Design of Scalable Java Communication Middleware for Multi-Core Systems

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

Computer Architecture Group, Department of Electronics and Systems, University of A Coruña, Campus de Elviña s/n, 15071 A Coruña, Spain
*Corresponding author: sramos@udc.es

This paper presents smdev, a shared memory communication middleware for multi-core systems. smdev provides a simple and powerful messaging application program interface 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 with 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)

Received 21 March 2012; revised 2 August 2012
Handling editor: Mostafa Bassiouni

1. INTRODUCTION

Java is the leading programming language both in academia and the industry environment. Nowadays, owing to the continuous advances in the Java virtual machine (JVM) technology and just-in-time 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 useful 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 application program interface (API) and 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. 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 message passing interface (MPI) [4], whereas in Java it is message-passing in Java (MPJ) [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 us 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 fast MPJ (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.

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. 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 partitioned global address space arise for programming hybrid shared/distributed memory systems, although generally their performance is lower than MPI [10].

Java, owing 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, and 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 sshm, sshm or sshm [15]. It also supports Nemesis [16], a communication middleware that 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 that 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, SysV resources) which requires at least two data transfers, from the source process to the shared memory resource and from this resource to the destination process. Thus, optimizations in these communications subsystems (like Large Message Transfers in Nemesis [15]) aim to optimize data transfers from and to shared memory through fragmentation and rendezvous protocols, but they are not able to eliminate the intermediate copies.

Although there are several message-passing projects in Java [3], 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 that allows us to combine the portability of the ‘pure’ Java communication devices with high-performance network support wrapping native communication libraries through Java native interface (JNI). Additionally, MPJ Express provides shared memory support [19, 20], whereas F-MPJ takes advantage of the middleware presented in this paper, smdev. Its buffer layer avoidance and the reduced synchronization overhead significantly improve the scalability of F-MPJ when communications involve a large number of MPJ processes. In [21], the MPJ Express-shared memory device was found to be the main bottleneck in the development of a hybrid shared/distributed memory communication middleware, with serious drawbacks in scalability due to coarse-grained synchronization and
buffering in the upper layers of the middleware. Furthermore, F-MPJ also outperforms MPJ Express for collective operations as it includes a scalable collectives library tuned for multi-core systems [22] 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] (Fig. 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, which 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 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 Fig. 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 100 MB. 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 when using a single JVM; this is a consequence of the fragmentation and multiple JVMs memory consumption.

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.

#### 3.2.1. Class loading in smdev

The use of threads in smdev for running as message-passing processes requires the isolation of the namespace 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 namespace 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, and so each loader defines its own namespace. 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. 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 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 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 that 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 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 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 are completed 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 first in, first out 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. To cope with duplication 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 into 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 are 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 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 (Fig. 3a), the sender copies the message data in the request buffer and assumes the communication to be 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 to use a rendez-vous protocol (Fig. 3b), where the sender leaves a reference to its own buffer. In this case, the data are copied directly to the receiver buffer when it is available, avoiding the extra copy (zero-copy protocol). However, the sender cannot assume the communication to be completed 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 KB by default. Nevertheless, when the receiver initiates the communication (Fig. 3c), 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 (Fig. 3d), 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 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 shown 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 that are created by sender threads contain the message data.

Regarding the first scenario (Fig. 4a), 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 3). When Thread 1 initiates the reception process (step 4), it finds a matching request in the...
Request $\texttt{isend}(\text{Object } \text{buf}, \text{int } \text{src}, \text{int } \text{tag})$
\begin{verbatim}
    if (!\text{buf}.isObject())
        \text{serialized} = \text{serialize data}(\text{buf});
    \textbf{synchronized}(\text{lock} [\text{dest}])
        
            //lock the receiver's queue
            \text{request} = \text{postedRecvQueue}[\text{dest}],
            \text{find and get message}();
        \text{found} = (\text{request} != \text{null});
        \text{if} (!\text{found})
            \text{request} = \text{new Request}(\ldots);
        \text{if} (!\text{buf}.isObject())
            \text{if} (!\text{eager}(\text{buf}))
                \text{request.buf} = \text{copy}(\text{buf});
                \text{request.setEager}();
            \text{else}
                \text{request.buf} = \text{buf};
                \text{//leave a reference to the message in the request}
        \text{else}
            \text{request.buf} = \text{serialized};
            \text{unexpectedRecvQueue[dest].insert(request)};
    \textbf{if}(\text{found})
        \text{if} (!\text{buf}.isObject())
            \text{request.buf} = \text{copy(buf)};
            \text{//direct copy in the receiver buffer}
        \text{else}
            \text{request.buf} = \text{serialized};
    \textbf{if}(\text{request.eager}() \text{|| found})
        \text{request.setCompleted(true);}
    \text{return req};
\end{verbatim}

\textbf{FIGURE 5.} Pseudo-code of the $\text{isend}$ method.

Request $\texttt{irecv}(\text{Object } \text{buf}, \text{int } \text{src}, \text{int } \text{tag},\text{ Status } \text{atts})$
\begin{verbatim}
    \textbf{synchronized}(\text{lock} [\text{me}])
        
            //lock both of my queues
            \text{request} = \text{unexpectedRecvQueue[me]},
            \text{find and get message}();
        \text{found} = (\text{req} != \text{null});
        \text{if} (!\text{found})
            \text{request} = \text{new Request}(\ldots);
            \text{postedRecvQueue[me].insert(request)};
    \textbf{if}(\text{found})
        \text{if} (!\text{request.serialized}() \&\& !\text{request.completed}())
            \text{buf} = \text{copy(\text{request.buf})};
        \text{else}
            \text{buf} = \text{deserialize(\text{request.buf});}
            \text{request.setCompleted(true);}
    \text{return request;}
\end{verbatim}

\textbf{FIGURE 6.} Pseudo-code of the $\text{irecv}$ method.

UnexpectedRecvQueue (step 5) and copies the message into the destination buffer (step 6).

Regarding the second scenario (Fig. 4b), first Thread 1 initiates the communication with a receive operation (step 1) which does not have a matching request in the UnexpectedRecvQueue (step 2), and 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 is completed by transferring the message data into the destination buffer (steps 6 and 7).

For further details, Figs 5 and 6 illustrate the pseudo-code of the non-blocking send and receive methods ($\text{isend}$ and $\text{irecv}$). Both methods have to check if the message has to be serialized and also search for a matching request in the corresponding queue. If there is no request found, a new pending request is inserted into the other queue; 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 $\text{iwait()}$ over the request finalizes the transfer.

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: one are 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 the other 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. In addition, 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.

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

### 3.3. 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 7 shows the communication support implemented in F-MPJ, 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 (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 the F-MPJ layers owing 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 (Section 3.2.1).

#### 3.3.1. Collective operations in smdev

F-MPJ includes a collects library with multi-core-aware optimized algorithms [22] that runs on top of the device layer, and therefore on smdev. These algorithms include minimum spanning tree (MST), BiDirectional Exchange, Bucket or Cyclic, Binomial Trees and Flat Trees. smdev also provides, through an extension of the xxdev API, its own implementation of collects without relying on point-to-point primitives. The use of these custom algorithms allows us 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 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. Figure 8 presents the broadcast pseudo-code for a more detailed explanation. Similar algorithms have been implemented for the rest of collective operations.

### 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 machine. The first one (‘Xeon E5’) has 2 Intel Xeon E5-2670 eight-core processors at 2.6 GHz [23] and 64 GB 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 9a presents the layout of one of the eight-core Xeon E5 processors, based on the Sandy-Bridge-E architecture, where up to 16 threads can run simultaneously owing 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 9b 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. This non-uniform memory access (NUMA) system supports DDR3-1600 MHz memory.
Design of Scalable Java Communication Middleware

void bcast(Object buf, int root, int tag){
    if(me == root){
        // wait if there is any other collective performing and set it
        busy-wait(collective is performing, busy);
        for(i=0; i<np; i++){
            if(me != i){
                requests[i] = new Request(...);
                if(buf instanceof Object){
                    requests[i].buf = copy serialized data(buf);
                } else{
                    requests[i].buf = buf;
                    // set a reference to the buffer in the request
                    synchronized(lock[me]){ // lock both queues of process
                        unexpectedRecvQueue[me].insert(requests[i]);
                    }
                    coll sem[i].set(written);
                    // wake-up the rest of the processes
                }
            }
        }
        wait(all(requests);
        collective.set(collective set finalized);
        // notify that this collective has finalized
    } else{
        busy-wait(wait until 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);
    }
}

FIGURE 8. The pseudo-code of the bcast method of smdev.

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 [25, 26] and 128 GB 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 10a 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 Level 3 cache) interconnected by Hypertransport (HT) links. Figure 10b 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 two NUMA regions, the quad-socket system has eight NUMA regions.

On the one hand, the Xeon E5 machine 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, which represents one of the highest computational performance per core among the currently available systems. On the other hand, the Magny-Cours machine has one of the highest aggregated CPU computational power in a shared memory system (403.2 GFlops) although its performance per core is quite reduced (8.4 GFlops). Thus, the Xeon E5 system has allowed us to analyze the most recent micro-architecture by Intel, whereas the Magny-Cours system presents the issues associated with the integration of multiple multi-core processors.

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 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. 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. Both libraries have been carefully configured in order to obtain the best performance.

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 [27] and our internal implementation of its Java counterpart.

Figures 11 and 12 show point-to-point performance results obtained on the Xeon E5 and Magny-Cours systems, respectively. The metric shown is half of the round-trip time of a pingpong test for short messages (up to 1 KB), and the bandwidth for messages larger than 1 KB. The transferred data are byte arrays, avoiding the 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 \texttt{smdev} primitives, showing quite similar performance.

To analyze the impact of the memory hierarchy on \texttt{smdev} performance, we have implemented affinity support in Java allowing pinning a thread to a particular core. This support is based on the \texttt{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 11 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 eight cores in each processor only share the L3 cache, the specific core mapping within a processor has no impact on performance.

![FIGURE 9. Intel Xeon E5-2670 system. (a) Intel Xeon E5-2670 processor. (b) Dual-socket Intel Xeon E5-2670 system.](https://example.com/figure9.png)

![FIGURE 10. Magny-Cours AMD Opteron 6172 system. (a) AMD Opteron 6172 processor. (b) Quad-socket Magny-Cours AMD Opteron 6172 system.](https://example.com/figure10.png)

![FIGURE 11. \texttt{smdev} performance on the Xeon E5.](https://example.com/figure11.png)
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 ≥1 KB, achieving the best performance for intra-processor communications, especially when messages are around the L1 (32 KB) or L2 (256 KB) 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 inter-processor large-message transfers in `smdev` benefit more than the intra-processor ones from the L3 cache, since the bandwidth of the former falls from 4 MB on and the latter from 2 MB 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 12 presents pingpong results on the Magny-Cours, communicating either 2 cores within a 6-core die (‘intra-die’ communication), 2 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 2 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.

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 with 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 KB, where `smdev` clearly outperforms MPI owing 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; moreover, the peak of bandwidth is obtained at 256 KB, whereas, in Xeon E5, the peak is at 32 KB, 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 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 with traditional MPI libraries, we have also evaluated it against the shared memory support implemented by MPJ Express, which is the MPJ library with more users and recent maintenance effort. Figures 13 and 14 show results for `smdev` and MPJ Express in the Xeon E5 and the Magny-Cours, respectively, using the best configuration for both scenarios (adjacent cores). In both cases, `smdev` clearly overcomes the performance of MPJ Express in latency and bandwidth. This is due to the avoidance of the double-buffering and the optimization on the synchronizations in `smdev`.

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 [28, 29] on Xeon E5 (Fig. 15) and Magny-Cours (Fig. 16). NetPIPE implementations in Java and C (native) sockets perform a pingpong test similar to the one
The performance of collective operations has a significant impact on the scalability of applications. The aggregated bandwidth for the broadcast, a representative collective of data movement, has been measured on 8 and 16 cores on the Xeon E5 (Fig. 17), and communicating 8 and 48 cores on the Magny-Cours testbed (Fig. 18). Two algorithms have been used for smdev: the broadcast device implementation, presented in Section 3.3.1, and the MST from the F-MPI collectives library [22]. The metric used, the aggregated bandwidth, has been selected as it takes into account the global amount of transferred data.

4.3. Collective operations micro-benchmarking

The performance of collective operations has a significant impact on the scalability of applications. The aggregated bandwidth for the broadcast, a representative collective of data movement, has been measured on 8 and 16 cores on the Xeon E5 (Fig. 17), and communicating 8 and 48 cores on the Magny-Cours testbed (Fig. 18). Two algorithms have been used

FIGURE 14. MPJ performance on the Magny-Cours.

FIGURE 15. Sockets performance on the Xeon E5.

FIGURE 16. Sockets performance on the Magny-Cours.

FIGURE 17. Broadcast performance on 8 and 16 cores (Xeon E5).

FIGURE 18. Broadcast performance (16 Cores, Xeon E5).
The results from Figs 17 and 18 show that the F-MPJ broadcasts generally obtain higher bandwidth than the MPI implementations owing to relying on a zero-copy communication protocol, as it happened in 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 decrease in performance. In fact, the broadcast of messages up to 1 MB 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 KB 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 the message size and number of cores. Like 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 MB). Owing to the lower computational power and memory performance of the Magny-Cours, the achieved bandwidth is significantly lower than on 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 NASA Advanced Supercomputing (NAS) Parallel Benchmarks (NPB) [30, 31], 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) [32] 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 with sockets, the default communication solution in Java. Four NPB kernels have been selected: conjugate gradient (CG), Fourier transform (FT), integer sort (IS) and multi-grid (MG), 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 Figs 19 and 20. With the aim of providing a reference of absolute performance, Table 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 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 versions.

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 worst 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 thread programming while taking advantage of the inherent parallelism of multi-core processors introducing performance improvements over the current alternatives in MPI and MPJ. This middleware has been successfully integrated in our Java message-passing implementation, F-MPJ. Thus, any MPJ application running on distributed memory systems can...
FIGURE 19. NPB performance (Xeon E5).

FIGURE 20. NAS parallel benchmarks performance (Magny-Cours).
also run efficiently on shared memory systems owing to relying on \texttt{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\,\mu s, 13 times higher performance than Java sockets latency on shared memory (10\,\mu s); (2) point-to-point bandwidths higher than 90\,Gbps, around six times better performance than Java sockets bandwidth on shared memory (around 15\,Gbps); (3) point-to-point performance results only around 1\,\mu s worse than MPI for small messages, but significantly better for medium and large messages (from 2\,KB on); (4) \texttt{smdev} presents higher performance and scalability for collective operations than MPI; (5) the use of \texttt{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 codes; (6) \texttt{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 shows scaling performance on the 48-core AMD-based system. Therefore, \texttt{smdev} is the key for high-performance Java applications on shared memory multi-core systems.

\section*{ACKNOWLEDGEMENTS}

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

\section*{FUNDING}

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

\section*{REFERENCES}


