(HPSC **5576** ELIZABETH JESSUP)

HIGH PERFORMANCE SCIENTIFIC COMPUTING

:: Homework / 4

:: Students / C. Preis *&&* F. Rappl

1 problem / **25** points

Problem 1

Task:

Programming assignment 11.11.01 from Pacheco's PPMPI textbook: T_s, T_c measurement (p. 258).

Write a short program that measures T_s and T_c -- the latency and (inverse) throughput -- of a MPI interconnect.

Solution:

The program itself was pretty straightforward and does not require much description. First of all we print out the processor name (which can help us a lot as Trestles and Frost) and then determine on which process we currently are. One of the processes (in this case we use 0) is the master process, the other one is the slave process (in our case 1).

Since we are timing the round-trip time (RTT) we have to be careful in our calculations later on. We thought that (since our experience shows that this is never wrong) a warm-up-run would be nice as well as a large amount of iterations (we took 4096) since the measured time for 1 iteration will be very short. In the end we will just divide the total time by $2 \cdot 4096$ (factor 2 due to RTT) and obtain results for the different tasks.

a. Large message latency and bandwidth

Task:

Run the program to examine messages between 1 byte and 1 MB in size, increasing the message size by powers of two, for the following five configurations (we did actually all **7**, i.e. included Blacklight).

Solution:

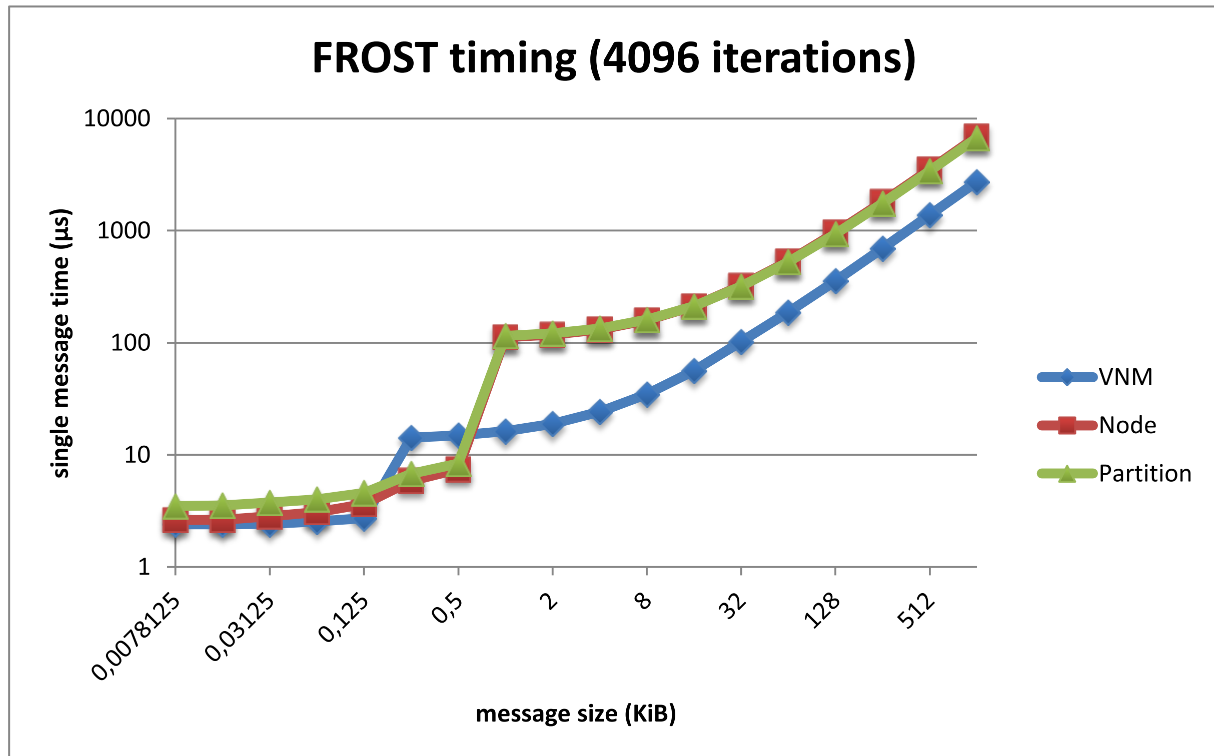
i. NCAR Frost | within a **node**
ii. NCAR Frost | within a **partition**
iii. NCAR Frost | **across partitions**

To determine T_s it was assumed that for very small messages the message round trip time is dominated by the latency, since $T_s \gg T_C$. So T_s was calculated dividing the time of 4096 round trips for a very small message (8 byte) divided by $(4096 \cdot 2)$.

Similarly to determine T_C it was assumed that for very big messages the message round trip time is dominated by the throughput. So T_C was calculated dividing the time of 4096 round trips for a very big message (1 MiB) divided by $(4096 \cdot 2)$.

We were able to see that the virtual node mode (VNM) of Frost has the best speed (lowest latency, best bandwidth) overall. For a certain region (above 128 bytes and below 512 bytes) we see that the partition and node (which are pretty equal in the long run, but the node has a better latency) are even better than the VNM.

Summary	within a node	within a partition	across partitions
Latency T_s (µs)	2,393310547	2,605712891	3,486694336
α (cycles)	**1675,317383**	**1823,999023**	**2440,686035**
Throughput (MiB/s)	365,935971	144,9020703	149,2079357
T_c (µs/byte)	0,002606123	0,006581509	0,006391579
β (cycles/byte)	**1,824286417**	**4,607056477**	**4,474105337**



iv. SDSC Trestles | within **a node**

v. SDSC Trestles | between two **different nodes**

The connection to Trestles was no problem since it was as easy to access as Frost over the TeraGrid portal. What makes Trestles a little bit harder to use is the required batch file or command line parser script language. Since the SDSC has a lot of documentations online for Trestles it was possible to create a batch script to be used for this purpose. For the measurements within a node we used

```
#!/bin/bash
#PBS -q normal
#PBS -A TG-SEE110002
#PBS -l nodes=1:ppn=2
#PBS -l walltime=00:10:00
#PBS -o single_node_job.output
#PBS -N PINGPONG_MPI
#PBS -V
cat $PBS_NODEFILE
cd $PBS_O_WORKDIR #change to the working directory
mpirun_rsh -np 2 -hostfile $PBS_NODEFILE ./pingpong
```

which did run well from the first time we used it.

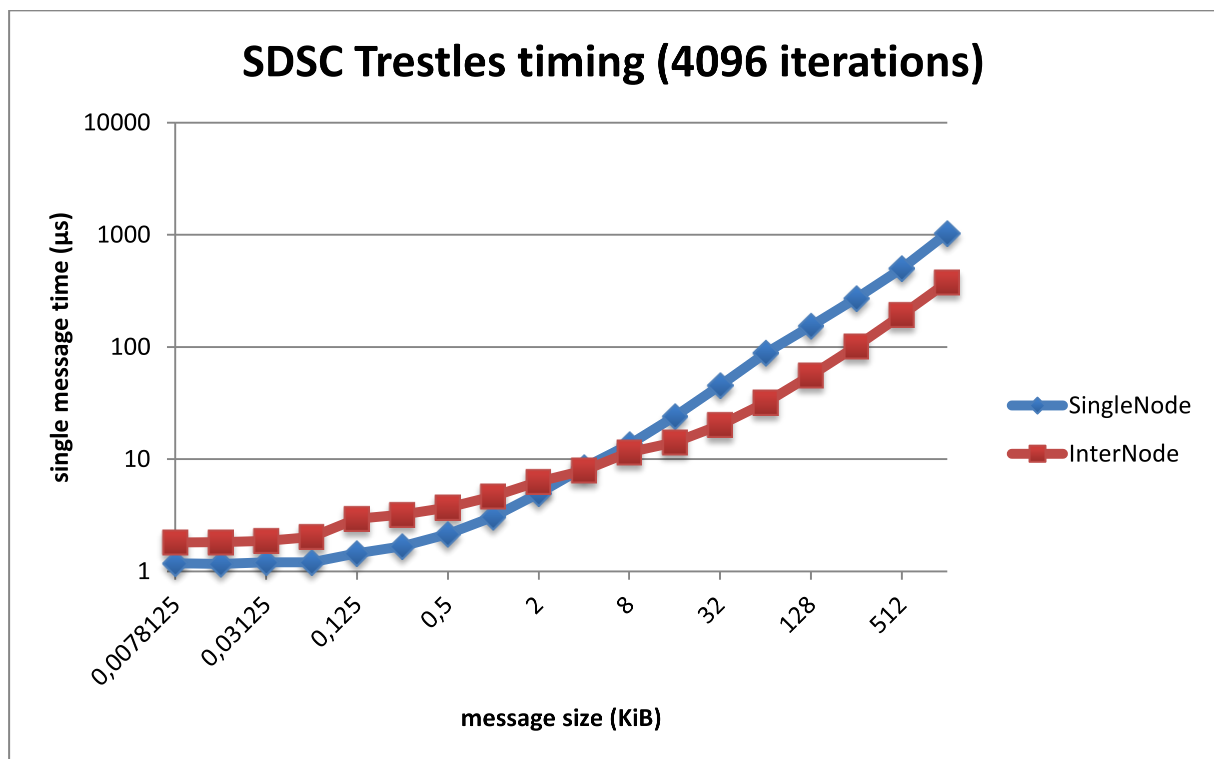
The other one is nearly the same except another node distribution (no *2 processors on one* node but *2 nodes with 1 processor each*):

```
#PBS -l nodes=2:ppn=1
#PBS -o inter_node_job.output
```

The timing results were quite remarkable. We were able to see that the InfiniBand network that the SDSC Trestles uses is in fact a lot faster than the one Frost uses. Furthermore we could see that the connection between two nodes is faster than the connection of the processors in a node.

Summary	within a node	different nodes
Latency T_s (µs)	1,177612305	1,80480957
α (cycles)	**2826,269531**	**4331,542969**
Throughput (MiB/s)	969,3681097	2642,596842
T_c (µs/byte)	0,00098381	0,000360885
β (cycles/byte)	**2,361144684**	**0,866124686**

For small messages however (really small ones) the connection between two processors at the same node wins due to a much shorter connection. The bandwidth of the Trestles system is approx. 9 times faster than the bandwidth of the VNM of the Frost system.



vi. PSC SGI Blacklight | within **a blade**

vii. PSC SGI Blacklight | between two **different blades**

Connecting to the SGI Blacklight system at Pittsburgh was also more than easy. The Java module allows us to connect to other systems even though they are not on the login page. By just entering

the right address of the computer (blacklight.psc.teragrid.org) we were able to connect over the TeraGrid portal. Blacklight uses a script language that is pretty similar to Trestles. Our batch file was:

```
#!/bin/csh
#PBS -l ncpus=16
#PBS -l walltime=10:00
#PBS -j oe
cd $HOME/hpsc/pingpong/
setenv MPI_DSM_CPULIST 1,2 #allocates the specified CPUs
setenv MPI_DSM_VERBOSE 1   #activates verbose mode
mpirun -np 2 ./pingpong
```

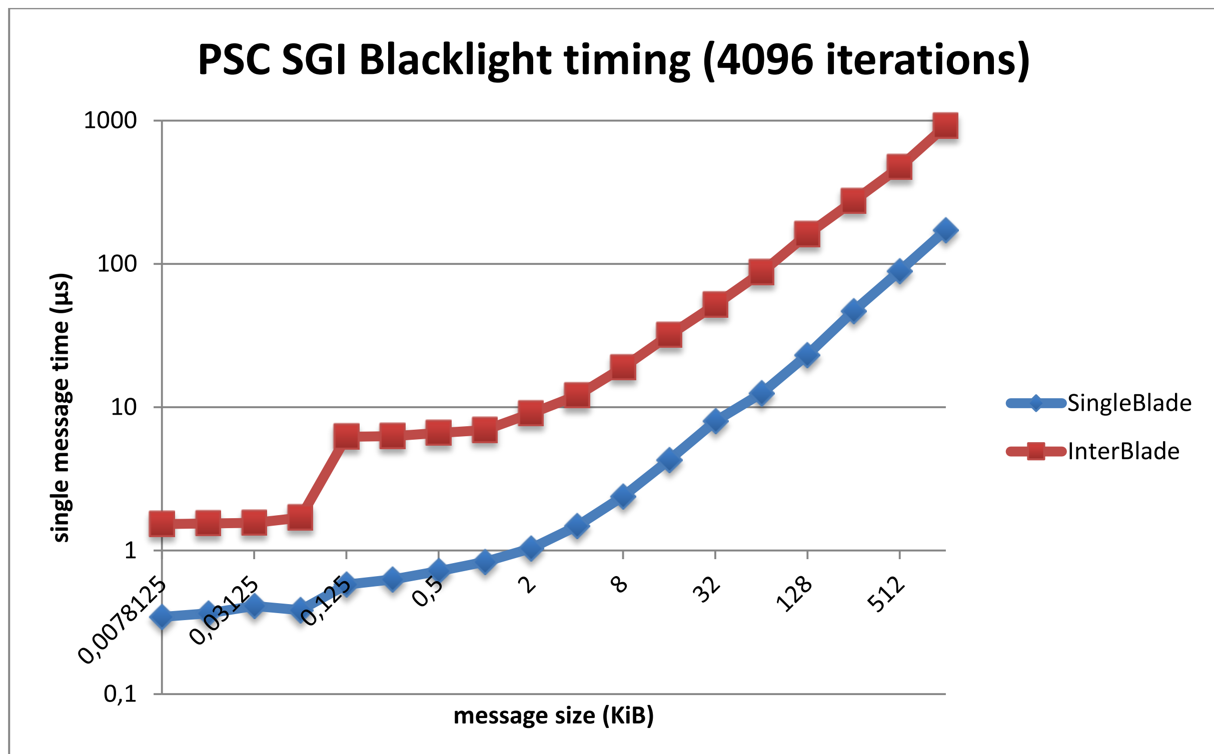
For Blacklight we have to be very cautious with our CPU allocation. We can only use a multiple of 16 (which is the amount of CPUs in a blade). Over the environment variable we can determine which of the 16 we allocate with our 2 MPI processes. This line was more interesting in the next one (for inter-blade connection), where we had to allocate 32 CPUs (to get 2 blades). We wrote there

```
setenv MPI_DSM_CPULIST 1,17 #allocates the specified CPUs
```

to get the first processor on the first blade and the second processor on the second blade.

Summary	within a blade	different blades
Latency T_s (µs)	0,345825195	1,530029297
α (cycles)	**785,0231934**	**3473,166504**
Throughput (MiB/s)	5826,109962	1087,889724
T_c (µs/byte)	0,00016369	0,000876628
β (cycles/byte)	**0,371575668**	**1,989944984**

This is actually the fastest of all networks with a bandwidth that is reaching 6 GiB/s.

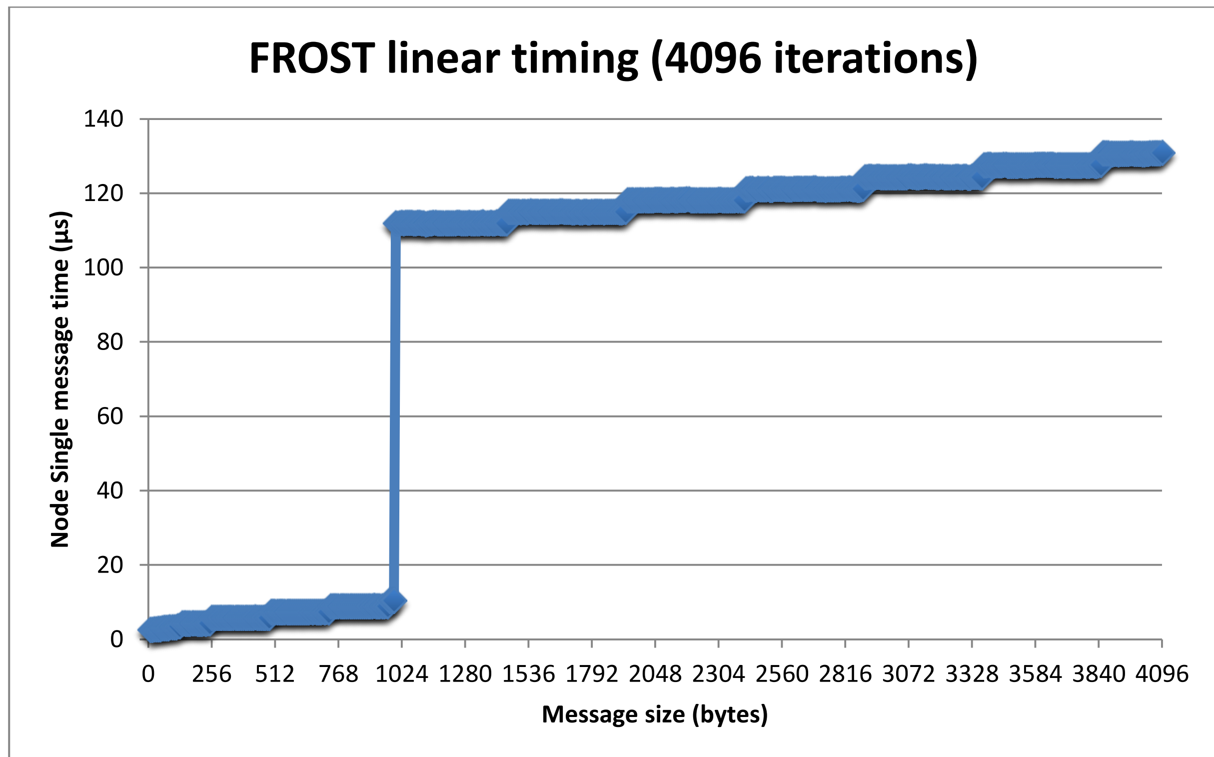


b. Small message latency and bandwidth

Task:

Using Frost, run the program between two nodes for messages between 1 byte and 4 KiB in size, increasing the message size linearly. Plot this data and determine if (and if so, at what data size) the MPI implementation switches delivery protocols.

Solution:



Obviously MPI switches the delivery protocol at 1000 bytes (1 KB or 1 KiB - 24 bytes). Besides that we see kind of steps in the plot which is an indicator that MPI delivers not an arbitrary number of bytes but a fixed amount of bytes, i.e. for a message bigger than 1000 bytes and smaller than 1456 bytes the amount of time is nearly the same. It then seems that the higher size is selected for about 480 bytes more, i.e. the next step goes till 1936, and then we have 2416, 2896 etc.

Summary:

	Frost 1 Node	1 Partition	2 Partitions	Trestles 1 Node	2 Nodes	Blacklight 1 Blade	2 Blades
α (cycles)	1675	1823	**2440**	2826	4331	**785**	3473
β (cycles/byte)	1,8	4,6	4,5	2,4	**0,9**	**0,4**	2,0

Output printout:

This section displays some output printouts of these three systems. A more detailed output can be viewed in the *.tar.zip file which is uploaded in the Moodle.

a. Frost

The following shows a sample output for an inner node Ping-Pong:

```
Process 1 is running on Processor <0,0,1,0> in a <4, 4, 2, 1> mesh
Process 0 is running on Processor <0,0,0,0> in a <4, 4, 2, 1> mesh
Iterations: 4096, Size: 1, Time: 0.021346
Iterations: 4096, Size: 2, Time: 0.021475
Iterations: 4096, Size: 4, Time: 0.023161
Iterations: 4096, Size: 8, Time: 0.025204
Iterations: 4096, Size: 16, Time: 0.029795
Iterations: 4096, Size: 32, Time: 0.047949
Iterations: 4096, Size: 64, Time: 0.060829
Iterations: 4096, Size: 128, Time: 0.917428
Iterations: 4096, Size: 256, Time: 0.969544
Iterations: 4096, Size: 512, Time: 1.074398
Iterations: 4096, Size: 1024, Time: 1.302112
Iterations: 4096, Size: 2048, Time: 1.739202
Iterations: 4096, Size: 4096, Time: 2.634494
Iterations: 4096, Size: 8192, Time: 4.359450
Iterations: 4096, Size: 16384, Time: 7.848610
Iterations: 4096, Size: 32768, Time: 14.805037
Iterations: 4096, Size: 65536, Time: 28.722646
Iterations: 4096, Size: 131072, Time: 56.534734
...
```

In order to shorten the output the final vector (to have an easy possibility for copying the data to *Excel / QtiPlot / MatLab* etc.) has been replaced by "…".

For a VNM we have a little change in the beginning of the output:

```
Process 1 is running on Processor <0,0,0,1> in a <4, 4, 2, 2> mesh
Process 0 is running on Processor <0,0,0,0> in a <4, 4, 2, 2> mesh
```

The beginning of the output for the across partitions task looked like this:

```
Process 1 is running on Processor <7,3,1,0> in a <8, 4, 2, 1> mesh
Process 0 is running on Processor <0,0,0,0> in a <8, 4, 2, 1> mesh
```

b. Trestles

Trestles looked pretty much the same except the first lines:

```
trestles-1-28
trestles-1-28
Process 0 is running on trestles-1-28.local
Process 1 is running on trestles-1-28.local
```

This shows a connection within a node. The first two lines come from the batch script, where we print out the $PBS_NODEFILE variable, which shows us what nodes we got allocated. The output between two nodes looked like this:

```
trestles-1-31
trestles-1-32
Process 1 is running on trestles-1-32.sdsc.edu
Process 0 is running on trestles-1-31.local
```

c. Blacklight

To get a good output at Blacklight we had to use the verbose environment variable as we already did in the batch script shown in the evaluation of the Blacklight data. The only lines that changed in the output were the lines after the *mpirun* command. Those lines print out the processor names on which our MPI program is actually running. For the single blade connection we received the following data from the machine:

```
MPI: DSM information
grank lrank pinning  node name                cpuid
    0     0 yes      r005i23b03#38_01-1218      305
    1     1 yes      r005i23b03#38_02-1220      306
Process 0 is running on bl1.psc.teragrid.org
Process 1 is running on bl1.psc.teragrid.org
```

We can actually see that the output from our program (C file) is not helpful at all, since it only prints out on which upper network (bl1) we currently are. We do not get any information about the blade used. This will be more obvious in a connection between two blades:

```
MPI: DSM information
grank lrank pinning  node name                cpuid
    0     0 yes      r007i23b11#182_01-5826    1457
    1     1 yes      r007i23b12#184_01-5890    1473
Process 0 is running on bl1.psc.teragrid.org
Process 1 is running on bl1.psc.teragrid.org
```

Here we see that the big difference is in the node name (b11, b12), where we had before (b03,b03). This way gives us the CPU-ID as well!

Code printout:

```
1  #define SIZE_ITERATIONS 18 /* 2^20 = 8 MB for doubles */
2  #define COMM_ITERATIONS 4096
3  #include <stdio.h>
4  #include <string.h>
5  #include "mpi.h"
6  
7  void Plot_Vec(int, int*, double*, int*);
8  
9  void main(int argc, char* argv[])
10 {
11       int          my_rank;        /* rank of process      */
12       int          p;              /* number of processes  */
13       int          source;         /* rank of sender       */
14       int          dest;           /* rank of receiver     */
15       int          tag = 0;        /* tag for messages     */
16       char         my_name[64];    /* name of machine      */
17       int          my_name_len;    /* length of my_name    */
18       MPI_Status   status;         /* return status 4 recv */
19       int          n = 1;          /* message size         */
20       double       start, end;     /* time measurements    */
21       double       my_time;        /* full time temporary  */
22       double       times[24];      /* time statistic vec   */
23       int          iterations[24];/* number of iterations */
24       int          mess_size[24]; /* message sizes (mb...)*/
25       int          i , j;          /* other loop counters  */
```

```
26          /* Setup (empty) arrays */
27          for (i = 0; i < 24; i++)
28          {
29                  times[i] = 0.0;
30                  niterations[i] = 0;
31                  mess_size[i] = 0;
32          }
33
34      /* Start up MPI */
35      MPI_Init(&argc, &argv);
36
37      /* Find out process rank  */
38      MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
39
40      /* Find out number of processes */
41      MPI_Comm_size(MPI_COMM_WORLD, &p);
42
43          /* print processor name for processes */
44      if (my_rank < 2)
45          {
46                  MPI_Get_processor_name( my_name, &my_name_len );
47                  printf("Process %i is running on %s\n", my_rank, my_name);
48          }
49
50          for (j = 0; j < SIZE_ITERATIONS; j++)
51          {
52                  /* define and fill message */
53                  double            message[n];
54                  for (i = 0; i < n; i++)
55                          message[i] = i * my_rank + 1.0;
56
57                  /* wait for all processors */
58                  MPI_Barrier(MPI_COMM_WORLD);
59
60                  /* determine if master or slave process */
61                  if (my_rank == 0)
62                  {
63                          /* send to slave process */
64                          dest = 1;
65                          source = 1;
66
67                          /* warm up iteration */
68                          MPI_Send(message, n, MPI_DOUBLE, dest, tag,
69                                          MPI_COMM_WORLD);
70                          MPI_Recv(message, n, MPI_DOUBLE, source, tag,
71                                          MPI_COMM_WORLD, &status);
72
73                          /* Take starttime */
74                          start = MPI_Wtime();
75                          /* Timed iterations */
76                          for (i = 0; i < COMM_ITERATIONS; i++)
77                          {
78                                  MPI_Send(message, n, MPI_DOUBLE, dest, tag,
79                                                  MPI_COMM_WORLD);
80                                  MPI_Recv(message, n, MPI_DOUBLE, source, tag,
81                                                  MPI_COMM_WORLD, &status);
82                          }
83                          /* Take endtime */
84                          end = MPI_Wtime();
85                          /* delta time = endtime - starttime */
86                          my_time = end - start;
```

```
87                      /* populate statistic vector */
88                      times[j] = my_time;
89                      niterations[j] = comm_iterations;
90                      mess_size[j] = n;
91
92                      /* Print iterations summary */
93                      printf("Iterations:\t%d\tSize:\t%d\tTime:\t%f\n",
94                              comm_iterations, n, my_time);
95              }
96              else if (my_rank == 1)
97              {
98                      /* send to master process */
99                      dest = 0;
100                     source = 0;
101
102                     /* warm up iteration */
103                     MPI_Recv(message, n, MPI_DOUBLE, source, tag,
104                             MPI_COMM_WORLD, &status);
105                     MPI_Send(message, n, MPI_DOUBLE, dest, tag,
106                             MPI_COMM_WORLD);
107
108                     /* Real iterations */
109                     for (i = 0; i < COMM_ITERATIONS; i++)
110                     {
111                             MPI_Recv(message, n, MPI_DOUBLE, source, tag,
112                                     MPI_COMM_WORLD, &status);
113                             MPI_Send(message, n, MPI_DOUBLE, dest, tag,
114                                     MPI_COMM_WORLD);
115                     }
116             }
117
118             n *= 2;
119     }
120
121     /* Print Summary */
122     if (my_rank == 0)
123     {
124             printf("\nNumber of Processes:\t%d\n", p);
125             printf("Sizes\t\tTimes\tIterations:n");
126
127             /* Plot the values in a good form */
128             Plot_Vec(24, mess_size, times, niterations);
129     }
130
131    /* Shut down MPI */
132    MPI_Finalize();
133 } /* main */
134
135 void Plot_Vec(int veclen, int* mess_size, double *A, int* B)
136 {
137     int k;
138     for (k = 0; k < veclen; k++)
139             printf ("%d\t\t%f\t%d\n", mess_size[k], A[k], B[k]);
140 } /* Plot_Vec */
```