The Performance of the MPI Collective Communication Routines for Large Messages on the Cray T3E-600, the Cray Origin 2000, and the IBM SP.

Glenn R. Luecke, Bruno Raffin and James J. Coyle
grl@iastate.edu, raffin@iastate.edu, jjc@iastate.edu
Iowa Sate University
Ames, Iowa 50011-2251 USA

July 1999

Abstract

We have implemented eight of the MPI collective routines using MPI point-to-point communication routines with algorithms designed to be efficient for large messages. The performance of our implementations of these collective routines is compared with the vendor implementations on the Cray T3E-600, the Cray Origin 2000 and on the IBM SP. Many of our implementations significantly outperformed vendor implementations on the T3E and the Origin 2000. On the SP, only our implementation of the broadcast significantly outperformed IBM's implementation.

Keywords: MPI; Collective Communication Routines for Large Messages; Cray T3E; Origin 2000; IBM SP.

1 Introduction

Today, MPI [15] is probably the most used message passing library for programming distributed memory parallel computers. Implementations of MPI are available for all commercially available parallel platforms. The MPI collective communication routines provide important functionality for scientific computing and the algorithms chosen for their implementation can greatly influence their performance. In papers [8, 13], the authors present a library of collective communication routines, called iCC, that uses different algorithms for small and large messages except for the gather and scatter routines. The gather and scatter routines use the same algorithm for all message sizes. They show that their algorithms for large messages lead to significant performance improvements on the Intel Paragon when compared with MPICH [11], BLACS [9] and NX [4]. The purpose of our study is to implement the MPI collective communication routines using their algorithms for large messages and to compare their performance with current vendor MPI implementations on the Cray T3E-600, the Cray Origin 2000, and the IBM SP. The MPI routines used for this study are: mpi_scatter, mpi_gather, mpi_allgather, mpi_reduce_scatter, mpi_bcast, mpi_reduce, mpi_allreduce and mpi_alltoall (this includes most but not all of the MPI collective routines). Papers [8, 13] present algorithms for the above listed collective routines, except for the all-to-all. We have developed an algorithm for mpi_alltoall and have implemented all of the above collective routines using mpi_ssend, mpi_isend, mpi_sendrecv and mpi_recv.

The Cray Origin 2000 [1, 6, 10] used was a 128 processor machine located in Eagan, Minnesota. Each processor is a MIPS R10000 microprocessor running at 250 MHz. The tests were run with MPI version 3.2.x.6 and the MIPSpro Fortran compiler version 7.2.1.2m.

The Cray T3E-600 [1, 14, 7] used was a 512 processor machine also located in Eagan, Minnesota. Each processor is a DEC Alpha EV5 microprocessor running at 300 MHz. The tests were run with MPI version 3.1.1.0 and the Fortran compiler version 3.1.0.0.2. This T3E had processors with the stream buffer problem and hence MPI performance is degraded in some circumstances [5].

The IBM SP [2, 3] used was a 251 processor machine located at the Maui High Performance Computing Center, Hawaii. Because of limited access to the machine, a maximum of 96 processors (P2SC microprocessors running at 160 MHz) were available for running our tests. The tests were run with the MPI from IBM's POE version 1.2.1 and the XLF Fortran compiler version 4.1.0.0.

Tests run on T3E and Origin 2000 were executed on machines dedicated to running only these tests. Tests run on the IBM SP were executed using IBM's scheduler Loadleveler so that only one job at a time would be executing on the requested processors.

The performance results reported in this paper were obtained with a variety of message sizes, all using 8 byte reals, and up to the maximum number of processors available for each machine (512 for the T3E, 128 for the Origin 2000 and 96 for the IBM SP).

Section 2 introduces the timing methodology used. Section 3 presents the implemented algorithms. Section 4 shows the performance results. The scalability of the implemented algorithms is studied in section 5 and the conclusions are discussed in section 6.

2 Timing Methodology

Timings were done by first flushing the cache on all processors by changing the values in the real array `cache(1:ncache)`, prior to timing the desired operation. The value `ncache` was chosen so the size of `cache` was the size of the secondary cache for the T3E (96 Kbytes) and the Origin 2000 (4 Mbytes), and the size of the primary cache for the IBM SP (64 Kbytes) since this machine does not have a secondary cache. Here 1 Kbyte means 1024 bytes and Mbyte means 1024^2 bytes. Note that by flushing the cache between each trial, the data that may have been loaded in the cache during the previous trial cannot be used to optimize the communications of the next trial, which a machine like the Origin 2000 could do [6]. The following shows the code used for the timings.

```
parameter (ntest=26)
real*8, dimension(ntest) ::  time, array_time
.  .  .
do k = 1, ntest
   cache = cache + 0.1
   call mpi_barrier(mpi_comm_world,ierr)
   time1 = mpi_wtime()

   .  .  .  call to the collective communication to be timed .  .  .

   time2 = mpi_wtime()
   array_time(k) = time2 - time1
   call mpi_barrier(mpi_comm_world,ierr)
end do
call mpi_reduce(array_time(1),time(1),ntest,&
   & mpi_real8,mpi_max,0,mpi_comm_world,ierr)
```

The first call to `mpi_barrier` guarantees that all processors reach this point before they each call the wall-clock timer, `mpi_wtime`. The second call to `mpi_barrier` is to make sure that no processor starts the next iteration (flushing the cache) until all processors have completed executing the `call to the collective communication to be timed`. The test is executed `ntest` times and the values of the differences in times on each participating processor are stored in `array_time`. The call to `mpi_reduce` calculates the maximum of `array_time(k)` for each fixed `k` and places this maximum in `time(k)` on processor 0 for all values of `k`. Thus, `time(k)` is the time to execute the test for the k-th trial.

The clock ticks for `mpi_wtime()` are every 0.013 microseconds on the T3E and the SP, and every 0.800 microseconds on the Origin. These timer resolutions are accurate enough to measure the execution times that are greater than 120 microseconds for the performed tests.

Observe from the above timing code that one run will measure the time to call the collective routine `ntest` times. For the T3E and SP machines the measured first time was always significantly larger than most of the subsequent times. For these machines, `ntest` was set to 26 and the first measured time was thrown away. Examination of data showed that there was no need to take larger values of `ntest` for these machines. The remaining data was then filtered as described below. All timings on T3E were done on a dedicated machine with no other jobs running. IBM's Loadleveler was used on the SP so there were no other jobs running on the processors requested, but other jobs could delay communication between the processors being used by our job. Therefore on the SP, the timing program was run twice and the data for the 50 trials was filtered as described below.

Timing on the Origin was more challenging. Figure 1 shows execution times for our implementation of `mpi_reduce` with `ntest` = 50. For some tests, the first 20 to 25 timings showed significant variation and then the timings would settle down. For this reason, `ntest` was set to 50 (except when 124 and 128 processors were used) and only the last 25 timings were not thrown away. When 124 and 128 processors were used, `ntest` was set to 100 and only the last 25 timings were kept because of timing fluctuations for roughly the first 75 trials. All Origin timings were done on a dedicated machine with no other jobs running. The remaining data was filtered as described below.

Among the remaining trials, occasionally a few trials have significantly times larger than

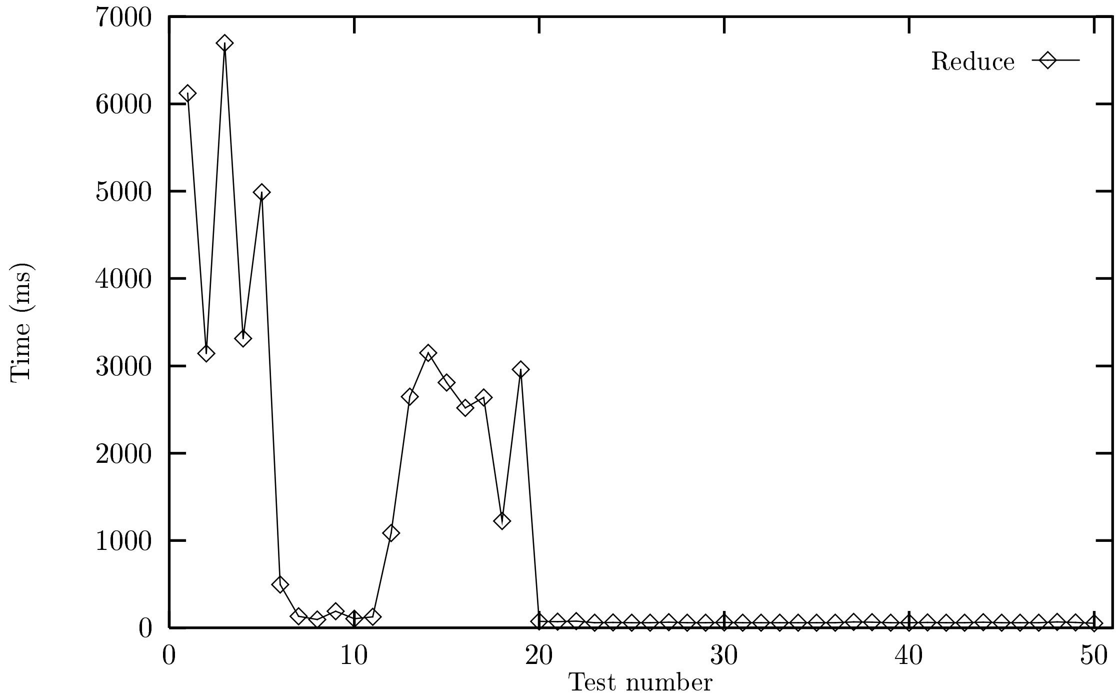


Figure 1: Execution times on the Origin 2000 for 50 executions of our implementation of `mpi_reduce` with 64 processors and a 1 Mbyte message.

the other trials, making the average over all trials to significantly increase (see [12] for more details about this behavior). These time "spikes" are likely due to operating system interruptions. It is our opinion that these spikes should be removed so that the performance data will reflect the times one will usually obtain. All data in this report has the spikes removed by the following process: First the median is computed and all times greater than 1.8 times this median are removed. There were a few cases where the above procedure would remove more than 10% of the data. We felt that this would not be appropriate, so in these cases only the 10% of the largest times were removed. An average was then calculated from the filtered data and this is what is presented for each test in this report. For all tests and all message sizes, this filtering process led to the removal of 0.001% of the trials for the T3E, 0.53% for the Origin 2000, 2% for the IBM SP.

3 The Algorithms

The time required to perform a collective operation is a function of the size of the message, the number of processors involved, the latency of the network, the bandwidth of the network and the algorithm used. Memory requirements for performing a collective operation with a large message can be an important factor for algorithm selection. This section presents the time and memory requirements for each collective routine assuming a linear model for communication. Notice that memory requirements are not presented in [8, 13].

Assume that the time required to send a message of n bytes from one processor to another is given by

$$\alpha + n\beta$$

where α is the latency and β is the communication rate. Let p be the number of processors involved in the collective operation. For ease of exposition, assume p is a power of two, n is a multiple of p, and the root processor is processor 0. For collective operations involving reductions, an arithmetic operation is used. Assume that this operation requires γ seconds to execute. Notice that all tests were run using the summation.

3.1 The Scatter

The scatter is implemented by an algorithm based on a binary tree. Initially the root processor 0 owns p messages m_0, ..., m_{p-1}, each of size n, that have to be sent to processors 1, ..., $p-1$ respectively. First, processor 0 sends $m_{p/2}$, ..., m_{p-1} to processor $p/2$. For the second step, processors 0 sends messages $m_{p/4}$, ..., $m_{p/2-1}$ to processor $p/4$ and concurrently processor $p/2$ sends messages $m_{3p/4}$, ..., m_{p-1} to $3p/4$. The scatter is completed by repeating these steps $\log(p)$ times. Based on the model described above,

$$\text{execution time} = \alpha \log p + (p-1)n\beta$$

Notice that to execute the above algorithm, some processors require the allocation of temporary arrays:

- Processor i with i even and $0 < i \leq p/2$ needs to store a temporary message of size ni.
- Processor i with i even and $p/2 < i < p-1$ needs to store a temporary message of size $n(p-i)$.

3.2 The Gather

The gather is implemented as a scatter in reverse. The execution time and the space required to store temporary messages are the same as the gather.

3.3 The Gather-To-All

For the gather-to-all, initially each processor i owns a message m_i of size n. These messages have to be gathered such that at the end each processor owns all the messages. This is performed by $p-1$ right-shifts executed by each of the p processors, where the j^{th} right-shift is defined by: each processor i receives the message $m_{(i-j) mod\ p}$ from processor $(i-1) mod\ p$. Here

$$\text{execution time} = (p-1)(\alpha + n\beta).$$

The space reserved to receive the final result of the gather-to-all is sufficient to contain the temporary messages.

3.4 The Reduce-Scatter

For the reduce-scatter, initially processor i owns a message m_i of size np that is cut in p parts. Let m_i^j be the j^{th} part of the message m_i. First, processor i receives $m^i_{i-1 mod\ p}$ from processor $(i-1) mod\ p$ and applies the reduction operator between $m^i_{(i-1) mod\ p}$ and m^i_i. For the other steps $1 < j < p$, processor i receives $m^i_{(i-j) mod\ p}$ from processor $(i-j) mod\ p$ and applies the reduction operator between $m^i_{(i-j) mod\ p}$ and the result of the reduction performed during the previous step. Here

$$\text{execution time} = (p-1)(\alpha + n\beta + n\gamma).$$

The space reserved to receive the final result of the reduce-scatter is not sufficient to contain the temporary messages. Each processor needs to store a temporary message of size n.

3.5 The Broadcast

For the broadcast, the root processor 0 owns a message m of size n that has to be sent to all other processors. It is implemented by a scatter followed by a gather-to-all. Here

$$\text{execution time} = (\log p + p - 1)\alpha + 2\frac{p-1}{p} n\beta.$$

The space reserved to receive the final result of the broadcast is sufficient to contain the temporary messages.

3.6 The Reduce

The reduce is implemented by a reduce-scatter followed by a gather where initially processor i owns a message m_i of size n. Here

$$\text{execution time} = (\log p + p - 1)\alpha + n\frac{p-1}{p}(2\beta + \gamma).$$

The space reserved to receive the final result of the reduce is not sufficient to contain the temporary messages:

- For the reduce-scatter, processor $i > 0$ needs a space of size $2n/p$ to store the incoming messages and the results of the reduction operations.
- For the scatter, processor i with i even and $0 < i \leq p/2$ needs to store a temporary message of size in/p. Processor i with i even and $p/2 < i < p-1$ needs to store a temporary message of size $(p-i)n/p$.

3.7 The All-Reduce

The all-reduce is the same as the reduce except the reduced message is placed on all the processors instead of only on the root processor. It is implemented by a reduce-scatter followed by a gather-to-all. Here

$$\text{execution time} = 2(p-1)\alpha + n\frac{p-1}{p}(2\beta + \gamma).$$

The space reserved to receive the final result of the all-reduce is sufficient to contain the temporary messages.

3.8 The All-To-All

Initially, processor i owns p messages of size n denoted by m_i^0, m_i^1, ..., m_i^{p-1}. The algorithm is completed when each processor i owns the p messages m_0^i, m_1^i, ..., m_{p-1}^i. This is performed by cyclically shifting the messages on the p processors. At the step $0 < j < p$, each processor i sends $m_i^{(i-j) mod\ p}$ to the processor $(i-j) mod\ p$. Here

$$\text{execution time} = (p-1)(\alpha + n\beta).$$

The space reserved to receive the final result of the all-to-all is sufficient to contain the temporary messages.

3.9 Remark

The reduce-scatter and the all-to-all algorithms can be modified by changing the execution ordering of the steps of the cyclic shift such that short and long-range communications are alternated. With such a scheme that balances short and long-range communications, less contention on the network is likely to occur. For the machines used in this study, these modified algorithms provide a performance improvement but only for a large number of processors. When there is not a "large" number of processors being used for the communication, these modified algorithms provide about the same performance as the original algorithms. For example, on the T3E a 20% improvement is achieved when using the modified algorithm for the all-to-all with 400 processors and a 50 Kbyte message.

4 Results

Figures 2–7 pages 9–14 present the ratio of the time measured for the MPI routines provided by the vendor divided by the time measured for our implementation. Figures 2, 4 and 6 present these ratios as a function of the message size for execution with 70 processors. For these Figures, message sizes vary from 1 Kbyte to 10 Mbytes or up to as large a message could be run with the memory available on the specified machine. Notice that the number of processors (70) is not a power of two in order to not favor any algorithm. Figure 3, 5 and 7 present these ratios as a function of the number of processors for a selected message size. The message size was chosen so that execution times were not too large and the machine being used had enough memory so the test could be run with most of the available processors. The tests were run with 10, 20, ..., 510 and 512 processors on the T3E, 16, 32, ..., 112, 126 and 128 processors on the Origin 2000, and 4, 8, ..., 96 processors on the SP. Limited access to the machines did not allow to run the tests with all possible numbers of processors (1, 2, 3,...).

When considering large messages, memory requirements can be a critical point that should be taken into consideration as well as performance in the selection of an algorithm. For example, our implementation of the gather-to-all executed with a 4 Mbyte message and 20 processors on the T3E was able to run whereas the vendor implementation could not execute because of insufficient memory (each processor had 128 MBytes of memory). But because we do not have access to the vendor codes, it is difficult to compare the memory requirements of the vendor implementations and of our implementations. Moreover, limited access to the machines did not allow us to systematically test memory requirements.

4.1 T3E Results

Figure 2 page 9 shows that our implementations significantly outperform the vendor implementations with large messages using 70 processors for all the tests except the scatter and the all-to-all. Notice that our implementation of the reduce-scatter is 4.5 faster than the vendor implementation for 100 Kbyte messages and is also faster for the smallest message size tested (1 Kbyte).

Figure 3 page 10 shows that our implementation and the vendor implementations of the scatter and all-to-all perform about the same. Our implementation of the gather, gather-to-all and reduce-scatter perform significantly better than the vendor implementations. Notice that the reduce, all-reduce and broadcast significantly outperform the vendor implementations reaching a maximum around 50-100 processors, but their relative performance then decrease as the number of processors approaches 512.

4.2 Origin 2000 Results

For the Origin 2000, Figure 4 page 11, shows that our implementations significantly outperform the vendor implementations with large messages using 70 processors for the gather-to-all, reduce-scatter, broadcast, reduce and all-reduce routines.

Figure 5 page 12 shows that our implementations of the gather-to-all and reduce-scatter significantly outperform the vendor implementations. This is not the case for our implementations for all of the other routines.

4.3 IBM SP Results

On the IBM SP, Figures 6 page 13 and 7 page 14 show that our implementation of the broadcast significantly outperform the IBM implementation. For all the other routines, the IBM implementation is either better or about the same as our implementation.

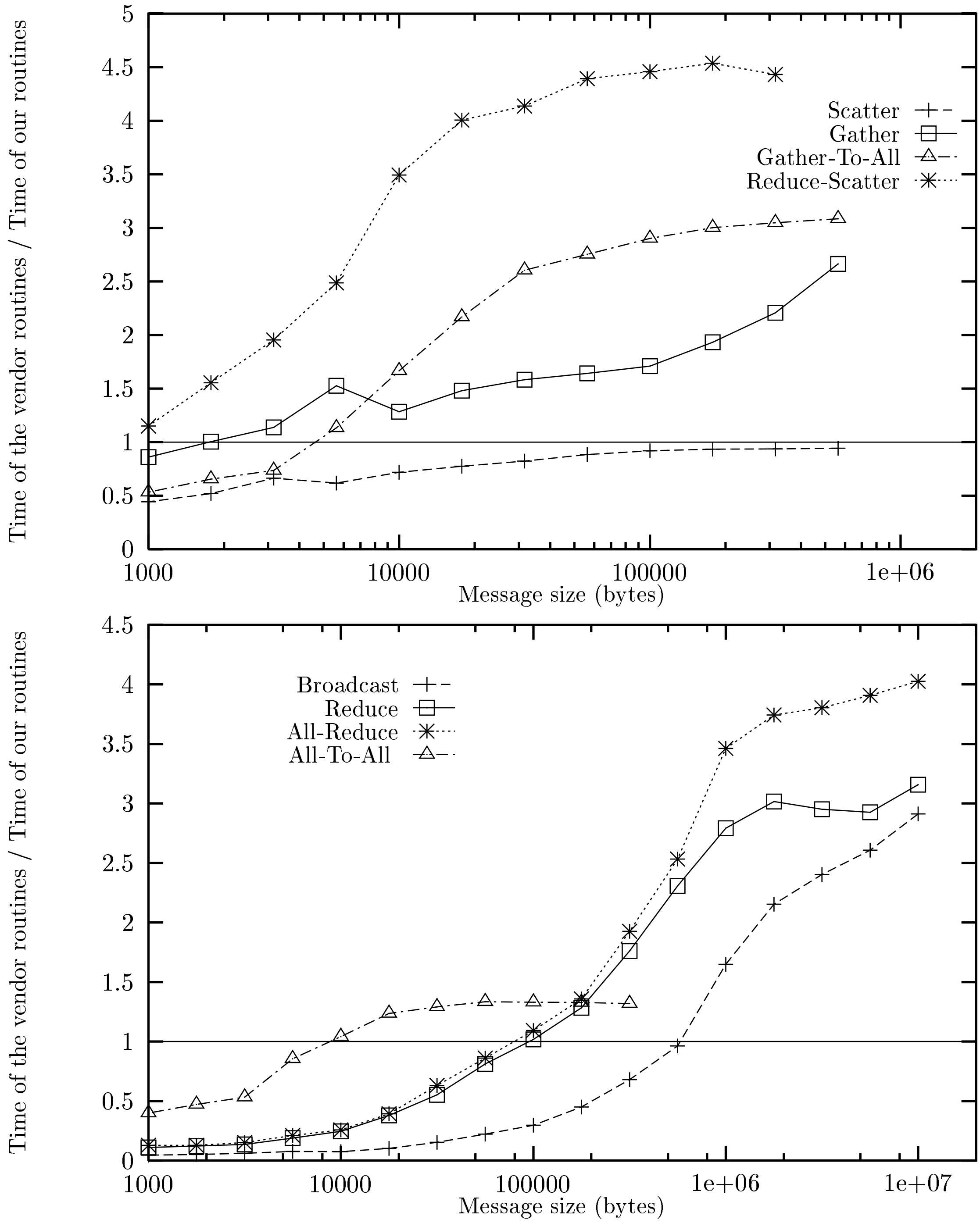


Figure 2: Cray T3E - 70 Processors

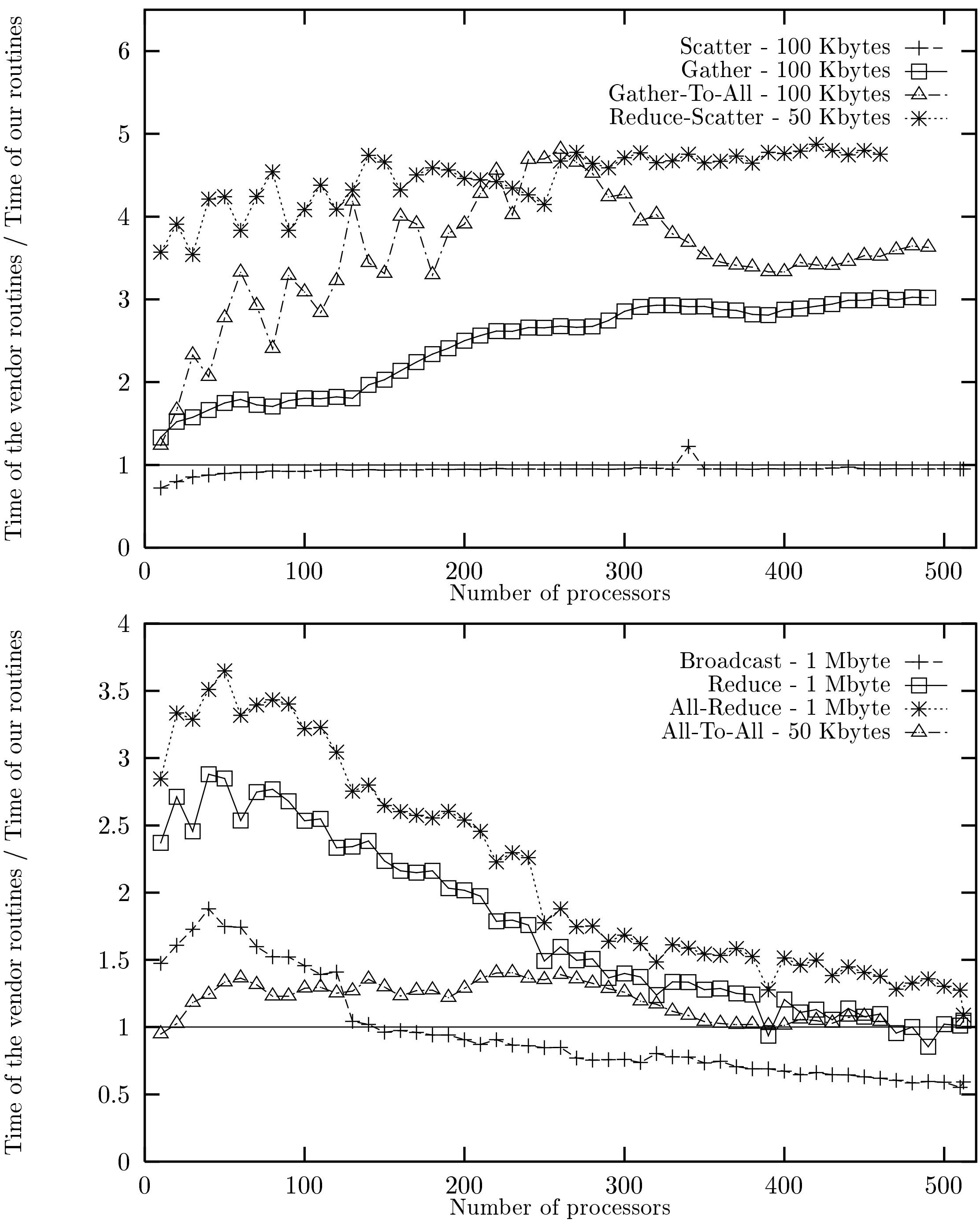


Figure 3: Cray T3E

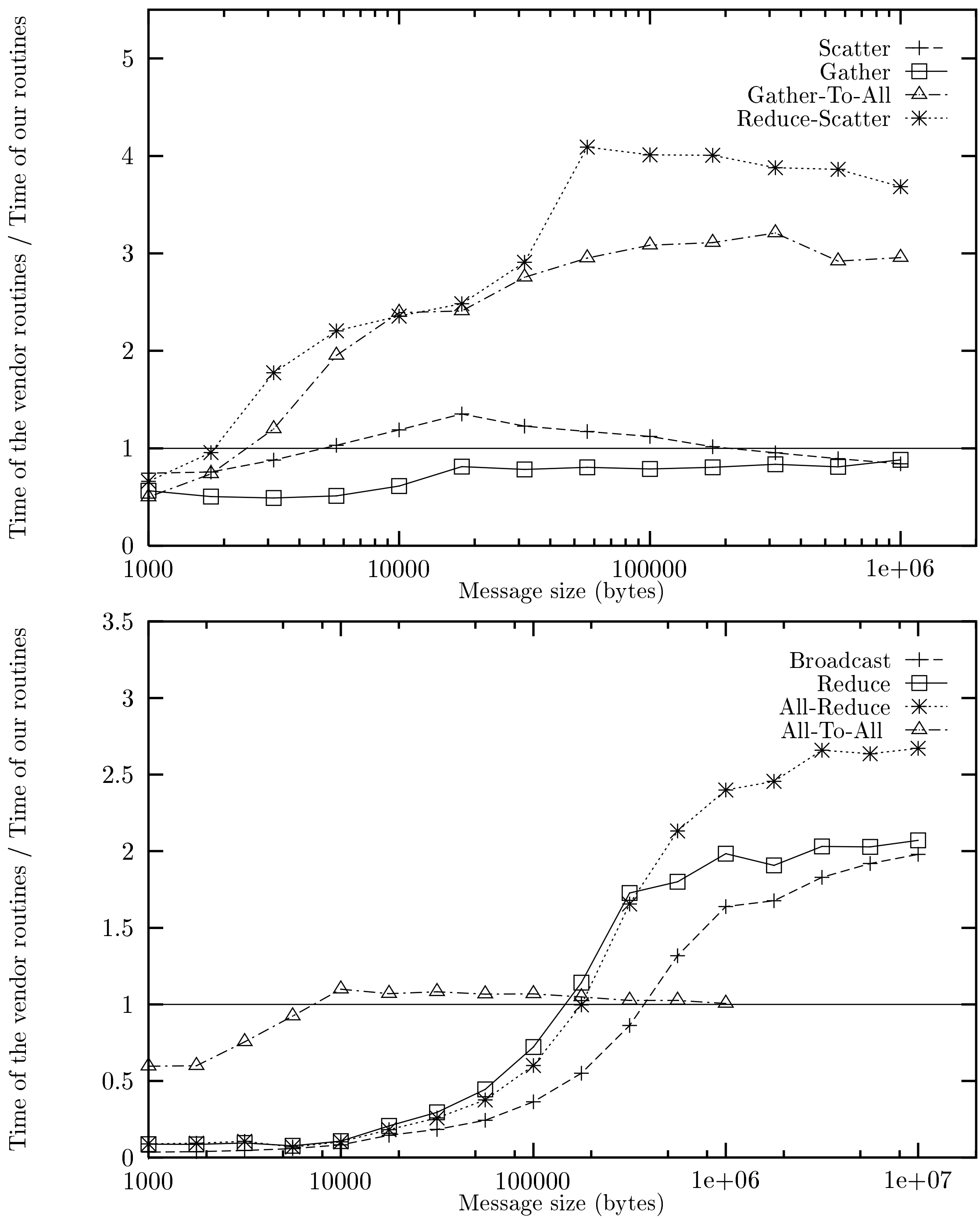


Figure 4: Origin 2000 - 70 processors

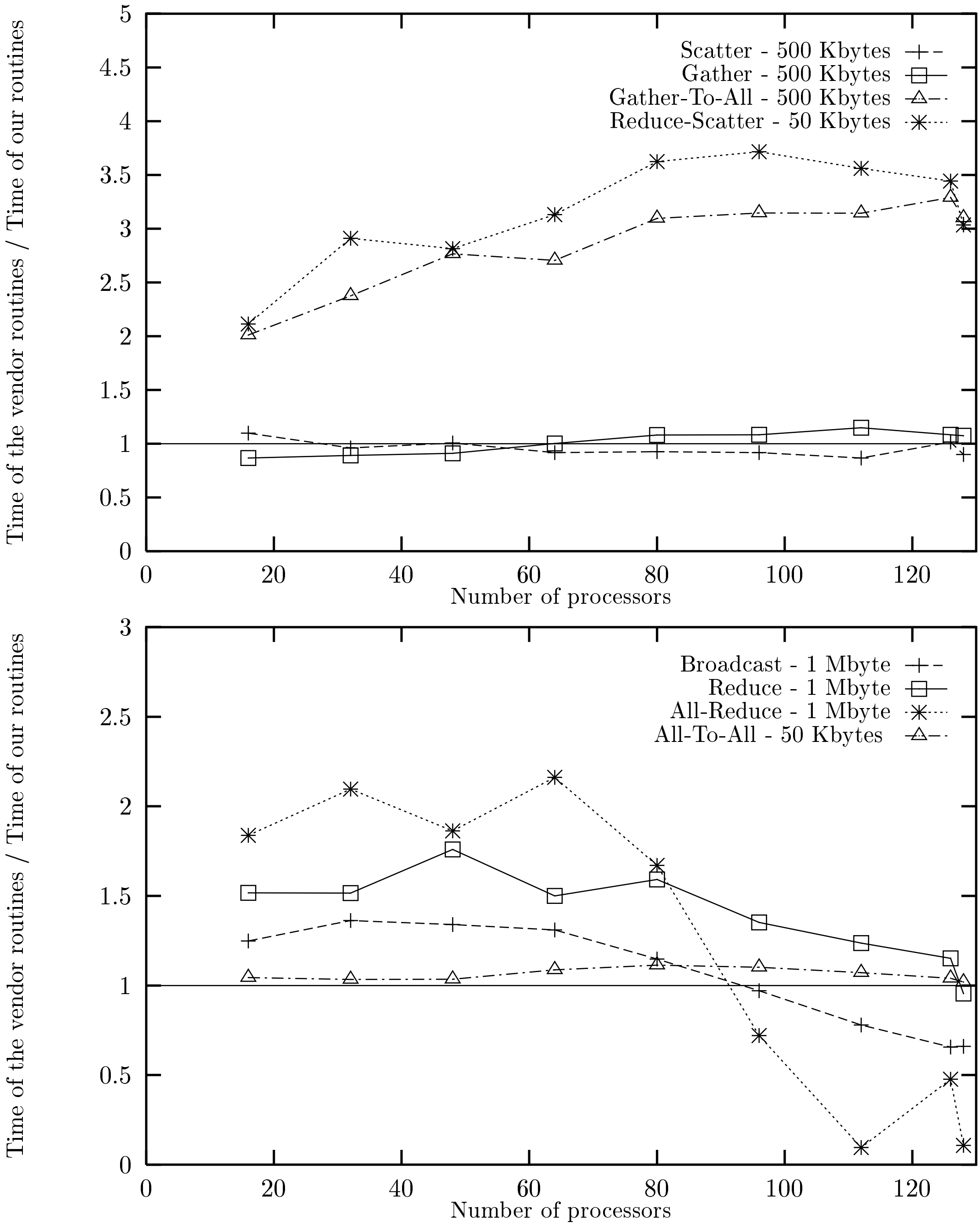
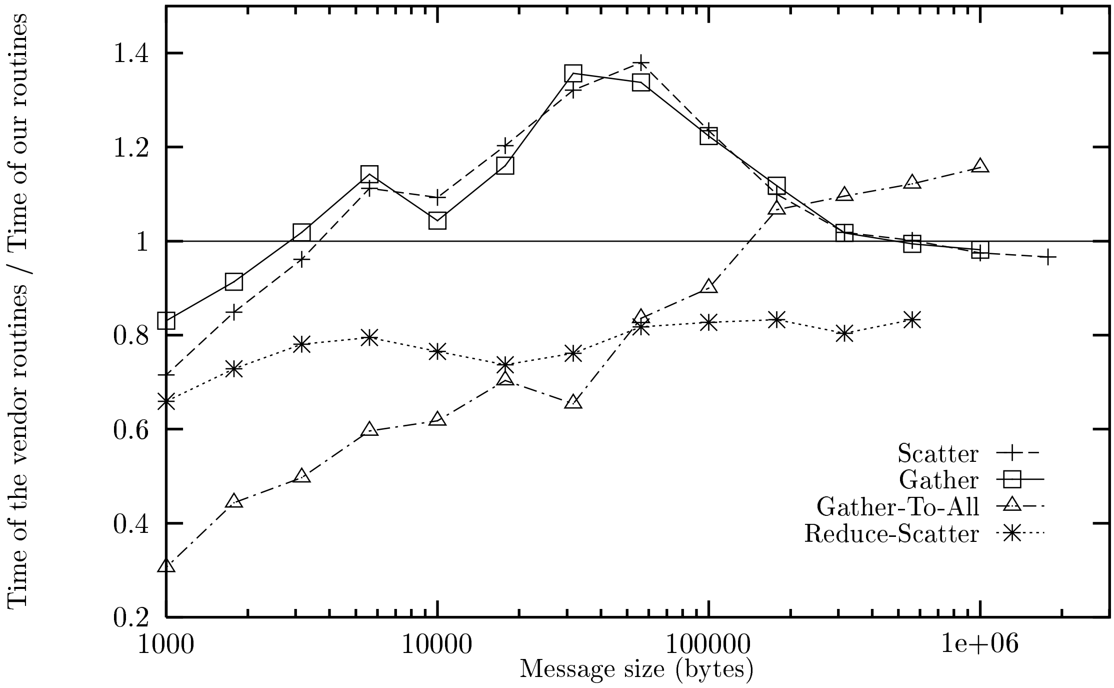


Figure 5: Origin 2000



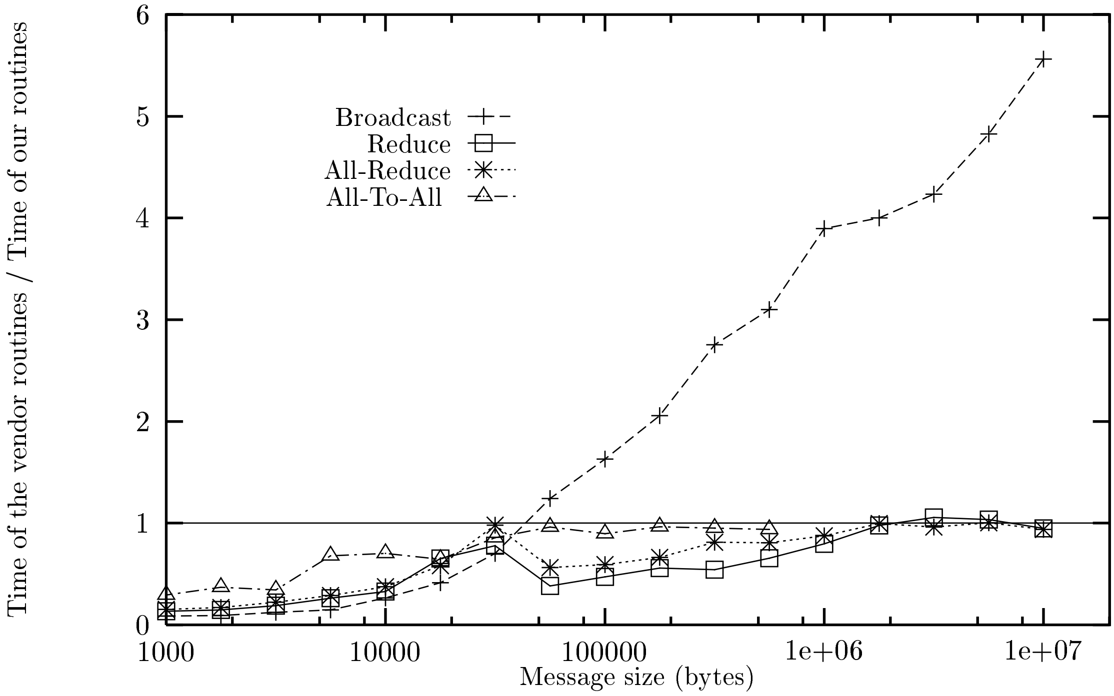


Figure 6: IBM SP - 70 Processors

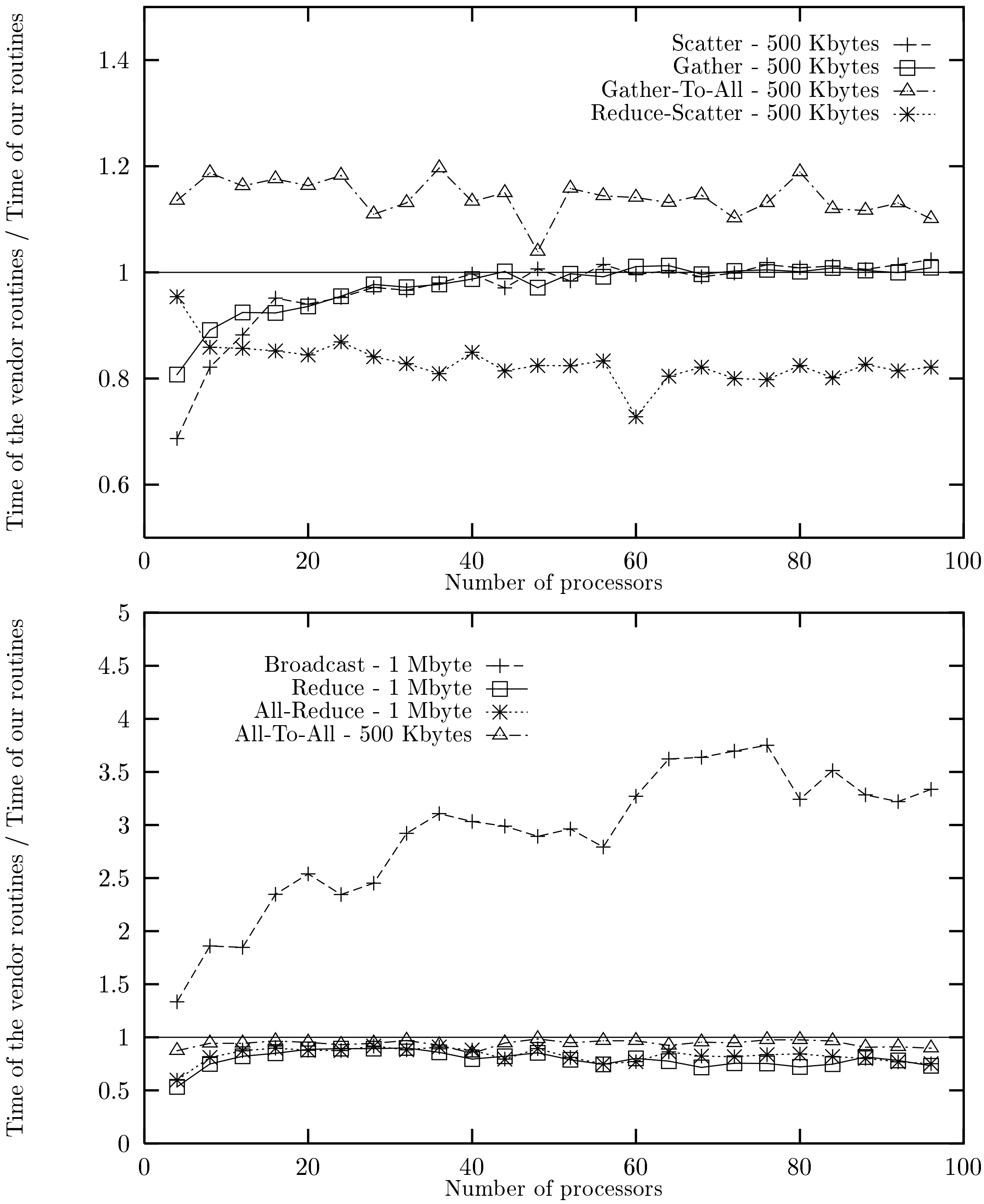


Figure 7: IBM SP

5 Scalability of the Collective Routines

In this section we discuss the scalability of the routines we implemented for a large number of processors. Assuming the simple executing model presented in section 3 and assuming there are p processors involved in the execution, the execution time for the routines we implemented increase linearly with p for large p. Table 5 gives the slopes for each of these routines where the routines have been ordered from largest slope to smallest, assuming $2\alpha < n\beta$.

Routine	Slope
Reduce-Scatter	$\alpha + n\beta + n\gamma$
All-To-All	$\alpha + n\beta$
Gather-To-All	$\alpha + n\beta$
Gather	$n\beta$
Scatter	$n\beta$
All-Reduce	2α
Broadcast	α
Reduce	α

Figure 8 on page 16 shows the execution time as a function of p for these routines for a 50 Kbyte message on the T3E, the machine with the most processors that we had access to. Notice that the graphs for the gather and scatter appear to be the same as the model predicts. Our simple execution model corresponds surprisingly well with the graphs in Figure 8. The two main exceptions occur between the all-to-all and gather-to-all, and between the all-reduce, reduce and broadcast. Table 5 shows that the simple execution model predicts the slope of the execution time to be the same for the all-to-all and gather-to-all. However, Figure 8 shows the slope of the all-to-all to be about three times that of the gather-to-all. A gather-to-all is executed by $p-1$ righ-shifts whereas for the all-to-all each processor j sends and receives a message to/from all other processors. Our simple execution model cannot distinguish between these. Notice that the slope for the all-reduce is about the same as the slope for the reduce and broadcast instead of twice this slope as predicted by the model.

We also would like to point out that Figures 3 and 5 show that the vendor routines for the broadcast, reduce and all-reduce scale better than our routines on the T3E and Origin 2000.

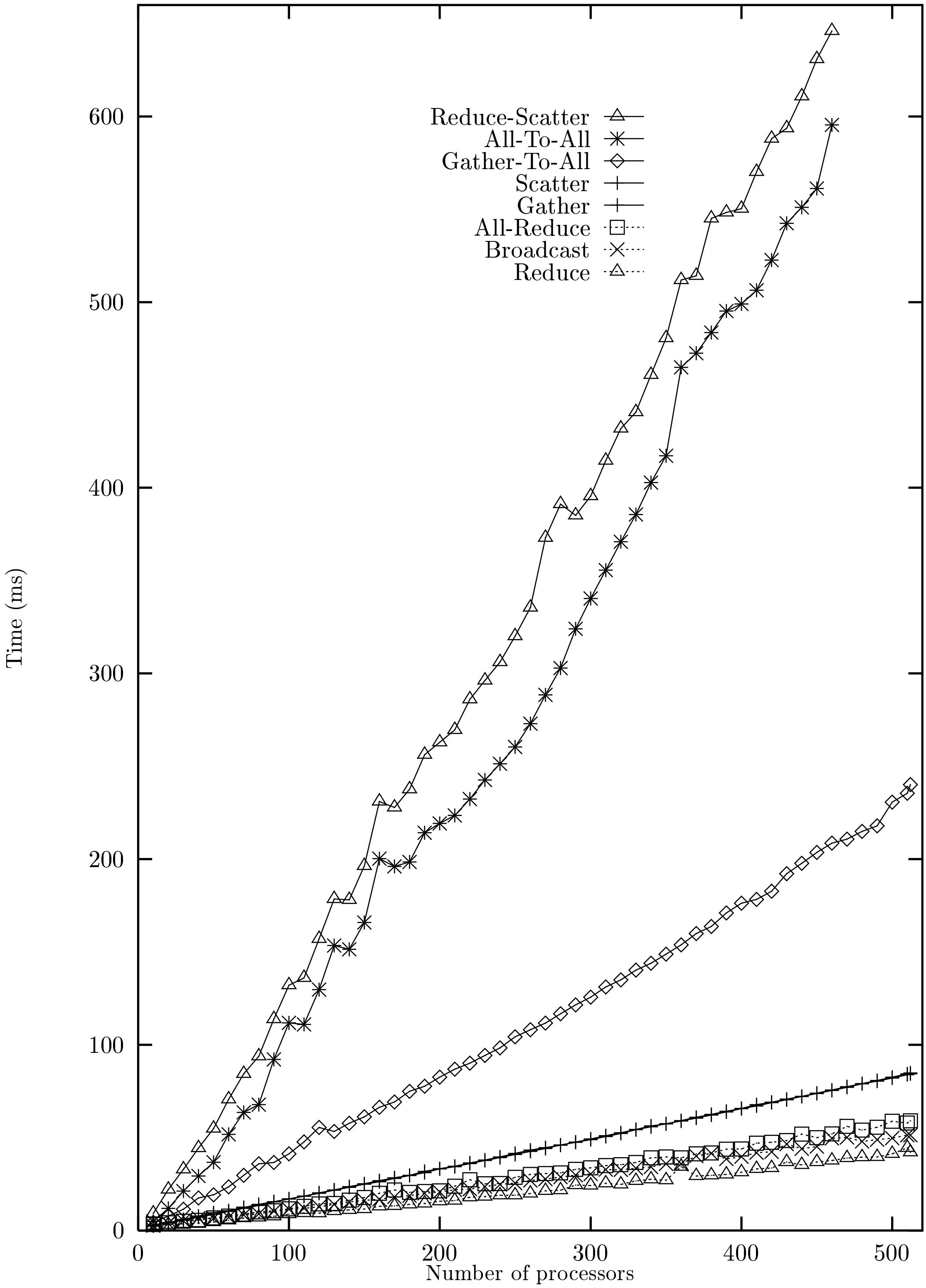


Figure 8: Scalability of our routines on the T3E with a 50 Kbyte message

6 Conclusion

Providing efficient implementations of the collective routines can be a difficult job, because the time required to perform a collective operation is a function of the size of the message, the number of processors involved, the latency of the network, the bandwidth of the network and the algorithm used. In addition, some algorithms require significant amounts of temporary storage that limit the size of messages that can be sent.

Based on the work in [8, 13], we have implemented eight of the MPI collective routines using MPI point-to-point communication routines with algorithms designed to be efficient for large messages. The performance of our implementations of these collective routines is compared with the vendor implementations on the Cray T3E-600, the Cray Origin 2000 and on the IBM SP. Many of our implementations significantly outperformed vendor implementations on the T3E and Origin for large messages. On the SP, only our implementation of the broadcast significantly outperformed IBM's implementation for large messages. Observe that the performance of our implementations depends on the performance of the vendor's implementation of the MPI point-to-point communication routines that we used. A "lower level" implementation of the algorithms we used might also enhance their performance (for example using SHMEM on the Origin and the T3E).

7 Acknowledgments

We would like to thank SGI for allowing us to use their T3E-600 and Origin 2000 located in Eagan, Minnesota, USA.

Computer time on the Maui High Performance Computer Center's P2SC was sponsored by the Phillips Laboratory, Air Force Material Command, under cooperative agreement number F29601-93-2-0001. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of Phillips Laboratory or the U.S. Government.

References

[1] Cray Research Web Server. http://www.cray.com.

[2] IBM Web Server. http://www.austin.ibm.com/hardware/largescale.

[3] Maui High Performance Computing Center Web Server. http://www.mhpcc.edu.

[4] Paragon Fortran System Calls Reference Manual. http://quest.cc.purdue.edu/Paragon/Docs/.

[5] CRAY T3E Programming with Coherent Memory Streams. Technical report, Cray Research, December 1996.

[6] Origin Servers. Technical report, Silicon Graphics, April 1997.

[7] A. Anderson, J. Brooks, C. Grassl, and S. Scott. Performance of the CRAY T3E Multiprocessor. In *Proceedings of SC97*, 1997.

[8] M. Barnett, S. Gupta, D. G. Payne, L. Shuler, R. van de Geijn, and J. Watts. Building a High-Performance Collective Communication Library. In *Supercomputing'94*, Washington D. C., November 1994. IEEE Computer Society Press.

[9] J. J. Dongarra and R. C. Whaley. The BLACS v1.0 User's Guide.

[10] J. Fier. *Performance Tuning Optimization for Origin 2000 and Onyx 2.* Silicon Graphics, 1996. http://techpubs.sgi.com.

[11] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard. http://www-c.mcs.anl.gov/mpi/mpich/.

[12] G. R. Luecke, B. Raffin, and J. J. Coyle. Comparing the Scalability of the Cray T3E-600 and the Cray Origin 2000 Using SHMEM Routines. *The Journal of Performance Evaluation and Modelling for Computer Systems*, December 1998. http://hpc-journals.ecs.soton.ac.uk/PEMCS/.

[13] P. Mitra, D. Payne, L. Shuler, R. van de Geijn, and J. Watts. Fast Collective Communication Libraries, Please. In *Proceedings of the Intel Supercomputing Users' Group Meeting.* 1995. http://www.cs.utexas.edu/users/rvdg/intercom/pubs.html.

[14] S. L. Scott and G. M. Thorson. The Cray T3E Network: Adapatative Routing in a High Perfromance 3D Torus. In *HOT Interconnects*, Stanford University, August 1996.

[15] M. Snir, S. W. Otto, S. Huss-Lederman, D. W. Walker, and J. Dongarra. *MPI, The Complete Reference.* Scientific and Engineering Computation. The MIT Press, 1996.