School of Arts & Sciences
Department of Computer Science & Mathematics
CSC 447—Parallel Programming for Multicore and Cluster Systems
Spring 2015

Final Examination—Solution, v1.0

Name:		**Student ID:**	
Signature:		**Duration:**	**75 minutes**

General Instructions

- There are 6 questions and 8 pages. Make sure that you have all of them.
- Exam questions are **NOT** sorted by order of difficulty. Scan the exam before you start and budget your time over the exam questions so you can maximize your grade.
- Your answers should be *brief* and *right to the point*. There is no need for essay answers! Use the back of the previous sheet if you need additional space.
- Your handwriting should be readable so it can be graded. You are liable to have points deducted from your grade if your handwriting is excessively difficult to decipher!
- The exam is a **closed** book, **closed** notes, and **closed** neighbor exam. **Any attempts at cheating or communicating with a neighbor will lead to expulsion from the exam!**

Question	Points	Score
1	21	
2	12	
3	20	
4	15	
5	17	
6	15	
Total:	100	

Multiple-Choice Questions

(21) 1. **Answer the following questions with a True or False. Justify the answer if it is false.**

(a) According to Amdahl's law, the maximum speed-up of a parallel computation given that 80% of the computation can be executed in parallel is 10.

(a) **False. Answer is 5**

(b) If we have an algorithm whose sequential complexity is $O(N)$ and we parallelize it with P processors, the maximum speedup we would expect is $N \log N$

(b) **False. We expect** N

(c) Flynn's taxonomy classifies computer systems into four categories (SISD, SPMD, MISD, MIMD) according to the number of instruction and data streams used.

(c) **False. SPMD should not be**

(d) Hypercube is the topology with the smallest diameter for a machine whose number of nodes is a power of 2.

(d) **False. diameter of hypercube**

(e) Whenever *producers* and *consumers* use locks, the producer thread must overwrite the shared buffer when the previous task has not been picked up by a consumer thread. The consumer threads must continue on picking up tasks until done. Individual consumer threads should pick up tasks one at at time.

(e) **False**

(f) If a program is running concurrently it is also running in parallel.

(f) **False**

(g) The main advantages for the shared memory parallel architecture are simplified programming and potential memory capacity. Some of the disadvantages are high cost and limited scalability

(g) **True**

(12) 2. **Select the appropriate answer.**

(a) Indicate which of the following phenomena contributed to the rise of multicore computing:

A. Limitations on cooling technologies
B. Limitations on software complexity
C. Limitations on on-chip bandwidth
D. Limitations on scalability of single core designs

(b) Which of the following is not an alternative for Shared Memory Multiprocessors.

A. Using sequential library routines with an existing sequential programming language
B. Using a sequential programming language with a parallelizing compiler
C. UNIX processes
D. Threads

(c) In programming shared address space, the following is true:

A. Threads assume all memory is global.
B. Threads assume all memory is local.
C. Memory manipulation is slower.
D. Process based models assume that memory is public and available to other processors.

(d) Which of the following is a valid reason for using threads.

A. Software Portability
B. Latency Hiding
C. Scheduling and Load Balancing
D. Ease of Programming
E. All of the above

(e) How many threads could be used for the computation below, each thread executing one or more of the instructions:

```
x++ ;
a = x + 2;
b = a + 3;
c++;
```

A. Two concurrent threads.
B. Three concurrent threads.
C. Four concurrent threads.
D. None of the above. The code is not parallelizable.

(f) Which of the following specifies that the variable is to be initialized in an OpenMP program.

A. private
B. firstprivate
C. shared
D. reduction

(g) If two threads execute the instruction $x{+}{+}$ where x is a shared variable initialized to 0, what are the possible values that x could have after the execution of the threads:

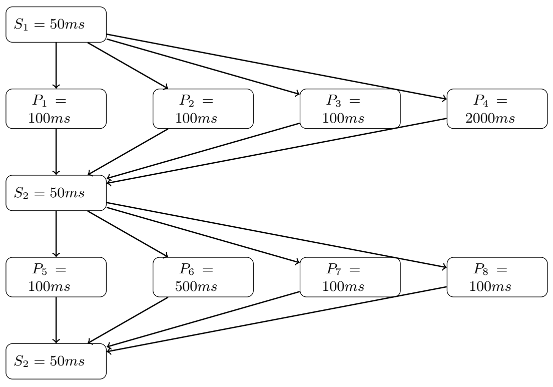
A. $x = 2$ if at different times and $x = 1$ if interleaved
B. $x = 1$ since threads will execute at different times
C. $x = 2$ since threads will be interleaved.
D. None of the above

(h) The only way to significantly increase the performance of microprocessors is to:

A. Improve power efficiency at about the same rate as the performance increase.
B. Use good compilers
C. Use faster chips

D. Use efficient algorithms

Parallel Performance



(5) 3. (a) Assuming a single worker thread (X10 NTHREADS=1) what is the runtime of this program?

Solution:
The single worker thread must execute all of the tasks serially, so T1 = 50 + (2000 + 100 + 100 + 100) + 50 + (100 + 500 + 100 + 100) + 50 = 3250ms

(5) (b) What is the speedup when X10 NTHREADS=8?

Solution:
The stages are now executed in parallel, with the duration of the stage determined by the slowest task: $T_8 = 50 + (2000) + 50 + (500) + 50 = 2650ms$, so the speedup is $\frac{T1}{T8}$ or or 1.226.

(5) (c) If each parallel task ($P_N = Xms$) were parallelized further to become two parallel tasks ($Q_N = \frac{X}{2}ms, R_N = \frac{X}{2}ms$), and again run with X10_NTHREADS=8, what would the runtime be? What is the speedup be relative to the previous run?

Solution:
The slowest task in the parallel stage is now twice as fast meaning: $T_8' = 50 + (\frac{2000}{2}) + 50 + (\frac{500}{2}) + 50 = 1400ms$, so the speedup is $\frac{T_8}{T_8'}$ or 1.89.

(5) (d) Why is the speedup not 2×?

Solution:
This is due to Amdahl's Law. To achieve a 2x overall speedup, the entire program would need to be sped up by a factor of two, but only a portion of the program was. The serial tasks were unchanged.

(15) 4. Parallelize Floyd's algorithm shortest all pairs shortest path using OpenMP. You can assume that matrix `A` is given as input and the number of nodes is `n`. Recall that the sequential Floyd's algorithm maybe implemented using the following code segment:

```
for (k=0; k<n; k++)
  for (i=0; i<n; i++)
    for (j=0; j<n; j++)
```

```
        if ( (d[i][k]+d[k][j]) < d[i][j] )
          d[i][j] = d[i][k]+d[k][j];
```

Solution:

```
 #pragma omp parallel default(shared) private(i,j,k)
{
for (k=0; k<n; k++)
  #pragma omp for
  for (i=0; i<n; i++)
    for (j=0; j<n; j++)
      if ( (d[i][k]+d[k][j]) < d[i][j] )
        d[i][j] = d[i][k]+d[k][j];
}
```

Comment: It is very important to use the private(i,j,k) clause, otherwise the OpenMP parallelization won?t work.

Multi-Core Programming using Pthreads

(17) 5. One of the OpenMP constructs that is the reduction operator. For instance, the dot product of two vectors can be computed as follows:

```
/* Compute and return the dot produce of vectors 'a' and 'b'. */
double vector_dot_product(double *a, double *b, int n)
{
   double dotproduct = 0;
   int i;

   #pragma omp parallel for reduction(+:dotproduct)

   for (i = 0; i < n; i++)
        dotproduct += a[i] * b[i];
  return dotproduct;
}
```

Show the above code using `pThreads`. Be sure to include all necessary synchronization code. To simplify the problem, you may assume that the number of threads `nT` is a constant, and that `n % nT == 0`. You may also assume that each parallel region spawns a new team of threads (though real-world OpenMP implementations reuse a pool of threads). Your code must be thread-safe, but must avoid unnecessary or excessive synchronization.

Solution:

```
/* A unit of work ? chunks of three vectors
* ?a?, ?b?, and ?sum? of equal length ?n?
*/
struct vector_chunk {
double *a, *b, dotproduct;
int n;
};
/* Compute partial dot product for a chunk of two vectors */
static void *dot_product_chunk(void *_chunk)
{
struct vector_chunk *chunk = _chunk;
int i;
double dotproduct = 0;
for (i = 0; i < chunk->n; i++)
dotproduct += chunk->a[i] * chunk->b[i];
chunk->dotproduct = dot_product;
return NULL;
}

/* Compute and return the dot produce of vectors 'a' and 'b'. */
double
vector_dot_product(double *a, double *b, int n)
{
int thread;
tid_t threadids[NT];
struct vector_chunk *chunk[NT];
// start NT-1 threads and assign a chunk of the vector
// dot product to each thread
for (thread = 0; thread < NT - 1; thread++) {
chunk[thread] = malloc(sizeof *chunk);
chunk[thread]->a = a + thread * n/NT;
chunk[thread]->b = b + thread * n/NT;
chunk[thread]->n = n / NT;
threadids[thread] =
thread_create(add_chunk, chunk[thread]);
}
// set up work unit for master thread
struct vector_chunk master_thread_chunk = {
.a = a + thread * n/NT,
.b = b + thread * n/NT,
.n = n / NT
};
dot_product_chunk(&master_thread_chunk);
double dotproduct = master_thread_chunk.dotproduct;
// wait until all threads have finished their work
for (thread = 0; thread < NT - 1; thread++) {
thread_join(threadids[thread]);
dotproduct += chunk[thread]->dotproduct;
free(chunk[thread]);
}
return dotproduct;
}
```

Multi-Core Programming using OpenMP

(15) 6. Write an OpenMP producer-consumer algorithm. The producer generates a set of random numbers and then adds them to the tail of the queue. The producer uses a work stealing algorithm where it would go through the head of the queue and retrieves one element at a time. The producer will then invoke a method `process` with the dequeued element as an argument. The algorithm repeats until the queue is empty and no random numbers are generated. There is no need to worry about the implementation of the `process` method. Furthermore, there is no need to worry about any methods declarations. A sequential version could look like:

```
void Producer(Queue *p) {
int myRand;

while (flag > 0)
{
     myRand = rand();
     enqueue(p, myRand);
     flag = (myRand > 1,0000,000)
}

// CONSUMER: Sum the data in A
double Consumer() {
int myRand;
Queue *p;

     while (p != NULL)
     {
      myRand = dequeue(p);
      process(p);
}
```

Solution:

```
flag = 0;
#pragma omp parallel
{
#pragma omp section
{
fillrand(N,A);
#pragma omp flush
flag = 1;
#pragma omp flush(flag)
}
#pragma omp section
{
while (!flag)
#pragma omp flush(flag) #pragma omp flush
sum = sum_array(N,A);
}
}
```

This page was left blank intentionally

Pthreads API Cheat Sheet

Pthread creation

```
pthread_t threads[N]
pthread_create(&threads[i], NULL, start_routine, void *args)
pthread_join(threads[i])
```

Mutex

```
pthread_mutex_t mutex;
pthread_mutex_init(&mutex);
pthread_mutex_lock(&mutex);
pthread_mutex_unlock(&mutex);
pthread_mutex_destroy(&mutex);
```

Semaphore

```
sem_t sem;
sem_init(&sem, 0, initial) -> initial = 0: lock, initial > 0: unlocked
sem_wait(&sem) -> sem = 0: wait, sem > 0 decrement and go
sem_post(&sem) -> increment value
sem_destroy(&sem)
```

Condition Variable

```
pthread_cond_t cond
pthread_cond_init(&cond)
pthread_cond_wait(&cond, &mutex) -> unlock mutex and wait on cond
pthread_cond_signal(&cond) -> wake up threads waiting on cond
pthread_cond_destroy(&cond)
```

Common Condition Variable Usage

```
pthread_mutex_lock(&mutex);
while(isnotready()) pthread_cond_wait(&cond, &mutex);
critical section
pthread_mutex_unlock(&mutex);
pthread_cond_signal(&cond2);
```