COSC 407
Intro to Parallel Computing

Topic 6 - Barriers and Mutexes
(Critical Sections, Atomics and Locks)

COSC 407: Intro to Parallel Computing

1

Outline

Previously (LiveStream):
- Student lead questions
- OpenMP
- Basics, HelloWorld
- Distributing the work
- Example: Summing it all up (as array)

Today:
- Synchronization (barriers, nowait)
- Mutual Exclusion (critical, atomic, locks)

2

Race Condition

- Unpredictable results when two (or more) threads attempt to read or write the same data simultaneously (previous unprotected example)

```
global_sum += my_sum;
```

- Consider two threads each increases the value of a shared integer variable by one

Good

Thread 1	Thread 2		value
			0
read value		←	0
Increase value			0
write back		→	1
	read value	←	1
	increase value		1
	write back	→	2

!!!

Thread 1	Thread 2		value
			0
read value		←	0
	read value	←	0
increase value			0
	increase value		0
write back		→	1
	write back	→	1

- ***Data Race*** happens when two or more threads work on a ***shared data*** item, where at least one thread is trying to update (***write to***) this item.
- Mutual Exclusion: only one thread at a time can have access to a shared resource **(there can only be one!)**

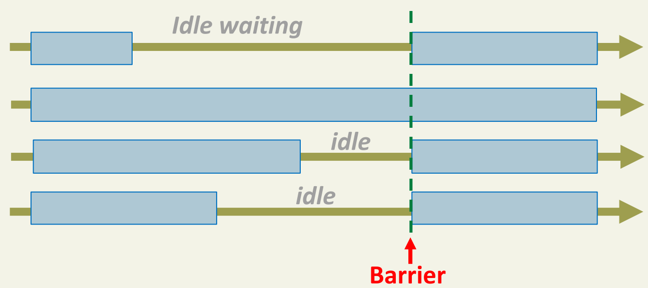
Barriers

- Barriers are used for **threads synchronization**
- All threads must reach the barrier before any of them can proceed
  - We may use barriers between parts of the code that read and write to the same memory locations
- Important
  - Each barrier needs to be encountered by all the threads in a team or none of them
  - The sequence of work-sharing regions and barrier regions encountered must be the same for every thread in the team (more on this soon….)

Barriers

- Two types:
  - ***Implicit barriers***: automatically added for you
    - eg end of critical section (as we saw previously(
  - ***Explicit barriers***: programmers add them explicitly using

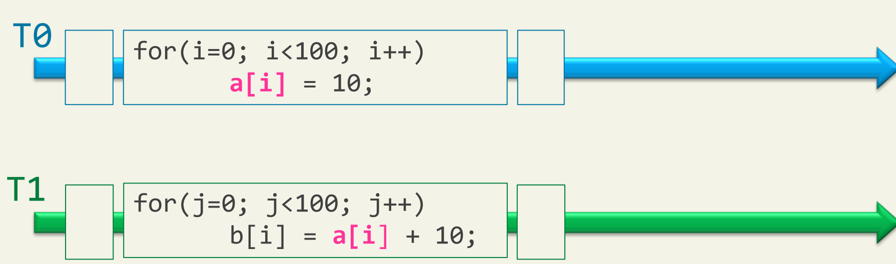
```
#pragma omp barrier
```



5

Barriers: *Example*

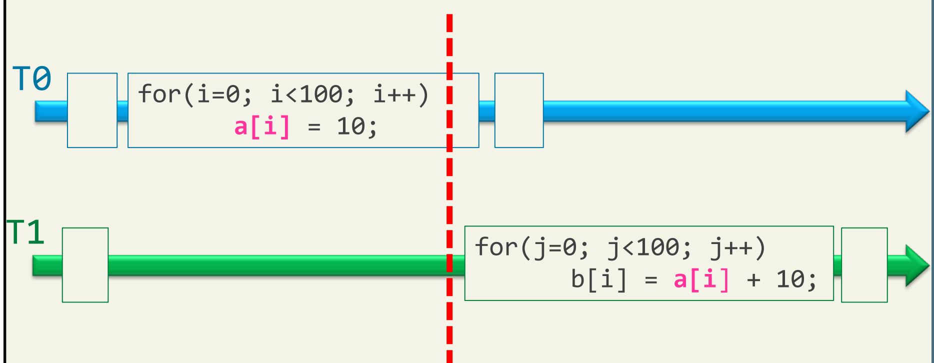
- Assume the following two for-loops are executed in parallel. Without a barrier, a race condition might occur.
  - T1 might read an old value of a[i] before it has been updated by T0.



6

Barriers: *Example, cont'd*

- If you add a barrier between these two parts (i.e., the two for blocks), then you force T1 to wait until the first loop finishes before it resume executing its own for loop



7

Illegal Barriers!

```
#pragma omp parallel
{
  if ( omp_get_thread_num() == 0 ){
    .....
    #pragma omp barrier  // Correction: the barrier should be out of the if-else region
  }
  else{
    .....
    #pragma omp barrier
  }
 // #pragma omp barrier  // The barrier should be added here.
} /*-- End of parallel region --*/
```

- The barrier is illegal as it is not encountered by all the threads in the team
  - Due to conditional statement, some threads may execute different paths (Thread Divergence)

8

nowait

- The use of barriers is expensive (as threads in a team will be idle for a while)
- To reduce synchronization, you can add the `nowait` clause.
- The **implicit barrier** at the end pragma construct can be cancelled with a `nowait`
  – i.e. If `nowait` is used in an OpenMP construct, threads will not synchronize (wait) at the end of these constructs.
- Example:

```
#pragma omp single nowait
{
  ..
}

..
```

Threads don't have to synchronize here

- You can use `nowait` **if there is no data dependency** between two parts of your code

Mutual Exclusion in OpenMP

- Sometimes, only one thread at a line can access a critical section of code or update a variable
  – Same concept as with POSIX threads
- OpenMP provides several mechanisms for insuring mutual exclusion in critical sections:
  – **critical directive**
    - Unnamed and named critical directives
  – **atomic directive**
  – **simple locks**

critical Directive

- Race condition may be controlled by defining a critical region

```
# pragma omp critical
  *global_result_p += my_result;
```

- Critical regions force threads to work one at a time
  - All threads run the code, but **<u>only one thread at a time can run the code in the critical region</u>**
  - When a thread encounters a critical construct, it waits until no other thread is executing a critical region with the same name

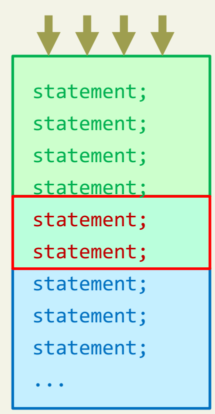
```
#pragma omp critical [(name)]
...... code block .....
```

- All structured blocks modified by an **unnamed critical directive** form a **single critical section**

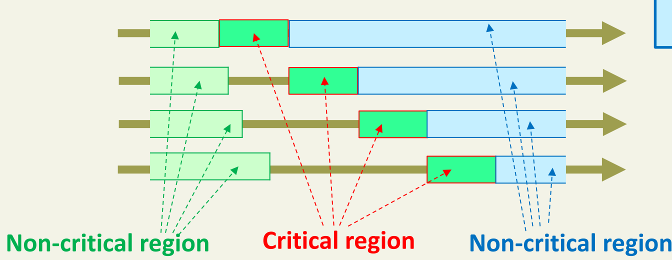
https://www.openmp.org/spec-html/5.0/openmpsu89.html

critical Directive cont'd

- On the right, assume we have a concurrent program that has three code sections: non-critical, critical, non-critical.
- The illustration below shows how 4 threads could be running this program



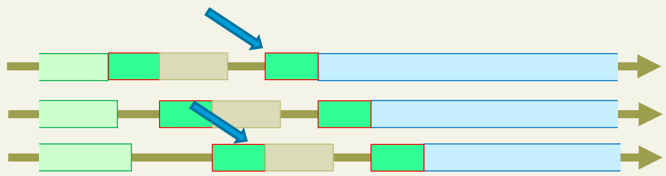
A thread has to wait if it reaches a critical section that is being executed by another thread

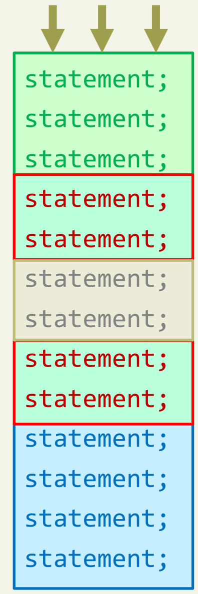


- Critical sections have **no sync barriers at their entry or exit** points within a thread
- Threads wait for each other at a **barrier**.

Multiple critical Sections

- All critical (unnamed) section belong to the same section. Assume we have this code: non-critical, critical, non-critical, critical, non-critical
- The illustration below shows a sample run of this program using 3 threads





Notice how the third thread caused the first to wait until the third one is done.

13

Named critical Sections

- Sometimes you have two critical section and each one must be executed by only one thread at any time **but** it is ok for **both to run in parallel**
  - Example: updating two shared variables a and b
    - It is ok that both are updated at the same time, but cannot be accessed by more than one thread at any time
- OpenMP provides the option of adding a name to a critical directive:

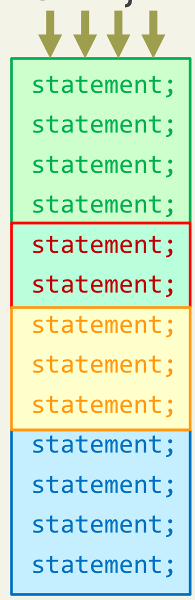
```
# pragma omp critical(name)
```

- When we do this, two blocks protected with critical directives with different names can be executed simultaneously
- Example:

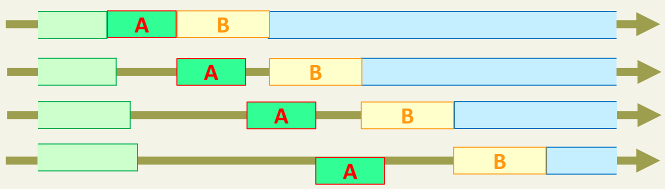
```
# pragma omp critical(name1)
x = f();
# pragma omp critical(name2)
y = g();
```


14

Named critical Sections, cont'd



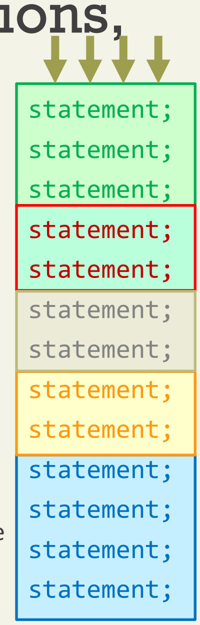
- In this illustration, assume we have a concurrent program that has four code sections with two named critical sections :
  non-critical, critical A, critical B, non-critical.
- Consider 4 threads could be running this program



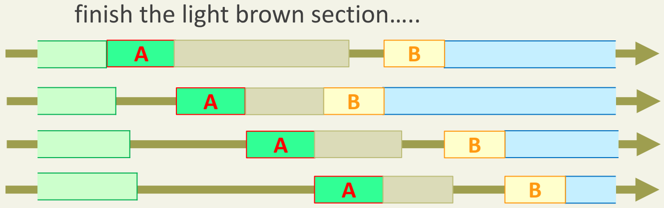
Threads can be running the two critical sections A and B concurrently

15

Named critical Sections, cont'd



- Consider the following code sections:
  non-critical, critical A, non-critical, critical B, non-critical
- The illustration below shows **one** of the scenarios of running this program
  - Assume the top thread took longer than others to finish the light brown section…..



In this case, the second thread entered B first, causing the first to wait until the first one is done (there can only be one section running the named section 'B' at any one time….

16

atomic Directive

- Allows threads to safely update a shared variable and avoids data race conditions
- Similar to critical directive (protects shared data) and can be faster on many processors (if implemented correctly -> but can cause problems if done incorrectly)
  - A critical section that only does a **load-modify-store** can be protected much more efficiently using this special instruction
  - Only the memory update is atomic

Example:

```
#pragma omp atomic
*global_result_p += my_result;
```


atomic Directive

- It can only be applied to load-modify-store operation in one of following forms:

  a) A single statement in the form:

```
x++, ++x, x--, --x
x <op>= <expression>
x = x <op> <expression>
x = <expression> <op> x
   where <op> is one of  +, -, *, /, &, ^, |, <<, >>
   and  <expression> does not reference x
```

  b) A structured block in the form:

```
{ value = x; //any of the expressions in (b) above; }
{ //any of the expressions in (b) above; value = x; }
e.g. {value = x; x += 1;}
```

Danger!

Case 1

- All elements are added sequentially, and there is performance penalty for using *atomic*, because the system coordinates all threads
- Slower than a serial code!
  - Each thread must wait!

```
sum = 0;
for (i=0; i<n; i++)
{
    #pragma omp atomic
    sum += a[i];
}  /*-- End of parallel for --*/
```

Case 2

- Each thread adds up its local sum.
- The *atomic* is only applied for adding up local sums to obtain the total sum

```
sumLocal = 0;
#pragma omp for
for (i=0; i<n; i++)  sumLocal += a[i];
#pragma omp atomic
sum += sumLocal;
}  /*-- End of parallel region --*/
```


Locks

A lock consists of a ***data structure*** and ***functions*** that allow the programmer to explicitly enforce mutual exclusion in a critical section.

```
INITIALIZES a lock-data-structure (by one thread, e.g. master)

#pragma omp parallel
{
   //..not critical section

   LOCK
   //..critical section
   UNLOCK

   //..not critical section
}

DESTROY the lock-data-structure (by e.g. the master)
```

Locks

A lock consists of a ***data structure*** and ***functions*** that allow the programmer to explicitly enforce mutual exclusion in a critical section

```
static omp_lock_t mylock;
omp_init_lock(&mylock);

#pragma omp parallel
{
    //not critical code

    omp_set_lock(&mylock);
    //critical region
    omp_unset_lock(&mylock);

    //not critical region
}

omp_destroy_lock(&mylock);
```

lock data structure is shared among the threads

One of the threads (e.g., master) initializes it

Only one thread can own (set) mylock, and only this thread can unset it. While a thread owns mylock, no other thread can enter this critical section. If another thread attempts to enter the critical section, it will *block* when it calls the lock function.

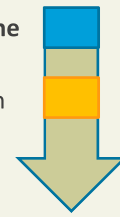
One of the threads destroys the lock.

When to Use Which?

- First, think of **`atomic`** (usually highest performance)
  - Use atomic for single load-modify-store statement (be careful!)
- Second, think of **`critical`**
  - No large difference between the performance between critical directive and locks.
- Finally, if neither `atomic` nor `critical` is possible then use locks
  - e.g., if you want to leave a locked region with jumps (*e.g. break or return).*
    - You cannot leave a region protected by 'critical' with a jump
    - Don't forget to unset the lock!!!!
  - e.g., if you want to protect a single data structure, not the complete block of code.

Some Caveats

1. **Different types** of mutual exclusion **don't** belong to the **same critical region**
   - Don't mix the different types for a **single** critical section
2. There is **no guarantee of fairness** in mutual exclusion constructs
3. It can be **dangerous to "nest"** mutual exclusion constructs
   – **Use named critical sections** if you have to nest mutual exclusion constructs, but even then there is no guarantee to avoid deadlocks



Conclusion/Up Next

- What we covered today (review key concepts):
  – Synchronization (barriers, nowait)
  – Mutual Exclusion (critical, atomic, locks)

- Next:
  – Variable scope (shared, private, firstprivate)
  – Reduction
  – Work Sharing

Homework

- Please review
  - OpenMP Resources (See week three module)
  - Additional resources on Canvas
  - Run the sample code and try the challenge
    - You need to be able to run and understand how to approach a problem
    - Be familiar with the OpenMP directives introduced so far