Performance Analysis of Java Message-Passing Libraries on Fast Ethernet, Myrinet and SCI Clusters

Guillermo L. Taboada, Juan Touriño and Ramón Doallo
Computer Architecture Group
Dep. of Electronics and Systems
University of A Coruña, Spain
E-mail: {taboada,juan,doallo}@udc.es

Abstract

The use of Java for parallel programming on clusters according to the message-passing paradigm is an attractive choice. In this case, the overall application performance will largely depend on the performance of the underlying Java message-passing library. This paper evaluates, models and compares the performance of MPI-like point-to-point and collective communication primitives from selected Java message-passing implementations on clusters with different interconnection networks. We have developed our own micro-benchmark suite to characterize the message-passing communication overhead and thus derive analytical latency models.

1. Introduction

Cluster computing architectures are an emerging option for organizations as they offer a reasonable price/performance ratio. The message-passing model provides programming flexibility and generally good performance on these architectures. Java message-passing libraries are an alternative for developing parallel and distributed applications due to appealing characteristics such as platform independence, portability and integration into existing Java applications, although probably at a performance cost. In this work, Java message-passing libraries are analyzed in order to estimate overheads with simple expressions. Our goal is to identify design faults in the communication routines, as well as to provide performance results for Fast Ethernet, Myrinet and SCI (Scalable Coherent Interface) clusters which can guide developers to improve their Java parallel applications.

Related work about Java message-passing evaluation are the papers by Stankovic and Zhang [18], and by Getov et al. [7]. Both works do not derive performance analytical models. Moreover, the experimental results in [7] are restricted to point-to-point primitives on an IBM SP-2, and the scenarios of the experiments in [18] are not representative (eg, they carry out collective measures using up to 5 processors of different architectures connected via 10Mbps Ethernet). Finally, these works are based on currently out-of-date Java message-passing libraries and programming environments.

2. Java Message-Passing Libraries

Research efforts to provide MPI for Java are focused on two main types of implementations: Java wrapper and pure Java. On the one hand, the wrapper-based approach provides efficient MPI communication through calling native methods (in C, C++ or Fortran) using the Java Native Interface (JNI). The major drawback is lack of portability: only a few combinations of platforms, message-passing libraries and JVMs are supported due to interoperability issues. The pure Java approach, on the other hand, provides a portable message-passing implementation since the whole library is developed in Java, although the communication could be relatively less efficient due to the use, in general, of the RMI protocol.

There is a great variety of Java message-passing libraries that use different approaches and APIs due to the lack of a standard MPI binding for Java, although the Message-Passing Working Group within the Java Grande Forum (`www.javagrande.org`) is working on a standard interface, named MPJ [1], in pure Java. One of the major issues that has arisen is how the Java mechanisms can be made useful in association with MPI: it is under study if and how Java's thread model can be used instead of the process-based approach of MPI.

In this work, we have focused on the following MPI-based libraries:

- mpiJava [2], the most active Java wrapper project,

consists of a collection of wrapper classes (with a C++-like interface) that call a native MPI implementation through JNI.

- JMPI [14], a pure Java implementation developed for academic purposes at the University of Massachusetts, following the MPJ specification.
- CCJ [16], a pure Java communication library with an MPI-like syntax not compliant with the MPJ specification. It makes use of Java capabilities such as a thread-based programming model or sending of objects.

Other Java message-passing libraries are:

- JavaMPI [13], an MPI Java wrapper created with the help of JCI, a tool for generating Java-to-C interfaces. The last version was released in January 2000.
- M-JavaMPI [9] is another wrapper approach with process migration support that runs on top of the standard JVM. Unlike mpiJava and JavaMPI, it does not use direct binding of Java programs and MPI. M-JavaMPI follows a client-server message redirection model that makes the system more portable, that is, MPI-implementation-independent. It is not publicly available yet.
- JavaWMPI [11] is a Java wrapper version built on WMPI, a Windows-based implementation of MPI.
- MPIJ is a pure Java MPI subset developed as part of the DOGMA project (Distributed Object Group Metacomputing Architecture) [8]. MPIJ has been removed from DOGMA since release 2.0.
- the commercial JMPI project [4] by MPI Software Technology (do not confuse with [14]) intends to build a pure Java version of MPI specialized for commercial applications. The project has been on hold since 1999.
- PJMPI [12] is a pure Java message-passing implementation strongly compatible with the MPI standard that is being developed at the University of Adelaide in conjunction with a non-MPI message-passing environment called JUMP (not publicly available yet).
- jmpi [5] is another pure Java implementation of MPI built on top of JPVM (see below). The project has been left idle since 1999.

Far less research has been devoted to PVM-based libraries. The most representative projects were JavaPVM (renamed as jPVM [20]), a Java wrapper to PVM (last released in April 1998), and JPVM [6], a pure Java implementation of PVM (last released in February 1999). Performance issues of both libraries were studied in [22].

3. Message-Passing Performance Models

In order to characterize Java message-passing performance, we have followed the same approach as in [3] and [21], where the performance of MPI C routines was modeled on a Fast Ethernet cluster (only MPI-I/O primitives) and on the Fujitsu AP3000 multicomputer, respectively.

Thus, in point-to-point communications, message latency (T) can be modeled as an affine function of the message length n: $T(n) = t_s + t_b n$, where t_s is the startup time, and t_b is the transfer time per data unit (one byte from now on). Communication bandwidth is easily derived as $Bw(n) = n/T(n)$. A generalization of the point-to-point model is used to characterize collective communications: $T(n,p) = t_s(p) + t_b(p)n$, where p is the number of processors involved in the communication.

The Low Level Operations section of the Java Grande Forum Benchmark Suite is not appropriate for our modeling purposes (eg, it only considers seven primitives and timing outliers are not discarded). We have thus developed our own micro-benchmark suite [19] which consists of a set of tests adapted to our specific needs. Regarding point-to-point primitives, a ping-pong test takes 150 measurements of the execution time varying the message size in powers of four from 0 bytes to 1 MB. We have chosen as test time the sextile (25th value) of the increasingly ordered measurements to avoid distortions due to timing outliers. Moreover, we have checked that the use of this value is statistically better than the mean or the median to derive our models (we have obtained minimal least square errors with this value). As the millisecond timing precision in Java is not enough for measuring short message latencies, in these cases we have gathered several executions to achieve higher precision. The parameters t_s and t_b were derived from a linear regression of T vs n. Similar tests were applied to collective primitives, but also varying the number of processors (from 2 up to the number of available processors in our clusters). A Barrier was included to avoid a pipelined effect and to prevent the network contention that might appear by the overlap of collective communications executed on different iterations of the test. The parameters of the model were obtained from the regression of T vs n and p. Double precision addition was the operation used in the experiments with reduction primitives.

4. Experimental Results

4.1. Experiment Configuration

The tests were performed using two clusters. The first cluster consists of 16 single-processor nodes (PIII at 1 GHz with 512 MB of main memory) interconnected via

Primitive	Library	$t_s(p)$ $\{\mu s\}$	$t_b(p)$ $\{ns/byte\}$	$T(_{16B,8})$ $\{\mu s\}$	$Bw(_{1MB,8})$ $\{MB/s\}$
Send	MPICH	69	90.3	72	11.061
	mpiJava	101	100.7	105	9.923
	CCJ	800	138.2	800	7.217
	JMPI	4750	154.4	4750	6.281
Barrier	MPICH	$26 + 33p$	N/A	N/A	N/A
	mpiJava	$44 + 33p$	N/A	N/A	N/A
	CCJ	$382 + 209p$	N/A	N/A	N/A
	JMPI	$1858 + 521p$	N/A	N/A	N/A
Broadcast	MPICH	$7 + 117\lceil lp \rceil$	$-0.3 + 90.4\lceil lp \rceil$	364	3.686
	mpiJava	$19 + 124\lceil lp \rceil$	$10.1 + 90.5\lceil lp \rceil$	406	3.546
	CCJ	$-430 + 1430\lceil lp \rceil$	$6.4 + 130.4\lceil lp \rceil$	3800	2.506
	JMPI	$-9302 + 7151p$	$-123.2 + 175.7p$	41600	0.752
Scatter	MPICH	$92 + 18p$	$39.7 + 11.1\lceil lp \rceil$	232	13.077
	mpiJava	$95 + 19p$	$52.6 + 10.0\lceil lp \rceil$	250	11.522
	CCJ	$534 + 333p$	$51.6 + 20.2\lceil lp \rceil$	5400	7.172
	JMPI	$-5276 + 6938p$	$102.1 + 1.5\lceil lp \rceil$	48000	6.347
Gather	MPICH	$31 + 46p$	$37.7 + 13.3(lp)$	321	12.418
	mpiJava	$45 + 45p$	$38.6 + 14.1(lp)$	358	10.790
	CCJ	$780 + 210p$	$47.1 + 9.6(lp)$	4200	12.221
	JMPI	$2000 + 2000p$	$70.8 + 0.9(lp)$	15600	11.497
Allgather	MPICH	$-78 + 84p$	$52.2 + 11.2\lceil lp \rceil$	620	4.250
	mpiJava	$-65 + 89p$	$47.9 + 15.2\lceil lp \rceil$	640	4.297
	CCJ	$442 + 679p$	$50.6 + 76.6\lceil lp \rceil$	9200	2.028
	JMPI	$-5259 + 7929p$	$108.6 + 63.1\lceil lp \rceil$	48000	2.922
Alltoall	MPICH	$32 + 34p$	$31.6 + 37.1(lp)$	277	3.434
	mpiJava	$38 + 35p$	$54.2 + 37.3(lp)$	284	3.411
	JMPI	$-6836 + 8318p$	$103.9 + 65.6\lceil lp \rceil$	45800	2.940
Reduce	MPICH	$103 + 25p$	$2.7 + 99.6\lceil lp \rceil$	383	3.320
	mpiJava	$113 + 25p$	$11.9 + 99.7\lceil lp \rceil$	394	3.051
	CCJ	$454 + 273p$	$11.9 + 92.7\lceil lp \rceil$	3000	3.218
Allreduce	MPICH	$90 + 32p$	$3.0 + 189.7\lceil lp \rceil$	453	1.747
	mpiJava	$102 + 32p$	$16.6 + 194.5\lceil lp \rceil$	473	1.663
	CCJ	$18 + 591p$	$-110.8 + 358.5\lceil lp \rceil$	6400	1.105
Reducesctr	MPICH	$78 + 41p$	$44.0 + 110.0\lfloor lp \rfloor$	418	2.649
	mpiJava	$90 + 42p$	$65.4 + 112.7\lceil lp \rceil$	440	2.459
Scan	MPICH	$83 + 25p$	$-90.2 + 97.6\lfloor 2lp \rfloor$	426	2.061
	mpiJava	$91 + 27p$	$-69.1 + 99.3\lceil 2lp \rceil$	456	1.936

Table 1. Fast Ethernet: analytical models and measured metrics ($lp = \log_2 p$)

Fast Ethernet and Myrinet 2000. Each node has a ServerWorks CNB20LE chipset and a M3F-PCI64C-2 Myrinet card plugged into a 64bit/33MHz PCI slot. The OS is Linux Red Hat 7.1, kernel 2.4.7-10 and the C compiler is gcc 2.96. This cluster was also used for benchmarking Fast Ethernet. The second cluster (#152 in June 2003 Cluster Top500 list) consists of 8 dual-processor nodes (PIV Xeon at 1.8 GHz with hyperthreading disabled and 1GB of memory) interconnected via Fast Ethernet and SCI (2-D torus topology). Each node has an Intel E7500 chipset and a D334 SCI card plugged into a 64bit/133MHz PCI-X slot. The OS is Red Hat 7.3, kernel 2.4.19 and the C compiler is gcc 3.2.2. We have used the Java Virtual Machine IBM JVM 1.4.0 with JITC technology.

Three Java message-passing libraries were analyzed for each network: the Java wrapper mpiJava 1.2.5, and the pure Java libraries CCJ 0.1 and JMPI. They have been selected as they are the most outstanding projects publicly available. We have used the same benchmark codes for mpiJava and JMPI, whereas CCJ codes are quite different as this library does not follow the MPJ standard. As the mpiJava wrapper implementation needs an underlying

Primitive	Library	$t_s(p)$ $\{\mu s\}$	$t_b(p)$ $\{ns/byte\}$	$T(_{16B,8})$ $\{\mu s\}$	$Bw(_{1MB,8})$ $\{MB/s\}$
Send	MPICH-GM	9	5.40	10	183.30
	mpiJava	15	5.26	16	189.21
	CCJ	650	33.68	700	29.13
	JMPI	3850	52.42	3850	17.88
Barrier	MPICH-GM	$-3+16\lceil lp\rceil$	N/A	N/A	N/A
	mpiJava	$5+15\lceil lp\rceil$	N/A	N/A	N/A
	CCJ	$817+171p$	N/A	N/A	N/A
	JMPI	$477+452p$	N/A	N/A	N/A
Broadcast	MPICH-GM	$3+8\lceil lp\rceil$	$0.012+5.741\lceil lp\rceil$	28	57.77
	mpiJava	$20+17\lceil lp\rceil$	$0.036+5.263\lceil lp\rceil$	101	62.78
	CCJ	$-800+1600\lceil lp\rceil$	$-10.64+40.69\lceil lp\rceil$	4000	8.72
	JMPI	$-8617+5356p$	$-61.57+66.70p$	32400	1.80
Scatter	MPICH-GM	$-7+9p$	$4.321+0.414\lceil lp\rceil$	65	190.26
	mpiJava	$42+10p$	$7.223-0.358\lceil lp\rceil$	131	167.95
	CCJ	$217+604p$	$19.11+10.38\lceil lp\rceil$	5000	18.26
	JMPI	$-8287+6438p$	$46.04+11.66\lceil lp\rceil$	40400	9.23
Gather	MPICH-GM	$7+5p$	$3.813+0.505\lceil lp\rceil$	40	184.90
	mpiJava	$47+5p$	$5.043+0.175\lceil lp\rceil$	91	181.35
	CCJ	$600+400p$	$17.79+4.668\lceil lp\rceil$	4000	28.03
	JMPI	$2165+1271p$	$44.56-1.752\lceil lp\rceil$	11800	18.99
Allgather	MPICH-GM	$-10+15p$	$5.308+1.098\lceil lp\rceil$	104	116.48
	mpiJava	$30+17p$	$8.560+0.483\lceil lp\rceil$	173	100.63
	CCJ	$817+944p$	$13.60+20.79p$	9400	5.25
	JMPI	$-8296+7506p$	$-31.16+41.36p$	42400	2.85
Alltoall	MPICH-GM	$-10+13p$	$4.215+2.717\lceil lp\rceil$	94	79.99
	mpiJava	$37+15p$	$7.477+1.87\lceil lp\rceil$	162	71.61
	JMPI	$-11443+7619p$	$-29.78+40.96p$	39800	2.90
Reduce	MPICH-GM	$12+3p$	$2.830+10.94\lceil lp\rceil$	35	28.08
	mpiJava	$45+4p$	$5.267+11.27\lceil lp\rceil$	91	25.08
	CCJ	$783+196p$	$9.956+32.68\lceil lp\rceil$	2400	9.14
Allreduce	MPICH-GM	$18+4p$	$3.112+16.51\lceil lp\rceil$	63	19.12
	mpiJava	$44+6p$	$4.01+15.54\lceil lp\rceil$	112	19.61
	CCJ	$565+531p$	$-67.66+136.4\lceil lp\rceil$	5600	3.06
Reducesctr	MPICH-GM	$-3+13p$	$9.383+10.91\lceil lp\rceil$	98	24.03
	mpiJava	$24+15p$	$11.59+11.62\lceil lp\rceil$	152	21.27
Scan	MPICH-GM	$13+4p$	$-4.505+9.376\lceil 2lp\rceil$	53	19.08
	mpiJava	$50+6p$	$-0.229+10.26\lceil 2lp\rceil$	112	16.10

Table 2. Myrinet: analytical models and measured metrics ($lp = \log_2 p$)

native MPI implementation (MPI C in our experiments), we have chosen the widely used MPICH implementation among several MPI libraries designed for IP networks. Thus, MPICH 1.2.4 was installed for the Fast Ethernet experiments. MPICH-GM, a port of MPICH on top of GM (a low-level message-passing system for Myrinet networks) developed by Myricom (`www.myri.com`) was used for the Myrinet tests (specifically, MPICH-GM-1.2.4..8 over GM 1.6.3). Regarding SCI, we have used ScaMPI 1.13.8, Scali's (`www.scali.com`) high performance MPI implementation for SCI Linux clusters.

4.2. Analytical Models

Table 1 presents the estimated parameters of the latency models (t_s and t_b) for the standard Send (we have checked that this is the best choice in blocking point-to-point communications) and for collective communications on the Fast Ethernet cluster. Two experimentally measured metrics ($T(n,p)$ for n=16 bytes and p=8 processors, and $Bw(n,p)$ for n=1MB and p = 8) are also provided in order to show short and long message-passing performance, respectively, as well as to compare the different libraries for

Primitive	Library	$t_s(p)$ $\{\mu s\}$	$t_b(p)$ $\{ns/byte\}$	$T(_{16B,8})$ $\{\mu s\}$	$Bw(_{1MB,8})$ $\{MB/s\}$
Send	ScaMPI	4	3.89	5	256.88
	mpiJava	10	3.92	10	254.26
	CCJ	500	86.16	500	11.54
	JMPI	950	90.71	950	10.91
Barrier	ScaMPI	$7 + 0.4p$	N/A	N/A	N/A
	mpiJava	$8 + 1.2p$	N/A	N/A	N/A
	CCJ	$236 + 278p$	N/A	N/A	N/A
	JMPI	$799 + 171p$	N/A	N/A	N/A
Broadcast	ScaMPI	$6\lceil lp \rceil$	$-0.105 + 4.128\lceil lp \rceil$	18	81.71
	mpiJava	$33 + 7\lceil lp \rceil$	$-0.197 + 4.458\lceil lp \rceil$	48	76.71
	CCJ	$-800 + 1400\lceil lp \rceil$	$-4.242 + 93.01\lceil lp \rceil$	3400	3.63
	JMPI	$-2800 + 2900p$	$-89.71 + 95.52p$	20600	1.45
Scatter	ScaMPI	$-5 + 6p$	$2.708 + 0.255\lceil lp \rceil$	43	284.00
	mpiJava	$27 + 6p$	$2.409 + 0.3976\lceil lp \rceil$	76	275.43
	CCJ	$-200 + 586p$	$31.01 + 20.63\lceil lp \rceil$	4400	10.57
	JMPI	$-1200 + 3243p$	$30.34 + 29.76\lceil lp \rceil$	26200	8.11
Gather	ScaMPI	$4 + p$	$0.617 + 1.233\lceil lp \rceil$	11	230.40
	mpiJava	$36 + p$	$1.423 + 1.010\lceil lp \rceil$	47	227.85
	CCJ	$300 + 364p$	$39.20 + 10.39\lceil lp \rceil$	3400	14.32
	JMPI	$2100 + 450p$	$38.18 + 10.95\lceil lp \rceil$	5000	13.17
Allgather	ScaMPI	$-6 + 14\lceil lp \rceil$	$3.555 + 1.340\lceil lp \rceil$	36	131.40
	mpiJava	$23 + 16\lceil lp \rceil$	$2.960 + 1.619\lceil lp \rceil$	71	126.47
	CCJ	$-200 + 1057p$	$-62.03 + 167.7\lceil lp \rceil$	8200	2.12
	JMPI	$-1000 + 3971p$	$18.74 + 61.59\lceil lp \rceil$	31400	4.84
Alltoall	ScaMPI	$-10 + 8p$	$1.717 + 2.333\lceil lp \rceil$	55	113.93
	mpiJava	$22 + 9p$	$2.363 + 2.141\lceil lp \rceil$	95	112.25
	JMPI	$-2400 + 3957p$	$18.30 + 61.77\lceil lp \rceil$	31600	4.78
Reduce	ScaMPI	$1 + 6\lceil lp \rceil$	$9.621 + 1.778\lceil lp \rceil$	19	66.69
	mpiJava	$13 + 8\lceil lp \rceil$	$9.732 + 2.008\lceil lp \rceil$	36	63.30
	CCJ	$500 + 193p$	$42.70 + 56.35\lceil lp \rceil$	2000	4.83
Allreduce	ScaMPI	$-1 + 12\lceil lp \rceil$	$9.332 + 2.561\lceil lp \rceil$	36	58.65
	mpiJava	$7 + 15\lceil lp \rceil$	$8.989 + 3.080\lceil lp \rceil$	55	54.63
	CCJ	$-200 + 657p$	$-1.837 + 183.3\lceil lp \rceil$	5000	1.84
Reducesctr	ScaMPI	$-1 + 8p$	$12.37 + 2.088\lceil lp \rceil$	67	53.55
	mpiJava	$23 + 9p$	$12.94 + 2.180\lceil lp \rceil$	100	51.16
Scan	ScaMPI	$-9 + 16p$	$-3.462 + 5.222p$	39	26.09
	mpiJava	$19 + 7p$	$-5.725 + 8.382p$	72	16.30

Table 3. SCI: analytical models and measured metrics ($lp = \log_2 p$)

each primitive. Tables 2 and 3 present the same results for Myrinet and SCI clusters, respectively. The Fast Ethernet and Myrinet analytical models were derived using up to 16 single-processor nodes, but the SCI models were obtained using only one processor per dual node (and thus the models are for only 8 processors). Reduction primitives are not implemented by the JMPI library and some advanced primitives (Alltoall, Reducescatter, Scan) are not available in the current version of CCJ.

As we can see, the MPI C primitives, which are used as a reference to compare the performance of the Java-based libraries, have the lowest communication overhead; mpiJava, as a wrapper implementation over native MPI, has slightly larger latencies due to the overhead of calling the native MPI routines through JNI. Regarding pure Java implementations (CCJ and JMPI), both the transfer time and, mainly, the startup time increase significantly because of the use of RMI for interprocess communication, which incurs a substantial overhead for each message passed. In fact, RMI was primarily designed for communication across the Internet, not for low latency networks. CCJ shows better performance than JMPI as it implements an RMI optimization that simulates asynchronous communications over the synchronous RMI protocol using multithreading: send operations are performed by separate sending threads. A pool of available sending threads is maintained to reduce thread cre-

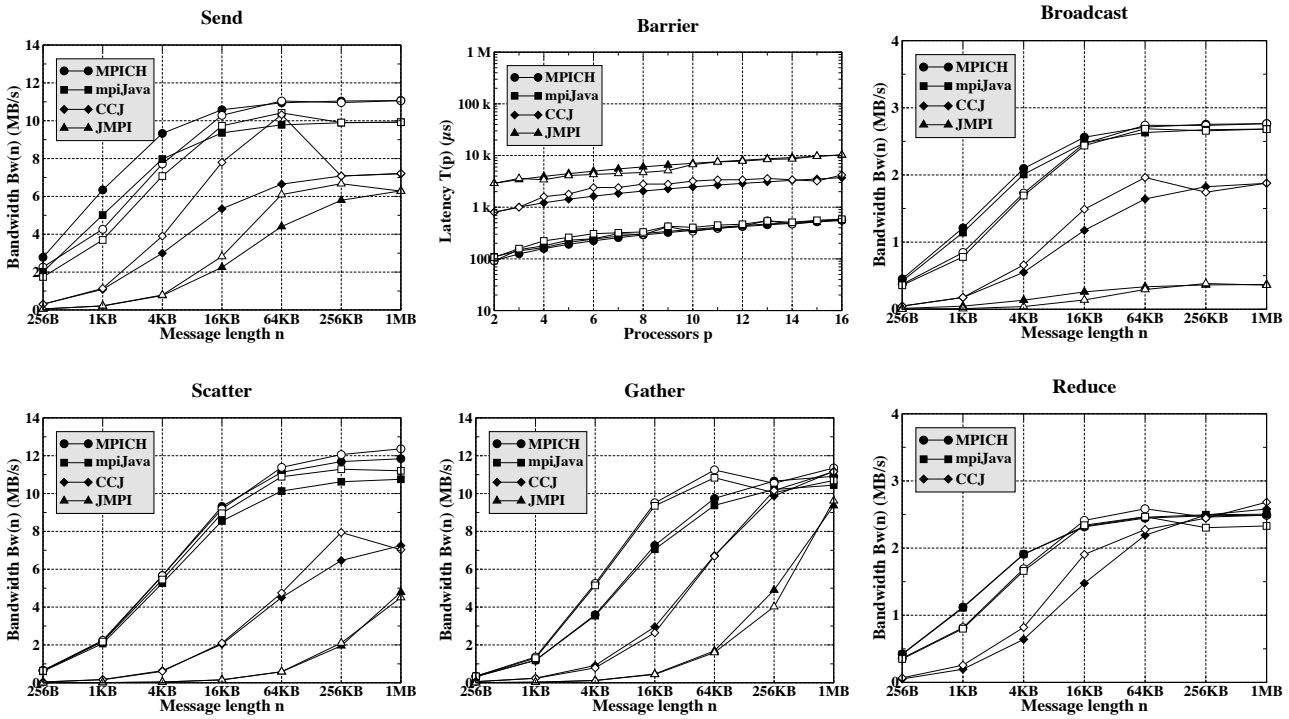

Figure 1. Fast Ethernet: measured and estimated metrics (collective primitives with 16 processors)

ation overhead.

Transfer times ($t_b(p)$) present $O(\log_2 p)$ complexities in almost all collective communications, which reveals a binomial tree-structured implementation of the primitives. Design faults were found in the CCJ implementations of Allgather and Allreduce, where surprisingly their latencies are higher than those of the equivalent primitives Gather+Broadcast and Reduce+Broadcast, respectively. The JMPI implementation of the Broadcast is also inefficient (it is $O(p)$), so that performance degrades significantly as p increases. For instance, the Broadcast latency of 1MB for 8 processors on the Fast Ethernet cluster is three times larger than that of the equivalent Scatter+Allgather.

4.3. Analysis of Performance Results

4.3.1. Point-to-point Primitives The first graph of Figures 1 – 3 shows experimentally measured (empty symbols) and estimated (filled symbols) bandwidths of the Send primitive as a function of the message length, for the different networks. As can be seen from the upper left graph in Figure 1, MPICH and mpiJava point-to-point bandwidths are very close to the theoretical bandwidth of Fast Ethernet (12.5 MB/s). We have observed that communication performance on Fast Ethernet clusters is limited by the interconnection technology. However, communication performance on the Myrinet and SCI clusters is limited by several factors, mainly network protocols, PCI buses and chipsets (processors and memory are not important because network cards have their own processing resources). According to Myrinet 2000 and SCI specifications, their theoretical bandwidths are 1960 MB/s and 667 MB/s (1333 MB/s bidirectional), respectively, whereas the startup time of Myrinet is below $7\mu s$ and $1.46\mu s$ for SCI. In both networks the chipset and the PCI bus are an important bottleneck for communication bandwidth, because they do not reach the transfer rate of the specifications. In our Myrinet nodes the Myrinet/PCI interface is plugged into a 64bit/33MHz PCI slot which achieves a maximum transfer rate of 264 MB/s. A motherboard with a 64bit/66MHz PCI interface would increase bandwidth above 300 MB/s [15]. In the SCI nodes the SCI D334 interface is plugged into a 64bit/133MHz PCI-X slot which achieves a maximum transfer rate of 1056 MB/s. Currently, there is no SCI card that takes advantage of the PCI-X capabilities and thus the interface operates at 66 MHz, resulting in a maximum transfer rate of 528 MB/s. Regarding wrapper point-to-point primitives, the bandwidth of mpiJava Send over MPICH-GM is 198 MB/s (230 MB/s for the low-level GM Send according to the *gm_debug* utility), whereas mpiJava Send over ScaMPI obtains 254 MB/s. Thus, mpiJava achieves 75% of available bandwidth on

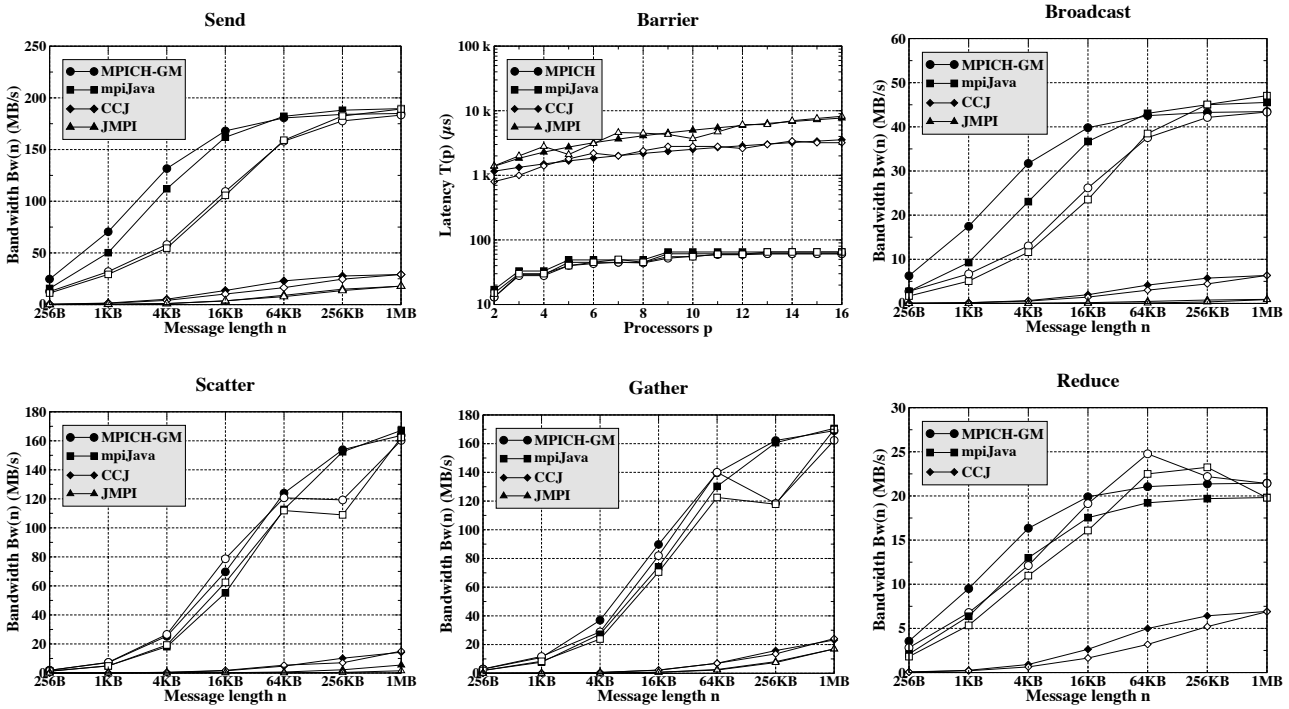

Figure 2. Myrinet: measured and estimated metrics (collective primitives with 16 processors)

Myrinet and only 48% on SCI.

Pure Java libraries (CCJ and JMPI) need IP to work. In order to provide IP over Myrinet we had to enable Ethernet emulation over GM, whereas ScaIP was installed to provide IP over SCI. According to the upper left graphs of Figures 2 and 3, pure Java Send on Myrinet and SCI slightly outperform Fast Ethernet: startup times are in the same order of magnitude and bandwidth is clearly higher on the Myrinet cluster, although it is quite low as compared with the theoretical network bandwidth. This poor performance is due to the communication technology (Java RMI), which is not optimized for low latency networks. Nevertheless, IP over GM achieves better performance than IP over SCI because three software layers are used in SCI to emulate IP (from upper to lower level: ScaIP, ScaMac and ScaSCI), which degrades SCI performance.

4.3.2. Collective Primitives Measured and estimated bandwidths for some collective primitives are also depicted in Figures 1 – 3 (except the Barrier graph that shows latencies). Note that bandwidths are not aggregated, as they are computed simply by dividing message length by communication time (T). In many cases, the estimated values (filled symbols) are hidden by the measured values (empty symbols), which means a good modeling.

Fast Ethernet results are presented in Figure 1. As expected, the bandwidth of the mpiJava routine and the underlying MPICH implementation are very similar (mpiJava calls to native MPI have low overhead). Pure Java primitives show poor performance for short messages (latencies are about one order of magnitude slower than on mpiJava), but they obtain closer results to those of the Java wrapper library as message size increases. CCJ shows better performance than JMPI (particularly for the Broadcast primitive) due to the use of asynchronous messages and a more efficient communication pattern design.

Figure 2 shows Myrinet bandwidths. As in Fast Ethernet mpiJava values remain close to those of the corresponding MPI native implementation (MPICH-GM), but the performance gap between Java wrapper and pure Java primitives increases significantly on Myrinet. Thus, mpiJava primitives have startup times between two and six times faster than on Fast Ethernet, and transfer times about one order of magnitude faster; nevertheless, pure Java primitives have startup times similar to those measured on Fast Ethernet and transfer times between only two and three times faster than on Fast Ethernet.

SCI bandwidths are depicted in Figure 3. As can be seen from these graphs (and according to the models of Table 3) mpiJava primitives on SCI have startup times between two and three times faster than on Myrinet, slightly faster transfer times (around 30%) for data movement primitives, and

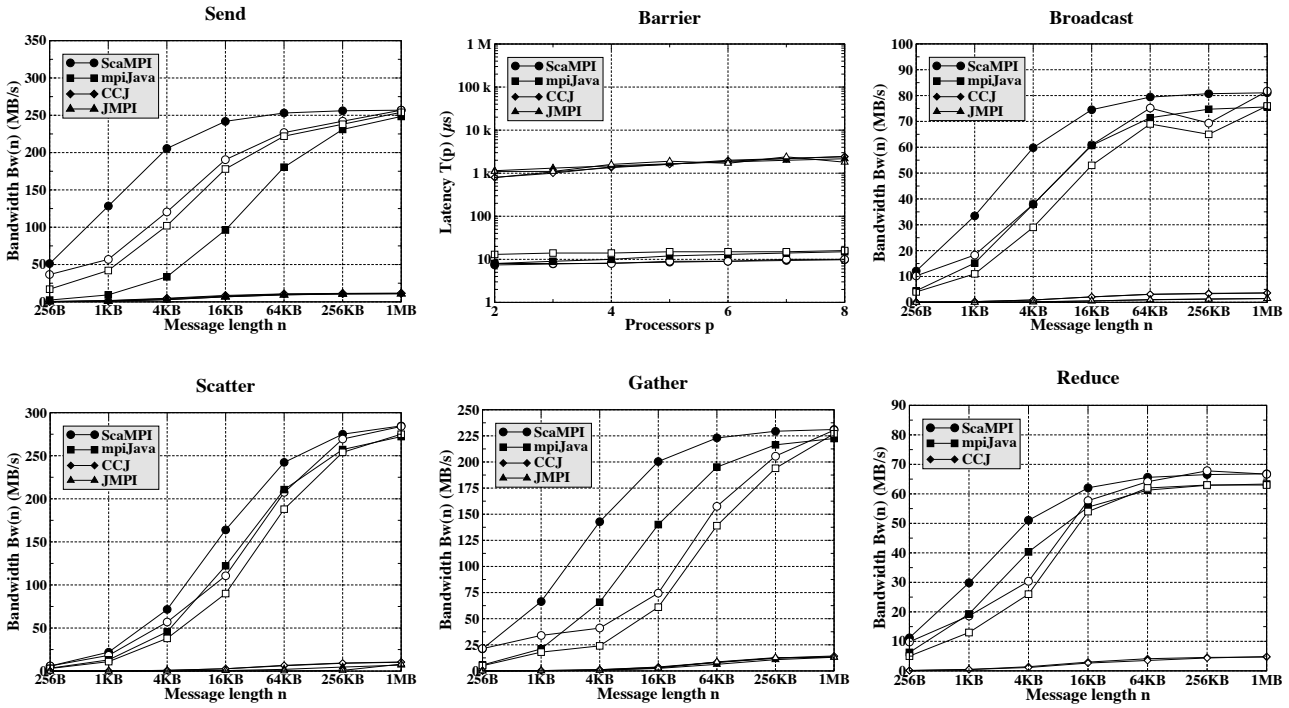

Figure 3. SCI: measured and estimated metrics (collective primitives with 8 processors)

between two and three times faster transfer times for reduction primitives. Although pure Java startup times are faster than on Myrinet, transfer times are around twice slower (due to the poor IP emulation), so the gap between Java wrapper and pure Java performance continues to widen on SCI, mainly for long messages.

4.3.3. Additional Performance Issues We have empirically observed that ScaMPI primitives show, in general, better performance results than SCI-MPICH, another native MPI implementation for SCI Linux clusters. Another remarkable issue is the influence of the JVM on performance. We have observed that, in general, IBM JVM with JITC obtains better results than Sun JVM with HotSpot technology, mainly for messages longer than 64KB. Startup times, nevertheless, are very similar.

5. Conclusions

The characterization of message-passing communication overhead is an important issue in developing environments. As Java message-passing is an emerging option in cluster computing, this kind of studies serve as objective and quantitative guidelines for cluster parallel programming. We have selected the most well-known Java-based libraries: mpiJava, CCJ and JMPI. Additionally, we have also analyzed several MPI C libraries for comparison purposes. The design of our own message-passing micro-benchmark suite allowed us to obtain more accurate models. From the evaluation of the experimental results using different networks, we can conclude that mpiJava presents a good performance, very similar to that of the corresponding native MPI library, although mpiJava is not a truly portable library. Pure Java implementations show poorer performance, mainly for short messages due to the RMI overhead. This performance is almost independent of the underlying network, due to the overhead added by IP emulation, which does not take advantage of Myrinet or SCI bandwidth (eg, the startup time of IP over GM is four times slower than on GM, and bandwidth is 32% smaller). Among the research efforts undertaken to optimize RMI calls, KaRMI [17] deserves to be mentioned. The KaRMI library can operate both over IP (for Ethernet networks) and GM (for Myrinet), and achieves significant latency reductions. The implementation of pure Java message-passing libraries using KaRMI would substantially improve performance, mainly startup times. Another possible optimization is to compile Java programs to native code, thus achieving performances comparable to C/C++ codes; but in this case Java's portability is lost. A straightforward option would be to use GCJ, the GNU compiler for Java, but currently it does not support RMI. An alternative approach was followed by CCJ developers: in order to improve performance, they use Manta [10], a Java to

native code compiler with an optimized RMI protocol. It is possible to install Manta on Myrinet, but it uses several software layers that require considerable configuration efforts.

There is still a long way for pure Java message-passing libraries to efficiently exploit low latency networks. Research efforts should concentrate on this issue to consolidate and enhance the use of pure Java message-passing codes. Middleware approaches that increase RMI performance, as well as the optimization of the communication pattern design for each primitive are necessary to reduce the performance gap between Java wrappers and pure Java libraries. The gap is even wider for SCI networks where far less research has been done, and thus the optimization of Java message-passing performance on SCI clusters is the goal of our future work.

Acknowledgments

This work was supported by Xunta de Galicia (Projects PGIDT01-PXI10501PR and PGIDIT02-PXI10502IF). We gratefully thank CESGA (Galician Supercomputing Center, Santiago de Compostela, Spain) for providing access to the Myrinet cluster.

References

[1] M. Baker and B. Carpenter. MPJ: a Proposed Java Message Passing API and Environment for High Performance Computing. In *2nd Int. Workshop on Java for Parallel and Distributed Computing (IPDPS 2000 Workshop), Cancun, Mexico, Lecture Notes in Computer Science*, volume 1800, pages 552–559. Springer, 2000.

[2] M. Baker, B. Carpenter, G. Fox, S. Ko, and S. Lim. mpiJava: an Object-Oriented Java Interface to MPI. In *1st Int. Workshop on Java for Parallel and Distributed Computing (IPPS/SPDP 1999 Workshop), San Juan, Puerto Rico, Lecture Notes in Computer Science*, volume 1586, pages 748–762. Springer, 1999.

[3] J. Barro, J. Touriño, R. Doallo, and V. Gulías. Performance Modeling and Evaluation of MPI-I/O on a Cluster. *Journal of Information Science and Engineering*, 18(5):825–836, 2002.

[4] G. Crawford, Y. Dandass, and A. Skjellum. The JMPI Commercial Message Passing Environment and Specification: Requirements, Design, Motivations, Strategies, and Target Users. Technical report, MPI Software Technology, Inc., 1998.

[5] K. Dincer. Ubiquitous Message Passing Interface Implementation in Java: jmpi. In *13th Int. Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing, IPPS/SPDP'99, San Juan, Puerto Rico*, pages 203–207, 1999.

[6] A. Ferrari. JPVM: Network Parallel Computing in Java. *Concurrency: Practice & Experience*, 10(11-13):985–992, 1998.

[7] V. Getov, P. Gray, and V. Sunderam. MPI and Java-MPI: Contrasts and Comparisons of Low-Level Communication Performance. In *Supercomputing Conference, SC'99, Portland, OR*, 1999.

[8] G. Judd, M. Clement, and Q. Snell. DOGMA: Distributed Object Group Metacomputing Architecture. *Concurrency: Practice & Experience*, 10(11-13):977–983, 1998.

[9] R. K. K. Ma, C. L. Wang, and F. C. M. Lau. M-JavaMPI: a Java-MPI Binding with Process Migration Support. In *2nd IEEE/ACM Int. Symposium on Cluster Computing and the Grid, CCGrid'02, Berlin, Germany*, pages 255–262, 2002.

[10] 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.

[11] P. Martin, L. M. Silva, and J. G. Silva. A Java Interface to WMPI. In *5th European PVM/MPI Users' Group Meeting, EuroPVM/MPI'98, Liverpool, UK, Lecture Notes in Computer Science*, volume 1497, pages 121–128. Springer, 1998.

[12] J. A. Mathew, H. A. James, and K. A. Hawick. Development Routes for Message Passing Parallelism in Java. In *the ACM 2000 Java Grande Conference, San Francisco, CA*, pages 54–61, 2000.

[13] S. Mintchev and V. Getov. Towards Portable Message Passing in Java: Binding MPI. In *4th European PVM/MPI Users' Group Meeting, EuroPVM/MPI'97, Crakow, Poland, Lecture Notes in Computer Science*, volume 1332, pages 135–142. Springer, 1997.

[14] S. Morin, I. Koren, and C. M. Krishna. JMPI: Implementing the Message Passing Standard in Java. In *4th Int. Workshop on Java for Parallel and Distributed Computing (IPDPS 2002 Workshop), Fort Lauderdale, FL*, pages 118–123, 2002.

[15] Myrinet Experiences Exchange Site: PCI Performance. http://www.conservativecomputer.com/myrinet/perf.html [Last visited: August 2003].

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

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

[18] N. Stankovic and K. Zhang. An Evaluation of Java Implementations of Message-Passing. *Software - Practice and Experience*, 30(7):741–763, 2000.

[19] G. L. Taboada. Java Message-Passing Micro-Benchmark. http://www.des.udc.es/~gltaboada/micro-bench/ [Last visited: August 2003].

[20] D. Thurman. jPVM – A Native Methods Interface to PVM for the Java Platform. http://www.chmsr.gatech.edu/jPVM, 1998. [Last visited: August 2003].

[21] J. Touriño and R. Doallo. Characterization of Message-Passing Overhead on the AP3000 Multicomputer. In *30th Int. Conference on Parallel Processing, ICPP'01, Valencia, Spain*, pages 321–328, 2001.

[22] N. Yalamanchilli and W. Cohen. Communication Performance of Java-Based Parallel Virtual Machines. *Concurrency: Practice & Experience*, 10(11-13):1189–1196, 1998.