Design of scalable Java communication middleware for multi-core systems

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

Computer Architecture Group, Dep. Electronics & Systems, University of A Coruña, Campus de Elviña s/n, 15071 A Coruña (Spain)
Email: {sramos, taboada, rreye, juan, doallo}@udc.es

This paper presents `smdev`, a shared memory communication middleware for multi-core systems. `smdev` provides a simple and powerful messaging API that is able to exploit the underlying multi-core architecture replacing inter-process and network-based communications by threads and shared memory transfers. The performance evaluation of `smdev` on several multi-core systems has shown noticeable improvements compared to other Java shared memory solutions, reaching and even overcoming the performance of natively compiled libraries. Thus, `smdev` has obtained start-up latencies around 0.76 μs and almost 90 Gbps bandwidth for point-to-point communications, as well as high performance and scalability both for collective operations and representative messaging kernels. This fact has motivated the integration of `smdev` in F-MPJ, our message-passing implementation in Java.

Keywords: Parallel Programming; Java Multi-threading; Shared Memory; Multi-core Architectures; Message-Passing in Java (MPJ)

1. INTRODUCTION

Java is the leading programming language both in academia and industry environments. Nowadays, thanks to the continuous advances in the Java Virtual Machine (JVM) technology and Just-In-Time (JIT) compilation, it is able to generate native executable code from the platform-independent bytecode, which is reducing the gap with natively compiled languages (e.g., C/C++) according to [1], enabling the use of Java in performance-bounded scenarios as real-time environments [2]. Furthermore, Java provides some interesting characteristics for parallel programming [3] such as built-in networking and multi-threading support, automatic memory management, platform independence, portability, security, object orientation, an extensive API and a wide community of developers.

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. The most extended API in natively compiled languages is MPI [4], whereas in Java it is MPJ (Message-Passing in Java) [5]. Nevertheless, current message-passing implementations generally do not take full advantage of multi-threading for intra-process transfers, and they 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. The communication middleware presented in this paper, `smdev`, provides a high-level message-passing API while taking full advantage of these multi-core architectures using multi-threading and shared memory transfers.

Multi-threading allows to exploit shared memory intra-process transfers, but thread programming increases the development complexity due to the need for thread control and management, task scheduling, synchronization, access and maintenance of shared data structures, and the presence of thread safety concerns.

Using the smdev messaging API, the developer does not have to deal with threads as it offers a simple API with 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 F-MPJ [6], our Java message-passing implementation. This integration supports the efficient execution of any MPJ application on multi-core systems.

The structure of this paper is as follows: Section 2 introduces the related work. Section 3 describes the design, implementation and operation of the developed middleware. Section 4 presents the analysis of the performance of smdev on two representative systems, evaluated comparatively against MPI and thread-based counterpart codes. Section 5 summarizes our concluding remarks.

2. RELATED WORK

The continuous increase in the number of available cores per processor emphasizes the need for scalable solutions in parallel programming to exploit multi-core shared memory architectures. Traditionally, the approach followed in compiled languages, such as C/C++, is the use of shared memory models like POSIX threads (pthreads) or OpenMP. However, the code developed with these approaches is limited to shared memory systems. In order to overcome this limitation, several tools that execute multi-threaded applications on distributed memory architectures have been proposed but, up to now, either their implementation is based on software translations to MPI [7] or it relies on Distributed Shared Memory (DSM) systems [8]. Another option is the use of a hybrid shared/distributed memory programming model combining MPI for inter-node communications and resort to a shared memory model to take advantage of intra-node parallelism [9]. 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 [10].

Java, thanks to its built-in multi-threading support, is widely used for shared memory programming. 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 computing parallel codes, where threads cooperate to reduce the runtime of a workload. Finally, codes developed using threads or tasks cannot run on distributed memory environments, unless relying on a Java DSM which generally involves portability issues due to the use of modified JVMs. The Parallel Java project [11] provides several abstractions over these concurrency utilities, also implementing the message-passing paradigm for distributed memory but providing its own interface. There are also OpenMP-like Java implementations such as JOMP [12] and JaMP [13]. 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 [14], a software-based Java DSM implementation, and its main drawback is the lack of portability since it cannot run on standard JVMs.

MPI libraries, such as MPICH2 and OpenMPI, are mostly optimized for distributed memory communications, although they are increasingly taking advantage of multi-core shared memory systems. Thus, the MPICH2 project includes several communication devices for shared memory such as ssm, shm or sshm [15]. It also supports Nemesis [16], 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. OpenMPI includes optimized communications among processes via shared memory (sm Byte Transfer Layer) [17] 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.

Although there are several message-passing projects in Java [3], Fast MPJ (F-MPJ) [6] and MPJ Express [18] currently have the most active development. They both include a modular design with a pluggable architecture of communication devices which allows to combine the portability of the "pure" Java communication devices with high performance network support wrapping native communication libraries through JNI (Java Native Interface). Additionally, MPJ Express provides shared memory support [19], whereas F-MPJ takes advantage of the middleware presented in this paper, smdev. Its buffer layer avoidance and the reduced synchronization overhead improve significantly the scalability of F-MPJ when communications involve a large number of MPJ processes. In [20], the MPJ Express shared memory device was found to be the main bottleneck in the development of a hybrid shared/distributed memory communication middleware. Furthermore, F-MPJ also outperforms MPJ Express for collective operations as it includes a scalable collectives library tuned for multi-core systems [21] which takes advantage of the reduced overhead of shared memory smdev transfers.

3. SMDEV: SCALABLE JAVA COMMUNICATIONS FOR SHARED MEMORY

This section presents the `smdev` communication middleware: its API, implementation and integration in F-MPJ.

3.1. `smdev` Message-Passing API

The `smdev` middleware provides a message-passing API that conforms with the `xxdev` API [6] (see Figure 1), which avoids data buffering by supporting direct communication of any serializable object.

It is composed of basic operations such as point-to-point communications, both blocking (send and recv) and non-blocking (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.

```
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();

}
```

FIGURE 1. API of the `xxdev.Device` class.

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 multi-threading programming.

3.2. `smdev` Implementation

The goal of `smdev` is to increase the scalability of Java applications through the use of 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 upper graph in Figure 2), whereas `smdev` runs several threads within a single JVM instance (see bottom 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. Thus `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, consequence of the fragmentation and multiple JVMs memory consumption.

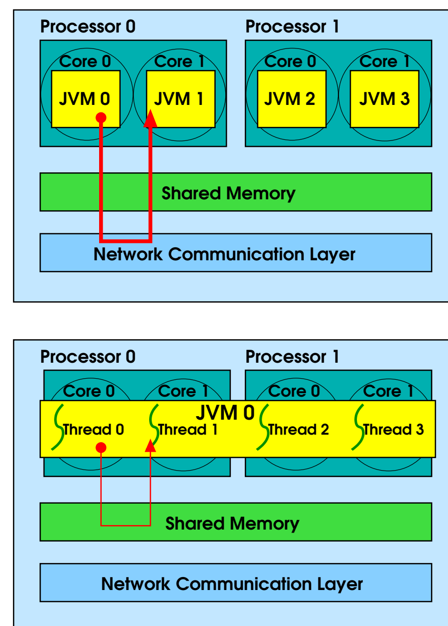


FIGURE 2. Java communications on a two-core dual processor using distributed (top) and shared (bottom) memory-oriented middleware.

The implementation of `smdev` over shared memory required handling with JVM class loaders to maintain shared structures (message queues) for the communication, involving also 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.

3.2.1. Class Loading in `smdev`

The use of threads in `smdev` for running as message-passing processes requires the isolation of the name space for each running thread, configuring a distributed memory space in which threads can exchange messages through shared memory references. This management relies on custom class loaders, a mechanism similar to the one used in MPJ Express [19].

The purpose of the name space isolation is 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. Thus, the user classes and the top-level smdev classes, as well as some related to the device management, have to be isolated to behave like independent processes. Nevertheless, the communication through shared memory transfers requires the access to several shared classes within the device.

A JVM identifies each loaded class by its fully qualified name and its class loader, so each loader defines its own name space. 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, which is in our case the custom class loader of the thread, is used. This mechanism implies that the system class loader is going to load every reachable class that, in consequence, is shared by all threads. Thus, its classpath has to be bounded in such a way that it only has access to shared packages (runtime and smdev) that contain the implementation of shared memory transfers among threads.

The class loading particularities of smdev also affects 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 less conflicts with the class loader structure than Cloneable. Besides, 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.

3.2.2. Message Queues

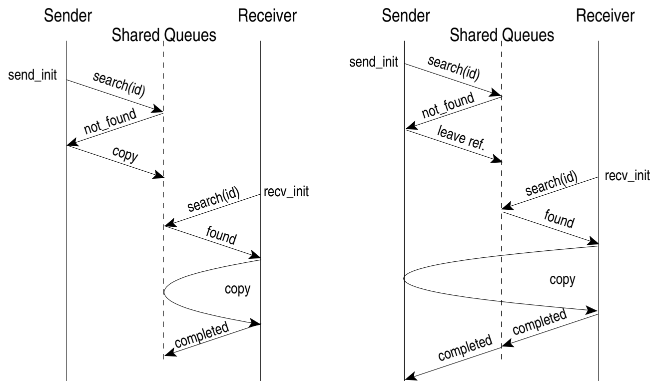
Communications in smdev are implemented by shared memory transfers. Thus, point-to-point communication operations delegate on a shared class which manages shared message queues to handle pending communication requests for sends and receives. Each thread has two queues assigned, one for the incoming messages 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.

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.

3.2.3. Message Transfer Protocols

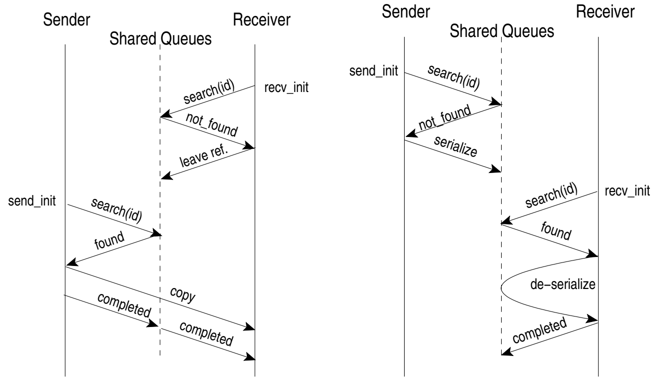
Sends and receives rely on the shared message queues already described, using as message identification the source identifier, a user tag and a context, which is managed internally by the device. In order to cope with duplicity of message identification, the sequence number is also taken into account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



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

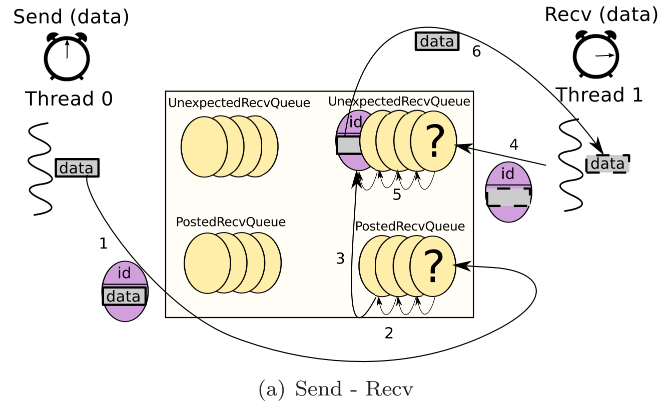
(b) Large message, starting with Send (Rendez-vous)



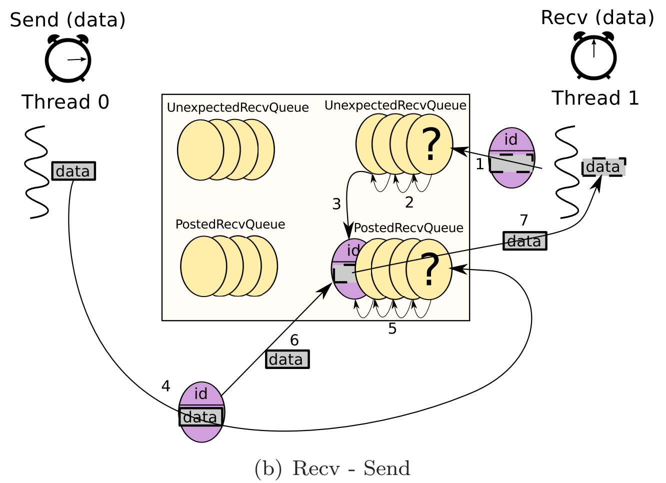
(c) Small or large message, starting with Recv

(d) Serializable message

FIGURE 3. Communication protocols in `smdev`.



(a) Send - Recv



(b) Recv - Send

FIGURE 4. Send/Recv operations in `smdev`.

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 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(a)), the sender copies the message data in the request buffer and assumes the communication as completed. Then, 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 the use of a rendez-vous protocol (Figure 3(b)), 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 completed until the receiver has copied the data. The boundary of data size 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(c)), 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(d)), 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. This makes unavoidable to store the serialized data in the request buffer, independently of the message size.

Figure 4 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 4(a)), Thread 0 sends a message (step 1) before Thread 1 posts the corresponding receive request. After checking the `PostedRecvQueue` of the destination for a matching request without success (step 2), the sender enqueues the send request in the `UnexpectedRecvQueue` (step

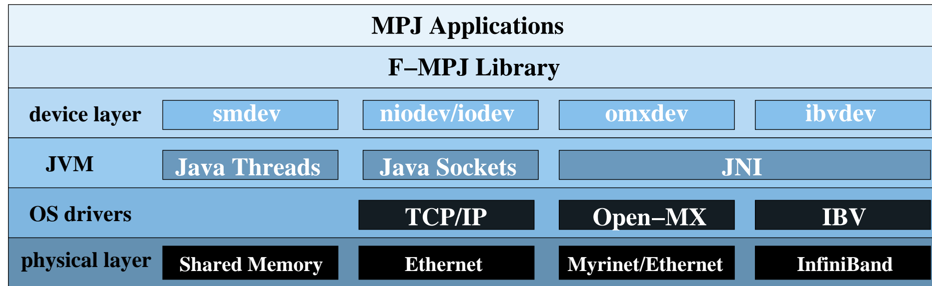


FIGURE 5. F-MPJ communication devices on shared memory and cluster networks.

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 4(b)), 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).

3.2.4. Synchronization

Synchronization is one of the main performance bottlenecks in shared memory communications. In `smdev`, synchronization among threads guarantees 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 completed in a short period of time. In this case, `smdev` trades off latency for CPU consumption contributing to code scalability. Besides, 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 pairing queue, creating a dependence condition in the consistency of the queues.

3.3. Integration of `smdev` in F-MPJ

The developed middleware has been integrated in the F-MPJ library providing Java message-passing applications with efficient support for shared memory communications.

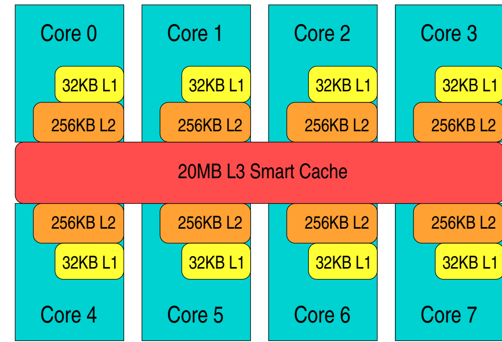
Figure 5 shows the communication support implemented in F-MPJ, either on JVM threads (`smdev`), sockets over TCP/IP stack (`niodev` and `iodev`), or on native communication layers such as Open-MX (`omxdev`) and InfiniBand Verbs (IBV) (`ibvdev`), which are accessed through JNI. `smdev` is the only F-MPJ device that takes advantage of intra-process shared memory transfers for point-to-point communications.

The integration has been almost transparent to the rest of F-MPJ layers thanks to the modular structure of the device layer. The upper layers of F-MPJ rely on the point-to-point `xxdev` API primitives from the communication devices, thus all the operations and algorithms from F-MPJ, such as the collective operations library, can benefit from the use of `smdev` without further knowledge of the communication system. 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 3.2.1).

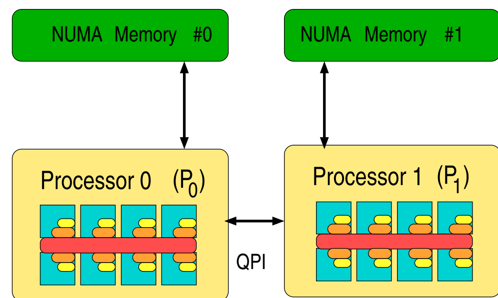
3.3.1. Collective Operations in `smdev`

F-MPJ includes a collectives library with multi-core-aware optimized algorithms [21] that runs on

top of the device layer, and therefore on smdev. These algorithms include Minimum Spanning Tree, BiDirectional Exchange, Bucket or Cyclic, Binomial Trees and Flat Trees. smdev also provides, through an extension of the xxdev API, its own implementation of collectives without relying on point-to-point primitives. The use of these custom algorithms allows to perform only one call to smdev per collective. Moreover, these operations optimize the use of the shared queues by using less explicit synchronizations since the communication pattern is already known. As an example, in the Broadcast algorithm, the root thread uses an atomic variable to indicate about 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 in 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 notification system establishes the order of operation, avoiding the need to check the queues for already-arrived messages. Similar algorithms have been implemented for the rest of collective operations.



(a) Intel Xeon E5-2670 processor



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

FIGURE 6. Intel Xeon E5-2670 system.

4. PERFORMANCE EVALUATION

The performance evaluation of smdev consists of a micro-benchmarking of point-to-point and collective operations, and an analysis of the impact of smdev on representative parallel codes.

4.1. Experimental Configuration

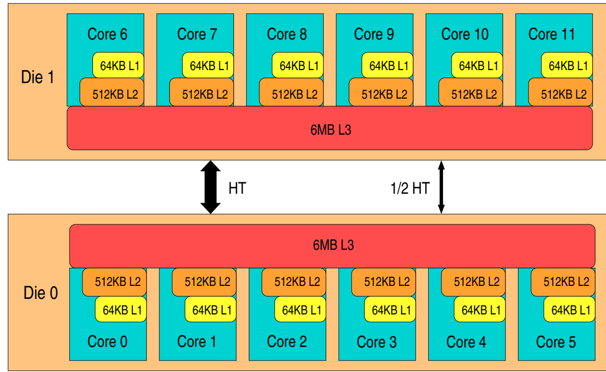
The developed middleware has been evaluated on two representative multi-core systems, a 16-core Intel-based and a 48-core AMD-based. The first one ("Xeon E5") has 2 Intel Xeon E5-2670 8-core processors at 2.6 GHz [22] and 64 GBytes of RAM. The OS is Linux CentOs with kernel 2.6.35, the GNU compilers are v4.4.4 and the JVM is OpenJDK Runtime Environment 1.6.0_20 (IcedTea6 1.9.8).

Figure 6(a) presents the layout of the 8-core Xeon E5 processor, based on the Sandy-Bridge-E architecture, where up to 16 threads can run simultaneously thanks to hyperthreading. The eight cores in this processor share the Level 3 cache, implemented as an Intel Smart Cache, where each core can access the whole cache when the rest of the cores are idle. Figure 6(b) shows the interconnection layout in a dual-socket Intel Xeon E5-2670 system where the processors and the memory are linked by an Intel QuickPath Interconnect (QPI). This NUMA system supports DDR3-1600 MHz memory.

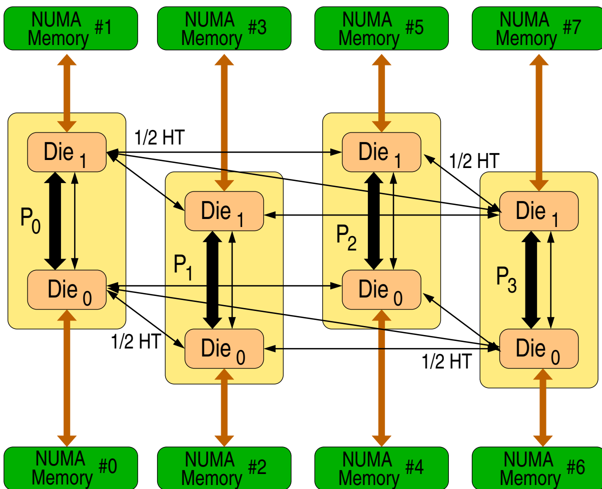
The second system, a fat node from the DAS-4 cluster [23], has 48 cores in 4 AMD Opteron 6172 processors ("Magny-Cours"), each one with 12 cores [24] [25] and 128 GBytes of RAM. The OS is Linux CentOS with kernel 2.6.32, the GNU compilers are v4.3.4 and the JVM is Sun JDK 1.6.0_05.

Figure 7(a) presents the layout of the 12-core Magny-Cours processor, which is composed of two 6-core AMD Opteron Istambul dies (the 6 cores share the Level 3 cache) interconnected by Hypertransport (HT) links. Figure 7(b) shows the HT interconnections among 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) while thick ones represent full HT links (16 bits). As each 12-core processor is a NUMA system with 2 NUMA regions, the quad-socket system has eight NUMA regions.

The performance of smdev has been evaluated comparatively against 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 1.4 and OpenMPI v1.4.3 on the Xeon E5, and OpenMPI v1.4.2 on the Magny-Cours. MPICH2 results have been omitted for clarity purposes since OpenMPI 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 OpenMPI and Nemesis in MPICH2.



(a) AMD Opteron 6172 processor



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

FIGURE 7. Magny-Cours AMD Opteron 6172 system.

4.2. 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 [26], and our internal implementation of its Java counterpart.

Figures 8 and 9 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 short 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, F-MPJ point-to-point routines are direct and thin wrappers over `smdev` primitives, showing therefore quite similar performance.

To analyze the impact of the memory hierarchy on `smdev` performance, we have implemented the 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. MPI libraries also support pinning control. The impact of thread allocation on performance is analyzed in this section for point-to-point transfers.

Figure 8 shows the performance of point-to-point communications between two cores on 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 L3 cache, the specific core mapping within a processor has no impact on performance.

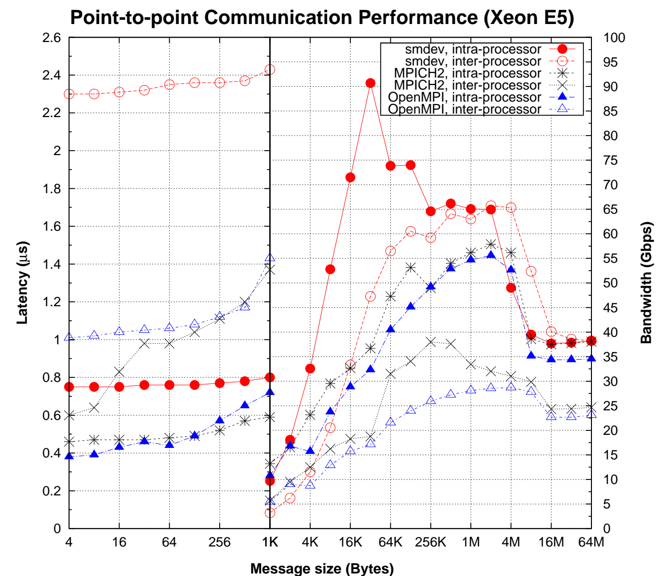


FIGURE 8. `smdev` performance on the Xeon E5.

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, for messages $\geq$ 1 KByte, `smdev` clearly outperforms MPI, achieving the best performance 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 evidences that inter-processor large-message transfers in `smdev` benefit more than intra-processor from 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 OpenMPI), and even `smdev` inter-processor transfers outperform MPI intra-processor ones.

Figure 9 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 OpenMPI.

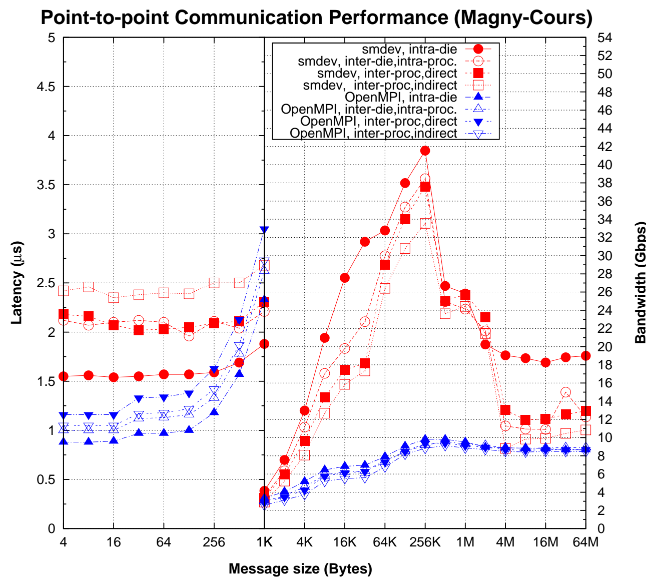


FIGURE 9. `smdev` performance on the Magny-Cours.

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 latencies are around 1 μs and `smdev` values around 1.5-2 μs. However, this superior performance of MPI for short 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 Xeon E5 and also the peak of bandwidth is obtained for 256 KBytes, while in Xeon E5 the peak is for 32 KBytes, 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 impacts severely on the performance of the communication middleware, not only for small messages, where the communication overhead is more computational-bounded than any other factor, but also for large-message bandwidth, showing around half of the Xeon E5 performance. Additionally, these results are also influenced by the performance of the memory, which has DDR3-1600 MHz support in Xeon E5 and DDR3-1333 MHz in Magny-Cours.

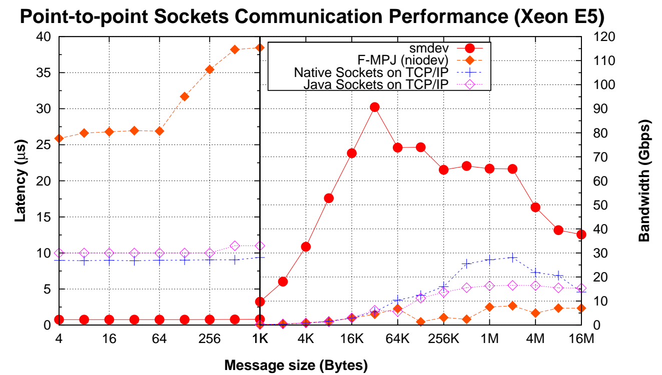


FIGURE 10. Sockets performance on the Xeon E5.

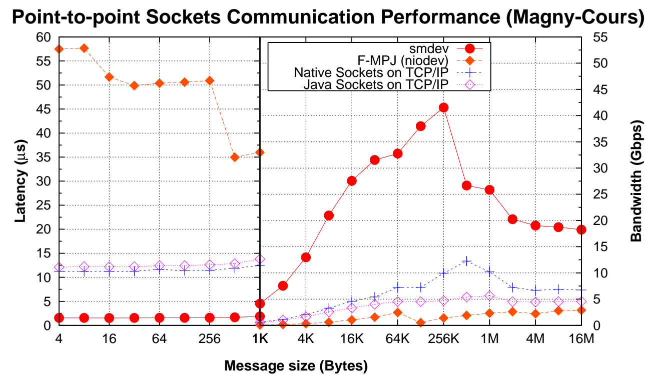


FIGURE 11. Sockets performance on the Magny-Cours.

As most Java communication middleware (e.g., JMS and RMI) is based on sockets, `smdev` performance has been evaluated comparatively against sockets using the NetPIPE benchmark suite [27] [28] on Xeon E5 (Figure 10) and Magny-Cours (Figure 11). 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 the results obtained with `smdev` and the NIO-socket communication device from F-MPJ (`niodev`). This benchmarking has been carried out with the default JVM/OS mapping policy, scheduling the threads in the intra-processor and intra-die configurations. As it comes from 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 also its non-blocking communication support imposes a high overhead.

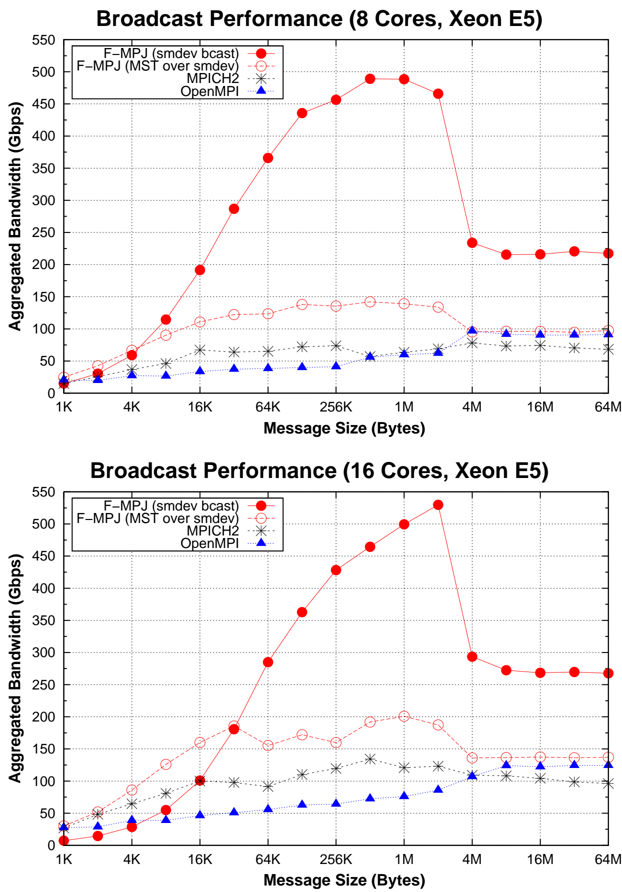


FIGURE 12. Broadcast performance on 8 and 16 cores (Xeon E5).

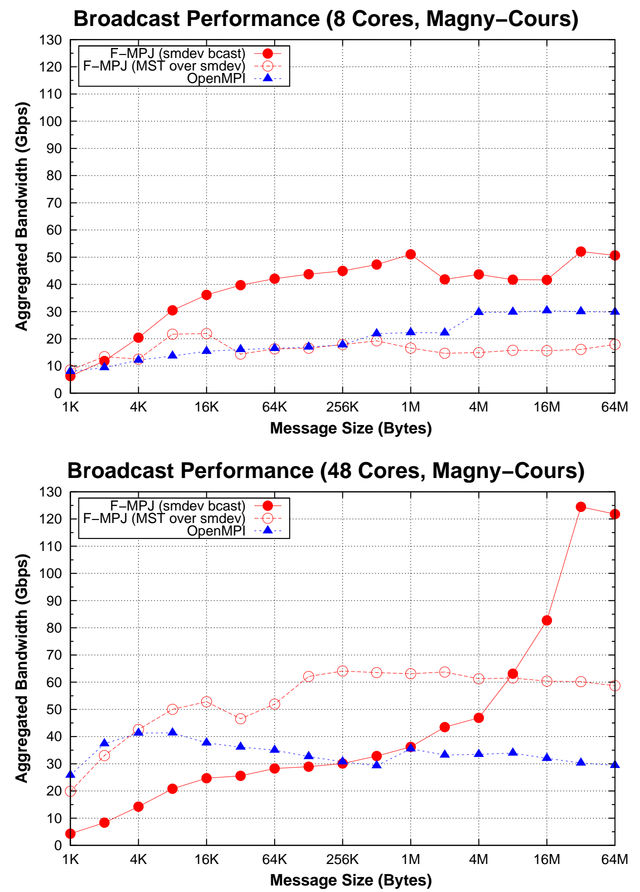


FIGURE 13. Broadcast performance on 8 and 48 cores (Magny-Cours).

4.3. Collective Operations Micro-benchmarking

The performance of collective operations has a significant impact on applications scalability. The aggregated bandwidth for the broadcast, a representative collective of data movement, has been measured on 8 and 16 cores on the Xeon E5 (Figure 12), and communicating 8 and 48 cores on the Magny-Cours testbed (Figure 13). Two algorithms have been used for `smdev`: the broadcast device implementation, presented in Section 3.3.1, and the Minimum Spanning Tree (MST) from the F-MPJ collectives library [21]. The metric used, the aggregated bandwidth, has been selected as it takes into account the global amount of transferred data.

The results from Figures 12 and 13 show that the F-MPJ broadcasts generally obtain higher bandwidth than the MPI implementations thanks to relying on a zero-copy communication protocol, as for point-to-point transfers. Regarding F-MPJ collective algorithms, the `smdev` internal implementation of the broadcast 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 this collective 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 introduce an important performance penalty. The `smdev` internal algorithm also shows poorer performance for messages up to 32 KBytes on 16 cores of the Xeon E5 system. However, the F-MPJ 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 `smdev` internal algorithm drops on Xeon E5 when the message cannot be fully stored in the L3 cache (from 2 MBytes). Due to the lower computational power and memory performance of the Magny-Cours, the achieved bandwidth is significantly lower than on Xeon E5.

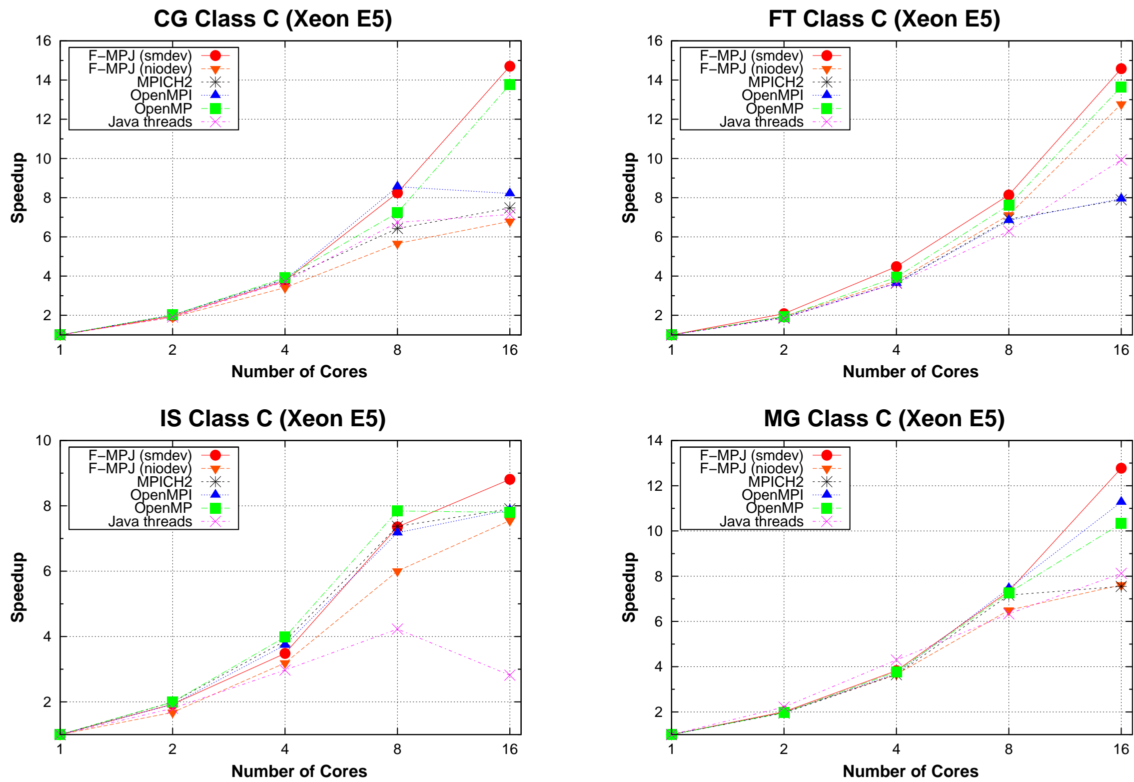


FIGURE 14. NAS parallel benchmarks performance (Xeon E5).

4.4. Scalability of `smdev` on Parallel Codes

The impact of `smdev` on the scalability of parallel codes has been analyzed with the NAS Parallel Benchmarks (NPB) [29] [30], 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) [31] have been used for this evaluation. Regarding NPB-MPJ 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: 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 number of cores power of two). Performance is shown in terms of speedup in Figures 14 and 15. With the aim of providing a reference of absolute performance, Table 1 includes performance in terms of 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 MG and FT due to the JVM start-up overhead combined with the higher maturity of the native codes, which are more refined than the Java version.

TABLE 1. MOPS of NPB codes on a single CPU core.

		CG	FT	IS	MG
Xeon E5	Native	381.8	1179.9	59.0	1741.2
	Java	379.8	695.3	52.6	1219.3
Magny-Cours	Native	201.3	711.4	58.6	847.6
	Java	168.1	461.9	45.0	548.2

Regarding the reported speedups, `smdev` is the most scalable solution in Xeon E5, and one of the best performers, together with MPI and OpenMP, in the Magny-Cours system. Regarding FT and MG, which showed the greatest performance gap among Java and native implementations using a single core, `smdev` generally obtains significant improvements in scalability. OpenMP obtains its worse results in FT and MG, where messaging implementations (MPI and F-MPJ) exploit the collective operations present in these kernels. Java threads and F-MPJ over Java NIO-sockets generally obtain the poorest results.

5. CONCLUSIONS

This paper has presented `smdev`, a shared memory Java communication middleware, which provides a simple messaging API that abstracts from thread

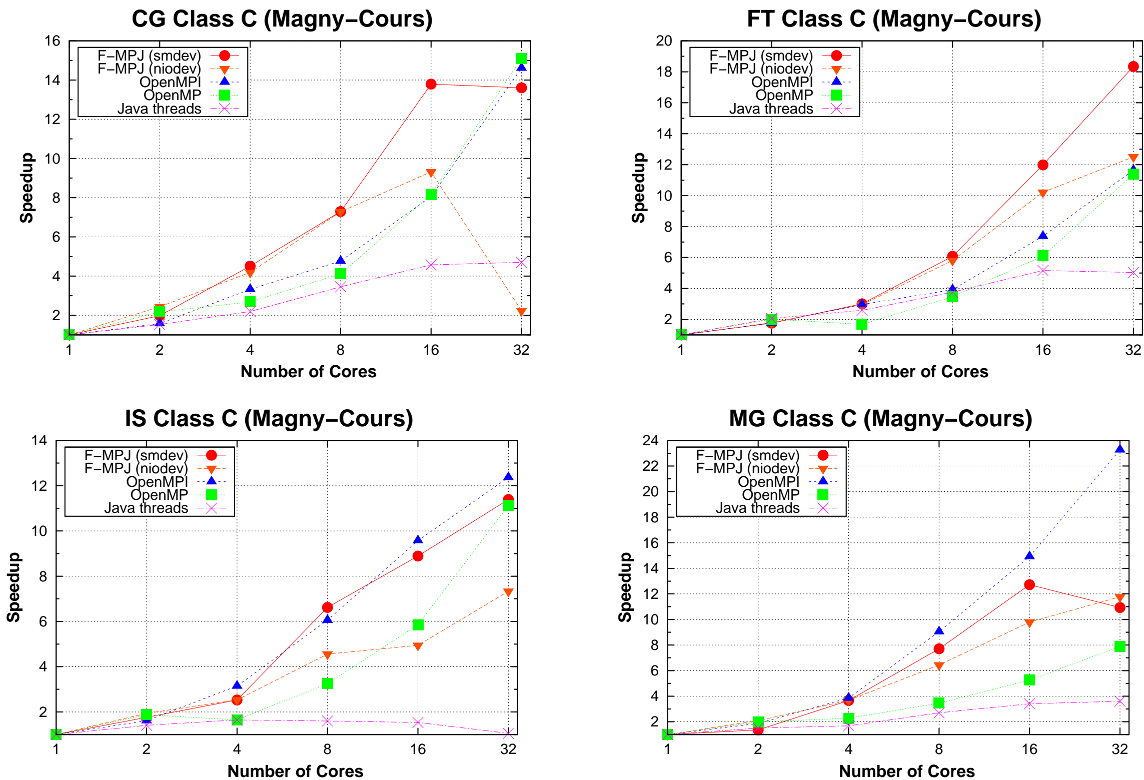


FIGURE 15. NAS parallel benchmaks performance (Magny-Cours).

programming while taking advantage of the inherent parallelism of multi-core processors. This middleware has been successfully integrated in our Java message-passing implementation, F-MPJ. Thus, any MPJ application running on distributed memory systems can also run efficiently on shared memory systems thanks to relying on `smdev`.

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, 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 more 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) `smdev` presents higher performance and scalability for collective operations than MPI; (5) the use of `smdev` in representative message-passing kernels (NPB) has generally achieved the highest speedups, which definitely contributes to bridge the performance gap between Java HPC applications and natively compiled codes; (6) `smdev` improves Java communications performance both on Intel- and AMD-based systems, taking advantage of their particular characteristics: small and fast caches in the Intel-based testbed, and generally scaling performance on the 48-core AMD-based system. Therefore, `smdev` is key for high performance Java applications on shared memory multi-core systems.

FUNDING

This work was supported by the Ministry of Science and Innovation of Spain [Project TIN2010-16735].

ACKNOWLEDGEMENTS

We gratefully thank the Advanced School for Computing and Imaging (ASCI) and VU University Amsterdam for providing access to the DAS-4 cluster.

REFERENCES

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

[2] Kim, M., and Wellings, A. (2011) Multiprocessors and Asynchronous Event Handling in the Real-Time Specifications for Java. *The Computer Journal*, **54 (8)**, 1308–1324.

[3] Taboada, G. L., Ramos, S., Expósito, R. R., Touriño, J., and Doallo, R. (2012) Java in the High Performance Computing Arena: Research, Practice and Experience. *Science of Computing Programming*, **(in press)**.

[4] Snir, M., Otto, S. W., Walker, D. W., Dongarra, J., and Huss-Lederman, S. (1995) *MPI: The Complete Reference*. MIT Press.

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

[6] Taboada, G. L., Touriño, J., and Doallo, R. (2012) F-MPJ: Scalable Java Message-passing Communications on Parallel Systems. *Journal of Supercomputing*, **60(1)**, 117–140.

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

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

[9] Wu, X., and Taylor, V. (2012) 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.

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

[11] Kaminsky, A. (2007) Parallel Java: A Unified API for Shared Memory and Cluster Parallel Programming in 100% Java. *Proc. 9th Intl. Workshop on Java and Components for Parallelism, Distribution and Concurrency (IWJacPDC'07)*, Long Beach, CA, USA, p. 231 (8 pages).

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

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

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

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

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

[17] The OpenMPI Project. OpenMPI Shared Memory Communications. `http://www.open-mpi.org/faq/?category=sm` [Last visited: March 2012].

[18] Shafi, A., Carpenter, B., and Baker, M. (2009) Nested Parallelism for Multi-core HPC Systems using Java. *Journal of Parallel and Distributed Computing*, **69 (6)**, 532–545.

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

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

[21] Taboada, G. L., Ramos, S., Touriño, J., and Doallo, R. (2011) Design of Efficient Java Message-Passing Collectives on Multi-core Clusters. *The Journal of Supercomputing*, **55 (2)**, 126–154.

[22] 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: March 2012].

[23] DAS-4. Accelerators and Special Compute Nodes. `http://www.cs.vu.nl/das4/special.shtml` [Last visited: March 2012].

[24] AMD Opteron™ Processor Solutions. `http://products.amd.com/en-us/OpteronCPUDetail.aspx?id=644` [Last visited: March 2012].

[25] 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: March 2012].

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

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

[28] NetPIPE. A Network Protocol Independent Performance Evaluator. `http://www.scl.ameslab.gov/netpipe/` [Last visited: March 2012].

[29] Bailey, D.H. et al (1991) The NAS Parallel Benchmarks Summary and Preliminary Results. *Proc. 1991 ACM/IEEE Intl. Conf. on Supercomputing (SC'91)*, Albuquerque, NM, USA, pp. 158–165.

[30] NAS Parallel Benchmarks. `http://www.nas.nasa.gov/publications/npb.html` [Last visited: March 2012].

[31] Mallón, D. A., Taboada, G. L., Touriño, J., and Doallo, R. (2009) NPB-MPJ: NAS Parallel Benchmarks Implementation for Message-Passing in Java. *Proc. 17th Euromicro Intl. Conf. on Parallel, Distributed*

and Network-Based Processing (PDP'09), Weimar, Germany, pp. 181–190.