CISC 372: Parallel Computing
Exam 2 Review

Stephen F. Siegel

Department of Computer and Information Sciences
University of Delaware

Reference sheet & Notes

- you will be given a sheet with signatures of all functions needed
  - also list of type names and constants
  - also basic pragmas in OpenMP
- you have to know what they mean, the syntax, and how to use the functions and other language primitives
- you may bring one (1) sheet of 8.5x11" paper with anything written/printed on both sides
- you can not use any notes on your computer or other devices

Summary

- multithreaded programming
- starting threads, joining threads
- communication and coordination with shared variables
- mutexes, condition variables, and concurrency flags
- data races and deadlocks: how to recognize them and how to avoid them
- critical sections, producer/consumer problems, and barrier algorithms
- directive-based multithreaded-programming and OpenMP
- private vs. shared variables
- work-sharing and loop distribution strategies
- hybrid MPI+threads programming

`pthread_t`

- a type
- a "thread ID" or reference to a running thread
- an opaque object
    - you cannot perform any operations on something of type `pthread_t`
    - you can only use that thing as an argument to functions in the Pthreads library
    - you don't know how it is actually defined
        - it may be a pointer, or an int, or ...
        - completely up to the Pthread implementation

pthread_create

```
int pthread_create(pthread_t * thread,
                   pthread_attr_t * attr,
                   void *(*start_routine)(void*),
                   void * arg);
```

`pthread_create(thread, attr, start_routine, arg)`

`thread`	where ID of new thread is returned (`pthread_t*`)
`attr`	thread attributes, can be `NULL` (`pthread_attr_t*`)
`start_routine`	function to run in new thread (`void *(*)(void*)`)
`arg`	argument to provide to `start_routine` (`void*`)

Note: `start_routine` consumes a `void*` and returns a `void*`

pthread_join

```
int pthread_join(pthread_t thread, void **value_ptr);
```

```
pthread_join(thread, value_ptr)
```

`thread`	ID of thread to wait on (`pthread_t`)
`value_ptr`	address in which to store return val (`void**`)

- blocks until the specified thread terminates
- deallocates the thread resources

Data races

A data race occurs whenever two threads access the same memory location concurrently, without proper synchronization, and at least one of the accesses is a write.
I.e.:

- one thread reads and the other writes, OR
- both threads write

A data race in a Pthreads program results in undefined behavior.
The program could do “anything” (crash, return weird results,. . .)
You can not assume the value written will be one of the two possible “reasonable” values.

- it is the programmer’s responsibility to avoid all data races

Question

Suppose two threads can access a shared variable `x` and execute as follows:

Thread 1: `x=1;`
Thread 2: `x=1;`

Is there a data race?

```
A. Yes
B. No
```

Yes! write-write

Mutexes

- mutex = “mutual exclusion lock”
- used to guarantee that at most one thread can access a shared object at any time
- supports “lock” and “unlock” operations
- example: a single mutex is used to control access to `sum`
- each thread obtains the lock before reading and modifying `sum`
- . . . and releases lock when it is done
- a thread will block when trying to obtain the lock if another thread owns the lock

Using mutexes to enforce critical sections

- a mutex is typically used to control access to some shared data
- this is purely a programming convention
  - no formal relationship between the mutex and the data
  - programmer should document the relationship clearly
- typical control flow:
  1. obtain lock;
  2. access the shared data;
  3. release the lock;
- do this wherever the data is accessed!
  - if you miss one case, all bets are off

Pthreads mutex interface

- type
  - pthread_mutex_t : opaque handle to a mutex
- functions

```
int pthread_mutex_init(pthread_mutex_t * mutex,
                       pthread_mutexattr_t * attr);
int pthread_mutex_destroy(pthread_mutex_t * mutex);
int pthread_mutex_lock(pthread_mutex_t * mutex);
int pthread_mutex_unlock(pthread_mutex_t * mutex);
```

- use NULL for the attribute argument for now
- all functions return error code (0=success)
- see add_fix.c

Condition variables

- a condition variable *c* is used with a mutex
- when a thread owns the mutex, it may want to wait until some condition holds (due to actions of other threads)
- it can do this by waiting on *c*
- this reliquishes the locks and the thread goes to sleep
- other threads run
- at some point in future, another thread can issue a notification on *c*
- the thread that is asleep may be notified
    - it wakes up and has the opportunity to regain the lock once the thread owning the lock relinquishes it
    - typically, after the thread wakes up, it will check some condition
    - if the condition holds, great, it continues running with the lock
    - otherwise, it waits again (loops are good for this)

Condition variables in Pthreads

- `pthread_cond_init(pthread_cond_t * cond, NULL)`
  - initialize a condition variable
- `int pthread_cond_destroy(pthread_cond_t *cond);`
  - destroy the previously initialized condition variable
- `int pthread_cond_signal(pthread_cond_t *cond);`
  - wake up one or more threads waiting on `cond`
- `int pthread_cond_broadcast(pthread_cond_t *cond);`
  - wake up all threads waiting on `cond`
- `int pthread_cond_wait(pthread_cond_t * cond, pthread_mutex_t * mutex);`
  1. release lock on `mutex`
  2. go to sleep
  3. when woken up: try to regain lock on `mutex`

Semantics of condition variables

- every thread is either running, blocked waiting for lock, or asleep
  - an asleep thread is not contending for resources/consuming CPU cycles
  - note: I am using "asleep" here in a non-standard sense
- state of condition variable: set of waiting threads
- `wait`
  - release lock on mutex, state changes from running to asleep, thread added to `cond`'s wait set
  - when signaled: state changes from asleep to blocked, thread removed from `cond`'s set
  - later the thread may regain the lock on `mutex`, just like any other thread trying to unlock `mutex`
- `signal`
  - changes state of one or more waiting threads as above, removes them from waiting set
  - usually called from thread that owns lock on `mutex`, but not required by Pthreads
- `broadcast`: signals all waiting threads, waitset become empty

Typical pattern for using condition variables

```
obtain lock on mutex;
...
while (!expr) {
  wait on cond;
}
// at this point you know expr holds
// assuming expr can only be changed
// by a thread holding lock on mutex!
...
release lock on mutex;
```

Question 1: condition variable

When a thread calls `wait` on a condition variable and mutex, it should own the mutex.

[A] True
[B] False

True

Question 2: condition variable

When a thread calls `signal` on a condition variable, every thread waiting on that condition variable will “wake up”.

[A] True
[B] False

False

Question 3: condition variable

When a waiting thread receives a signal, it will automatically re-obtain the mutex lock.

[A] True
[B] False

False

Question 4: condition variable

When a waiting thread receives a signal and regains the lock, the condition upon which it was waiting must hold.

[A] True
[B] False

False

Concurrency flags

- a flag is a boolean variable
- a concurrency flag is a shared boolean variable used in a particular disciplined way
  - also known as a "binary semaphore"
- concurrency `flags` are basic concurrency building blocks
- can be used to construct all kinds of complex synchronization patterns and data structures
  - mutual exclusion protocols, barriers, reductions, . . .
- state: a flag has two values, 0 and 1
- atomic operations
  - `raise`
    - can only be invoked when value is 0, otherwise error
    - sets value to 1
  - `lower`
    - blocks until value is 1, then sets value to 0 in one atomic step
    - no other thread can perform any operation on flag between check that value is 1 and set to 0

Interface for flags: `flag.h`

```
typedef ... flag_t;

/* Initializes the flag with the given value.  Must be called before
   the first time the flag is used. */
void flag_init(flag_t * f, _Bool val);

/* Destroys the flag */
void flag_destroy(flag_t * f);

/* Increments f atomically, and returns the result.  Notifies threads
   waiting for a change on f. An assertion is violated if f is 1 when
   this function is called. */
void flag_raise(flag_t * f);

/* Waits for f to be 1, then sets it to 0, all atomically. */
void flag_lower(flag_t * f);
```

Synchronization patterns: barriers

A very common pattern in multi-threaded programs:

```
while (true) {
  compute something;
  barrier();
}
```

- barrier(): no thread can leave until every thread has entered
- thread 1 needs to read something produced by thread 2 in previous iteration
- how to construct a “barrier” for threads?
- many ways, using synchronization primitives we have already learned
- solutions differ in their performance charactertistics
- desired characteristics of barriers:
  1. no one leaves until everyone enters
  2. no unnecessary delay: after last thread enters, everyone can leave without further delay
  3. reuseable : need to use the same barrier object over and over

A 2-thread barrier using flags

- two flags are used `f1` and `f2`
  - `f1` is used by Thread 1 to send a signal to Thread 2 saying "I have arrived at barrier"
  - `f2` is used by Thread 2 to send a signal to Thread 1 saying "I have arrived at barrier"
- Thread 1
  1. raises `f1`
  2. lowers `f2`
- Thread 2
  1. lowers `f1`
  2. raises `f2`

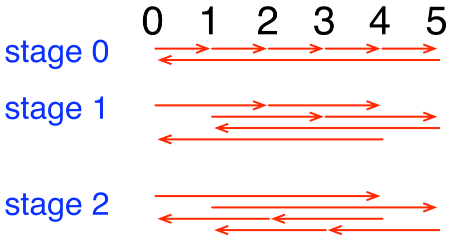
Is it a correct, re-useable barrier with no unnecessary delay?

See `2barrier.c`.

General *n*-thread barrier implementations: counter barrier, coordinator barrier, combining tree barrier, butterfly barrier, dissemination barrier.

Dissemination Barrier: `dissem_barrier.c`

- butterfly requires n to be power of 2 or have exceptional code which breaks symmetry
- dissemination barrier works for any n
- uses cyclic order of threads
- two flags (`a` and `b`) for each thread, in each stage
- in stage i each thread
    - synchs with thread 2^i to the right using `a` and `b` flags of that thread
    - synchs with thread 2^i to the left using its `a` and `b` flags
- stages: $0 \leq i < \lceil \log_2 n \rceil$



Dissemination barrier: code

```
for (int stage=0, i=1; stage<nstages; stage++, i*=2) {
  flag_raise(&bs->a[stage][(tid+i)%nthreads]);
  flag_lower(&bs->a[stage][tid]);
  flag_raise(&bs->b[stage][tid]);
  flag_lower(&bs->b[stage][(tid+i)%nthreads]);
}
```

OpenMP: the `parallel` directive

```
#pragma omp parallel [clauses]
S
```

- a program begins execution with one thread
- executing a parallel directive creates a parallel region
- when control enters the region, a team of threads is created
  - the team includes the original thread, known as the master thread
- all of the threads in the team execute the statement S concurrently
- *S* is typically a big compound statement
- additional directives inside *S* control how threads in the team behave
- at the end of *S* there is an implicit barrier
- all threads join up at this point
- all threads other than the master essentially disappear
- the master continues execution

`hello1.c`: parallel directive example

```
#include <stdio.h>

int main (int argc, char *argv[]) {
  printf("I am the master.\n"); // just the master
  #pragma omp parallel
  {
    printf("Hello, world.\n"); // all threads
  } /* end of parallel region */
  printf("Goodbye, world.\n"); // just the master
}
```

```
omp$ gcc-mp-4.8 -fopenmp hello1.c
omp$ ./a.out
I am the master.
Hello, world.
Hello, world.
Goodbye, world.
omp$
```

Basic OMP functions

- need to `#include <omp.h>`
- `int omp_get_num_threads()`
  - returns the number of threads in the team in the current region
- `int omp_get_thread_num()`
  - returns the ID of the calling thread
  - threads within a team are numbered $0, 1, \ldots$
  - master thread is always thread 0

Private vs. shared variables

- if a variable is declared within the parallel region, all threads have their own copy of that variable
- if a variable is declared before the parallel region and is visible in the region, you have a choice: variable can be
  - private: all threads have their own copy, or
  - shared: one shared variable
- specify what you want by clauses of the form
  - `shared(u1,u2,...)`
  - `private(v1,v2,...)`
- some (obvious) points
  - the `u1`,`u2`,... and `v1`,`v2`,... must all be visible at this point
  - a variable cannot be both `shared` and `private`

The `default` clause

- sets the default for private vs. shared in the parallel region
- `default(none)`
  - no default
  - every variable used in parallel region must be explicitly listed in `shared` or `private`
- `default(shared)`
  - if not listed, the variable is shared
- there are rules specifying what happens if you don't have a default clause
  - but ignore them for now
  - explicitly declare every variable used in the region as either `private` or `shared`

hello4.c

```
#include <omp.h>
#include <stdio.h>

int main () {
  int nthreads, tid;

  #pragma omp parallel private(tid) shared(nthreads)
  {
    tid = omp_get_thread_num();
    printf("Hello World from thread = %d\n", tid);
    if (tid == 0) { // only master
      nthreads = omp_get_num_threads();
      printf("Number of threads = %d\n", nthreads);
    }
  } // end of parallel region
}
```

`num_threads()`: requesting the number of threads

- you can request that the team have a specified number of threads
- clause: `num_threads(expr)`
  - where `expr` is an expression which evaluates to a positive integer
- the runtime system may give you the requested number of threads, or it may give you fewer
- if you really need to know how many there are, ask

The threadprivate directive

- consider the program semiprivate.c. What is the output?

```
#include <stdio.h>
#include <omp.h>
int x = 99;
void f() {
  x=omp_get_thread_num();
}
int main() {
#pragma omp parallel private(x) num_threads(5)
  {
    int tid = omp_get_thread_num();
    f();
    printf("Thread %d: x = %d\n", tid, x);
  }
  printf("Final x = %d\n", x);
}
```

semiprivate.c: output

```
omp$ gcc-mp-4.8 -fopenmp semiprivate.c
omp$ ./a.out
Thread 1: x = -348111896
Thread 2: x = -348111896
Thread 3: x = -348111896
Thread 4: x = 19907219
Thread 0: x = 0
Final x = 0
omp$
```

Why?

- the `private` clause affects only references to the variable inside the construct (the static extent), not the region (dynamic extent).
- if you want `x` to be `private` everywhere, you need to use the `threadprivate` directive.

threadprivate.c

```
#include <stdio.h>
#include <omp.h>
int x;
#pragma omp threadprivate(x)
void f() {
  // this updates the private copy of x...
  x=omp_get_thread_num();
}
int main() {
#pragma omp parallel num_threads(5)
  {
    int tid = omp_get_thread_num();
    f();
    printf("Thread %d: x = %d\n", tid, x);
  }
  printf("Final x = %d\n", x);
}
```

threadprivate.c: output

```
omp$ ./a.out
Thread 1: x = 1
Thread 2: x = 2
Thread 0: x = 0
Thread 3: x = 3
Thread 4: x = 4
Final x = 0
omp$
```

- use this when you have a global variable you wish to share between functions
  - and you want it private
- you can even make the shared variable persist between parallel regions
  - certain requirements must be met
  - in particular, all the parallel regions in which variable is used must have same number of threads
- note the variable must be initialized inside a parallel region before it is used

Work-sharing

- you usually don't want all threads in the team to do the same thing
- you can code in branches on thread ID manually, but this is very tedious
- OpenMP provides more convenient, higher-level constructs
  - these are specified using directives within a parallel region
- one class of such constructs are the work-sharing constructs
  - these specify how work is to be divided up among members of the team
  - kinds of work-sharing constructs
    - for loops: distribute iterations to team members
    - sections: distribute independent code bocks (work units)
    - single: let only one thread execute a block

Worksharing constructs: `for` loops

- syntax

```
#pragma omp for [clauses]
for (init-expr; var relop b; incr-expr)
  body
```

- semantics
  - each iteration is executed by exactly one thread in the team
  - barrier at end of loop
  - in general, everything else is unspecified
    - how the iterations are distributed among the team members
    - the order in which the iterations are executed
    - what happens concurrently
- syntactic restrictions on the `for` statement:
  - `init-expr`: `var = expr`, integer type
  - `relop` is one of: `<`, `<=`, `>`, `>=`
  - `b` is a loop-invariant integer expression
  - `incr-expr` has one of a few forms; see OpenMP 4.0 Standard, Section 2.6

Allowed forms for increment expression in `for` loop

- `++var`
- `var++`
- `--var`
- `var--`
- `var += incr`
- `var -= incr`
- `var = var + incr`
- `var = incr + var`
- `var = var - incr`

where `incr` is a loop invariant integer expression

- i.e., throughout one execution of the loop
  - `incr` will have the same value each time control reaches the top of the loop
- however `incr` could have different values in different loop executions

Clauses for the `for` loop directive

- `private(v1,v2,...)`
  - make a shared variable private for the duration of the loop
- `firstprivate(v1,v2,...)`
  - make a variable private and initialize it in every thread
- `lastprivate(v1,v2,...)`
  - make a variable private and copy the final value of variable in the last iteration back to the shared variable at end
- `reduction(...)`
  - apply some associative and commutative operation (like `+`) across all iterations for some variable
- `ordered`: declares that an `ordered` construct may occur in loop body
- `schedule`: options to control how iterations are distributed to threads
- `nowait`: remove the barrier at the end of the loop
- `collapse(`*n*`)`: apply directive to next *n* loops in loop nest
  - *n* is an expression that evaluates to a positive integer
  - iteration space of the *n* loops is collapsed into a single space
  - the iterations in the resulting space are distributed to threads

Question

Can this loop be parallelized with an OpenMP `for` construct?

```
for (i=0; i<n && a[i]>0; i++)
  b[i] = b[i] - a[i];
```

A. Yes
B. No

No! Condition not in standard form.

Reductions: `reduction(`*`reduction-identifier`*` : `*`list`*`)`

- this is another clause that can be added to an `omp for` directive
- performs an (approximately) associative and commutative operation across all threads
- each variable v in the list should be a shared variable
- v should be initialized before entering the loop
- effectively, a private copy of v is created
- each private v is initialized to the default initial value corresponding to the operation
  - 0 for $+$, 1 for $*$, etc.
- all operations in loop body take place on the private copies
- when a thread finishes its iterations:
  - it adds (or whatever the operation is) its private value back to the shared v
  - this happens atomically to prevent races

Reduction example: `reduce.c`

```c
#include <stdio.h>
#include <omp.h>
#define n 10
int a[n], s=1000000;
int main() {
  printf("Start s = %d\n", s);
#pragma omp parallel default(none) shared(a,s)
  {
    int tid = omp_get_thread_num();
#pragma omp for
    for (int i=0; i<n; i++) a[i] = i;
#pragma omp for reduction(+:s) schedule(static,1)
    for (int i=0; i<n; i++) {
      s+=a[i];
      printf("Local s on thread %d = %d\n", tid, s);
    }
  }
  printf("Final s = %d\n", s);
}
```

Reduction example: output

```
  omp$ make run-reduce
gcc-mp-4.8 -fopenmp -o reduce reduce.c
./reduce
Start s = 1000000
Local s on thread 0 = 0
Local s on thread 0 = 2
Local s on thread 0 = 6
Local s on thread 0 = 12
Local s on thread 0 = 20
Local s on thread 1 = 1
Local s on thread 1 = 4
Local s on thread 1 = 9
Local s on thread 1 = 16
Local s on thread 1 = 25
Final s = 1000045
omp$
```

Reduction operations

operation	operator	initial value
addition	+	0
multiplication	*	1
subtraction (?)	-	0
bitwise and	&	~0
bitwise or	|	0
bitwise exclusive or	^	0
logical and	&&	1
logical or	||	0

`schedule(static,` *chunk_size*`)`

- iterations are partitioned into chunks of size *chunk_size*
- chunks are distributed in round-robin order to threads
- last chunk may be smaller
- distribution is "static": determined upon reaching the loop
- you can omit *chunk_size*
  - iteration space divided into chunks of approximately equal size
  - at most one chunk given to each thread

`schedule(dynamic, chunk_size)`

- iterations are partitioned into chunks of size *chunk_size*
- chunks are distributed to threads as they request them
  - similar to the "manager-worker" pattern
  - as soon as a thread completes its chunk, it asks for a new one
- last chunk may be smaller
- advantageous when time to execute an iteration varies in an unpredictable way
- distribution is "dynamic": determined as loop executes

Worksharing constructs: `sections`

```
#pragma omp sections
{
  #pragma omp section
      ...
  #pragma omp section
      ...
          ⋮
}
```

- specifies explicit code blocks which can execute in parallel
- each block (or section) is executed once, by exactly one thread
- a thread may execute several sections, or no sections
- in general: you cannot assume anything about how sections are distributed to threads
- barrier at end (unless overridden with `nowait`)

Worksharing constructs: `single`

```
#pragma omp single
S
```

- indicates that you want only one thread in the team to execute *S*
  - you don't care which thread
- barrier at end (unless overridden with `nowait`)
- typical use: initialization of shared variable

Clauses:

- `private(list)`, `firstprivate(list)`, `nowait`: usual semantics
- `copyprivate(list)`
  - applies to private variables
  - copies final value of variable in the single thread to corresponding variables in all other threads
  - copy occurs at end, before threads leave the barrier

Synchronization constructs

These constructs control synchronization among threads.

- `barrier`
- `ordered`
- `critical`
- `atomic`
- `master`

Note:

- except for `barrier`, these do not impose barriers

OpenMP loop problems

For each of the following code segments, use OpenMP pragmas to make the loop parallel, or explain why the code segment is not suitable for parallel execution.
Choose A if it can be parallelized; B if not.
Assume a and b are disjoint arrays.

```
   for (int i = 0; i < (int)sqrt(x); i++) {
      a[i] = 2.3 * i;
      if (i < 10) b[i] = a[i];
   }
```

```
#pragma omp parallel for shared(a,b,x)
   for (int i = 0; i < (int)sqrt(x); i++) {
      a[i] = 2.3 * i;
      if (i < 10) b[i] = a[i];
   }
```

OpenMP loop problems

```
flag = 0;
for (int i = 0; (i < n) & (!flag); i++) {
  a[i] = 2.3 * i;
  if (a[i] < b[i]) flag = 1;
}
```

No! loop condition not in standard form

OpenMP loop problems, cont.

Assume `foo(i)` does not modify array `a` or `i`.

```
   for (int i = 0; i < n; i++)
      a[i] = foo(i);
```

```
#pragma omp parallel for shared(a,n)
   for (int i = 0; i < n; i++)
      a[i] = foo(i);
```

OpenMP loop problems, cont.

```
   for (int i = 0; i < n; i++) {
      a[i] = foo(i);
      if (a[i] < b[i]) a[i] = b[i];
   }
```

```
#pragma omp parallel for shared(a,b,n)
   for (int i = 0; i < n; i++) {
      a[i] = foo(i);
      if (a[i] < b[i]) a[i] = b[i];
   }
```

OpenMP loop problems, cont.

```
for (int i = 0; i < n; i++) {
   a[i] = foo(i);
   if (a[i] < b[i]) break;
}
```

No! Cannot break out of an OpenMP for loop.

OpenMP loop problems, cont.

Attempt to parallelize the nested loop as much as possible:
(N, M, L are constants):

```
    for (int i=0; i<N; i++)
      for(int j=0; j<M; j++)
        for (int k=0; k<L; k++)
          c[i][j] += a[i][k] * b[k][j];
```

```
#pragma omp parallel for collapse(2) shared(a,b,c,N,M,L)
    for (int i=0; i<N; i++)
      for(int j=0; j<M; j++)
        for (int k=0; k<L; k++)
          c[i][j] += a[i][k] * b[k][j];
```

What will this print

Assume the following is executed with exactly 3 threads.
Explain all possible things it could print.

```
#pragma omp parallel
{
  #pragma omp sections
  {
    #pragma omp section
    {
      printf ("%d", omp_get_thread_num());
    }
    #pragma omp section
    {
      printf ("%d", omp_get_thread_num());
    }
  }
}
```

Answer: 00, 01, 02, 10, 11, 12, 20, 21, 22