4.2 runtime and driver API, allowing the direct interaction with the device, including memory management, and providing support to launch CUDA kernels from Java. The main strength of this project is that it provides support for several optimized libraries from CUDA like CUBLAS (CUDA Basic Linear Algebra Subprograms), CUFFT (Fast Fourier Transforms), CUDPP (Data Parallel Prim-

Table 1.1: Available solutions for GPGPU computing in Java

	Java bindings	**User-friendly projects**
CUDA	JCUDA [144] jCuda [55]	Java-GPU [18] Rootbeer [104]
OpenCL	JOCL [56] JogAmp JOCL [57]	Aparapi [4]

itives), CURAND (Random Number Generation), CUSPARSE (Sparse Matrices) and NPP (NVIDIA Performance Primitives). The jCuda API consists of a group of static methods which are very similar to the native library functions since the aim of jCuda is to keep the API as close to the original as possible, including also functions in order to use user defined kernels in CUDA language, as well as pointer handling functions.

Java-GPU [18] introduces directives to offload Java code into the GPU, whereas Rootbeer [104], which has recently been published, provides a specific high-level API for Java and translates the generated bytecode into CUDA.

Java OpenCL binding solutions include JOCL [56] and JogAmp JOCL [57]. The main difference between them is that while the former provides support for OpenCL 1.2, the latter only handles OpenCL 1.1.

Finally, Aparapi [4] is the most up-to-date Java OpenCL project and provides OpenCL 1.2 support. The Aparapi programmer is provided with a high-level API to express data parallel workloads in Java, being released from all the GPU implementation details. Nevertheless, in order to obtain higher performance, the user must be aware of some architectural details, although no OpenCL knowledge is needed. The runtime system translates Java parallel workloads to OpenCL and offload them on an OpenCL device (a GPU or a CPU) or on a pool of threads (in this case, no translation is needed). Aparapi is supported by AMD and its source code has been released with a GPL license.

Chapter 2

Shared Memory Architectures

The lack of frequency scaling has motivated the increase in the number of cores per processor, and thus current shared memory architectures aim to provide higher computational power by the aggregation of smaller processing units. In this chapter, in order to analyze the shared memory architectures used in this Thesis, we have classified them in multi- and many-core processors. Although it is not clear the number of cores that determine the division, we considered as multi-core the stand-alone processors, i.e., those providing parallel features by the combination of general purpose cores. And, as many-core, accelerators or coprocessors with an aggregation of specific purpose cores that are not expected to be efficient for sequential computation.

2.1. Multi-core Architectures

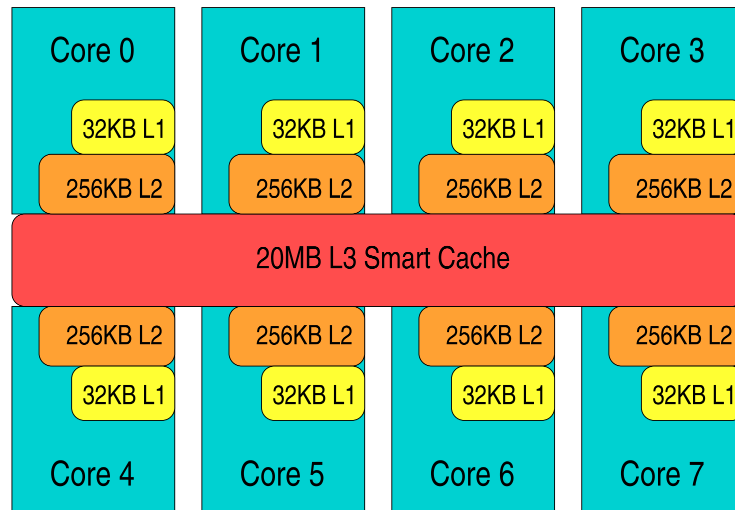
Multi-core architectures appeared as a solution to the lack of power scaling and dissipation issues that arose when trying to increase the clock frequency on uni-core processors. Thus, multi-core systems provide more computational power by combining several processing cores within the same chip. Some of them can also be combined within a node using high-speed connectors such as the Intel QuickPath Interconnect (QPI) or the AMD HyperTransport (HT) technology.

Usually, these cores are arranged as NUMA systems where the cores within the

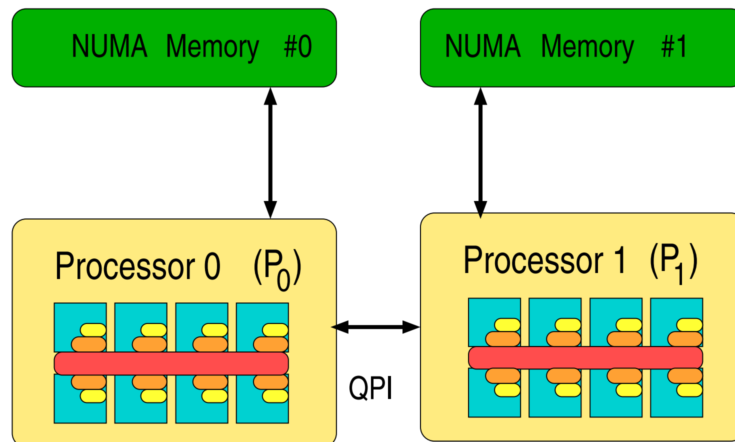
same chip get access to the memory regions through a shared bus. When there are several processes connected by QPI or HT, cores from one processor have to go through these links to access the NUMA regions of another processor, thus experiencing a higher access overhead. Within each chip, cores are linked in different manners, e.g., Intel Nehalem uses a crossbar system, which is highly efficient but not scalable in terms of cost, and Intel Sandy Bridge uses a ring bus which is slower but more scalable. Another feature of these multi-core architectures is that cache coherency is kept in each node (using links among processors with this purpose). Usually, cores have some private caches and a larger shared cache used to maintain coherency within each processor.

The rest of this section focuses on describing two representative up-to-date multi-core systems, based on the Intel Sandy Bridge (Xeon E5) and on the AMD Magny-Cours. Both of them will also be used for the performance evaluation of the libraries developed in this Thesis, along with others with similar characteristics, so their main architectural features are detailed. On the one hand, the Xeon E5 processor presents a quite recent micro-architecture (Intel Sandy Bridge) for high performance shared memory systems, achieving up to 20.8 GFlops of peak performance per CPU core in the model used (2.6 GHz), which represents one of the best peak performances per core amongst the currently available systems. On the other hand, the 48-core Magny-Cours-based system has one of the highest aggregated performance numbers in a shared memory system (403.2 GFlops), although its performance per core is quite reduced (8.4 GFlops). Hence, the Xeon E5 system has allowed us to analyze a quite recent Intel micro-architecture, whereas the Magny-Cours system presents the issues associated with the integration of multiple multi-core processors.

Figure 2.1a presents the layout of an 8-core Xeon E5 processor, based on the Sandy Bridge-EP architecture, where up to 16 threads can run simultaneously thanks to hyperthreading. The eight cores in this processor share the L3 cache (called LLC or Last Level Cache), implemented as an Intel Smart Cache, where each core can access the whole cache when the rest of the cores are idle. This cache is divided in physical slices connected to an internal ring bus. Figure 2.1b shows the interconnection layout in a dual-socket Intel Xeon E5-2670 system where the processors and the memory are linked by a QuickPath Interconnect (QPI). This NUMA system supports DDR3-1600 MHz memory.



(a) Intel Xeon E5-2670 processor

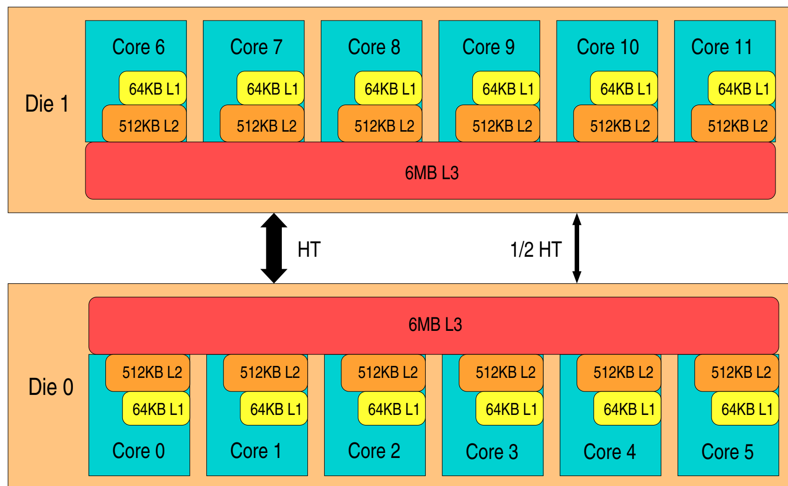


(b) Dual-socket Intel Xeon E5-2670 system

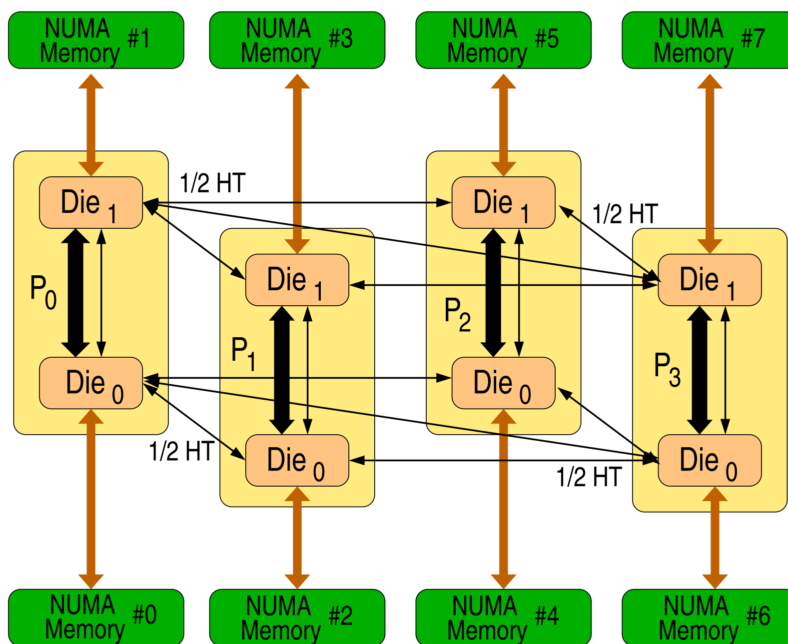
Figure 2.1: Architecture of an Intel Xeon E5-2670 Sandy Bridge-based system

The second system, a fat node from the DAS-4 cluster [24], has 48 cores in 4 AMD Opteron 6172 processors (Magny-Cours), each one with 12 cores [3, 77] and 128 GBytes of RAM. Figure 2.2a presents the layout of the 12-core Magny-Cours processor, which is composed of two 6-core AMD Opteron Istanbul dies (the 6 cores share the L3 cache) interconnected by HyperTransport (HT) links. Figure 2.2b shows the HT interconnections between the different dies as well as the direct access of each die to its memory region with DDR3-1333 MHz support. Thin arrows represent half HT links (8 bits), whereas thick ones represent full HT links (16

bits). As each 12-core processor is a NUMA system with 2 NUMA regions, this quad-socket system has eight NUMA regions.



(a) AMD Opteron 6172 processor



(b) Quad-socket Magny-Cours AMD Opteron 6172 system

Figure 2.2: Architecture of a Magny-Cours AMD Opteron 6172-based system

2.1.1. Thread Affinity Control in Java

A consequence of the popularization of NUMA architectures for multi-core systems is that the control of the location of running threads within a node or a processor can have significant effects on performance. For instance, if two threads are communicating through shared memory transfers and they share the last level of cache, transfers will go through this cache instead of across an interconnection link.

If the communication is performed between two JVMs (thus between two different processes), OS utilities can be used when launching the processes to state the core in which each process is running (e.g., the `numactl` tool can be used in UNIX). However, when using threads within a single JVM, these tools only support to bound the JVM execution to a set of cores, but the thread mapping within this set is managed internally by the JVM. Thus, if a more accurate mapping is needed, such as a thread-to-core mapping, it would have to be controlled by the application.

Unfortunately, Java does not provide any affinity or pinning control over threads. Nevertheless, JVM threads are directly mapped to native OS threads and, taking advantage of this property, it was possible to develop an affinity tool in order to enable a fine-grained thread mapping control within the JVM. This tool accesses native OS primitives (in particular, `pthread_setaffinity_np`) through JNI, thus supporting affinity management in a Java multithreaded application. In the experimental evaluations included in the following chapters, this tool will be used to improve performance by defining optimal thread-to-core mappings. In fact, in the next chapter, the effects of thread mapping will be evaluated and analyzed in detail.

2.2. Many-core Architectures

Multi-core processors provide high aggregated computational performance but, when the number of cores is large, the probabilities of taking full advantage of all cores at the same time are reduced, because parallel codes have also sequential parts and many of them cannot exploit all the potential provided by a multi-core processor more than in a small portion of the code. A more common situation is a parallel code with unbalanced and heterogeneous parts that will not take advantage

of having several similar cores. In this scenario, having a large number of cores is highly inefficient in terms of energy consumption, and even in terms of cost because it could be solved by having a cheaper set of smaller cores that provide a specific functionality. This is the aim of many-core accelerators or coprocessors. They provide massively parallel architectures with simple computing units or cores to accelerate sections of an application that is run in a general purpose processor. Among these many-core architectures, this section is focused on two of them that are highly interesting: the Intel Xeon Phi, the main accelerator with a x86 architecture, and the GPUs (specifically, the Kepler architecture), which are currently present in the most powerful supercomputers [134].

2.2.1. Many-core x86 Architectures: Intel Xeon Phi

The Intel Xeon Phi is the latest and most scalable many-core x86 coprocessor. It is the first commercial product of the MIC (Many Integrated Cores) architecture, and reflects the efforts made by Intel to develop a many-core coprocessor. The main goal was to provide an accelerator to be programmed with traditional paradigms and languages, avoiding the need to resort to GPGPU computing and complex programming models. This approach was also explored by the IBM Cell Broadband Engine [49], which implemented a new architecture also used in popular video game consoles such as Sony's PlayStation3 and that even reached the 1st position in the Top 500 list [134] in June 2008 with the IBM Roadrunner, but finished its production in 2009.

The precursor of the Xeon Phi is the Larrabee architecture [115], a project that started in 2006 (it was canceled in 2010) and aimed to provide, in a single chip, a combination of GPU and vector accelerator (64-byte VPU unit with scatter and gather loads/stores and mask registers) with a bidirectional ring bus (512 bits per direction), private L1 and 256-KByte L2 sliced caches. Moreover, it implemented the support for running 4 threads per core in some sort of Intel hyperthreading. However, as the performance obtained was below the expected results, the chip was never commercialized. Although the Xeon Phi has inherited most of its features, the main changes are the lack of graphics processing support, the OS running on the coprocessor instead of on the host and that main memory is GDDR5 (it was

DRAM on Larrabee).

In the meantime (around 2009), Intel was also involved in the development of the Single-chip Cloud Computer (SCC) processor [100], with multiple independent cores (48) without cache coherency and interconnected with a 2D mesh network. The cores (Pentium I P54C at 800 MHz) have 16-KByte L1 and 256-KByte L2 caches and access to off-chip private DDR3 memories (physically shared). The cores are organized in tiles that contain two cores and a message-passing buffer (MPB, 8 KBytes), and each tile is connected to a router. The main goal of this architecture was to achieve high scalability, but it turned out to be harder to program than a cache coherent architecture.

Intel Xeon Phi is the brand name of all products based on the MIC architecture, which inherits many architectural features from Larrabee except for the GPU-specific hardware. Although its development started in 2010, it was not until November 2012 that Intel announced the first commercial Xeon Phi (named 5110P), and it is expected that different models, varying the number of cores and small hardware features, will shortly be available. Figure 2.3 shows an overview of the 5110P Xeon Phi architecture. It has 60 simplified Intel CPU cores running at 1056 MHz and supports 4 threads per core thanks to hyperthreading (thus, 240 threads in the die). The cores, as the Larrabee ones, have a vector unit with 64-byte registers featuring a new vector instruction set known as Intel Initial Many Core Instructions (IMCI). Each core has a 32-KByte L1 data cache, a 32-KByte L1 instruction cache, and a private 512-KByte L2 unified cache which is kept coherent by a Distributed Tag Directory system. Cores, Tag Directories (TDs) and memory are connected to a bidirectional ring that consists of three independent rings in each direction [22]: the data block ring (64 bytes wide), the address ring (send/write commands and memory addresses) and the acknowledgment ring (flow control and coherency messages). There are 64 TDs connected to the ring and the address mapping to the TDs is based on hash functions over the memory addresses, leading to an even distribution around the ring. Although this causes all cache transfers to go through the TDs, increasing the minimum latency, it also provides homogeneity in the access time. The memory controllers provide access to the GDDR5 memory (8 GB of global memory). The coprocessor runs a simplified Linux-based OS in one of the cores. The Xeon Phi can be used as a mere coprocessor in which the host offloads code

to be accelerated, as an independent unit that runs a whole application, or as an independent unit that communicates in a symmetric manner with the host [50].

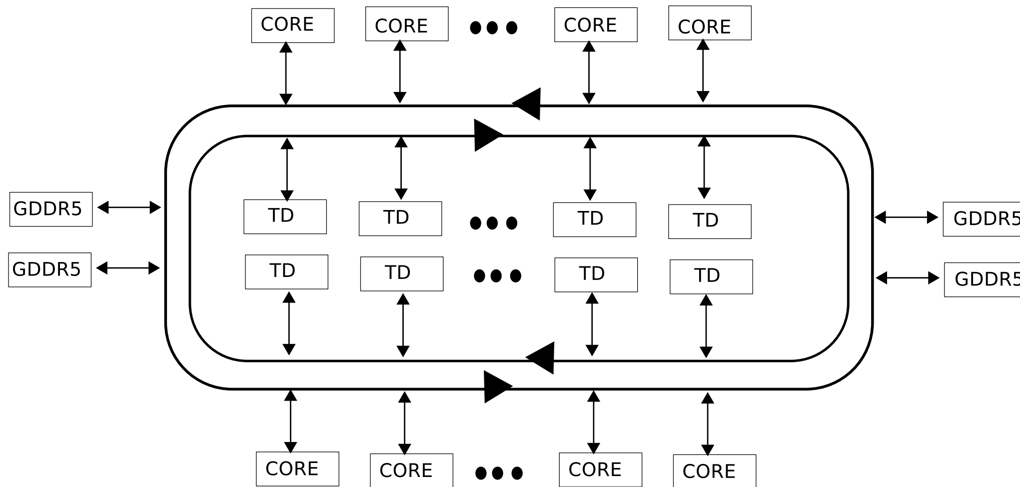


Figure 2.3: Architecture of the Intel Xeon Phi coprocessor

2.2.2. Graphics Processing Units (GPUs)

Graphics Processing Units (GPUs) appeared as a means to leverage graphics computing to special units that take advantage of their particularities to improve performance. Generally, graphics computing presents massive parallelism that can be exploited in terms of stream and vector operations. However, this high parallelism, the floating point capabilities, and the generalization of GPU architectures, made it an interesting choice for general purpose computation (GPGPU) [32, 71].

Modern GPUs use unified architectures of scalar cores that enable their use not only for graphics purposes but for any other vector-based computation. They follow the Single Instruction Multiple Thread (SIMT) model, providing multiple small processing units to exploit data parallelism with threads managed by hardware. The main GPU manufacturers are AMD and NVIDIA and, although they present some differences in their architecture and their nomenclature differs, they both exploit the idea of having several multi-processors composed by Single Instruction Multiple Data (SIMD) processors. As a consequence, GPGPU is mainly based on stream

computing and the codes are structured in small kernels that consist of operations that are applied to many elements of a stream in parallel.

This section describes the Kepler architecture, the last GPU family released by NVIDIA, which is the coprocessor vendor with the highest share in the Top 500 list [134], including the number one (Titan) in the November 2012 list (second in the June 2013 list). In a NVIDIA architecture, the main processing unit is a Streaming Multi-processor ("SM", or "SMX" in Kepler architectures) composed by SIMD Processors ("SP") that can schedule the same instruction on a group of threads ("warp"). Kepler GPUs are based on the GK110 architecture (15 SMXs by default) [95]. Each SMX has 192 FP32 and 64 FP64 CUDA cores, 256 KBytes of register file space, 64 KBytes of on-chip memory (one per SMX) and 48 KBytes of uniform cache. The 64-KByte on-chip memory can be configured as 48 KBytes of shared memory with 16 KBytes of L1 cache or vice versa (or in a 32/32 KBytes manner). Moreover, the 48-KByte uniform cache supports full speed unaligned memory access patterns automatically managed by the compiler. The L2 cache (1.5 MBytes by default) serves as data unification point among the SMX units. Among the Kepler series, we used the NVIDIA Tesla K20m in the experiments of Chapter 5, with 2496 SPs (13 SMXs) at 706 MHz, 5 GB GDDR5 (5.2 GHz) and a 1.25-MByte L2 cache.

The SMX schedules threads in warps of 32, having 4 schedulers and 8 instruction dispatch units, allowing 4 warps to be issued and executed concurrently, and 2 independent instructions per warp can be dispatched each cycle. Moreover, the GK110 allows double precision instructions to be paired with other instructions. Each GK110 SMX can have up to 2K threads at any time.

The GK110 incorporates the Hyper-Q technology, that expands the number of hardware work queues from 1 (on previous architectures like the GF100) to 32, allowing connections from multiple CUDA streams, multiple MPI processes, or multiple threads within a processor; and operations in one stream will not block other streams, avoiding false dependencies. It also offers significant benefits to legacy MPI-based algorithms optimized for multi-core CPU systems where a single MPI process ends up with insufficient work to take full advantage of the GPU. Using only one queue, MPI processes could also share a GPU but they were bottlenecked by false dependencies that can now be removed.

The Kepler architecture also implements dynamic parallelism, which consists of the ability for kernels to be able to dispatch other kernels without CPU interaction, implicitly relying on recursion. Since both CPU and GPU can launch new kernels, Kepler includes a Grid Management Unit that controls and prioritizes them.

Figure 2.4 shows the architecture of a GK110 SMX including the internal memory subsystem and the cores. Other elements shown are the Double Precision Units (DP Unit), the Special Function Units (SFU) and the Load/Store Units (LD/ST). “Tex” represents 16 Texture Units used for graphics processing.

Figure 2.4: Architecture of the Kepler GK110 SMX, basic block of the NVIDIA K20

2.3. Java Support for Shared Memory Architectures

The integration of several cores in a single processor has become a popular resource to address the physical limits of clock frequency scaling. However, the generalization of multi- and many-core processors has also led to the spread of a large variety of shared memory architectures whose particularities have to be taken into account when developing efficient codes. Moreover, not only the differences between x86 and GPU accelerators are remarkable, but also x86 multi- and many-core systems present noticeable differences. The main concerns of many-core processor designers are scalability and throughput: the goal is having a large number of processes or threads carrying out many small tasks. However, multi-core processors have to find a trade-off between throughput and latency since they are also meant to provide reasonably good performance for sequential codes. Regarding multi-core systems, the differences are mostly related to topologies and interconnection strategies and technologies. Hence, having some insight into memory hierarchies, interconnection between cores and management of shared resources, among other architectural aspects, is crucial to provide efficient and optimized parallel codes.

Programming languages, like Java, have provided several approaches to take advantage of multi-core architectures, usually through multithreading techniques. Although Java multithreading support enables the efficient use of underlying multi-core systems, it has to be managed carefully, especially when the number of threads increases, given that synchronization and contention in the access to shared resources can severely limit its scalability and increase complexity. A solution based on a message-passing API over Java multithreading in order to increase scalability and ease of use is proposed in Chapter 3, and an efficient library of collective operations optimized for multi-core architectures is presented in Chapter 4. In addition, in a scenario of multi-core processors with NUMA architectures, it is also necessary to take into account the location of threads and use affinity techniques to exploit locality. The message-passing middleware of these chapters benefits heavily from the fine-grained affinity control described in Section 2.1.1 and the multi-core processors described in Section 2.1 will serve as testbeds.

Other shared memory architectures like coprocessors or accelerators do not sup-

port directly the use of traditional programming languages (e.g., the GPUs) or do not provide JVM support (e.g., the Intel Xeon Phi). This lack of support prevents the use of multithreading techniques to exploit this kind of architectures with the Java language. As it has been mentioned in Section 1.3, several projects have tried to address this issue by providing Java support for OpenCL or CUDA languages through JNI access. In Chapter 5, two of these existing solutions were selected to be analyzed and benchmarked in both a NVIDIA K20 GPU and an Intel Xeon Phi.

Chapter 3

A Shared Memory Communication Device for Message Passing in Java

Parallelism for shared memory systems is inherently supported in Java through multithreading, which allows the simultaneous execution of multiple tasks in a JVM, hence taking advantage of shared memory intra-process transfers. However, thread programming increases the development complexity due to the need for thread control and management, task scheduling, synchronization, and access to and maintenance of shared data structures, which is always accompanied by the presence of thread-safety concerns. An alternative for overcoming these limitations is the use of message passing on shared memory systems, whose support has been implemented in a low-level communication device named `smdev` that is presented in this chapter. This device includes the implementation of communications using shared memory data transfers, and hence it is possible to program efficiently on shared memory without dealing with threads, offering a high level of abstraction that supports handling threads as message-passing processes. This ease of use has been demonstrated in the straight integration of `smdev` in our Java message-passing implementation, FastMPJ [29, 123] (see Figure 1.1 for a global overview).

3.1. State of the Art of Message Passing for Shared Memory

Message passing was selected as the programming model to support shared memory as it is the most widely used in parallel programming, especially for distributed memory. The device developed, `smdev`, is based on threads while current message-passing implementations, either MPI for natively compiled languages or Java message passing, do not take full advantage of multithreading for intra-process transfers, and they generally resort to inter-process communication protocols and, in some cases, to network-based communication protocols for data transfers within the same node. This is even more critical with the current increase in the number of cores per processor, which demands scalable shared memory communication solutions.

MPI libraries, such as MPICH and Open MPI, are mostly optimized for distributed memory communications, although they are increasingly taking advantage of multi-core shared memory systems. Thus, the MPICH project includes several communication devices for shared memory such as ssm, shm or sshm [28]. It also supports Nemesis [17], a communication middleware which selects the best-fit communication device for the underlying architecture. Nemesis also contains its own highly optimized shared memory communication subsystem. Open MPI includes optimized communications among processes via shared memory (sm Byte Transfer Layer) [130], providing a management subsystem which uses shared memory transfers when possible. Other MPI libraries (mainly proprietary ones) are generally capable of selecting the most appropriate fabric combination automatically, including shared memory optimizations. However, current MPI libraries have to rely on inter-process communications using shared memory resources (e.g., memory mapped regions or SysV resources), which requires at least two data transfers: one from the source process to the shared memory resource, and another one from this resource to the destination process. Thus, optimizations in these communication subsystems (like Large Message Transfers in Nemesis [28]) aim to optimize data transfers from and to shared memory through fragmentation and rendez-vous protocols, but they are not able to eliminate the intermediate copies. The only solution to avoid extra copies is the use of kernel modules such as KNEM or XPMEM (only for SGI machines). Both MPICH and Open MPI provide support for direct memory transfers

with KNEM, which has been used in [76] to develop hierarchical and topology aware collectives. The main drawback is the lack of portability due to the need of external kernel modules.

Although there are several message-passing projects in Java (see Section 1.2), nowadays FastMPJ [29, 123] and MPJ Express [116] have the most active development. MPJ Express has already implemented its shared memory support [118] through its multi-core device, but it presents serious drawbacks in scalability due to coarse-grained synchronization and buffering in the upper layers of the middleware. This causes its performance not to be competitive compared to current MPI shared memory devices. FastMPJ includes the communication device presented in this chapter, `smdev`, whose buffer layer avoidance and reduced synchronization overhead improve significantly the scalability of FastMPJ when communications involve a large number of Message Passing in Java (MPJ) processes on shared memory.

3.2. Design and Implementation of the Shared Memory Device `smdev`

The goal of `smdev` is to increase the scalability of Java applications by providing an efficient communication middleware for multi-core shared memory architectures. Messaging libraries usually require the use of several instances of the JVM per shared memory system, thus incurring high communication overhead and memory consumption (see left graph in Figure 3.1), whereas `smdev` runs several threads within a single JVM instance (see right graph), thus taking advantage of thread-based intra-process data transfers, as well as lower memory consumption.

Although the minimum memory required by a JVM is system- and JVM implementation-dependent, it is usually around a hundred MBytes. Hence, `smdev` saves this memory for the second and consecutive cores communicating in a system. Additionally, garbage collection represents a higher overhead when using several JVMs, as they have a more limited amount of memory than using a single JVM; this is a consequence of the fragmentation and high memory consumption when using multiple JVMs in a shared memory system.

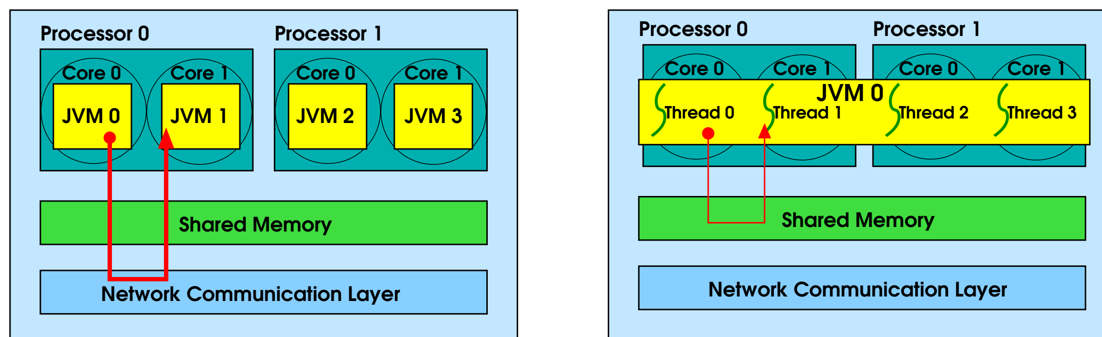


Figure 3.1: Java communications on a dual-core dual processor using distributed (left) and shared (right) memory-oriented middleware

3.2.1. Low-level Message-Passing API: `xxdev`

The `smdev` device provides a message-passing API that conforms with the `xxdev` API [29, 123] (see Listing 3.1 and Figure 1.1), which avoids data buffering by supporting direct communication of any serializable object.

Listing 3.1: API of the `xxdev.Device` class

```
public abstract class Device {

  static public Device newInstance (String deviceImpl);

  public int[] init (String[] args);
  public int id ();
  public void finish ();


  public void send (Object buf, int dst, int tag);
  public Status recv (Object buf, int src, int tag);

  public Request isend (Object buf, int dst, int tag);
  public Request irecv (Object buf, int src, int tag, Status stts);

  public void ssend(Object buf, int dst, int tag);
  public Request issend(Object buf, int dst, int tag);


  public Status probe (int src, int tag, int context);
  public Status iprobe (int src, int tag, int context);
  public Request peek ();

}
```

The API 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), functions to check incoming messages without actually receiving them (probe and iprobe), and the peek operation, that only receives a message that has already arrived.

The use of a simple message-passing API supports a direct migration to distributed memory systems, thus benefiting from higher portability and ease of use, avoiding the issues associated with multithreading programming.

The implementation of `smdev` over shared memory required handling with JVM class loaders to maintain shared structures (message queues) for communication, and the optimization of the synchronization among threads in the access to these shared structures. The details of these implementation issues, along with the presentation of the communication protocols, are discussed next.

3.2.2. Class Loading in `smdev`

The use of threads in `smdev` as message-passing processes requires the isolation of the namespace for each running thread, simulating the distributed memory environment in which threads can exchange messages that are actually transferred through shared memory references. This management relies on custom class loaders, a mechanism similar to the one used in MPJ Express [118].

The purpose of the namespace isolation is, therefore, to implement the abstraction of MPJ processes over threads. While processes from different JVMs are completely independent entities, threads within a JVM are instances of the same application class, sharing all static variables. Hence, the user classes and the high-level `smdev` classes, as well as some related to the device management, have to be isolated to behave like independent processes, having private static variables. Nevertheless, the communication through shared memory transfers requires the access to several shared classes within the device.

To achieve this dual behavior, let us start by analyzing the JVM load class system. A JVM identifies each loaded class by its fully qualified name and its class loader, so each loader defines its own namespace. Through creating each thread with

its custom class loader, all the non-shared classes within a thread can be directly isolated. The JVM uses a loader hierarchy in which the system class loader is first invoked when trying to load a class. When the system loader does not find a class, the next class loader in the hierarchy is used; in our case, the next loader is the custom class loader. This mechanism implies that the system class loader is going to load every reachable class that, in consequence, is shared by all threads. Its classpath has therefore to be bounded in such a way that it only has access to shared packages (`runtime` and `smdev`) that contain the implementation of shared memory transfers among threads.

The class loading particularities of `smdev` also affect communications. If the data type sent in a message is a user object, which must agree with the `Serializable` interface, there is a serialization/de-serialization process involved. `smdev` could have managed these communications using the `Cloneable` interface instead, but there are more classes that conform with the `Serializable` interface, and it is also more flexible and presents fewer conflicts with the class loader structure than `Cloneable`. Moreover, `Serializable` is a well-established constraint by standard Java for input/output operations. Thus, the object to be sent is serialized using the thread-local class loader of the sender. However, if the de-serialization is done by the JDK `ObjectInputStream` class, which relies on the system class loader by default, the JVM will consider that the de-serialized object has a different class from the expected one and a `ClassNotFoundException` will be thrown. To deal with this issue, a custom class which overrides the `resolveClass` method of `ObjectInputStream` is used, making the local class loader of the invoking thread load the class in the `Class.forName` method. This technique requires the serialization to be run by the sender thread and the de-serialization by the receiver thread, a constraint that has to be taken into account in the message transfer protocols when any of the communicating threads can eventually complete the communication.

3.2.3. Message Queues

As communications in `smdev` are implemented by shared memory transfers, point-to-point communication operations delegate on a shared class that manages shared message queues to handle pending communication requests for sends and re-

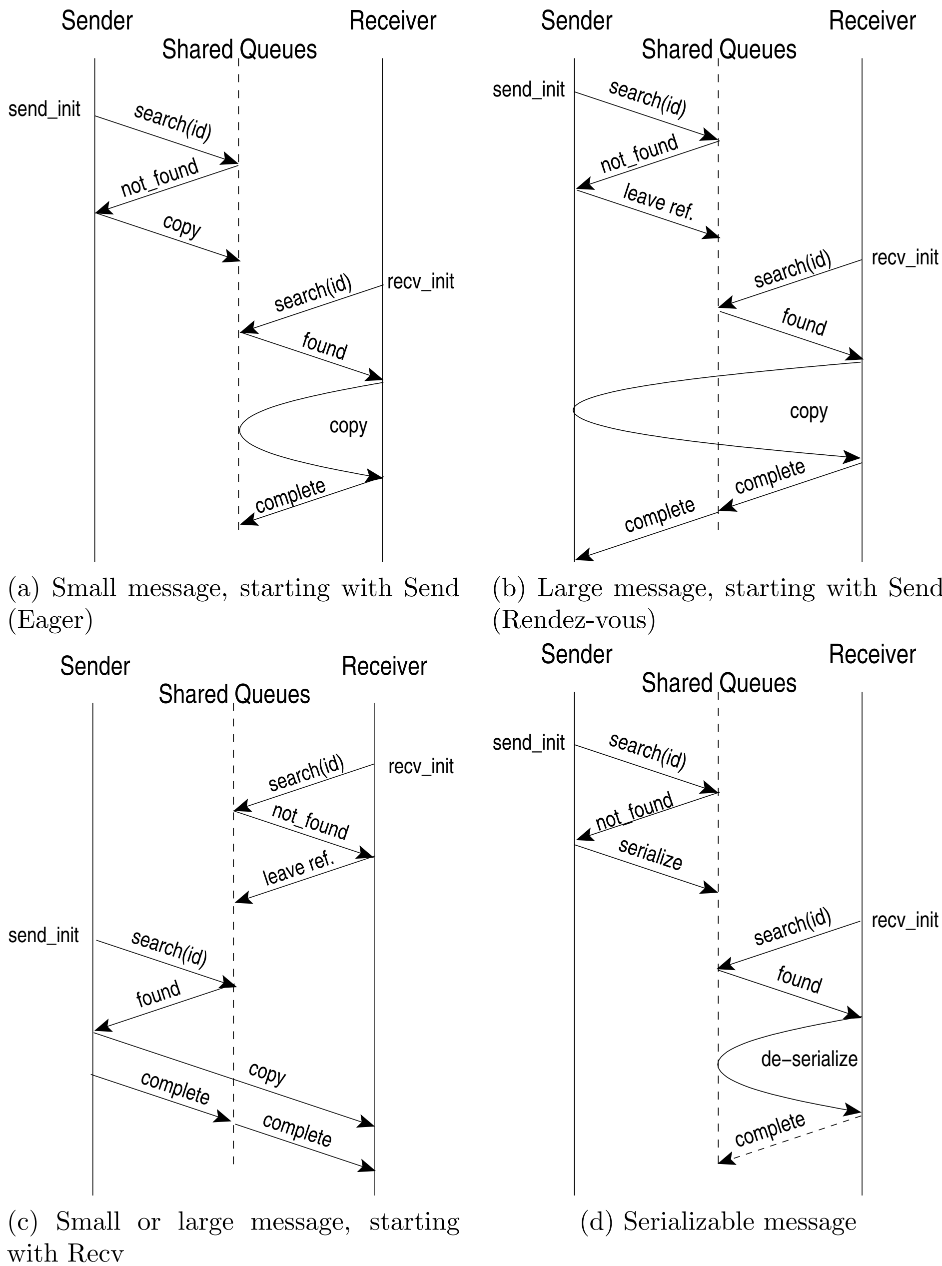
ceives. Each thread has two queues assigned, one for the incoming messages posted by senders (from now on, `UnexpectedRecvQueue`) and the other one for pending receive requests posted by itself (from now on, `PostedRecvQueue`). The access to each pair of queues is synchronized to avoid inconsistency. Nevertheless, having a pair of queues per thread instead of a global one reduces the contention and makes it possible to optimize a fine-grained synchronization.

Each message queue consists of a linked list, which is implemented over a combination of a fixed-size array and a dynamic structure, where the first incoming message is posted on the head of the list, the next one is enqueued after that and so on. In a common working situation, senders and receivers operate in parallel and communications complete quite soon. Thus, the expected number of pending requests is not large and they usually fit in the static array. However, there might be situations where pending requests exceed the size of the static array. To manage them, the device stores new messages in the dynamic structure. As the static structure gets available room, new messages will be stored in it, so that the dynamic structure only stores new requests when the array is full. One of the requirements of messaging libraries is that, when two pending requests have the same identification, messages should be dequeued in FIFO ordering. Since our pending requests in the static array are not necessarily older than the requests in the dynamic structure, a sequence number is included in each request to identify which one should be dequeued.

3.2.4. Message Transfer Protocols

Sends and receives rely on the shared message queues already described, using as message identification the source identifier, a user tag and a context, which is managed internally by the device. In order to cope with duplicity of message identification, the sequence number is also taken into account for retrieving pending messages.

A thread sending a message to another thread first has to check if there is already a matching receive request in the destination `PostedRecvQueue`. If there is a match, the sender copies the message in the destination address and the request is marked as completed. When there is no match, the sender inserts the message request in the `UnexpectedRecvQueue`. Depending on the communication protocol, the sender



(a) Small message, starting with Send (Eager)

(b) Large message, starting with Send (Rendez-vous)

(c) Small or large message, starting with Recv

(d) Serializable message

Figure 3.2: Communication protocols in `smdev`

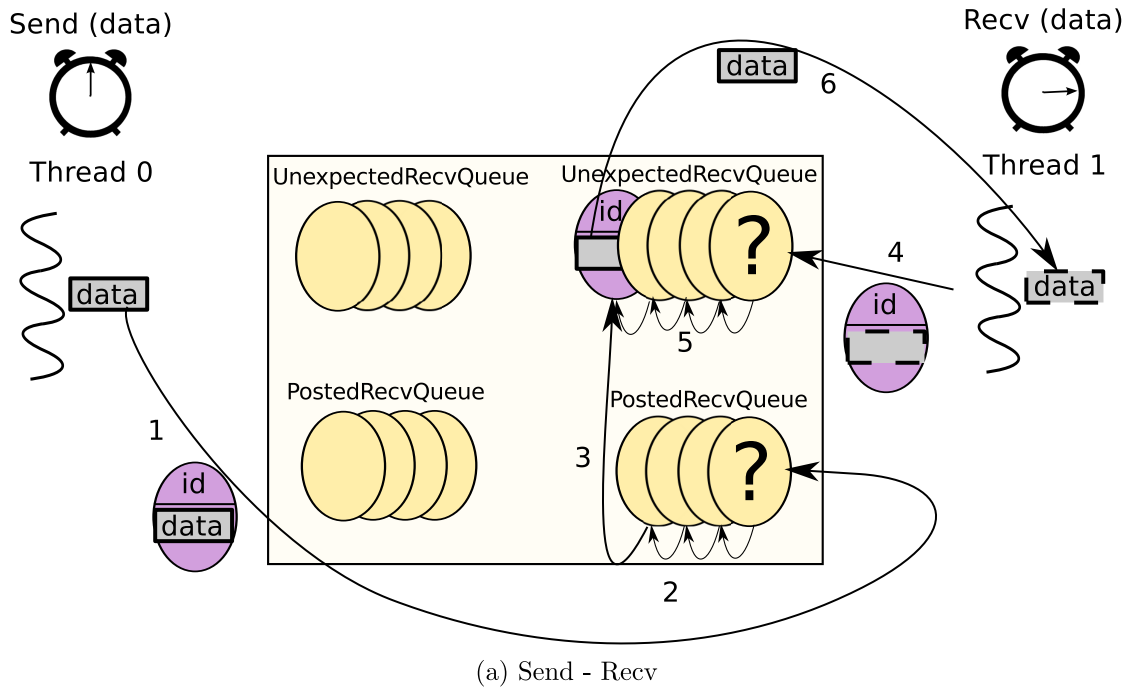
will store the data in the queue or it will leave a reference to it. This send request will be queued until the destination posts a receive request for this message. The reception operation works inversely. The receiver checks its `UnexpectedRecvQueue` and, if there is a matching message request, the data is copied into the destination

address and the communication is completed. If there is no match, it enqueues a receive request in the PostedRecvQueue, where it will be queued until a matching message request is received.

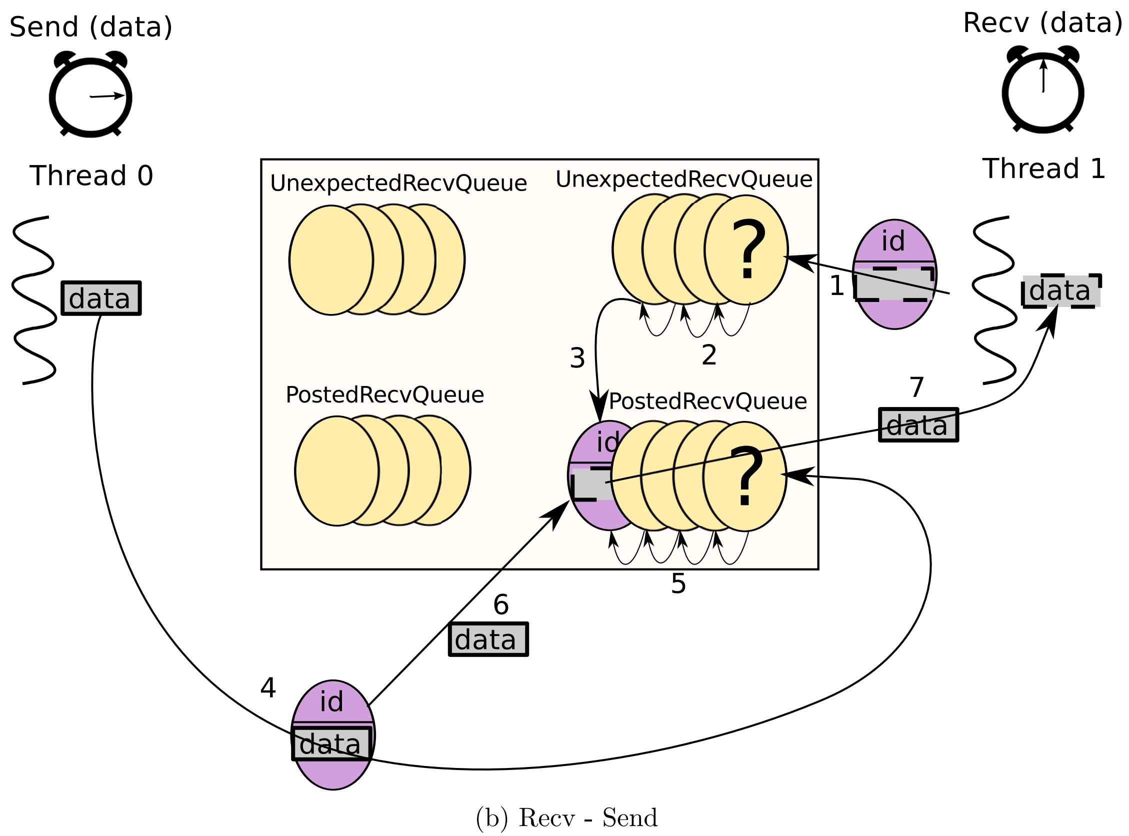
The communication protocol establishes the management of the request in the shared queues. Figure 3.2 includes the protocol operation for the different communication situations. For primitive types, we can distinguish between the eager and the rendez-vous protocol. With an eager protocol (Figure 3.2a), the sender copies the message data in the request buffer and assumes the communication as completed. Next, another copy is performed from the intermediate buffer to the receiver. When the amount of data is large, the cost of this extra copy becomes a bottleneck and it is more convenient to use a rendez-vous zero-copy protocol (Figure 3.2b), where the sender leaves a reference to its own buffer. In this case, the data is copied directly to the receiver buffer when it is available, avoiding the extra copy (zero-copy protocol). However, the sender cannot assume the communication as complete until the receiver has copied the data. The data size boundary to choose between both protocols is established via a "Protocol Size Limit" parameter, which is 64 KBytes by default. Nevertheless, when the receiver initiates the communication (Figure 3.2c), it has to leave a reference to its own buffer in the request, allowing the sender to make a direct copy in the receiver buffer and thus avoiding the extra copy. When using a serializable message (Figure 3.2d), as discussed before, the serialization has to be carried out by the sender thread and the de-serialization has to be run in the receiver thread. In this case, the serialized data must be stored in the request buffer, regardless of the message size.

Figure 3.3 shows two threads communicating on two scenarios, according to the thread which initiates the communication. The numbers in each scenario indicate the order in which the actions are taken. Message requests are represented by ovals and the active one is in dark. The "id" tag represents the identification of the request and the small rectangle represents the message data (if the border is continuous) or a buffer (if the border is dotted). Requests that are posted by a receiver thread have empty buffers, while requests which are created by sender threads contain the message data.

Regarding the first scenario (Figure 3.3a), Thread 0 sends a message (step 1) before Thread 1 posts the corresponding receive request. After checking the



(a) Send - Recv



(b) Recv - Send

Figure 3.3: Send/Recv operations in `smdev`

PostedRecvQueue of the destination for a matching request without success (step 2), the sender enqueues the send request in the UnexpectedRecvQueue (step 3). When Thread 1 initiates the reception process (step 4), it finds a matching request in the UnexpectedRecvQueue (step 5) and copies the message into the destination buffer (step 6).

Regarding the second scenario (Figure 3.3b), first Thread 1 initiates the communication with a receive operation (step 1) which does not have a matching request in the UnexpectedRecvQueue (step 2), so it is posted in the PostedRecvQueue (step 3). Next, the sender (Thread 0) sends the message (step 4) and finds the matching receive request in the PostedRecvQueue (step 5). The communication completes by transferring the message data into the destination buffer (steps 6 and 7).

For further details, Listings 3.2 and 3.3 illustrate the pseudo-code of the non-blocking receive and send methods (irecv and isend), respectively. Both methods have to check if the message has to be serialized and also search for a matching request in the corresponding queue. If no request is found, a new pending request is

Listing 3.2: Pseudo-code of the irecv method

```
Request irecv (Object buf, int src, int tag, Status stts){

    /* lock both of my queues */
    synchronized(lock[me]){
        request = unexpectedRecvQueue[me].find_and_get_message();
        found = (request != null);

        if(!found){ // the request has to be inserted
            request = new Request(...);
            postedRecvQueue[me].insert(request);
        }
    }
    /* end of the lock */

    if(found){
        if(!request.serialized() && !request.completed())
            buf = copy(request.buf);
        else
            buf = deserialize(request.buf);

        request.setCompleted(true);
    }

    return request;
}
```

inserted into the other queue, and otherwise the communication is performed, unless the message is serialized, in which case, and as explained above, only the receiver can perform the de-serialization. When the communication is not completed in either one of these calls, the call to `iwait()` over the request finalizes the transfer. The `iwait` method has to be called over a request to complete a nonblocking communication (`irecv` or `isend`).

Listing 3.3: Pseudo-code of the `isend` method

```
Request isend (Object buf, int src, int tag){

    if (buf.isObject())
        serialized = serialize_data(buf);

    /* lock the receiver's queues */
    synchronized(lock[dest]){
        request = postedRecvQueue[dest].find_and_get_message();
        found = (request != null);

        if(!found){ // the request has to be inserted
            request = new Request(...);
            if(!buf.isObject()){
                if(eager(buf)){
                    request.buf = copy(buf);
                    request.setEager();
                }
                else // zero-copy protocol
                    request.buf = buf; //leave a reference to the message
            }
            else
                request.buf = serialized;
            unexpectedRecvQueue[dest].insert(request);
        }
    }
    /* end of the lock */

    if(found){
        if(!buf.isObject())
            request.buf = copy(buf); //direct copy in the receiver buffer
        else
            request.buf = serialized;
    }

    if(request.eager() || found)
        request.setCompleted(true);

    return request;
}
```

3.2.5. Synchronization

Synchronization is one of the main performance bottlenecks in shared memory communications. In smdev, synchronization among threads is needed to guarantee thread safety, avoiding race conditions. There are two types of scenarios in which synchronization is required. On the one hand, situations where the number of threads that are going to perform a well-determined task is known. This is the case of the initialization of the device, where every thread has to register itself, or when a thread is waiting for a message request to be completed (it is known that only one thread has to complete the operation). In these cases, the middleware resorts to busy waits over atomic variables in order to minimize the communication latency. The introduced overhead of the busy wait is acceptable because these are small tasks that are expected to be complete in a short period of time. In this case, smdev trades off latency for CPU consumption contributing to code scalability. Moreover, a busy wait avoids context switch overheads and, as the number of scheduled threads is expected to be lower or equal to the number of available cores in a shared memory system, a blocking wait would not report any benefit since there are no other threads waiting for CPU resources.

On the other hand, there are scenarios where the interactions among threads are more complex or unpredictable. This is the case of the access to the message queues. Each thread can read and insert requests in its own reception queues, but every other thread can also search and insert requests in these queues when sending a message. Thus, in this scenario, explicit synchronization with locks is needed to avoid inconsistency in the shared queues. To reduce the overhead and contention, a lock per each pair of queues is used. Therefore, a thread trying to send or receive only blocks the queues needed to perform the operation. Both queues of each thread are blocked simultaneously because a thread only makes insertions in a queue if it has not found a matching request in the other paired up queue, creating a dependence condition in the consistency of the queues.

The use of a pair of queues per processor enables smdev to include fine-grained synchronizations, combining busy waits and locks, and thus reducing contention in the access to the shared structures. As an example, MPJ Express shared memory support uses a global pair of queues with class lock-based synchronization, which

can result in a very inefficient approach in applications that involve more than a pair of threads.

3.2.6. Integration of `smdev` in FastMPJ

The developed device has been integrated in the FastMPJ library providing Java message-passing applications with efficient support for shared memory communications. The integration has been almost transparent to the rest of the FastMPJ layers thanks to the modular structure of the device layer. The upper layers of FastMPJ rely on the point-to-point `xxdev` API primitives from the communication devices, and thus all the operations and algorithms from FastMPJ, such as the collective operations library, can benefit from the use of `smdev` without further knowledge of the communication system (see Figure 1.1). Besides the device module, only a specific multi-core boot class had to be added. This class, independent of the rest of the runtime system, implements the scheduling of threads within the custom class loaders (see Section 3.2.2).

3.3. Performance Evaluation

The performance evaluation of `smdev` consists of a micro-benchmarking of point-to-point operations and an analysis of the impact of `smdev` on representative parallel codes. An analysis of collective operations performance using `smdev` as underlying device will also be shown in Chapter 4.

3.3.1. Experimental Configuration

The developed device has been evaluated on two representative multi-core systems described in Section 2.1, a 16-core Intel-based and a 48-core AMD-based machine. The first one ("Xeon E5") has 2 Intel Xeon E5-2670 8-core processors at 2.6 GHz [51] and 64 GBytes of RAM (see Figure 2.1). The second one ("Magny-Cours") has 48 cores in 4 AMD Opteron 6172 processors, each one with 12 cores [3, 77] and 128 GBytes of RAM (see Figure 2.2). The OS is Linux CentOs with kernel v2.6.35,

the GNU compilers are v4.4.4 and the JVM is OpenJDK Runtime Environment v1.6.0_20 (IcedTea6 v1.9.8).

The performance of `smdev` has been evaluated comparatively against MPJ Express v0.38 and two representative MPI implementations which provide efficient communication protocols for distributed and shared memory systems for natively compiled languages (C/C++, Fortran). The implementations selected for this evaluation are MPICH2 v1.4 and Open MPI v1.4.3 on the Xeon E5, and Open MPI v1.4.2 on the Magny-Cours. MPICH2 results have been omitted for clarity purposes since Open MPI obtains better performance on the Magny-Cours. In order to present a fair comparison with `smdev`, these implementations have been benchmarked using their shared memory communication devices: sm BTL in Open MPI and Nemesis in MPICH2. Both libraries have been carefully configured in order to obtain the best performance.

3.3.2. Point-to-point Micro-benchmarking

The performance of point-to-point communications has been evaluated using a representative micro-benchmarking suite, the Intel MPI Benchmarks [110], and our internal implementation of its Java counterpart [124].

Figures 3.4 and 3.5 show point-to-point performance results obtained on the Xeon E5 and Magny-Cours systems, respectively. The metric shown is the half of the round-trip time of a pingpong test for small messages (up to 1 KByte), and the bandwidth for messages larger than 1 KByte. The transferred data are byte arrays, avoiding Java serialization overhead, in order to present a fair comparison with MPI. Moreover, for point-to-point operations, FastMPJ point-to-point routines are direct and thin wrappers over `smdev` primitives, showing quite similar performance.

To analyze the impact of the memory hierarchy on `smdev` performance, we have implemented affinity support in Java allowing pinning a thread to a particular core. This support is based on the `pthread_setaffinity_np` system call invoked by each thread through JNI (see Section 2.1.1). MPI libraries also support pinning control. The impact of thread allocation on performance is analyzed in this section for point-to-point transfers.

Figure 3.4 shows the performance of point-to-point communications between two cores on the Xeon E5. The results have been obtained for transfer operations within a processor ("intra-processor") and between two cores from different processors ("inter-processor"). Since the 8 cores in each processor only share the L3 cache, the specific core mapping within a processor has no impact on performance.

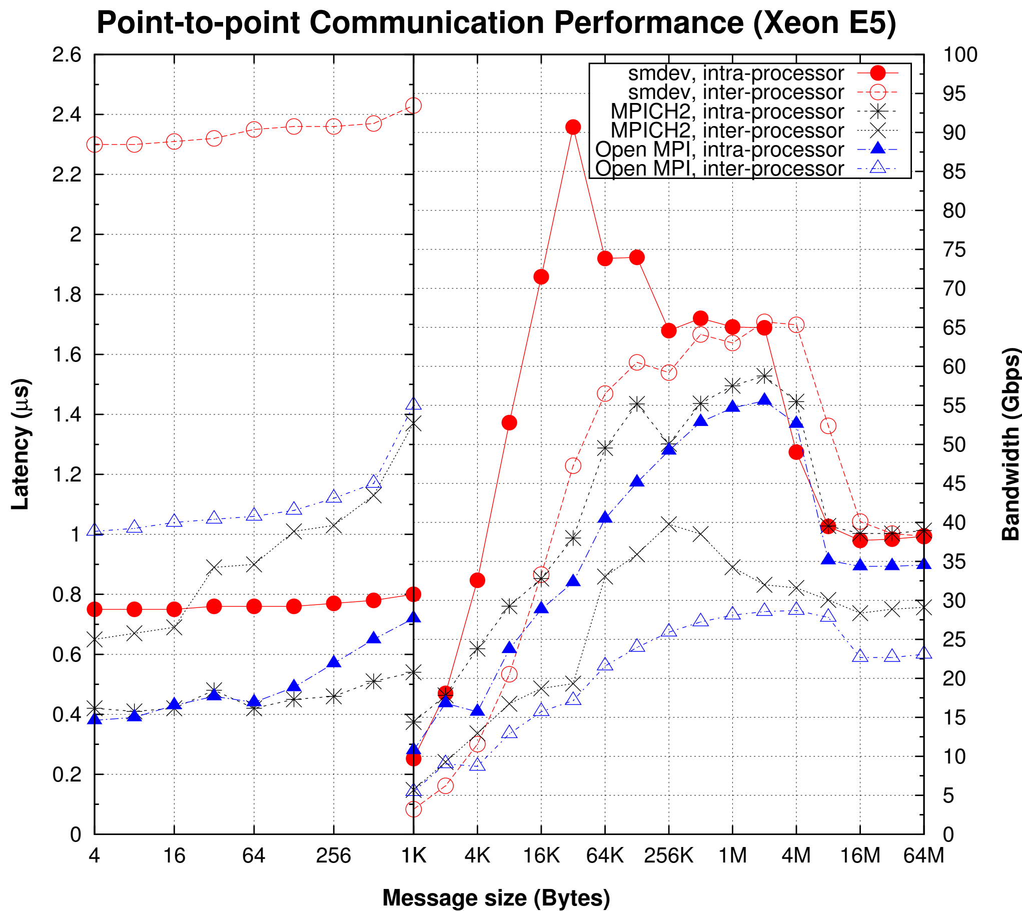


Figure 3.4: `smdev` performance on the Xeon E5

Intra-processor transfers show lower small-message latency than inter-processor ones. This is consistent with the benchmarking configuration, where no cache invalidation is performed and small messages fit in the L1 cache. Although `smdev` doubles the latency obtained by MPI for very small messages, regarding bandwidth results, `smdev` clearly outperforms MPI for messages $\geq$ 1 KByte, achieving the best perfor-

mance for intra-processor communications, especially when messages are around the L1 (32 KBytes) or L2 (256 KBytes) cache size. As the message size increases and it does not fit in the L2 cache, the performance gap between intra-processor and inter-processor `smdev` transfers reduces, which evidences the impact of the memory hierarchy on shared memory performance. It also shows that large-message inter-processor transfers in `smdev` benefit more than intra-processor from the L3 cache, since the bandwidth of the former falls from 4 MBytes on and the latter from 2 MBytes on. Moreover, the zero-copy protocol implemented in `smdev` outperforms the one-copy protocol of MPI (both MPICH2 and Open MPI), and even `smdev` inter-processor transfers outperform MPI intra-processor ones.

Figure 3.5 presents pingpong results on the Magny-Cours, communicating either two cores within a 6-core die ("intra-die" communication), two cores from the same 12-core processor but from different dies ("inter-die, intra-proc."), cores from two dies from different processors directly connected with half HT ("inter-proc, direct"), or two cores from two dies not directly connected ("inter-proc, indirect"). As in the Xeon E5 system, the specific core mapping within a processor has no impact on performance. Two libraries have been evaluated on these four scenarios, `smdev` and Open MPI.

As it can be observed, the lowest latency results are obtained for intra-die transfers, although the start-up latencies are relatively high, at least compared to the latencies measured on the Xeon E5. Thus, MPI results are around 1 μs and `smdev` values around 1.5-2 μs. However, this superior performance of MPI for small messages contrasts with the higher performance of `smdev` for messages larger than 2 KBytes, where `smdev` clearly outperforms MPI thanks to the use of a zero-copy protocol. In fact, `smdev` achieves up to 42 Gbps bandwidth whereas MPI hardly reaches 10 Gbps. These results are significantly lower than the ones obtained on the Xeon E5. Moreover, the peak of bandwidth is obtained at 256 KBytes, while it is at 32 KBytes on the Xeon E5, taking advantage of the messages fitting in the L1 cache. This difference is due to the lower computational power of a Magny-Cours core, which is approximately half of the performance of a Xeon E5 core. This fact severely impacts the performance of the communication middleware, not only for small messages, where the communication overhead is more strongly bound to computation than any other factor, but also for large-message bandwidth, showing around half of the

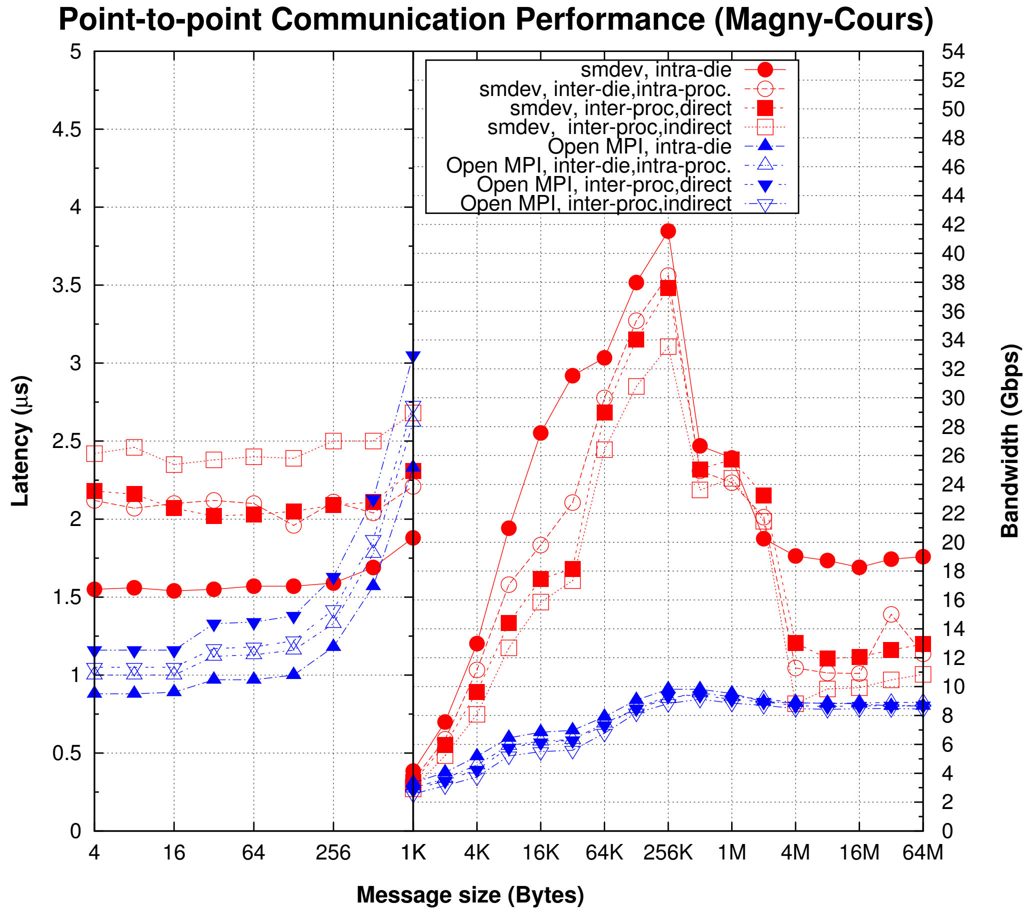


Figure 3.5: `smdev` performance on the Magny-Cours

Xeon E5 performance. Additionally, these results are influenced by the performance of the memory, which has DDR3-1600 MHz support in Xeon E5 and DDR3-1333 MHz in Magny-Cours.

Although `smdev` performance is clearly comparable to traditional MPI libraries, we have also evaluated it against the shared memory support implemented by MPJ Express, which is the MPJ library with the highest number of users and recent maintenance effort. Figures 3.6 and 3.7 show results for `smdev` and MPJ Express on the Xeon E5 and the Magny-Cours, respectively, using the best configuration for both scenarios (i.e., adjacent cores within the same die or processor). In both cases, `smdev` clearly ouperforms MPJ Express in latency and bandwidth. This is due to

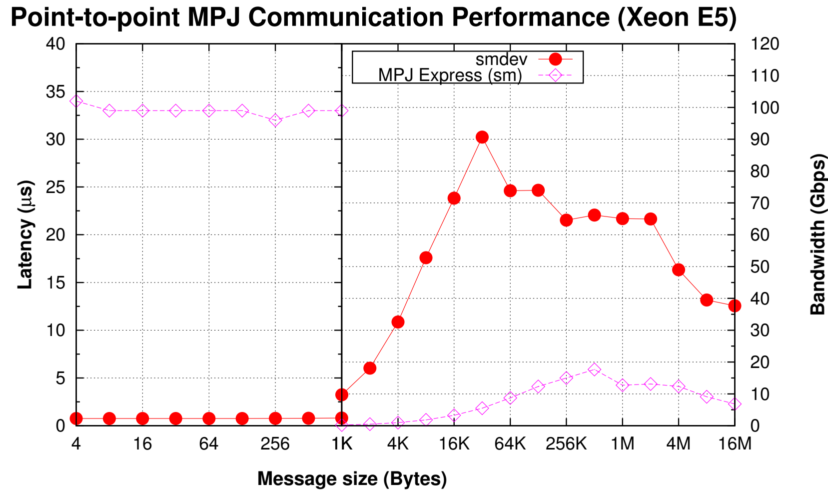


Figure 3.6: Message Passing in Java performance on the Xeon E5

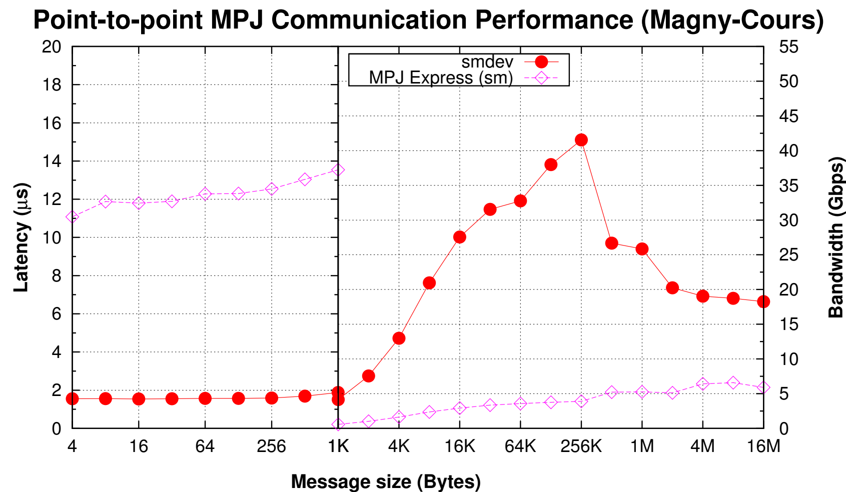


Figure 3.7: Message Passing in Java performance on the Magny-Cours

the avoidance of double buffering and the optimization of the synchronizations in `smdev`.

Since most Java communication middleware (e.g., JMS and RMI) is based on sockets, `smdev` performance has also been evaluated comparatively against sockets using the NetPIPE benchmark suite [90, 139] on the Xeon E5 (Figure 3.8) and the

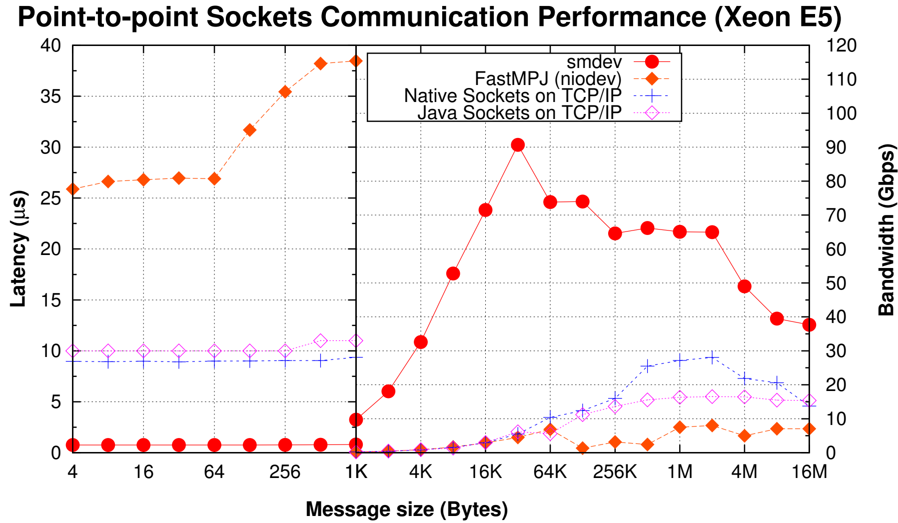


Figure 3.8: Sockets performance on the Xeon E5

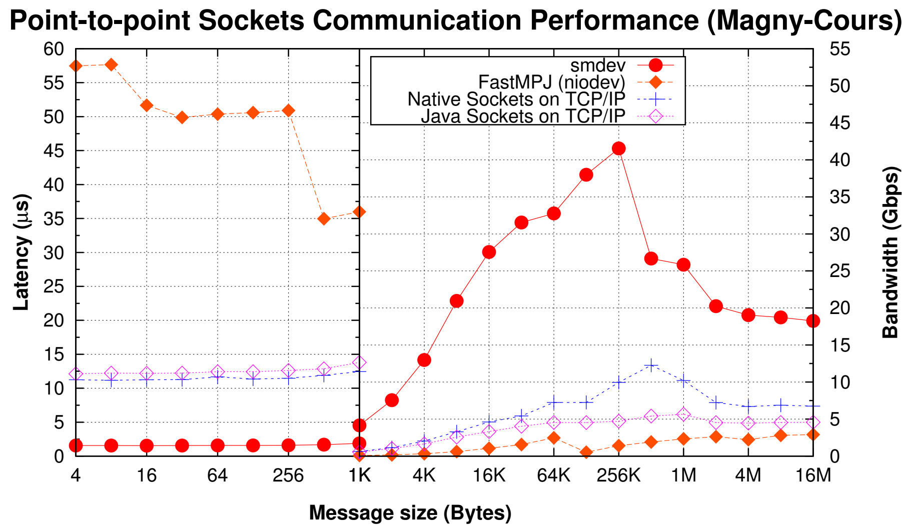


Figure 3.9: Sockets performance on the Magny-Cours

Magny-Cours (Figure 3.9). NetPIPE implementations in Java and C (native) sockets perform a pingpong test similar to the one implemented for message-passing benchmarking, so their results are directly comparable with those obtained with `smdev` and the NIO-socket communication device from FastMPJ (`niodev`, see Figure 1.1).

This benchmarking has been carried out with the default JVM/OS mapping policy, scheduling threads in intra-processor and intra-die configurations. As shown in the figures, sockets show significantly poorer shared memory performance than `smdev` as they rely on the TCP/IP loopback interface, suffering from significant latency overheads and bandwidth limitations due to the use of several communication layers. Finally, the Java NIO-socket device presents the poorest performance since it is based on TCP/IP, and its nonblocking communication support imposes a high overhead.

In order to compare the results obtained with the theoretical bandwidth limit, Figures 3.10 and 3.11 show the results obtained with a benchmark based on the COPY benchmark from the STREAM2 suite [131] (used to measure memory bandwidth) using Java and a native language (C) both on Xeon E5 and Magny-Cours. This benchmark measures the bandwidth of memory copies using arrays of different sizes and it does not perform cache invalidation between iterations, thus performance heavily relies on the size of the different cache levels (in this scenario, a 32-KByte L1, 256-KByte L2 and 20-MByte L3 for Xeon E5; and a 64-KByte L1, 512-KByte L2 and 6-MByte L3 for Magny-Cours).

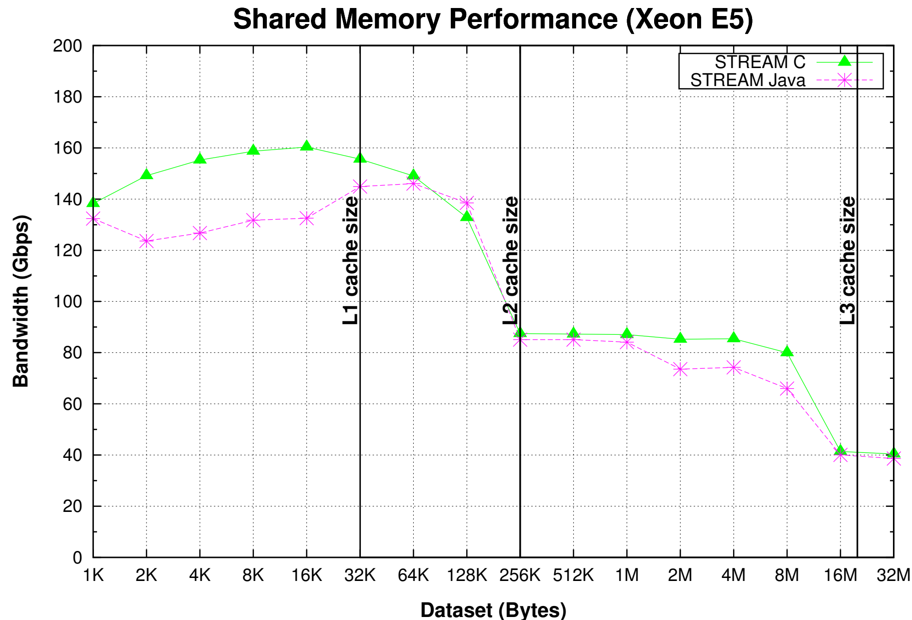


Figure 3.10: Memory performance on the Xeon E5

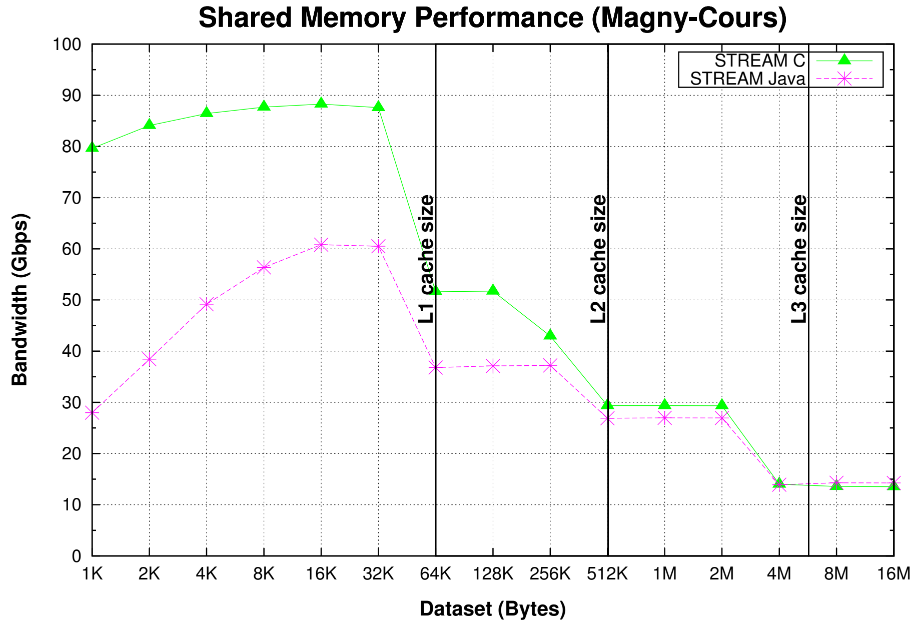


Figure 3.11: Memory performance on the Magny-Cours

These figures show that Java suffers from a higher start-up latency than the C implementation, especially on the Magny-Cours, which makes bandwidth decrease for small messages, whereas large-message performance is similar for both languages. When comparing these figures with the previous ones (Figures 3.8 and 3.9), it can be observed that `smdev` achieves bandwidths which are comparable to the theoretical limit exposed here by STREAM2.

3.3.3. Impact of `smdev` on the Scalability of Parallel Codes

The impact of `smdev` on the scalability of parallel codes has been analyzed with the NAS Parallel Benchmarks (NPB) [6, 88], which have been selected for their representativeness in the evaluation of languages, libraries and middleware for scientific computing. The NPB implementations for MPI, OpenMP, Java threads and MPJ (NPB-MPJ) [79] have been used for this evaluation. Regarding NPB-MPJ (FastMPJ) codes, they have been executed both with `smdev` and the NIO-sockets support (`niodev`) in order to analyze the actual impact of `smdev` compared to sockets, the default communication solution in Java.

Four NPB kernels have been selected due to their communication intensiveness: CG (Conjugate Gradient), FT (Fourier Transform), IS (Integer Sort) and MG (Multi-Grid), measuring the performance with the class C data size. Furthermore, these kernels have been executed using 1, 2, 4, 8 and 16 cores (also 32 for Magny-Cours, not 48 as the kernels only work for a power-of-two number of cores).

Performance is shown in terms of speedup in Figures 3.12 and 3.13. With the aim of providing a reference of absolute performance, Table 3.1 includes performance in millions of operations per second (MOPS) for Java and native (C/Fortran) implementations on a single core. These results show that CG and IS obtain similar performance for both native and Java implementations, but there are important differences in FT and MG due to the JVM start-up overhead combined with the higher maturity of the native codes, which are more refined than the Java versions.

Table 3.1: MOPS of NPB codes on a single core

<table>
<tr><th></th><th></th><th>CG</th><th>FT</th><th>IS</th><th>MG</th></tr>
<tr><td rowspan="2">Xeon E5</td><td>Native</td><td>381.8</td><td>1179.9</td><td>59.0</td><td>1741.2</td></tr>
<tr><td>Java</td><td>379.8</td><td>695.3</td><td>52.6</td><td>1219.3</td></tr>
<tr><td rowspan="2">Magny-Cours</td><td>Native</td><td>201.3</td><td>711.4</td><td>58.6</td><td>847.6</td></tr>
<tr><td>Java</td><td>168.1</td><td>461.9</td><td>45.0</td><td>548.2</td></tr>
</table>

Regarding the reported speedups on Figures 3.12 and 3.13, `smdev` is the most scalable solution on the Xeon E5, and one of the best performers, together with MPI, on the Magny-Cours system. Java threads and FastMPJ over Java NIO sockets generally obtain the poorest results.

The CG kernel relies on point-to-point primitives to perform the communications, and `smdev` takes huge advantage of the optimized point-to-point shared memory transfers. However, contention in the access to shared structures and resources limits scalability for 32 cores on the Magny-Cours.

IS is the most communication-intensive code, making a frequent use of Alltoall and Allreduce operations. On the Xeon E5 there is a significant loss of scalability when the number of cores forces the use of the two processors (16 cores, i.e., 8 cores per processor), thus increasing the cost of communications.

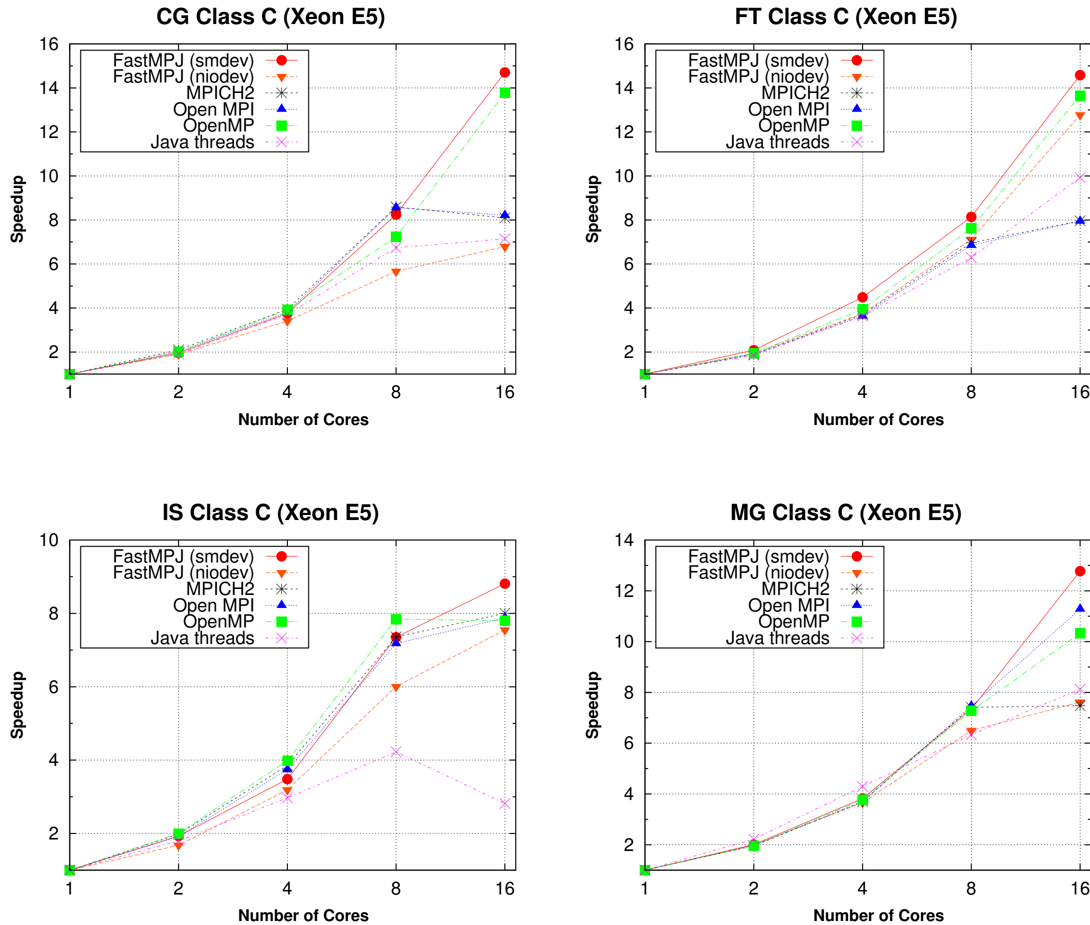


Figure 3.12: NAS Parallel Benchmarks performance on the Xeon E5

Regarding FT and MG, which showed the greatest performance gap between Java and native implementations using a single core, `smdev` generally obtains significant scalability on both systems except for MG on the Magny-Cours. MG heavily relies on Allreduce operations that may need further optimizations on FastMPJ to scale beyond 16 threads. OpenMP obtains its worst results for MG, especially on the Magny-Cours, where the messaging implementations (MPI and FastMPJ) exploit the Allreduce operation present in this kernel. `niodev` obtains its best scalability in FT because this is a computation-intensive code that, with large problem sizes (like the C size used here), limits the impact of communications on the overall performance.

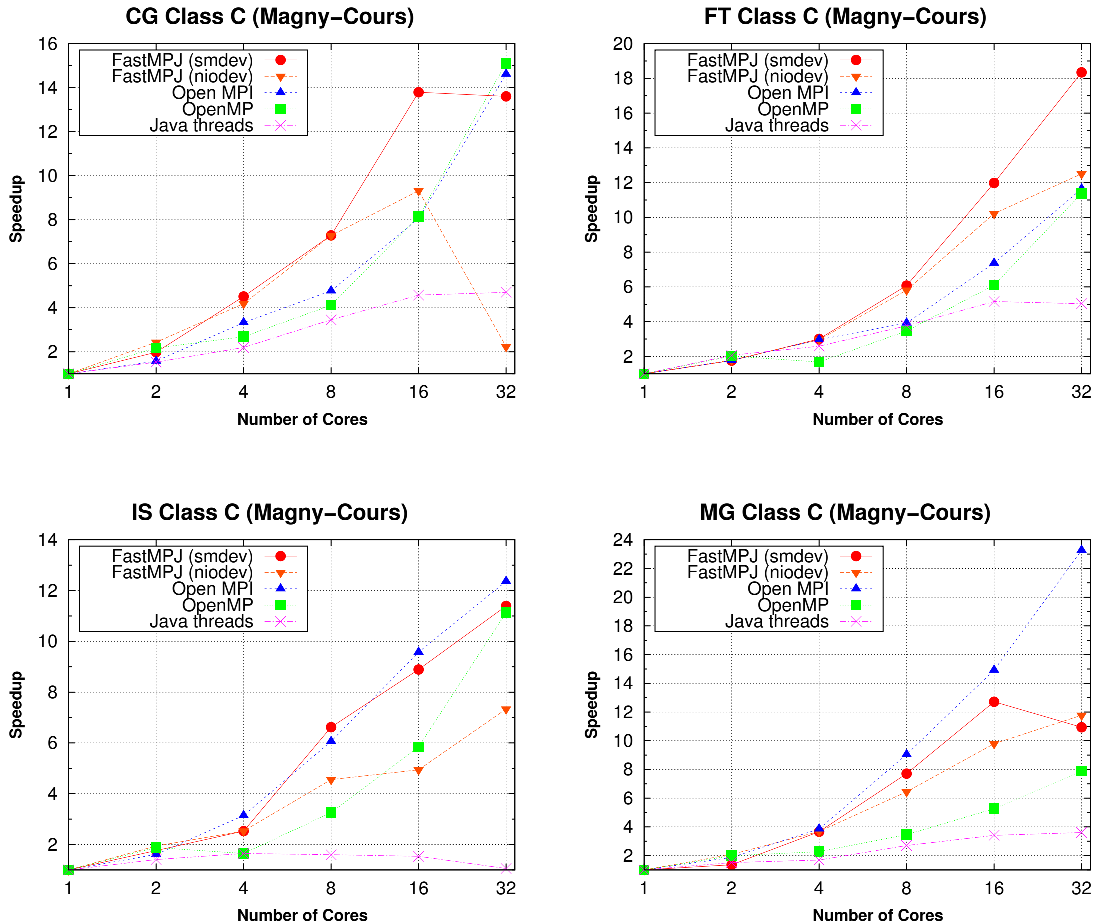


Figure 3.13: NAS Parallel Benchmaks performance on the Magny-Cours

3.4. Main Contributions of the smdev Device

This chapter has presented smdev, a shared memory Java communication device which provides a simple messaging API that abstracts thread programming while taking advantage of the inherent parallelism of multi-core processors and Java multithreading. This causes smdev to introduce performance improvements over the current alternatives in MPI and MPJ. This device has been successfully integrated in our Java message-passing implementation, FastMPJ. Hence, any MPJ application running on distributed memory systems can also run efficiently on shared memory systems thanks to relying on smdev. This shared memory communication middle-

ware has also been reported to provide high performance results when used in cloud HPC resources [30, 31], with similar or better efficiency than MPI libraries in shared memory scenarios.

The performance evaluation carried out on two representative shared memory systems, a 16-core Intel-based and a 48-core AMD-based, has shown:

(1) point-to-point start-up latencies as low as 0.76 μs, i.e. 13 times higher performance than Java sockets latency on shared memory (10 μs);

(2) point-to-point bandwidths higher than 90 Gbps, around 6 times better performance than Java sockets bandwidth on shared memory (around 15 Gbps);

(3) point-to-point performance results only around 1 μs worse than MPI for small messages, but significantly better for medium and large messages (from 2 KBytes on);

(4) the use of `smdev` in representative message-passing kernels (NPB) has generally achieved the highest speedups, which definitely helps to bridge the performance gap between Java HPC applications and natively compiled counterparts;

(5) `smdev` improves Java communications performance both on Intel- and AMD-based systems, taking advantage of their particular characteristics such as small and fast caches in the Intel-based testbed, and generally scaling performance on the 48-core AMD-based system.

In conclusion, the `smdev` device has been shown to be key for high performance Java applications on shared memory multi-core systems.

Chapter 4

Efficient Support of Collective Communications in Java

The increasing number of cores per system demands efficient and scalable message-passing communication middleware, not only for shared memory systems but also for clusters of multi-core processors. However, Message Passing in Java (MPJ) implementations have been focused on providing production-quality implementations of the full MPJ specification, rather than concentrating on developing scalable communications. The basic operations which benefit the most from these optimizations are collective operations. MPJ application developers use collective primitives for performing standard data movements (e.g., Broadcast, Scatter and Gather) 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 and reduces the risk of introducing bugs. Thus, collective algorithms 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.

This chapter presents the development of an MPJ collectives library which aims to provide an efficient solution for collective communications not only on shared memory multi-core systems (through specific collectives directly based on the `smdev` device presented in Chapter 3), but also on clusters of multi-core nodes. This

chapter also analyzes the feasibility and benefits of introducing the new nonblocking collectives from the MPI 3.0 specification in MPJ, discussing its performance results on a shared memory architecture. The developed MPJ collectives library has been integrated in FastMPJ.

4.1. Blocking Collectives for Multi-core Systems

MPJ collectives are an important part of any MPJ library, not only in terms of functionality but also in terms of development effort required. A wrapper MPJ library (e.g., Open MPI Java bindings) consists of a collection of wrapper classes that rely on an underlying MPI collectives library implementation, whereas a pure Java MPJ library, such as FastMPJ or MPJ Express, requires a full collectives implementation, usually on top of point-to-point communications. Generally each MPJ library comes with its own collectives library implementation, although it is possible to integrate third-party collectives, especially if they are based on MPJ point-to-point operations. Although it could seem that the translation of successful research in MPI collectives optimization into the MPJ arena would suffice, the particularities of the Java execution environment pose several additional challenges to the usually complex development of efficient collective operations. Thus, the new collectives library has to cope with the latency jitter, the JVM runtime execution behavior, the poor Java high-speed network latency, and the restriction to the use of point-to-point operations in the collectives implementation. These issues are discussed next.

MPJ collective primitives might show an important variation for the minimum, average and maximum latencies. This variability of their performance results over time is known as jitter, or latency variation. One of the most important factors for Java communications jitter is the JVM operation, especially the Just-In-Time (JIT) compiler and the JVM runtime execution. The impact of the jitter on the overall collectives performance is minimized by reducing the synchronization points in their algorithms.

Moreover, the JVM presents several execution modes, such as interpreted bytecode, and several levels of native code generation from the JIT compiler. The specific

performance of an MPJ collective call depends significantly on the JVM operation, which tends to further optimize commonly used methods with high overheads. Thus, it is quite common to find a collective operation with theoretically higher overhead outperforming a collective call with smaller message payload. This issue can be addressed reusing communication methods, or exploding recursion in order to make them eligible for further optimizations of the JIT compiler. Moreover, continuous changes in the communication protocols and algorithms, quite common in native MPI bindings, are also avoided for this reason.

Another challenge is that, except for FastMPJ, there is a lack of efficient high-speed network support in Java, due to its inability to control the underlying specialized hardware. Thus, MPJ libraries generally present lower performance than MPI, especially for small messages. In fact, when using TCP/IP over high-speed networks MPJ hardly obtains around 10% of MPI small-message performance, whereas it can achieve up to 90-95% of MPI bandwidth. As a consequence of this, most of the techniques used in MPI to speed up collectives performance, such as message fragmentation and synchronous protocols (rendez-vous protocol), are useless. Nevertheless, new optimization techniques arise, such as message aggregation and asynchronous operations (eager protocol). The efficient high-speed network support implemented in FastMPJ has been able to overcome most of these limitations, providing performance results quite close to those of MPI [29].

Finally, MPI implementations, looking for performance, can rely on collective operations implemented natively in the communication hardware or in low-level communication libraries, whereas MPJ libraries have been restricted to implement collective algorithms on top of point-to-point Java communication primitives. Although it is possible to rely on native collective methods through JNI, this option presents several additional drawbacks associated to the use of native methods (e.g., lack of portability, JVM security and instability issues, and significant JNI copy overhead). Nevertheless, since the `smdev` device has a pure Java implementation, it is able to provide collective operations based on the direct use of its shared structures (not relying on point-to-point communications) without causing any portability issues.

The development of efficient collectives taking into account the characteristics of the Java execution model constitutes the main research effort presented in this chap-

ter. The main contributions of the developed MPJ collectives library are: (1) the implementation of multi-core aware collectives through the minimization of internode communications and favoring multithreading-based solutions; and (2) the selection of the most efficient algorithm at runtime, based on the number of processes and the message size.

4.1.1. State of the Art of MPJ Collectives

As far as we know, this is the first work devoted to the optimization of MPJ collective communications, as up to now MPJ libraries have disregarded the development of scalable and efficient collective primitives. Moreover, the design and implementation of MPJ collectives libraries is usually discussed in the related literature together with their corresponding MPJ projects, as the collectives are an essential part of their tightly coupled designs. Therefore, although a few papers consider MPJ collective communications, their significance and impact on Java HPC performance has been reduced.

Hence, the most relevant related literature in MPJ collective communications comprises the papers that introduce the MPJ/Ibis [13], MPJava [105], and FastMPJ [29, 123] projects. Moreover, the collectives of MPJ Express are evaluated in [122]. Regarding MPJ/Ibis, its collectives library only implements one algorithm per primitive. Unfortunately, the selected algorithms are poorly scalable as they are usually based on blocking point-to-point communications (except for Alltoall/Alltoallv). Additionally, MPJava implements a subset of the MPJ collective operations, showing also poor scalability. Their performance results highlighted, for the first time in the MPJ community, the importance of choosing an appropriate collective communication algorithm according to the characteristics of the code being executed and the hardware configuration employed. Regarding the scalability of MPJ Express collectives library, several performance evaluations [116, 122] have pointed out that their results are generally poor due to the use of algorithms that do not take advantage of multi-core architectures and nonblocking communications, in order to exploit data locality and overlap communications, respectively. Finally, our own MPJ implementation, FastMPJ, includes the scalable collectives library implemented on top of point-to-point calls to a low-level communication device (explained later in this

section). Moreover, the FastMPJ collectives library implements several algorithms per primitive, selected at runtime.

The collectives library that has been developed improves previous MPJ collective implementations (combining different techniques) by: (1) providing multi-core aware collective primitives; (2) implementing up to five algorithms per collective operation; (3) selecting the most scalable algorithm at runtime depending on the specific multi-core system architecture and the number of cores and message size involved in the collective operation; and finally (4) allowing an easy integration with any MPJ implementation as it is based on MPJ point-to-point standard operations.

Most of the contributions of the implemented library have been motivated by the success of related works in native message-passing libraries, where far more research has been done. Thus, our library has adapted the research in MPI to MPJ, taking into account the particularities of Java, namely high variations in the communication latencies (jitter), the JVM execution modes and the impact of the JIT compiler on collective communications, and generally a high start-up time in communications.

Regarding the optimization of MPI collective operations, in [21] Chan et al. discuss thoroughly the design and high performance implementation of collective communications on distributed memory architectures, and [33] presents Tuned, a highly optimized collectives library included in Open MPI. In [127] Thakur et al. suggest the use of different algorithms in order to select the most scalable one for a particular message size and number of processes involved in the communication. Moreover, in [10] and [101] two model-based approaches for selecting the communication strategy that better adapts to a particular scenario are presented. As it is highly desirable that this selection can be made automatically, the efficiency of this process has been tackled in [102], obtaining less than a 5% performance penalty on average.

With respect to the efficiency of collective communications on multi-core architectures, several research lines have been explored. Thus, the hierarchical approach has provided significant performance increase for the Alltoall operation [113]. An extension of this strategy is a two-level intra-node and inter-node hierarchy [145], but this discrimination between intra-node and inter-node hierarchies scarcely increased performance because most of the overhead was in the inter-node communication.

However, the increase in the number of cores per node highlights the interest in these hierarchical approaches to take advantage of highly optimized shared memory communications. In fact, both MPICH and Open MPI include hierarchical collectives libraries. Recently, in [76], the use of kernel modules to improve intra-node communications has also been analyzed, and in [72] the use of a hybrid shared/distributed memory MPI approach, using multithreading for intra-node communications, provides perfomance improvements in collective operations. Furthermore, the optimization based on hierarchical virtual topologies presented in [138] has achieved significant performance gains thanks to the use of cache aware intra-node communications. Another approach is the maximization of the use of shared memory and the reduction of network communications on hybrid shared/distributed memory systems. This strategy usually provides significant performance advantages [133]. Finally, the effect on collective operations of the efficient placement of MPI processes in a multi-core system (runtime process attachment to specific cores) is discussed in [83].

Additional projects involving the optimization of Java collective communications for HPC are CCJ [89] and the Java Adlib collectives library [73]. CCJ is an RMI-based Java collective communication library for HPC which implements a simple low-level API. This library was intended to support collective operations in higher-level libraries, and thus MPJ/Ibis included CCJ collective implementations. However, CCJ is poorly scalable, mainly due to the use of RMI, and lacks widely used collective primitives such as Alltoall and reduction operations. Regarding the Java Adlib collectives library, it is a high-level collective communication library primarily focused on HPJava [73], a data-parallel Java programming environment for HPC, so its applicability to message-passing communications is quite limited.

4.1.2. Multi-core Aware Collectives

The main motivation of the work presented in this chapter is to implement a scalable and efficient MPJ collective communication library, taking advantage of communications overlapping, multi-core awareness, and runtime selection of the most appropriate algorithm, which depends on the message size and the number of processes involved in the communication. As MPJ performance heavily depends on

the scalability of collective communications, the implemented library is of special interest; thus, the design of the collectives library has considered especially the ease of integration in different MPJ implementations and so, aiming at portability, all collective algorithms are implemented using MPJ point-to-point operations. In fact, the library has been integrated not only in FastMPJ but also in MPJ Express.

Collective Communication Algorithms

The collective algorithms implemented in our library can be classified in six types, namely Flat Tree (FT), Four-ary Tree (FaT), Minimum-Spanning Tree (MST), Binomial Tree (BT), BiDirectional Exchange (BDE) or recursive doubling, and Bucket (BKT) or cyclic. These algorithms are thoroughly described in [21]. Although more complex algorithms (e.g. Fibonacci Tree for small-message Broadcast in specific environments) have been proven to be optimal under certain models [61], the higher start-up latency caused by the use of Java communications usually breaks down the latency improvements obtained with them.

The simplest algorithm is the Flat Tree (FT), where all communications are performed sequentially. For instance, in a Broadcast operation, the root process sends the message sequentially to all processes. We will distinguish between *b*FT and *nb*FT depending on whether the algorithm uses blocking communications or exploits the use of nonblocking primitives in order to overlap communications, respectively. As a general rule, valid for all collective algorithms, the use of nonblocking primitives avoids unnecessary waits and thus increases the scalability of the collective primitive. However, an intensive use of nonblocking operations could also collapse the network mechanisms that manage them. For instance, the version of the Alltoall primitive based on nonblocking operations has three variants: using both nonblocking sends and receives, using nonblocking sends and blocking receives, and using blocking sends and nonblocking receives. Although it might seem that the use of both nonblocking sends and receives is the most scalable solution, this option has some drawbacks, such as the contention and congestion that might appear in the underlying communication layer (thus causing serious performance bottlenecks), as this algorithm involves an important number of messages.

Four-ary Tree (FaT) algorithms configure a tree in which each node has four

successors at most. Hence, the communication operation consists of traversing this "four-ary" tree. This algorithm can be easily configured to use a different number of descendants. Although it prevents the communications from being performed serially, it does not minimize communications between processes that are located in different sockets or processors.

Minimum-Spanning Tree (MST) algorithms present a binomial tree in which the total number of processes involved in the operation is recursively halved and each half has a process that acts as root and communicates with the other half. If the operation is a Broadcast, after each division the root process sends its message to a process of the other subset, that becomes the root for its subset. This task is recursively repeated until all processes have performed their expected operations. MST algorithms minimize the communications between distant processes, but recursion causes implicit synchronizations that can reduce the performance gain. Although this algorithm can be implemented avoiding recursion, since the JVM usually compiles functions that are called several times, the collective would benefit from having recursive calls. Figures 4.1 and 4.2 show MST operations for a Broadcast and Gather communication pattern, respectively. The system represented uses three levels of core encapsulation (with different communication costs) represented by boxes, where the smallest ones are the cores; for instance, two dual-core dual-socket processors, or a dual-socket processor with two dual-core dies. Additionally, the thickness of the arrow symbolizes the communication cost of a point-to-point communication, i.e. the thicker the arrow the higher the communication cost. This representation will also be used for the description of the communication operation of the following algorithms.

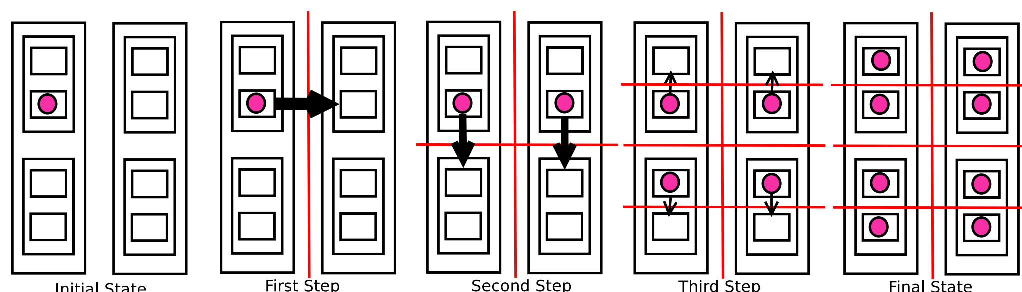


Figure 4.1: Overview of the Broadcast MST algorithm

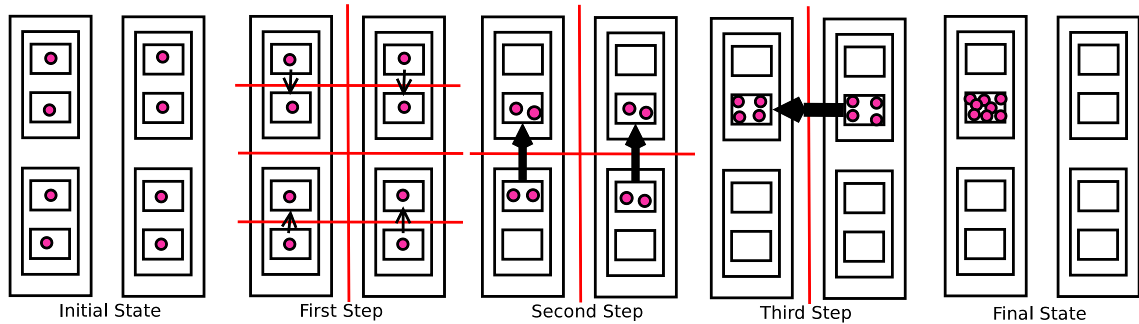


Figure 4.2: Overview of the Gather MST algorithm

Regarding the BiDirectional Exchange (BDE) or recursive doubling algorithm, it subdivides the set of processes just like MST, but communications are performed in a bidirectional manner: each process selects a counterpart from the other subset in order to perform a communication exchange, as shown in Figure 4.3 for an Allgather communication pattern.

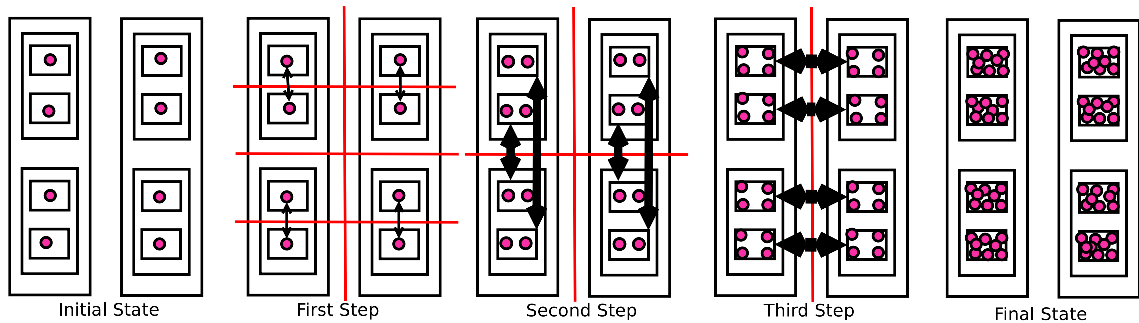


Figure 4.3: Overview of the Allgather BDE algorithm

The BDE algorithm can also be seen as a particular case of the Bucket algorithm (BKT). In BKT all processes are organized like a ring, and they send at each step data to the process on the right. Thus, data eventually arrives to all of them. Figure 4.4 shows an example of the algorithm for the Allgather primitive. The main difference between BDE and BKT for a given scenario lies on the number of steps involved in the communication, showing logarithmic and linear complexity for BDE and BKT, respectively. Moreover, the size of the message recursively increases for BDE, while it remains constant for BKT. Regarding the suitability of these

algorithms, a shared memory environment would benefit from the parallelism in BKT communications, whereas the aggregation of small messages into a larger one in BDE significantly improves performance on high-latency networks. Finally, these algorithms are much more scalable than the Flat Tree ones.

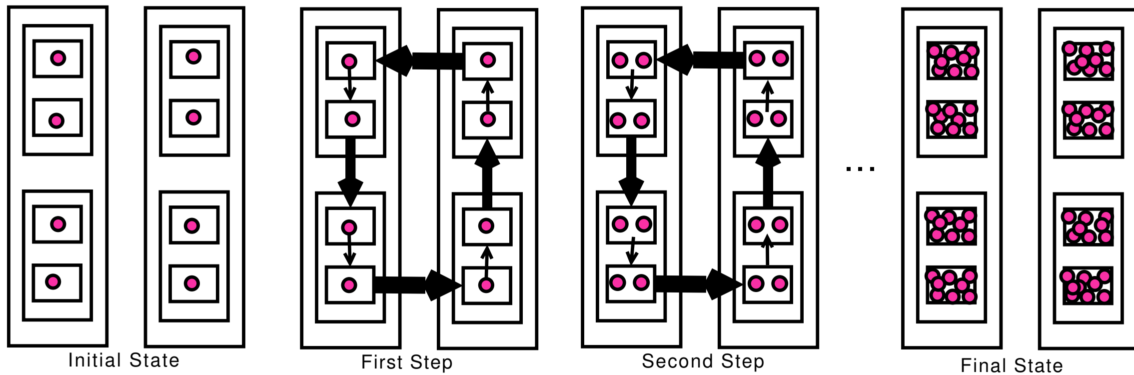


Figure 4.4: Overview of the Allgather BKT algorithm

Table 4.1: Algorithms implemented in the MPJ collectives library

Primitive	**Algorithms**
Barrier	Gather+Bcast, BT
Broadcast (Bcast)	*b*FT, *nb*FT, FaT, MST, Scatter+Allgather
Scatter	*nb*FT, MST
Scatterv	*nb*FT, MST
Gather	*b*FT, *nb*FT, MST
Gatherv	*b*FT, *nb*FT, MST
Allgather	*nb*FT, BT, BDE, BKT, Gather+Bcast
Allgatherv	*nb*FT, BT, BDE, BKT, Gather+Bcast
Alltoall	*b*FT, *nb*FT, *nb1*FT, *nb2*FT
Alltoallv	*b*FT, *nb*FT, *nb1*FT, *nb2*FT
Reduce	*b*FT, *nb*FT, MST
Allreduce	*nb*FT, BT, BDE, Reduce+Bcast
Reduce-Scatter	BDE, BKT, Reduce+Scatter
Scan	*nb*FT, linear

Table 4.1 presents a list of the collective algorithms implemented in the collectives library. The implementation variants are correlatively numbered (e.g., *nb1*FT

and *nb2*FT are variants of *nb*FT, as it was previously explained for the Alltoall primitive). As it can be seen, the high number of algorithms implemented allows a wide selection, being feasible to take advantage of the underlying hardware. Besides FT algorithms, it is assumed that nonblocking point-to-point primitives are used when communication overlapping could benefit the overall performance. The Scan primitive includes an algorithm, named linear, that has not been mentioned before. This algorithm is similar to BKT, i.e., data is received from the left neighbor, combined with the own data and sent to the right neighbor. However, there is no wrap-around communication and thus the first process does not receive data from the last one.

Thread-based Collective Primitives

Current MPI libraries usually support hybrid shared/distributed memory communications by the combination of communication devices, implementing intra-node communications through inter-process mechanisms such as sockets and System V IPC shared memory. However, the combination of two or more communication devices (e.g., `smdev` and `niodev`) has not been implemented yet in any MPJ library. Our previous work [109] explored this possibility using MPJ Express, but the synchronization overhead required for recovering the network messages, combined with the lack of efficiency and scalability of the shared memory support from MPJ Express, made it difficult to scale performance with hybrid communication devices in Java. Unlike MPI, where processes only share memory through shared locations explicitly defined, `smdev` implements several MPJ processes within a single JVM. Thus, combining `smdev` with a network device implies an increase in contention and in the need for synchronization, limiting the scalability of a hybrid MPJ device. In order to overcome this limitation the best approach is to include shared memory optimizations in network devices, such as for the InfiniBand support, using several processes per node. Hence, to take full advantage of the underlying hardware, we explored the simultaneous use of the message-passing paradigm together with thread-based solutions. This hybrid message-passing/multithreading approach requires a thread-safe MPJ implementation, and FastMPJ supports the highest level of thread safety, `MPI_THREAD_MULTIPLE`, like MPJ Express, which has already been used for hybrid message-passing and multithreading programming such as Java threads or Java OpenMP implementations (e.g., JOMP) [116].

However, although this hybrid programming approach can provide good performance, it requires the use of two programming paradigms, increasing the complexity of parallel programming. In order to explore the feasibility of this option, the developed collectives library has been extended in order to support this approach. This thread-based message-passing collectives library will serve as a proof of concept of the performance benefits that can be obtained following the aforementioned approach.

This library extension has been implemented by spawning multiple threads per MPJ process, with only one process per cluster node. Additionally, one thread per node is selected (`rootThread`) in order to be in charge of the inter-node message-passing communications. Furthermore, this thread will also serve as root thread for the intra-node execution of the collective operation. Thus, the thread-safety requirement of the developed thread-based message-passing collectives is limited to the support of the `MPI_THREAD_FUNNELED` level, which means that a process may be multithreaded but MPJ calls are only invoked by the thread that initializes the MPJ parallel environment (which internally sets up the communications among MPJ processes). Finally, in order to exploit data locality and affinity, each thread defines its storage space in its TLA (Thread Local Area).

Figure 4.5 presents the algorithm of the thread-based Broadcast, where `x` is the message, `rootProcess` is the root process, `rootThread` is the root thread within each process, `myThreadRank` is the rank of each thread within each process, and `NUM_THREADS` is the number of threads per process.

The data access synchronization is controlled by `ready` and `done`, `AtomicInteger` objects from the concurrency framework. An `AtomicInteger` is an int value (counter) that is updated atomically through several methods, three of them used in our library: `compareAndSet(int expect, int update)`, which atomically sets the counter to a given value if the current value equals the expected one; `incrementAndGet()`, which atomically increments the counter by one; and `get()`, which gets the current counter value.

This thread-based algorithm first broadcasts the data among the MPJ processes, involving only the `rootThread`. After this process-level operation, a thread-level intra-node Broadcast is performed. This operation consists of all the threads but the

```
Method Threaded Bcast(x,rootProcess)
AtomicInteger ready = new AtomicInteger(0);
AtomicInteger done = new AtomicInteger(0);
if myThreadRank = rootThread then
    Bcast (x,rootProcess);
    while not ready.compareAndSet(NUM_THREADS-1,0) do Wait ();
    while not done.compareAndSet(NUM_THREADS-1,0) do Wait ();
else
    ready.incrementAndGet();
    while ready.get()>0 do Wait ();
    System.arraycopy(x,0,TLA[myThreadRank].x,0,x.size);
    done.incrementAndGet();
    while done.get()>0 do Wait ();
```

Figure 4.5: Threaded Broadcast

`rootThread` copying in parallel the Broadcast message into thread-local variables. This local copy of the data is needed in order to exploit data locality, avoiding bottlenecks in shared memory accesses, as well as cache invalidation, quite common performance penalties in multi-core systems.

Portability and Selection of Algorithms

Our aim is to provide a portable library that can be easily integrated in any MPJ implementation. In order to achieve this goal, the communication algorithms have been implemented in the MPJ `Intracomm` class, which contains the collectives implementation according to the main MPJ API proposals (mpiJava 1.2 and JGF MPJ, see Section 1.2). In order to avoid dependencies that would break the portability of the developed library, all collective algorithms were implemented using MPJ point-to-point operations, which show almost no variation among the different APIs. In fact, the library was initially implemented for MPJ Express and now it is integrated in FastMPJ with the mpiJava 1.2 API. Moreover, in order to support the JGF MPJ API, implemented by MPJ/Ibis, the only changes required are renaming the MPJ package (e.g., for datatypes MPI.BYTE→MPJ.BYTE), as well as some methods (e.g., Send→send and Recv→recv).

The objective of the easy integration has therefore been fulfilled, as the de-

veloped library can be directly used from FastMPJ and MPJ Express, this latter considered because of its popularity, active development and, especially, for being distributed with a poorly scalable collective communications library. Moreover, the new collectives library is fully transparent to the user.

The runtime selection of the collective algorithm that provides the highest performance in a given multi-core system, among the several algorithms available, is based on the message size and the number of processes, defining a threshold for each of them. Moreover, these thresholds can be configured for a particular system by means of a benchmarking process, in order to obtain an optimal selection of algorithms, based on the performance results on a specific system and taking into account the particularities of the Java execution model.

This selection is stored in a configuration file (`collectives.properties`) that is loaded by a static initializer at MPJ initialization. This configuration file contains information about which algorithm has to be selected depending on the message size and the number of processes involved in the communication. The use of the selected algorithms is fully transparent to the user: if the `collectives.properties` file exists, the MPJ implementation will select the appropriate collective algorithm, otherwise a default algorithm will be used.

The configuration process consists of the execution of our own MPJ collectives micro-benchmark suite [124], the gathering of their performance results, and finally the generation of the configuration file that contains the algorithms that maximize performance. The results have been obtained on a power-of-two number of processes, up to the total number of cores of the system, and for power-of-two message sizes. The parameter thresholds, which are independently configured for each collective, are those that maximize, in relative terms, the performance measured by the micro-benchmark suite. Moreover, this tuning process is done once per system configuration, previous to the execution of the applications. A dynamic runtime system approach would present several drawbacks, such as a higher overhead of the algorithm selection process, and probably the use of a wider range of algorithms, something penalized by the JIT compiler, which in turn benefits commonly reused methods.

Finally, the overhead of the communication algorithm selection is reduced since it

is done through a light mapping method based on the number of processes and message size of a particular data transfer. This mapping is backed by a Java HashMap with approximately 60 keys and an access overhead of few microseconds ($<$10 μs), as it handles the different algorithms and scenarios represented by integer codes. Moreover, the access to the HashMap is not synchronized, as the algorithms do not change dynamically and there are not write operations.

Collective Operations for Shared Memory Systems in `smdev`

As mentioned before, the collectives library runs on top of MPJ point-to-point operations which actually run on the implementation of the point-to-point operations of the communication devices. In addition, `smdev` also provides, through an extension of the `xxdev` API (Section 3.2.1), its own implementation of collectives precluding to rely on point-to-point primitives. Having the collective operations implemented at the communication device level requires only a single call to `smdev` per collective and enables to optimize the use of the communication queues.

The design of these collectives starts from the thread-based implementation already explained, but using the `smdev` queues (see Section 3.2.3). The optimization in the use of the shared queues relies on the use of less explicit synchronizations, taking advantage of knowing in advance the communication pattern. As an example, in a Flat Tree algorithm for the Broadcast operation, the root thread relies on an atomic variable to indicate the state of an ongoing execution of a collective operation, and directly inserts a send request, which contains a reference to the message, in each `UnexpectedRecvQueue`. The rest of the threads, meanwhile, are waiting on another atomic variable to be notified that they can safely receive the message. Once the notification is received, they lock their own queue to find the request, and copy the message directly from the reference left by the root. In this case, the use of busy waits as a notification system establishes the order of operation, avoiding the need to check the queues for already arrived messages. Listing 4.1 presents the pseudo-code of this Broadcast for a more detailed explanation. Similar algorithms have been implemented for the remainder of collective operations.

These `smdev` collective operations do not rely on point-to-point primitives but on specific structures from the device. Since `smdev` is already included in FastMPJ,

it was possible to integrate these shared memory collectives into the mechanism for selecting the most appropriate algorithm at runtime.

Listing 4.1: Pseudo-code of the Broadcast method of `smdev`

```
void bcast(Object buf, int root, int tag){
  if(me == root){
    // wait if there is any other collective running and set it to busy
    busy-wait(other_collective,busy);
    for(i=0; i<nthreads; i++){
      if(me != i){
        requests[i] = new Request (...);
        if(buf.isObject())
          requests[i].buf = copy_serialized_data(buf);
        else
          requests[i].buf = buf; //let a reference to the buffer in the request
        synchronized(lock[i]){ //lock both queues of thread i
          unexpectedRecvQueue[i].insert(requests[i]);
        }
        other_threads[i].set(written); //wake up the rest of threads
      }
    }
    waitall(requests);
    collective.set(finalized); //notify that this collective has finalized
  }
  else{
    busy-wait(written); //wait until the root has written the message in the queue
    synchronized(lock[me]){//lock my queues
      request = unexpectedRecvQueue[me].find_and_get_message();
    }
    if(request.isSerialized())
      buf = deserialize_and_copy(req.buf);
    else
      buf = copy(req.buf);
    request.setCompleted(true);
  }
}
```

4.1.3. Performance Evaluation

This section presents the performance evaluation of the multi-core aware collectives library, first on a multi-core cluster with 128 cores organized in a hierarchy of multiple levels, and second, on two representative shared memory systems. A micro-benchmarking of collective primitives is presented for both scenarios. Moreover, for the multi-core cluster, an analysis of the impact of the use of the collectives library on the overall performance of two representative MPJ codes is also presented. The

analysis of these codes using `smdev` was shown in Section 3.3.3.

Micro-benchmarking of MPJ Collectives on a Multi-core Cluster

The evaluation of the developed library has been carried out on the DAS-4 cluster (ASCI, Vrije University Amsterdam) using up to 16 nodes, each of them with a dual-socket Intel Xeon E5620 quad-core Nehalem Westmere processor at 2.4 GHz and 24 GBytes of RAM. Each core has a 32-KByte L1 data cache and a 256-KByte L2 unified cache, and each processor has a 12-MByte shared Intel Smart cache. The interconnection network is InfiniBand (16 Gbps of maximum theoretical bandwidth) with OFED driver v2.7.0. The OS is CentOS v6.3, the C compiler GNU v4.4.6, the JVM Oracle v1.7.0, and the message-passing libraries are FastMPJ v1.0 (internal release), Open MPI v1.6.4 and MVAPICH2 v1.6. The performance results on this system have been obtained running up to 128 processes, distributing them evenly among the different nodes (e.g., for 128 processes, 8 processes are run per node), except for the thread-based collectives, which use one process per node and 8 threads per process (hence 128 threads).

Figures 4.6 and 4.7 present the aggregated bandwidth results (using 128 cores) obtained by four representative MPJ collective operations, Broadcast, Allgather, Reduce (sum) and Allreduce (sum) for the different algorithms implemented in the library (see Table 4.1) and the proposed thread-based collectives (labeled as "Threaded"). In order to micro-benchmark collective operations our own MPJ micro-benchmark suite [124], similar to the Intel MPI Benchmarks, has been used due to the lack of suitable micro-benchmarks for MPJ evaluation. The aggregated bandwidth metric has been selected as it takes into account the global amount of data transferred. The data used are byte arrays, avoiding serialization (the process of transforming an object into a byte array for communication) in order to present the collectives performance without incurring in additional overheads that would distort the analysis of the results. For clarity purposes, Table 4.2 presents a summary of the configuration file that maximizes performance (i.e., the `collectives.properties` file mentioned in Section 4.1.2). The thresholds that determine the number of processes and message size vary depending on the primitive.

The presented results show that there are significant differences between the

Table 4.2: Algorithms that maximize collectives performance on the DAS-4 multi-core cluster

Primitive	**small message/ small number of processes**	**small message/ large number of processes**	**large message/ small number of processes**	**large message/ large number of processes**
Broadcast (Bcast)	MST	MST	MST	MST
Allgather	BT	BT	BKT	BKT
Reduce	*b*FT	MST	*b*FT	MST
Allreduce	*nb*FT	MSTReduce+ MSTBcast	*b*FTReduce+ MSTBcast	MSTReduce+ MSTBcast

implemented algorithms, especially between the thread-based and the rest of algorithms. Data movement collectives (Broadcast and Allgather in Figure 4.6) take advantage of thread-based collectives and the efficient exploitation of shared memory transfers. Among the rest of algorithms, these collectives benefit from the use of multi-core awareness. For instance, MST implements communication patterns that reduce the number of costly network communications while taking advantage of the high bandwidth of high-speed intra-node and intra-processor transfers. Thus, the MST Broadcast algorithm only requires, in the configuration used (a 16-node cluster), a maximum of 15 inter-node communications, whereas the Four-ary Tree (FaT) or the Flat Tree (FT) would need up to 120 inter-node transfers. Moreover, in FT, all communications are performed by the root and thus they have to be serialized. This number of inter-node transfers has been determined with the default mapping provided by the FastMPJ runtime, which consists of an even distribution of the processes among the cluster nodes, trying to maintain the adjacency of the process ranks. It is possible to define alternative mappings that would increase the FaT Broadcast performance, however, applications and algorithms usually exploit the default mapping and network locality, hence, it is quite likely that alternative mappings would negatively affect the overall performance.

Regarding reduction operations (Reduce and Allreduce in Figure 4.7), thread-based collectives provide less benefit since communications within each node have to be serialized due to the need for computation. As an example, in a thread-based Broadcast all threads can copy the message from the root at the same time

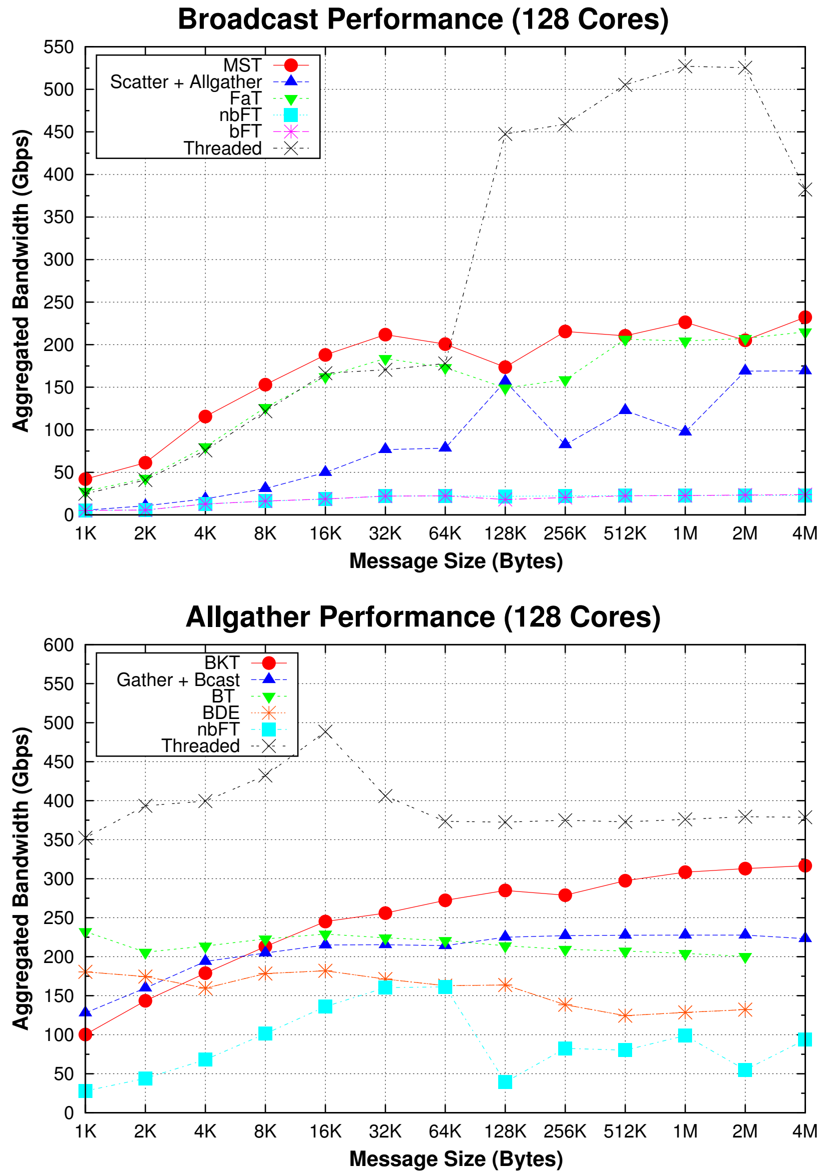


Figure 4.6: MPJ Broadcast and Allgather performance on the DAS-4 multi-core cluster

(see Figure 4.5) and, in a Reduce operation, the computation defines a critical region in which each thread uses its own data and the data from the root to carry out the computation, storing the result in the root's buffer. However, in MST the computation is performed recursively between pairs of processes in each step. This causes MST to reach and improve the aggregated bandwidth of the threaded

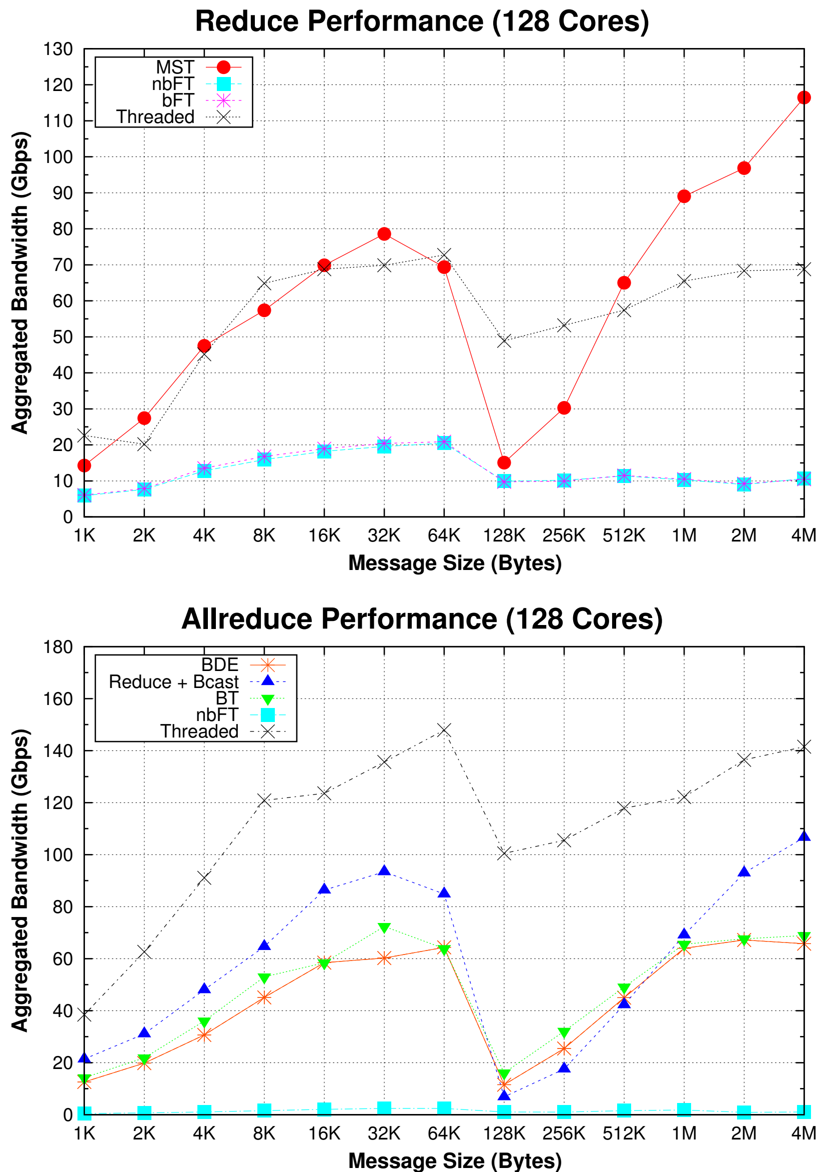


Figure 4.7: MPJ Reduce and Allreduce performance on the DAS-4 multi-core cluster

collectives for the Reduce operation. The threaded Allreduce involves an additional stage of communication, in which the result of the reduction has to be sent to all processes and threads, and hence the overhead introduced in the threaded reduction step has less impact on the overall performance.

The performance scalability of the selected collective primitives has also been

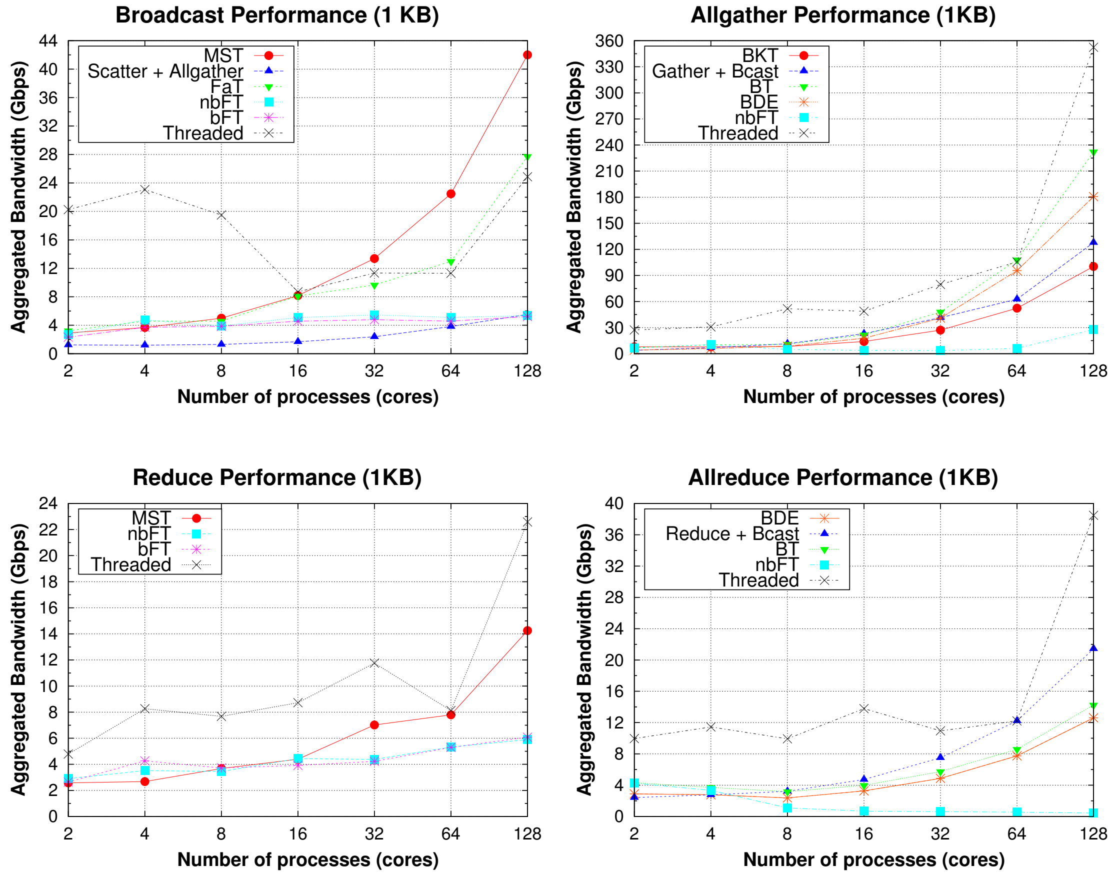


Figure 4.8: Scalability of MPJ collectives for a 1-KByte message

evaluated. Figures 4.8, 4.9 and 4.10 present the aggregated bandwidth for representative small (1 KByte), medium (32 KBytes) and large (1 MByte) message sizes, respectively. Smaller messages have not been considered as the relatively high start-up latency of the JVM makes the latency for a 1-KByte collective operation quite similar to the latency for a 1-Byte collective operation. These results confirm that the developed library presents generally good scalability, especially the thread-based implementation except for the Reduce operation, as explained before. The decrease in performance for 16 processes using the thread-based collectives (especially noticeable for the Broadcast operation) is caused by the need for communications between two different nodes. When only one node is involved, thread-based collectives only carry out one stage: the communications among threads; but when using more than one node, an additional stage of communication between nodes is needed. For large

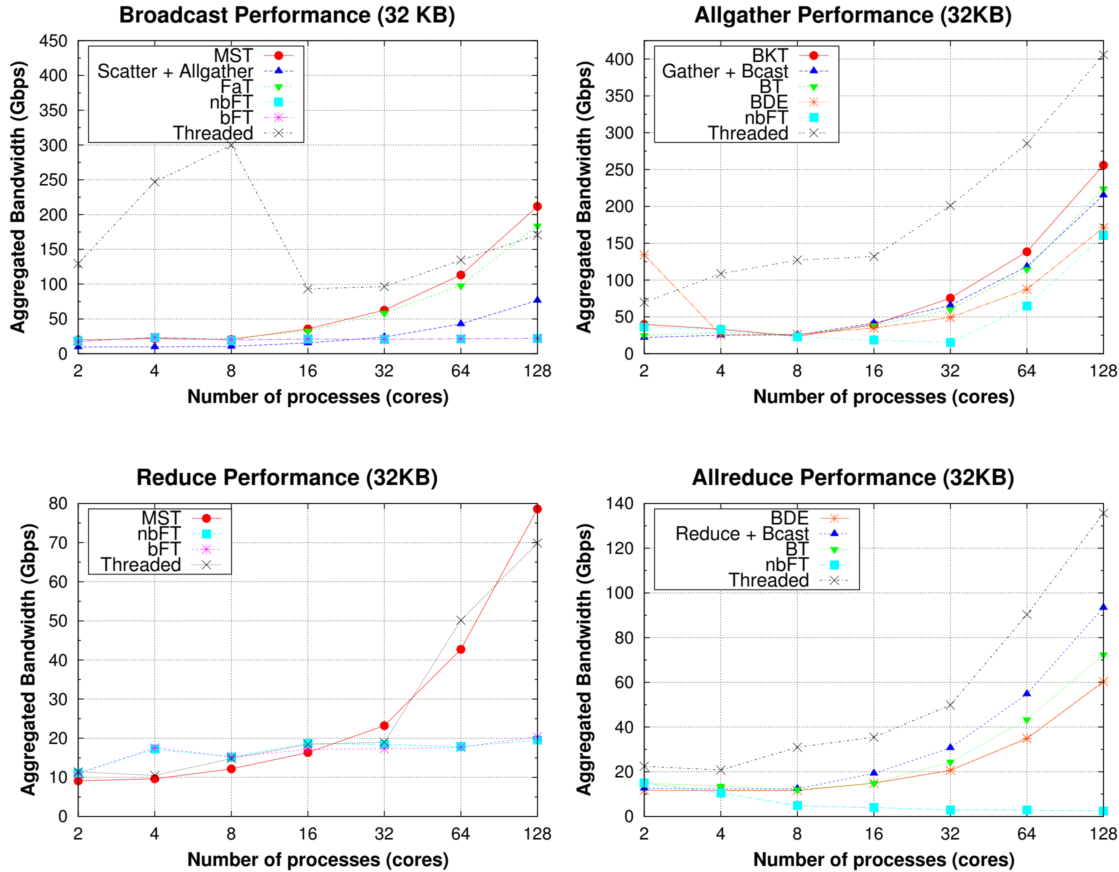


Figure 4.9: Scalability of MPJ collectives for a 32-KByte message

messages (1 MByte in Figure 4.10), it is also remarkable that for 8 processes the use of two different sockets with separate NUMA regions also influences perfomance in the data movement collectives (Broadcast and Allgather).

As a global conclusion of the micro-benchmarking analysis, it can be said that the multi-core awareness significantly improves MPJ collectives performance when compared to generic algorithms such as FT. Moreover, the use of threads has shown huge potential benefits in order to exploit the use of hybrid shared/distributed memory support in forthcoming FastMPJ communication devices. A comparison between the developed library and the collectives included in MPJ Express was presented in [122], where our library significantly outperformed MPJ Express collectives performance, due to the use of more efficient and scalable algorithms for multi-core architectures. However, most of the benefits of the implemented library come also from

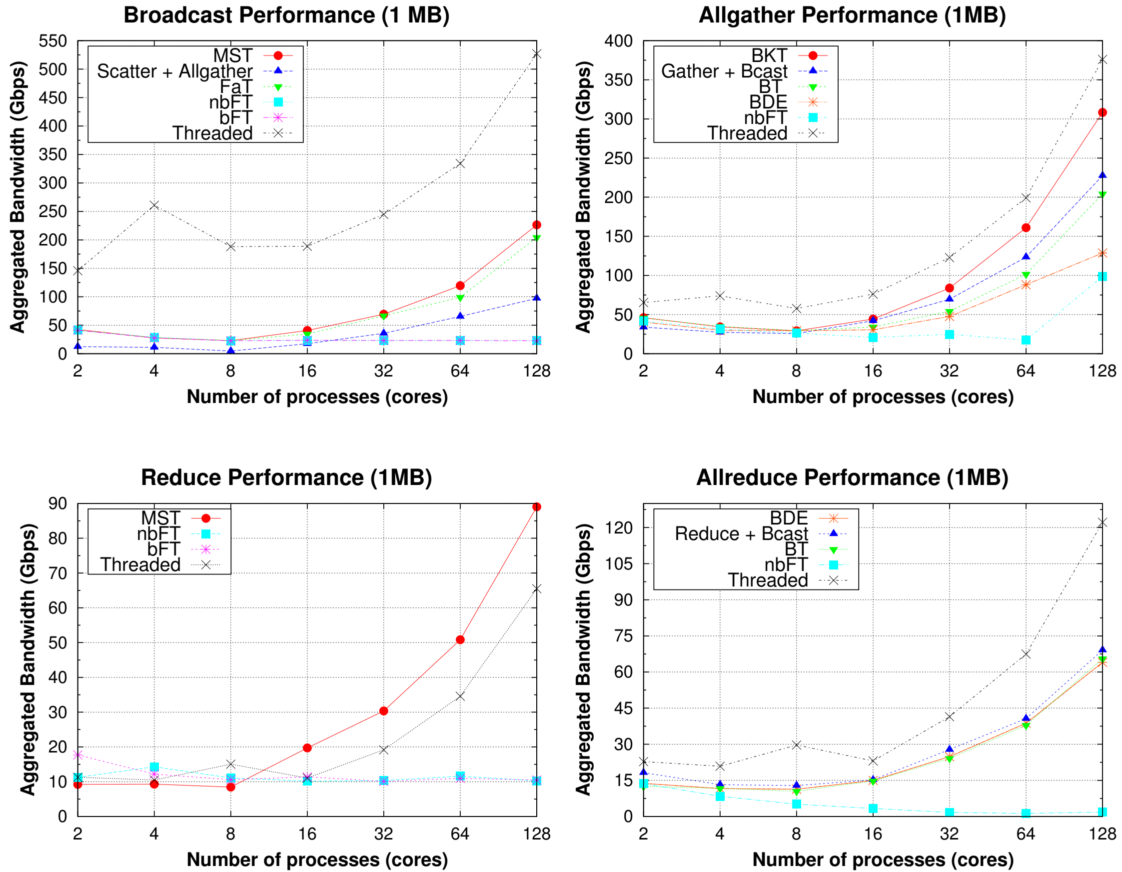


Figure 4.10: Scalability of MPJ collectives for a 1-MByte message

the automatic selection of the algorithm that maximizes the performance through the best adaptation of the communication pattern to the underlying architecture. As the library includes up to five algorithms per collective primitive, this feature is of special interest in a scenario of increasing complexity in the architecture of current systems.

Benchmarking of MPJ Kernels on a Multi-core cluster

The impact of the use of the developed library on representative MPJ kernels (IS and FT, using the class C workload) from the NAS Parallel Benchmark Suite (NPB-MPJ) [79] has been evaluated. These MPJ codes were selected as previous analysis of their performance showed poor scalability due to the collective primitives

overhead (Allreduce and Alltoall operations for IS, and Alltoall for FT) [79, 117]. In fact, these are communication-intensive codes, where additional factors, other than communications performance, are usually negligible (e.g., data locality exploitation).

Figures 4.11 and 4.12 present the performance of the selected codes in terms of speedup and MOPS, respectively. The performance of the MPI (MVAPICH2 and Open MPI) implementations of NPB IS and FT are also presented. As MPI collectives are highly optimized, the speedups obtained with them can serve as a reference for assessing the quality of the MPJ collectives library included in FastMPJ. For comparison purposes, performance results of FastMPJ using the basic MPJ collectives that were present on MPJ Express are also included (labeled as "FastMPJ (basic)"). FastMPJ does not support the use of hybrid shared/distributed memory devices to take advantage of multi-core clusters. Consequently, the use of thread-based collectives would require the implementation of the user codes with a hybrid MPJ/threads paradigm. Therefore, the thread-based collectives are not considered in the graphs as it would require the reimplementation of the NPB-MPJ kernels.

Regarding the measured speedups in Figure 4.11, as IS is a quite communication-intensive code, with several Allreduce, Alltoall and point-to-point communication operations, its scalability is low (in fact, the scalability using our MPJ collectives drops from 64 cores). However, our MPJ collectives always outperform the basic ones, especially from 32 cores. MPI libraries present better scalability than MPJ because both MVAPICH2 and Open MPI have a highly optimized collectives library and take advantage of hybrid shared/distributed memory transfers with hybrid communication devices.

With respect to FT, this kernel presents better scalability. The communication overhead of the FT kernel is heavily based on the Alltoall primitive, whose algorithm is a Flat Tree with underlying nonblocking point-to-point primitives (nbFT, see Table 4.1). Here the performance using our MPJ collectives is significantly better than using the basic ones, and only slightly worse than MPI. This is derived from the low performance of the Java sequential FT implementation (see Table 3.1), thus the performance improvements introduced by our MPJ collectives heavily impacts on scalability. This is evidenced in Figure 4.12, where there is a significant performance gap between MPI and MPJ that cannot be appreciated in a speedup graph (Figure 4.11). For IS, the difference between Java and C on a single core is

reduced (see Table 3.1), and thus there is almost no difference between Figures 4.11 and 4.12 regarding the relative behavior of MPJ and MPI.

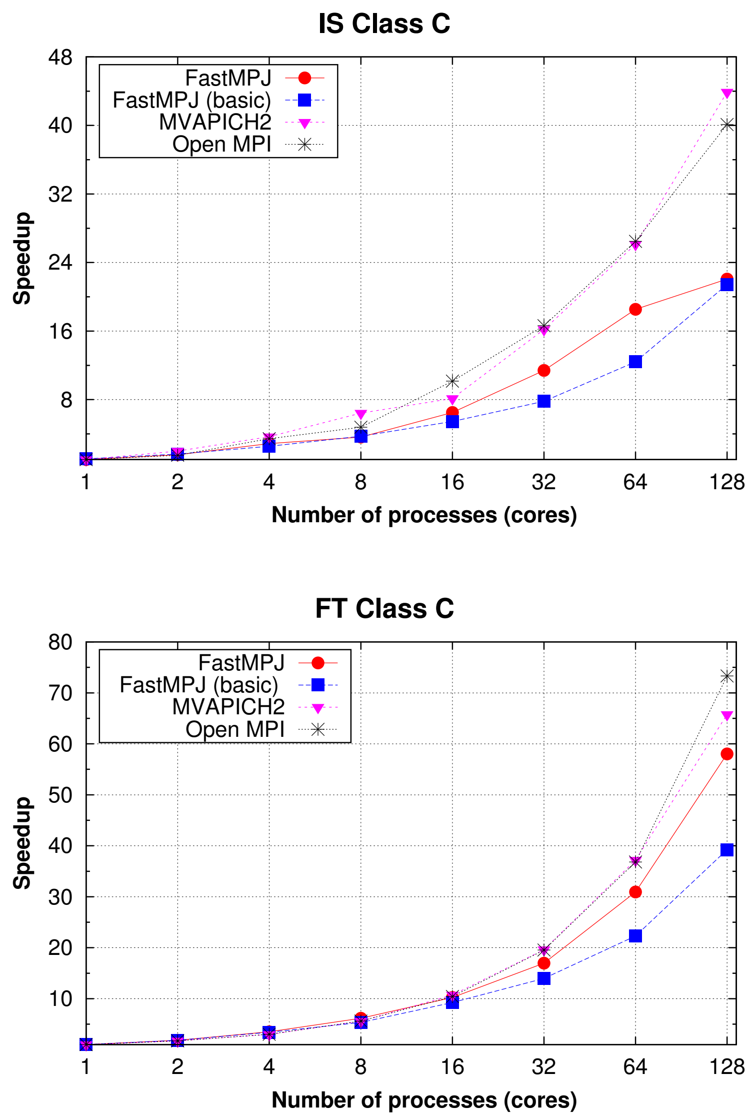


Figure 4.11: Scalability of NPB kernels on the DAS-4 multi-core cluster

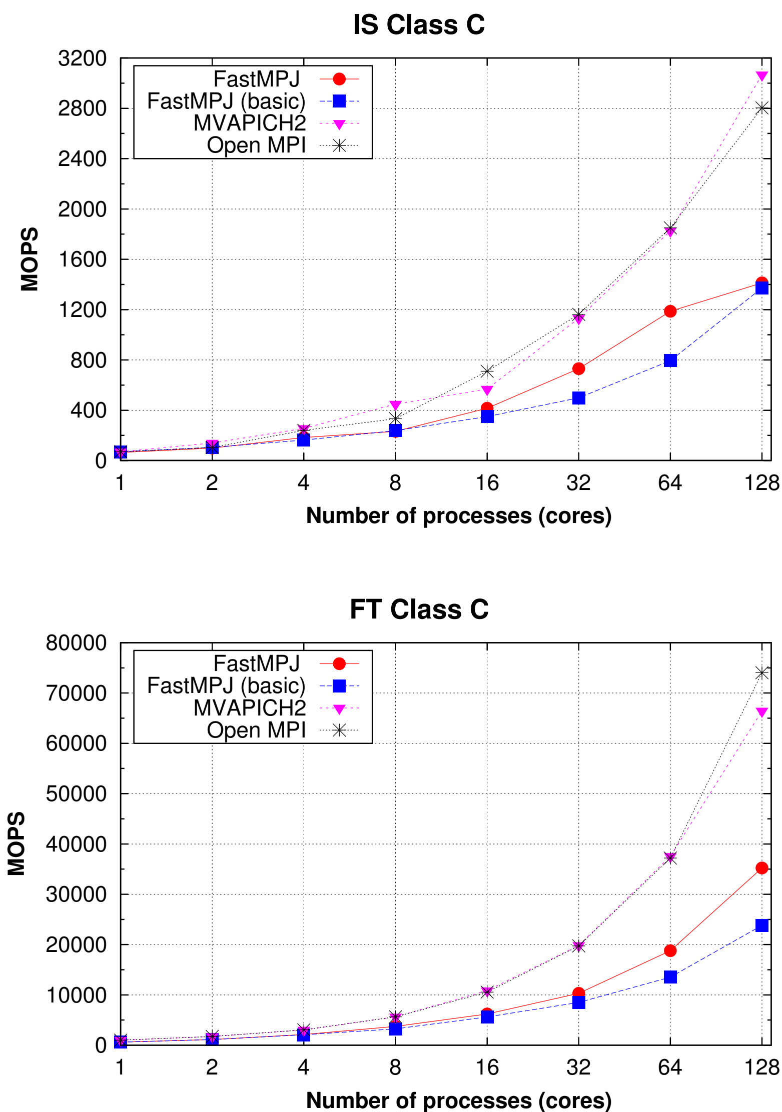


Figure 4.12: Performance (in MOPS) of NPB kernels on the DAS-4 multi-core cluster

Micro-benchmarking of `smdev` Collective Primitives on Shared Memory Systems

The `smdev` collectives have been benchmarked on the same representative shared memory systems used for the evaluation of the `smdev` device (see Section 3.3.1): (1) a 16-core Intel-based server, composed of two Intel Xeon E5-2670 8-core processors at 2.6 GHz and 64 GBytes of RAM (see Figure 2.1), and (2) a 48-core AMD-based

server, composed of four Magny-Cours AMD Opteron 6172 processors, each one with 12 cores and 128 GBytes of RAM (see Figure 2.2). The OS is Linux CentOs with kernel v2.6.35, the GNU compilers are v4.4.4 and the JVM is OpenJDK Runtime Environment v1.6.0_20 (IcedTea6 v1.9.8).

The performance of FastMPJ collectives, either implemented at MPJ level or relying directly on the `smdev` device, has been evaluated comparatively against two representative MPI implementations which provide efficient communication protocols on shared memory systems for natively compiled languages (C/C++, Fortran). The MPI libraries selected for this evaluation are MPICH2 v1.4 and Open MPI v1.4.3 on the Xeon E5, and Open MPI v1.4.2 on the Magny-Cours. MPICH2 results have been omitted for clarity purposes since Open MPI generally obtains better performance on the Magny-Cours. In order to present a fair comparison with FastMPJ when relying on `smdev`, these MPI implementations have been benchmarked using their shared memory communication devices: sm and KNEM BTL in Open MPI and Nemesis in MPICH2. Both libraries have been carefully configured in order to achieve the best performance results.

The aggregated bandwidth for the Broadcast, as representative data movement collective, has been measured on 8 and 16 cores on the Xeon E5 (Figure 4.13), and communicating 8 and 48 cores on the Magny-Cours testbed (Figure 4.14). Two algorithms have been used for FastMPJ: the specific `smdev` implementation, and the Minimum-Spanning Tree (MST) from the developed collectives library which relies on the point-to-point primitives from `smdev`.

The results from Figures 4.13 and 4.14 show that the FastMPJ `smdev` Broadcast generally obtains higher bandwidth than the MPI implementations thanks to relying on a zero-copy communication protocol, as it happened in `smdev` point-to-point transfers. In comparison with the MST Broadcast, it also shows the highest bandwidth, except for 48 cores on Magny-Cours, in which the contention in the access to the shared queues causes a performance decrease. In fact, the Broadcast of messages up to 1 MByte obtains lower aggregated bandwidth on 48 cores than on 8 cores of the Magny-Cours, showing a decrease in the scalability of the `smdev` implementation. The MST algorithm balances the load among the cores involved in the communication, which is a more scalable approach as it increases performance with the number of cores. However, this algorithm relies on several synchronizations that

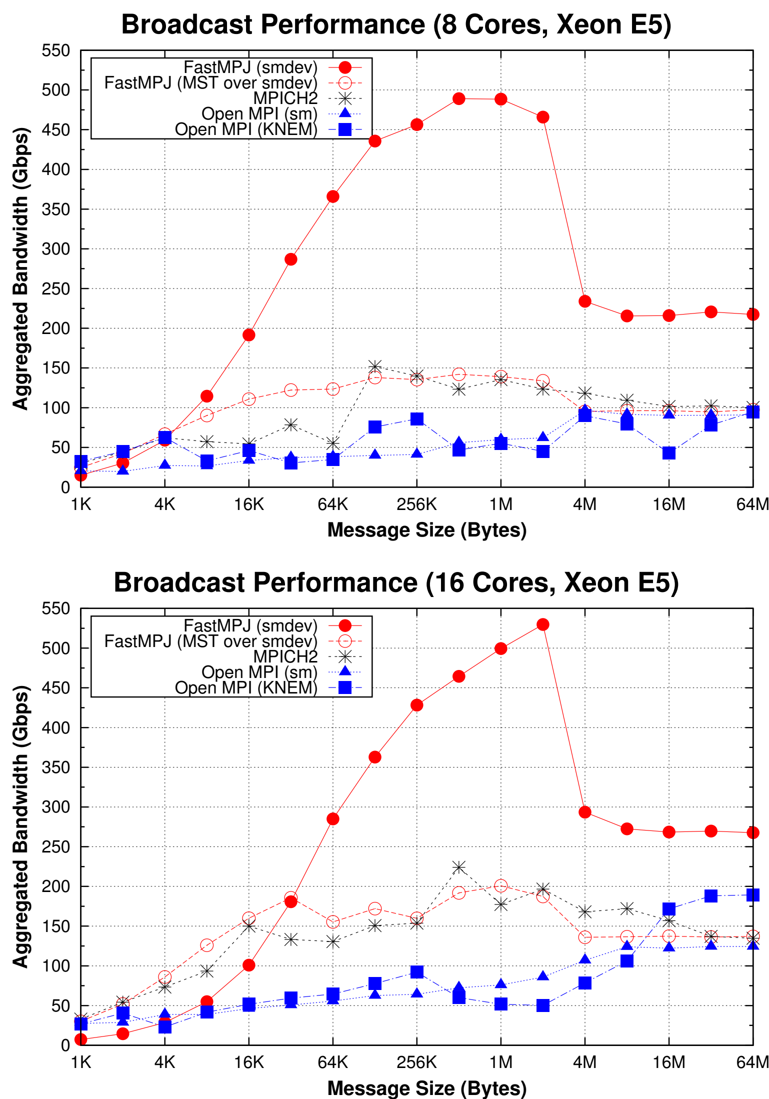


Figure 4.13: Broadcast performance on 8 and 16 cores (Xeon E5)

introduce an important performance penalty. The specific `smdev` implementation also shows poor performance for small messages on 16 cores of the Xeon E5 system. However, the FastMPJ library supports the selection of collective algorithms at runtime, thus the best algorithm is selected depending on message size and number of cores. As in point-to-point transfers, the performance of the specific `smdev` implementation drops on the Xeon E5 when the message cannot be fully stored in the L3 cache (from 2 MBytes). Finally, due to the lower computational power and

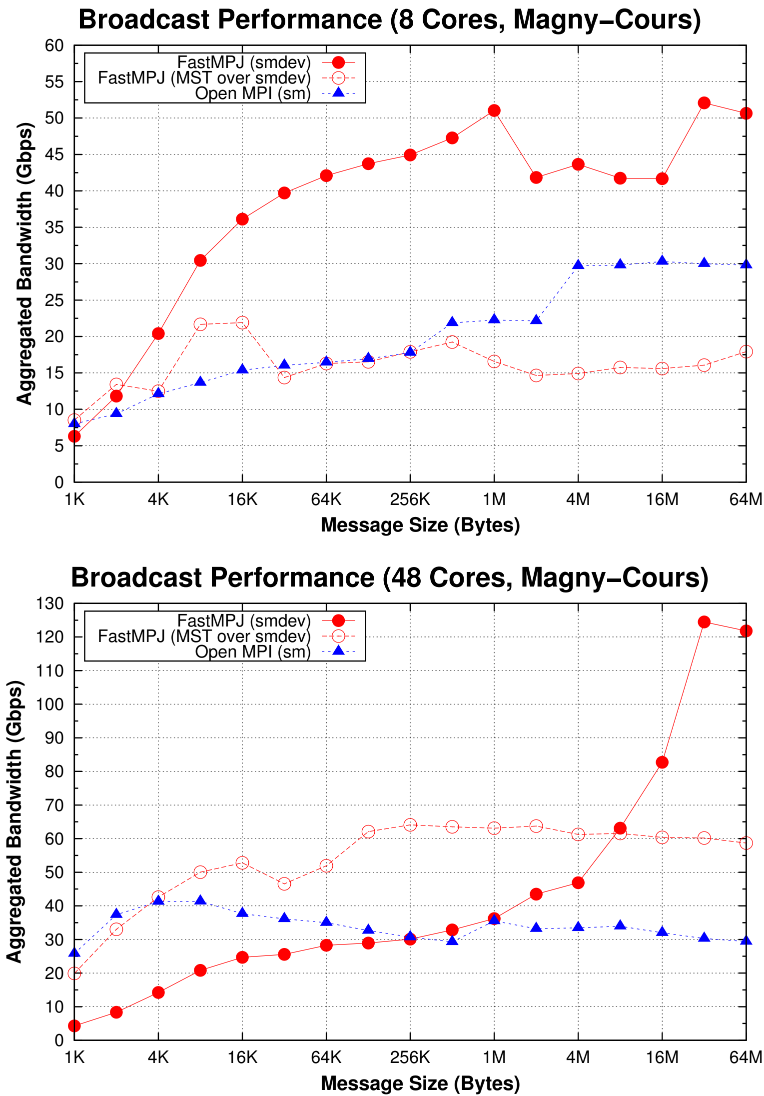


Figure 4.14: Broadcast performance on 8 and 48 cores (Magny-Cours)

memory performance of the Magny-Cours, the achieved bandwidth is significantly lower than on the Xeon E5.

In addition to the Broadcast, a collective operation whose performance is key in parallel and distributed codes is the explicit synchronization among several cooperating threads, generally known as Barrier. This operation is usually implemented as a combination of sends and receives that guarantees that every thread that calls

the barrier blocks until all the threads involved in the collective operation have reached this routine. The developed collectives library implements two algorithms for this operation. The first one, Binomial Tree (BT), organizes the threads into a binomial tree performing several communications across this tree that guarantee the synchronization among the threads. The second one, Duplex, uses a Broadcast (sending a token from one thread to the rest of threads) followed by a Gather (one thread receives an ACK message from each one of the rest of threads). The Java threading API also provides an implementation of the synchronization operation, the Cyclic Barrier, which implements the concept of Barrier using locks for synchronizing a configurable number of threads. Regarding MPI libraries, both Open MPI and MPICH2 use their own shared memory communication devices with the same considerations than for the Broadcast micro-benchmarking.

The performance of this representative collective operation has been analyzed on the Xeon E5 system (using 8 and 16 cores) and on the Magny-Cours system (using 8 and 48 cores). Table 4.3 shows the minimum and average latencies measured for the Barrier algorithms described above. As a direct consequence of the typical variability of the JVM measured results, some outliers increased significantly the average latency. Thus, the Java micro-benchmarks execute the Barrier operation inside a loop obtaining the minimum and the average latency, instead of using the Intel MPI Benchmarks (IMB) [110] approach: obtaining the average by dividing the total latency of the loop by the number of operations. The approach used for benchmarking the Java Barrier does not reduce variability but allows to obtain the actual minimum (in the IMB results, the minimum represents the "minimum average" of a set of repetitions of the benchmark).

As it can be observed, the BT algorithm generally achieves the best performance of the Java algorithms, both for the minimum and average latencies. The main reason is that it needs fewer communications than the Duplex algorithm and the cost of those communications is lower than the overhead of the explicit locks in the Cyclic Barrier, which involve context switches. Moreover, the minimum latencies of the BT and the MPI Barriers are very similar. Regarding MPI, both Open MPI and MPICH2 generally present the lowest latencies, and although the usage of the KNEM-based device shows significant variability, its minimum latency is the lowest one.

Table 4.3: Latency of Barrier algorithms (time in μseconds)

			Fast-MPJ (BT)	**Fast-MPJ (Dupl.)**	**Fast-MPJ (Cycl.)**	**Open MPI (sm)**	**Open MPI (KNEM)**	**MPICH2**
Xeon E5	8 cores	Min.	4.40	3.84	6.55	4.44	3.46	3.73
		Avg.	9.28	13.21	203.06	4.56	10.62	4.07
	16 cores	Min.	5.87	6.87	6.50	6.98	4.20	5.81
		Avg.	14.61	223.81	425.02	8.56	24.64	7.49
Magny-Cours	8 cores	Min.	7.16	8.64	12.74	6.14	–	–
		Avg.	11.71	21.62	192.43	6.97	–	–
	48 cores	Min.	14.41	23.27	16.96	28.49	–	–
		Avg.	752.51	127.72	1621.78	31.49	–	–

4.2. Nonblocking Collectives in Java

As seen previously in this chapter, communications may become one of the major bottlenecks in the scalability of parallel codes, especially as the number of cores per processor keeps increasing. Message passing reduces this communication overhead by overlapping communication and computation via nonblocking point-to-point primitives. Nevertheless, regarding collective operations, only blocking primitives have been traditionally supported by the MPI standard, forcing programmers to implement their own collective communications involving nonblocking point-to-point primitives when needed. This imposes higher costs of development, risks of bugs and lack of efficiency as it is not possible to take advantage of the highly optimized collective algorithms included in message-passing libraries, which usually exploit the underlying hardware. Thus, nonblocking collectives were proposed to be part of the MPI 2.2 standard but they were postponed until the recently published MPI 3.0 specification.

Blocking collectives impose an implicit synchronization that can be avoided by the adoption of a nonblocking paradigm for these primitives. Thus, a nonblocking collectives library, which is next described, has been developed and benchmarked to show that both overlapping of communication and computation, and the avoidance of extra synchronization improve performance of message-passing codes on shared

memory systems [107].

4.2.1. State of the Art of Nonblocking Collectives

The optimization of message-passing libraries for exploiting multi-core shared memory architectures has become a necessity due to the increase in the number of cores per processor. In fact, as explained in Chapter 3, MPI libraries such as Open MPI [130] and MPICH2 [17, 28], and MPJ implementations such as MPJ Express [116] and FastMPJ [108], include custom communication devices which exploit shared memory transfers. The advantage of Java over traditional languages in HPC (C, Fortran) is that it supports multithreading in the core of the language and shared memory programming naturally emerges from it, whereas natively compiled languages have to rely on the shared resources management of the operating systems.

Moreover, when the number of communicating processes is large, blocking communications generally impose a high overhead that can be overcome by overlapping communication and computation using nonblocking communications, and thus message-passing libraries provide nonblocking point-to-point primitives. The benefits of overlapping in message-passing libraries have been well studied. For instance, in [15] the authors analyze the benefits for an MPI library which supports overlapping with offloading and independent progress. Another work [69] presents a benchmark to assess the ability of hardware and software to overlap MPI communication and computation, whereas in [112] a theoretical analysis for scientific applications is shown, and in [103] the benefits of overlapping in an MPI-2 application are evaluated.

Since its inception, MPI aimed to provide nonblocking communications. In fact, in [26], while describing the MPI standard, the authors mention that not only nonblocking point-to-point primitives but also collective ones might be useful and should be included in subsequent versions. Nonblocking collectives were attempted to be part of the MPI 2.2 standard but they were finally put off since it would suppose a major change which would fit best with MPI 3.0 [86]. However, they have been explored ever since, and different MPI implementations provided their own suite of nonblocking collective operations. Some examples are Adaptive MPI [48], that

extends MPI to use virtual processors; the optimization of MPI collectives for the MPICH2-based library used in BlueGene/L [2] and BlueGene/P [66, 68]; or the Component Collective Messaging Interface (CCMI) [67], which is not an MPI implementation but provides a messaging interface with nonblocking collectives. The potential benefits of nonblocking collectives in different applications are analyzed in [14], and [36] studies how noise affects MPI performance, concluding that nonblocking collectives can help, and demonstrating this statement empirically through the evaluation of its own nonblocking allreduce implementation. 3D FFT (Fast Fourier Transform) has been stated to be able to take advantage of nonblocking collectives in [59] and [111]. Moreover, there is also a large number of projects which intend to provide low-level nonblocking support. The work [60] presents the implementation of a nonblocking Broadcast taking advantage of the Mellanox ConnectX-2 InfiniBand adapters that offer a task-offload interface (CORE-Direct), being evaluated with the High Performance Linpack (HPL) benchmark. The work [125] extends PAMI, a low-level messaging interface, to support the implementation of nonblocking collectives in Power7 IH supercomputers. KACC [93] is a new nonblocking communication facility implemented in the OS kernel interrupt context to perform nonblocking asynchronous collective operations without the help of an extra thread, and it was moved to the user level in uKACC [94], which uses the Marcel thread library and the PIOMan's scheduler from Madeleine [135, 136] to implement nonblocking collectives. In [114] the authors discuss a possible implementation of the flexible Group Operation Assembly Language (GOAL) framework to support nonblocking collectives. Additionally, PGAS languages like Unified Parallel C (UPC) also support them [92].

One of the most relevant projects related to nonblocking collectives is the LibNBC library [42, 45], which is being integrated in Open MPI. In its first version, each operation needs user interaction to progress, but micro-benchmark results show that overlapping computation and communication in collective operations could potentially provide significant performance improvements. The authors state the benefits that nonblocking collectives could add to MPI and show benchmarking results of their implementation, based on avoiding implicit synchronization and taking advantage of nonblocking features of modern network hardware. This implementation aimed to support a strong case in favor of the inclusion of nonblocking collectives in the MPI standard. The library has been optimized for InfiniBand in [44], and

the benefits and drawbacks of including an extra thread to manage progress instead of user interaction are evaluated in [43]. This work compares a polling strategy (beneficial when there are free CPU resources) with an interruption system using communications over InfiniBand. In [46] the authors present an analysis of the methodology for benchmarking nonblocking collective operations using overlapping in the latency measurements, which is estimated using both time and workload measurements. More recently, the successful approach of overlapping communication with costly computation has also been applied to I/O operations, as shown in [142].

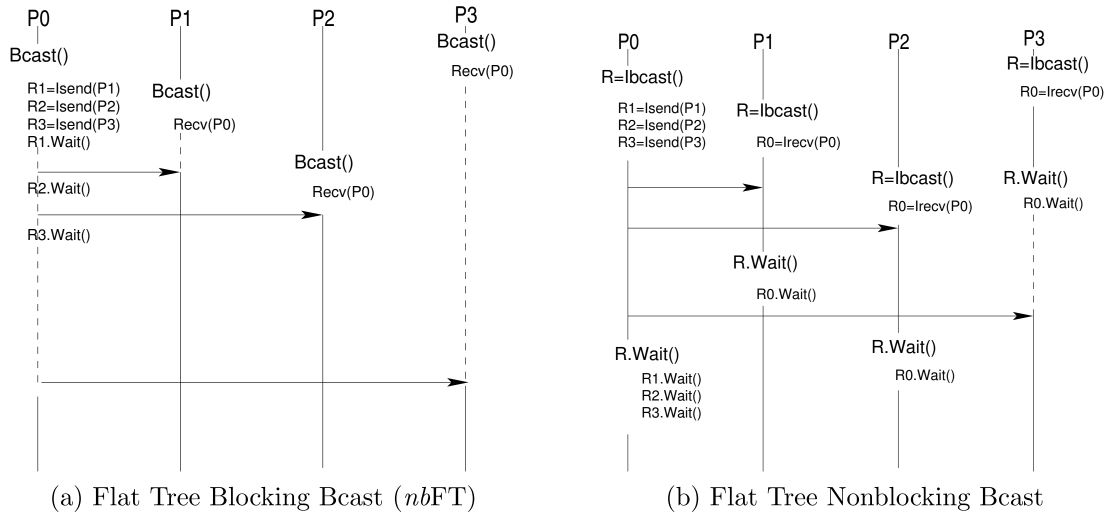
4.2.2. Nonblocking MPJ Collectives

The benefits of nonblocking collectives have been extensively studied for distributed memory systems and communication across the network. However, although shared memory architectures are becoming increasingly supported by message-passing libraries, there is no previous assessment of the capabilities of the shared memory communication support to take advantage of nonblocking collectives. With this purpose, a Java message-passing nonblocking collectives library has been developed and integrated in FastMPJ [29, 107], and then it has been benchmarked in a Sandy Bridge-based shared memory system.

Implementation of Nonblocking Collectives for FastMPJ

The blocking collective operations of FastMPJ described in Section 4.1.2 do not allow the overlapping of computation and communication and, furthermore, they impose implicit synchronizations. In an environment where threads and processes are supposed to perform independent workloads, any synchronization can potentially cause major overheads. An initial approach to the implementation of nonblocking collectives could be based on a Flat Tree algorithm upon nonblocking point-to-point primitives. Figure 4.15 compares this initial approach with its blocking counterpart (using also nonblocking point-to-point primitives, see *nb*FT in Table 4.1) for a Broadcast in an example scenario with four processes. Dotted lines indicate that the process has to wait and it is not able to perform any other computation while the operation is not complete, whereas continuous lines represent useful computation.

Figure 4.15a represents the blocking version of the algorithm, where nonblocking point-to-point primitives are used but the `Wait` operations are immediately invoked after them, thus blocking the calling processes until the whole collective is complete. In Figure 4.15b, when the nonblocking collective is invoked, the point-to-point primitives are called, and the corresponding `Wait` operations can be invoked later. Communications are therefore performed by an asynchronous progress mechanism while the process is able to continue its computation. Nevertheless, this is a naive approach with dubious benefits and hardly scalable that can collapse the progress system of the communication devices with excessive requests when the number of processes is large or when allowing concurrent nonblocking collectives.



(a) Flat Tree Blocking Bcast (*nb*FT) (b) Flat Tree Nonblocking Bcast

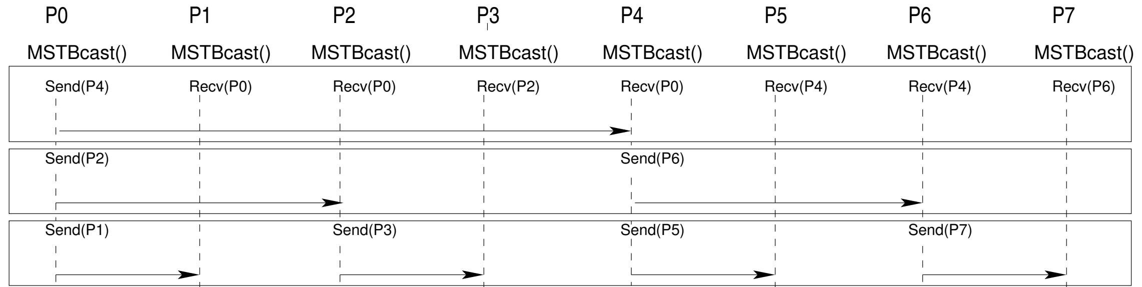
Figure 4.15: Flat Tree-based Blocking and Nonblocking Broadcast implementation

The approach followed relies on a queue of stages per process that calls a collective operation. The queues hold two types of stages: "dependent" or "non-dependent". The former prevents the progress of the collective operation in the process until the stage is complete. The latter only needs to be complete when returning from the `Wait` operation. With this mechanism, it is possible to implement the multi-core aware algorithms presented in Section 4.1.2 by splitting them in a stage manner to take advantage of their optimized performance. The use of "non-dependent" stages enables the scheduling of several stages that can issue nonblocking point-to-point primitives simultaneously.

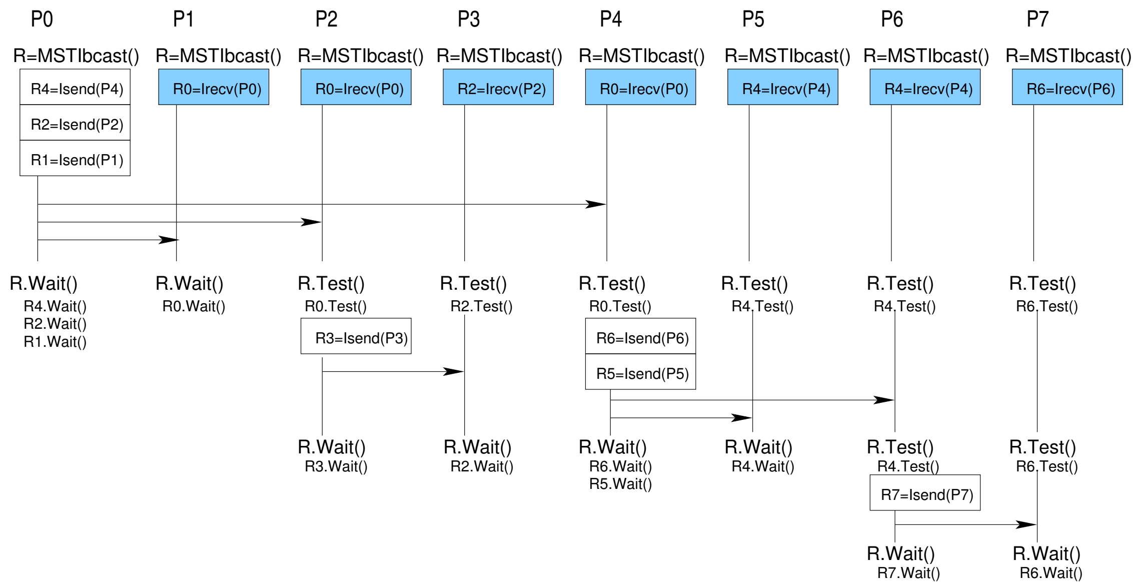
The issue that arises here is the progress of the operation. The decision to be

made is if a specific mechanism is required for these operations or if it is possible to rely on each device to make progress happen, which is only possible if the operations are based on nonblocking point-to-point primitives. Since nonblocking collectives are in a very early stage of adoption, the priority is to assess the feasibility of these operations, which can be achieved relying on existing device mechanisms. Nevertheless, if the library does not create any thread to be in charge of stage progress, the user is responsible of being aware of it and making some calls to a testing function (`Test`). This function will check the stage queue and, if there is not any dependent stage pending, it will launch the subsequent stages. If there are neither pending stages nor subsequent ones, it will complete the operation. The `Wait` operation is equivalent to perform several tests until the operation is finished.

Figure 4.16 compares a blocking Minimum-Spanning Tree (MST) Broadcast with a nonblocking counterpart implemented with stages for an example scenario using eight processes. The figure assumes that the collective operation is issued at the same time in every process and, in the nonblocking scenario, the calls to the `Test`/`Wait` operations are also made simultaneously. This is not a realistic scenario, but the aim of the figures is only to show the differences among both approaches. As in Figure 4.15, dotted lines represent idle time spent waiting for the operation to be complete and continuous lines represent computation. Figure 4.16a represents the blocking implementation which uses blocking point-to-point primitives since it is a recursive algorithm (see Section 4.1.2). This recursion causes the algorithm to be executed in three implicit steps (marked by rectangles). In Figure 4.16b, the nonblocking staged implementation of the MST algorithm is represented, using dark rectangles for dependent stages and white ones for non-dependent stages. In this scenario there are processes that schedule three or less stages, depending on how many communications they have to perform. With the purpose of ease the representation of the algorithm, it is assumed that a process that has already scheduled every stage, calls `Wait` instead of `Test`. In addition, when `Test` is called and the stage is already finished, this stage is never tested again. It can be seen that even considering simultaneous calls, this algorithm yields less implicit synchronization and requires less ordering than the blocking one. In fact, in the blocking version every process will end almost at the same time whereas in the nonblocking one, even when the calls are simultaneous, each process can finalize the collective when its stages are complete.



(a) MST Blocking Bcast



(b) MST Nonblocking Bcast

Figure 4.16: MST-based Blocking and Nonblocking Broadcast Implementation

Optimization of the Nonblocking Collectives Library for Shared Memory

The implementation being discussed, based on stages, is portable to every communication device in FastMPJ, although the maximum potential could be reached on shared memory where the stage-based design can be combined with the specific shared memory algorithms implemented in `smdev` and described in Section 4.1.2. The operation of the nonblocking algorithms proposed here is the same as described before: instead of keeping the thread waiting when the condition to progress is not yet fulfilled, the thread checks if it has to remain in the same stage or if it is able to advance to the next one. These checks are performed in the `Test` or `Wait` function.

Flat Tree algorithms have shown significant performance improvements for shared memory communications in [108], and the stage-based design can be implemented in these algorithms by using a single stage per thread instead of a stage queue. Moreover, as for shared memory blocking collectives, shared structures will only contain references to messages instead of real data, barely involving any memory overhead. It is thus feasible to allow the scheduling of concurrent collective operations storing references of multiple messages. This is possible through the replication on an array of the shared structure that maintains the references along with the semaphores that manage the stage progress. Hence, there is a limited number of concurrent operations bounded by the number of replications managed by a tag parameter. The tag is a user parameter needed by point-to-point operations to identify each message. Collective operations based on point-to-point primitives use this parameter internally and nonblocking collectives for shared memory can take advantage of it, since it can be used as a sequence number. This tag modulo the number of concurrent operations is an index in the array of references. If the slot of this index is free, the operation can continue and the slot will be marked as busy with the operation tag but, if not, the operation must remain in the previous stage. Nevertheless, since this array is a shared structure, the index can be marked as busy by another thread that has started the same operation that the new thread is trying to perform. This situation does not incur any problem because the index is marked with the current tag, so the new thread will realize that it is occupied by the same operation and will be able to perform its stage.

To illustrate the implementation of the shared memory nonblocking collectives, Listings 4.2 and 4.3 show the pseudo-code for the main methods used in the nonblocking Broadcast: the `ibcast` and `Test` calls, and the internal function to advance between stages.

Listing 4.2 shows the pseudo-code for the `ibcast` and `Test` operations, that can be called from the user application. Nonblocking collectives, like nonblocking point-to-point primitives, return a request over which the `Test` and `Wait` operations can be invoked. Before returning the request, this collective operation issues an advance call to make as much progress as possible. This advance call will not make the thread wait if any of the conditions prevent it from progressing, but it will just return the stage of the operation without advancing. The `Test` method also schedules the

advance returning immediately even if it was not able to move forward. The `Wait` function would perform the same operation but blocking until the collective has been completed.

Listing 4.2: Pseudo-code of the `ibcast` and `Test` methods for the shared memory nonblocking Broadcast

```
public static Request ibcast(Object buf,int root,int tag){
  int indexTag = tag % NUMBER_OF_CONCURRENT_OPERATIONS;
  int stage = ibcastAdvance(indexTag,buf,root,tag,INITIAL_STAGE);
  return new IbcastRequest(indexTag,buf,root,tag,stage);
}

public Status Test() {
  if(stage == FINAL_STAGE)
    return COMPLETE;
  else{
    stage = ibcastAdvance(indexTag, buf, root, tag, stage);
    return null;
  }
}
```

The pseudo-code of the advance method used for the Broadcast is shown in Listing 4.3. This is the main function that controls the progress throughout stages. It uses two condition variables implemented as `AtomicInteger` type: `collectives_nbc` and `ended_collective_nbc`. To support a fixed number of concurrent collective operations (`NUMBER_OF_CONCURRENT_OPERATIONS`), these variables are replicated in two arrays of `AtomicInteger` indexed by the modulo of the operation tag. Hence, each operation will have an assigned slot which consists of the condition variables and a shared buffer in which the root stores the reference to the data.

The `collectives_nbc` variable controls the start and end of a collective and it has three possible states: `FREE`, `INIT` and `BUSY`. In a Broadcast operation, when the root finds the `collectives_nbc` variable in the `FREE` state, it sets this variable to `INIT` to mark it as occupied but not yet prepared for the rest of threads to perform the copy. Then, after copying the reference to the message data, the root sets the condition variable to `BUSY` with the operation tag, to notify that the data is ready to be copied. All threads but the root will not be able to start the communication operation until the variable is set to the operation tag by the root. The `ended_collective_nbc` variable indicates how many threads have already performed the copy and when the root would be able to reset and free the slot.

Listing 4.3: Pseudo-code of the `ibcastAdvance` method for the shared memory nonblocking Broadcast

```
public static int ibcastAdvance(int indexTag, Object buf, int root, int tag,
                                int stage){
  boolean isRoot = (getRank()==root);
  if(isRoot){
    if(stage==INITIAL_STAGE){
      if(!collectives_nbc[indexTag].compareAndSet(FREE,INIT){
        return INITIAL_STAGE;
      }
      else{
        ended_collective_nbc[indexTag].set(INIT);
        shared_buffers[indexTag] = buf;
        collectives_nbc[indexTag].set(tag);
        stage = FIRST_ROOT_STAGE;
      }
    }
    if(stage==FIRST_ROOT_STAGE){
      if(!ended_collective_nbc[indexTag].compareAndSet(nthreads,FREE)){
        return FIRST_ROOT_STAGE;
      }
      else{
        shared_buffers[indexTag] = null;
        ended_collective_nbc[indexTag].set(FREE);
        collectives_nbc[indexTag].set(FREE);
        return FINAL_STAGE;
      }
    }
    if(stage==FINAL_STAGE)
      return FINAL_STAGE;
  }
  else{
    if(stage==INITIAL_STAGE){
      if(!collectives_nbc[indexTag].compareAndSet(tag,tag){
        return INITIAL_STAGE;
      }
      else{
        copy(shared_buffers[indexTag],buf);
        ended_collective_nbc[indexTag].increment();
        return FINAL_STAGE;
      }
    }
    if(stage==FINAL_STAGE)
      return FINAL_STAGE;
  }
}
```

4.2.3. Performance Evaluation

The performance evaluation of the shared memory nonblocking collectives has been carried out on a representative 16-core shared memory testbed next described. The benchmarking consists of a micro-benchmark which compares the blocking and nonblocking versions of two collective operations (Broadcast and Scatter), and a production application which combines I/O operations with nonblocking collectives.

Experimental Configuration

The nonblocking collectives library has been evaluated on the Sandy Bridge Intel Xeon E5 system already described in Chapter 2 (see Figure 2.1) and used for the evaluation of the blocking collectives on shared memory (Section 4.1.3). This shared memory system consists of 2 Intel Xeon E5-2670 octa-core processors at 2.6 GHz (a total of 16 cores in the system, 32 with hyperthreading) and 64 GBytes of RAM. Each core has a 32-KByte L1 and a 256-KByte L2 cache and the eight cores in each processor share a 20-MByte Intel Smart L3 Cache. Although the system had the hyperthreading enabled, the results are shown for 16 cores since the use of the 32 available threads does not provide any benefit in terms of performance. The OS is Linux CentOs with kernel v2.6.35, together with OpenJDK JVM v1.6.0_20 (IcedTea6 v1.9.8) and FastMPJ v1.0 internal release.

Micro-benchmarking of MPJ Nonblocking Collectives

Figures 4.17-4.21 show the performance results for Broadcast (Figures 4.17-4.19) and Scatter (Figures 4.20 and 4.21) of a comparative benchmark among: (1) a blocking algorithm (labeled as “block” in the figures), (2) a nonblocking algorithm without overlapping computation, i.e. with an immediate call to `Wait` after calling the collective (labeled as “nbc”), and (3) the nonblocking algorithm overlapping the communication with a synthetic workload (“nbc+overlap”). The benchmark is based on the test published for LibNBC [45] and the nbc version is used to assess the overhead introduced by the nonblocking operation. For the workload simulation, it is estimated how long it takes to perform the nonblocking operation together with its corresponding `Wait` to approximate it with the synthetic workload.

The latency is measured for the collective, `Test` and `Wait` calls, which make up the effective communication. The performance evaluation methodology has been carefully designed following the recommendations addressed in [37] to avoid bias caused by side effects of the use of the JVM.

For the Broadcast, it is shown the comparison between the shared memory nonblocking implementation and three blocking implementations: the blocking Broadcast for shared memory (Figure 4.17), the MST algorithm (Figure 4.18), and the Flat Tree (*nb*FT) algorithm (Figure 4.19). Besides the specific shared memory algorithm, on which the nonblocking implementation is based, the *nb*FT algorithm has been selected as it is the point-to-point-based counterpart of the shared memory algorithm. MST has also been included for comparison purposes.

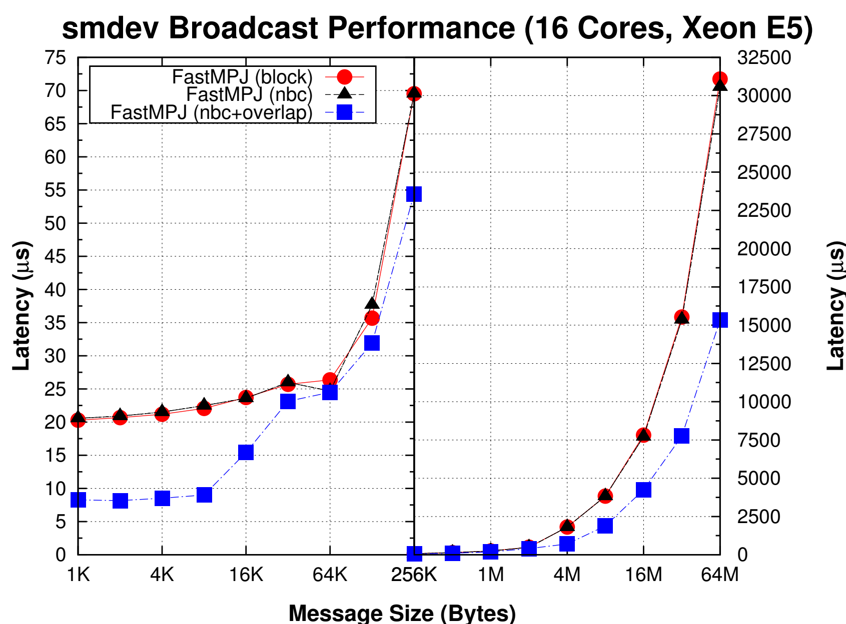


Figure 4.17: Shared memory Broadcast: Blocking vs. Nonblocking

Regarding Figure 4.17, as expected, there is almost no overhead imposed on nonblocking collectives when compared to the shared memory blocking counterpart. Moreover, it can be observed that the overlapping with a computational workload reduces the time spent in the actual communication. This is because of the lack of imposed synchronization, thus when a thread calls the `Wait` function, it is more probable that other threads have already finished and they will not have to wait

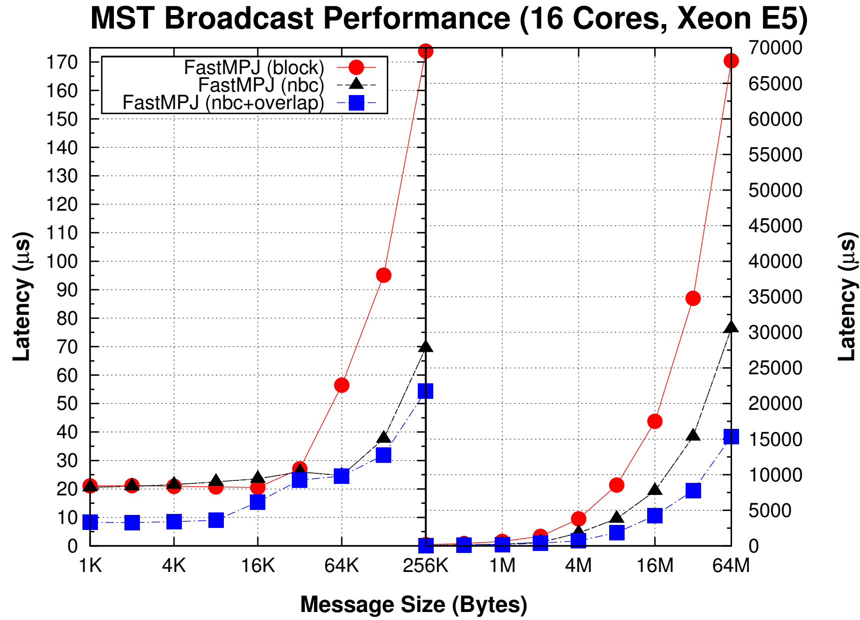


Figure 4.18: MST Broadcast vs. Shared memory Nonblocking Broadcast

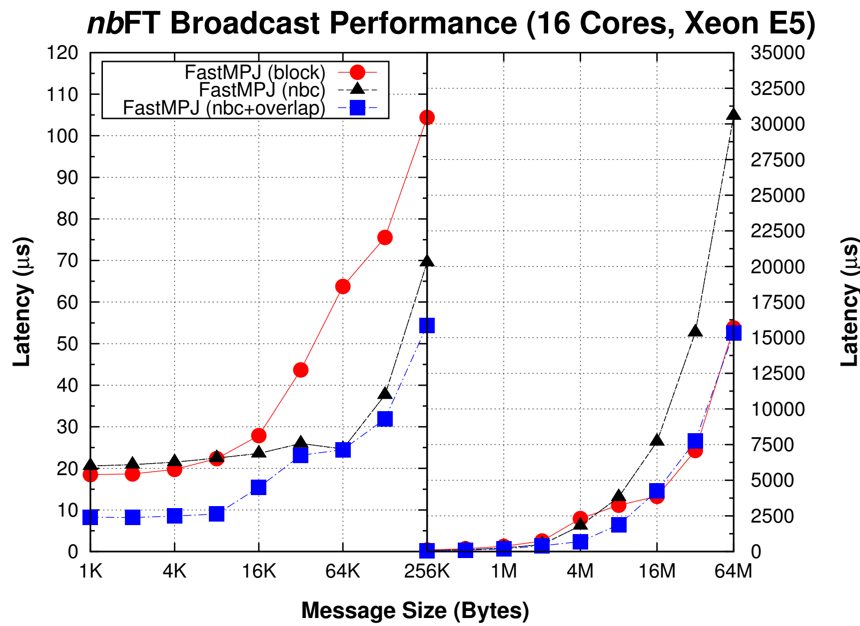


Figure 4.19: *nb*FT Broadcast vs. Shared memory Nonblocking Broadcast

to perform their own operations. The differences increase from 4 MBytes on, since messages do not fit in the L3 cache (taking into account that there is a 20-MByte L3 cache shared among 8 cores), thus taking more advantage of the overlapping.

The results of the comparison with the MST algorithm in Figure 4.18 show that the nbc version overcomes MST, and thus the shared memory blocking implementation also overcomes MST in the same way according to the results of Figure 4.17. The nbc+overlap version shows again its benefits. Finally, Figure 4.19 shows that, although the *nb*FT algorithm has a higher start-up latency than the algorithms based on shared memory transfers, it provides more scalability (in terms of message size) since it avoids contention in the access to shared buffers. Nevertheless, nbc+overlap achieves better performance than *nb*FT, mainly providing less start-up latency for small messages.

Regarding the Scatter, the blocking versions selected were the shared memory (Figure 4.20) and the *nb*FT (Figure 4.21) algorithms. MST was discarded due to its poor performance. Results are quite similar to the ones observed for Broadcast. Again, the nonblocking collective barely imposes any overhead over the shared

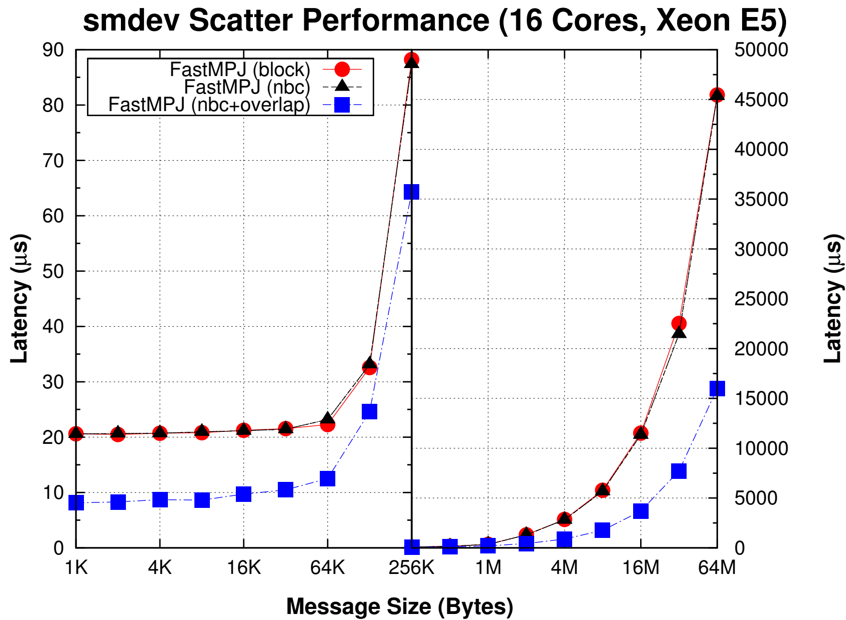


Figure 4.20: Shared memory Scatter: Blocking vs. Nonblocking

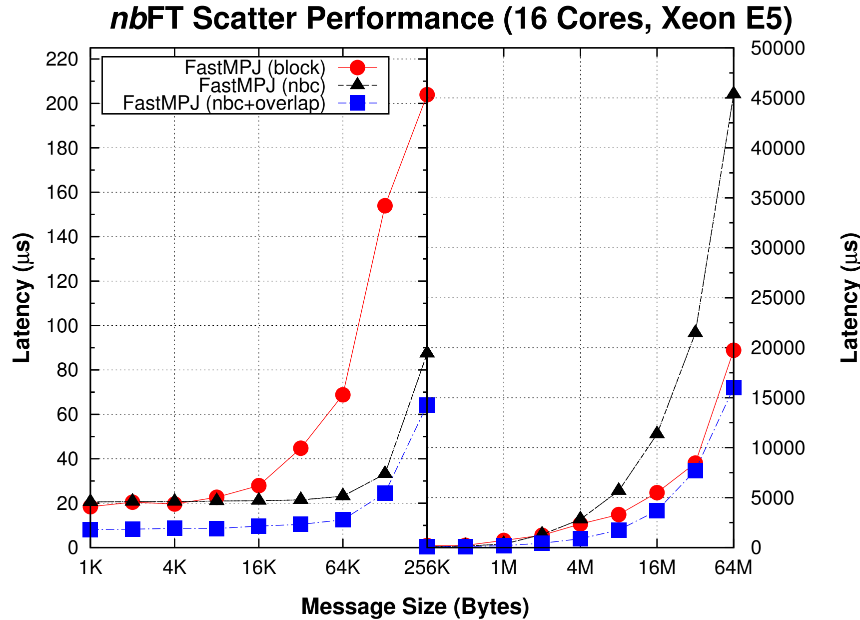


Figure 4.21: *nb*FT Scatter vs. Shared memory Nonblocking Scatter

memory blocking implementation and, when compared to the *nb*FT algorithm, it overcomes the scalability issues of the shared memory algorithm, also reducing the latency obtained with *nb*FT, as it happened for the Broadcast operation.

Benchmarking of an MPJ Application

The application used to assess the performance of the implemented nonblocking collectives has been selected as it overlaps collective communications with computation and I/O operations. MPI includes the MPI-I/O library to deal with input/output operations. The feasibility of the use of I/O nonblocking collectives has been studied in [142]. In MPJ there are no MPJ-I/O libraries available, so parallel codes have to deal directly with the standard Java I/O libraries, generally imposing a large overhead which makes them suitable for overlapping. Although it is possible to use an extra thread to perform the I/O operations, this mechanism is far more complicated to manage than the overlapping of nonblocking collectives and I/O operations for MPJ.

The original application reads a group of zip files which represent two years of financial data from the Spanish Market of Financial Futures, including strings of information for options and futures over the IBEX-35 (Spanish exchange index) shares and the National Spanish Bond. The application has to extract and read each file and process the lines. Each line is processed at the moment that it is read and available (using the `readLine` method). For the processing of each line and to evaluate the effect of the overlapping, a synthetic workload that is measured in terms of the number of operations over each line has been created. The parallelization with MPJ uses the `readChar` method to read groups of chars which make up a buffer scattered among all threads. Only one thread extracts one zip file at a time and reads groups of chars until the buffer is complete. After that, it performs a Scatter and each thread runs the workload over each line received. Since a whole buffer of chars is scattered, the application has also to deal with the possibility that a line could be broken between two threads and it is solved by overlapping the scattered fragments. This imposes a certain overhead compared to the sequential version, but it is more efficient for the parallelization in that there is no need for serialization nor conversion of strings to arrays of chars to build the sending buffer. Finally, every thread (including the one in charge of the I/O operations) performs the operations over the received lines. The application benchmark does not take into account the return of the results to the thread in charge of I/O operations because the goal is to measure the effect of using a nonblocking Scatter that allows the overlapping of communications with read operations and computation.

Figure 4.22 shows the performance of the parallel application on the Xeon E5 testbed using 16 cores and different workloads. For the parallelization with blocking collectives ("block"), the Flat Tree (*nb*FT) algorithm was chosen since, although the shared memory algorithm shows better performance for small and medium size messages, *nb*FT is better for large messages, which are extensively used in this application. The nonblocking version ("nbc") allows a defined number of concurrent nonblocking Scatters for both the sender and the receivers (see 4, 8 and 16 in the legend of the figure). The results are shown in terms of execution time varying the workload according to the number of operations per line. A buffer size of 512 KBytes is received by each thread, thus having an 8-MByte scattered message. The buffer size was selected to take advantage of memory locality and L3 cache size.

It can be observed that, while the overhead when there is no computation (i.e. 0 operations per line) is negligible, the use of nonblocking collectives achieves performance gains up to 30% when the number of operations (and thus the workload) increases. This is due to the overhead imposed by the implicit synchronization of the blocking collectives as opposed to the overlapping of communication and computation in the nonblocking implementation, especially when increasing the number of concurrent nonblocking collectives.

The results of this benchmark show that the use of nonblocking collectives in shared memory architectures is beneficial for communication-intensive codes that also involve large amounts of computation assigned to threads in an unbalanced way. Hence, when having a costly I/O operation and significant workloads in each thread, introducing nonblocking collectives can provide important performance benefits.

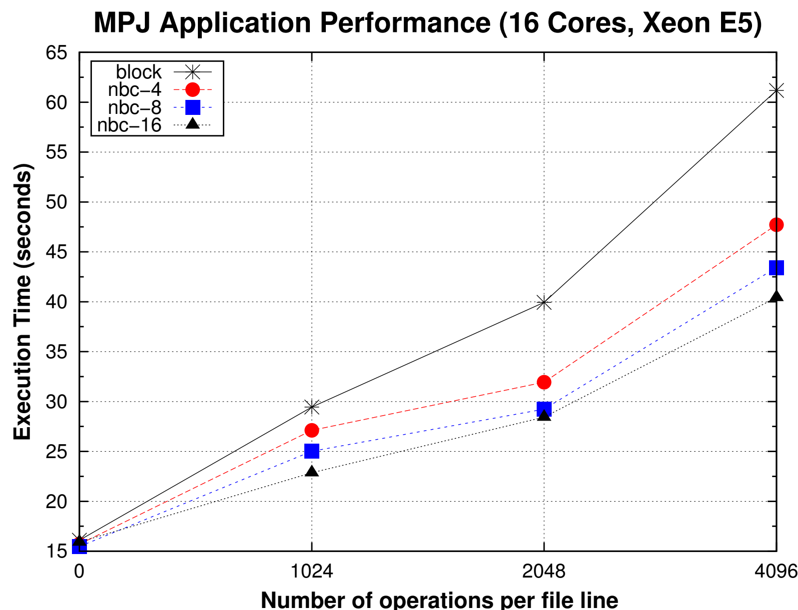


Figure 4.22: Performance comparison of an I/O-intensive MPJ application using Blocking ("block") and Nonblocking ("nbc") collectives

4.3. Main Contributions of the MPJ Collectives Support

This chapter has presented a scalable and efficient MPJ collectives library for parallel computing on multi-core architectures. This library efficiently exploits multi-core systems with multiple levels of core hierarchies, taking advantage of a thread-based MPJ collectives implementation. Additionally, the library transparently provides Java message-passing applications with several multi-core aware algorithms per collective primitive that can be selected automatically at runtime, depending on relevant parameters, in order to increase the communications performance.

The evaluation of the collectives library on representative shared memory systems has shown that it is able to outperform significantly the MPI counterpart. This high performance is also achieved when considering different levels of core hierarchies, for instance in a multi-core cluster with InfiniBand interconnect. Furthermore, the developed library increases the speedup of collective communication-intensive Java HPC kernels, as shown on a representative 128-core system. In our work [122] it was also experimentally assessed that the lower the scalability and performance provided by the communication hardware, the higher the relative performance benefits achieved by the MPJ collectives library. Thus, this library can contribute significantly to bridge the performance gap between Java and native languages in HPC.

Regarding nonblocking primitives, this chapter has presented an analysis of the feasibility of the MPI 3.0 nonblocking collectives for message passing in Java focused on shared memory architectures. The performance evaluation on a representative multi-core shared memory system and the analysis of the micro-benchmarking performance results have shown that: (1) no additional overhead is imposed by this nonblocking implementation, and (2) performance improvements are obtained when overlapping communication and computation. As representative results, with the proposed nonblocking collectives there is a performance gain up to 50% for Broadcast and 66% for Scatter in comparison with shared memory blocking counterparts.

The impact on a real I/O-intensive MPJ application has also been analyzed using a synthetic workload to assess the performance improvements regarding concurrent collective operations and workload overlapping. Performance results confirmed that

shared memory nonblocking collectives are able to exploit the avoidance of implicit synchronization, as well as the overlapping of computation and communication. For instance, it was obtained around a 30% reduction in execution time when using 16 concurrent collectives. These results demonstrate the benefits of nonblocking collectives in shared memory environments, which is crucial when the trend is to increase the number of cores per processor, showing that nonblocking collectives allow communication-intensive MPJ applications to reduce significantly their overhead, thus improving the scalability of the communications.

Chapter 5

Java Heterogeneous Computing

The use of specific coprocessors, such as Graphics Processing Units (GPUs), composed by a large number of small and specialized cores in order to increase the performance of specific regions of parallel codes, has gained popularity over the last years. These accelerators provide high performance and energy efficiency, having an increasing presence in the Top 500 supercomputer list [134]. The recently released Intel Xeon Phi, from the Intel MIC family, aims at sharing this popularity by providing many-core coprocessors with a x86 architecture to enable the use of traditional programming languages and paradigms to exploit performance [106]. The efficient combination of multi-core processors and many-core coprocessors is a timely and very important research topic that will next be discussed in the context of high performance heterogeneous computing in Java.

This chapter presents a performance and productivity analysis of currently available solutions that enable Java programmers to exploit many-core accelerators. These solutions have been evaluated using the NVIDIA K20m GPU and the Intel Xeon Phi coprocessor presented in Chapter 2.

5.1. General Purpose GPU Computing in Java

Nowadays Java generally supports General Purpose GPU (GPGPU) computing relying on native solutions, such as CUDA or OpenCL, due to the lack of direct JVM

support, which is still an ongoing effort [96]. Section 1.3 presented a detailed analysis of current Java GPGPU solutions, among which two main approaches were identified: the performance-oriented or architectural-friendly approach, which consists of JNI bindings to CUDA/OpenCL, and the productivity-oriented or user-friendly approach, based on Java pre-processing. This section presents an evaluation of two Java GPGPU projects that have been selected due to their representativeness and active development: jCuda and Aparapi. The jCuda project [55] is a direct wrapper implementation over a native library, thus architectural-friendly and aiming to obtain similar performance to native solutions, whereas Aparapi [4] translates bytecode into OpenCL at runtime providing a user-friendly API. Moreover, jCuda is based on CUDA whereas Aparapi relies on OpenCL, although this fact is not especially relevant since our work [32], which evaluated comparatively CUDA and OpenCL performance, has shown that they are able to provide roughly the same performance, being the actual implementation of a given code the main reason for performance differences.

5.1.1. Experimental Configuration

The evaluation has been done using our own Java implementation of representative GPGPU synthetic kernels [25]. They are code snippets that provide basic building blocks widely extended in HPC applications (e.g., a matrix multiplication kernel), selected from the benchmark suite Scalable HeterOgeneous Computing (SHOC) [23]. This suite includes several levels of benchmarks (with both CUDA and OpenCL versions for some of them), being the level 0 the most simple one aiming to obtain specific and architectural features like the memory bandwidth or the computation capacity. Level 1 benchmarks are typical computational kernels to measure the performance of higher-level operations such as the Fast Fourier Transform (FFT) or matrix multiplications. Finally, Level 2 benchmarks are real application kernels. The selected codes came from the first two levels since they are general, well-known and widely used benchmark codes. Table 5.1 presents the four synthetic kernels selected.

The benchmarks were executed on the K20m GPU described in Chapter 2 (See Section 2.2.2 and Figure 2.4), being the host machine a dual-socket 8-core Intel Xeon

Table 5.1: Selected kernels for Java GPGPU performance analysis

Kernel	Suite	Description
MaxFlops	SHOC Level 0	Peak GFLOPS
GEMM	SHOC Level 1	Matrix multiplication
Stencil2D	SHOC Level 1	A two-dimensional nine-point Stencil calculation
FFT	SHOC Level 1	Fast Fourier Transform

E5-2660 at 2.20 GHz. The software configuration is CentOS v6.4, JVM Oracle JDK v1.7.0 and GCC v4.4.7; NVIDIA CUDA 5.0 with the v304.54 NVIDIA driver and the NVIDIA OpenCL support, and SHOC v1.1.5.

5.1.2. Analysis of Experimental Results

Table 5.2 shows the MaxFlops kernel performance in GFLOPS for jCuda and Aparapi compared to the CUDA version of SHOC MaxFlops (considered as the baseline –100%– for the comparison). Results show that, when using the single precision benchmark, jCuda achieves around 76% of the CUDA SHOC performance, whereas Aparapi reaches almost 70%. It should be noticed that the CUDA version is around 15% below the theoretical peak performance, which points out that the benchmark does not take full advantage of the GPU features for single precision. When using the double precision benchmark, jCuda gets 99% of CUDA SHOC performance, whereas Aparapi only 62%. To sum up, on the one hand, in the single precision scenario, jCuda and Aparapi present quite similar performance; on the other hand, for double precision, jCuda obtains almost the same performance as native CUDA SHOC, whereas Aparapi achieves almost 40% less performance.

Figures 5.1-5.3 show the results obtained for the selected SHOC Level 1 kernels: matrix multiplication (GEMM), Stencil2D and FFT. Figure 5.1 presents the performance results of the matrix multiplication kernel for CUDA SHOC, jCuda, Aparapi and a serial Java implementation. A logarithmic scale has been used to better appreciate the differences among the Java versions, where jCuda is the one that achieves the highest performance. This is because it relies on the same matrix

Table 5.2: MaxFlops performance on the NVIDIA K20 GPU (in GFLOPS)

	Single precision		**Double precision**	
Theoretical	3520.00		1170.00	
SHOC (CUDA)	3007.00	100%	1168.58	100%
jCuda	2281.06	75.86%	1162.26	99.46%
Aparapi	2101.15	69.87%	730.23	62.49%

multiplication routine of the CUBLAS library used by CUDA SHOC, being the performance gap between jCuda and CUDA SHOC motivated by the overhead of the data movements between Java and CUBLAS.

As currently GPUs do not directly support standard JVMs, serial Java performance results have been obtained on the CPU, running a pure Java (without relying on native methods) matrix multiplication code which yields around 1.5 GFLOPS both for single and double precision. Although Java would be able to achieve higher performance calling any BLAS library with Java bindings (e.g., the Intel MKL library), we opted for using a standard and fully portable Java code as baseline. The introduction of Aparapi in the serial Java implementation increases performance up to 39x for single precision (from 1.63 GFLOPS for serial Java to 64.43 GFLOPS for Aparapi), and up to 23x for double precision (from 1.49 to 35 GFLOPS). In case that Aparapi does not find a GPU in the system it would run the code using either OpenCL or a pure Java Thread Pool on the CPU, depending on the availability of OpenCL multi-core support, so portability is not compromised with this solution.

Figure 5.2 presents the results of the Stencil2D kernel. Once again CUDA SHOC achieves the highest performance and the serial Java version (running on the CPU) is around 25-60 times slower. However, jCuda and Aparapi are able to achieve around 50-80% of the CUDA SHOC performance, increasing the performance of serial Java up to 45 times.

Note that jCuda does not rely on any CUDA library, so the performance gap with CUDA is larger in comparison with the GEMM results. Moreover, as both the jCuda and Aparapi versions of the Stencil2D kernel implement a similar algorithm, they achieve similar performance results. In fact, Aparapi is able to outperform jCuda. In terms of productivity, Aparapi is a better option as it has been much

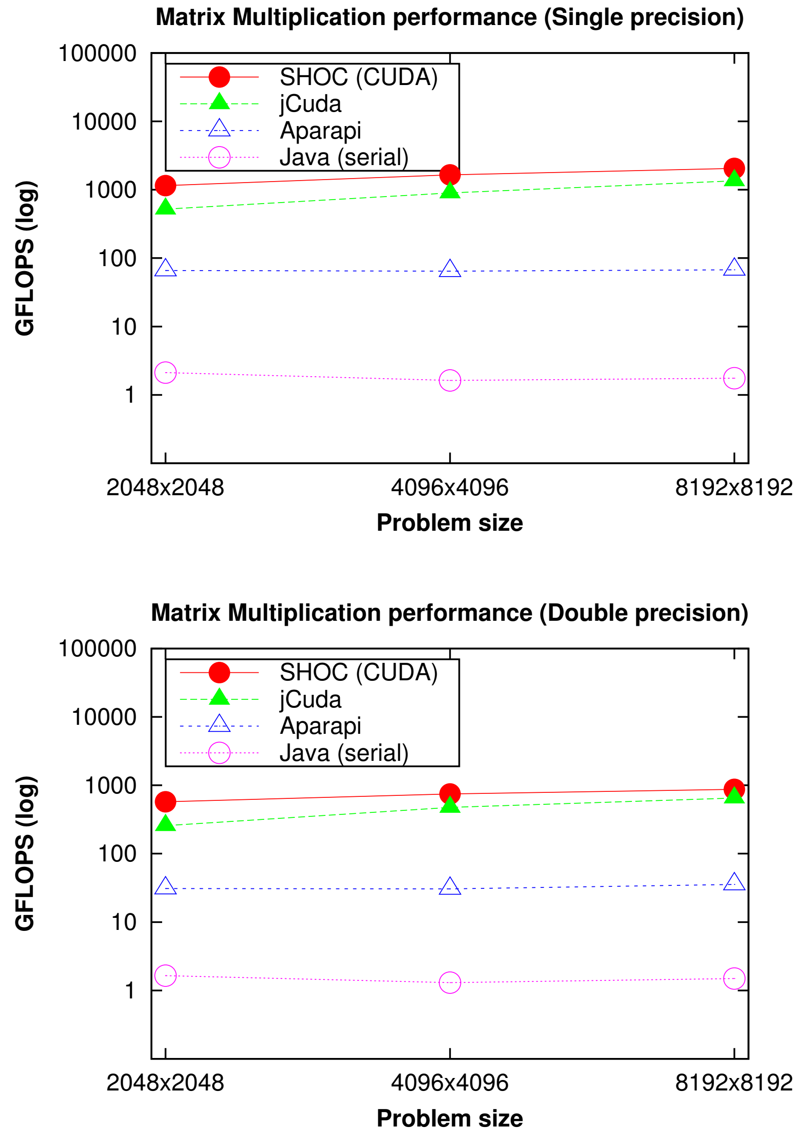


Figure 5.1: Matrix multiplication (GEMM) kernel performance on the NVIDIA K20 GPU

easier to develop the Aparapi code than the jCuda version. This lower time-to-solution and the significant performance achieved suggest that Aparapi is the best Java GPGPU choice when jCuda cannot rely on an optimized CUDA library such as CUBLAS.

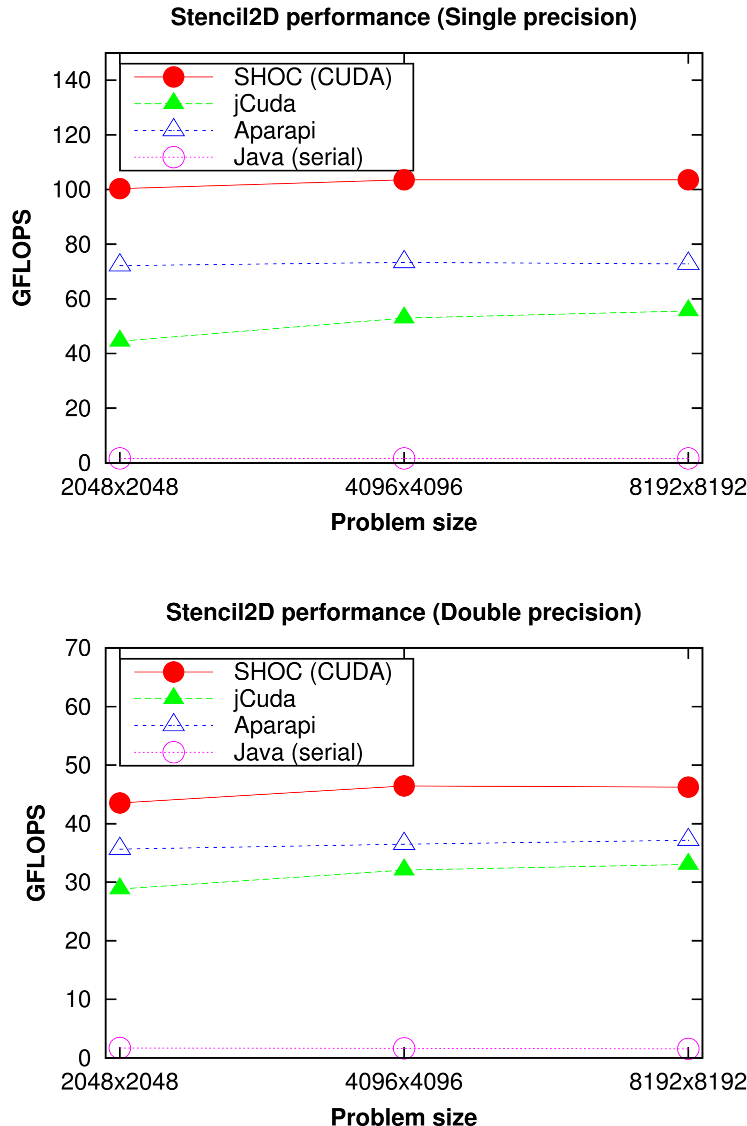


Figure 5.2: Stencil2D kernel performance on the NVIDIA K20 GPU

The performance results of the FFT kernel are shown in Figure 5.3. In this case, CUDA SHOC (and thus jCuda) relies on the CUFFT library and the comparison between jCuda and Aparapi is similar to the matrix multiplication scenario. As it can be observed, Aparapi FFT is around 13-15.5 times faster than the serial Java implementation (CPU-only) but 2-6.5 times slower than jCuda and around 4-10 times slower than CUDA SHOC. Nevertheless, for Java programmers, Aparapi could represent a portable and productive option for accelerating their standard

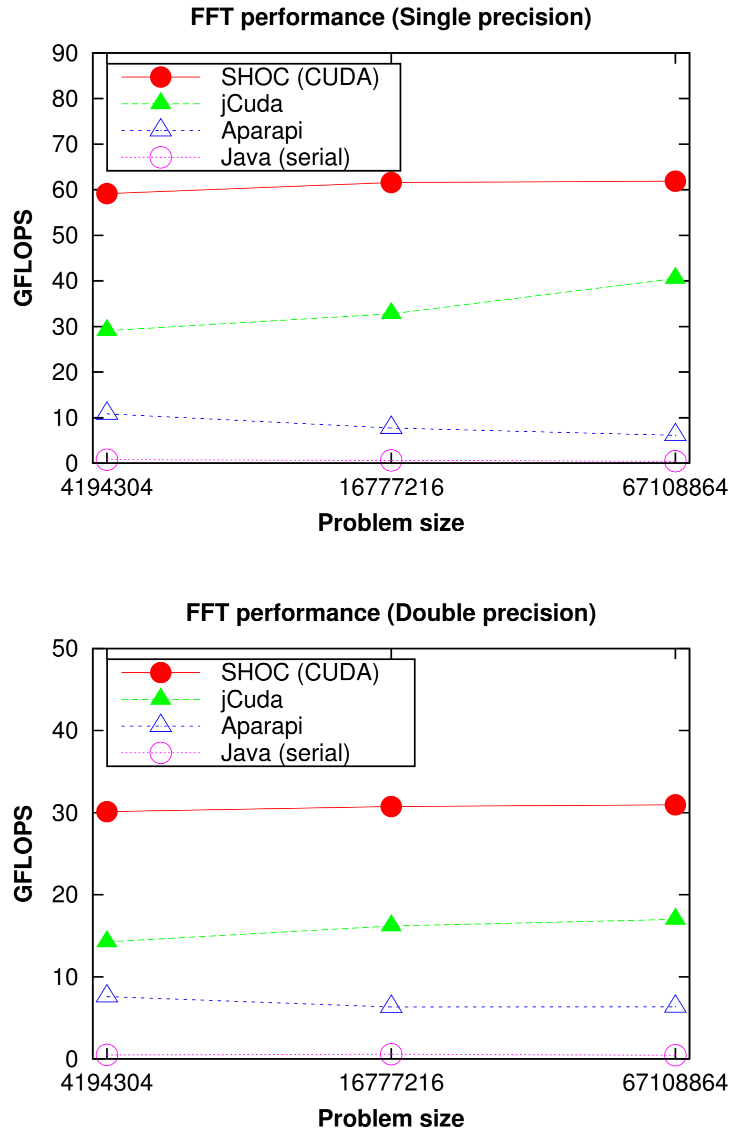


Figure 5.3: FFT kernel performance on the NVIDIA K20 GPU

Java applications on GPUs. However, when performance is critical, jCuda, and even writing CUDA native methods accessible through JNI, is the way to go, especially when there are highly optimized libraries available, like CUBLAS or CUFFT, although at the cost of losing portability and productivity.

5.2. Many-core x86-based Computing in Java

Currently, the best ratio between computational performance and power consumption in x86 architectures is provided by the many-core Intel Xeon Phi coprocessor. As mentioned in Section 1.3, there is no direct Java support for taking advantage of this coprocessor as an accelerator, although its x86-based architecture should ease this support, so it is feasible that it will be seamless integrated in the JVM with the generalization of this kind of coprocessor. However, the OpenCL support of the Xeon Phi by Intel [97] makes it possible to exploit the architecture through JOCL (Java binding to OpenCL) [56] and Aparapi.

5.2.1. Experimental Configuration

This section presents a performance analysis of the Xeon Phi using the SHOC kernels (both the offloading and OpenCL versions) and their Aparapi implementations, thus allowing a comparison with the GPU performance results of the previous section. The offloading SHOC implementation has been specifically developed to take advantage of the Xeon Phi by offloading parallel sections of the benchmark to be executed on the accelerator using OpenMP. Hence, this benchmark suite is able to exploit the full performance provided by the Xeon Phi coprocessor by being adapted to its architectural features, also relying on the Intel MKL library for mathematical operations. The main drawback of the OpenCL version is that, although it is able to run on different devices, from mainstream multi-core processors to GPUs and the Xeon Phi, this does not mean that the performance is portable, and thus an OpenCL code can be efficient on a given device but perform poorly on another. Moreover, another main issue that OpenCL codes face is the transfer overhead, both to the accelerator and between its cores. Nevertheless, these codes can take advantage of automatic vectorization and 512-bit vector instructions.

The evaluation has been carried out on an Intel Xeon Phi (see Section 2.2.1 and Figure 2.3 for more details) using a dual-socket 8-core Intel Xeon E5-2660 at 2.20 GHz as host machine, with CentOS v6.4, JVM Oracle JDK v1.7.0, Intel C compiler v13.1.1, Intel OpenCL SDK v.3.0.67279, SHOC v1.1.5 (OpenCL version) and SHOC MIC alpha pre-release (offloading version).

5.2.2. Analysis of Experimental Results

Table 5.3 shows the MaxFlops performance obtained with the two versions of SHOC and Aparapi, as well as the theoretical peak performance. The table also shows the percentage of Aparapi performance compared to OpenCL SHOC, which is quite low (52.18%) for single precision, although it reaches 96.75% for double precision. Regarding the SHOC versions, the offloading implementation is the best performer, with results quite close to the theoretical peak performance, whereas the OpenCL implementation is around 90% of the offloading version.

Table 5.3: MaxFlops performance on the Intel Xeon Phi (in GFLOPS)

	Single precision		**Double precision**	
Theoretical	2016.00		1080.00	
SHOC (Offloading)	1938.22		975.19	
SHOC (OpenCL)	1765.04	100%	897.85	100%
Aparapi	921.05	52.18%	868.72	96.75%

Figures 5.4-5.6 present the serial Java, Aparapi and OpenCL SHOC performance results, thus showing graphically the benefits of introducing Aparapi in the serial Java code and also the performance of common OpenCL codes in this coprocessor. For clarity purposes offloading SHOC results are not shown, as it is a custom native implementation, specifically developed for the Intel Xeon Phi architecture, which is able to rival CUDA SHOC performance on the K20 GPU, but whose performance benefits are much more limited when invoked by Java through JNI.

Figure 5.4 presents the matrix multiplication (GEMM) results both for single and double precision. The native OpenCL SHOC benchmark obtained the best performance. The Aparapi implementation is around 5.5-8 times slower than OpenCL SHOC, but it overcomes the serial Java approach by 5-8 times. These results are significantly lower than those obtained on the K20 GPU. In fact, SHOC results for single precision are up to 16 times slower, whereas Aparapi is around 3.3-4.5 times slower.

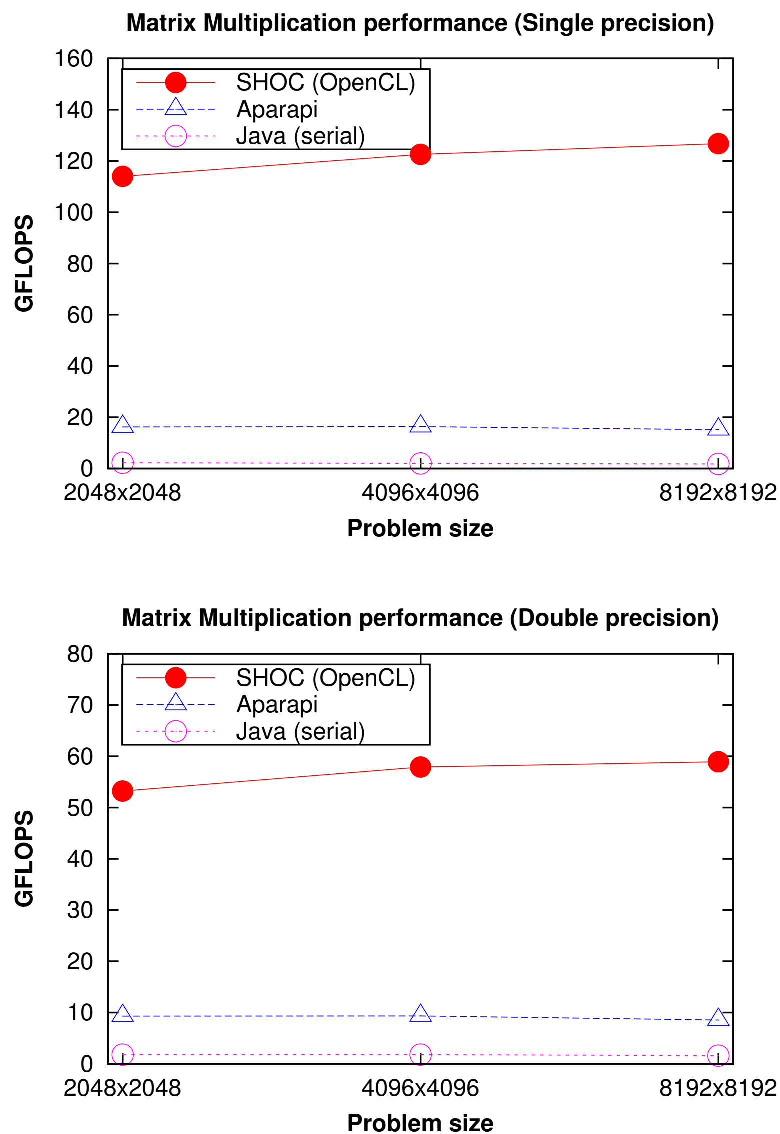


Figure 5.4: Matrix multiplication (GEMM) kernel performance on the Intel Xeon Phi

The results for the Stencil2D kernel are shown in Figure 5.5. The overall performance is, once again, quite reduced compared to the results obtained on the K20 GPU. Aparapi is able to achieve similar results to OpenCL SHOC, both for single and double precision, but it slightly outperforms the serial Java code. Here the main bottleneck is the transfer time between the host and the accelerator card.

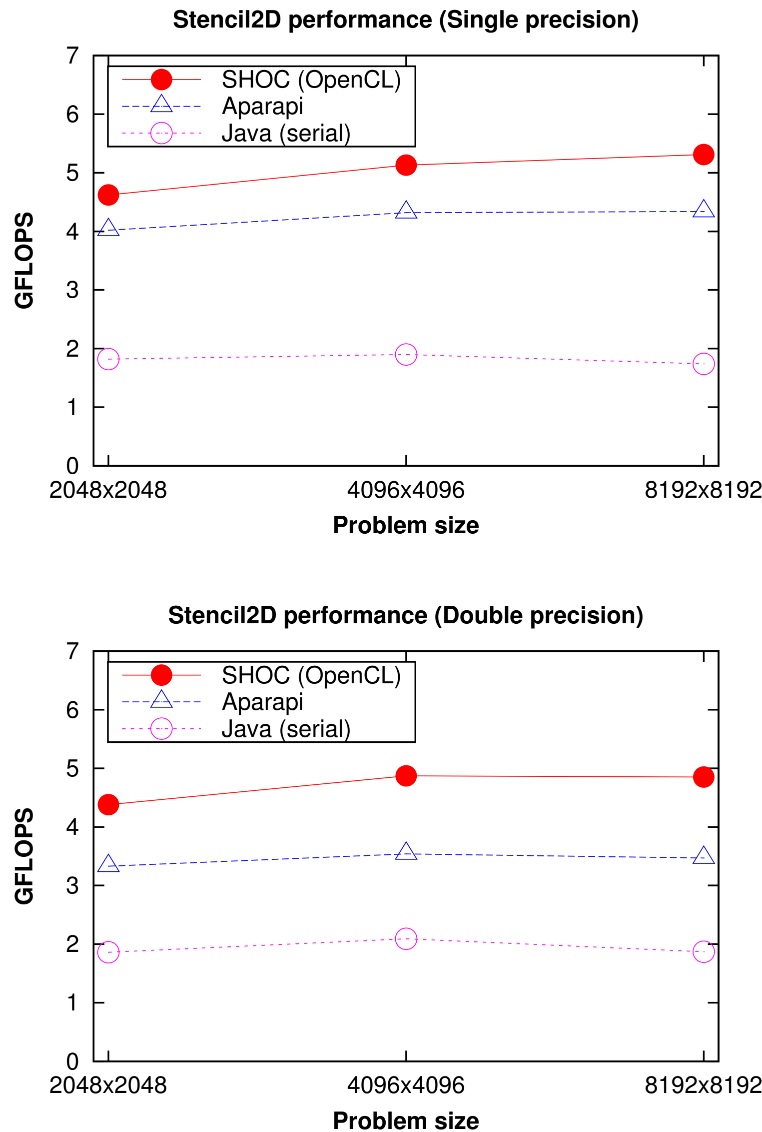


Figure 5.5: Stencil2D kernel performance on the Intel Xeon Phi

Figure 5.6 shows the results for the FFT kernel. OpenCL SHOC is again the best performer, generally around 1.2 times (and up to 5 times) faster than Aparapi,

although it is around 3.6 times slower compared to CUDA SHOC on the K20 GPU (see Figure 5.3). Finally, the performance of Aparapi drops when the problem size increases, although introducing Aparapi in the serial Java code increases its performance around 4-14 times.

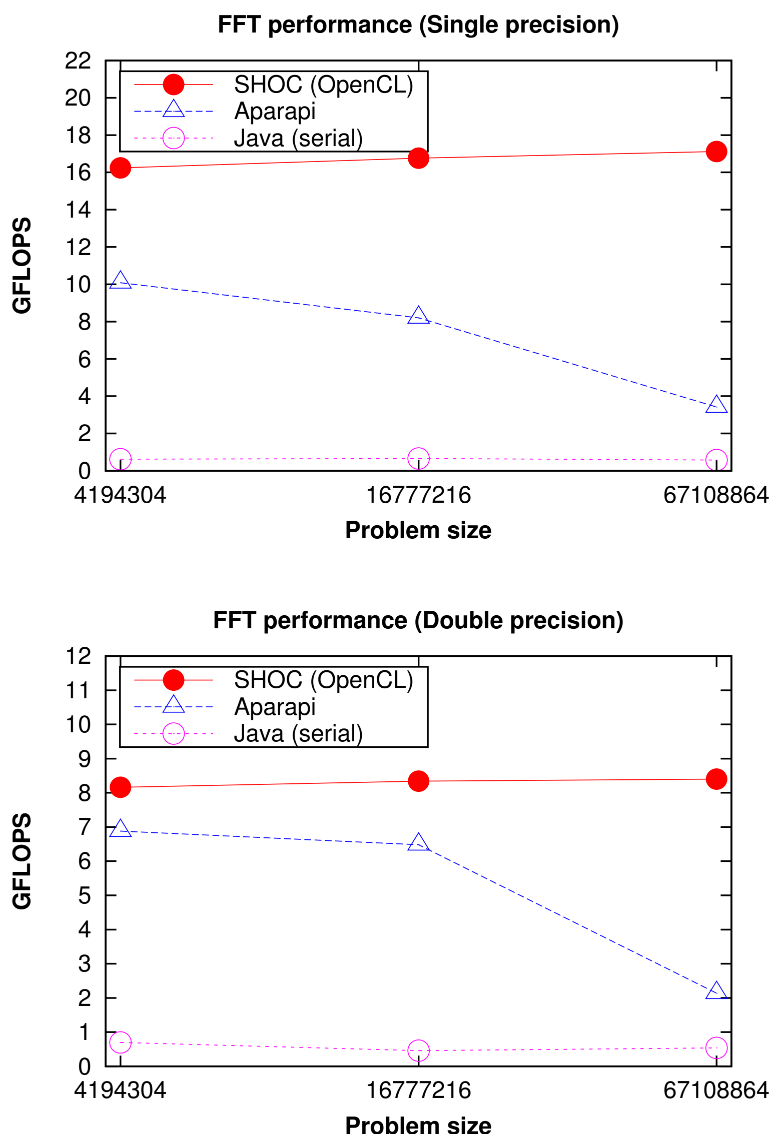


Figure 5.6: FFT kernel performance on the Intel Xeon Phi

As a general conclusion, although Aparapi outperforms the serial Java implementation of the selected kernels, it is far from taking full advantage of the Xeon Phi capabilities, mainly because it relies on OpenCL codes that are not especially

optimized for this coprocessor. This lack of efficiency of the OpenCL SHOC codes is therefore derived from the generality of the benchmarks, used to measure the performance of GPUs and accelerators, not taking into account the special features of the Xeon Phi and the overhead imposed by data movements, not only from or to the host, but also between cores within the coprocessor. These are the main reasons that prevent Aparapi from obtaining competitive results on the Xeon Phi. Achieving high performance is extremely complicated for an automatic framework that transforms Java code into OpenCL, since it would require either knowledge of the platform where the code is going to be run, or resorting to low-level programming mechanisms. And, in this latter case, it would be more efficient to use a direct wrapper over OpenCL, such as JOCL, or just to invoje native code from the JVM.

5.3. Analysis of Productivity of Java Heterogeneous Computing Codes

The measurement of the productivity of programming languages has always been an important concern [82, 146]. The High Productivity Computer Systems (HPCS) project [39], funded by the Defense Advanced Research Projects Agency (DARPA) of the USA, had the goal of providing more productive systems, also developing techniques to measure both complexity and programmability. One of the main problems that the productivity analysis faces since some years ago is the inclusion of parallel programming given its inherent complexity [75], and thus different metrics have been proposed and analyzed with this purpose [38, 62, 119]. Some works like [1, 40, 41, 80, 99, 126] propose to explore complexity by the comparison of codes developed by students that have just learned different parallel languages, and [87] proposes a similar technique but using experts. However, all these studies need to be complemented with a static analysis of the features of the developed codes [35], where the number of Lines Of Code (LOC) [98] is among the most popular metrics.

However, since the information of the LOC metric can be insufficient to provide an accurate measurement of productivity, in [19] the authors combine the number of LOC with other parameters like the number of characters per line and the structures needed by the language (e.g., type declarations, function calls...), to provide a better

insight into the productivity difference between MPI and UPC. The authors of this work propose to group metrics in two main categories: manual programming effort and conceptual programming effort. The former uses the number of LOC and the Number Of Characters (NOC). NOC is added to clarify the results provided by the number of LOC, because in case that the language uses long complex lines, the number of LOC can be not significant enough in terms of effort needed to write a program. To measure the conceptual programming effort, it is necessary to take into account the effort to learn the language concepts and apply them. To capture this effort, this work suggests the use of the number of parameters passed to a function, function calls, keywords, types and constructions, and other relevant features of the language, counted for some common parallel programming tasks like work distribution and synchronization among others.

This section provides a productivity analysis of Aparapi and jCuda based on the work presented in [19] and using the four SHOC kernels selected in the performance comparison with the native solutions of the previous sections (see Table 5.1). These kernels present different features that will allow to compare productivity in several scenarios. Due to the lack of relevant differences in terms of productivity between single and double precision codes, only the single precision version has been evaluated. Regarding relevant characteristics of the codes, it is worth mentioning that jCuda uses the CUBLAS and CUFFT libraries for matrix multiplication (GEMM) and FFT, respectively, whereas Aparapi implements its own kernels. Moreover, in order to increase performance, the Aparapi FFT kernel does explicit memory management, thus increasing code complexity. Another remarkable characteristic is that jCuda supports loading PTX and CUBIN modules to execute CUDA C kernels from a Java application. Hence, for the MaxFlops and Stencil2D kernels (which do not access optimized CUDA libraries), jCuda has to create and manage the PTX file and write a CUDA C kernel. Both the PTX creation and the CUDA kernel are not included neither in the manual effort nor in the conceptual effort analysis. The code needed to create the PTX file is the same for both benchmarks and it basically consists in I/O operations to create the file, and thus it does not use library-specific methods or types from jCuda. The CUDA C kernel is not analyzed because it is written in other language different than Java. However, the fact that these benchmarks need extra programming, even in a different language, has to be taken into account, especially in the conceptual effort analysis.

5.3.1. Characterization of the Manual Effort

The characterization of the manual effort, i.e. the cost of developing a code, consists of measuring the number of LOC and the NOC. Parts of the code that are irrelevant (e.g., the verification of the result by comparing it to the sequential one, reads and writes from/to files or standard I/O, standard Java imports...) are not considered. Moreover, the try-catch statement is considered as a single line of code that manages the exception, to avoid bias related to different error managements. The NOC does not include spaces or blank lines.

Table 5.4 shows the metrics and results for the selected kernels. Kernels marked with (*) are the ones that need to create a PTX file and a CUDA C kernel. The code needed to create the PTX file has 24 lines and 959 characters, and it is the same for both benchmarks, but, as mentioned before, it is not included in the table. jCuda needs fewer LOC for matrix multiplication (GEMM) and FFT since it relies on external libraries (CUBLAS and CUFFT, respectively), otherwise Aparapi has a more compact syntax, as it can be seen for MaxFlops and Stencil2D. However, when Aparapi implements optimizations (e.g., the explicit memory management in FFT) then its number of LOC largely increases.

Table 5.4: Manual effort for the development of the Aparapi and jCuda implementations of representative SHOC kernels

		Aparapi	**jCuda**
MaxFlops	#LOC	86	181 (*)
	NOC	2118	5562 (*)
GEMM	#LOC	49	40
	NOC	1050	1283
Stencil2D	#LOC	55	99 (*)
	NOC	1348	3780 (*)
FFT	#LOC	167	27
	NOC	3653	836

The NOC indicates that jCuda is more verbose than Aparapi, e.g., although Aparapi has a higher number of lines in GEMM, jCuda presents more characters and thus longer lines. Another example is FFT, where Aparapi has around 6 times

more LOC than jCuda, but the NOC is only 4 times higher.

5.3.2. Characterization of the Conceptual Effort

The characterization of the conceptual effort has been done in [19] considering features of the language such as the number of keywords, number of functions and types, and number of parameters of a function, among other factors. In our scenario, the keywords used are reserved words from standard Java (`new`, `public`...) and thus only parameters, method calls, and constructs (constructors) and types are considered. The number of classes and library-specific objects have been considered to quantify the number of types. Each method call invoked through `object.method()` or `Class.method()` is considered as a method call plus one type because an object or a class is necessary to write the method call. Calls to constructors with the form `Class name = new Class()` are considered to be one method call plus one type (there is only one class involved), but if the creation of a new instance has the form `Class name = object.method()`, it is considered to be a method call plus two types (the class and the object used, which does not have to be an instance of the same class). Another consideration is that the quantification is made taking into account the written code, by counting how many times the items are written, but not how many times they are executed/invoked (and thus loops and branches are not considered in a different way).

In [19], the conceptual effort was also quantified for several common parallel code structures like synchronization or data distribution. In our scenario, although Aparapi and jCuda codes have the same purpose, they are completely different paradigms that can present different structures depending on how the benchmark is implemented (i.e., calling a library kernel, or whether explicit memory management is used in Aparapi). Hence, every statement that is library-specific or uses classes and objects related to Aparapi and jCuda has been selected and classified according to the benchmark features. The Aparapi codes present a very similar structure and the specific sentences are related to: (1) kernel instantiation or creation, (2) configuration and execution (and device selection), (3) sentences used to release resources, and (4) the proper kernel implementation (extension of `Kernel` class), which usually implements the algorithm and presents barely one or two sentences

using Aparapi types and methods. As explained before, a special case is the FFT kernel, where Aparapi is configured to use explicit memory management in order to increase performance, which also increases its complexity.

In jCuda, the sections of code identified differ depending on the benchmark features. The access in jCuda to external libraries, such as CUBLAS or CUFFT, means that jCuda has to manage the kernel calls, the library configuration, and some memory allocation and release of resources if the library method does not manage it (as it happens for the CUBLAS method for matrix multiplication). However, when the kernel has to be written, the code will include handling low-level configurations of the device and the context, memory management, kernel calls, synchronization or event management, and release of resources. Moreover, in these scenarios (marked with (*) in the tables), a PTX file has to be created and a C kernel has to be implemented. These extra costs have not been included for clarity purposes, as explained before, but they should be taken into account when analyzing the results.

Tables 5.5-5.8 present the results obtained from the conceptual effort characterization of the Aparapi and jCuda codes. To ease the comparison of results, the number of items has been summed and an overall score, as the sum of all the measurements, is provided. Except for FFT, the overall score for jCuda is generally an order of magnitude higher than for Aparapi, which indicates that it is noticeably more difficult to develop codes using jCuda than Aparapi. Moreover, the gap in the development effort would be even higher in case of considering the creation of the PTX file and that a CUDA C kernel has to be implemented. The FFT kernel is a special case as Aparapi implements an optimization (explicit memory management), and the jCuda code is quite simple as it relies on an external library, CUFFT.

Besides the low number of specific constructions of the language, which makes Aparapi easier to learn and develop with is that the codes always present a similar structure, making it possible for the programmer to focus on the algorithmic part. Even when explicit memory management is used, it only affects when reading/ copying data from/to the device, but the increase in the number of parameters and method calls is due to the number of data structures that are moved, since there are only two different methods involved (`get` and `put`).

Table 5.5: Conceptual effort in MaxFlops

(a) Aparapi

	kernel creation	config. & exec.	release	kernel	sum	overall score
# Parameters	3	2	0	0	5	16
# Method calls	1	2	1	1	5	
# Constructs & types	1	3	1	1	6	

(b) jCuda

	config. device & context	mem. manag.	kernel call	events	release	sum	overall score
# Param.	13	6	17	10	1	47	119 (*)
# Funct. calls	11	4	5	9	1	30	
# Constr. & types	11	4	8	18	1	42	

Table 5.6: Conceptual effort in matrix multiplication (GEMM)

(a) Aparapi

	kernel creation	config. & exec.	release	kernel	sum	overall score
# Parameters	3	2	0	0	5	17
# Method calls	1	2	1	2	6	
# Constructs & types	1	3	1	1	6	

(b) jCuda

	library config.	memory manag.	kernel call	release	sum	overall score
# Parameters	1	36	14	4	55	110
# Method calls	2	16	1	4	23	
# Constructs & types	2	18	8	4	32	

Table 5.7: Conceptual effort in Stencil2D

(a) Aparapi

	kernel creation	**config. & exec.**	**release**	**kernel**	**sum**	**overall score**
# Parameters	3	3	0	0	6	19
# Method calls	1	2	1	3	7	
# Constructs & types	1	3	1	1	6	

(b) jCuda

	config. device & context	**memory manag.**	**kernel call**	**synch.**	**release**	**sum**	**overall score**
# Param.	13	20	34	0	2	69	146 (*)
# Funct. calls	11	12	10	1	2	36	
# Constr. & types	11	12	16	0	2	41	

Table 5.8: Conceptual effort in FFT

(a) Aparapi

	kernel creation	**config. & exec.**	**release**	**kernel**	**memory manag.**	**sum**	**overall score**
# Param.	1	1	0	0	22	24	71
# Funct. calls	1	2	1	1	22	27	
# Constr. & types	1	3	1	1	14	20	

(b) jCuda

	library config.	**kernel call**	**release**	**sum**	**overall score**
# Parameters	4	8	1	13	30
# Method calls	2	2	1	5	
# Constructs & types	4	6	2	12	

5.4. Lessons Learned from Java Heterogeneous Computing

The lack of direct Java support for general purpose accelerators and coprocessors is forcing Java developers to resort to wrappers or frameworks that access native libraries via JNI. The choice between the use of direct wrappers or high-level frameworks is usually a trade-off between efficiency of the codes and productivity. If performance is key, when the device or accelerator has a common or well-known architecture, the high-level framework can provide reasonably good performance, as it happens for the K20 GPU with Aparapi, but to take advantage of specific features it is necessary to use low-level programming tools, such as jCuda, which also comes with the advantage of providing access to high performance libraries (e.g., CUBLAS and CUFFT). However, looking for productivity, the use of high-level frameworks can reduce significantly the development effort, as it does not require learning a direct low-level wrapper or having specific knowledge of the underlying hardware architecture.

The inclusion of the transparent support for GPUs and accelerators in the JVM, expected for upcoming releases, will represent a qualitative improvement in order to directly exploit the offloading of standard Java code to accelerators, something that looks quite feasible, at least for the x86 coprocessors. However, in the meantime, there are competitive solutions that allow us to leverage some computation-intensive Java codes. Finally, in order to achieve extremely high performance results it is necessary to rely on native libraries and an in-depth knowledge of the underlying hardware.

Conclusions and Future Work

This Thesis has presented an analysis of the suitability of Java for High Performance Computing (HPC) on multi- and many-core shared memory architectures, as well as the design, implementation and optimization of Java communication solutions for shared memory. Hence, one of the main outcomes of the Thesis is the development and evaluation of a Java message-passing middleware for parallel programming in shared memory systems and an optimized library of collective operations (both blocking and nonblocking) for Message Passing in Java (MPJ). The Thesis also includes a thorough evaluation of current possibilities for heterogeneous programming in Java, considering both the latest architectures in GPUs (NVIDIA Kepler) and x86-based accelerators (Intel Xeon Phi).

One of the most important conclusions is that Java is able to achieve high performance results, which are competitive when compared to natively compiled languages, also taking advantage of shared memory systems in a more productive and portable manner. However, on the one hand, although the Java language usually provides high productivity, the multithreading API, which enables the efficient exploitation of multi-core architectures, has a complex and error-prone interface where the synchronization management can result in inefficient codes or, even worse, in race conditions and thus inconsistencies. On the other hand, the higher-level Java concurrency framework is oriented to high-throughput task programming, requiring the rewriting of many parallel codes. Nevertheless, the multithreading support allows to develop high performance tools and libraries with simple interfaces that manage the Java threading API internally in a transparent manner. In fact, it was possible to develop a message-passing middleware, which is the traditional solution for distributed memory programming, relying on specific shared memory mechanisms, and thus enabling the efficient exploitation of multi-core processors with this

scalable and well-known paradigm.

Regarding the optimization of communications in Java, when trying to maximize the performance of Java applications, it is necessary to focus not only on the hardware but also on the JVM features since, for example, initialization costs or lack of compilation of some parts of the code can cause latency increases that hide other optimizations developed using traditional techniques. The JVM presents improvements in this field to make it more hardware-friendly and allowing to take advantage of hardware characteristics, e.g., JVM threads mapped to OS threads, which enables the use of thread affinity to specific cores in Java codes.

A key contribution of this Thesis is the development of a highly efficient communication middleware for shared memory, the `smdev` device, which provides very low latency and high bandwidth, very close to the peak limits of the hardware. In order to achieve this high performance, `smdev` implements an efficient zero-copy protocol and takes advantage of low-level hardware characteristics and fine-grained synchronization. Moreover, `smdev` provides an MPJ API that allows the abstraction from thread-level programming.

The MPJ collectives library developed, including both blocking and nonblocking primitives, showed that the awareness of the underlying architecture, although in a high-level manner, can help to optimize the communication algorithms. The library includes several algorithms per collective allowing the selection of the most suitable one at runtime, depending on the number of cores and the message size. Moreover, the implementation of a nonblocking collectives library in Java proved that shared memory message-passing applications can take advantage of computation and communication overlapping and lack of implicit synchronization, getting rid of the restrictions of traditional blocking operations.

Finally, although manufacturers do not provide specific support in Java for many-core architectures, there are a few libraries that enable the use of Java to program these systems, like jCuda and Aparapi. These projects are based on native support, generally accessed by JNI, and provide different interfaces, from direct Java bindings (jCuda) to high-level APIs that internally deal with the native support (Aparapi), thus combining efficiency and productivity.

In terms of future work, one of the most interesting topics would be the analysis

of the benefits and drawbacks of the direct Java support for many-core architectures. Since the GPU support, given its architecture, mainly depends on the manufacturers, the Intel Xeon Phi, with a x86 architecture, is the main target where it should be feasible to run JVMs. However, in order to support efficiently this coprocessor in Java, the development of optimization techniques to handle the complexity of this architecture and taking full advantage of the scalability of its hardware resources would be required.

Moreover, the `smdev` device and the collectives library can be further optimized by including hardware parametrization in the communication algorithms and implementing a more accurate selection of the algorithm, which could be achieved by developing performance models taking into account both JVM and hardware characteristics. Another interesting topic related to `smdev` is to carry out a new analysis about the feasibility of a hybrid communication device for shared/distributed memory environments. Although this possibility was already considered in [109], the contention and need for synchronization in the access to shared network resources limited this option. The lack of parallel codes adapted to exploit the performance of hybrid architectures is also a barrier to demonstrate the benefits of this hybrid support. Nevertheless, the development of a new approach based on thread-based collectives, combining multithreading for intra-node transfers with communications across the network, is the way to go as they showed high scalability.

Bibliography

[1] R. Alameh, N. Zazwork, and J. Hollingsworth. Performance Measurement of Novice HPC Programmers Code. In *Proc. 3rd Intl. Workshop on Software Engineering for High Performance Computing (SE-HPC'07)*, pages 3–8, Minneapolis, MN, USA, 2007. pages 115

[2] G. Almási, P. Heidelberger, C. J. Archer, X. Martorell, C. C. Erway, J. E. Moreira, B. Steinmacher-Burow, and Y. Zheng. Optimization of MPI Collective Communication on BlueGene/L Systems. In *Proc. 19th ACM Intl. Conf. on Supercomputing (ICS'05)*, pages 253–262, Cambridge, MA, USA, 2005. pages 85

[3] AMD Opteron™. Processor Solutions. `http://products.amd.com/en-us/OpteronCPUDetail.aspx?id=644`. [Last visited: June 2013]. pages 17, 40

[4] Aparapi. API for Data Parallel Java. `http://code.google.com/p/aparapi/`. [Last visited: June 2013]. pages 13, 104

[5] H. Arndt, M. Bundschus, and A. Naegele. Towards a Next-Generation Matrix Library for Java. In *Proc. 33rd Annual IEEE Intl. Computer Software and Applications Conf. (COMPSAC'09)*, pages 460–467, Seattle, WA, USA, 2009. pages 8

[6] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, and S. Weeratunga. The NAS Parallel Benchmarks Summary and Preliminary Results. In *Proc. 1991 ACM/IEEE Supercomputing Conf. (SC'91)*, pages 158–165, Albuquerque, NM, USA, 1991. pages 48

[7] M. Baitsch, N. Li, and D. Hartmann. A Toolkit for Efficient Numerical Applications in Java. *Advances in Engineering Software*, 41(1):75–83, 2010. pages 8

[8] M. Baker and B. Carpenter. MPJ: a Proposed Java Message Passing API and Environment for High Performance Computing. In *Proc. 2nd Intl. Workshop on Java for Parallel and Distributed Computing (IWJPDC'00)*, pages 552–559, Cancun, Mexico, 2000. pages 9

[9] M. Baker, B. Carpenter, G. Fox, S. Ko, and S. Lim. mpiJava: an Object-Oriented Java Interface to MPI. In *Proc. 1st Intl. Workshop on Java for Parallel and Distributed Computing (IWJPDC'99)*, pages 748–762, San Juan, Puerto Rico, 1999. pages 9

[10] L. A. Barchet-Estefanel and G. Mounie. Fast Tuning of Intra-cluster Collective Communications. In *Proc. 11th European PVM/MPI Users' Group Meeting (EuroPVM/MPI'04)*, pages 28–35, Budapest, Hungary, 2004. pages 57

[11] A. Basumallik, S.-J. Min, and R. Eigenmann. Programming Distributed Memory Systems using OpenMP. In *Proc. 12th Intl. Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS'07)*, page 181 (8 pages), Long Beach, CA, USA, 2007. pages 7

[12] R. F. Boisvert, J. J. Dongarra, R. Pozo, K. A. Remington, and G. W. Stewart. Developing Numerical Libraries in Java. *Concurrency: Practice and Experience*, 10(11–13):1117–1129, 1998. pages 8

[13] M. Bornemann, R. V. van Nieuwpoort, and 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)*, pages 217–224, Sorrento, Italy, 2005. pages 10, 56

[14] R. Brightwell, S. Goudy, A. Rodrigues, and K. D. Underwood. Implications of Application Usage Characteristics for Collective Communication Offload. *Intl. Journal of High Performance Computing and Networking*, 4(3-4):104–116, 2006. pages 85

[15] R. Brightwell and K. D. Underwood. An Analysis of the Impact of MPI Overlap and Independent Progress. In *Proc. 18th ACM Intl. Conf. on Supercomputing (ICS'04)*, pages 298–305, Saint Malo, France, 2004. pages 84

[16] J. Bull, M. Westhead, and J. Obdržálek. Towards OpenMP for Java. In *Proc. 2nd European Workshop on OpenMP (EWOMP'00)*, pages 98–105, Edinburgh, UK, 2000. pages 7

[17] D. Buntinas, G. Mercier, and W. Gropp. Implementation and Evaluation of Shared-memory Communication and Synchronization Operations in MPICH2 using the Nemesis Communication Subsystem. *Parallel Computing*, 33(9):634–644, 2007. pages 28, 84

[18] P. Calvert. Parallelisation of Java for Graphics Processors. Final-year Dissertation, Computer Laboratory, University of Cambridge, UK, 2010. pages 13

[19] F. Cantonnet, Y. Yao, M. Zahran, and T. El-Ghazawi. Productivity Analysis of the UPC Language. In *Proc. 18th IEEE Intl. Parallel and Distributed Processing Symposium (IPDPS'04)*, pages 254–260, Santa Fe, NM, USA, 2004. pages 115, 116, 118

[20] 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: June 2013]. pages 9

[21] E. Chan, M. Heimlich, A. Purkayastha, and R. A. van de Geijn. Collective Communication: Theory, Practice, and Experience. *Concurrency and Computation: Practice and Experience*, 19(13):1749–1783, 2007. pages 57, 59

[22] G. Chrysos. Intel® Xeon Phi™ Coprocessor (Codename Knights Corner). In *Proc. 24th Hot Chips: A Symposium on High Performance Chips (HC24)*, pages 323–353, Cupertino, CA, USA, 2012. pages 21

[23] A. Danalis, G. Marin, C. McCurdy, J. Meredith, P. Roth, K. Spafford, V. Tipparaju, and J. Vetter. The Scalable HeterOgeneous Computing (SHOC) Benchmark Suite. In *Proc. 3rd Workshop on General Purpose Processing on*

Graphics Processing Units (GPGPU-3), pages 63–74, Pittsburgh, PA, USA, 2010. pages 104

[24] DAS-4 Cluster. Accelerators and Special Compute Nodes. `http://www.cs.vu.nl/das4/special.shtml`. [Last visited: June 2013]. pages 17

[25] J. Docampo, S. Ramos, G. L. Taboada, R. R. Expósito, J. Touriño, and R. Doallo. Evaluation of Java for General Purpose GPU Computing. In *Proc. Intl. Workshop on Engineering Object-Oriented Parallel Software (EOOPS'13)*, pages 1398–1404, Barcelona, Spain, 2013. pages xv, 3, 104

[26] J. J. Dongarra, S. W. Otto, M. Snir, and D. Walker. An Introduction to the MPI Standard. Technical report, University of Tennessee, Knoxville, TN, USA, 1995. pages 84

[27] Efficient Java Matrix Library (EJML). `http://code.google.com/p/efficient-java-matrix-library/`. [Last visited: June 2013]. pages 8

[28] P. Ekman and P. Mucci. Design Considerations for Shared Memory MPI Implementations on Linux NUMA Systems: an MPICH/MPICH2 Case Study. *Advanced Micro Devices (AMD)*, 16 pages, 2005. pages 28, 84

[29] R. R. Expósito, S. Ramos, G. L. Taboada, J. Touriño, and R. Doallo. FastMPJ: a Scalable and Efficient Java Message-Passing Library. 2013 (Submitted for journal publication). pages 10, 27, 29, 30, 55, 56, 86

[30] 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 52

[31] 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*, 2013 (In press). pages 52

[32] 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*, 2013 (In press). pages 22, 104

[33] G. E. Fagg, G. Bosilca, J. Pjesivac-Grbovic, T. Angskun, and J. J. Dongarra. Tuned: a Flexible High Performance Collective Communication Component Developed for Open MPI. In *Proc. 6th Austrian-Hungarian Workshop on Distributed and Parallel Systems (DAPSYS'06)*, pages 65–72, Innsbruck, Austria, 2006. pages 57

[34] Z. Fan, F. Qiu, A. Kaufman, and S. Yoakum-Stover. GPU Cluster for High Performance Computing. In *Proc. 16th ACM/IEEE Intl. Conf. for High Performance Computing, Networking, Storage and Analysis (SC'04)*, page 47 (12 pages), Pittsburgh, PA, USA, 2004. pages 11

[35] S. Faulk, J. Gustafson, P. Johnson, A. Porter, W. Tichy, and L. Votta. Measuring HPC Productivity. *Intl. Journal of High Performance Computing Applications*, 18(4):459–473, 2004. pages 115

[36] K. Ferreira, P. Bridges, R. Brightwell, and K. Pedretti. The Impact of System Design Parameters on Application Noise Sensitivity. In *Proc. 12th IEEE Intl. Conf. on Cluster Computing (CLUSTER'10)*, pages 146–155, Heraklion, Crete, Greece, 2010. pages 85

[37] A. Georges, D. Buytaert, and L. Eeckhout. Statistically Rigorous Java Performance Evaluation. In *Proc. 22nd Annual ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'07)*, pages 57–76, Montreal, QC, Canada, 2007. pages 94

[38] J. Gustafson. Purpose-Based Benchmarks. *Intl. Journal of High Performance Computing Applications*, 18(4):475–487, 2004. pages 115

[39] High Productivity Computer Systems. `http://highproductivity.org/`. [Last visited: June 2013]. pages 115

[40] L. Hochstein, V. R. Basili, U. Vishkin, and J. Gilbert. A Pilot Study to Compare Programming Effort for Two Parallel Programming Models. *Journal of Systems and Software*, 81(11):1920–1930, 2008. pages 115

[41] L. Hochstein, J. Carver, F. Shull, S. Asgari, V. R. Basili, J. K. Hollingsworth, and M. V. Zelkowitz. Parallel Programmer Productivity: a Case Study of Novice Parallel Programmers. In *Proc. 17th ACM/IEEE Intl. Conf. for High*

Performance Computing, Networking, Storage and Analysis (SC'05), page 35 (9 pages), Seattle, WA, USA, 2005. pages 115

[42] T. Hoefler, P. Kambadur, R. L. Graham, G. M. Shipman, and A. Lumsdaine. A Case for Standard Non-blocking Collective Operations. In *Proc. 14th European PVM/MPI Users' Group Meeting (EuroPVM/MPI'07)*, pages 125–134, Paris, France, 2007. pages 85

[43] T. Hoefler and A. Lumsdaine. Message Progression in Parallel Computing - To Thread or not to Thread? In *Proc. 10th IEEE Intl. Conf. on Cluster Computing (CLUSTER'08)*, pages 213–222, Tsukuba, Japan, 2008. pages 86

[44] T. Hoefler and A. Lumsdaine. Optimizing Non-blocking Collective Operations for InfiniBand. In *Proc. 8th Workshop on Communication Architecture for Clusters (CAC'08)*, 8 pages, Miami, FL, USA, 2008. pages 85

[45] T. Hoefler, A. Lumsdaine, and W. Rehm. Implementation and Performance Analysis of Non-blocking Collective Operations for MPI. In *Proc. 20th ACM/IEEE Intl. Conf. for High Performance Computing, Networking, Storage and Analysis (SC'07)*, page 52 (10 pages), Reno, NV, USA, 2007. pages 85, 93

[46] T. Hoefler, T. Schneider, and A. Lumsdaine. Accurately Measuring Overhead, Communication Time and Progression of Blocking and Nonblocking Collective Operations at Massive Scale. *Intl. Journal of Parallel, Emergent and Distributed Systems*, 25(4):241–258, 2010. pages 86

[47] HPPC: High Performance Primitive Collections for Java. `http://labs.carrotsearch.com/hppc.html`. [Last visited: June 2013]. pages 8

[48] C. Huang, O. Lawlor, and L. Kalé. Adaptive MPI. In *Proc. 6th Intl. Workshop on Languages and Compilers for Parallel Computing (LCPC'03)*, pages 306–322, College Station, TX, USA, 2003. pages 84

[49] IBM Research. The Cell Project. `https://www.research.ibm.com/cell/home.html`. [Last visited: June 2013]. pages 20

[50] Intel® Xeon Phi™ Coprocessor: Software Developers Guide. `https://www-ssl.intel.com/content/www/us/en/processors/xeon/xeon-phi-`

coprocessor-system-software-developers-guide.html. [Last visited: June 2013]. pages 22

[51] Intel® Xeon® E5 Series. http://newsroom.intel.com/community/intel_newsroom/blog/2011/11/15/intel-reveals-details-of-next-generation-high-performance-computing-platforms. [Last visited: June 2013]. pages 40

[52] JAMA: a Java Matrix Package. http://math.nist.gov/javanumerics/jama. [Last visited: June 2013]. pages 8

[53] Java Grande Forum. http://www.javagrande.org. [Last visited: June 2013]. pages 9

[54] Java Grande Forum. JavaNumerics. http://math.nist.gov/javanumerics/. [Last visited: June 2013]. pages 8

[55] jCuda. Java Bindings for CUDA. http://jcuda.org. [Last visited: June 2013]. pages 12, 13, 104

[56] JOCL. Java Bindings for OpenCL. http://www.jocl.org/. [Last visited: June 2013]. pages 13, 110

[57] JogAmp JOCL. Java OpenCL. http://jogamp.org/jocl/www/. [Last visited: June 2013]. pages 13

[58] 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)*, page 231 (8 pages), Long Beach, CA, USA, 2007. pages 7

[59] K. Kandalla, H. Subramoni, K. Tomko, D. Pekurovsky, S. Sur, and D. K. Panda. High-Performance and Scalable Non-blocking All-to-All with Collective Offload on InfiniBand Clusters: a Study with Parallel 3D FFT. *Computer Science - Research and Development*, 26(3):237–246, 2011. pages 85

[60] K. Kandalla, H. Subramoni, J. Vienne, S. P. Raikar, K. Tomko, S. Sur, and D. K. Panda. Designing Non-blocking Broadcast with Collective Offload on InfiniBand Clusters: a Case Study with HPL. In *Proc. 19th IEEE Annual Sym-*

posium on High Performance Interconnects (HOTI'11), pages 27–34, Santa Clara, CA, USA, 2011. pages 85

[61] R. M. Karp, A. Sahay, E. E. Santos, and K. E. Schauser. Optimal Broadcast and Summation in the LogP Model. In *Proc. 5th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA'93)*, pages 142–153, Velen, Germany, 1993. pages 59

[62] J. Kepner. High Performance Computing Productivity Model Synthesis. *Intl. Journal of High Performance Computing Applications*, 18(4):505–516, 2004. pages 115

[63] M. Klemm, M. Bezold, R. Veldema, and M. Philippsen. JaMP: an Implementation of OpenMP for a Java DSM. *Concurrency and Computation: Practice and Experience*, 19(18):2333–2352, 2007. pages 7

[64] H. Kredel. Java Algebra System (JAS) Project. `http://krum.rz.uni-mannheim.de/jas/`. [Last visited: June 2013]. pages 8

[65] H. Kredel. On the Design of a Java Computer Algebra System. In *Proc. 4th Intl. Symposium on Principles and Practice of Programming in Java (PPPJ'06)*, pages 143–152, Mannheim, Germany, 2006. pages 8

[66] S. Kumar, G. Dózsa, G. Almási, P. Heidelberger, D. Chen, M. E. Giampapa, M. Blocksome, A. Faraj, J. Parker, J. Ratterman, B. Smith, and C. J. Archer. The Deep Computing Messaging Framework: Generalized Scalable Message Passing on the Blue Gene/P Supercomputer. In *Proc. 22nd ACM Intl. Conf. on Supercomputing (ICS'08)*, pages 94–103, Island of Kos, Greece, 2008. pages 85

[67] S. Kumar, G. Dózsa, J. Berg, B. Cernohous, D. Miller, J. Ratterman, B. E. Smith, and P. Heidelberger. Architecture of the Component Collective Messaging Interface. In *Proc. 15th European PVM/MPI Users' Group Meeting (EuroPVM/MPI'08)*, pages 23–32, Dublin, Ireland, 2008. pages 85

[68] S. Kumar, P. Heidelberger, D. Chen, and M. Hines. Optimization of Applications with Non-blocking Neighborhood Collectives via Multisends on the Blue Gene/P Supercomputer. In *Proc. 24th IEEE Intl. Parallel and Distributed*

Processing Symposium (IPDPS'10), 11 pages, Atlanta, GA, USA, 2010. pages 85

[69] W. Lawry, C. Wilson, A. Maccabe, and R. Brightwell. COMB: a Portable Benchmark Suite for Assessing MPI Overlap. In *Proc. 4th IEEE Intl. Conf. on Cluster Computing (CLUSTER'02)*, pages 472–475, Chicago, IL, USA, 2002. pages 84

[70] D. Lea. A Java Fork/Join Framework. In *Proc. ACM 2000 Java Grande Conf. (JAVA'00)*, pages 36–43, San Francisco, CA, USA, 2000. pages 6

[71] A. Leist, D. Playne, and K. Hawick. Exploiting Graphical Processing Units for Data-Parallel Scientific Applications. *Concurrency and Computation: Practice and Experience*, 21(18):2400–2437, 2009. pages 11, 22

[72] S. Li, T. Hoefler, and M. Snir. NUMA-Aware Shared Memory Collective Communication for MPI. In *Proc. 22nd ACM Intl. Symposium on High-Performance Parallel and Distributed Computing (HPDC'13)*, pages 85–96, New York City, NY, USA, 2013. pages 58

[73] S. Lim, B. Carpenter, G. Fox, and H. Lee. Collective Communications for Scalable Programming. In *Proc. 3rd Intl. Symposium on Parallel and Distributed Processing and Applications (ISPA'05)*, pages 286–297, Nanjing, China, 2005. pages 58

[74] Linear Algebra for Java (jblas). `http://jblas.org/`. [Last visited: June 2013]. pages 8

[75] E. Lusk and K. Yelick. Languages for High-Productivity Computing: the DARPA HPCS Language Project. *Parallel Processing Letters*, 17(1):89–102, 2007. pages 115

[76] T. Ma, G. Bosilca, U. Bouteiller, and J. J. Dongarra. Kernel-assisted and Topology-aware MPI Collective Communications on Multi-core/Many-core Platforms. *Journal of Parallel and Distributed Computing*, 2013 (In press). pages 29, 58

[77] Magny-Cours and Direct Connect Architecture 2.0. `http://developer.amd.com/documentation/articles/pages/Magny-Cours-Direct-Connect-Architecture-2.0.aspx`. [Last visited: June 2013]. pages 17, 40

[78] D. A. Mallón, G. L. Taboada, C. Teijeiro, J. Touriño, B. B. Fraguela, A. Gómez, R. Doallo, and J. C. Mouriño. Performance Evaluation of MPI, UPC and OpenMP on Multicore Architectures. In *Proc. 16th European PVM/MPI Users' Group Meeting (EuroPVM/MPI'09)*, pages 174–184, Espoo, Finland, 2009. pages 7

[79] D. A. Mallón, G. L. Taboada, J. Touriño, and 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)*, pages 181–190, Weimar, Germany, 2009. pages 48, 75, 76

[80] J. Manzano, Y. Zhang, and G. Gao. P3I: The Delaware Programmability, Productivity and Proficiency Inquiry. In *Proc. 2nd Intl. Workshop on Software Engineering for High Performance Computing (SE-HPC'05)*, pages 32–36, St. Louis, MO, USA, 2005. pages 115

[81] Matrix Toolkits Java (MTJ). `https://github.com/fommil/matrix-toolkits-java`. [Last visited: June 2013]. pages 8

[82] T. J. McCabe. A Complexity Measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, 1976. pages 115

[83] G. Mercier and J. Clet-Ortega. Towards an Efficient Process Placement Policy for MPI Applications in Multicore Environments. In *Proc. 16th European PVM/MPI Users' Group Meeting (EuroPVM/MPI'09)*, pages 104–115, Espoo, Finland, 2009. pages 58

[84] D. Millot, A. Muller, C. Parrot, and F. Silber-Chaussumier. STEP: a Distributed OpenMP for Coarse-Grain Parallelism Tool. In *Proc. 4th Intl. Workshop on OpenMP (IWOMP'08)*, pages 83–99, West Lafayette, IN, USA, 2008. pages 7

[85] J. Moreira, S. Midkiff, M. Gupta, P. Artigas, M. Snir, and R. Lawrence. Java Programming for High-Performance Numerical Computing. *IBM Systems Journal*, 39(1):21–56, 2000. pages 8

[86] MPI 3.0 Standardization Effort. `http://meetings.mpi-forum.org/MPI_3.0_main_page.php`. [Last visited: June 2013]. pages 84

[87] S. Nanz, S. West, and K. S. da Silveira. Benchmarking Usability and Performance of Multicore Languages. Technical report, ETH Zürich and Google Inc., Zürich, Switzerland, 2013. pages 115

[88] NAS Parallel Benchmarks. `http://www.nas.nasa.gov/publications/npb.html`. [Last visited: June 2013]. pages 48

[89] A. Nelisse, J. Maassen, T. Kielmann, and H. E. Bal. CCJ: Object-Based Message-Passing and Collective Communication in Java. *Concurrency and Computation: Practice and Experience*, 15(3-5):341–369, 2003. pages 58

[90] NetPIPE. A Network Protocol Independent Performance Evaluator. `http://www.scl.ameslab.gov/netpipe/`. [Last visited: June 2013]. pages 45

[91] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable Parallel Programming with CUDA. *Queue*, 6(2):40–53, 2008. pages 11

[92] R. Nishtala, Y. Zheng, P. H. Hargrove, and K. A. Yelick. Tuning Collective Communication for Partitioned Global Address Space Programming Models. *Parallel Computing*, 37(9):576–591, 2011. pages 85

[93] A. Nomura and Y. Ishikawa. Design of Kernel-Level Asynchronous Collective Communication. In *Proc. 17th European MPI Users' Group Meeting (EuroMPI'10)*, pages 92–101, Stuttgart, Germany, 2010. pages 85

[94] A. Nomura, Y. Ishikawa, N. Maruyama, and S. Matsuoka. Design and Implementation of Portable and Efficient Non-blocking Collective Communication. In *Proc. 12th IEEE/ACM Intl. Symposium on Cluster, Cloud and Grid Computing (CCGrid'12)*, pages 1–8, Ottawa, ON, Canada, 2012. pages 85

[95] NVIDIA. NVIDIA's Next Generation CUDA™ Compute Architecture: Kepler™ GK110. NVIDIA Whitepaper v.10 (24 pages), 2012. pages 23

[96] Open JDK. Project Sumatra. `http://openjdk.java.net/projects/sumatra/`. [Last visited: June 2013]. pages 12, 104

[97] OpenCL* Design and Programming Guide for the Intel® Xeon Phi™ Coprocessor. `http://software.intel.com/en-us/articles/opencl-design-and-programming-guide-for-the-intel-xeon-phi-coprocessor`. [Last visited: June 2013]. pages 110

[98] R. E. Park. Software Size Measurement: a Framework for Counting Source Statements. Technical report, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, USA, 1992. pages 115

[99] I. Patel and J. Gilbert. An Empirical Study of the Performance and Productivity of Two Parallel Programming Models. In *Proc. 22nd IEEE Intl. Parallel and Distributed Processing Symposium (IPDPS'08)*, 7 pages, Miami, FL, USA, 2008. pages 115

[100] D. Petrović, O. Shahmirzadi, T. Ropars, and A. Schiper. High-performance RMA-based Broadcast on the Intel SCC. In *Proc. 24th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'12)*, pages 121–130, Pittsburgh, PA, USA, 2012. pages 21

[101] J. Pjesivac-Grbovic, T. Angskun, G. Bosilca, G. E. Fagg, E. Gabriel, and J. J. Dongarra. Performance Analysis of MPI Collective Operations. *Cluster Computing*, 10(2):127–143, 2007. pages 57

[102] J. Pjesivac-Grbovic, G. E. Fagg, T. Angskun, G. Bosilca, and J. J. Dongarra. MPI Collective Algorithm Selection and Quadtree Encoding. In *Proc. 13th European PVM/MPI Users' Group Meeting (EuroPVM/MPI'06)*, pages 40–48, Bonn, Germany, 2006. pages 57

[103] S. Potluri, P. Lai, K. Tomko, S. Sur, Y. Cui, M. Tatineni, K. W. Schulz, W. L. Barth, A. Majumdar, and D. K. Panda. Quantifying Performance Benefits of Overlap using MPI-2 in a Seismic Modeling Application. In *Proc. 24th ACM Intl. Conf. on Supercomputing (ICS'10)*, pages 17–25, Tsukuba, Japan, 2010. pages 84

[104] P. Pratt-Szeliga, J. Fawcett, and R. Welch. Rootbeer: Seamlessly using GPUs from Java. In *Proc. 14th IEEE Intl. Conf. on High Performance Computing and Communications (HPCC'12)*, pages 375–380, Liverpool, UK, 2012. pages 13

[105] B. Pugh and 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)*, pages 323–339, College Station, TX, USA, 2003. pages 56

[106] S. Ramos and T. Hoefler. Modeling Communication in Cache-Coherent SMP Systems - A Case-Study with Xeon Phi. In *Proc. 22nd ACM Intl. Symposium on High-Performance Parallel and Distributed Computing (HPDC'13)*, pages 97–108, New York City, NY, USA, 2013. pages xv, 3, 103

[107] S. Ramos, G. L. Taboada, R. R. Expósito, and J. Touriño. Nonblocking Collectives for Scalable Java Communications. 2013 (Submitted for journal publication). pages xv, 3, 84, 86

[108] S. Ramos, G. L. Taboada, R. R. Expósito, J. Touriño, and R. Doallo. Design of Scalable Java Communication Middleware for Multi-Core Systems. *The Computer Journal*, 56(2):214–228, 2013. pages xiv, 3, 84, 90

[109] S. Ramos, G. L. Taboada, J. Touriño, and R. Doallo. Scalable Java Communication Middleware for Hybrid Shared/Distributed Memory Architectures. In *Proc. 13th Intl. Conf. on High Performance Computing and Communications (HPCC'11)*, pages 221–228, Banff, AB, Canada, 2011. pages xiv, xiv, 3, 63, 125

[110] S. Saini, R. Ciotti, B. T. Gunney, T. E. Spelce, A. Koniges, D. Dossa, P. A. Adamidis, R. Rabenseifner, S. R. Tiyyagura, and M. Müller. Performance Evaluation of Supercomputers using HPCC and IMB Benchmarks. *Journal of Computer and System Sciences*, 74(6):965–982, 2008. pages 41, 82

[111] R. S. Saksena. On Non-blocking Collectives in 3D FFTs. In *Proc. 2nd Workshop on Scalable Algorithms for Large-scale Systems (ScalA'11)*, pages 15–18, Seattle, WA, USA, 2011. pages 85

[112] J. C. Sancho, K. J. Barker, D. J. Kerbyson, and K. Davis. Quantifying the Potential Benefit of Overlapping Communication and Computation in Large-scale Scientific Applications. In *Proc. 20th ACM/IEEE Intl. Conf. for High Performance Computing, Networking, Storage and Analysis (SC'06)*, page 125 (17 pages), Tampa, FL, USA, 2006. pages 84

[113] P. Sanders and J. L. Träff. The Hierarchical Factor Algorithm for All-to-All Communication. In *Proc. 8th Intl. Euro-Par Conf. (Euro-Par'02)*, pages 799–804, Padeborn, Germany, 2002. pages 57

[114] T. Schneider, S. Eckelmann, T. Hoefler, and W. Rehm. Kernel-Based Offload of Collective Operations - Implementation, Evaluation and Lessons Learned. In *Proc. 17th Intl. Euro-Par Conf. (Euro-Par'11)*, pages 264–275, Bordeaux, France, 2011. pages 85

[115] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan. Larrabee: a Many-core x86 Architecture for Visual Computing. *ACM Transactions on Graphics*, 27(3):18:1–18:15, 2008. pages 20

[116] A. Shafi, B. Carpenter, and M. Baker. Nested Parallelism for Multi-core HPC Systems using Java. *Journal of Parallel and Distributed Computing*, 69(6):532–545, 2009. pages 9, 29, 56, 63, 84

[117] A. Shafi, B. Carpenter, M. Baker, and A. Hussain. Study of Java and C Performance in Two Large-scale Parallel Applications. *Concurrency and Computation: Practice and Experience*, 21(15):1882–1906, 2009. pages 76

[118] A. Shafi, J. Manzoor, K. Hameed, B. Carpenter, and M. Baker. Multicore-enabling the MPJ Express Messaging Library. In *Proc. 8th Intl. Conf. on Principles and Practice of Programming in Java (PPPJ'10)*, pages 49–58, Vienna, Austria, 2010. pages XIV, 29, 31

[119] M. Snir and D. Bader. A Framework for Measuring Supercomputer Productivity. *Intl. Journal of High Performance Computing Applications*, 18(4):417–432, 2004. pages 115

[120] J. 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 11

[121] 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 xiv, 3, 5, 6, 8

[122] G. L. Taboada, S. Ramos, J. Touriño, and R. Doallo. Design of Efficient Java Message-passing Collectives on Multi-core Architectures. *Journal of Supercomputing*, 55(2):126–154, 2011. pages xv, 3, 56, 74, 100

[123] 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 10, 27, 29, 30, 56

[124] G. L. Taboada, J. Touriño, and 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)*, pages 118–126, Hong Kong, China, 2003. pages 41, 66, 69

[125] G. Tanase, G. Almási, C. Archer, and H. Xue. Hybrid Collective Operations on Power7 IH. In *Proc. 26th ACM Intl. Conf. on Supercomputing (ICS'12)*, pages 215–224, Venice, Italy, 2012. pages 85

[126] C. Teijeiro, G. L. Taboada, J. Touriño, B. B. Fraguela, R. Doallo, D. A. Mallón, A. Gómez, J. C. Mouriño, and B. Wibecan. Evaluation of UPC Programmability using Classroom Studies. In *Proc. 3rd Conf. on Partitioned Global Address Space Programming Models (PGAS'09)*, page 10 (7 pages), Ashburn, VA, USA, 2009. pages 115

[127] R. Thakur, R. Rabenseifner, and W. Gropp. Optimization of Collective Communication Operations in MPICH. *Intl. Journal of High Performance Computing Applications*, 19(1):49–66, 2005. pages 57

[128] The Open MPI Project. `http://www.open-mpi.org/`. [Last visited: June 2013]. pages 10

[129] The Open MPI Project. Java Support. http://www.open-mpi.org/faq/?category=java. [Last visited: June 2013]. pages 10

[130] The Open MPI Project. Open MPI Shared Memory Communications. http://www.open-mpi.org/faq/?category=sm. [Last visited: June 2013]. pages 28, 84

[131] The STREAM2 Homepage. http://www.cs.virginia.edu/stream/stream2/. [Last visited: June 2013]. pages 47

[132] M. Thompson. LMAX Disruptor. High Performance Inter-Thread Messaging Library. http://lmax-exchange.github.com/disruptor/. [Last visited: June 2013]. pages 7

[133] V. Tipparaju, J. Nieplocha, and D. K. Panda. Fast Collective Operations using Shared and Remote Memory Access Protocols on Clusters. In *Proc. 17th IEEE Intl. Parallel and Distributed Processing Symposium (IPDPS'03)*, pages 84–93, Nice, France, 2003. pages 58

[134] TOP500 Supercomputing Site. http://www.top500.org. [Last visited: June 2013]. pages 11, 20, 23, 103

[135] F. Trahay, E. Brunet, and A. Denis. An Analysis of the Impact of Multi-threading on Communication Performance. In *Proc. 9th Workshop on Communication Architecture for Clusters (CAC'09)*, page 142 (7 pages), Rome, Italy, 2009. pages 85

[136] F. Trahay, E. Brunet, A. Denis, and R. Namyst. A Multithreaded Communication Engine for Multicore Architectures. In *Proc. 8th Workshop on Communication Architecture for Clusters (CAC'08)*, page 167 (7 pages), Miami, FL, USA, 2008. pages 85

[137] TROVE, High Performance Collections for Java. http://trove.starlight-systems.com/. [Last visited: June 2013]. pages 8

[138] B. Tu, J. Fan, J. Zhan, and X. Zhao. Performance Analysis and Optimization of MPI Collective Operations on Multi-core Clusters. *Journal of Supercomputing*, 60(1):141–162, 2012. pages 58

[139] D. Turner, A. Oline, C. Xuehua, and T. Benjegerdes. Integrating New Capabilities into NetPIPE. In *Proc. 10th European PVM/MPI Users' Group Meeting (EuroPVM/MPI'03)*, pages 37–44, Venice, Italy, 2003. pages 45

[140] Universal Java Matrix Package (UJMP). `http://www.ujmp.org`. [Last visited: June 2013]. pages 8

[141] R. Veldema, R. F. H. Hofman, R. A. F. Bhoedjang, and H. E. Bal. Run-time Optimizations for a Java DSM Implementation. *Concurrency and Computation: Practice and Experience*, 15(3-5):299–316, 2003. pages 7

[142] V. Venkatesan, M. Chaarawi, E. Gabriel, and T. Hoefler. Design and Evaluation of Nonblocking Collective I/O Operations. In *Proc. 18th European MPI Users' Group Meeting (EuroMPI'11)*, pages 90–98, Santorini, Greece, 2011. pages 86, 97

[143] X. Wu and V. Taylor. Performance Characteristics of Hybrid MPI/OpenMP Implementations of NAS Parallel Benchmarks SP and BT on Large-Scale Multicore Clusters. *The Computer Journal*, 55(2):154–167, 2012. pages 7

[144] Y. Yan, M. Grossman, and V. Sarkar. JCUDA: a Programmer-Friendly Interface for Accelerating Java Programs with CUDA. In *Proc. 15th Intl. Euro-Par Conf. (Euro-Par'09)*, pages 887–899, Delft, The Netherlands, 2009. pages 12, 13

[145] H. Zhu, D. Goodell, W. Gropp, and R. Thakur. Hierarchical Collectives in MPICH2. In *Proc. 16th European PVM/MPI Users' Group Meeting (EuroPVM/MPI'09)*, pages 325–326, Espoo, Finland, 2009. pages 57

[146] H. Zuse. *Framework of Software Measurement*, chapter History of Software Measurement, pages 57–80. Walter de Gruyter, 1998. pages 115