High Performance Java Sockets for Parallel Computing on Clusters

Guillermo L. Taboada, Juan Touriño, and Ramón Doallo

Computer Architecture Group
Dept. of Electronics and Systems, University of A Coruña
A Coruña, 15071 Spain
{taboada,juan,doallo}@udc.es

Abstract

The use of Java for parallel programming on clusters relies on the need of efficient communication middleware and high-speed cluster interconnect support. Nevertheless, currently there are no solutions that fully fulfill these issues. In this paper, a Java sockets library has been tailored to increase the efficiency of Java parallel applications on clusters. This library supports high-speed cluster interconnects and its API has been extended to meet the requirements of a high performance Java RMI implementation and Java parallel applications on clusters. Thus, it provides Java with a more efficient communication middleware on clusters. The performance evaluation of this middleware on a Gigabit Ethernet (GbE) and a Scalable Coherent Interface (SCI) cluster has shown experimental evidence of throughput increase. Moreover, qualitative aspects of the solution such as transparency to the user, interoperability with other systems and no need of source code modifications are decisive to boost the performance of existing Java parallel applications and their developments in high performance Java cluster computing.

1. Introduction

Cluster computing architectures are a well established option for organizations as they deliver outstanding parallel performance at a reasonable price/performance ratio. Java has gained popularity in all phases of systems development because of appealing characteristics such as platform independence, portability and increasing integrability into existing applications. Nevertheless, the use of Java for parallel application development for clusters is still an emerging option, since performance concerns, especially in the I/O arena, have delayed its use. On clusters, efficient communication performance is key to delivering scalability to parallel applications, but Java lacks efficient communication middleware. Even if the cluster nodes were interconnected by a high-speed network, such as SCI, Myrinet, Infiniband or Giganet, Java would not take advantage of this mainly because these interconnection technologies are not efficiently supported.

Currently, the only way Java is fully supported on high-speed interconnects is resorting to TCP/IP protocol stack emulations. Nevertheless, in this case, one of the main advantages of high-speed interconnects, offloading the host CPU from communication processing, is wasted by the processing of the IP emulation libraries, that add a significant overhead [16]. Examples of IP emulations are IP over GM [12] on Myrinet, LANE driver [8] over Giganet, IP over Infiniband (IPoIB) [7] and ScaIP [2] and SCIP [5] on SCI.

Besides the lack of efficiency supporting high-speed interconnects, the Java Virtual Machine (JVM) does not provide efficient protocols for cluster communications. Some efforts have been done to optimize communications in Java Distributed Shared Memory (DSM) implementations (e.g., CoJVM [10], JESSICA2 [21] and JavaSplit [6]), in high performance Java message-passing libraries (e.g., MPJ Express [1] and MPJ/Ibis [3]), and implementing new Java RMI libraries. Nevertheless, these projects usually lack desirable features such as the use of widely spread standard APIs, no need of source code modification, transparency to the user and full interoperability with other systems.

Our goal is to provide a more efficient Java communication middleware that consists of a high performance sockets library extended to cope with the requirements of Java parallel applications (Section 3). Every Java parallel application can run on top of this library, and even it can serve as transport layer for another Java communication middleware, such as Java RMI. A Java RMI implementation has been adapted (Section 4) to take full advantage of this high performance Java sockets implementation on high-speed interconnection clusters. The use of these two widely spread

1-4244-0910-1/07/$20.00

APIs assures a broad range of communication performance optimization. Moreover, the objective is to optimize communications transparently to the user, not modifying source code, and maintaining interoperability.

2. Related Work

Previous efforts at obtaining efficient Java Sockets have been focused on providing non-blocking communications in order to increase scalability in server applications. In this context two pioneer projects: *NBIO* [19] and *Jaguar* [20] have led to the introduction of some facilities in Java NIO Sockets. Nevertheless, both solutions do not provide neither high-speed interconnection support nor high performance computing tailoring. Currently, most of these problems have been solved by the *Ibis* framework [13], a "pure" Java solution for grid and cluster computing, that supports high-speed networks (Myrinet) and provides several efficient sockets implementations (TCP and Java NIO Sockets) and serialization methods. The process of transforming objects in stream bytes to send across the network can be efficiently done using its own *ibis* serialization. Ibis supports a wide range of object-based communication: method invocation on remote objects (RMI/Ibis) and object groups (GMI), as well as divide-and-conquer parallelism via spawned method invocations (Satin/Ibis) and message-passing applications (MPJ/Ibis). Nevertheless, its numerous communication layers add a significant overhead (see Section 5).

Regarding Java RMI optimizations, different frameworks have been implemented with the efficiency of RMI communication on clusters as their goal. The most relevant ones are KaRMI [14], RMIX [9], Manta [11] and RMI/Ibis. KaRMI is a drop-in replacement for the Java RMI framework that uses a completely different protocol and introduces new abstractions (such as "export points") to improve communications specifically in cluster environments. Nevertheless, KaRMI suffers from performance losses when dealing with large data sets and its interoperability is limited to the cluster nodes. RMIX extends Java RMI functionality to cover a wide range of communication protocols, but the performance on high performance clusters is not satisfactory. The Manta project is a different approach for implementing RMI, based on Java to native code compilation. This approach allows for better optimization, avoids data serialization and class information processing at runtime, and uses a lightweight communication protocol. Finally, RMI/Ibis benefits from being integrated in the *Ibis* framework. Looking for performance RMI/Ibis uses efficient serialization and avoids runtime type inspections.

3. Java Sockets for Parallel Computing

In order to support efficiently Java parallel applications a High Performance Java Socket implementation, named Java Fast Sockets (*JFS*), has been extended. This library was sketched in [17] and its prototype was implemented later. The work presented in this paper is the tailoring of *JFS* to high performance Java parallel applications on clusters. The main achievements are: (1) higher performance on high-speed networks, (2) the increase of communication performance due to reduction of unnecessary copies by implementing more efficient communication protocols, and (3) the reduction of the cost of serialization.

Higher Performance on High-speed Networks. *JFS* supports high-speed interconnects through Java Native Interface (JNI) access to high performance native communication libraries. On SCI, *JFS* makes JNI calls to SCI Sockets, SCILib and SISCI, three native communication libraries [15] on SCI. On Ethernet-based networks, *JFS* makes JNI calls to native TCP/IP sockets. Furthermore, the lightness of the *JFS* implementation, without lose of functionality, is key to delivering the low latency and high bandwidth of the high-speed interconnect to the applications.

JFS Communication Performance Increase. Current Java sockets implementations (e.g., from Sun and IBM JVMs) are implemented in a general way, buffering data and making unnecessary copies, apart from not supporting high performance communication libraries. Figure 1 shows a sketch of the default scenario in Java sockets communications: up to six copies and two serialization/deserialization processes are performed. This inefficiency has been solved in *JFS* implementing a one-copy protocol (see Figure 2) using a direct *ByteBuffer*, a Java NIO Buffer accessible through Java and JNI code, and a zero-copy protocol (see Figure 3) moving data directly to the communication driver. The choice of protocol is based by default on message size, but can be defined by the user based on the application needs. The zero-copy protocol maximizes bandwidth, whereas the one-copy protocol minimizes latency, but increasing the CPU load. The default choice is to use the one-copy protocol for small messages and the zero-copy protocol for long messages. These protocols reduce significantly the number of copies needed.

Serialization Cost Reduction. Java Sockets can only send byte arrays, so Java objects have to be transformed in stream bytes to be sent across the network. In order to avoid, or at least minimize this costly process, *JFS* implements a new native, generic method that can process arrays of any primitive data type and transform them into byte arrays. Thus, one of the most common communication patterns in Java parallel applications, primitive data type arrays, can be seen as byte series by native methods, with even no need of {de}serialization for handling data in native memory.

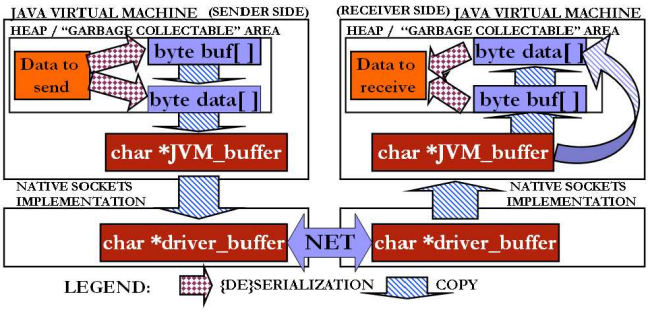


Figure 1. Default Java sockets communication

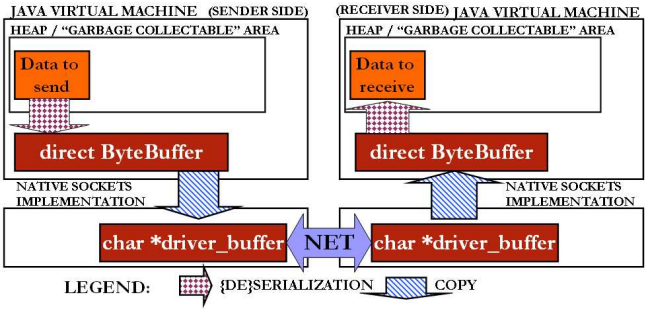


Figure 2. One-copy JFS communication

Besides the increase of communication throughput, *JFS* is transparent to the user, does not require source code modifications and it is a lightweight and portable solution. In fact, the use of native libraries by *JFS* does not jeopardize its portability. In order to take advantage of native code efficiency maintaining the portability of the solution the strategy to follow is the Ibis-based approach. Thus, an efficient pure Java solution is implemented together with native solutions to access low-level cluster native protocols through JNI. At establishing connections, *JFS* looks for a native-based protocol. If any, it will take over the communications. Otherwise, *JFS* resorts to the pure Java efficient socket implementation.

An application can use *JFS* replacing the default *SocketFactory*, in charge of creating sockets using the default

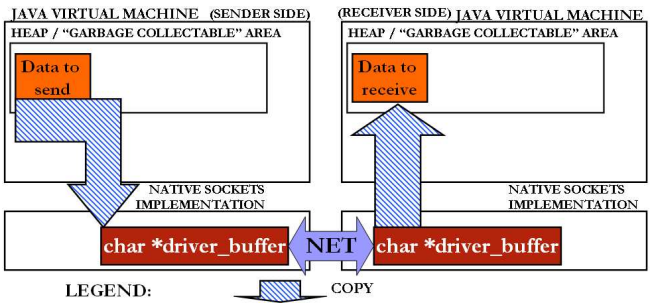


Figure 3. Zero-copy JFS communication

socket implementation (*PlainSocketImpl*), by *JFSFactory*, which creates sockets using the JFS implementation (*JFSImpl*). The transparency to the user is achieved by means of a small Java application that invokes through Java's reflection the *main()* method of the main class of the application after replacing the default sockets implementation by *JFS*. Listing 1 illustrates this procedure:

Listing 1. Replacing default Sockets by JFS

```
SocketImplFactory factory = new JFSFactory();
Socket.setSocketImplFactory(factory);
ServerSocket.setSocketFactory(factory);

Class cl = Class.forName(className);
Method method = cl.getMethod("main", argsTypes);
method.invoke(null, parameters);
```

4. Java RMI Optimization

In addition to the *JFS* extension, a high performance Java RMI library has been designed and implemented. Current Java RMI implementations, in order to increase performance through the use of *JFS*, can only benefit from replacing the transport protocol. Nevertheless, the Java RMI protocol can be implemented in order to obtain more benefits from the new features of *JFS*: (1) the native serialization, although it can be avoided sending directly any primitive data type array, (2) reduction of unnecessary copies, and (3) reduction of the protocol overhead. These reduction of protocol overhead can be made on clusters under some basic assumptions: (1) the use of a shared file system from which the classes can be loaded, (2) homogeneous architecture of the cluster nodes, and (3) the use of a single JVM version.

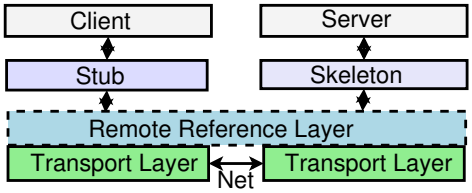


Figure 4. Java RMI layered architecture

Java RMI has been designed following a layered architecture approach, shown in Figure 4. From bottom to top it can be seen: the transport layer, responsible for handling all communications; the remote reference layer, responsible for managing and keeping all references to objects; the stub/skeleton layer, aware of the invocation and execution, respectively, of the methods exported by the objects; and the client and server layer, also known as service layer. At this service layer it can also be found the activation, registry and distributed garbage collection (DGC) services.

In order to optimize Java RMI efficiently, an analysis of the overhead of the methods involved in handling an RMI call has been accomplished. These methods can be grouped in four categories: (1) *Network*, (2) RMI *Protocol* processing, mainly stub and skeleton operation, (3) *Serialization*, and (4) *DGC*. From the analysis of a typical Java RMI call profile (3KB object send), it has been obtained that almost 84% of the overhead belongs to *Network*, 12.7% to the costly serialization process, 3.3% to *Protocol*, and a minimal 0.2% to *DGC*. Having these overheads into account, performance has been improved, focusing on: (1) transport improvements: managing data to reduce buffering and socket delays, (2) serialization overhead reduction, and (3) protocol overhead reduction.

RMI Data Transport Management. By default, all serialized data are inserted in a data block, in order to distinguish data from different objects. These data blocks are created in small buffers, increasing the number of data copies. In order to increase communication efficiency, this strategy has been disabled and the management of the buffer has been simplified, with only a minimal control to avoid serialization and deserialization incoherences, and writing directly to destination buffers instead of buffering the data blocks.

Serialization Overhead Reduction. Java primitive data type arrays have to be serialized in an element-by-element approach. Nevertheless, *JFS* native serialization allows for sending primitive data type arrays, without need of serialization, reducing significantly the serialization overhead.

Protocol Overhead Reduction. Under the basic assumptions previously made for high performance clusters: single JVM version, shared file system and homogeneous architecture, it is possible to (1) avoid the versioning information (the description of a serialized class), (2) avoid the class annotations (the location of the classes), and (3) improve the array processing. Under the considered assumptions, the versioning and annotation information is not needed. Moreover, Java RMI protocol processes arrays as objects, with the consequent useless type checks and reflection operations. The implemented solution creates a specific serialization method to deal with arrays, hence avoiding that useless processing. Thus, an early array detection check is performed, and subsequently the array type is obtained through checking against a list of primitive data types. This list has been empirically obtained from the frequency of primitive data type appearance in high performance Java applications. This list (*double, integer, float, long, byte, Object, char, boolean*) optimizes the type casting compared to the default list (*Object, integer, byte, long, float, double, char, boolean*). Furthermore, this specific serialization method can use *JFS* native serialization and thus reduce its overhead significantly.

5. Performance Evaluation

5.1. Experimental Configuration

The testbed consists of a cluster of dual-processor nodes (PIV Xeon at 3.2 GHz with hyper-threading enabled and 2GB of memory) interconnected via SCI and Gigabit Ethernet (GbE). The SCI interface is a D334 card plugged into a 64bits/66MHz PCI slot, whereas the GbE is an Intel PRO/1000 MT 82546 GB with an MTU of 1500 bytes. The OS is Linux CentOS 4.2 with compilers gcc 3.4.4 and Sun JDK 1.5.0_05. The SCI libraries are SCI Sockets 3.0.3, DIS 3.0.3 (SCILib and SISCI) and the IP Emulation library SCIP 1.2.0. The *Ibis* version is 1.4.

In order to benchmark communications, a Java version of NetPIPE [18] has been developed (there is no Java NetPIPE publicly available). The results considered in this section are the half of the round trip time of a ping-pong test. It has been taken into account that Java micro-benchmarking has some particularities: in order to obtain JVM Just in Time (JIT) results, from running fully optimized native compiled bytecode, 10000 warm-up iterations have to be executed before the actual measurements. It has been measured in Java sockets and Java RMI benchmarks the communication of integer arrays as it is a frequent communication pattern in Java parallel applications.

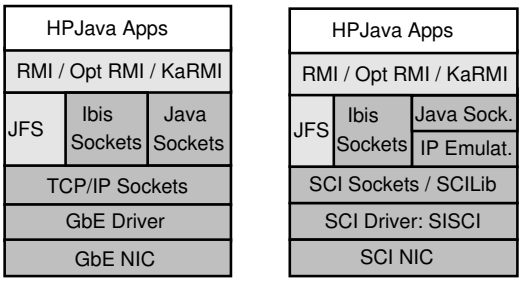


Figure 5. High performance Java parallel applications: software architecture overview

Figure 5 shows an overview of the six-layered proposed architecture for high performance Java parallel applications on GbE and SCI. Given components are colored in dark gray, whereas the contributions presented in this paper: (1) the extension to *JFS* and (2) the optimized Java RMI, are depicted in light gray. *Ibis Sockets* have been adapted to run on top of SCI Sockets. Thus, *Ibis Sockets* implementation and *JFS*'s implementation can be compared fairly, as both use the same base communication libraries. From bottom to top it can be seen the Network Interface Card (NIC) layer, NIC drivers, native sockets, Java Sockets and required IP emulation libraries, Java RMI implementations and high performance Java parallel applications.

5.2. JFS Performance Evaluation

Figures 6 and 7 show experimentally measured latencies and bandwidths of Java Sockets and *JFS* as a function of the message length, for byte and integer arrays on GbE and SCI. Bandwidth graphs (right side of the figures) are useful to compare long-message performance, whereas latency graphs (left side of the figures) serve to compare short-message performance. Regarding the upper two graphs in Figure 6, *JFS* and *Ibis Sockets* present results quite similar to Java Sockets as the three libraries use the same underlying library, native TCP/IP sockets. Nevertheless, the situation is quite different on SCI (see the two lower graphs in Figure 6) where *JFS* clearly outperforms Java Sockets over SCIP, due to the use of an emulation library that adds considerable overhead. Moreover, *JFS* also outperforms *Ibis Sockets*.

Figure 7 presents the results of sending integer arrays. The performance achieved by *JFS*, *Ibis Sockets* and Java Sockets is lower (up to 10-20%) than sending byte arrays, due to the serialization overhead. The *ibis* serialization has been used in *Ibis Sockets*, but the performance of this library is lower (up to three times lower throughput and up to eleven times higher startup) than using *JFS*.

5.3. Optimized Java RMI

Figure 8 presents the results for communicating integer arrays through RMI calls using KaRMI [14], Java RMI and the optimized RMI (labeled as "Opt RMI" in the legends) described in Section 4. Regarding the two upper graphs (GbE), KaRMI shows the lowest latency for short messages ($<$ 1KB), but for larger messages its performance is the worst of the three implementations. The optimized RMI obtains slightly better results than Java RMI. Regarding SCI graphs, KaRMI and Java RMI on SCIP show the poorest results. Nevertheless, substituting Java Sockets as transport protocol by *JFS* improves the results significantly. In this case, KaRMI presents slightly better performance than Java RMI, for all message sizes. KaRMI shows better performance on SCI than on GbE, mainly for being designed to cope with high performance libraries and systems. Thus, KaRMI results on GbE are poorer due to the use of TCP/IP. Regarding the RMI bandwidth on SCI, it can be seen that Java RMI and KaRMI performance drops for messages longer than 256KB caused by a native communication protocol change at this boundary. The optimized RMI presents slightly lower latencies than Java RMI and KaRMI for short messages. For longer messages, specially $>$ 256KB, its performance benefits increase significantly.

5.4. Applications Performance

Figure 9 shows the speedups achieved with two representative Java parallel applications from the Java Grande MPJ benchmark [4]. The Molecular Dynamics application, **MolDyn** *SizeB* version, is an N-body code. For each iteration, six reduce-to-all addition operations update atoms' properties. The Ray Tracer application, **RayTracer** *SizeC* version, renders a scene of 64 spheres, considering images of 2000x2000 pixels. Each node calculates a checksum over its part of the scene, and a reduce operation is used to combine these checksums into a single value.

These applications use MPJ/Ibis on SCI, running on top of SCIP and on top of *JFS* on SCI. *JFS* specially improves **MolDyn** performance on 16 processors, whereas **RayTracer** *SizeC* obtains results very close to the ideal. Regarding these two representative applications, it can be concluded that *JFS* improves significantly performance on communication intensive Java parallel applications on high-speed interconnects.

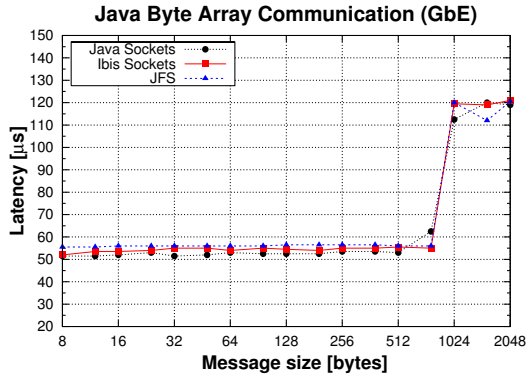
6. Conclusions

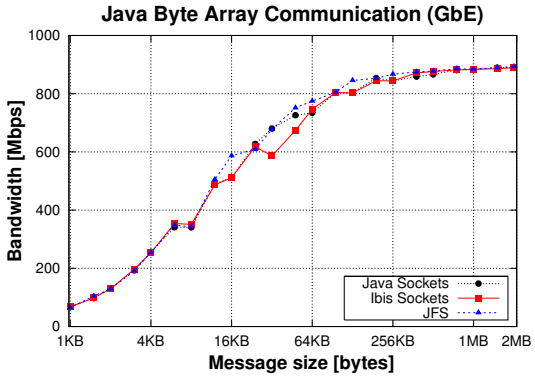
A more efficient Java communication middleware has been presented. This middleware consists of a high performance sockets library, *JFS*, extended to cope with the requirements of Java parallel applications on high-speed interconnection clusters. Moreover, a more efficient Java RMI library has been implemented based on the *JFS* extensions: more efficient communication and serialization. Thus, *JFS* is not only the transport layer, but also the underlying library for the optimized Java RMI implementation. The proposed middleware is transparent to the user, interoperable with other systems, does not need source code modification and offers widely spread APIs (Java Sockets and Java RMI).

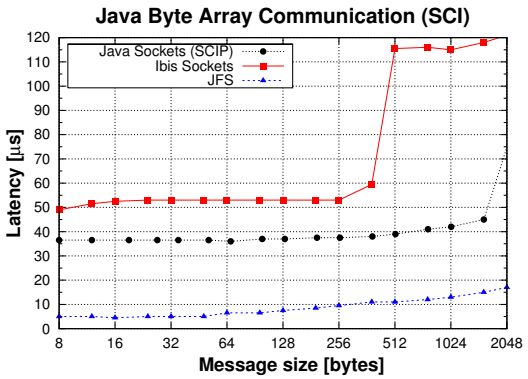
Experimental results have shown that *JFS* greatly improves Java Sockets performance, especially on high-speed interconnects and for communication patterns frequently used in high performance parallel applications, such as arrays of primitive data types. Moreover, the optimized Java RMI reduces significantly the RMI call overhead, especially on supported high-speed interconnects. Furthermore, *JFS* improves Java parallel applications efficiency. These conclusions are backed by the experimental evaluation carried out on a Gigabit Ethernet (GbE) and SCI cluster.

Acknowledgments

This work was funded by the Ministry of Education and Science of Spain under Project TIN2004-07797-C02 and by the Galician Goverment (Xunta de Galicia) under Project PGIDIT06PXIB105228PR.







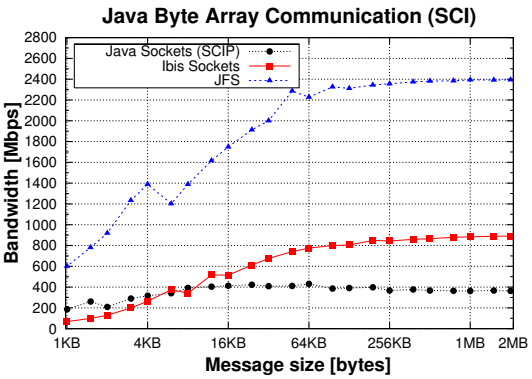
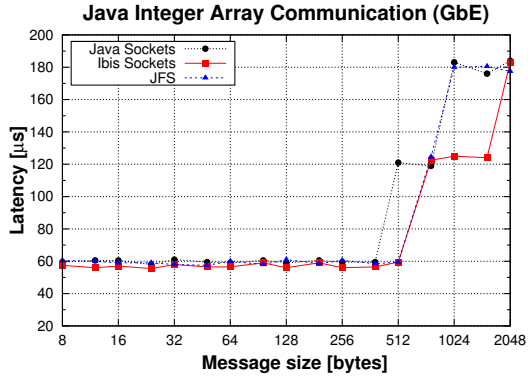
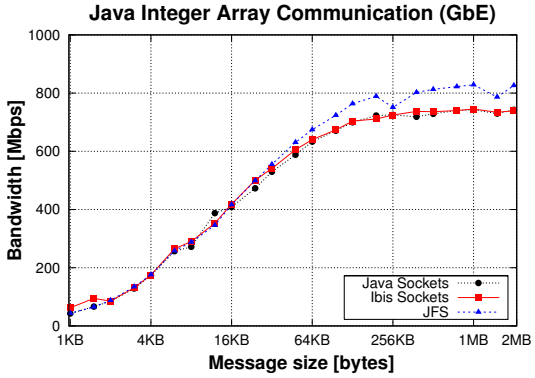
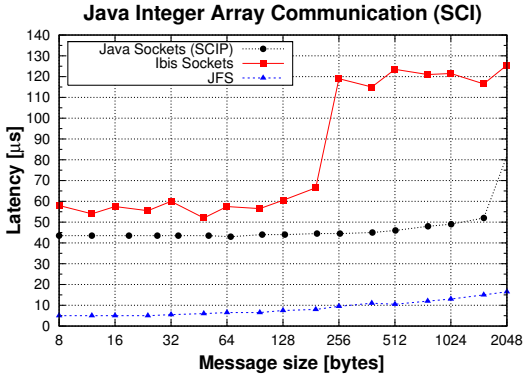


Figure 6. Java Sockets communication performance







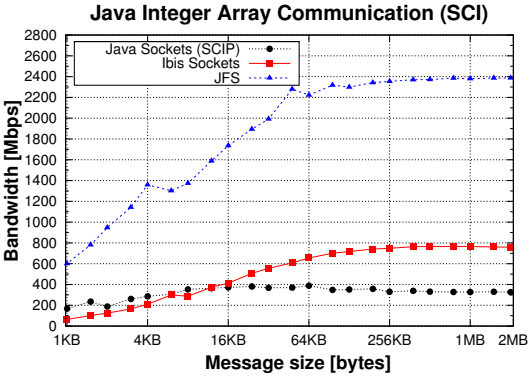


Figure 7. Integer array communication performance

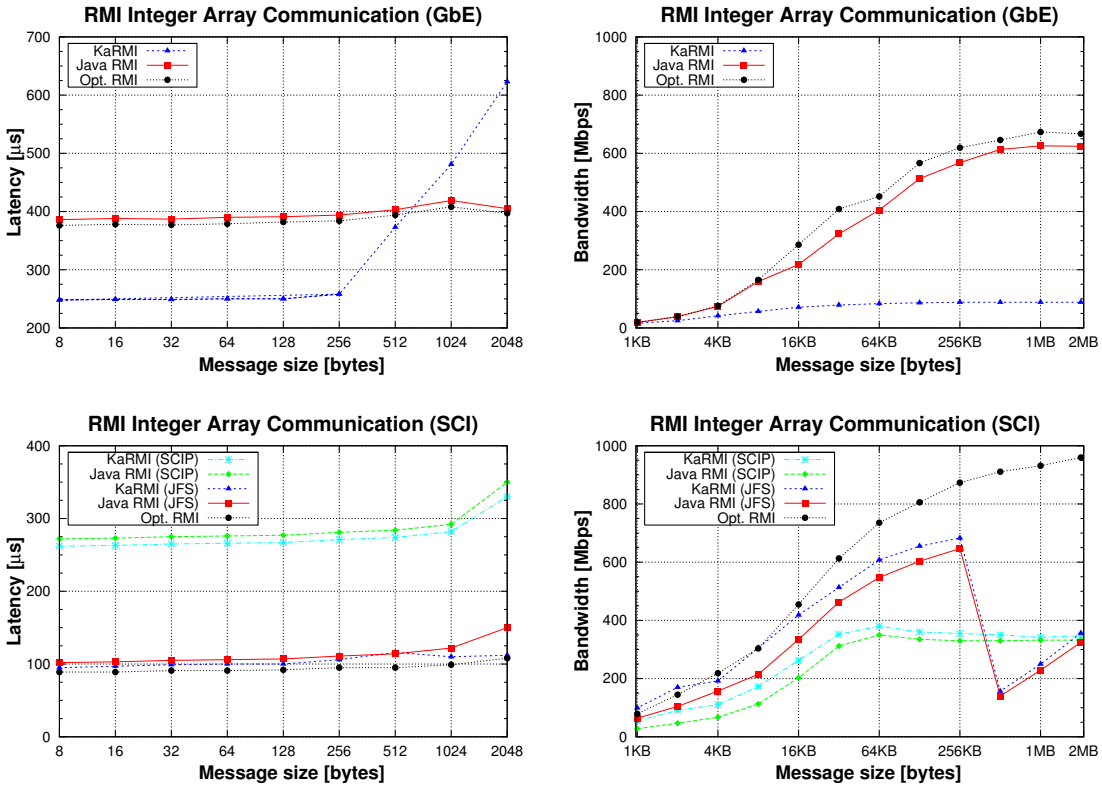


Figure 8. Java RMI communication performance

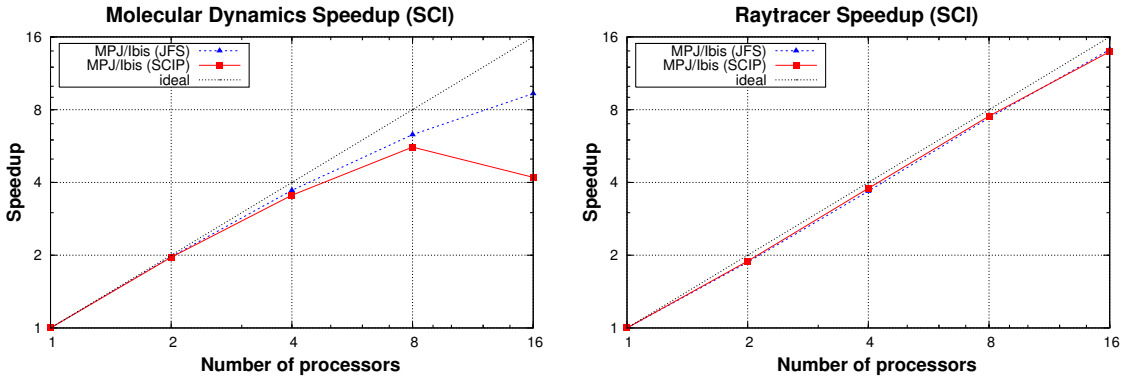


Figure 9. Speedups of MPJ/Ibis Java Grande applications using JFS

References

[1] M. Baker, B. Carpenter, and A. Shafi. MPJ Express: Towards Thread Safe Java HPC. In *Proc. of 8th IEEE Intl. Conference on Cluster Computing (CLUSTER'06)*, Barcelona, Spain, 2006.

[2] R. G. Börger, R. Butenuth, and H.-U. Hei. IP over SCI. In *Proc. 2nd IEEE Intl. Conf. on Cluster Computing (CLUSTER'00)*, pages 73–77, Chemnitz, Germany, 2000.

[3] M. Bornemann, R. V. van Nieuwpoort, and T. Kielmann. MPJ/Ibis: A Flexible and Efficient Message Passing Platform for Java. In *12th European PVM/MPI Users' Group Meeting, (PVM/MPI'05), LNCS 3666, Springer-Verlag*, pages 217–224, Sorrento, Italy, 2005.

[4] J. M. Bull, L. A. Smith, M. D. Westhead, D. S. Henty, and R. A. Davey. A Benchmark Suite for High Performance Java. *Concurrency: Practice and Experience*, 12(6):375–388, 2000.

[5] Dolphin Interconnect Solutions, Inc. IP over SCI. Dolphin ICS Website. http://www.dolphinics.com/products/software/sci_ip.html. [Last visited: January 2007].

[6] M. Factor, A. Schuster, and K. Shagin. JavaSplit: a Runtime for Execution of Monolithic Java Programs on Heterogenous Collections of Commodity Workstations. In *Proc. 5th IEEE Intl. Conf. on Cluster Computing (CLUSTER'03)*, pages 110–117, Hong Kong, China, 2003.

[7] IETF Draft. IP over IB. IETF Website. http://www.ietf.org/ids.by.wg/ipoib.html. [Last visited: January 2007].

[8] J.-S. Kim, K. Kim, and S.-I. Jung. SOVIA: A User-level Sockets Layer Over Virtual Interface Architecture. In *Proc. 3rd IEEE Intl. Conf. on Cluster Computing (CLUSTER'01)*, New Port Beach, CA, 2001.

[9] D. Kurzyniec, T. Wrzosek, V. Sunderam, and A. Slominski. RMIX: A Multiprotocol RMI Framework for Java. In *Proc. 5th Intl. Workshop on Java for Parallel and Distributed Computing (JAVAPDC'03)*, page 140 (7 pages), Nice, France, 2003.

[10] M. Lobosco, A. F. Silva, O. Loques, and C. L. de Amorim. A New Distributed Java Virtual Machine for Cluster Computing. In *Proc. 9th Intl. Euro-Par Conf. (EuroPAR'03)*, pages 1207–1215, Klagenfurt, Austria, 2003.

[11] J. Maassen, R. Nieuwpoort, R. Veldema, H. Bal, T. Kielmann, C. Jacobs, and R. Hofman. Efficient Java RMI for Parallel Programming. *ACM Transactions on Programming Languages and Systems*, 23(6):747–775, 2001.

[12] Myricom Inc. GM/Myrinet. http://www.myri.com. [Last visited: January 2007].

[13] R. V. v. Nieuwpoort, J. Maassen, G. Wrzesinska, R. Hofman, C. Jacobs, T. Kielmann, and H. E. Bal. Ibis: a Flexible and Efficient Java-based Grid Programming Environment. *Concurrency and Computation: Practice & Experience*, 17(7-8):1079–1107, 2005.

[14] M. Philippsen, B. Haumacher, and C. Nester. More Efficient Serialization and RMI for Java. *Concurrency: Practice & Experience*, 12(7):495–518, 2000.

[15] F. Seifert and H. Kohmann. SCI SOCKET - A Fast Socket Implementation over SCI. Dolphin ICS Website. http://www.dolphinics.com/pdf/whitepapers/sci-socket.pdf. [Last visited: January 2007].

[16] G. L. Taboada, J. Touriño, and R. Doallo. Performance Analysis of Java Message-Passing Libraries on Fast Ethernet, Myrinet and SCI Clusters. In *Proc. 5th IEEE Intl. Conf. on Cluster Computing (CLUSTER'03)*, pages 118–126, Hong Kong, China, 2003.

[17] G. L. Taboada, J. Touriño, and R. Doallo. Designing Efficient Java Communications on Clusters. In *Proc. 7th Intl. Workshop on Java for Parallel and Distributed Computing (JAVAPDC'03)*, page 182a, Denver, CO, 2005.

[18] D. Turner and X. Chen. Protocol-Dependent Message-Passing Performance on Linux Clusters. In *Proc. 4th IEEE Intl. Conf. on Cluster Computing (CLUSTER'02)*, pages 187–194, Chicago, IL, 2002.

[19] M. Welsh. NBIO: Nonblocking I/O for Java. http://www.eecs.harvard.edu/~mdw/proj/java-nbio/. [Last visited: January 2007].

[20] M. Welsh and D. E. Culler. Jaguar: Enabling Efficient Communication and I/O in Java. *Concurrency: Practice & Experience*, 12(7):519–538, 2000.

[21] W. Zhu, C.-L. Wang, and F. C. M. Lau. JESSICA2: A Distributed Java Virtual Machine with Transparent Thread Migration Support. In *Proc. 4th IEEE Intl. Conf. on Cluster Computing (CLUSTER'02)*, pages 381–388, Chicago, IL, 2002.