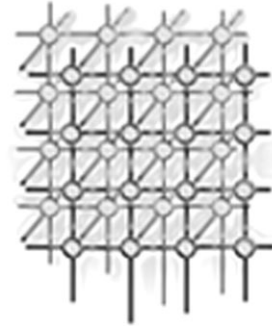


## Performance

# Performance and scalability of MPI on PC clusters

Glenn R. Luecke<sup>\*,†</sup>, Marina Kraeva, Jing Yuan and Silvia Spanoyannis

291 Durham Center, Iowa State University, Ames, IA 50011, U.S.A.



---

## SUMMARY

The purpose of this paper is to compare the communication performance and scalability of MPI communication routines on a Windows Cluster, a Linux Cluster, a Cray T3E-600, and an SGI Origin 2000. All tests in this paper were run using various numbers of processors and two message sizes. In spite of the fact that the Cray T3E-600 is about 7 years old, it performed best of all machines for most of the tests. The Linux Cluster with the Myrinet interconnect and Myricom's MPI performed and scaled quite well and, in most cases, performed better than the Origin 2000, and in some cases better than the T3E. The Windows Cluster using the Gigaset Full Interconnect and MPI/Pro's MPI performed and scaled poorly for small messages compared with all of the other machines. Copyright © 2004 John Wiley & Sons, Ltd.

KEY WORDS: PC clusters; MPI; MPI performance

## 1. INTRODUCTION

MPI [1] is the standard message-passing library used for programming distributed memory parallel computers. Implementations of MPI are available for all commercially available parallel platforms, including PC clusters. Nowadays, clusters are considered as an inexpensive alternative to 'traditional' parallel computers. The purpose of this paper is to compare the communication performance and scalability of MPI communication routines on a Windows Cluster, a Linux Cluster, a Cray T3E-600, and an SGI Origin 2000.

---

\*Correspondence to: Glenn R. Luecke, 291 Durham Center, Iowa State University, Ames, IA 50011-2251, U.S.A.

†E-mail: grl@iastate.edu



## 2. TEST ENVIRONMENT

The SGI Origin 2000 [2,3] used for this study is a 128-processor machine in which each processor is MIPS R10000 running at 250 MHz. There are two levels of cache:  $32 \times 1024$  byte first level instruction and data caches, and a  $4 \times 1024 \times 1024$  byte second level cache for both data and instructions. The communication network is a hypercube for up to 32 processors and is called a 'fat bristled hypercube' for more than 32 processors since multiple hypercubes were interconnected via a Cray Link Interconnect. For all tests, the IRIX 6.5 operating system, the Fortran compiler version 7.3.1.1m and the MPI library version 1.4.0.0.1 were used.

The T3E-600 [2,4] used is a 512-processor machine located in Eagan, Minnesota. Each processor is a DEC Alpha EV5 microprocessor running at 300 MHz. 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 is a three-dimensional, bi-directional torus. For all tests, the UNICOS/mk 2.0.5 operating system, the Fortran compiler version cf90.3.3.0.2 and the MPI library version 1.4.0.0.2 were used.

The Dell Windows Cluster of PCs [5] used is a 128-processor (64-node) machine located at the Computational Materials Institute (CMI) at the Cornell Theory Center. Each node consists of a 64 dual PIII 1 GHz processor with 2 GB of memory. Each processor has 256 KB cache. Nodes are interconnected via a full 64-way Gigaset Interconnect (100 MB/s). To use the Gigaset Interconnect the environment variable `MPI_COMM` was set to `VIA`. Other than setting the `MPI_COMM` environment variable, default settings were used for all tests. The system runs Microsoft Windows 2000, which is key to the seamless desktop-to-HPC model that CTC is pursuing. For all tests, the MPI/Pro library from MPI Software Technology (MPI/Pro distribution version 1.6.3) and Compaq Visual Fortran Optimizing Compiler version 6.1 were used. The Cornell Theory Center also has two other Velocity and Velocity+ clusters. The results for the Velocity Cluster for MPI tests were similar to the results for the cluster at CMI.

The Linux Cluster of PCs [6] used is a 1024-processor machine located at the National Center for Supercomputing Applications (NCSA) in Urbana-Champaign, Illinois. The cluster is a 512-node cluster, each node consisting of a dual processor Intel Pentium III (1 GHz) machine with 1.5 GB ECC SDRAM. Each processor has 256 KB full-speed L2 cache. Nodes are interconnected via a Myrinet (full-duplex  $1.28 + 1.28$  GB/s) network. The system was running Linux 2.4.9 (RedHat 7.2). For all tests, version 3.0-1 Intel Fortran compiler (ifc) with the `-Qoption,ld,-Bdynamic` compiler options, and the MPICH-GM for the Myrinet Linux Cluster were used. All the tests were run both with 1 process and 2 processes per node. Thus, later in the paper the results of the tests run with 1 process per node are shown under 'Linux Cluster (1 ppn)' and the results with 2 processes per node are shown under 'Linux Cluster (2 ppn)'.

The performance results reported in this paper were obtained with a large message size and a small message size, all using 8 byte reals, and up to 128 processors. Section 3 introduces the timing methodology used. Section 4 presents the performance results. The conclusions are listed in Section 5.

## 3. TIMING METHODOLOGY

All tests were timed using the following template:



```
integer, parameter :: ntest=51
real*8, dimension(ntest) :: array_time, time
. . .
do k = 1, ntest
  flush(1:ncache) = flush(1:ncache) + 0.1
  call mpi_barrier(mpi_comm_world, ierror)
  t = mpi_wtime()

  ... mpi routine(s) ...

  array_time(k) = mpi_wtime() - t
  call mpi_barrier(mpi_comm_world, ierror)
  A(1) = A(1) + flush(mod(k, ncache))
enddo
call mpi_reduce(array_time, time, ntest, mpi_real8, mpi_max, 0, &
  mpi_comm_world, ierror)

...

write(*,*) "prevent dead code elimination", A(1), flush(1)
```

Throughout this paper, `ntest` is the number of trials of a timing test performed in a single job. The value of `ntest` should be chosen to be large enough to access the variability of the performance data collected. For the tests in this paper, setting `ntest=51` (the first timing was always discarded) was satisfactory. Timings were done by first flushing the cache on all processors by changing the values in the real array `flush(1:ncache)` prior to timing the desired operation. The value of `ncache` was chosen so the size of the array `flush` was the size of the secondary cache for the T3E-600 ( $96 \times 1024$  bytes), the Origin 2000 ( $8 \times 1024 \times 1024$  bytes), the Windows and Linux Clusters ( $512 \times 1024$  bytes). Note that by flushing the cache before 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 [3]. Figure 1 shows that the execution time without cache flushing is ten times smaller than the execution time with cache flushing for the broadcast communication of an 8-byte message with 128 processors on the Origin.

In the above timing code, the first call to `mpi_barrier` guarantees that all processors reach this point before they each call the wallclock 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 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`. Some compilers might split the whole timing loop into two loops with cache flushing statement in the one and timed MPI routine in the other. To prevent this, the statement '`A(1) = A(1) + flush(mod(k, ncache))`' was added into the timing loop, where `A` is the array involved in the communication being timed. To prevent the compiler from considering all or part of the 'code to be timed' as dead code and eliminating it, later in the program the value of `A(1)` and `flush(1)` were used in the `write` statement. 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.

Figure 2 shows times in seconds for 100 trials for `mpi_bcast` test with 128 processors and 10 KB message. Note that there are several 'spikes' in the data with the first spike being the first timing.

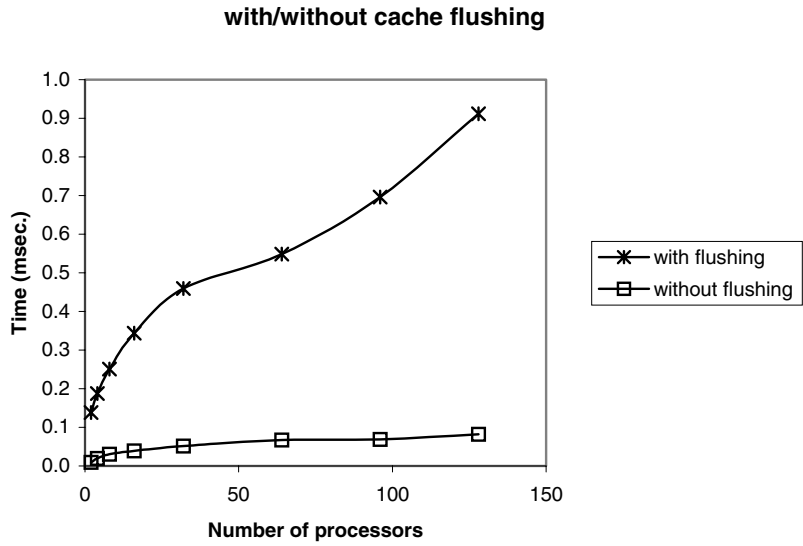


Figure 1. Execution times for `mpi_bcast` for an 8-byte message on the Origin.

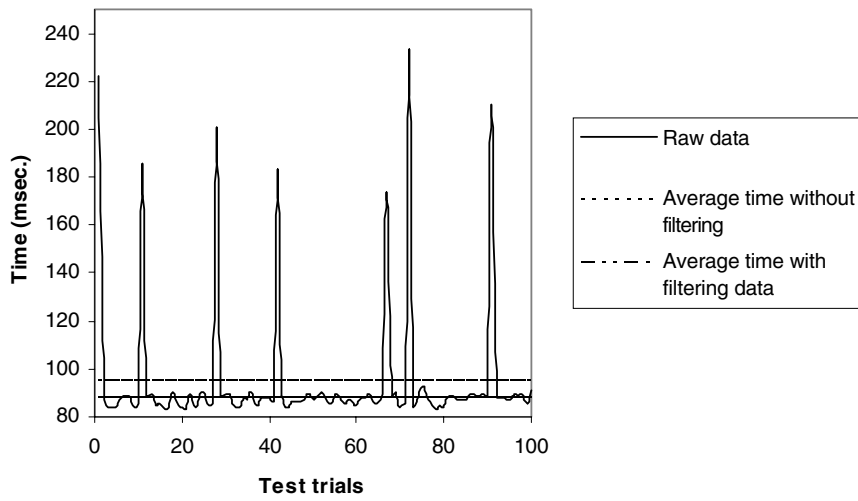


Figure 2. Time for 100 trials for `mpi_bcast` for a 10 KB message on the Origin with 128 processors.



Table I. Time ratios for average of filtered data for all processes for the ping-pong test.

Message size	8 bytes	1 MB
Origin/T3E	2.8	1.6
Windows Cluster/T3E	4.3	1.1
Linux Cluster (1 ppn)/T3E	1.1	1.3
Linux Cluster (2 ppn)/T3E	1.8	1.3

The first timing usually is significantly longer than most of the other timings (likely due to the additional setup time required for the first call to subroutines and functions), so the time for the first trial is always removed. The other spikes are probably due to the operating system interrupting the execution of the program. The average of the 99 trials (the first trial is removed) is 94.7 ms, which is much longer than most of the other trials. The authors decided to measure times for each operation by first filtering out the spikes as follows. Compute the median value after the first time trial is removed. All times greater than 1.8 times this median are candidates to be removed. The authors consider it to be inappropriate to remove more than 10% of the data. If more than 10% of the data would be removed by the above procedure then only the largest 10% of the spikes are removed. Authors thought that these additional (smaller) spikes should influence the data reported. However, for all tests in this paper, the filtering process always removed less than 10% of the data. Using this procedure, the filtered value for the time in Figure 2 is 87.7 ms instead of 94.7 ms.

#### 4. TEST DESCRIPTIONS AND PERFORMANCE RESULTS

This section contains performance results for 11 MPI communication tests. All tests except for the ping-pong test were run using 2, 4, 8, 16, 32, 64, 96 and 128 processors on all machines using `mpi_comm_world` for the communicator. There are two processors on each node of the Linux Cluster and MPI jobs can be run with either one MPI process per node (1 ppn) or with two MPI processes per node (2 ppn). For the Linux Cluster, all tests were run using 1 ppn and using 2 ppn since there are 256 nodes on this machine. Since there are only 64 nodes on the Windows Cluster, all tests were run with two MPI processes per node.

##### 4.1. Test 1: ping-pong between 'near' and 'distant' processors

Ideally, one would like to be able to run parallel applications with large numbers of processors without the communication network slowing down execution. One would like the time for sending a message from one processor to another to be independent of the processors used. To determine how each machine deviates from this ideal, we measure the time required for processor 0 to send a message and receive it back from processor  $j$  for  $j = 1-127$  for all machines. Because of time limitations we did not test ping-pong times between all processors. The code for processor 0 to send a message of size  $n$  to processor  $j = 1$  to 127 and to receive it back is:

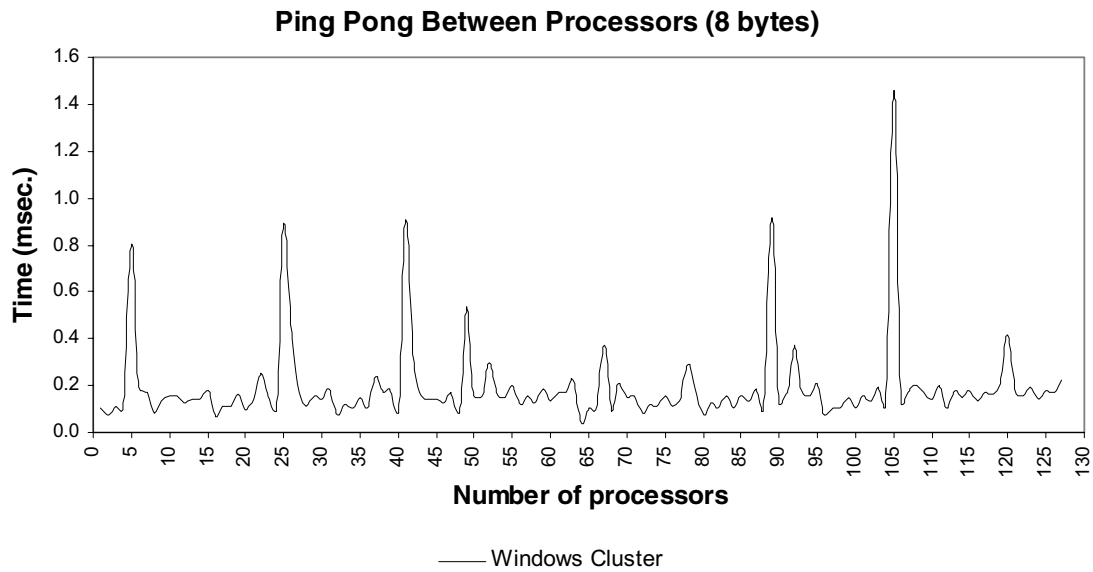
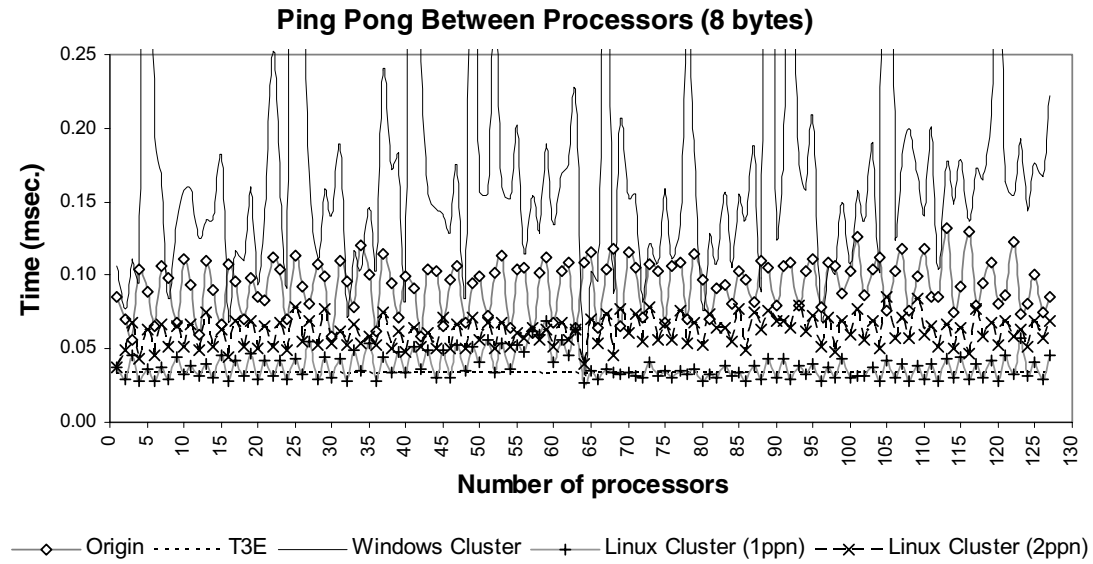


Figure 3. Test 1 (ping-pong between 'near' and 'distant' processors) with times in milliseconds.

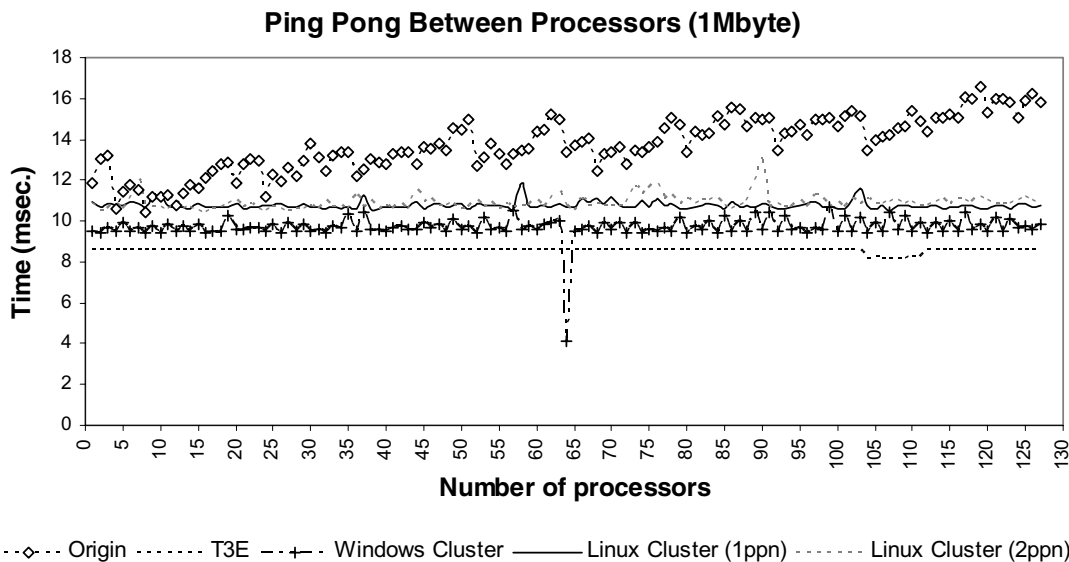


Figure 4. Test 1 (ping-pong between 'near' and 'distant' processors) with times in milliseconds.

```
if (rank == 0) then
    call mpi_send (A,n,mpi_real8,j,1, mpi_comm_world,ierr)
    call mpi_recv(B,n,mpi_real8,j,2,mpi_comm_world,status,ierr)
endif
if (rank == j) then
    call mpi_recv(B,n,mpi_real8,0,1,mpi_comm_world,status,ierr)
    call mpi_send (A,n,mpi_real8,0,2, mpi_comm_world,ierr)
endif
```

Notice that processor  $j$  receives the message in array B and sends the message in array A. If the message were to be received in A instead of B, then this would put A into the cache making the sending of the second message in the ping-pong faster than the first. The results of this test are based on one run per machine (with many trials) because the assignment of ranks to physical processors will vary from one run to another.

Figures 3 and 4 present the performance data for this test. Each ping-pong time is divided by two, indicating the average one-way communication time. For both message sizes, the T3E shows the best performance. Ideally, these graphs would all be horizontal lines. Note that many of the graphs are 'reasonably close' to this ideal. The Origin and Windows Cluster have the largest variation in times for this test for an 8-byte message. The performance data for 8-byte messages shows that the Windows Cluster has significantly higher latency than the other machines. However, for the 1 MB message the Windows Cluster performs better than the Linux Cluster and the Origin. Note that on the Windows Cluster the time to send and receive a message between processes 0 and 64 is much less than the

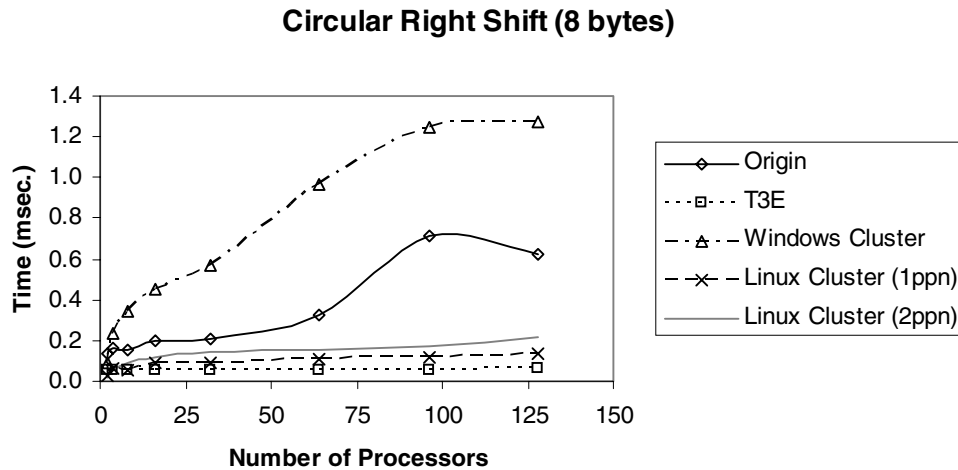


Figure 5. Test 2 (circular right shift) with times in milliseconds.

other times. This is likely due to process 0 and 64 being assigned to the same node resulting in communication within the node instead of between nodes. Also note the large spikes in the performance data for 8-byte messages for the Windows Cluster. Recall that for each fixed number of processes and for each fixed rank, the test was run 51 times and these data were then filtered as described in Section 3. Figure 3 presents the results of these data after filtering. The authors examined the raw data (51 trials) for the large spikes shown for the Windows Cluster in Figure 3 and discovered that most of the data were large; however, if one takes the minimum of the raw data, then the performance data are then more in line with the other data shown in Figure 3. It is likely therefore, that these large spikes in the Windows Cluster are due to operating system activity occurring during the execution of the ping-pong test.

To give an idea of the overall performance achieved on this test on each machine, the data for all the different numbers of MPI processes were filtered (as described in Section 3) and an average was taken. Since the performance data for the Cray T3E were best, these averages were then compared with the average times computed for the T3E, see Table I.

#### 4.2. Test 2: the circular right shift

The purpose of this test is to measure the performance of the ‘circular right shift’ where each process receives a message of size  $n$  from its ‘left’ neighbor, i.e.  $\text{modulo}(\text{myrank}-1, p)$  is the left neighbor of  $\text{myrank}$ . Ideally, the execution time for this test would not depend on the number of processors, since these operations have the potential of executing at the same time. The code used for this test is:

```
call mpi_sendrecv(a,n,mpi_real8,module(myrank+1,p),tag,b,n, &
  mpi_real8,module(myrank-1,p),tag,mpi_comm_world,status,ierr)
```



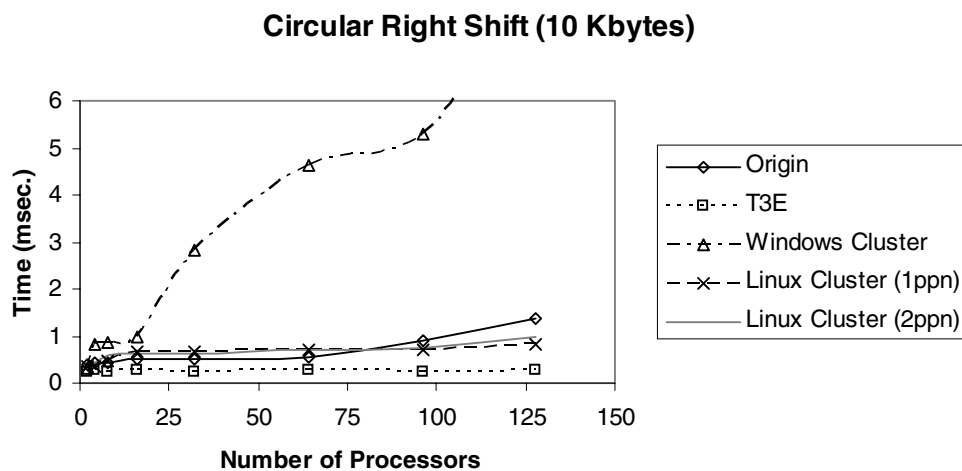


Figure 6. Test 2 (circular right shift) with times in milliseconds.

Figures 5 and 6 present the performance data for this test. Note that for both message sizes, the T3E performs best. Also note that both the T3E and the Linux Cluster scale well for both message sizes. The Windows Cluster performs and scales poorly for both message sizes. This is likely due to a poor implementation of `mpi_sendrecv` since the Windows Cluster performs well for the ping-pong test for 1 MB messages.

#### 4.3. Test 3: the barrier

An important performance characteristic of a parallel computer is its ability to efficiently execute a barrier synchronization. This test evaluates the performance of the MPI barrier:

```
call mpi_barrier(mpi_comm_world, ierror)
```

Figure 7 presents the performance and scalability data for `mpi_barrier`. Note that the T3E and the Origin scale and perform significantly better than the Windows and Linux Clusters. The Linux Cluster performs better than the Windows Cluster. Table II shows the performance of all machines relative to the T3E for 128 processors.

#### 4.4. Test 4: the broadcast

This test evaluates the performance of the MPI broadcast:

```
call mpi_bcast(A, n, mpi_real8, 0, mpi_comm_world, ierror)
```

for  $n = 1$  and  $n = 125\,000$ .

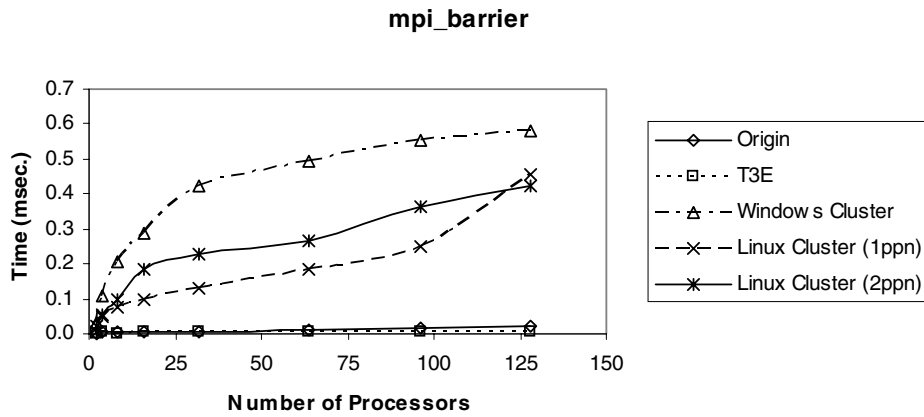


Figure 7. Test 3 (`mpi_barrier`) with times in milliseconds.

Table II. Time ratios for 128 processors for the `mpi_barrier` test.

Origin/T3E	5.8
Windows Cluster/T3E	143
Linux Cluster (1 ppn)/T3E	112
Linux Cluster (2 ppn)/T3E	104

Figures 8 and 9 present the performance data for `mpi_bcast`. The T3E and the Linux Cluster (both with 1 and 2 processes per node) show good performance for 8-byte messages and they both perform about three times better than the Origin. The performance of the Windows Cluster was so poor that a separate graph was required to show its performance. The results for 1 MB messages are closer to each other, with the T3E performing best and the Windows Cluster performing the worst. Table III shows the performance of all machines relative to the T3E for 128 processors. Notice that for an 8-byte message the Linux Cluster with 1 process per node outperformed the T3E.

To better understand how well the machines scale implementing `mpi_bcast`, let us consider the following simple execution model. Assume the time to send a message of size  $M$  bytes from one processor to another is

$$\alpha + M\beta$$

where  $\alpha$  is the latency and  $\beta$  is the communication rate of the network. This assumes there is no contention and there is no difference in time when sending a message between any two processors. Assume that the number of processors,  $p$ , used is a power of 2, and assume that `mpi_bcast` is

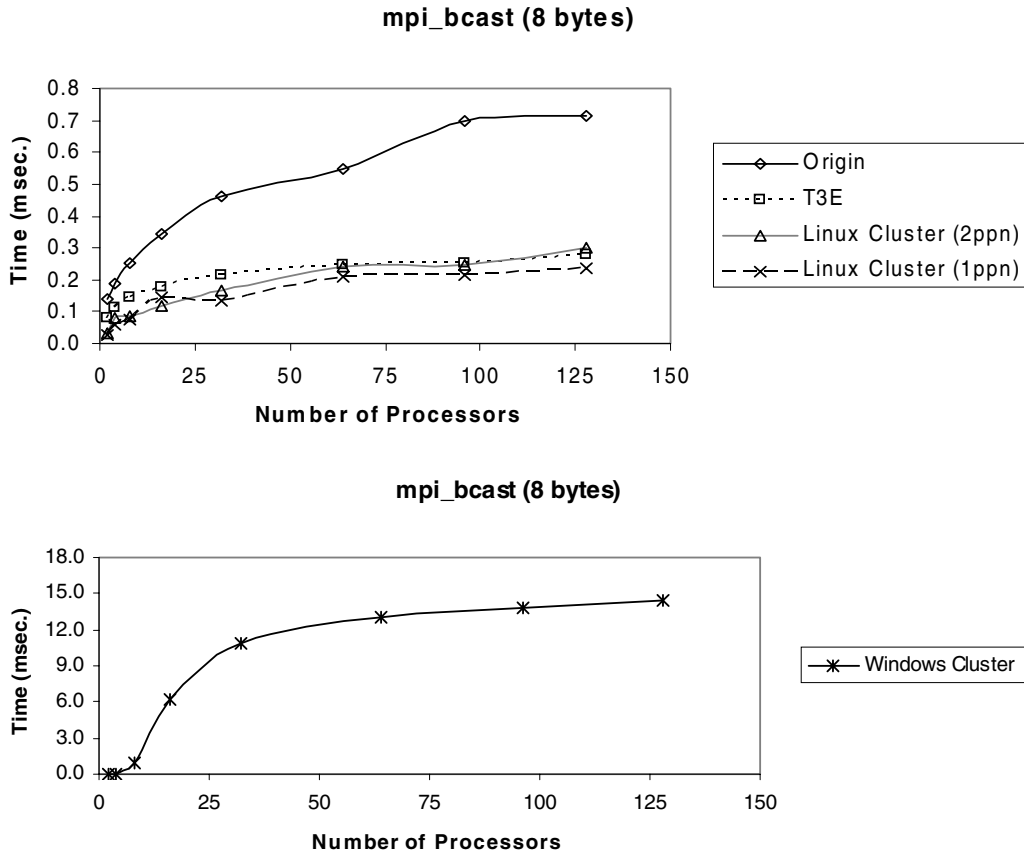


Figure 8. Test 4 (mpi\_bcast) with times in milliseconds.

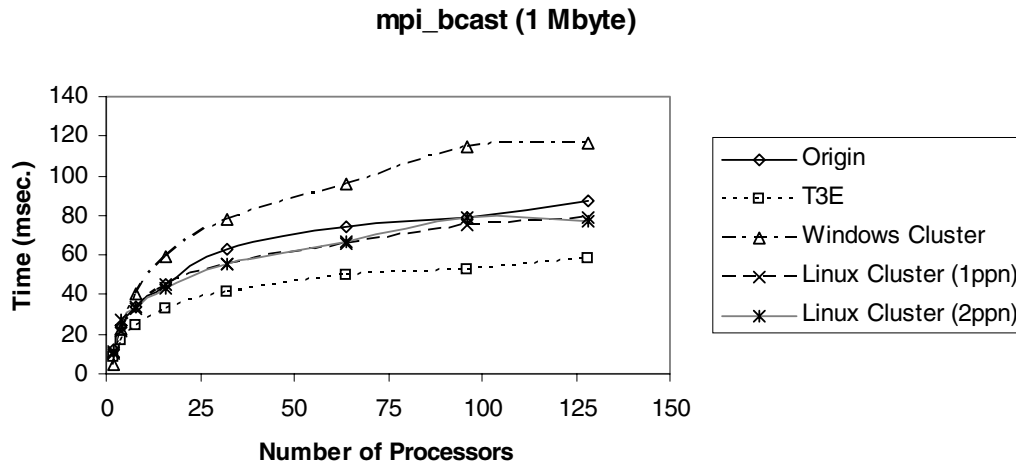
implemented using a binary tree algorithm. If  $p = 2^k$ , then with the above assumptions the time to execute a binary tree broadcast for  $p$  processors would be

$$(\log(p)) (\alpha + M\beta)$$

Thus, ideally (execution time)/ $\log(p)$  would be a constant for all such  $p$  for a fixed message size. Thus, plotting

$$(\text{execution time})/\log(p)$$

will provide a way to better understand the scalability of mpi\_bcast for each machine. Figure 10 shows these results. Note that when  $p < 8$ , the (execution time)/ $\log(p)$  is nearly constant on all machines, except the Windows Cluster for an 8-byte message.

Figure 9. Test 4 (`mpi_bcast`) with times in milliseconds.Table III. Time ratios for 128 processors for the `mpi_bcast` test.

Message size	8 bytes	1 MB
Origin/T3E	2.5	1.5
Windows Cluster/T3E	51	2
Linux Cluster (1 ppn)/T3E	0.83	1.37
Linux Cluster (2 ppn)/T3E	1.08	1.33

#### 4.5. Test 5: the scatter

This test measures the time to execute

```
call mpi_scatter(B, n, mpi_real8, A, n, mpi_real8, 0,
  mpi_comm_world, ierror)
```

for  $n = 1$  and  $n = 1250$ . For both message sizes, the T3E has the best performance, see Figures 11 and 12. Notice that for 8 byte messages the performance of the Windows Cluster is much poorer than the other machines. Table IV shows the performance of all machines relative to the T3E for 128 processors.

Let us assume that the `mpi_scatter` was implemented by an algorithm based on a binary tree. Initially the root processor 0 owns  $p$  messages  $m_0, \dots, m_{p-1}$ , each of size  $M$  bytes, that have to be

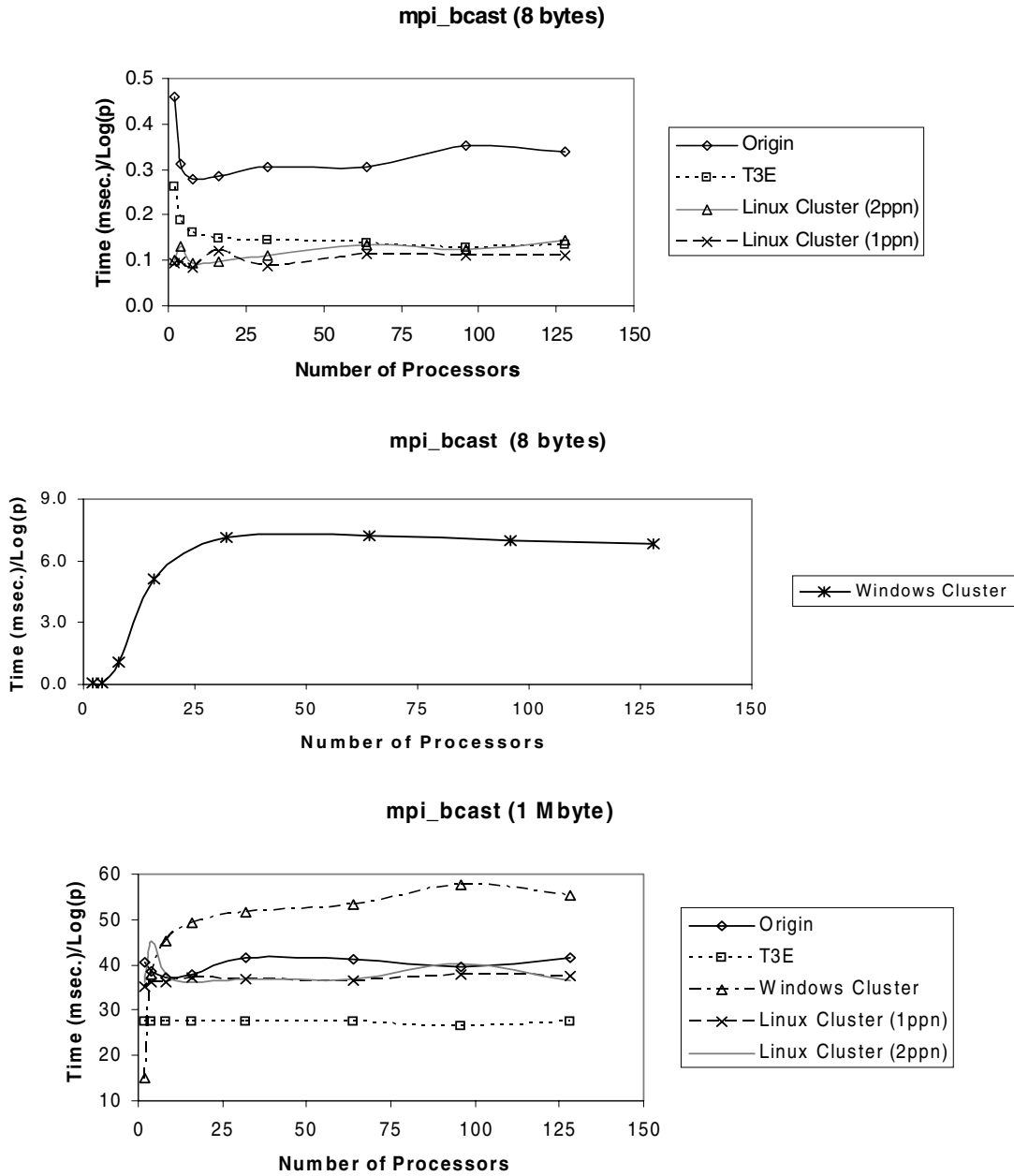


Figure 10. Test 4 (mpi\_bcast) plotting (execution time)/log( $p$ ), where  $p$  is the number of processors.

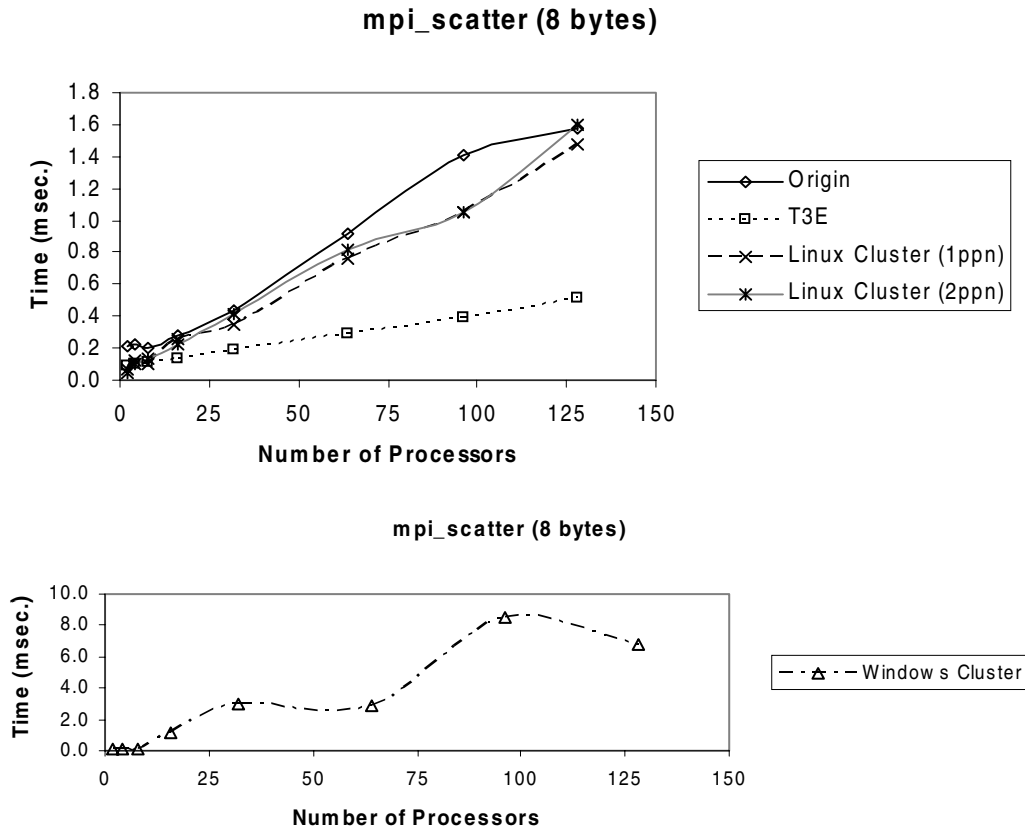
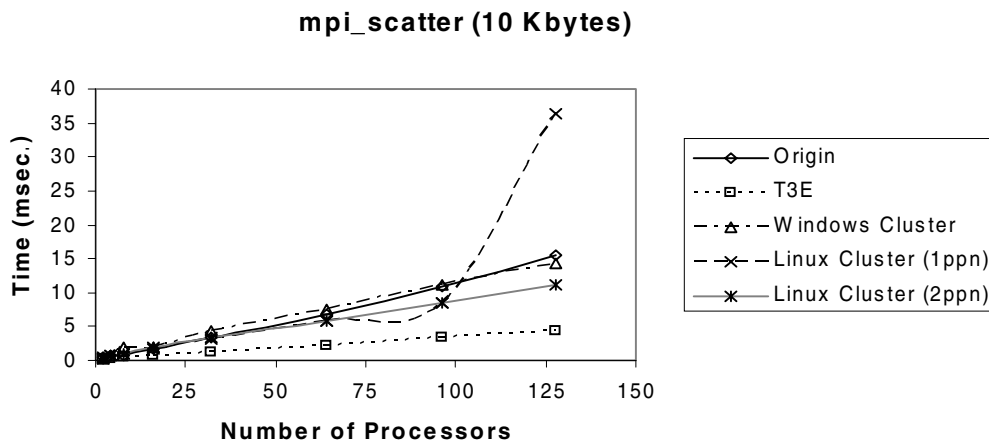


Figure 11. Test 5 (mpi\_scatter) with times in milliseconds.

sent to processors  $1, \dots, p-1$ , respectively. First, processor 0 sends  $m_{p/2}, \dots, m_{p-1}$  to processor  $p/2$ . For the second step, processors 0 send messages  $m_{p/4}, \dots, m_{p/2-1}$  to processor  $p/4$  and concurrently processor  $p/2$  sends messages  $m_{3p/4}, \dots, m_{p-1}$  to  $3p/4$ . The scatter is completed by repeating these steps  $\log(p)$  times. Based on the model described above, the execution time would be

$$\alpha \log(p) + (p - 1)M\beta$$

If we now assume that a large message is being scattered, then  $M$  is large and the execution time will be dominated by  $(p - 1)M\beta$ . Thus, the (execution time)/( $p - 1$ ) would be constant for a fixed message size as  $p$  increases. This allows us to better understand the scalability of mpi\_scatter for large messages. Figure 13 shows that the (execution time)/( $p - 1$ ) is nearly constant on all machines when more than eight processes participate in communications. When  $M$  is small, then both terms of the above expression are significant. That makes it difficult to evaluate the scalability for the simple execution-time model presented above.

Figure 12. Test 5 (`mpi_scatter`) with times in milliseconds.Table IV. Time ratios for 128 processors for the `mpi_scatter` test.

Message size	8 bytes	10 KB
Origin/T3E	3	3.5
Windows Cluster/T3E	13.36	3.35
Linux Cluster (1 ppn)/T3E	2.9	8.44
Linux Cluster (2 ppn)/T3E	3.14	2.6

#### 4.6. Test 6: the gather

This test measures the time to execute

```
call mpi_gather(A, n, mpi_real8, B, n, mpi_real8, 0,  
mpi_comm_world, ierror)
```

for  $n = 1$  and  $n = 1250$ .

Figures 14 and 15 present the performance data for this test. For an 8-byte message, the performance on all machines except Windows Cluster is similar, with the Linux Cluster (2 processes per node) performing best. For a 10-KB message, the T3E has the best performance. Table V shows the performance of all machines relative to the T3E for 128 processors.

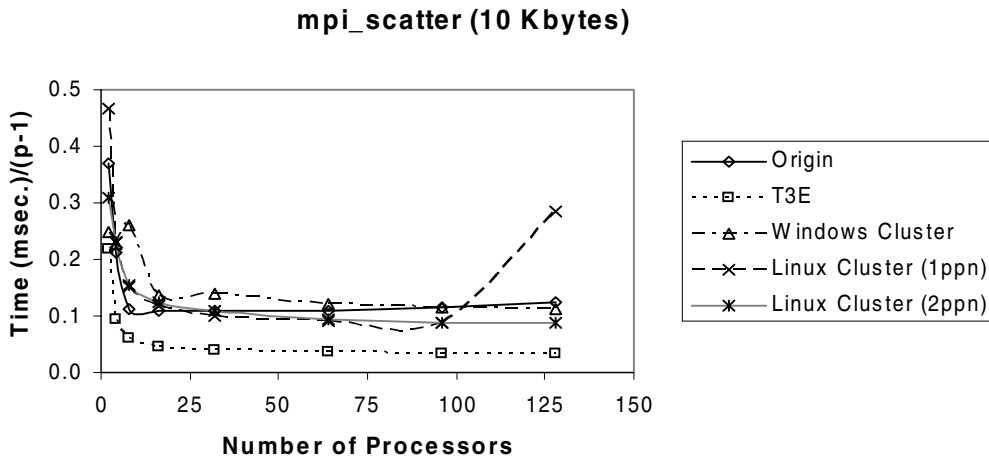


Figure 13. Test 5 (`mpi_scatter`) plotting the (execution time)/( $p - 1$ ), where  $p$  is the number of the processors.

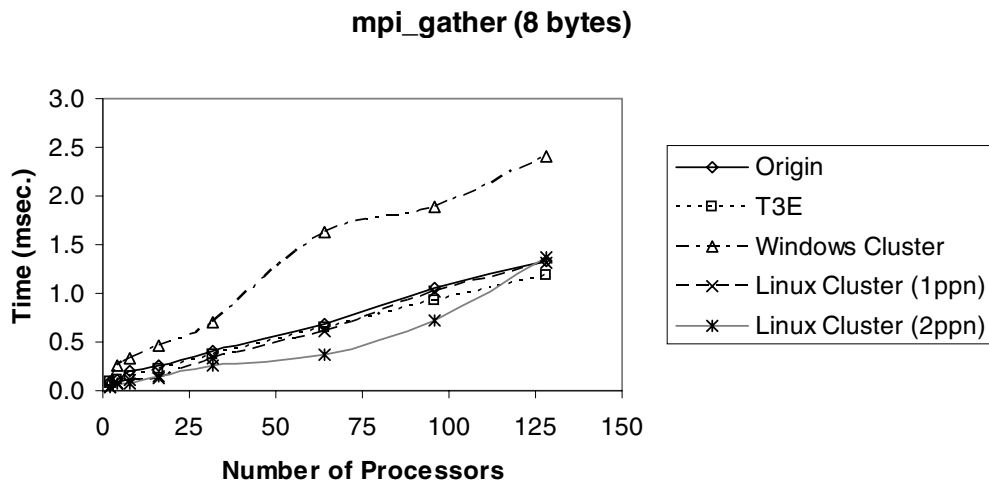
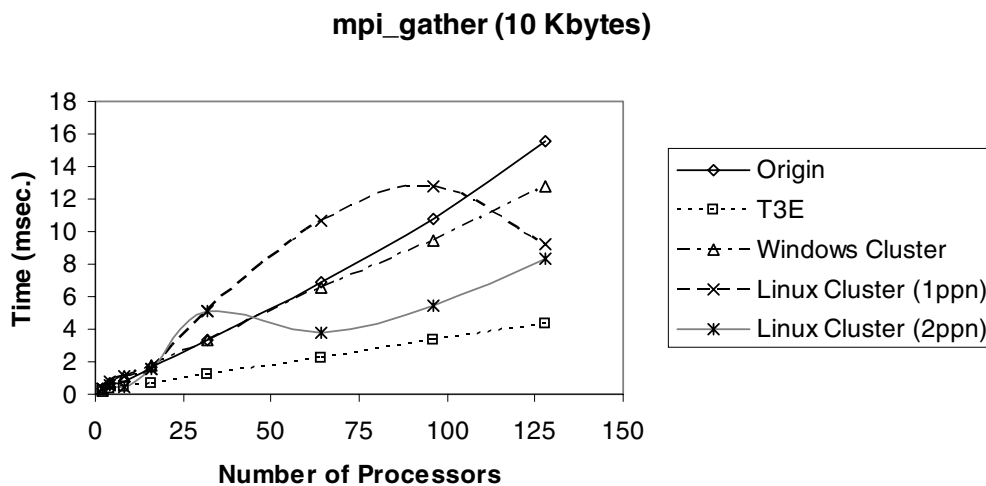


Figure 14. Test 6 (`mpi_gather`) with times in milliseconds.



Figure 15. Test 6 (`mpi_gather`) with times in milliseconds.Table V. Time ratios for 128 processors for the `mpi_gather` test.

Message size	8 bytes	10 KB
Origin/T3E	1.1	3.5
Windows Cluster/T3E	2	3
Linux Cluster (1 ppn)/T3E	1.1	2.15
Linux Cluster (2 ppn)/T3E	1.2	1.9

The scalability analysis for `mpi_gather` is the same as that for `mpi_scatter`. Figure 16 shows that the (execution time)/( $p - 1$ ) is roughly constant on the T3E, the Origin and the Windows Cluster for the large message size when more than eight processes participate in the communications.

#### 4.7. Test 7: the all-gather

This test measures the time to execute:

```
call mpi_allgather(A(1), n, mpi_real8, B(1,0), n, mpi_real8,  
    mpi_comm_world, ierror)
```

for  $n = 1$  and  $n = 1250$ .

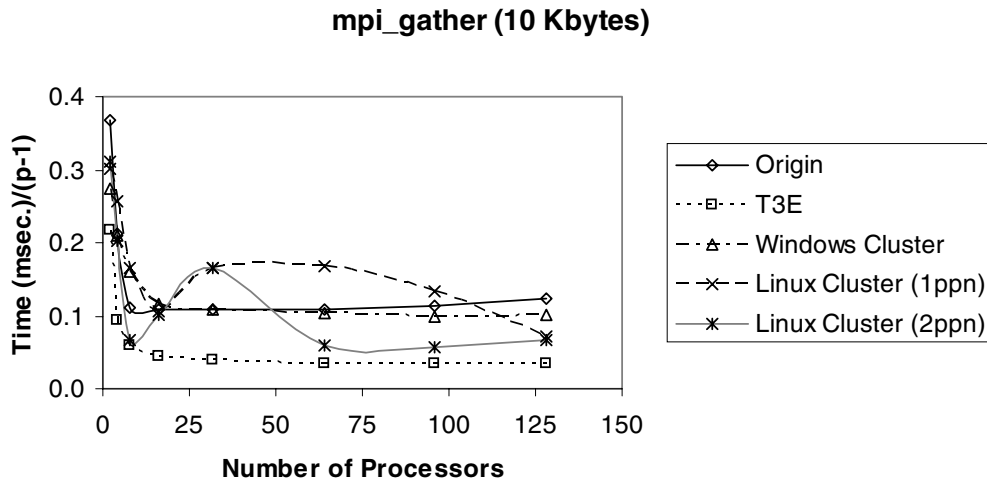


Figure 16. Test 6 (`mpi_gather`) plotting the (execution time)/( $p - 1$ ), where  $p$  is the number of the processors.

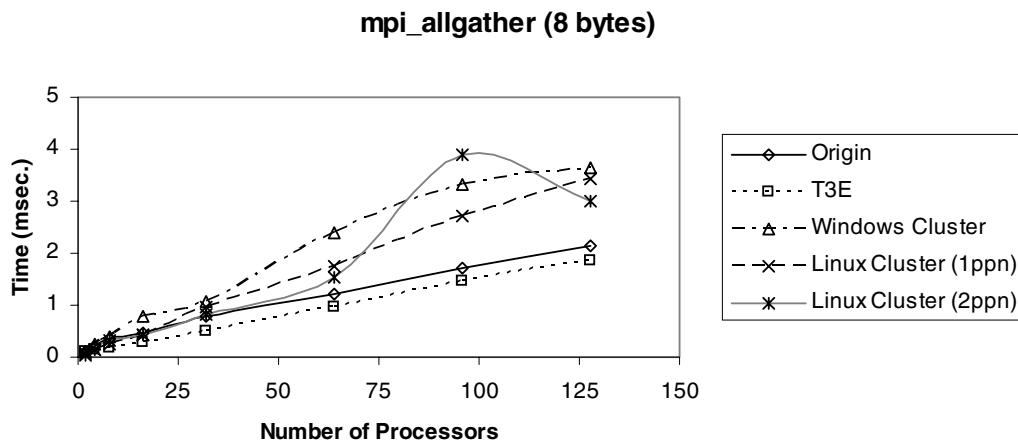


Figure 17. Test 7 (`mpi_allgather`) with times in milliseconds.

Figures 17 and 18 present the performance data for this test. For the 8-byte message, the performance and scalability of the T3E and the Origin are good compared with those of the Windows and Linux Clusters. For the 10-KB message, the Linux Cluster (both 1 and 2 processes per node) performs slightly better than the T3E, while the Origin and the Windows Cluster demonstrate poor performance. Table VI shows the performance of all machines relative to the T3E for 128 processors.

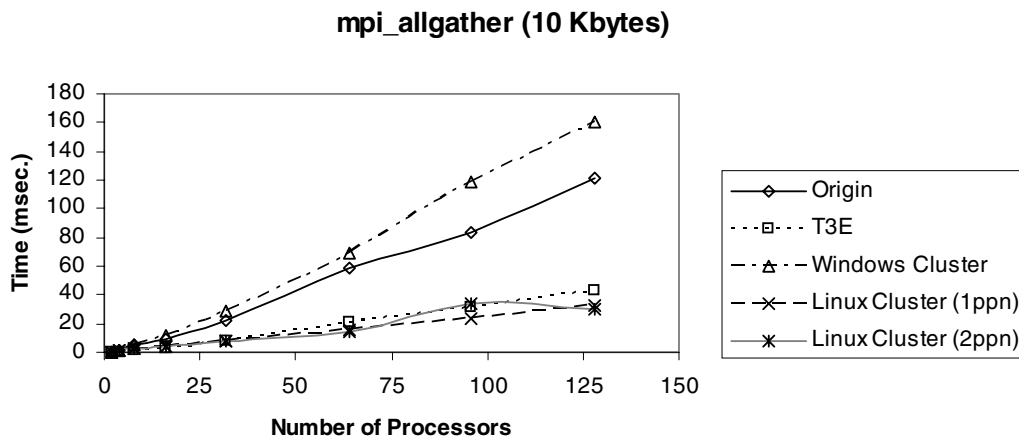


Figure 18. Test 7 (`mpi_allgather`) with times in milliseconds.

Table VI. Time ratios for 128 processors for the `mpi_allgather` test.

Message size	8 bytes	10 KB
Origin/T3E	1.2	2.8
Windows Cluster/T3E	1.97	3.72
Linux Cluster (1 ppn)/T3E	1.86	0.75
Linux Cluster (2 ppn)/T3E	1.64	0.68

The `mpi_allgather` is sometimes implemented as  $p - 1$  circular right shifts executed by each of the  $p$  processors [7], where each processor  $i$  initially owns a message  $m_i$  of size  $M$  bytes. The  $j$ th right-shift is defined by: each processor  $i$  receives the message  $m_{(p+(i-j)) \bmod p}$  from processor  $(i - 1) \bmod p$ . Assuming each right shift can be executed in the amount of time to send a single message of  $m_i$  of size  $M$  bytes from one to another processor, the execution time of `mpi_allgather` is

$$(p - 1)(\alpha + M\beta)$$

Thus,  $(\text{execution time})/(p - 1)$  will be a constant for all such  $p$  for a fixed message size. Hence, plotting  $(\text{execution time})/(p - 1)$  can provide a way to better understand the scalability of `mpi_allgather` for each machine. Figures 19 and 20 show that when the number of processes is more than eight,  $(\text{execution time})/(p - 1)$  is roughly constant for all the machines except for the Windows Cluster and the Origin for the 10 KB message size.

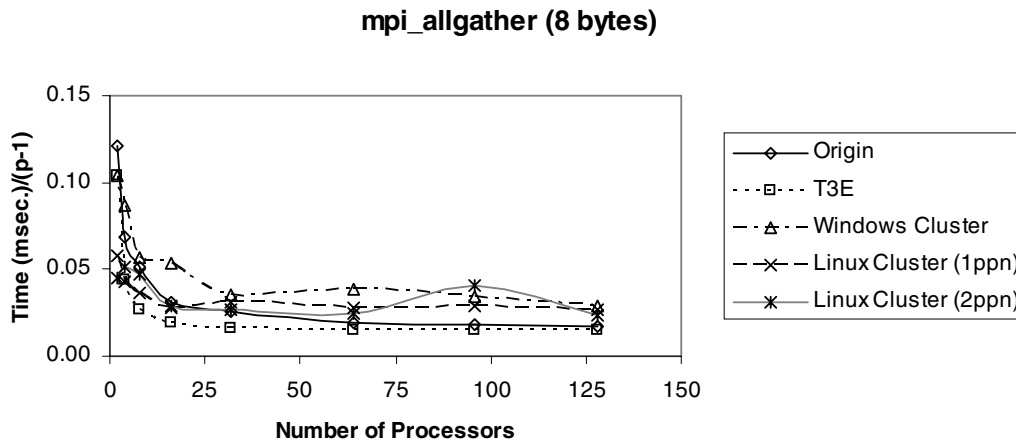


Figure 19. Test 7 (mpi\_allgather) plotting (execution time)/( $p - 1$ ), where  $p$  is the number of processors.

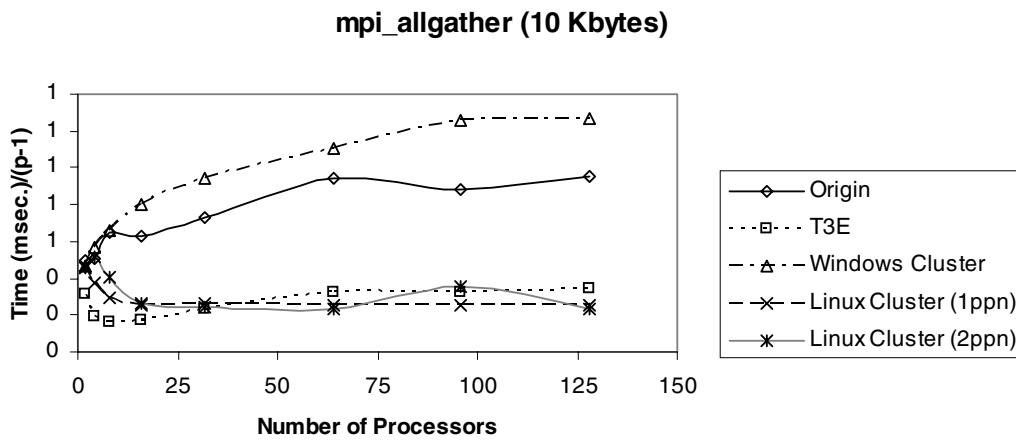
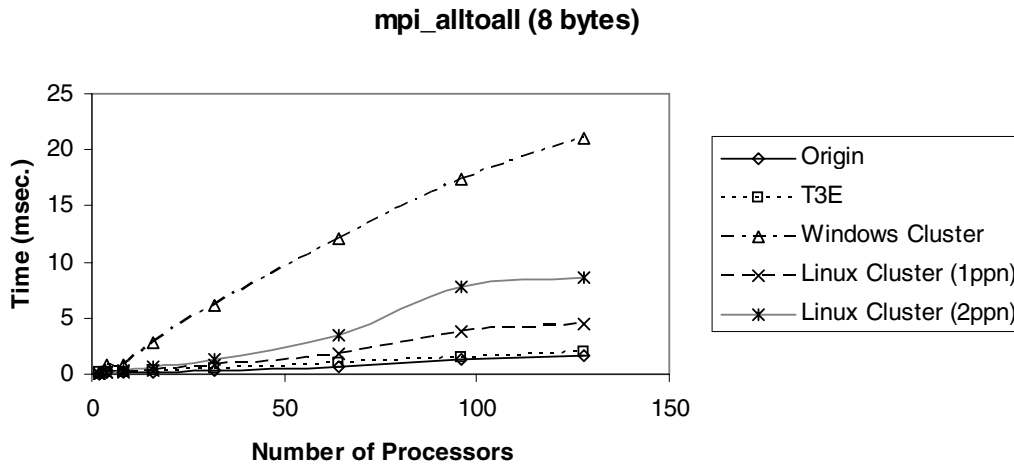


Figure 20. Test 7 (mpi\_allgather) plotting (execution time)/( $p - 1$ ), where  $p$  is the number of processors.

Figure 21. Test 8 (`mpi_alltoall`) with times in milliseconds.

#### 4.8. Test 8: the all-to-all

For `mpi_alltoall`, each processor sends a distinct message of the same size to all other processors and hence produces a large amount of traffic on the communication network. This test measures the time to execute

```
call mpi_alltoall(C(1,0), n, mpi_real8, B(1,0), n, mpi_real8,  
mpi_comm_world, ierror)
```

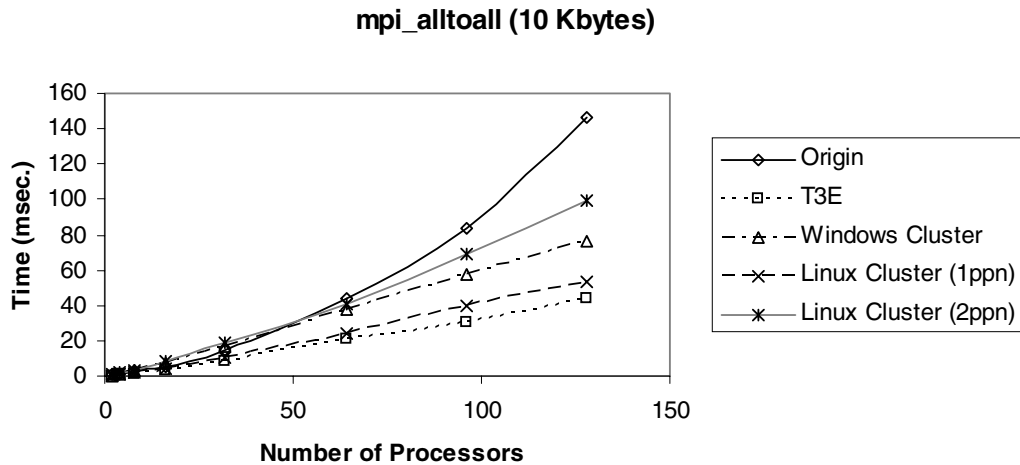
for  $n = 1$  and  $n = 1250$ .

Figures 21 and 22 present the performance data for this test. Note that for the 8-byte message, both the T3E and the Origin perform better than the Linux Clusters and significantly better than the Windows Cluster. However, for the 10-KB message, the Origin does not perform as well as the other machines. Table VII shows the performance of all machines relative to the T3E for 128 processors.

Like the `mpi_allgather`, the `mpi_alltoall` is usually implemented as cyclically shifting the messages on the  $p$  processors [7]. Initially processor  $i$  owns  $p$  messages of size  $M$  bytes denoted by  $m_i^0, m_i^1, \dots, m_i^{p-1}$ . At the step  $0 < j < p$ , each processor  $i$  sends  $m_i^{(p+(i-j)) \bmod p}$  to the processor  $(i - j) \bmod p$ . Thus the execution time would be

$$(p - 1)(\alpha + M\beta)$$

Thus, (execution time)/( $p - 1$ ) will be a constant for all such  $p$  for a fixed message size, just like the case of the `mpi_allgather`. Then, plotting (execution time)/( $p - 1$ ) will provide a way to better understand the scalability of `mpi_alltoall` for each machine. Figures 23 and 24 show these results.

Figure 22. Test 8 (`mpi_alltoall`) with times in milliseconds.Table VII. Time ratios for 128 processors for the `mpi_alltoall` test.

Message size	8 bytes	10 KB
Origin/T3E	0.9	3.3
Windows Cluster/T3E	10.9	1.72
Linux Cluster (1 ppn)/T3E	2.32	1.2
Linux Cluster (2 ppn)/T3E	4.48	2.24

Note that though the Origin scales well for the 8-byte message, it shows really poor scalability for a bigger message size.

#### 4.9. Test 9: the reduce

This test measures the time to execute:

```
call mpi_reduce(A,C,n,mpi_real8,mpi_sum,0,mpi_comm_world,ierror)
```

for  $n = 1$  and  $n = 125\,000$ .

Figures 25 and 26 present the performance data for MPI reduce with the sum operation. The results of the min and max operations are similar. The Origin has the best performance for 1-MB messages

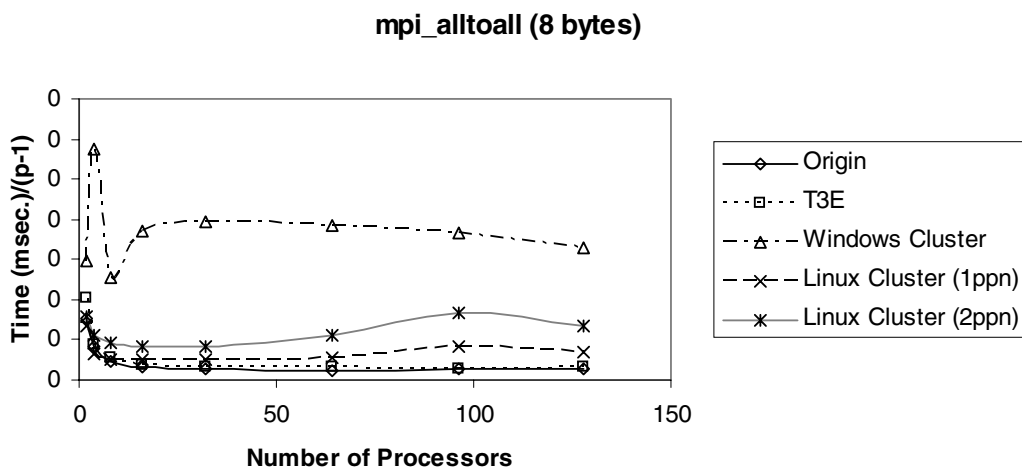


Figure 23. Test 8 (`mpi_alltoall`) plotting (execution time)/( $p - 1$ ), where  $p$  is the number of processors.

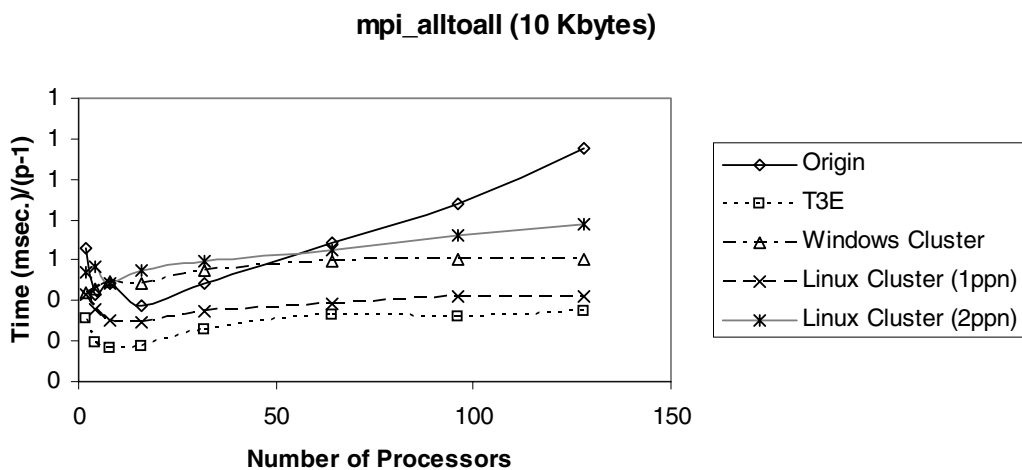


Figure 24. Test 8 (`mpi_alltoall`) plotting (execution time)/( $p - 1$ ), where  $p$  is the number of processors.

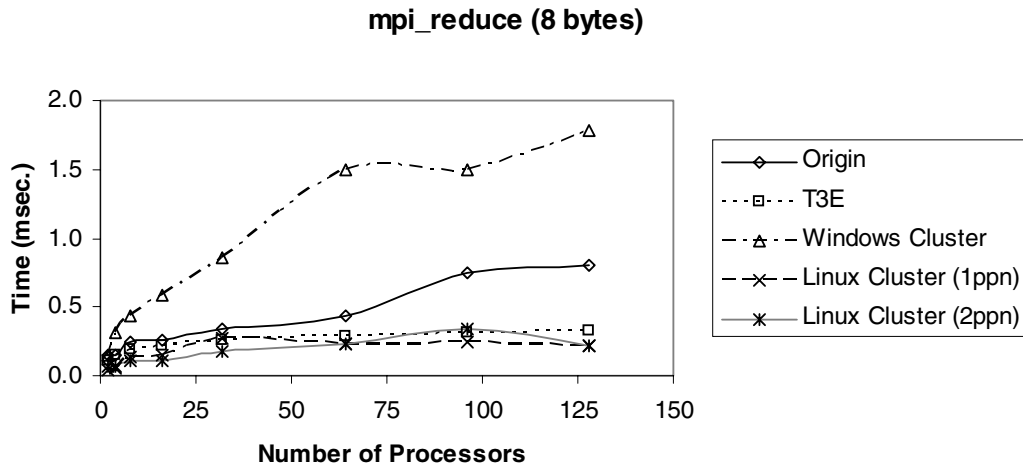


Figure 25. Test 9 (mpi\_reduce sum) with times in milliseconds.

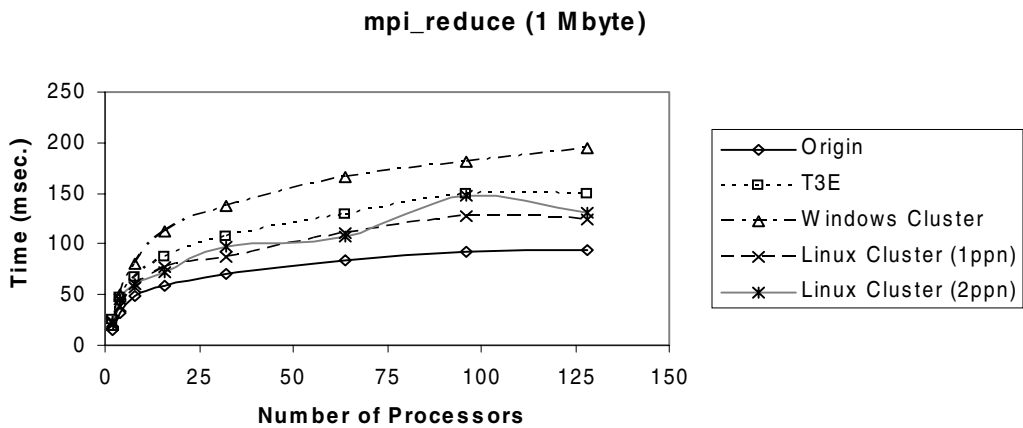


Figure 26. Test 9 (mpi\_reduce sum) with times in milliseconds.

and the Windows Cluster again gives the worst performance for 8-byte messages. Table VIII shows the performance of all machines relative to the T3E for 128 processors.

The discussion of the scalability of this algorithm is beyond the scope of this paper; see [7] for an algorithm for implementing mpi\_reduce.





Table VIII. Time ratios for 128 processors for the `mpi_reduce` test.

Message size	8 bytes	1 MB
Origin/T3E	2.5	0.6
Windows Cluster/T3E	5.4	1.3
Linux Cluster (1 ppn)/T3E	0.65	0.83
Linux Cluster (2 ppn)/T3E	0.67	0.87

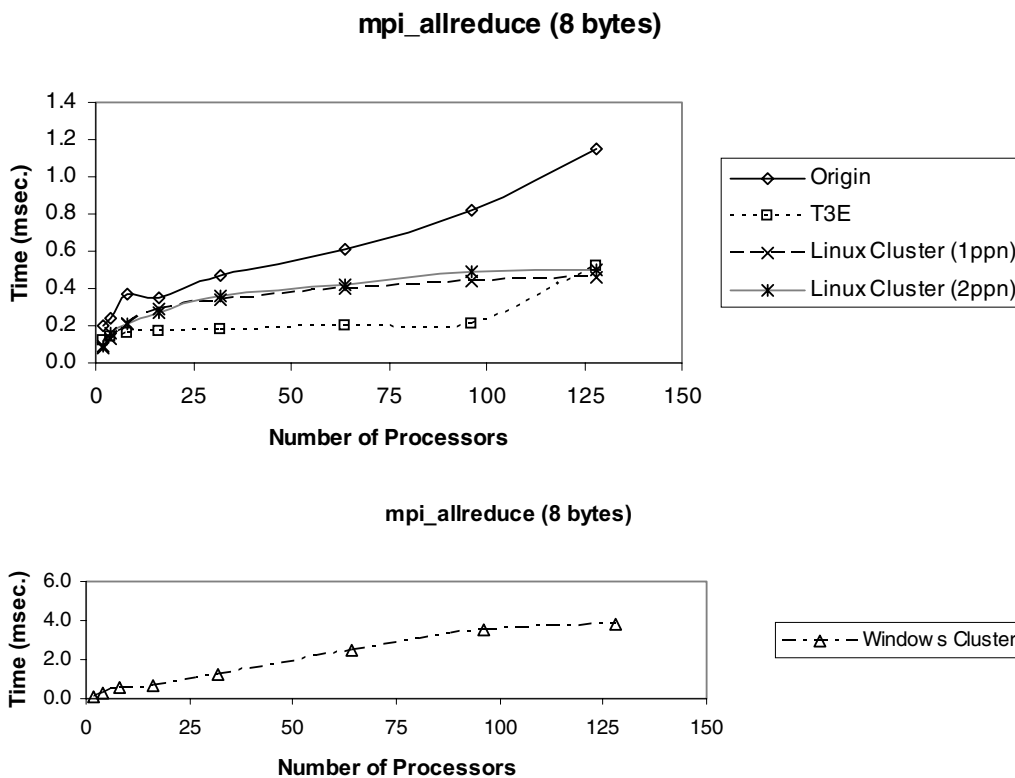


Figure 27. Test 10 (`mpi_allreduce` sum operation) with times in milliseconds.

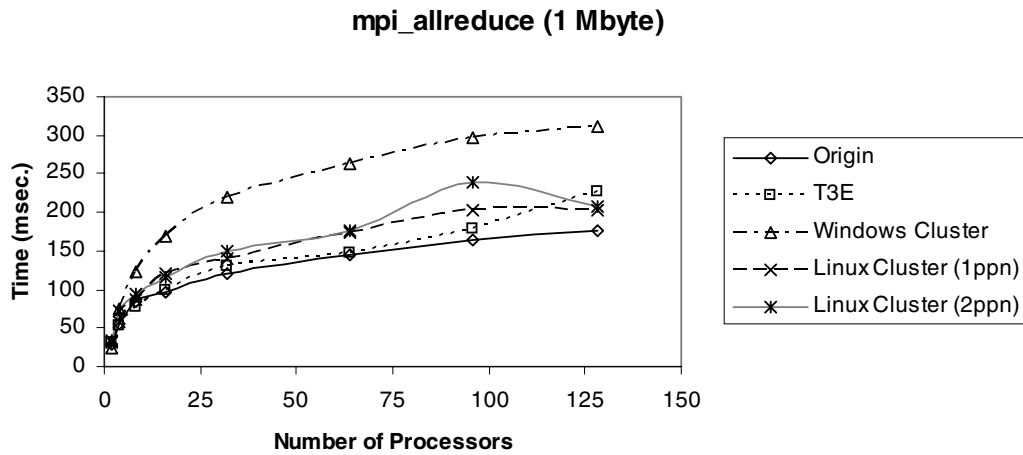


Figure 28. Test 10 (`mpi_allreduce` sum operation) with times in milliseconds.

Table IX. Time ratios for 128 processors for the `mpi_allreduce` test.

Message size	8 bytes	1 MB
Origin/T3E	2.2	0.78
Windows Cluster/T3E	7.4	1.37
Linux Cluster (1 ppn)/T3E	0.88	0.89
Linux Cluster (2 ppn)/T3E	0.97	0.91

#### 4.10. Test 10: the all-reduce

The `mpi_allreduce` is same as the `mpi_reduce` except that the result is sent to all processors instead of only to the root processor. This test measures the time to execute

```
call mpi_allreduce(A,C,n,mpi_real8,mpi_sum,mpi_comm_world,ierror)
```

for  $n = 1$  and  $n = 125\,000$ .

Figures 27 and 28 present the performance data for `mpi_allreduce` sum operation. The results of the min and max operations are similar, including the ‘spike’ on the T3E for the 8-byte message when 128 processes were used in the communications. Except this ‘spike’, the T3E performs best for the 8-byte message size. For the 1-MB message, the Origin performs the best with the T3E and the Linux Cluster showing close performance. Table IX shows the performance of all machines relative to the T3E for 128 processors.

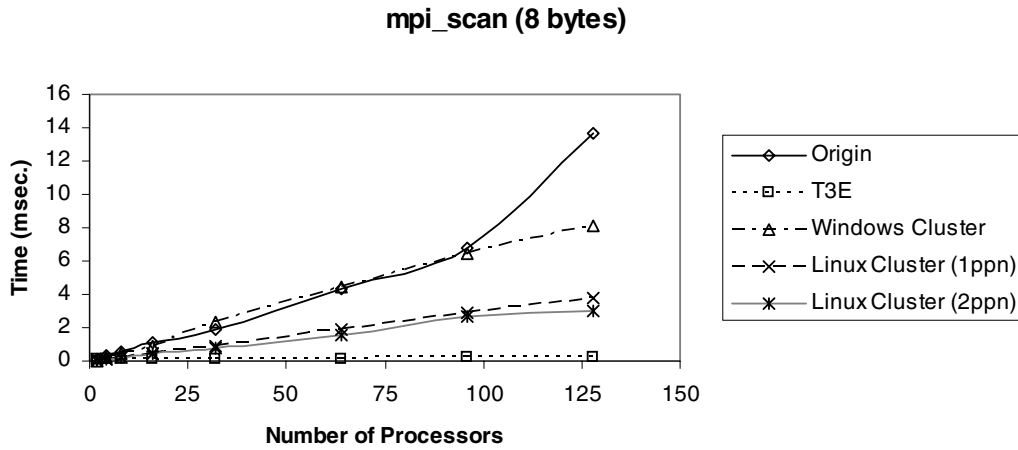


Figure 29. Test 11 (mpi\_scan) with times in milliseconds.

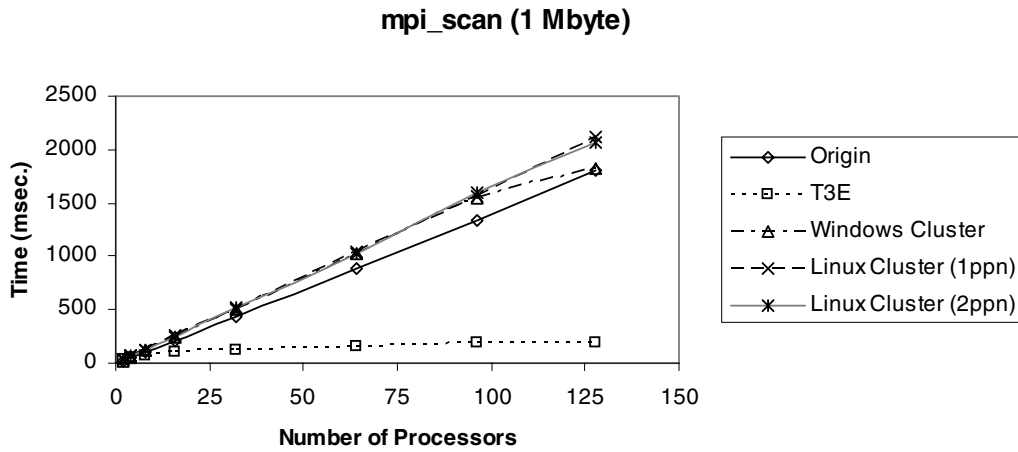


Figure 30. Test 11 (mpi\_scan) with times in milliseconds.

Table X. Time ratios for 128 processors for the `mpi_scan` test.

Message size	8 bytes	1 MB
Origin/T3E	76	9.5
Windows Cluster/T3E	45.6	9.6
Linux Cluster (1 ppn)/T3E	21	11
Linux Cluster (2 ppn)/T3E	17	10.9

#### 4.11. Test 11: the scan

This test measures the time to execute

```
call mpi_scan(A, C, n, mpi_real8, mpi_sum, mpi_comm_world, ierror)
```

for  $n = 1$  and  $n = 125\,000$ . `Mpi_scan` is used to perform a prefix reduction on data distributed across the group. The operation returns, in the receive buffer of the process with rank  $i$ , the reduction of the values in the send buffers of processes with ranks  $0, \dots, i$  (inclusive).

Figures 29 and 30 present the performance data. The results for the min and max operations are similar. Note that the T3E performs and scales significantly better than all the other machines. Table X shows the performance of all machines relative to the T3E for 128 processors.

## 5. CONCLUSION

The purpose of this paper is to compare the communication performance and scalability of MPI communication routines on a Windows Cluster, a Linux Cluster, a Cray T3E-600, and an SGI Origin 2000. All tests in this paper were run using various numbers of processors and two message sizes. In spite of the fact that the Cray T3E-600 is about 7 years old, it performed best of all machines for most of the tests. The Linux Cluster with the Myrinet Interconnect and Myricom's MPI performed and scaled quite well and, in most cases, performed better than the Origin 2000, and in some cases better than the T3E. The Windows Cluster using the Giganet Full Interconnect and MPI/Pro's MPI performed and scaled poorly for small messages compared with all of the other machines. These high-latency problems have been reported to MPI Software Technology, Inc. for fixing.

## ACKNOWLEDGEMENTS

We thank SGI and Cray Research for allowing us to use their Origin 2000 and T3E-600 located in Eagan, Minnesota. We would like to thank the National Center for Supercomputing Applications at the University of Illinois in Urbana-Champaign, Illinois, for allowing use to use their IA-32 Linux Cluster. We would also like to thank the Cornell Theory Center for access to their Windows Clusters. This work utilizes the cluster at the Computational Materials Institute.



---

**REFERENCES**

1. Snir M, Otto SW, Huss-Lederman S, Walker DW, Dongarra J. *MPI, the Complete Reference. Scientific and Engineering Computation*. MIT Press: Cambridge, MA, 1996.
2. Cray Research Web Server. <http://www.cray.com> [November 2002].
3. Origin Server. *Technical Report*. Silicon Graphics, April 1997.
4. Anderson A, Brooks J, Grassl C, Scott S. Performance of the CRAY T3E multi-processor. *Proceedings of Supercomputing 97*, San Jose, CA, 15–21 November 1997. ACM Press: New York, 1997.
5. Lifka D. High performance computing with Microsoft Windows 2000. <http://www.tc.cornell.edu/ac3/tech/cluster20012.pdf> [November 2002].
6. NCSA IA-32 Linux Cluster. <http://www.ncsa.uiuc.edu/UserInfo/Resources/Hardware/IA32LinuxCluster/> [November 2002].
7. Luecke GR, Raffin B, Coyle JJ. The performance of the MPI collective communication routines for large messages on the Cray T3E600, the Cray Origin 2000, and the IBM SP. *Journal of Performance Evaluation and Modelling for Computer Systems*, July 1999. <http://dsg.port.ac.uk/Journals/PEMCS/papers/paper10.pdf>.