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 2011

Midterm Exam

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!**
- The exam has 18 bonus points.

Question	Points	Score
1	20	
2	10	
3	30	
4	15	
5	15	
6	25	
Total:	115	

Multiple-Choice Questions

(20) 1. (a) A shared memory computer has access to:

A. the memory of other nodes via a proprietary high-speed communications network
B. a directives-based data-parallel language
C. a global memory space
D. communication time

(b) In the message passing approach:

A. serial code is made parallel by adding directives that tell the compiler how to distribute data and work across the processors.
B. details of how data distribution, computation, and communications are to be done are left to the compiler.
C. is not very flexible.
D. it is left up to the programmer to explicitly divide data and work across the processors as well as manage the communications among them

(c) In Parallel programing, total execution time does not involve:

A. computation time.
B. compiling time.
C. communications time.
D. idle time

(d) Which of the following is true for all send routines?

A. It is always safe to overwrite the sent variable(s) on the sending processor after the send returns.
B. Completion implies that the message has been received at its destination.
C. It is always safe to overwrite the sent variable(s) on the sending processor after the send is complete.
D. All of the above.
E. None of the above.

(e) Consider the following MPI pseudo-code, which sends a piece of data from processor 1 to processor 2:

```
MPI_INIT()
MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
if (myrank = 1)
         MPI_SEND (some data to processor 2 in MPI_COMM_WORLD)
else {
         MPI_RECV (data from processor 1 in MPI_COMM_WORLD)
         print "Message received!"
}
MPI_FINALIZE()
}
```

where `MPI_SEND` and `MPI_RECV` are blocking send and receive routines. Thus, for example, a process encountering the `MPI_RECV` statement will block while waiting for a message from processor 1. If this code is run on a single processor, what do you expect to happen?

A. The code will print "Message received!" and then terminate.
B. The code will terminate normally with no output.
C. The code will hang with no output.
D. An error condition will result.

Amdahl's Law

(10) 2. Amdahl's law determines the speedup of a parallel program that has a sequential fraction $0 \leq f \leq 1$. Suppose we have $f = 1/5$ and $P = 6$.

(a) What is the speedup of this parallel program?

(b) What is the parallel efficiency?

OpenMP Loop Parallelization

(30) 3. (a) Give the contents of array x after the execution of the following OpenMP code fragment if possible, or give the error if there is one. Assume that there are 5 threads all together:

```
int i, p,
     x[5] = {1,1,1,1,1},
     y[5]={0,1,2,3,4};
#pragma omp parallel private (p)
{
  p=5;
  #pragma omp for
     for (i=0; i<5; i++)
         x[i]=y[i]+p;
} /* omp end parallel */
```

(b) Give the output of the following OpenMP code fragment if possible, or give the error if there is one. Assume that there are 4 threads all together:

```
int i;
double sum = 0.0;
#pragma omp parallel for reduction(+:sum)
  for (i=1; i <= 4; i++)
    sum = sum + i;
printf("The sum is %lf\n", sum);
```

(c) Give the output of the following OpenMP code fragment if possible, or give the error if there is one. Assume that there are 4 threads all together:

```
int i;
double sum;
#pragma omp parallel private (sum)
{
  sum = 0.0;
  for (i=1; i <= 4; i++)
    sum = sum + i;
  printf("The sum is %lf\n", sum);
}
```

(d) Give the output of the following OpenMP code fragment if possible, or give the error if there is one. Assume that there are 4 threads all together:

```
int i;
double sum = 0.0;
#pragma omp parallel shared (sum)
{
  for (i=1; i <= 4; i++)
    sum = sum + i;
}
printf("The sum is %lf\n", sum);
```

(e) Give the output of the following OpenMP code if possible, or give the error if there is one. Assume that there are 4 threads all together:

```
int i;
double sum = 0.0;
#pragma omp parallel private (sum)
{
  for (i=1; i <= 4; i++)
    sum = sum + 1;
}
printf("The sum is %lf\n", sum);
```

Parallel Programming Using Shared-Memory

(15) 4. Assuming two threads, write a parallel program using `OpenMP` to sum n numbers that are held in an array $a[]$ where each thread computes half of the sum. The final result is to be held in a single location. Give clear comments explaining the code.

Parallel Rank Sort Using OpenMP

(15) 5. Rank sort is a sorting technique that counts the number of *numbers* that are smaller than each selected *number*. The count provides the position of selected *number* in the sorted list; that is, its "rank." Thus, $a[0]$ is read and compared with each of the other numbers, $a[1] \ldots a[n-1]$, recording the number of numbers less than $a[0]$. Suppose this number is x. This is the index of the location in the final sorted list. The number $a[0]$ is copied into the final sorted list $b[0] \ldots b[n-1]$, at location $b[x]$. Actions repeated with the other numbers. The algorithm has an overall sorting time complexity of $O(n^2)$.

(a) Write a sequential function for the Rank Sort using C.

(b) Write a parallel program using openMP for ranksort. Give clear comments explaining the code. Analyze the code and discuss the speed up attained. Is it worth it?

Life-Long Learning

(25) 6. David Patterson states that "the recent switch to parallel microprocessors is a milestone in history of computing. Industry has laid out a roadmap for multicore designs that preserve the programming paradigm of the past via binary-compatibility and cache-coherence. Conventional wisdom is now to double the number of cores on a chip with each silicon generation. [...] Our investigations into the future opportunities led to the following recommendations which are more revolutionary than what industry plans to do:

1. The target should be 1000s of cores per chip;
2. Maximize application efficiency, programming models should support a wide range of data types and successful models of parallelism: data-level parallelism, independent task parallelism, and instruction-level parallelism.
3. Should play a larger role than conventional compilers in translating parallel programs.

Comment on the above and describe the technology drivers that have led to the multi-core paradigm shift in an essay that contrasts various multicore approaches and explains the drivers behind each one of these technologies. Explain the barriers that may not help achieve the above objectives or recommendations. You will be graded on the technical merit as well as on the **English** and language structure!

(continued)

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