Name KEY **Class** _____

CPE 412 Omitted Problem Number ____

CPE 412/512 Exam I

Fall Semester 2010

INSTRUCTIONS: **Work in a clear, neat and detailed manner on this paper the equally weighted problems given on this exam. CPE 412 Students should work and 5 of the 6 problems. Clearly indicate which problem you desire to omit. CPE 512 Students are to complete all 6 problems. This is a closed book examination. Allowable items on desk include pencil or pen, basic function calculator, and blank scratch paper. All other items, including personal electronic devices are not to be accessed during the examination. Students are expected to do their own independent work.**

1. What is meant by the term *collective communication*? Outline what is meant by the *broadcast*, *gather*, and *scatter* operations?

Collective communications are message passing communication or synchronization operations in which a set of processes participate concurrently. Unlike the point-to-point message passing operations, collective communications often involve a group of processes that are larger than two.

A ***broadcast*** collective communication operation involves the sending a copy of a data item or structure to the other processes in the system.

A ***gather*** collective communication operation involves the receiving of different segments of a data structure from separate processes and reassembling it into the appropriate segments of a single data structure that is contained in the memory space of one of the processes (root process).

A ***scatter*** collective communication operation involves the distribution of a data structure among the set of processes that are present in the system. The processes can then perform parallel computation on the segment that it has received. It is the opposite of the gather operation.

What two major items does the programmer need to know to write general SPMD code that will automatically divide the problem out among the available set of processes at run time? How does the programmer get this information in MPI?

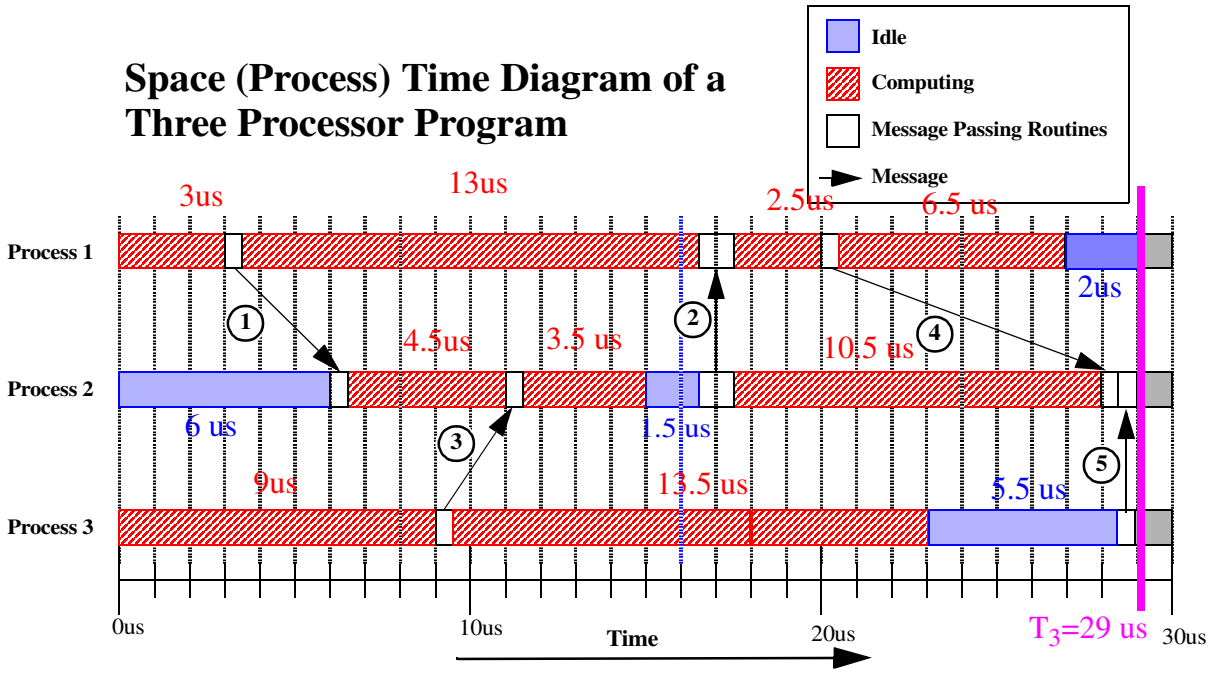
The process id or ***rank*** of the implementation which is assigned from 0 to number of processes created, and the number of processes that have been created in the implementation. This information is obtained in MPI through the

```
MPI_Comm_size(MPI_COMM_WORLD,&numprocs);// find total number of processes
MPI_Comm_rank(MPI_COMM_WORLD,&rank);      // get process identity number
```

What are the fundamental differences between the MPMD and SPMD paradigms?

In MPMD, Multiple Program Multiple Data, separate programs are written for each process. Communication between the separate interactive programs in a deadlock free manner that produces correct results is a challenge in this style of programming. In SPMD, Single Program Multiple Data, a single program is written that executes on each process. Its execution is usually performed on different segments of the data which is distributed among the local memory associated with the process. The processes are loosely synchronized and in general performing different instructions at a given time. Central program does behave differently on each process by utilizing the process id (rank) and the number of processes in the group information as discussed previously. Debugging of SPMD is generally easier than MPMD.

2. A modern parallel profiling routine has produced a space time diagram which has been annotated as shown below.



Answer the following questions assuming a homogeneous system where the three-processor parallel implementation of the program code required no additional computation as compared to the single processor implementation:

a. Determine the sequential execution time, T_1 and the parallel execution time, T_3, for the three-processor implementation.

$$T_1 = \sum \text{computational segments} = 3us + 13us + 2.5us + 6.5us + 4.5us + 3.5us + 10.5us + 9us + 13.5us = 66us$$

T_3 = time last module completes all computational/communication operations

$= 29us$

b. Determine the estimated Speedup, S_3, Efficiency, E_3, and Cost, C_3, for the three-processor implementation.

$$S_3 = \frac{T_1}{T_3} = \frac{66us}{29us} \approx 2.28 \qquad E_3 = \frac{T_1}{T_3 \bullet 3} = \frac{66us}{29us \bullet 3} \approx 0.76$$

$$C_3 \propto T_3 \bullet 3 = 29us \bullet 3 = 87us$$

c. Identify which of the five labeled communications are synchronous or locally blocking in nature.

Two synchronous communications are the ones labeled #2 and #5. In both cases the sending routine has to wait until the receiving routine can accept the data. Would classify the other routines as locally blocking in nature. Here the send routine is able to perform the communication and proceed with additional computation before the corresponding receive is reached on the other processor.

d. Determine the Computation to Communication Ratio.

Tricky question. Difficult to determine when there are not distinct computation and communication phases and some communications between processors overlap computations on other processors. Acceptable answers vary. One answer is average the computation time over the three processors and average the computation then take the ratio of average computation to communication.

$$Avg(t_{comp}) = \frac{66}{3} = 22us$$

$$Avg(t_{comm}) = \frac{(0.5 + 1 + 0.5 + 0.5 + 0.5 + 1 + 0.5 + 0.5 + 0.5 + 0.5)}{3} = \frac{6}{3} = 2$$

$$Avg(\frac{Avg(t_{comp})}{Avg(t_{comm})}) = \frac{22us}{2us} = 11$$

e. Identify which processor has the greatest computational load and which processor presents the greatest bottleneck to the performance.

$$T_{\text{processor 1}} = \sum \text{computational segments} = 3us + 13us + 2.5us + 6.5us$$
$$= 25us$$

$$T_{\text{processor 2}} = \sum \text{computational segments} = 4.5us + 3.5us + 10.5us$$
$$= 18.5us$$

$$T_{\text{processor 3}} = \sum \text{computational segments} = 9us + 13.5us = 22.5us$$

The processor with the greatest computational load is Processor 1 which computes a total of 25us of the required 66us processing load.

The greatest bottleneck in terms of performance is processor 2 which performs the minimum computation, 18.5 us worth, and has the maximum idle time, 7.5 us, and the maximum time spent communicating with other processors, 3us. This processor is a good candidate to accept computation from the other processors.

3. You are working on a commercial contract to compute through numeric simulation the levels of radiation exposure of sensitive equipment that is present on a deep space probe that has already been deployed. The analysis is needed as quickly as possible to aid NASA in determining the degree to which protective measures are to be employed with the general trade-off being the greater the radiation protection employed by the space craft the less science that can be produced. A major scientific event is occurring in the next 1200 hours but there are other major events that will also be occurring in the next four years of the expected duration of the mission and radiation has a cumulative effect. You have been asked to produce a simulation run that approximates this phenomena and produce your results as soon as possible so NASA can command the probe in the best manner possible to maximize the science that it can obtain during the probe's lifetime. NASA is paying your company on an incentive basis using the following formula.

$$R = \$200(1200 - W)$$

where R is the total revenue, and W is the total number of hours it takes to successfully complete the simulation

In previous contract work, you have developed and fully tested a parallel program to perform this analysis that can run on 1 to 64 processing cores at the government owned computing center. The run time characteristics of this program are given by the following equation.

$$T_p = \frac{2000}{p} + 2p$$

where T_p is the program execution time in hours, and p is the number of processing cores employed.

The full cost accounting government computing facility charges \$10 per CPU core hour for this rush job. What is the number of processors that will maximize your company's profit. What is this profit? Fully explain how you arrived at your answer.

Revenue

$$R = \$200(1200 - W) = \$200(1200 - T_p) \quad \text{because} \quad W = T_p$$

Cost

$$C = \$10pT_p$$

Parallel Execution Time

$$T_p = \frac{\$2000}{p} + \$2p$$

Profit

$$P = R - C = \$240000 - \$200T_p - \$10pT_p$$

$$= \$240000 - \left(\frac{\$400000}{p} + \$400p\right) - (\$20000 + \$20p^2)$$

$$= \$240000 - \left(\frac{\$400000}{p} + \$400p\right) - (\$20000 + \$20p^2)$$

$$= \$220000 - \frac{\$400000}{p} - \$400p - \$20p^2$$

Find critical point (maximum profit)

$$\frac{d}{dp}P = 0$$

$$\frac{d}{dp}P = \frac{d}{dp}\left(\$220000 - \frac{\$400000}{p} - \$400p - \$20p^2\right) = \frac{\$400000}{p^2} - \$400 - \$40p = 0$$

or

$$\$10000 - \$10p^2 - p^3 = 0$$

$$p^3 + \$10p^2 = \$10000$$

upward bound of *p* can be found by temporarily ignoring the middle term

$$p^3 \approx 10000 \longrightarrow p < \sqrt[3]{\$100000} \approx 21.5$$

We know that *p* = 21 is probably too large so we will start at *p*=21 and then decrease *p* by one until the first place we see the error [i.e. abs(10000-left side of equation)] increase as *p* is decreased.

***p*=21**	$p^3 + \$10p^2 = 13671$	**error ($3671) -- left side too large**
***p*=20**	$p^3 + \$10p^2 = 12000$	**error ($2000) -- left side too large**
***p*=19**	$p^3 + \$10p^2 = 10469$	**error ($469) -- left side too large**
***p*=18**	$p^3 + \$10p^2 = 9072$	**error ($928) -- left side now too small**

Minimum Error Solution

The profit at *p* = 19 is then

$$P = \$220000 - \frac{\$400000}{(19)} - \$400(19) - \$20(19)^2 \approx \$184{,}127.4$$

4. [10 points] For the following problem assume that α represents the proportion of the original workload that must execute serially, β represents the proportion of the original workload that can be evenly distributed to execute in parallel on up to two processors, and δ represents the proportion of the original workload that can be evenly distributed to execute in parallel on up to four (4) processors. Also assume that the remaining workload can be evenly balanced among the remain processors that are employed and $0 \leq \alpha+\beta+\delta \leq 1.0$. Determine an expression for the maximum speed up that is possible in terms of α, β and δ, if the actual number of processors employed is assumed to arbitrarily large and no additional computation is added by the parallelization process. Also assume that communication and synchronization effects can be neglected and that the different portions associated with α, β, and δ, cannot be overlapped in their execution with any other portions of the code.

T_1 = serial execution time

$$T_p = \text{parallel execution time} = \alpha T_1 + \frac{\beta T_1}{2} + \frac{\delta T_1}{4} + \frac{(1-\alpha-\beta-\delta)T_1}{p}$$

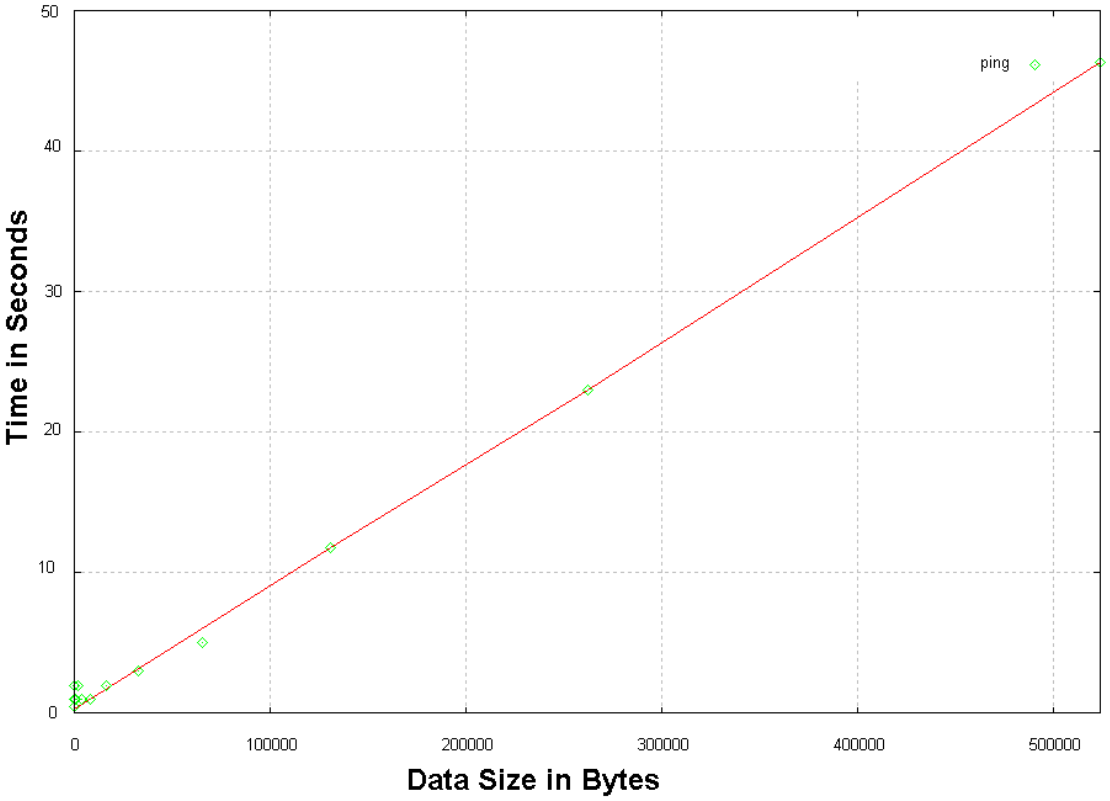
Maximum Speedup when no overhead is present occurs when parallelism is maximum

$$S_{p_{max}} = \lim_{p \to \infty} \frac{T_1}{T_p} = \lim_{p \to \infty} \frac{T_1}{\alpha T_1 + \frac{\beta T_1}{2} + \frac{\delta T_1}{4} + \frac{(1-\alpha-\beta-\delta)T_1}{p}}$$

$\frac{(1-\alpha-\beta-\delta)T_1}{p}$ approaches 0 as p approaches infinity

$$= \frac{1}{\alpha T_1 + \frac{\beta T_1}{2} + \frac{\delta T_1}{4}} = \frac{4}{4\alpha + 2\beta + \delta}$$

5. The following MPI program has been executed on two processors to determine the communication attributes of the system. It produced the data that is shown in the graph below. Assuming the linear model for communication time that is discussed in the text, use this data to calculate T_{data} and $T_{startup}$ for the system. Clearly show how you obtained your results and state any assumptions that were made.




```
using namespace std;
#include <iostream>
#include <mpi.h> /* MPI Prototype Header Files */

main( int argc, char *argv[]) {
   int tag,i,data_size;
   char data[60000000];
   double tm_start,tm_end,tm;

   int numprocs,rank;
   MPI_Status status;
   MPI_Init(&argc,&argv); /* initalize MPI environment */
   MPI_Comm_size(MPI_COMM_WORLD,&numprocs);/* find total number of processes*/
   MPI_Comm_rank(MPI_COMM_WORLD,&rank); /* get processor identity number */

   tag =123;
   for (data_size=1;data_size<=524288;data_size*=2) {
      // Start recording the execution time
      tm_start = MPI_Wtime();
      for (i=0;i<1000;i++) {
         if(rank==0) {
            MPI_Send(data, data_size, MPI_CHAR, 1, tag, MPI_COMM_WORLD);
            MPI_Recv(data,data_size,MPI_CHAR,1,tag,MPI_COMM_WORLD,&status);
         }
         else {
            MPI_Recv(data,data_size,MPI_CHAR,0,tag,MPI_COMM_WORLD,&status);
            MPI_Send(data,data_size, MPI_CHAR, 0, tag, MPI_COMM_WORLD);
         }
      }
      // stop recording the execution time
      tm_end = MPI_Wtime();
      tm=tm_end-tm_start;

      if (rank==0) cout << data_size << " bytes " << tm << " seconds" << endl;
  }
   /* Terminate MPI Program -- clear out all buffers */
   MPI_Finalize();
}
```

Taking two points along the line that connects the data points (last point and 3rd from last), (X_1,Y_1), and (X_2,Y_2) where X_1 is found to be 524288 from the program and X_2 is found by determining that the data point before that will have an X coordinate of 524288/4 = 131072.

The $Y_1 = 46.2$ and $Y_2=11.9$ were estimated from the graph.

$$(X_1, Y_1) = \left(524288, \frac{46.2}{(2 \bullet 1000)}\right) = (524288, 0.0231) \text{ bytes, seconds}$$

#number of Send/Rec Pairs between timing points

Send/Receive Pair

$$(X_2, Y_2) = \left(262144, \frac{11.9}{(2 \bullet 1000)}\right) = (131072, 0.00595) \text{ bytes, seconds}$$

#number of Send/Rec Pairs between timing points

Send/Receive Pair

$$T_{comm} = T_{startup} + T_{data} W$$

Y: T_{comm}; X: W

$T_{startup}$: message latency in seconds

T_{data}: per byte transfer rate in seconds/byte

W: data size in bytes

Applying standard algebraic linear techniques

$$T_{data} = \frac{Y_1 - Y_2}{X_1 - X_2} = \frac{0.0231 - 0.00595}{524288 - 131072} = \frac{0.01715}{393216} = 4.36 \times 10^{-8} \text{ seconds/byte}$$

$$T_{startup} = T_{comm} - T_{data} W \approx 2.33 \times 10^{-4} \text{ seconds}$$

Y_1: T_{comm} = 0.0231

X_1: W = 524288

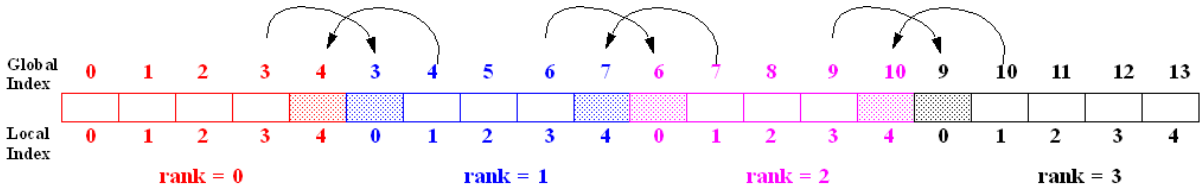
note: $T_{startup}$ in this example is very close to zero

6. The following *compute_temp* function is the heart of a one dimensional heat transfer simulation program that was presented in class. It is designed to execute on any multiple of 2 processors (2, 4, 6, 8, etc.). The program computes the temperature at each point along a one dimensional metal strip at a set of points that are evenly distributed among the set of MPI processes. The input parameters are as follows: *temp[ ]* represents the even subset of points to be computed by a given MPI process, *total_points* represents the total number of points that the set of processors are to calculate the temperature (note: there are two additional points on either end of the line that serve as boundary conditions -- these are not included in the value of *total_points* parameter since the boundary conditions themselves are never computed and thus never change), *num_iterations*, represents the total number of iterations that are to occur (where each iteration represents a certain increment in time), *rank* is the process' MPI rank id, and *numprocs* represents the total number of MPI processes in the system. If it is assumed that the *total_points* parameter is an even multiple of *numprocs* then write a function *print_temp ()* that can be called from the *main* routine that will output the temperature values from the leftmost position (lower global index) to the rightmost point (highest global index) in secession (one after the other based on the global index) by only employing MPI_Send and MPI_Recv point-to-point communication functions. You do not have to output the boundary values.

```
#define MAX_POINTS_PER_PROCESS 20000
void compute_temp(double temp[],int total_points,int num_iterations,
                  int rank, int numprocs) {
   int points_per_process,right_pe,left_pe,i,j;
   double temp_buf[MAX_POINTS_PER_PROCESS+2];
   MPI_Status status;

   points_per_process = total_points/numprocs;
   right_pe = rank+1;
   left_pe  = rank-1;

   for (i=0;i<num_iterations;i++) {
      if (rank%2==0) { // even numbered processes
         MPI_Send(&temp[points_per_process],1,MPI_DOUBLE,right_pe,
            123,MPI_COMM_WORLD);
         MPI_Recv(&temp[points_per_process+1],1,MPI_DOUBLE,right_pe,
            123,MPI_COMM_WORLD,&status);
         if (rank>0) {
            MPI_Send(&temp[1],1,MPI_DOUBLE,left_pe,
               123,MPI_COMM_WORLD);
            MPI_Recv(&temp[0],1,MPI_DOUBLE,left_pe,
               123,MPI_COMM_WORLD,&status);
         }
      }
      else { // odd numbered processes
         MPI_Recv(&temp[0],1,MPI_DOUBLE,left_pe,
            123,MPI_COMM_WORLD,&status);
         MPI_Send(&temp[1],1,MPI_DOUBLE,left_pe,
            123,MPI_COMM_WORLD);
         if (rank < numprocs-1) {
            MPI_Recv(&temp[points_per_process+1],1,MPI_DOUBLE,right_pe,
               123,MPI_COMM_WORLD,&status);
            MPI_Send(&temp[points_per_process],1,MPI_DOUBLE,right_pe,
               123,MPI_COMM_WORLD);
         }
      }
      for (j=1;j<=points_per_process;j++) {
         temp_buf[j]=0.5*(temp[j-1]+temp[j+1]);
      }
      for (j=1;j<=points_per_process;j++) {
         temp[j]=temp_buf[j];
      }
   }
}
```



Example where total points = 12 and numprocs=4

```
// function to display the values in the temp array in a global manner where
// the value are to be displayed in secession from global index 1 through the
// final global index. Function does not have to display the boundary
// conditions. It should display the temp value, local index, and global index
print_temp(double temp[],int total_points, int rank, int Numprocs) {
   char flg;
   int i,left_pr,right_pr,points_per_proc;
   MPI_Status status;

   left_pr=rank-1;
   right_pr=rank+1;
   points_per_proc = total_points/numprocs;

   // wait for turn to print out local temp array -- if rank 0 process you can start right away
   // if you are rank 0 process then go ahead and print boundary condition.
   if (rank!=0) {// skip MPI_Recv if process 0 -- otherwise wait for lower number processes to finish!
      MPI_Recv(&flg,1,MPI_CHAR,left_pr,123,MPI_COMM_WORLD,&status);
   }
   else {
      // if rank 0 process print header and then value of boundary point
      // i.e. global index, local index, and temp of points
      cout << "global local temperature" << endl;
      cout << "    0    0 " << setw(5)<<temp[0] << endl;
   }

   // when it is your turn print out all points that the process has computed.
   // information include global index, local index, and temperature value.
   // print out data points on all processes labeled 1 to points_per_process
   for (i=1;i<=points_per_proc;i++) {
      // print global index, local index, and temp of points
      cout << setw(5) << rank*points_per_proc+i << setw(5) << i << " " << setw(5) << temp[i] << endl;
   }

   // if you are not the last MPI process signal the rightmost process that it is now
   // its turn. if this is the last process then print out rightmost boundary condition.
   if (rank!=numprocs-1) {// skip MPI-Send if last process -- otherwise allow the next process to start
      MPI_Send(&flg,1,MPI_CHAR,right_pr, 123,MPI_COMM_WORLD);
   }
   else {
      // print global index, local index, and temp of points
      cout << setw(5) << points_per_proc*(rank+1)+1 << setw(5)<< points_per_proc+1 << " "
           << setw(5) << temp[points_per_proc+1] << endl;
   }


}
```

Reference Sheet

MPI_Recv - Provides a basic receive operation

SYNOPSIS

```
#include <mpi.h>

int MPI_Recv ( void *buf, int count, MPI_Datatype datatype,
               int source, int tag, MPI_Comm comm,
               MPI_Status *status );
```

DESCRIPTION

The MPI_Recv routine provides a basic receive operation. This routine accepts the following parameters:

buf	Returns the initial address of the receive buffer (choice)
status	Returns the status object (status)
count	Specifies the maximum number of elements in the receive buffer (integer)
datatype	Specifies the data type of each receive buffer element (handle)
source	Specifies the rank of the source (integer)
tag	Specifies the message tag (integer)
comm	Specifies the communicator (handle)
ierror	Specifies the return code value for successful completion, which is in MPI_SUCCESS. MPI_SUCCESS is defined in the mpif.h file.

MPI_Send - Performs a basic send operation

```
#include <mpi.h>

int MPI_Send ( void *buf, int count, MPI_Datatype datatype,
               int dest, int tag, MPI_Comm comm );
```

DESCRIPTION

The MPI_Send routine performs a basic send operation. This routine accepts the following parameters:

buf	Specifies the initial address of the send buffer (choice)
count	Specifies the number of elements in the send buffer (nonnegative integer)
datatype	Specifies the data type of each send buffer element (handle)
dest	Specifies the rank of the destination (integer)
tag	Specifies the message tag (integer)
comm	Specifies the communicator (handle)
ierror	Specifies the return code value for successful completion, which is in MPI_SUCCESS. MPI_SUCCESS is defined in the mpif.h file.