SOLUTION

THE UNIVERSITY OF ALABAMA IN HUNTSVILLE

Department of Electrical and Computer Engineering

CPE 412/512

Fall Semester 2013

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 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 Under what conditions, if any, may the efficiency metric of a parallel program be greater than one? Is this normally the case? If it is then explain the reasons why this is so, if not explain why the efficiency can never be greater than one.

Case 1: It is possible for the efficiency to be greater than one if the data and/or code segments are partitioned in the parallel problem in such a manner that it will fit into faster cache memory than the entire problem can be placed when a single CPU core is used. It is possible that this improved use of locality can overcome the added communication overhead associated with the parallelization process.
Case 2: It is also possible for the efficiency to be greater than one if the problem is non-deterministic in nature. A problem such as this is a search problem where the first solution that meets the stated critiera is selected. If this search space is divided for independent processing among the cores and the solution falls in an area of the search space that would normally be considered late in the search process but due to dividing up the space now falls at the beginning of the search space of the new partition then this speedup would be arbitrarily large (much greater than the number of processing cores) resulting in the efficiency being much greater than one.
In general, most cases, efficiency goes down as the number of processing cores increase because of the extra overhead due to load imbalance and inter-process communication.

2) Let f be the percentage of a program code which can be executed simultaneously by p processor in a computer system. Assume that the remaining code must be executed sequentially by a single processor. Each processor has an execution rate of Δ MFLOPS, and all the processors are assumed equally capable.

(a) Derive an expression for the effective MFLOPS rate when using the system for exclusive execution of this program, in terms of the parameters p, f, and Δ..

$$\Delta_{eff} = \frac{W}{T_p}$$

$$T_p = T_1(1-f) + \frac{f^2 T_1}{p}$$

If we define W = number of instructions (i.e. computational workload)

Then $T_1 = \frac{W}{\Delta}$ or $\Delta = \frac{W}{T_1}$ thus

$\Delta_{eff} = \frac{W}{T_p}$ is a reasonable definition of the Effective MIPS rate of a p processor system

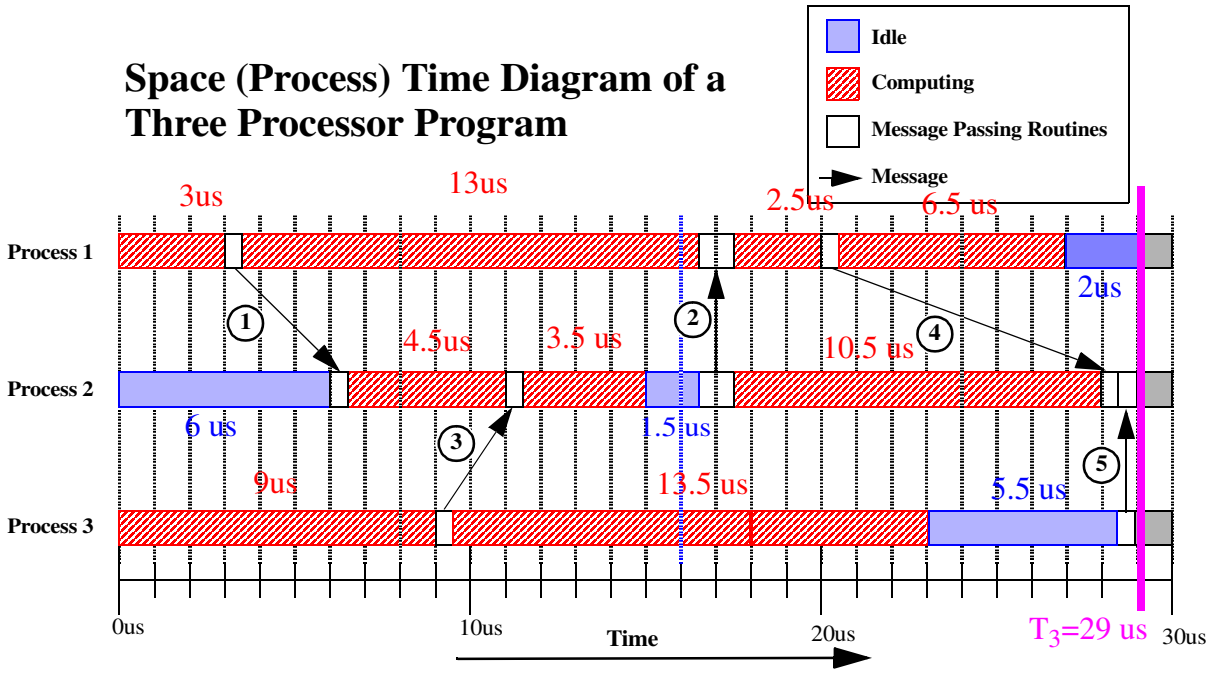
since $T_p = \frac{W}{\Delta}\left((1-f) + \frac{f}{p}\right)$

$$\Delta_{eff} = \frac{W}{\frac{W}{\Delta}\left((1-f) + \frac{f}{p}\right)} = \frac{\Delta p}{((1-f)p + f)}$$

(b) If p = 16 and Δ = 400 MFLOPS, determine the value of f which will yield a system performance of 8,000 MFLOPS.

With 16 processing cores the maximum possible execution rate is 16 x 400 = 6,400 MFLOPS. This would occur when f=1 which is the maximum value it can obtain -- meaning 100% of the computation can execute simultaneously on the processors in the system. If you plug this value into the formula from part a you get a required value of f of approximately 1.01 which is beyond the range of the parameter.

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 Relative Speedup, S_3, Relative Efficiency, E_3, and Cost, C_3, for the three-processor implementation (assume a unit cost constant).

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

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

c. Identify which of the four labeled communications appear to be synchronous in nature and explain why you believe this to be the case.

Two synchronous communications are the ones labeled #2 and #5. In both cases the sending routines appears to wait until the receiving routine can accept the data.

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.

4) A parallel square matrix/matrix algorithm has a parallel execution time expression that is given the expression $T_p = (K_3n^3 + K_2n^2 + K_1n)/p + K_0 + (p-1)(p+2)n^2/p\ T_{data} + 3\ (p-1)T_{startup}$ where K_3, K_2, K_1, K_0 are computation related constants and T_{data}, $T_{startup}$ are inter-processing core communication derived constants. Each of these values are assumed to be obtained through empirical profiling techniques. The value of n represents the dimension of the matrices to be multiplied.

a) If it is assumed that this same performance approximation expression holds for the serial version of the algorithm, determine a general expression for the relative speed up and relative efficiency for this algorithm if it is to be executed on a homogeneous computing platform, where the inter-core communication network is uniform and the processors are identical in terms of processing power.

$$S_p = \frac{T_1}{T_p} = \frac{K_3n^3 + K_2n^2 + K_1n + K_0}{\frac{K_3n^3 + K_2n^2 + K_1n}{p} + K_0 + \frac{(p-1)(p+2)n^2}{p}T_{data} + 3(p-1)T_{startup}}$$

$$= \frac{\left(K_3n^3 + K_2n^2 + K_1n + K_0\right)p}{K_3n^3 + K_2n^2 + K_1n + K_0p + (p^2 + p - 2)n^2T_{data} + 3p(p-1)T_{startup}}$$

$$E_p = \frac{T_1}{pT_p} = \frac{K_3n^3 + K_2n^2 + K_1n + K_0}{K_3n^3 + K_2n^2 + K_1n + K_0p + (p^2 + p - 2)n^2T_{data} + 3p(p-1)T_{startup}}$$

b) What is the number of processing cores that will result in the maximum speedup for n=200, $K_3=2\text{x}10^{-9}$,$K_2=2\text{x}10^{-7}$, $K_1=2\text{x}10^{-5}$, $K_0=2\text{x}10^{-4}$, $T_{startup}=200\text{x}10^{-6}$, and $T_{data}=2\text{x}10^{-9}$. Justify your answer.

Since T_1 is constant, the maximum S_p will occur when T_p is minimum

$$T_p = \left(K_3n^3 + K_2n^2 + K_1n\right)p^{-1} + K_0 + (p + 1 - 2p^{-1})n^2T_{data} + 3(p-1)T_{startup}$$

$$\frac{dT_p}{dp} = \frac{d}{dp}\left(K_3n^3 + K_2n^2 + K_1n\right)p^{-1} + K_0 + (p + 1 - 2p^{-1})n^2T_{data} + 3(p-1)T_{startup}$$

$$= -\frac{K_3n^3 + K_2n^2 + K_1n}{p^2} + n^2T_{data} + \frac{2n^2T_{data}}{p^2} + 3T_{startup}$$

$$= -\frac{K_3n^3 + K_2n^2 + K_1n - 2n^2T_{data}}{p^2} + n^2T_{data} + 3T_{startup}$$

Finding critical point, p, by setting $\frac{dT_p}{dp} = 0$ to yield

$$p = \sqrt{\frac{K_3n^3 + K_2n^2 + K_1n - 2n^2T_{data}}{n^2T_{data} + 3T_{startup}}} \approx 6$$

For the parameter values assumed in this problem

c) What is the number of processing cores that will result in the maximum efficiency? Justify your answer explaining why this is the case.

You can answer this by inspection because the parallel execution time has an overhead component that grows with increasing p. This will cause the efficiency to decrease as the number of processing cores increases. The best value is when p=1. This value is E_p=1 because all the other overhead terms go to zero when this occurs. One should not that the best (lowest) execution time does not occur at this point but occurs when p=6 as discovered in part b of the problem.

If the best known serial algorithm is $O(n^3)$ then is this algorithm cost optimal? Explain what is meant by cost optimality and discuss your answer in a manner that conforms to this definition.

The *cost* of a parallel algorithm is the product of the number of processing cores employed and its run time. A parallel algorithm is considered to be *cost optimal* when its costs has the same asyntotic order as the cost of the best know serial algorithm. In other words when $O(T_s) = O(pT_p)$, where T_s is the run time of the best know serial algorithm and T_p is the parallel algorithm. The run time characteristics of both algorithms are a function of the dimension parameter, n, and the number of processing cores, p, in the parallel case. T_p is a polonomial. Its asymtotic behavior depends upon the leading coefficient of the varable around which is to approach infinity, which is the problem size variable n. For T_p this dominate term is K_3n^3/p. This $O(T_s)=O(n^3)$ was given for best serial algorithm and $O(pT_p)=O(p\ K_3n^3/p)=O(K_3n^3)=O(n^3)$ which means that the parallel algorithm <u>is</u> indeed <u>cost optimal</u>. For a large enough value of n then the cost of the parallel version will be in the same order as the cost of the serial one.

5. a) Using your knowledge of MPI and the linear model for point-to-point interprocess communication routines, analyze the following scalar ***reduce()*** 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 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). You may ignore the time costs associated with the actual reduction operation -- only the communication cost should be reflected in your expression.

```
int reduce(int myval, int rank, int nmtsks) {

   MPI_Status status;

   for (int step = 1; step < nmtsks; step = step*2) {
      int rec;
      if (rank % step == 0) {
         // 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(&rec,1,MPI_INT,rank+step,0,MPI_COMM_WORLD,&status);
            myval += rec;
         }
         else {
           MPI_Send(&myval,1,MPI_INT,rank-step,0,MPI_COMM_WORLD);
         }
      }
      else {
         break;
      }
   }
   return myval;
}
```

See diagram on next page for explaination

Expression for $\mathbf{T_{comm}}$ **[note: Tprocs=mntsks]**

$$\mathbf{T_{comm} = (T_{startup}+T_{data})log_2(N_{procs})}$$

b) Describe in detail how you would modify the above program so that it would perform a minimization type reduction operation.

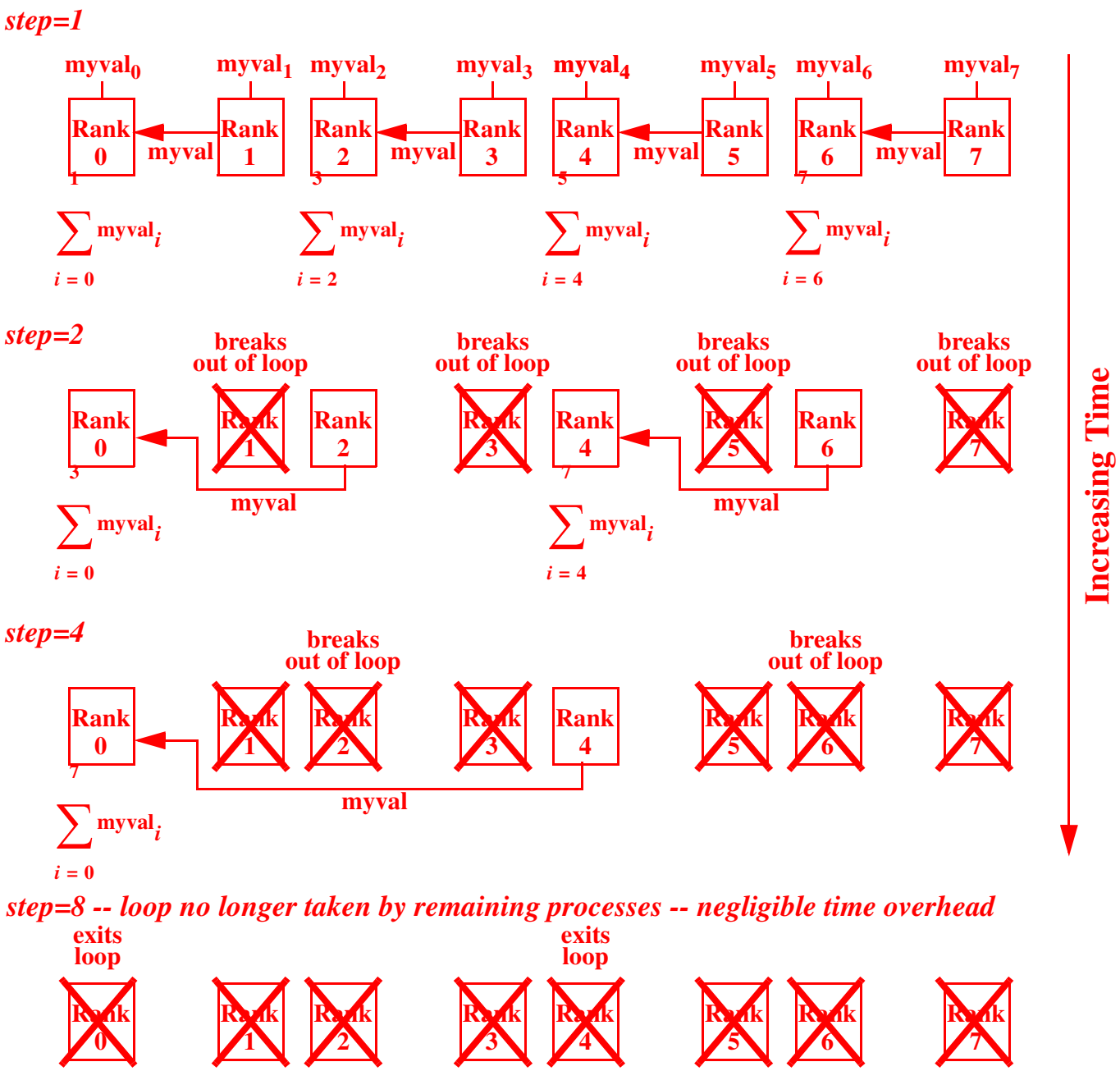
Replace

```
  myval += rec;
```

line with something like

```
  if (rec < myval) {
    myval = rec;
  }
```

Runtime Evaluation of reduce() routine



Number of communication phases is 3 or in general it will be $\log_2(N_{procs})$, where N_{procs} is the number of MPI processes that are executing in parallel on separate processing cores.

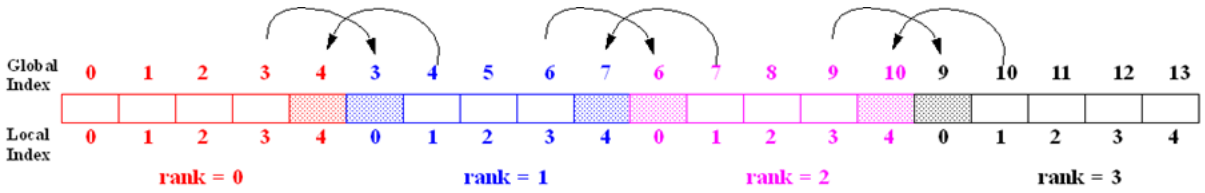
6 The following ***compute_temp*** function is the major portion of a one dimensional heat transfer simulation that is designed to execute on any multiple of 2 processors (2, 4, 6, 8, etc.). The program computes the temperature at each point along a one dimensional metal strip at a set of points that are evenly distributed among the set of MPI processes. 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's MPI rank id, and *numprocs* represents the total number of MPI processes in the system. A known limitation of this program is that the total_points needs to be a multiple of the number of MPI processes.

```
void compute_temp(double temp[],int total_points,int num_iterations,
                  int rank, int numprocs) {

   MPI_Status status;
   int points_per_process = total_points/numprocs;
   double *temp_buf = new double[points_per_process+2];

   int right_pe = rank+1;
   int left_pe = rank-1;
   for (int 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 (int j=1;j<=points_per_process;j++) {
         temp_buf[j]=0.5*(temp[j-1]+temp[j+1]);
      }
      for (int j=1;j<=points_per_process;j++) {
         temp[j]=temp_buf[j];
      }
   }
   delete temp_buf;
```

Rewrite the communication section of the program so that it efficiently utilizes the nonblocking **MPI_ISend** and **MPI_Wait** point-to-point communication routines instead of the **MPI_Send** routines and still functions correctly. You are to continue to utilize the blocking **MPI_Recv** routines. It is assumed that the interconnection network allows for simultaneous transfer of data between any two processing cores..

```
void compute_temp(double temp[],int total_points,int num_iterations,
                  int rank, int numprocs) {

   MPI_Status status;
   MPI_Request req1, req2;

   int points_per_process = total_points/numprocs;
   double *temp_buf = new double[points_per_process+2];
   int right_pe = rank+1;
   int left_pe = rank-1;

   for (int i=0;i<num_iterations;i++) {
     // Send right and left at the same time -- then use blocking receives
     if (rank!=numprocs-1) { // if not right most process send right
        MPI_Isend(&temp[points_per_process],1,MPI_DOUBLE,right_pe,123,MPI_COMM_WORLD,&req1);
     }
     if (rank!=0) { // if not left-most process send to the left
        MPI_Isend(&temp[1],1,MPI_DOUBLE,left_pe,123,MPI_COMM_WORLD,&req2);
     }
     // receive right and then left -- messages transfers should be occurring on both sides
     //                               at the same time for intermeddiate processes
     if (rank!=numprocs-1) { // if not right-most process receive from the right
        MPI_Recv(&temp[points_per_process+1],1,MPI_DOUBLE,right_pe,123,MPI_COMM_WORLD,&status);
     }
     if (rank!=0) { // if not the left-most process then receive from the left
        MPI_Recv(&temp[0],1,MPI_DOUBLE,left_pe,123,MPI_COMM_WORLD,&status);
     }
     // make sure all data has been sent from both left and right (where applicable)
     // before finishing this section -- would not want to modify send buffer if everything
     // has not been sent!
     if (rank!=numprocs-1) MPI_Wait(&req1,&status); // wait on right send to complete
     if (rank!=0) MPI_Wait(&req2,&status);          // wait on left send to complete

       for (int j=1;j<=points_per_process;j++) {
          temp_buf[j]=0.5*(temp[j-1]+temp[j+1]);
       }
       for (int j=1;j<=points_per_process;j++) {
          temp[j]=temp_buf[j];
       }
   }
   delete temp_buf;
}
```

Enter your Code Here!



Example where total points = 12 and numprocs=4