Name KEY Class _____

CPE 412 Omitted Problem Number ____

CPE 412/512 Exam I

Fall Semester 2011

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 any 5 of the 6 problems. Clearly indicate which problem you desire to omit. CPE 512 Students are to complete <u>all</u> 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. The self-consistent field (SCF) method in computational chemistry involves two operations that must be performed in sequence: Fock matrix construction and matrix diagonalization. Assuming that diagonalization portion accounts for 0.5 per cent of total execution time on a uniprocessor computer, use Amdahl's law to determine the maximum speedup that can be obtained if only the Fock matrix construction operation is parallelized..

Matrix Diagonalization Portion **Fock Matrix Portion**

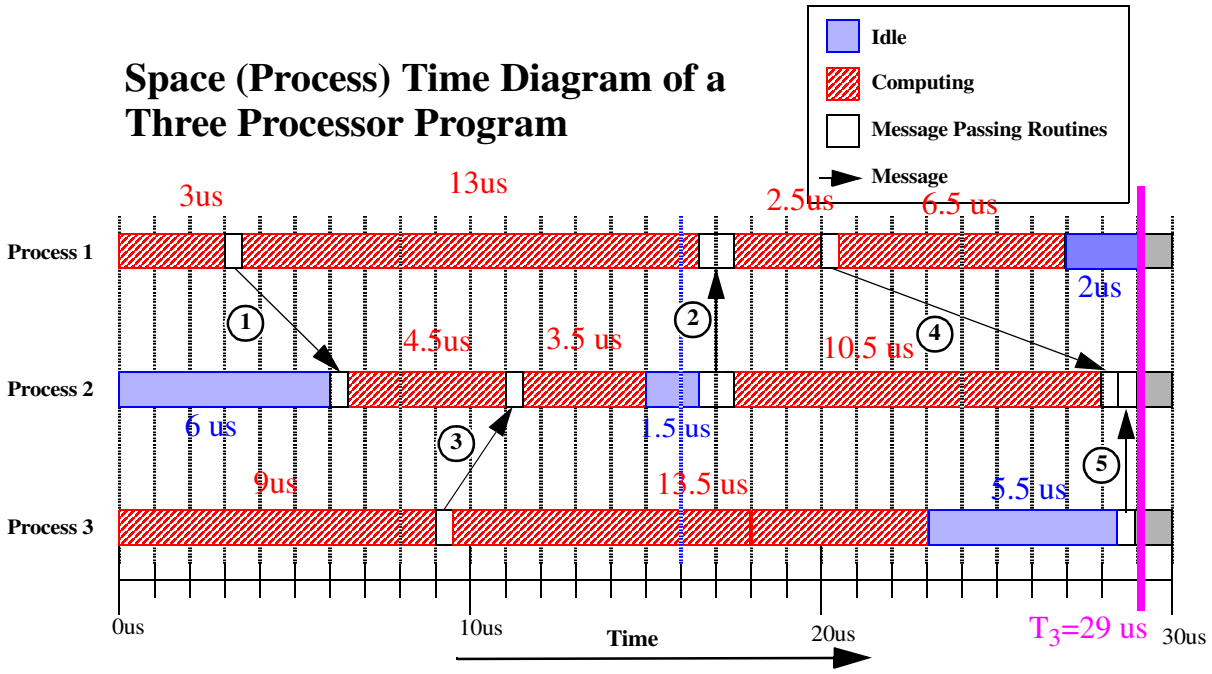
$\alpha = 0.005$ $1 - \alpha = 0.995$

$$T_p = \alpha T_1 + \frac{(1-\alpha)T_1}{p}$$

$$S_p = \frac{T_1}{T_p} = \frac{T_1}{\alpha T_1 + \frac{(1-\alpha)T_1}{p}} = \frac{p}{\alpha(p-1)+1}$$

$$\lim_{p \to \infty} S_p = \lim_{p \to \infty} \frac{p}{\alpha(p-1)+1} = \frac{1}{\alpha} = \frac{1}{0.005} = 200$$

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

   $T_p = 2000/p+0.01p^3$, 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

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

Cost

$$C = \$10pT_p$$

Parallel Execution Time

$$T_p = \frac{\$2000}{p} + \$0.01p^3$$

Profit

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

$$= \$240000 - \left(\frac{\$400000}{p} + \$2p^3\right) - (\$20000 + \$0.1p^4)$$

$$= \$240000 - \frac{\$400000}{p} - \$2p^3 - \$20000 - \$0.1p^4$$

$$= \$220000 - \frac{\$400000}{p} - \$2p^3 - \$0.1p^4$$

Find critical point (maximum profit)

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

$$\frac{d}{dp}P = \frac{d}{dp}\left(\$220000 - \frac{\$400000}{p} - \$2p^3 - \$0.1p^4\right) = \frac{\$400000}{p^2} - \$6p^2 - \$0.4p^3 = 0$$

or $$\$400000 - \$6p^4 - \$0.4p^5 = 0$$

$$p^5 + \$15p^4 = \$1000000$$

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

$$p^5 \approx 10000 \longrightarrow p < \sqrt[5]{\$10000} \approx 15.8$$

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*=15** $p^5 + \$15p^4 = 1518750$ **error ($518750) -- left side too large**

***p*=14** $p^5 + \$15p^4 = 1114064$ **error ($114064) -- left side too large** **Minimum Error Solution**

***p*=13** $p^5 + \$15p^4 = 799708$ **error ($200292) -- left side now too small**

***p*=12** $p^5 + \$15p^4 = 559872$ **error ($440128) -- left side now too small**

The profit at *p* = 14 is then

$$P = \$220000 - \frac{\$400000}{(14)} - \$2(14)^3 - \$0.1(14)^4 \approx \$182{,}099$$

4. The self-consistent field (SCF) method in computational chemistry involves two operations: Fock matrix construction and matrix diagonalization that must be performed in sequence. You are charged with designing a parallel SCF program. You estimate your Fock matrix construction algorithm portion of the SCF program to be 90 percent efficient on your target computer. You must choose between two parallel diagonalization algorithms, which on five hundred processors achieve speedups of 50 and 10, respectively. What overall efficiency do you expect to achieve with each of these two diagonalization algorithms?

assuming Fock matrix and diagonal computation are being performed serially.

$$T_1 = T_{1_{Fock}} + T_{1_{diagonal}}$$ where $T_{1_{Fock}} = wT_1 \quad T_{1_{diagonal}} = (1-w)T_1$

and w is the portion of the overal T_1 that was associated with the Fock matrix construction.

$$T_p = T_{p_{Fock}} + T_{p_{diagonal}}$$ **where**

$$T_{p_{Fock}} = \frac{T_{1_{Fock}}}{pE_{p_{Fock}}} = \frac{wT_1}{pE_{p_{Fock}}} \text{ and } T_{p_{diagonal}} = \frac{T_{1_{diagonal}}}{pE_{p_{diagonal}}} = \frac{(1-w)T_1}{pE_{p_{diagonal}}}$$

Then overall Efficiency can be give by

$$E_p = \frac{T_1}{pT_p} = \frac{T_1}{p(T_{p_{Fock}} + T_{p_{diagonal}})} = \frac{T_1}{p\left(\frac{wT_1}{pE_{p_{Fock}}} + \frac{(1-w)T_1}{pE_{p_{diagonal}}}\right)}$$

$$= \frac{1}{\frac{w}{E_{p_{Fock}}} + \frac{(1-w)}{E_{p_{diagonal}}}} = \frac{E_{p_{Fock}} E_{p_{diagonal}}}{wE_{p_{diagonal}} + (1-w)E_{p_{Fock}}}$$

If $w = 0.995$ as in Problem 1

Case 1:

$$E_{p_{Fock}} = 0.90 \qquad E_{p_{diagonal}} = \frac{S_p}{p} = \frac{50}{500} = 0.1$$

$$E_p = \frac{E_{p_{Fock}} E_{p_{diagonal}}}{wE_{p_{diagonal}} + (1-w)E_{p_{Fock}}} = \frac{(0.90)(0.1)}{(0.995)(0.1) + (0.005)(0.90)} \approx 0.865$$

Case 2:

$$E_{p_{Fock}} = 0.90 \qquad E_{p_{diagonal}} = \frac{S_p}{p} = \frac{10}{500} = 0.02$$

$$E_p = \frac{E_{p_{Fock}} E_{p_{diagonal}}}{wE_{p_{diagonal}} + (1-w)E_{p_{Fock}}} = \frac{(0.90)(0.02)}{(0.995)(0.02) + (0.005)(0.90)} \approx 0.738$$

Bonus [Up to 15 points extra points credit] In problem 4, if your time is as valuable as the computer's, and you expect the more efficient diagonalization algorithm to take one hundred hours longer to program, for how many hours must you plan to use the parallel program if the more efficient algorithm is to be worthwhile?

case 2 case 1

$$T_p - T_p = 100\text{hours}$$

Fock matrix part is the same in both expressions so this portion cancels out leaving only the difference in the two diagonal cases

$$T_{p_{diagonal}}^{\text{case 1}} - T_{p_{diagonal}}^{\text{case 2}} = 100\text{hours}$$

$$\frac{(1-w)T_1}{pE_{p_{diagonal}}^{\text{case 2}}} - \frac{(1-w)T_1}{pE_{p_{diagonal}}^{\text{case 1}}} = 100\text{hours}$$

solving for T_1

$$T_1 = \frac{pE_{p_{diagonal}}^{\text{case 1}} E_{p_{diagonal}}^{\text{case 2}}}{(1-w)(E_{p_{diagonal}}^{\text{case 1}} - E_{p_{diagonal}}^{\text{case 2}})} 100\text{hours}$$

$$= \frac{(500)(0.1)(0.02)}{(1-0.995)(0.1-0.02)} 100\text{hours} = 250000\text{hours}$$

This corresponds to an overall parallel execution time of case 1 of

$$T_p = \frac{T_1}{pE_{p}^{\text{case 1}}} \approx \frac{250000\text{hours}}{(500)(0.865)} \approx 578\text{hours}$$

need to run it at least this long to 'pay' for time it took you to develop it!

5. 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.

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

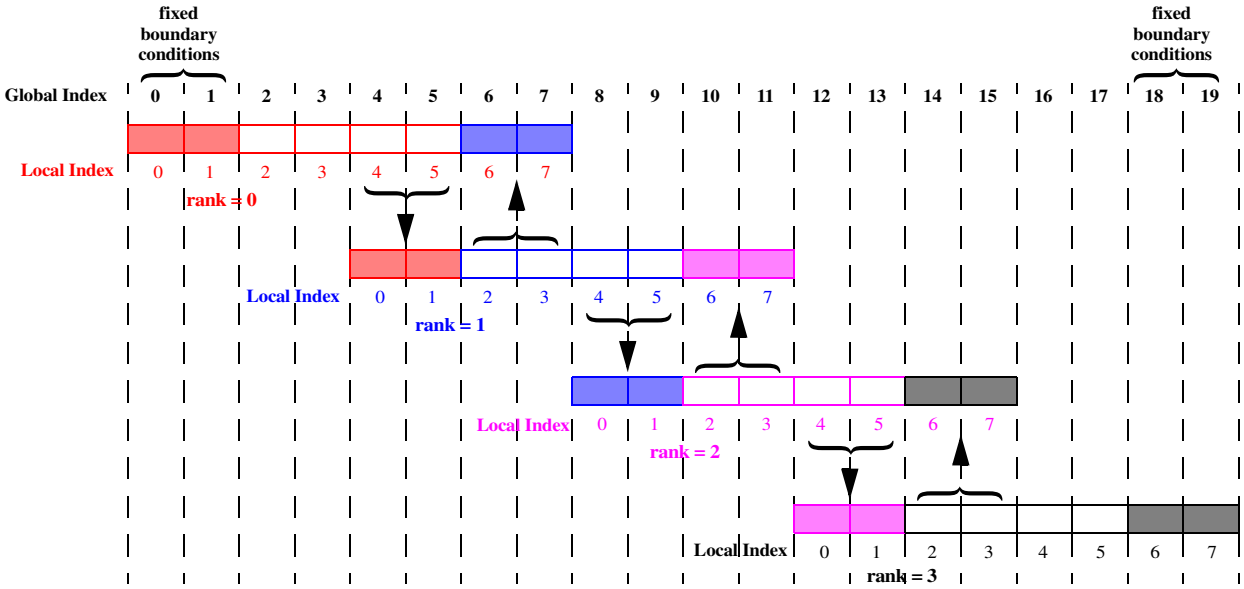
```
MPI_Comm_size(MPI_COMM_WORLD,&numprocs);// find total number of processes
MPI_Comm_rank(MPI_COMM_WORLD,&rank);      // get process identity number
```

6. The following *compute_temp* function is the heart of a one dimensional heat transfer simulation program that 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. It does at each iteration by averaging the two left most and two right most data points that are adjacent to each point that is computed. 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* and that all floating point operations are of unit time weight, determine a general expression for the parallel execution time of the *compute_temp* function. This expression should be an algebraic function of the parameters *numprocs*, *total_points, num_iterations,* and the $T_{startup}$, and T_{data} components of the MPI point-to-point message passing routines. Assume that the buffer size for this implementation is so small that the both the *MPI_Send* and the *MPI_Recv* routines will always block until the entire message has been transferred (i.e. assume they operate synchronously). Also assume that the network that is being used does not block simultaneous point-to-point transfers of data between distinct source and destination processor pairs (the very best case in this regard).

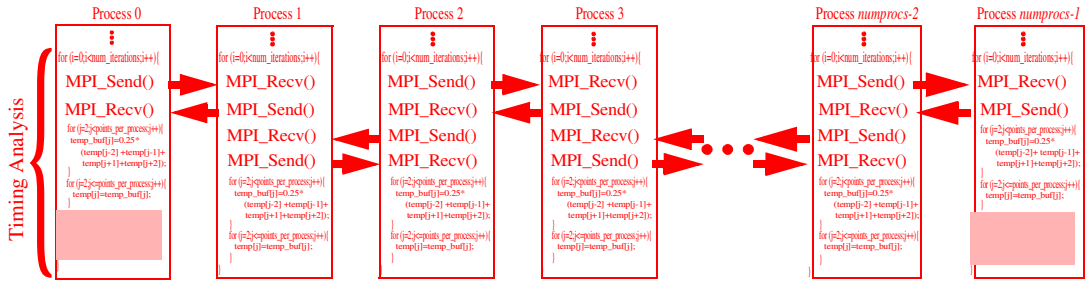
```
#define MAX_POINTS_PER_PROCESS 20000
void compute_temp(double temp[],int total_points,int num_iterations,
                  int rank, int numprocs) {
   int points_per_process,right_pe,left_pe,i,j;
   double temp_buf[MAX_POINTS_PER_PROCESS+4];
   MPI_Status status;

   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],2,MPI_DOUBLE,right_pe,
            123,MPI_COMM_WORLD);
         MPI_Recv(&temp[points_per_process+2],2,MPI_DOUBLE,right_pe,
            123,MPI_COMM_WORLD,&status);
         if (rank>0) {
            MPI_Send(&temp[2],2,MPI_DOUBLE,left_pe,
               123,MPI_COMM_WORLD);
            MPI_Recv(&temp[0],2,MPI_DOUBLE,left_pe,
               123,MPI_COMM_WORLD,&status);
         }
      }
      else { // odd numbered processes
         MPI_Recv(&temp[0],2,MPI_DOUBLE,left_pe,
            123,MPI_COMM_WORLD,&status);
         MPI_Send(&temp[2],2,MPI_DOUBLE,left_pe,
            123,MPI_COMM_WORLD);
         if (rank < numprocs-1) {
            MPI_Recv(&temp[points_per_process+2],2,MPI_DOUBLE,right_pe,
               123,MPI_COMM_WORLD,&status);
            MPI_Send(&temp[points_per_process],2,MPI_DOUBLE,right_pe,
               123,MPI_COMM_WORLD);
         }
      }
      for (j=2;j<points_per_process+2;j++) {
         temp_buf[j]=0.25*(temp[j-2]+temp[j-1]+temp[j+1]+temp[j+2]);
      }
      for (j=2;j<points_per_process+2;j++) {
         temp[j]=temp_buf[j];
      }
   }
}
```



Example where *total_points* = 16 and *numprocs* = 4



$$T_{\text{numprocs}} = \text{num\_interations}(T_{comm} + T_{comp})$$

for numprocs > 2

$$T_{comp} = \frac{\text{total\_points}}{\text{numprocs}} \bullet 4 \text{ floating point operations (3 additions, 1 multiplication)}$$

$$T_{comm} = 4(T_{startup} + (2)T_{data}) \quad \text{for numprocs} > 2$$

$$T_{comm} = 2(T_{startup} + (2)T_{data}) \quad \text{for numprocs} = 2$$

Therefore for numprocs > 2

$$T_{\text{numprocs}} = \text{num\_interations}\left(4T_{startup} + 8T_{data} + 4\frac{\text{total\_points}}{\text{numprocs}}\right)$$

for numprocs = 2

$$T_{\text{numprocs}} = \text{num\_interations}\left(2T_{startup} + 4T_{data} + 4\frac{\text{total\_points}}{\text{numprocs}}\right)$$

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.