CMOR 421/521
Shared memory parallelism

Jesse Chan

Some basic parallel theory

Computing trends

Slower processors but more of them

- Trend is towards slower CPUs
  - Higher CPU clock speeds generate more heat and consume more power
  - Supercomputers consume huge amounts of energy
  - Some of that energy is from computing, some is from cooling; industry is experimenting with hot server rooms, water cooling, etc because of this
- Easier to manufacture: fast CPUs are more sensitive to manufacturing defects, easier to fabricate lots of slower CPUs instead.

Challenge of parallelism

- Not all work is parallelizable; “9 women can’t make a baby in 1 month”
- If a small amount of work is serial / sequential, this limits scalability
- Suppose p, s are “parallelizable” and “serial” fractions of work ($s = 1 - p$).
  - Let n denote the number of processors.
- Total runtime of a program is $T(n) = T_s(n) + T_p(n)$
  - $T_s(n)$ is the time spent on the sequential portion
  - $T_p(n)$ is the time spent on the parallel portion

Some scaling estimates

- Total runtime of a program is $T(n) = T_s(n) + T_p(n)$
  - Assume sequential portion is independent of n, so $T_s(n) = sT(1)$
  - Assume parallel portion is perfectly parallelizable, so $T_p(n) = \frac{p}{n}T(1)$
- Total runtime is then $T(n) = \left(s + \frac{p}{n}\right) T(1).$
- Can analyze *scaling* behavior as the number of processors increases.

Amdahl's law: strong scaling

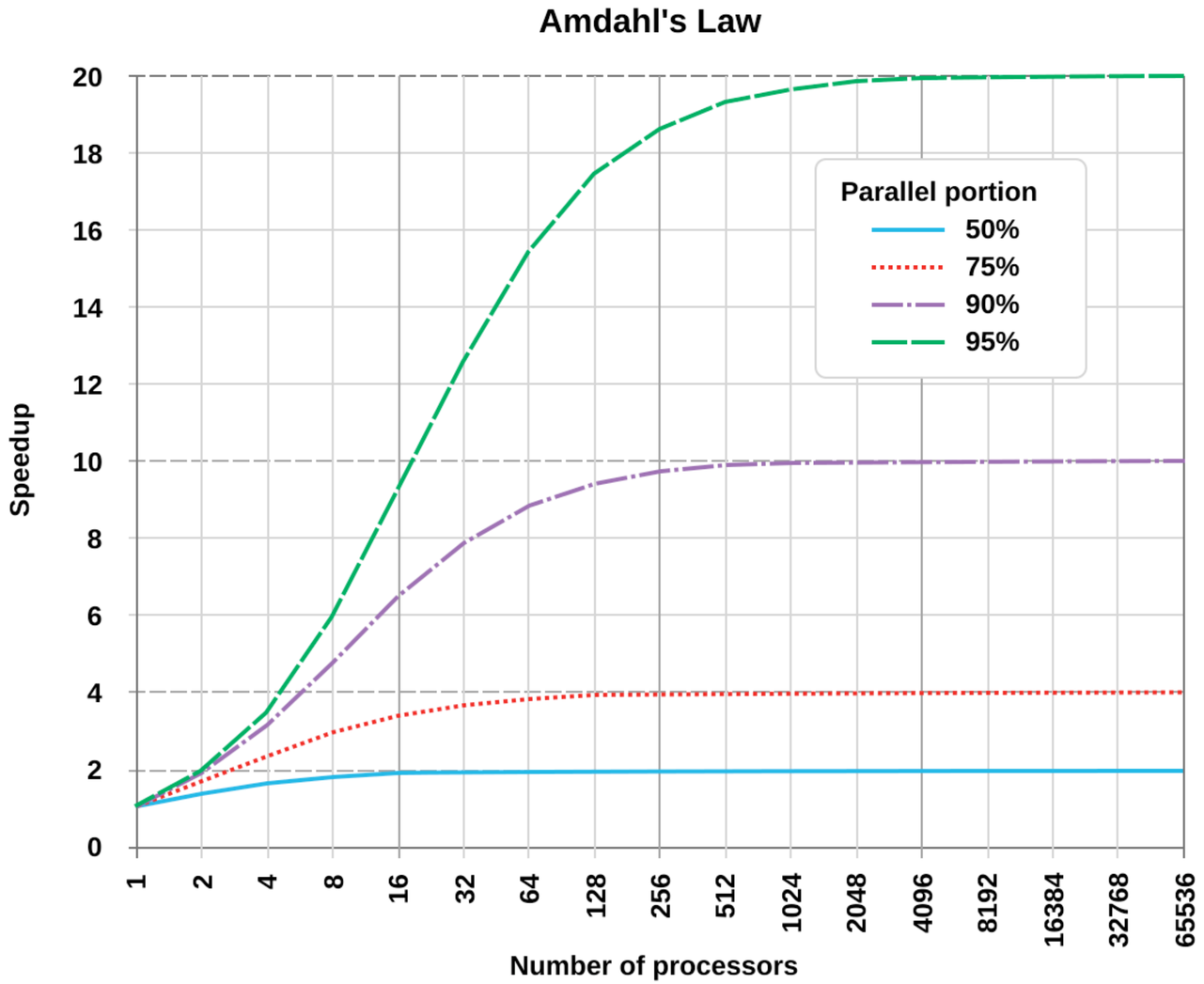
- There are different ways of measuring speed-up related to scaling
  - **Strong scaling**: amount of work is fixed
  - **Weak scaling**: amount of work scales with the number of processors
- For strong scaling, parallel speed-up $S(n) = \frac{T(1)}{T(n)}$ is the main metric
- Substitute in $T(n) = \left(s + \frac{p}{n}\right) T(1)$ to get Amdahl's law:
  - $S(n) = \frac{T(1)}{T(n)} = \frac{s+p}{s+p/n} = \frac{1}{s+p/n} < \lim_{n\to\infty} S(n) = \frac{1}{s}$

What does Amdahl's law mean?

- Parallel speed-up $S(n) = \frac{T(1)}{T(n)}$ is bounded by the sequential work
  - $S(n) = \frac{T(1)}{T(n)} = \frac{s+p}{s+p/n} = \frac{1}{s+p/n} < \lim_{n\to\infty} S(n) = \frac{1}{s}$
- Cannot expect arbitrarily large parallel speed-ups; eventually you will hit a sequential bottleneck.
  - Suppose only 50% of a program is parallelizable; maximum parallel speed-up is at most 2.
  - If 80% is parallelizable, then maximum parallel speed-up is 1 / .2 = 5.
  - Relatively modest speed-ups unless *everything* is parallelizable.

Amdahl’s law implies the sequential part matters

- Speeding up the sequential part asymptotically improves max speed-up by same factor
- Implies that the parallel *efficiency* $E(n) = \frac{S(n)}{n}$ must eventually decrease as n increases.
- Note: Amdahl assumes fixed parallel resources (potentially good), ignores overhead from synchronization (bad)



Weak scaling

- Strong scaling increases the number of processors n but keeps work fixed
- Weak scaling increases both number of processors and work together
  - Performance if work per processor is held constant
- Why weak scaling? For many programs, the sequential part does not increase as the problem size increases. Examples:
  - Monte Carlo simulations; minimal setup, “embarrassingly” parallel
  - Numerical simulations: often dominated by multiple matrix multiplies. Serial part typically negligible or scales very slowly with problem size.

Relating weak and strong scaling

- Weak scaling changes the assumptions of strong scaling. Suppose the total work depends on the problem size N
  - Total work: $W(N) = W_s(N) + W_p(N)$.
  - Recover serial/parallel parts: $1 = \frac{W_s(N)}{W(N)} + \frac{W_p(N)}{W(N)} = s_N + p_N$
- Strong scaling assumes that N is fixed; weak scaling assumes the serial part of the work $W_s(N)$ does not change with N.
- Only way this can happen is if s_N decreases and p_N increases with N.

Weak scaling: Gustafson's law

- Suppose the problem size is proportional to number of processors n
  - This implies that $T(1) = (s + np)T(n)$, e.g., cost on one processor is serial part plus n times the parallel part.
- Assume without loss of generality $T(n) = 1$.
  - Then parallel speed-up is $S(n) = \dfrac{T(1)}{T(n)} = \dfrac{s + np}{s + p} = s + np$
- In other words, if we increase the problem size with n, parallel speed-up increases *linearly* with respect to the number of processors.

Caveats of Gustafson's law

- Gustafson's law: under weak scaling, parallel speed-up is $S(n) = s + np$
  - Speed-up can be arbitrarily large as long as we grow the problem size!
  - Weak speed-up measures how much time it would take for one processor to do the work of n processors.
- Assumes that the serial part is independent of problem size
- Assumes that the cost scales *linearly* with respect to number of processors. This may not be true for algorithms which are not perfectly scalable or have nonlinear complexity.

Strong and weak scaling

- Both strong and weak scaling are useful measures of parallel efficiency
  - Strong vs weak scaling: “9 women can’t make a baby in 1 month, but could possibly make 81 babies in 9 months?”
  - Good weak scaling is usually easier to achieve.
  - Strong scaling is still important! Shows how large your problem size must be on each individual worker/processor.
- Both measures can be gamed!
  - Imagine if the parallelizable part of a program is scalable but implemented in a highly inefficient manner; strong / weak scaling limits may not be observed until the number of processors is very large.

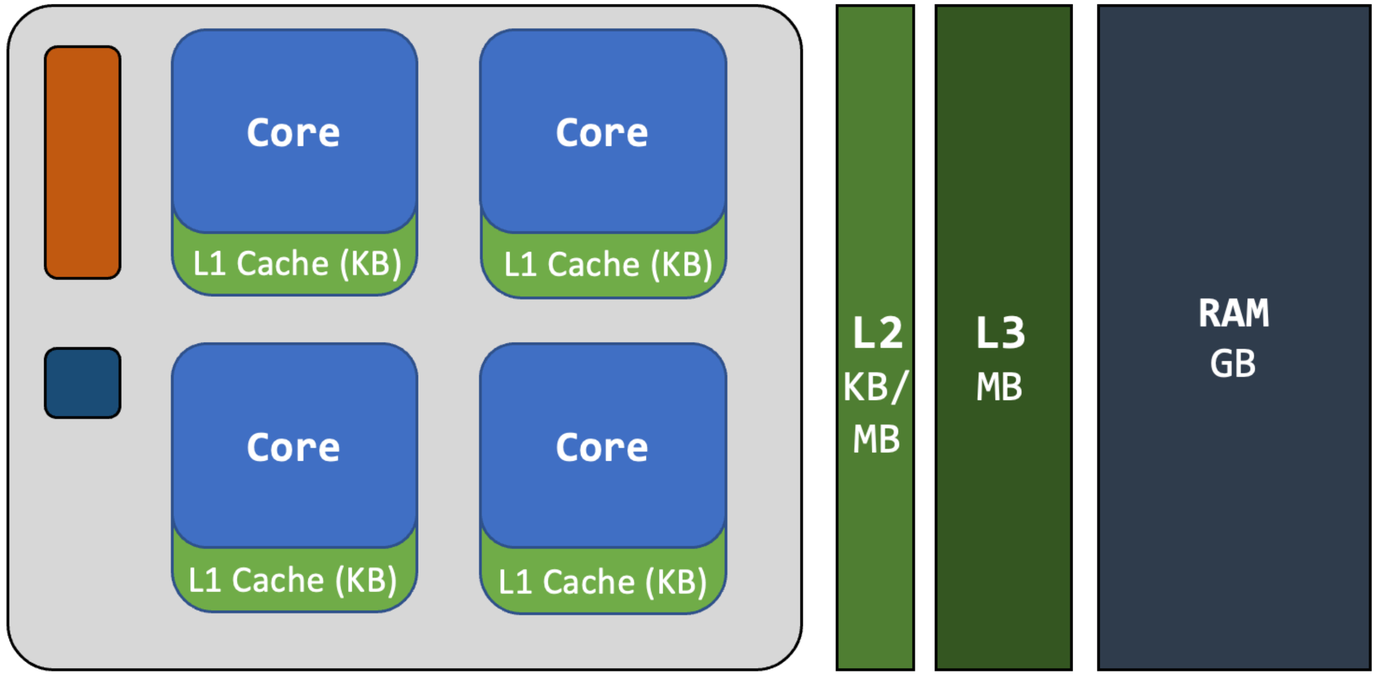
Parallelism and communication

- Parallel work can be split into **computation** and **communication**
- Parallel computation is usually pretty straightforward: divide work up (as evenly as possible) among multiple workers.
- However, there are several ways to implement communication among workers, each of which corresponds to a different parallel “paradigm”.
- We’ll focus on **shared memory** and **distributed memory** parallelism.
  - **Shared memory:** multiple workers communicate through centralized storage. Appropriate for multi-core CPUs (OpenMP) and GPUs.
  - **Distributed memory:** workers communicate by explicit *message-passing.* Appropriate for clusters, uses MPI: message passing interface

Shared memory parallelism and OpenMP

Shared memory programming: threads vs cores

- Cores/threads have shared memory that they all read from or write to.
- Cores are related to hardware, threads are related to software
  - One thread is similar to an independent subroutine
  - Threads get created and joined / destroyed when the program runs
  - Threads are independent from each other and communicate through shared variables and synchronization.



Pthreads: low level thread management

- **P**osix (Portable Operating System Interface) **threads**
- Requires manual thread management (creation/ joining of threads)
- Can avoid unnecessary thread creation, which has significant computational overhead.

```
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

OpenMP

- **Open** specification for **M**ulti-**P**rocessing.
  - Specification is determined by the OpenMP Architecture Review Board (ARB): a nonprofit specifically for OpenMP.
  - Actively managed and updated; OpenMP 6.0 released Nov. 2024.
  - See openmp.org for talks, examples, forums, etc.
- Motivation: capture common multi-threaded programming patterns and simplify implementation compared to POSIX-threads.
  - OpenMP syntax additionally aims to be “light” and relatively non-intrusive.

How does OpenMP work?

- Because it's a *specification*, OpenMP requires compiler-side implementation.
  - OpenMP typically converts OpenMP code to lower level Pthreads code.
  - Behavior laid out in the specification should be consistent across compilers.
- Main idea: rather than fork-joining individual threads, OpenMP introduced the idea of *serial and parallel regions.*
  - OpenMP hides thread management from the user and provides convenience methods for common thread synchronization/communication patterns.
- OpenMP *does not* guarantee automatic parallel speedup or memory safety issues (e.g., data race conditions).

An OpenMP “Hello world”

- #pragma (e.g., a directive or “pragmatic”) specifies an OpenMP parallel *region.*
  - Everything inside the parallel region is run on every single thread.
  - Threads execute in any order.
- The #pragma is a preprocessor directive; if a compiler doesn’t support OpenMP, it gets ignored.
- Can query for total (active) number of threads and local thread id number.

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

using namespace std;

int main()
{
#pragma omp parallel
  {
    cout << "Hello world " << endl;
  }
  return 0;
}
```

An OpenMP “Hello world”

- #pragma (e.g., a directive or “pragmatic”) specifies an OpenMP parallel *region.*
  - Everything inside the parallel region is run on every single thread.
  - Threads execute in any order.
- The #pragma is a preprocessor directive; if a compiler doesn’t support OpenMP, it gets ignored.
- Can query for total (active) number of threads and local thread id number.

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

using namespace std;

int main()
{
#pragma omp parallel
  {
    int tid = omp_get_thread_num();
    cout << "Hello world from thread " <<
      tid << " / " << omp_get_num_threads()
      << " threads" << endl;
  }
```

How to build an OpenMP program

- On NOTSx, compile OpenMP via “g++ **-fopenmp** hello_omp.cpp”
  - Need “include <omp.h>” to define OpenMP directives/functions.
  - “-fopenmp” links OpenMP headers/library but also includes compiler symbols necessary to process parallel directives.
- On your own machine, installing gcc/g++ should also install OpenMP
  - On Macs, can install gcc/g++ via Xcode command line tools or Homebrew.
  - I use Homebrew’s “g++-14” on my laptop; Apple’s “g++” redirects to a version of the “llvm” compiler which doesn’t support -fopenmp.
- Demo…

How to configure OpenMP threads

- How to configure the number of threads? Three (four?) ways:
  - Within an OpenMP program:
    - “omp_set_num_threads(4)” sets the number of threads globally
    - “#pragma omp parallel num_threads(4)” sets the number of threads for a specific parallel region
  - Outside of an OpenMP program using environment variables: “export OMP_NUM_THREADS = …”
- On NOTSx:

```
srun --pty --partition=scavenge --reservation=cmor421 –
ntasks=1 –cpus-per-task=4 –mem=1G --time=00:30:00 $SHELL
```

Example: setting the number of threads

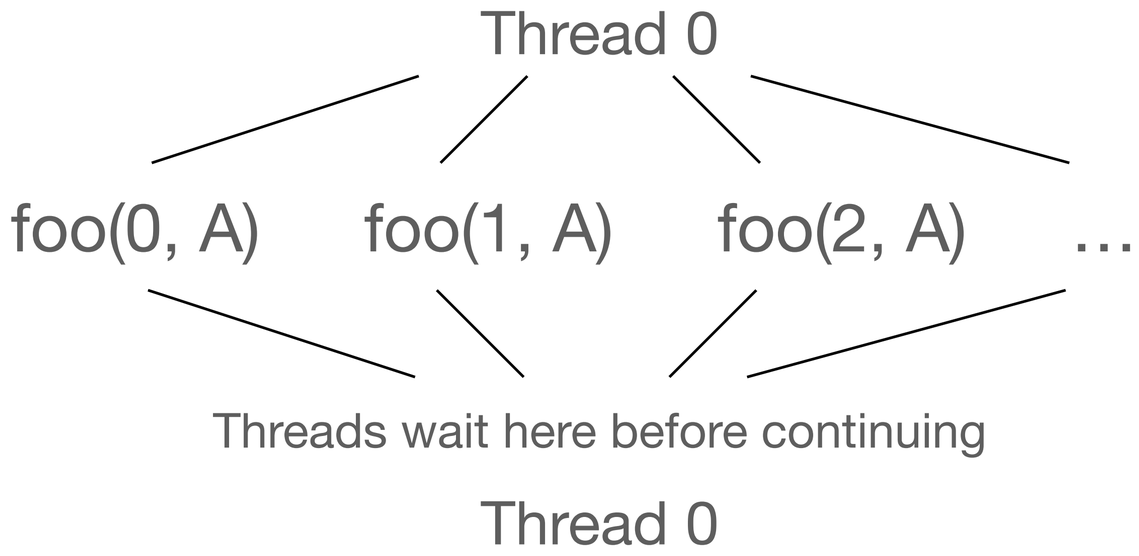
- When setting the number of threads in multiple places, there is a priority.
  - The “num_threads(…)” clause takes precedence
  - Next is the number of threads set by “omp_set_num_threads(…)”
  - Last is the “OMP_NUM_THREADS” environment variable.

```
int main()
{
  omp_set_num_threads(1);

#pragma omp parallel
  {
    int tid = omp_get_thread_num();
    cout << "Hello world from thread " <<
      tid << " / " << omp_get_num_threads()
      << " threads" << endl;
  }

#pragma omp parallel num_threads(2)
  {
    int tid = omp_get_thread_num();
    cout << "Hello world again from thread " <<
      tid << " / " << omp_get_num_threads()
      << " threads" << endl;
  }
  return 0;
```

Basic OpenMP directive: “omp parallel”

- Recall “#pragma omp parallel” creates a parallel region
- “A” is shared among all threads
- OpenMP provides an implicit barrier (*synchronization*) at the end of the parallel region
- Each thread waits at the end of the parallel region until all threads are finished before continuing.

```
    double A[1000];
#pragma omp parallel{
    int tid = omp_get_thread_num();
    foo(tid, A);
}
```

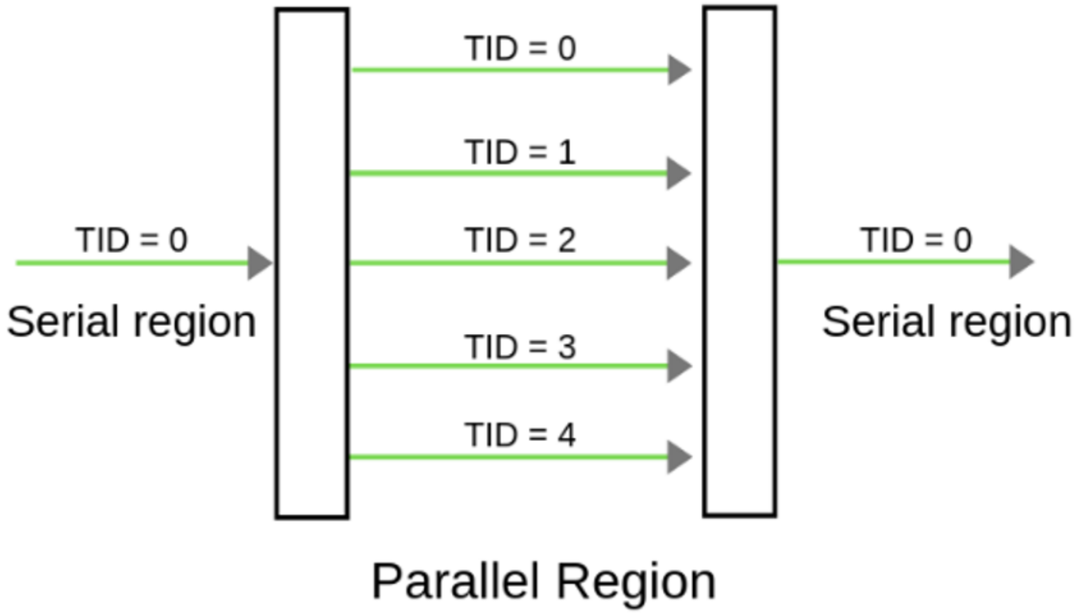


“How many threads” vs “how many cores"?

- Recall cores are hardware units and threads are software units. However, you can have fewer threads than cores, or more threads than cores.
  - Fewer threads than cores: cores sit idle
  - More threads than cores: cores process some threads threads, then switch to processing other threads.
- Many more threads than cores can require switching processes often, resulting in higher overhead.
- Rule of thumb: aim for more threads than cores, with the number of threads up to 2x the number of cores.

OpenMP's model of parallelism

- Specifying a parallel *region* instead of individual threads (like Pthreads)
- The thread with ThreadId = 0 is referred to as a "master" thread, and persists in both the serial and parallel regions
- Instead of assigning work to specific threads, OpenMP assumes the same code is executed on every thread.



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

Fork-join parallelism

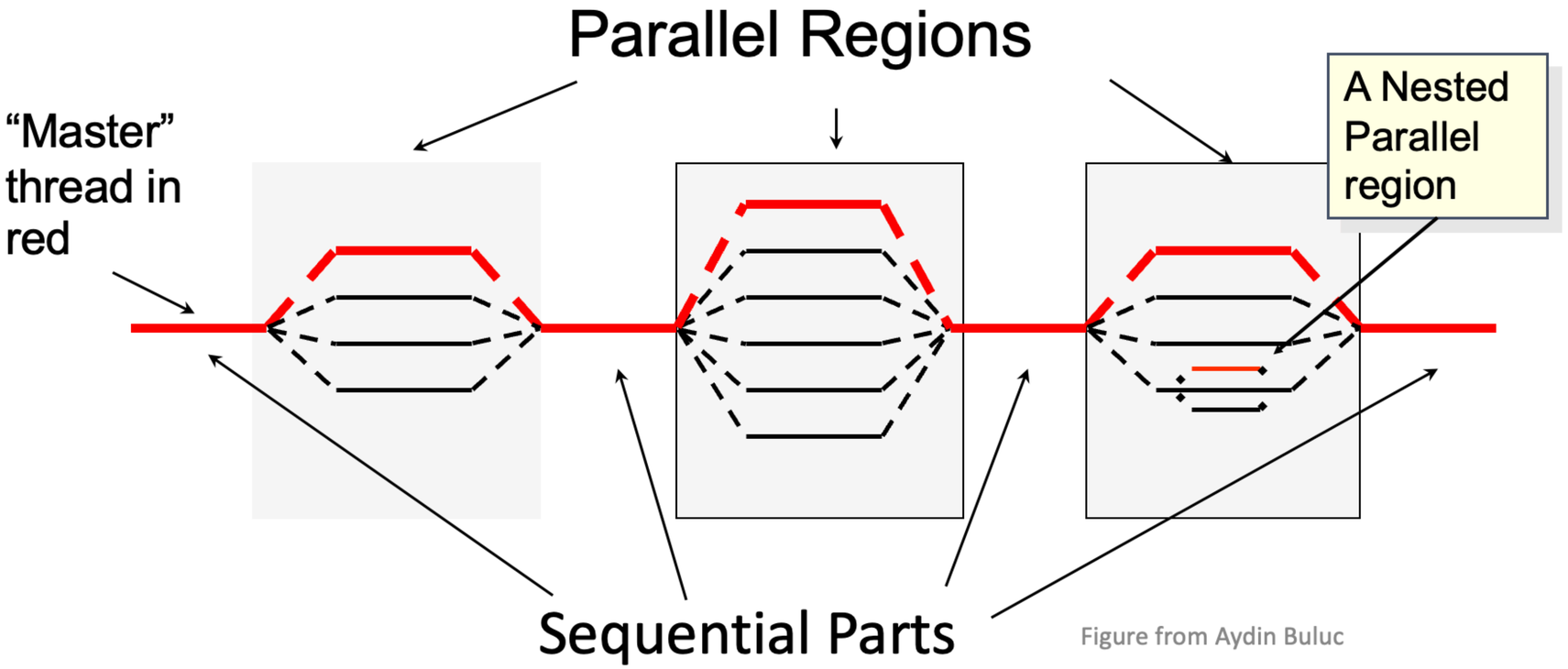
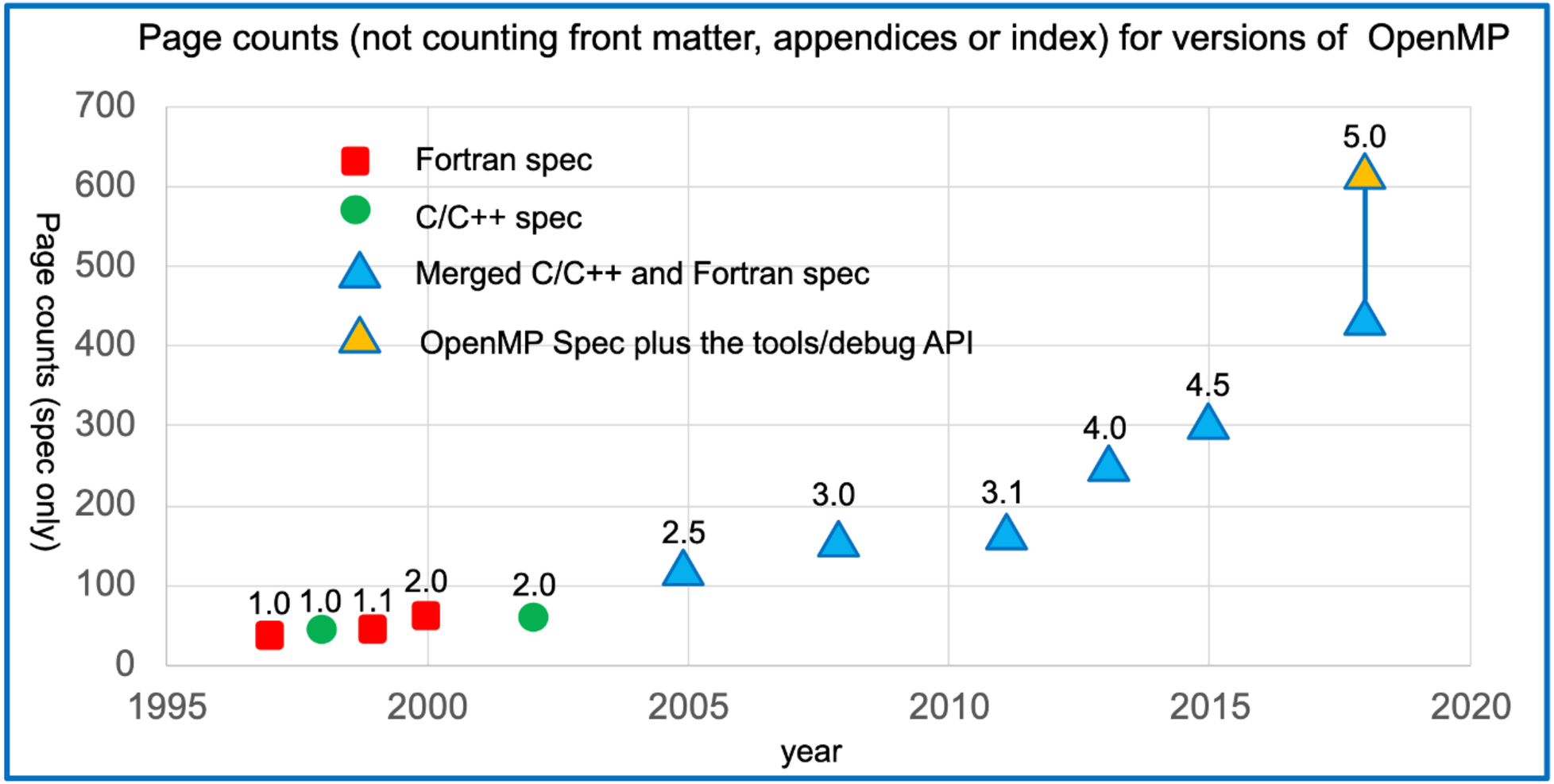


Figure from Aydin Buluc

- OpenMP is flexible with respect to constructing parallel regions.
- Different numbers of threads per parallel region, nested parallel regions, dynamic thread allocation (not shown here) are all possible.

What else can we do? Quite a lot

OpenMP is huge! However, not all the functionality is necessary

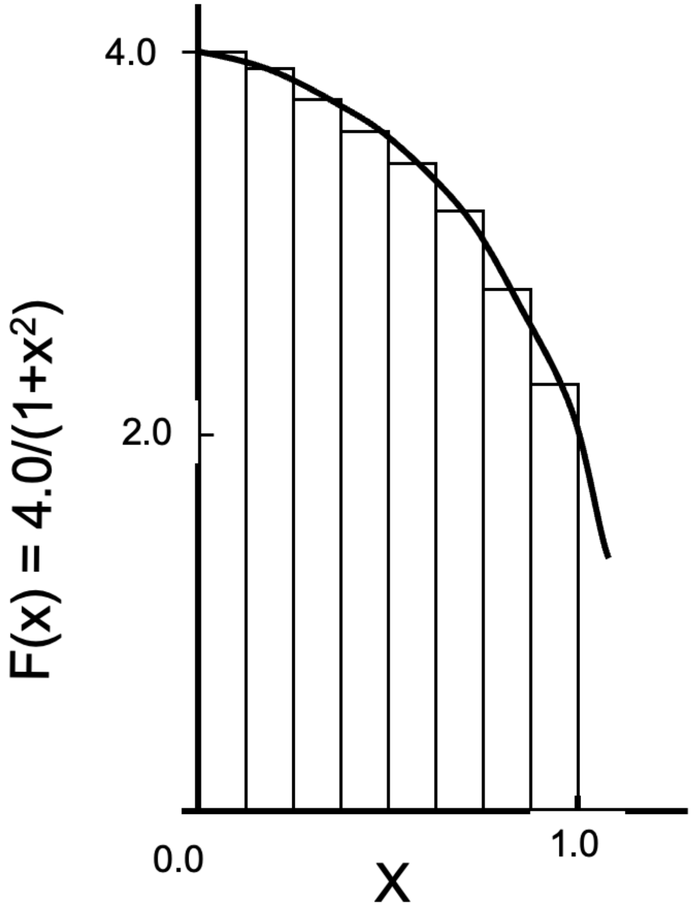


Most OpenMP programs use ~20 directives

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.

OpenMP directives + shared memory parallelism concepts using an example problem

An example problem: computing an integral



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.

A serial implementation

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.

- Note we are timing with “omp_get_wtime()” now.

```
int main(){
  double sum = 0.0;
  int num_steps = 100000000;
  double step = 1.0 / (double) num_steps;

  double elapsed_time = omp_get_wtime();
  for (int i = 0; i < num_steps; i++){
    double x = (i + 0.5) * step;
    sum = sum + 4.0 / (1.0 + x * x);
  }
  double pi = step * sum;
  elapsed_time = omp_get_wtime() - elapsed_time;
  cout << "pi = " << setprecision(7) << pi <<
    " in " << elapsed_time << " secs" << endl;
}
```

Integration with OpenMP: version 1

- Just putting “#pragma omp parallel” around the for loop doesn’t do anything.
  - The “for loop” is simply run on every single thread.
- Simple fix: divide up work so that each thread processes “1 / nthreads” of the total work.
- What do we observe?

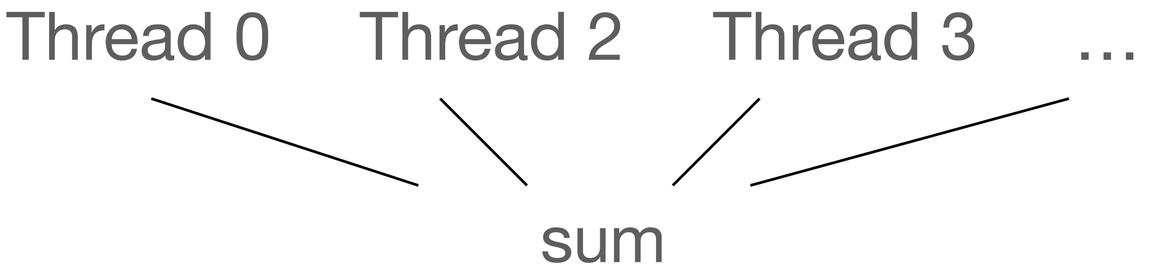
```
#pragma omp parallel
  {
    for (int i = 0; i < num_steps; i++){
      double x = (i + 0.5) * step;
      sum = sum + 4.0 / (1.0 + x * x);
    }
  }
```

```
  int nthreads = 8;
  omp_set_num_threads(nthreads);
  double elapsed_time = omp_get_wtime();
#pragma omp parallel
  {
    for (int i = 0; i < num_steps; i += nthreads){
      double x = (i + 0.5) * step;
      sum = sum + 4.0 / (1.0 + x * x);
    }
  }
```

Data races cause inconsistent results

- Threads communicate by writing to shared variables which other threads can read from.
- If one or more threads attempts to update shared memory at the same time, this can lead to a *data race* (or race condition).
- Memory writes are not synchronized; one thread can overwrite another thread's update

```
    int sum = 0;
#pragma omp parallel
    {
        sum += 1;
    }
    cout << "sum = " << sum << endl;
```



Race conditions disasters

Therac-25 radiation therapy machine

The Therac-25 was involved in at least six accidents between 1985 and 1987, in which some patients were given massive overdoses of radiation.[2]:425 Because of concurrent programming errors (also known as race conditions), it sometimes gave its patients radiation doses that were hundreds of times greater than normal, resulting in death or serious injury.[3] These accidents highlighted the dangers of software control of safety-critical systems.

Northeast blackout of 2003

A software bug known as a race condition existed in General Electric Energy's Unix-based XA/21 energy management system.[13] Once triggered, the bug stalled FirstEnergy's control room alarm system for over an hour. System operators were unaware of the malfunction. The failure deprived them of both audio and visual alerts for important changes in system state.[14][15]

Integration with OpenMP: version 2

- To avoid data races, each thread writes to an independent location in memory.
  - Sum the results in a serial region at the end
  - Serial overhead should be small since number of threads is much smaller than “num_steps”.
- The result for pi should now be correct! What about scaling?

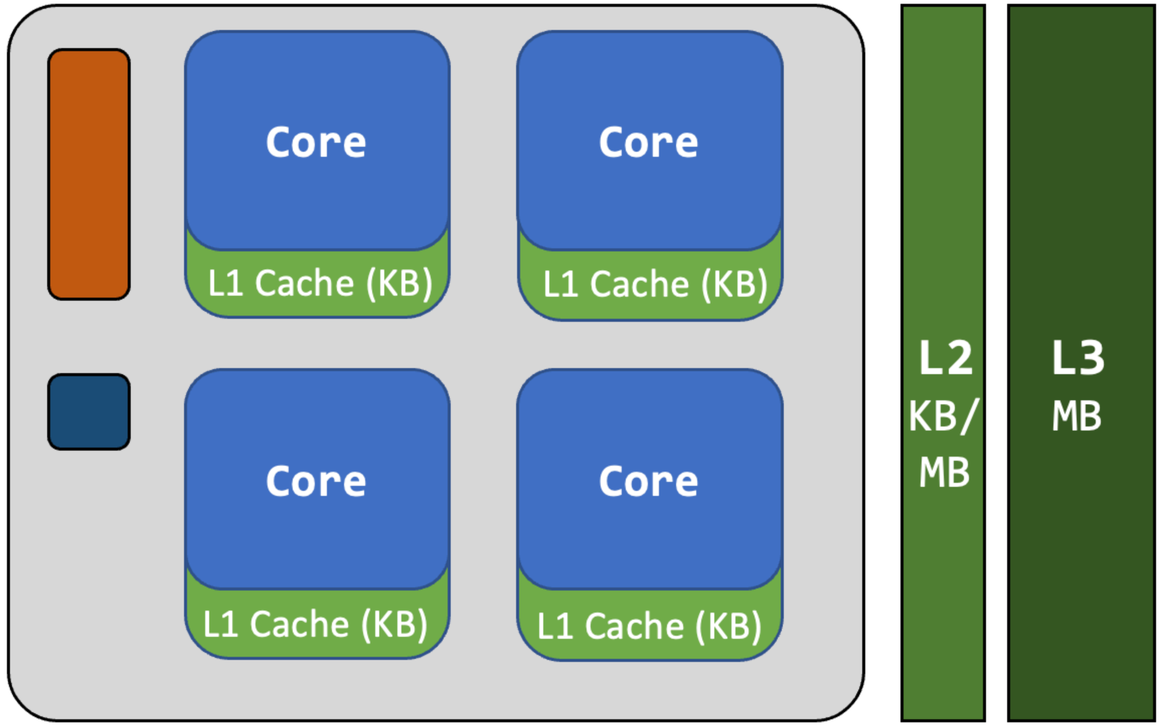
```
double sum[NUM_THREADS];
```

```
#pragma omp parallel
  {
    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[id] = sum[id] + 4.0 / (1.0 + x * x);
    }
  }
```

```
double pi = 0.0;
for (int id = 0; id < NUM_THREADS; ++id){
  pi += step * sum[id];
}
```

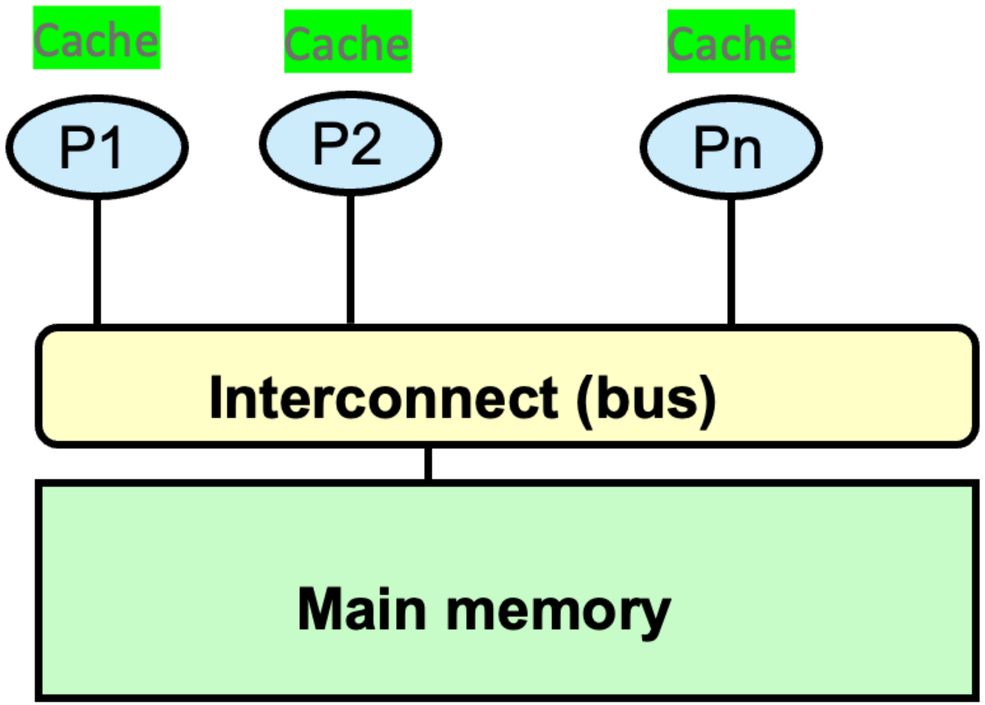
Why is “version 2” not scalable?

- Speedup limited to ~2x due to memory hierarchy (e.g., cache) issues.
- For multi-core architectures, each core has its own L1 cache. However, local caches must synchronize when **shared** variables are updated.
  - If you update elements of an array in one cache, those elements should get updated in the other cache too.



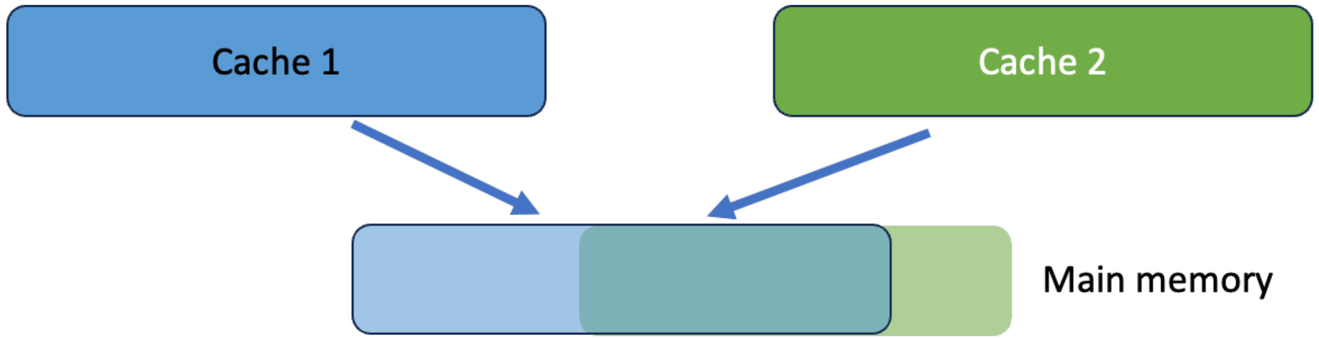
Cache coherency protocols

- Strategies for synchronizing multi-core caches are referred to as “cache coherency protocols”
- Typical approach is to “snoop” on memory operations and synchronize on “relevant transactions”
  - A transaction is relevant if it involves a **cache line currently contained in this cache.**
- Synchronization may go through memory bus, can be slow



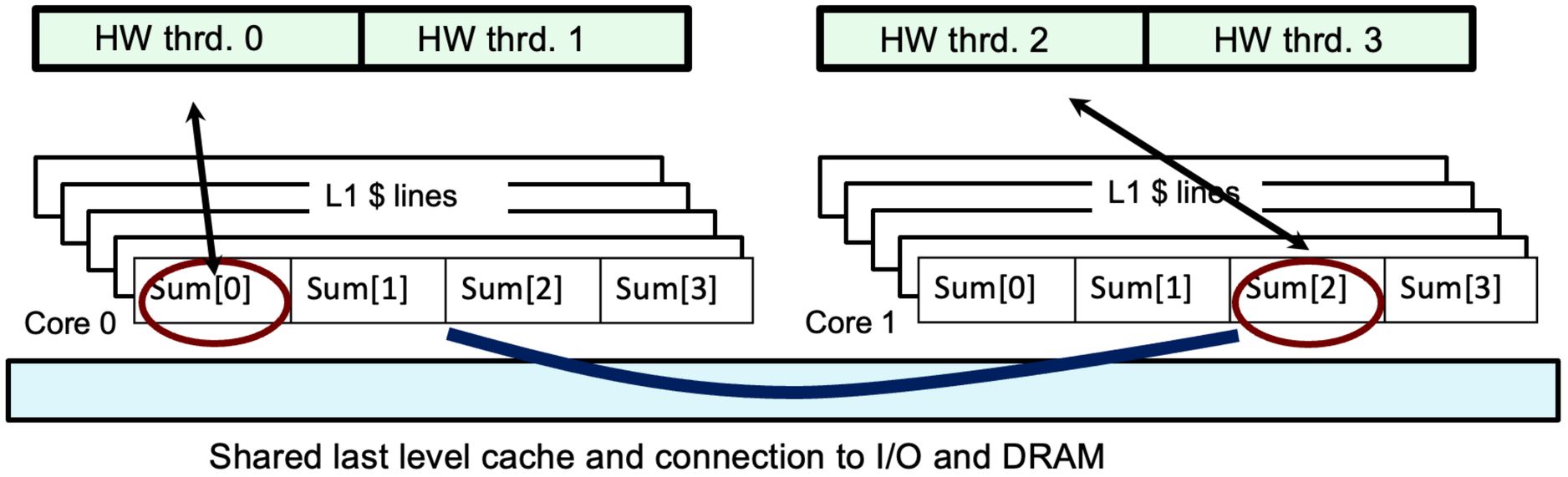
What triggers cache coherency?

- Recall: a memory transaction from any thread will triggers synchronization if it involves a **cache line currently contained in this cache.**
- Suppose you have two threads which access memory locations close to each other (specifically, they are in the same cache line).
  - Thread 1 and Thread 2 both copy the cache line into their fast memory
  - What happens if Thread 1 updates Cache 1 memory? Cache 2 must synchronize as well.



The cache controller will take action (invalidate, update, or supply value) to ensure coherence after a relevant memory transaction.

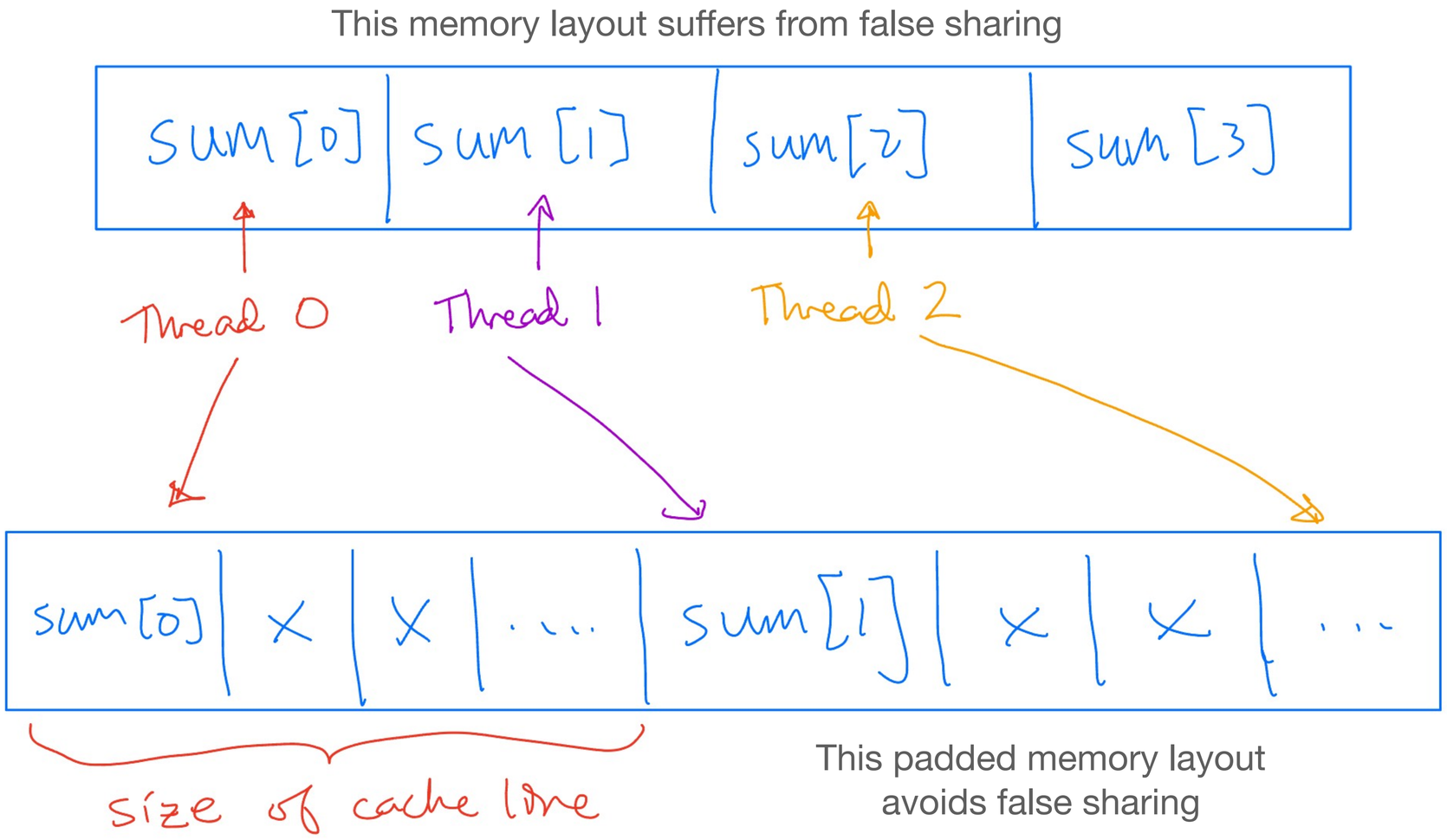
False sharing



- 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
- In other words, threads **shouldn’t** access contiguous array entries!
- Mostly seen in multi-core CPUs, becoming less of an issue w/modern processors.

Avoiding false sharing via padding

- False sharing only happens if threads access the same cache line (e.g., threads access entries close together in memory).
- Avoid this by **padding** an array with unused entries so consecutive elements don't lie in the same cache line.



Cache *lines* are typically much smaller than L1 caches
64 bytes (8 doubles) for x86 architectures, 128 bytes (16 doubles) for M-series Mac?

Integration with OpenMP: version 3

```
double sum[NUM_THREADS][PAD];
```

- Simple way to pad: use that arrays are row major in C/C++
  - Add an extra dimension of size PAD; each row now has stride PAD in memory
- PAD should be set based on your cache line size.
- How do we scale now?

```
#pragma omp parallel
  {
    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[id][0] = sum[id][0] + 4.0 / (1.0 + x * x);
    }
  }
  double pi = 0.0;
  for (int id = 0; id < NUM_THREADS; ++id){
    pi += step * sum[id][0];
  }
```

OpenMP directives: making fast code less ugly

```
#pragma omp <directive> <clauses> <other stuff>
```

- Padding is a fairly invasive and ugly operation. OpenMP provides several directives and clauses to avoid explicit padding.
- Examples of <directives>: “#pragma omp parallel”, “#pragma omp for”, …
- Examples of <clauses>:
  - Setting “num_threads(4)”, declaring a variable “shared” or “private”
  - Synchronization (e.g., “reduction”, “nowait”), scheduling (specifies thread management strategies)

Variable scoping in OpenMP

- Recall the idea of variable *scope:* a variable's scope defines where the variable can be accessed (e.g., if a variable is defined inside a function or within a loop).
- OpenMP blocks also introduce variable scope: some variables are thread-local, some are shared (global). Default behavior:
  - Variables used inside the parallel block (but declared before it) are **shared**, i.e., all threads have access to those variables.
  - Variables declared inside the block are **private**, i.e. each thread has their own local copy of that variable.
- We can change this behavior using OpenMP directives with *clauses*

Integration with OpenMP: version 4

- Instead of padding the “sum” array, make “sum” a private variable local to each thread.
  - Avoids need for padding; private variables should not be in same cache line.
- Still need to add up local “sum” accumulators. To avoid race conditions, use the **critical** directive.
  - Only allows one thread to access this section at a time.

```
#pragma omp parallel
{
    double sum = 0.0; // private variable
    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);
    }
    // only one thread does this per time
    #pragma omp critical
    {
        pi += step * sum;
    }
}
```

“private” vs “firstprivate”, “lastprivate”

- Can also use the “firstprivate” clause to initialize the private variable “sum”
  - Uses the value of “sum” outside of the parallel region
  - Makes a copy of “sum” in local parallel scope.
- “lastprivate” exports the last value a variable takes in a sequential iteration.

```
double sum = 0.0; // private variable
#pragma omp parallel firstprivate(sum)
{
    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);
    }
    // only one thread does this per time
    #pragma omp critical
    {
        pi += step * sum;
    }
}
```

Variable sharing behavior: “default” clause

- Can set default variable sharing behavior with “**default**” clause
  - default(shared): all external variables defined outside the OpenMP construct are turned into shared variables (except for OpenMP tasks)
  - default(private): all external variables defined outside the OpenMP construct are copied into thread-local private variables
  - default(none): no default for variables; all variables transferred into an OpenMP construct must be explicitly listed. Useful for debugging.

```
#pragma omp parallel default(none)
```

Synchronization in OpenMP

- In addition to communicating with each other through shared memory, OpenMP threads can be *synchronized* using various directives.
- Example: “#pragma omp critical” forces synchronization between threads
  - Any code in this region will only have one thread at a time running. This serializes the code, but avoids race conditions.
  - Similar to a “lock” in pthreads
- Other types of synchronization in OpenMP:
  - Implicit and explicit *barriers,* ordered sections, atomics.
  - We’ll revisit this after going over OpenMP basics.

Avoiding manual parallel indexing

- Private variables allowed us to avoid manual padding.
- Would be nice to avoid manually setting up the for loop in the parallel region.
  - We are essentially manually scheduling and load balancing work among OpenMP threads
- Can fix this using OpenMP's **"parallel for"** directive and associated clauses

```
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);
}
```

The way we break up the for loop is non-unique; can have a thread loop through every "nthreads" entry or have each thread loop through a "num_steps / threads" chunk

Integration with OpenMP: version 5

- Several things are the same as v4:
  - “sum” is still a private variable local to each thread
  - we still use the **critical** directive to avoid race conditions.
- Main change: “#pragma omp for” instead of manual thread-dependent loop indexing.
  - Automatically splits up iterations of the loop among the threads

```
  double pi = 0.0;
#pragma omp parallel
{
    double sum = 0.0; // private variable
    #pragma omp for
        for (int i = 0; i < num_steps; ++i){
            double x = (i + 0.5) * step;
            sum += 4.0 / (1.0 + x * x);
        }

    // only one thread does this per time
    #pragma omp critical
    {
        pi += step * sum;
    }
}
```

Two ways to do parallel for loops in OpenMP

- Can define as a nested directive within an “omp parallel” block
  - Can be useful if you need to initialize private variables prior to the parallel for loop.
- Can also define as a standalone OpenMP directive (more on this later)
- Can split work unevenly using clauses (useful in applications with varying work per loop iteration, such as sparse matrix multiplication)

```
#pragma omp parallel
#pragma omp for
for (int i = 0; i < 8; ++i){
  // ...
}
```

```
#pragma omp parallel for
for (int i = 0; i < 8; ++i){
  // ...
}
```

Guidelines for parallel for loops in OpenMP

- OpenMP allows for nested parallel for loops. Can either spawn more threads or parallelize across nested loops using “collapse” directive.
- For efficiency, should ensure each loop iteration has similar work (load balancing).
- Cannot directly parallelize a loop with a *loop-carried dependence*!

```
double x[n];
x[0] = 1;
x[1] = 1;
for (int i = 2; i < n; ++i){
  x[i] = x[i-1] + x[i-2]
}
```

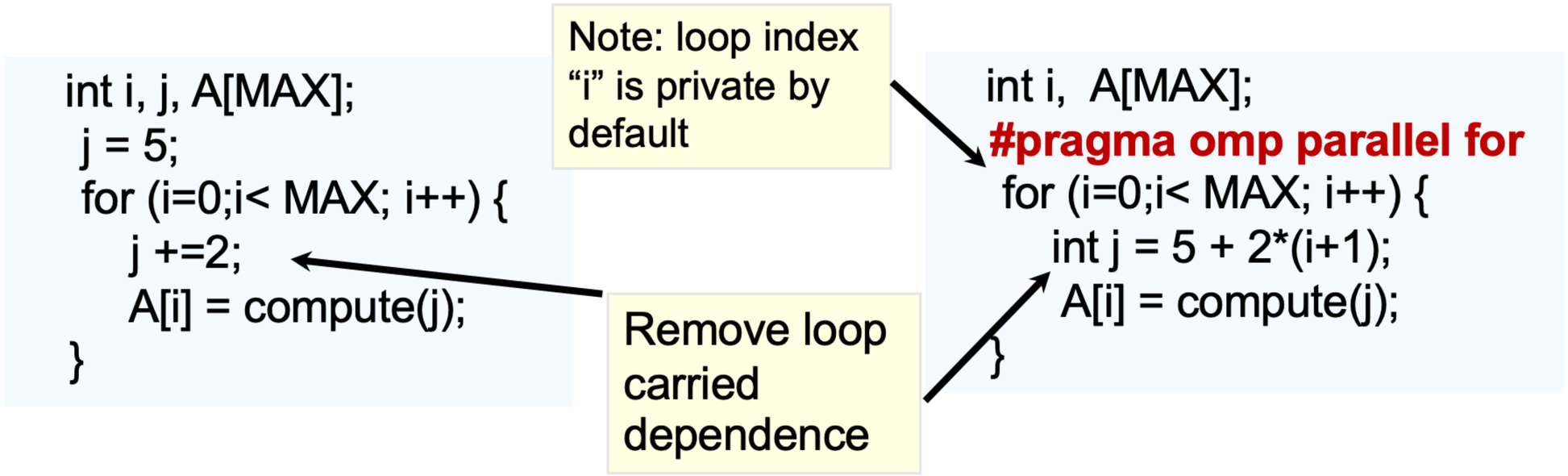
An example of a “loop-carried dependence”

OpenMP loop collapsing

```
#pragma omp parallel for collapse(2)
  for (int i = 0; i < n; ++i)
  {
    for (int j = 0; j < m; ++j)
```

- If you have nested loops, can use the “collapse” clause to combine nested loops into one large single loop
  - Larger loop = more work over which to parallelize (more efficient)
  - Cannot be used on loops with bounds which depend on outer loop indices

Guidelines for parallel for loops, cont



- Cannot directly parallelize a loop with a *loop-carried dependence.* Try to rewrite loop iterations to be independent as possible.
- Not all loop iterations can be re-written to avoid loop-carried dependences! Example?

Guidelines for parallel for loops, cont

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

- Cannot directly parallelize a loop with a *loop-carried dependence.* Try to rewrite loop iterations to be independent as possible.
- Not all loop iterations can be re-written to avoid loop-carried dependences! Example?

Reductions: specific loop-carried dependencies

- Computing norms, averages, min/max, dot products, “all true/false” checks, etc.
  - Taking an array (or multiple entries) and collapsing them to a single value.
- Very common in HPC and parallel programming (e.g., “map-reduce”)
- Natively supported in most parallel programming environments: OpenMP, but also MPI and CUDA.

```
double xmax = 0.0;
for (int i = 0; i < n; ++i)
{
  if (xmax < x[i])
  {
    xmax = x[i];
  }
}
```

Examples of “+” and “max” reductions

Reductions: specific loop-carried dependencies

- Computing norms, averages, min/max, dot products, “all true/false” checks, etc.
  - Taking an array (or multiple entries) and collapsing them to a single value.
- Very common in HPC and parallel programming (e.g., “map-reduce”)
- Natively supported in most parallel programming environments: OpenMP, but also MPI and CUDA.

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

```
double xmax = 0.0;
for (int i = 0; i < n; ++i)
{
  if (xmax < x[i])
  {
    xmax = x[i];
  }
}
```

Examples of “+” and “max” reductions

OpenMP reductions

- OpenMP supports the reduction clause within a parallel for loop.
- Syntax: “reduction (**op** : **variable**)”,
  - **op** is a supported operation
  - **variable** is the shared variable which stores the result of the reduction

```
  double sum = 0.0;
#pragma omp parallel for reduction (+:sum)
  for (int i = 0; i < n; ++i)
  {
    sum += x[i];
  }
  cout << "sum = " << sum << endl;
```

OpenMP reductions

```
  double sum = 0.0;
  double sum2 = 0.0;
  int nmax = n / 2;
#pragma omp parallel for reduction (+:sum,sum2)
```

Reducing two variables

```
  int N = 10;
  int A[] = {84, 30, 95, 94, 36, 73, 52, 23, 2, 13};
  int S[N] = {0};
#pragma omp parallel for reduction(+ : S[:N])
  for (int n = 0; n < N; ++n)
  {
    for (int m = 0; m <= n; ++m)
    {
      S[n] += A[m];
    }
  }
```

An array reduction

```
  double sum = 0.0;
#pragma omp parallel for reduction (+:sum)
  for (int i = 0; i < n; ++i)
  {
    sum += x[i];
  }
  cout << "sum = " << sum << endl;
```

A simple reduction.

```
  double sum = 0.0;
  int nmax = n / 2;
#pragma omp parallel for
#pragma omp for reduction (+:sum)
  for (int i = 0; i < n; ++i)
  {
    if (i < nmax){
      sum += x[i];
    }
  }
  cout << "sum = " << sum << endl;
```

A more customized reduction

How OpenMP reductions work

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

- A local copy of each shared reduction variable is created and initialized depending on the operation “op” (e.g. the variable is initialized to 0 for “+”).
- The local copy is updated within the parallel loop.
- Local copies are reduced to a single value and stored within the shared reduction variable.

Exercise

- Implement integration using an OpenMP reduction.
- Check the scalability of your program
  - If on your own computer, use 1, 2, …, max number of cores
  - If on nots, use up to 32 threads (request the same number of cores)
- Compare the runtimes to one other OpenMP integration code:
  - integration_omp_padding.cpp
  - integration_omp_private.cpp
  - integration_omp_parallel_for.cpp

OpenMP so far

- What we've covered:
  - "#pragma omp parallel" and "#pragma omp for"
  - Basic scoping: shared/private variables
  - Basic synchronization: "#pragma omp critical", implicit barriers
  - Reduction operations
- These are sufficient to handle many "easily parallelizable" programs.
- What remains: synchronization, load balancing, task-based parallelism

Synchronization

- Useful for:
  - Respecting ordering requirements for an algorithm
  - Ensuring thread correctness (e.g., avoiding race conditions)
  - Printing information or implementing interactivity
- You've already seen a synchronization directive: "critical"
  - Sequentializes a section of code; only one thread can run at a time.
- Some directives: barrier, atomic, single, master, ordered

OpenMP synchronization: “barrier”

- Waits until all threads arrive at the location of the barrier until proceeding
- Example: parallel initialization of an array “x”, then thread-local computations with the full array.
  - Useful for “scatter” type operations; will cover more in MPI section.

```
#pragma omp parallel
  {
    int id = get_omp_thread_num();
    initialize_x(&x[i], id);

// barrier ensures all entries
// of x are computed
#pragma omp barrier

#pragma omp for
    for (int i = 0; i < n; ++i)
    {
      calculate_y(y, x, id);
    }
  }
```

OpenMP synchronization: “nowait”

- Barriers (implicit or explicit) are expensive; the “nowait” clause removes the **implicit barrier** at the **end of a parallel region**

```
#pragma omp parallel
  int id = omp_get_thread_num();
#pragma omp for nowait
  for (int i = 0; i < n; ++i)
  {
    initialize_x(&x[i], id);
  }
  // no barrier at end of
  // the "omp for" loop
```

- For example: #pragma omp for {…}
  - TID 0 —> do work —> wait longer —> proceed
    TID 1 —> do more work —> wait —> proceed
    TID 2 —> significantly more work —> proceed

OpenMP synchronization: “nowait”

- Barriers (implicit or explicit) are expensive; the “nowait” clause removes the **implicit barrier** at the **end of a parallel region**

```
#pragma omp parallel
  int id = omp_get_thread_num();
#pragma omp for nowait
  for (int i = 0; i < n; ++i)
  {
    initialize_x(&x[i], id);
  }
  // no barrier at end of
  // the "omp for" loop
```

- For example: #pragma omp for nowait {…}
  - TID 0 —> do work —> proceed without waiting
    TID 1 —> do more work —> proceed without waiting
    TID 2 —> significantly more work —> proceed

OpenMP synchronization: “atomic”

- Similar to OpenMP “critical” region, but less general. However, atomics have much lower overhead.
- Older versions of OpenMP could only do simple atomic updates
- Newer versions of OpenMP provide clauses to enable a wider range of atomic operations.
  - Read, write, update, capture

```
  int counter = 0;
#pragma omp parallel
  {
#pragma omp atomic
    counter += 1;
  }
```

```
  double max_val = 0.0;
#pragma omp parallel for
  for (int i = 0; i < n; ++i)
  {
#pragma omp atomic compare
    if (max_val < x[i])
    {
      max_val = x[i];
    };
  }
```

OpenMP synchronization: “single”/“master”

- “#pragma omp single” runs a region of code with only one single thread
- “#pragma omp master” does the same thing as “single”, but the code runs specifically on the master thread.
- Can use “nowait” clause with “single” directive but not with “master”
- May be necessary to restrict to master thread for some operations, e.g., MPI.

```
#pragma omp parallel
  {
#pragma omp single // or "omp master"
    {
      // do stuff
    }
  }
```

OpenMP synchronization: “ordered”

```
#pragma omp parallel for ordered
  for (int i = 0; i < omp_get_num_threads(); ++i)
  {
#pragma omp ordered
    cout << "on thread " << omp_get_thread_num() << endl;
  }
```

- Runs a block of code in an ordered fashion; useful for printing information
  - “ordered” is both a clause and its own directive.
  - The clause defines an ordered OpenMP loop; however, the ordered *directive* specifies which lines within this loop need to be ordered.
  - Useful if you want to print info but also retain some threaded computations

OpenMP synchronization: “flush”

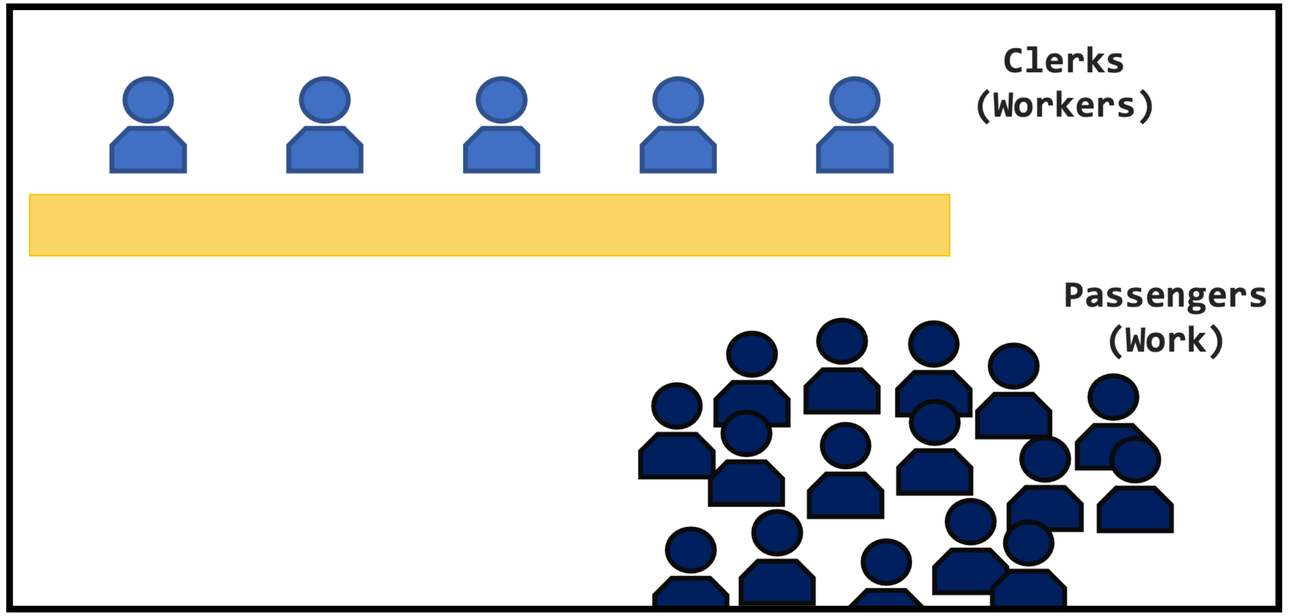
```
  double x;
  #pragma omp parallel
{
  calculate_something_with(x);
  #pragma omp flush(x)
}
```

- OpenMP flush synchronizes *memory*; “flush(x)” synchronizes the value of “x” so that all other threads get the most recent value.
  - Analogous to a “fence” in shared memory parallelism, is done automatically by OpenMP at implicit and explicit barriers.
- Why “flush”? Can synchronize only on specific variables, allows compiler to better optimize (i.e., rearrange) code.

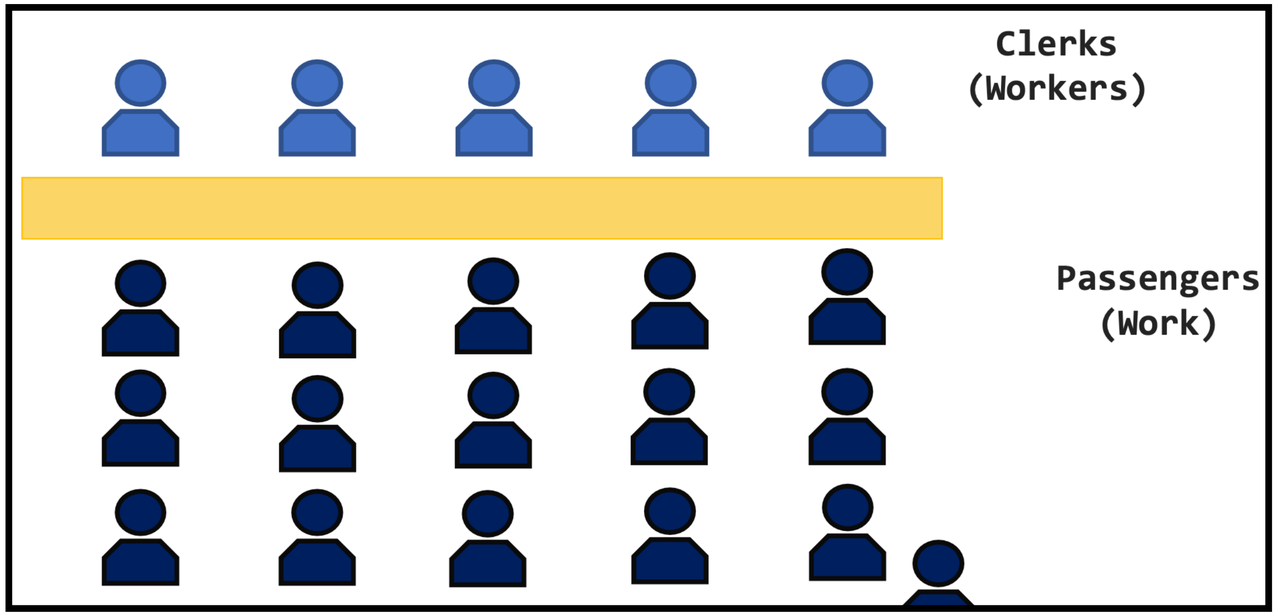
OpenMP parallel for loops: scheduling

- Up to now, we've assumed that each iteration of our parallel for loop is roughly the same cost so the iterations can be evenly divided among threads.
- When they are not, the iterations should be divided up unequally, so that the amount of work each thread is doing is roughly equal.
  - This is commonly referred to as "load balancing" in parallel computing
- OpenMP provides a "schedule" clause to determine how to divide up for loop iterations and schedule them among threads

Illustration of load balancing

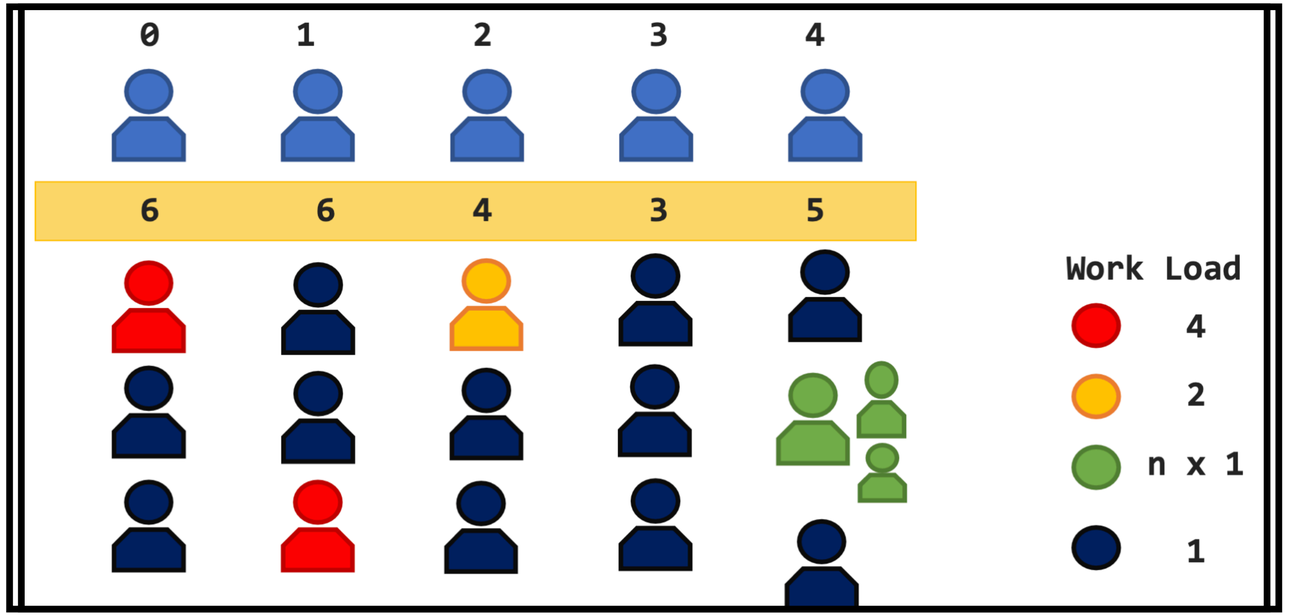


Unbalanced

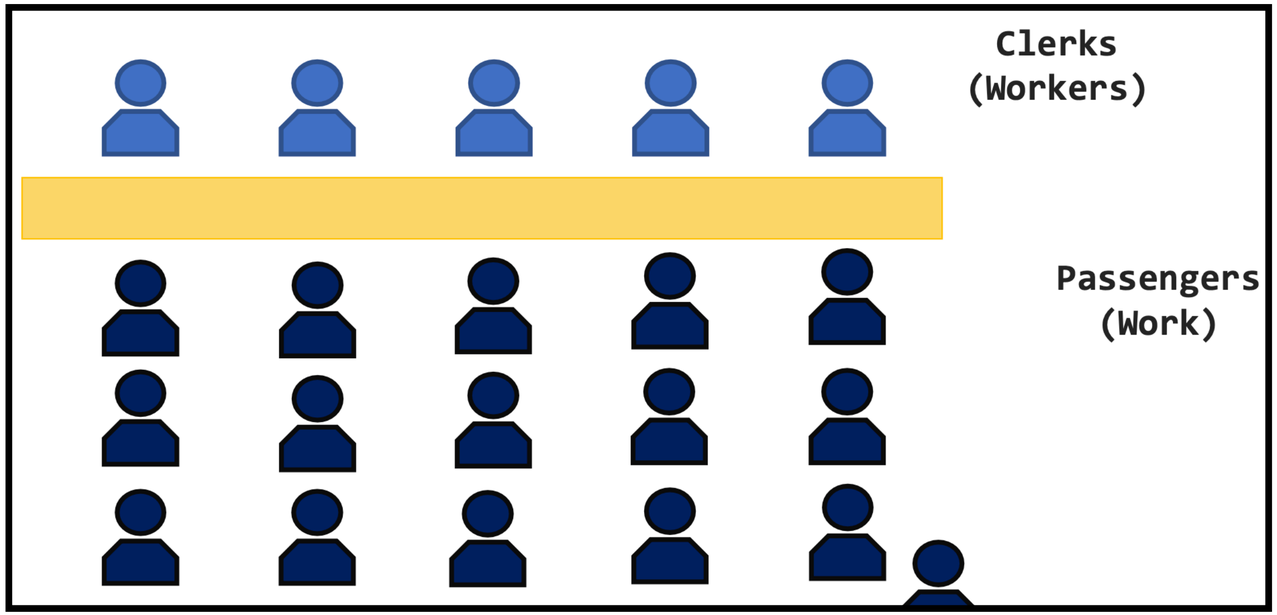


Balanced

Illustration of load balancing



Unbalanced



Balanced

OpenMP scheduling

- Syntax: “#pragma omp [parallel] for **schedule(schedule type, chunk_size)**”
  - Schedule types: static, dynamic, guided, auto, runtime
  - Iterations are distributed in groups of “chunk_size”. This parameter defaults to the number of loop iterations / number of threads.
- “Static” is (usually) the default if no schedule clause is specified
  - Divides up iterations equally among available threads

OpenMP non-static scheduling

- Other options: dynamic, guided, auto
- “Dynamic”: each thread is given a chunk of the specified size. Once the work is completed, each thread grabs a new chunk until no chunks are left.
- “Guided”: like dynamic, each thread works on a chunk then grabs a new chunk once it is finished. However, the chunk size starts large then decreases.
  - Chunk size set based on (remaining iterations) / (number of threads). This is intended to reduce scheduler overhead.

```
#pragma omp parallel for schedule(guided)
  for (int i = 0; i < n; ++i)
  {
    for (int j = 0; j < i; ++j)
      x[i] += 1;
  }
```

For n = 50000, elapsed time was
- static: 0.753789 secs
- dynamic: 0.668723 secs
-guided: 0.45944 secs

OpenMP scheduling: auto

- “Auto”: lets the compiler decide the optimal strategy (may not do anything)
- Implementation in GCC actually reverts to static scheduling

```
case GFS_AUTO:
  /* For now map to schedule(static), later on we could play with feedback
     driven choice.  */
```

Task-based parallelism

- Up to now, we've parallelized over regions of code and loop iterations
- OpenMP enables parallelization over an unstructured set of *tasks* instead.
  - Tasks should be independent, mappable to independent threads
  - Runtime system decides when a task is executed (i.e., can optimize task queue by deferring execution)

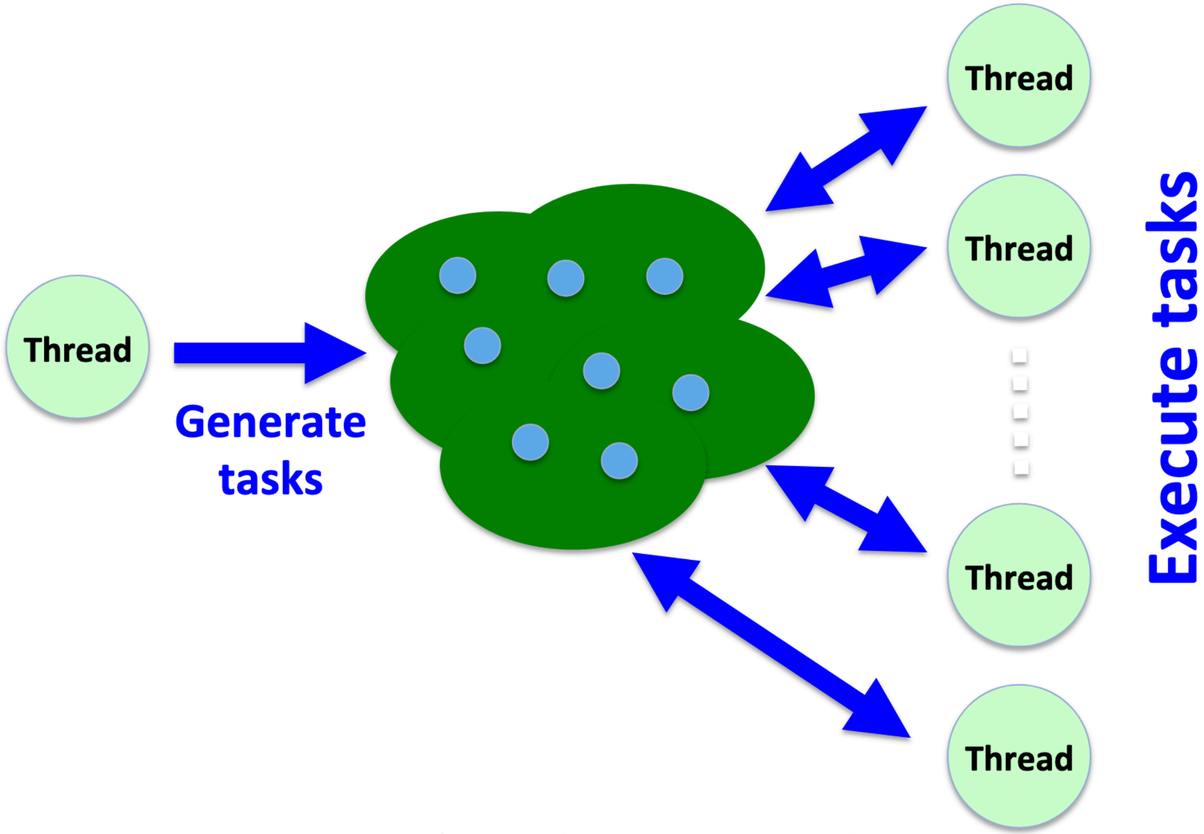


Figure from Ruud van der Pas

Task-based parallelism syntax

- Tasks are useful for divide-and-conquer parallelization.
- Syntax: “#pragma omp task” specifies a task within a parallel region
  - Within each task region, the code can be fairly complex.
  - Note: “#pragma omp single” is often used to launch tasks; without this, each thread would generate a task.
- Tasks are guaranteed to be complete at thread barriers or *task barriers*

```
#pragma omp parallel num_threads(3)
  {
#pragma omp single
    {
#pragma omp task
      cout << "executing task 1\n";

#pragma omp task
      cout << "executing task 2\n";
    }

#pragma taskwait

#pragma omp single
#pragma omp task
    cout << "executing task 3\n";
}
```

Task-based parallelism syntax

- Generating tasks is often done using a single thread to generate tasks
  - “#pragma omp single” + a loop + “pragma omp task”
- #pragma omp taskloop” basically replaces the “omp single” and “omp task” directives.
- Tasks are also good options if you want to parallelize a while loop without explicit bounds.

```
#pragma omp parallel
  {
    #pragma omp single
    {
      for (int i = 0; i < ntasks; ++i){
        #pragma omp task
        cout << "task " << i << "\n";
      }
    }
  }
```

```
#pragma omp parallel
  {
    #pragma omp taskloop
    for (int i = 0; i < ntasks; ++i){
      cout << "task " << i << "\n";
    }
  }
```

OpenMP task example: Fibonacci

- Recall recursive implementation of Fibonacci; hard to parallelize using existing OpenMP techniques.
- Tasks allow us to identify the two recursive Fibonacci calls and parallelize among them.
- Demo.

OpenMP tasks require explicit variable scoping

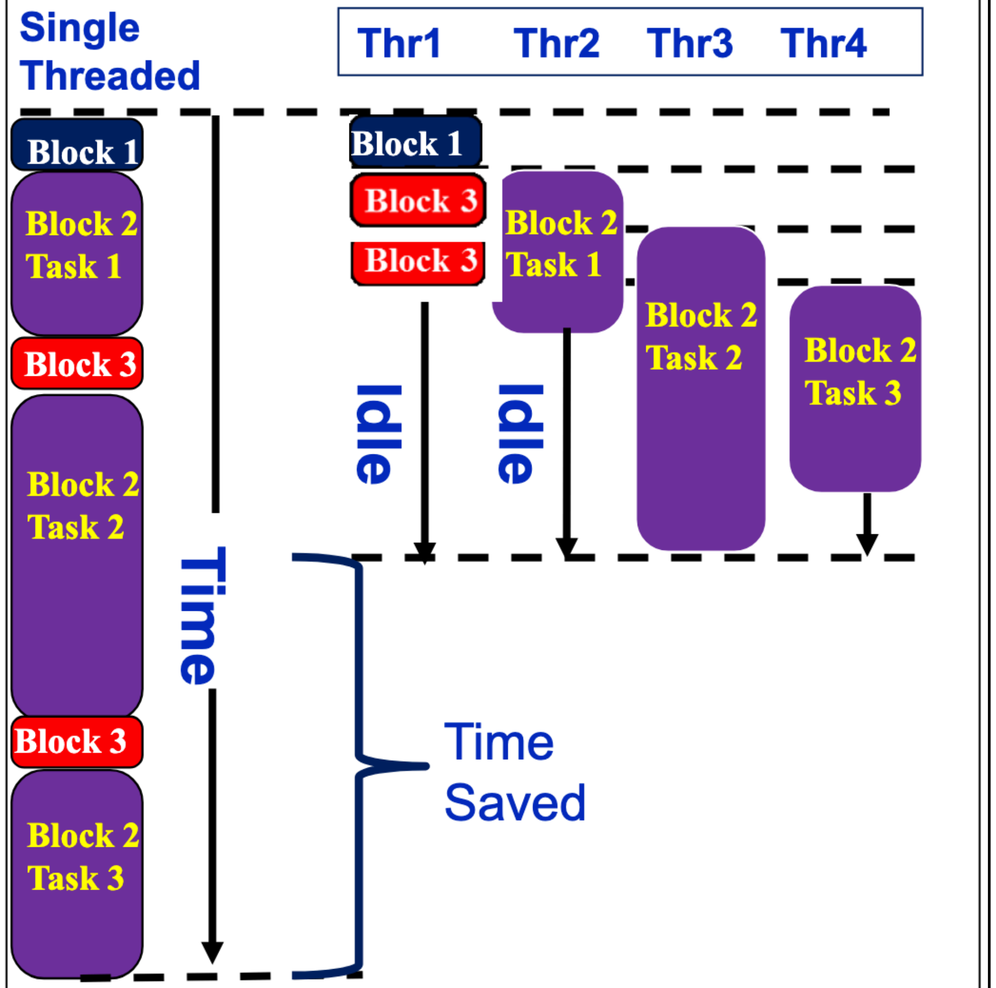
- Implicit variable scoping rules do not apply to tasks.
- Need to explicitly specify shared variables that tasks will operate on.
  - If not specified, *all* variables inside a task are assumed to be private.
  - Tasks are more flexible so they require stricter variable scoping.

```
  int fib1, fib2;

#pragma omp task shared(fib1)
  {
    fib1 = fib(n - 1);
  }

#pragma omp task shared(fib2)
  {
    fib2 = fib(n - 2);
  }

#pragma omp taskwait
  return fib1 + fib2;
```

How tasks execute

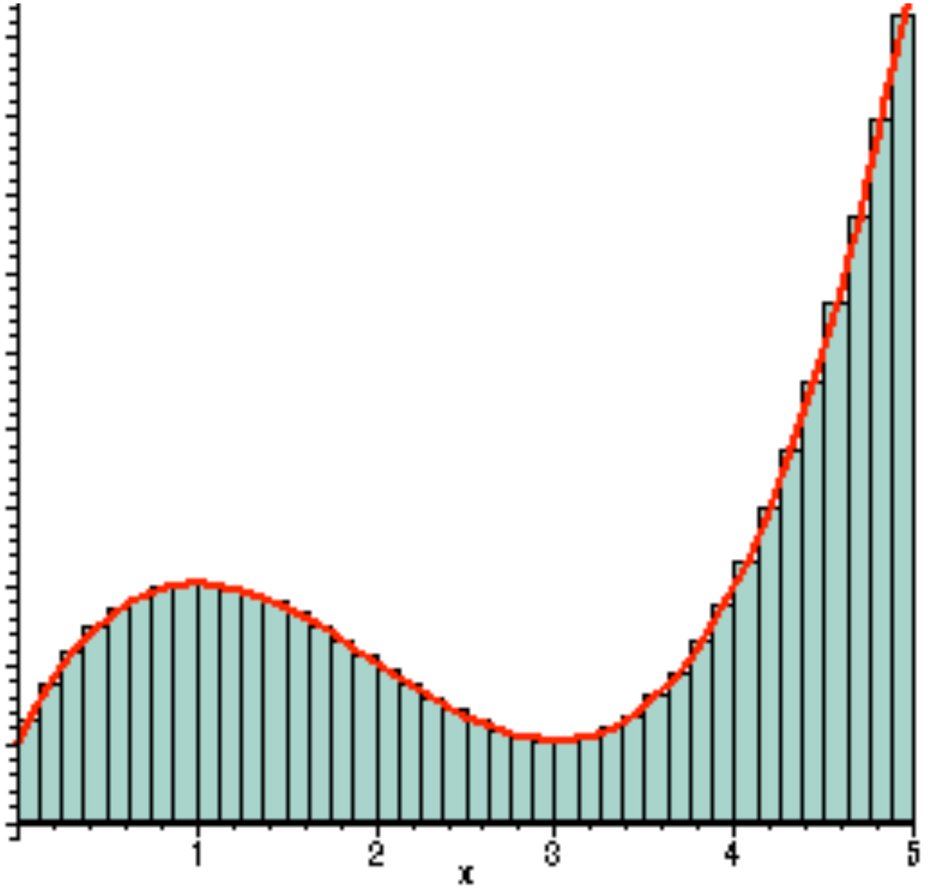
- Flexible execution
- However, relies on scheduler to determine task order and load balancing.
- Comes with some overhead.

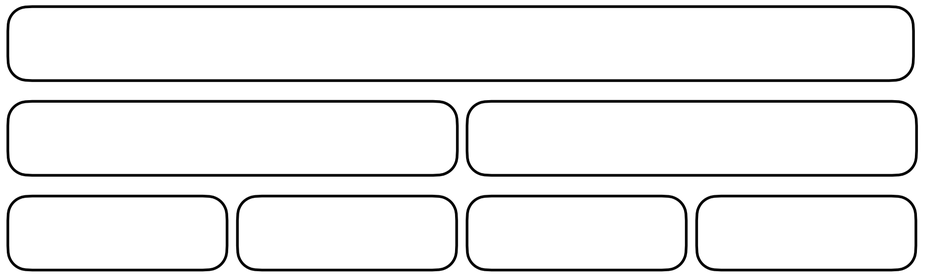
```
#pragma omp parallel
{
  #pragma omp single
  {  //block 1
    node * p = head;
    while (p) { // block 2
    #pragma omp task
      process(p);
    p = p->next;   //block 3
    }
  }
}
```



Integration but with tasks

- Lets compare tasks to other threading for our favorite integration example.
- How to make it recursive? “Divide and conquer”
  - “integrate_sum(start, end, step_size)” recursively calls itself until “end - start” is small enough
  - Similar to recursive matrix multiply
- Demo





Integration divide and conquer

- How is the performance compared with an OpenMP reduction?
  - Not bad! Depends on the recursion termination condition (minimum chunk size).
  - More aggressive compiler optimization yields similar runtime with reduction.

Launch recursive tasks

```
#pragma omp parallel
  {
#pragma omp single
    sum = integral_sum(0, num_steps, step);
  }
```

Divide and conquer recursion

```
    double sum1, sum2;
    int iblk = end - start;
#pragma omp task shared(sum1)
    sum1 = integral_sum(start, end - iblk / 2, step);
#pragma omp task shared(sum2)
    sum2 = integral_sum(end - iblk / 2, end, step);
#pragma omp taskwait
    sum = sum1 + sum2;
```

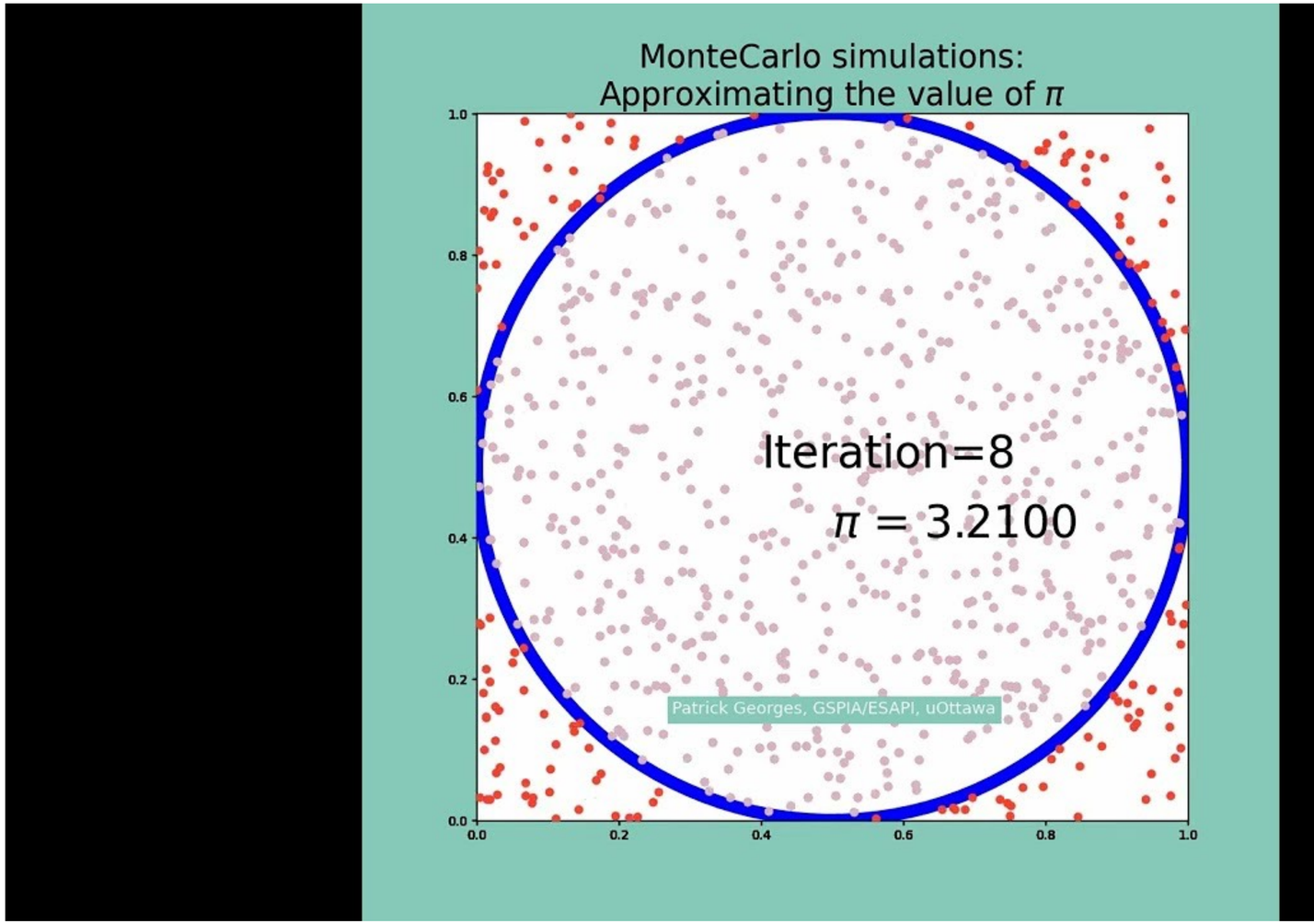
OpenMP sections

- Similar to tasks, but assume a *static* number of parallel sections (e.g., sections cannot be launched from a loop like tasks).
- Sections are simpler, but tasks are better for dynamic parallelism (e.g., adding tasks without knowing how many there will be total).

```
#pragma omp sections
{
  int id = omp_get_thread_num();

  #pragma omp section
  {
    first_calculation(x, id);
  }

  #pragma omp section
  {
    second_calculation(x, id);
  }
}
```

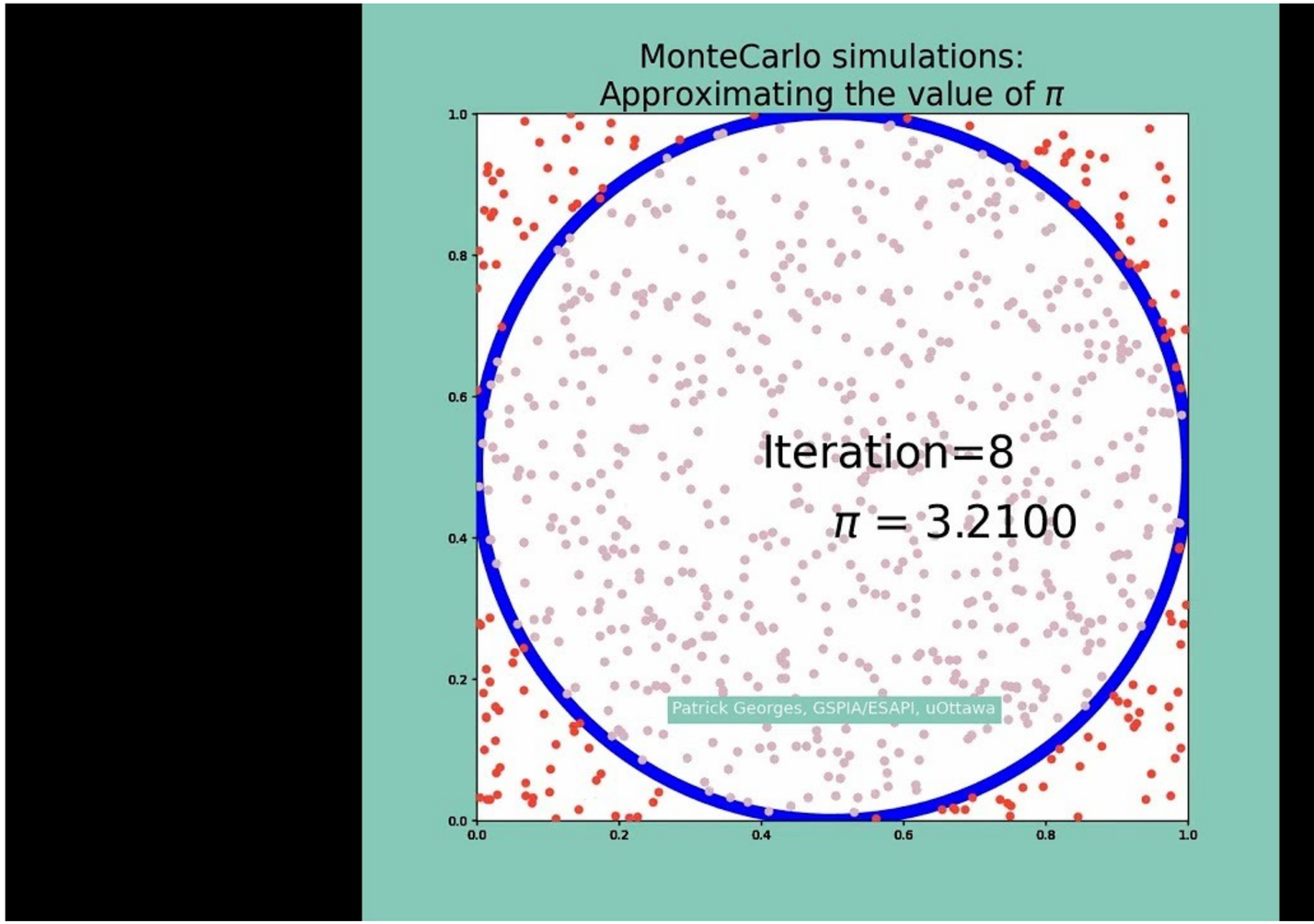
Random numbers: a Monte Carlo example

- Let’s consider a Monte Carlo simulation as an example
- Pretty simple idea: randomly sample points, count how many points are inside the circle (check if “x^2+y^2 < 1”)
- What ratio of points falls inside the circle?
  - Area of circle / area of square
  - Easier if we use a 1/4 circle.



Random numbers: a Monte Carlo example

- Let’s consider a Monte Carlo simulation as an example
- Pretty simple idea: randomly sample points, count how many points are inside the circle (check if “x^2+y^2 < 1”)
- What ratio of points falls inside the circle?
  - Area of circle / area of square
  - Easier if we use a 1/4 circle.



A Monte Carlo example

- Demo serial code using rand() - random number generator
- How to parallelize using what we've learned in class?
- What do we observe about the result?

A brief note on (pseudo)random numbers

- Random number generators aren't truly random, they typically depend on parameters and a *seed.*
  - "rand()" by default assumes the seed is 1, but this is set only on the first call.
- Since each thread is operating independently, we need to set the seed for each thread in order for the result to be deterministic.

$$X_{n+1} = (aX_n + c) \mod m$$

m, $0 < m$ — the "modulus"

a, $0 < a < m$ — the "multiplier"

c, $0 \leq c < m$ — the "increment"

X_0, $0 \leq X_0 < m$ — the "seed" or "start value"

Linear congruential generator (LCG) equation and parameters from Wikipedia