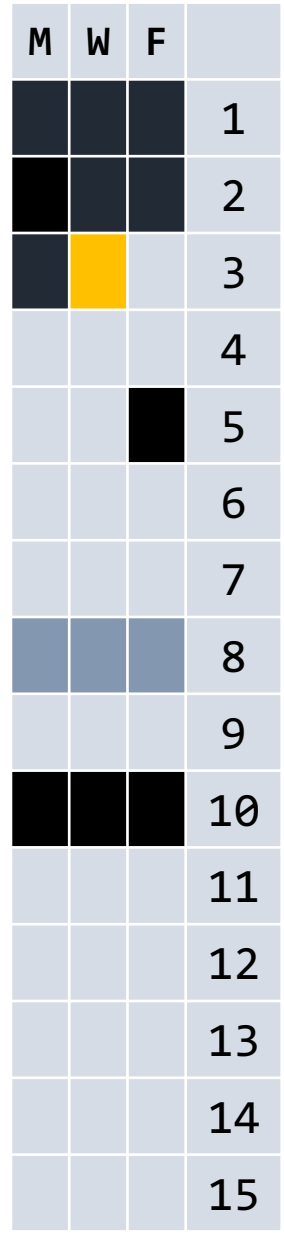
CMOR 421/521:
Threads and OpenMP

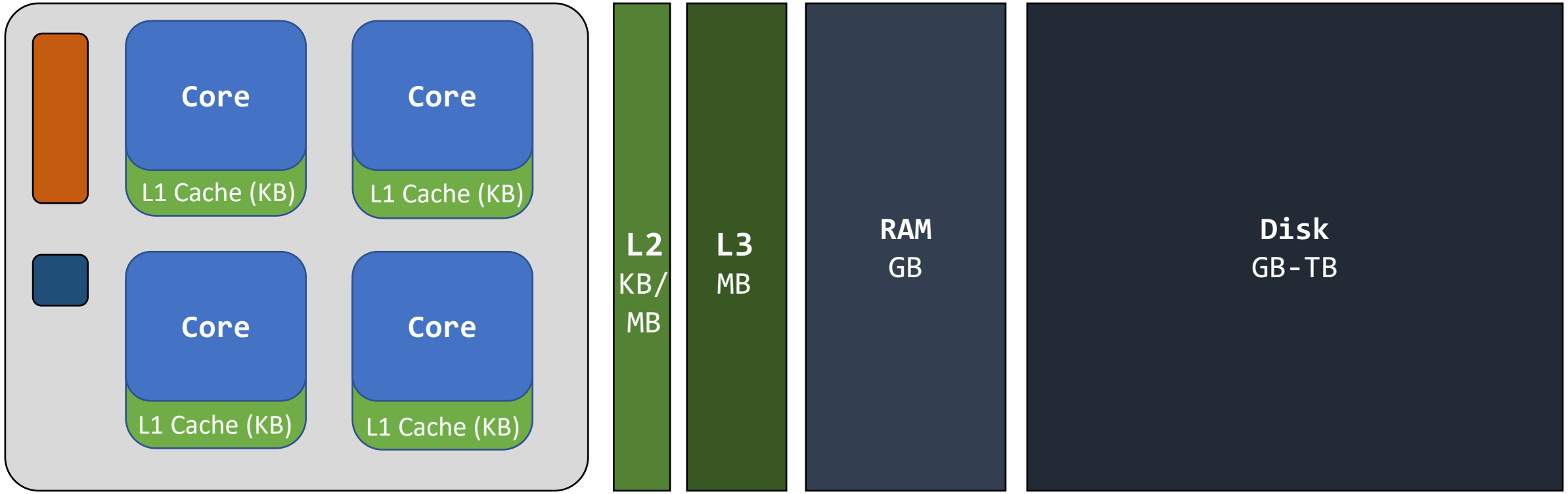


Topics

- What are threads, how do they work?
- Adding OpenMP to a program
- OpenMP for-Loops
  - Thread branching
  - Variable declarations
  - Load balancing

OpenMP: Shared Memory Paradigm

- Cores/**threads** have shared memory that they can all read/write
- Works for single computers/CPUs



Threads vs cores

- Cores are hardware, **threads** are not
  - A thread is like a subroutine or set of instructions
  - Created or joined (destroyed) during program execution
  - Threads are independent from each other and communicate through *shared variables* and synchronization

- Challenges:
  - Manual thread management is cumbersome
    - Creation/joining of threads
    - Required for imperative programming
  - Thread creation has significant **overhead**

Pthreads “hello world”

```
void* SayHello(void *foo) {
  printf( "Hello, world!\n" );
  return NULL;
}

int main() {
  pthread_t threads[16];
  int tn;
  for(tn=0; tn<16; tn++) {
    pthread_create(&threads[tn], NULL, SayHello, NULL);
  }
  for(tn=0; tn<16 ; tn++) {
    pthread_join(threads[tn], NULL);
  }
  return 0;
}
```

Compile using gcc –lpthread

Example from Aydin Buluc

What is OpenMP?

- **Open** specification for Multi-**P**rocessing
  - openmp.org – talks, examples, forums, etc.
  - Specification controlled by the OpenMP Architecture Review Board (ARB): a nonprofit for the OpenMP Spec
  - OpenMP 6.0 coming in 2024

- Motivation: capture common usage and simplify programming compared with POSIX-threads

For programmers: what is OpenMP?

- OpenMP is a portable, threaded, shared-memory programming *specification* with "light" syntax
  - Requires compiler-side implementation
- OpenMP will:
  - Allow users to separate a program into *serial regions* and *parallel regions* (less low-level than p-threads)
  - Hide stack management, provide synchronization constructs
- OpenMP will not:
  - Parallelize automatically
  - Guarantee speedup
  - Address issues such as data race conditions

Popularity/complexity of OpenMP

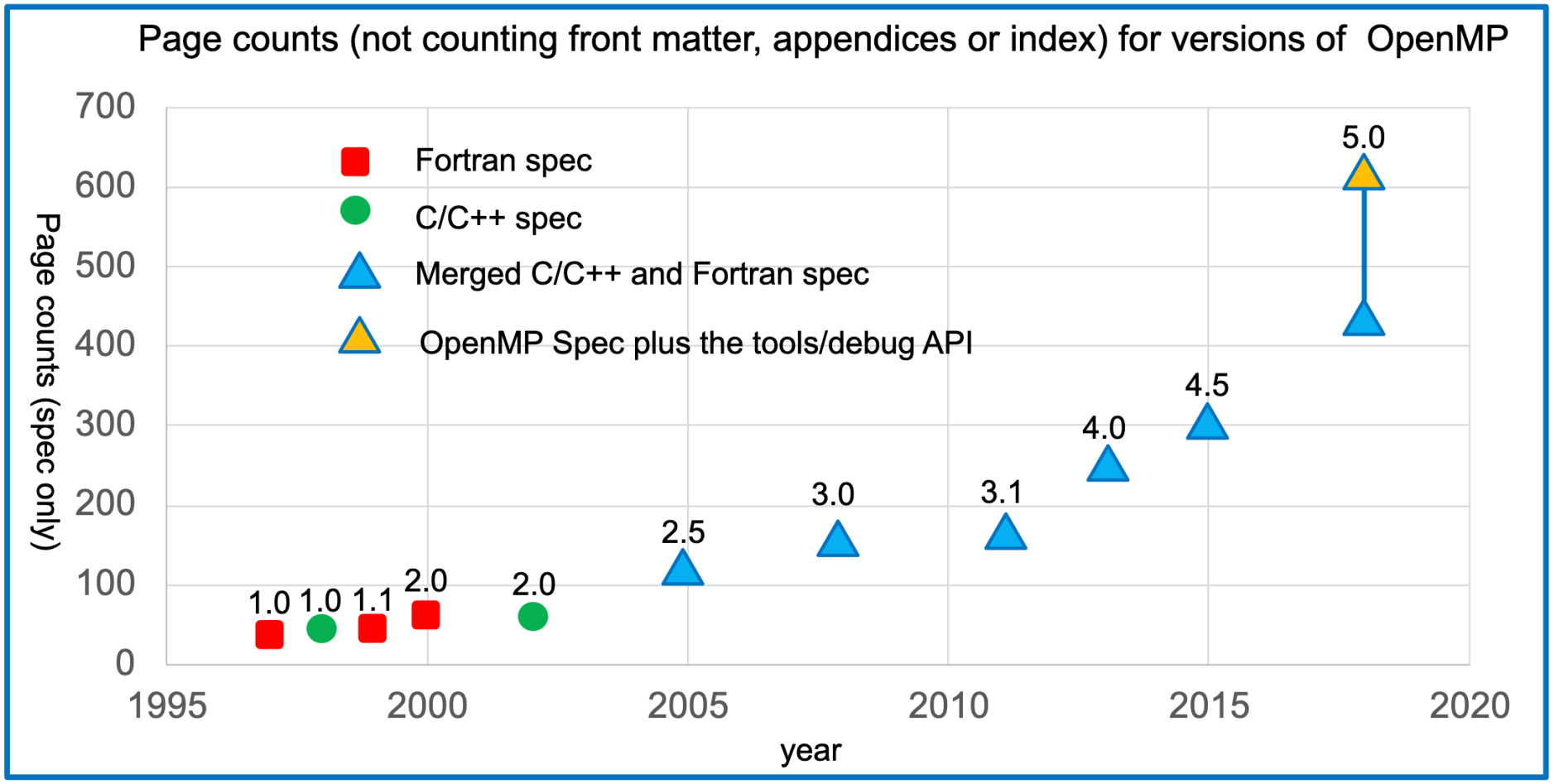


Figure from Tim Mattson

Most OpenMP programs only use ~20 directives and functions

OpenMP pragma, function, or clause	Concepts
#pragma omp parallel	Parallel region, teams of threads, structured block, interleaved execution across threads
int omp_get_thread_num() int omp_get_num_threads()	Create threads with a parallel region and split up the work using the number of threads and thread ID
double omp_get_wtime()	Speedup and Amdahl's law. False Sharing and other performance issues
setenv OMP_NUM_THREADS N	Internal control variables. Setting the default number of threads with an environment variable
#pragma omp barrier #pragma omp critical	Synchronization and race conditions. Revisit interleaved execution.
#pragma omp for #pragma omp parallel for	Worksharing, parallel loops, loop carried dependencies
reduction(op:list)	Reductions of values across a team of threads
schedule(dynamic [,chunk]) schedule (static [,chunk])	Loop schedules, loop overheads and load balance
private(list), firstprivate(list), shared(list)	Data environment
nowait	Disabling implied barriers on workshare constructs, the high cost of barriers, and the flush concept (but not the flush directive)
#pragma omp single	Workshare with a single thread
#pragma omp task #pragma omp taskwait	Tasks including the data environment for tasks.

What does OpenMP look like?

- High-level API for programming in C/C++ and Fortran
  - **Preprocessor (compiler) directives (~80%)**
    - #pragma omp […*stuff*…]
  - **Library Calls (~19%)**
    - #include <omp.h>
  - **Environment Variables (~1%):**
    - all caps, added to srun, etc.
    - E.g., export OMP_NUM_THREADS = 8

Adding OpenMP to a Program

- OpenMP is supported by the GNU compiler (gcc, g++)

- **In the code:** OpenMP is added as a library

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

- **Compiling:**

```
g++ -o <exe name> -fopenmp <.cpp files>
```

- **Running:**

```
export OMP_NUM_THREADS = #
```

Add “--cpus-per-task=…” flag to srun

Basic OpenMP Functions

- Getting the number of threads:
  - `num_threads = omp_get_num_threads();`
  - Gets the number of **active** threads in a parallel section

- Getting the active thread index
  - `thread_id = omp_get_thread_num();`

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

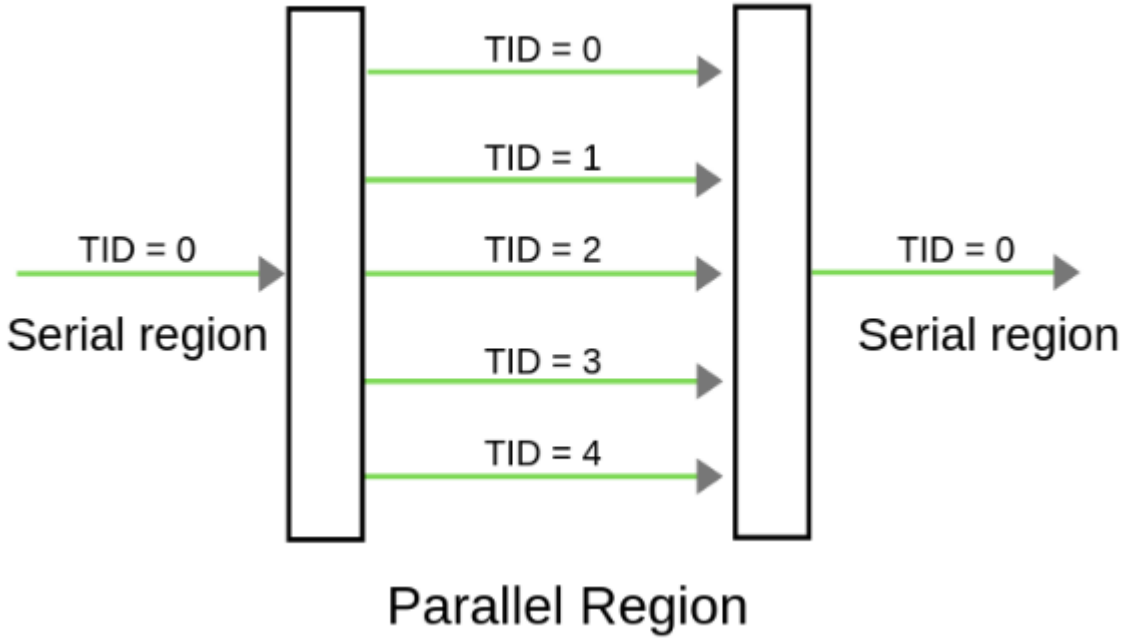


Image credit: https://www.geeksforgeeks.org/openmp-hello-world-program/

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.

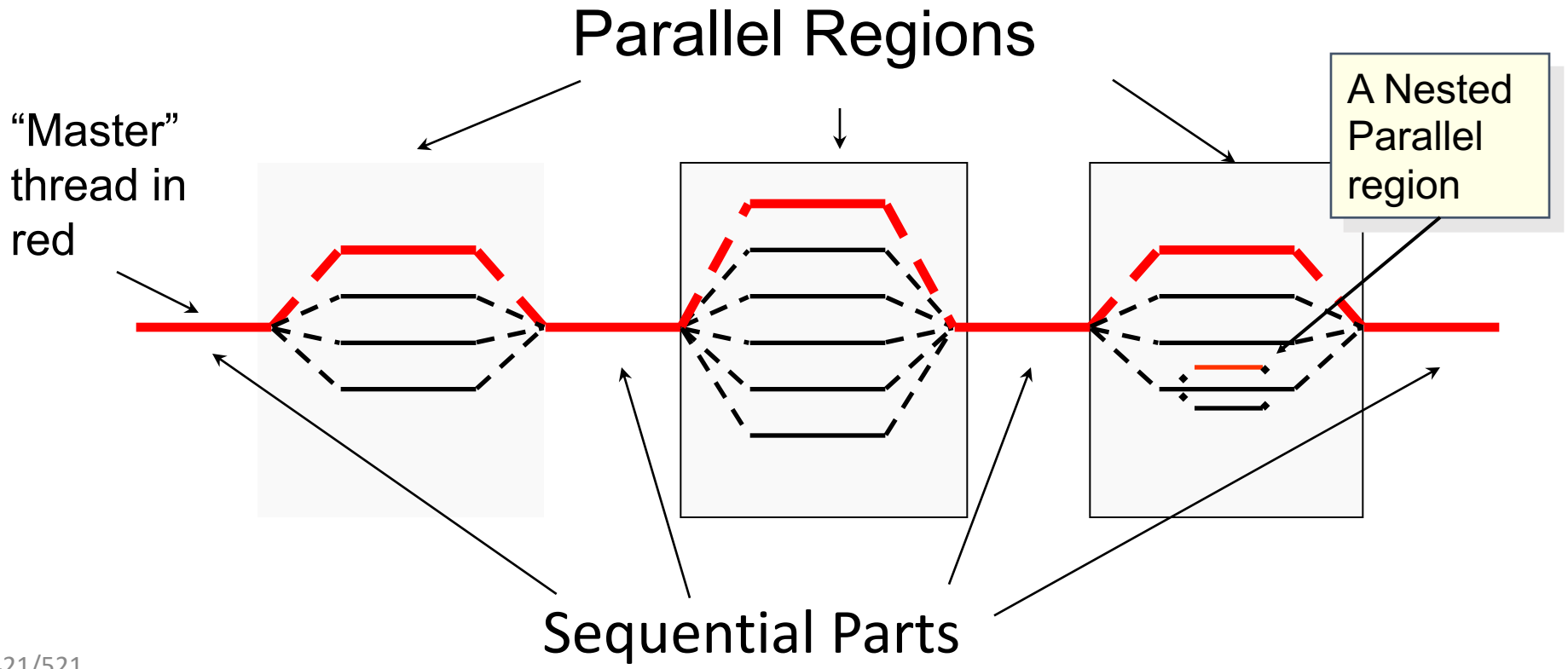


Figure from Aydin Buluc

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!

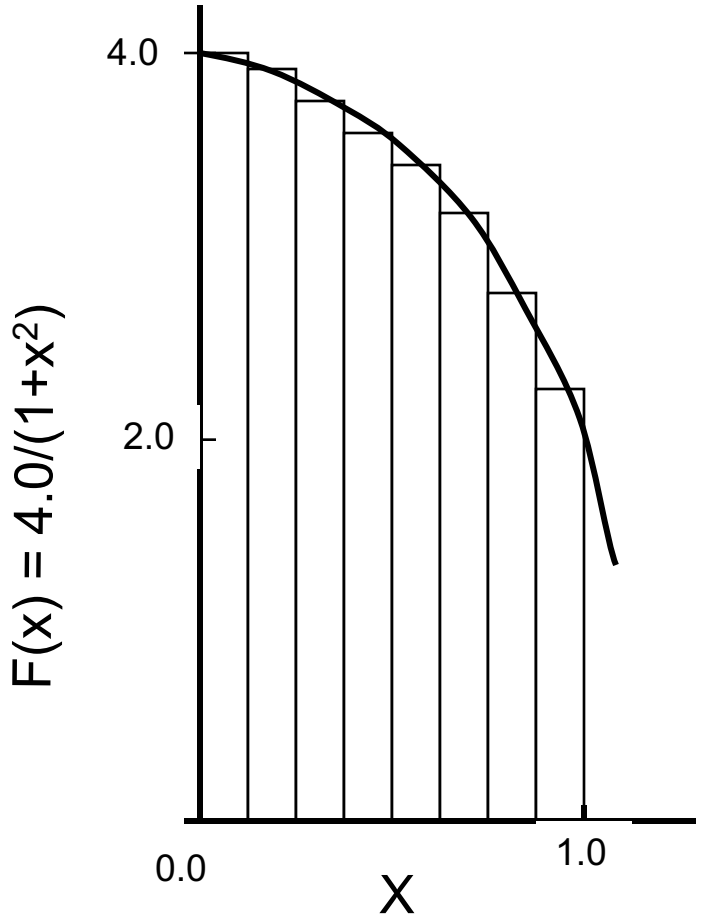
OpenMP: Thread management

- Note: you can't directly set the number of threads, you can only *request* a certain number of threads
  - export OMP_NUM_THREADS=… in the terminal
  - Can also request via set_omp_num_threads(…) at runtime
- You will get up to the number of threads set in the environment variable
  - An implementation can silently decide to give you a team with fewer threads.
  - Once a team of threads is established, the system will not reduce the size of the team.

How many threads vs cores?

- How many threads? We want a well-used system…
- **A well-used system is never idle**
  - If we limit the number of threads to the number of cores, we are short changing ourselves
- **A well-used system does more work than overhead**
  - If there are too many threads, the cores spend more of their time just switching programs/processes
- **Suggestion:** more threads than cores is okay, usually <2x number of cores

An example problem: numerical integration (aka quadrature)



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.

An example OpenMP code

- See integration_omp.cpp
  - All this code does is throw a “#pragma omp parallel” around the for loop.

```
#pragma omp parallel {
   for (int i = 0; i < num_steps; i++){
        double x = (i + 0.5) * step;
        sum = sum + 4.0 / (1.0 + x * x);
   }
}
```

What is going wrong?

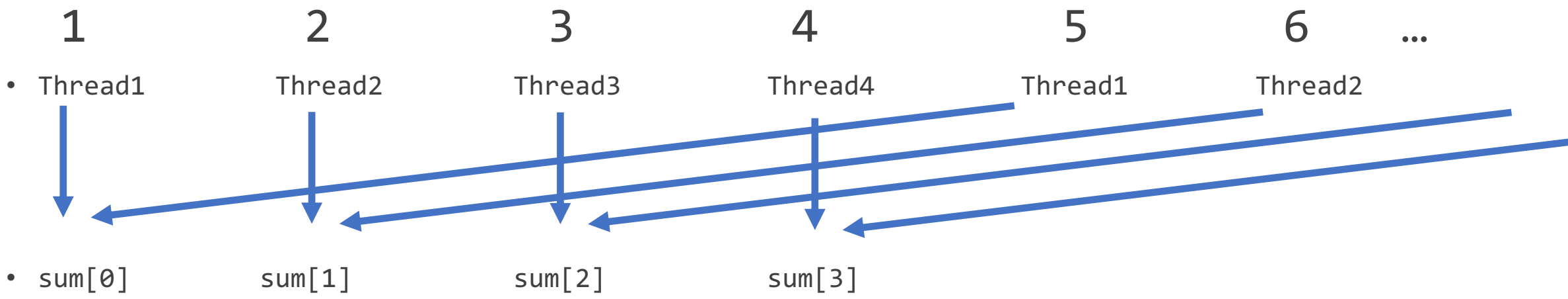
Problem: Data Race

- Problem is a *race condition* on the variable "sum" in the program
- A race condition or data race occurs when:
  - two processors/threads access the same variable, and at least one does a write.
  - The accesses are concurrent (not synchronized) so they could happen simultaneously

First try at a solution:

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

- Simple parallelization pattern: for 4 threads

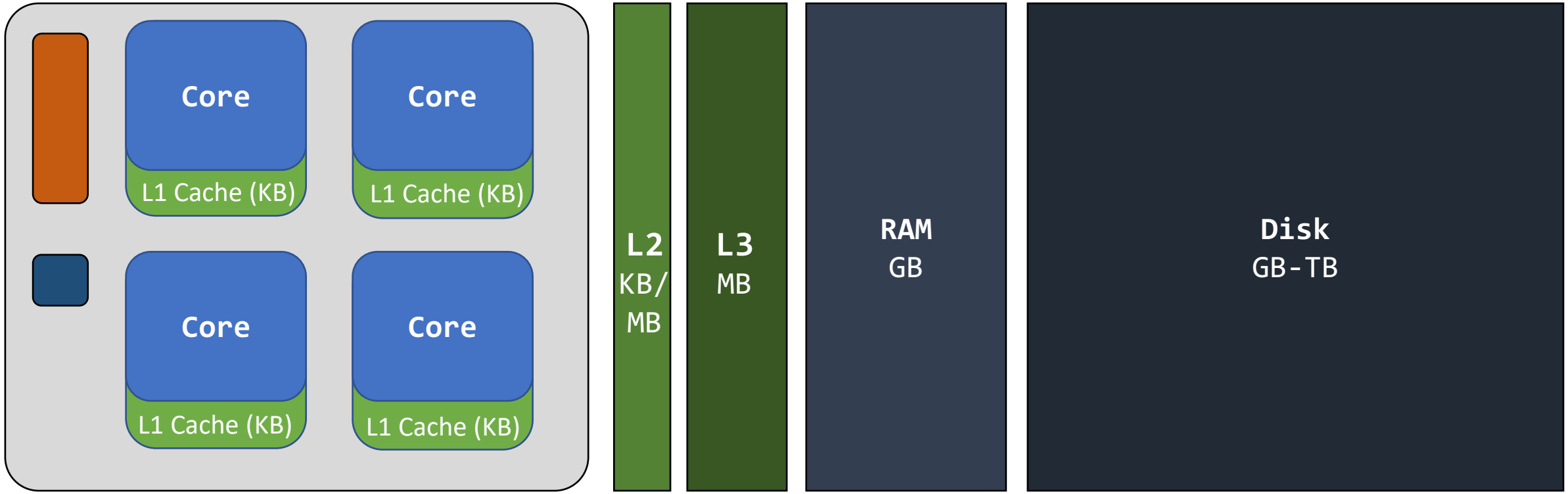


What about scalability?

- See integration_omp_no_race.cpp

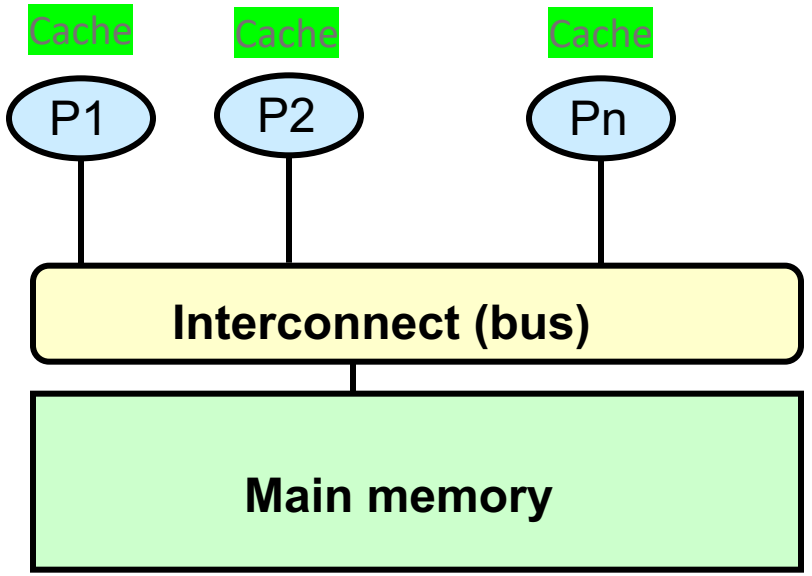
What is the issue?

- The memory hierarchy!



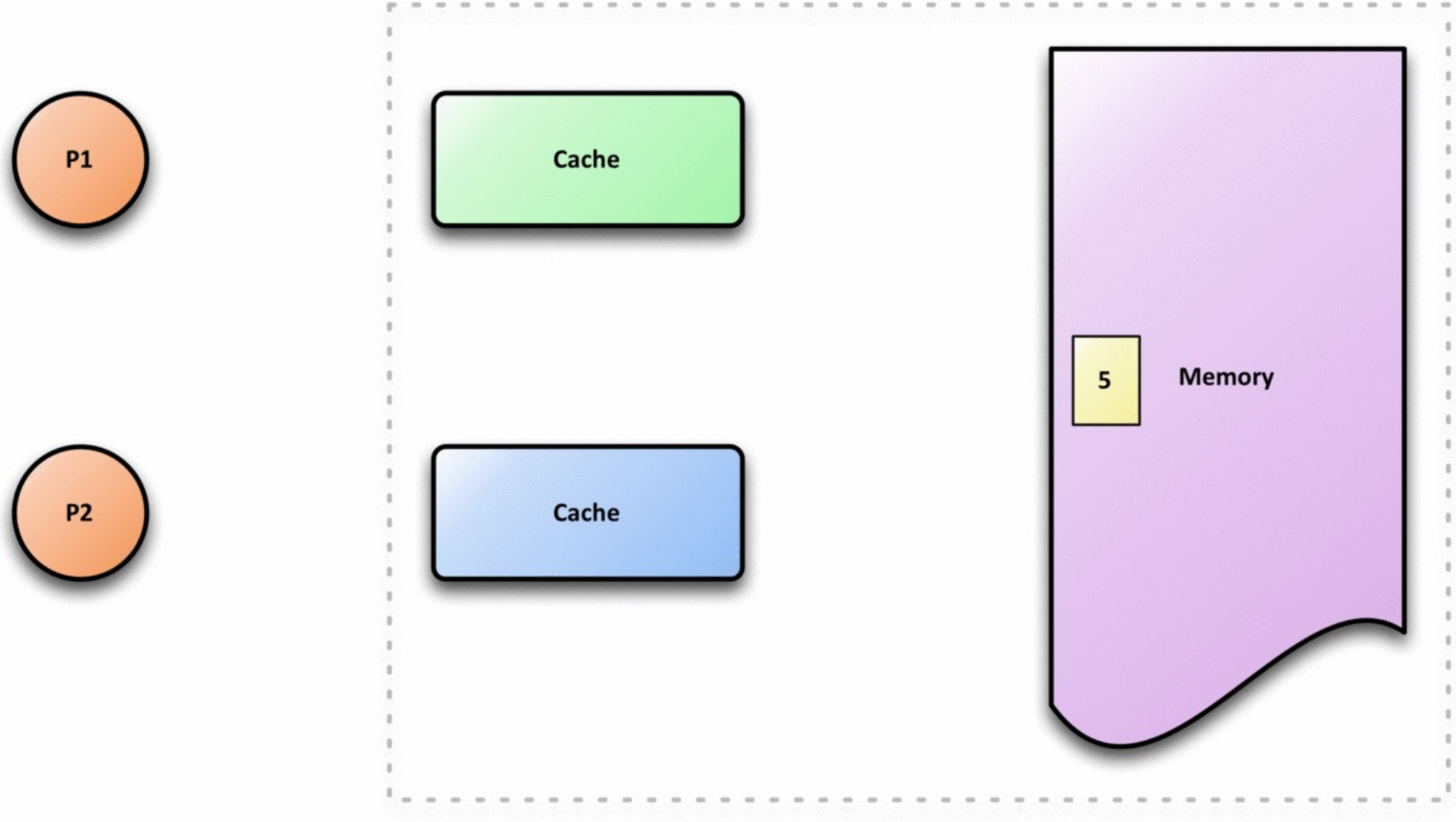
What is the issue?

- 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

Cache coherence protocol illustration



Gif from Wikipedia

Cache coherence protocols for a single processor

- **Write-through:** cache and main memory are both updated at the same time whenever a memory write occurs.
- **Write-back:** you update the cache first and update main memory at regular intervals
- Write-back is generally faster, especially with multiple processors.

Gif from Wikipedia

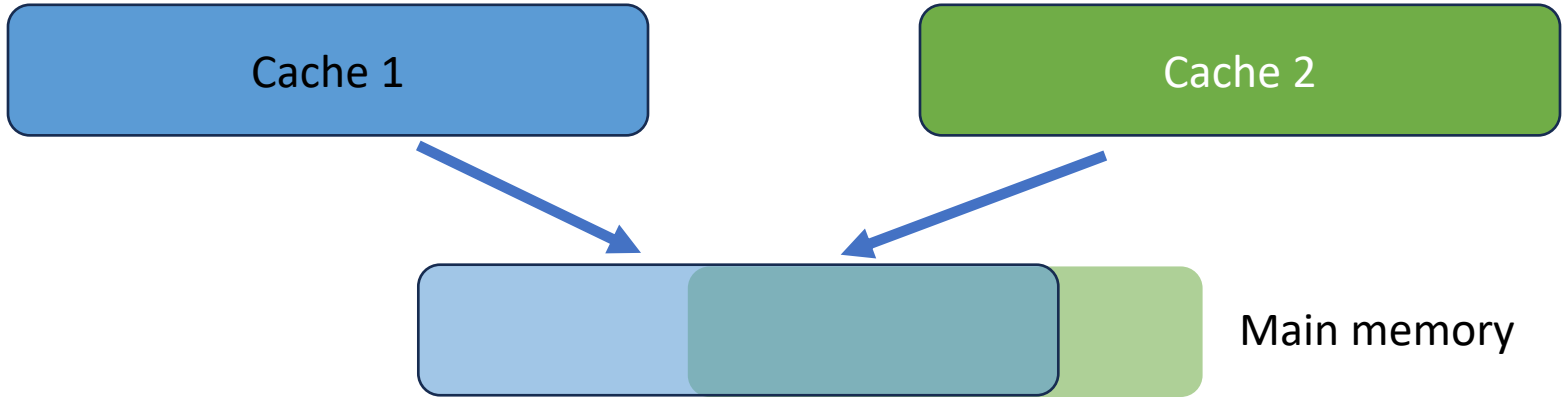
Cache coherence for multiple processors

- Caches can be independent of main memory, but how to keep them synchronized with each other?
- Memory bus is a broadcast medium
  - Caches contain information on which addresses they store
- Naïve approach: just broadcast every memory operation through the bus
  - Slow (lots more memory traffic!)
  - Scalability limited by bus bandwidth

Gif from Wikipedia

Cache coherence for multiple processors: snooping

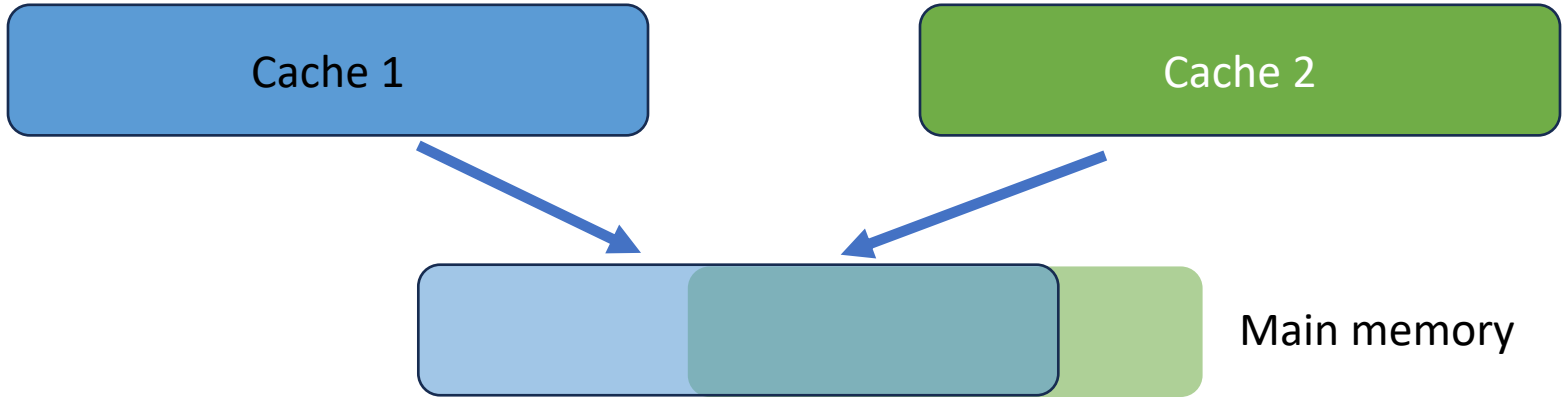
- Cache Controller “snoops” on all “relevant transactions” through the bus
- A transaction is *relevant* if it involves a cache block **currently contained in this cache**
  - Take action (invalidate, update, or supply value) to ensure coherence



Gif from Wikipedia

Why poor scaling?

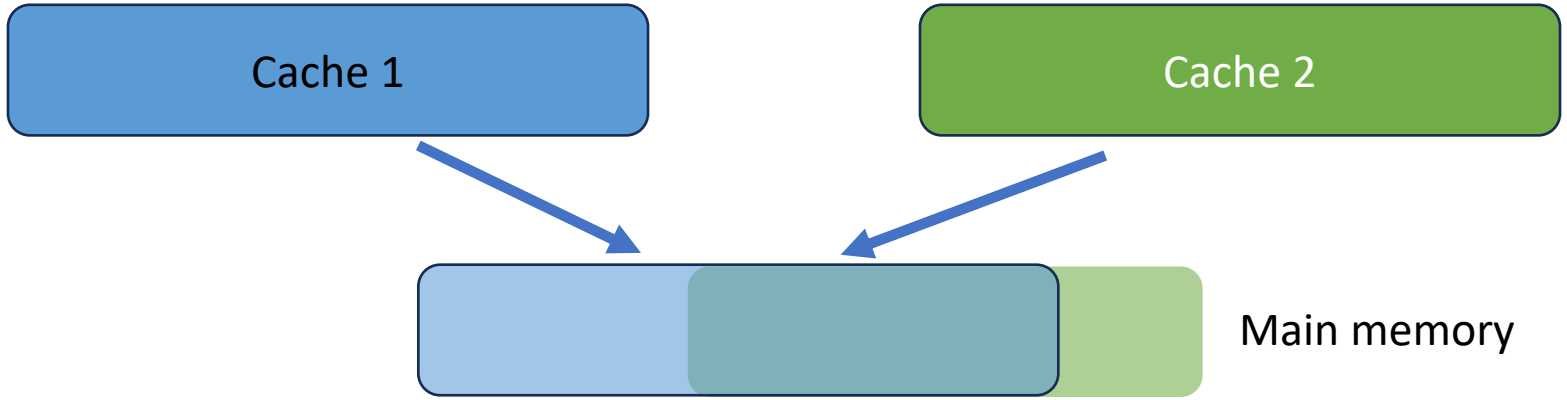
- "A transaction is relevant if it involves a **cache block** currently contained in this cache"
- Suppose you have two threads which access memory locations close to each other.
  - Thread 1 pulls a cache line into Cache 1
  - Thread 2 pulls a cache line into Cache 2



Why poor scaling?

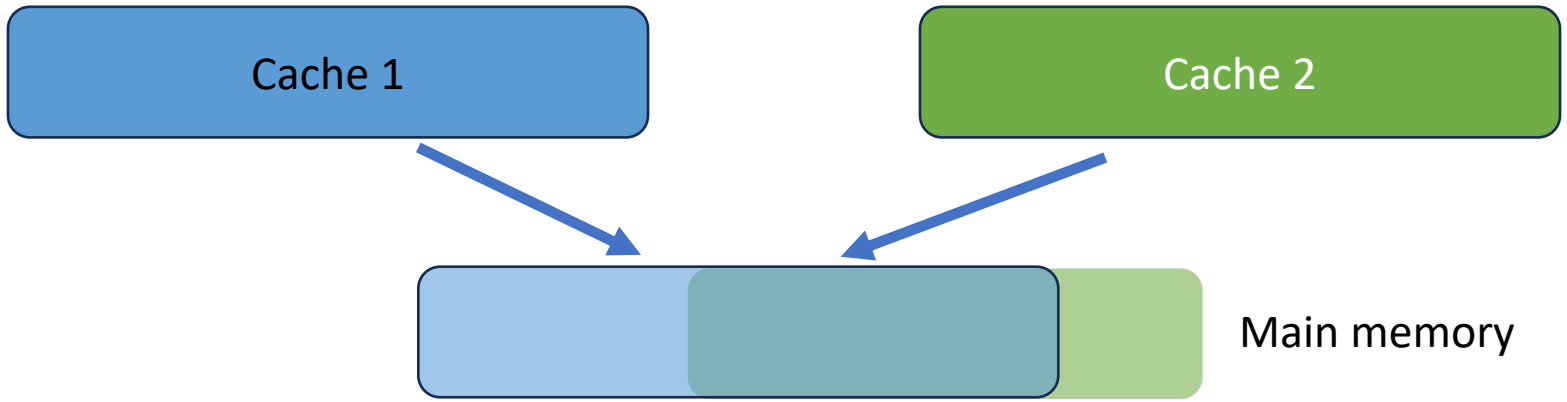
- "A transaction is *relevant* if it involves a cache block **currently contained in this cache**"
- Suppose you have two threads which access memory locations close to each other.
  - Thread 1 pulls a cache line into Cache 1
  - Thread 2 pulls a cache line into Cache 2

What happens if Cache 1 memory gets updated?



False sharing

- If independent data elements happen to sit on the same cache line, each update will cause the cache lines to “slosh back and forth” between threads …
- This is called “false sharing”.
  - Somewhat unique to OpenMP-type parallelism, where individual processors have their own caches.



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

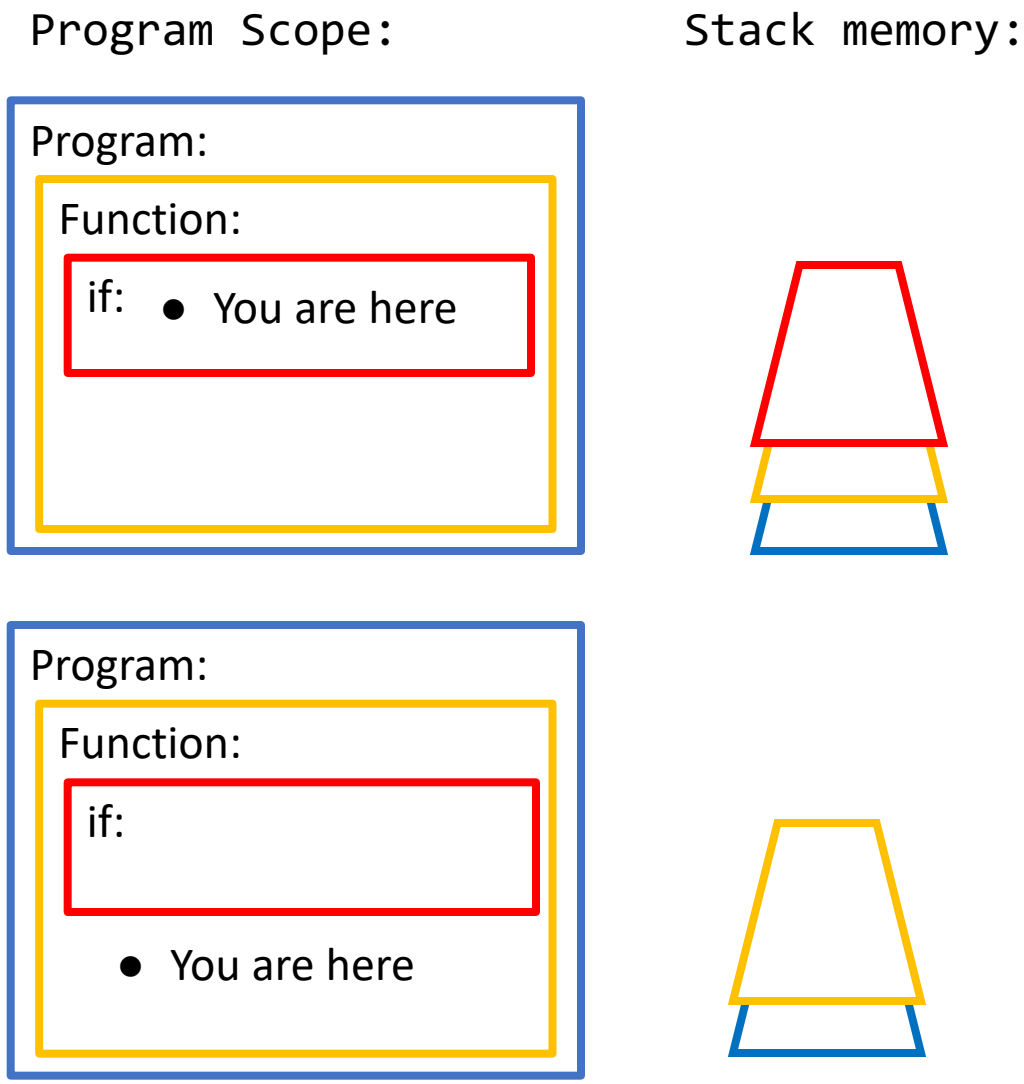
OpenMP: Variable Scope

- The parallel block has braces around it: it scopes variables

- OpenMP uses a shared memory paradigm: which variables are shared, which are not?

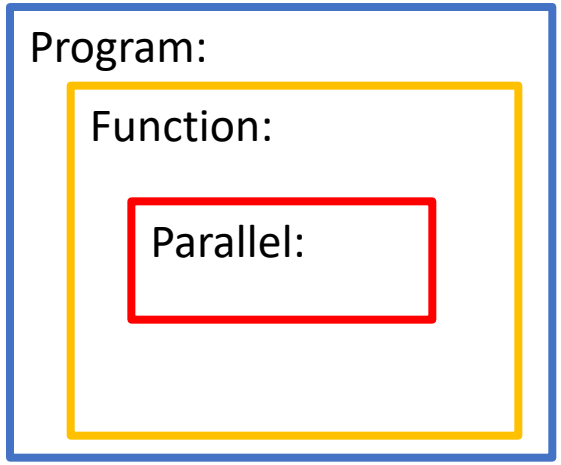
The Stack Revisited

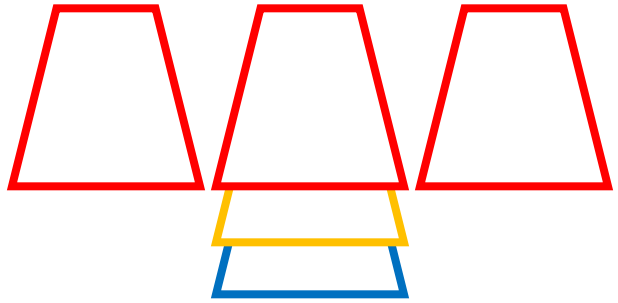
- The stack data structure is like a stack of cups: Last-In-First-Out (LIFO)
- Stack memory allocations operate within a single program as a stack
- Allocation and deallocation uses LIFO ordering to ensure scope



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

Preview of synchronization

- "#pragma omp critical" forces a type of **synchronization** between threads
  - Any code in this region will only have *one thread at a time running*
  - Serializes the code, but avoids race conditions
  - Similar to a "lock" in pthreads
- Other types of OpenMP synchronization: explicit and implicit barriers, ordered, atomics, etc.
- Christina will talk more about this (as well as scheduling options, load balancing) next week!

OpenMP: Scoping Clauses

- **firstprivate:** every thread get their own instance, + initialized

```
int x = 4;
#pragma omp parallel firstprivate(x)
{   // x will now be 4
    tid = omp_get_thread_num();
    do_stuff(tid, x);
}
```

Indexing is annoying: example 1

```
int id = omp_get_thread_num();
int nthreads = omp_get_num_threads();
for (int i = id; i < num_steps; i += nthreads){
    double x = (i + 0.5) * step;
    sum += 4.0 / (1.0 + x * x);
}
```

Indexing is annoying: example 2

Sequential code

```
for(i=0;i<N;i++)   { a[i] = a[i] + b[i];}
```

OpenMP parallel region

```
#pragma omp parallel
{
        int id, i, Nthrds, istart, iend;
        id = omp_get_thread_num();
        Nthrds = omp_get_num_threads();
        istart = id * N / Nthrds;
        iend = (id+1) * N / Nthrds;
        if (id == Nthrds-1)iend = N;
        for(i=istart;i<iend;i++)   { a[i] = a[i] + b[i];}
}
```

Worksharing via "omp parallel for"

```
#pragma omp parallel
#pragma omp for
        for(i=0;i<N;i++)   { a[i] = a[i] + b[i];}
```


Code from Aydin Buluc

OpenMP: for-loops

- Another OpenMP directive: **parallel for**
- Takes a for-loop and splits its iterations *evenly* amongst all threads
  - Can ask OpenMP to split the work unevenly using clauses (useful in applications such as sparse matrix multiply)
- **Two ways to define:**
  - Nested in a parallel block
  - Standalone

OpenMP: Parallel for

- **Version 1:** Nested in a parallel block

```
#pragma omp parallel
{
    #pragma omp for
    for(int i = 0; i < n; i++) {
        A[i] += i;
    } // Note only the word “for” was used
}
```

OpenMP: Parallel for

- **Version 2:** Standalone

```
#pragma omp parallel for
for(int i = 0; i < n; i++) {
    A[i] += i;
}
// Note now we use “parallel for” and
// the parallel block just uses the for’s
// braces
```

Example: integration

- See integration_omp_parallel_for.cpp

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
- **lastprivate:** same as private but the value from the thread that executes the last iteration should be returned to the original variable (for-loops)

OpenMP: Scoping Clauses

- **lastprivate:** every thread gets their own instance (not initialized). The value at the final **iteration** is stored in the shared variable x.

```
int x = 4;
#pragma omp parallel for lastprivate(x)
for(int i = 0; i < n; i++) {
    // x can be anything
    x = i + 1;
}
// x will equal n
```

OpenMP: for-loops

- Why use the nested notation?
- There are other OpenMP directives
- Sometimes it makes more sense to nest the “for” directive inside of a single parallel block with other directives
- **DO NOT modify the index variable** of a parallel for loop that OpenMP is acting on
  - It needs to just be a regular, vanilla for-loop

OpenMP: Nested Parallel Environments?

- Nested parallel environments will try to spawn more threads
- Typically not what we want

```
#pragma omp parallel for
for(int i = 0; i < n; i++) {
    #pragma omp parallel for
    for(int j = 0; j < n; j++) {
        do_stuff(A[i,j]);
    }
}
```

OpenMP: Nested Loops

- Sometimes we have nested for-loops though, and the first dimension may not be very large
- Can use the “collapse” clause

```
#pragma omp parallel for collapse(2)
for(int i = 0; i < n; i++) { // n small
      for(int j = 0; j < n; j++) {
         do_stuff(A[i,j]);
     }
}
```

How to Divide Work

- OpenMP for loops split the work evenly between all threads
  - A small number of iterations relative to the number of threads can be suboptimal when the work per iteration is small. Super bad: more cores than iterations
- What if the amount of work per iteration is not the same?

```
#pragma omp parallel for
for(int i = 0; i < n; i++) {
    for(int j = 0; j < i; j++)
        do_stuff();
}
```

Load Balancing

- **"…a well-used system is never idle…"**
  - We want workers to have similar amounts of work so they complete their work around the same time
  - Otherwise, our program's reserved workers may sit around with nothing to do, which is a waste of computing resources
- Trying to evenly distribute the amount of work across all workers is called ***load balancing***

Another Thought

- What about operation order dependence?

```
// Our friends the Fibonacci numbers, now
// computed iteratively
f[0] = 1;
f[1] = 1;

for(int i = 2; i < n; i++) {
    f[i] = f[i-1] + f[i – 2];
}
// Can’t compute f[i] until after f[i-1]
```

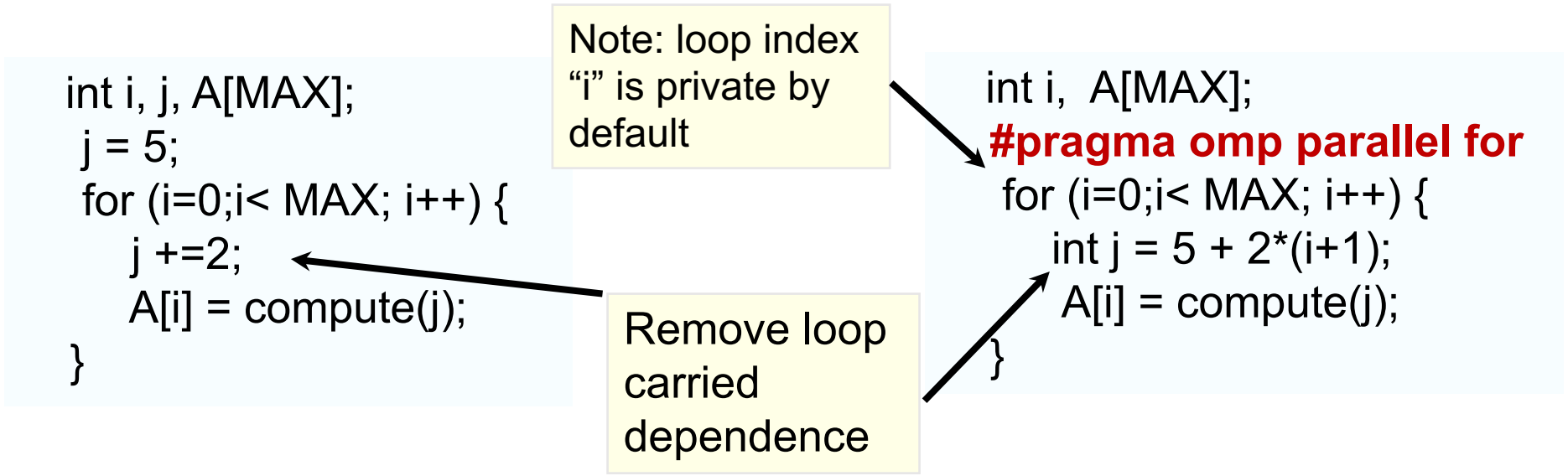
Guidelines for OpenMP

- Basic approach
  - Find compute intensive loops
  - Make the loop iterations *independent* so they can safely execute in any order without loop-carried dependencies
  - Test the appropriate OpenMP directives

- Caveats: need to account for load balancing (scheduling clauses can help), certain special operations

Guidelines for OpenMP

- A toy example:



Reduction operations

- How do we handle this?
  - Cannot remove the loop-carried dependency

```
double  avg=0.0, A[N];
  for (int i=0; i < N; ++i) {
        avg += A[i];
  }
  avg = avg / N;
```

- This is known as a reduction
  - Norms, averages, min/max, etc.
- Support for reduction operations is included in most parallel programming environments (OpenMP, MPI, CUDA).

Reductions in OpenMP

```
double  avg=0.0, A[N];
for (int i=0; i < N; ++i) {
        avg += A[i];
}
avg = avg / N;
```

```
double  avg=0.0, A[N];
#pragma omp parallel for reduction (+:avg)
for (int i=0; i < N; ++i) {
        avg += A[i];
}
avg = avg / N;
```

- OpenMP reduction clause: “reduction (op : list)”
- Inside a parallel or a work-sharing construct:
  - A local copy of each list variable is made and initialized depending on the “op” (e.g. variable initialized to 0 for “+”).
  - Updates occur on the local copy.
  - Local copies are reduced into a single value and combined with the original global value.
- “list” must be shared variables in the parallel region.

OpenMP: Reduction operands and initial values

- Many different associative operands can be used with reduction:

Operator	Initial value
+	0
*	1
-	0
min	Largest pos. number
max	Most neg. number

C/C++ only	
Operator	Initial value
&	~0
|	0
^	0
&&	1
||	0

- Initial values are the ones that make sense mathematically.
- There are also Fortran-specific reduction ops.

Example: integration with reduction

- See integration_omp_reduction.cpp

- Compare timing with other 7 versions of integration*.cpp

OpenMP so far

- #pragma omp parallel/for
- Basic scoping
  - Shared/private variables
- Basic synchronization
  - #pragma omp critical
  - Implicit barriers
- Reduction operations

Can handle many “easily parallelizable” programs.

What is left?

- Parallel scalability cares about parallelizable vs not parallelizable (sequential) parts
- Parallelizable does not imply:
  - Balanced work load (load balancing)
  - Independent operations (execution order)
- These are important real-world considerations

Next Time (feat. Christina Taylor)

- Details of data sharing/scoping
- Load balancing options for OpenMP parallel for loops
- Ensuring correct execution order:
  - Synchronization
  - Barriers
  - More detail: thread safety and race conditions