LAU
الجامعة اللبنانية الأميركية
Lebanese American University

School of Arts & Sciences
Department of Computer Science & Mathematics
CSC 447—Parallel Programming for Multicore and Cluster Systems
Spring 2016

# Midterm Examination

**Name:** _____ **Student ID:** _____

**Signature:** _____ **Duration:** **75 minutes**

## General Instructions

- There are 6 questions and 11 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) _____

(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) _____

(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) _____

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

(d) _____

(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) _____

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

(f) _____

(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) _____

(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

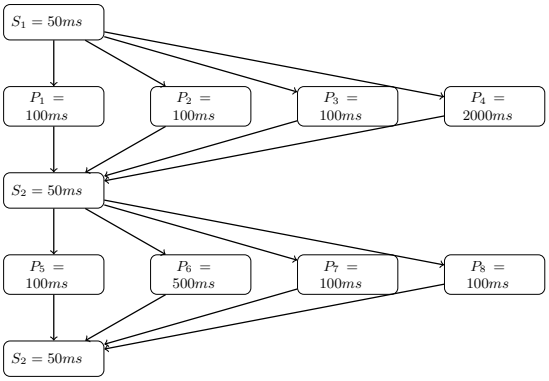
(g) If two threads execute the instruction $x$++ where $x$ is a shared variable initialized to 0, what are the possible values that $x$ could have after the execution of the threads:

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?

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

(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?

(5) (d) Why is the speedup not 2×?

(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];
```

# 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.

# 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);
}
```

*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);
```

# OpenMP Reference Sheet for C/C++

## Constructs

*<parallelize a for loop by breaking apart iterations into chunks>*
**#pragma omp parallel for** [shared(vars), private(vars), firstprivate(vars), lastprivate(vars), default(shared|none), reduction(op:vars), copyin(vars), if(expr), ordered, schedule(type[,chunkSize])]
*<A,B,C such that total iterations known at start of loop>*
for(A=C;A<B;A++) {
    *<your code here>*

    *<force ordered execution of part of the code. A=C will be guaranteed to execute before A=C+1>*
    **#pragma omp ordered** {
        *<your code here>*
    }
}

*<parallelized sections of code with each section operating in one thread>*
**#pragma omp parallel sections** [shared(vars), private(vars), firstprivate(vars), lastprivate(vars), default(shared|none), reduction(op:vars), copyin(vars), if(expr)] {
    **#pragma omp section** {
        *<your code here>*
    }
    **#pragma omp section** {
        *<your code here>*
    }
    ....
}

*<grand parallelization region with optional work-sharing constructs defining more specific splitting of work and variables amongst threads. You may use work-sharing constructs without a grand parallelization region, but it will have no effect (sometimes useful if you are making OpenMP'able functions but want to leave the creation of threads to the user of those functions)>*
**#pragma omp parallel** [shared(vars), private(vars), firstprivate(vars), lastprivate(vars), default(private|shared|none), reduction(op:vars), copyin(vars), if(expr)] {
    *<the work-sharing constructs below can appear in any order, are optional, and can be used multiple times. Note that no new threads will be created by the constructs. They reuse the ones created by the above parallel construct.>*

    *<your code here (will be executed by all threads)>*

    *<parallelize a for loop by breaking apart iterations into chunks>*
    **#pragma omp for** [private(vars), firstprivate(vars), lastprivate(vars), reduction(op:vars), ordered, schedule(type[,chunkSize]), nowait]
    *<A,B,C such that total iterations known at start of loop>*
    for(A=C;A<B;A++) {
        *<your code here>*

        *<force ordered execution of part of the code. A=C will be guaranteed to execute before A=C+1>*
        **#pragma omp ordered** {
            *<your code here>*
        }
    }

    *<parallelized sections of code with each section operating in one thread>*
    **#pragma omp sections** [private(vars), firstprivate(vars), lastprivate(vars), reduction(op:vars), nowait] {
        **#pragma omp section** {
            *<your code here>*
        }
        **#pragma omp section** {
            *<your code here>*
        }
        ....
    }

    *<only one thread will execute the following. NOT always by the master thread>*
    **#pragma omp single** {
        *<your code here (only executed once)>*
    }
}

## Directives

**shared(vars)** *<share the same variables between all the threads>*
**private(vars)** *<each thread gets a private copy of variables. Note that other than the master thread, which uses the original, these variables are not initialized to anything.>*
**firstprivate(vars)** *<like private, but the variables do get copies of their master thread values>*
**lastprivate(vars)** *<copy back the last iteration (in a for loop) or the last section (in a sections) variables to the master thread copy (so it will persist even after the parallelization ends)>*
**default(private|shared|none)** *<set the default behavior of variables in the parallelization construct. shared is the default setting, so only the private and none setting have effects. none forces the user to specify the behavior of variables. Note that even with shared, the iterator variable in for loops still is private by necessity >*
**reduction(op:vars)** *<vars are treated as private and the specified operation(op, which can be +,*,-,&,|,&,&&,||) is performed using the private copies in each thread. The master thread copy (which will persist) is updated with the final value.>*

**copyin(vars)** *<used to perform the copying of threadprivate vars to the other threads. Similar to firstprivate for private vars.>*
**if(expr)** *<parallelization will only occur if expr evaluates to true.>*
**schedule(type [,chunkSize])** *<thread scheduling model>*

| *type* | *chunkSize* |
|---|---|
| *static* | *number of iterations per thread pre-assigned at beginning of loop (typical default is number of processors)* |
| *dynamic* | *number of iterations to allocate to a thread when available (typical default is 1)* |
| *guided* | *highly dependent on specific implementation of OpenMP* |

**nowait** *<remove the implicit barrier which forces all threads to finish before continuation in the construct>*

---

## Synchronization/Locking Constructs *<May be used almost anywhere, but will only have effects within parallelization constructs.>*

*<only the master thread will execute the following. Sometimes useful for special handling of variables which will persist after the parallelization.>*
**#pragma omp master** {
<your code here (only executed once and by the master thread).
}

*<mutex lock the region. name allows the creation of unique mutex locks.>*
**#pragma omp critical** [(name)] {
<your code here (only one thread allowed in at a time)>
}

*<force all threads to complete their operations before continuing>*
**#pragma omp barrier**

*<like critical, but only works for simple operations and structures contained in one line of code>*
**#pragma omp atomic**
*<simple code operation, ex. a += 3; Typical supported operations are ++,--,+,*,-,/,&,^,<<,>>,| on primitive data types>*

*<force a register flush of the variables so all threads see the same memory>*
**#pragma omp flush**[(vars)]

*<applies the private clause to the vars of any future parallelize constructs encountered (a convenience routine)>*
**#pragma omp threadprivate(vars)**

---

## Function Based Locking *< nest versions allow recursive locking>*

void **omp_init_[nest_]lock**(omp_lock_t*) *<make a generic mutex lock>*
void **omp_destroy_[nest_]lock**(omp_lock_t*) *<destroy a generic mutex lock>*
void **omp_set_[nest_]lock**(omp_lock_t*) *<block until mutex lock obtained>*
void **omp_unset_[nest_]lock**(omp_lock_t*) *<unlock the mutex lock>*
int **omp_test_[nest_]lock**(omp_lock_t*) *<is lock currently locked by somebody>*

---

## Settings and Control

int **omp_get_num_threads**() *<returns the number of threads used for the parallel region in which the function was called>*
int **omp_get_thread_num**() *<get the unique thread number used to handle this iteration/section of a parallel construct. You may break up algorithms into parts based on this number.>*
int **omp_in_parallel**() *<are you in a parallel construct>*
int **omp_get_max_threads**() *<get number of threads OpenMP can make>*
int **omp_get_num_procs**() *<get number of processors on this system>*
int **omp_get_dynamic**() *<is dynamic scheduling allowed>*
int **omp_get_nested**() *<is nested parallelism allowed>*
double **omp_get_wtime**() *<returns time (in seconds) of the system clock>*
double **omp_get_wtick**() *<number of seconds between ticks on the system clock>*
void **omp_set_num_threads**(int) *<set number of threads OpenMP can make>*
void **omp_set_dynamic**(int) *<allow dynamic scheduling (note this does not make dynamic scheduling the default)>*
void **omp_set_nested**(int) *<allow nested parallelism; Parallel constructs within other parallel constructs can make new threads (note this tends to be unimplemented in many OpenMP implementations)>*

*<env vars- implementation dependent, but here are some common ones>*
**OMP_NUM_THREADS** "number" *<maximum number of threads to use>*
**OMP_SCHEDULE** "type,chunkSize" *<default #pragma omp schedule settings>*

---

## Legend

vars is a comma separated list of variables
[optional parameters and directives]
*<descriptions, comments, suggestions>*
.... above directive can be used multiple times
For mistakes, suggestions, and comments please email e_berta@plutospin.com