Performance Analysis of Message-Passing Libraries on High-Speed Clusters

Guillermo L. Taboada, Juan Touriño and Ramón Doallo
Department of Electronics and Systems
University of A Coruña, Spain
{taboada,juan,doallo}@udc.es

Abstract

This paper presents a performance analysis of message-passing overhead on high-speed clusters. Communication performance is critical for the overall high-speed cluster performance. In order to analyze the communication overhead, a new linear model proposed in this work is used for its characterization. Performance models have been derived using our own micro-benchmark suite on MPI C and emerging Java message-passing libraries. These models predict communication overhead quite accurately. Representative performance metrics have also been obtained in order to evaluate message-passing performance and establish comparisons among different message-passing libraries and clusters. Besides the evaluation process, communication models are useful to optimize parallel applications. Several model-based performance optimizations have been reported. Thus, inefficient primitives have been replaced by more efficient equivalent combinations of primitives. Through an application-level kernel benchmarking it has been analyzed the influence of multiple processor nodes and the message-passing overhead on the overall application performance. From this analysis, it has been concluded that current message-passing implementations do not fully benefit from multiple processor nodes.

Keywords: Cluster, Myrinet, SCI, Message-passing, MPI, Java, Performance Analysis

1 Introduction

There is a growing interest of both scientific and enterprise environments in high-speed clusters as they deliver outstanding parallel performance at a competitive cost. A high-speed cluster consists of computing nodes connected together by a specific purpose high-speed network for achieving higher communication performance on clusters. SCI (Scalable Coherent Interface), Myrinet, Quadrics, Infiniband and 10 Gigabit Ethernet are examples of high-speed interconnects. Scalability is a key factor to confront new challenges in cluster computing, and it depends heavily on the use of high-speed interconnects. But this scalability must reach the parallel application level, and here is where the message-passing paradigm plays an important role, providing programming flexibility and good performance on these architectures.

In this work, C and Java message-passing libraries are analyzed on high-speed clusters in order to estimate overheads. The inclusion of Java message-passing libraries is motivated by the emergence of Java as an option for high performance computing [1]. The goal of this paper is to identify inefficient primitive implementations, as well as to report performance results for these specific environments, particularly for Myrinet and SCI clusters, which can guide developers to improve their parallel applications. A proposal of a more accurate analytical model for high-speed cluster communications, as well as a micro-benchmark suite (`http://www.des.udc.es/~gltaboada/micro-bench/`), are made available to parallel programmers. These tools provide a useful way to quantify the influence of the message-passing libraries and system configuration on the overall application performance. This influence has been corroborated through an application-level kernel benchmarking. The obtained analytical performance models are also useful in optimizing message-passing performance. Thus, communication overhead can be reduced through replacing inefficient communication primitives by more efficient equivalent combinations of primitives.

The paper is organized as follows: the next section introduces existing message-passing performance models and analyzes their suitability for evaluation purposes. As the accuracy and simplicity of these models have not been as expected, a new linear model is proposed, focused on obtaining higher accuracy on high-speed clusters. Section 3 presents the formulation of this model, some performance metrics derived from it, the micro-benchmarking process and a preliminary accuracy analysis. Section 4 presents experimental results: the communication performance of two clusters, with two representative high-speed interconnects, SCI and Myrinet, has been modeled and analyzed. A further discussion on the experimental results and performance estimation is the focus of Section 5, together with a proposal of a model-based performance optimization. Section 6 presents an analysis of the influence of message-passing overhead on applications through an application-level kernel benchmarking. Section 7 analyzes the influence of the use of multiple processor nodes on the overall cluster performance. This evaluation has been done with the aid of the previous application-level kernel benchmarking. Finally, Section 8 concludes the paper with a summary of contributions and future research directions.

2 Message-Passing Performance Models

The appropriateness of existing communication models has been evaluated in terms of their simplicity and accuracy for high-speed clusters. Models discussed in this paper can be classified into LogP- and linear-based models.

The LogP model [2] characterizes communications by four parameters: network communication time L, overhead o, gap g and number of processors P. Some LogP variants have been proposed to support additional characteristics by adding parameters to the model. Thus, LogGP [3] introduces G, gap per byte, to support long messages, LoPC [4] and LoGPC [5] add C to model resource contention, LogGPS [6] incorporates synchronization costs by adding S, and LogPQ [7] introduces Q, referring to communication queues. Additional models are *memory* logP [8] which applies and augments the original LogP model to estimate overheads in a hierarchical memory subsystem, parametrized LogP (P-LogP) [9], which presents a gap $g(m)$ that depends on the message size m, $\log_n$P [10], that addresses the communication cost as a sum of middleware, memory and interconnection network overheads, and HLogP [11], which is targeted to model Grid systems.

Regarding the appropriateness of these models, LogP is too basic to perform a thorough analysis. This model assumes single processor nodes and small messages, determining that it is only effective when L dominates overall cost. In this case, the influence of message size and data distribution on memory communication overhead is negligible. The need to include these parameters has led models to include G, gap per byte, or the data size. However, this is effective only in tightly synchronized communication patterns. In fact, the contention C for message-processing resources is a significant factor in the total application runtime for many fine-grain message-passing algorithms, particularly on clusters. Nevertheless, LogP with additional G and/or C parameters usually omits significant costs, such as the influence of the memory gap on performance. *memory* logP models this influence, although only for shared memory architectures. The model $\log_n$P extends *memory* logP (in fact, *memory* logP is $\log_1$P) taking into account the number of communication steps. Thus, $\log_3$P would describe communications on high-speed clusters: (1) communication memory / Network Interface Card (NIC), (2) communication NIC / NIC, and (3) communication NIC / memory. Experimental results from characterizing communication overhead using these models on high-speed clusters report average absolute relative errors of 28% for LogGP predictions, and of 5% for $\log_3$P [10]. Nevertheless, these accurate results are limited to regular access patterns.

Linear models are also a popular method to characterize message-passing overhead. These models are usually based on Hockney's model for point-to-point communications and on Xu and Wang's model for collective primitives [12]. Thus, message latency (T) of point-to-point communications is modeled as an affine function of the message length n: $T(n) = t_0 + t_b n$, where t_0 is the startup time, the time taken for a zero length message, and t_b is the transfer time per byte. 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: $T(n,p) = t_0(p) + t_b(p)n$, where p is the number of processors involved in the communication. This characterization of message-passing overhead is relatively easy to develop and usually provides good predictions, but its simplicity is thought to be a restricting factor to its accuracy.

The lack of accuracy of linear models on high-speed clusters affects both to t_0 and t_b parameters. The combination into a unique parameter t_0 of the overhead (o) and network communication time (L) differentiated in the LogP model is considered to be only appropriate for long messages, not giving enough detail for short messages [3]. Moreover, as linear models usually assume constant t_b, the accuracy of the models turned out to be much better on Ethernet-based than on high-speed clusters, where different high performance communication protocols, with different t_b, are used depending on the message size. A previous work [13] on modeling communication performance on high-speed clusters has shown the limitations of the Hockney's model to predict performance accurately. In fact, Hockney's model on Fast Ethernet predicts performance with average absolute relative errors of 13% for Send and 21% for collective communications. Hockney's model on SCI presented average absolute relative errors of 18% and 28%, respectively.

Once turned out to be unsuitable the linear model due to the dearth of accuracy, the log$_n$P model was selected as the most suitable choice among LogP-based models. Nevertheless, apart from its lack of direct collective primitive support, it exhibits a certain complexity in its formulation. Although the possibility of simplification by ignoring some parameters exists, sometimes this is not an advantageous choice. In fact, while too many parameters keep non-experts from drawing conclusions about performance, too few parameters do not provide enough information.

3 Modeling Process and Performance Metrics

This paper aims at using a model realistic enough to characterize more accurately communication overhead despite the complexity of current communication middleware, but simple enough for programmers to design and analyze parallel algorithms overhead. As existing models do not fit completely this purpose, a new linear model is proposed to address the main drawbacks of high-speed cluster communications modeling. This model takes into account the influence of different protocols involved in the communication process. This is done by augmenting the linear model described in [12] with a new parameter, t_i, which is the intercept from the linear regression of $T(n) - t_0$ versus n. In high-speed clusters, t_0 is quite small and t_i is usually higher. According to the previous considerations, message latency (T) of point-to-point communications on high-speed clusters should be modeled as $T(n) = t_0 + t_i + t_b n$. Nevertheless, this tentative model predicts inaccurately T for short messages (e.g. $T(0) = t_0$ and the model predicts $T(0) = t_0 + t_i$). In order to solve this issue, t_i must be weighted by the ratio of transfer time ($t_b n$) to the latency predicted by Hockney's model ($t_0 + t_b n$). Thus, point-to-point communications are modeled as:

$$T(n) = t_0 + t_i(\frac{t_b n}{t_0 + t_b n}) + t_b n$$

and collective communications are modeled generalizing the point-to-point model:

$$T(n,p) = t_0(p) + t_i(p)(\frac{t_b(p) n}{t_0(p) + t_b(p) n}) + t_b(p) n$$

Regarding point-to-point communications, this new model predicts accurately $T(0) = t_0$, and shows higher accuracy than Hockney's model, especially for medium messages. In fact, the higher relative difference between this model and Hockney's model occurs at a t_0/t_b-byte message. This maximum relative difference has been obtained by setting the derivative of $(T_{proposed}(n) - T_{Hockney}(n))/T_{Hockney}(n)$ equals to zero and solving for n. This value, t_0/t_b, varies on high-speed clusters from 1KB to tens of KB, in the range of medium messages. In fact, Hockney's model usually underestimates latency of medium messages on high-speed clusters. The reason for this is that message-passing libraries use different communication protocols for short and long messages. Long message protocols usually show lower t_b than short message protocols, focused on lower t_0. As t_b is obtained from a linear regression of T vs. n in which the long message performance dominates, its value is quite similar to the t_b of long message protocols. Thus, using the obtained t_b, short message latency is underestimated.

In order to illustrate this scenario, an example is provided: an MPI C primitive on an SCI cluster presents $t_0 = 4\mu s$, $t_i = 13\mu s$ and $t_b = 3.89ns/byte$ (see ScaMPI Send in Table 2). The estimates of the models are $T_{Hockney}(4KB) = 20\mu s$ and $T_{proposed}(4KB) = 30\mu s$. As $T_{measured}(4KB) = 33\mu s$ the proposed model estimates performance more accurately.

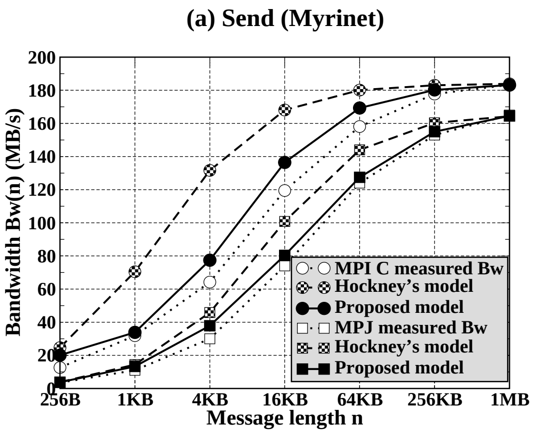
The addition to Hockney's model of a new explanatory variable (t_i) has shown that increasing slightly the complexity of the model, higher accuracy can be obtained, specially for medium messages. A different alternative would be defining a function in pieces for each communication protocol. Nevertheless, this approach requires knowledge about protocol boundaries.

A benchmark suite for both C and Java message-passing libraries appropriate for the modeling process has not been found. Thus, a micro-benchmark suite has been developed (`http://www.des.udc.es/~gltaboada/micro-bench/`). It consists of a set of tests for both C and Java codes adapted to the modeling needs. Regarding point-to-point primitives, a ping-pong test takes 150 measurements of the runtime varying the message size in powers of four from 0 bytes. It has been chosen as test time the minimum value to avoid distortions due to timing outliers. The parameter t_0 is the startup time. The parameters t_i (intercept) and t_b (slope) were derived from a linear regression of $T(n) - t_0$ 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 the testbed). The parameter $t_0(p)$ was derived from a linear regression of startup times vs p. The parameters $t_i(p)$ and $t_b(p)$ were derived from a regression of $T(n,p) - t_0(p)$ vs n and p. 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. Double precision addition was the operation used in computational primitives (Reduce, Allreduce, Reduce-scatter and Scan).

In order to test the accuracy of the proposed model the average absolute relative error of 20 random messages for each primitive has been calculated. The results, a 7% error for Send and below 7% error for collective primitives, are much better than the 18% and 28% error for Hockney's model for Send and collective primitives, respectively. Moreover, the predictions obtained from this model (Section 4) are consistent with the application-level kernel benchmarking (Section 6).

Figure 1 illustrates, through bandwidth graphs, the better fitting of experimentally measured bandwidth (empty symbols) by the proposed model compared to Hockney's model. Graph (a) shows Send bandwidth on Myrinet, and Graph (b) Broadcast bandwidth on SCI. The complete details of the experimental results and models are presented in Section 4. It can be seen that the estimates improve especially on the native message-passing library (MPI C), as there are more major differences among native communication protocols than among Java communication protocols for Java message-passing (MPJ) [14]. It can also be observed that the higher relative difference between the proposed model and Hockney's model occurs at a t_0/t_b-byte message. t_0/t_b is 1KB for MPI C and 11KB for MPJ. In fact, at this point, the proposed model estimates the bandwidth much better than Hockney's model.

Two metrics are derived from the model: the asymptotic bandwidth $Bw_{as}(p) = 1/t_b(p)$, the maximum throughput achievable when $n \rightarrow \infty$, and the specific performance $\pi_0(p) =$



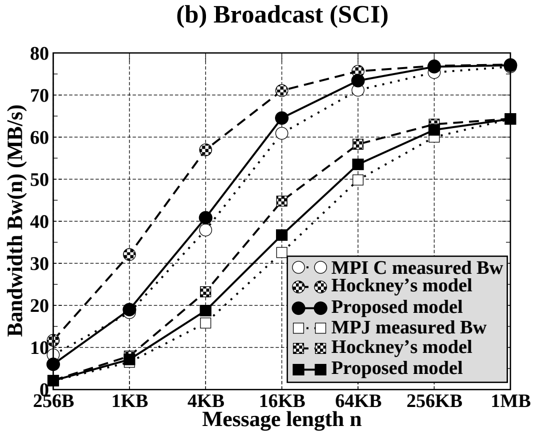


Figure 1: Hockney's model vs. proposed model comparison

$1/t_0(p)$. Bw_{as} shows long message performance, whereas π_0 characterizes short message bandwidth. Another metric is the aggregated asymptotic bandwidth $Bw_{as}^{ag}(p) = f(p)Bw_{as}(p)$, defined as the ratio of the total number of bytes transferred in the collective operation to the time required to perform the operation, as $n \to \infty$. The function $f(p)$ is the relationship between the total number of bytes transferred in the collective primitive and the message length. $f(p)$ depends on the communication pattern of each primitive: e.g. a Broadcast of n bytes to p processors implemented with a binomial tree sends $p-1$ messages of n bytes. In our case $f(p) = p-1$ for Broadcast, Alltoall, Reduce and Scan; $f(p) = (p-1)/p$ for Scatter and Gather; $f(p) = 2(p-1)$ for Allreduce, and $f(p) = (p^2-1)/p$ for Allgather and Reduce-scatter. Similarly, the aggregated specific performance is defined as $\pi_0^{ag}(p) = f(p)\pi_0(p)$ to show the performance of a collective operation for short messages. All these metrics for collective primitives are functions depending on p. In order to have numbers rather than functions to straightforwardly compare the performance of the different message-passing libraries, peak metrics have also been used in our experimental results (see Tables 1–4): the peak aggregated bandwidth $Bw_{as}^{pag} = \max_{2 \le p \le p^{max}} Bw_{as}^{ag}(p)$, and the peak aggregated specific performance $\pi_0^{pag} = \max_{2 \le p \le p^{max}} \pi_0^{ag}(p)$, being p^{max} the maximum p available. For point-to-point communications $Bw_{as}^{pag} = 1/t_b$ and $\pi_0^{pag} = 1/t_0$.

4 Experimental Results

4.1 Cluster Hardware/Software Configuration

Performance analytical models have been obtained from two high-speed clusters. The first cluster consists of 16 single-processor nodes (PIII at 1 GHz and 512 MB of memory) interconnected via Myrinet 2000 cards plugged into 64bit/33MHz PCI slots. The OS is Linux Red Hat 7.1, kernel 2.4, C compiler gcc 2.96, and Java Virtual Machine (JVM) Sun 1.5.0. The second cluster consists of 8 dual-processor nodes (PIV Xeon with hyperthreading at 1.8 GHz and 1GB of memory) interconnected via D334 SCI cards plugged into 64bit/66MHz PCI slots in a 2-D torus topology. The OS is Red Hat 7.3, kernel 2.4, C compiler gcc 3.2.2, and JVM Sun 1.5.0. Three different hardware configurations have been used for the SCI cluster: SCI-single, running one message-passing process on each node; SCI-dual, running two message-passing processes on each node; and SCI-dual w/HT (with hyperthreading enabled), running four message-passing processes on each node. The hyperthreading allows one processor to operate as two processors internally, with a potential increase in performance claimed to be of about 30%, according to the manufacturer, Intel. Thus, a dual node with hyperthreading enabled has 4 "virtual" processors. The other two configurations, SCI-single and SCI-dual, have hyperthreading disabled.

Two MPI C libraries have been analyzed on the SCI cluster: ScaMPI (version 1.13.8), and SCI-MPICH (version 1.2.1), an MPICH implementation for SCI. As ScaMPI has shown better performance than SCI-MPICH, especially on SCI-dual and SCI-dual w/HT, only some SCI-MPICH models are shown for comparative purposes (Tables 2 and 3). MPICH-GM (version 1.2.4..8), a port of MPICH on top of GM (a low-level message-passing system for Myrinet) was selected for Myrinet.

Three representative Java message-passing libraries have been selected: mpiJava [15] (version 1.2.5), MPJ/Ibis [16] (version 1.4) and MPJ Express [17] (version 0.26). The mpiJava library consists of a collection of wrapper classes that call a native MPI implementation through Java Native Interface (JNI). On Myrinet, mpiJava calls MPICH-GM, whereas on SCI, it calls ScaMPI. This wrapper-based approach provides efficient communication relying on native libraries, adding just a small JNI overhead. Nevertheless, its major drawback is the lack of portability, caused by the need of a native MPI implementation. This problem is overcome with the use of "pure" Java message-passing libraries that implement the whole messaging system in Java. Nevertheless, these implementations are less efficient than native implementations. MPJ/Ibis is an MPI-like "pure" Java message-passing implementation integrated in the Ibis framework [18]. It is implemented on top of TCPIbis Sockets (similar to Java I/O Sockets). MPJ Express is another MPI-like "pure" Java message-passing implementation based on Java NIO Sockets. It implements more high-level MPI features than MPJ/Ibis, like derived datatypes, virtual topologies and inter-communicators. It also includes a runtime execution environment. Despite these differences, in terms of performance both "pure" Java libraries behave similarly. In fact, the differences between these libraries are mainly explained by the underlying communication layer. TCPIbis Sockets obtain lower t_0 and higher t_b than Java NIO Sockets. Thus, MPJ/Ibis shows slightly better performance for short messages, whereas MPJ Express achieves higher bandwidths. For conciseness, only one "pure" Java message-passing library has been modeled. MPJ/Ibis has been selected as the representative library for showing slightly better short message performance, an extremely important characteristic on high-speed clusters.

4.2 Analytical Models and Metrics

Table 1 presents the parameters of the latency models ($t_0(p)$, $t_i(p)$ and $t_b(p)$) for the standard Send and for collective communications on the Myrinet cluster. Two peak metrics derived from the models (π_0^{pag} and Bw_{as}^{pag}, see Section 3) are also provided in order to show short and long message performance, respectively, as well as to compare among libraries for each primitive. Regarding these two metrics, the higher, the better. Tables 2, 3 and 4 present the same results for the different SCI configurations: SCI-single, SCI-dual and SCI-dual w/HT, respectively. These models are valid for communications from 2 nodes up to the total number of processors of the cluster. Thus, the models are valid for $2 \leq p \leq 16$ on Myrinet, for $2 \leq p \leq 8$ on SCI-single, for $4 \leq p \leq 16$ on SCI-dual, and for $8 \leq p \leq 32$ on SCI-dual w/HT. Transfer times, $t_b(p)$, show $O(\log_2 p)$ complexity in almost all collective communications, which reveals a binomial tree-structured implementation of the primitives. Nevertheless, inefficient communication patterns have been detected on ScaMPI and MPJ/Ibis Scan (they are $O(p)$). Other implementations, e.g. MPJ/Ibis Allreduce, performs badly. In this particular case a Reduce followed by a Broadcast performs better than the equivalent Allreduce. This statement can be obtained from the values of $t_0(p)$ and $t_b(p)$ from the tables (e.g. $t_{b\_Allreduce}(p) > t_{b\_Reduce}(p) + t_{b\_Broadcast}(p)$). Both $t_0(p)$ and $t_i(p)$ usually present $O(p)$ complexities.

4.2.1 Native Communications Libraries

As can be observed from Tables 1–4, native primitives on the SCI cluster show, in general, lower startups and transfer times per byte than on the Myrinet cluster. These differences can be attributed to: (1) the lower theoretical startup of the interconnect: $1.46\mu s$ for SCI and $7\mu s$ for Myrinet, (2) the higher theoretical bandwidth of the PCI bus, 528MB/s on the SCI cluster and 264MB/s on the Myrinet cluster, and (3) the higher computational power of the nodes, dual PIV Xeon at 1.8GHz on the SCI cluster and PIII at 1GHz on the Myrinet cluster.

Regarding performance metrics Bw_{as}^{pag} and π_0^{pag} from the tables, it can be seen that ScaMPI outperforms SCI-MPICH, except for Reduce-scatter and Scan. Generally, these metrics present the highest values (best performance) on SCI-dual, although communication primitives with more complex communication patterns, such as Alltoall, present the highest values on SCI-single.

4.2.2 Java Communication Libraries

From the models it can be observed that mpiJava adds little overhead to the underlying message-passing library. Regarding MPJ/Ibis, both the transfer time and, mainly, the startup time, increase significantly with respect to the native libraries. This overhead corresponds to: (1) the additional communication layers involved in the communication, TCPIbis Sockets and Ibis Portability Layer (IPL), and (2) the interpreted nature of the JVM, basic for the portability of the library. The most immediate way of running this library on high-speed interconnects is on top of IP emulation libraries: IP over GM on Myrinet and ScaIP on SCI. Nevertheless, MPJ/Ibis was slightly adapted to run on top of Sockets-GM on Myrinet, and on top of SCI Sockets.

Regarding peak performance metrics, it can be observed that MPJ/Ibis collective primitives generally present the highest values (best performance) on SCI-single configuration, except for computational primitives (Reduce, Allreduce, Reduce-scatter and Scan).

Table 1: Myrinet: analytical models and peak aggregated metrics ($lp = \log_2 p$)

Primitive	Library	$t_0(p)$	$t_i(p)$	$t_b(p)$	π_0^{pag}	Bw_{as}^{pag}
		$\{\mu s\}$	$\{\mu s\}$	$\{ns/byte\}$	$\{KB/s\}$	$\{MB/s\}$
Send	MPICH-GM	9	20	5.330	111.1	187.6
	mpiJava	15	20	5.360	66.67	186.6
	MPJ/Ibis	65	69	5.951	15.38	168.0
Barrier	MPICH-GM	$-3 + 16\lceil lp \rceil$	N/A	N/A	245.9	N/A
	mpiJava	$5 + 15\lceil lp \rceil$	N/A	N/A	230.8	N/A
	MPJ/Ibis	$194 + 73p$	N/A	N/A	11.01	N/A
Broadcast	MPICH-GM	$3 + 8\lceil lp \rceil$	$17 + 23\lceil lp \rceil$	$0.017 + 5.649\lceil lp \rceil$	428.6	663.3
	mpiJava	$20 + 17\lceil lp \rceil$	$33 + 31\lceil lp \rceil$	$0.136 + 5.741\lceil lp \rceil$	170.5	649.4
	MPJ/Ibis	$22 + 21p$	$3 + 24p$	$3.006 + 6.670\lceil lp \rceil$	41.90	505.3
Scatter	MPICH-GM	$-7 + 9p$	$1 + 11p$	$4.271 + 0.412\lceil lp \rceil$	45.45	158.9
	mpiJava	$42 + 10p$	$39 + 13p$	$4.336 + 0.421\lceil lp \rceil$	9.146	156.3
	MPJ/Ibis	$37 + 19p$	$8 + 23p$	$4.421 + 0.673\lceil lp \rceil$	6.667	135.9
Gather	MPICH-GM	$7 + 5p$	$13 + 7p$	$3.782 + 0.503\lceil lp \rceil$	29.41	165.4
	mpiJava	$47 + 5p$	$44 + 7p$	$4.981 + 0.174\lceil lp \rceil$	11.19	140.4
	MPJ/Ibis	$78 + 8p$	$83 + 16\lceil lp \rceil$	$6.216 + 0.487\lceil lp \rceil$	6.818	114.8
Allgather	MPICH-GM	$-10 + 15p$	$3 + 19p$	$5.272 + 1.093\lceil lp \rceil$	75.00	1653
	mpiJava	$30 + 17p$	$41 + 23p$	$8.489 + 0.479\lceil lp \rceil$	52.77	1532
	MPJ/Ibis	$17 + 61p$	$4 + 72p$	$4.096 + 2.970\lceil lp \rceil$	16.05	997.6
Alltoall	MPICH-GM	$-10 + 13p$	$-6 + 16p$	$4.182 + 2.690\lceil lp \rceil$	75.76	1004
	mpiJava	$37 + 15p$	$28 + 19p$	$7.371 + 1.83\lceil lp \rceil$	54.15	1014
	MPJ/Ibis	$296 + 523p$	$213 + 465p$	$5.810 + 3.857\lceil lp \rceil$	1.731	706.3
Reduce	MPICH-GM	$12 + 3p$	$9 + 5p$	$2.698 + 10.83\lceil lp \rceil$	250.0	326.0
	mpiJava	$45 + 4p$	$29 + 6p$	$5.161 + 11.16\lceil lp \rceil$	137.6	301.2
	MPJ/Ibis	$107 + 98\lceil lp \rceil$	$63 + 100\lceil lp \rceil$	$7.618 + 15.38\lceil lp \rceil$	30.06	217.0
Allreduce	MPICH-GM	$18 + 4p$	$21 + 6p$	$3.219 + 16.35\lceil lp \rceil$	365.9	437.2
	mpiJava	$44 + 6p$	$58 + 8p$	$4.319 + 15.39\lceil lp \rceil$	214.3	455.4
	MPJ/Ibis	$223 + 290\lceil lp \rceil$	$381 + 256\lceil lp \rceil$	$5.536 + 22.03\lceil lp \rceil$	21.69	320.3
Reducescttr	MPICH-GM	$-3 + 13p$	$2 + 16p$	$9.326 + 10.81\lceil lp \rceil$	77.97	303.2
	mpiJava	$24 + 15p$	$18 + 19p$	$11.37 + 11.51\lceil lp \rceil$	60.37	277.6
	MPJ/Ibis	$13 + 76p$	$7 + 89p$	$13.91 + 17.83\lceil lp \rceil$	12.97	187.0
Scan	MPICH-GM	$13 + 4p$	$31 + 6p$	$-4.487 + 9.284\lfloor 2lp \rfloor$	194.8	357.7
	mpiJava	$50 + 6p$	$67 + 8p$	$-0.234 + 10.15\lfloor 2lp \rfloor$	102.7	296.9
	MPJ/Ibis	$-1 + 97p$	$9 + 112p$	$3.380 + 21.62p$	9.671	42.94

Table 2: SCI-single: analytical models and peak aggregated metrics ($lp = \log_2 p$)

Primitive	Library	$t_0(p)$ $\{\mu s\}$	$t_i(p)$ $\{\mu s\}$	$t_b(p)$ $\{ns/byte\}$	π_0^{pag} $\{KB/s\}$	Bw_{as}^{pag} $\{MB/s\}$
Send	ScaMPI	4	13	3.890	250.0	257.1
	SCI-MPICH	6	5	4.560	166.7	219.3
	mpiJava	10	11	3.924	100.0	254.8
	MPJ/Ibis	49	43	4.272	20.41	234.1
Barrier	ScaMPI	$7 + 0.4p$	N/A	N/A	686.2	N/A
	SCI-MPICH	$-2 + 9\lceil lp\rceil$	N/A	N/A	280.0	N/A
	mpiJava	$8 + 1.2p$	N/A	N/A	397.7	N/A
	MPJ/Ibis	$133 + 48p$	N/A	N/A	13.54	N/A
Broadcast	ScaMPI	$6\lceil lp\rceil$	$12 + 8\lceil lp\rceil$	$-0.093 + 4.099\lceil lp\rceil$	388.9	573.6
	SCI-MPICH	$6\lceil lp\rceil$	$17 + 7\lceil lp\rceil$	$3.403 + 2.987\lceil lp\rceil$	388.9	566.1
	mpiJava	$33 + 7\lceil lp\rceil$	$59 + 9\lceil lp\rceil$	$0.391 + 4.451\lceil lp\rceil$	129.6	509.3
	MPJ/Ibis	$-7 + 15p$	$-9 + 16p$	$0.720 + 4.870\lceil lp\rceil$	61.95	456.6
Scatter	ScaMPI	$-5 + 6p$	$2 + 8p$	$2.714 + 0.251\lceil lp\rceil$	71.43	252.4
	SCI-MPICH	$5 + 2p$	$19 + 5p$	$2.011 + 0.718\lceil lp\rceil$	57.69	217.6
	mpiJava	$27 + 6p$	$58 + 11p$	$2.443 + 0.394\lceil lp\rceil$	14.71	241.4
	MPJ/Ibis	$11p$	$-16 + 16p$	$2.412 + 0.511\lceil lp\rceil$	19.23	221.8
Gather	ScaMPI	$4 + p$	$18 + 2p$	$0.612 + 1.222\lceil lp\rceil$	93.75	272.6
	SCI-MPICH	$2 + 2p$	$22 + 3p$	$2.139 + 0.719\lceil lp\rceil$	83.33	209.7
	mpiJava	$36 + p$	$53 + 4p$	$1.411 + 0.989\lceil lp\rceil$	19.89	221.3
	MPJ/Ibis	$54 + p$	$6 + 2\lceil lp\rceil$	$1.333 + 0.970\lceil lp\rceil$	14.11	229.1
Allgather	ScaMPI	$-6 + 14\lceil lp\rceil$	$12 + 18\lceil lp\rceil$	$3.510 + 1.327\lceil lp\rceil$	218.8	1051
	SCI-MPICH	$-1 + 5p$	$13 + 9p$	$1.936 + 2.571\lceil lp\rceil$	201.9	816.1
	mpiJava	$23 + 16\lceil lp\rceil$	$49 + 22\lceil lp\rceil$	$2.963 + 1.603\lceil lp\rceil$	110.9	1013
	MPJ/Ibis	$32p$	$-15 + 37p$	$1.101 + 2.679\lceil lp\rceil$	30.76	861.8
Alltoall	ScaMPI	$-10 + 8p$	$3 + 11p$	$1.693 + 2.310\lceil lp\rceil$	166.7	811.8
	SCI-MPICH	$-6 + 9p$	$12 + 12p$	$2.412 + 2.230\lceil lp\rceil$	106.1	769.1
	mpiJava	$22 + 9p$	$39 + 14p$	$2.347 + 2.120\lceil lp\rceil$	74.47	804.0
	MPJ/Ibis	$92 + 307p$	$73 + 271p$	$1.408 + 2.658\lceil lp\rceil$	2.747	746.1
Reduce	ScaMPI	$1 + 6\lceil lp\rceil$	$7 + 9\lceil lp\rceil$	$9.834 + 1.761\lceil lp\rceil$	368.4	463.1
	SCI-MPICH	$7 + 2p$	$18 + 4p$	$-3.718 + 6.381\lceil lp\rceil$	304.3	453.8
	mpiJava	$13 + 8\lceil lp\rceil$	$24 + 11\lceil lp\rceil$	$9.681 + 1.911\lceil lp\rceil$	189.2	454.1
	MPJ/Ibis	$26 + 42\lceil lp\rceil$	$6 + 42\lceil lp\rceil$	$1.507 + 9.299\lceil lp\rceil$	46.05	238.1
Allreduce	ScaMPI	$-1 + 12\lceil lp\rceil$	$11 + 15\lceil lp\rceil$	$9.281 + 2.536\lceil lp\rceil$	400.0	828.9
	SCI-MPICH	$11 + 5p$	$14 + 6p$	$5.591 + 3.859\lceil lp\rceil$	274.5	815.5
	mpiJava	$7 + 15\lceil lp\rceil$	$26 + 18\lceil lp\rceil$	$8.819 + 3.048\lceil lp\rceil$	269.2	779.4
	MPJ/Ibis	$-60 + 258\lceil lp\rceil$	$39 + 165\lceil lp\rceil$	$0.666 + 15.49\lceil lp\rceil$	19.61	297.0
Reducescatter	ScaMPI	$-1 + 8p$	$17 + 10p$	$12.51 + 2.068\lceil lp\rceil$	125.0	420.8
	SCI-MPICH	$-6 + 9p$	$5 + 12p$	$9.138 + 2.345\lceil lp\rceil$	125.0	486.9
	mpiJava	$23 + 9p$	$39 + 12p$	$13.04 + 2.149\lceil lp\rceil$	82.89	404.1
	MPJ/Ibis	$42 + 25p$	$31 + 29p$	$4.267 + 10.05\lceil lp\rceil$	32.54	228.8
Scan	ScaMPI	$-9 + 6p$	$13 + 9p$	$-3.361 + 5.183p$	333.3	183.7
	SCI-MPICH	$-1 + 4p$	$10 + 10p$	$3.799 + 1.544\lfloor 2lp\rfloor$	225.8	701.8
	mpiJava	$19 + 7p$	$42 + 12p$	$-5.423 + 8.299p$	93.33	114.8
	MPJ/Ibis	$-62 + 39p$	$-77 + 43p$	$-5.650 + 8.989p$	62.50	105.6

Table 3: SCI-dual: analytical models and peak aggregated metrics ($lp = \log_2 p$)

Primitive	Library	$t_0(p)$	$t_i(p)$	$t_b(p)$	π_0^{pag}	Bw_{as}^{pag}
		$\{\mu s\}$	$\{\mu s\}$	$\{ns/byte\}$	$\{KB/s\}$	$\{MB/s\}$
Barrier	ScaMPI	$5 + 2\lceil lp \rceil$	N/A	N/A	1154	N/A
	SCI-MPICH	$-169 + 140\lceil lp \rceil$	N/A	N/A	38.36	N/A
	mpiJava	$11 + \lceil lp \rceil$	N/A	N/A	1000	N/A
	MPJ/Ibis	$204 + 42p$	N/A	N/A	17.12	N/A
Broadcast	ScaMPI	$-3 + 6\lceil lp \rceil$	$7 + 9\lceil lp \rceil$	$-0.605 + 4.297\lceil lp \rceil$	714.2	904.6
	SCI-MPICH	$-11 + 11\lceil lp \rceil$	$3 + 18\lceil lp \rceil$	$-0.531 + 4.919\lceil lp \rceil$	454.5	783.5
	mpiJava	$21 + 7\lceil lp \rceil$	$39 + 12\lceil lp \rceil$	$-0.629 + 4.371\lceil lp \rceil$	306.1	889.9
	MPJ/Ibis	$6 + 15p$	$2 + 15p$	$-2.096 + 4.887\lceil lp \rceil$	60.97	859.5
Scatter	ScaMPI	$-12 + 6p$	$3 + 9p$	$2.199 + 0.339\lceil lp \rceil$	62.50	272.1
	SCI-MPICH	$6 + 2p$	$21 + 7p$	$2.158 + 1.702\lceil lp \rceil$	53.57	134.8
	mpiJava	$17 + 6p$	$38 + 13p$	$2.833 + 0.212\lceil lp \rceil$	18.29	254.7
	MPJ/Ibis	$-1 + 12p$	$-21 + 17p$	$2.417 + 0.408\lceil lp \rceil$	15.96	240.3
Gather	ScaMPI	$7 + 2p$	$34 + 4p$	$0.921 + 0.949\lceil lp \rceil$	50.00	266.1
	SCI-MPICH	$-41 + 35p$	$-3 + 53p$	$0.941 + 1.778\lceil lp \rceil$	7.575	166.8
	mpiJava	$41 + p$	$51 + 6p$	$1.037 + 0.944\lceil lp \rceil$	17.85	256.4
	MPJ/Ibis	$83 + 3p$	$70 + 4\lceil lp \rceil$	$0.968 + 0.995\lceil lp \rceil$	8.177	253.5
Allgather	ScaMPI	$4 + 2p$	$24 + 4p$	$4.515 + 1.863\lceil lp \rceil$	442.7	1332
	SCI-MPICH	$55 + 28p$	$63 + 33p$	$11.42 + 3.688\lceil lp \rceil$	31.68	608.9
	mpiJava	$41 + 2p$	$49 + 4p$	$5.831 + 1.592\lceil lp \rceil$	218.3	1306
	MPJ/Ibis	$-51 + 50p$	$-105 + 55p$	$2.713 + 2.374\lceil lp \rceil$	20.13	1305
Alltoall	ScaMPI	$-24 + 14p$	$4 + 18p$	$0.369 + 4.499\lceil lp \rceil$	93.75	816.7
	SCI-MPICH	$-221 + 103p$	$-193 + 121p$	$-2.97 + 8.391\lceil lp \rceil$	15.71	490.3
	mpiJava	$8 + 14p$	$35 + 21p$	$1.190 + 4.331\lceil lp \rceil$	64.66	810.2
	MPJ/Ibis	$-57 + 377p$	$-84 + 315p$	$-3.012 + 5.481\lceil lp \rceil$	2.510	793.1
Reduce	ScaMPI	$9 + p$	$8 + 2p$	$6.519 + 3.352\lceil lp \rceil$	600.0	752.7
	SCI-MPICH	$38 + 23p$	$51 + 38p$	$8.017 + 3.695\lceil lp \rceil$	36.94	657.9
	mpiJava	$24 + p$	$38 + 3p$	$7.598 + 3.616\lceil lp \rceil$	375.0	679.9
	MPJ/Ibis	$74 + 38\lceil lp \rceil$	$1 + 48\lceil lp \rceil$	$1.748 + 8.176\lceil lp \rceil$	66.37	435.4
Allreduce	ScaMPI	$7 + 2p$	$9 + 4p$	$11.41 + 3.693\lceil lp \rceil$	769.2	1145
	SCI-MPICH	$198 + 71p$	$228 + 83p$	$-15.03 + 20.94\lceil lp \rceil$	22.48	436.4
	mpiJava	$29 + 2p$	$41 + 5p$	$11.04 + 4.177\lceil lp \rceil$	491.8	1081
	MPJ/Ibis	$-61 + 288\lceil lp \rceil$	$-3 + 192\lceil lp \rceil$	$-9.143 + 22.39\lceil lp \rceil$	27.50	373.1
Reducesctr	ScaMPI	$-10 + 9p$	$3 + 11p$	$10.48 + 3.248\lceil lp \rceil$	144.2	679.0
	SCI-MPICH	$-673 + 216p$	$-540 + 239p$	$7.711 + 3.62\lceil lp \rceil$	19.63	718.2
	mpiJava	$22 + 8p$	$41 + 14p$	$11.31 + 3.761\lceil lp \rceil$	106.2	604.7
	MPJ/Ibis	$62 + 24p$	$46 + 28p$	$3.563 + 9.245\lceil lp \rceil$	35.73	393.1
Scan	ScaMPI	$-5 + 4p$	$1 + 6p$	$-2.939 + 5.050p$	272.7	192.7
	SCI-MPICH	$-24 + 82p$	$17 + 87p$	$-0.726 + 2.015\lfloor 2lp \rfloor$	11.64	1604
	mpiJava	$16 + 5p$	$32 + 8p$	$-4.596 + 7.903p$	156.2	123.1
	MPJ/Ibis	$-85 + 49p$	$-33 + 48p$	$-11.68 + 8.462p$	27.03	135.3

Table 4: SCI-dual w/HT: analytical models and peak aggregated metrics ($lp = \log_2 p$)

Primitive	Library	$t_0(p)$ $\{\mu s\}$	$t_i(p)$ $\{\mu s\}$	$t_b(p)$ $\{ns/byte\}$	π_0^{pag} $\{KB/s\}$	Bw_{as}^{pag} $\{MB/s\}$
Barrier	ScaMPI	$3 + 2\lceil lp \rceil$	N/A	N/A	2067	N/A
	mpiJava	$8 + 4\lceil lp \rceil$	N/A	N/A	1937	N/A
	MPJ/Ibis	$331 + 32p$	N/A	N/A	22.88	N/A
Broadcast	ScaMPI	$-7 + 7\lceil lp \rceil$	$-3 + 11\lceil lp \rceil$	$3.210 + 4.480\lceil lp \rceil$	1107	1210
	mpiJava	$45 + 9\lceil lp \rceil$	$57 + 14\lceil lp \rceil$	$-1.097 + 5.719\lceil lp \rceil$	344.4	1127
	MPJ/Ibis	$11 + 15p$	$37 + 13p$	$-0.997 + 5.397\lceil lp \rceil$	63.14	1191
Scatter	ScaMPI	$-17 + 6p$	$3 + 8p$	$0.519 + 1.150\lceil lp \rceil$	28.22	220.4
	mpiJava	$18 + 8p$	$41 + 11p$	$1.63 + 0.937\lceil lp \rceil$	10.67	197.0
	MPJ/Ibis	$9 + 13p$	$-3 + 16p$	$0.553 + 1.295\lceil lp \rceil$	7.743	197.2
Gather	ScaMPI	$15 + 2p$	$83 + 5p$	$-1.403 + 2.017\lceil lp \rceil$	28.23	188.3
	mpiJava	$55 + 2p$	$131 + 9p$	$-1.031 + 2.053\lceil lp \rceil$	12.32	170.6
	MPJ/Ibis	$71 + 3p$	$-39 + 44\lceil lp \rceil$	$0.344 + 1.835\lceil lp \rceil$	9.211	149.6
Allgather	ScaMPI	$-1 + 3p$	$45 + 5p$	$10.23 + 1.987\lceil lp \rceil$	342.4	1585
	mpiJava	$74 + 2p$	$128 + 7p$	$9.648 + 2.238\lceil lp \rceil$	231.7	1534
	MPJ/Ibis	$-68 + 67p$	$-162 + 79p$	$5.645 + 3.320\lceil lp \rceil$	16.83	1437
Alltoall	ScaMPI	$-123 + 36p$	$-83 + 43p$	$-7.585 + 12.41\lceil lp \rceil$	42.42	569.2
	mpiJava	$-114 + 44p$	$-45 + 60p$	$-5.969 + 12.10\lceil lp \rceil$	29.41	568.5
	MPJ/Ibis	$-575 + 547p$	$-773 + 465p$	$-1.950 + 11.92\lceil lp \rceil$	1.842	537.7
Reduce	ScaMPI	$14 + p$	$31 + 2p$	$13.31 + 4.690\lceil lp \rceil$	673.9	843.3
	mpiJava	$47 + p$	$77 + 3p$	$12.91 + 5.796\lceil lp \rceil$	392.4	740.0
	MPJ/Ibis	$56 + 51\lceil lp \rceil$	$-11 + 49\lceil lp \rceil$	$1.486 + 10.93\lceil lp \rceil$	99.68	552.2
Allreduce	ScaMPI	$15 + p$	$33 + 2p$	$22.31 + 5.513\lceil lp \rceil$	1319	1243
	mpiJava	$51 + 2p$	$83 + 4p$	$20.70 + 7.131\lceil lp \rceil$	539.1	1100
	MPJ/Ibis	$-135 + 358\lceil lp \rceil$	$82 + 219\lceil lp \rceil$	$-133.3 + 62.74\lceil lp \rceil$	37.46	306.3
Reducesctr	ScaMPI	$7 + 8p$	$30 + 11p$	$18.72 + 4.798\lceil lp \rceil$	121.6	748.5
	mpiJava	$44 + 8p$	$79 + 15p$	$15.92 + 6.586\lceil lp \rceil$	106.6	654.4
	MPJ/Ibis	$81 + 27p$	$75 + 28p$	$6.588 + 11.09\lceil lp \rceil$	33.83	515.3
Scan	ScaMPI	$-3 + 5p$	$6 + 9p$	$-7.813 + 5.407p$	197.4	197.5
	mpiJava	$29 + 6p$	$41 + 11p$	$-7.864 + 9.645p$	140.3	103.1
	MPJ/Ibis	$-120 + 58p$	$-325 + 78p$	$-5.730 + 8.071p$	20.35	122.8

5 Analysis and Discussion of Performance Results

5.1 Point-to-Point Communication

Figure 2 shows experimentally measured (empty symbols) and estimated (filled symbols) latencies and bandwidths of the Send primitive as a function of the message length for the different networks. Bandwidth graphs are useful to compare long message performance, whereas latency graphs serve to compare short message performance (note that their scale is logarithmic).

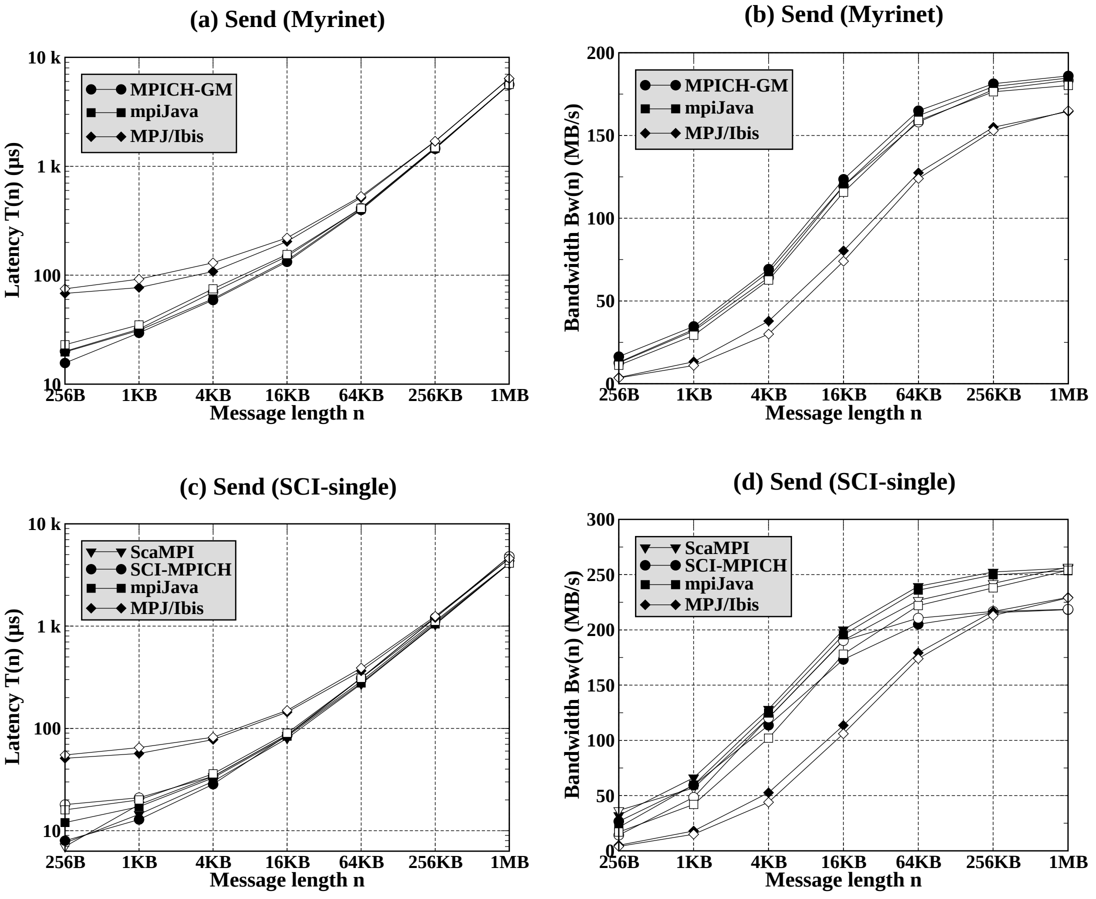


Figure 2: Measured and estimated latencies and bandwidths of Send

Regarding MPI C point-to-point primitives (see Tables 1-2), asymptotic bandwidths are 188 MB/s for MPICH-GM Send, and 257 MB/s for ScaMPI Send. In this case, the PCI bus is the main performance bottleneck as it limits the bandwidth to 264MB/s on the Myrinet cluster and to 528MB/s on the SCI cluster. Experimentally measured MPI C point-to-point startups, $9\mu s$ on Myrinet and $4\mu s$ on SCI, are very close to their theoretical values, $7\mu s$ and $1.46\mu s$, respectively. The different computational power of the nodes has a minor influence on these values.

Regarding the message-passing libraries, on the one hand, mpiJava obtains results quite similar to the underlying native message-passing library. On the other hand, MPJ/Ibis shows startups of $65\mu s$ on Myrinet and of $49\mu s$ on SCI, and values of t_b slightly higher (around 10%) than the native library values. This overhead, quite far from the theoretical values of the high-speed interconnects, especially for t_0, must be attributed to the messaging-protocol (around $40\mu s$ overhead for t_0). The underlying communication library, TCPIbis Sockets, shows $t_0 = 22\mu s$ on Myrinet, and $t_0 = 11\mu s$ on SCI, thanks to the use of high performance sockets libraries (Sockets-GM and SCI Sockets). Using IP emulation libraries TCPIbis obtains $t_0 = 196\mu s$ on Myrinet and $t_0 = 131\mu s$ on SCI. The benefits of using the emerging high performance Sockets libraries instead of IP emulations are clear on MPJ/Ibis. Using message-passing libraries based on RMI, such as CCJ [19] and JMPI [20], these benefits are relatively much less important as the procotocol overheads are much higher (from $0.5ms$ to $4ms$) [13].

5.2 Collective Communications

Measured and estimated bandwidths for some collective primitives are depicted in the graphs of Figures 3 and 4. The results were obtained using the maximum number of available processors for each cluster configuration (16 for Myrinet and SCI-dual, 8 for SCI-single and 32 for SCI-dual w/HT). Note that bandwidths are not aggregated, as they are computed simply by dividing n by $T(n,p)$. In many cases, the estimated values (filled symbols) are hidden by the measured values (empty symbols), which means a good modeling. As expected, the bandwidth of the mpiJava routines and the underlying MPI C implementations are very similar (mpiJava calls to native MPI have low overhead), and pure Java primitives show lower performance. In fact, MPJ/Ibis t_b is slightly higher than the native library value, and therefore, the derived performance metric, Bw_{as}^{pag}, presents slightly lower value.

A gap between short message performance between Myrinet and SCI can be observed. For instance, the 4KB MPI C Broadcast bandwidth is 3.3 times higher on SCI-single than on Myrinet (see Figures 3(a) and 3(c)). Similarly, the 4KB MPI C Reduce bandwidth is 2.8 times higher on SCI-single (see Figures 3(b) and 3(d)). A higher t_0 on Myrinet is the main cause of this lower performance. Regarding the different system configurations, it can be observed that the highest bandwidths are obtained by SCI-single (see Figures 3(c), 3(d), 4(c) and 4(d)), followed by SCI-dual, and finally, by Myrinet and SCI-dual w/HT, which obtain similar results. MPJ/Ibis shows lower performance for Alltoall, especially for short and medium messages (see Figures 4(b), 4(d), 4(f) and 4(h), and the metric π_0^{pag} in Tables 1–4).

5.3 Model-based Performance Optimization

Message-passing performance models have been used to identify inefficient communication primitives. From this process, it has been detected that ScaMPI and MPJ/Ibis Scan show non-optimal $O(p)$ complexities. Other implementations, e.g. MPJ/Ibis Allreduce, just show bad performance. To reduce the inefficiency, a primitive can be replaced by a more efficient equivalent combination of primitives. Examples of equivalences existing in message-passing libraries are: Broadcast=Scatter+Allgather (Van der Geijn algorithm [21]), Allgather=Gather+Broadcast, Reduce-scatter=Reduce+Scatterv and Allreduce=Reduce+Broadcast. The conditions for carrying out the replacement are actually obtained from the models: $T_{basic\_primitive}(n,p) > T_{primitive\#1}(n,p) + T_{primitive\#2}(n,p)$, where *basic_primitive* is

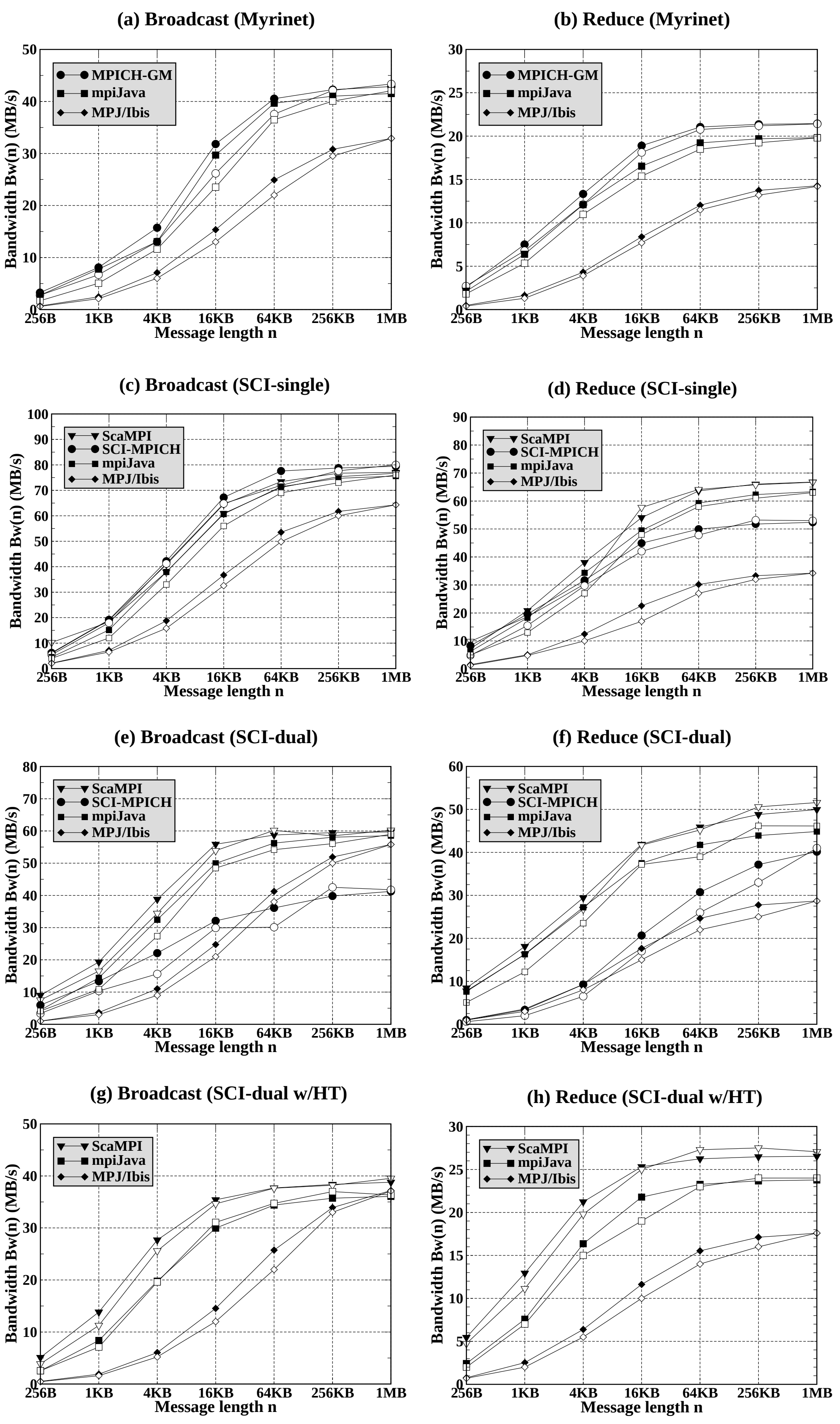


Figure 3: Measured and estimated bandwidths for Broadcast and Reduce

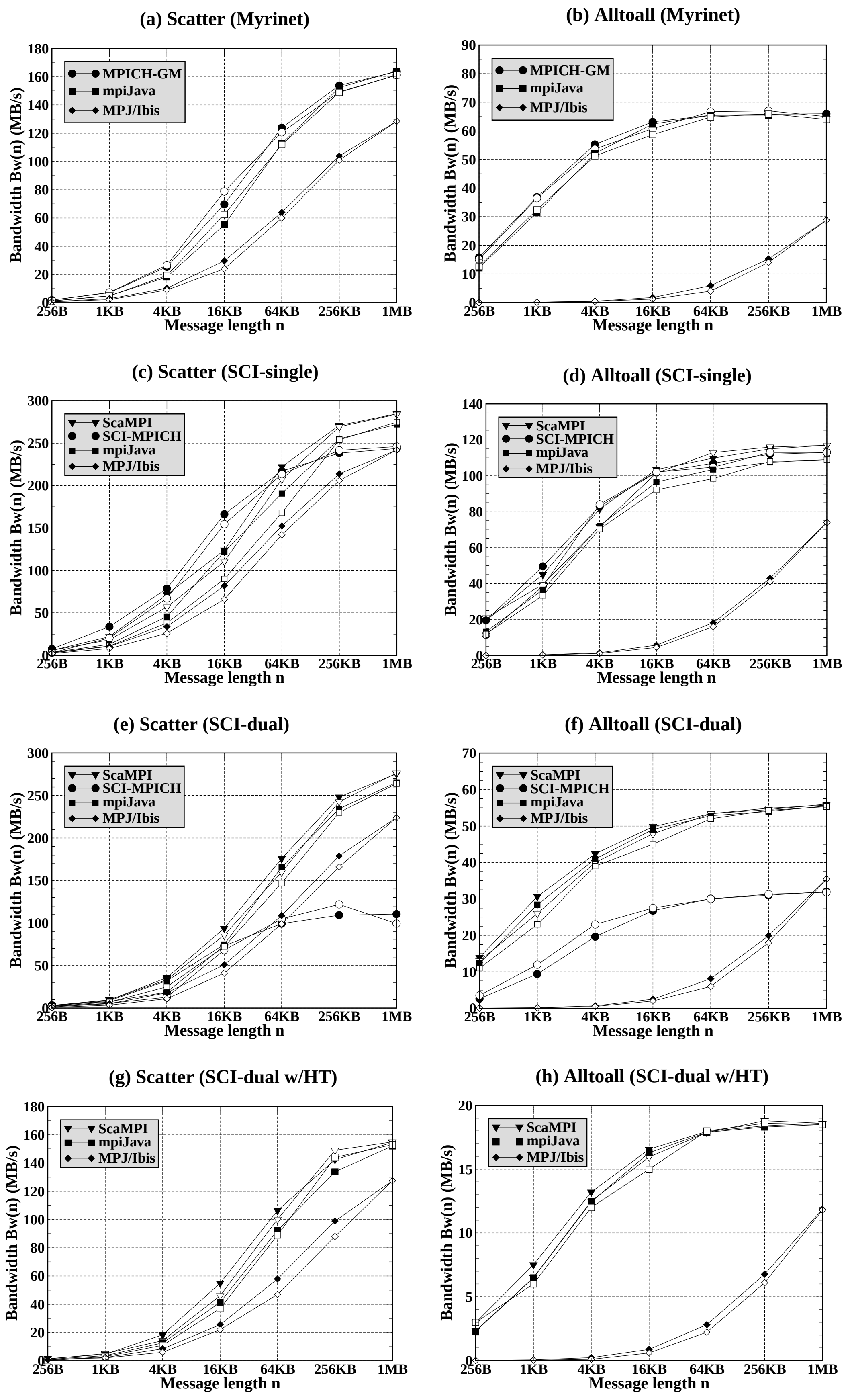


Figure 4: Measured and estimated bandwidths for Scatter and Alltoall

equivalent to *primitive#1+primitive#2*. These conditions are the parameter values of n and p for which this inequality is satisfied.

For illustrative purposes, some examples of latency reduction using this technique are presented in Table 5. Several conditions to replace communication primitives are shown, together with some examples that meet these conditions. The obtained latency reductions for these examples are shown in the last column. mpiJava examples have been omitted as this library performs similarly to the underlying native library. The Reduce-scatter primitive has also been omitted as it is implemented in MPICH, ScaMPI and MPJ/Ibis using a Reduce followed by a Scatterv. The equivalent combination of primitives: (1) can present lower startups than the original primitive (e.g. Allgather in Table 5), or (2) can show lower transfer times than the original primitive (e.g. Broadcast in Table 5), or (3) can present both situations (e.g. Allreduce in Table 5); in this case the substitution conditions are always met.

This model-based performance optimization can be easily automatized. By determining cross-over points between communication primitives and its equivalent combinations, the message-passing library can replace at runtime inefficient primitives by their equivalents. Related projects on automatic collective communication optimization [22, 23] use the P-LogP model, that operates in a lower level. Nevertheless, these papers present only Broadcast and Scatter optimizations, due to the complexity of their approximations. In fact, in order to determine the best communication pattern, the optimization procedure consists of finding out the best algorithm for each message size, and the best segment size to fragment the message. This procedure must be repeated for each number of processors taken into account, although it can be speeded up with the aid of the P-LogP model. The originality of our higher-level approach relies on its generality, as it works straightforwardly for every collective primitive, and on its simplicity, as the optimization procedure requires less phases than the related approaches.

6 Application-level Kernel Benchmarking

An application-level kernel benchmarking has been carried out in order to analyze the impact of message-passing overhead on the overall application performance. This benchmarking has also served to analyze the influence of message-passing overhead on multiple processor nodes (see Section 7). Both analyses are consistent with the predictions obtained from the models. This process has been carried out on the SCI cluster, and the selected benchmarks have been the MPJ application-level kernels from the Java Grande Forum (JGF) Benchmark Suite [24] and their corresponding MPI C versions. The kernels are, from higher to lower computation/communication ratio: Series, Crypt, SOR, Sparse and LUFact. For each of them there are three predetermined problem sizes: small (A), medium (B) and large (C). This benchmark suite is the only one that includes MPJ kernels, although there have been some attempts to develop MPJ NAS Parallel Benchmarks [25].

Figure 5 shows the speedups obtained from running LUFact and Series kernels using ScaMPI, mpiJava, MPJ/Ibis and MPJ Express on the SCI cluster. These kernels have been

Table 5: Parameter values for latency (**T**) reduction through primitive substitution

	Library	**Testbed**	**Parameter values** $\{(n,p)\}$	**Example (n,p)**	**T**
Broadcast	MPICH-GM	Myrinet	{(n>64KB,p=8),(n>78KB,p=16)}	n=256KB, p=16	↓ 20%
	ScaMPI	SCI-single	{(n>103KB,p=8)}	n=256KB, p=8	↓ 18%
		SCI-dual	{(n>128KB,p=16)}	n=256KB, p=16	↓ 13%
	MPJ/Ibis	Myrinet	{(n>273KB,p=8),(n>994KB,p=16)}	n=512KB, p=8	↓ 43%
		SCI-single	{(n>462KB,p=4),(n>202KB,p=8)}	n=1MB, p=8	↓ 53%
		SCI-dual	{(n>1173KB,p=16)}	n=2MB, p=16	↓ 9%
Allgather	MPICH-GM	Myrinet	{(n<256B,p=8),(n<2KB,p=16)}	n=256B, p=16	↓ 20%
	ScaMPI	SCI-single	{(n<128B,p=4),(n<256B,p=8)}	n=128B, p=8	↓ 43%
	MPJ/Ibis	Myrinet	{(n<25KB,p=8),(n<40KB,p=16)}	n=1KB, p=8	↓ 42%
		SCI-single	{(n<2KB,p=4),(n<7KB,p=8)}	n=1KB, p=8	↓ 35%
		SCI-dual	{(n>17KB,p=8),(n>45KB,p=16)}	n=1KB, p=16	↓ 50%
		SCI-w/HT	{(n>66KB,p=8),(n>109KB,p=16)}	n=1KB, p=16	↓ 61%
Allreduce	MPJ/Ibis	Myrinet	Replace always	n=1KB, p=8	↓ 39%
				n=256KB, p=8	↓ 18%
		SCI-single	Replace always	n=1KB, p=8	↓ 50%
				n=256KB, p=8	↓ 24%
		SCI-dual	Replace always	n=1KB, p=16	↓ 44%
				n=256KB, p=16	↓ 41%
		SCI-w/HT	Replace always	n=1KB, p=16	↓ 52%
				n=256KB, p=16	↓ 65%

selected as representatives of communication intensive applications (LUFact) and computation intensive applications (Series). Labels in the x-axis represent the kernel problem size (A,B,C) and the number of processes per node (1, 2 and 4; using SCI-single, SCI-dual and SCI-dual w/HT configurations, respectively). Regarding the speedup results, ScaMPI shows generally the best scalability; mpiJava presents slightly lower performance than ScaMPI; and MPJ/Ibis and MPJ Express results are slightly lower than mpiJava results. LUFact shows modest parallel efficiencies, and even slowdowns for size A, especially for A4, and also for size B with MPJ Express. Series as a whole presents higher parallel efficiencies (between 48% and 99%). Nevertheless, these efficiencies fall in two groups, modest parallel efficiencies for size C or using SCI-dual w/HT (labels C1, C2, A4, B4 and C4), and higher values for the remaining cases. Regarding the "pure" Java libraries, MPJ Express Series shows better performance than MPJ/Ibis Series, whereas MPJ/Ibis performs better for LUFact. These differences can be explained by the fact that MPJ/Ibis uses TCPIbis Sockets as communication technology, which has lower t_0 but higher t_b than Java NIO Sockets, base of MPJ Express. Thus, MPJ/Ibis performs better for applications with short message communication patterns, whereas MPJ Express shows better performance for medium and long message communication patterns.

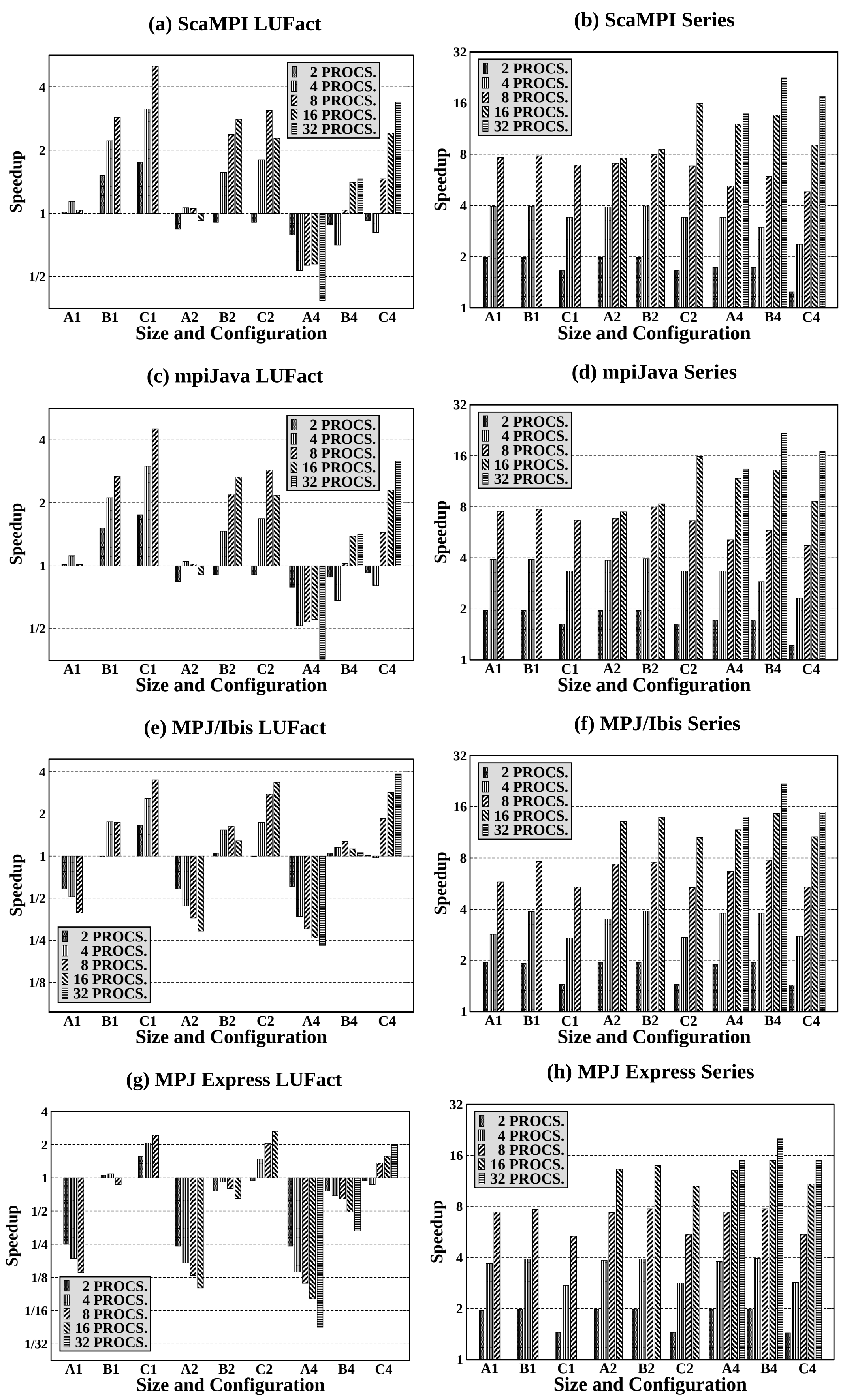


Figure 5: Speedups of selected Java Grande application-level kernels

Although Sparse, Crypt and SOR experimental results have also been analyzed, they have been omitted for conciseness and only the main conclusions are presented. Thus, on the one hand, Sparse results are slightly lower than LUFact speedups. On the other hand, Crypt and SOR results are similar to Series results, but showing lower speedups, especially for SOR.

From this benchmarking process, it has been observed that MPJ/Ibis and MPJ Express show parallel efficiencies comparable with native libraries performance. Only small, communication intensive applications show clearly poorer scalability.

7 Performance Analysis on Multiple Processor Nodes

The kernel benchmarking has also served to analyze the influence of message-passing overhead on multiple processor nodes, more specifically on dual nodes with and without hyperthreading. This analysis has been carried out on the SCI cluster using single, dual and dual w/HT configurations. ScaMPI and MPJ/Ibis have been selected as representative libraries of native and Java message-passing libraries, respectively.

7.1 Performance Analysis on Dual Processor Nodes

According to the graphs of Figure 5, LUFact speedups are higher using 1 process per node than using 2 processes (A1 speedups $>$ A2 speedups, B1 $>$ B2 , C1 $>$ C2), whereas Series speedups remain similar. In order to quantify the influence of using 2 processes per node instead of 1 process per node a metric has been derived. This metric consists of the ratio $T_{SCI-single}(p)/T_{SCI-dual}(p)$ for p processes, where p nodes are used on SCI-single and $p/2$ nodes on SCI-dual. A ratio higher than 1 means that the kernel benefits from running p processes on SCI-dual, instead of running on SCI-single. From Table 6 it can be observed that LUFact, Sparse and SOR, communication intensive kernels, do not benefit from using 2 processes per node, whereas Series and Crypt, computation intensive kernels, can slightly benefit from this. The reason is that with 2 processes per node each process has available approximately half of the resources of the node, instead of the resources of the whole node (as it happens with 1 process per node). As communication intensive kernels need more resources for communications than computation intensive kernels (the message-passing libraries use additional buffers and threads when communicating), the performance benefits of intra-node communication do not make up for the reduction of available resources for inter-node communication.

Once running p processes on $p/2$ nodes instead of on p nodes seems to be little beneficial, another interesting comparison is running the kernel on nd nodes assigning 1 process per node against running the kernel on nd nodes with 2 processes per node. The associated metric is the ratio $T_{SCI-single}(nd)/T_{SCI-dual}(2 \times nd)$. A ratio higher than 1 means that the kernel benefits from running 2 processes per node instead of running only 1 for a fixed number of nodes nd. From the discussion in Subsection 4.2, both t_0 and t_b are higher on SCI-dual than on SCI-single. Moreover, the communication overhead is higher for $2 \times nd$ processes instead of for nd processes. Thus, clearly the communication cost is higher for $T_{SCI-dual}(2 \times nd)$

Table 6: Ratio $T_{SCI-single}(p)/T_{SCI-dual}(p)$

		Small Size (A)		**Medium Size (B)**		**Large Size (C)**	
		$p = 4$	$p = 8$	$p = 4$	$p = 8$	$p = 4$	$p = 8$
ScaMPI	LUFact	0.93	**1.01**	0.69	0.82	0.56	0.64
	Series	0.95	0.88	0.99	**1.01**	**1.00**	0.99
	SOR	0.70	0.77	0.63	0.70	0.59	0.68
	Sparse	0.56	0.51	0.47	0.53	0.46	0.50
	Crypt	**1.02**	**1.01**	**1.01**	**1.00**	**1.00**	0.98
MPJ/Ibis	LUFact	0.86	0.91	0.88	0.92	0.68	0.79
	Series	**1.22**	**1.27**	**1.00**	0.99	**1.00**	**1.00**
	SOR	0.90	**1.03**	0.85	**1.04**	0.76	0.91
	Sparse	0.89	0.81	0.94	0.83	0.86	0.79
	Crypt	0.98	**1.00**	**1.02**	**1.02**	**1.00**	**1.04**

Table 7: Ratio $T_{SCI-single}(nd)/T_{SCI-dual}(2 \times nd)$

		Small Size (A)		**Medium Size (B)**		**Large Size (C)**	
		$4Nodes$	$8Nodes$	$4Nodes$	$8Nodes$	$4Nodes$	$8Nodes$
ScaMPI	LUFact	0.92	0.88	**1.03**	0.99	0.97	0.97
	Series	**1.67**	0.95	**1.99**	**1.96**	**1.95**	**1.92**
	SOR	**1.02**	**1.02**	0.99	0.99	0.98	0.98
	Sparse	0.62	0.65	0.71	0.68	0.72	0.69
	Crypt	**1.66**	**1.34**	**1.78**	**1.62**	**1.78**	**1.72**
MPJ/Ibis	LUFact	0.70	0.75	0.93	0.74	**1.08**	0.95
	Series	**1.58**	**1.25**	**1.95**	**1.82**	**1.97**	**1.96**
	SOR	**1.01**	0.82	**1.10**	0.94	**1.08**	0.96
	Sparse	0.58	0.61	0.63	0.64	0.67	0.64
	Crypt	**1.65**	**1.44**	**1.87**	**1.76**	**1.89**	**1.84**

Table 8: Ratio $T_{SCI-dual}(2 \times nd)/T_{SCI-dual\ w/HT}(4 \times nd)$

		Small Size (A)		**Medium Size (B)**		**Large Size (C)**	
		$2Nodes$	$8Nodes$	$2Nodes$	$8Nodes$	$2Nodes$	$8Nodes$
ScaMPI	LUFact	0.61	0.39	0.78	0.55	0.88	0.72
	Series	**1.30**	**1.23**	**1.24**	**1.24**	**1.36**	**1.24**
	SOR	0.80	0.46	0.87	0.64	0.89	0.52
	Sparse	0.54	0.53	0.74	0.63	0.88	0.65
	Crypt	0.94	0.72	**1.22**	0.90	**1.24**	0.95
MPJ/Ibis	LUFact	0.46	0.49	0.60	0.48	0.70	0.69
	Series	**1.27**	**1.18**	**1.30**	**1.29**	**1.29**	**1.30**
	SOR	0.70	0.49	0.70	0.61	0.75	0.62
	Sparse	0.44	0.39	0.44	0.39	0.49	0.43
	Crypt	**1.14**	**1.07**	**1.22**	**1.18**	**1.27**	**1.22**

than for $T_{SCI-single}(nd)$. Nevertheless, the workload for each of the $2 \times nd$ processes on SCI-dual is approximately half of the workload for each of the nd processes on SCI-single. Therefore, ratios slightly below 2 can be predicted for computation intensive kernels (Series and Crypt), whereas more modest ratios, even significant slowdowns, can be predicted for communication intensive kernels (LUFact, Sparse and SOR). Table 7 presents the obtained ratios, that are in tune with these predictions.

7.2 Performance Analysis on Hyperthreaded Dual Processor Nodes

The influence of enabling the hyperthreading has not been taken into account in the previous analyses. This influence can be characterized by the ratio of latency from running the kernels on nd SCI-dual nodes against the runtime on nd SCI-dual w/HT nodes. Thus, the metric is $T_{SCI-dual}(2 \times nd)/T_{SCI-dual\ w/HT}(4 \times nd)$, where the number of processes is $2 \times nd$ on SCI-dual and $4 \times nd$ on SCI-dual w/HT. A ratio higher than 1 means that the kernel benefits from enabling hyperthreading, for a fixed number of nodes nd. From Subsection 4.2, it can be predicted that both t_0 and t_b are higher on SCI-dual w/HT than on SCI-dual. From Subsection 4.1 it can be obtained that the computational performance should be slightly higher. Table 8 shows the obtained ratios, that are in tune with these predictions. Thus, computation intensive kernels (Series and Crypt) benefit from enabling hyperthreading (up to a 36% performance increase), whereas communication intensive kernels (LUFact, Sparse and SOR) reduce their performance, especially on 8 nodes.

It has been observed that representative message-passing implementations do not benefit from systems with multiple processor nodes. A solution could be the use of multithreading instead of interprocess communication for handling intra-node communications. The development of shared memory communication protocols for intra-node communications and its combination with current inter-node protocols would achieve higher performance. Nevertheless, the message-passing library must implement thread-safe communication mechanisms. This issue is of special importance with the advent of multicore systems. Several related projects, e.g. USFMPI [26] and pCoR [27], propose to integrate multithreading and message-passing communications.

8 Conclusions

The characterization of the message-passing communication overhead on high-speed clusters is extremely important. Message-passing performance is critical for the overall system scalability and performance. Representative native MPI (MPICH-GM, ScaMPI and SCI-MPICH) and Java message-passing libraries (mpiJava, MPJ/Ibis and MPJ Express) have been selected for performance modeling and evaluation. For this purpose, a more accurate message-passing communication model, together with a message-passing micro-benchmark suite to derive these models, have been proposed. The predictions obtained by this model have been validated against experimental results obtaining better estimates than preceding models. The estimates have shown only a 7% average absolute relative error. Moreover, performance metrics derived from the models have been used to evaluate message-passing primitives implementations and their performance on high-speed clusters. These models have also served to identify inefficient communication primitives. To solve these inefficiencies, some primitives can be replaced by a more efficient equivalent combination of primitives. This process has obtained important latency reductions and can be easily automatized.

From the analysis of message-passing performance, it can be concluded that native libraries and mpiJava benefit from the low startup and high bandwidth of the high-speed interconnects. Nevertheless, these libraries are not portable. MPJ/Ibis and MPJ Express overcome this issue, but this involves an important added overhead.

Besides the message-passing performance analysis on high-speed interconnects, it has been carried out a kernel benchmarking. This process has been performed in order to analyze the influence of message-passing overhead and the use of multiple processor nodes on the overall application performance. The main conclusion is that message-passing implementations, especially "pure" Java libraries, do not take much advantage of these systems.

Finally, this work intends to provide parallel programmers and library developers with guidelines for efficiently exploiting high-speed cluster interconnects and multiple processor nodes. The design of low-level communication middleware that increases Java performance on high-speed clusters, where far less research has been done, is the goal of our current work.

Acknowledgments

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 PGIDIT06PXIB105228PR. We gratefully thank CESGA (Galician Supercomputing Center, Santiago de Compostela, Spain) for providing access to the Myrinet cluster.

References

[1] M. Lobosco, C. L. de Amorim, and O. Loques. Java for High-Performance Network-Based Computing: a Survey. *Concurrency and Computation: Practice and Experience*, 14(1):1–31, 2002.

[2] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian and T. von Eicken. LogP: Towards a Realistic Model of Parallel Computation. In *Proc. 4th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP'93)*, pages 1 – 12, San Diego, CA, 1993.

[3] A. Alexandrov, M. F. Ionescu, K. E. Schuser and C. Scheiman. LogGP: Incorporating Long Messages into the LogP Model – one Step Closer Towards a Realistic Model for Parallel Computation. *Journal of Parallel and Distributed Computing*, 44(1):71–79, 1997.

[4] M. I. Frank, A. Agarwal and M. K. Vernon. LoPC: Modeling Contention in Parallel Algorithms. In *Proc. 6th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP'97)*, pages 276 – 287, Las Vegas, NV, 1997.

[5] C. A. Moritz and M. Frank. LoGPC: Modeling Network Contention in Message-Passing Programs. *IEEE Transactions on Parallel and Distributed Systems*, 12(4):404–415, 2001.

[6] F. Ino, N. Fujimoto and K. Hagihara. LogGPS: A Parallel Computational Model for Synchronization Analysis. In *Proc. 8th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP'01)*, pages 133 – 142, Snowbird, UT, 2001.

[7] Y. Touyama and S. Horiguchi. Performance Evaluation of Practical Parallel Computation Model LogPQ. In *Proc. 1999 IEEE Int. Symp. on Parallel Architectures, Algorithms and Networks (ISPAN'99)*, pages 215 – 221, Fremantle, Australia, 1999.

[8] K. W. Cameron and X.-H. Sun. Quantifying Locality Effect in Data Access Delay: Memory logP. In *Proc. 17th Int. Parallel & Distributed Processing Symp. (IPDPS'03)*, page 48 (8 pages), Nice, France, 2003.

[9] T. Kielmann, H. Bal, S. Gorlatch, K. Verstoep, and R. Hofman. Network Performance-aware Collective Communication for Clustered Wide Area Systems. *Parallel Computing*, 27(11):1431–1456, 2001.

[10] K. W. Cameron and R. Ge. Predicting and Evaluating Distributed Communication Performance. In *Proc. 16th Int. Conf. on High Performance Computing and Communications (SC'04)*, page 43 (15 pages), Pittsburgh, PA, 2004.

[11] B. A. Vianna, A. A. Fonseca, N. T. Moura, L. T. Mendes, J. A. Silva, C. Boeres and V. E. F. Rebello. A Tool for the Design and Evaluation of Hybrid Scheduling Algorithms for Computational Grids. In *Proc. 2nd ACM Workshop on Middleware for Grid Computing (MGC'04)*, pages 41 – 46, Toronto, Canada, 2004.

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

[13] 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 Int. Conf. on Cluster Computing (CLUSTER'03)*, pages 118–126, Hong Kong, China, 2003.

[14] M. Baker and B. Carpenter. MPJ: a Proposed Java Message Passing API and Environment for High Performance Computing. In *Proc. 2nd Int. Workshop on Java for Parallel and Distributed Computing (JAVAPDC'00), LNCS 1800, Springer-Verlag*, pages 552–559, Cancun, Mexico, 2000.

[15] M. Baker, B. Carpenter, G. Fox, S. Ko, and S. Lim. mpiJava: an Object-Oriented Java Interface to MPI. In *Proc. 1st Int. Workshop on Java for Parallel and Distributed Computing (JAVAPDC'99), LNCS 1586, Springer-Verlag*, pages 748–762, San Juan, Puerto Rico, 1999.

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

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

[18] R. V. van 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.

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

[20] S. Morin, I. Koren, and C. M. Krishna. JMPI: Implementing the Message Passing Standard in Java. In *Proc. 4th Int. Workshop on Java for Parallel and Distributed Computing (JAVAPDC'02)*, pages 118–123, Fort Lauderdale, FL, 2002.

[21] R. Thakur, R. Rabenseifner, and W. Gropp. Optimization of Collective Communication Operations in MPICH. *Int. Journal of High Performance Computing Applications*, 19(1):49–66, 2005.

[22] S. S. Vadhiyar, G. E. Fagg, and J. J. Dongarra. Towards an Accurate Model for Collective Communications. *Int. Journal of High Performance Computing Applications*, 18(1):159–167, 2004.

[23] L. A. Barchet-Estefanel and G. Mounie. Fast Tuning of Intra-cluster Collective Communications. In *Proc. 11th European PVM/MPI Users' Group Meeting (EuroPVM/MPI'04), LNCS 3241, Springer-Verlag*, pages 28 – 35, Budapest, Hungary, 2004.

[24] L. A. Smith, J. M. Bull, and J. Obdržálek. A Parallel Java Grande Benchmark Suite. In *Proc. of the 2001 ACM/IEEE Conf. on Supercomputing (SC'01)*, page 8 (10 pages), Denver, CO, 2001.

[25] J. A. Mathew, Paul D. Coddington, and Kenneth A. Hawick. Analysis and Development of Java Grande Benchmarks. In *Proc. of the ACM 1999 Conference on Java Grande (JAVA'99)*, pages 72–80, San Francisco, CA, 1999.

[26] S. G. Caglar, G. D. Benson, Q. Huang, and C.-W. Chu. USFMPI: A Multi-threaded Implementation of MPI for Linux Clusters. In *Proc. 15th Int. Conf. on Parallel and Distributed Computing and Systems (PDCS'03)*, pages 674 – 680, Marina del Rey, CA, 2003.

[27] A. A. G. Alves, A. Pina, J. L. P. Exposto, and J. Rufino. Scalable Multithreading in a Low Latency Myrinet Cluster. In *Proc. 5th Int. Conf. on High Performance Computing in Computational Sciences (VECPAR'02), LNCS 2565, Springer-Verlag*, pages 579–592, Porto, Portugal, 2002.