CMOR 421/521:
Order Dependency and Synchronization

Date: 2/13/2024

This Week:

Tu: Order Dependency and Synchronization

Th: Load Balancing, Reductions, and Thread Mapping

Note: I will be available after class today and Thursday to help with code issues.

Tu	Th	
		1
		2
		3
		4
		5
		6
		7
		8
		9
		10
		11
		12
		13
		14
		15

Today

- Recap of some things from last Tuesday
- Race Conditions
- Order Dependency
- Synchronization

How to OpenMP

- OpenMP uses precompiler ***directives*** to define parallel regions
- Directives can be modified using clauses

```
#pragma omp <directive> <clauses> <myb other stuff>
{
    // Code you want to run in parallel
}
```

Using OpenMP: Parallel Blocks

- The simplest OpenMP directive: **parallel**

```
// Parallel hello world

#pragma omp parallel
{ // Code standards: allowed for OpenMP
    tid = omp_get_thread_num();
    printf("Hello from thread %d\n", tid);
}
// Note: threads are zero-indexed
```

OpenMP: Thread Branching

- Once you enter a parallel block, the code branches to each thread

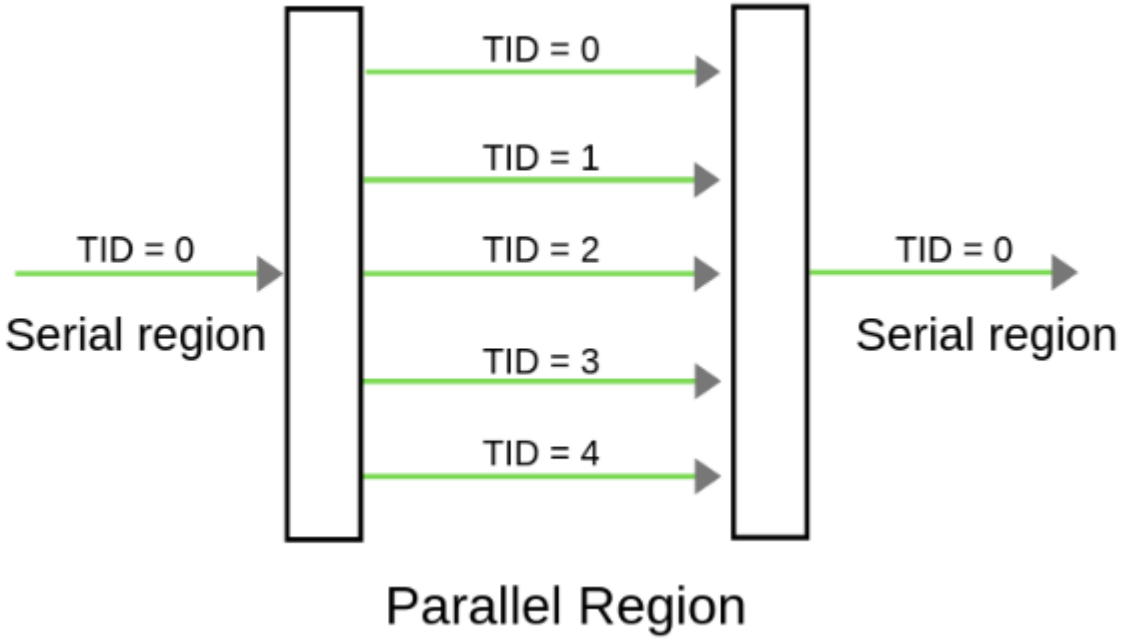


Image credit: https://www.geeksforgeeks.org/openmp-hello-world-program/

OpenMP: Fork-Join Parallelism

- Master thread spawns a team of threads as needed.
- Parallelism added incrementally until performance goals are met, i.e., the sequential program evolves into a parallel program.

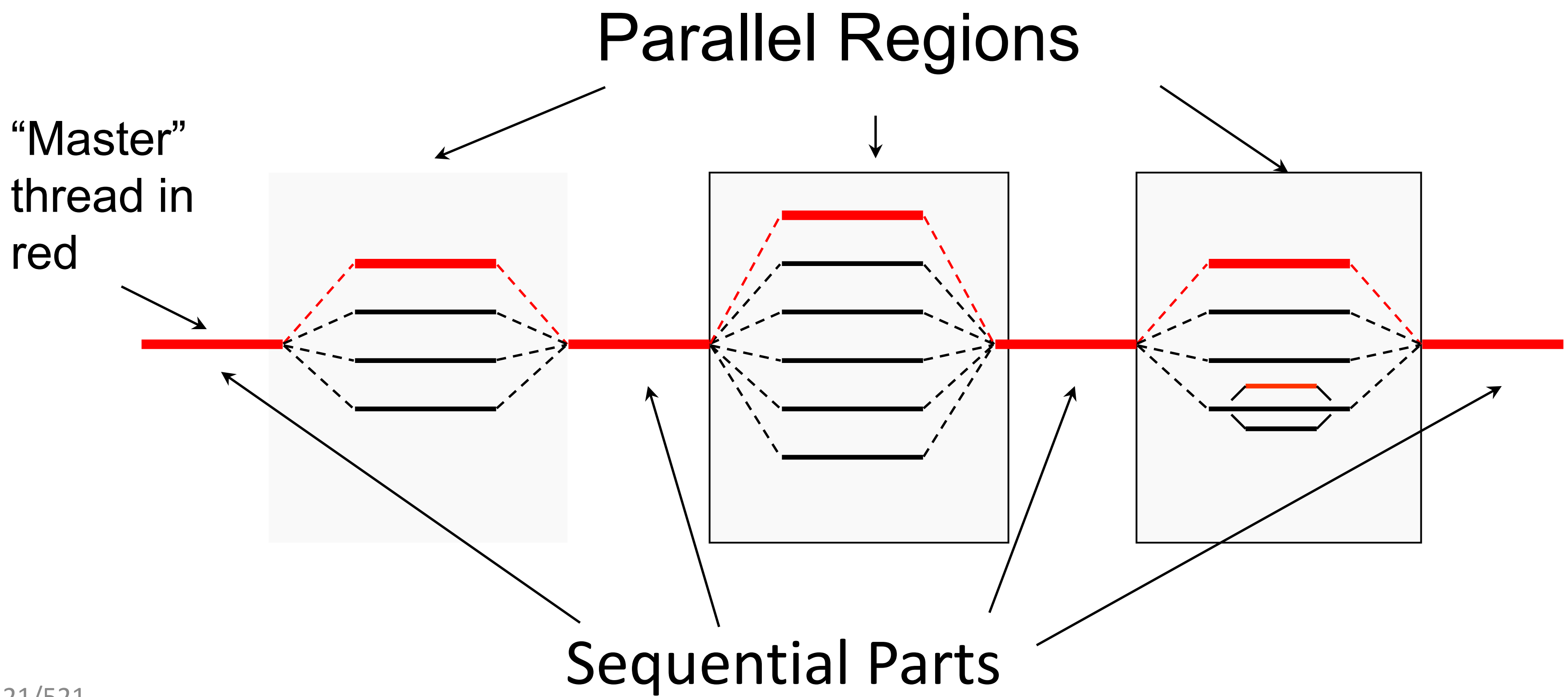


Figure from Aydin Buluc

OpenMP: Parallel Blocks Cont.

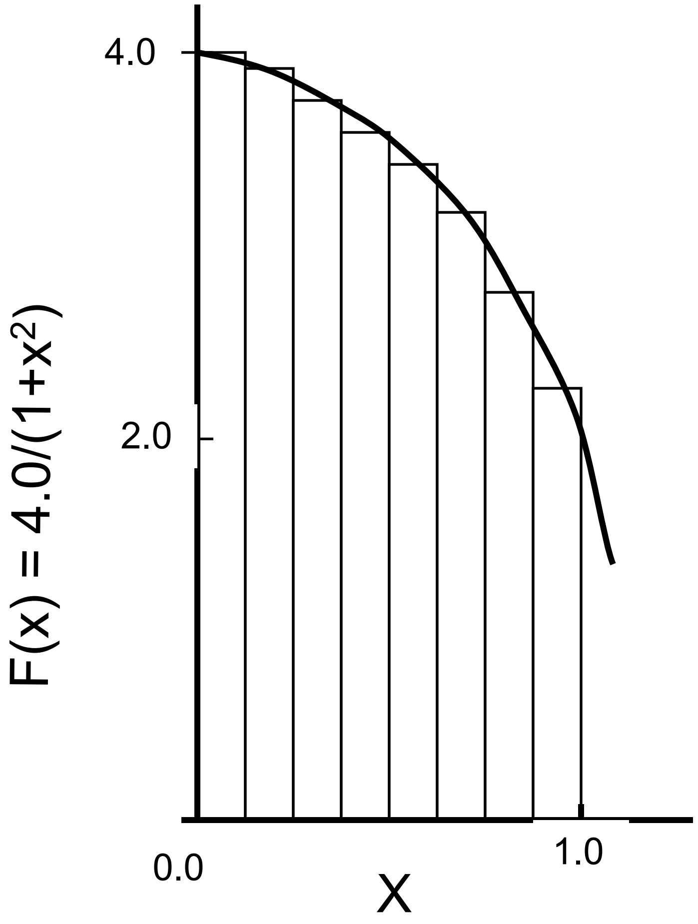
- Each thread does **everything** in the parallel block

Example: you can put a regular for-loop inside a parallel block

```
#pragma omp parallel
{
    for(int i = 0; i < n; i++) {
        A[i] += i;
    } // Every thread will do the whole loop
}
```

- **Threads execute in any order!**
  - They are also in parallel, and can interrupt each other
  - Example: printing is a mess!

An example problem: numerical integration



Mathematically, we know that:

$$\int_0^1 \frac{4.0}{(1+x^2)}\,dx = \pi$$

We can approximate the integral as a sum of rectangles:

$$\sum_{i=0}^{N} F(x_i)\Delta x \approx \pi$$

where each rectangle has width Δx and height $F(x_i) = 4/(1+x^2)$ at the middle of interval i.

An example serial code

- See integration.cpp
- This is a “second order” accurate method; roughly speaking, if we want 14 digits of accuracy, we’ll need 1e7 function evaluations.
- This code resulted in a ***race condition***

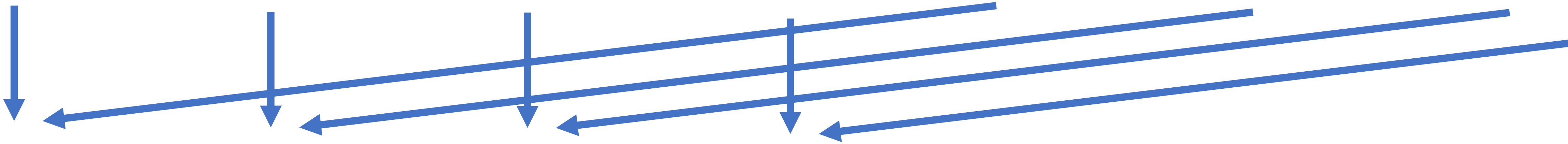
First try at a (correct) solution:

- Ensure no two threads write to the same blocks of memory.

- Simple parallelization pattern: for 4 threads

1 2 3 4 5 6 …

- Thread1 Thread2 Thread3 Thread4 Thread1 Thread2



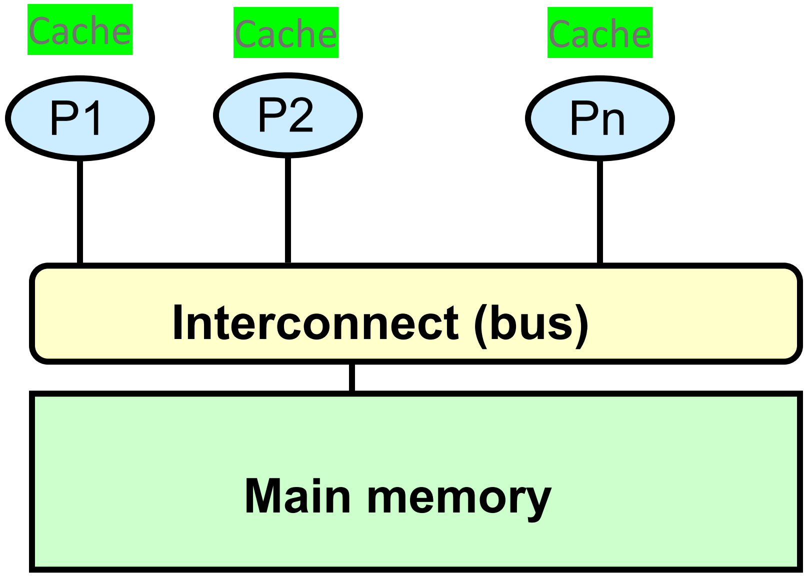
- sum[0] sum[1] sum[2] sum[3]

What about scalability?

- See integration_omp_no_race.cpp

What is the issue?

- The memory hierarchy!
- Each processor has *its own L1 cache (or cache hierarchy)*
  - Cache hierarchy behaves similarly to the single processor case.
- Challenge: how do you ensure **cache coherency?**

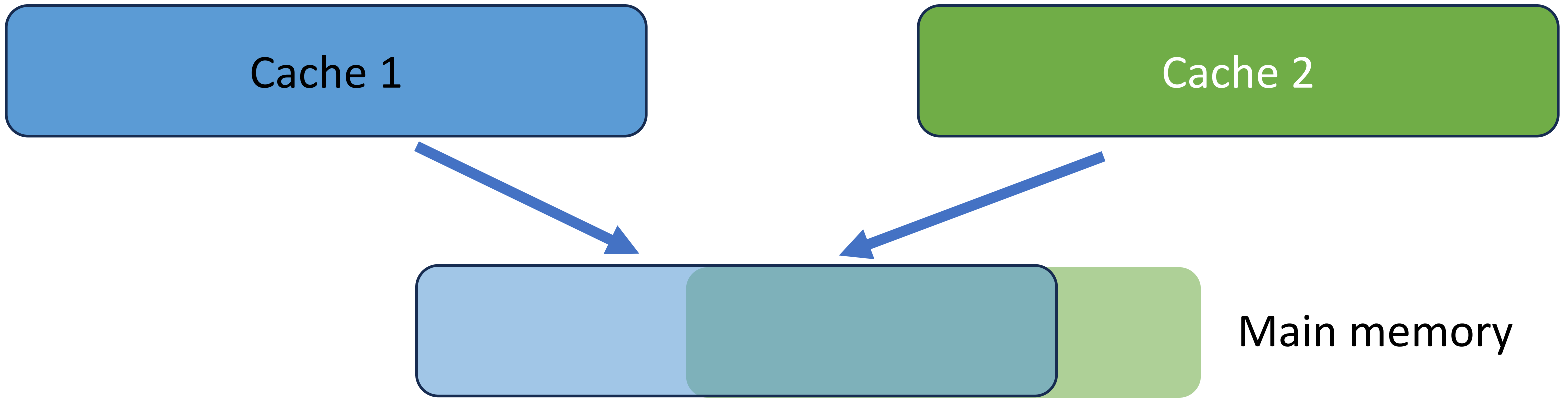


Idealized model of shared memory for processors P1, …, Pn

Why poor scaling?

- "A transaction is *relevant* if it involves a cache block **currently contained in this cache**"
- Suppose you have two threads which access memory locations close to each other.
  - Thread 1 pulls a cache line into Cache 1
  - Thread 2 pulls a cache line into Cache 2

What happens if Cache 1 memory gets updated?



How to fix false sharing?

- **Padding:** ensure memory accessed by different threads do not share the same cache line
  - Add unused entries in between each
- Note: padding may or may not result in the same speedup on all architectures.
  - Cache coherence protocol might be less expensive
  - Smaller number of threads? Compiler takes care of it?
- See integration_omp_padding.cpp

How to fix, but less ugly?

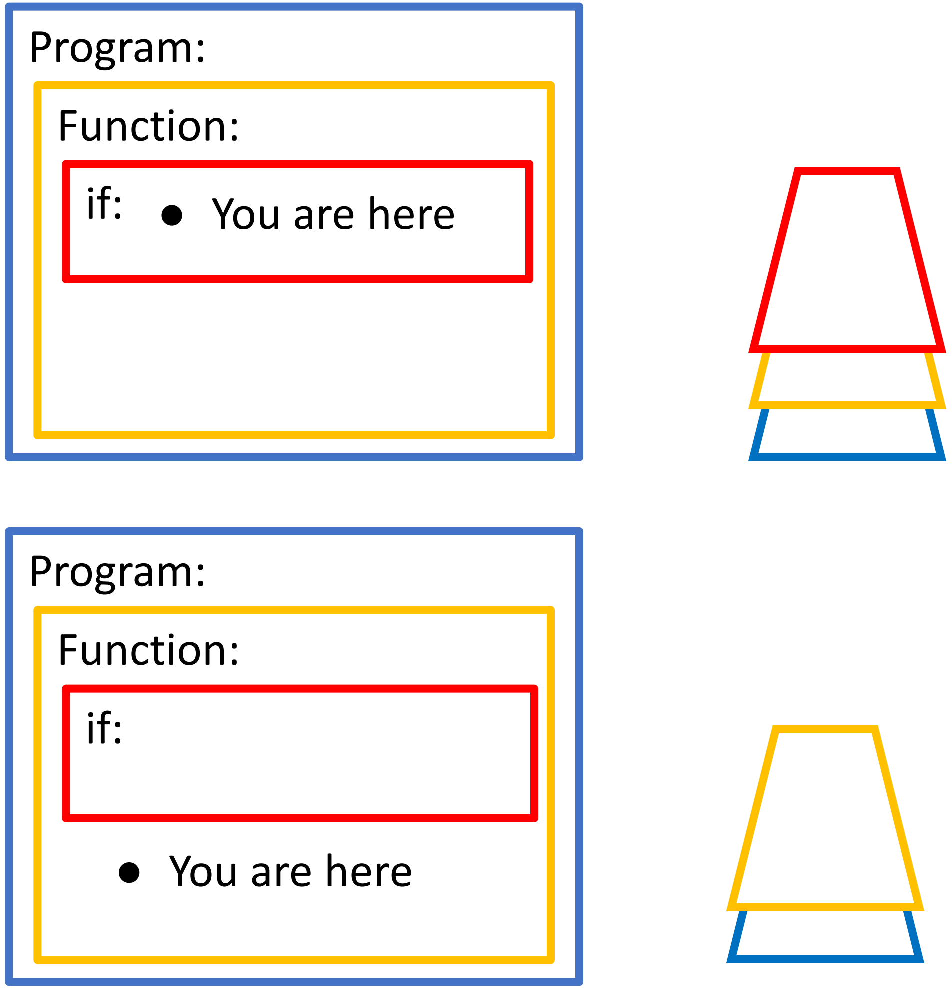
- Padding is fairly invasive

- OpenMP provides several nice alternatives
  - Option 1: **private** variables and **critical** regions
  - Option 2: **omp for**: tries to automate worksharing

The Stack Revisited

- The stack data structure is like a stack of cups: Last-In-First-Out (LIFO)
- Stack memory allocations operate within a single program as a stack
- Allocation and deallocation uses LIFO ordering to ensure scope

Program Scope: Stack memory:

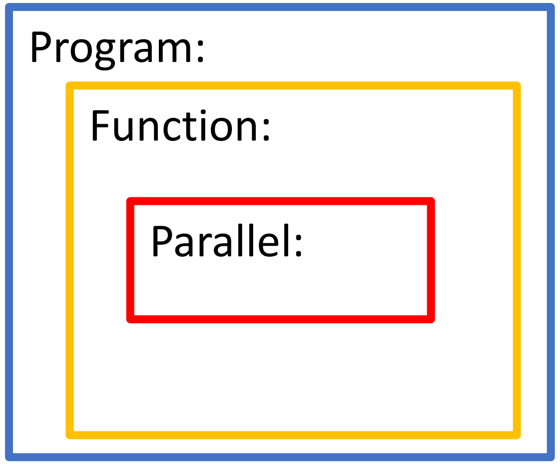


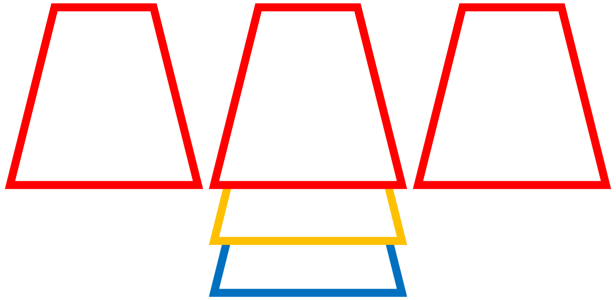
OpenMP: Variable Scope

- The parallel block has braces around it: it scopes variables
- OpenMP uses a shared memory paradigm: which variables are shared, which are not?

The Stack in Parallel

- Each thread has its own variables on the stack
  - Imagine being able to have multiple cups in a given layer
- LIFO structure remains, so does scope





OpenMP: Variable Scope & Shared Memory

- **Q:** Which variables are shared, which are not?
- **A:** By default:
  - (Stack) Variables declared before the parallel block are **shared**, i.e. all threads have access to those variables
  - (Stack) Variables declared inside the block are **private**, i.e. each thread has their own version of that variable
  - **We can change this using *clauses* with our directives**

Recall: How to OpenMP

- OpenMP uses precompiler directives to define parallel regions
- Directives can be modified using clauses

```
#pragma omp <directive> <clauses> <myb other stuff>
{
    // Code you want to run in parallel
}
```

OpenMP: Scoping Clauses

- **shared:** all threads have access to the same variable
- **private:** every thread get their own instance of a variable, but it is not initialized
- **firstprivate:** same as private, but initialized
- **default:** sets the scoping for all variables that are used in the block

OpenMP: Scoping Clauses

- **shared:** all threads have access to the same variable

```
#pragma omp parallel shared(x)
{
    int x;
    tid = omp_get_thread_num();
    do_stuff(tid, x);
}
```

- Variables which are defined outside the parallel section but used inside it are assumed to be **shared**

OpenMP: Scoping Clauses

- **private:** every thread get their own instance, not initialized

```
int x = 4;
#pragma omp parallel private(x)
{   // x may not be 4 in here
    tid = omp_get_thread_num();
    do_stuff(tid, x);
}
```

- Variables defined inside #pragma omp parallel are private

Example: integration

- **private:** every thread get their own instance, not initialized
  - We can make the “sum” accumulator a private variable

- Needs a synchronization directive: #pragma omp critical

- See integration_omp_private.cpp

Scaling Revisited

- Weak and strong scaling laws were ideal cases
  - They only cared about parallel (p), sequential (s) portion
  - **The act of parallelizing a code may change p and s!**
- Think of the final loop that combines the thread-local sums

Amdahl's (Strong)

$$S(n) = \frac{1}{s + \frac{p}{n}}$$

Order Dependency

- Order dependencies can be operator, algorithm, or instruction based
  - We can influence algorithm and instruction dependencies
  - **Algorithm:** Rectify *then* normalize a vector
  - **Instruction:** Modify a variable value
  - Operator dependencies are handled by the compiler
- **Order dependencies ensure correctness**
  - Order *sensitivities* can impact efficiency, e.g. locality

Race Conditions

- A *race condition/data race* occurs when two threads try to read/write to the same variable
  - If only reads occur, there will be no issue
  - If even one thread tries to write, there can be an issue
    - Other threads may read and use a *stale* value
- Race conditions are an instruction-based violation of order dependency unique to parallelized codes

Parallel Codes and Order Dependency

- Parallel codes can also have other problems:
  - Threads can execute in any order
  - They can start/**finish** in any order as a result
- What if you need to make sure the last (parallel) task was completed before moving to the next?
- This gives rise to the idea of ***thread safety***
  - Can a given routine **guarantee** its correctness in a parallel setting?

Thread Safety: Synchronization

- To preserve order dependency, we must introduce *synchronization*
- Synchronization is an operation between threads/their memory that restricts threads ability to move forward or enter a given block of code
  - Can be implicit or explicit
  - May involve all threads or just some
  - Example: **critical** directive from last week

Implicit Synchronization

- The previously described commands are examples of explicit synchronization
  - i.e., synchronization explicitly requested by the user
- Parallel regions are also subject to implicit synchronization at their closing brace
- Note the number of threads may change from one parallel region to the next though

Implicit Synchronization: Join

- Threads rejoin at the end of a parallel region
  - When we enter the enclosing region, we know all threads have completed the inner region’s code

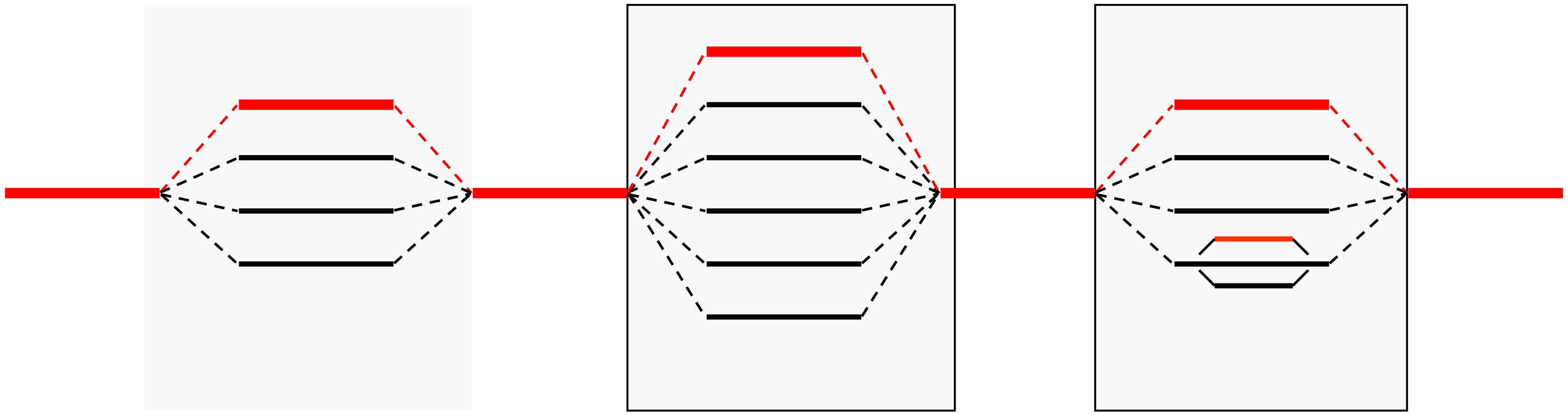


Figure from Aydin Buluc

Implicit Synchronization

- Parallel regions are also subject to implicit synchronization at their closing brace

```
#pragma omp parallel for
for(i = 0; i < n; i++) {
    // Routine 1
} // Implicitly synchronizes here


#pragma omp parallel for
for(i = 0; i < n; i++) {
    // Routine 2
}
```

```
#pragma omp parallel
{
    #pragma omp for
    for(i = 0; i < n; i++) {
        // Routine 1
    }
    // Explicit syn. required here
    #pragma omp for
    for(i = 0; i < n; i++) {
        // Routine 2
    }
}
```

OpenMP Synchronization Directives

OpenMP provides several directives for synchronization which do different things

Important:
- barrier
- critical
- single

Less important:
- atomic
- master
- ordered
- flush

Synchronization: barrier

- **barrier** makes threads wait until all threads have reached that point in the code
- Binds to the innermost parallel region

```
#pragma omp parallel
{
    // Routine 1
    #pragma omp barrier

    // Routine 2
}
```

Barrier and Nested Regions

- **barrier** binds the innermost parallel region
- Threads on the enclosing region are not affected

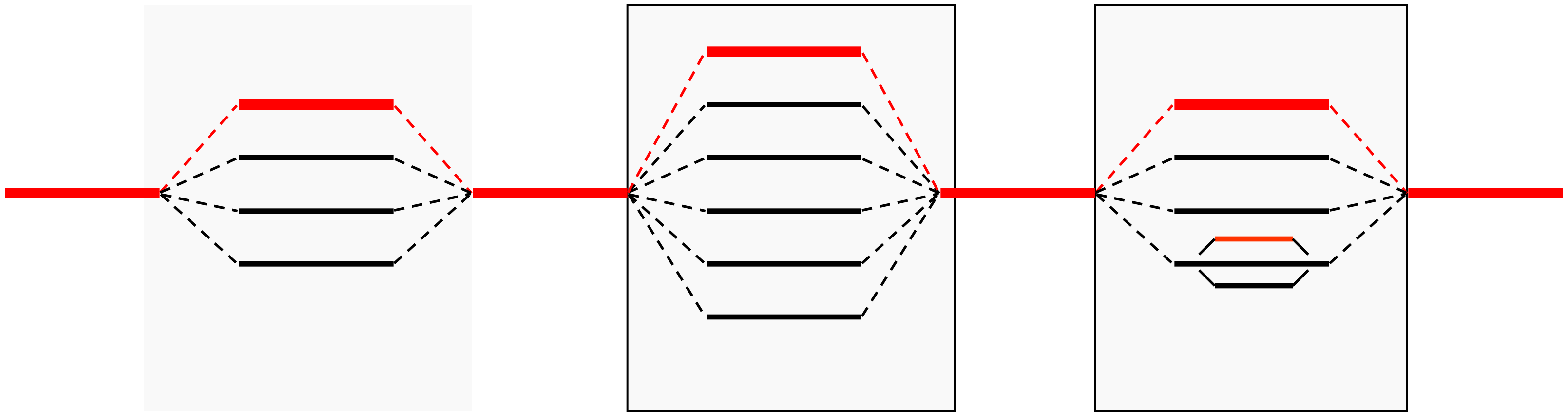


Figure from Aydin Buluc

Synchronization: critical

- **critical** only allows one thread to execute that block of code at a time

```
#pragma omp parallel shared(x,y)
{
    #pragma omp critical
    {
        x++;
        y++;
    }
}
```

Synchronization: single

- **single** code that should only be run by one thread, but which thread does it doesn't matter

```
#pragma omp parallel
{    …
     // for-loop for some operation
     #pragma omp single
     {
          print("Completed work\n");
     }
}
```

Synchronization: atomic

- **atomic** is similar to **critical** but can only deal with one operation

```
#pragma omp parallel shared(x)
{
    #pragma omp atomic
        x++;
}
```

Synchronization: master

- **master** same as single, but now the master thread (thread 0) must be the one run the code

```
#pragma omp parallel
{    …
     // for-loop for some operation
     #pragma omp master
     {
          print("Completed work.\n");
     }
}
```

Synchronization: ordered

- **ordered** specifies that a block of code in a parallel for loop must be run in order

```
#pragma omp parallel for
for(i = 0; i < n; i++) {
    // do actual work
    #pragma omp ordered
    {
        print(“%d\n”, i);
    }
}
```

Synchronization: flush

- **flush** write all shared variable to memory; i.e. make memory (RAM/disk) consistent with cache values
- Can specify which vars or do all

```
#pragma omp parallel shared(x,y)
{
    // Does stuff in parallel w x,y
    #pragma omp flush
    #pragma omp flush(x)
}
```

Effect of Synchronization

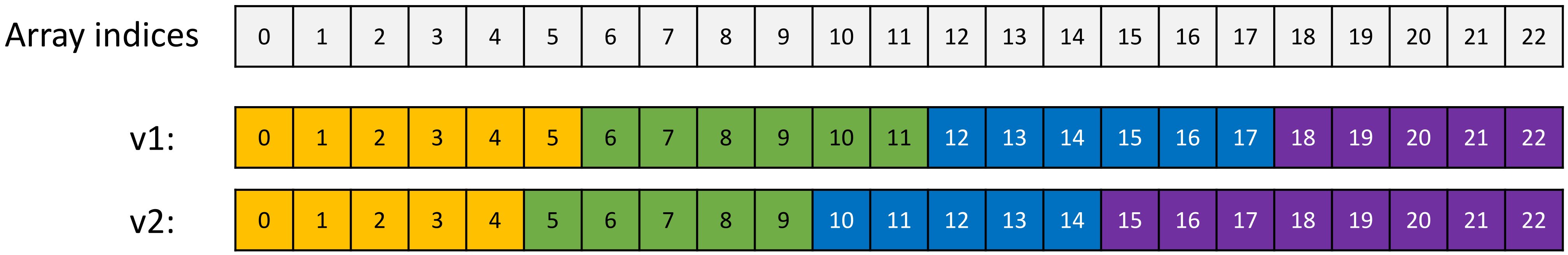
Some guiding principles:

- **A well-used system is never idle**
  - Having threads sit idle is wasteful
- **A well-used system does more work than overhead**
  - Synchronizing threads costs overhead

- **Synchronization is <u>necessary</u>, but expensive**
  - Don’t use it unnecessarily, but don’t be afraid to use it
  - Its expense also impacts how you should write code

Synchronization and Performance

- Synchronization causes idle threads and creates overhead
  - If threads start and finish around the same time, there is less idle time
- Example problem: 23 elements, 4 threads
  - Better to have one thread do more or less than the others?



Synchronization and Performance

- How much time will the different versions take with perfect parallelization?
- i.e., assume all threads:
  - Start at the same time
  - Take the same amount of time to update one element
- How much “work potential” is wasted?
  - v1: (1 idle thread)*(1 step)
  - v2: (3 idle threads)*(3 steps)
- **This is the major motivation behind *load balancing***

v1:

t	0	1	2	3
1	0	6	12	18
2	1	7	13	19
3	2	8	14	20
4	3	9	15	21
5	4	10	16	22
6	5	11	17	

v2:

t	0	1	2	3
1	0	5	10	15
2	1	6	11	16
3	2	7	12	17
4	3	8	13	18
5	4	9	14	19
6				20
7				21
8				22

Load Balancing with OpenMP

- How does OpenMP’s parallel-for distributes iterations of the for-loop?
- Scientific computing often feature nD arrays…
  - What if there are nested for-loops?
  - What if there are fewer iterations than threads?
- **What if some iterations cost more than others?**

More food for thought…

- **Can synchronization cause problems?**
- Consider the following code:

```
#pragma omp parallel
{
    thread_id = omp_get_thread_num();
    if thread_id == 0
        do_stuff();
    else
        #pragma omp barrier
}
```