Performance and Scalability of MPI on PC Clusters

Glenn R. Luecke, Jing Yuan, Silvia Spanoyannis, Marina Kraeva
grl@iastate.edu, yjing@iastate.edu, spanoyan@iastate.edu, kraeva@iastate.edu
292 Durham Center
Iowa State University
Ames, Iowa 50011, USA
January 23, 2000

Abstract. The purpose of this paper is to compare the communication performance and scalability of MPI communication routines on an NT cluster, a Myrinet Linux cluster, an Ethernet Linux cluster, a Cray T3E-600, and an SGI Origin 2000. All tests in this paper were run for the various numbers of processors and 2 message sizes. For most of the MPI tests used in this paper, the T3E-600 and Origin 2000 outperform the NT cluster, the Myrinet and Ethernet Linux clusters. In spite of the fact that the Cray T3E-600 is about 5 years old, it performs best of all machines for most of the tests. For `mpi_bcast`, `mpi_allgather`, and `mpi_alltoall`, the Myrinet Linux cluster outperforms the NT cluster. For all other MPI collective routines, the NT cluster outperforms the Myrinet Linux cluster. For all MPI collective routines, the Myrinet Linux cluster performs significantly better than the Ethernet Linux cluster.

1 Introduction

Today, MPI [6] 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. Today, 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 an NT cluster, a Myrinet Linux cluster, an Ethernet Linux cluster, a Cray T3E-600, and an SGI Origin 2000.

2 Test Environment

The SGI Origin 2000 [1, 4] used for this study is a 128-processor machine in which each processor is MIPS R10000 running at 250Mhz. There are two levels of cache: 32*1024 byte first level instruction and data caches, and an 4*1024*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 [1, 5] used is a 512-processor machine located in Eagan, Minnesota. Each processor is a DEC Alpha EV5 microprocessor running at 300 MHz. There were two levels of cache: 8*1024 byte first level instruction and data caches and a 96*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 NT cluster of PCs [3] used is a 128-processor machine located at the National Center for Supercomputing Applications in Urbana-Champaign, Illinois. The cluster is a 64-nodes machine, each node consisting of a dual processor Intel 550 MHz Xeon Pentium III. There are two levels of cache: 16*1024 byte first level instruction and data caches and a 512*1024 byte second level cache for both data and instructions. Nodes are interconnected via a Myrinet network (full-duplex 1.28+1.28 Gigabit/second). The system was running Windows NT Server 4.0 and HPVM 1.1 for Myrinet-clustered compute nodes developed by the Concurrent Systems Architecture Group at the University of Illinois. For all tests, the MPI library and its HPVM interface were used (mpi.lib, fm.lib, kernel32.lib, advapi32.lib, and wsock32.lib).

The Linux cluster of PCs [2] used is a 128-processor machine located at the Albuquerque High Performance Computing Center in Albuquerque, New Mexico. The cluster is a 64-nodes AltaCluster, by Alta Technology Corporation, each node consisting of a dual processor Intel 450 MHz Pentium II. There are two levels of cache: 16*1024 byte first level instruction and data caches and a 512*1024 byte second level cache for both data and instructions. Nodes were interconnected via a Myrinet (full-duplex 1.28+1.28 Gigabit/second) or Fast Ethernet (full-duplex 100Mbit/second) network. The system was running Linux 2.2.10. For all tests, version 3.0-1 Portland Group Fortran compiler with the -O3 compiler option, and the MPICH-GM for the Myrinet Linux cluster or MPICH-ETH library for the Ethernet Linux cluster were used.

The performance results reported in this paper were obtained with a large message size and a small message size, all using 8 bytes 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 collective routine …

  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*1024 bytes), the Origin 2000 (8*1024*1024 bytes), the NT and the Linux clusters of PCs (512*1024). 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 [4]. 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 bytes message with 128 processors on the Origin.

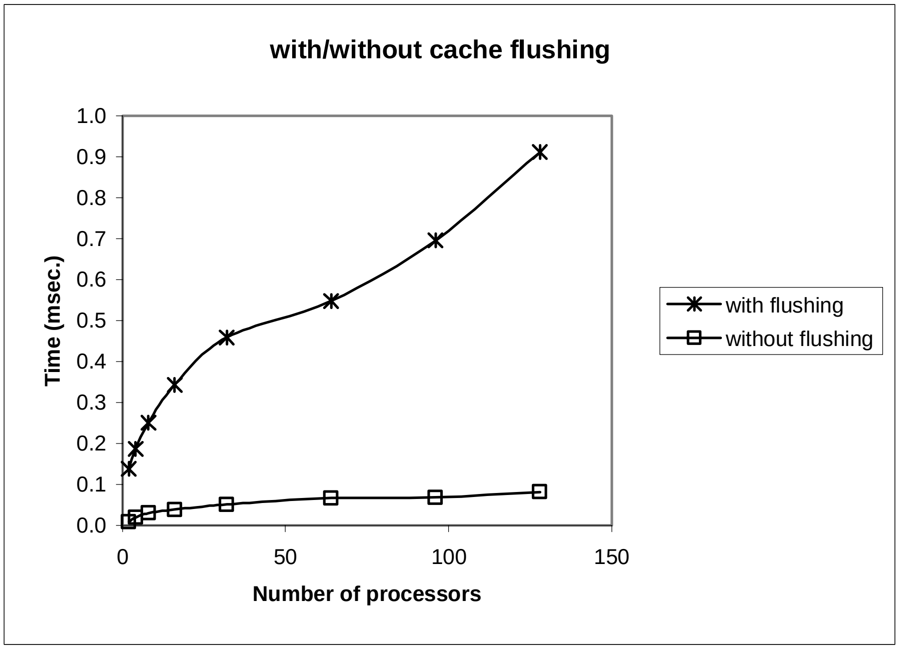


Figure 1: Execution times for `mpi_bcast` for an 8 bytes message on the Origin.

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 that the statement "**A(1) = A(1) + flush(mod(k, ncache)**" was added into 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 Kbytes message. Notice that there are several "spikes" in the data with the first spike being the first timing. The first timing usually was 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 was 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 milliseconds, 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 milliseconds instead of 94.7 milliseconds.

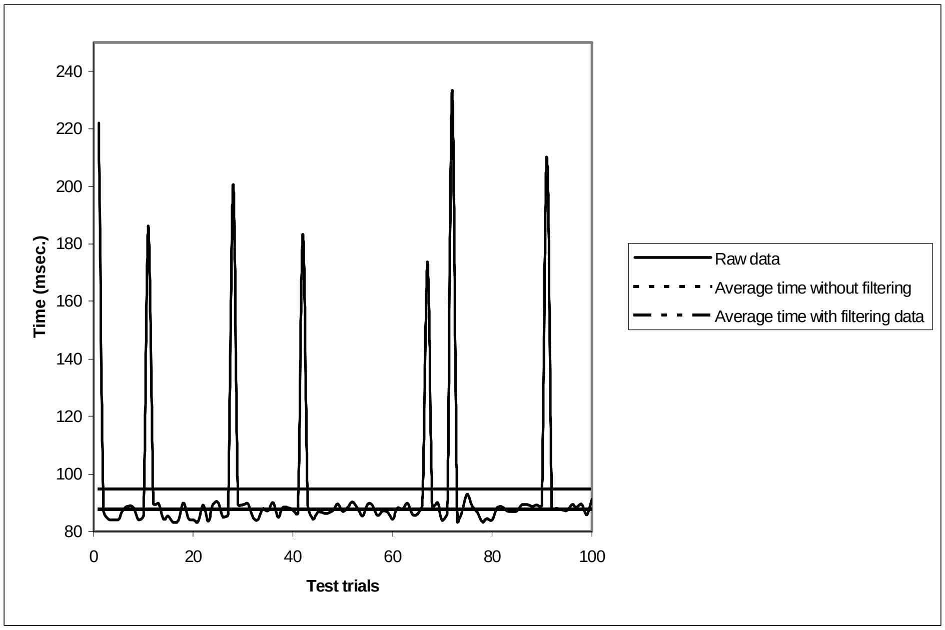


Figure 2: Time for 100 trials for `mpi_bcast` for a 10 Kbytes message on the Origin with 128 processors.

4 Test Descriptions and Performance Results

This section contains performance results for 11 MPI communication tests using 2, 4, 8, 16, 32, 64, 96 and 128 processors on all machines using `mpi_comm_world` for the communicator.

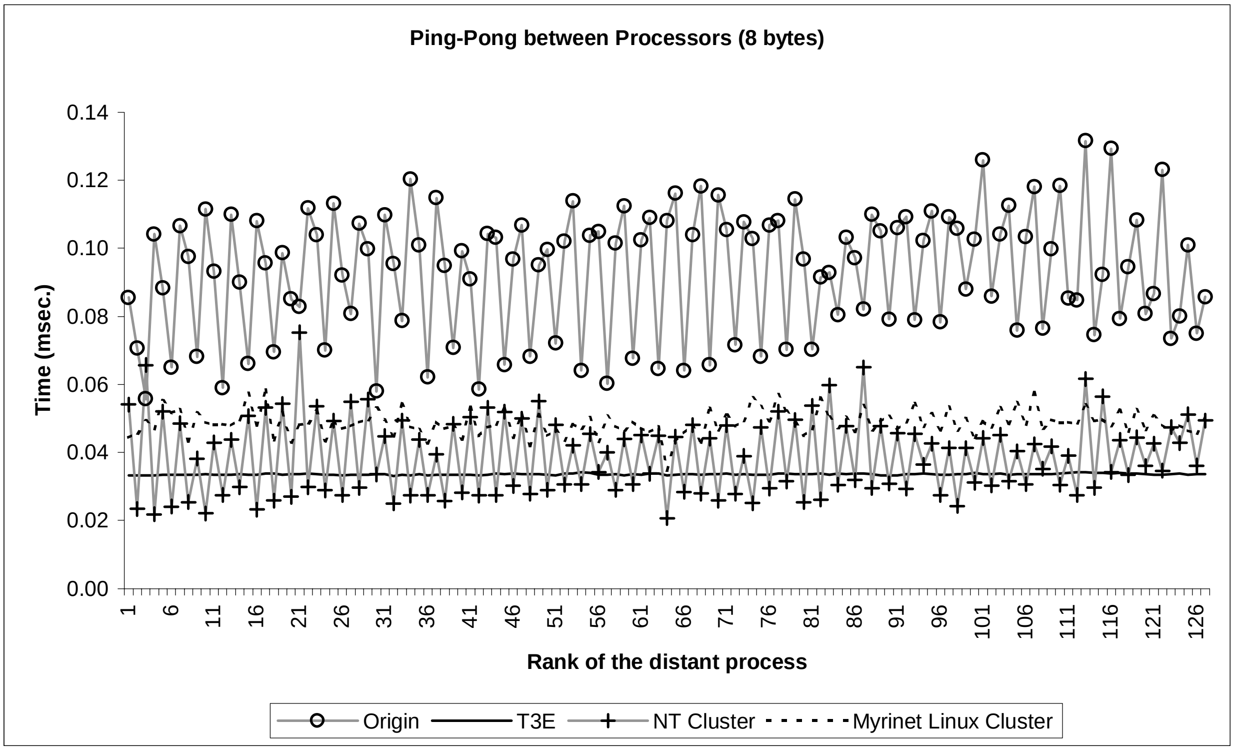
Test 1: Ping-Pong Between Processors

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 to 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:

```
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 would 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. Figure 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. Notice that many of the graphs are "reasonably close" to this ideal. The Origin has the largest variation in times for this test for an 8 bytes message. Notice that the Ethernet Linux cluster performs roughly 18 times slower for 8 bytes and 5 times slower for 1 Mbyte than the Myrinet Linux cluster.

Notice that for 8 bytes messages, the Myrinet Linux cluster shows better behavior that the Myrinet NT cluster. This seems to indicate a problem in the MPI implementation for the Myrinet NT cluster. The authors do not have access to any tools to determinate why there is so much variation in the performance for 8 bytes messages for the Origin.



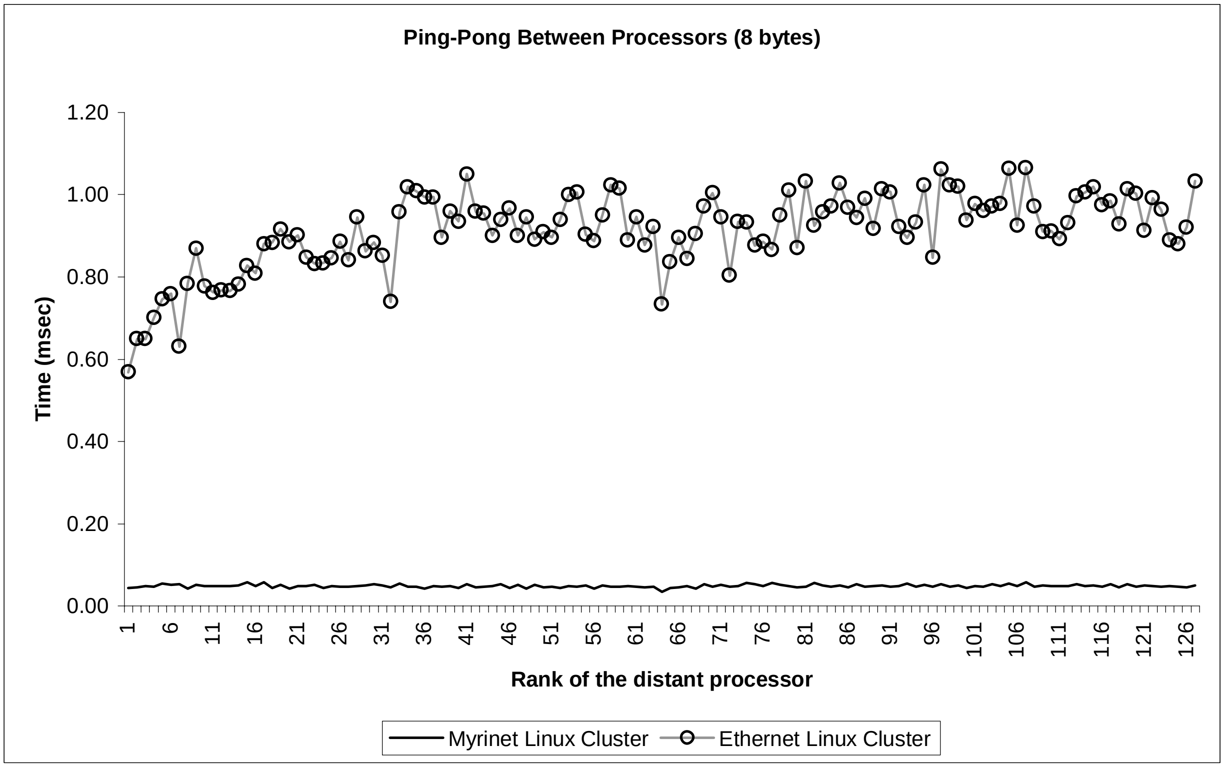
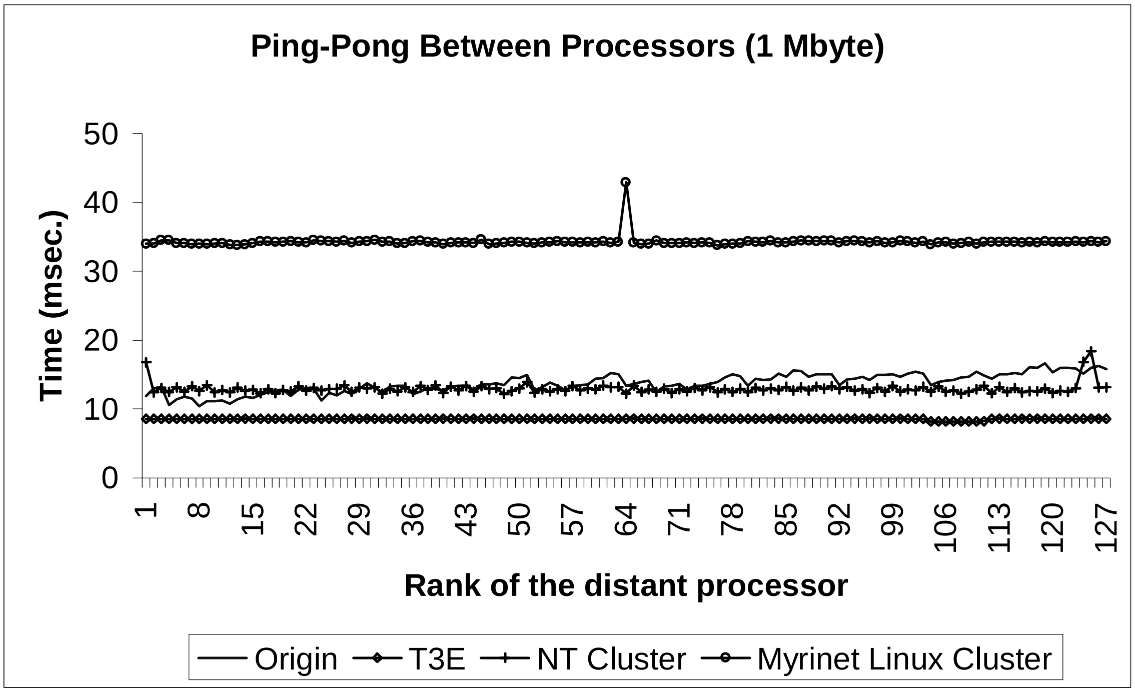


Figure 3: Test 1 (Ping Pong Between Processors) with times in milliseconds



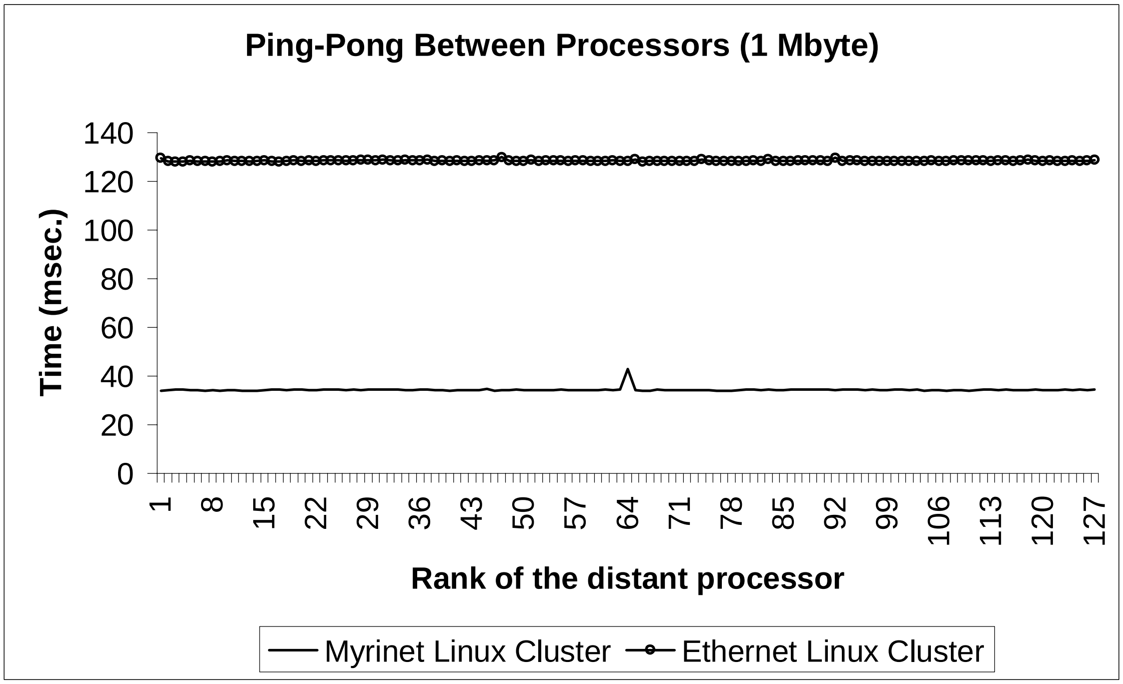


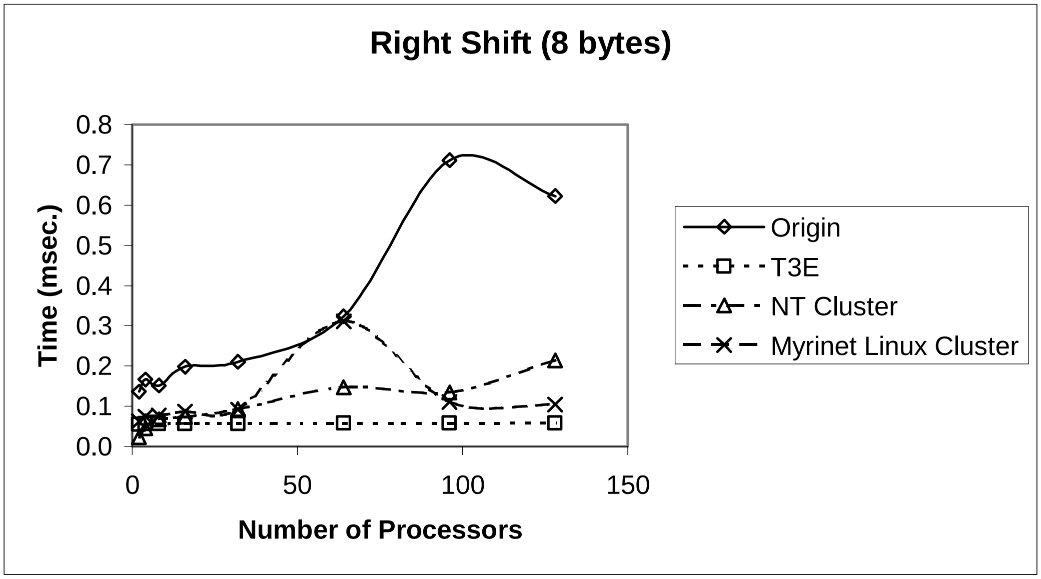
Figure 4: Test 1 (Ping Pong Between Processors) with times in milliseconds

Test 2: The Right Shift

The purpose of this test is to measure the performance of the "right shift" where each processor receives a message of size n from its "left" neighbor, i.e., `modulo(myrank-1,p)` is the left neighbor of `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,modulo(myrank+1,p),tag,b,n, &
      mpi_real8,modulo(myrank-1,p),tag,mpi_comm_world,status,ierr)
```

Figure 5 and 6 present the performance data for this test. Notice that for both message sizes, the T3E performs and scales best. The Myrinet Linux cluster scales well compared with the rest of machines.



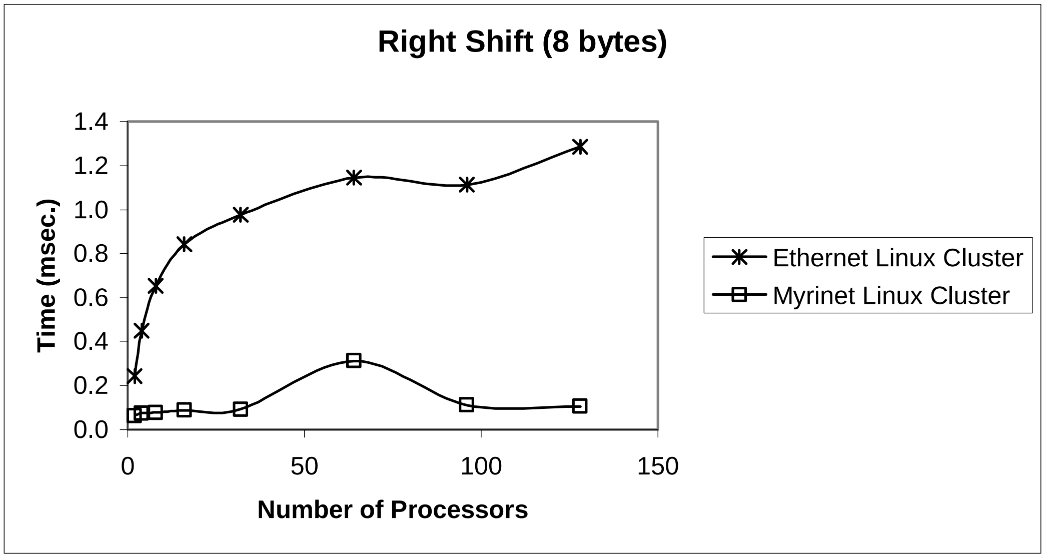
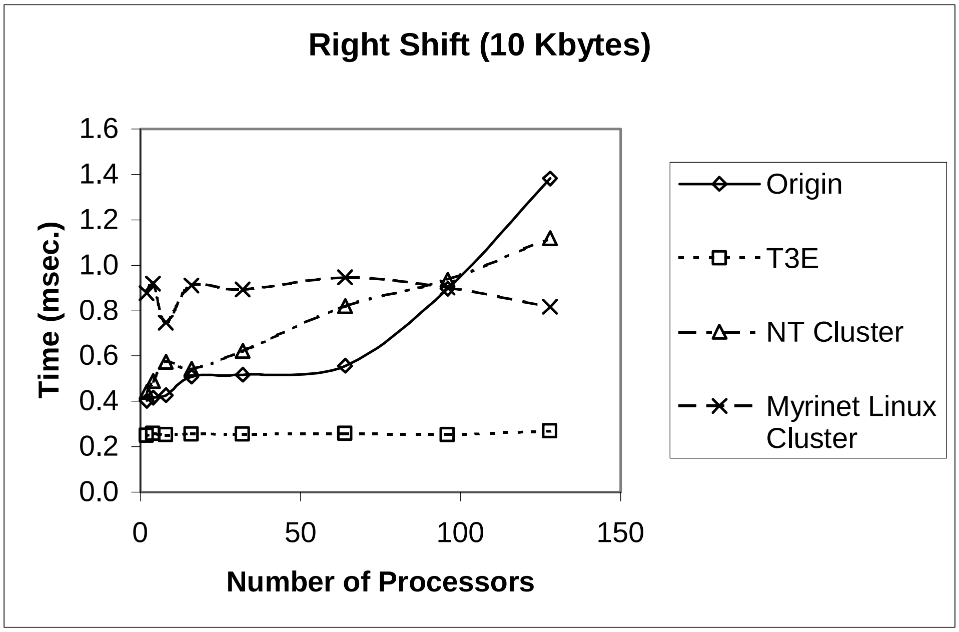


Figure 5: Test 2 (Right Shift) with times in milliseconds



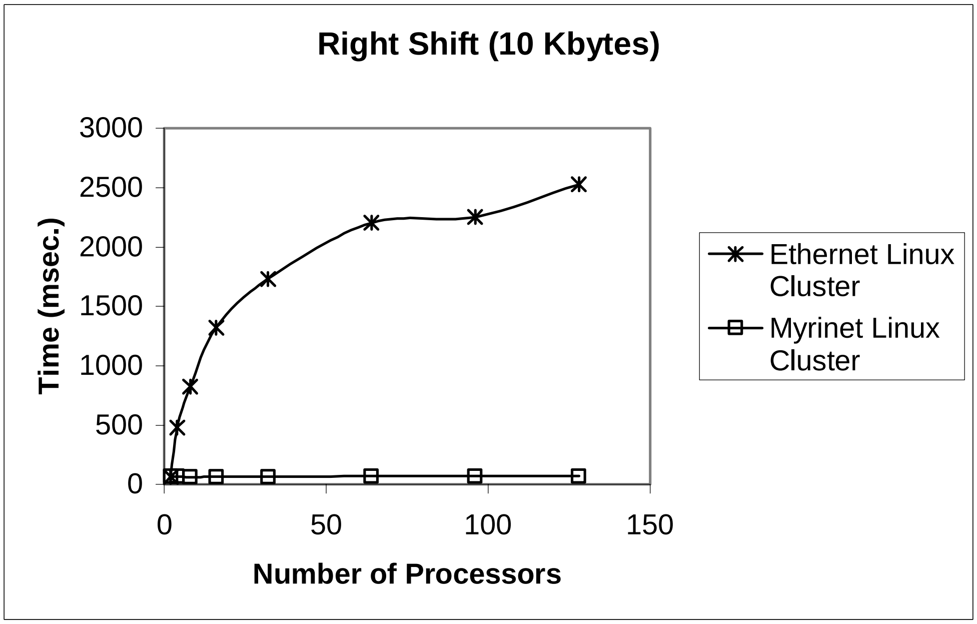


Figure 6: Test 2 (Right Shift) with times in milliseconds

Test 3: The Barrier

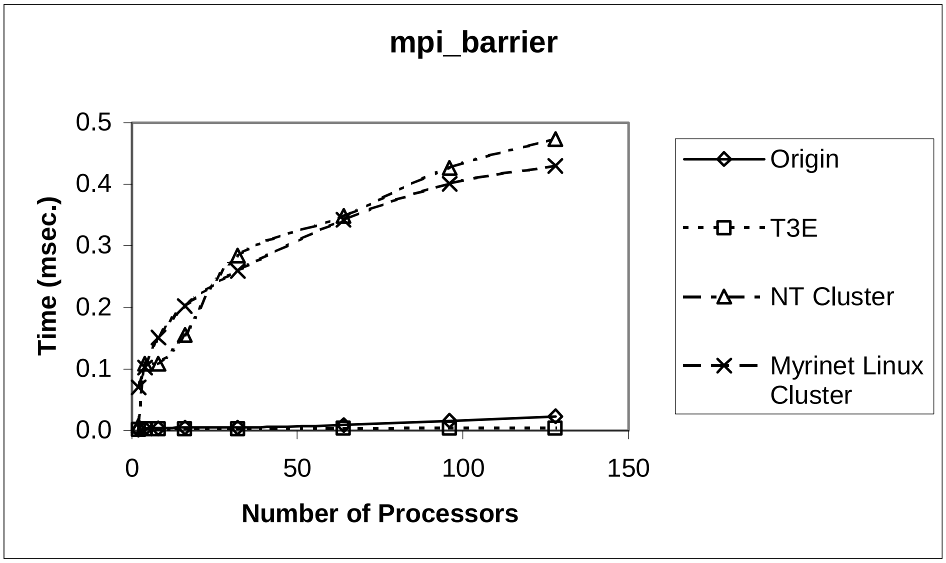
An important performance characteristic of a parallel computer is its ability to efficiently execute a barrier synchronization. This test evaluates the performance of the MPI barrier:

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

Figure 7 presents the performance and scalability data for `mpi_barrier`. Notice that the T3E and the Origin scale and perform significantly better than the NT cluster and the Myrinet Linux cluster. Also notice that the Myrinet Linux cluster significantly outperforms the Ethernet Linux cluster. Table 1 shows the performance of all machines relative to the T3E for 128 processors.

Origin/T3E	5.8
NT Cluster/T3E	116
Myrinet Linux Cluster/T3E	106
Ethernet Linux Cluster/T3E	933

Table 1: Time ratios for 128 processors for the mpi_barrier test.



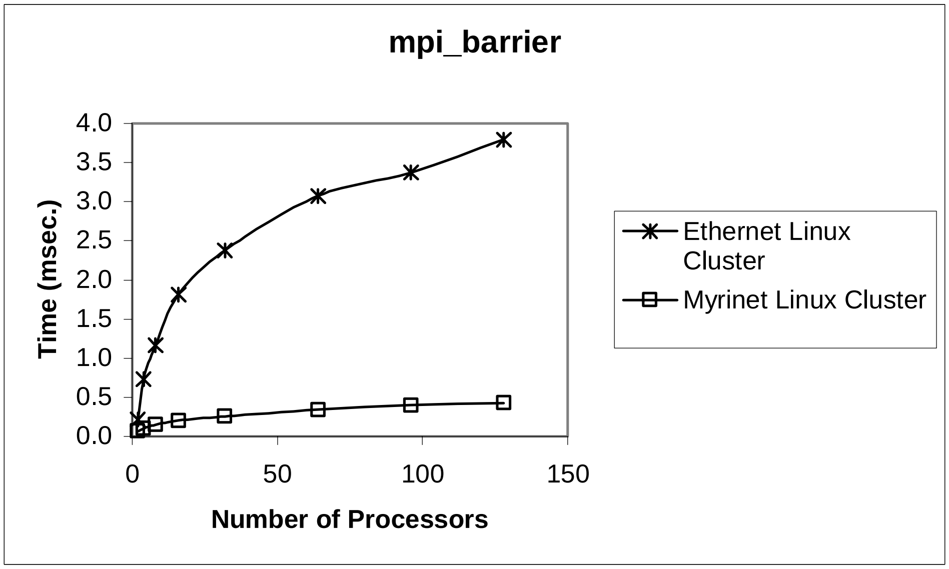


Figure 7: Test 3 (`mpi_barrier`) with times in milliseconds.

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=125000.

Figures 8 and 9 present the performance data for `mpi_bcast`. The NT cluster, the T3E and the Myrinet Linux cluster show good performance for 8 bytes messages while the Origin shows poor performance. However, for 1 Mbyte message, the Origin and T3E significantly outperform the two Myrinet clusters. Notice that the Myrinet Linux cluster significantly outperforms the Ethernet Linux cluster. Table 2 shows the performance of all machines relative to the T3E for 128 processors.

Message Size	8 bytes	1 Mbyte
Origin/T3E	2.5	1.5
NT Cluster/T3E	0.6	3.8
Myrinet Linux Cluster/T3E	1.3	5.6
Ethernet Linux Cluster/T3E	10	25.6

Table 2: Time ratios for 128 processors for the mpi_bcast test.

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

$$\alpha + M\beta.$$

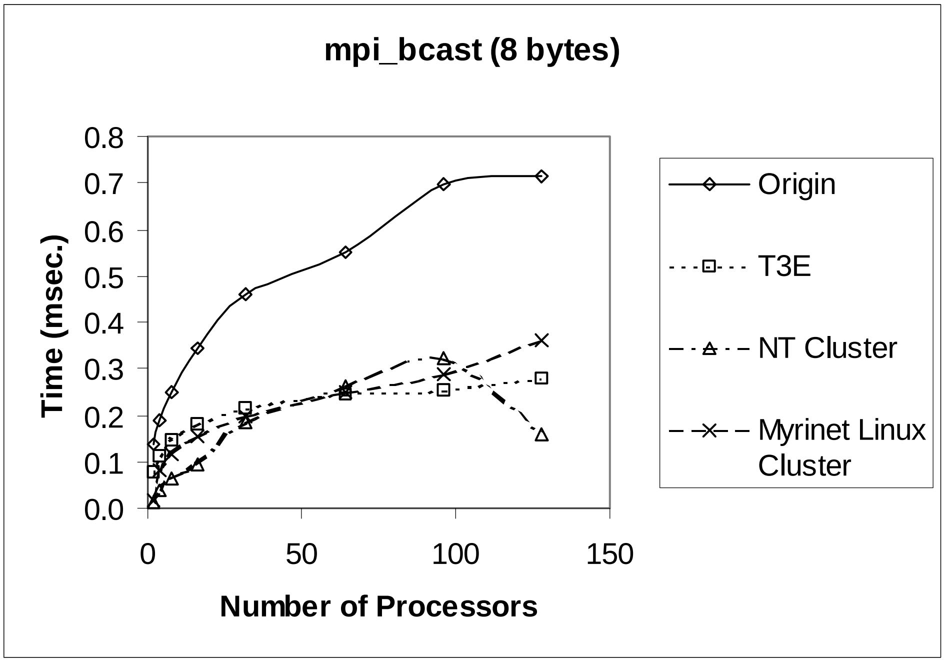
where α is the latency and β is the communication rate of the network. This assumes there is no contention and there is no difference in time when sending a message between any two processors. Assume that the number of processors, p, used is a power of 2, and assume that `mpi_bcast` is implemented using 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. Notice that the (execution time)/log(p) is nearly constant on all machines, except when p < 8.



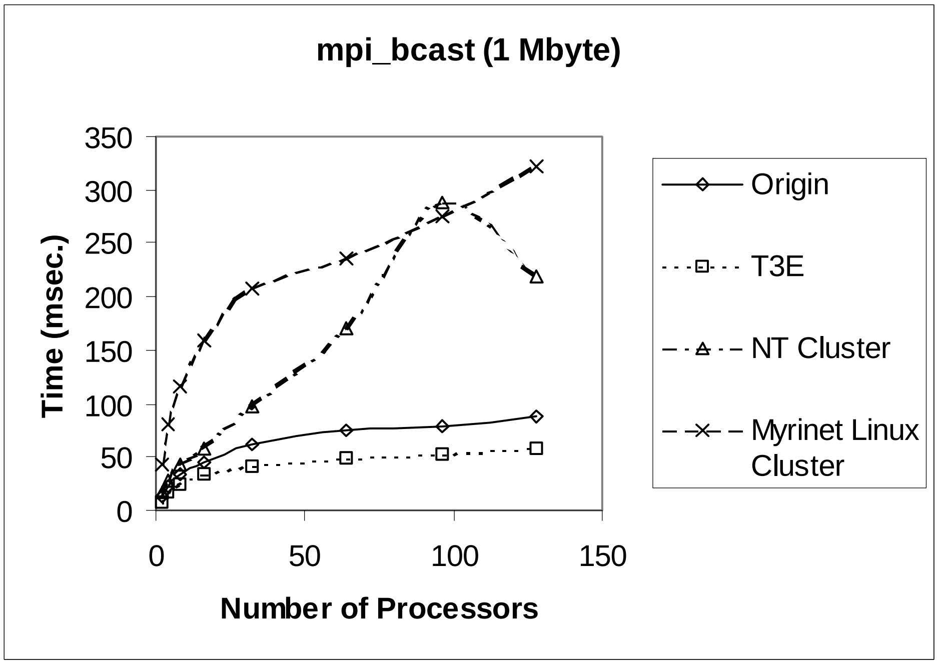
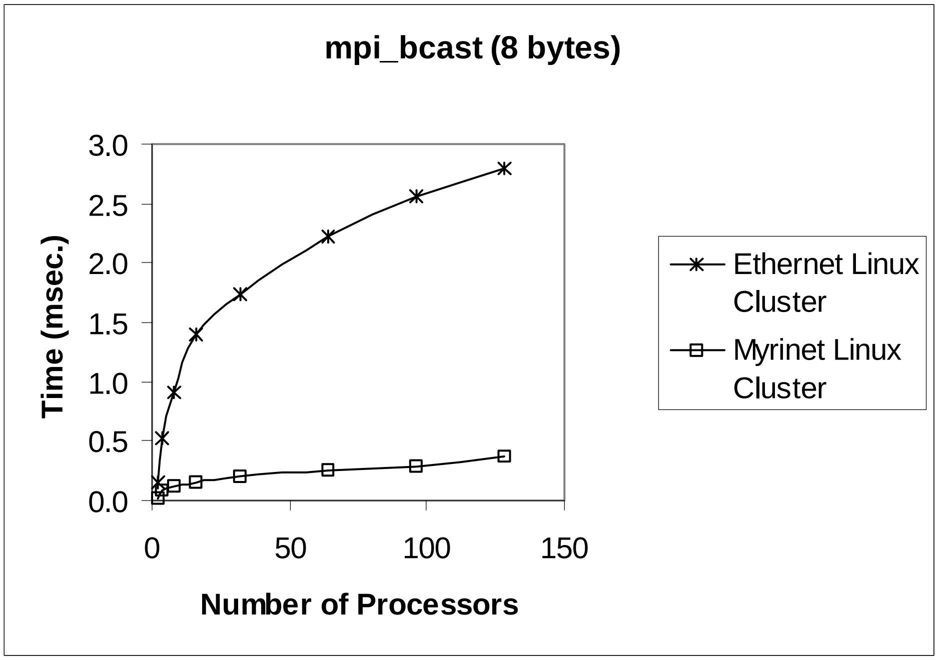


Figure 8: Test 4 (`mpi_bcast`) with times in milliseconds.



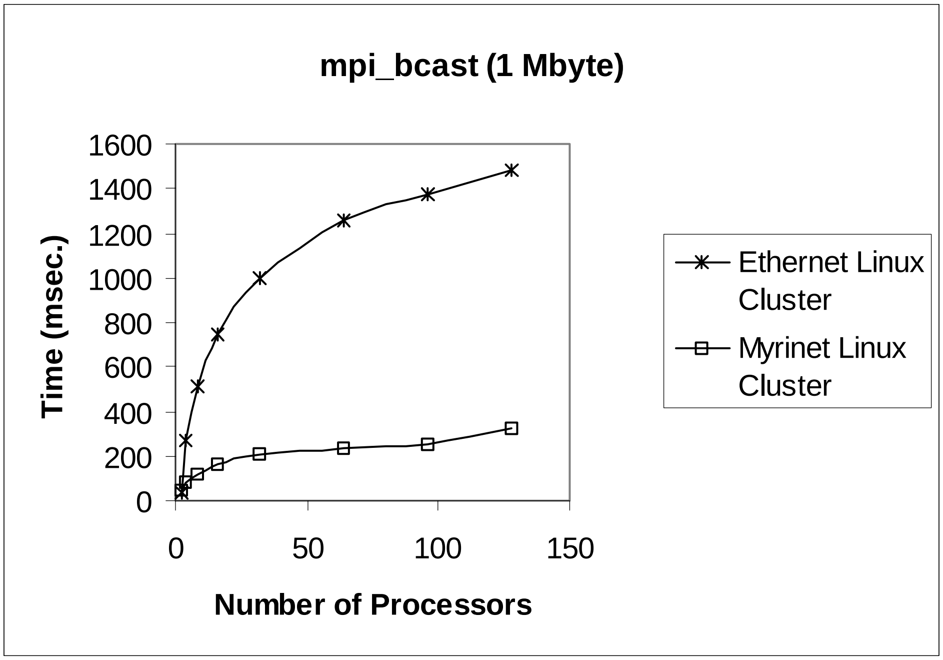
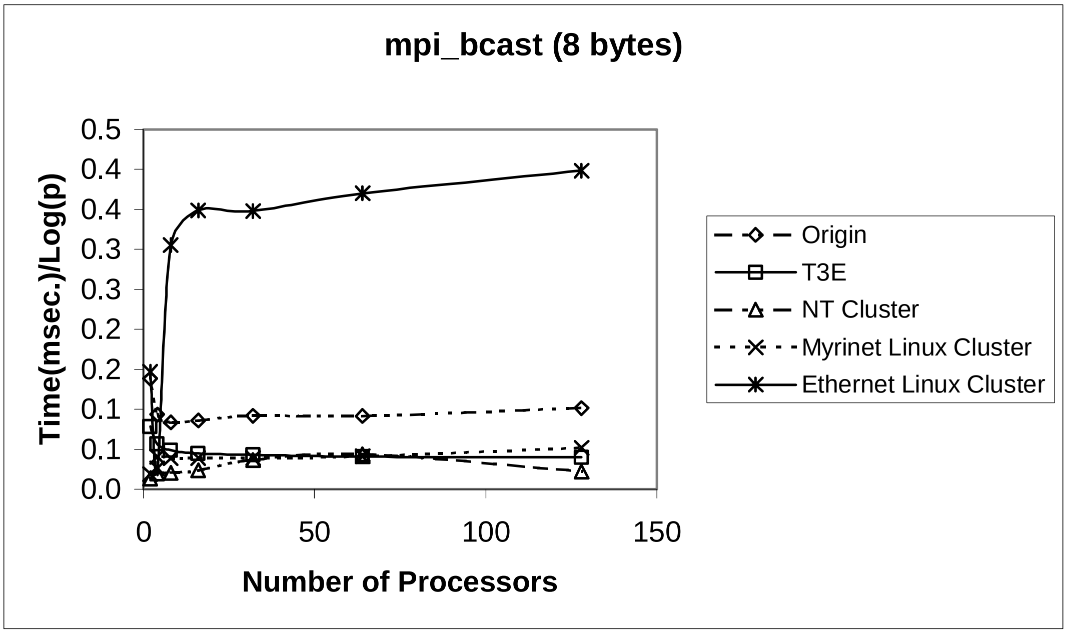


Figure 9: Test 4 (`mpi_bcast`) with times in milliseconds.



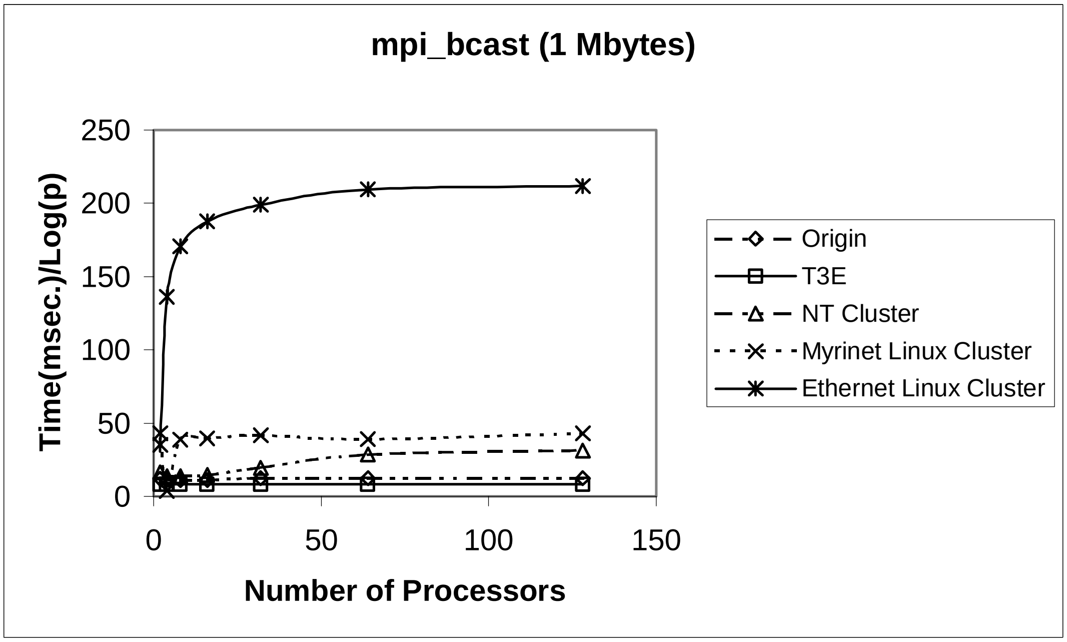


Figure 10: Test 4 (`mpi_bcast`) plotting (execution time)/log(p), where p is the number of the processors.

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 Figure 11 and 12. Table 3 shows the performance of all machines relative to the T3E for 128 processors.

Message Size	8 bytes	10 Kbytes
Origin/T3E	3	3.5
NT Cluster/T3E	1	3
Myrinet Linux Cluster/T3E	3.5	5
Ethernet Linux Cluster/T3E	19	228

Table 3: Time ratios for 128 processors for the mpi_scatter test.

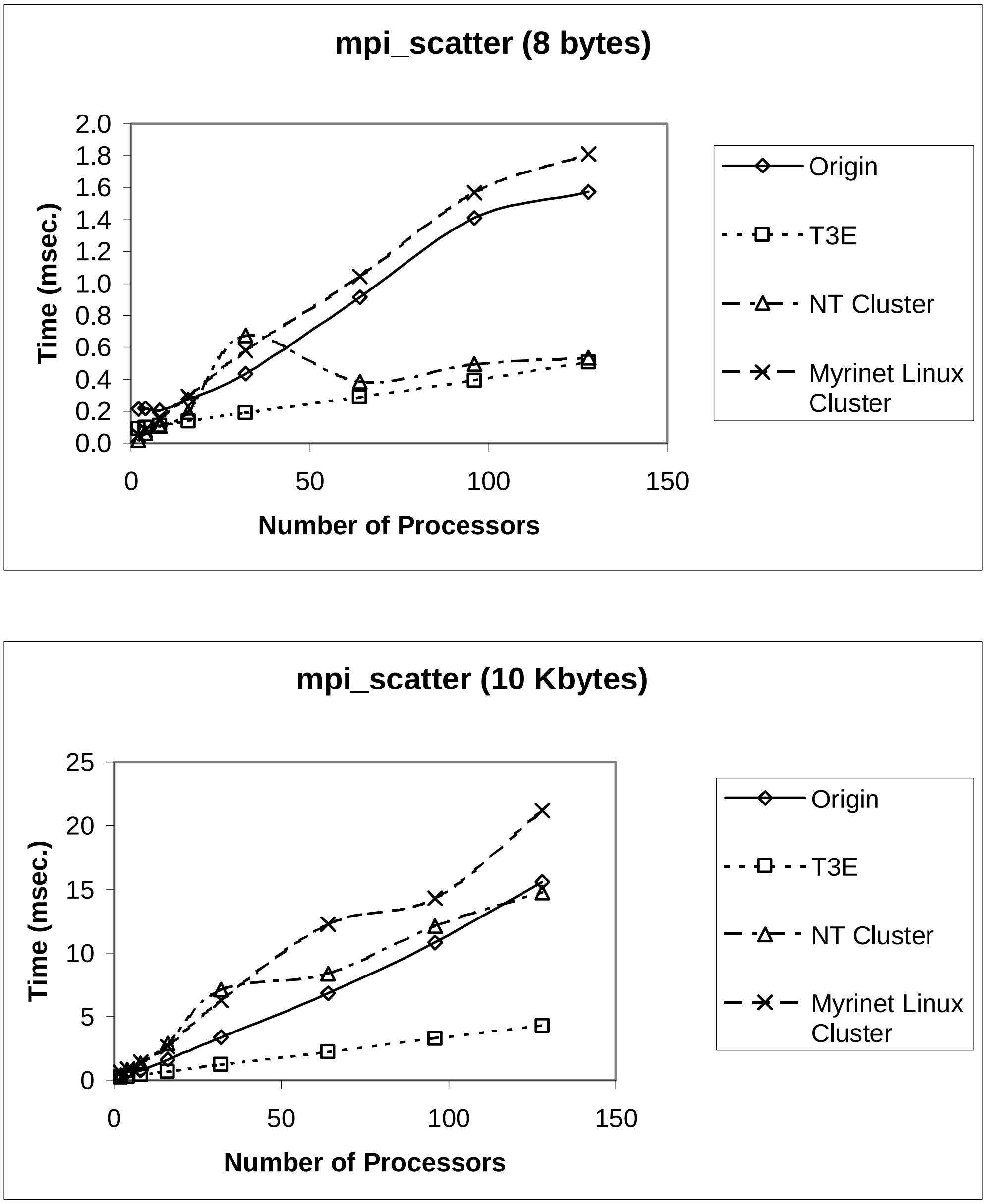
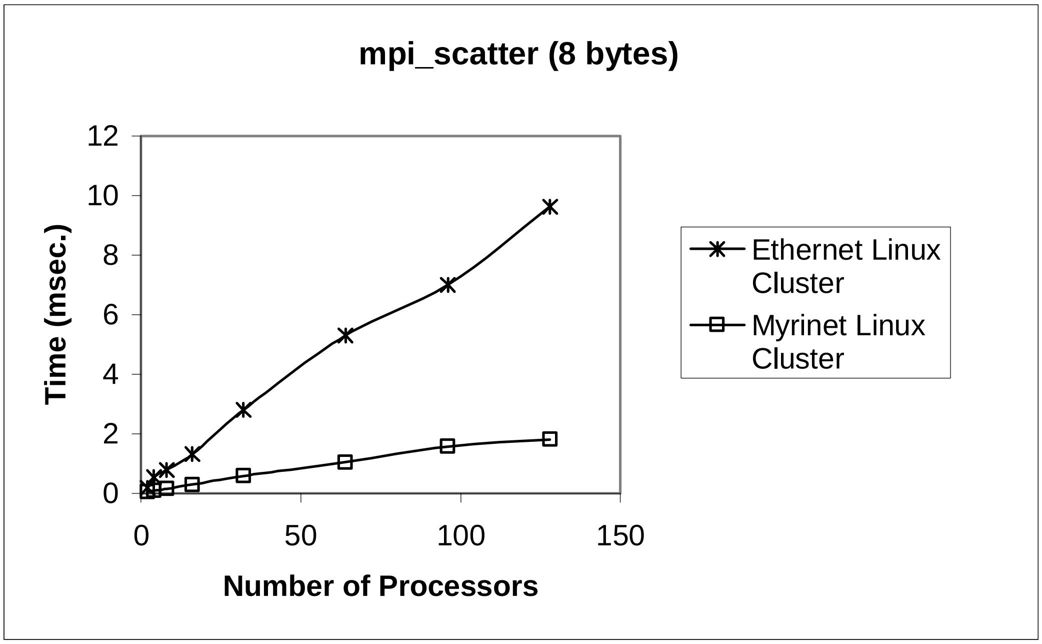


Figure 11: Test 5 (`mpi_scatter`) with times in milliseconds.



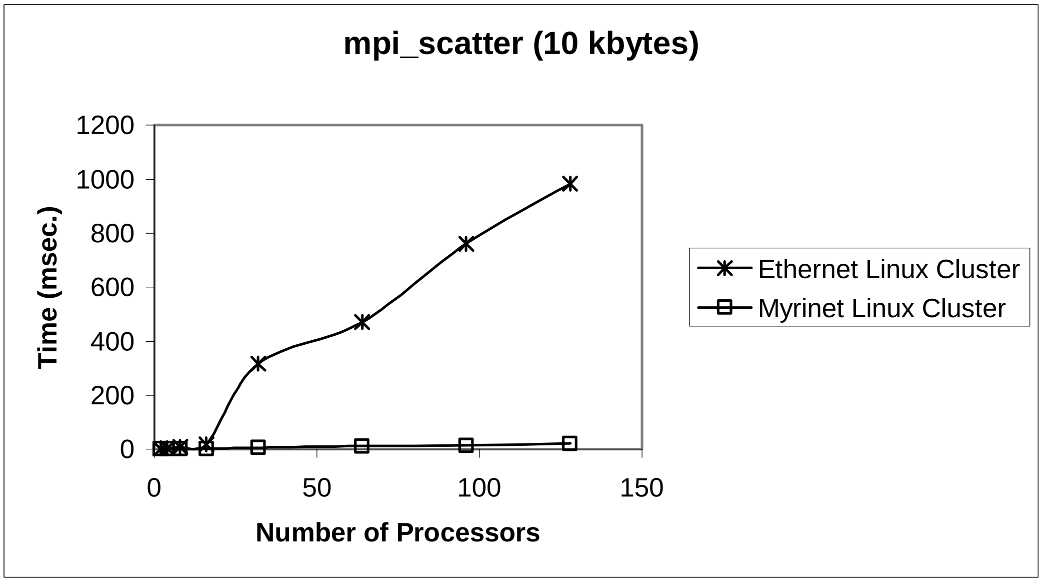
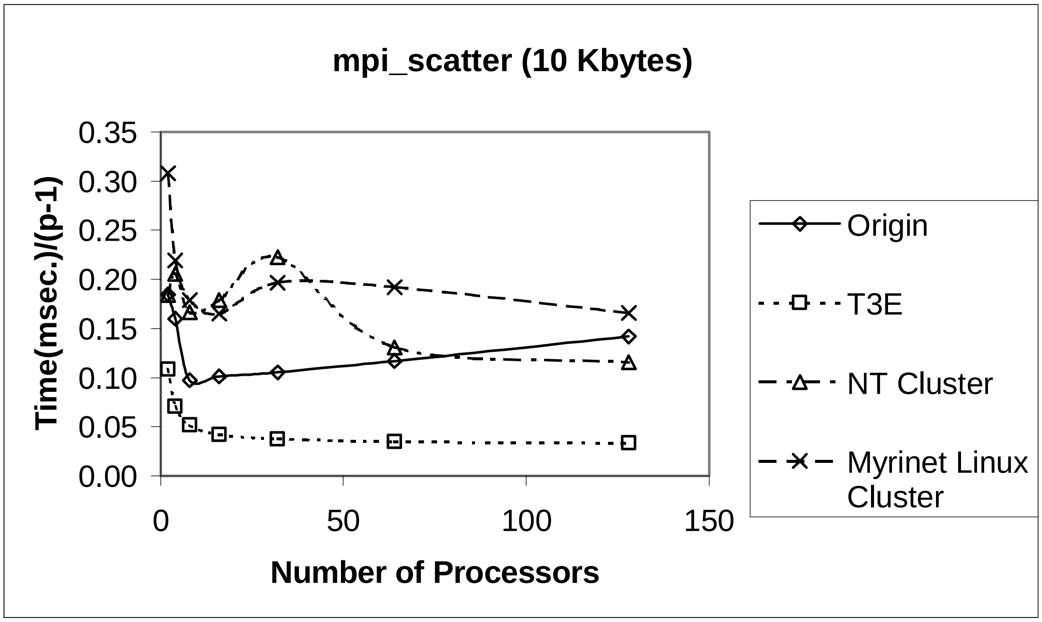


Figure 12: Test 5 (`mpi_scatter`) with times in milliseconds.

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, ..., m_{p-1}, each of size M bytes, 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 send 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, 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)Mβ. 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 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.



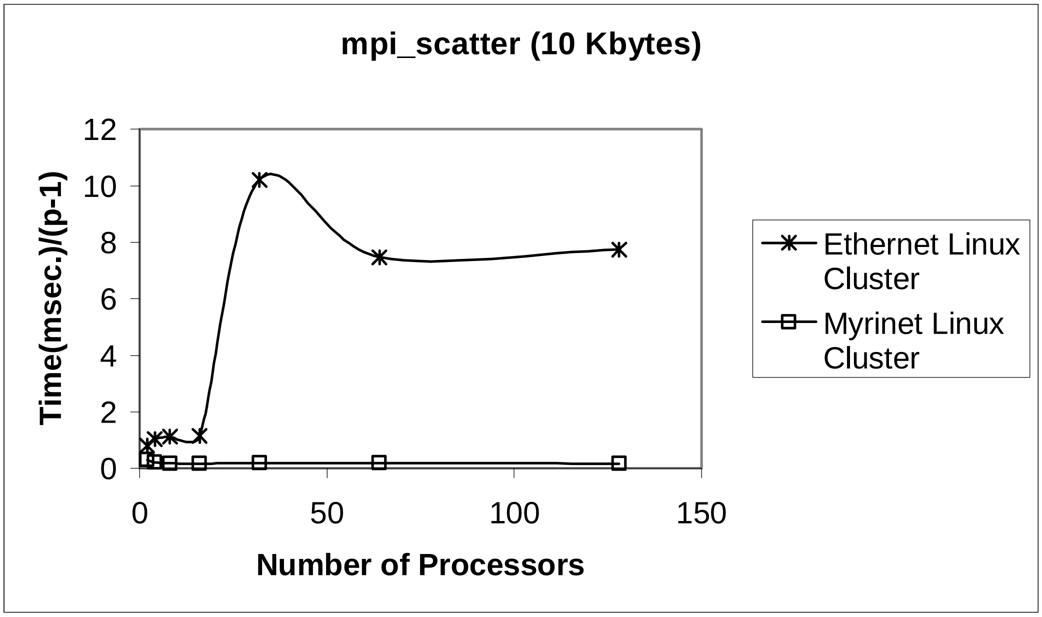


Figure 13: Test 5 (`mpi_scatter`) plotting the (execution time)/(p-1), where p is the number of the processors.

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 bytes message, the performance on all machines is similar with the NT cluster performing best. For 10 Kbytes message, the T3E has the best performance, then the Origin and the NT cluster, finally the Myrinet Linux cluster. Table 4 shows the performance of all machines relative to the T3E for 128 processors.

Message Size	10 Kbytes
Origin/T3E	3.5
NT Cluster/T3E	3
Myrinet Linux Cluster/T3E	5
Ethernet Linux Cluster/T3E	286

Table 4: Time ratios for 128 processors for the mpi_gather test.

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 all machines except the Ethernet Linux cluster for the large message size. Notice that the Ethernet Linux cluster does not perform well compared with the other machines.

Test 7: The All-Gather

This test measures the time to execute:

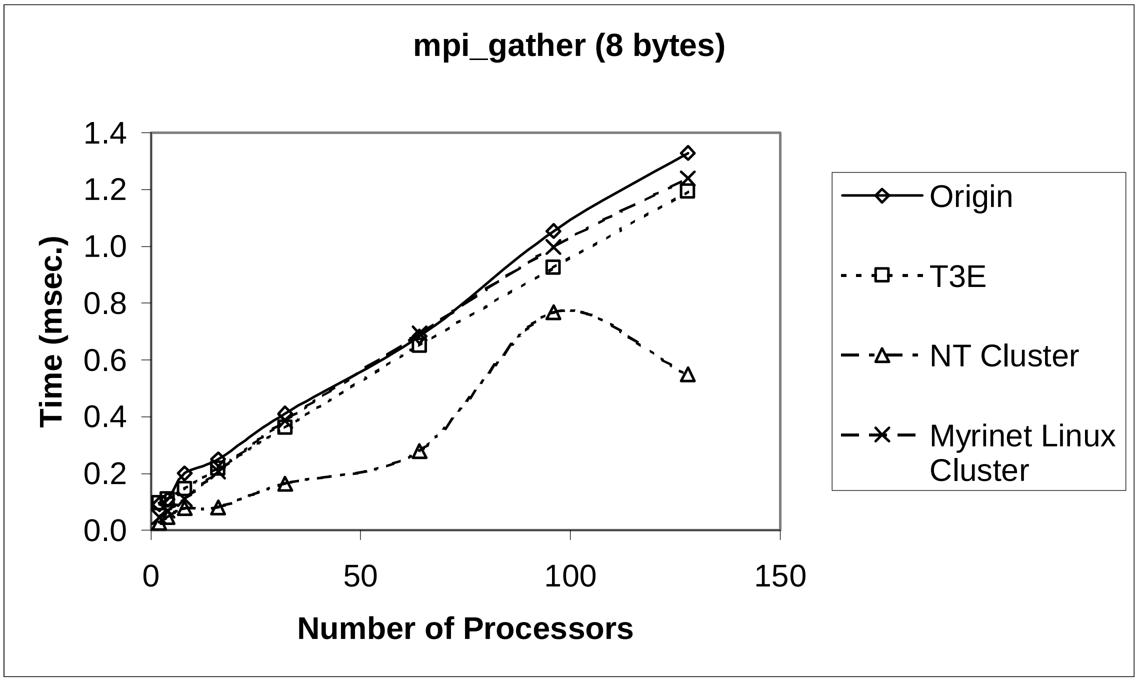
```
  call mpi_allgather(A(1), n, mpi_real8, B(1,0), n, mpi_real8,
     mpi_comm_world, ierror)
```

for n=1 and n=1250.

Figures 17 and 18 present the performance data for this test. For the 8 bytes message, the performance and scalability of the T3E, the Origin and the NT cluster are good compared to that of the Myrinet Linux cluster. For the 10 Kbytes message, the NT cluster performs poorly relative to the T3E, the Origin, and both the Linux clusters. Table 5 shows the performance of all machines relative to the T3E for 128 processors.

Message Size	8 bytes	10 Kbytes
Origin/T3E	1.2	2.8
NT Cluster/T3E	0.7	8.9
Myrinet Linux Cluster/T3E	3.1	1.4
Ethernet Linux Cluster/T3E	353	8

Table 5: Time ratios for 128 processors for the mpi_allgather test.



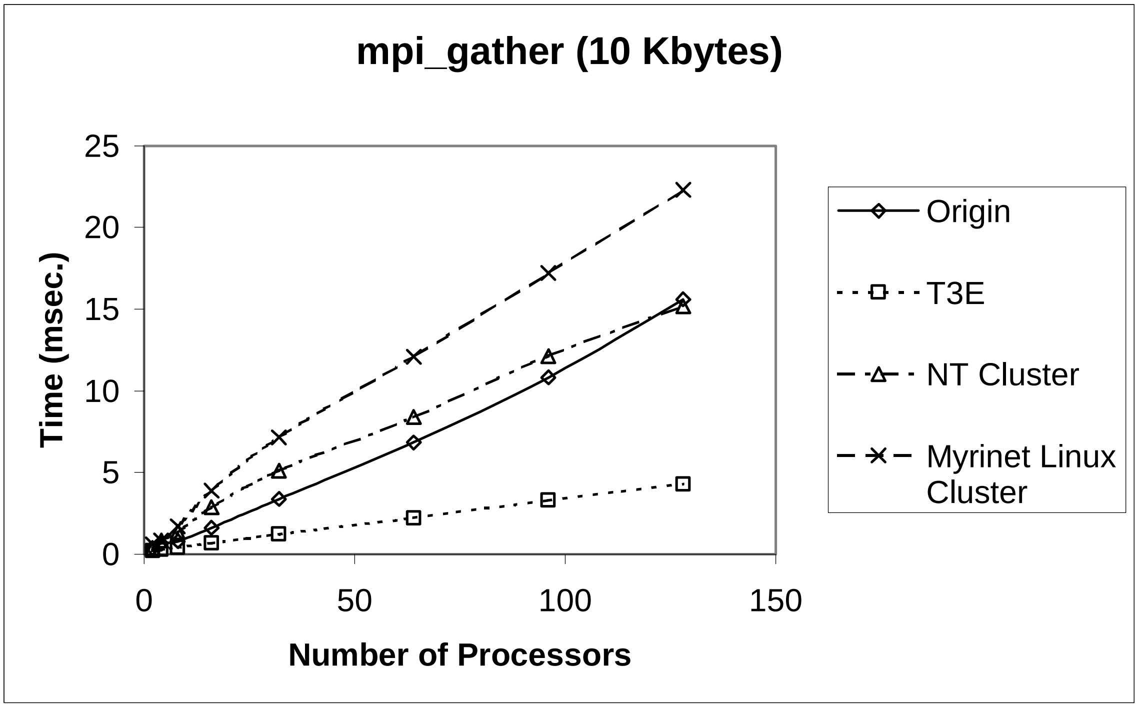
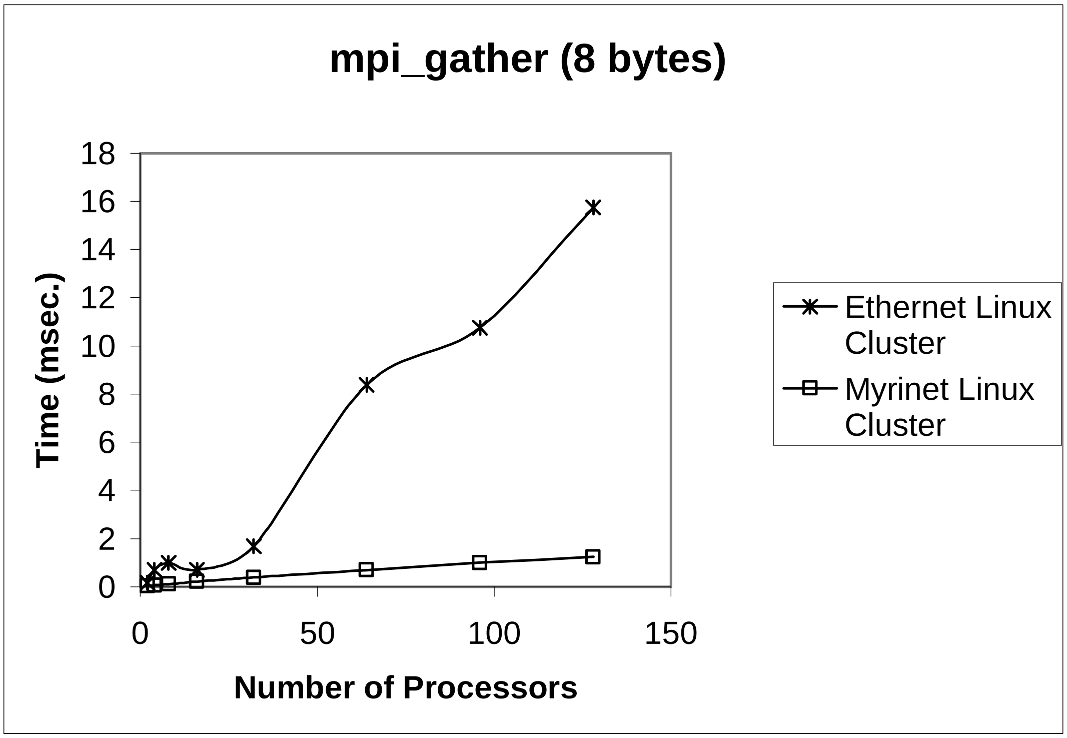


Figure 14: Test 6 (`mpi_gather`) with times in milliseconds.



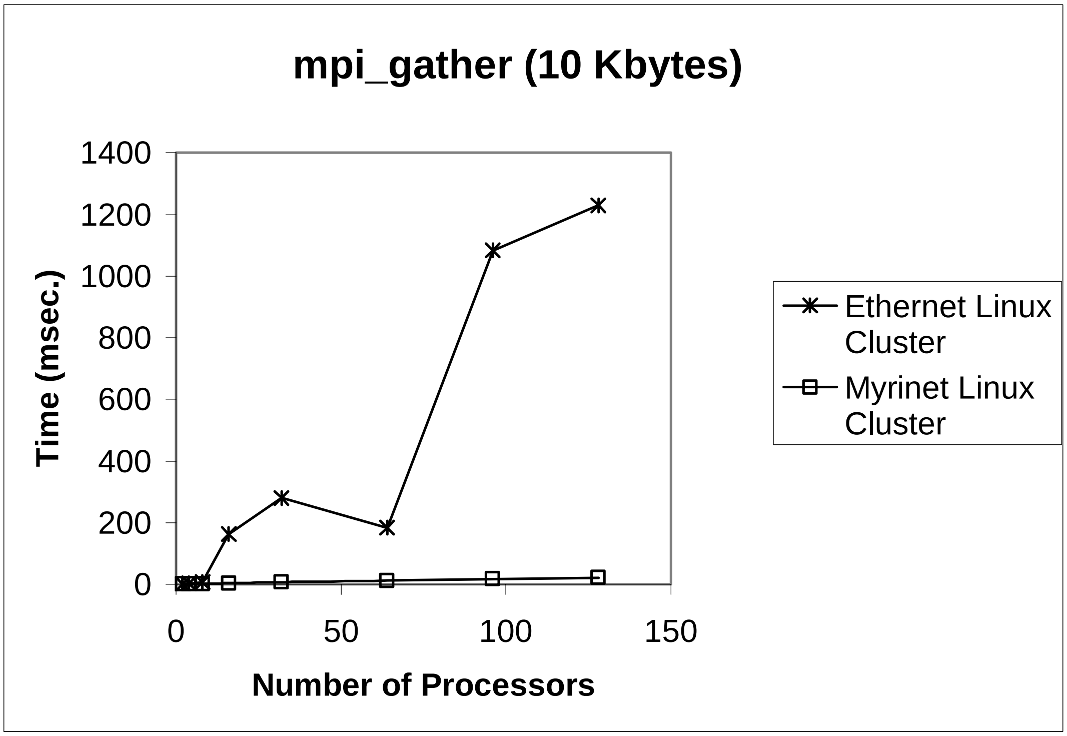
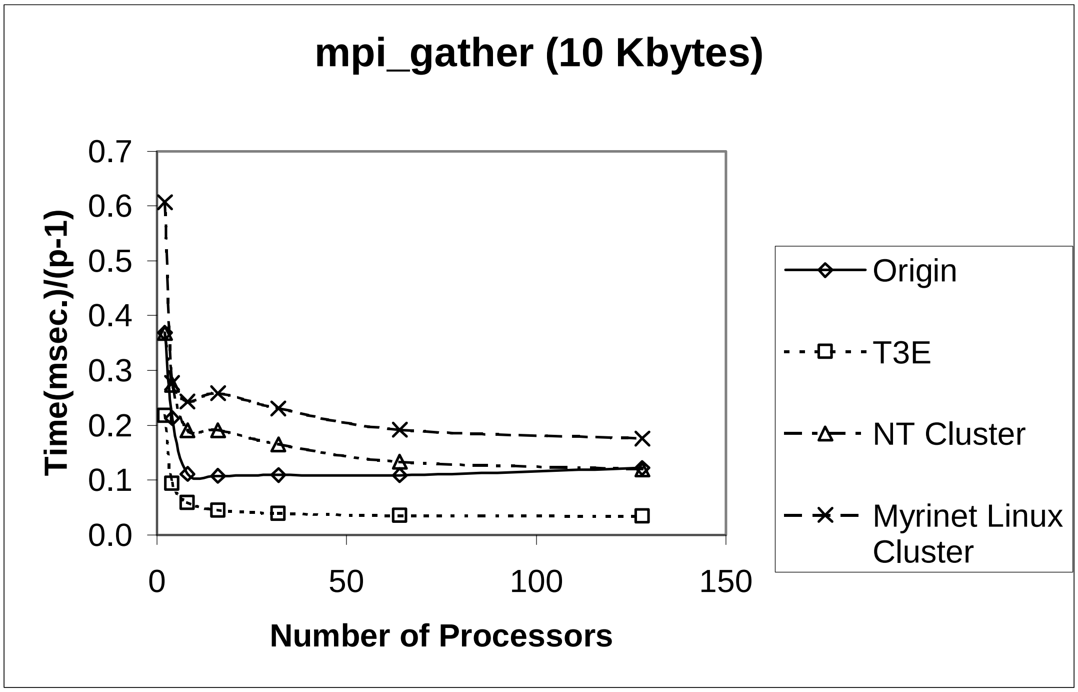


Figure 15: Test 6 (`mpi_gather`) with times in milliseconds.



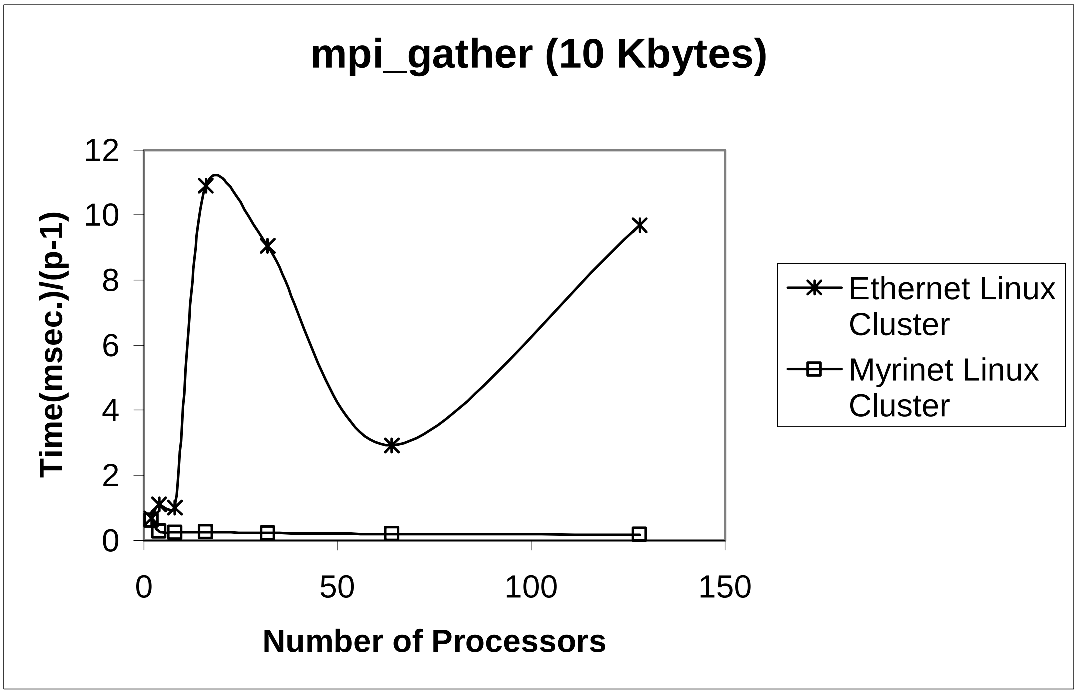
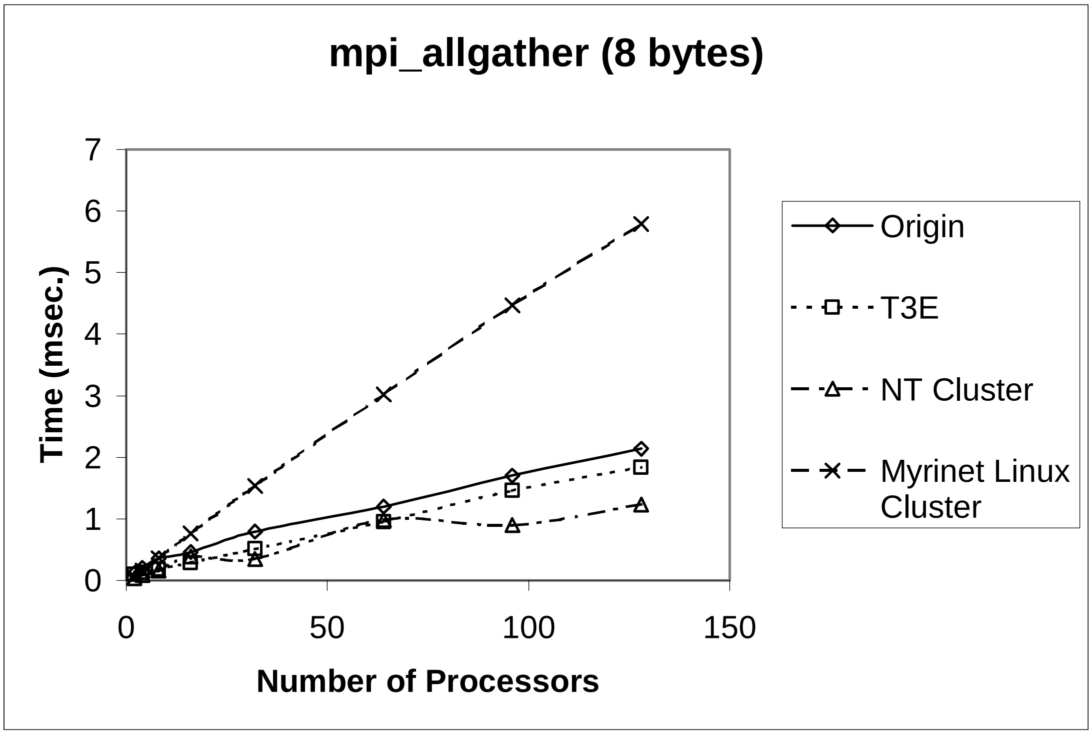


Figure 16: Test 6 (`mpi_gather`) plotting the (execution time)/(p-1), where p is the number of the processors.



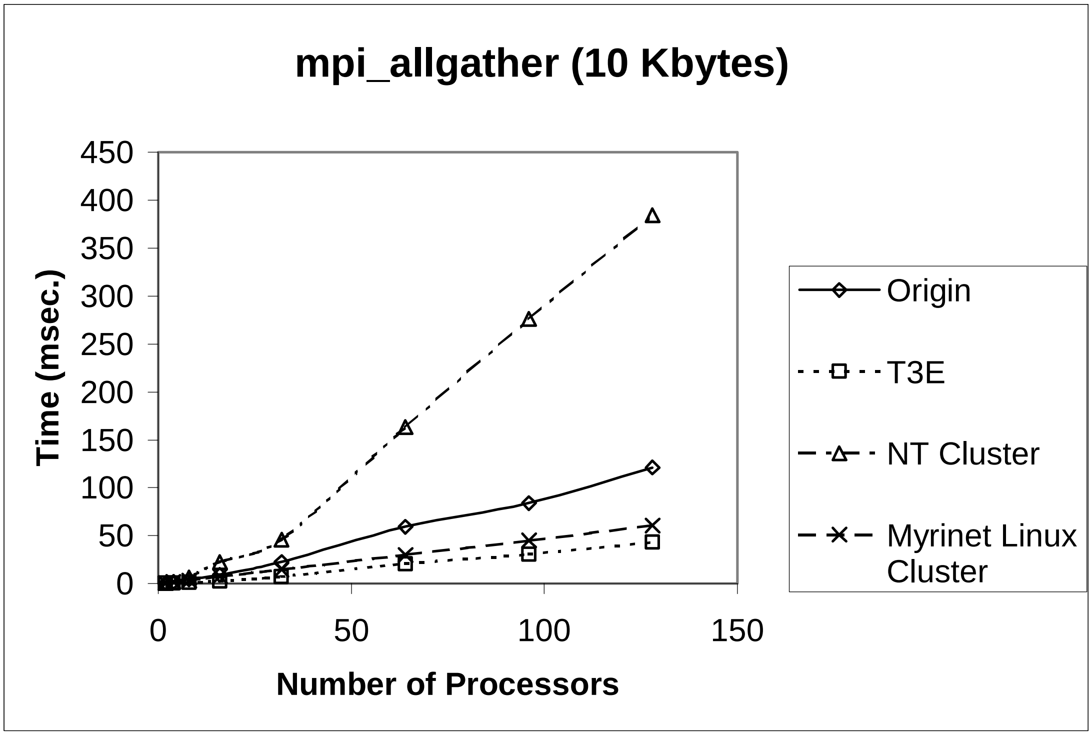
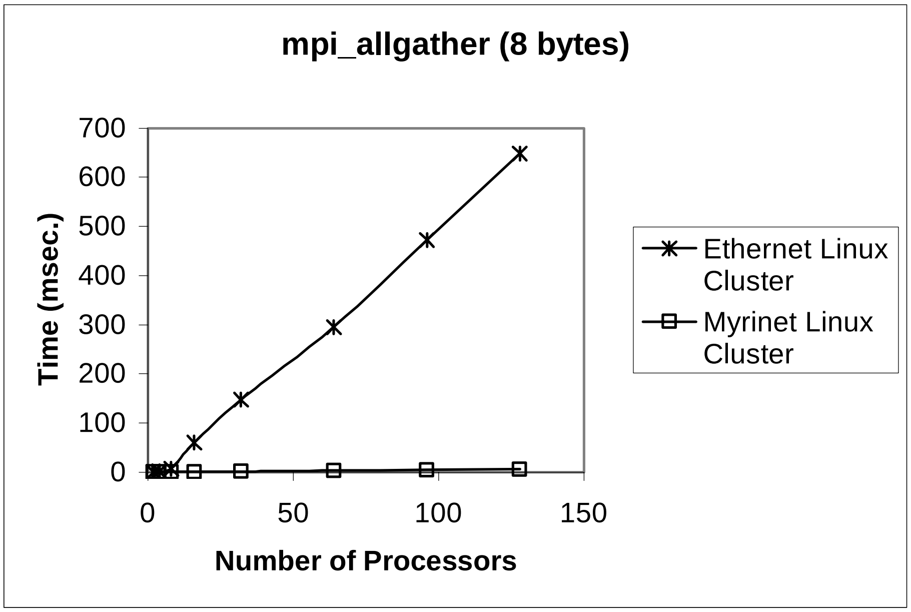


Figure 17: Test 7 (`mpi_allgather`) with times in milliseconds.

Notice that on Ethernet Linux cluster the times for 8 bytes messages are larger than the ones for 10Kb messages when more than 8 processors were used.



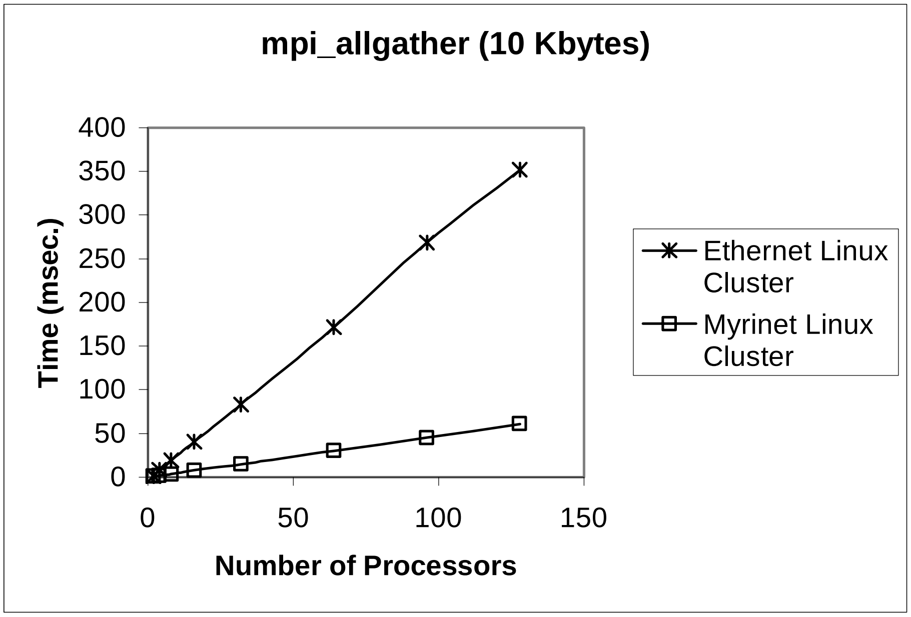


Figure 18: Test 7 (`mpi_allgather`) with times in milliseconds.

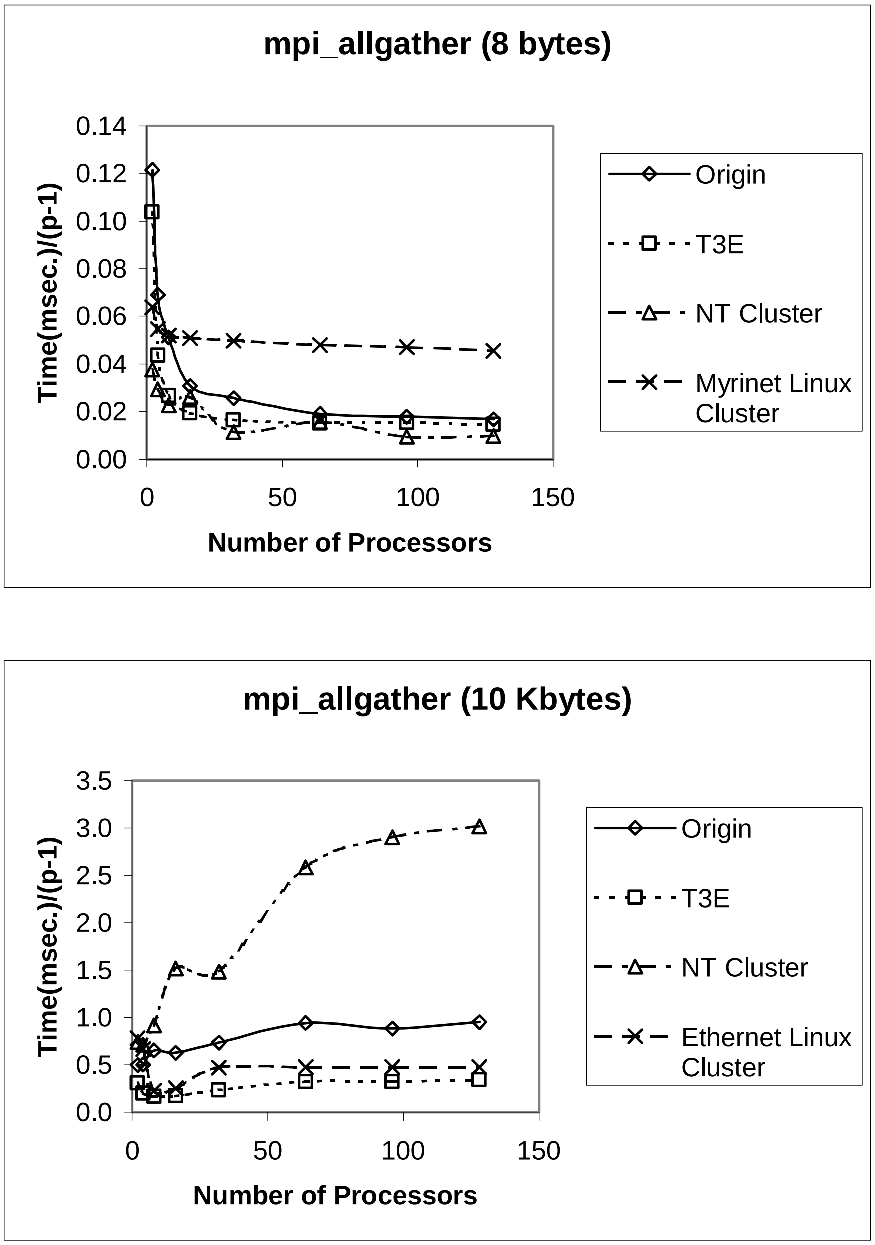
The `mpi_allgather` is sometimes implemented as p-1 right shifts executed by each of the p processors [7], where each processor i initially owns a message m_i of size M bytes. The jth right-shift is defined by: each processor i receives the message $m_{(p+(i-j)) \bmod p}$ from processor (i-1) mod p. 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, (execution time)/(p-1) will be a constant for all such p for a fixed message size. Thus, plotting

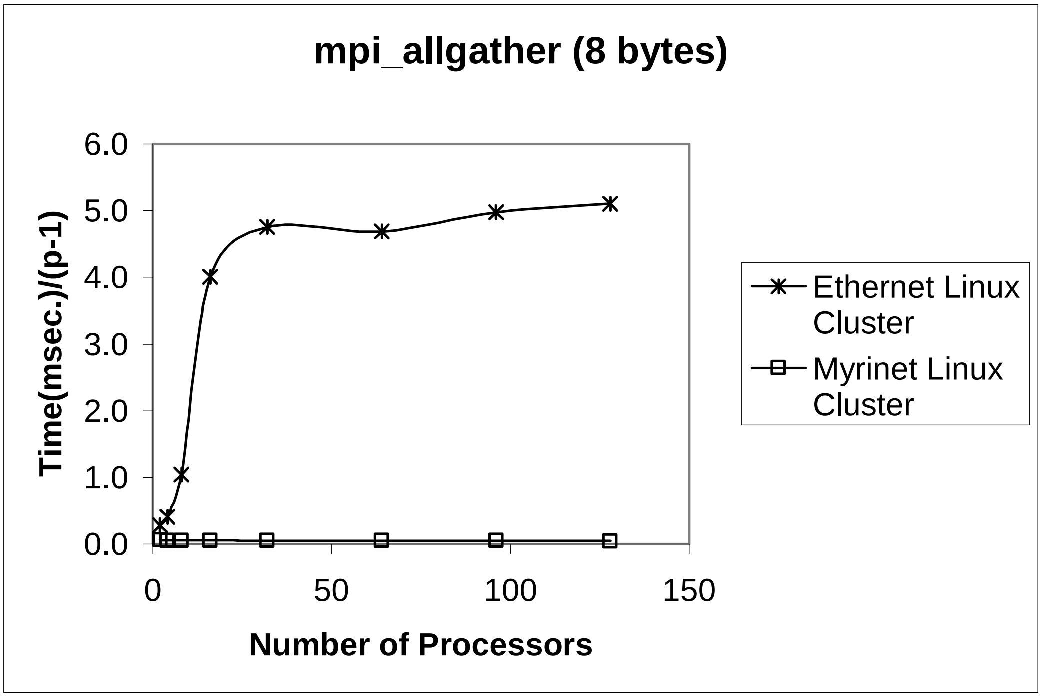
(execution time)/(p-1)

will provide a way to better understand the scalability of `mpi_allgather` for each machine. Figures 19 and 20 show that (execution time)/(p-1) is roughly constant for all the machines except for the NT cluster for large p.


mpi_allgather (8 bytes)

mpi_allgather (10 Kbytes)

Figure 19: Test 7 (`mpi_allgather`) plotting (execution time)/(p-1), where p is the number of the processors.



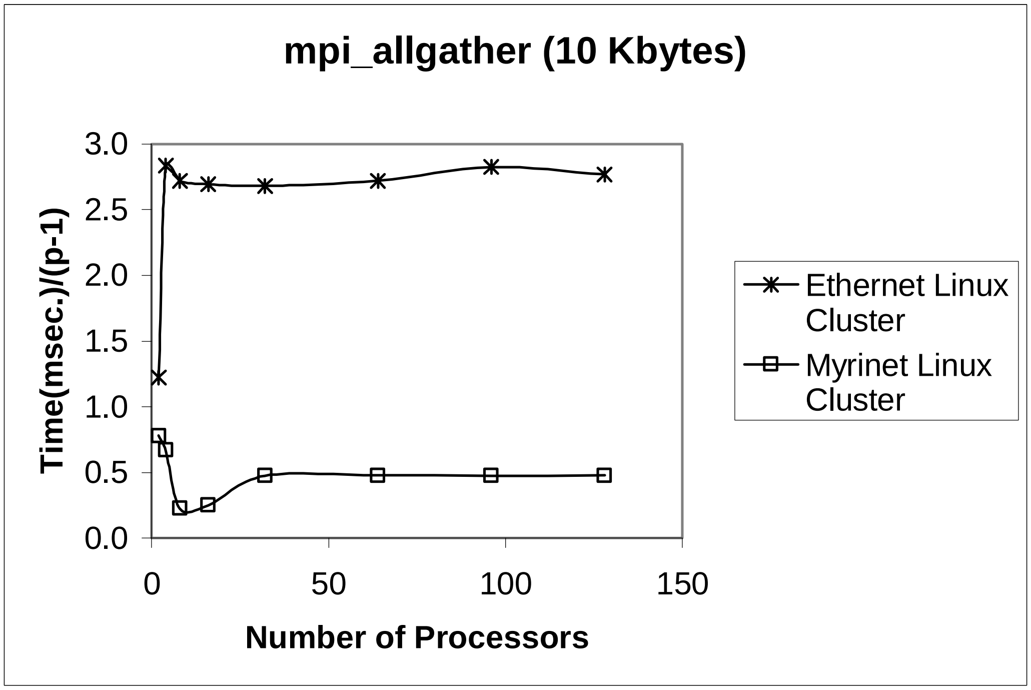


Figure 20: Test 7 (`mpi_allgather`) plotting (execution time)/(p-1), where p is the number of the processors.

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. Notice that for the 8 bytes message, the T3E and NT cluster perform and scale significantly better than the Origin and Myrinet Linux cluster. However, for the 10 Kbyte message, the NT cluster performs and scales poorly. Table 6 shows the performance of all machines relative to the T3E for 128 processors.

Message Size	8 bytes	10 Kbytes
Origin/T3E	0.9	3.3
NT Cluster/T3E	2.7	7.3
Myrinet Linux Cluster/T3E	4.7	3.1
Ethernet Linux Cluster/T3E	117	116

Table 6: Time ratios for 128 processors for the mpi_alltoall test.

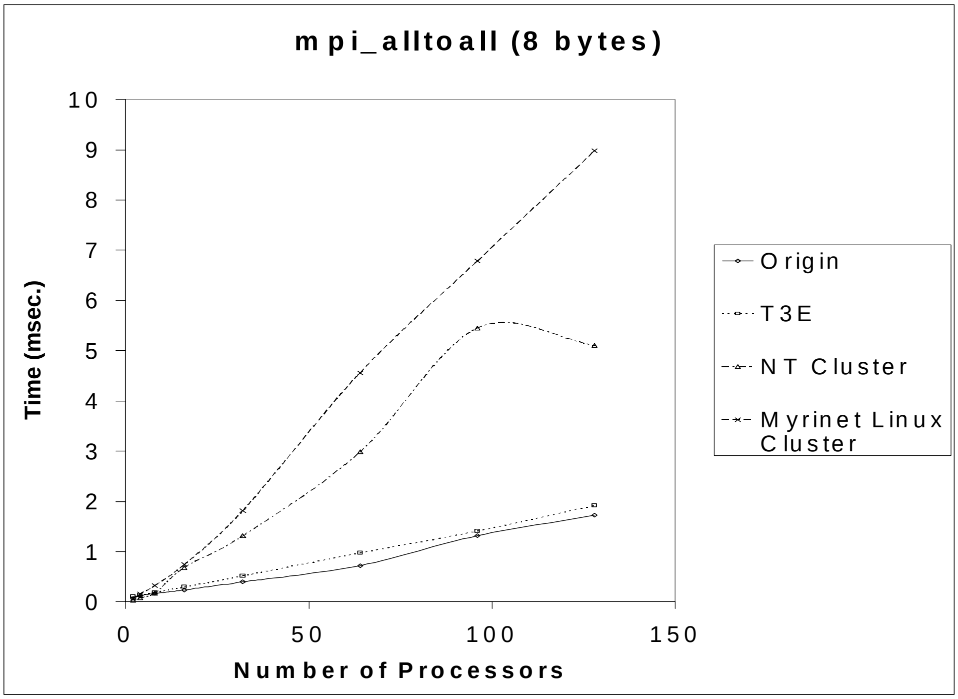
Like the mpi_allgather, the mpi_alltoall is usually implemented as cyclically shifting the messages on the p processors [7]. Initially processor i owns p messages of size M bytes denoted by m_i^0, m_i^1,…,m_i^{p-1}. At the step 0 < j < p, each processor i sends $m_i^{(p+(i-j)) \bmod p}$ to the processor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

$$\text{(execution time)/(p-1)}$$

will provide a way to better understand the scalability of mpi_alltoall for each machine. Figures 23 and 24 show these results. Notice that the (execution time)/(p-1) remains nearly the same with the various number of processors for the T3E and Myrinet Linux cluster. Also notice that the NT cluster scales poorly for 10 Kbyte messages and the Ethernet Linux cluster shows poor performance and scaling for 8 bytes messages.



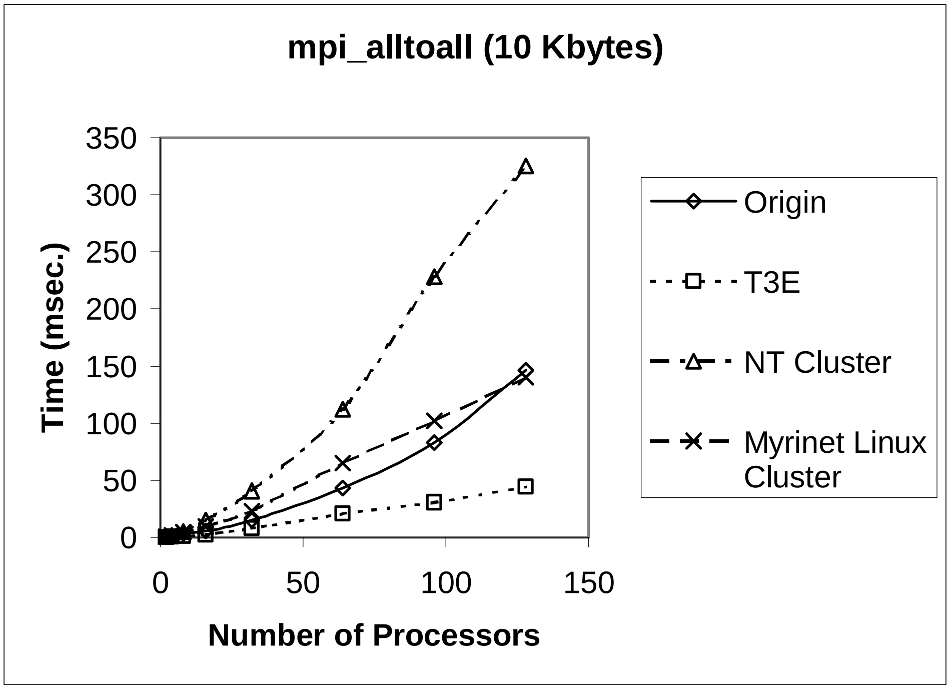
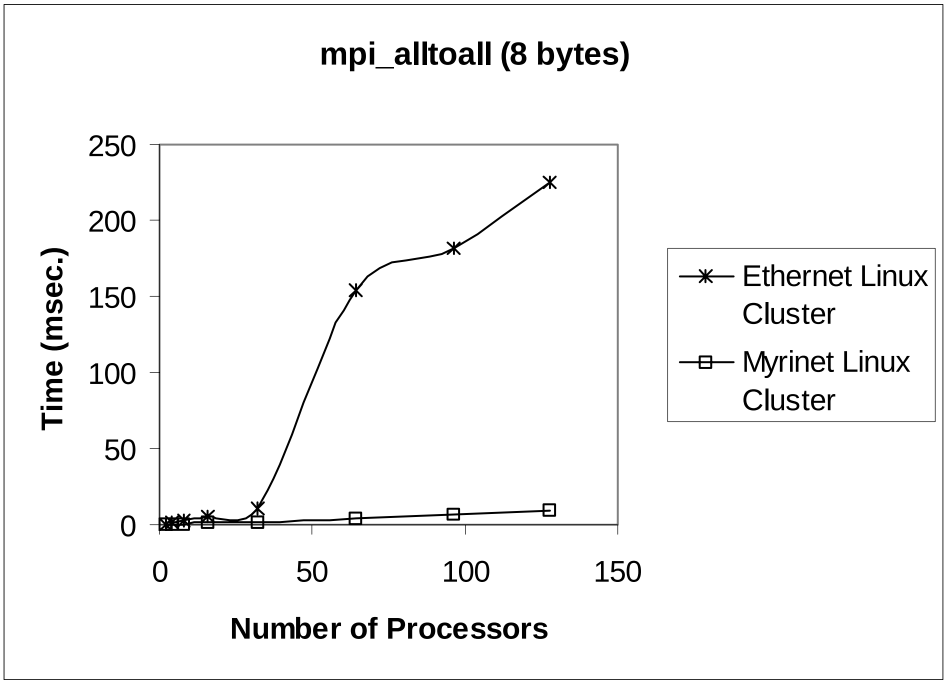


Figure 21: Test 8 (`mpi_alltoall`) with times in milliseconds.



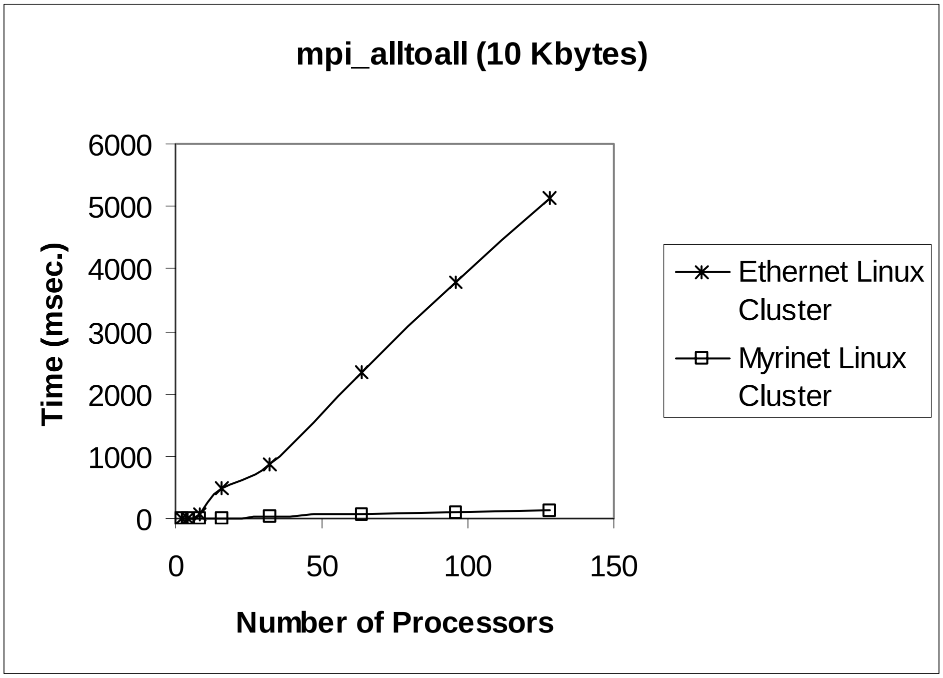


Figure 22: Test 8 (`mpi_alltoall`) with times in milliseconds.

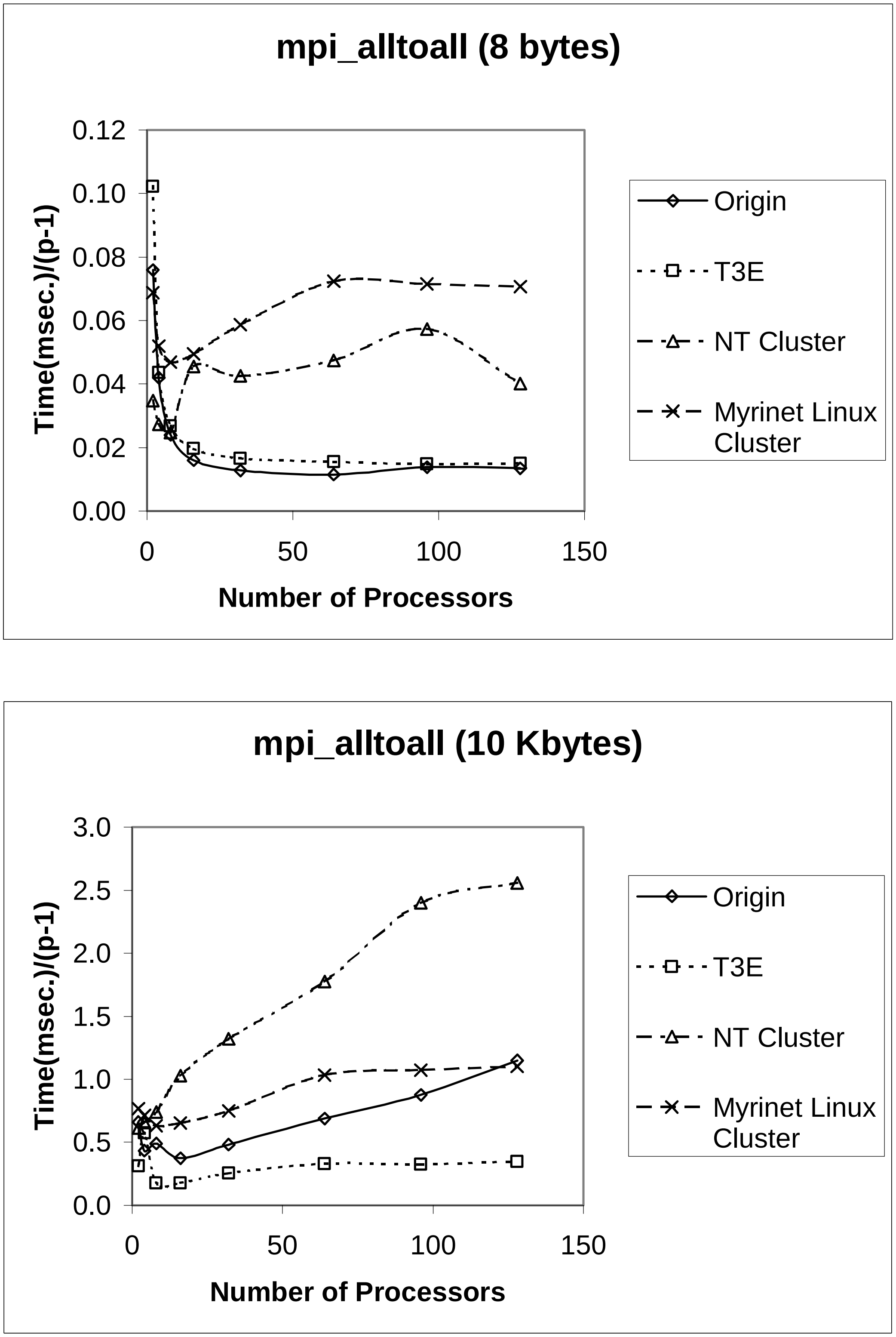
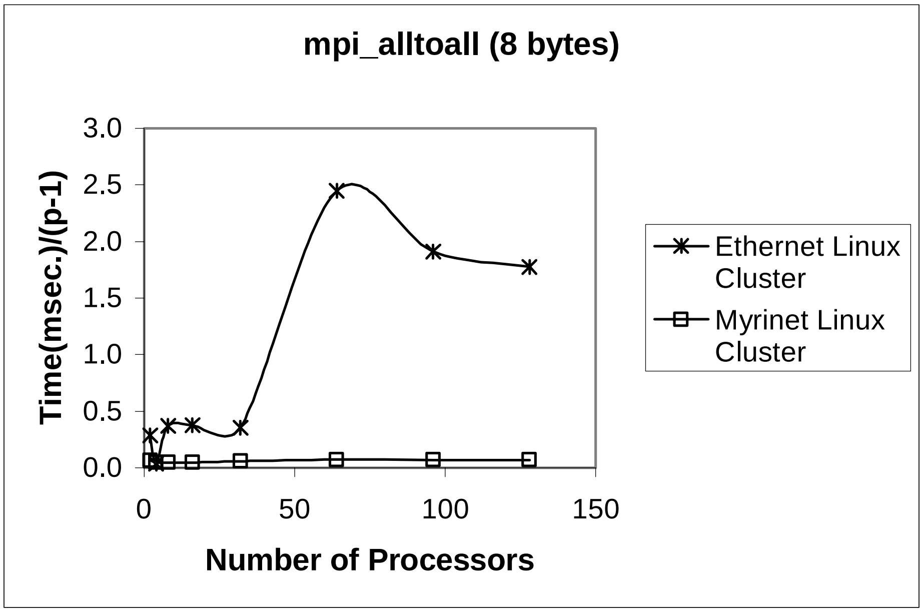


Figure 23: Test 8 (`mpi_alltoall`) plotting (execution time)/(p-1), where p is the number of the processors.



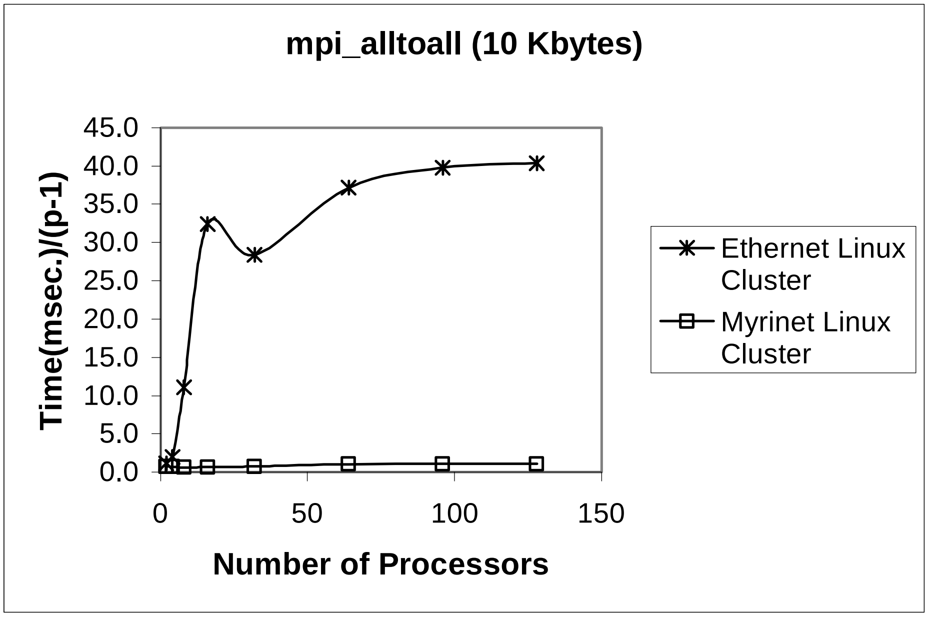


Figure 24: Test 8 (`mpi_alltoall`) plotting (execution time)/(p-1), where p is the number of the processors.

Test 9: The Reduce

This test measures the time to execute:

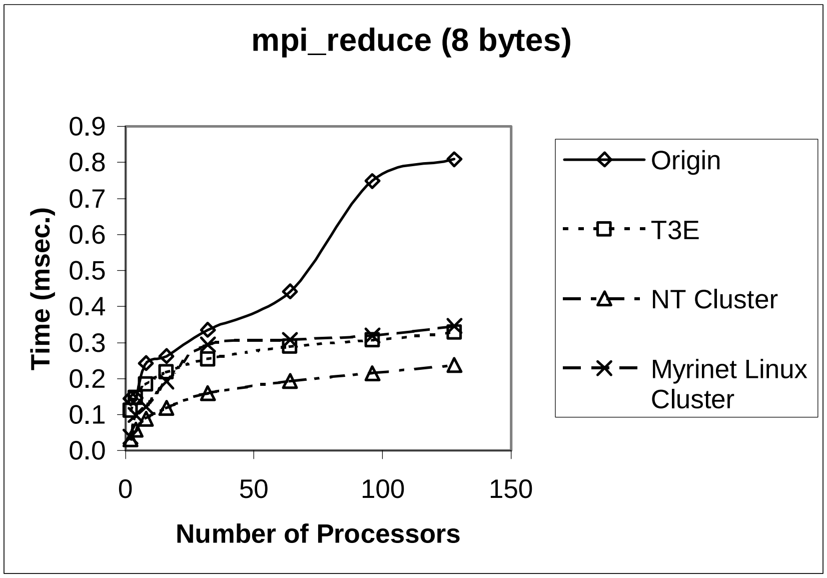
```
    call mpi_reduce(A,C,n,mpi_real8,mpi_sum,0,mpi_comm_world,ierror)
```

for n=1 and n=125000.

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 worst performance for the 8 bytes message and has the best performance for a 1 Mbyte message. Table 7 shows the performance of all machines relative to the T3E for 128 processors.

Message Size	8 bytes	1 Mbyte
Origin/T3E	2.5	0.6
NT Cluster/T3E	0.7	2.4
Myrinet Linux Cluster/T3E	1.1	2.6
Ethernet Linux Cluster/T3E	9.8	11

Table 7: Time ratios for 128 processors for the mpi_reduce test.



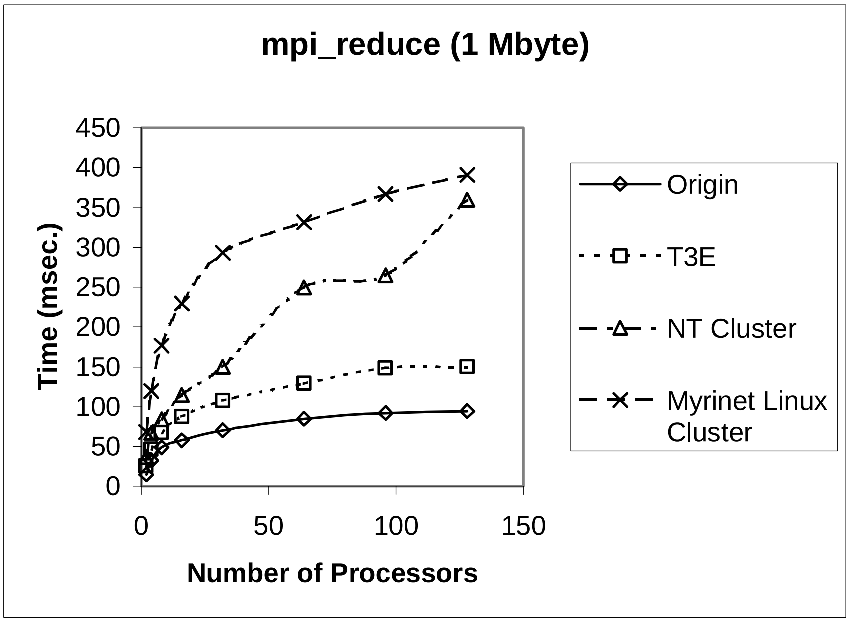


Figure 25: Test 9 (`mpi_reduce` sum) with times in milliseconds.

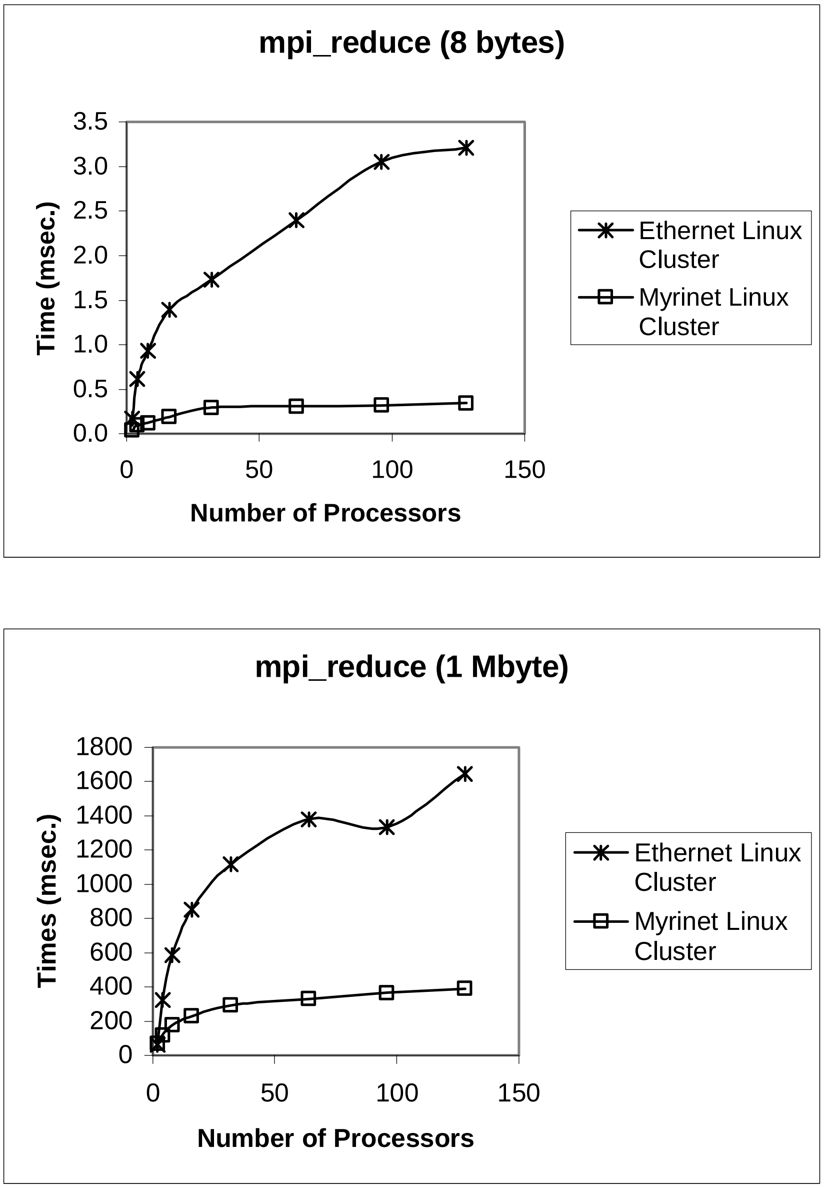


Figure 26: Test 9 (`mpi_reduce` sum) with times in milliseconds.

The discussion of the scalability of this algorithm is beyond the scope of this paper, see [7] for an algorithm for implementing `mpi_reduce`.

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

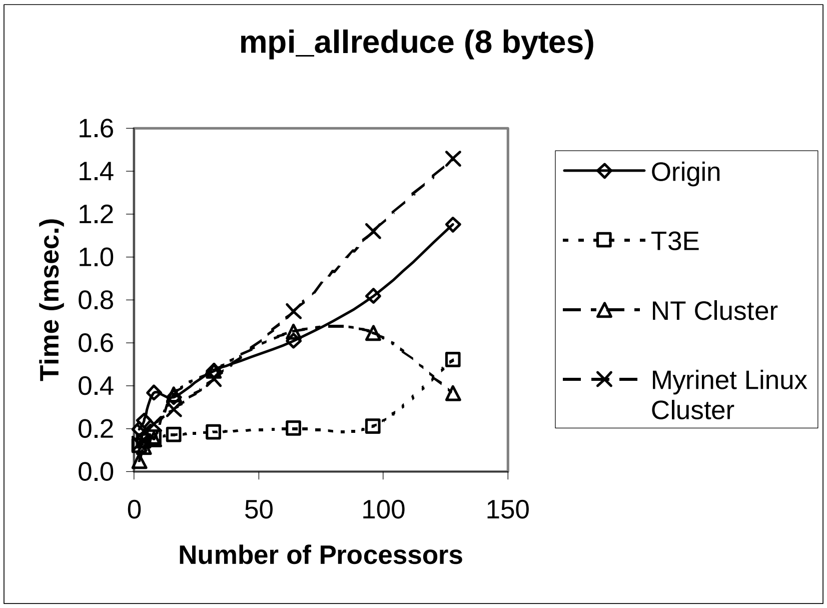
```
    call mpi_allreduce(A,C,n,mpi_real8,mpi_sum,mpi_comm_world,ierror)
```

for n=1 and n=125000.

Figures 27 and 28 present the performance data for `mpi_allreduce` sum operation. The results of the min and max operations are similar. Notice that for the 8 bytes message, the T3E performs best, and for the 1 Mbyte message, both the T3E and NT cluster perform well. Table 8 shows the performance of all machines relative to the T3E for 128 processors.

Message Size	8 bytes	1 Mbyte
Origin/T3E	2.2	0.9
NT Cluster/T3E	0.7	2.5
Myrinet Linux Cluster/T3E	2.8	3
Ethernet Linux Cluster/T3E	9.9	13.8

Table 8: Time ratios for 128 processors for the mpi_allreduce test.



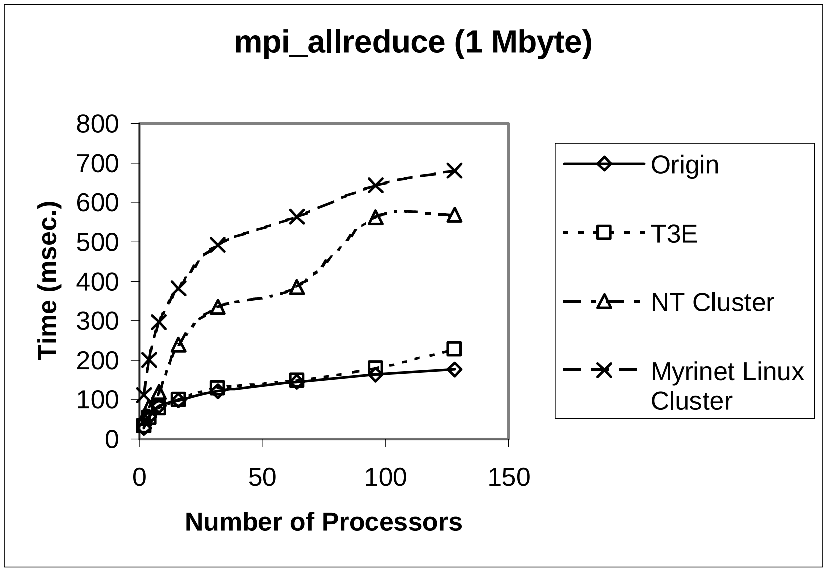


Figure 27: Test 10 (`mpi_allreduce` sum operation) with times in milliseconds.

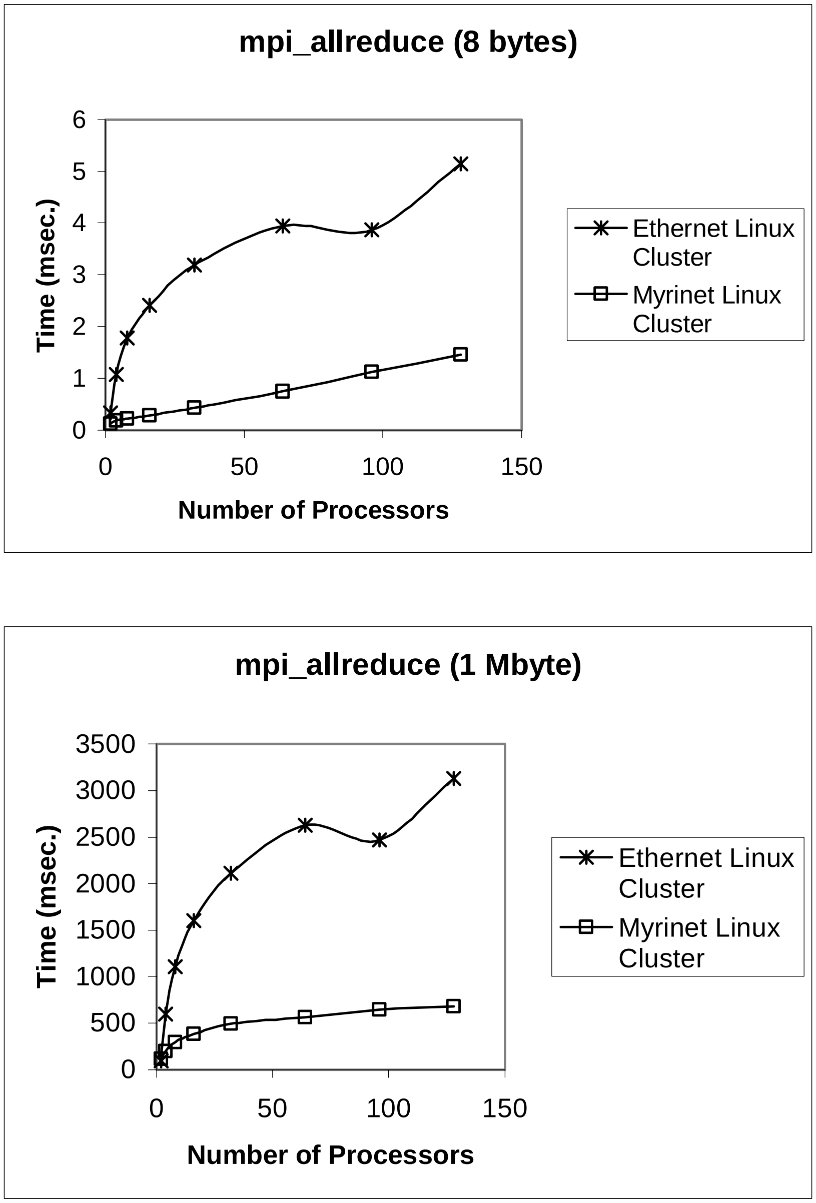


Figure 28: Test 10 (`mpi_allreduce` sum operation) with times in milliseconds.

Test 11: The Scan

This test measures the time to execute

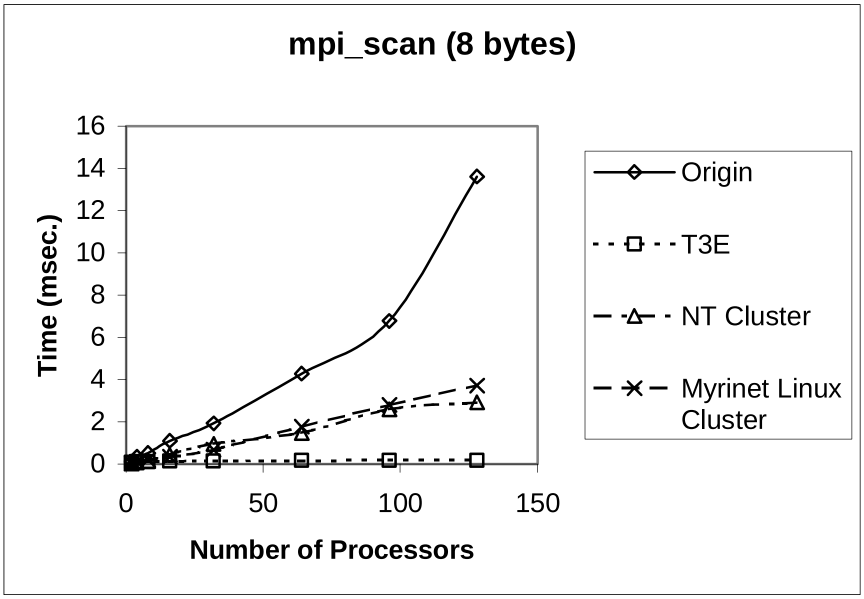
```
call mpi_scan(A, C, n, mpi_real8, mpi_sum, mpi_comm_world, ierror)
```

for n=1 and n=125000. `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,...,i` (inclusive).

Figures 29 and 30 present the performance data. The results for the min and max operations are similar. Notice that the T3E performs and scales significantly better than all the other machines. Table 9 shows the performance of all machines relative to the T3E for 128 processors.

Message Size	8 bytes	1 Mbyte
Origin/T3E	76	9.5
NT Cluster/T3E	16.4	12
Myrinet Linux Cluster/T3E	21	27
Ethernet Linux Cluster/T3E	24	31

Table 9: Time ratios for 128 processors for the mpi_scan test.



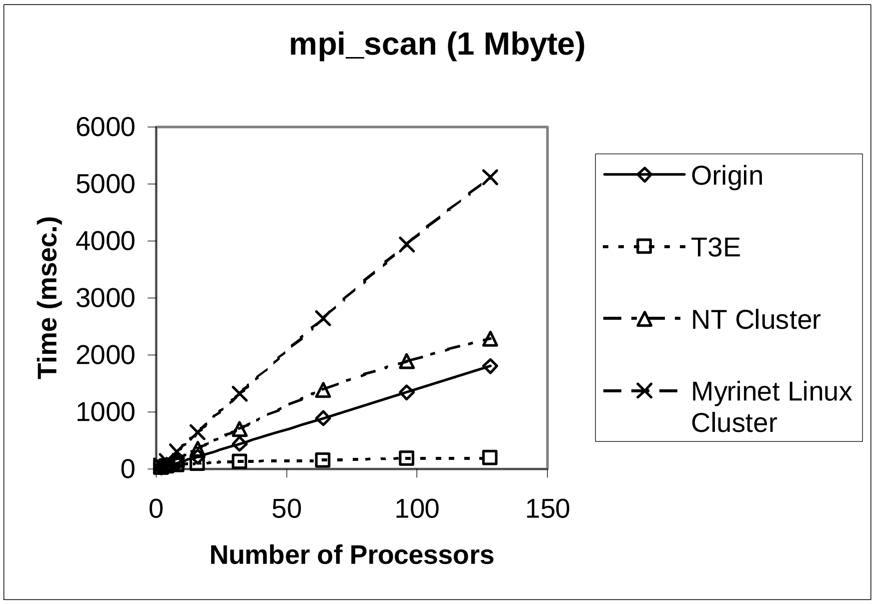


Figure 29: Test 11 (`mpi_scan`) with times in milliseconds.

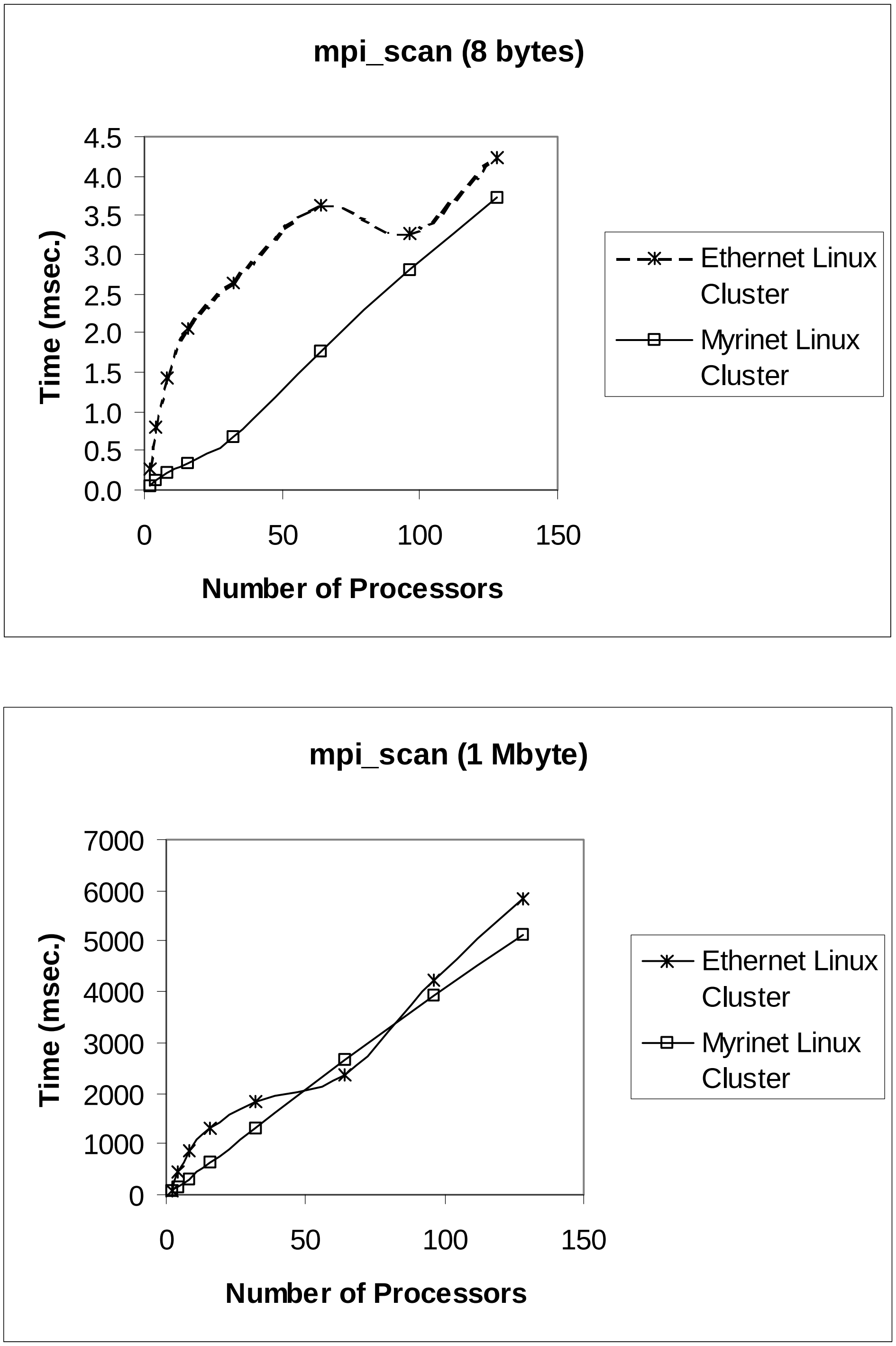


Figure 30: Test 11 (`mpi_scan`) with times in milliseconds.

5 Conclusion

The purpose of this paper is to compare the communication performance and scalability of MPI communication routines on a NT cluster, a Myrinet Linux cluster, an Ethernet Linux cluster, a Cray T3E-600, and a SGI Origin 2000. For most of the MPI tests used in this paper, the T3E-600 and Origin 2000 outperform the NT cluster, the Myrinet and Ethernet Linux clusters. In spite of the fact that the Cray T3E-600 is about 5 years old, it performs best of all machines for most tests. For `mpi_bcast`, `mpi_allgather`, and `mpi_alltoall`, the Myrinet Linux cluster outperforms the NT cluster. For all other MPI collective routines, the NT cluster outperforms the Myrinet Linux cluster. For all MPI collective routines, the Myrinet Linux cluster performs significantly better than the Ethernet Linux cluster with the performance difference increasing as the number of processors increases.

6 References:

1. Cray Research Web Server. http://www.cray.com
2. AHPCC Linux Supercluster. http://www.alliance.unm.edu/
3. NCSA NT Cluster. http://www.ncsa.uiuc.edu/General/CC/ntcluster/
4. Origin Server. Technical report, Silicon Graphics, April 1997.
5. A. Anderson, J. Brooks, C. Grassl, and S. Scott. Performance of the CRAY T3E Multiprocessor. In Proceedings of SC97, 1997.
6. 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.
7. G. R. Luecke, B. Raffin, and J. J. Coyle. The Performance of the MPI Collective Communication Routines for Large Messages on the Cray T3E600, the Cray Origin 2000, and the IBM SP. The Journal of Performance Evaluation and Modelling for Computer Systems, July 1999.
8. Cray Research. Application Programmer Library Reference Manual. Publication SR-2165.
9. H. Dietz and T. Mattox. Inside the KLAT2 Supercomputer: The Flat Neighborhood Network & 3D.
10. 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.

7 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 NT Supercluster.

We would also like to thank the University of New Mexico for access to their Albuquerque High Performance Computing Center. This work utilized the UNM-Alliance Roadrunner Supercluster.