Characterization of Message-Passing Overhead on the AP3000 Multicomputer

Juan Touriño and Ramón Doallo
Department of Electronics and Systems, University of A Coruña
Campus de Elviña, s/n, 15071 A Coruña, Spain
juan@udc.es, doallo@udc.es

Abstract

The performance of the communication primitives of parallel computers is critical for the overall system performance. The characterization of the communication overhead is very important to estimate the global performance of parallel applications and to detect possible bottlenecks. In this work, we evaluate, model and compare the performance of the message-passing libraries provided by the Fujitsu AP3000 multicomputer: MPI/AP, PVM/AP and APlib. Our aim is to fairly characterize the communication primitives using general models and performance metrics.

1. Introduction

The performance of the communication primitives of a parallel computer does not only depend on the underlying hardware, but also on their implementation. Users do not know the quality of the message-passing implementations and they can find that the performance of their parallel applications makes worse in other machine or using other message-passing library. We have used low-level tests to study basic communication primitives on the AP3000. Our aim is to estimate communication overheads with simple expressions, which can help AP application developers to design or migrate parallel programs more efficiently.

We have used the AP3000 to develop a parallel algorithm in the field of fluid mechanics [1]. The tuning of the communication routines of our algorithm is critical to achieve good performance. We predict communication costs using simple models, which can easily serve for extracting a set of rules or practices to select the appropriate primitive. There are related works that report message-passing performance on other machines, such as Cray T3E and SGI Origin 2000 [9], IBM SP2 [7] [12] or large clusters of workstations [4].

The next section introduces the target machine and the programming environment, the performance models and their associated metrics, as well as a brief description of our experiments. In Section 3 point-to-point communications (from now on, p-t-p) are analyzed. Section 4 presents modeling results for collective communications, mainly focusing on MPI primitives. Section 5 describes an example of the practical application of our work to a real program. Finally, conclusions are discussed in Section 6.

2. AP3000 message-passing environment

The Fujitsu AP3000 [6] is a distributed-memory multiprocessor consisting of multiple nodes (4-1024) connected via a high-speed network, AP-Net, in a 2D-torus topology. The AP-Net transfers 16-bit data in parallel for a throughput of 200 Mbytes/s (bidirectional).

There are two message-passing modes in the AP3000: *ult* (user-level transfer) and *slt* (system-level transfer). The *ult* mode provides dedicated use of the allocated processors. Besides, the message communication can be activated without any help from the OS and there is direct access to the message controller, the interface hardware of the processor to the AP-Net. The *slt* mode uses the TCP/IP protocol, which results in a higher overhead. It allows to run many programs at one time in a processor partition.

Sitsky and Mackerras proposed in [10] a high-performance message-passing library for the AP3000, LWSLT (Light-Weight SLT), as basic communication layer in an MPI implementation for the AP3000. The name indicates a multiuser communication library without the heavy overhead of the *slt* mode by using the AP-Net directly. The goal is to obtain a robust library with low startup overhead for small messages and high bandwidths for large messages, taking into account the underlying communication hardware of the AP3000. It would involve the use of two different protocols for small and large messages, respectively. Tan et al. [11] were developing EPVM (Enhanced PVM), a PVM implementation for the AP3000. It uses the AP-Net and allows heterogeneous parallel computing, dynamic task spawning and MIMD programming style (PVM/AP and MPI/AP only allow the SPMD paradigm). Nevertheless, it does not achieve the PVM/AP communication speed.

APlib is a proprietary library that provides a programming and execution environment same as the AP1000 multicomputer [5], the predecessor of the AP3000.

2.1. Communication models

In p-to-p communication, 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). Distance is not a factor when considering latency between any 2 processors, due to the wormhole routing on the AP3000. We have experimentally checked that latencies vary slightly depending on the processor location. Hockney [3] proposed the following model to characterize p-t-p communications:

$$T(n) = \frac{n_{\frac{1}{2}} + n}{Bw_{as}} \qquad (1)$$

where Bw_{as} ($= 1/t_b$) is the asymptotic bandwidth, that is, the maximum throughput achievable when n approaches infinity and $n_{\frac{1}{2}}$ ($= t_s/t_b$) is the half-peak length, the message length required to obtain half of Bw_{as}. Bw_{as} shows long-message performance; the Hockney's model also defines the specific performance $\pi_s = 1/T(0) = 1/t_s$, to characterize short-message performance.

Collective communications among p processors can be characterized by the model proposed by Xu and Hwang [12], which is a generalization of the p-to-p model:

$$T(n,p) = t_s(p) + \frac{n}{Bw_{as}(p)} \qquad (2)$$

$Bw_{as}(p)$ can also be expressed as $1/t_b(p)$. The half-peak length for collective communications is defined as $n_{\frac{1}{2}}(p) = t_s(p)/t_b(p)$. Similarly, we define $\pi_s(p) = 1/t_s(p)$.

In order to get more information about AP3000 collective primitives, an additional metric is derived from the model of Eq. 2: the aggregated asymptotic bandwidth Bw_{as}^{ag}, which is the ratio of the total number of bytes transfered in the collective operation and the time required to perform the operation, as $n \to \infty$. For a broadcast, scatter, gather or reduction, $Bw_{as}^{ag}(p) = (p-1)Bw_{as}(p)$, but it is $p(p-1)Bw_{as}(p)$ for a total exchange operation. Similarly, we have derived a new metric from the aggregated concept, the aggregated specific performance: $\pi_s^{ag}(p) = Bw_{as}^{ag}(p)/n_{\frac{1}{2}}^{ag}(p) = (p-1)\pi_s(p)$. It shows the performance of a collective operation for short messages. The aggregated half-peak performance $n_{\frac{1}{2}}^{ag}(p)$ can also be defined as the message length that achieves an aggregated bandwidth $Bw_{as}^{ag}(p)/2$. It can be easily derived that $n_{\frac{1}{2}}^{ag}(p) = n_{\frac{1}{2}}(p)$.

We have also defined peak metrics: the peak aggregated bandwidth $Bw_{as}^{pag} = \max_{2 \leq p \leq p^{max}} Bw_{as}^{ag}(p)$, the peak aggregated specific performance $\pi_s^{pag} = \max_{2 \leq p \leq p^{max}} \pi_s^{ag}(p)$ and the minimum aggregated half-peak length $n_{\frac{1}{2}}^{mag} = \min_{2 \leq p \leq p^{max}} n_{\frac{1}{2}}^{ag}(p)$, being p^{max} the maximum p available.

2.2. Experimental conditions

The configuration of our parallel machine consists of 20 processors (UltraSparc-II at 300 Mhz), but only a maximum of a 12-processor partition can be accessed to execute parallel programs. The tests were executed in *ult* mode for different message sizes (from 0 bytes to 1 Mbyte) and, for collective communications, using different number of processors (2-12). In APlib we have used messages of up to 768 Kbytes, because the machine fails for 1 Mbyte messages. Timing overhead, cache warm-up effects and timing outliers were taken into account to avoid distortions in the latency measurements. The ping-pong test was used to model p-t-p communications; t_s and t_b were derived from a least-squares fit of latency (T) against message length (n). Regarding collective communications, as each test consisted of hundreds of iterations, 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. Nupairoj and Ni proposed in [8] a method (the Collective Communication Flow Model) to measure the performance of collective communications accurately.

3. Modeling point-to-point primitives

Only blocking p-to-p communications were considered. Initially, the primitives under evaluation were: `MPI_Send/MPI_Recv`, `pvm_psend/pvm_precv` and `l_asend/l_arecv` (APlib). Table 1 shows the estimated parameters and metrics.

As can be observed, Bw_{as} is the same for the three primitives, which results into a similar behavior of the libraries for large messages: latencies tend to be equal as the message length increases. The main difference lies in short messages, because there are appreciable differences in the startup times. APlib has the lowest startup time and MPI the highest. Consequently, the best $n_{\frac{1}{2}}$ is achieved by APlib.

Besides the standard MPI p-t-p primitive, we have also modeled the buffered send `MPI_Bsend` (the programmer allocates a buffer into which data can be placed until it is delivered), the synchronous send `MPI_Ssend` (the send does not return until the destination begins to receive the message), and the ready send `MPI_Rsend` (the sending processor assumes that the matching receive has already been posted; so, communication performance can be improved). According to the results of Table 1, the buffered send involves a considerable overhead (both startup and transfer time); the synchronous send also incurs a higher latency, but

Table 1. Point-to-point communication parameters and metrics.

	MPI/AP				PVM/AP		APlib	
	Send	Bsend	Ssend	Rsend	psend	send	l_asend	xy_send
t_s (μs)	69	84	75	69	53	50	46	46
t_b (μs)	0.0162	0.0213	0.0171	0.0162	0.0162	0.0263	0.0162	0.0162
π_s (Kbytes/s)	14.15	11.63	13.02	14.15	18.43	19.53	21.23	21.23
Bw_{as} (Mbytes/s)	58.87	44.77	55.77	58.87	58.87	36.26	58.87	58.57
$n_{\frac{1}{2}}$ (bytes)	4260	3944	4386	4260	3272	1902	2840	2840

Table 2. Broadcast parameters and metrics ($k_1 = 10^6/2^{10}, k_2 = 10^6/2^{20}$).

	MPI/AP	PVM/AP	APlib
$t_s(p)$ (μs)	69log$_2$p	22p	46+25(p-2)
$t_b(p)$ (μs)	0.0162log$_2$p	0.0110p	0.0162+0.0110(p-2)
$\pi_s(p)$ (Kbytes/s)	k$_1$/(69 log$_2$p)	k$_1$/(22p)	k$_1$/(25p-4)
$\pi_s^{ag}(p)$ (Kbytes/s)	k$_1$(p-1)/(69 log$_2$p)	k$_1$(p-1)/(22p)	k$_1$(p-1)/(25p-4)
π_s^{pag} (Kbytes/s)	43.43	40.69	36.29
$Bw_{as}(p)$ (Mbytes/s)	k$_2$61.73/log$_2$p	k$_2$90.91/p	k$_2$/(0.0110p-0.0058)
$Bw_{as}^{ag}(p)$ (Mbytes/s)	k$_2$61.73(p-1)/log$_2$p	k$_2$90.91(p-1)/p	k$_2$(p-1)/(0.0110p-0.0058)
Bw_{as}^{pag} (Mbytes/s)	180.63	79.47	83.13
$n_{\frac{1}{2}}(p) = n_{\frac{1}{2}}^{ag}(p)$(bytes)	4260	2000	(25p-4)/(0.0110p-0.0058)
$n_{\frac{1}{2}}^{mag}$(bytes)	4260	2000	2346

the parameters are closer to the standard mode. The ready send works as the normal send, that is, the protocol between sending and receiving processors was not optimized in the AP implementation of `MPI_Rsend`. It is clear that the standard send is the best option.

`pvm_send/pvm_recv` were also tested (see Table 1). The estimated startup is slightly lower than `pvm_psend/ pvm_precv`, $t_s \approx 50\mu$s, but the transfer time increases excessively, $t_b = 0.0263\mu$s, which results in a poor bandwidth, $Bw_{as} = 36.26$ Mbytes/s. This is due to packing and unpacking operations (`pvm_pk*` and `pvm_upk*` routines).

The APlib primitives `xy_send/xy_recv` were low latency p-to-p communications that used the torus network (T-net) [5] of the AP1000; `xy_recv` reads the received message directly in a specified variable (in `l_arecv`, an additional `readmsg` routine is necessary to read the received message). These routines have exactly the same latencies as `l_asend/l_arecv`, as can be observed in Table 1 (it seems that they have the same implementation).

4. Modeling collective primitives

4.1. Broadcast

In a broadcast a source processor (root) sends the same data to a group of destination processors. The routines under consideration are: `MPI_Bcast`, `pvm_mcast` and `cbroad` (APlib). We have not used the `pvm_bcast` routine because the PVM/AP group operations are not reliable, as we will comment in Section 4.3.

`MPI_Bcast` is called by all the processors involved in the broadcast. In PVM, `pvm_mcast` (and `pvm_bcast`) is only called by the root processor, which broadcasts a message previously stored in the active send buffer. The other processors receive the broadcast message using a receive routine (`pvm_precv` in our experiment). Another difference is that the MPI standard establishes that the root processor broadcasts the message to all the processors of the group, itself included; in PVM, the message is not sent to the caller (even if listed in the array of destination processors of `pvm_mcast` or if the caller is in the destination group of `pvm_bcast`). The APlib `cbroad` routine works as `pvm_mcast`. There is another APlib broadcast routine: `xy_brd`. As `MPI_Bcast`, it is called by all the processors. We have found that `cbroad` and `xy_brd` have the same latencies in the AP3000.

The fitting of the components of Eq. 2 and the performance metrics are shown in Table 2. The constants k_1 and k_2 are introduced to consider the base 2 vs base 10 discrepancy. Note that, for 2 processors, in MPI and APlib the values of the model's parameters are the same as the ones of the p-to-p model (see Table 1). Therefore, the p-to-p model in MPI and APlib can be considered as a particular case of the broadcast model for *p*=2. That is not the case for PVM,

Table 3. Parameters for MPI collective data movement primitives.

	Scatter	Gather	Allgather	Total exchange
$t_s(p)$ (μs)	58+53p	1/(0.0135-10^{-3}2.96$\log_2$p)	95p-24	89p
$t_b(p)$ (μs)	0.0209($\log_2$p)$^{-0.3830}$	0.0252($\log_2$p)$^{-0.1473}$	0.0239+10^{-3}2.79$\log_2$p	0.0291+10^{-4}8.84p

where latencies are higher for a 2-processor broadcast than in a p-to-p communication, although they are slightly better for short messages due to a lower startup time.

As can be observed, latency has a complexity $O(log_2p)$ in MPI and, therefore, the asymptotic bandwidth is $O(1/log_2p)$. This fact reveals that the broadcast in MPI was implemented using a binomial tree-structured approach (top-down traversing of the tree). Bernaschi et al. proposed in [2] an MPI broadcast implementation based on quasi-optimal spanning trees, instead of binomial trees. In PVM, latency is $O(p)$ (and $Bw_{as}(p)$ is $O(1/p)$). This is an inefficient PVM implementation. It seems that `pvm_mcast` was implemented as a sequence of sends all originating from the root processor. Surprisingly, the APlib broadcast is also $O(p)$. Therefore, MPI performance is the best, and it is better as p increases. The results for PVM and APlib are very similar, although $t_b(p)$ is slightly better for the APlib broadcast and the startup time is a bit lower in PVM. The Fujitsu AP1000, but not the AP3000, included a specific network (B-net, Broadcast network) for one-to-all communications between a host processor and the other processors. Using this network, broadcast routines (and other collective primitives) with latency independent of p (that is, $O(1)$) could be implemented. Regarding $n_{\frac{1}{2}}(p)$, in MPI and PVM is a constant and in APlib is almost constant, because the complexities of $t_s(p)$ and $t_b(p)$ are the same within each library.

4.2. Data movement primitives

A broadcast is a data movement routine. Table 3 includes the estimated parameters for other MPI data movement primitives.

In a scatter operation (`MPI_Scatter`), a data structure that is stored on a single processor is distributed across the processors. In a gather (`MPI_Gather`), a distributed data structure is collected onto a single processor. In `MPI_Scatter` the message length n of Eq. 2 is the number of bytes of the message to be scattered from the root processor, and it is the total number of bytes of the message gathered onto the root processor in `MPI_Gather`. In both primitives, $t_b(p)$ is a decreasing function, and it tends to be constant as p increases. It can be due to the fact that, as p increases, the size of the message to be sent/received by each processor is smaller and these communications can be overlapped through the torus interconnection network. Nevertheless, the startup increases with p and is specially high (it is $O(p)$) in the scatter operation because the root processor has to split the message among the processors.

`MPI_Allgather` gathers a distributed array on all the processors. As expected, according to the modeling results, it is clearly better to use `Allgather` instead of the equivalent `Gather+Broadcast`, except for short messages (due to the high $O(p)$ startup of `Allgather`).

In a total exchange (`MPI_Alltoall`) each processor sends a distinct collection of data to every processor. This operation is the basis of the parallel FFT. The startup time is $O(p)$ and is the highest of the collective routines under evaluation, together with `Allgather`. Although the transfer time is also $O(p)$, the slope increases very slowly and it is almost constant.

4.3. Reduction primitives

The model of Eq. 2 hides an additional parameter that cannot be ignored (mainly for long messages) in reduction primitives: t_c, the cost per byte of the operation performed by the reduction. Therefore, we propose an extension of the model shown in Eq. 2, valid for all the collective primitives:

$$T(n,p) = t_s(p) + \frac{n}{Bw_{as}(p)} + t_c(p)n \tag{3}$$

where $Bw_{as}(p) = 1/t_b(p)$. The parameter t_c depends on the operation (sum, maximum ...) and the data type (integer, double ...). Clearly, for a broadcast $t_c(p) = 0$. We propose a new metric, the ratio transfer time-computation time: $r_{cc}(p) = t_b(p)/t_c(p)$, which provides a view of the weight of the computation factor as opposed to the communication factor in the total latency. As this metric is specially interesting when considering long messages (in which the effect of t_b and t_c is greater), it can be formally defined as:

$$r_{cc}(p) = \lim_{n\to\infty} \frac{t_s(p) + t_b(p)n}{t_c(p)n} = \frac{t_b(p)}{t_c(p)} \tag{4}$$

We have tested MPI and PVM reductions, `MPI_Reduce` and `pvm_reduce` (the APlib reduction can only be applied to single numbers and the result is stored in all the processors involved in the reduction). The parameters of the model were obtained as follows. First, a user-defined empty operation (`MPI_Nop`, `PvmNop`) was created. It allowed us to obtain $t_s(p)$ and $t_b(p)$ as in the broadcast model. Second, the regression procedure is repeated using the desired operation (sum, maximum ...) and $t_c(p)$ is estimated by substracting the first model (`Nop` model) from the

Table 4. Parameters for MPI reduction primitives.

	Reduce	Allreduce	Reduce_scatter	Parallel prefix
$t_s(p)$ (μs)	90log$_2$p-15	144log$_2$p+15	279log$_2$p-57	70p+38
$t_b(p)$ (μs)	0.0171log$_2$p+0.0037	0.0344log$_2$p	0.0147log$_2$p+0.0262	0.0145p+0.0116
$t_c(p)$ (μs)	0.0051log$_2$p-0.0037	0.0038log$_2$p-0.0026	0.0046log$_2$p+0.0049	0.0071p-0.0079

second one (`Op` model). Specifically, we have used a sum operation (`MPI_Sum`, `PvmSum`) of double precision floating point numbers (`MPI_DOUBLE`, `PVM_DOUBLE`) (8 bytes in our system) for all reduction routines.

We have found that the PVM reduction (and, in general, the group routines) are poorly implemented. The reduction routine is not robust: it fails for $p > 6$. Also, latencies are dominated by very high startup times; for instance, using the `PvmNop` operation for only 2 processors, $t_s \approx 5.55ms$ and it seems that it is $O(p)$. Another example: for p=3 and n=64 Kbytes, t_s represents $\approx$ 80% of latency.

Table 4 presents the estimated parameters of the model for MPI reduction routines. As expected, `MPI_Reduce` is $O(log_2p)$, which means that it uses a tree-structured communication pattern (bottom-up traversing of the tree). We can derive $r_{cc}(p)$=(0.0171log_2p+0.0037)/(0.0051log_2p-0.0037). Although it varies from 14.86 (for p=2) to 4.46 (for p=12), it tends to be a constant since p=4 and the transfer time per byte is approximately five times (on average) the computation time per byte.

`MPI_Allreduce` stores the result on all the processors. If we compare this routine with the equivalent `Reduce+Broadcast` (see the corresponding models), we can observe that there is not much difference (`Allreduce` is slightly better as p increases). It seems that the implementation of `Allreduce` could be improved by using an efficient butterfly-structured communication pattern.

The `MPI_Reduce_scatter` routine scatters the result across the processors (it is like `Reduce+Scatterv`). This primitive can be used as a basis for other primitives like `Allreduce`. Although it is $O(log_2p)$, the startup and the transfer time are very high. In general, the latencies achieved are similar to the equivalent `Reduce+Scatter` ones (provided that it was possible to use `Scatter` instead of `Scatterv`), even worse for large messages according to the models. Clearly, the implementation of `MPI_Reduce_scatter` on the AP is inefficient.

The `MPI_Scan` primitive carries out a parallel prefix operation. It is much like an `Allreduce` operation, but on each processor with rank r stores the result of operating the input values on processors with ranks $\leq r$. `MPI_Scan` is $O(p)$ on the AP, which leads to have the highest transfer time of all MPI routines under evaluation. It is not an optimized implementation because $O(log_2p)$ complexity could be achieved.

4.4. Experimental results

Figures 1 and 2 show some experimental results for the broadcast routines described in Section 4.1, by fixing p=8 (log scale) and n=64 Kbytes (linear scale), respectively. The filled symbols represent the estimated values of the latencies, following the derived models, whereas the dotted symbols are the experimental values of the latencies. The latter graph shows that the model is very accurate for PVM and APlib. This graph also reveals that in MPI, for that message size, the startup time of the model should be a bit higher, although the fitting is acceptable.

Figures 3-9 show an overview of the latencies of the MPI collective communications on the AP3000. In many cases the estimated values are hidden by the measured values, which means a good modeling.

Figures 3, 4 and 5 show latencies in an 8-processor configuration (log scale). In Figures 6 and 7, 8 and 9, the message size is set to 16 bytes and 256 Kbytes, respectively, and p varies from 2 up to 12 (linear scale). Figures 6, 7 focus on short-message latency (dominated by the startup time) and Figures 8, 9 on long-message performance (dominated by the transfer time). As can be observed, `MPI_Allgather` has the highest startup time (it is $O(p)$), and `MPI_Bcast` the lowest. Regarding the transfer time, `MPI_Scan` (also $O(p)$) is clearly the most expensive, while `MPI_Scatter` is the least expensive. It can be also observed that the implementations of `MPI_Allgather` and `MPI_Alltoall` are very similar: they have almost the same startup and their transfer times are represented by a similar curve, although `MPI_Alltoall` is more expensive. It is not surprising because an `Allgather` operation can be expressed in terms of an `Alltoall` operation. The poor implementation of `MPI_Reduce_scatter` pointed out in Section 4.3 is confirmed by the fact that, for short messages, it is less expensive to have the whole result of a reduction in all the processors (`Allreduce`) instead of distributed, due to the high startup of `MPI_Reduce_scatter` (see Figure 7).

5. Case study

We developed in [1] a parallel numerical algorithm for solving an elastohydrodynamic piezoviscous lubrication problem that appears in industrial devices. It involves very high execution times because very fine finite element

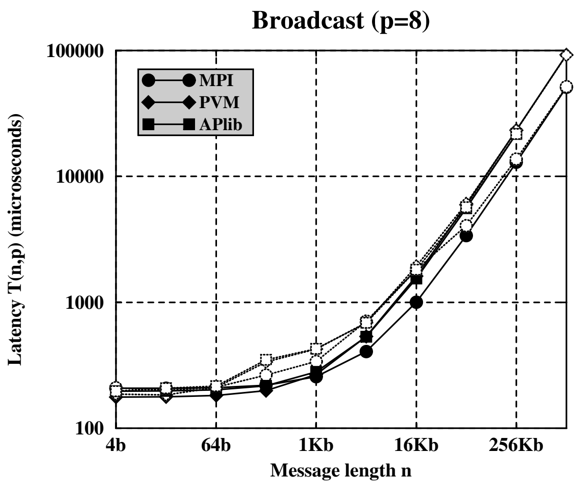

Figure 1. Broadcast latencies for various message sizes.

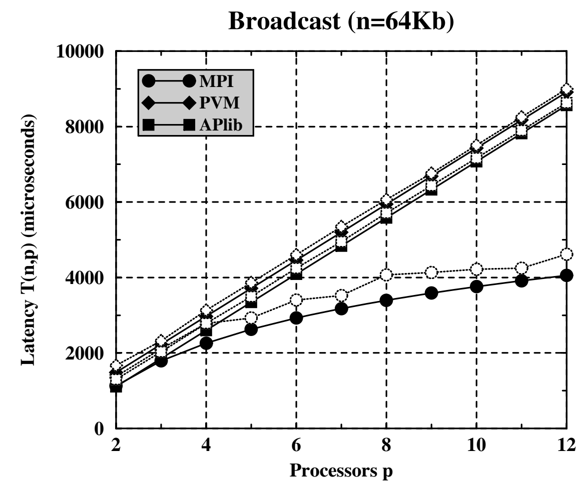

Figure 2. Broadcast latencies for various machine sizes.

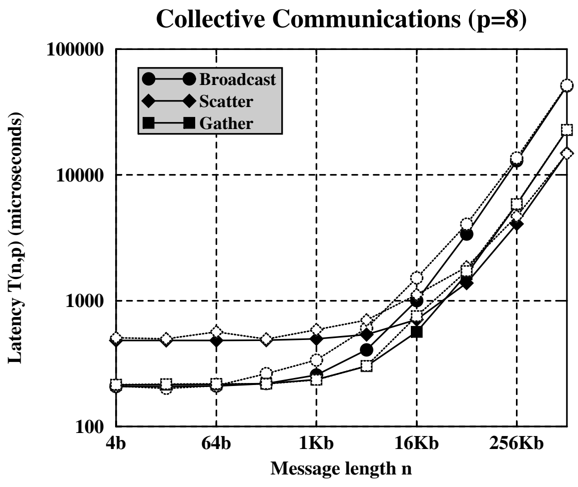

Figure 3. Latencies for various message sizes (broadcast, scatter, gather).

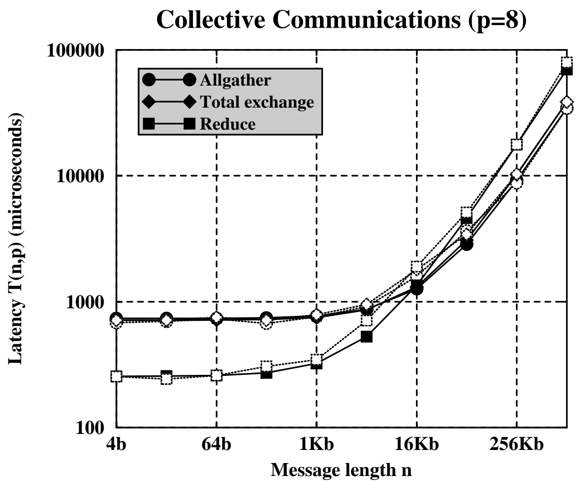

Figure 4. Latencies for various message sizes (allgather, total exchange, reduce).

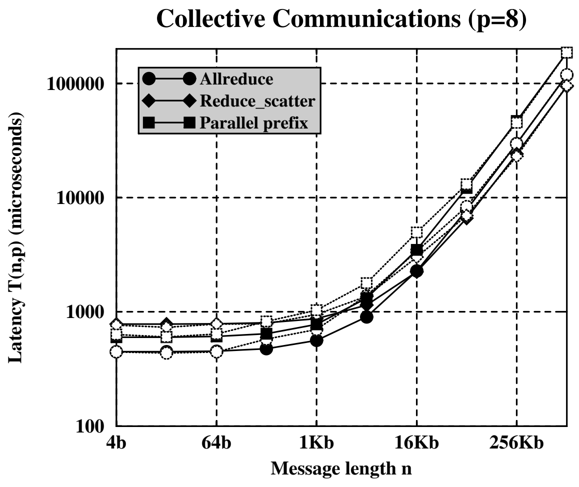

Figure 5. Latencies for various message sizes (allreduce, reduce_scatter, parallel prefix).

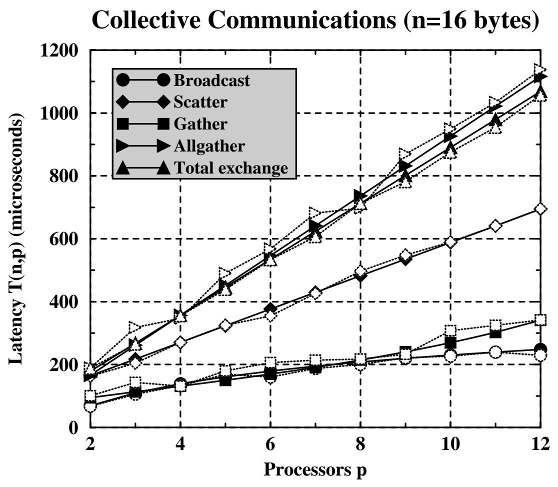

Figure 6. Latencies for various machine sizes (data movement routines).

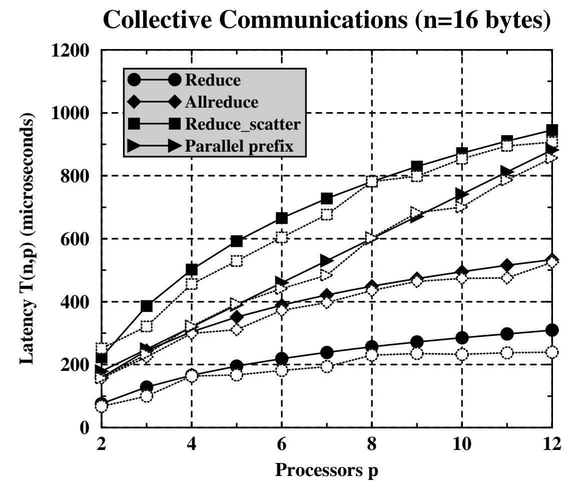

Figure 7. Latencies for various machine sizes (reduction routines).

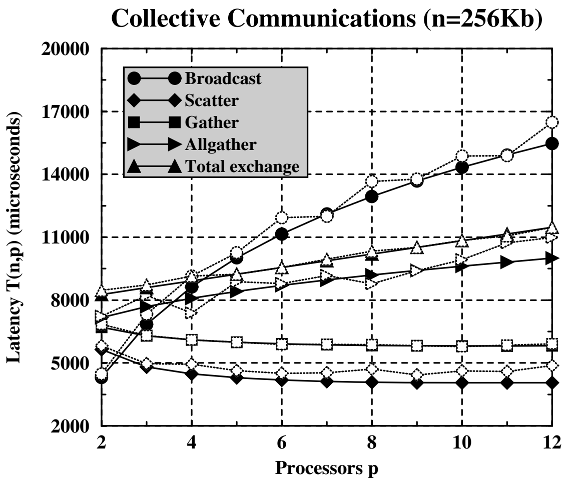

Figure 8. Latencies for various machine sizes (data movement routines).

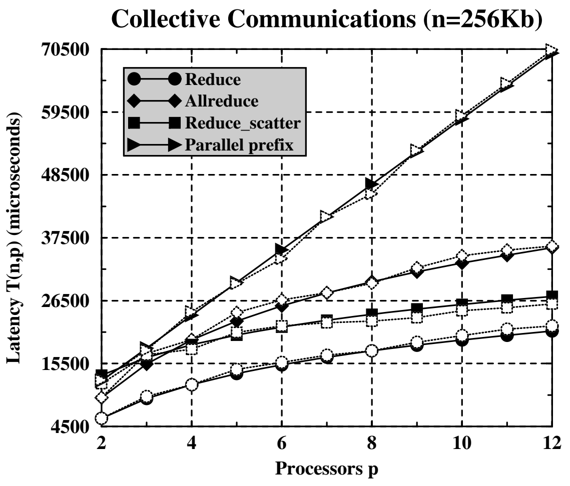

Figure 9. Latencies for various machine sizes (reduction routines).

meshes are required to obtain accurate solutions to the problem. We applied our models to this code, where an appropriate choice of the communication primitives can involve a substantial reduction in the execution times.

In one stage of the algorithm, the pressure approximation s of the lubricant fluid is computed by solving the sparse linear equation system $Cs = b$, where C is a finite element matrix. The right-hand side is computed by means of a reduction operation with processor 0 as root. Next, processor 0 distributes vector b in order to solve the system in parallel. It seems that the primitive `MPI_Reduce_Scatter` fits well for obtaining b. Nevertheless, in Section 4.3 we pointed out that, according to the models, it would be better to use (if possible) `Reduce+Scatter` instead of `Reduce_Scatter`, and the difference would be clearer for large messages. That is the case of our problem: we have used a standard block distribution and vector b has as many elements as nodes of the mesh. If we consider a mesh of 60000 nodes (vector b of 480000 bytes), the estimated and measured results of both approaches are shown in Table 5. The entry $\%$Red. represents the percentage of reduction in the measured execution times by using `Reduce+Scatter` instead of `Reduce_scatter`. As can be observed, this percentage goes down as p increases. Although the estimated times for this size of b are greater than the corresponding measured times, the difference between the measured times and the difference between the estimated times of both approaches follow the same tendency. Taking into account that this stage of the algorithm takes place over 1.6 million times (it is executed in the innermost loop of the algorithm), the profits obtained following the trends ruled by our modeling are evident.

The approximations to unknown parameters of the lubrication problem are computed using iterative methods. Some convergence tests involve gathering one element (double) per processor onto each processor. Therefore, an `Allgather` is needed to obtain an $8{*}p$-byte vector on all processors. According to the models of Section 4, it would be better to use `Gather+Broadcast` instead of `Allgather` for short messages, where latency is dominated by the startup time. This fact was experimentally proved in our code, as shown in the results of Table 5. It is clear that the improvement using the second approach increases with p, due to the $O(p)$ complexity of the startup of `Allgather`.

6. Conclusions

The characterization of the communication overhead is very important in the development of parallel codes. In this work, we have presented a comprehensive study of the AP3000 message-passing primitives. Such primitives are the basis for the design of more complex communication patterns that can appear in parallel applications. The models

Table 5. Estimated and measured execution times (in μs) of the example code.

p	Reduce_scatter		Reduce+Scatter		%Red.	Allgather		Gather+Broad.		%Red.
	Estim.	Meas.	Estim.	Meas.		Estim.	Meas.	Estim.	Meas.	
2	24414	21568	20927	18920	12.28%	166.43	177.61	164.54	168.146	5.33%
4	33957	31167	29440	27376	12.16%	356.94	354.43	271.69	269.717	23.90%
8	43500	39534	39291	36269	8.26%	738.07	718.67	427.93	420.668	41.47%
12	49082	47181	45355	43444	7.92%	1119.25	1122.82	601.14	530.717	52.73%

and metrics used in the previous sections helped us to identify design faults in the communication routines and, furthermore, to estimate and improve the performance of the most time-consuming stages of parallel programs, as shown in Section 5. The primitives could be more accurately modeled by defining different functions for different message length intervals. Nevertheless, we found more interesting to show global functions which have been experimentally proved to have a reasonable accuracy and that provide a clearer overview of the primitives' behavior. Machine vendors should provide the parameters of these models (or, at least, complexities) for basic communication routines.

Regarding the AP3000 libraries, we can conclude that the PVM/AP library (particularly, the group routines) is a naive implementation. The APlib routines are not robust for long messages, the broadcast implementation is inefficient and the APlib reduction routines are only available for single numbers. Besides, APlib is a proprietary library with a small set of primitives compared to MPI. It is clear that APlib is available for compatibility with the AP1000 machine, where this library had a good behavior.

Currently, MPI/AP is the best choice to program the AP3000 because the basic primitives present acceptable latencies; nevertheless, the implementation of more sophisticated collective communications, such as `Reduce_Scatter` and `Scan` should be revised. Furthermore, the AP3000 hardware is not fully exploited by the MPI/AP library. Message latencies could be reduced by re-designing the low-level communication mechanisms. In fact, MPI/AP was constructed using too many communication layers. Software developments to achieve high-performance communication primitives for the AP3000 were mentioned in Section 2. Hardware improvements, such as the SBus design (the I/O bus which connects the processor and the message controller), could also help to reach this aim.

Acknowledgments

We gratefully thank CESGA (Galician Supercomputing Center) for providing access to the AP3000. This work was funded by the Spanish Ministry of Science and Technology and the European Union under Contract 1FD97-0118-C02.

References

[1] M. Arenaz, R. Doallo, J. Touriño, and C. Vázquez. A Parallel Algorithm for an Elastohydrodynamic Piezoviscous Lubrication Problem. In *Parallel Numerical Computations with Applications*, chapter 13, pages 191–201. Kluwer Academic Publishers, 1999.

[2] M. Bernaschi, G. Iannello, and M. Lauria. Experimental Results about MPI Collective Communication Operations. In *7th Int'l Conference on High-Performance Computing and Networking*, volume 1593 of *Lecture Notes in Computer Science*, pages 774–783. Springer-Verlag, 1999.

[3] R. W. Hockney. The Communication Challenge for MPP: Intel Paragon and Meiko CS-2. *Parallel Computing*, 20(3):389–398, 1994.

[4] L. S. Huse. Collective Communication on Dedicated Clusters of Workstations. In *6th European PVM/MPI Users' Group Meeting*, volume 1697 of *Lecture Notes in Computer Science*, pages 469–476. Springer-Verlag, 1999.

[5] H. Ishihata, T. Horie, and T. Shimizu. Architecture for the AP1000 Highly Parallel Computer. *Fujitsu Sci. Tech. Journal*, 29(1):6–14, 1993.

[6] H. Ishihata, M. Takahashi, and H. Sato. Hardware of AP3000 Scalar Parallel Server. *Fujitsu Sci. Tech. Journal*, 33(1):24–30, 1997.

[7] J. Miguel, A. Arruabarrena, R. Beivide, and J. A. Gregorio. Assessing the Performance of the New IBM SP2 Communication Subsystem. *IEEE Parallel & Distributed Technology*, 4(4):12–22, 1996.

[8] N. Nupairoj and L. M. Ni. Performance Metrics and Measurement Techniques of Collective Communication Services. In *1st Int'l Workshop on Communication and Architectural Support for Network-Based Parallel Computing*, pages 212–226, San Antonio, TX, 1997.

[9] M. Prieto, D. Espadas, I. M. Llorente, and F. Tirado. Message Passing Evaluation and Analysis on Cray T3E and SGI Origin 2000 Systems. In *5th Int'l Euro-Par Conference*, volume 1685 of *Lecture Notes in Computer Science*, pages 173–182. Springer-Verlag, 1999.

[10] D. Sitsky and P. Mackerras. A High-Performance Message-Passing Library for the AP3000. In *8th Int'l Parallel Computing Workshop*, pages 245–251, Singapore, 1998.

[11] C. P. Tan, W. F. Wong, and C. K. Yuen. PVM Enhancement for AP3000. In *7th Int'l Parallel Computing Workshop*, pages P1D1–P1D8, Canberra, 1997.

[12] Z. Xu and K. Hwang. Modeling Communication Overhead: MPI and MPL Performance on the IBM SP2. *IEEE Parallel & Distributed Technology*, 4(1):9–23, 1996.