Name General Solution **CPE 412 or CPE 512**
circle one

THE UNIVERSITY OF
ALABAMA IN HUNTSVILLE
Department of Electrical and Computer Engineering

CPE 412/512
Fall Semester 2015

INSTRUCTIONS: **Work in a clear, neat and detailed manner on the paper provided the equally weighted problems given on this exam. CPE 412 Students should work any 4 of the 5 problems. For these students, clearly indicate which problem you desire to omit. CPE 512 Students are to complete <u>all</u> 5 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. *a) True or [✓ false]: In MPI you set the number of processes when you write the source code using the SPMD paradigm. For credit you must fully explain your answer.*

False. While it is possible to write programs that will only work with a specified number of processors the number of processes is actually set at run time using a command such as mpirun.

b) Explain the purpose of each of the library calls listed.

"MPI_Init

Initialize the MPI Environment, setting up buffers, general internal data structures, file and I/O handlers, communicator indexing schemes, underlying communication/synchronization mechanisms, etc.

"MPI_Finalize

Remove the MPI Environment that was setup by the MPI_Init command, flushing and removing buffers, communicator indexing schemes, underlying communication/synchronization mechanisms, etc.

"MPI_Comm_rank

Returns a unique MPI process id relative to the MPI Communicator.

"MPI_Comm_size

Returns the number of MPI processors associated with the specified MPI Communicator. In the case of the MPI_COMM_WORLD communicator then the number returned is equal to the entire set of MPI processes that are associated with the program.

"MPI_Send

This is a locally blocking point to point communication operation where data is sent from the current MPI process to the specified MPI process. If there is room for the message in the internal MPI buffer then the message will be placed in that buffer and the MPI_Send command will exit allowing the MPI process to execute other commands while the actual data is being sent. In cases where there is insufficient room in the buffer, the MPI process will block until the message can be placed in the buffer or received by the target process.

"MPI_Recv

This is a blocking point-to-point communication that received data into the current MPI process from another specified MPI process. If the specified data is not present in the receive buffer this process will block the current MPI process until such data is received.

"MPI_Barrier

This is a synchronization routine that is used to coordinate all MPI processes that are associated with the specified communicator. All MPI processes are suspended (can proceed no further) until every process has executed the MPI_Barrier routine. When this occurs all processes are allowed to continue their operation. No actual data is communicated between MPI processes with this routine.

"MPI_Bcast

This is a routine that copies data from a specified root process to the memory space of all of the processes associated with the specified MPI communicator.

"MPI_Scatter

This routine distributes the data that is present on the specified root MPI process among the memory space of all of the processes that are associated with the specified MPI communicator. Reverse of MPI_Gather.

"MPI_Gather

This routine copies data segments from the local memory of individual MPI processes (that are associated with the specified communicator) into a unified global data structure that is created on the specified MPI root process. Reverse of MPI_Scatter.

"MPI_Reduce

This routine employes a specified commutable numeric or logical operation on a distributed data set that is present in the local memory of the individual MPI processes (that are associated with the specified MPI communicator).

b) Explain if the following MPI code segment is correct or not, and why:

Process 0 executes:

```
MPI_Bcast(&mydata, 1, MPI_FLOAT, 0, MPI_COMM_WORLD);
```

All other processes:

```
MPI_Recv(&mydata, 1, MPI_FLOAT, 0, tag, MPI_COMM_WORLD, &status);
```

This is not correct. All MPI processes should enter the collaborative MPI_Bcast routine for it to function correctly.

c) Suppose that process 0 has variable A, and process 1 also has a variable A. Write MPI-like pseudocode to correctly exchange these values between the processes.

A solution using locally blocking point-to-point MPI_Send/MPI_Recv Routines

```
  if (rank==0) {
     MPI_Send(&A,1,MPI_INT,1,0,MPI_COMM_WORLD);
     MPI_Recv(&A,1,MPI_INT,1,0,MPI_COMM_WORLD,&status);
   }
   else if (rank==1) {
     MPI_Recv(&A_buf,1,MPI_INT,0,0,MPI_COMM_WORLD,&status);
     MPI_Send(&A,1,MPI_INT,0,0,MPI_COMM_WORLD);
     A=A_buf;
   }
```

Note: A-buf assumed declared elsewhere in Code

A solution using nonblocking point-to-point MPI_ISend and blocking MPI_Recv Routines

```
 int other_proc;
 MPI_Request req;
 if (rank==0) other_proc=1;
 else other_proc=0;
 // Both Processes Send A
 MPI_Isend(&A,1,MPI_INT,other_proc,0,MPI_COMM_WORLD,&req);
 // Both Processes Receive A but but in Buffer, A_buf
  MPI_Recv(&A_buf,1,MPI_INT,other_proc,0,MPI_COMM_WORLD,&status);
 // Both Processes check to make sure their send is complete
 MPI_Wait(&req,&status); // wait on to make sure own send is complete
 A=A_buf; // Transfer buffered value into A
```

Note: A-buf assumed declared elsewhere in Code

A solution using MPI_Sendrecv Routine

```
 int other_proc;
 if (rank==0) other_proc=1;
 else other_proc=0;
 // Both Processes Send/Receive A relying on internal buffering
  MPI_Sendrecv(&A,1, MPI_INT,other_proc, 0, &A, 1, MPI_INT,
              other_proc,0,MPI_COMM_WORLD,&status);
```

d) Explain if the following MPI code segment is correct or not, and why:

Process 0 executes:

```
MPI_Recv(&yourdata, 1, MPI_FLOAT, 1, tag, MPI_COMM_WORLD, &status);
MPI_Send(&mydata, 1, MPI_FLOAT, 1, tag, MPI_COMM_WORLD);
```

Process 1 executes:

```
MPI_Recv(&yourdata, 1, MPI_FLOAT, 0, tag,MPI_COMM_WORLD, &status);
MPI_Send(&mydata, 1, MPI_FLOAT, 0, tag, MPI_COMM_WORLD);
```

Not correct. MPI_Recv are both blocking -- Each process would stall waiting for data to be sent from the other process. The MPI_Send routine in both cases which would send that data is never executed. This situation is often referred to as a deadlock.

2. From the three process MPI code fragment that is shown below determine the following serial and parallel timing properties given the stated timing characteristics of each of the functions and the MPI point to point communication functions 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.

```
MPI_Init(&argc,&argv); /* initialize MPI environment */
MPI_Comm_rank(MPI_COMM_WORLD,&rank); /* get processor identity number */

if (rank==0) {
   int *a = new int [vec_size];
   int *b = new int [vec_size];
   int *c = new int [vec_size];
   int *d = new int [vec_size];
   function1(a);
   MPI_Ssend(a,vec_size,MPI_INT,1,tag,MPI_COMM_WORLD);
   function2(b,a);
   MPI_Recv(c,vec_size,MPI_INT,1,tag,MPI_COMM_WORLD,&status);
   function3(d,b,c);
   MPI_Ssend(d,vec_size,MPI_INT,1,tag,MPI_COMM_WORLD);
   delete a; delete b; delete c; delete d;
}
else if (rank==1) {
   int *e = new int [vec_size];
   int *f = new int [vec_size];
   int *g = new int [vec_size];
   int *h = new int [vec_size];
   int *i = new int [vec_size];
   int *j = new int [vec_size];
   int *k = new int [vec_size];
   MPI_Recv(e,vec_size,MPI_INT,0,tag,MPI_COMM_WORLD,&status);
   function4(f,e);
   MPI_Recv(g,vec_size,MPI_INT,2,tag,MPI_COMM_WORLD,&status);
   function5(h,f);
   MPI_Ssend(h,vec_size,MPI_INT,0,tag,MPI_COMM_WORLD);
   function6(i,g);
   MPI_Recv(j,vec_size,MPI_INT,0,tag,MPI_COMM_WORLD,&status);
   MPI_Recv(k,vec_size,MPI_INT,2,tag,MPI_COMM_WORLD,&status);
   delete e; delete f; delete h; delete i; delete j; delete k;
}
else if (rank==2) {
   int *l = new int [vec_size];
   int *m = new int [vec_size];
   int *o = new int [vec_size];
   function7(l);
   MPI_Ssend(l,vec_size,MPI_INT,1,tag,MPI_COMM_WORLD);
   function8(m,l);
   function9(o,m);
   MPI_Ssend(o,vec_size,MPI_INT,1,tag,MPI_COMM_WORLD);
   delete l; delete m; delete o;
}
```

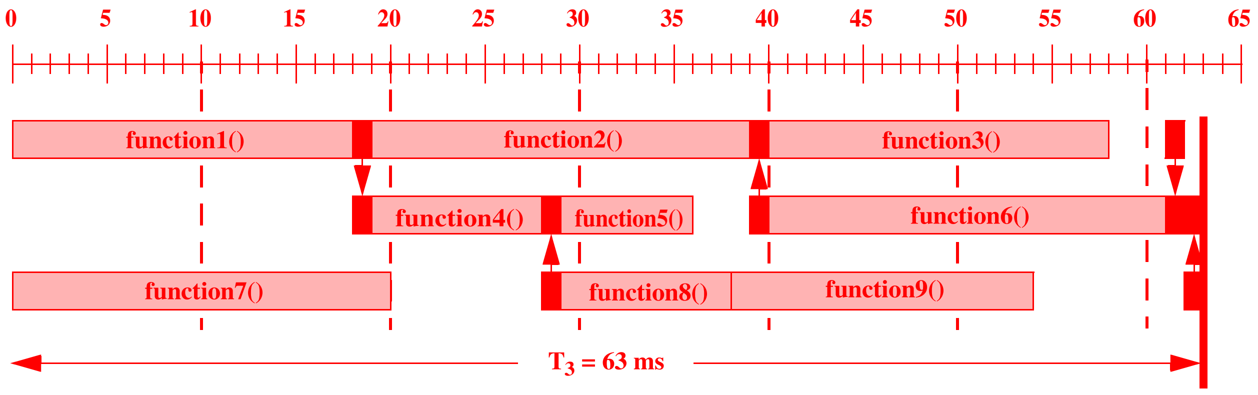
Run Time Characteristics

function1() -- 18 ms
function2() -- 20 ms
function3() -- 18 ms
function4() -- 9 ms
function5() -- 7 ms
function6() -- 21 ms
function7() -- 20 ms
function8() -- 9 ms
function9() -- 16 ms

MPI_Ssend/MPI_Recv -- 1 ms

In determining parallel execution time measure the point in time where all MPI processes have been completed. In the comparative analysis assume that no additional computation as compared to the single processor implementation:

The following execution Gannt Chart can be created from the execution time and communication time data that was presented:



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

$$T_1 = \sum_{i=1}^{9} T_{\text{function}_i} = 18\text{ms} + 20\text{ms} + 18\text{ms} + 9\text{ms} + 7\text{ms} + 21\text{ms} + 20\text{ms} + 9\text{ms} + 16\text{ms}$$

$$= 138\text{ms}$$

b. Determine the estimated Relative Speedup, S_3, Relative Efficiency, E_3, and Cost, C_3, for the three-processor implementation (assume a unit cost constant). Compare the cost to the cost associated with the one processor implementation.

$T_3 = 63\text{ms}$ from Gantt Chart above

$$S_3 = \frac{T_1}{T_3} = \frac{138\text{ms}}{63\text{ms}} \approx 2.19 \qquad E_3 = \frac{T_1}{3T_3} = \frac{138\text{ms}}{(3)63\text{ms}} \approx 0.73$$

$$C_3 = 3T_3 = 3(63\text{ms}) \approx 189$$

Cost of Serial (one processor) Implementation ≈ 138

Parallel Implementation has ≈ 37% more cost than the One-processor one

c. Determine the overall Computation to Communication Ratio of the three processor representation.

$$\text{Computation/Communication Ratio} = \frac{T_{\text{Total}_{comp}}}{T_{\text{Total}_{comm}}} = \frac{138\text{ms}}{10\text{ms}} = 13.8$$

$$= 13.8:1$$

3. For the following problem assume that α represents the proportion of the original workload that must execute serially and β represents the proportion of the original workload that can be evenly distributed to execute in parallel on up to four (4) processing cores. Also assume that the remaining workload can be evenly balanced among the remaining processing cores that are employed and $0 \leq \alpha + \beta \leq 1.0$. Determine an expression for the maximum speed up that is possible in terms of α, and β, 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, and β, cannot be overlapped in their execution with any other portions of the code.

$$T_p = T_1\alpha + \frac{T_1\beta}{4} + \frac{T_1(1-\alpha-\beta)}{p} = \frac{4pT_1\alpha + pT_1\beta + 4T_1(1-\alpha-\beta)}{4p}$$

$$S_p = \frac{T_1}{T_p} + \frac{4p}{4p\alpha + p\beta + 4(1-\alpha-\beta)} = \frac{4p}{(4\alpha+\beta)p + 4(1-\alpha-\beta)}$$

$$S_p = \frac{T_1}{T_p} + \frac{4p}{4p\alpha + p\beta + 4(1-\alpha-\beta)} = \frac{4p}{(4\alpha+\beta)p + 4(1-\alpha-\beta)}$$

$$\lim_{p\to\infty} S_p = \overset{\text{L'Hopital's Rule}}{\lim_{p\to\infty}} \frac{\frac{d}{dp}4p}{\frac{d}{dp}(4\alpha+\beta)p + 4(1-\alpha-\beta)} = \lim_{p\to\infty}\frac{4}{4\alpha+\beta}$$

$$= \frac{4}{4\alpha+\beta}$$

4. Using your knowledge of MPI and the linear model for point-to-point interprocess communication routines, analyze the following ***homemade_gather()*** function in terms of its total communication time $\mathbf{T_{comm}}$. The expression for $\mathbf{T_{comm}}$ that you obtain should be a function of the number of processing cores, $\mathbf{N_{procs}}$, employed, number of elements in the send buffer, **send_rec_count_per_process**, and the point-to-point communication paramters of $\mathbf{T_{startup}}$, and $\mathbf{T_{data}}$. Assume that a separate processing core is used for each MPI process and that the numbers of processing cores employed is always a power of two. Also 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 that they in effect will operate synchronously). Perform this calculation assuming that the network that is being used is uniform in structure and does not block simultaneous point-to-point transfers of data between distinct source and destination processing core pairs (the very best case in this regard).

```
// send_rec_buf -- starting address of each processes send buffer
//                 also starting address of receive buffer (significant only
//                 on root, rank==0, process)
// send_rec_counter_per_process -- number of elements in each send buffer
// rank -- process id
// nmtsks -- number of MPI processes relative to MPI_COMM_WORLD
void homemade_gather(int *send_rec_buf, int send_rec_count_per_process,
                     int rank, int nmtsks) {
   MPI_Status status;
   for (int step = 1; step < nmtsks; step = step*2) {
      if (rank % step == 0) {
         int data_size=step*send_rec_count_per_process;
         // is_even(x) returns true (or 1) if x is even or zero
         // returns a false (or 0) if x is odd
         if (is_even(rank/step)) {
            MPI_Recv(&send_rec_buf[data_size],data_size,MPI_INT,
                     rank+step,0,MPI_COMM_WORLD,&status);
          }
          else {
             MPI_Send(send_rec_buf,data_size,MPI_INT,
                     rank-step,0,MPI_COMM_WORLD);
          }
      }
      else {
         break;
      }
   }
}
```

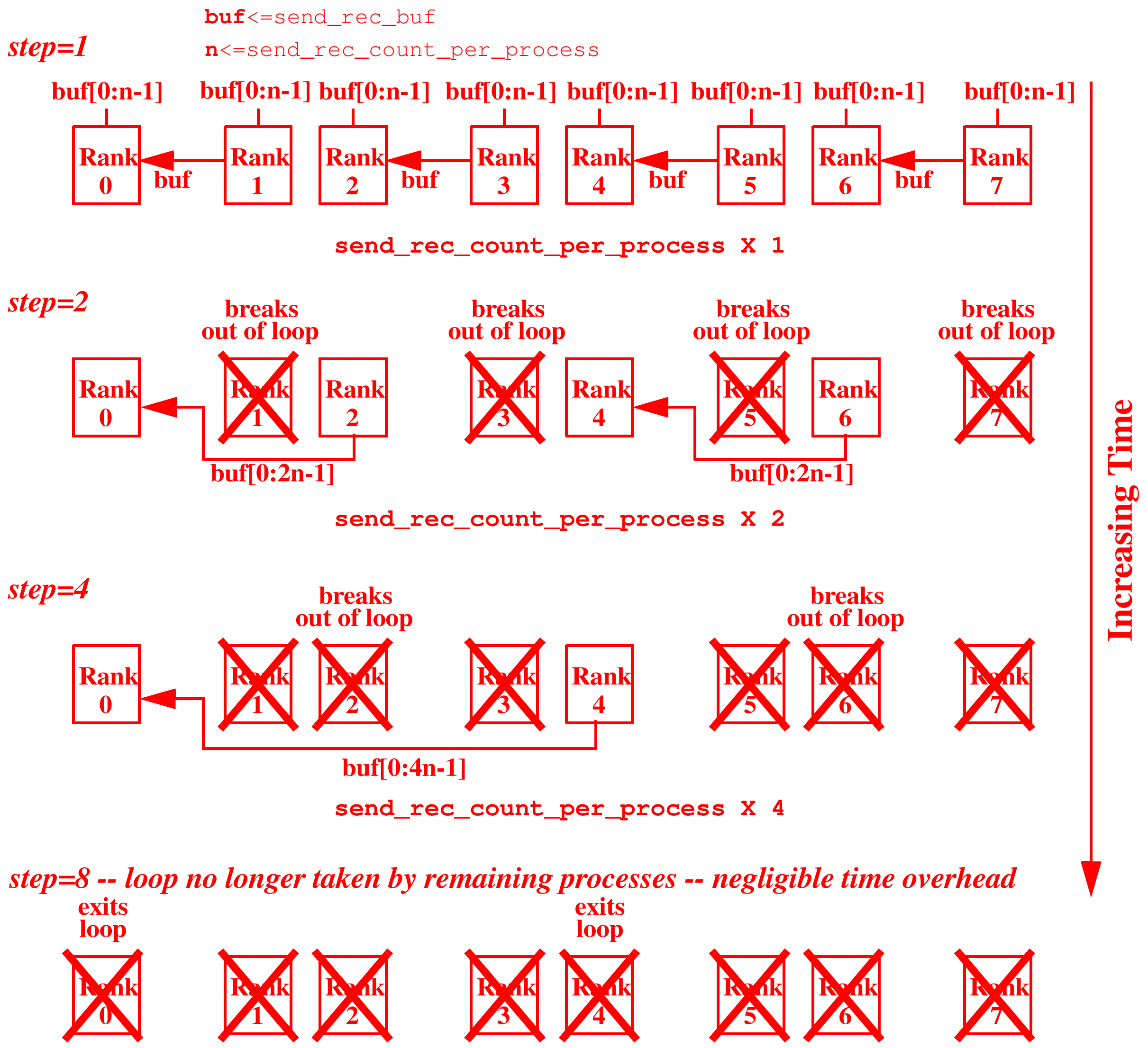
Expression for $\mathbf{T_{comm}}$

$$\mathbf{T_{comm}} = \sum_{i=1}^{\log_2(N_{procs})} \left(T_{startup} + 2^{i-1}(\text{send\_rec\_count\_per\_process})T_{data}\right)$$

$$= \sum_{i=1}^{\log_2(N_{procs})} T_{startup} + (\text{send\_rec\_count\_per\_process})T_{data} \sum_{i=1}^{\log_2(N_{procs})} 2^{i-1}$$

$$= \log_2(N_{procs})T_{startup} + (\text{send\_rec\_count\_per\_process})T_{data}(N_{procs} - 1)$$

Runtime Evaluation of *homemade_gather()* routine



buf<=send_rec_buf
n<=send_rec_count_per_process
step=1
buf[0:n-1]
Rank 0
Rank 1
Rank 2
Rank 3
Rank 4
Rank 5
Rank 6
Rank 7
buf
send_rec_count_per_process X 1
step=2
breaks out of loop
buf[0:2n-1]
send_rec_count_per_process X 2
step=4
buf[0:4n-1]
send_rec_count_per_process X 4
step=8 -- loop no longer taken by remaining processes -- negligible time overhead
exits loop
Increasing Time

5. The following ***compute_temp*** function part of a two dimensional heat transfer simulation that is designed to execute on any multiple of 2 processors (2, 4, 6, 8, etc.). This function is a simple extension of the one dimensional heat transfer simulation discussed in class. The function computes the temperature at each point along a two dimensional surface where the points are evenly distributed along the rows and columns. The function assumes that the overall temperature data is distributed in standard row major ordering in such a way that each MPI process is responsible the same number of rows. The input parameters are as follows: *temp* represents the even subset of row data to be computed by a given MPI process, *total_rows* represents the total number of rows that make up the overall problem (this does not include two additional rows at the top and bottom that serve as boundary conditions), *columns_per_row,* represents the total number of columns each row contains (this does include the two boundary condition points associated with each row), *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's MPI rank id, and *numprocs* represents the total number of MPI processes in the system relative to MPI_COMM_WORLD. A known limitation of this program is that the total_rows needs to be a multiple of the number of MPI processes.

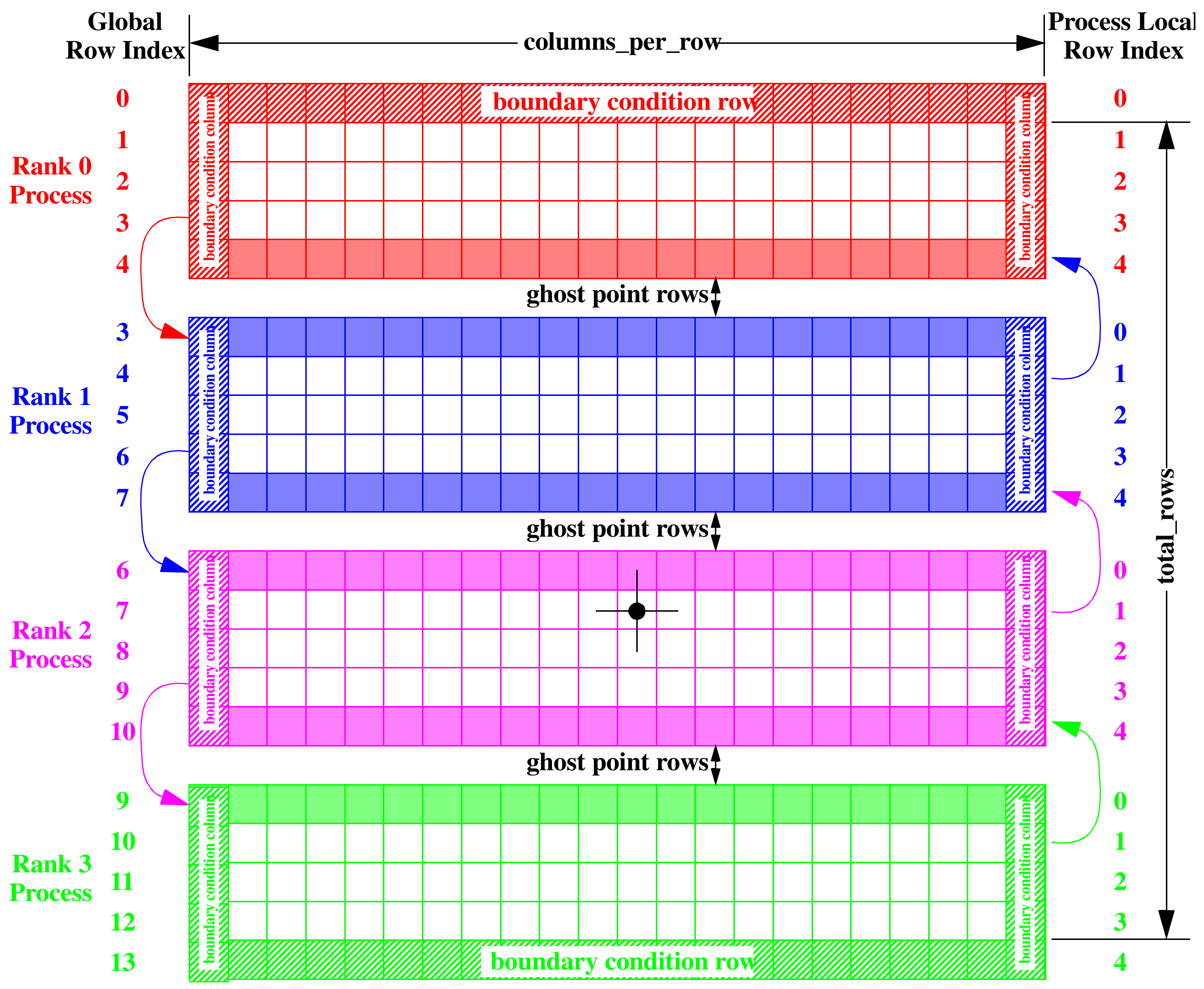
```
void compute_temp(double temp[],int total_rows,int columns_per_row, int num_iterations,
                  int rank, int numprocs) {
   MPI_Status status;
   #define TEMP_BUF(x,y) temp_buf[x*columns_per_row+y] // *(temp_buf+x*columns_per_row+y)
   #define TEMP(x,y)     temp[x*columns_per_row+y]     // *(temp+x*columns_per_row+y)

   int rows_per_process = total_rows/numprocs;
   double *temp_buf = new double[(rows_per_process+2)*columns_per_row];

   int up_pe = rank+1;
   int down_pe = rank-1;
   for (int i=0;i<num_iterations;i++) {
      if (rank%2==0) { // even numbered processes
         MPI_Send(&temp[rows_per_process*columns_per_row],columns_per_row,MPI_DOUBLE,
                  up_pe,123,MPI_COMM_WORLD);
         MPI_Recv(&temp[rows_per_process*columns_per_row+1],columns_per_row,MPI_DOUBLE,
                  up_pe,123,MPI_COMM_WORLD,&status);
         if (rank>0) {
            MPI_Send(&temp[columns_per_row],columns_per_row,MPI_DOUBLE,
                     down_pe,123,MPI_COMM_WORLD);
            MPI_Recv(&temp[0],columns_per_row,MPI_DOUBLE,down_pe,
                     123,MPI_COMM_WORLD,&status);
         }
      }
      else { // odd numbered processes
         MPI_Recv(&temp[0],columns_per_row,MPI_DOUBLE,down_pe,123,MPI_COMM_WORLD,&status);
          MPI_Send(&temp[columns_per_row],columns_per_row,MPI_DOUBLE,down_pe,123,MPI_COMM_WORLD);
         if (rank < numprocs-1) {
            MPI_Recv(&temp[rows_per_process*columns_per_row+1],columns_per_row,MPI_DOUBLE,up_pe,
                     123,MPI_COMM_WORLD,&status);
            MPI_Send(&temp[rows_per_process*columns_per_row],columns_per_row,MPI_DOUBLE,up_pe,
                     123,MPI_COMM_WORLD);
         }
      }

      for (int j=1;j<=rows_per_process;j++) {
         for (int k=1;k<=columns_per_row-2;k++) {
            TEMP_BUF(j,k)=0.25*(TEMP(j-1,k)+TEMP(j+1,k)+TEMP(j,k-1)+TEMP(j,k+1));
         }
      }
      for (int j=1;j<=rows_per_process;j++) {
         for (int k=1;k<=columns_per_row-2;k++) {
            TEMP(j,k)=TEMP_BUF(j,k);
         }
      }
   }
   delete temp_buf;
}
```

If it is assumed that the *total_rows* 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_row*, *columns_per_row*, *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).



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

$$T_{comp} = \frac{\text{total\_rows(columns\_per\_row-2)}}{\text{numprocs}} \bullet 4 \text{ floating point operations (3 additions, 1 multiplication)}$$

for numprocs > 2 (multiples of 2)

$$T_{comm} = 4(T_{startup} + \text{columns\_per\_row}\,T_{data})$$

for numprocs = 2

$$T_{comm} = 2(T_{startup} + \text{columns\_per\_row}\,T_{data})$$

Therefore overall Parallel Execution Time is given by

for numprocs > 2 (multiples of 2)

$$T_{\text{numprocs}} = 4\left(\text{num\_interations}\right)\left(T_{startup} + \left(\text{columns\_per\_row}\right)T_{data} + \frac{\text{total\_rows(columns\_per\_row-2)}}{\text{numprocs}}\right)$$

for numprocs = 2

$$T_{\text{numprocs}} = 2(\text{num\_interations})(T_{startup} + \left(\text{columns\_per\_row}\right)T_{data} + \text{total\_rows(columns\_per\_row-2)})$$

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

SYNOPSIS

C:

```
#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.

MPI_Ssend - Performs a blocking synchronous send operation

SYNOPSIS

C:

```
#include <mpi.h>

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

DESCRIPTION

The MPI_Ssend 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.