Name KEY Omitted Problem Number (CPE 412 only) _____

Class _____

CPE 412/512 Exam II

Spring Semester 2009
(60% of Total Exam Grade)

INSTRUCTIONS: **CPE 512 students are to work all 5 problems on this exam. CPE 412 students should omit one problem from this exam. Problems are equally weighted. This is a closed-book examination.**

1. Assume that a heterogeneous system is made up of four processors that each are identical in computational capability (i.e. same type and number of internal functional units, etc.) but have differing clocking rates that cause them to exhibit different computational rates as measured in Million Floating Point Operations per second, Mflops. These estimated computational rates of each processor are 100 Mflops for Processor P_0, 200 Mflops for Processor P_1, 400 Mflops for Processor P_2, and 800 Mflops for Processor P_3.

   If there is no restriction as to the amount of parallelism that is exploitable within any segment of the computational workload associated with the algorithm to be executed and ignoring any synchronization or communication effects on performance then

   a) What is the percent distribution of the workload among the four processors that would achieve the minimal execution time?

$$min(T_p) = \frac{W_0}{\Delta_0} = \frac{W_1}{\Delta_1} = \frac{W_2}{\Delta_2} = \frac{W_3}{\Delta_3}$$ when there is no idle time (all processors are busy all the time)

where $W = \sum_{i=0}^{p-1} W_i$

This occurs when $W_i = \frac{\Delta_i}{\Delta_T} W$ where $\Delta_T = \sum_{j=0}^{p-1} \Delta_j = 100 + 200 + 400 + 800 = 1500 Mflops$

$$min(T_p) = \frac{W}{\Delta_T}$$ for each component of the equation

Optimal Percent Distribution is:

processor 0: $\frac{\Delta_0}{\Delta_T} = \frac{100}{1500} 100\% \approx 6.7\%$

processor 1: $\frac{\Delta_1}{\Delta_T} = \frac{200}{1500} 100\% \approx 13.3\%$

processor 2: $\frac{\Delta_2}{\Delta_T} = \frac{400}{1500} 100\% \approx 26.7\%$

processor 3: $\frac{\Delta_3}{\Delta_T} = \frac{800}{1500} 100\% \approx 53.3\%$

   b) If this workload was determined to be 1,000 million floating point operations what is the execution time associated with the above optimal percent distribution of the workload among the four processors?

$$min(T_p) = \frac{W}{\Delta_T} = \frac{1000 \text{ million floating point operations}}{1500 \text{ million floating point operations per second}} \approx 0.67 \text{ seconds}$$

c) If it were determined that 10% of the workload could not be parallelized but must execute in a serial manner (but there were no restrictions on the rest of the workload) what is the best possible execution time possible to execute this algorithm on the four processor system assuming the the workload was 1,000 million floating point operations?

$W = W_{serial} + W_{parallel}$ where

$W_{serial} = 0.1W = 100$ million floating point operations

$W_{parallel} = 0.9W = 900$ million floating point operations

Assuming that parallel workload cannot be executed at the same time the serial workload.

$$T_{min} = min(T_{serial}) + min(T_{parallel})$$

$$min(T_{serial}) = \frac{W_{serial}}{\Delta_3} = \frac{100}{800} = 0.125 \text{ seconds}$$

Δ_3 ← fastest processor available

$$min(T_{parallel}) = \frac{W_{parallel}}{\Delta_T} = \frac{900}{1500} = 0.6 \text{ seconds}$$

$$T_{min} = 0.125 + 0.6 = 0.725 \text{ seconds}$$

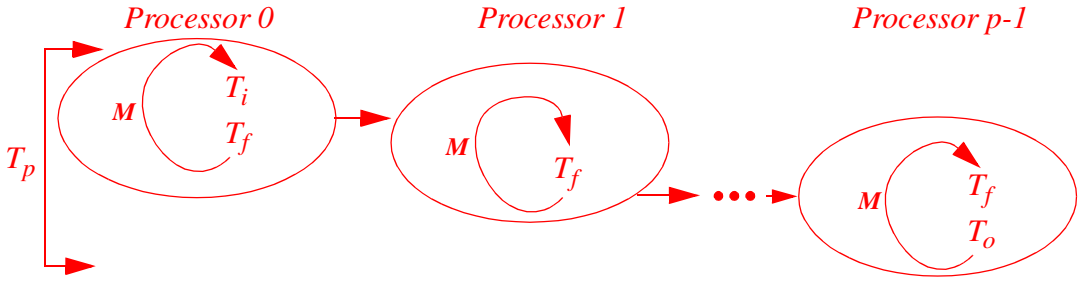
Assuming that parallel workload can be executed at the same time the serial workload. (i.e. while one processor is executing the serial workload the others can execute the parallel workload.)

From previous calculations it has been shown that the minimum time for the serial calculations on the fastest processor was 0.125 seconds which is less than the required calculations for the ideal case where there is no restrictions placed on the number of processors that can simultaneously execution portions of the load with all 4 processors participating in this calculation. This means that such a serialization component would have no effect on the overall execution time because there is enough work to keep the three remaining processors busy during the time the one processor was busy calculating the serial portion of the workload. Thus the overall execution time would be 0.67 seconds as before. This was not the intent of this problem, but is an acceptable answer if the above assumption was stated.

2. A SPMD MPI code fragment for a parallel program is shown below:

```
int main( int argc, char *argv[])
{
   int i,j,type;
   int numprocs,rank;
   MPI_Status status;
   char fun_input[MESG_SIZE],fun_output[MESG_SIZE];

   MPI_Init(&argc,&argv); /* initalize MPI environment */
   /* find total number of processors*/
   MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
   /* get processor identity number */
   MPI_Comm_rank(MPI_COMM_WORLD,&rank);
   type = 123; // set message type to something

   TIMER_CLEAR;
   TIMER_START;
   for (i=0;i<M;i++) {
      if (rank==0) {
         input(fun_input);
      }
      else {
         MPI_Recv(&fun_input,MESG_SIZE,MPI_CHAR,rank-1,type,
             MPI_COMM_WORLD,&status);
      }

      function(numprocs,rank,fun_input,fun_output);

      if (rank<numprocs-1) {
         MPI_Send(&fun_output,MESG_SIZE,MPI_CHAR,rank+1,type,
              MPI_COMM_WORLD);
      }
      else {
         output(fun_output);
      }
   }
   TIMER_STOP;
   printf("time=%e seconds\n\n",(double) TIMER_ELAPSED/1000000.0);
   MPI_Finalize();
```

Note, TIMER_START, TIMER_STOP, and TIMER_ELAPSED are macros that are defined elsewhere in the code. They are used to calculate the elapsed time between the TIMER_START and TIMER_STOP portion of the code. The constant **M** is also defined elsewhere in the code. Assuming that the communication cost is negligible, develop an expression for speed up for a ***p*** processor implementation. Assume that the execution time of the *function()* function is T_f, and also assume that the execution time for the *input()* and *output()* functions are represented by T_i and T_o, respectively. For the other symbols in your expression use the variable or constant names taken from the program code.



$$T_p \approx (M-1+p)(max((T_i+T_f),(T_o+T_f),(T_f)))$$

$T_s = M(T_i + pT_f + T_o)$ comparing the result when an equivalent amount of computation is performed on one processor

If $T_f = 100/\mathbf{p}$, and $T_i = T_o = 1$ time units, then what is the speed up for a 10 processor implementation if M = 1?

$$T_p \approx (M-1+p)(max((T_i+T_f),(T_o+T_f),(T_f))) = (1-1+10)\left(1+\frac{100}{10}\right)$$
$$\approx 10 \bullet 11 = 110$$
$$T_s = M(T_i + pT_f + T_o) = 1\left(1+10\left(\frac{100}{10}\right)+1\right) = 102$$
$$S_p = \frac{T_s}{T_p} = \frac{102}{110} \approx 0.92$$

by the equation but the actual speedup would be 1 because on the first pass, there is no pipeline delay associated with waiting for the input or output function to complete

If $T_f = 100/\mathbf{p}$, and $T_i = T_o = 1$ time units, then what is the speed up for a 10 processor implementation, if M=100?

$$T_p \approx (M-1+p)(max((T_i+T_f),(T_o+T_f),(T_f))) = (100-1+10)\left(1+\frac{100}{10}\right)$$
$$\approx 109 \bullet 11 = 1199$$
$$T_s = M(T_i + pT_f + T_o) = 100\left(1+10\left(\frac{100}{10}\right)+1\right) = 10200$$
$$S_p = \frac{T_s}{T_p} = \frac{10200}{1199} \approx 8.51$$

If $T_f = 100/\mathbf{p}$, and $T_i = 1$, $T_o = 5$ time units, then what is the speed up for a 10 processor implementation if M=100?

$$T_p \approx (M-1+p)(max((T_i+T_f),(T_o+T_f),(T_f))) = (100-1+10)\left(5+\frac{100}{10}\right)$$
$$\approx 109 \bullet 15 = 1635$$
$$T_s = M(T_i + pT_f + T_o) = 100\left(1+10\left(\frac{100}{10}\right)+5\right) = 10600$$
$$S_p = \frac{T_s}{T_p} = \frac{10600}{1635} \approx 6.48$$

4. The MPI program below utilizes one Monte Carlo Technique to estimate pi/4.

```
// Monte Carlo method for calculating pi/4 using two randomly generated
// coordinates (x,y) and determining if they fall within the upper
// quadrant of the unit circle
// B. Earl Wells October 2009
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <mpi.h> /* MPI Prototype Header Files */

#define PI (double) 3.14159265358979323846
int main( int argc, char *argv[])
{
   double x,y,num,rel_err,pi_4,local_sum,gl_sum;
   int i,rank,numprocs,group_sz,N,seed;

   MPI_Status status;

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

   if (rank==0) {
      printf("Enter the number of sample points you desire:\n");
      scanf("%d",&N);
      printf("Enter random number seed\n");
      scanf("%d",&seed);
   }
   MPI_Bcast (&N, 1, MPI_INT,0, MPI_COMM_WORLD);
   MPI_Bcast (&seed, 1, MPI_INT,0, MPI_COMM_WORLD);
   srand48(seed); // seed random number

   // round up to even distribution on the set of process
   group_sz = ceil((float) N/(float) numprocs);
   N = group_sz*numprocs; // insure that N is the next most multiple of the number of processes

   local_sum = 0; // clear local sum variable
   for (i=0;i<group_sz;i++) {
      x = drand48(); // drand48() produces a uniformly distributed value
      y = drand48(); //           between 0 and 1
      if ((x*x + y*y) < 1.0) local_sum++;
   }

   MPI_Reduce (&local_sum, &gl_sum, 1,MPI_DOUBLE,MPI_SUM,0,MPI_COMM_WORLD);
   if (rank==0) {
      num = (double) gl_sum/(double) N;
      pi_4 = PI/4.0;
      rel_err = fabs(num-pi_4)/pi_4;
      printf("answer=%e [ref=%e](relative error=%e)\n",num,pi_4,rel_err);
   }
   MPI_Finalize();
}
```

(a) The estimates that for pi/4 that the above program produces do not appear to improve as the number of processors is increased and the number of data points present on each process remains constant. Why is this the case? What needs to be done to improve its accuracy?

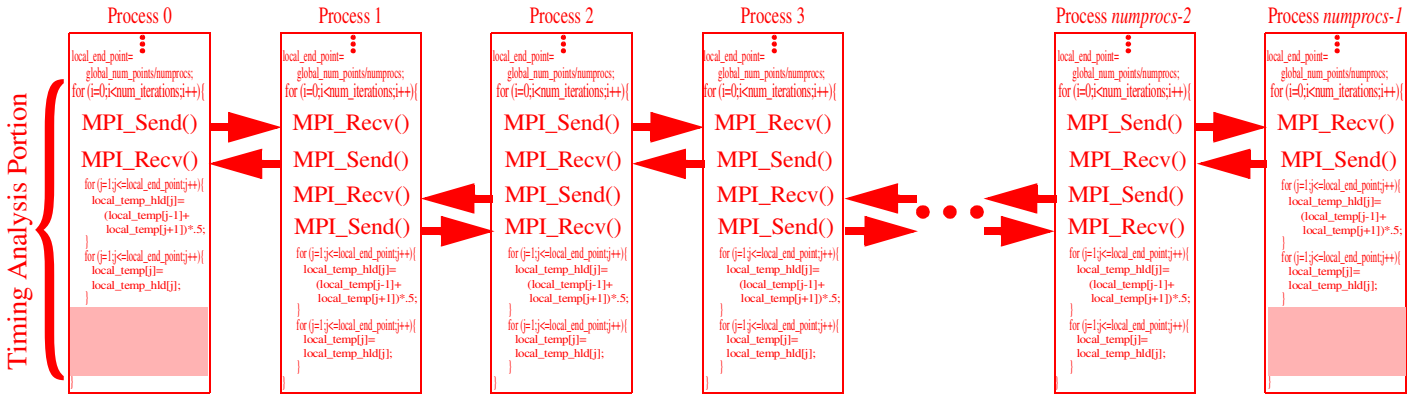
Each processor uses the same psuedo random number generator with the same seed, resulting in the same numerical Monte Carlo experiment being performed on each processor (The average of identical concurrent experiments are the same as if only one were performed -- i.e. we are doing the same thing simultaneiously and averaging the results!!) To improve the accuracy we could utilize a conventional random number generator and scatter the random numbers to each process as discussed in the text, we could have a master process send its random numbers to each of the processes when they request, or we could have a parallel random number generation scheme where each process generates its set of random numbers independently from the other processes (this has to be done in a manner that keeps the random number streams independent from one another).

(b) Rewrite the highlighted portions of the program so that only one call to *drand48()* per iteration is used but the technique is still considered to be Monte Carlo based.

```
local_sum = 0; // clear local sum of function variable
 for (i=0;i<group_sz;i++) {
      x = drand48();
      local_sum  +=  sqrt(1-x*x);
 }
```

5. Evaluate the time complexity between the two points that are indicated in the following code segment of a parallel temperature calculation program shown below. Determine a general expression that is a function of the number of processors, *numprocs*, the data size variable, *global_num_points*, the number of iterations variable, *num_iterations*, and the $T_{startup}$, and T_{data} constants. Assume that the data size, *global_num_points*, is evenly divisible by the number of processors and 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). Perform this calculation assuming 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). In computing the computation cost only include floating point operations that operate on the *double* type. Assume in your analysis assume that time has been normalized so that each floating point operation takes one time unit and the constants $T_{startup}$, and T_{data} take into account this normalization.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <mpi.h> /* MPI Prototype Header Files */

#define MX_SZ 100000
int main( int argc, char *argv[])
{
   int numprocs,rank,type=123;
   int global_num_points,local_end_point,num_iterations,i,j;
   MPI_Status status;
   double local_temp[MX_SZ],local_temp_hld[MX_SZ];

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

   if (rank==0) {
      printf("Enter Number of Grid Points:\n");
      scanf("%d",&global_num_points);
      printf("Enter Number of Iterations:\n");
      scanf("%d",&num_iterations);
   }
   MPI_Bcast (&global_num_points, 1, MPI_INT,0,MPI_COMM_WORLD);
   MPI_Bcast (&num_iterations, 1, MPI_INT,0,MPI_COMM_WORLD);
   local_end_point = global_num_points/numprocs;
   for (i=1;i<=local_end_point;i++) local_temp[i]=0.0;  // initial conditions temp = 0 C
   //set up left boundary temperature
   if (rank==0)local_temp[0] = 100.0; // boundary condition temp = 100 C
   //set up right boundary temperature
   if (rank==0)local_temp[local_end_point+1] = 0.0; // boundary condition temp = 0 C
                                                    <-------------- Start point for analysis
   for (i=0;i<num_iterations;i++) {
      if (rank%2==0) {
         if (rank<numprocs-1) {
            MPI_Send(&local_temp[local_end_point],1,MPI_DOUBLE,rank+1,type,MPI_COMM_WORLD);
            MPI_Recv(&local_temp[local_end_point+1],1,MPI_DOUBLE,rank+1,type,MPI_COMM_WORLD,&status);
         }
         if (rank>0) {
            MPI_Send(&local_temp[1],1,MPI_DOUBLE,rank-1,type,MPI_COMM_WORLD);
            MPI_Recv(&local_temp[0],1,MPI_DOUBLE,rank-1,type,MPI_COMM_WORLD,&status);
         }
      }
      else {
         if (rank>0) {
            MPI_Recv(&local_temp[0],1,MPI_DOUBLE,rank-1,type,MPI_COMM_WORLD,&status);
            MPI_Send(&local_temp[1],1,MPI_DOUBLE,rank-1,type,MPI_COMM_WORLD);
         }
         if (rank<numprocs-1) {
            MPI_Recv(&local_temp[local_end_point+1],1,MPI_DOUBLE,rank+1,type,MPI_COMM_WORLD,&status);
            MPI_Send(&local_temp[local_end_point],1,MPI_DOUBLE,rank+1,type,MPI_COMM_WORLD);
         }
      }
      for (j=1;j<=local_end_point;j++) {
         local_temp_hld[j]=(local_temp[j-1]+local_temp[j+1])*.5;
      }
      for (j=1;j<=local_end_point;j++) {
         local_temp[j]=local_temp_hld[j];
      }
   }
         <-------------- End point for analysis

   MPI_Finalize();
}
```



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

for numprocs > 2

$$T_{comp} = \frac{\text{global\_num\_points}}{\text{num\_procs}} \bullet 2 \text{ floating point operations}$$

$$T_{comm} = 4(T_{startup} + (1)T_{data}) \quad \text{for numprocs} > 2$$

$$T_{comm} = 2(T_{startup} + (1)T_{data}) \quad \text{for numprocs} = 2$$

Therefore for numprocs > 2

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

for numprocs = 2

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

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.