Name KEY Omitted Problem Number (CPE 412 only) _____

Class _____

# CPE 412/512 Exam II

## Fall Semester 2010
## (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. Allowable items on desk include, pencils or pens and calculator.**

1. A parallel program has been implemented in a manner that exploits type 2 linear pipelining techniques. In this program, 8 homogeneous processors are used to implement 8 stages of the pipeline. Processor 1 reads data from the global input and processor 8 outputs the data to a global output. It is assumed that the computational workload is distributed in the following manner: 11% to processor 1, 13% to processor 2, 15% to processor 3, 11% to processor 4, 16% to processor 5, 13% to processor 6, 8% to processor 7, 13% to processor 8.

   a) Neglecting communication/io costs what is the pipeline speedup and efficiency, compared to a nonpipeline version for the following cases

$T_1$ is the time it takes to process $M$ items on 1 processor without pipelining

$T_1 = MT_s$ where $T_s$ is the time it takes to execute 1 complete item on a single processor and $M$ is the total number of items to be processed.

$T_8$ approximate time it takes to process $M$ items on 8 process with pipelining

$$T_8 \approx (8 + M - 1)max(0.11T_s, 0.13T_s, 0.15T_s, 0.11T_s, 0.16T_s, 0.13T_s, 0.08T_s, 0.13T_s)$$
$$\approx 0.16(7 + M)T_s$$

$S_8$ approximate speedup ratio to process $M$ items on 8 processors with pipelining compared to the nonpipelined case.

$$S_8 = \frac{T_1}{T_8} \approx \frac{MT_s}{0.16(7 + M)T_s} = \frac{6.25M}{(7 + M)}$$

$E_8$ approximate efficiency to process $M$ items on 8 processors with pipelining compared to the nonpipelined case.

$$E_8 = \frac{T_1}{8T_8} \approx \frac{0.78125M}{(7 + M)}$$

i) 1 data item is to be processed

from pipeline equations

$$S_8 \approx \frac{6.25(1)}{(7 + 1)} = 0.78125 \qquad E_8 \approx \frac{0.78125}{(7 + 1)} = 0.09765625$$

but if only one data item flows through there is not the bottleneck that is present when data from one stage is slowed down ultimately to that of the slowest stage -- in this case data simply trickles through at varying rates. Since there is no overhead with the pipelining then one would expect $T_8 = T_1$, resulting in

$$S_8 = \frac{T_1}{T_8} = 1.0 \qquad E_8 = \frac{S_8}{8} = \frac{1}{8} = 0.125$$

ii) 10 data items are to be processed

$$S_8 \approx \frac{6.25M}{(7+M)} = \frac{6.25(10)}{(7+10)} \approx 3.68 \qquad E_8 \approx \frac{0.78125M}{(7+M)} = \frac{0.78125(10)}{(7+10)} \approx 0.46$$

iii) 100 data items are to be processed

$$S_8 \approx \frac{6.25M}{(7+M)} = \frac{6.25(100)}{(7+100)} \approx 5.84 \qquad E_8 \approx \frac{0.78125M}{(7+M)} = \frac{0.78125(100)}{(7+100)} \approx 0.73$$

b) If processing a total of 1000 data items in sequence costs $200 of CPU time when the problem is implemented on a single CPU without pipelining, then what is the expected cost of processing 1000 data items on the 8 stage pipeline representation discussed in part a of this problem?

$$\text{cost}_{\text{single CPU}} = kT_1 = kMT_s = k(1000)T_s = \$200$$

$$k = \frac{0.2}{T_s}$$

$$\text{cost}_{\text{8 pipelined CPUs}} = kT_8p = k(0.16(7+M)T_s)8$$

$$= \left(\frac{0.2}{T_s}\right)(0.16(7+1000)T_s)8 = \$257.79$$

2. a) What are at least three major general differences between pThreads and OpenMP, in terms of language constructs, thread management responsibilities, and philosophical considerations?

Many answers possible. Here are some of the differences

Language Constructs:

Both OpenMP and pThreads utilize library function calls but OpenMP also makes extensive use of compiler directives. This requires that the compiler itself be modified to handle OpenMP pragmas. The advantage of this approach is that OpenMP can allow certain optimizations to be handled automatically in a fine grained manner.

Thread management responsibilities:

pThreads requires that the user explicitly execute commands that create threads and join threads. OpenMP these actions are usually implicit being part of a higher level construct.

Philosophical considerations:

OpenMP attempts to abstract away some of the lower-level aspects of thread management, synchronization, and control. pThreads requires that the user programmer take into account many of the low-level activities such as thread creation and destruction. Loop level parallelism and other fine grain parallel optimizations is better supported in OpenMP, whereas pThreads is much more flexible allowing for easier dynamic spawning of threads to meet the needs of a nondetermistic application. Such a spawning scheme does not have to adhere to the fork-join model. (The task construct in OpenMP is designed to provide this feature but is somewhat difficult to implement).

b) What are at least the major differences between the MPI message passing environment and the OpenMP shared memory environment in terms of process/thread creation, data transfer/synchronization, and memory space utilization?

Process/thread creation:

In MPI this is usually performed at run time with the number of processes not changing during the scope of the application. In OpenMP the number of threads is easily changed dynamically as the program executes.

Data Transfer/synchronization:

In MPI data is transferred through message passing. In OpenMP data is transferred using shared memory, with the appropriate synchronization mechanism such as creating critical sections.

Memory Space Utilization:

In MPI data is local to the memory space of the process. No global data is allowed. In OpenMP data can be global to the entire program or thread local.

c) Why would one ever consider utilizing a hybrid message-passing/shared memory approach to implement a real-world application? Why not just use a single paradigm?

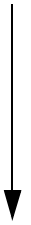
MPI is usually better suited to expressing parallelism across the cluster because its paradigm is closer to the underlying message-passing based socket communication paradigm. and OpenMP is better suited to exploit the multicore nature of each node in the cluster.Other reasons include: OpenMP exhibits good usage of shared memory system resources within a node (memory, latency, and bandwidth). It avoids the extra communication overhead with MPI within node.OpenMP adds fine granularity and allows increased and/or dynamic load balancing. Some problems naturally have two-level parallelism. Could have better scalability than both pure MPI and pure OpenMP.

3. a) Clearly state Bernstein's conditions:

Bernstein's conditions are a set of conditions which must exist if two (or more) processes can execute in parallel. If the input sets of two processes are independent of each other's output sets, and if the two output sets are independent these processes can execute in parallel.

Apply Bernstein's conditions for parallelism to the following five functions that are executed in the original program in sequential order starting from the top to the bottom.

```
B = function1(A,F); // statement 1
A = function2(H);   // statement 2
C = function3(B,C); // statement 3
E = function4(A,B); // statement 4
C = function5(D);   // statement 5
```

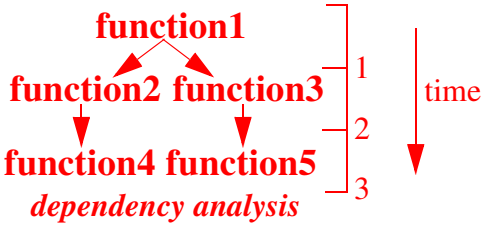


Original Sequential Program Order

b) Considering these functions on a pair-wise basis, list the sets of functions that can execute in parallel to one another a preserve sequential consistency (not violating Bernstein's Conditions).

| 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 | Notes |
|---|---|---|---|---|---|
| **function1 & function2** | $\phi$ | A | $\phi$ | **function1 must execute before function2** | |
| **function1 & function3** | B | $\phi$ | $\phi$ | **function1 must execute before function3** | |
| **function1 & function4** | $\phi$ | $\phi$ | $\phi$ | **function1 and function4 may be able to execute in parallel** | **turns out because of transitivity this is actually not possible** |
| **function1 & function5** | $\phi$ | $\phi$ | $\phi$ | **function1 and function5 may be able to execute in parallel** | **turns out because of transitivity this is actually not possible** |
| **function2 & function3** | $\phi$ | $\phi$ | $\phi$ | **function2 and function3 may be able to execute in parallel** | |
| **function2 & function4** | A | $\phi$ | $\phi$ | **function2 must execute before function4** | |
| **function2 & function 5** | $\phi$ | $\phi$ | $\phi$ | **function2 and function5 may be able to execute in parallel** | |
| **function3 & function4** | $\phi$ | $\phi$ | $\phi$ | **function3 and function4 may be able to execute in parallel** | |
| **function3 & function5** | $\phi$ | $\phi$ | C | **function3 must execute before function5** | |
| **function4 & function5** | $\phi$ | $\phi$ | $\phi$ | **function4 and function5 may be able to execute in parallel** | |

c) In a homogenous parallel processing system where all functions require the same time to complete, what is the maximum speedup that could be obtained while adhering to Bernstein's conditions and any implied transitivity relationships between functions?




$$max(S_p) = \frac{T_1}{T_p} = \frac{5}{3} \approx 1.67$$

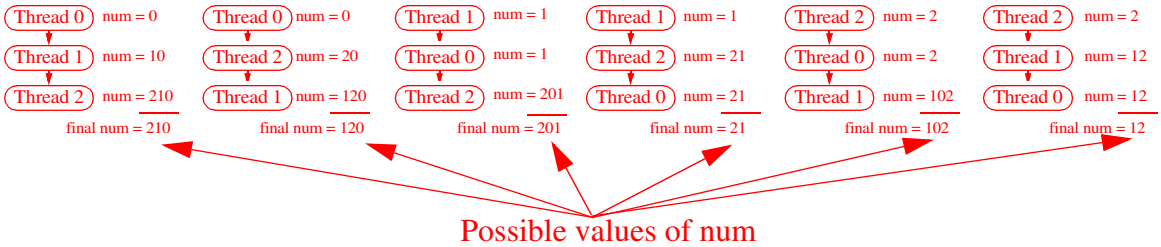
Assuming ***p*** is as large as needed (2 in this case)

4. For the OpenMP code fragment shown below, list the value or set of possible values for the variable *num* that would be outputted by the *cout* statement in the program. Assume that the *thread_function()* routine has a non predictable execution time (it can vary every time it is executed).

```
#define NM_THREADS 3
int main(int argc, char *argv[]) {
   double fun_out;
   int num=0,prod=1;
                                   ⋮
   omp_set_num_threads(NM_THREADS); // set the number of threads
                                   ⋮
   int thread_id;
   double fun_out;
   #pragma omp parallel private(thread_id,fun_out)
   {
      thread_id = omp_get_thread_num(); // get thread id
      fun_out = thread_function(...);
                                   ⋮
      // critical section
      #pragma omp critical
      {
         num += thread_id*prod;
         prod *= 10;
      }
   }
                                   ⋮
   cout << "num = " << num << endl;
}
```

Value(s) of num?

Note that there are three threads and the time that each thread finishes is not predictable before the program executes which makes it possible for the threads to get to the critical section in any order. Since there are 3 threads there are 3! = 6 possible orderings which are listed below. The result of the num variable is dependent upon the particular ordering as shown.



5. a) What are locks, semaphores, monitors, and condition variables in relationship to their ability to be used to control shared data access and mutual exclusion. What are the advantages and disadvantages of each of these constructs.

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 uniterruptable 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

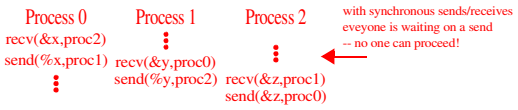
A monitor is a suite of procedures that are often treated as a single object (in which the data and the operations that operate upon the data are encapsulated) that provides the only method to access a shared resource. The monitor is written such that only one process at a time can access a procedure 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

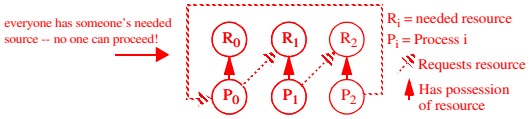
Condition variables are shared memory entities that allow a thread or a process to wait for an event to complete. Unlike locks or semaphores, one cannot retrieve a value or store a value into a condition variable. Instead these variables indicate that a global event has occurred. A process or thread 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.

b) In general what is *deadlock*? In a message passing program illustrate how deadlock could occur. In a similar manner illustrate in a shared memory program how deadlock could occur.

Deadlock occurs when two or more processes (or threads) simultaneously enter a wait state because they are waiting on an event that must be generated by another processes (or threads) within a group. The other process(es) (or thread(s)) cannot trigger this event because it are must wait for one or more other events to be triggered by other processes (or threads) within the group but the process(es) or thread(s) that must generate these events cannot do so because they are themselves suspended. Often these events are centered around the ability to access system resources.




Deadlock in Message Passing System




Deadlock in a Shared Memory System

c) What do the terms *embarrassingly parallel*, *divide and conquer*, and *partitioning* mean in the context of Chapters 3 and 4?

Embarrassingly parallel computation is computation that can obviously be divided into a number of completely independent parts, each of which can be executed by a separate process(or). No communication or very little communication between processes. Each process can do its tasks without any interaction with other processes