Computer Communications 31 (2008) 4049–4059

Contents lists available at ScienceDirect

Computer Communications

journal homepage: www.elsevier.com/locate/comcom

Java Fast Sockets: Enabling high-speed Java communications on high performance clusters

Guillermo L. Taboada *, Juan Touriño, Ramón Doallo

Computer Architecture Group, Department of Electronics and Systems, University of A Coruña, 15071 A Coruña, Spain

ARTICLE INFO

Article history:
Received 29 November 2007
Received in revised form 12 August 2008
Accepted 13 August 2008
Available online 28 August 2008

Keywords:
Java sockets
High performance cluster
Scalable coherent interface
Myrinet
Gigabit Ethernet

ABSTRACT

This paper presents Java Fast Sockets (JFS), an optimized Java socket implementation on clusters for high performance computing. Current socket libraries do not efficiently support high-speed cluster interconnects and impose substantial communication overhead. JFS overcomes these performance constraints by: (1) enabling high-speed communication on cluster networks such as Scalable Coherent Interface (SCI), Myrinet and Gigabit Ethernet; (2) avoiding the need of primitive data type array serialization; (3) reducing buffering and unnecessary copies; and (4) reimplementing the protocol for boosting shared memory (intra-node) communication. Its interoperability and user and application transparency allow for immediate applicability on a wide range of parallel and distributed target applications. A performance evaluation conducted on a dual-core cluster has shown experimental evidence of throughput increase on SCI, Myrinet, Gigabit Ethernet and shared memory communication. It has also been analyzed the impact of this improvement on the overall application performance of representative parallel codes.

1. Introduction

Several appealing features have made Java particularly attractive for many computing environments, becoming a widespread option. The benefits are many: platform independence, portability, higher programming productivity typical of object-oriented languages, widely spread knowledge and better integration into existing applications. Nevertheless, there are environments where more traditional languages have predominance and Java is still an emerging option, usually where performance is a critical issue. Although continuous advances in JIT (Just In Time) compilers, Java Virtual Machines (JVMs) and runtime library optimizations have brought Java performance close to natively compiled languages (C/C++/Fortran), this is usually restricted to sequential applications. Parallel and distributed Java applications usually suffer from inefficient communication middleware, most of them based on protocols with high communication overhead such as Java Remote Method Invocation (RMI). The emergence of multicore architectures heightens the need of languages with out-of-the-box multithreading and concurrency support, like Java. Furthermore, efficient communication middleware is also needed. Regarding current computing platforms, clusters, especially with high-speed networks, are the choice of both industry and academia as they deliver outstanding throughput at a reasonable price/performance ratio. In this context, the trend is to move to multicore clusters with high-speed interconnects. The adoption of Java as a mainstream language on these systems depends on the availability of efficient communication middleware in order to benefit from its appealing features at a reasonable overhead.

Our goal is to provide parallel and distributed Java applications with an efficient socket implementation, Java Fast Sockets (JFS), for high performance computing on clusters with high-speed networks. Several projects have previously attempted to increase Java communication performance, especially on high-speed cluster networks, but they lack desirable features as will be discussed in Section 2. JFS optimizes the JVM socket protocol reducing communication overhead, especially for shared memory transfers, as will be presented in Section 3. It also provides high-speed cluster interconnects with efficient Java communication support and allows for immediate throughput increase thanks to its user and application transparency as will be shown in Section 4. Section 5 presents the performance evaluation conducted on a dual-core cluster with SCI, Myrinet and Gigabit Ethernet where JFS has shown significant throughput improvement. Its impact on the overall application performance has also been analyzed on representative parallel applications, as will be discussed in Section 6. The paper concludes in Section 7 with a summary of the main results and contributions.

2. Related work

Efficient communication middleware is key to deliver scalable application performance. Although most clusters have high-speed

* Corresponding author. Tel.: +34 981167000; fax: +34 981167160.
E-mail addresses: taboada@udc.es (G.L. Taboada), juan@udc.es (J. Touriño), doallo@udc.es (R. Doallo).

0140-3664/$ - see front matter © 2008 Elsevier B.V. All rights reserved.
doi:10.1016/j.comcom.2008.08.012

networks to boost communication performance, Java cannot take advantage of them as shown in [1] because it has to resort to inefficient TCP/IP emulations for full networking support. These emulation libraries present high start-up latency (the 0-byte message latency), low bandwidth and high CPU load as shown in [2]. The main reason behind this poor throughput is that the IP protocol was designed to cope with low speed, unreliable and prone to failure links in WAN environments, whereas current cluster networks are high-speed, hardware reliable and non-prone to failure in LAN environments. Examples of IP emulations are IPoMX and IPoGM [3] on top of the Myrinet low-level libraries MX (Myrinet eXpress) and GM, LANE driver [4] over Giganet, IP over Infiniband (IPoIB) [5], and ScaIP [6] and SCIP [7] on SCI.

A direct implementation of native sockets on top of low-level communication libraries can avoid the TCP/IP overhead, and thus increases performance. Representative examples are next presented. FastSockets [8] is a socket implementation on top of Active-Messages, a light-weight protocol with high-speed network access. SOVIA [4] has been implemented on VIA (Virtual Interface Architecture); and Sockets over Gigabit Ethernet [9] and GAMMAsockets [10] have been developed for Gigabit Ethernet. The Socket Direct Protocol (SDP) over Infiniband [11] is the representative socket library of the Offload Sockets Framework (OSF). Sockets-MX and Sockets-GM [3] are the developments on Myrinet, where MX is intended to supersede GM thanks to a more efficient protocol implementation. The high performance native sockets library on SCI is SCI Sockets [12]. However, from these implementations only SDP, Sockets-MX/GM and SCI Sockets are currently available. The Windows Sockets Direct components for Windows platforms provide access to certain high-speed networks. A related project is XenSocket [13], an optimized socket library restricted to Xen virtual machine intra-node communication that replaces TCP/IP by shared memory transfers.

However, the previous socket libraries usually implement a subset of socket functionality on top of low-level libraries, resorting to the system socket library for unimplemented functions. Thus, some applications such as kernel-level network services and Java codes can request features not present in the underlying libraries and thus failover to system sockets. In order to provide Java with full and more efficient support on high-speed networks several approaches have been followed: (1) VIA-based projects, (2) RMI optimizations, (3) Java Distributed Shared Memory (DSM) middleware on clusters and (4) low-level libraries on high-speed networks.

Javia [14] and Jaguar [15] provide access to high-speed cluster interconnects through VIA, communication library implemented on Giganet, Myrinet, Gigabit Ethernet and SCI [16], among others. More specifically Javia reduces data copying using native buffers, and Jaguar acts as a replacement of the Java Native Interface (JNI). Their main drawbacks are the use of particular APIs, the need of modified Java compilers and the lack of non-VIA communication support. Additionally Javia exposes programmers to buffer management and uses a specific garbage collector.

Representative works about RMI optimization are Manta [17], a Java to native code compiler with a fast RMI protocol, and KaRMI [18], that improves RMI through a more efficient object serialization that reduces protocol latency. Serialization is the process of transforming objects in byte series, in this case to be sent across the network. However, the use of specific high-level solutions with substantial protocol overhead and focused on Myrinet has restricted the applicability of these projects. In fact, their start-up latency is from several times up to an order of magnitude larger than socket latencies. Therefore, current Java communication middleware such as MPJ Express [19] and MPJ/Ibis [20], two Message-Passing in Java (MPJ) libraries use sockets (Java NIO and Ibis sockets, respectively) instead of RMI, due to their lower overhead. In this case, the higher programming effort required by the lower-level API allows for higher throughput, key in communication middleware for high performance computing.

Java DSM projects worth mentioning are CoJVM [21], JESSICA2 [22] and JavaSplit [23]. As these are socket-based projects, they benefit from socket optimizations, especially in shared memory communication [24]. However, they share unsuitable characteristics such as the use of modified JVMs, the need of source code modification and limited interoperability.

Other approaches are low-level Java libraries restricted to specific networks. For instance, Jdib [25] accesses Mellanox Verbs Interface (VAPI) on Infiniband through a low-level API which directly exploits RDMA and communication queues. Thus, this library achieves almost native performance on Infiniband.

This paper presents JFS, an efficient Java socket library for multicore clusters with high-speed networks. By optimizing the widely used socket API, parallel and distributed Java applications based on it improve performance transparently. A previous project NBIO [26] has led to introduce significant non-blocking features in Java NIO sockets which are key to increase scalability in server applications. Nevertheless, NBIO does not provide high-speed network support nor high performance computing tailoring. Ibis sockets partly solve these issues adding Myrinet support and being the base of the Ibis framework [27], a pure Java (without native code) optimized grid and cluster middleware. However, their implementation on top of JVM sockets limits their performance increase to serialization improvements.

A preliminary Java socket implementation for SCI and Gigabit Ethernet has been presented in previous works [28] [29], but current JFS implementation shows major improvements: (1) adds Myrinet support, (2) optimizes Java I/O sockets instead of NIO sockets in order to extend its applicability, (3) avoids the need of primitive data type array serialization, (4) reduces even more buffering and unnecessary copies and (5) adds an optimized shared memory protocol. The performance evaluation presented in Sections 5 and 6 shows the significant throughput improvement obtained by the current JFS implementation on SCI, Myrinet, Gigabit Ethernet and especially on shared memory communication, and on the overall application performance of representative parallel codes.

3. Efficient Java socket implementation

The development of an optimized Java socket library poses several challenges such as serialization overhead reduction and protocol performance increase, especially through a more efficient data transfer implementation. JFS has contributed to these goals by: (1) avoiding primitive data type array serialization (see Section 3.1); (2) reducing buffering and unnecessary copies in the protocol (see Section 3.2); and (3) providing shared memory communication with an optimized transport protocol as will be shown in Section 3.3.

3.1. Serialization overhead reduction

Serialization imposes severe performance penalties as this process involves the extraction of the byte values from the data to be sent. An example of this is shown in Listing 1, where `java.io.Bits.putInt()` writes an int `val` to the stream `b` at the position `off`. As Java socket restriction of sending only byte arrays does not hold for native sockets, JFS defines native methods (see Listing 2) to transfer primitive data type arrays directly without serialization.

Listing 1. Example of a costly serialization operation of an int value

```
static void putInt(byte[] b, int off, int val) {
  b[off+3]=(byte) (val ≫ 0);
  b[off+2]=(byte) (val ≫ 8);
  b[off+1]=(byte) (val ≫ 16);
  b[off+0]=(byte) (val ≫ 24);
}
```

Listing 2. JFS extended API for direct communication of primitive data type arrays

```
jfs.net.SocketOutputStream.write(int buf[],
int offset, int length);
jfs.net.SocketOutputStream.write(double  buf[],
int offset, int length);
jfs.net.SocketOutputStream.write(float   buf[],
int offset, int length);
...
jfs.net.SocketInputStream.read(int buf[],
int offset, int length);
...
```

3.2. Socket protocol optimization

Sun JVM socket operation has been analyzed. Fig. 1 shows its diagram representing the data structures used and the path followed by socket messages. It has been selected a primitive data type array transfer for representativeness and illustrative purposes. First, `ObjectOutputStream`, the class used to serialize objects, writes `sdata` to a block data buffer (`blockdata`). As recommended, serialized data is buffered in order to reduce the number of accesses to native sockets. Then, the socket library uses the JNI function `GetByteArrayRegion(byte[] buf)` to copy the buffered data to `jvmsock_buf`, a native buffer that is dynamically allocated for messages longer than 2 KB (configurable size). The native socket library and its buffer `nativesock_buf` are involved in the next copy. Then, data are transferred through the network thanks to the network driver. The receiving side operates in reverse order, and thus the whole process involves nine steps: a serialization, three copies, a network transfer, other three copies and a deserialization. Potential optimizations detected in this analysis in order to improve performance are the reduction in the number of copies and the decrease of the serialization overhead.

These optimizations have been included in JFS as shown in Fig. 2. The function `GetPrimitiveArrayCritical(⟨primitive data type⟩ {s/r} data[])` allows native code to obtain through JNI a direct pointer to the Java array in order to avoid serialization. Thus, a one-copy protocol can be implemented as only one copy is needed to transfer `sdata` to the native socket library. However, data can be transferred with a zero-copy protocol without involving the CPU on RDMA-capable high-speed cluster interconnects (such as SCI, Myrinet and Infiniband). This zero-copy protocol obtains higher bandwidths and lower CPU loads than the one-copy protocol, although RDMA imposes a higher start-up latency. Therefore, one copy is used only for short messages (size below a configurable threshold). A related issue is the receiving strategy, obtaining polling lower start-up latency but higher CPU load than blocking. Thus, polling is preferred only for short messages. These protocols and strategies are handled by JFS. The whole optimized process involves up to two copies and a network communication in the worst case. Furthermore, it has been detected a potential optimization for shared memory communication, presented in the following subsection.

3.3. Efficient shared memory socket communication

The emergence of multicore architectures has increased the use of shared memory socket communication, the most efficient way to exchange messages between two Java applications running on the same machine. However, JVM sockets handle intra-node transfers as TCP/IP transmissions. Some optimizations exist, like using a larger Maximum Transfer Unit (MTU) size, usually an order of magnitude higher, in order to reduce IP packet fragmentation, but TCP/IP overhead is still the throughput bottleneck. In order to reduce this performance penalty JFS has implemented shared memory transfers resorting to UNIX sockets (or similar lightweight non-TCP/IP sockets when available) and direct memory transfers, and therefore avoiding TCP/IP (see Fig. 2). Thus, JFS first sends the `sdata` direct pointer (`arr_sref`) to the receiver, which next moves `sdata` content into `rdata` array through a native copy (`memcpy` or analogous). Finally, the sender polls for the copy end notification, a control message or a flag setting by the receiver. JFS greatly benefits from this optimization achieving memory-to-memory bandwidth, although for short messages the start-up latency of this three-step protocol can be enhanced by sending the data in only one transaction. This efficient shared memory support, together with optimized inter-node transfers, allows socket-based parallel applications to achieve good performance on multicore clusters. This is due to the combination of the scalability provided by the distributed memory paradigm and the high performance of the shared memory communication.

4. High performance Java communication on clusters

JFS provides efficient socket communication through an optimized protocol. However, the usefulness of these improvements depends on the range of potential target systems and applications. Thus, in order to extend this range, JFS adds efficient support for high-speed cluster interconnects (next presented in Section 4.1). JFS also provides application transparency, in order to be used by Java applications without source code modification, as will be shown in Section 4.2.

4.1. Efficient Java communication on high-speed cluster interconnects

JFS includes a more efficient high-speed cluster network support than the use of IP emulations. Thus, JFS relies on native socket operation that does not experience problems with the JVM. An example is the avoidance of IPv6, preferred by JVM sockets and usually not implemented for high-speed networks. This high-speed interconnect support is implemented specifically for each network through JNI, which provides native socket throughput to Java. JNI is also used by JVM sockets, although their generic access to the network layer is inefficient for high-speed networks as they do not take advantage of the underlying native libraries.

Fig. 3 shows a schema of the components involved in socket operation on high-speed networks. From bottom to top, the first layer is the Network Interface Card (NIC) for each high-speed network, then appears the native middleware (two layers), next the Java middleware (two layers), and finally the applications. Java

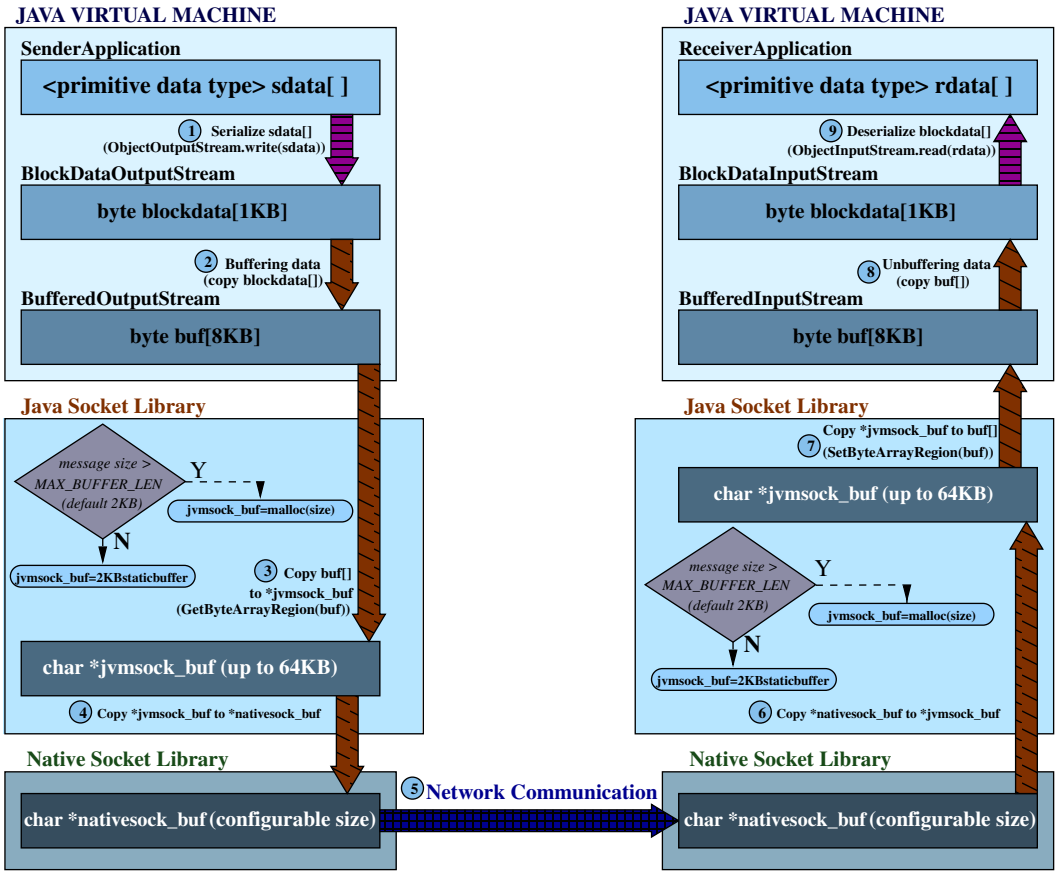

Fig. 1. Sun JVM socket operation.

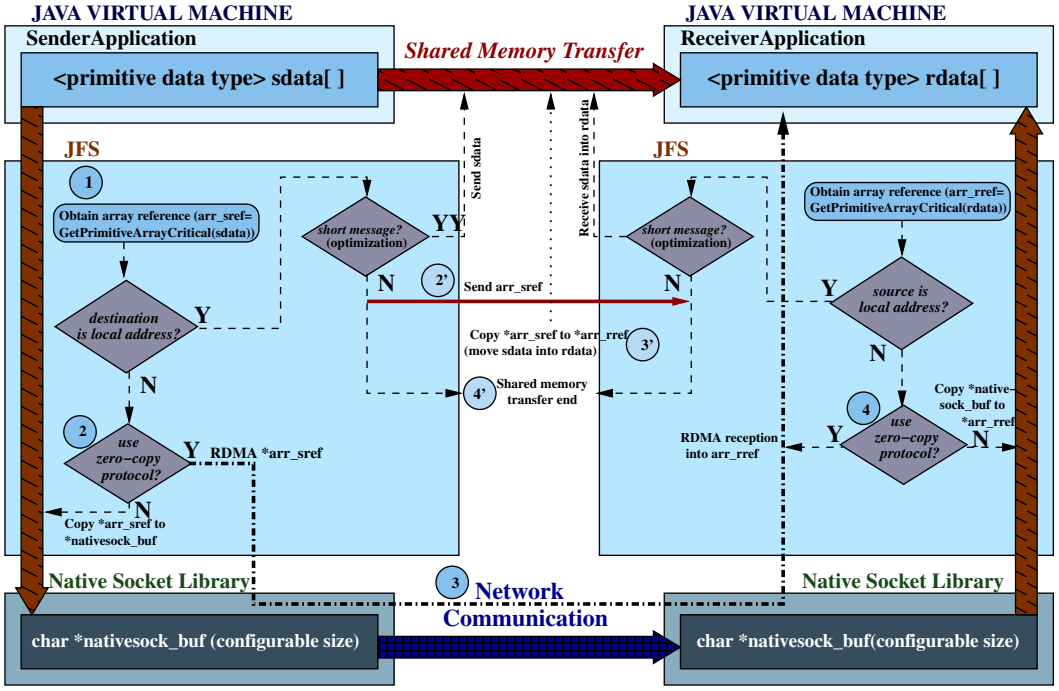

Fig. 2. JFS optimized protocol.

applications access Java sockets usually through Java communication middleware, e.g., MPJ libraries, typically based either on RMI or directly on sockets. Regarding Gigabit Ethernet, the Java support is direct on native sockets. The SCI low-level drivers are IRM (Interconnect Resource Manager) and SISCI (Software Infrastructure for SCI), whereas SCILib is a communication protocol on top of SISCI that offers unidirectional message queues. On SCI JFS resorts to SCI Sockets and SCILib, higher level solutions than IRM and SISCI

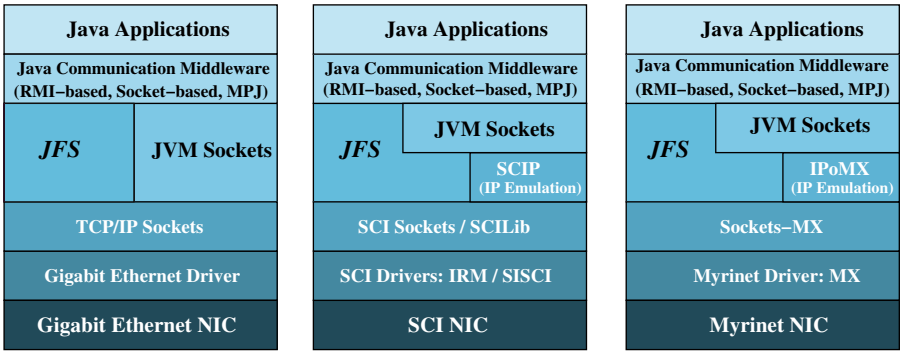

Fig. 3. Java sockets on high-speed networks: communication middleware overview.

but still efficient libraries. On Myrinet JFS relies on Sockets-MX for providing Java applications with efficient communication. JFS also provides JVM sockets with high-speed network support in order to avoid IP emulations. Furthermore, JFS aims to transparently obtain the highest performance on systems with several communication channels through a failover approach. Thus, JFS first tries to use the option with the highest performance. If this fails, it follows, in descending order of performance, with the remaining communication channels that are available.

4.2. JFS application transparency

By implementing the socket API, a wide range of parallel and distributed target applications can take advantage transparently of the efficient JFS communication protocol. As Java has a built-in procedure (setting factories) to swap the default socket library, it is easy to replace the JVM sockets by JFS. However, the JVM socket design has to be followed in order to implement a swappable socket library. Fig. 4 presents JFS core classes: `PlainSocketImpl` is the Sun JVM socket implementation, `FastSocketImplFactory` creates custom JFS sockets, and the I/O stream classes, whose package is `java.net` for Sun JVM sockets and `jfs.net` for JFS. The stream classes are in charge of managing the transport protocol. The JFS setting as the default socket library is shown in Listing 3. From then on the application will use this implementation. As this procedure requires source code modification, Java's reflection has been used in order to obtain a transparent solution. Thus, a small application *launcher* swaps its default socket factory and then invokes the `main` method of the target class (see Listing 4). The target application will use JFS transparently even without source code availability.

Listing 3. Swapping Java socket implementation

```
SocketImplFactory factory =
new jfs.net.FastSocketImplFactory();
Socket.setSocketImplFactory(factory);
ServerSocket.setSocketFactory(factory);
```

Listing 4. JFS *launcher* application code

```
[Swap Java socket implementation]
Class cl=Class.forName(className);
Method method=cl.getMethod("main",
parameterTypes);
method.invoke(null, parameters);
```

JFS extends the socket API by adding methods that avoid serialization and eliminate unnecessary copies when sending portions of primitive data type arrays. Listing 5 presents an example of this feature. As JVM sockets can not send array portions (except for parts of byte arrays) a new array must be created to store the data to be serialized and then sent. This costly process is repeated at the receiver side. Listing 5 shows the handling of this communication scenario in a portable way in order to use the efficient JFS methods when they are available. This feature is of special interest in communication middleware such as Java message-passing libraries and RMI, yielding significant benefits to end applications without modifying their source code.

Parallel and distributed Java applications and, especially, communication middleware can benefit transparently from the higher performance of JFS on high-speed networks and shared memory communication. This can be done without losing portability, using particular JFS features such as serialization avoidance only when this socket library is available. Moreover, this solution is interoperable as it can communicate with JVM sockets, although relying only on features shared by both implementations. Thus, the buffering and copying reduction could be used, but not the high-speed network support nor the optimized shared memory transfers. Next sections evaluate JFS performance (Section 5) and its impact on representative applications (Section 6).

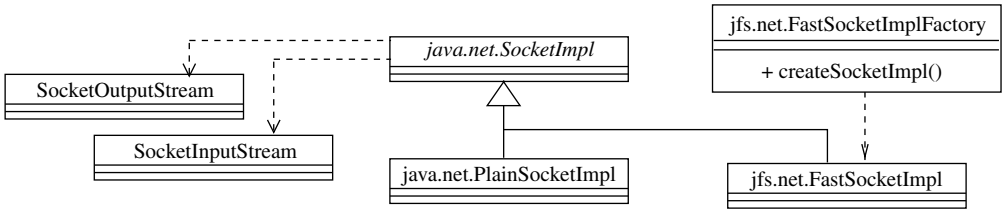

Fig. 4. JFS core class diagram.

Listing 5. JFS direct send of part of an int array

```
if (os instanceof jfs.net.SocketOutputStream) {
  jfsExtendedAPI = true;
  jfsos = (jfs.net.SocketOutputStream) os;
}
oos = new ObjectOutputStream(os);
int int_array[] = new int[20];
[...]
// Writing the first ten elements of int_array
if (jfsExtendedAPI)
  jfsos.write(int_array,0,10);
else {
  int[]     ints     =     (int[])Array.newIn-
stance(int.class, 10);
  System.arraycopy(int_array, 0, ints, 0, 10);
  oos.writeUnshared(ints);
}
```

5. JFS Performance evaluation

5.1. Experimental configuration

The testbed consists of a cluster of eight dual-core nodes (Pentium IV Xeon at 3.2 GHz, 4 GB of memory) interconnected via SCI, Myrinet and Gigabit Ethernet. The SCI NIC is a D334 card and the Myrinet NIC is an "F" Myrinet2000 card (M3F-PCIXF-2 Myrinet-Fiber/PCI-X NIC). Both are plugged into 64 bits / 66 MHz PCI slots. The Gigabit Ethernet NIC is an Intel PRO/1000 using a 3Com 2816-SFP Plus switch. It has also been used a DGS-1216T Dlink switch for evaluating Gigabit Ethernet Jumbo Frames performance. The OS is Linux CentOS 4.4 with C compiler gcc 3.4.6 and Sun JDK 1.5.0_07. This JDK has been selected as it obtained slightly better performance than IBM JDK 1.5 for the benchmarks used in Sections 5 and 6. The SCI libraries are SCI Sockets 3.1.4, DIS 3.1.11 (IRM, SISCI and SCILib) and SCIP 1.2.0, whereas the Myrinet libraries are MX 1.1.1 and Sockets-MX 1.1.0 (see Fig. 3).

In order to microbenchmark JFS performance, a Java socket version of NetPIPE [30] has been developed. The results considered in this section are the half of the round trip time of a ping-pong test running JIT compiled bytecode. In order to obtain optimized JIT results, 10,000 warm-up iterations have to be executed before the actual measurements. It has been benchmarked the performance of byte, integer and double arrays, as they are data structures frequently used in parallel and distributed applications. For clarity purposes it has been used the JNI array notation. Thus, `B]` denotes a byte array, `I]` an int array and `D]` a double array. When using serialization, it has been pointed out the procedure through the use of the keys `OOS` and `OBOS`. `OOS` indicates a `java.net.ObjectOutputStream` object wrapping a `SocketOutputStream` object, whereas `OBOS` is a `java.net.ObjectOutputStream` object wrapping a `BufferedOutputStream` around the supplied `SocketOutputStream`. `OOS` writes directly to the stream the serialized byte series in order to reduce the start-up latency, whereas `OBOS` buffers the serialized data in a byte array (by default an 8 KB buffer) in order to minimize the stream accesses and thus increase bandwidth. As `OOS` improves performance only for short messages, and slightly as can be seen in Figs. 9 and 10, its results have been omitted in the remaining figures for clarity purposes.

5.2. JFS Microbenchmarking on high-speed networks

Figs. 5–10 show the latencies and bandwidths of native and Java socket libraries as a function of the message length, for byte, integer and double arrays on SCI, Myrinet and Gigabit Ethernet. The native libraries considered are SCI Sockets, Sockets-MX and the native TCP/IP sockets, whereas the Java sockets libraries are Sun JVM sockets and JFS. The latency graphs (at the top) serve to compare short message performance, whereas the bandwidth graphs (bottom) are useful to compare long message performance.

Figs. 5 and 6 present latency and bandwidth results on SCI. The two available transport layers with Java support, the IP emulation SCIP and JFS, obtain significantly different results. Thus, JFS start-up latency is 6 μs compared to 36–48 μs for SCIP, showing an overhead reduction of up to 88%. Regarding bandwidth, JFS achieves up to 2366 Mbps whereas SCIP results are below 450 Mbps, up to 1305% performance increase for JFS (`B] JFS` vs. `OBOS(D])` for a 2 MB message). `B]`, `I]` and `D]` JFS results are quite similar among them as they use the same protocol, a direct send avoiding serialization. Thus, for clarity purposes, only `B]` values are presented as the representative results under the label `B],I],D] JFS`. As JFS is implemented on top of SCI Sockets (see Fig. 3) it can be estimated its processing overhead (the difference between JFS and SCI Sockets performance) in around 1-2 μs for short messages, and around a 5% bandwidth penalty for long messages. Therefore, JFS obtains quite similar results to SCI Sockets, the high performance native

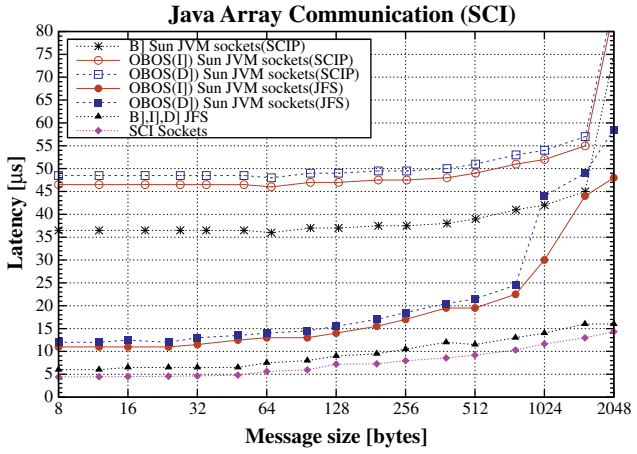

Fig. 5. SCI latency.

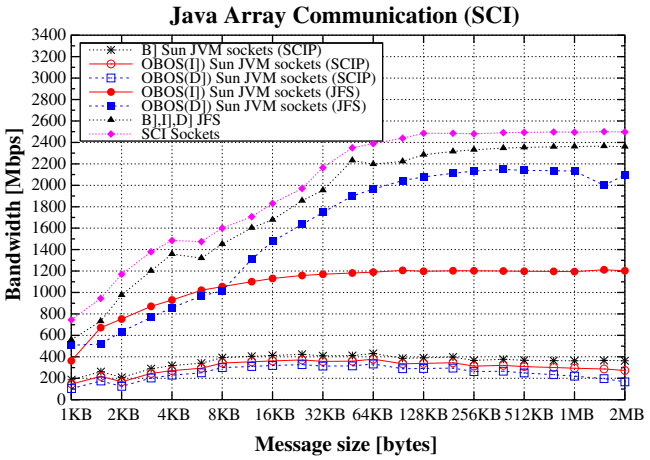

Fig. 6. SCI bandwidth.

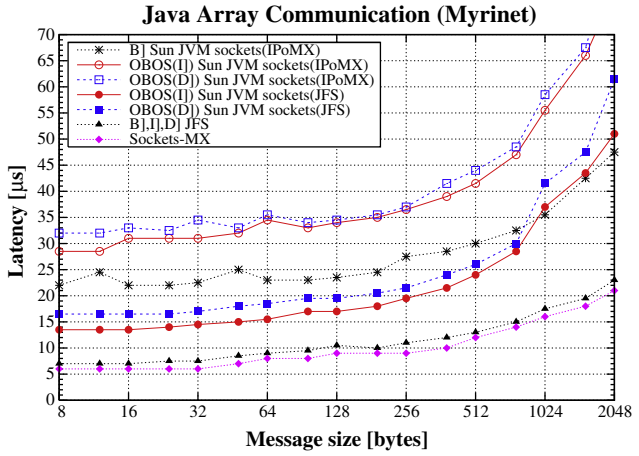

Fig. 7. Myrinet latency.

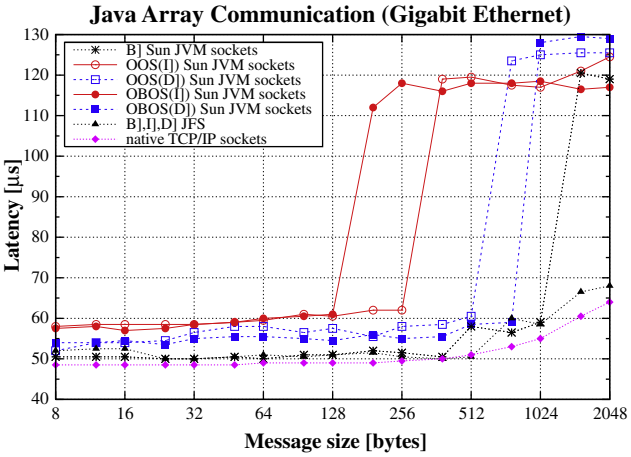

Fig. 9. Gigabit Ethernet latency.

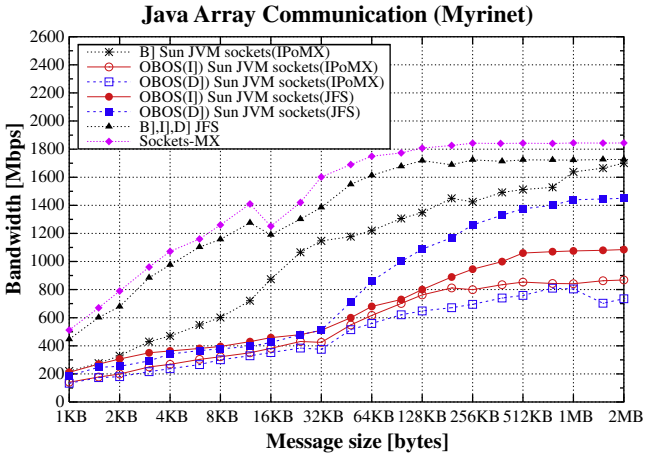

Fig. 8. Myrinet bandwidth.

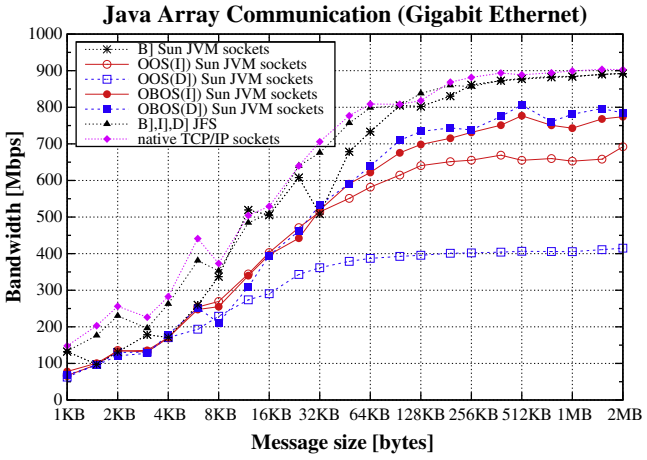

Fig. 10. Gigabit Ethernet bandwidth.

socket library on SCI. `OBOS` serialization imposes overheads on start-up latencies around 6 µs and 12 µs using JFS and SCIP, respectively. `OBOS` over SCIP bandwidths are quite poor, under 400 Mbps. Regarding `OBOS` over JFS results, `OBOS(D])` bandwidth is close to `JFS` (around 90%) thanks to its optimized native implementation. Sun JVM provides optimized native methods for float and double array serialization, overcoming the pure Java serialization bottleneck at 1200 Mbps for `OBOS(I])` over JFS. However, for short messages `OBOS(I])` obtains better performance than `OBOS(D])`.

Figs. 7 and 8 present latency and bandwidth results on Myrinet. The best Java sockets results have been obtained using JFS as transport layer, although using the IP emulation IPoMX the differences narrow as the message size increases, showing similar long message bandwidth for byte arrays. The reason of this behavior is the higher start-up latency of IPoMX (22 µs) compared to JFS (7 µs, 68% less than IPoMX), and that the Myrinet NIC is the communication bottleneck limiting the maximum transfer rate to 2 Gbps. In fact, the experimentally measured JFS and IPoMX bandwidths can only rise up to 85% of this value (1700 Mbps). JFS Myrinet support is based on Sockets-MX rather than Sockets-GM for its better performance. This has been experimentally assessed on our testbed, where JFS resorting to Sockets-GM obtained higher start-up latency (23 µs) and lower bandwidths than using Sockets-MX. The presented Sockets-MX results show that JFS overhead on Myrinet is quite reduced, obtaining JFS almost native performance. `OBOS` serialization imposes an overhead on start-up latency of around 7–10 µs. Regarding bandwidth, `OBOS` over JFS performs better than using IPoMX. The native serialization method in `OBOS(D])` improves the performance of the pure serialization method used in `OBOS(I])` only over JFS, but not over IPoMX. However, `B],I],D] JFS` clearly outperforms `OBOS` results with a performance increase of up to 412%.

Figs. 9 and 10 present latency and bandwidth results on Gigabit Ethernet. There is not a significant difference in byte array performance between socket implementations, although JFS slightly outperforms Sun JVM sockets for medium-sized messages. However, JFS performance improvement of sending int and double arrays (`I]` and `D]`) is up to 119%, result obtained for a 2 MB message, thanks to avoid serialization. It can be seen that serialization imposes an overhead of 4–7 µs in start-up latency and that the native serialization used in `OBOS(D])` does not increase performance significantly. In fact, the data link and network layers are the performance bottlenecks as they impose high start-up latencies, around 50 µs, and low bandwidths, below the 1 Gbps maximum network transfer rate, severely limiting throughput improvement. This analysis is confirmed by the presented native TCP/IP sockets results, which show almost the same performance as Java sockets due to the high impact of the communication bottlenecks on the overall performance. Moreover, socket latencies are clustered around 55 and 120 µs (see Fig. 9) caused by the operation of the underlying layers. The effect of these clustered latencies can also be observed for JFS in Fig. 10 where the bandwidth for [1–16 KB] messages presents a saw-teeth shape. Looking for potential

improvements in order to partly overcome these limitations it has been evaluated the use of Gigabit Ethernet Jumbo Frames.

5.3. JFS on Gigabit Ethernet Jumbo Frames

The Ethernet default MTU of 1500 bytes has been maintained for backward compatibility in order to handle any communication between 10/100/1000 Mbps devices without any Ethernet frame fragmentation or reassembly. Nevertheless, this is a rather small size that increases CPU load due to handling numerous frames when sending long messages. A larger MTU reduces CPU overhead and therefore increases long message bandwidth, although for medium-sized messages waiting for filling larger Ethernet frames increases latency. Jumbo Frames is the technology that extends MTU size up to 9000 bytes.

It has been evaluated the use of JFS with Jumbo Frames for MTU sizes of 3000, 4500, 6000 and 9000 bytes. Jumbo Frames increase slightly JFS long message performance. For a 2 MB message the bandwidth rises from 892 Mbps, with the default MTU, up to 932 Mbps using an MTU of 9000 bytes. This improved result is 93% of the maximum theoretical bandwidth, 4% more than using the default MTU. Regarding medium-sized messages, the use of Jumbo Frames increases JFS latency in the range [1.5–256 KB] up to 90% (this peak latency increase was obtained for a 6 KB message with an MTU of 9000 bytes). This latency increase is especially high for [1.5–16 KB] messages, while for larger messages the negative impact of Jumbo Frames is reduced as the message size increases.

An additional characteristic of the use of Jumbo Frames is the CPU communication processing offloading. Table 1 presents the CPU overhead of two Java socket implementations in terms of percentage of CPU load (using a Xeon 3.2 GHz) devoted to socket communication processing. It has been used the NetPIPE benchmark sending from 9 KB up to 2 MB messages (range with Ethernet frame fragmentation) for measuring these values. It can be seen that Jumbo Frames reduce significantly CPU overhead. Nevertheless, as Jumbo Frames trade off medium-sized message performance for CPU offloading, this is not an especially useful feature.

As main conclusion, the use of Jumbo Frames is recommended for applications sending only long messages. Regarding the CPU offloading, Jumbo Frames contribution is not especially important as JFS already reduces CPU load without using Jumbo Frames (MTU = 1500), from 30% to 12% (60% reduction) as can be seen in Table 1, and therefore without trading off performance for CPU offloading.

5.4. Java shared memory communication

Fig. 11 presents JFS shared memory protocol performance for short messages (see Section 3.3). Although the default underlying library for this protocol is UNIX sockets, JFS performance using TCP sockets is shown for comparison purposes. JFS start-up latency is 8 μs, half of Sun JVM sockets start-up. However, this value is larger than the 6 and 7 μs JFS start-up latencies on SCI and Myrinet, respectively (see Section 5.2), as the underlying native library, UNIX sockets, imposes higher start-up overhead than the native sockets on these high-speed networks. In a multicore scenario it is key to reduce the high start-up latency of shared memory native communication.

Table 1
CPU load percentage of sockets processing using Gigabit Ethernet Jumbo Frames

		MTU (bytes)				
		1500	3000	4500	6000	9000
Socket implementation	Sun JVM	30%	25%	15%	16%	5%
	JFS	12%	10%	4%	4%	3%

Fig. 12 shows the significant bandwidth increase of JFS communication due to the use of the optimized shared memory protocol for messages longer than 16 KB. This protocol increases the peak bandwidth from 9 Gbps, using JFS without this optimized protocol, up to 34 and 41 Gbps for JFS using TCP and UNIX sockets, respectively. These peak bandwidths are obtained for 256–512 KB message sizes as memory-to-memory transfers also obtain their peak bandwidths for this range. The performance of the optimized shared memory protocol has also been measured for the native UNIX sockets library, showing that JFS also obtains almost native performance on shared memory. Sun JVM socket performance is under 1.5 Gbps for int and double arrays and under 6 Gbps for byte arrays. The observed bandwidth increase is up to 4411%, peak value obtained by comparing a 512 KB `D]` message sent with `JFS (UNIX sockets)` versus sent with `OBOS(D])`.

6. Performance analysis of parallel applications with JFS

JFS microbenchmarking has shown significant performance improvement, but its usefulness depends on its impact on the overall application performance. The range of JFS applicability covers socket-based MPJ applications and MPJ libraries such as MPJ Express [19] and MPJ/Ibis [20], RMI applications and RMI-based middleware like ProActive [31] [32], a middleware for parallel, multithreaded and distributed computing focused on Grid applications. In short, any sockets-based parallel or distributed Java application running on a cluster can use JFS. These applications can

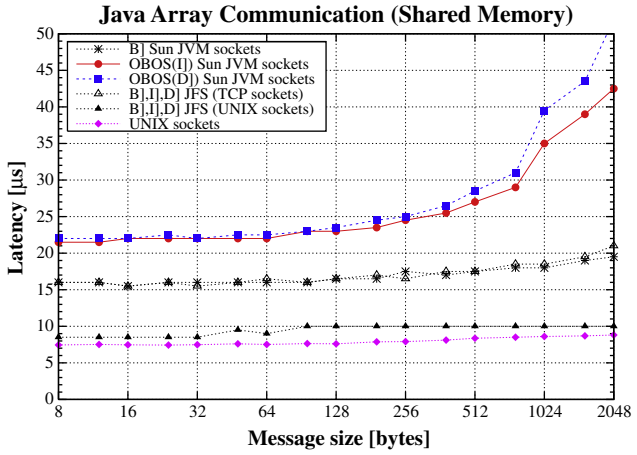

Fig. 11. Latency of shared memory communication.

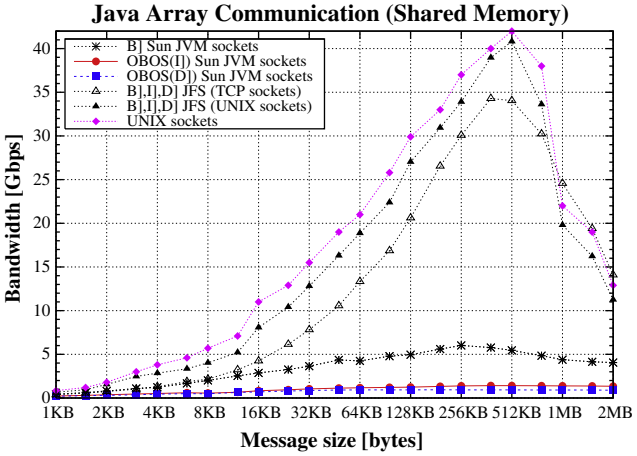

Fig. 12. Bandwidth of shared memory communication.

benefit immediately from JFS thanks to its user and application transparency. The impact on the overall application performance has been analyzed in the current section where the following representative parallel applications have been selected for evaluation: (1) two message-passing applications, LUFact and Moldyn, from the Java Grande Forum Benchmark Suite [33], and (2) two parallel applications, MG and CG, from the ProActive implementation of the NAS Parallel Benchmarks (NPB) [34]. Two high-speed networks have been also selected: Gigabit Ethernet due to its wide deployment, and SCI as the JFS microbenchmarking has shown the best performance on this network. In order to isolate the impact of these networks on performance, only one processor core per node has been used for running the benchmarks on two, four and eight processors. Two processor cores per node have been used for obtaining 16-processor results in order to analyze the behavior of hybrid high-speed network/shared memory (inter-node/intra-node) communication. As the trend is to move to multicore clusters with high-speed networks, the performance of this hybrid approach is of special interest.

6.1. JFS-based message-passing

Two message-passing application kernels, LUFact, a matrix LU factorization, and Moldyn, a molecular dynamics N-body parallel simulation, have been selected (the size `C` benchmark versions) in order to analyze the performance impact of the use of JFS-based message-passing middleware. These kernels have been benchmarked using three MPJ libraries: MPJ/Ibis, MPJ Express and a JFS-based MPJ, developed specifically for showing the benefits of JFS-based middleware, especially the serialization avoidance. On SCI, it has been used JFS instead of JVM sockets over SCIP as underlying layer for MPJ/Ibis and MPJ Express (see Fig. 3) in order to avoid the IP emulation and thus ensure a fair comparison. Therefore, the three MPJ implementations use the same underlying socket library on SCI and the performance differences are exclusively due to their implementation. Thus, the benefits of the JFS-based implementation can be easily noticed.

Fig. 13 shows MPJ LUFact runtimes and speedups. The performance differences on two, four and eight processors are explained exclusively by the communication efficiency of these MPJ libraries on high-speed networks. However, results on 16 processors combine network communication (inter-node) with shared memory communication (intra-node). JFS-based MPJ significantly outperforms MPJ/Ibis and MPJ Express, especially using 16 processors and SCI, obtaining a speedup increase of up to 179%. Both MPJ/Ibis and MPJ Express scale performance only up to eight processors, decreasing their speedups for 16 processors. Nevertheless, JFS-based MPJ obtains higher speedups on 16 processors than on eight processors, thanks to combining efficiently its inter-node and intra-node communication. Fig. 14 shows Moldyn runtimes and speedups. Moldyn is a more computation-intensive code than LUFact, obtaining almost linear speedups for up to eight processors. Nevertheless, for 16 processors MPJ Express and MPJ/Ibis scale performance significantly worse than JFS-based MPJ, which outperforms these libraries up to 14% and 42% on Gigabit Ethernet and SCI, respectively. MPJ Express performs better than MPJ/Ibis for this benchmark, except for 16 processors. However, these differences are small due to the limited influence of communication overhead on the overall performance.

This analysis of MPJ libraries performance has also been useful for evaluating two additional Java sockets libraries: Java NIO and Ibis sockets (see Section 2). Thus, the differences observed among MPJ/Ibis, MPJ Express and the JFS-based MPJ are mainly explained by the socket libraries used in their implementation, Ibis sockets, Java NIO and JFS, respectively. Java NIO sockets obtain the lowest performance, as this implementation is more focused on providing scalability in distributed systems rather than efficient message-passing communication. Ibis sockets significantly outperform Java NIO sockets. Moreover, they are a good estimate for JVM sockets performance, as they show similar results [29]. Finally, JFS clearly achieves the highest performance, showing its improvements a significant impact of on the overall MPJ applications performance.

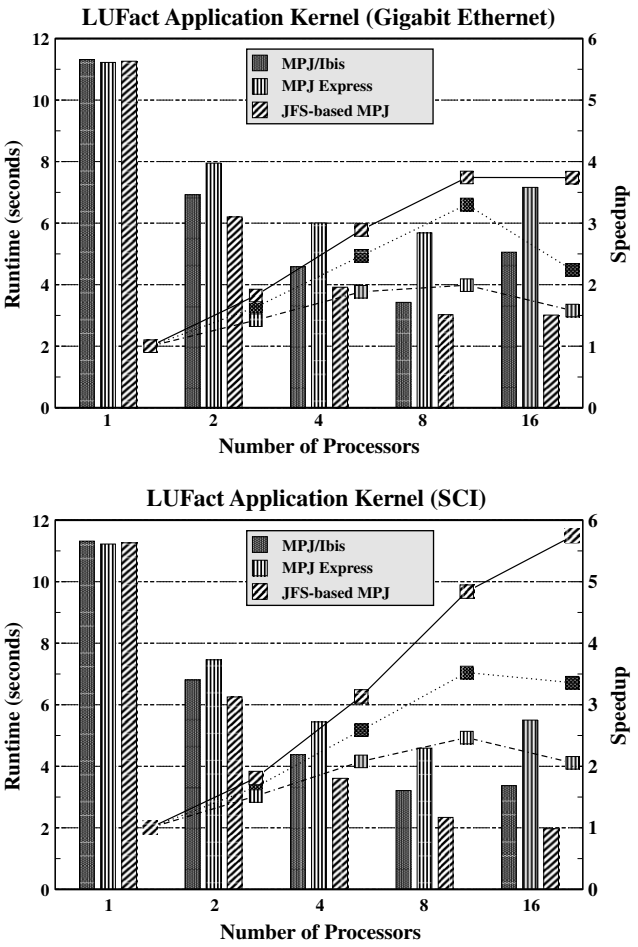

Fig. 13. MPJ LUFact performance.

6.2. JFS-based RMI

As shown in a previous work [35], JFS reduces significantly RMI overhead, up to 10% and 60% for primitive data type arrays on Gigabit Ethernet and SCI, respectively, and up to 63% for object communication, especially sensitive to start-up latency, on SCI. As RMI imposes a significant performance penalty on the ProActive middleware [31] [32], JFS has been used for reducing its overhead. The benefits have been evaluated using two representative communication-intensive applications, MG and CG, from the ProActive NPB [34]. MG is a 3D MultiGrid method with a Poisson solver algorithm, whereas CG solves an unstructured sparse linear system. Fig. 15 shows MG and CG results using SCIP and JFS on SCI. The high memory requirements of MG have prevented this benchmark from being run on a single node. The MG speedups have then been calculated using the runtime on two processors as reference. As can be seen in Fig. 15, JFS increases speedup up to 24% for MG and up to 157% for CG. JFS improves transparently the performance of RMI applications, especially for intra-node communications and on high-speed networks. Therefore, JFS enables parallel and distributed high-level programming without compromising performance.

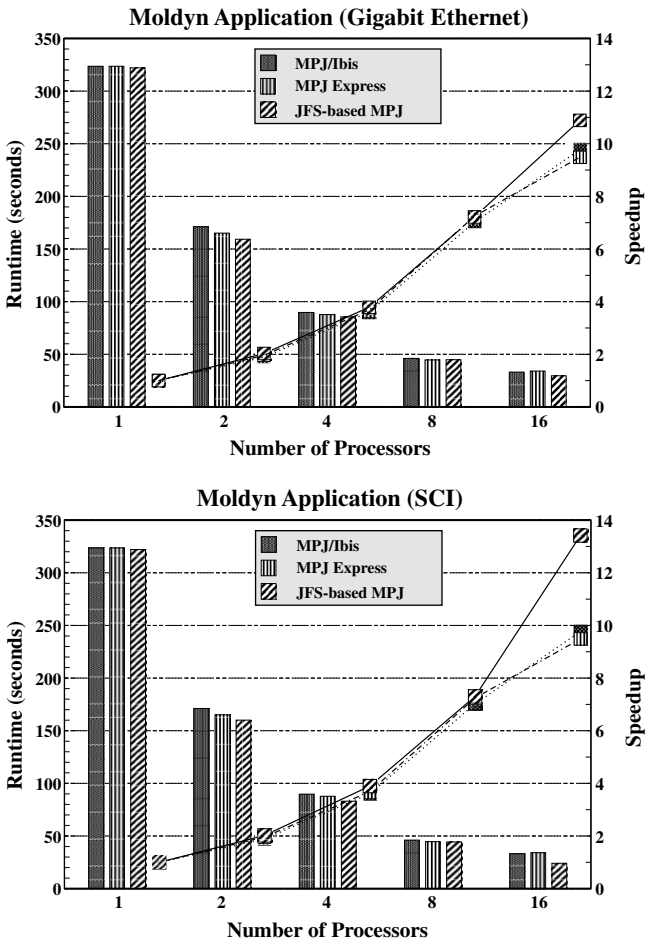

Fig. 14. MPJ Moldyn performance.

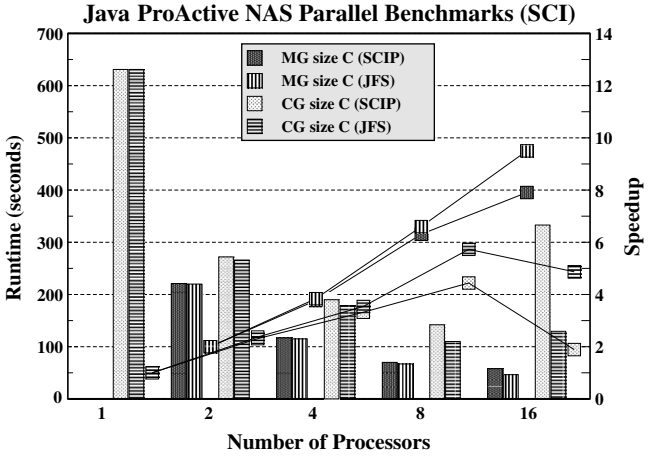

Fig. 15. ProActive NAS parallel benchmarks.

7. Conclusions

The steady increase of cluster components performance, especially for CPUs and high-speed networks, has led to a substantial improvement of the potential overall performance of clusters. In order to take full advantage of the hardware resources, applications have to resort to efficient middleware. Nevertheless, there is a shortage of optimized Java communication libraries. The use of IP emulations on high-speed networks incurs in considerable overhead. Several RMI optimizations for these networks have also been developed, but their performance is not competitive enough. Additional Java communication libraries for high-speed networks lack desirable characteristics such as user and application transparency and the use of widely adopted APIs.

Table 2
JFS performance improvement compared to Sun JVM sockets

	SCI	Myrinet	Gigabit Ethernet	Shared memory
JFS start-up reduction	88%	68%	0%	50%
JFS bandwidth increase	Up to 1305%	Up to 412%	Up to 119%	Up to 4411%

This paper has presented Java Fast Sockets (JFS), an efficient Java communication middleware for high performance clusters. JFS implements the widely used socket API for a broad range of target applications. Furthermore, the use of standard Java compilers and JVMs, and its interoperability and transparency allow for immediate performance increase. The main contributions of JFS are: (1) enabling efficient communication on high performance clusters interconnected via high-speed networks through a general and easily portable solution; (2) avoiding the need of primitive data type array serialization; (3) reducing buffering and unnecessary copies; and (4) the optimization of shared memory (intra-node) communication.

A detailed performance evaluation of JFS has been conducted on a cluster with dual-core nodes for shared memory communication and SCI, Myrinet and Gigabit Ethernet as high-speed interconnects. Table 2 summarizes the performance improvement obtained. JFS has also enhanced the performance of communication-intensive parallel applications obtaining speedup increases of up to 179% (LUFact benchmark on 16 processors) compared to the analyzed socket-based Java message-passing libraries. However, the observed improvements significantly depend on the amount of communication involved in the applications. Additionally, JFS reduces socket processing CPU load up to 60% compared to Sun JVM sockets.

Although JFS has significantly improved parallel and distributed Java applications performance, this library is also intended for middleware developers in order to implement JFS-based higher level communication libraries like Java message-passing and RMI implementations.

Further information, additional documentation and software downloads of this project are available from the JFS Project webpage http://jfs.des.udc.es.

Acknowledgement

This work was funded by the Ministry of Education and Science of Spain under Project TIN2004-07797-C02 and by the Galician Government (Xunta de Galicia) under Project PGIDIT06PXIB 105228PR.

References

[1] G.L. Taboada, J. Touriño, R. Doallo, Performance analysis of Java message-passing libraries on Fast Ethernet, Myrinet and SCI Clusters, in: Proceedings of the 5th IEEE International Conference on Cluster Computing (Cluster'03), Hong Kong, China, 2003, pp. 118–126.
[2] A. Barak, I. Gilderman, I. Metrik, Performance of the communication layers of TCP/IP with the Myrinet Gigabit LAN, Computer Communications 22 (11) (1999) 989–997.
[3] Myricom Inc., GM/MX/Myrinet. Available from: <http://www.myri.com> [Last visited: August 2008].

[4] J.-S. Kim, K. Kim, S.-I. Jung, SOVIA: a user-level sockets layer over Virtual Interface Architecture, in: Proceedings of the 3rd IEEE International Conference on Cluster Computing (Cluster'01), Newport Beach, CA, 2001, pp. 399–408.
[5] IETF Draft, IP over IB. Available from: <http://www.ietf.org/ids.by.wg/ipoib.html> [Last visited: August 2008].
[6] R.G. Börger, R. Butenuth, H.-U. Hei, IP over SCI, in: Proceedings of the 2nd IEEE International Conference on Cluster Computing (Cluster'00), Chemnitz, Germany, 2000, pp. 73–77.
[7] Dolphin Interconnect Solutions Inc., IP over SCI. Dolphin ICS Website. Available from: <http://www.dolphinics.com/products/software.html> [Last visited: August 2008].
[8] S.H. Rodrigues, T.E. Anderson, D.E. Culler, High-performance local-area communication with fast sockets, in: Proceedings of the Winter 1997 USENIX Symposium, Anaheim, CA, 1997, pp. 257–274.
[9] P. Balaji, P. Shivan, P. Wyckoff, D.K. Panda, High performance user level sockets over Gigabit Ethernet, in: Proceedings of the 4th IEEE International Conference on Cluster Computing (Cluster'02), Chicago, IL, 2002, pp. 179–186.
[10] S. Petri, L. Schneidenbach, B. Schnor, Architecture and implementation of a socket interface on top of GAMMA, in: Proceedings of the 28th IEEE Conference on Local Computer Networks (LCN'03), Bonn, Germany, 2003, pp. 528–536.
[11] Intel Corporation, Offload Sockets Framework and Sockets Direct Protocol High Level Design. Available from: <http://infiniband.sourceforge.net/archive/OSF_SDP_HLD.pdf> [Last visited: August 2008].
[12] F. Seifert, H. Kohmann, SCI SOCKET – A Fast Socket Implementation over SCI. Dolphin ICS Website, Available from: <http://www.dolphinics.com/userfiles/files/Whitepaper/sci-socket.pdf> [Last visited: August 2008].
[13] X. Zhang, S. McIntosh, P. Rohatgi, J.L. Griffin, XenSocket: a high-throughput interdomain transport for VMs, in: Proceedings of the 8th ACM/IFIP/USENIX International Middleware Conference (Middleware'07), Newport Beach, CA, 2007, pp. 184–203.
[14] C.-C. Chang, T. von Eicken, Javia: a Java interface to the Virtual Interface Architecture, Concurrency: Practice and Experience 12 (7) (2000) 573–593.
[15] M. Welsh, D.E. Culler, Jaguar: enabling efficient communication and I/O in Java, Concurrency: Practice and Experience 12 (7) (2000) 519–538.
[16] K. Ghouas, K. Omang, H.O. Bugge, VIA over SCI: consequences of a zero copy implementation and comparison with VIA over Myrinet, in: Proceedings of the 1st International Workshop on Communication Architecture for Clusters (CAC'01), San Francisco, CA, 2001, pp. 1632–1639.
[17] J. Maassen, R. Nieuwpoort, R. Veldema, H. Bal, T. Kielmann, C. Jacobs, R. Hofman, Efficient Java RMI for parallel programming, ACM Transactions on Programming Languages and Systems 23 (6) (2001) 747–775.
[18] M. Philippsen, B. Haumacher, C. Nester, More efficient serialization and RMI for Java, Concurrency: Practice and Experience 12 (7) (2000) 495–518.
[19] M. Baker, B. Carpenter, A. Shafi, MPJ Express: towards thread safe Java HPC, in: Proceedings of the 8th IEEE International Conference on Cluster Computing (Cluster'06), Barcelona, Spain, 2006, pp. 1–10.
[20] M. Bornemann, R.V. van Nieuwpoort, T. Kielmann, MPJ/Ibis: a flexible and efficient message passing platform for Java, in: Proccedings of the 12th European PVM/MPI Users' Group Meeting (EuroPVM/MPI'05), Sorrento, Italy, 2005, pp. 217–224.
[21] M. Lobosco, A.F. Silva, O. Loques, C.L. de Amorim, A new distributed Java Virtual Machine for cluster computing, in: Proceedings of the 9th International Euro-Par Conference (Euro-Par'03), Klagenfurt, Austria, 2003, pp. 1207–1215.
[22] W. Zhu, C.-L. Wang, F.C.M. Lau, JESSICA2: A distributed Java Virtual Machine with transparent thread migration support, in: Proceedings of the 4th IEEE International Conference on Cluster Computing (Cluster'02), Chicago, IL, 2002, pp. 381–388.
[23] M. Factor, A. Schuster, K. Shagin, JavaSplit: a Runtime for execution of monolithic Java programs on heterogenous collections of commodity workstations, in: Proceedings of the 5th IEEE International Conference on Cluster Computing (Cluster'03), Hong Kong, China, 2003, pp. 110–117.
[24] P.J. Keleher, Update protocols and cluster-based shared memory, Computer Communications 22 (11) (1999) 1045–1055.
[25] W. Huang, H. Zhang, J. He, J. Han, L. Zhang, Jdib: Java Applications interface to unshackle the communication capabilities of InfiniBand networks, in: Proceedings of the 4th IFIP International Conference Network and Parallel Computing (NPC'07), Dalian, China, 2007, pp. 596–601.
[26] M. Welsh, NBIO: Nonblocking I/O for Java. Available form: <http://www.eecs.harvard.edu/~mdw/proj/java-nbio> [Last visited: August 2008].
[27] R.V. van Nieuwpoort, J. Maassen, G. Wrzesinska, R. Hofman, C. Jacobs, T. Kielmann, H.E. Bal, Ibis: a flexible and efficient Java-based grid programming environment, Concurrency and Computation: Practice and Experience 17 (7–8) (2005) 1079–1107.
[28] G.L. Taboada, J. Touriño, R. Doallo, Efficient Java communication protocols on high-speed cluster interconnects, in: Proceedings of the 31st IEEE Conference on Local Computer Networks (LCN'06), Tampa, FL, 2006, pp. 264–271.
[29] G.L. Taboada, J. Touriño, R. Doallo, High performance Java sockets for parallel computing on clusters, in: Proceedings of the 9th International Workshop on Java and Components for Parallelism, Distribution and Concurrency (IWJPDC'07), Long Beach, CA, 2007, p. 197b (8 pages).
[30] D. Turner, X. Chen, Protocol-dependent message-passing performance on Linux clusters, in: Proceedings of the 4th IEEE International Conference on Cluster Computing (Cluster'02), Chicago, IL, 2002, pp. 187–194.
[31] L. Baduel, F. Baude, D. Caromel, Object-oriented SPMD, in: Proceedings of the 5th IEEE International Symposium on Cluster Computing and the Grid (CCGrid'05), Cardiff, UK, 2005, pp. 824–831.
[32] INRIA, ProActive Website. Available from: <http://proactive.inria.fr> [Last visited: August 2008].
[33] J.M. Bull, L.A. Smith, M.D. Westhead, D.S. Henty, R.A. Davey, A benchmark suite for high performance Java, Concurrency: Practice and Experience 12 (6) (2000) 375–388.
[34] INRIA, NAS Parallel Benchmarks: ProActive implementation. Available from: <http://proactive.inria.fr/nas_benchmarks.htm> [Last visited: August 2008].
[35] G.L. Taboada, C. Teijeiro, J. Touriño, High performance Java Remote Method Invocation for parallel computing on clusters, in: Proceedings of the 12th IEEE Symposium on Computers and Communications (ISCC'07), Aveiro, Portugal, 2007, pp. 233–239.