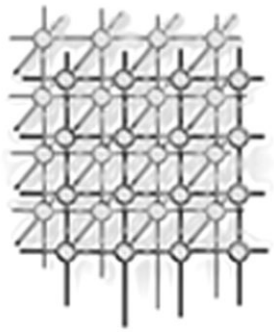
CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
*Concurrency Computat.: Pract. Exper.* 2004; **16**:1037–1060 (DOI: 10.1002/cpe.796)

**Performance**



# The performance and scalability of SHMEM and MPI-2 one-sided routines on a SGI Origin 2000 and a Cray T3E-600

Glenn R. Luecke*,†, Silvia Spanoyannis and Marina Kraeva

*Iowa State University, Ames, IA 50011-2251, U.S.A.*

**SUMMARY**

**This paper compares the performance and scalability of SHMEM and MPI-2 one-sided routines on different communication patterns for a SGI Origin 2000 and a Cray T3E-600. The communication tests were chosen to represent commonly used communication patterns with low contention (accessing distant messages, a circular right shift, a binary tree broadcast) to communication patterns with high contention (a 'naive' broadcast and an all-to-all). For all the tests and for small message sizes, the SHMEM implementation significantly outperformed the MPI-2 implementation for both the SGI Origin 2000 and Cray T3E-600. Copyright © 2004 John Wiley & Sons, Ltd.**

KEY WORDS: parallel computers; MPI-2 one-sided routines; SHMEM routines; SGI Origin 2000; Cray T3E-600; MPI-2 library; SHMEM library

## 1. INTRODUCTION

In 1997, MPI-2 [1,2] was introduced. MPI-2 includes one-sided communication routines, parallel I/O, dynamic process management, and other features introduced to make the previous MPI-1 more robust and convenient to use. MPI-1 [3,4] contains only two-sided point-to-point routines. This means that the program executing on the sending process must execute a send operation, and the process receiving the data must execute a receive operation. This is not the case for one-sided communication routines. There is just one routine executed by either the sending process or by the receiving process, but never by both.

*Correspondence to: Glenn R. Luecke, Iowa State University, Ames, IA 50011-2251, U.S.A.
†E-mail: grl@iastate.edu

Published online 10 June 2004

*Received 20 November 2001*
*Revised 10 September 2002*
*Accepted 13 December 2002*

The Cray shared memory library (SHMEM) [5–7] was originally developed to take advantage of the logically shared memory on Cray T3D machines. Later SHMEM implementations were made available on distributed memory parallel computers, such as Cray T3E, and on SGI CC-NUMA shared memory parallel computers. SHMEM is a collection of routines which provide low-latency, high-bandwidth communication, atomic memory operations, one-sided communication operations, collective operations (broadcast, reduction, etc.), and synchronization barriers among processes. SHMEM routines can be also used in programs that perform computations in separate address spaces and that explicitly pass data to and from different processes, which are also called processing elements (PEs). Data transfer, for both MPI-2 and SHMEM, is implemented through *put* operations, which transfer data to a different PE, and *get* operations which transfer data from a different PE.

The primary purpose of this paper is to compare the performance and scalability of SHMEM and MPI-2 one-sided routines on five communication patterns for a Cray T3E-600 and for a SGI Origin 2000. The communication tests were chosen to represent commonly used communication patterns with low contention (accessing distant messages, a circular right shift, a binary tree broadcast) to communication patterns with high contention (a 'naive' broadcast and an all-to-all) [8,9]. All tests were written in Fortran 90. The MPI-2 tests use the mpi_get and mpi_put communication routines, and the mpi_barrier, mpi_win_fence routines for synchronization. The SHMEM tests use the shmem_get and shmem_put communications routines, and the shmem_barrier_all and shmem_fence routines for synchronization.

This work is similar to [9] and uses the same SHMEM tests. For this paper these SHMEM tests were implemented using MPI-2 one-sided routines. The structure of the MPI-2 one-sided routines is more complex than that of the SHMEM routines. Implementing these five tests using MPI-2 one-sided routines was sometimes challenging, especially for the binary tree broadcast test.

No MPI collective communication routines, and no SHMEM collective routines were used. For example, the mpi_bcast and shmem_broadcast routines were not used for the broadcast test, which was implemented using the mpi_get and the shmem_get routines. This was done to directly compare the performance of the SHMEM and MPI-2 one-sided communication routines.

Tests were run with 16, 32, ..., 256 processors for both the SGI Origin 2000 and for the T3E-600. Large numbers of processors were used to evaluate the scalability of the SHMEM and MPI-2 implementations. All messages used 8 bytes reals. The tests were run with messages of size 8 bytes and 10 Kbytes for the 'naive' broadcast test; with 8 bytes, 1 Kbyte and 10 Kbytes for the all-to-all test; and for 8 bytes, 10 Kbytes and 1 Mbyte for the other tests.

The Origin 2000 used [10–13] was a 256 processor (128 nodes) machine located in Eagan, MN, with MIPS R12000 processors, running at 300 MHz. Each node consists of two processors sharing a common memory. There are two levels of cache: a $32 \times 1024$ byte first-level instruction cache and a $32 \times 1024$ byte first-level data cache, and an unified $8 \times 1024 \times 1024$ byte second-level cache for both data and instructions. The communication network is a hypercube for up to 16 nodes and is called a 'fat bristled hypercube' for more than 16 nodes since multiple hypercubes are interconnected via a CrayLink Interconnect. Notice that throughout this paper 1 Kbyte means $10^3$ bytes and 1 Mbyte means $10^6$ bytes. For all tests, the IRIX 6.5 operating system, the Fortran 90 compiler version 7.3 with the -03 -64 compiler options, and the MPI library version 1.4.0.1 were used.

The Cray T3E-600 used [14–16] was a 512 processor machine located in Chippewa Falls, WI. Each processor is a DEC Alpha EV5 microprocessor running at 300 MHz with a peak theoretical performance of 600 Mflop/s. There are two levels of cache: $8 \times 1024$ byte first-level instruction and

data caches and a $96 \times 1024$ byte second-level cache for both data and instructions. The communication network of both machines is a three-dimensional bi-directional torus. For all tests, the UNICOS/mk 2.0.5 operating system, the Fortran 90 compiler version 3.4.0.0 with the -O3 compiler option and the MPI library version 1.4.0.0.2 were used.

All tests were executed on machines dedicated to running only these tests. In order to achieve better communication performance on the T3E-600, Cray recommends turning the stream buffers off (set SCACHE_D_STREAMS 0), and setting buffering to 4096 (set MPI_BUFFER_MAX 4096).

Section 2 introduces the timing methodology employed, and Section 3 presents each of the tests and performance results. The conclusions are discussed in Section 4. Appendix A presents both the SHMEM and MPI-2 versions of the binary tree broadcast algorithm.

## 2. TIMING METHODOLOGY

Timings were done by first flushing the cache [17,18] on all processors by changing the values in the real array `flush(1:ncache)`, prior to timing the desired operation. The value `ncache` was chosen so the size of `flush` was the size of the secondary cache, 8 Mbytes for the Origin 2000, and 96 Kbytes for the T3E-600.

All the SHMEM tests were timed using the following code:

```
integer, parameter :: ntrial=51, n=1
                      !or 125, 1250, 125000 depending on the test
real*8, dimension(ntrial) ::  time, pe_time
real*8 :: A(n)
call random_number(A)
do k = 1, ntrial
   flush(1:ncache) = flush(1:ncache) + 0.1
   call shmem_barrier_all()
   time1 = timef()

   ... SHMEM code to be timed ...

    pe_time(k) = timef() - time1
    call shmem_barrier_all()
    A(1) = flush(1)
 end do
 print *,flush(1) + A(1)
 call shmem_real8_max_to_all(time,pe_time,ntrial,0,0,p,pWrk8,pSync1)
```

All the MPI-2 tests were timed using the following code:

```
integer, parameter :: ntrial=51, n=1
                      !or 125, 1250, 125000 depending on the test
real*8 A(n)
integer size_window, disp_unit, info
integer communicator, win, ierror, sizeofreal
real*8, dimension(ntrial) ::  time, pe_time
```

```
!Creation of the processor memory window
call mpi_type_extent(mpi_real8,sizeofreal,ierror)
size_window = n * sizeofreal
call mpi_win_create(A,size_window,sizeofreal, &
                    info,mpi_comm_world,win,ierror)
call mpi_win_fence(0,win,ierror)

do k = 1, ntrial
   flush(1:ncache) = flush(1:ncache) + 0.1
   call mpi_barrier(mpi_comm_world,ierror)
   time1 = mpi_wtime()

   ... MPI-2 code to be timed ...

   call mpi_win_fence(0,win,ierror)
   pe_time(k) = mpi_wtime() - time1
   call mpi_barrier(mpi_comm_world,ierror)
   A(1) = flush(1)
end do
print *,flush(1) + A(1)
call mpi_reduce(pe_time,time,ntrial,mpi_real8,mpi_max, &
                0,mpi_comm_world,ierror)
call mpi_win_free(win,ierror)
```

The first call to shmem_barrier/mpi_barrier guarantees that all processors reach this point before they each call the wall-clock timer, timef/mpi_wtime. The second call to a synchronization barrier is to ensure that no processor starts the next iteration (flushing the cache) until all processors have completed executing the 'SHMEM/MPI-2 code to be timed'. The first call to mpi_win_fence is required to begin the synchronization epoch for RMA operations. The second call to mpi_win_fence is needed to ensure the completion of all RMA calls in the window `win` since the previous call to mpi_win_fence. To prevent the compiler's optimizer from removing the cache flushing from the $k$-loop in SHMEM/MPI-2 code, the line `A(1) = flush(1)` was added, where `A` is an array involved in the communication. The tests are executed `ntrial` times and the values of the differences in times on each participating processor are stored in `pe_time`. The call to shmem_real8_max_to_all/mpi_reduce calculates the maximum of `pe_time(k)` over all participating processors for each fixed `k` and places this maximum in `time(k)` for all values of $k$. Thus, time($k$) is the time to execute the test for the $k$th trial. The print statement is added to ensure that the compiler does not consider the timing loop to be 'dead code' and does not remove it.

Figure 1 shows the performance data in milliseconds on the SGI Origin 2000 for 51 executions of the MPI-2 all-to-all test with a message of size 10 Kbytes using 32 MPI processes. For all the tests the first measured time was usually larger than most of the subsequent times. This is likely due to start-up overheads. This is illustrated in Figure 1, where the first measured time is 20 ms compared to the average times which is 12 ms. Timings for a number of trials were taken to view variation among trials and consistency of the timing data. Taking `ntrial` = 51 (the first timing was always thrown away), provided enough trials to do this for this study. Some time trials would be significantly larger than others and would significantly affect the average over all trials (see [8,9] and Figure 1). In Figure 1, the dotted horizontal line at 14 ms shows the average over 51 trials. It was the authors' opinion that 'spikes' should be removed so that the data will reflect the times one will usually obtain. All data in this report

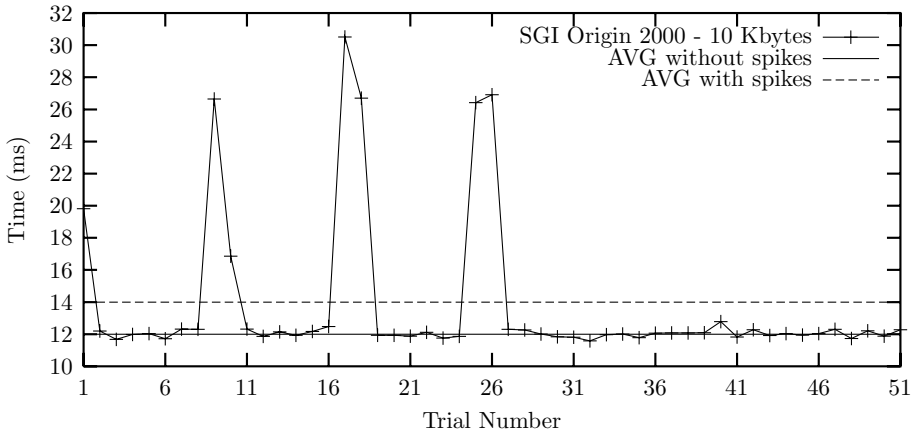



Figure 1. Timing data (in milliseconds) on the SGI Origin 2000 for 51 executions of the MPI-2 all-to-all test with 10-Kbyte messages using 32 MPI processes.

have been filtered by the following process [17,18]. The first timing is removed. The median of the remaining data 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. The authors felt that removing more than 10% of the data would not be appropriate, so in these cases only the 10% of the largest times were removed. Thus, taking 51 ntrials leads to the removal of at most five of the largest times. An average was calculated from the filtered data and this is what is presented for each test in this paper.

For the data shown in Figure 1 this filtering process gives a time of 14 ms whereas the average over all data is 12 ms (20% increase!). For all the tests, the filtering process led to the removal of 6% of performance data for the Cray T3E-600 and 10% for the Origin 2000.

## 3. COMMUNICATION TESTS AND PERFORMANCE RESULTS

### 3.1. Test 1: accessing distant messages

The purpose of this test is to determine the performance differences of sending messages between 'close' processors and 'distant' processors using SHMEM and MPI-2 one-sided routines. In a 'perfectly scalable' computer no difference would occur. This test measures the time required for processor 0 to get a message of size n from processor $j$, for $j = 1, 2, 3, \ldots, 255$ when 256 processors were used on both the Origin 2000 and the Cray T3E-600.

The SHMEM code for this test is implemented as follows:

```
if (myrank == 0) then
   call shmem_get8(A,A,n,j)
end if
```

where `myrank` is the rank of the executing processor, and the array `real*8 A(n)` represents the memory target/source buffers on processor 0 and $j$.

The MPI-2 code for processor 0 to get a message of size `n` from processor $j$'s window `win` is implemented as follows:

```
if (myrank == 0) then
   call mpi_get(A,n,mpi_real8,j,0,n,mpi_real8,win,ierror)
end if
```

where `myrank` is the rank of the executing processor, and the array `real*8 A(n)` is the destination buffer of processor `myrank`.

Performance data of this test are shown in Figure 2 for the Origin 2000, and in Figure 3 for the T3E-600. For 8-byte messages, on the Origin 2000 the MPI-2 implementation is about eight times slower than the SHMEM implementation. For 10-Kbyte and 1-Mbyte messages, the MPI-2 and SHMEM implementations are very close in performance. For all message sizes, performance data show 'high points' indicating an increase in time to send a message from processor of rank 0 to processor of the rank associated with the 'high point'.

Experiments showed that these 'high points' are not repeatable. It appears that these 'high points' randomly show up in correspondence to different processors. Recall that process assignment to physical processors is dynamically done when a job is submitted.

On the T3E-600 the SHMEM implementation significantly outperforms the MPI-2 implementation in all cases tested. Notice that the difference in time between the two implementations becomes smaller as the message size increases. For 8-byte messages, MPI-2 is about 24 times slower than SHMEM. For 10-Kbyte messages, MPI-2 is about four times slower than SHMEM. For 1-Mbyte messages, SHMEM and MPI-2 are very close in performance.

On the T3E-600, both MPI-2 and SHMEM implementations show good scalability. Notice that on the T3E-600 there is not much difference in time between messages sent to 'close' processors and messages sent to 'distant' processors and one cannot see the architecture of the communication network (a three-dimensional bi-directional torus) in the data.

With the Origin 2000, the SHMEM data shows less variability than the MPI-2 data. As was the case for the T3E-600, one cannot see the architecture of the communication network of the Origin 2000 in the data.

Table I shows the average performance results of the filtered data for this test on the Origin 2000 and on the T3E-600, where the average is taken over all processors. Notice that for 8-byte messages, the SHMEM implementation on the T3E-600 is 1.5 times slower than the SHMEM implementation on the Origin 2000. Notice that for the Origin 2000 for 8-byte messages, the average time reported in Table I is 0.816 ms for the MPI-2 implementation. Looking at the data in Figure 2(a), one might expect this average time to be about 0.04 ms instead of 0.816 ms. The reason for the average being so large is that some of the 'high points' are huge (i.e. on the order of 50 ms). A similar behavior occurs for the Origin 2000 for 10 Kbyte and 1 Mbyte messages for the MPI-2 implementation. Timings on the Origin 2000 for the SHMEM ping-pong test do not exhibit this behavior. For the T3E, this behavior does not occur for either the SHMEM or the MPI-2 ping-pong test.

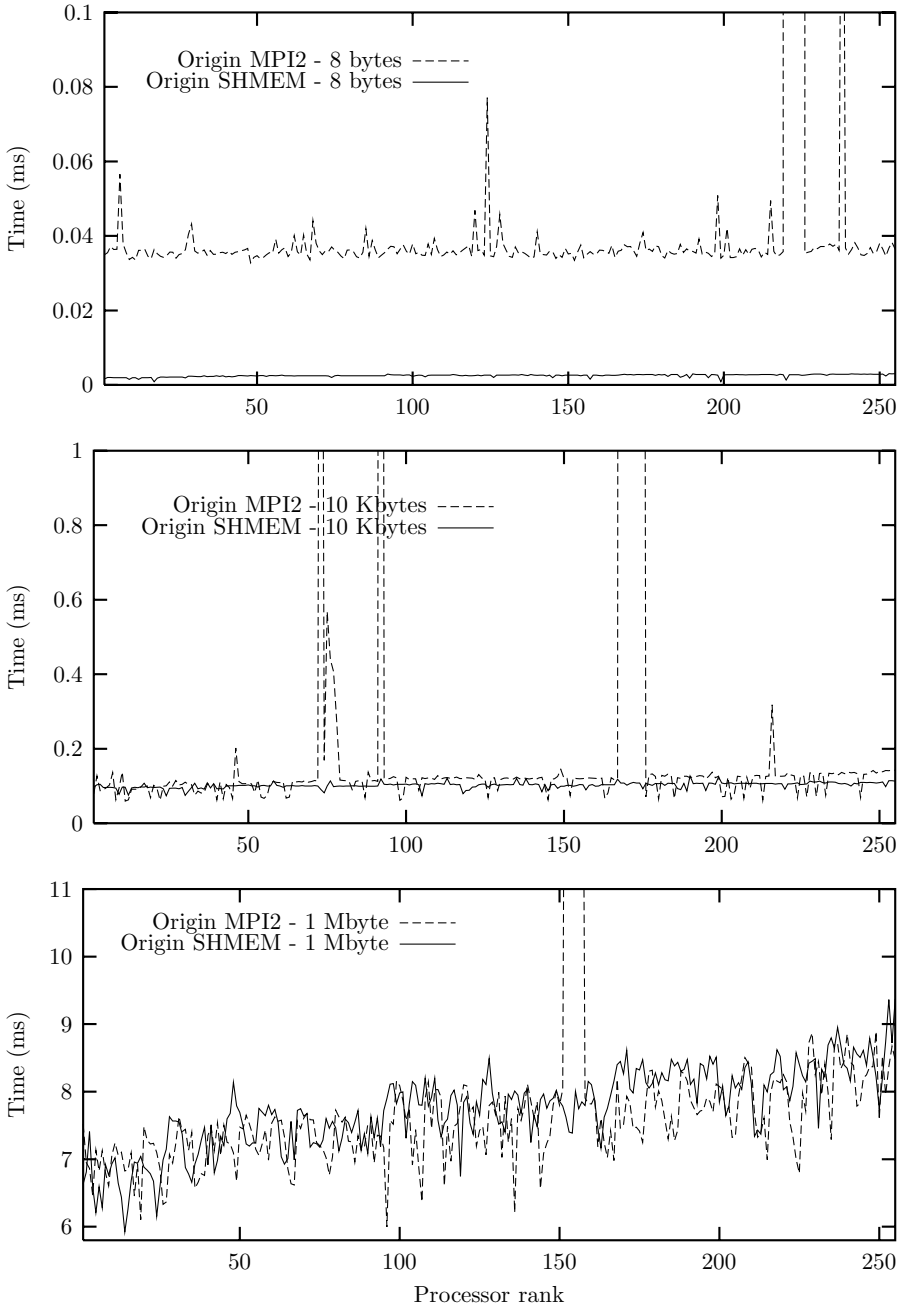



Figure 2. Accessing distant messages test for the SGI Origin 2000 with 8-byte, 10-Kbyte and 1-Mbyte messages.

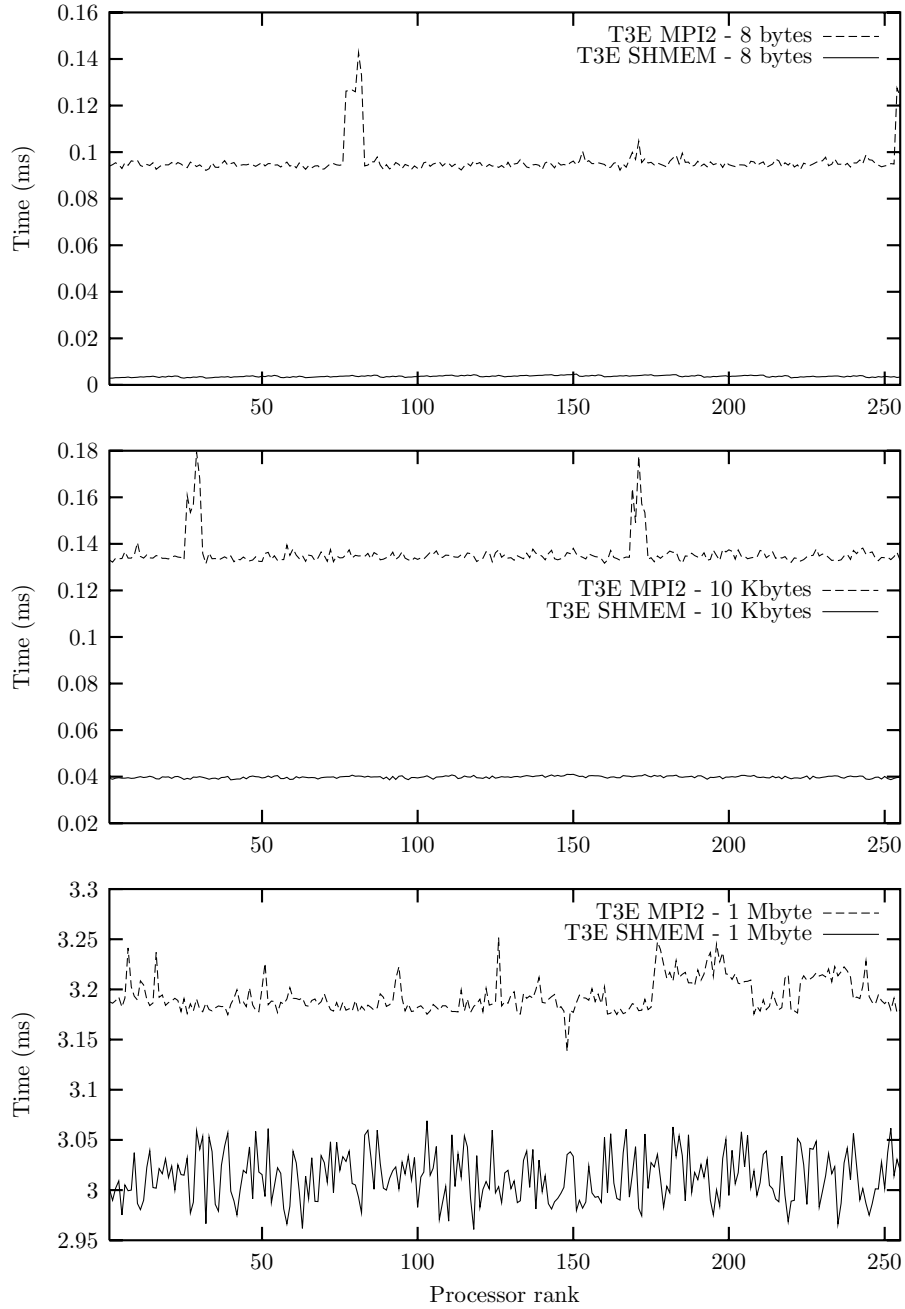



Figure 3. Accessing distant messages test for the T3E-600 with 8-byte, 10-Kbyte and 1-Mbyte messages.

Table I. Test 1: average times (in milliseconds) for the accessing distant messages test.

| | Origin SHMEM | Origin MPI-2 | T3E SHMEM | T3E MPI-2 |
|---|---|---|---|---|
| 8 bytes | 0.0025 | 0.816 | 0.0037 | 0.096 |
| 10 Kbytes | 0.102 | 1.24 | 0.04 | 0.136 |
| 1 Mbyte | 7.74 | 8.18 | 3.01 | 3.19 |

### 3.2. Test 2: circular right shift

The purpose of this test is to compare the performance and scalability of SHMEM and MPI-2 one-sided routines for the circular right shift operation. For this test, each processor gets a message of size `n` from its 'left' neighbor, i.e. from `modulo(myrank-1, p)`, where `myrank` is the rank of the executing processor. Since these operations can be done concurrently, one would expect the execution time for this test to be independent of the number of processors `p`.

The code for the SHMEM implementation of this test is

```
call shmem_get8(B,A,n,modulo(myrank-1,p))
```

where the array `real*8 B(n)` is the destination buffer of the executing processor, and the array `real*8 A(n)` is the source buffer on the processor of rank `modulo(myrank-1, p)`.

The code for the MPI-2 implementation of this test is

```
call mpi_get(B,n,mpi_real8,modulo(myrank-1,p),0,n,mpi_real8,win,ierror)
```

where the array `real*8 B(n)` is the destination buffer of the executing processor, and `win` is the memory window created on processor of rank `modulo(myrank-1, p)`.

Performance data of this test are shown in Figure 4. On the T3E-600 for 8-byte messages, the MPI-2 implementation is about five times slower than the SHMEM implementation. For 10-Kbyte and 1-Mbyte messages, the MPI-2 and SHMEM implementations perform about the same.

On the Origin 2000, for 8-byte messages, the SHMEM implementation gives much better performance than the MPI-2 implementation. For 10-Kbyte messages, both the SHMEM and MPI-2 implementations show times nearly the same until 80 processors. When more than 80 processors are used, MPI-2 and SHMEM present larger time variations. For 1-Mbyte messages, both SHMEM and MPI-2 implementations show similar performance. The unusually large times for 8-byte and 10-Kbyte messages reported for the MPI-2 implementation for 212 processors is likely due to a MPI implementation problem.

For all messages and for both MPI-2 and SHMEM implementations, the T3E-600 outperforms the Origin 2000. On the T3E-600, for all messages, both SHMEM and MPI-2 implementations show good scalability. On the Origin 2000, for 8-byte messages the MPI-2 implementation scales poorly. For 10-Kbyte messages, both MPI-2 and SHMEM implementations show poor scalability, while for 1-Mbyte messages, both implementations show good scalability.

Since some of the data for 8 byte messages is difficult to distinguish from zero, Table II shows the data for 16, 32, 64, 128 and 256 MPI processes.

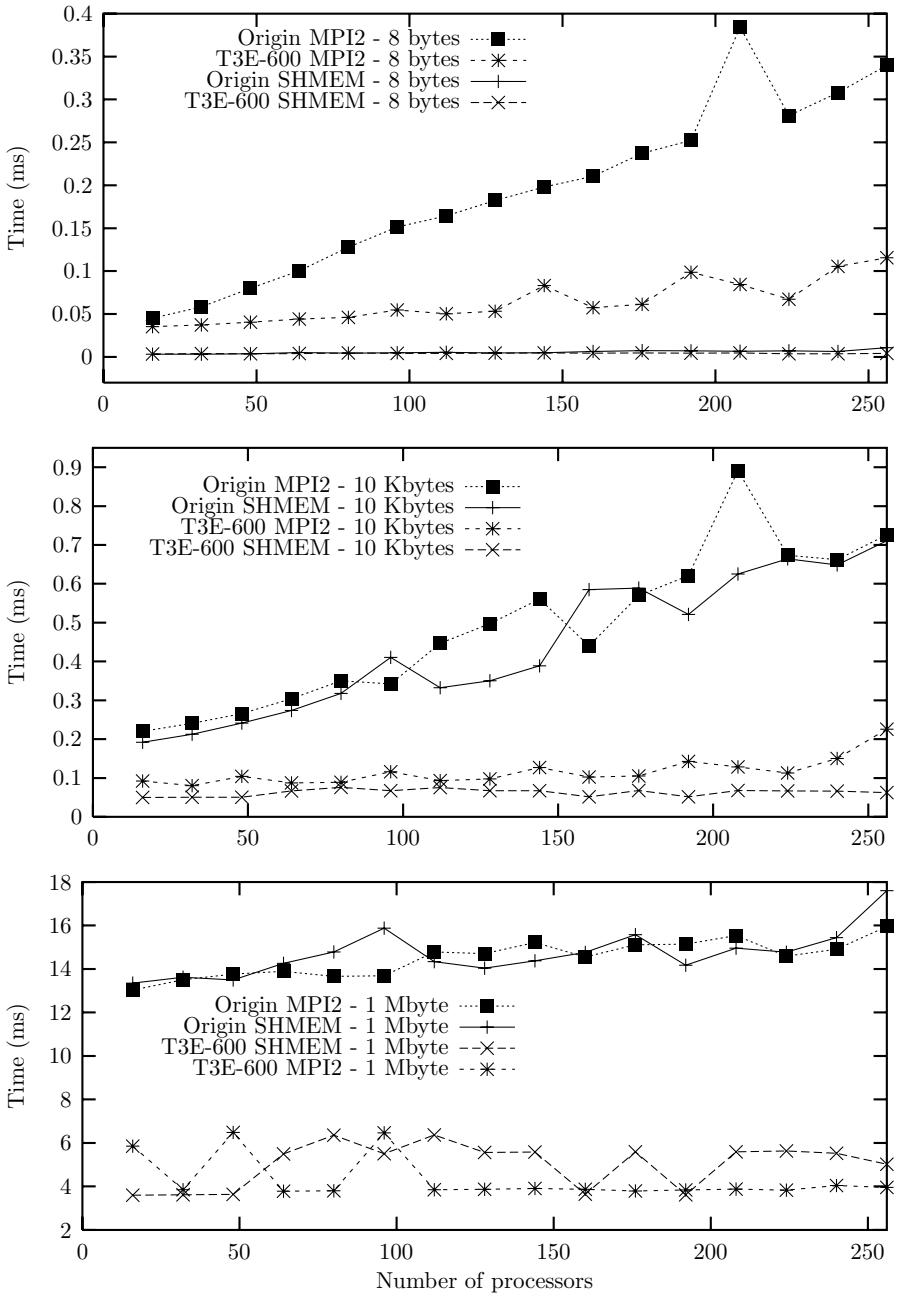



Figure 4. Circular right shift test with 8-byte, 10-Kbyte and 1-Mbyte messages.

Table II. Test 2: circular right shift for 8-byte messages (times in microseconds).

| | 16 PE | 32 PE | 64 PE | 128 PE | 256 PE |
|---|---|---|---|---|---|
| Origin SHMEM | 3.12 | 3.08 | 5.15 | 4.51 | 10.86 |
| Origin MPI-2 | 45.1 | 58.3 | 100.3 | 182.4 | 339.9 |
| T3E SHMEM | 3.36 | 3.84 | 3.77 | 4.32 | 4.02 |
| T3E MPI-2 | 35.3 | 37.2 | 44.1 | 53.2 | 115.7 |

### 3.3. Test 3: the 'naive' broadcast

The purpose of this test is to compare the performance and scalability of SHMEM and MPI-2 one-sided implementations of a 'naive' broadcast. This test is called 'naive' broadcast, because neither MPI-2 nor SHMEM collective routines (mpi_bcast and shmem_broadcast) are used. For this test, each processor with rank greater than 0 'gets' a message from processor 0. Since the window of memory of processor 0 only allows a single `get` or `put` operation at a time, one would expect the execution time for this test would grow linearly with the number of processors.

The SHMEM code for this test with `p` processors is

```
if ((0 < myrank) .and. (myrank < p)) then
     call shmem_get8(A,A,n,0)
end if
```

where `myrank` is the rank of the executing processor, and the array `real*8 A(n)` represents the memory destination/source buffers on processor `myrank` and 0. In the MPI-2 implementation of this test, each processor with `0 < myrank < p` gets a message of size `n` from the window `win` of processor 0.

The MPI-2 code of this test for `p` processors is

```
if ((0 < myrank) .and. (myrank < p)) then
     call mpi_get(A,n,mpi_real8,0,0,n,mpi_real8,win,ierror)
end if
```

where `myrank` is the rank of the executing processor, the array `real*8 A(n)` is the destination buffer of the executing processor, and `win` is the memory window created on processor of rank 0.

Performance data of this test are shown in Figure 5. For 8-byte messages, on the Origin 2000, the SHMEM implementation outperforms the MPI-2 implementation by a factor of 2–4. For 10-Kbyte messages, the SHMEM and MPI-2 implementations show similar performances.

For 8-byte messages, on the T3E-600, the SHMEM implementation is about three to five times faster than the MPI-2 implementation. SHMEM presents a more stable pattern than MPI-2. For 10-Kbyte messages, both implementations present similar performance.

For 8-byte messages, the T3E-600 outperforms the Origin 2000 for both MPI-2 and SHMEM implementations. For 10-Kbyte messages, the Origin 2000 presents slightly better performance than the T3E-600 for both SHMEM and MPI-2 implementations.

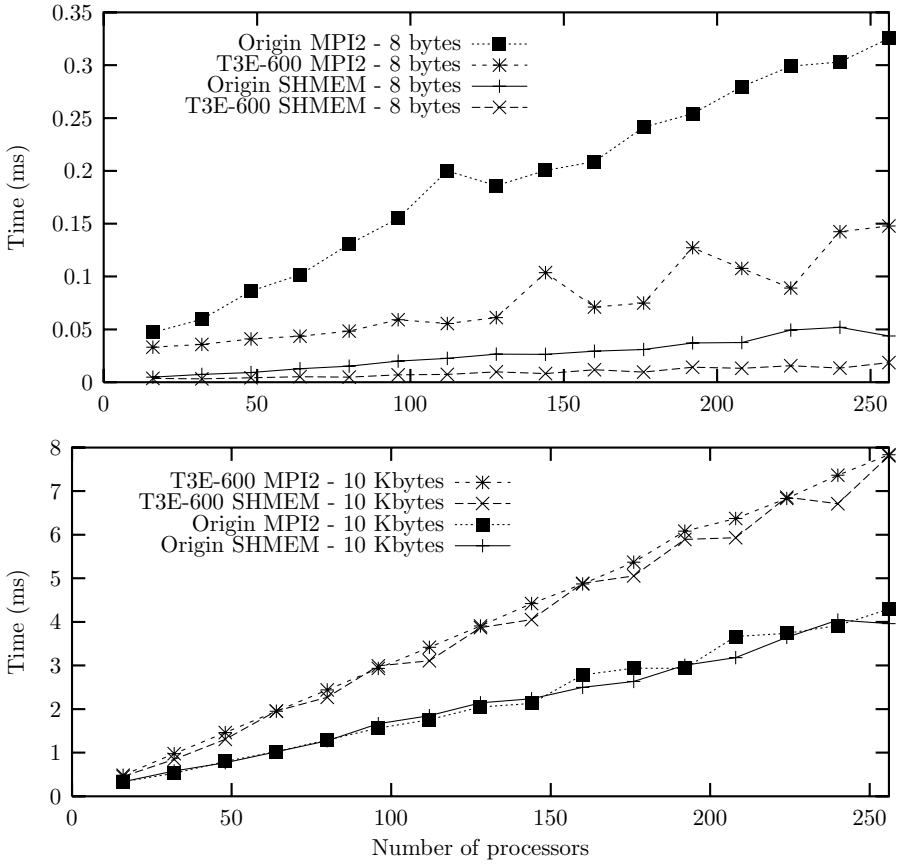



Figure 5. 'Naive' broadcast test with 8-byte and 10-Kbyte messages.

For 8-byte messages and for both machines, the SHMEM implementation shows better scalability than the MPI-2 implementation. Notice that in this case the MPI-2 implementation on the Origin 2000 scales poorly. For 10-Kbyte messages, both SHMEM and MPI-2 implementations show good scalability for both machines.

Table III shows times (in milliseconds) for the 'naive' broadcast test for 8-byte messages.

### 3.4. Test 4: the binary tree broadcast

The purpose of this test is to compare the performance and scalability of SHMEM and MPI-2 one-sided routines for a binary tree broadcast operation. This communication pattern is often used to implement

Table III. Test 3: the 'naive' broadcast for 8-byte messages (times in microseconds).

| | 16 PE | 32 PE | 64 PE | 128 PE | 256 PE |
|---|---|---|---|---|---|
| Origin SHMEM | 4.77 | 7.49 | 12.8 | 26.5 | 43.8 |
| Origin MPI-2 | 47.2 | 5.96 | 101.6 | 185.9 | 325.5 |
| T3E SHMEM | 3.61 | 3.2 | 5.16 | 9.76 | 18.6 |
| T3E MPI-2 | 33.2 | 35.8 | 43.5 | 61.2 | 147.9 |

other common operations such as reduce, gather and scatter [9,19]. Since a binary tree is used for sending messages one would expect executing time to increase logarithmically with the number of processors.

As explained in [8], when $\mathtt{p} = 2^n$, the SHMEM implementation of this test for broadcasting a message `A` can be described as follows.

(i) Using shmem_put8, processor 0 writes `A` into processor $p/2$'s memory. Next, processor 0 sends a message to processor $p/2$ using shmem_put8 to indicate that the transfer of `A` is done. To ensure that this last message arrives after writing `A` into processor $p/2$'s memory, processor 0 executes a shmem_fence between the two calls to shmem_put8.
(ii) Using shmem_int8_wait, processor $p/2$ waits for processor 0 to post the message signaling that processor 0 has written `A` into $p/2$'s memory. Processors 0 and $p/2$ write `A` into processor $p/4$'s memory and $3p/4$'s memory, respectively, before to execute a shmem_fence and to post a message indicating that the transfer of `A` is done.
(iii) The above pattern is repeated until all processors have received `A`.
(iv) A final synchronization executed by calling shmem_barrier_all ensures that the next broadcast does not begin until the first one has been completed.

The SHMEM algorithm for this test is listed in Appendix A. There are two MPI-2 versions of the SHMEM algorithm: one using mpi_get, see Appendix B, and the other using mpi_put, see Appendix C. The synchronization executed by calling the shmem_barrier_all routine in the SHMEM code, was performed using the mpi_barrier routine in the MPI-2 code. Synchronization among processors was implemented using the shmem_fence, and shmem_int8_wait routines in the SHMEM code, and the mpi_win_fence routine for the MPI-2 code.

In the MPI-2 implementations of the binary tree broadcast test, processors get messages from the windows of processors belonging to a 'higher level' in the tree. It is important that those processors get messages only after these messages have been written in the windows of the higher level processors. If this is not guaranteed, processors may get invalid data from the windows of the higher level processors.

The binary tree broadcast test written with mpi_get required the use of a synchronization variable, `syn`, in addition to the mpi_win_fence routine. Once a processor gets the data from the window of the upper level processor, it must check that the get operation has completed before notifying the target processor to start getting the data. This checking was implemented using a `do while` statement that

Table IV. Test 4: the binary tree broadcast for 8-byte messages (times in milliseconds).

| | 16 PE | 32 PE | 64 PE | 128 PE | 256 PE |
|---|---|---|---|---|---|
| Origin SHMEM | 0.033 | 0.049 | 0.07 | 0.094 | 0.179 |
| Origin MPI2 GET | 0.59 | 0.79 | 1.03 | 1.45 | 2.57 |
| Origin MPI2 PUT | 33 | 36.6 | 34.7 | 35 | 40.8 |
| T3E SHMEM | 0.026 | 0.032 | 0.037 | 0.045 | 0.052 |
| T3E MPI2 GET | 0.26 | 0.32 | 0.39 | 0.47 | 0.6 |
| T3E MPI2 PUT | 33.2 | 33.3 | 33.3 | 33.4 | 33.5 |

verifies if data from a higher level has been received. To run this test, the synchronization variable `syn`, was sent/received between processors using mpi_send and mpi_recv routines. This was done to avoid calling mpi_win_fence that requires a high overhead.

The binary tree broadcast test written with mpi_put is slightly different from the mpi_get version. Processors in the higher level of the tree, put data in the window of selected lower level processors. These lower level processors check if the *put* operation has completed, and then they start a new *put* operation into the windows of the target lower level processors in the tree. This checking was done using the `do while` statement described for the mpi_get implementation. For this version there was no need to use the synchronization variable `syn`.

Figures 6–8 present the performance results of this test. Notice the different time scales in Figures 6 and 7. On the T3E-600, for 8-byte and 10-Kbyte messages, the MPI-2 'put' implementation is significantly slower than the MPI-2 'get' implementation and than the SHMEM 'get' implementation. The SHMEM 'get' implementation is much faster than MPI-2 'get' one. For 1-Mbyte messages, the MPI-2 'get' implementation is about 1.5 times slower than the MPI-2 'put'implementation, and three times slower then SHMEM 'get'one. The SHMEM 'get' implementation is about 1.5 times faster than MPI-2 'put' one.

On the Origin 2000, for 8-byte messages, the MPI-2 'put' implementation is significantly slower than the MPI-2 'get' and SHMEM 'get' implementations. The 'high point' on the MPI-2 'put' graph on the Origin 2000 sometimes appears and sometimes does not. The MPI-2 'get' implementation is quite close in time to the SHMEM 'get' implementation, which shows the best performance. For 10-Kbyte messages, the MPI-2 'put' implementation is about 13 times slower than the MPI-2 'get' implementation and 30 times slower than SHMEM 'get' one. The SHMEM 'get' implementation is about twice as fast as the MPI-2 'get' one. For 1-Mbyte messages, the SHMEM implementation shows the worst performance. The SHMEM implementation is 1.5 times slower than the MPI-2 'put' one and twice as slow as the MPI-2 'get' one. The MPI-2 'put' implementation is about 1.4 times slower than the MPI-2 'get' one.

With the exception of 1-Mbyte messages, the SHMEM implementation shows better scalability than the MPI-2 'put' and the MPI-2 'get' implementations on both machines. For all message sizes, the MPI-2 'get' implementation shows better scalability than the MPI-2 'put' implementation.

Table IV shows times (in ms) for the binary tree broadcast test for 8-byte messages.

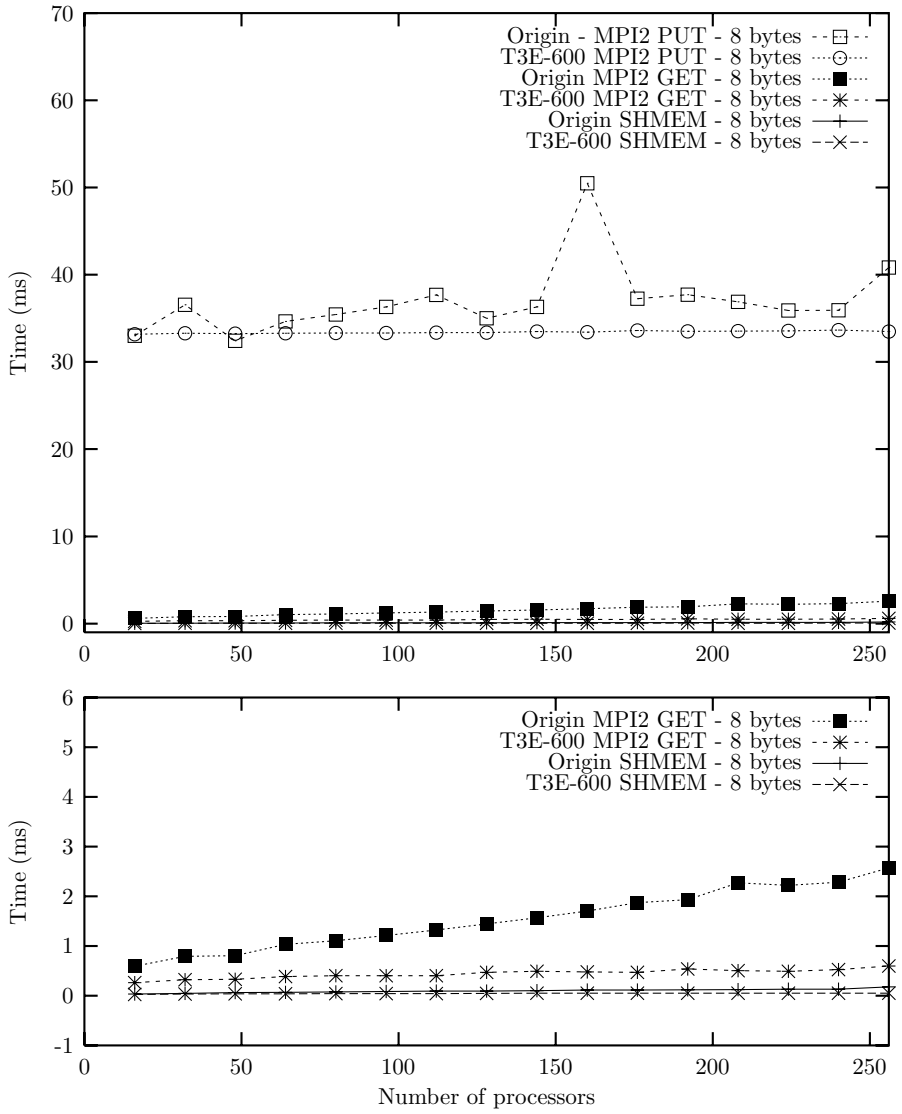



Figure 6. The binary tree broadcast test with 8-byte messages.

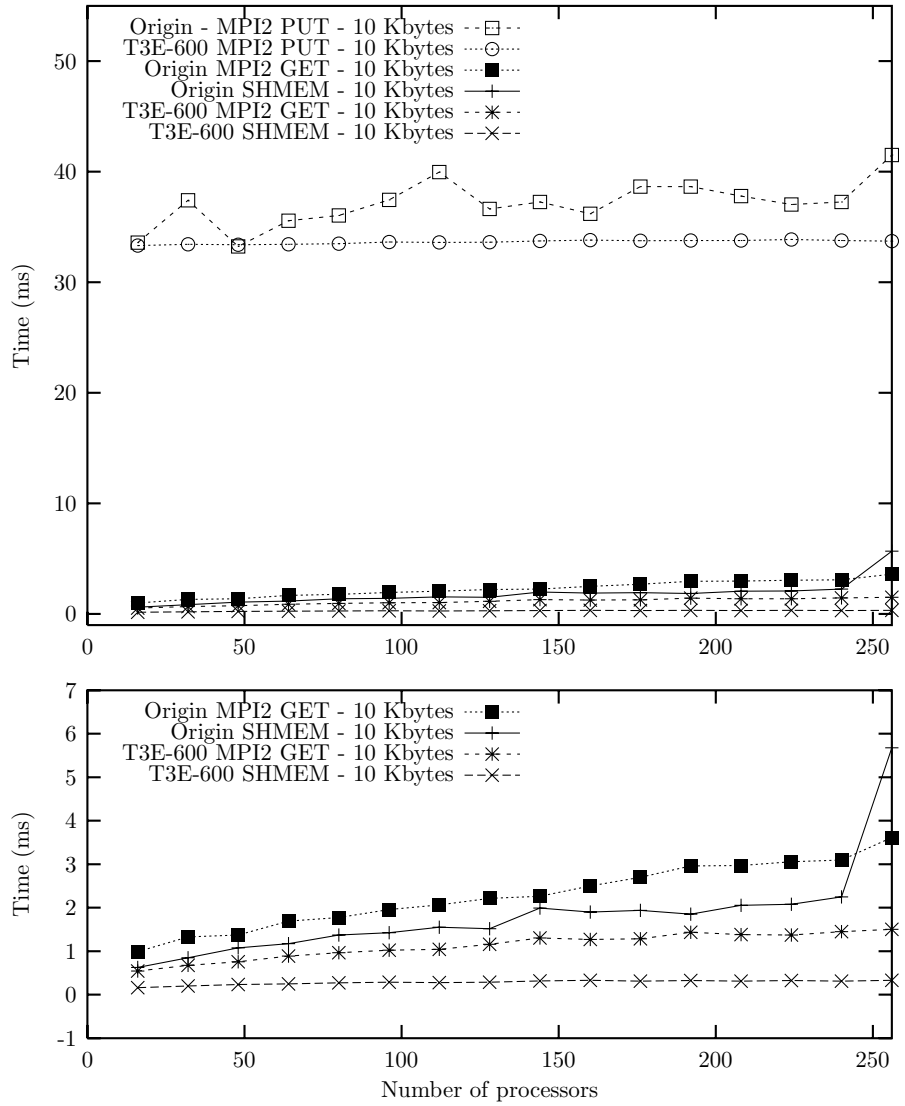



Figure 7. The binary tree broadcast test with 10-Kbyte messages.

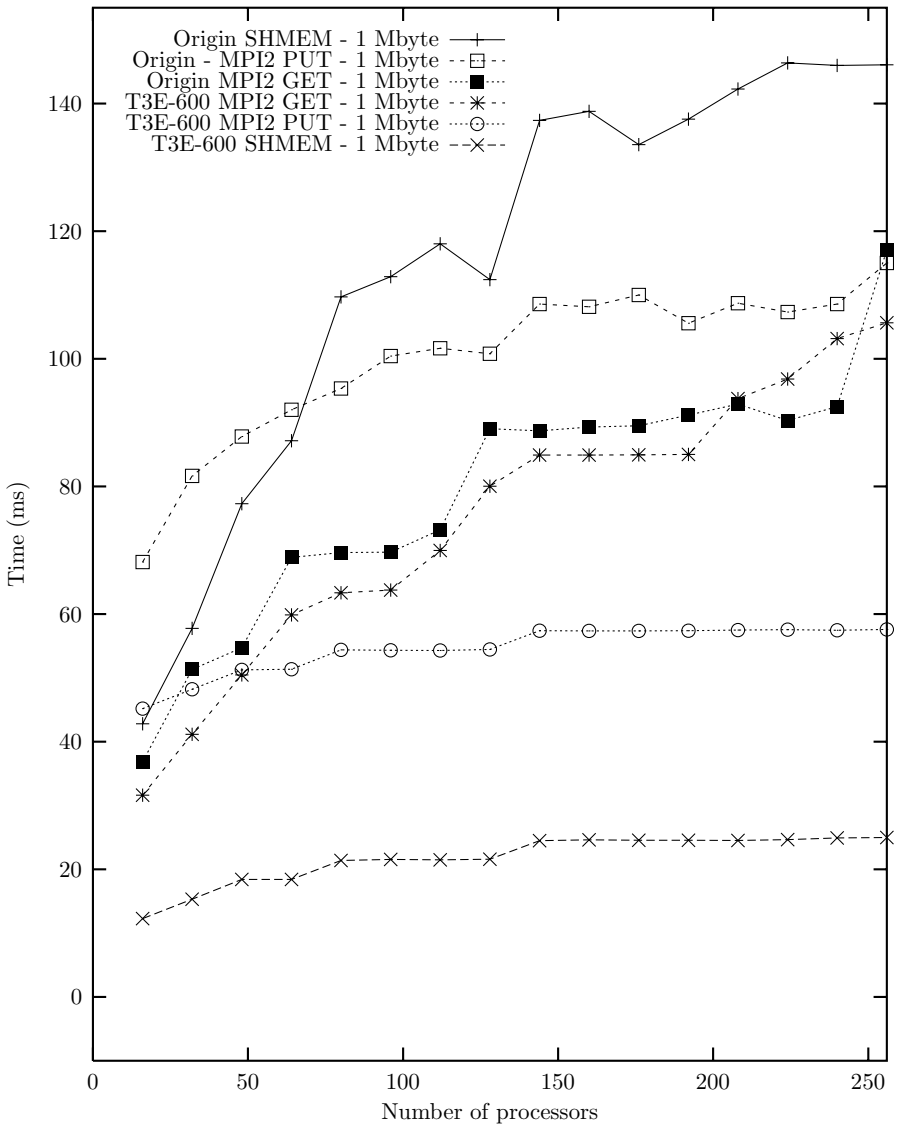



Figure 8. The binary tree broadcast test with 1-Mbyte messages.

### 3.5. Test 5: the all-to-all

The purpose of this test is to compare the performance and scalability of SHMEM and MPI-2 one-sided routines for the high contention all-to-all communication pattern. For this test, each processor gets a message from each of the other processors. Thus, in a 'perfectly scalable' computer the execution time for this test would increase linearly with the number of processors.

As in [8], care must be taken when implementing the all-to-all test to minimize contention. For example, if the first operation performed by all processors is to *get* data from one processor, then there will be high contention at this processor. If one were to write

```
do j=0, p-1
   if (j .NE. myrank) then
      call mpi_get(B(n*j),n,mpi_real8,j,n*myrank,n,mpi_real8,win,ierror)
   endif
end do
```

then all processors will first access elements from the window `win` of processor 0, next all processors will access elements from the window of processor of rank $1, 2, \ldots$. To avoid this unnecessary high contention, each processor can instead access elements from the window of processor `myrank-1, myrank-2, ...`. Since there are `p` processors of rank `0, 1, ..., p-1`, processors access elements from the window of `modulo(myrank-j, p)` for $j = 1, 2, \ldots, p-1$. Therefore, the code used for the MPI-2 implementation of this test is as follows:

```
do j = 1, p-1
   i = modulo(myrank-j,p)
   call mpi_get(B(n*i),n,mpi_real8,i,n*myrank,n,mpi_real8,win,ierror)
end do
```

where `myrank` is the rank of the executing processors, `i` is the rank of the target processor, `n` is the message size, `real*8 B` is the source buffer and `n*myrank` represents the displacement in the memory window `win`.

The SHMEM implementation of this test is as follows:

```
do j = 1, p-1
   i = modulo(myrank-j,p)
   call shmem_get8(B(n*i),A(n*myrank),n,i)
end do
```

where `myrank` is the rank of the executing processors, `i` is the rank of the target processor, `n` is the message size, `real*8 B` is the source buffer and `real*8 A` is the destination buffer.

Figure 9 presents the results of this test. Notice that the MPI-2 implementation for the Origin 2000 does not scale or perform well. For all other cases, both the MPI-2 and SHMEM implementations performed and scaled about the same.

Table V, shows times (in ms) for the all-to-all test for 8-byte messages.

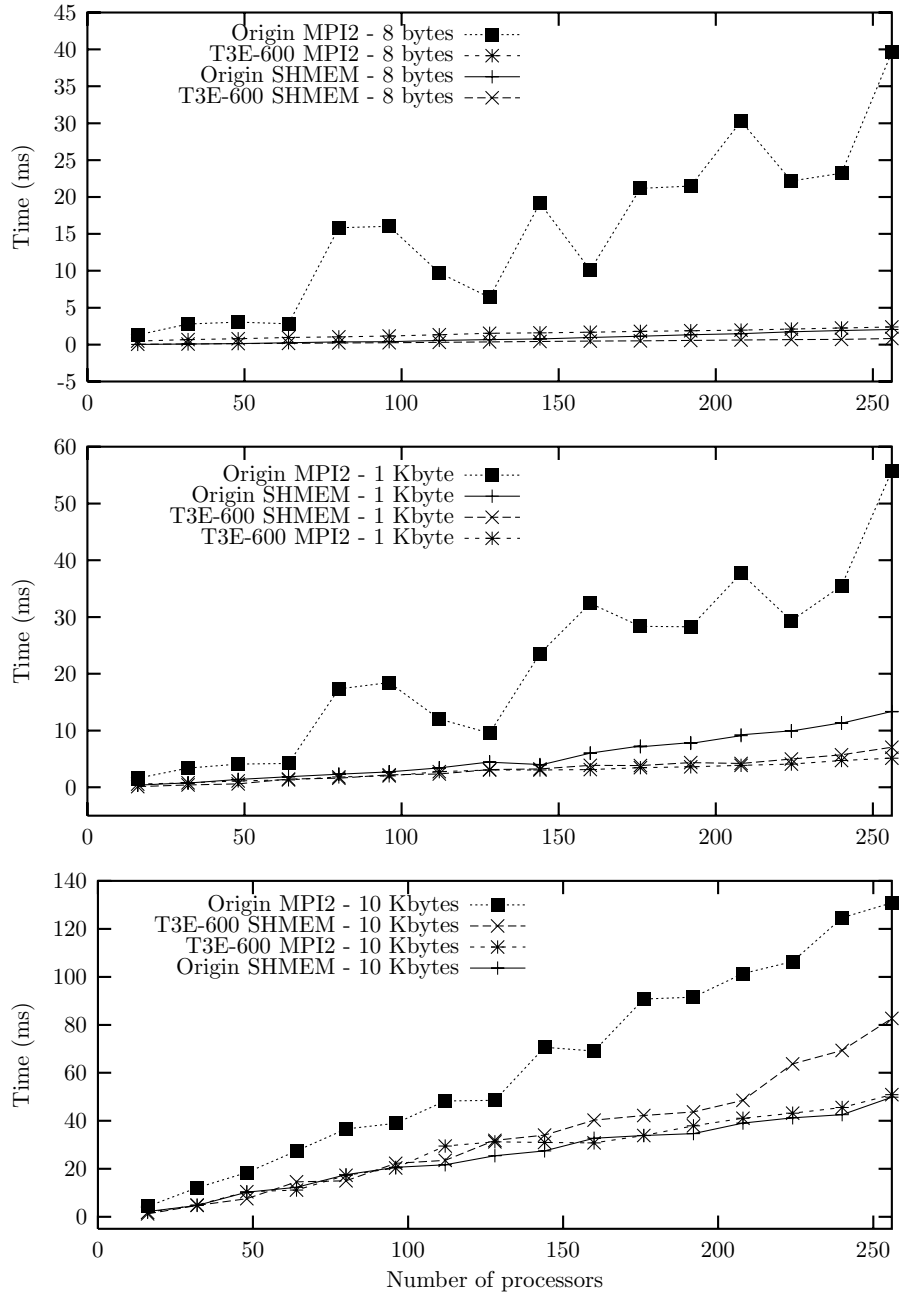



Figure 9. All-to-all test with 8-byte, 1-Kbyte and 10-Kbyte messages.

Table V. Test 5: the all-to-all for 8-byte messages (times in milliseconds).

| | 16 PE | 32 PE | 64 PE | 128 PE | 256 PE |
|---|---|---|---|---|---|
| Origin SHMEM | 0.032 | 0.078 | 0.25 | 0.67 | 2.04 |
| Origin MPI-2 | 1.32 | 2.8 | 2.81 | 6.39 | 39.6 |
| T3E SHMEM | 0.044 | 0.086 | 0.17 | 0.37 | 0.82 |
| T3E MPI-2 | 0.48 | 0.67 | 0.97 | 1.54 | 2.39 |

## 4. CONCLUSION

This paper compares the performance and scalability of SHMEM and MPI-2 one-sided routines on five communication patterns for a SGI Origin 2000 machine and a Cray T3E-600 machine. The communication tests were chosen to represent commonly used communication patterns with low contention (accessing distant messages, a circular right shift, a binary tree broadcast) to communication patterns with high contention (a 'naive' broadcast and an all-to-all). The structure of the MPI-2 one-sided routines is more complex than that of the SHMEM ones. Implementing these five tests using MPI-2 one-sided routines was sometimes challenging, especially the binary tree broadcast test.

For small messages the SHMEM implementations were typically five to 40 times faster than the MPI-2 implementations for all tests and for both machines. For large messages, performance results of the SHMEM and the MPI-2 implementations were mixed. Sometimes the MPI-2 outperformed the SHMEM but most of the time SHMEM outperformed MPI-2 for both machines.

## APPENDIX A: SHMEM BINARY TREE BROADCAST ALGORITHM

```
if (myrank == 0) then
   target = 1
   do while(target <= p-1)
      target = ishft(target,+1)
   end do
   target = ishft(target,-1)
   do while(target>0)
      call shmem_put8(a,a,n,target)
      call shmem_fence()
      call shmem_put8(put_ok,YES,1,target)
      target = ishft(target,-1)
   end do
else
   if((btest(myrank,0) .eqv. .FALSE.) .AND. (myrank /= p-1)) then
      position = 1
      do while(btest(myrank,position) .eqv. .FALSE.)
         position = position+1
      end do
      position = position-1
```

```
      target = ibset(myrank,position)
      do while(target > p-1)
         position = position-1
         target = ibset(myrank,position)
      end do
      call shmem_int8_wait(put_ok,NO)
      put_ok = NO
      do while(position > 0)
         call shmem_put8(a,a,n,ibset(myrank,position))
         call shmem_fence()
         call shmem_put8(put_ok,YES,1,ibset(myrank,position))
         position = position-1
      end do
      call shmem_put8(a,a,n,myrank+1)
   end if
end if
call shmem_barrier_all()
```

## APPENDIX B: MPI-2 BINARY TREE BROADCAST ALGORITHM (`mpi_get`)

```
call mpi_barrier(mpi_comm_world,ierroror)
call mpi_win_fence(0,win,ierror)
if (myrank == 0) then
    target=1
    do while(target <= pe)
              target=ishft(target,+1)
    end do
    target=ishft(target,-1)
    syn = 1.0
    do while(target>0)
       call mpi_send(syn,1,mpi_real8,target,1,mpi_comm_world,ierror)
       target=ishft(target,-1)
    enddo
else
    if((btest(myrank,0) .eqv. .TRUE.) .OR. (myrank==pe)) then
        position=0
        do while(btest(myrank,position) .eqv. .FALSE.)
           position=position+1
        enddo
        call mpi_recv(syn,1,mpi_real8,ibclr(myrank,position), &
                      1,mpi_comm_world,status,ierror)
        call mpi_get(a,n,mpi_real8,ibclr(myrank,position), &
                     0,n,mpi_real8,win,ierror)
        check = 0.0
        do while (check == 0.0 )
           call mpi_get(check,1,mpi_real8,myrank, &
                        (n-1),1,mpi_real8,win,ierror)
        enddo
    else
        position=1
        do while(btest(myrank,position) .eqv. .FALSE.)
```

```
                position=position+1
        end do
        j=position-1
        target=ibset(myrank,j)
        do while(target > p-1)
           j=j-1
           target=ibset(myrank,j)
        end do
        call mpi_recv(syn,1,mpi_real8,ibclr(myrank,position), &
                      1,mpi_comm_world,status,ierror)
        call mpi_get(a,n,mpi_real8,ibclr(myrank,position), &
                     0,n,mpi_real8,win,ierror)
        check = 0.0
        do while (check == 0.0 )
           call mpi_get(check,1,mpi_real8,myrank, &
                        (n-1),1,mpi_real8,win,ierror)
        enddo
        syn = 1.0
        do while(j >= 0)
           call mpi_send(syn,1,mpi_real8,ibset(myrank,j), &
                         1,mpi_comm_world,ierror)
           j=j-1
        end do
    endif
endif
call mpi_win_fence(0,win,ierror)
call mpi_barrier(mpi_comm_world,ierror)
```

## APPENDIX C: MPI-2 BINARY TREE BROADCAST ALGORITHM (`mpi_put`)

```
call mpi_barrier(mpi_comm_world,ierror)
call mpi_win_fence(0,win,ierror)
if (myrank == 0) then
    target=1
    do while(target <= p-1)
             target=ishft(target,+1)
    end do
    target=ishft(target,-1)
    do while(target>0)
             call mpi_put(a,n,mpi_real8,target,0,n,mpi_real8,win,ierror)
             target=ishft(target,-1)
    enddo
else
    if((btest(myrank,0) .eqv. .TRUE.) .OR. (myrank==pe)) then
         check = 0.0
         do while(check == 0)
           call mpi_get(check,1,mpi_real8,myrank,(n-1),1,mpi_real8,win,ierror)
    else
         position=1
         do while(btest(myrank,position) .eqv. .FALSE.)
                    position=position+1
```

```
        end do
        j=position-1
        target=ibset(myrank,j)
        do while(target > p-1)
                  j=j-1
                  target=ibset(myrank,j)
        end do
        check = 0.0
        do while (check == 0.0 )
           call mpi_get(check,1,mpi_real8,myrank, &
                        (n-1),1,mpi_real8,win,ierror)
        enddo
        do while(j >= 0)
           call mpi_put(a,n,mpi_real8,ibset(myrank,j), &
                        0,n,mpi_real8,win,ierror)
           j=j-1
        end do
    endif
endif
call mpi_barrier(mpi_comm_world,ierror)
call mpi_win_fence(0,win,ierror)
```

**ACKNOWLEDGEMENTS**

The authors would like to thank SGI for allowing them to use their SGI Origin 2000 machine. They also would like to thank Cray for giving them access to their Cray T3E-600 machine.

**REFERENCES**

1. Gropp W, Lusk E, Thakur R. *Using MPI-2 Advanced Features of the Message-Passing Interface*. MIT Press: Cambridge, MA, 1999.
2. Message Passing Interface Forum. *MPI-2: Extensions to the Message-Passing Interface*, 1997. http://www-unix.mcs.anl.gov/mpi/.
3. Snir M, Otto S, Huss-Lederman S, Walker D, Dongarra J. *MPI: The Complete Reference* (2nd edn), vol. 1. MIT Press: Cambridge, MA, 1998.
4. MPI Web server. http://www-unix.mcs.anl.gov/mpi/ [2003].
5. Feind K. *Shared Memory Access (SHMEM) Routines*. Cray Research, Inc.: Eagan, MN, 1995.
6. Feind K. *SHMEM Library Implementation on IRIX Systems*. Silicon Graphics, Inc.: Mountain View, CA, 1997.
7. Man Page Collection: Shared Memory Access (SHMEM). *Cray Publication S-2383-23*, 2003. http://www.cray.com/craydoc/20/.
8. Luecke GR, Raffin B, Coyle JJ. 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* 1998. http://hpc-journals.ecs.soton.ac.uk/PEMCS/.
9. Luecke GR, Raffin B, Coyle JJ. Comparing the communication performance and scalability of a SGI Origin 2000, a cluster of Origin 2000's and a Cray T3E-1200 using SHMEM and MPI routines. *The Journal of Performance Evaluation and Modelling for Computer Systems* 1999. http://hpc-journals.ecs.soton.ac.uk/PEMCS/.
10. SGI Web server. http://www.sgi.com/ [2002].
11. Fier J. *Performance Tuning Optimization for Origin 2000 and Onyx 2*. Silicon Graphics, Inc.: Mountain View, CA, 1996. http://techpubs.sgi.com/.
12. Laudon J, Lenosky D. *The SGI Origin: A ccNUMA Highly Scalable Server*. Silicon Graphics, Inc.: Mountain View, CA, 1997.

13. Ammon J. *Hypercube Connectivity within a ccNUMA Architecture*. Silicon Graphics, Inc.: Mountain View, CA, 1998.
14. The Cray T3E Fortran Optimization Guide. *Cray Publication 004-2518-002*, 2003. http://www.cray.com/craydoc/20/.
15. Cray Research Inc. *CRAY T3E Fortran Optimization Guide SG-2518 3.0*. Cray Research, Inc.: Eagan, MN, 1997.
16. Cray Research Inc. *CRAY T3E Programming with Coherent Memory Streams*. Cray Research, Inc.: Eagan, MN, 1996.
17. Luecke GR, Kraeva M, Yuan J, Spanoyannis S. Performance and scalability of MPI on PC clusters. *Concurrency and Computation: Practice and Experience* 2004; **16**(1):79–107.
18. Luecke GR, Coyle JJ. Comparing the performances of MPI on the Cray T3E-900, the Cray Origin 2000 and the IBM P2SC. *The Journal of Performance Evaluation and Modelling for Computer Systems* 1998. http://hpc-journals.ecs.soton.ac.uk/PEMCS/.
19. Barnett M, Gupta S, Payne DG, Shuler L, van de Geijn R, Watts J. Building a high-performance collective communication library. *Proceedings of Supercomputing 94*. IEEE Computer Society Press: Los Alamitos, CA, 1994.