Name KEY Omitted Problem Number _____

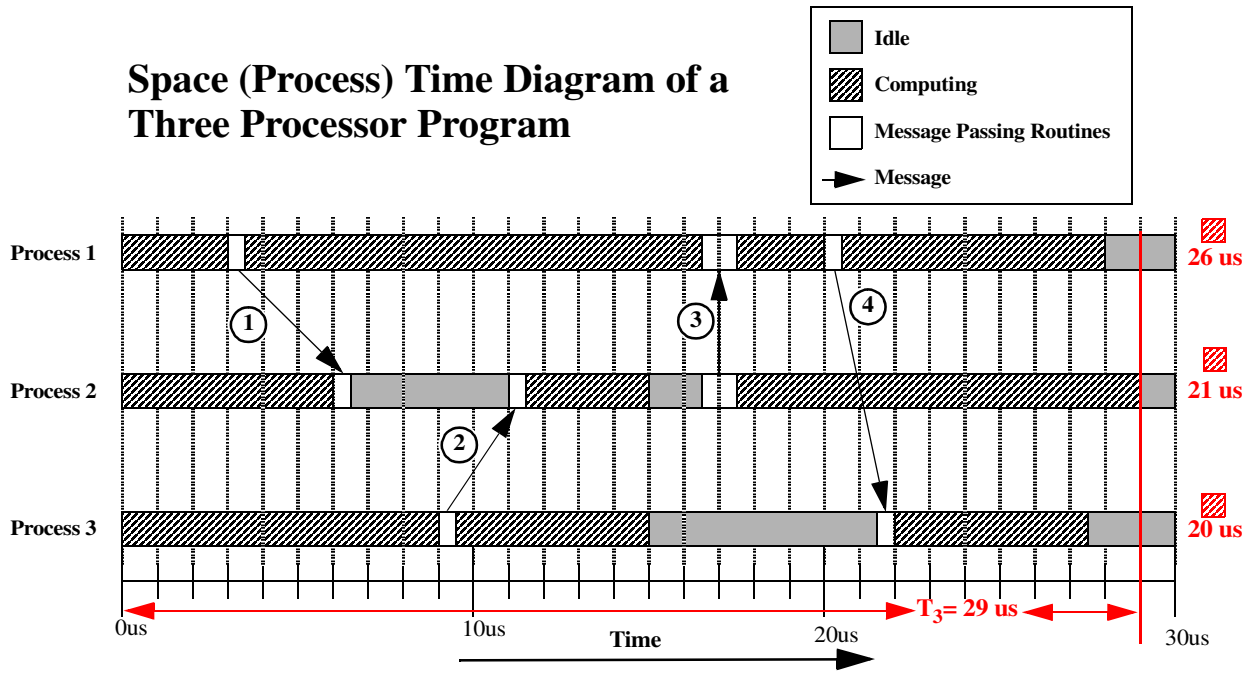
Class _____

CPE 412/512 Exam 1

Fall Semester 2008

INSTRUCTIONS: **All students must complete the first problem on this exam. Students may then chose one additional problem to omit from the set of problems numbered 2 through 6. All graduate students must work problem 7 in addition to the other 5 problems on the exam. Undergraduate students are not assigned problem 7. Problems are equally weighted. This is a closed-book examination.**

1. 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.

$T_1 = \Sigma T_{comp}$ = 26 us + 21 us + 20 us = 67 us

T_3 = Max($T_{overall\ execution\ time\ on\ each\ processor}$) = Max(28 us, 29 us + 27.5 us = 29 us

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

$S_3 = T_1/T_3$ = 67 us /29 us = 2.31

$C_3 = T_p * p = T_3 * 3$ = 29 us * 3 = 87 us

$E_3 = T_1/(T_3\ p)$ = 67 us /(29 us * 3) =0.77

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

Synchronous Numbers 3
Other (locally blocking or nonblocking) Numbers 1, 2, and 4.

2. A parallel program has been created to execute on a four processor system. In this program the original computational workload is distributed in the following manner: 20% is assigned to processor 1, 24% to processor 2, 29% to processor 3, and 27% to processor 4. In addition, the parallelization process has led to an additional 2% extra computation that must be performed by each of the processors (i.e. the parallel representation incurs a total overhead of 8% extra computations but this overhead is distributed evenly among the four processors). Neglecting the communication overhead and synchronization effects associated with the parallel implementation, what is the speedup, and efficiency when it is compared to a functionally equivalent sequential implementation. If the run time of the best sequential version of the program is 100 seconds what is the cost of this four processor implementation?

T_4 = parallel execution time on 4 processors

T_1 = sequential execution time on 1 processor = 100 seconds

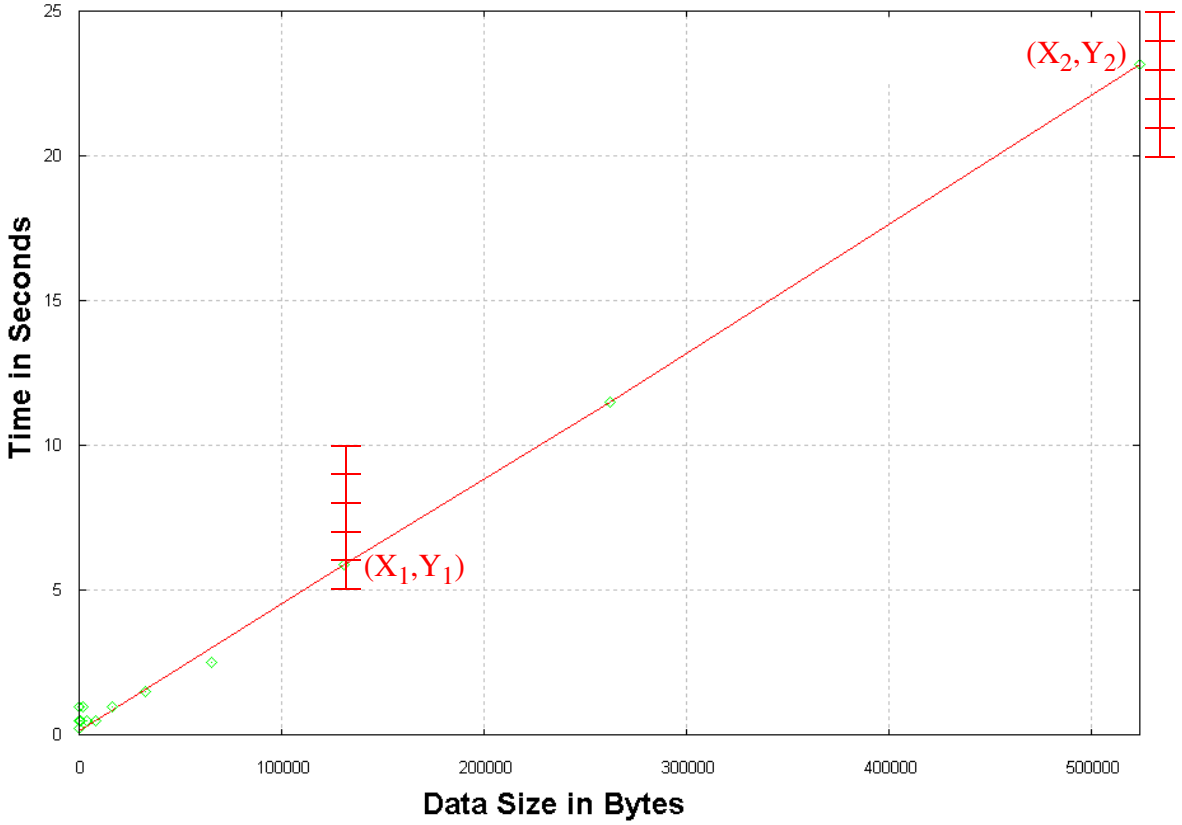
$T_4 = \text{Max}[(.20+.02)*T_1,(.24+.02)*T_1, (.29+.02)*T_1, (.27+.02)*T_1] = .31*T_1$

$S_4 = T_1/(0.31*T_1) = 3.22$ 

$E_4 = S_4/p = 3.22/4 = 0.81$ 

$C_4 = p * T_p = 4 * .31*T_1 = 4 * 31$ seconds = 124 seconds (assuming K=1) 

3. 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.




```
#include <stdio.h>
#include <string.h>
#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 processors*/
   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) printf("%d bytes  %e seconds\n",data_size,tm);
  }
   /* 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 $Y_1 = 23.1$ and $Y_2=5.95$ were estimated from the graph.

$$(X_1, Y_1) = \left(524288, \frac{23.1}{(2 \bullet 1000)}\right) = (524288, 0.01155) \text{ bytes, seconds}$$

#number of Send/Rec Pairs between timing points

Send/Receive Pair

$$(X_2, Y_2) = \left(262144, \frac{5.95}{(2 \bullet 1000)}\right) = (131072, 0.002975) \text{ bytes, seconds}$$

#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

$$T_{data} = \frac{Y_1 - Y_2}{X_1 - X_2} = \frac{0.01155 - 0.002975}{524288 - 131072} = \frac{0.008575}{393216} = 2.18 \times 10^{-8} \text{ seconds/byte}$$

$$T_{startup} = T_{comm} - T_{data} W \approx 1.17 \times 10^{-4} \text{ seconds}$$

Y_1 = 0.01155

X_1 = 524288

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

4. Calculate the communication time T_{comm} that is associated with the following scatter type routine that was implemented using point-to-point MPI communication routines. Determine a general expression that is a function of the number of processors, *numprocs*, the data size variable, *data_size*, and the $T_{startup}$, and T_{data}. Assume that the data size is evenly divisible by the number of processors, the number of processors is always a power of two, 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)..

```
float * scatter(int data_size,float *numbers,int rank,int numprocs)
{
   MPI_Status status;
   int i,pe,tag,msg_size;
   int active_bit_pos,bit_pos_mask;

   tag = 234;
   active_bit_pos = 1; // active_bit_pos = 00000000...00000001 binary
   bit_pos_mask = (~1);// bit_pos_mask   = 11111111...11111110 binary
   msg_size = data_size/2;
   for(i=0;i<log2((float) numprocs);i++) {
      if(!(bit_pos_mask&rank)) {
         pe=rank^active_bit_pos;
         /* receive value from other nearest neighbor pe */
         if (rank&active_bit_pos) {
            MPI_Recv(numbers,msg_size,MPI_FLOAT,pe,tag,
               MPI_COMM_WORLD,&status);
         }
         /* send value to other nearest neighbor pe */
         else {
            MPI_Send(&numbers[msg_size],msg_size,MPI_FLOAT,
                  pe,tag,MPI_COMM_WORLD);
         }
      }
      msg_size /= 2; // decrease message size by a factor of two
      active_bit_pos <<= 1;
      bit_pos_mask <<= 1;
   }
   return numbers;
}
```

$$T_{comm} = \sum_{i=1}^{\log_2(numprocs)} \left(T_{startup} + \frac{data\_size}{2^i} T_{data} \right)$$

$$T_{comm} = \log_2(numprocs) T_{startup} + \frac{data\_size \bullet T_{data}}{2} \sum_{i=0}^{\log_2(numprocs)-1} \frac{1}{2^i}$$

where by the geometric series

$$\sum_{i=0}^{\log_2(numprocs)-1} \frac{1}{2^i} = \frac{1 - 2^{-\log_2(numprocs)}}{1 - 2^{-1}} = 2\left(1 - \frac{1}{numprocs}\right) = 2\frac{(numprocs - 1)}{numprocs}$$

$$T_{comm} = \log_2(numprocs) T_{startup} + data\_size \bullet T_{data} \frac{(numprocs - 1)}{numprocs}$$

5. 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 MIPS, and all the processors are assumed equally capable.

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

(b) If $p = 16$ and $\Delta = 4$ MIPS, determine the value of f which will yield a system performance of 40 MIPS.

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

(a)

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

If we define

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

Then

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

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

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

for $p = 16, \Delta = 4$ MIPS, $\Delta_{eff} = 40$ MIPS,

$$f = \frac{p(\Delta_{eff} - \Delta)}{(p-1)\Delta_{eff}} = \frac{16(40-4)}{15(40)} = 0.96$$

96% of the code must be parallelizable!
only 4% can be sequential

6. What is meant by the term *collective communication*. What are the differences between a *broadcast*, *gather*, and *scatter* operations?

Collective communication are communication routines that generally employ a set of processes to achieve a desired communication operation such as a broadcast, scatter, or gather operation. This set of processes can be generally expanded to a value greater than two.

broadcast operations send a data item from a single source to all the other processes in the group.

gather operations combine together in a single data structure on a root process the data elements present in each of the other processes in the group.

scatter operations distribute portions of the data present in a single data structure that is on the root process to the other processes in the group.

What are the fundamental difference between shared memory and message passing systems/paradigms?

Shared memory assumes a global address space where all memory elements can be accessed by any processor by applying the appropriate global address where as message passing systems usually incorporate an address space that is localized from the point of view of each processor.

From a programmer's point of view major differences include:
Shared memory synchronization and communication is often expressed using the global memory address of the shared object.
Message passing synchronization and communication is most often expressed using messages not the memory address of shared objects. Point-to-point communication between processors requires that processors know and explicitly specify the label number associated with the other processors in the system. The memory address of objects is most often assumed to be local to that process.

The startup latency of many shared memory systems is considerably less than many message passing systems.

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

MPMD is the most general way to program a MIMD type architecture. In this paradigm the programmer must write a separate program for each process that is to execute on a targeted processing element of a multi-core system.

SPMD is a programming paradigm where there is only a single program that is created. This program is then replicated and executed on each process that makes up the system. The SPMD program is designed in such a way that it operates on different data segments of the problem and simultaneously executes different sections of the program depending upon the current process id (rank) and the number of processes being used to solve the problem.

MPMD is most general and may be easier to adapt to dynamic process creation. SPMD is easier to write and debug scalable code.

Graduate Student Problem:

7. Describe how you would implement a general parallel program that would approximate the following integral that computes the value of pi using standard Monte Carlo techniques. Describe the steps and issues involved, use pseudo code where necessary.

$$\pi = 4\int_0^1 \sqrt{1-x^2}\,dx$$

process id => ***rank*** number of processors => ***p*** Number of sample points => ***N***

Only one method is required for this problem!

Parallel Monte Carlo Method 1

Initialization Phase

broadcast value of ***N*** to all processors then generate random numbers using one of the following techniques

random number generation technique 1

generate 2****N*** random numbers between 0 & 1on processor ***p*** = 0

scatter random numbers between ***p*** processors

random number generation technique 2

on each processor seed random number using a function that is dependent on rank

generate 2****N/p*** random numbers between 0 &1 using these unique seeds

Embarrassing Parallel Phase

using your local pool of random numbers, *rand_nums[]*,

```
pt_sum = 0;
for (i=0;i<2*N/p;i=i+2) {
   x = rand_nums[i];
   y = rand_nums[i+1];
   if (y*y +x*x<=1) pt_sum = pt_sum + 1;
}
```

End Communication and Output Phase

reduce *pt_sums* to global sum *g_sum*

if (*rank* == 0) output (*g_sum**4/*N*);

Parallel Monte Carlo Method 2

Initialization Phase

broadcast value of ***N*** to all processors then generate random numbers using one of the following techniques

random number generation technique 1

generate ***N*** random numbers between 0 & 1on processor ***p*** = 0

scatter random numbers between ***p*** processors

random number generation technique 2

on each processor seed random number using a function that is dependent on rank

generate ***N/p*** random numbers between 0 &1 using these unique seeds

Embarrassing Parallel Phase

using your local pool of random numbers, *rand_nums[]*,

```
pt_sum = 0;
for (i=0;i<N/p;i=i+1) {
   x = rand_nums[i];
   pt_sum = pt_sum + sqrt(1-x*x);
}
```

End Communication and Output Phase

reduce *pt_sums* to global sum *g_sum*

if (*rank* == 0) output (*g_sum**(1-0)*4/*N*);

i

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

```
#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.