Name KEY

CPE 412/512 Exam I

Fall Semester 2009

1. Let *f* be the percentage of a program code which must be executed sequentially by a single processor p processor in a computer system. Assume that the remaining code can be executed in parallel in an evenly divisible manner on all *p* processors. Also assume that 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.

$$\text{Computation Rate} = \frac{\text{Total Instructions Performed}^*}{\text{Time it takes to Execute these Instructions}} = \Delta$$

$$\text{Total Number of Instructions Executed by Application}^* = W$$

*note: in units of million

Workload

$$\text{Sequential Execution Time} = T_s = \frac{W}{\Delta} \quad (1)$$

Single Processor Computation Rate in MIPS

Effective Compuation Rate using p indentical processors $= \Delta_{eff_p}$

$$\text{Parallel Execution time} = T_p = \frac{W}{\Delta_{eff_p}} \quad (2)$$

Assumption: No additional workload is added to the algorithm when it is executed in parallel

also by Amdahl's Law

$$T_p = fT_s + \frac{(1-f)T_s}{p} \quad (3)$$

where *f* is the serial portion

Solving for Δ_{eff_p} in (2) and substituting for T_p from (3)

$$\Delta_{eff_p} = \frac{W}{T_p} = \frac{W}{fT_s + \frac{(1-f)T_s}{p}} = \frac{Wp}{(fp + (1-f))T_s}$$

Then Substituting for T_s from (1)

$$\Delta_{eff_p} = \frac{Wp}{(fp + (1-f))\frac{W}{\Delta}} = \frac{p\Delta}{(fp + (1-f))}$$

note: an equivalent expression is:

$$\Delta_{eff_p} = S_p\Delta = pE_p\Delta$$

(b) If p = 16 and $\Delta = 28$ MIPS, determine the value of f which will yield a system performance of 300 MIPS.

$$\Delta_{eff_p} = \frac{p\Delta}{(fp + (1-f))} \rightarrow fp + (1-f) = \frac{p\Delta}{\Delta_{eff_p}} \rightarrow f(p-1) + 1 = \frac{p\Delta}{\Delta_{eff_p}}$$

$$f = \frac{p\Delta - \Delta_{eff_p}}{(p-1)\Delta_{eff_p}}$$

In this problem:

$$f = \frac{(16)(28\text{MIPS}) - 300MIPS}{(16-1)300MIPS} \approx 0.0329$$

2. A parallel program has been created that can be comprised of an instructional workload that can be divided in a manner that is dictated by the underlying algorithm into four distinct sections. Due to data dependencies, Section 1 must execute first, followed in sequence by Section 2. Section 3 must execute after Section 2, and finally Section 4 must execute after Section 3. The first portion of the code (Section 1) is composed of 5% of the total instructional workload and can only be executed on a single processor no matter how many processors are available in the system. The next section (Section 2) of the code represents 15% of the total workload which can be evenly assigned to execute on up to 2 processors. The next section (Section 3) represents 25% of the total instructional workload which can be evenly assigned for execution on up to 4 processors. The remaining 55% of the instructional workload (Section 4) can be evenly assigned to execute on an unbounded number of processors (as many processors as are available).

Neglecting the effects of communication delay and synchronization delay, develop a general expression for speedup (assuming a homogeneous processing system where each processor is identical to one another). This expression should be a function of the number of processors present in the system and the workload distribution as described above.

$$T_p = \frac{W \bullet 0.05}{\Delta} + \frac{W \bullet 0.15}{2\Delta} + \frac{W \bullet 0.25}{4\Delta} + \frac{W \bullet 0.55}{p\Delta}$$

$$= \frac{W}{\Delta}\left(0.05 + 0.075 + 0.0625 + \frac{0.55}{p}\right) = \frac{W}{\Delta}\left(0.1875 + \frac{0.55}{p}\right) \quad \text{for } p \geq 4$$

$$T_1 = \frac{W}{\Delta}$$

$$S_p = \frac{T_1}{T_p} = \frac{W/\Delta}{(W/\Delta)(0.1875 + 0.55/p)} = \frac{p}{0.1875p + 0.55} \quad \text{for } p \geq 4$$

What is the maximum speed up of the parallel program as the number of processors is increased without bound?

$$Max(S_p) = \lim_{p \to \infty} \frac{p}{0.1875p + 0.55} = \lim_{p \to \infty} \frac{1}{0.1875} \approx 5.33$$

If it were possible to improve the performance (decrease the execution time) of only <u>one</u> of the sections of the parallel program by a factor of 50%, which section should be chosen to obtain a maximum speedup for a 16 processor system? What would be this speedup?

$$Old(T_{16}) = \frac{W}{\Delta}(\overset{\text{section 1}}{0.05} + \overset{\text{section 2}}{0.075} + \overset{\text{section 3}}{0.0625} + \overset{\text{section 4 (p=16)}}{0.034375}) \qquad Old(T_1) = \frac{W}{\Delta}$$

Largest Impact on Time

Choosing Section 2 because this has the maximum Impact on Parallel Execution Time

$$New(T_{16}) = \frac{W}{\Delta}(0.05 + 0.075 \bullet 0.5 + 0.0625 + 0.034375) = \frac{W}{\Delta}0.184375$$

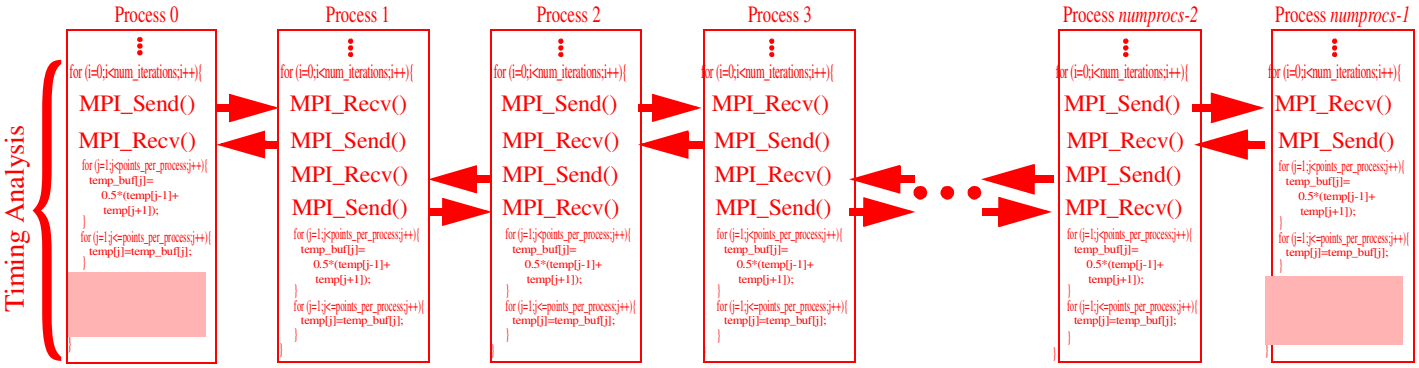
$$New(T_1) = \frac{W(1 - 0.15 \bullet 0.5)}{\Delta} = 0.925\frac{W}{\Delta} \qquad New(S_{16}) = \frac{0.925(W/\Delta)}{0.184375(W/\Delta)} \approx 5.02$$

3. 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. 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+2];
   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],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 (j=1;j<=points_per_process;j++) {
         temp_buf[j]=0.5*(temp[j-1]+temp[j+1]);
      }
      for (j=1;j<=points_per_process;j++) {
         temp[j]=temp_buf[j];
      }
   }
}
```



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

for numprocs > 2

$$T_{comp} = \frac{\text{total\_points}}{\text{numprocs}} \bullet 2 \text{ floating point operations}$$

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

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

Therefore for numprocs > 2

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

for numprocs = 2

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

4. The following OpenMP code fragment is based on the distributed add number routine that was discussed extensively in class. (Threads in this program are assigned segments of the global list of numbers in the ***numbers[]*** array to add in a local manner with the results being accumulated in the global sum variable ***g_sum***. The program works correctly for the specified number of threads but does not exhibit good performance characteristics.

```
   // use high resolution timer
   TIMER_CLEAR;
   TIMER_START;

   /* Fork a team of threads giving them their own copies of local */
   /* variables */
   int num_threads, tid,st_index,nm_ele,l_sum;
   #pragma omp parallel private(num_threads,tid,st_index,nm_ele,l_sum)
   {
      l_sum=0; // clear local sum
      tid = omp_get_thread_num(); //obtain thread number
      num_threads = omp_get_num_threads();
      nm_ele=list_size/num_threads;
      st_index=tid*nm_ele;

      /* Only master thread does this */
      if (tid == 0) {
         printf("Number of threads = %d\n", num_threads);
      }
      #pragma omp critical
      {
         for (i=st_index;i<st_index+nm_ele;i++) {
            l_sum+=numbers[i];
         }

         g_sum += l_sum;
      }

   }  /* All threads join master thread and disband */

   // stop recording the execution time
   TIMER_STOP;

   /* output global sum */
   printf("Sum of numbers is %e\n",g_sum);
   printf("time=%e seconds\n\n",(double) TIMER_ELAPSED/1000000.0);
}
```

Move for loop

Why is this the case? Identify one or more changes in the code that should improve performance without altering the correctness of the result. Clearly state what was the source of the performance bottleneck.

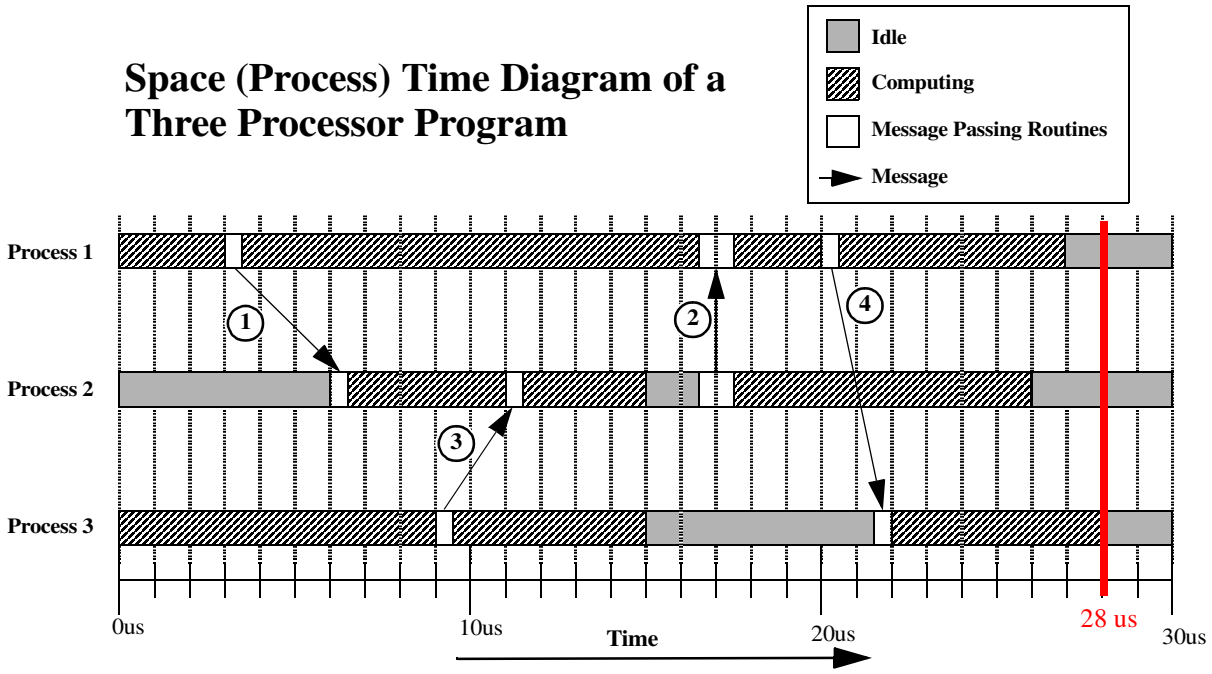
The computation of the local sum (l_sum) is part of the critical section. This means that only one thread at a time can compute it which effectively serializes the processing of the code because the last thread must wait until all the other threads have executed.

Suggested change is to move the loop

```
for (i=st_index;i<st_index+nm_ele;i++) {
   l_sum+=numbers[i];
}
```

out of the critical region by placing it above the "#pragma omp critical" statement.

5. 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.

$$T_1 = T_{com_{proc1}} + T_{com_{proc2}} + T_{com_{proc3}} = 25us + 16.5us + 20.5us = 62us$$

$T_3 = 28us$ schedule completion time of most busy process

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

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

Synchronous -- Label 2

$$S_3 = \frac{T_1}{T_3} = \frac{62\text{us}}{28\text{us}} \approx 2.21 \quad E_3 = \frac{S_3}{3} \approx \frac{2.21}{3} \approx 0.738 \quad C_3 \propto T_3 \bullet 3 \approx 28(3) \approx 84$$

Locally blocking -- Labels 1, 3, and 4.

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.

$$Avg(t_{comp}) = \frac{62}{3} \approx 20.67$$

$$Avg(t_{comm}) = \frac{(0.5 + 1 + 0.5 + 0.5 + 0.5 + 1 + 0.5 + 0.5)}{3} \approx \frac{5}{3} \approx 1.67$$

$$Avg(\frac{t_{comp}}{t_{comm}}) \approx \frac{20.67}{1.67} \approx 12.38$$

e. Identify which processor has the greatest computational load and which processor presents the greatest bottleneck to the performance.

Processor 1 has the greatest computational load at 25 us.

Processor 2 has the greatest idle time/communication time component (i.e. does the least amount of useful computation).

To improve performance we need to take some computation off processor 1 and assign it to processor 2 in a manner that does not significantly increase communication and idle time on any of the other processors.

6.a For the following **pthread** code what is (are) the possible value(s) of *num* if there are no assumptions regarding the targeted system or the manner in which the ***pthread*** scheduler

will operate?

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

int num=1;
void *thread1(void *dummy) {
   num = num + 1;
}
void *thread2(void *dummy) {
   num = num * 3;
}
void *thread3(void *dummy) {
   num = 123;
}

int main(int argc, char *argv[]) {
   pthread_t thread1_id,thread2_id,thread3_id;
   pthread_create(&thread1_id,NULL,thread1,NULL);
   pthread_create(&thread2_id,NULL,thread2,NULL);
   pthread_create(&thread3_id,NULL,thread3,NULL);
   printf("num = %d\n",num);
}
```

Original Program Order

Possible orders of num variable updates

Thread 1, Thread 2, Thread 3 → num=123

Thread 1, Thread 3, Thread 2 → num=369

Thread 3, Thread 1, Thread 2 → num=372

Thread 3, Thread 2, Thread 1 → num=370

Thread 2, Thread 3, Thread 1 → num=124

Thread 2, Thread 1, Thread 3 → num=123

But note that this is the detached thread model. The main thread may output the value of num and terminate before the other threads complete their operations! Some additional values of num could be
num=1; -- main thread finishes before the others
num=6; -- thread 1, thread 2, main thread terminates
num=2; -- thread 1, main thread terminates
num=3; -- thread 2, main thread terminates
num=4; -- thread 2, thread 1, main thread terminates

Is the output deterministic in nature? Explain why this may or may not be the case.

The output is not deterministic in the sense that we can accurately predict the result of the variable num before we run the code. There are several possible values of num depending upon the run time conditions and the thread scheduling methodology employed. This is complicated by the fact that the threads operate in a detached manner and there is no synchronization through a thread join operation.

b) Apply Bernstein's conditions for parallelism to the following functions that are labeled S1, S2, S3, S4, and S5 in program order.

S1: C = function1(D,E)

S2: M = function2(G,C)

S3: A = function3(B,C)

S4: C = function4(L,M)

S5: F = function5(G,E)

Considering these functions on a pair-wise basis, which functions is it possible for parallel execution?

Functions	$I_2 \cap O_1$ **True Data Dependencies**	$I_1 \cap O_2$ **Anti Data Dependencies**	$O_1 \cap O_2$ **Output Data Dependencies**	Results of applying Bernstein's Conditions
function1 & function2	C	ϕ	ϕ	**function1 must execute before function2**
function1 & function3	C	ϕ	ϕ	**function1 must execute before function3**
function1 & function4	ϕ	ϕ	C	**function1 must execute before function4**
function1 & function5	ϕ	ϕ	ϕ	**function1 and function5 are able to execute in parallel**
function2 & function3	ϕ	ϕ	ϕ	**function2 and function3 may be able to execute in parallel**
function2 & function4	M	C	ϕ	**function2 must execute before function4**
function2 & function 5	ϕ	ϕ	ϕ	**function2 and function5 are able to execute in parallel**
function3 & function4	ϕ	C	ϕ	**function3 must execute before function4**
function3 & function5	ϕ	ϕ	ϕ	**function3 and function5 are able to execute in parallel**
function4 & function5	ϕ	ϕ	ϕ	**function4 and function5 are able to execute in parallel**

What is the maximum number of functions that can execute in parallel? What are they?

Three, functions 2,3, and 5 all could execute in parallel

7.a What does the term cache coherency mean?

Cache coherency is the property that insures that access to a variable that is present in the local cache will always produce the latest copy of the variable even if this variable is also present in another cache. A cache coherency protocol is needed to insure that this is the case.

Define what is meant by the term false sharing in multiprocessing caches and describe how this can adversely affect performance in a multiprocessor system.

False sharing can occur when two separate data items reside close in memory and are assigned to the same cache block (or line). If one of the variables is to be repetitively updated by one processor that have a local cache and the other by another local-cache processor then though the processors are not writing to the same memory variable location, the cache on the other processor has to either be updated or invalidated because it also has a copy of the variable that has been updated. This can result in a large amount of waisted time being spent re-fetching entire blocks on each processor because the other processor is updating its adjacent variable.

7.b Define and compare, locks, semaphores, monitors, and condition variables in terms of their ability to be used to control shared data access and mutual exclusion. Specify at least one advantage and one disadvantage of each construct.

locks

A lock is a one bit shared variable that is used to indicate when a process (or thread) has entered the critical region. Processes (or threads) have the capability to set or reset the lock. The lock will be set before a process or thread enters the critical section and reset when the process or thread exits this section. Most lock variable implementations support atomic operations where the lock can be checked to see if it is in use and then set if it is not in use as one un-iterruptable operation. Locks are most often implemented using a spin lock mechanism where a lock is continuously examined by a particular process or thread until is become available. Major advantage: Simplicity of use and implementation. Locks are usually supported by at least one low-level machine language instruction. Major disadvantage: Locks that utilized busy waiting techniques can be very inefficient in terms of the overhead that they produce.

semaphores

Semaphores are shared positive integers (including zero) that support the P (decrement) and V (increment) operations. The P operation on semaphore, s, waits until s is greater than 0 and then decreases s by 1 and allows the process to continue. the V operation increments the semaphore variable, s, to release any waiting processes (or threads) if any. Before use semaphores must be initialized to a number of 1 are greater. If they are initialized to a value of 1 then they are called binary semaphores. Major advantage: Semaphores are implemented in the operating system and utilize thread or process scheduling techniques to eliminate the need for busy waiting (i.e. they will suspend the thread or process until the semaphore has a value greater than 0 and then place the suspended entity in the ready queue). Major disadvantage: the requirement for operating system support and the lack of support in all environments.

monitors

A monitor is a suite of procedures that are often treated as a single object (in which the data and the operations that operate upon the data are encapsulated) that provides the only method to access a shared resource. The monitor is written such that only one process at a time can access a procedure that reads/writes data to the shared structure. Major advantage: Easier for the programmer to implement than locks or semaphore, higher level abstraction, less error prone. Major disadvantage: Requires language or advanced operating system support. Not available in many environments.

condition variables

Condition variables are shared memory entities that allow a thread or a process to wait for an event to complete. Unlike locks or semaphores, one cannot retrieve a value or store a value into a condition variable. Instead these variables indicate that a global event has occurred. A process or thread can use raise a condition by signaling the specified condition variable is signaled. The operating system maintains of queue of processor or threads that are waiting on the condition variable whose execution can be restarted upon whenever the condition variable is signaled. Major Advantage: Condition variables allow one to avoid entering a busy waiting state by allowing threads and processes to signal each other about events of interest. Major disadvantage: Requires operating system/thread scheduling support.