( HPSC **5576** ELIZABETH JESSUP )

# HIGH PERFORMANCE SCIENTIFIC COMPUTING

## :: Homework / 3


## :: Student / Florian Rappl


**1** problem / **30** points

# Problem 1

**Task**:

- Write your own implementation of *MPI_Allreduce* using the log P butterfly allreduce algorithm.
- Compare the performance of your implementation with the built-in *MPI_Allreduce* function.

**Solution:**

My implementation of the *MPI_Allreduce* function is a function that is prototyped the following way:

```
void my_allreduce(double* sndvalue, double* recvalue, int count, unsigned
int rank, int processors, int tag, MPI_Comm comm, int type, char verbose);
```

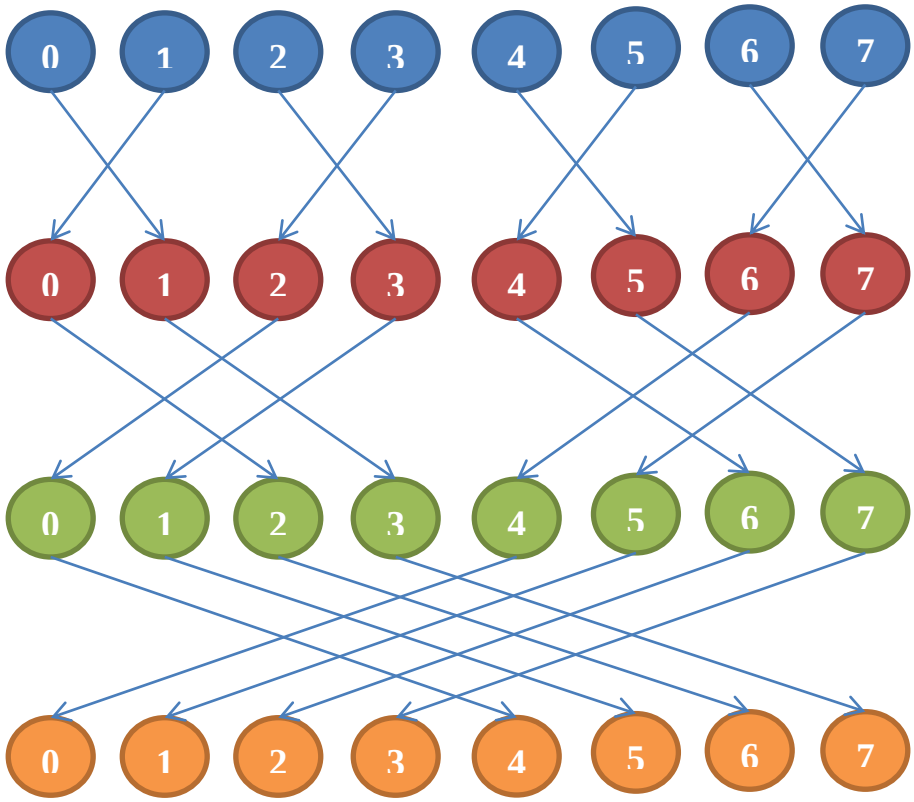
As the original MPI function I have a parameter for the send vector which won't be changed and a parameter to the receive vector. The count specifies the length of the data vector. A difference is that I give my function the current rank and the processor-count – whereas the MPI function does that probably under the hood (I guess those two values are stored in the running MPI instance and do not have to be re-calculated again – therefore this won't change any time measurements). While the tag and the MPI_Comm Object are two variables one also has to pass to *MPI_Allreduce* , the *my_allreduce* has a type and a verbose argument for Low to High (1) or High to Low (-1) bit traversal and showing the communication output. The details of the implementation are described in the code and later on.

**Communication Diagrams:**

[L→H] This output shows the communication for low to high given by the program with the verbose option set to true. I set the processor count to 8. To produce this output I used the –u option (def.).

```
Communication from 0 to 1
Communication from 1 to 0
Communication from 2 to 3
Communication from 3 to 2
Communication from 4 to 5
Communication from 5 to 4
Communication from 6 to 7
Communication from 7 to 6
Communication from 0 to 2
Communication from 1 to 3
Communication from 2 to 0
Communication from 3 to 1
Communication from 4 to 6
Communication from 5 to 7
Communication from 6 to 4
Communication from 7 to 5
Communication from 0 to 4
Communication from 1 to 5
Communication from 2 to 6
Communication from 3 to 7
Communication from 4 to 0
Communication from 5 to 1
Communication from 6 to 2
Communication from 7 to 3
```

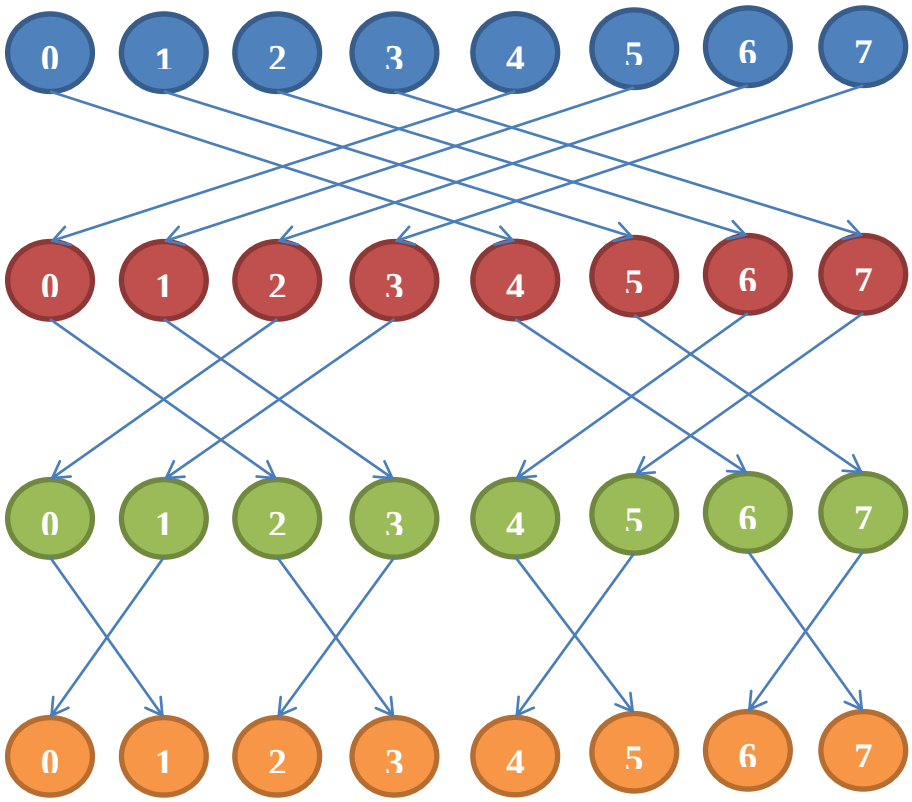
This can be drawn as the following for initial state (top) to final state (bottom).




[H→L] This output shows the communication for high to low given by the program with the verbose option set to true. Again with processor count set to 8. To produce this output I used the –d option.

```
Communication from 0 to 4
Communication from 1 to 5
Communication from 2 to 6
Communication from 3 to 7
Communication from 4 to 0
Communication from 5 to 1
Communication from 6 to 2
Communication from 7 to 3
Communication from 0 to 2
Communication from 1 to 3
Communication from 2 to 0
Communication from 3 to 1
Communication from 4 to 6
Communication from 5 to 7
Communication from 6 to 4
Communication from 7 to 5
Communication from 0 to 1
Communication from 1 to 0
Communication from 2 to 3
Communication from 3 to 2
Communication from 4 to 5
Communication from 5 to 4
Communication from 6 to 7
Communication from 7 to 6
```

This can be drawn as the following for initial state (top) to final state (bottom).




*Correctness of my implementation:*

The output for the communication diagrams already proofs that the communication between the processes is happening in the desired order.

Since my program has the processors ID in every element of its initial sending vector, we can use the small Gauss for calculating the sum in the outcome (for every possible processor count). We have

$$n \cdot (p-1) \cdot \frac{p}{2} = n \cdot \frac{p^2 - p}{2} = R.$$

This means for a vector with $n = 5$ elements and a processor count with $p = 4$ we obtain $R = 30$.

```
The final result is 30.000000
The time for    my_allreduce    in      15      Iterations was  0.000718
The final result is 30.000000
The time for    mpi_allreduce   in      15      Iterations was  0.003542
```

Therefore everything is working as intended. Let's note that on my machine my function is faster.

*Performing the tests to evaluate the speed:*

a.) **network throughput analysis**

Since this test was supposed to be in a range from 1 double (8 bytes) to 1 MB (131072 doubles) I figured out that the most efficient (meaning getting the most information out with the least amount of trials) would be to test it with a facultative but falling, i.e. somehow log related way of distributing the measurement values for the array size with a factor of 8 then 4 and then finally 2 for the

ascending values. In the end I got the following distribution for the number of doubles: 1, 8, 64, 256, 1024, 4096, 16384, 65536, and 131072. That means I have to measure nine times with a processor count of 128.

I made 16 iterations for both algorithms, always dropping out the first iteration – the remaining 15 have been stopped using *MPI_Wtime*.

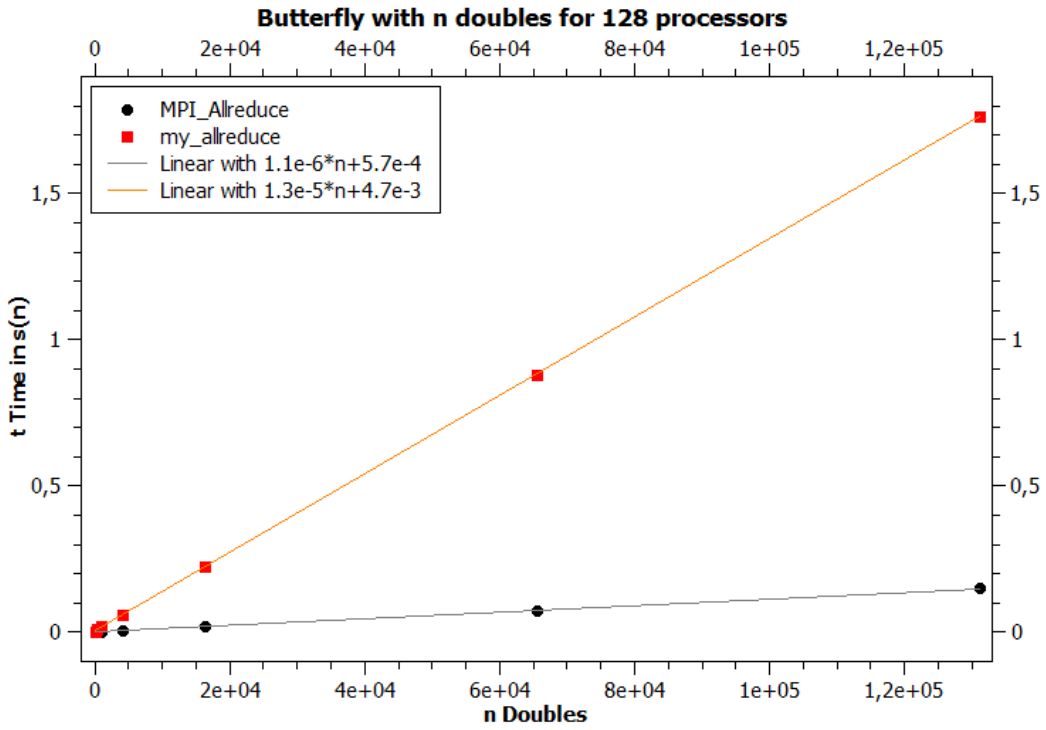



Fig. 1 Comparison of both algorithms in terms of network throughput (both had 15 iterations)

The plot is quite easy to interpret. The MPI function beats my algorithm by an approx. factor of 10. Remarkable is that also the offset is smaller, i.e. the MPI function is also faster to start up.

It is hard to beat the *MPI_Allreduce* since the implementation seems much optimized and it seems to use a more optimized topology for Frost than the Butterfly. Since *my_allreduce* beats the *MPI_Allreduce* on my machine constantly I guess that the topology of my machine differs from the topology of Frost, i.e. I do not have a Torus but probably a star, ring or even a mesh.

Therefore I conclude that the MPI routine is very hard if not impossible to beat by using a specific non-Torus topology, in this case the Butterfly.

b.) **network latency analysis**

Here our data set was predefined, i.e. we set our array to 1 (double) and increase the processor count in factors of 2. Therefore we measure np= 2, 4, 8, 16, 32, 64, 128, which gives us 6 measurements to do, since the 7$^{th}$ value was already measured in the evaluation before.

I made again 16 iterations for both algorithms and dropped the first one. The remaining 15 have been stopped using *MPI_Wtime*.

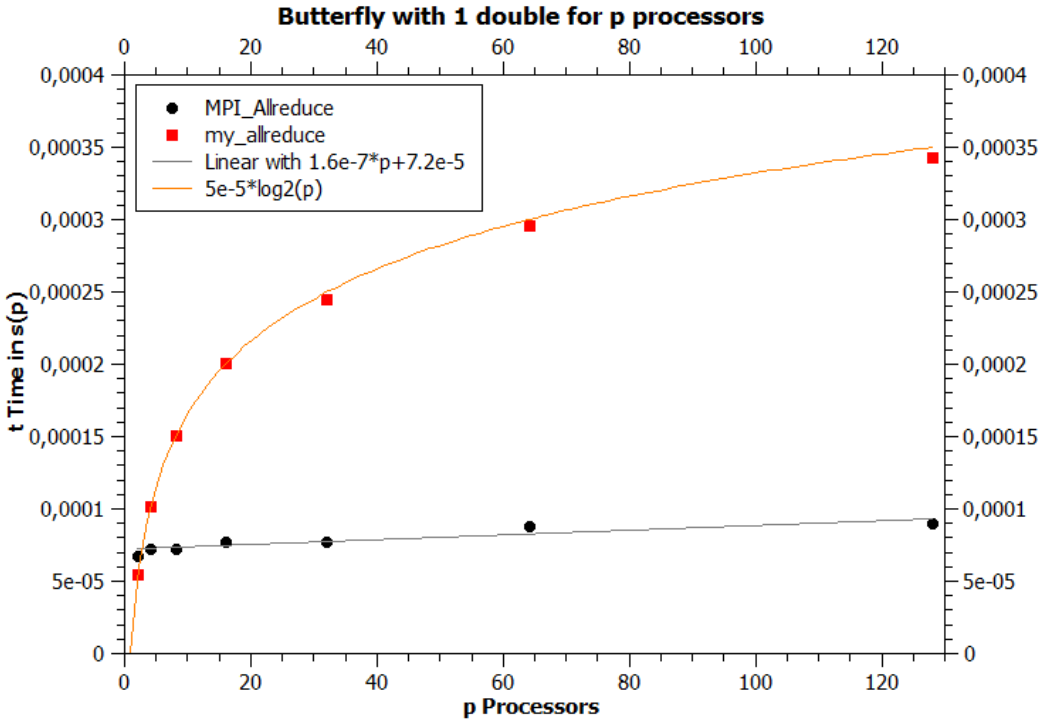



Fig. 2 Comparison of both algorithms in terms of network latency analysis (both had 15 iterations)

While my function shows a clear $\log_2 p$ behavior, the MPI function shows some weird outcome. Between the first and second measurement you can find a slope – whereas between the 2nd and the 3rd one is just finding a straight line without any slope – constant behavior. The 3rd and 4th are then again connected with a line that has a small slope, while the 4th and 5th have a slope of approx. $A = 0$. Therefore the implementation of the *MPI_Allreduce* must explicitly use the Torus or some topology that can be easily embedded in the Torus topology.

**Question (1):**

How does the performance of your implementations compare with *MPI_Allreduce*?

*Answer (1):*

The *MPI_Allreduce* is a lot faster (roughly about $10$ times). The gap could be closed using some techniques but in my opinion it is hard to get closer and since it is a topological thing (look at the $t(p)$ plot) it would not help – scalability is not given at all.

**Question (2):**

Does the bit traversal order matter for your tests?

*Answer (2):*

My first (and naïve clue) would have been that it does not matter. To be sure that my interpretation is correct I just ran a few (5) other tests on NCAR / Frost. I created another plot using QtiPlot. I took the following values (number of doubles, processor count): $(1,2)_1$, $(1,64)_2$, $(1,128)_3$, $(4096, 128)_4$, and $(131072, 128)_5$.

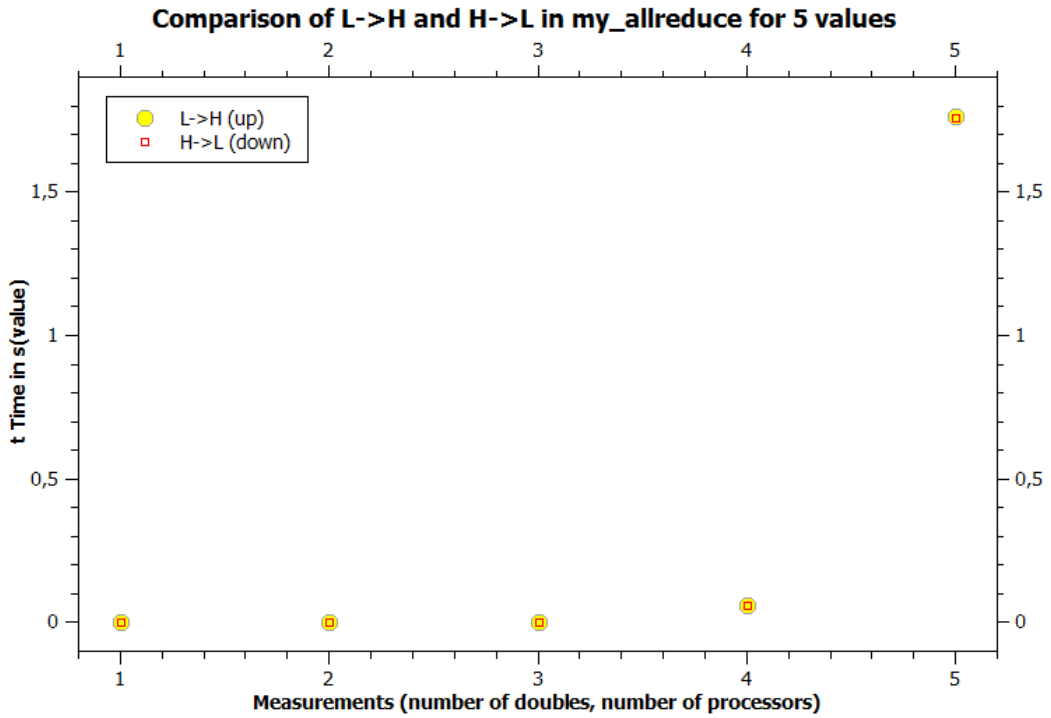



Fig. 3 A simple test to confirm that the bit traversal does not matter

The differences between the values are that small (way below 1%) that all differences can be projected to "tolerance issues". On a normal machine I would just argue that the operating system has given some other task more computation time or something like that. On the Blue Gene machine something like this is not so probable I suppose therefore I can move those differences to "other node with different processor who had probably a different temperature (semiconductor not working at peak point) / network (wire) issues" etc.

So my naïve clue has been confirmed and it does not make a difference at all.

**Question (3):**

How do the vector size and number of processes influence *allreduce* performance?

*Answer (3):*

The *MPI_Allreduce* seems to be in a pretty good shape. They used the Torus to the maximum which can be seen in those steps the plot takes. You can always connect 2 data points together with a straight line (plateau), and then the next two in an ascending line with a small slope. Since computation is much faster than network transfer one is actually able to read out the latency and network transfer rate in s/byte.

However the *my_allreduce* follows the rules of the binary tree and just gets a $\log_2 p$ behavior. The factor before the $\log_2 p$ is also nearly as big as the offset of the *MPI_Allreduce* function, which is not a good sign since the average slope of the MPI inbuilt function is about 440 times smaller than the offset. Therefore the bigger the processor count and the more data that is transferred, the better is the MPI inbuilt function for the allreduce operation.

*Code printout:*

```
1  #define ITERATIONS 32
2
3  #include <stdio.h>
4  #include <stdlib.h>
5  #include <string.h>
6  #include "mpi.h"
7
8  /* prototype of my_allreduce function */
9  void my_allreduce(double* sndvalue, double* recvalue, int count,
10                    unsigned int rank, int processors, int tag,
11                    MPI_Comm comm, int type, char verbose);
12
13 int main(int argc, char* argv[])
14 {
15       int               i,j;
16       int               my_rank;        /* rank of process     */
17       int               p;              /* number of processes */
18       int               tag = 0;        /* tag for messages    */
19       int               count = 1;      /* vector size         */
20       int               type = 1;       /* up or down arg 1|-1 */
21       double*           sndvec;         /* send vector         */
22       double*           recvec;         /* receive vector      */
23       double            final = 0.0;    /* final result        */
24       char              verbose = 0;    /* verbose option 0|1  */
25       double            starttime = 0.0;/* measurement start   */
26       double            endtime = 0.0;  /* measurement end     */
27       double            deltatime = 0.0;/* the final run-time  */
28
29       /* Command line args parser */
30       for(i = 1; i < argc; i++)
31       {
32             /* if we have the -n */
33             if(strcmp(argv[i],"-n") == 0)
34             {
35                   /* but nothing else specified */
36                   if(i == argc - 1)
37                   {
38                         printf("A veclen must be specified using -n.\n");
39                         break;
40                   }
41                   /* or we probably have a number */
42                   count = atoi(argv[++i]);
43                   if(count < 1)
44                   {
45                         /* but that is not a valid number */
46                         printf("Wrong input for n. Must be > than 0.\n");
47                         count = 10000;
48                   }
49             }
50             /* if we have the verbose statement */
51             else if(strcmp(argv[i],"-v") == 0)
52                   verbose = 1;
53             /* if we have the explicit up state */
54             else if(strcmp(argv[i],"-u") == 0)
55                   type = 1;
56             /* if we have the down state option */
57             else if(strcmp(argv[i],"-d") == 0)
58                   type = -1;
59             /* if we have some help statement */
```

```
60              else if(strcmp(argv[i],"-?") == 0)
61              {
62                    printf("Command line arguments\n");
63                    printf("======================\n");
64                    printf("-n X\t sets the length of the vector to X\n");
65                    printf("-v\t verbose mode\n");
66                    printf("-d\t switches to down mode\n");
67                    printf("-u\t switches to up mode\n");
68                    printf("-?\t displays this help\n");
69                    printf("======================");
70                    return 0;
71              }
72        }
73
74      /* Start up MPI */
75      MPI_Init(&argc, &argv);
76
77      /* Find out process rank  */
78      MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
79
80      /* Find out number of processes */
81      MPI_Comm_size(MPI_COMM_WORLD, &p);
82
83        /* Check if processor count is %2 - else finish */
84        if(p % 2 != 0)
85        {
86              if(my_rank == 0)
87                    printf("The execution is limited to 2^n processors.");
88        }
89        else
90        {
91              /* create vector for sending data */
92              sndvec = (double*)malloc(count * sizeof(double));
93
94              /* filling the vector with data - my_rank */
95              for(i = 0; i < count; i++)
96                    sndvec[i] = (double)my_rank;
97              /* measuring process */
98              for(i = 0; i < ITERATIONS; i++)
99              {
100                   final = 0.0;
101                   /* create and set the receive vector */
102                   recvec = (double*)malloc(count * sizeof(double));
103                   for(j = 0; j < count; j++)
104                         recvec[j] = 0.0;
105                   /* start measuring process */
106                   starttime = MPI_Wtime();
107                   /* call the specified function */
108                   if(i < ITERATIONS/2)
109                         my_allreduce(sndvec, recvec, count,
110                               (unsigned int)my_rank, p, tag,
111                               MPI_COMM_WORLD, type, verbose);
112                   else
113                         MPI_Allreduce(sndvec, recvec, count,
114                               MPI_DOUBLE_PRECISION, MPI_SUM,
115                               MPI_COMM_WORLD);
116                   /* end measuring process */
117                   endtime = MPI_Wtime();
118                   /* add the time to the counter or throw away */
119                   if(i == 0 || i % (ITERATIONS / 2) == 0)
120                         deltatime = 0.0;
```

```
121                   else
122                         deltatime += endtime-starttime;
123                   /* gather the data for the final result (chk) */
124                   for(j = 0; j < count; j++)
125                         final += recvec[j];
126                   /* print out final result if all iterations done */
127                   if(my_rank == 0 && (i+1)%(ITERATIONS/2) == 0)
128                   {
129                         /* this depends on if we run 1st half or 2nd */
130                         printf("The final result is %f\n", final);
131                         printf("The t for\t%s\tin\t%d\tIter was\t%f\n",
132                               i < ITERATIONS/2 ? "my" : "mpi",
133                               ITERATIONS/2-1, deltatime);
134                   }
135                   /* clear memory */
136                   free(recvec);
137             }
138             free(sndvec);
139             if(my_rank == 0)
140             {
141                   printf("PROCESSORS:\t%d\n", p);
142                   printf("VECTORLENGTH:\t%d\n", count);
143             }
144       }
145     /* Shut down MPI */
146     MPI_Finalize();
147     return 0;
148 } /* main */
149
150 void my_allreduce(double* sndvalue, double* recvalue, int count,
151                         unsigned int rank, int processors, int tag,
152                         MPI_Comm comm, int type, char verbose)
153 {
154       int               i,j;          /* Loop counters */
155       unsigned int      mask = 1;     /* the bit mask  */
156       unsigned int      dest = 0;     /* destination   */
157       MPI_Status        status;       /* status buffer */
158       char              message[100];/* messages buff */
159       double*           tmpvalue =    /* temporary vec */
160                                       (double*)malloc(count * sizeof(double));
161       /* set the starting mask properly for H->L (down) */
162       if(type == -1)
163             mask = processors / 2;
164       /* get the receive vector set up */
165       for(j = 0; j < count; j++)
166             recvalue[j] += sndvalue[j];
167       for(i = 1; i < processors; i *= 2)
168       {
169             /* bit shift to det. the partner */
170             dest = mask ^ rank;
171             /* communication */
172             MPI_Send(recvalue, count, MPI_DOUBLE_PRECISION, dest, tag,
173                         comm);
174             MPI_Recv(tmpvalue, count, MPI_DOUBLE_PRECISION, dest, tag,
175                         comm, &status);
176             /* if verbose is on then show communication */
177             if(verbose == 1)
178             {
179                   if(rank == 0)
180                   {
181                         printf("Comm. from 0 to %d\n", dest);
```

```
182                         for(j = 1; j < processors; j++)
183                         {
184                               MPI_Recv(&message, 100, MPI_CHAR, j, 1, comm,
185                                           &status);
186                               printf(message);
187                         }
188                   }
189                   else
190                   {
191                         sprintf(message, "Comm. from %d to %d\n", rank,
192                                     dest);
193                         MPI_Send(&message, 100, MPI_CHAR, 0, 1, comm);
194                   }
195             }
196             /* do the desired operation - in this case sum up */
197             for(j = 0; j < count; j++)
198                   recvalue[j] += tmpvalue[j];
199             /* do the selected bit shift - for L->H (up) */
200             if(type == 1)
201                   mask = mask << (unsigned int)1;
202             else /* or H->L (down) */
203                   mask = mask >> (unsigned int)1;
204       }
205 } /* my_allreduce */
```