An Introduction to Parallel Programming with OpenMP

Tim Mattson

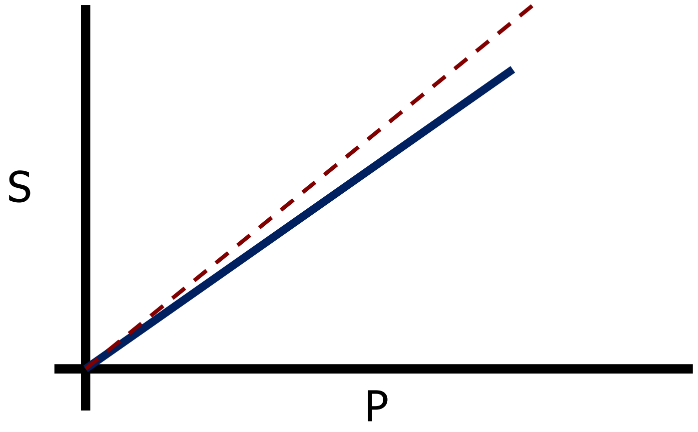
Download tutorial materials onto your laptop:
git clone https://github.com/tgmattso/CoDaS_HEP.git

An Introduction to me

I'm just a simple kayak instructor

Photo © by Greg Clopton, 2014

To support my kayaking habit, I work as a parallel programmer



Which means I know how to turn math into lines on a speedup plot

Preliminaries:

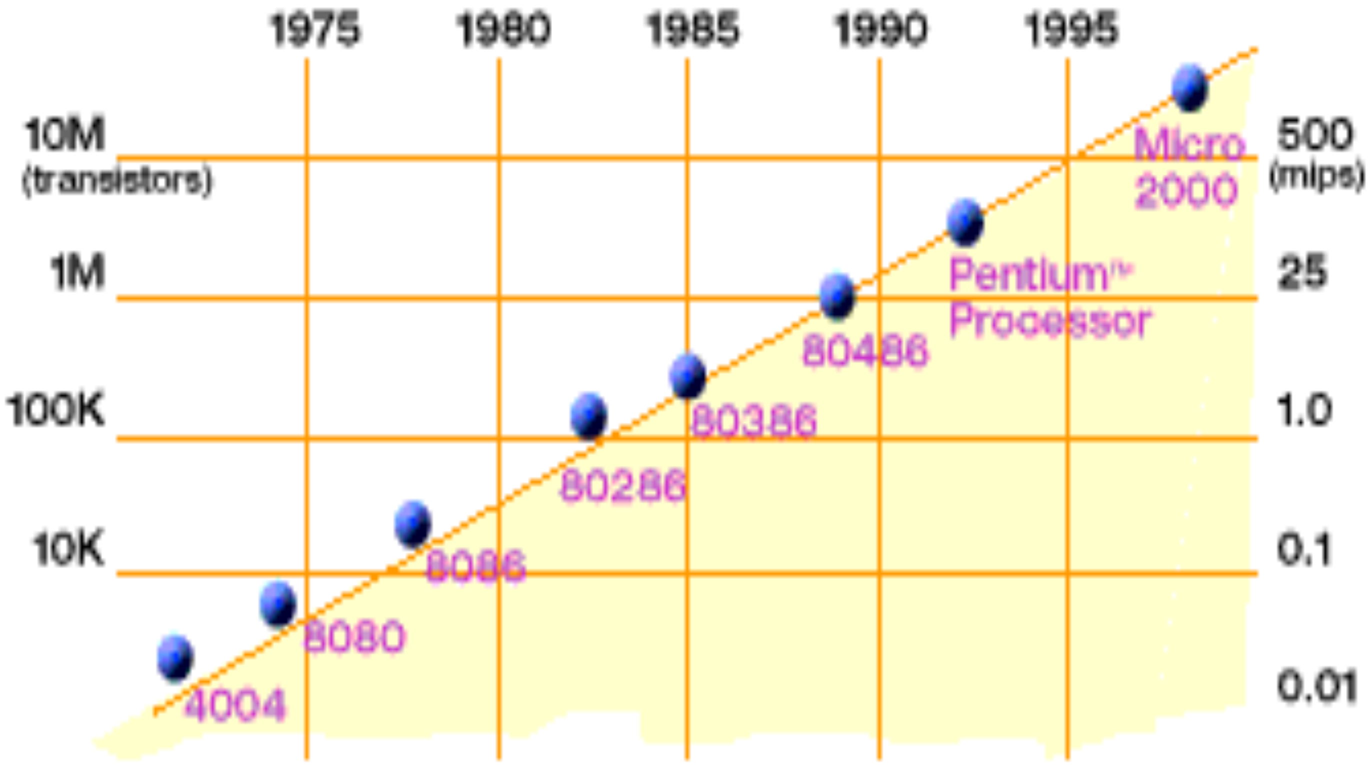
- Our plan ... Active learning!
  - We will mix short lectures with short exercises.
  - I assume you all have setup your laptops to use a gnu compiler, OR you have access to a remote server with an OpenMP compiler.

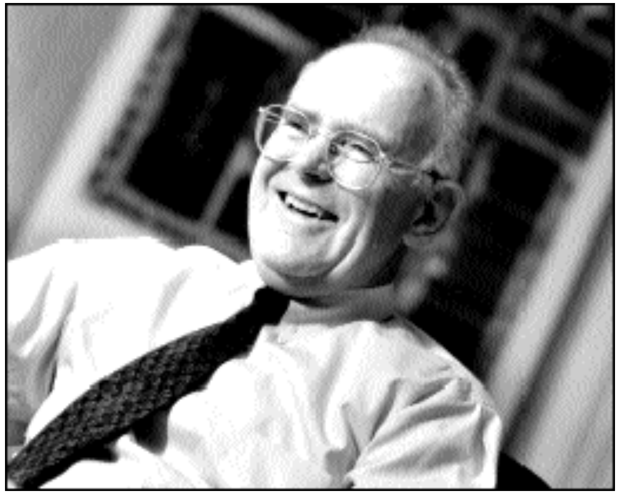
- Please follow these simple rules
  - Do the exercises that we assign and then change things around and experiment.
    - Embrace active learning!
  - <u>Don't cheat</u>: Do Not look at the solutions before you complete an exercise … even if you get really frustrated.

Download tutorial materials onto your laptop:
git clone https://github.com/tgmattso/CoDaS_HEP.git

Why do you need to understand parallel computing?

Moore's Law



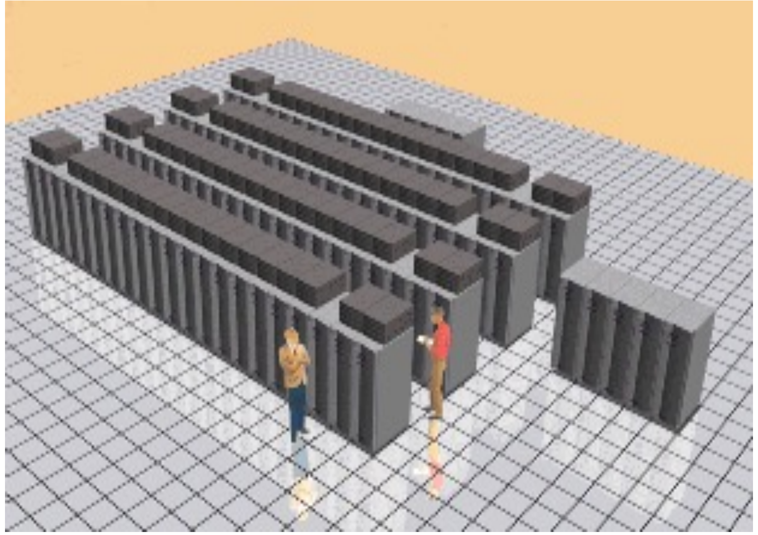


- In 1965, Intel co-founder Gordon Moore predicted (from just 3 data points!) that semiconductor density would double every 18 months.
  - ***He was right!*** Over the last 50 years, transistor densities have increased as he predicted.

Slide source: UCB CS 194 Fall’2010

Moore's Law: A personal perspective

First TeraScale* computer: 1997



Intel's ASCI Option Red

Intel's ASCI Red Supercomputer

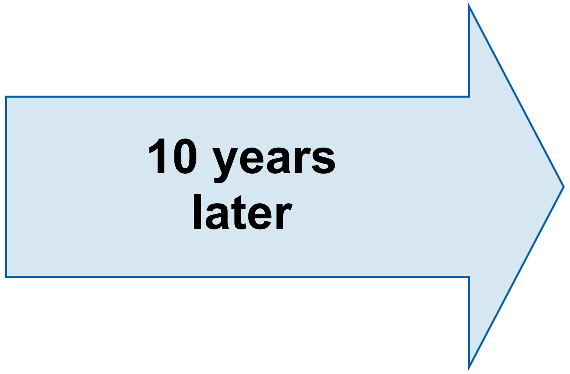
9000 CPUs

one megawatt of electricity.

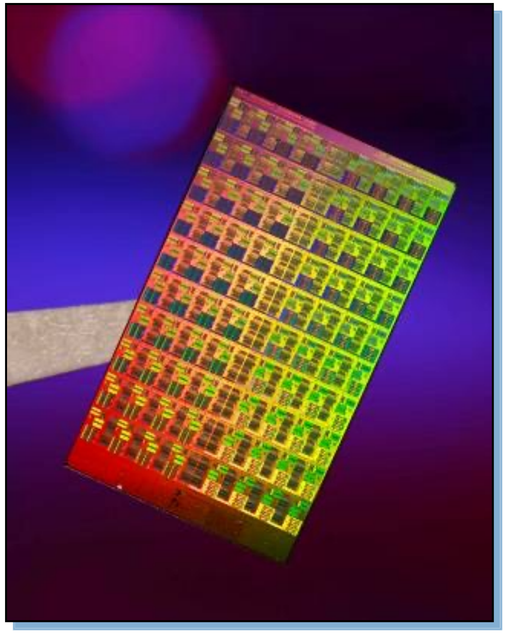
1600 square feet of floor space.

*Double Precision TFLOPS running MP-Linpack

A TeraFLOP in 1996: The ASCI TeraFLOP Supercomputer, Proceedings of the International Parallel Processing Symposium (1996), T.G. Mattson, D. Scott and S. Wheat.



First TeraScale% chip: 2007



Intel's 80 core teraScale Chip

1 CPU

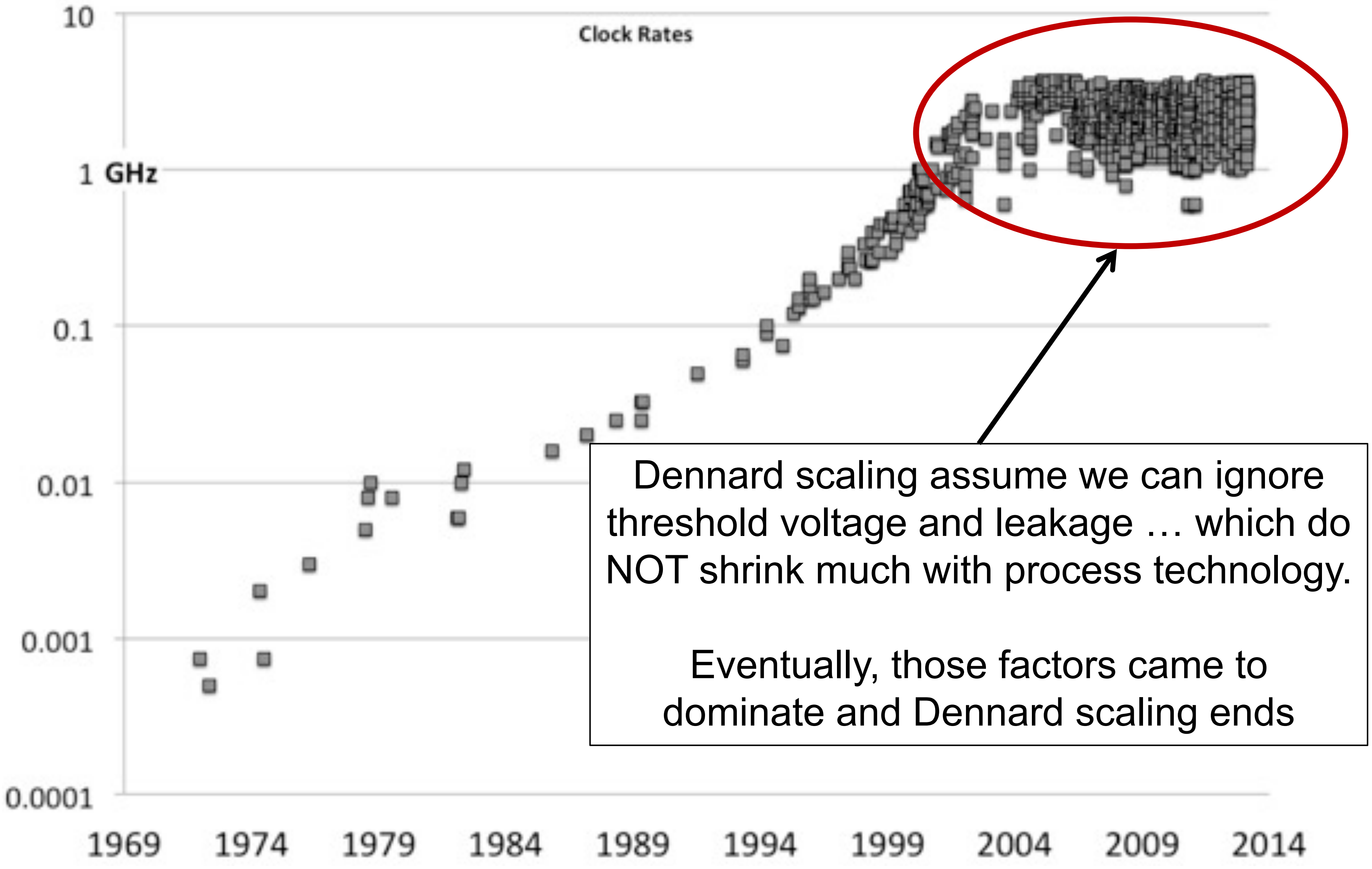
97 watt

275 mm2

%Single Precision TFLOPS running stencil

Programming Intel's 80 core terascale processor
SC08, Austin Texas, Nov. 2008, Tim Mattson, Rob van der Wijngaart, Michael Frumkin

CPU Frequency (GHz) over time (years)

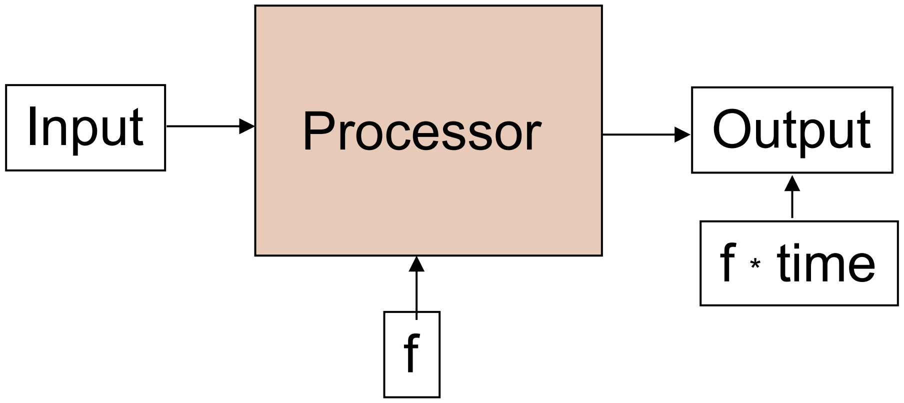


Dennard scaling assume we can ignore threshold voltage and leakage … which do NOT shrink much with process technology.

Eventually, those factors came to dominate and Dennard scaling ends

Source: James Reinders (from the book “structured parallel programming”)

Consider power in a chip …



Capacitance = C
Voltage = V
Frequency = f
Power = CV^2f

C = capacitance … it measures the ability of a circuit to store energy:

$$C = q/V \rightarrow \quad q = CV$$

Work is pushing something (charge or q) across a "distance" … in electrostatic terms pushing q from 0 to V:

$$V * q = W.$$

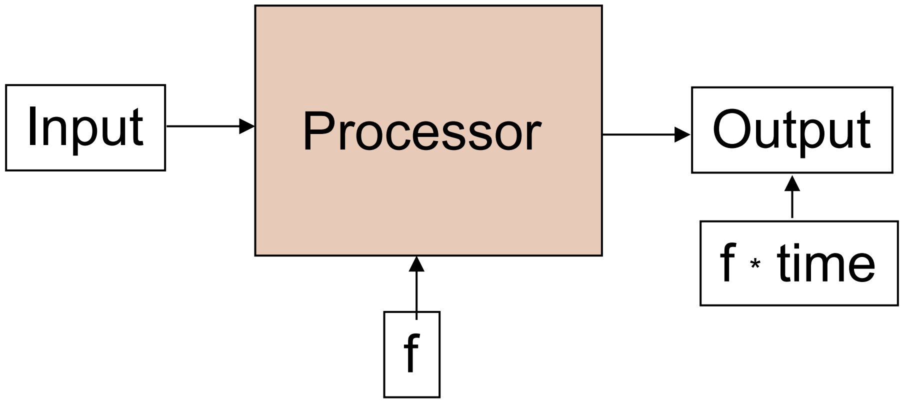
But for a circuit $q = CV$ so

$$W = CV^2$$

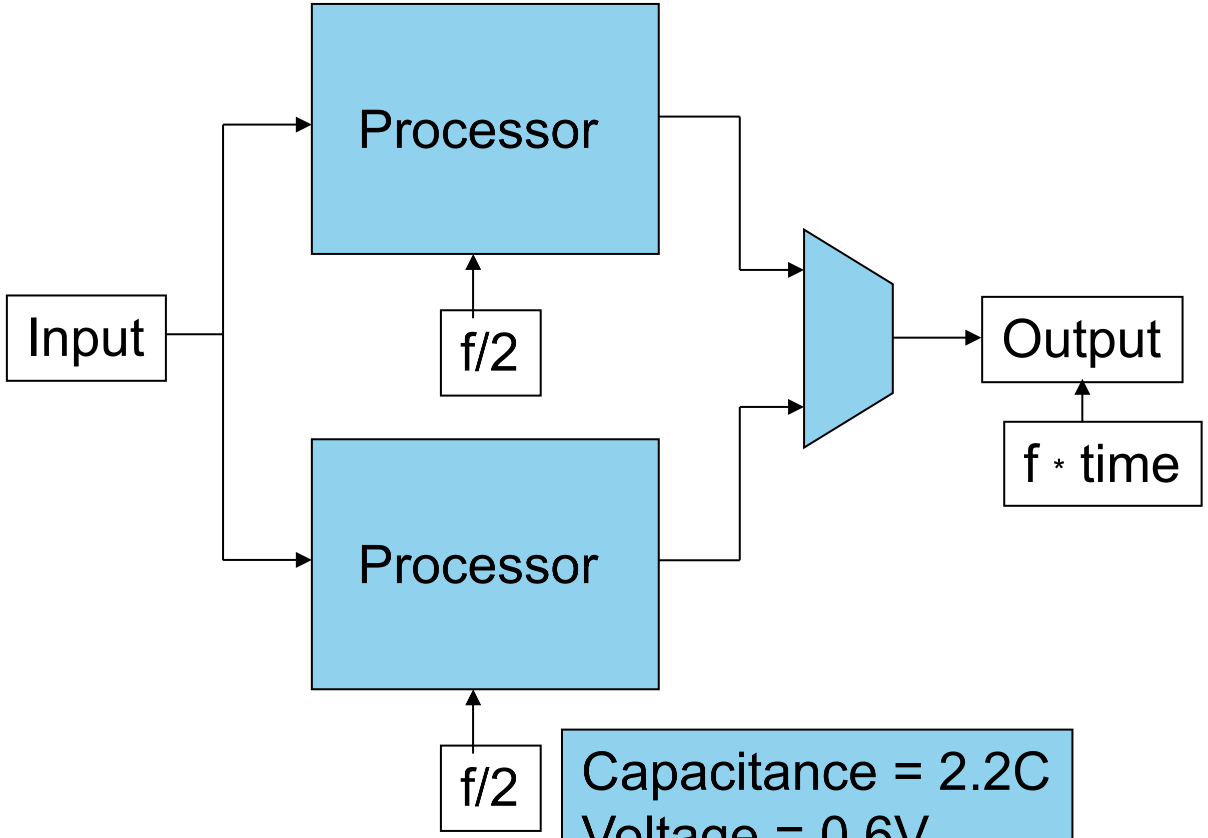
power is work over time … or how many times per second we oscillate the circuit

$$\text{Power} = W * F \quad \rightarrow \quad \text{Power} = CV^2f$$

... Reduce power by adding cores



Capacitance = C
Voltage = V
Frequency = f
Power = CV^2f



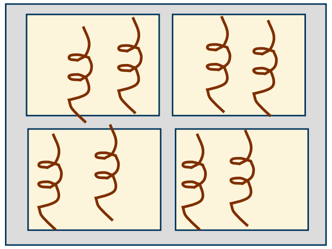
Capacitance = 2.2C
Voltage = 0.6V
Frequency = 0.5f
Power = $0.396CV^2f$

Chandrakasan, A.P.; Potkonjak, M.; Mehra, R.; Rabaey, J.; Brodersen, R.W., "Optimizing power using transformations," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,*, vol.14, no.1, pp.12-31, Jan 1995

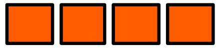
Source: Vishwani Agrawal

For hardware … parallelism is the path to performance

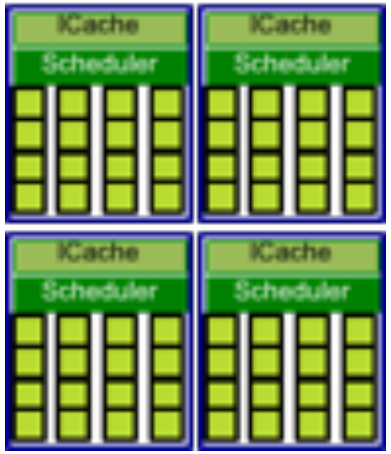
All hardware vendors are in the game … parallelism is ubiquitous so if you care about getting the most from your hardware, you will need to create parallel software.



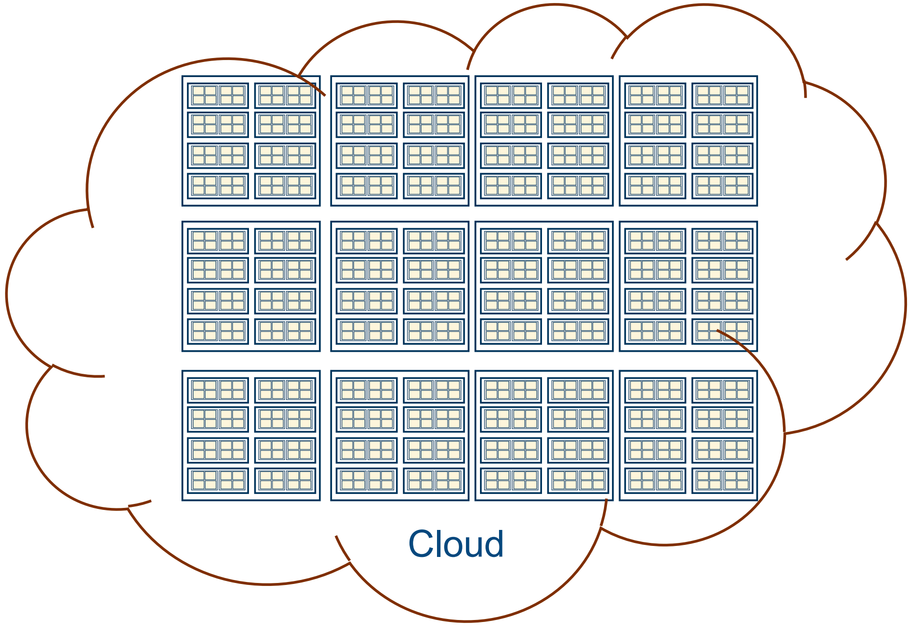
CPU



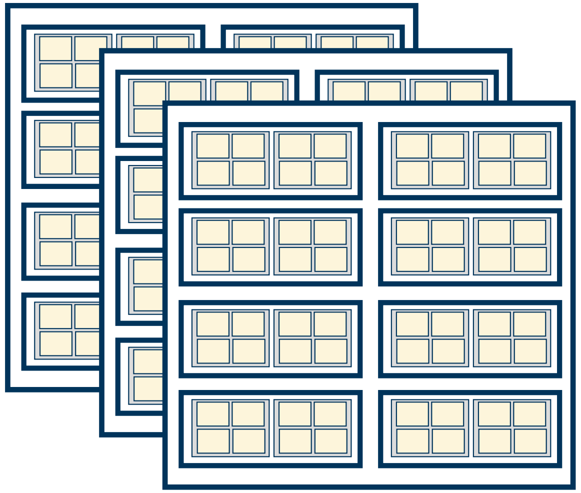
SIMD/Vector



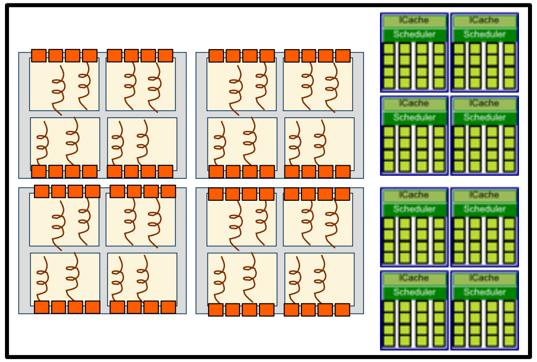
GPU



Cloud



Cluster



Heterogeneous node

The best way to master parallel computing …

start with a simple approach to parallelism and build an intellectual foundation by writing parallel code.

… and the simplest API for parallelism is?

OpenMP®

Outline

- Introduction to OpenMP
- Creating Threads
- Synchronization
- Parallel Loops
- Data Environment
- Memory Model
- Irregular Parallelism and Tasks
- OpenMP: Beyond the CPU
- Recap

OpenMP* Overview

```
C$OMP FLUSH
#pragma omp critical
#pragma omp single
C$OMP THREADPRIVATE(/ABC/)
C$OMP ATOMIC
CALL OMP_SET_NUM_THREADS(10)
cal
C$O
ED
```

OpenMP: An API for Writing Parallel Applications

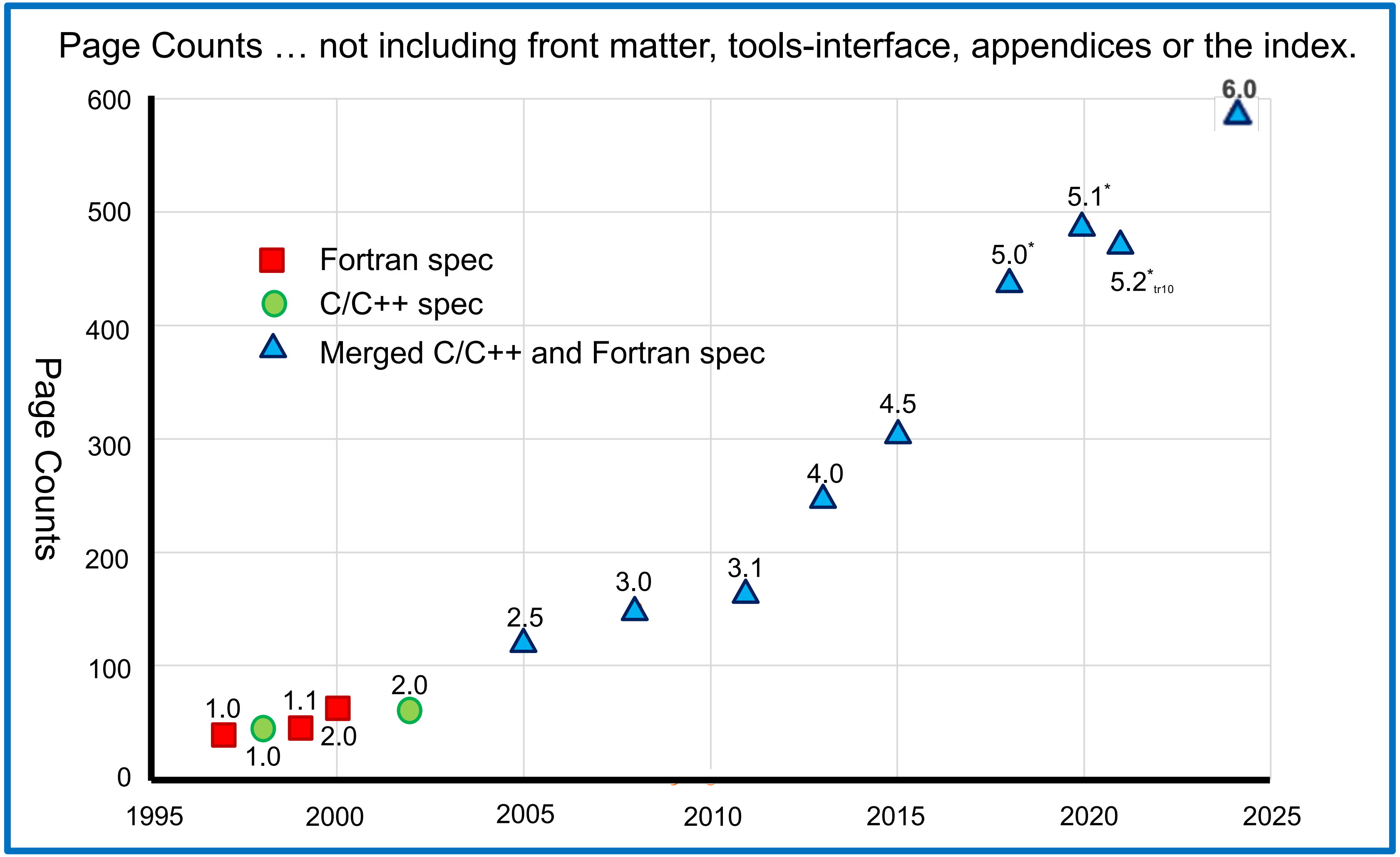
- A set of compiler directives and library routines for parallel application programmers
- Originally … Greatly simplifies writing multithreaded programs in Fortran, C and C++
- Later versions … supports non-uniform memories, vectorization and GPU programming

```
#pragma omp parallel for private(A, B)
C$OMP PARALLEL REDUCTION (+: A, B)
C$OMP PARALLEL COPYIN(/blk/)
C$OMP DO lastprivate(XX)
#pragma omp atomic seq_cst
Nthrds = OMP_GET_NUM_PROCS()
omp_set_lock(lck)
```

* The name “OpenMP” is the property of the OpenMP Architecture Review Board.

The Growth of Complexity in OpenMP

Our goal in 1997 … A simple interface for application programmers

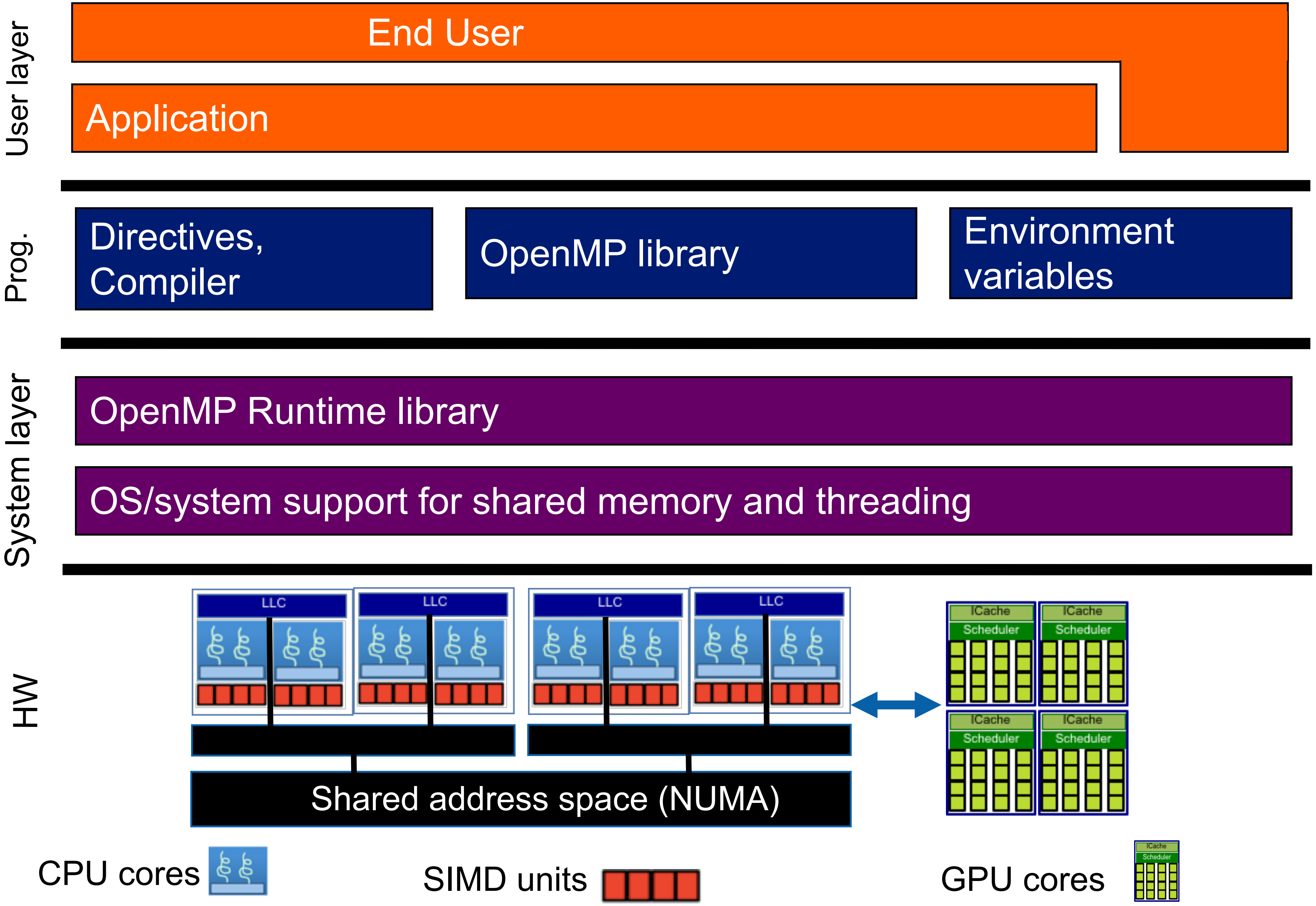


The OpenMP specification is so long and complex that few (if any) humans understand the full document

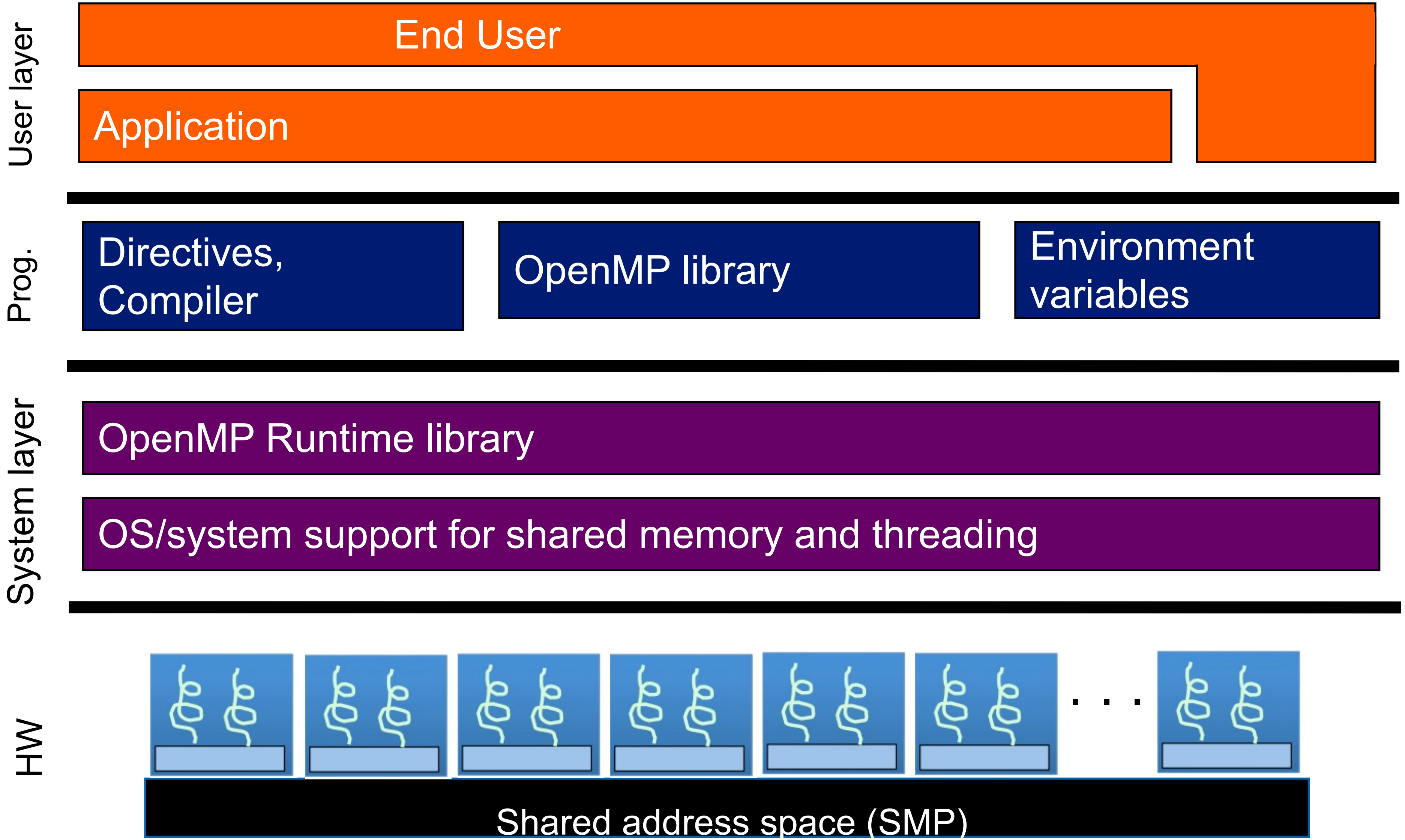
The OpenMP Common Core: Most OpenMP programs only use these 21 items

OpenMP pragma, function, or clause	Concepts
#pragma omp parallel	Parallel region, teams of threads, structured block, interleaved execution across threads.
void omp_set_thread_num() int omp_get_thread_num() int omp_get_num_threads()	Default number of threads and internal control variables. SPMD pattern: Create threads with a parallel region and split up the work using the number of threads and the thread ID.
double omp_get_wtime()	Speedup and Amdahl's law. False sharing and other performance issues.
setenv OMP_NUM_THREADS N	Setting the internal control variable for 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 (static [,chunk]) schedule(dynamic [,chunk])	Loop schedules, loop overheads, and load balance.
shared(list), private(list), firstprivate(list)	Data environment.
default(none)	Force explicit definition of each variable's storage attribute
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 Basic Definitions: Basic Solution Stack



OpenMP Basic Definitions: Basic Solution Stack



For the OpenMP Common Core, we focus on Symmetric Multiprocessor Case ….
i.e., lots of threads with “equal cost access” to memory

OpenMP Basic Syntax

- Most of OpenMP happens through compiler directives.

C and C++	Fortran
Compiler directives	
#pragma omp construct [clause [clause]…]	***!$OMP construct [clause [clause] …]***
Example	
#pragma omp parallel private(x) ***{*** ***}***	***!$OMP PARALLEL PRIVATE(X)*** ***!$OMP END PARALLEL***
Function prototypes and types:	
#include <omp.h>	***use OMP_LIB***

- Most OpenMP constructs apply to a "structured block".
  - **Structured block**: a block of one or more statements with one point of entry at the top and one point of exit at the bottom.
  - It's OK to have an exit() within the structured block.

Exercise, Part A: Hello World

Verify that your environment works

- Write a program that prints “hello world”.

```
#include<stdio.h>
int main()
{




     printf(“ hello ”);
     printf(“ world \n”);

}
```

Download tutorial materials onto your laptop:
git clone https://github.com/tgmattso/CoDaS_HEP.git

Exercise, Part B: Hello World

Verify that your OpenMP environment works

- Write a multithreaded program that prints “hello world”.

```
#include <omp.h>
#include <stdio.h>
int main()
{
 #pragma omp parallel
 {

    printf(" hello ");
    printf(" world \n");
 }
}
```

Switches for compiling and linking

gcc -fopenmp	**Gnu (Linux, OSX)**
cc –qopenmp	**Intel (Linux@NERSC)**
icc -fopenmp	**Intel (Linux, OSX)**

Solution

A Multi-Threaded “Hello World” Program

- Write a multithreaded program where each thread prints “hello world”.

```
#include <omp.h>
#include <stdio.h>
int  main()
{

#pragma omp parallel
 {


    printf(" hello ");
    printf(" world \n");
  }
}
```

OpenMP include file

Parallel region with default number of threads

End of the Parallel region

Sample Output:

hello hello world

world

hello hello world

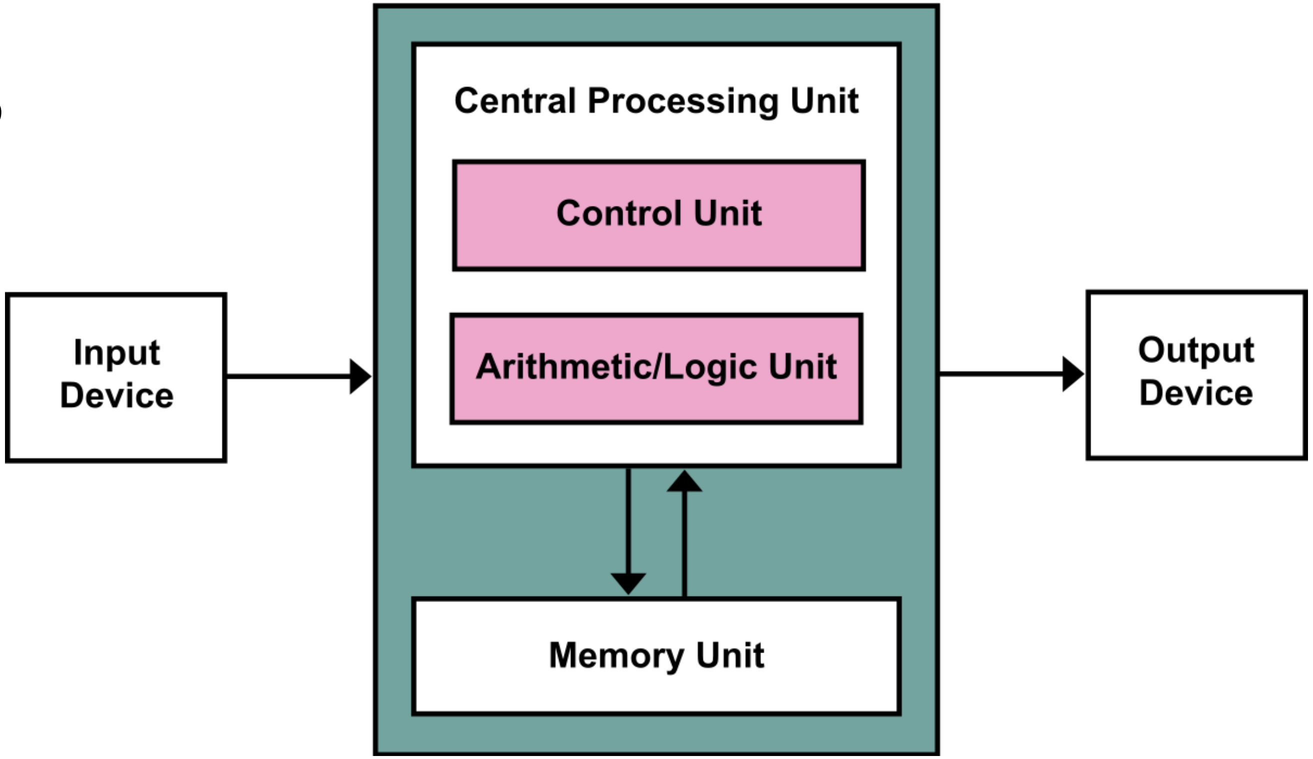
world

The statements are interleaved based on how the operating schedules the threads

A brief digression on the terminology of parallel computing

Let's agree on a few definitions:

- **Computer:**
  - A machine that transforms *input values* into *output values*.
  - Typically, a computer consists of Control, Arithmetic/Logic, and Memory units.
  - The transformation is defined by a stored **program** (von Neumann architecture).



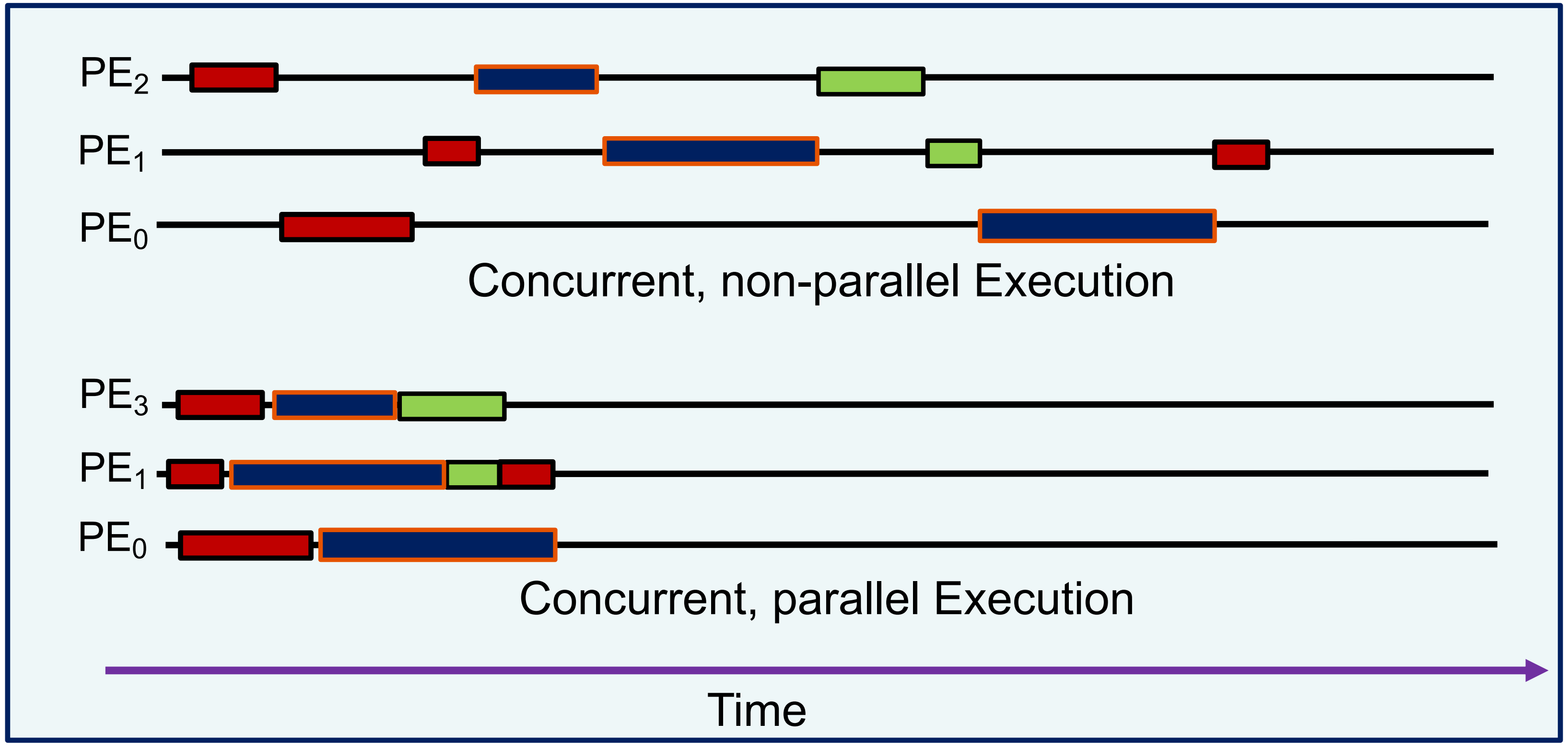
- **Task:**
  - A sequence of instructions plus a data environment. A program is composed of one or more tasks.

- **Active task:**
  - A task that is available to be scheduled for execution. When the task is moving through its sequence of instructions, we say it is making **forward progress**

- **Fair scheduling:**
  - When a scheduler gives each active task an equal *opportunity* for execution.

Concurrency vs. Parallelism

- Two important definitions:
  - Concurrency: A condition of a system in which multiple tasks are active and unordered. If **scheduled fairly**, they can be described as ***logically*** making **forward progress** at the same time.
  - Parallelism: A condition of a system in which multiple tasks are **actually** making **forward progress** at the same time.



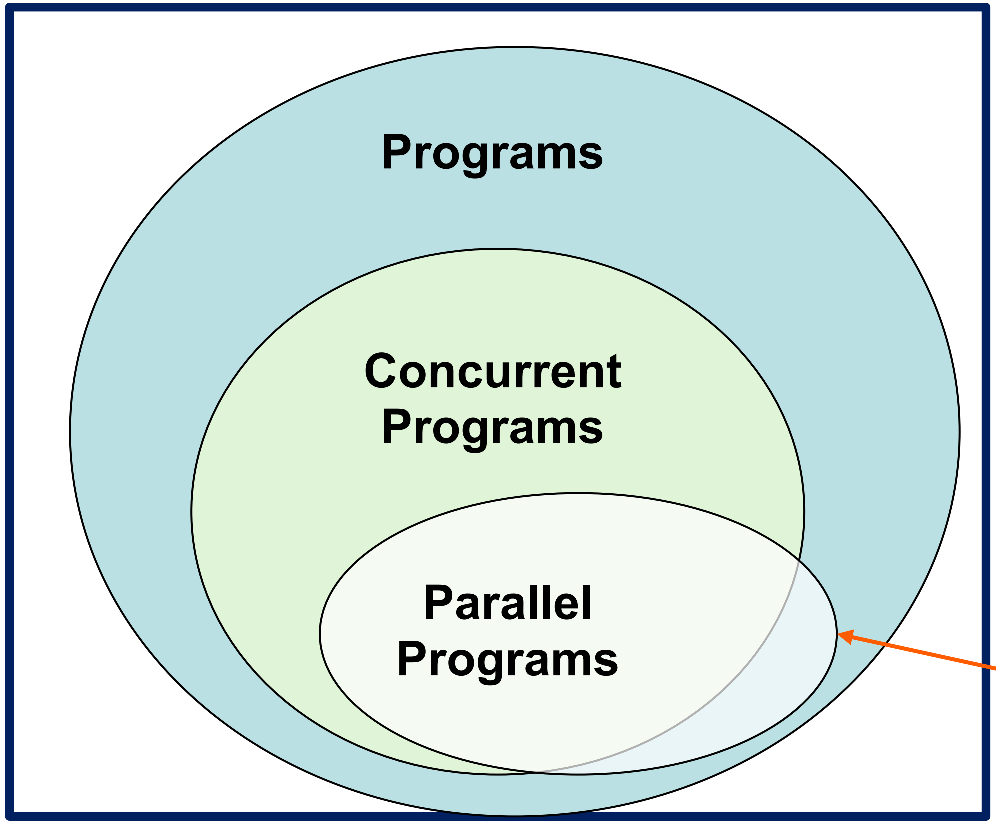
PE = Processing Element

Figure from "An Introduction to Concurrency in Programming Languages" by J. Sottile, Timothy G. Mattson, and Craig E Rasmussen, 2010

Concurrency vs. Parallelism

- Two important definitions:
  - <u>Concurrency</u>: A condition of a system in which multiple tasks are active and unordered. If **scheduled fairly**, they can be described as ***<u>logically</u>*** making **forward progress** at the same time.

  - <u>Parallelism</u>: A condition of a system in which multiple tasks are **<u>actually</u>** making **forward progress** at the same time.



In most cases, parallel programs exploit concurrency in a problem to run tasks on multiple processing elements

We use Parallelism to:
- Do more work in less time
- Work with larger problems

If tasks execute in "lock step" they are not concurrent, but they are still parallel. Example … a SIMD unit.

Figure from "An Introduction to Concurrency in Programming Languages" by J. Sottile, Timothy G. Mattson, and Craig E Rasmussen, 2010

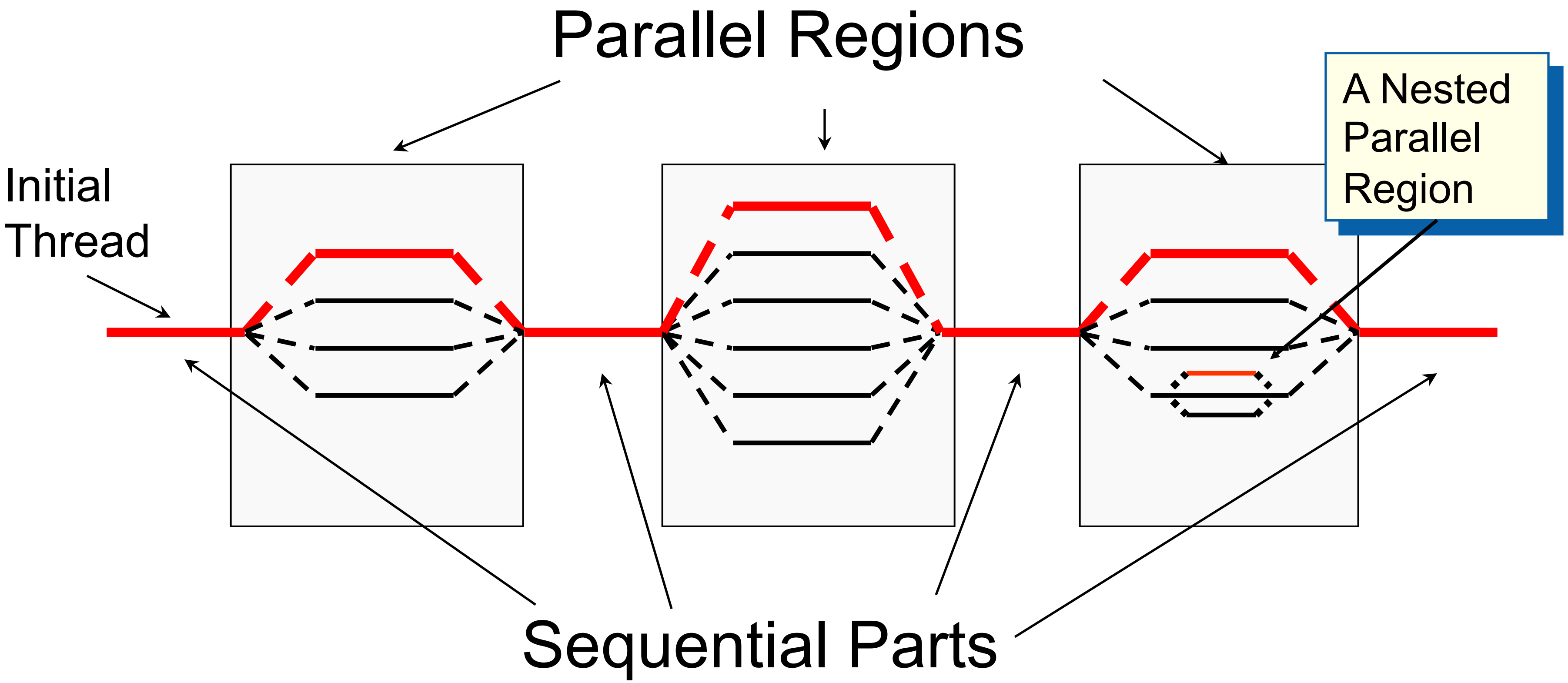
Outline

- Introduction to OpenMP
- Creating Threads
- Synchronization
- Parallel Loops
- Data Environment
- Memory Model
- Irregular Parallelism and Tasks
- OpenMP: Beyond the CPU
- Recap

OpenMP Execution model:

Fork-Join Parallelism:

- Initial 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.



Thread Creation: Parallel Regions

- You create threads in OpenMP with the parallel construct.
- For example, to create a 4 thread Parallel region:

Each thread executes a copy of the code within the structured block

```
double A[1000];
omp_set_num_threads(4);
#pragma omp parallel
{
        int ID = omp_get_thread_num();
        pooh(ID,A);
}
```

Runtime function to request a certain number of threads

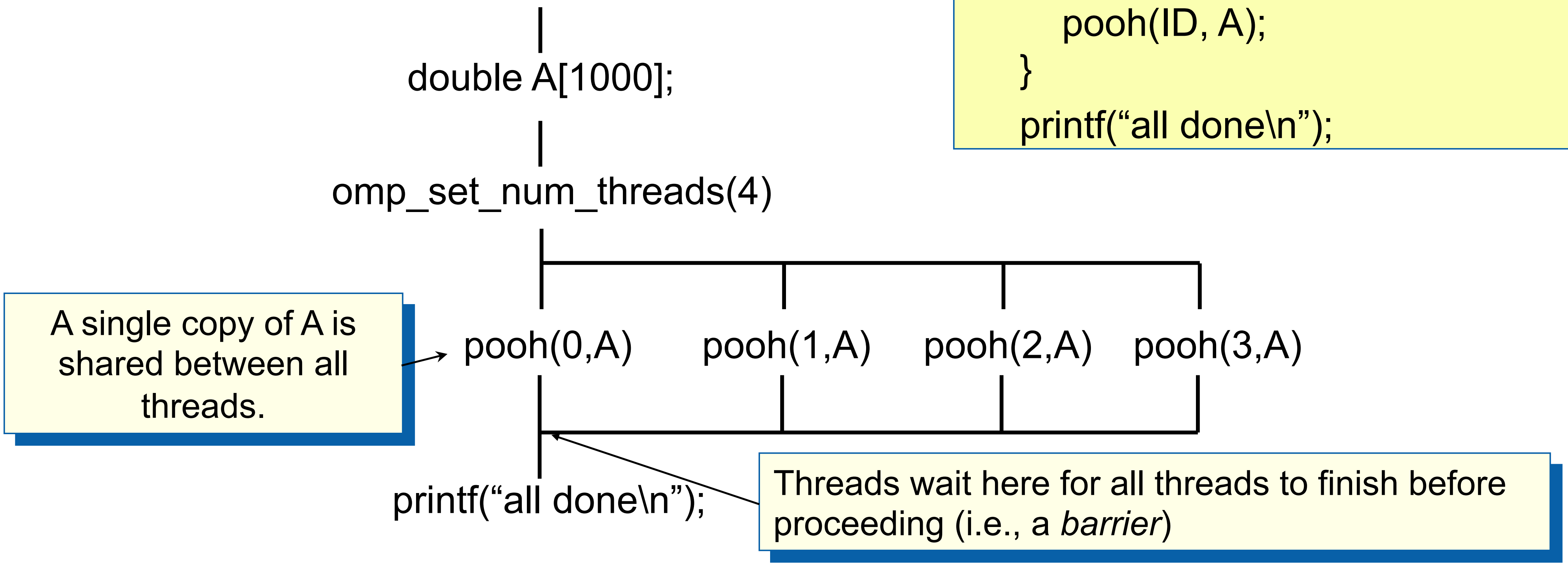
Runtime function returning a thread ID

- Each thread calls pooh(ID,A) for ID = 0 to 3

Thread Creation: Parallel Regions Example

- Each thread executes the same code redundantly.

```
double A[1000];
omp_set_num_threads(4);
#pragma omp parallel
{
    int ID = omp_get_thread_num();
    pooh(ID, A);
}
printf("all done\n");
```



Thread creation: How many threads did you actually get?

- Request a number of threads with omp_set_num_threads()

- The number requested may not be the number you actually get.
  - An implementation may silently give you fewer threads than you requested.
  - Once a team of threads has launched, it will not be reduced.

Each thread executes a copy of the code within the structured block

```
double A[1000];
omp_set_num_threads(4);
#pragma omp parallel
{
        int ID      = omp_get_thread_num();
        int nthrds = omp_get_num_threads();
        pooh(ID,A);
}
```

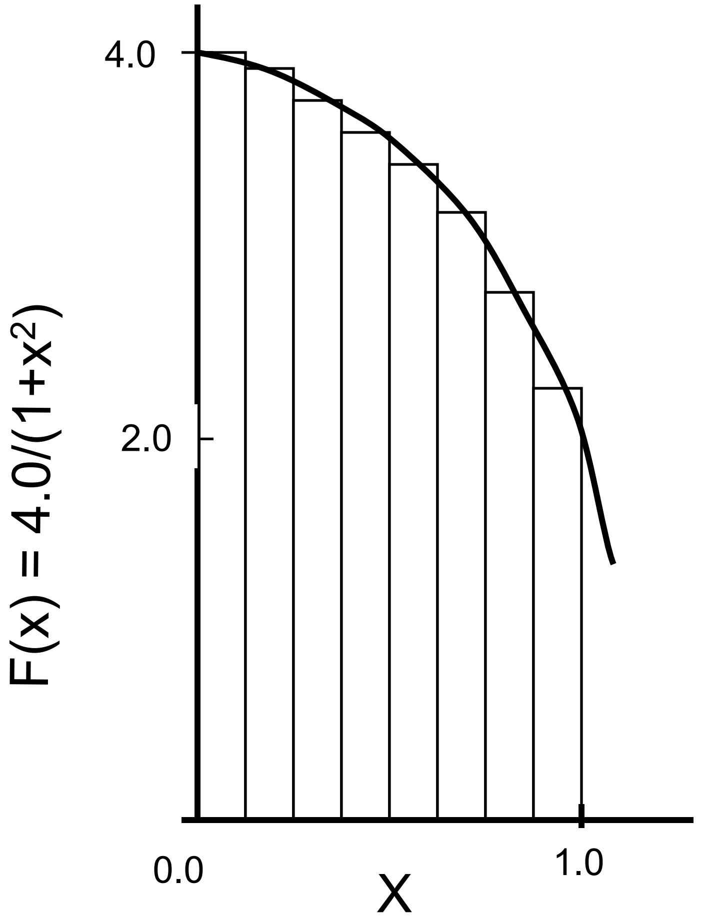
Runtime function to request a certain number of threads

Runtime function to return actual number of threads in the team

- Each thread calls pooh(ID,A) for ID = 0 to nthrds-1

An Interesting Problem to Play With

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 N rectangles:

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

Where each rectangle has width Δx and height $F(x_i)$ at the middle of interval i.

Serial PI Program

```
static long num_steps = 100000;
double step;
int main ()
{           double x, pi, sum = 0.0;

            step = 1.0/(double) num_steps;

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

Serial PI Program

```
#include <omp.h>
static long num_steps = 100000;
double step;
int main ()
{          double x, pi, sum = 0.0;

           step = 1.0/(double) num_steps;
           double tdata = omp_get_wtime();
           for (int i=0;i< num_steps; i++){
                   x = (i+0.5)*step;
                   sum = sum + 4.0/(1.0+x*x);
           }
           pi = step * sum;
           tdata = omp_get_wtime() - tdata;
           printf(" pi = %f in %f secs\n",pi, tdata);
}
```

The library routine get_omp_wtime() is used to find the elapsed "wall time" for blocks of code

Exercise: the Parallel Pi Program

- Create a parallel version of the pi program using a parallel construct:

  #pragma omp parallel

- Pay close attention to shared versus private variables.
- In addition to a parallel construct, you will need the runtime library routines
  - int omp_get_num_threads(); — Number of threads in the team
  - int omp_get_thread_num(); — Thread ID or rank
  - double omp_get_wtime(); — Time in seconds since a fixed point in the past
  - omp_set_num_threads(); — Request a number of threads in the team

git clone https://github.com/tgmattso/ParProgForPhys.git

Hints: the Parallel Pi Program

- Use a parallel construct:

  #pragma omp parallel

- The challenge is to:
  - divide loop iterations between threads (use the thread ID and the number of threads).
  - Create an accumulator for each thread to hold partial sums that you can later combine to generate the global sum.

- In addition to a parallel construct, you will need the runtime library routines
  - int omp_set_num_threads();
  - int omp_get_num_threads();
  - int omp_get_thread_num();
  - double omp_get_wtime();

Example: A simple SPMD* pi program

```
#include <omp.h>
static long num_steps = 100000;        double step;
#define NUM_THREADS 2
void main ()
{    int i, nthreads;  double pi, sum[NUM_THREADS];
     step = 1.0/(double) num_steps;
     omp_set_num_threads(NUM_THREADS);
    #pragma omp parallel
    {
            int i, id,numthrds;
            double x;
            id = omp_get_thread_num();
            numthrds = omp_get_num_threads();
            if (id == 0)   nthreads = numthrds;
              for (i=id, sum[id]=0.0;i< num_steps; i=i+numthrds) {
                        x = (i+0.5)*step;
                        sum[id] += 4.0/(1.0+x*x);
              }
     }
     for(i=0, pi=0.0;i<nthreads;i++) pi += sum[i] * step;
}
```

Promote scalar to an array dimensioned by number of threads to avoid race condition.

Only one thread should copy the number of threads to the global value to make sure multiple threads writing to the same address don't conflict.

This is a common trick in SPMD programs to create a cyclic distribution of loop iterations

*SPMD: Single Program Multiple Data

Example: A simple SPMD pi program … an alternative solution

```
#include <omp.h>
static long num_steps = 100000;        double step;
#define NUM_THREADS 2
void main ()
{    int i, nthreads;  double pi, sum[NUM_THREADS];
     step = 1.0/(double) num_steps;
     omp_set_num_threads(NUM_THREADS);
    #pragma omp parallel
    {
             int i, id,numthrds, istart, iend;
             double x;
             id = omp_get_thread_num();
             numthrds = omp_get_num_threads();
             istart = id*(num_steps/numthrds );      iend=(id+1)*(num_steps/numthrds);
             if(id == (numthrds-1)) iend = num_steps;
             if (id == 0)   nthreads = numthrds;
               for (i=istart, sum[id]=0.0;i< iend; i++) {
                          x = (i+0.5)*step;
                          sum[id] += 4.0/(1.0+x*x);
               }
     }
     for(i=0, pi=0.0;i<nthreads;i++) pi += sum[i] * step;
}
```

This is a common trick in SPMD algorithms … it's a **blocked distribution** with one block per thread.

SPMD: Single Program Multiple Data

Results*

- Original Serial pi program with 100000000 steps ran in 1.83 seconds.

```
#include <omp.h>
static long num_steps = 100000;        double step;
#define NUM_THREADS 2
void main ()
{    int i, nthreads;  double pi, sum[NUM_THREADS];
     step = 1.0/(double) num_steps;
     omp_set_num_threads(NUM_THREADS);
    #pragma omp parallel
    {
            int i, id,nthrds;
            double x;
            id = omp_get_thread_num();
            nthrds = omp_get_num_threads();
            if (id == 0)   nthreads = nthrds;
            for (i=id, sum[id]=0.0;i< num_steps; i=i+nthrds) {
                    x = (i+0.5)*step;
                    sum[id] += 4.0/(1.0+x*x);
            }
    }
    for(i=0, pi=0.0;i<nthreads;i++)pi += sum[i] * step;
}
```

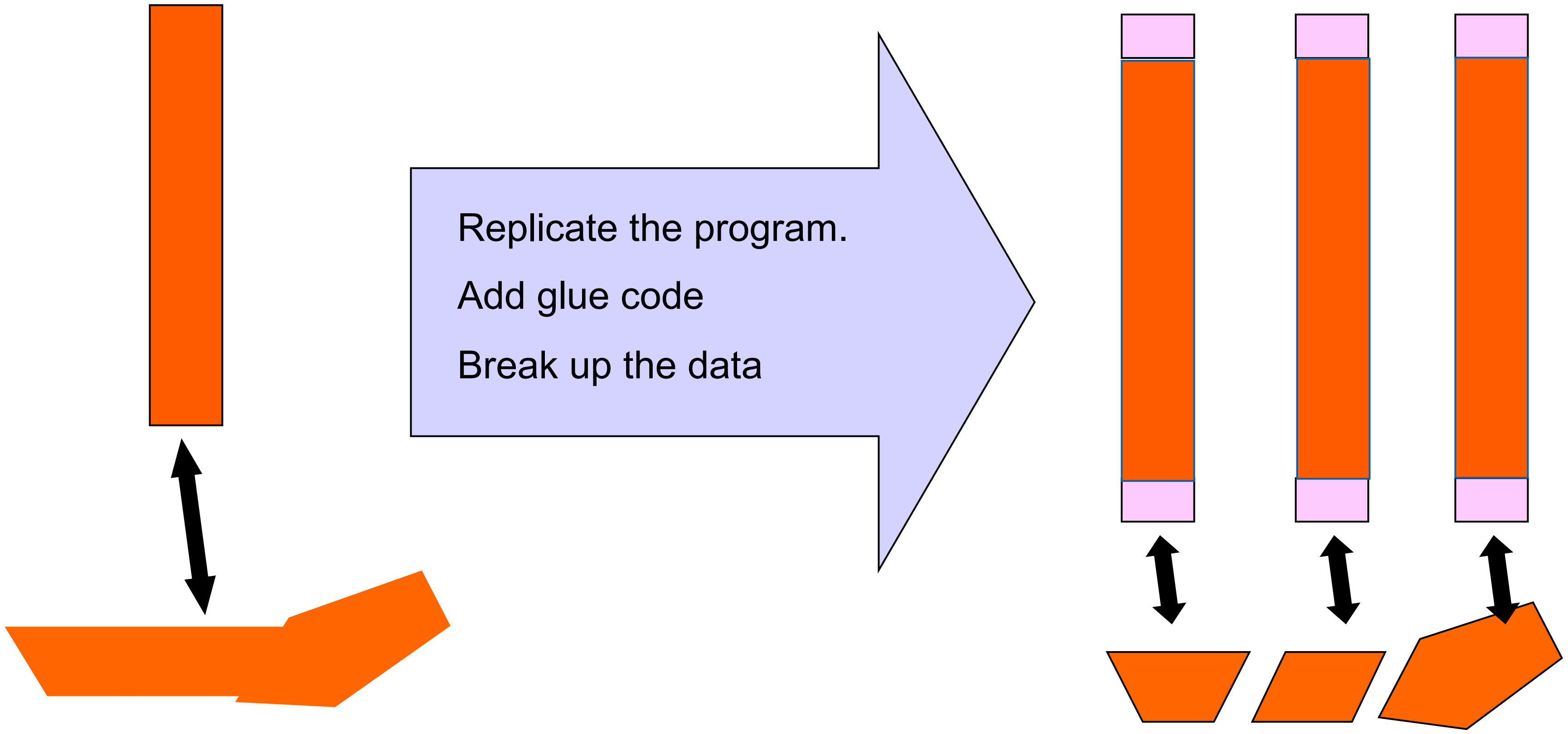
threads	1st SPMD*
1	1.86
2	1.03
3	1.08
4	0.97

Intel compiler (icpc) with default optimization level (O2) on Apple OS X 10.7.3 with a dual core (four HW thread) Intel® Core™ i5 processor at 1.7 Ghz and 4 Gbyte DDR3 memory at 1.333 Ghz.

*SPMD: Single Program Multiple Data

SPMD: Single Program Multiple Data

- Run the same program on P processing elements where P can be arbitrarily large.



- Use the rank … an ID ranging from 0 to (P-1) … to select between a set of tasks and to manage any shared data structures.

MPI programs almost always use this pattern … it is probably the most commonly used pattern in the history of parallel programming.

How do we describe performance in parallel programs

Consider performance of parallel programs

Compute N independent tasks on one processor

Load Data | Compute T_1 | … | Compute T_N | Consume Results

$$Time_{seq}(1) = T_{load} + N*T_{task} + T_{consume}$$

Compute N independent tasks with P processors

Load Data

Compute T_1
…
Compute T_N

Consume Results

Ideally Cut runtime by ~1/P

(Note: Parallelism only speeds-up the concurrent part)

$$Time_{par}(P) = T_{load} + (N/P)*T_{task} + T_{consume}$$

Talking about performance

- Speedup: the increased performance from running on P processors.

$$S(P) = \frac{Time_{seq}(1)}{Time_{par}(P)}$$

- Perfect Linear Speedup: happens when no parallel overhead and algorithm is 100% parallel.

$$S(P) = P$$

- Efficiency: How well does your observed speedup compare to the ideal case?

$$\varepsilon(P) = \frac{S(P)}{P}$$

Amdahl's Law

- What is the maximum speedup you can expect from a parallel program?

- Approximate the runtime as a part that can be sped up with additional processors and a part that is fundamentally serial.

$$Time_{par}(P) = (serial\_fraction + \frac{parallel\_fraction}{P}) * Time_{seq}$$

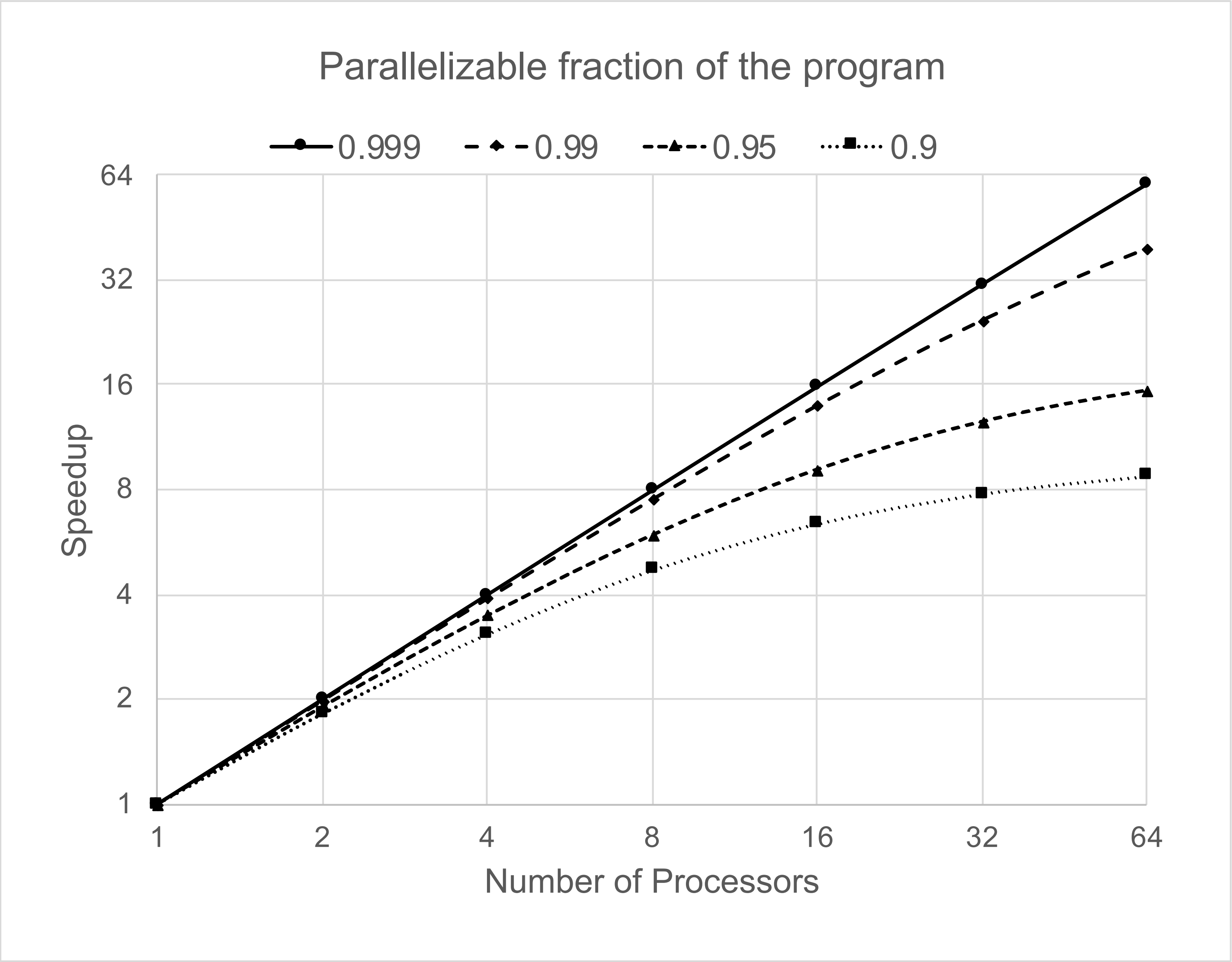
- If the serial fraction is α and the parallel fraction is (1- α) then the speedup is:

$$S(P) = \frac{Time_{seq}}{Time_{par}(P)} = \frac{Time_{seq}}{(\alpha + \frac{1-\alpha}{P}) * Time_{seq}} = \frac{1}{\alpha + \frac{1-\alpha}{P}}$$

- If you had an unlimited number of processors: $P \rightarrow \infty$

- The maximum possible speedup is: $S = \frac{1}{\alpha}$ ← Amdahl's Law

Amdahl's Law … It's not just about the maximum speedup

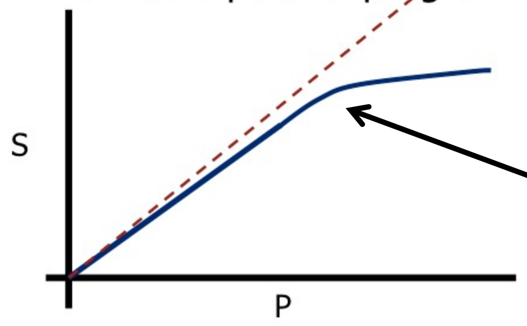


So now you should understand my silly introduction slide.

Introduction

Photo © by Greg Clopton, 2014

To support my kayaking habit I work as a parallel programmer



Which means I know how to turn math into lines on a speedup plot

4

We measure our success as parallel programmers by how close we come to ideal linear speedup.

A good parallel programmer always figures out when you fall off the linear speedup curve and why that has occurred.

…. Now that we know how to describe performance for parallel computations, lets get back to OpenMP

Internal control variables and how to control the number of threads in a team

- We've used the following construct to control the number of threads. (e.g. to request 12 threads):
  - omp_set_num_threads(12)

- What does omp_set_num_threads() actually do?
  - It **resets** an "**internal control variable**" the system queries to select the default number of threads to request on subsequent parallel constructs.

- Is there an easier way to change this internal control variable … perhaps one that doesn't require re-compilation? Yes.
  - When an OpenMP program starts up, it queries an environment variable OMP_NUM_THREADS and sets the appropriate internal control variable to the value of **OMP_NUM_THREADS**
  - For example, to set the initial, default number of threads to request in OpenMP from my apple laptop

```
> export OMP_NUM_THREADS=12
```

Exercise

- Go back to your parallel pi program and explore how well it scales with the number of threads.
- Can you explain your performance with Amdahl's law? If not what else might be going on?
  - int omp_get_num_threads();
  - int omp_get_thread_num();
  - double omp_get_wtime();
  - omp_set_num_threads();
  - export OMP_NUM_THREADS = N

An environment variable to set the default number of threads to request to N

Results*

- Original Serial pi program with 100000000 steps ran in 1.83 seconds.

```
#include <omp.h>
static long num_steps = 100000;         double step;
#define NUM_THREADS 2
void main ()
{    int i, nthreads;  double pi, sum[NUM_THREADS];
     step = 1.0/(double) num_steps;
     omp_set_num_threads(NUM_THREADS);
    #pragma omp parallel
    {
            int i, id,nthrds;
            double x;
            id = omp_get_thread_num();
            nthrds = omp_get_num_threads();
            if (id == 0)   nthreads = nthrds;
            for (i=id, sum[id]=0.0;i< num_steps; i=i+nthrds) {
                    x = (i+0.5)*step;
                    sum[id] += 4.0/(1.0+x*x);
            }
    }
    for(i=0, pi=0.0;i<nthreads;i++)pi += sum[i] * step;
}
```

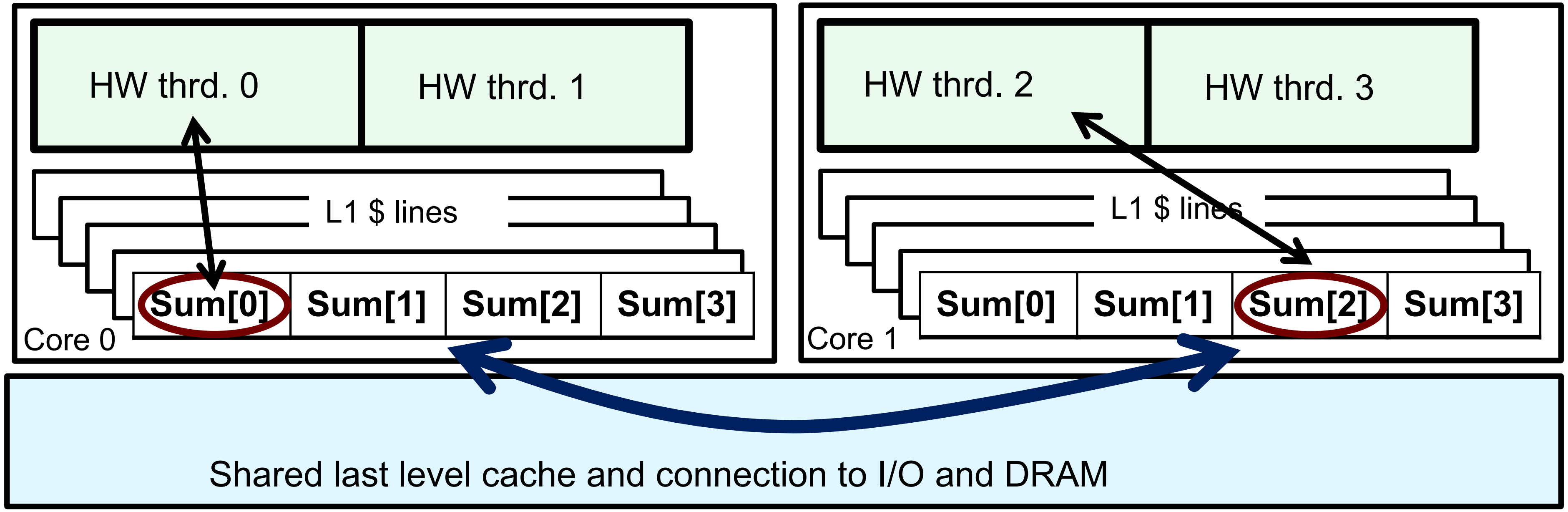
threads	1^{st} SPMD*
1	1.86
2	1.03
3	1.08
4	0.97

Intel compiler (icpc) with default optimization level (O2) on Apple OS X 10.7.3 with a dual core (four HW thread) Intel® Core™ i5 processor at 1.7 Ghz and 4 Gbyte DDR3 memory at 1.333 Ghz.

*SPMD: Single Program Multiple Data

Why Such Poor Scaling? 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”**.



- If you promote scalars to an array to support creation of an SPMD program, the array elements are contiguous in memory and hence share cache lines … Results in poor scalability.
- Solution: Pad arrays so elements you use are on distinct cache lines.

Example: Eliminate false sharing by padding the sum array

```
#include <omp.h>
static long num_steps = 100000;        double step;
#define NUM_THREADS 2
#define PAD 8        // assume 64 byte L1 cache line size
void main ()
{    int i, nthreads;  double pi, sum[NUM_THREADS][PAD] ;
     step = 1.0/(double) num_steps;
     omp_set_num_threads(NUM_THREADS);
     #pragma omp parallel
     {
          int i, id,nthrds;
          double x;
          id = omp_get_thread_num();
          nthrds = omp_get_num_threads();
          if (id == 0)   nthreads = nthrds;
          for (i=id, sum[id]=0.0;i< num_steps; i=i+nthrds) {
               x = (i+0.5)*step;
               sum[id][0] += 4.0/(1.0+x*x);
          }
     }
     for(i=0, pi=0.0;i<nthreads;i++)pi += sum[i][0] * step;
}
```

Pad the array so each sum value is in a different cache line

Results*: PI Program, Padded Accumulator

- Original Serial pi program with 100000000 steps ran in 1.83 seconds.

```
#include <omp.h>
static long num_steps = 100000;         double step;
#define NUM_THREADS 2
#define PAD 8         // assume 64 byte L1 cache line size
void main ()
{    int i, nthreads;  double pi, sum[NUM_THREADS][PAD] ;
     step = 1.0/(double) num_steps;
     omp_set_num_threads(NUM_THREADS);
     #pragma omp parallel
     {
          int i, id,nthrds;
          double x;
          id = omp_get_thread_num();
          nthrds = omp_get_num_threads();
          if (id == 0)   nthreads = nthrds;
          for (i=id, sum[id]=0.0;i< num_steps; i=i+nthrds) {
               x = (i+0.5)*step;
               sum[id][0] += 4.0/(1.0+x*x);
          }
     }
     for(i=0, pi=0.0;i<nthreads;i++)pi += sum[i][0] * step;
}
```

threads	1st SPMD	1st SPMD padded
1	1.86	1.86
2	1.03	1.01
3	1.08	0.69
4	0.97	0.53

*Intel compiler (icpc) with default optimization level (O2) on Apple OS X 10.7.3 with a dual core (four HW thread) Intel® Core™ i5 processor at 1.7 Ghz and 4 Gbyte DDR3 memory at 1.333 Ghz.

Outline

- Introduction to OpenMP
- Creating Threads
- Synchronization
- Parallel Loops
- Data Environment
- Memory Model
- Irregular Parallelism and Tasks
- Recap

Synchronization

Synchronization is used to impose order constraints and to protect access to shared data

- High level synchronization included in the common core:
  - critical
  - barrier
- Other, more advanced, synchronization operations:
  - atomic
  - ordered
  - flush
  - locks (both simple and nested)

Synchronization: critical

- Mutual exclusion: Only one thread at a time can enter a **critical** region.

```
float  res;
#pragma omp parallel
{     float B;   int i, id, nthrds;
      id = omp_get_thread_num();
      nthrds = omp_get_num_threads();
      B =  big_SPMD_job(id, nthrds);
      #pragma omp critical
            res += consume (B);

}
```

Threads wait their turn – only one thread at a time calls consume()

Synchronization: barrier

- Barrier: a point in a program all threads much reach before any threads are allowed to proceed.
- It is a “stand alone” pragma meaning it is not associated with user code … it is an executable statement.

```
double Arr[8], Brr[8];          int numthrds;
omp_set_num_threads(8)
#pragma omp parallel
{   int id, nthrds;
     id = omp_get_thread_num();
     nthrds = omp_get_num_threads();
     if (id==0) numthrds = nthrds;
     Arr[id] = big_ugly_calc(id, nthrds);
#pragma omp barrier
     Brr[id] = really_big_and_ugly(id, nthrds, Arr);
}
```

Threads wait until all threads hit the barrier. Then they can go on.

Exercise

- In your first Pi program, you probably used an array to create space for each thread to store its partial sum.
- If array elements happen to share a cache line, this leads to false sharing.
  - Non-shared data in the same cache line so each update invalidates the cache line … in essence "sloshing independent data" back and forth between threads.

- Modify your "pi program" to avoid false sharing due to the partial sum array.

```
int omp_get_num_threads();
int omp_get_thread_num();
double omp_get_wtime();
omp_set_num_threads();
#pragma omp parallel
#pragma omp critical
```

PI Program with False Sharing

- Original Serial pi program with 100000000 steps ran in 1.83 seconds.

```
#include <omp.h>
static long num_steps = 100000;        double step;
#define NUM_THREADS 2
void main ()
{    int i, nthreads;  double pi, sum[NUM_THREADS];
     step = 1.0/(double) num_steps;
     omp_set_num_threads(NUM_THREADS);
    #pragma omp parallel
    {
           int i, id,nthrds;
           double x;
           id = omp_get_thread_num();
           nthrds = omp_get_num_threads();
           if (id == 0)   nthreads = nthrds;
           for (i=id, sum[id]=0.0;i< num_steps; i=i+nthrds) {
                x = (i+0.5)*step;
                sum[id] += 4.0/(1.0+x*x);
           }
    }
    for(i=0, pi=0.0;i<nthreads;i++)pi += sum[i] * step;
}
```

Recall that promoting sum to an array made the coding easy, but led to false sharing and poor performance.

threads	1st SPMD
1	1.86
2	1.03
3	1.08
4	0.97

*Intel compiler (icpc) with no optimization on Apple OS X 10.7.3 with a dual core (four HW thread) Intel® Core^{TM} i5 processor at 1.7 Ghz and 4 Gbyte DDR3 memory at 1.333 Ghz.

Example: Using a critical section to remove impact of false sharing

```
#include <omp.h>
static long num_steps = 100000;        double step;
#define NUM_THREADS 2
void main ()
{  int nthreads; double  pi=0.0;          step = 1.0/(double) num_steps;
   omp_set_num_threads(NUM_THREADS);
  #pragma omp parallel
  {
       int i, id, nthrds;    double x, sum;
       id = omp_get_thread_num();
       nthrds = omp_get_num_threads();
        if (id == 0)   nthreads = nthrds;
        for (i=id, sum=0.0;i< num_steps; i=i+nthrds) {
              x = (i+0.5)*step;
              sum += 4.0/(1.0+x*x);
         }
        #pragma omp critical
              pi += sum * step;
  }
}
```

Create a scalar local to each thread to accumulate partial sums.

No array, so no false sharing.

Sum goes "out of scope" beyond the parallel region … so you must sum it in here. Must protect summation into pi in a critical region so updates don't conflict

Results*: pi program critical section

- Original Serial pi program with 100000000 steps ran in 1.83 seconds.

```
#include <omp.h>
static long num_steps = 100000;        double step;
#define NUM_THREADS 2
void main ()
{  int nthreads; double  pi=0.0;         step = 1.0/(double) num_steps;
   omp_set_num_threads(NUM_THREADS);
  #pragma omp parallel
  {
      int i, id, nthrds;    double x, sum;
      id = omp_get_thread_num();
      nthrds = omp_get_num_threads();
       if (id == 0)   nthreads = nthrds;
       for (i=id, sum=0.0;i< num_steps; i=i+nthrds) {
             x = (i+0.5)*step;
             sum += 4.0/(1.0+x*x);
        }
       #pragma omp critical
              pi += sum * step;
  }
}
```

threads	1^{st} SPMD	1^{st} SPMD padded	SPMD critical
1	1.86	1.86	1.87
2	1.03	1.01	1.00
3	1.08	0.69	0.68
4	0.97	0.53	0.53

*Intel compiler (icpc) with no optimization on Apple OS X 10.7.3 with a dual core (four HW thread) Intel® Core™ i5 processor at 1.7 Ghz and 4 Gbyte DDR3 memory at 1.333 Ghz.

Example: Using a critical section to remove impact of false sharing

```
#include <omp.h>
static long num_steps = 100000;        double step;
#define NUM_THREADS 2
void main ()
{  int nthreads; double  pi=0.0;          step = 1.0/(double) num_steps;
   omp_set_num_threads(NUM_THREADS);
  #pragma omp parallel
  {
      int i, id, nthrds;    double x, sum;
      id = omp_get_thread_num();
      nthrds = omp_get_num_threads();
       if (id == 0)   nthreads = nthrds;
       for (i=id, sum=0.0;i< num_steps; i=i+nthrds) {
             x = (i+0.5)*step;
             #pragma omp critical
                 sum += 4.0/(1.0+x*x);
       }
    }
}
```

What would happen if you put the critical section inside the loop?

Outline

- Introduction to OpenMP
- Creating Threads
- Synchronization
- Parallel Loops
- Data Environment
- Memory Model
- Irregular Parallelism and Tasks
- OpenMP: Beyond the CPU
- Recap

The Loop Worksharing Construct

- The loop worksharing construct splits up loop iterations among the threads in a team

```
#pragma omp parallel
{
#pragma omp for
        for (I=0;I<N;I++){
                NEAT_STUFF(I);
        }
}
```

Loop construct name:

- C/C++: for
- Fortran: do

The loop control index I is made "private" to each thread by default.

Threads wait here until all threads are finished with the parallel loop before any proceed past the end of the loop

Loop Worksharing Construct
A motivating example

Sequential code

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

OpenMP parallel region

(SPMD Pattern)

```
#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];}
}
```

OpenMP parallel region and a worksharing for construct

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

Combined Parallel/Worksharing Construct

- OpenMP shortcut: Put the "parallel" and the worksharing directive on the same line

```
double  res[MAX];  int i;
#pragma omp parallel
{
   #pragma omp for
   for (i=0;i< MAX; i++) {
        res[i] = huge();
   }
}
```

```
double  res[MAX];  int i;
#pragma omp parallel for
   for (i=0;i< MAX; i++) {
        res[i] = huge();
   }
```

These are equivalent

Working with loops

- Basic approach
  - Find compute intensive loops
  - Make the loop iterations independent ... So they can safely execute in any order without loop-carried dependencies
  - Place the appropriate OpenMP directive and test

```
int i, j, A[MAX];
 j = 5;
 for (i=0;i< MAX; i++) {
    j +=2;
    A[i] = big(j);
 }
```

Note: loop index "i" is private by default

```
int i,  A[MAX];
#pragma omp parallel for
 for (i=0;i< MAX; i++) {
    int j = 5 + 2*(i+1);
    A[i] = big(j);
 }
```

Remove loop carried dependence

Reduction

- How do we handle this case?

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

- We are combining values into a single accumulation variable (ave) … there is a true dependence between loop iterations that can't be trivially removed.

- This is a very common situation … it is called a "reduction".

- Support for reduction operations is included in most parallel programming environments.

Reduction

- 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. 0 for “+”).
  - Updates occur on the local copy.
  - Local copies are reduced into a single value and combined with the original global value.

- The variables in “list” must be shared in the enclosing parallel region.

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

OpenMP: Reduction operands/initial-values

- Many different associative operands can be used with reduction:
- Initial values are the ones that make sense mathematically.

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

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

Fortran Only	
Operator	Initial value
.AND.	.true.
.OR.	.false.
.NEQV.	.false.
.IEOR.	0
.IOR.	0
.IAND.	All bits on
.EQV.	.true.

OpenMP includes user defined reductions and array-sections as reduction variables (we just don't cover those topics here)

Exercise: PI with loops

- Go back to the serial pi program and parallelize it with a loop construct
- Your goal is to minimize the number of changes made to the serial program.

```
#pragma omp parallel
#pragma omp for
#pragma omp parallel for
#pragma omp for reduction(op:list)
#pragma omp critical
int omp_get_num_threads();
int omp_get_thread_num();
double omp_get_wtime();
```

Example: PI with a loop and a reduction

```
#include <omp.h>
static long num_steps = 100000;        double step;
void main ()
{   int i;          double x, pi, sum = 0.0;
     step = 1.0/(double) num_steps;
     #pragma omp parallel
     {
         double x;
        #pragma omp for reduction(+:sum)
           for (i=0;i< num_steps; i++){
                  x = (i+0.5)*step;
                  sum = sum + 4.0/(1.0+x*x);
           }
      }
         pi = step * sum;
}
```

Create a team of threads … without a parallel construct, you'll never have more than one thread

Create a scalar local to each thread to hold value of x at the center of each interval

Break up loop iterations and assign them to threads … setting up a reduction into sum. Note … the loop index is local to a thread by default.

Example: PI with a loop and a reduction

```
#include <omp.h>
static long num_steps = 100000;         double step;
void main ()
{
    double pi, sum = 0.0;
     step = 1.0/(double) num_steps;

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

Using modern C style, we put declarations close to where they are used … which lets me use the parallel for construct.

Results*: PI with a loop and a reduction

- Original Serial pi program with 100000000 steps ran in 1.83 seconds.

Example: Pi with a

```
#include <omp.h>
static long num_steps = 1000
void main ()
{   int i;          double x, pi, su
     step = 1.0/(double) num_s
     #pragma omp parallel
     {
        double x;
       #pragma omp for reduction(+:sum)
          for (i=0;i< num_steps; i++){
                x = (i+0.5)*step;
                sum = sum + 4.0/(1.0+x*x);
          }
     }
        pi = step * sum;
}
```

threads	1st SPMD	1st SPMD padded	SPMD critical	PI Loop
1	1.86	1.86	1.87	1.91
2	1.03	1.01	1.00	1.02
3	1.08	0.69	0.68	0.80
4	0.97	0.53	0.53	0.68

*Intel compiler (icpc) with no optimization on Apple OS X 10.7.3 with a dual core (four HW thread) Intel® Core^{TM} i5 processor at 1.7 Ghz and 4 Gbyte DDR3 memory at 1.333 Ghz.

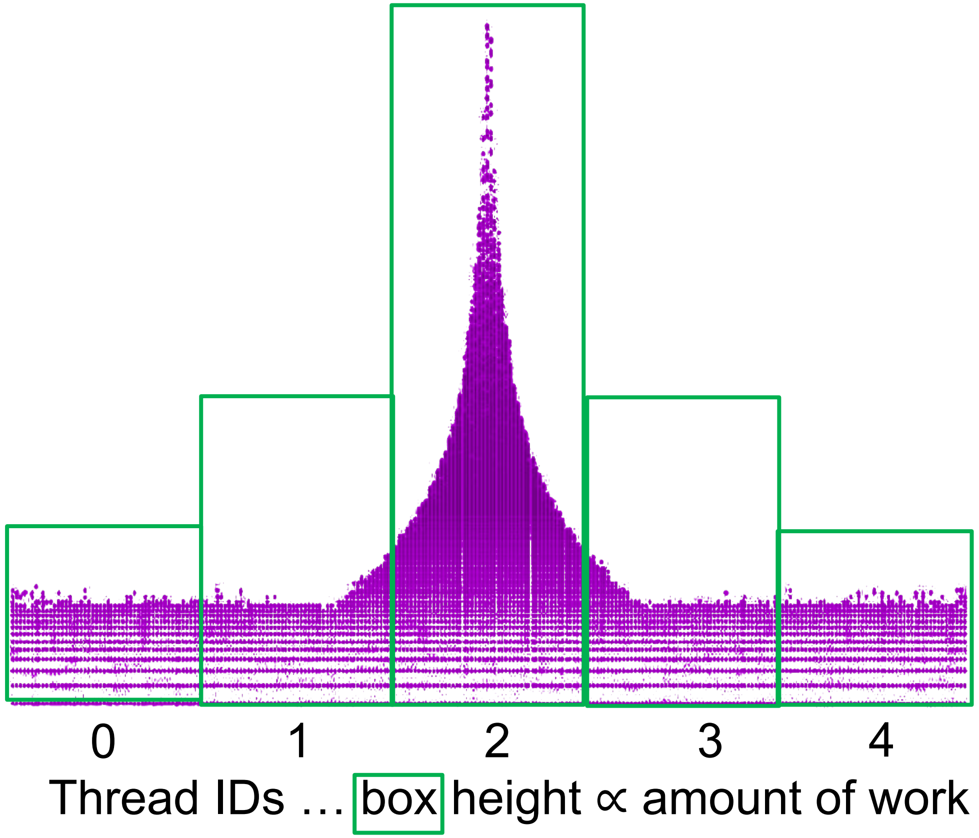
…. Let’s pause a moment and consider one of the fundamental issues EVERY parallel programmer must grapple with

Load Balancing

- A parallel job isn't done until the last thread is finished

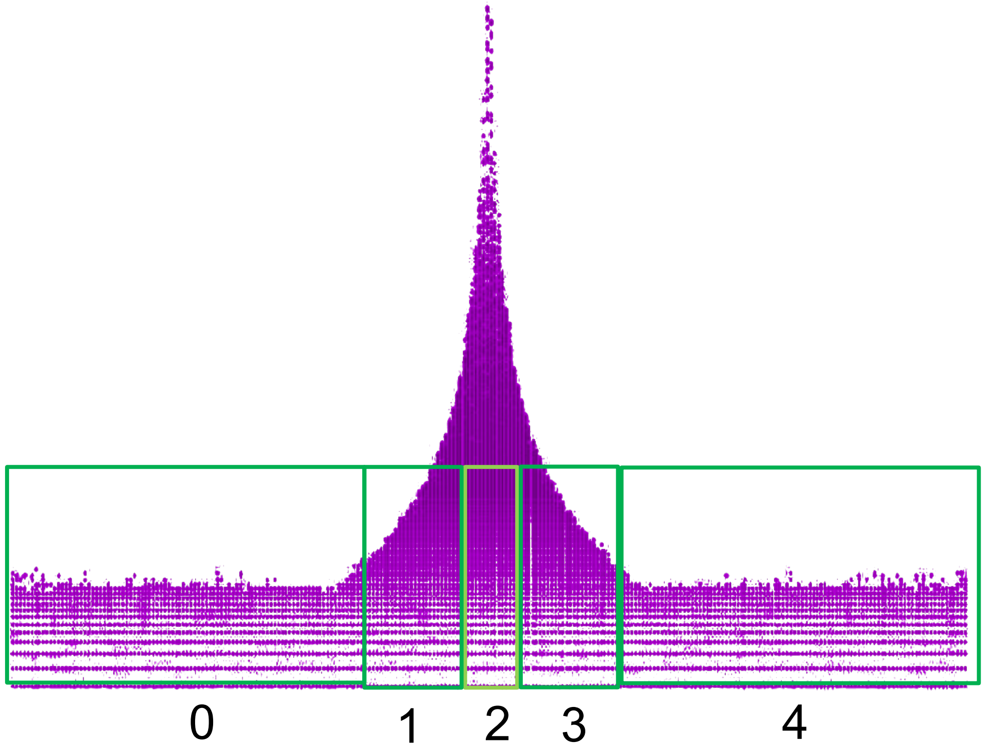
- Example: Partition a problem into equal sized chunks but for work that is unevenly distributed spatially.
  - Thread 2 has MUCH more work. The uneven distribution of work will limit performance.

- A key part of parallel programming is to design how you partition the work between threads so every thread has about the same amount of work. This topic is referred to as *Load Balancing*.



Load Balancing

- A parallel job isn't done until the last thread is finished
- The work in our problem is unevenly distributed spatially.
- A key part of parallel programming is to design how you partition the work between threads so every thread has about the same amount of work.
- This topic is referred to as *Load Balancing*.
- In this case we adjusted the size of each chunk to equalize the work assigned to each thread.
  - Getting the right sized chunks for a variable partitioning (as done here) can be really difficult.

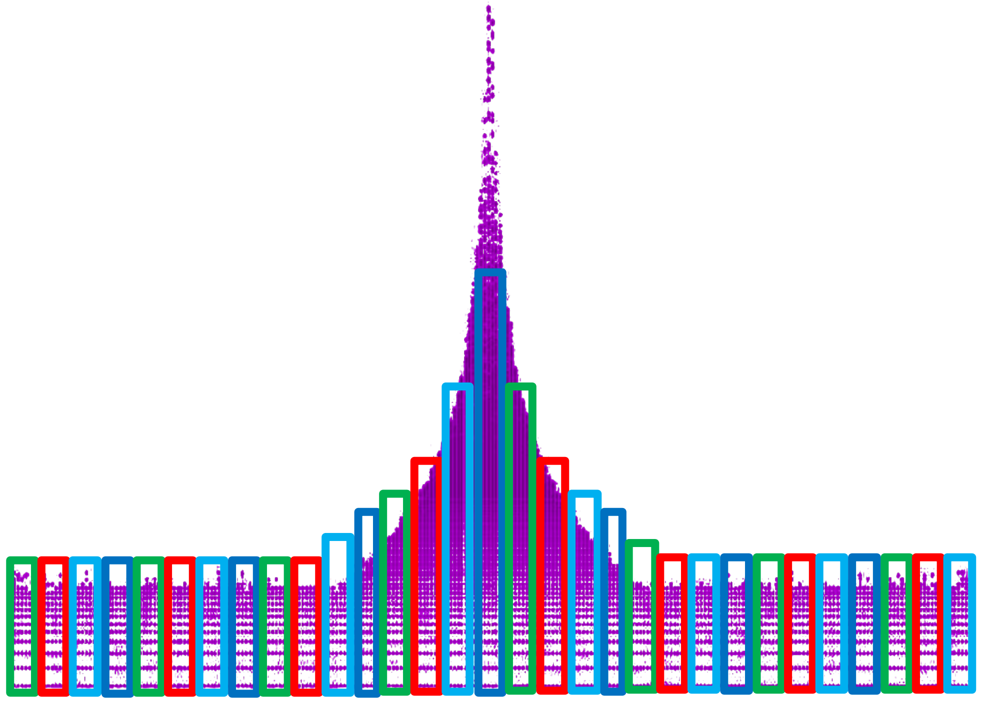


Thread IDs … box height ∝ amount of work

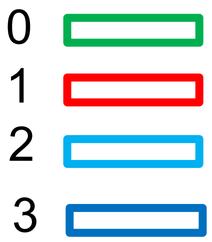
Load Balancing

- A parallel job isn't done until the last thread is finished

- An easier path to *Load Balancing*.
  - Over-decompose the problem into small, fine-grained chunks
  - Spread the chunks out among the threads (in this case using a cyclic distribution)
  - The work is spread out and statistically, you are likely to get a good distribution of work



Colors mapped to 4 different Threads

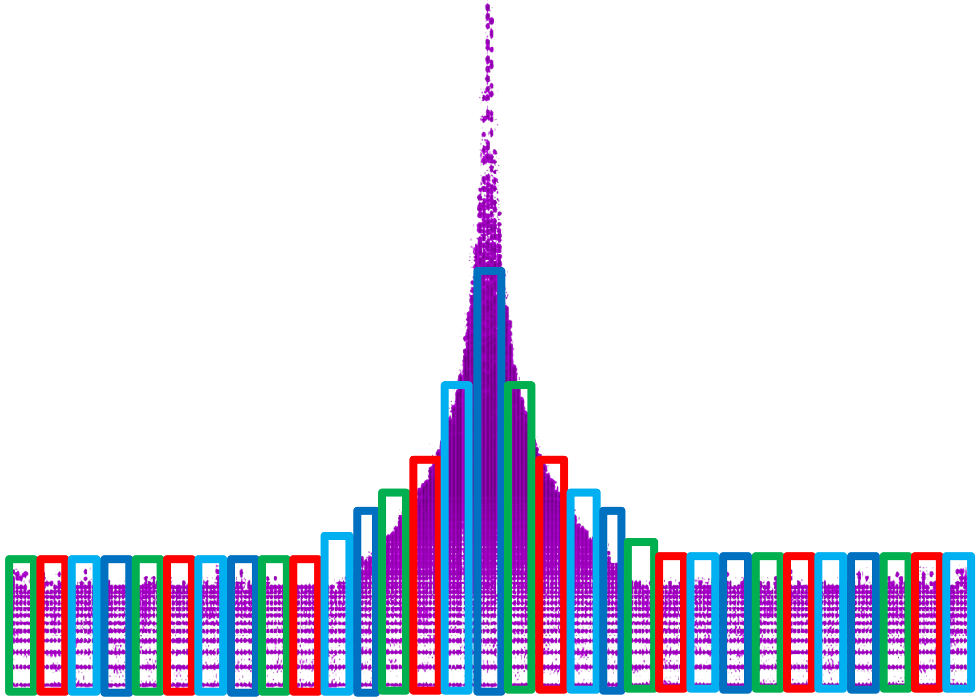


Load Balancing

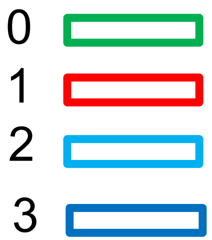
- A parallel job isn't done until the last thread is finished

- An easier path to *Load Balancing*.
  - Over-decompose the problem into small, fine-grained chunks
  - Spread the chunks out among the threads (in this case using a cyclic distribution)
  - The work is spread out and statistically, you are likely to get a good distribution of work

- Vocabulary review
  - **Load Balancing** … giving each thread work sized so all threads take the same amount of time
  - **Partitioning** or **decomposition** … breaking up the problem domain into partitions (or chunks) and assigning different partitions to different threads.
  - **Granularity** … the size of the block of work. Find grained (small chunks) vs coarse grained (large chunks)
  - **Over-decomposition** … when you decompose your problem into partitions such that there are many more partitions than threads to do the work



Colors mapped to 4 different Threads



Loop Worksharing Constructs: The schedule clause

- The schedule clause affects how loop iterations are mapped onto threads
  - **schedule(static [,chunk])**
    - Deal-out blocks of iterations of size “chunk” to each thread.
  - **schedule(dynamic[,chunk])**
    - Each thread grabs “chunk” iterations off a queue until all iterations have been handled.
- Example:
  - #pragma omp for schedule(dynamic, 10)

Schedule Clause	When To Use
STATIC	**Pre-determined and predictable by the programmer**
DYNAMIC	**Unpredictable, highly variable work per iteration**

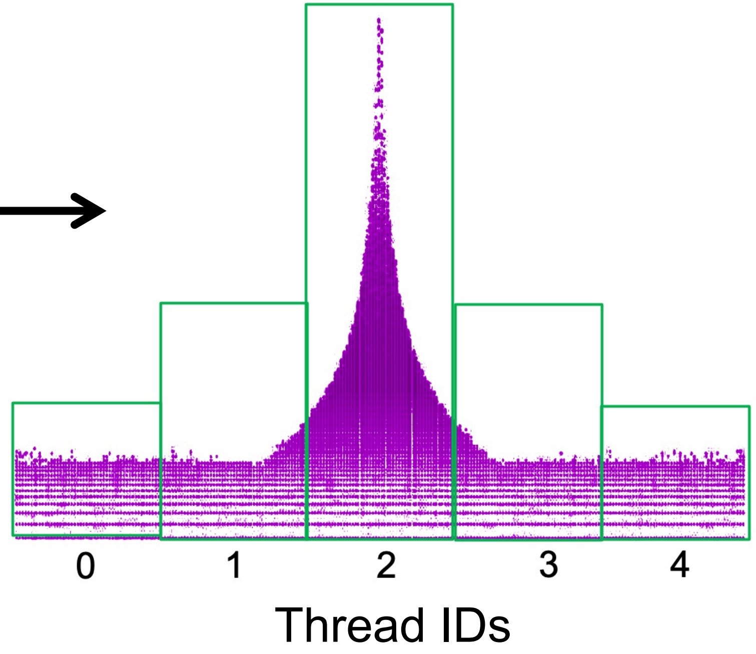
Least work at runtime : scheduling done at compile-time

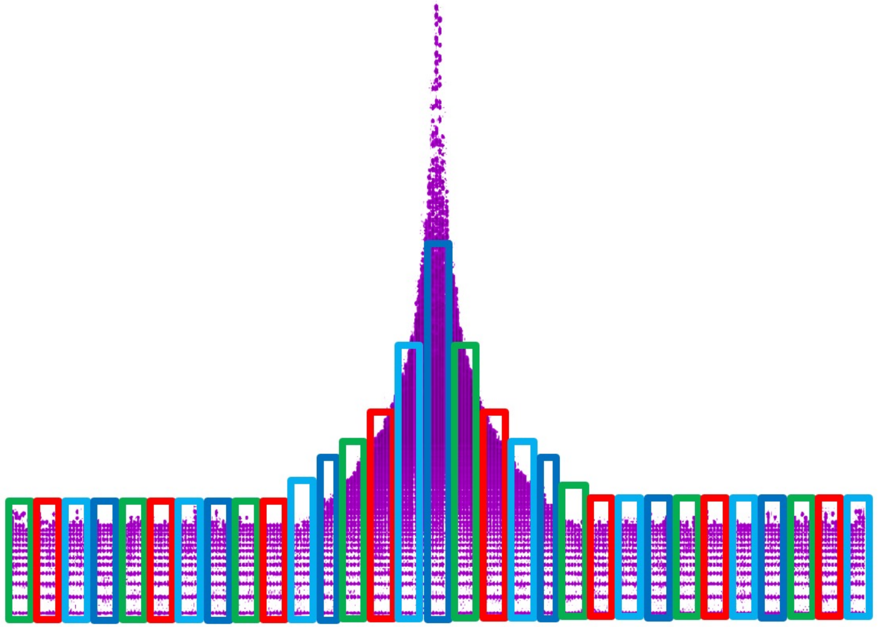
Most work at runtime : complex scheduling logic used at run-time

Loop Worksharing Constructs: The schedule clause

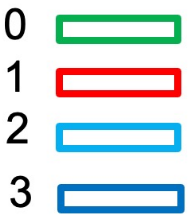
- The schedule clause … most common cases:

#pragma omp parallel for schedule (static)





Colors mapped to 4 different Threads



Int small = 8; // loop iterations, i.e., width of boxes in the figure

#pragma omp parallel for schedule (static, small)

We'll finish with loops by looking one more time at synchronization overhead

The nowait clause

- Barriers are really expensive. You need to understand when they are implied and how to skip them when it's safe to do so.

```
double A[big], B[big], C[big];

#pragma omp parallel
{
        int id=omp_get_thread_num();
        A[id] = big_calc1(id);
#pragma omp barrier
#pragma omp for
        for(i=0;i<N;i++){C[i]=big_calc3(i,A);}
#pragma omp for nowait
        for(i=0;i<N;i++){ B[i]=big_calc2(C,  i); }
        A[id] = big_calc4(id);
}
```

implicit barrier at the end of a for worksharing construct

no implicit barrier due to nowait

implicit barrier at the end of a parallel region

Outline

- Introduction to OpenMP
- Creating Threads
- Synchronization
- Parallel Loops
- Data Environment
- Memory Model
- Irregular Parallelism and Tasks
- Recap

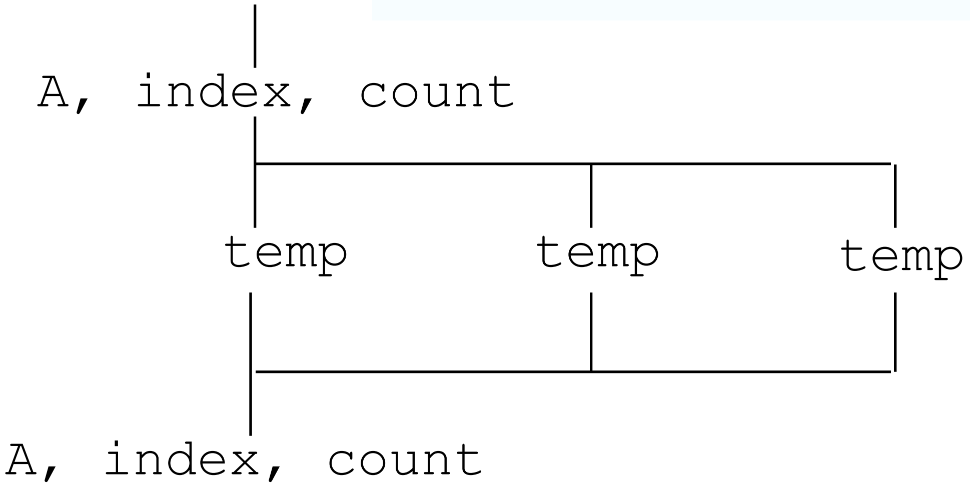
Data Environment: Default storage attributes

- Shared memory programming model:
  - Most variables are shared by default
- Global variables are SHARED among threads
  - Fortran: COMMON blocks, SAVE variables, MODULE variables
  - C: File scope variables, static
  - Both: dynamically allocated memory (ALLOCATE, malloc, new)
- But not everything is shared...
  - Stack variables in subprograms(Fortran) or functions(C) called from parallel regions are PRIVATE
  - Automatic variables within a statement block are PRIVATE.

Data Sharing: Examples

```
double A[10];
int main() {
int index[10];
#pragma omp parallel
      work(index);
printf("%d\n", index[0]);
}
```

```
extern double A[10];
void work(int *index) {
  double temp[10];
  static int count;
  ...
}
```



A, index and count are shared by all threads.

temp is local to each thread

Data Sharing: Changing storage attributes

- One can selectively change storage attributes for constructs using the following clauses (note: ***list*** is a comma-separated list of variables)
  - **shared(list)**
  - **private(list)**
  - **firstprivate(list**)
- These can be used on **parallel** and **for** constructs … other than **shared** which can only be used on a **parallel** construct

- Force the programmer to explicitly define storage attributes
  - **default (none)**

default() can only be used on parallel constructs

Data Sharing: Private clause

- private(var) creates a new local copy of var for each thread.

```
int N = 1000;
extern void init_arrays(int N, double *A, double *B, double *C);


void example () {
    int i, j;
    double A[N][N], B[N][N], C[N][N];
    init_arrays(N, *A, *B, *C);


    #pragma omp parallel for private(j)
    for (i = 0; i < 1000; i++)
        for( j = 0; j<1000; j++)
            C[i][j] = A[i][j] + B[i][j];
}
```

OpenMP makes the loop control index on the parallel loop (i) private by default … but not for the second loop (j)

Data Sharing: Private clause

- private(var) creates a new local copy of var for each thread.
  - The value of the private copies is uninitialized
  - The value of the original variable is unchanged after the region

```
incr = 0;
#pragma omp parallel for private(incr)
for (i = 0; i <= MAX; i++) {
        if ((i%2)==0) incr++;
        A[i] = incr;
}
printf(“ incr= %d\n”, incr);
```

incr was not initialized

incr is 0 here

When you need to refer to the variable incr that exists prior to the construct, we call it the **original variable**.

Firstprivate clause

- Variables initialized from a shared variable
- C++ objects are copy-constructed

```
incr = 0;
#pragma omp parallel for firstprivate(incr)
for (i = 0; i <= MAX; i++) {
        if ((i%2)==0) incr++;
        A[i] = incr;
}
```

Each thread gets its own copy of incr with an initial value of 0

Data sharing:
A data environment test

- Consider this example of PRIVATE and FIRSTPRIVATE

```
variables:  A = 1,B = 1, C = 1
#pragma omp parallel private(B)  firstprivate(C)
```

- Are A,B,C private to each thread or shared inside the parallel region?
- What are their initial values inside and values after the parallel region?

Inside this parallel region ...

- “A” is shared by all threads; equals 1
- “B” and “C” are private to each thread.
  - B’s initial value is undefined
  - C’s initial value equals 1

Following the parallel region ...

- B and C revert to their original values of 1
- A is either 1 or the value it was set to inside the parallel region

Exercise: Mandelbrot set area

- The supplied program (mandel.c) computes the area of a Mandelbrot set.
- The program has been parallelized with OpenMP, but we were lazy and didn't do it right.
- Find and fix the errors.
- Once you have a working version, try to optimize the program.

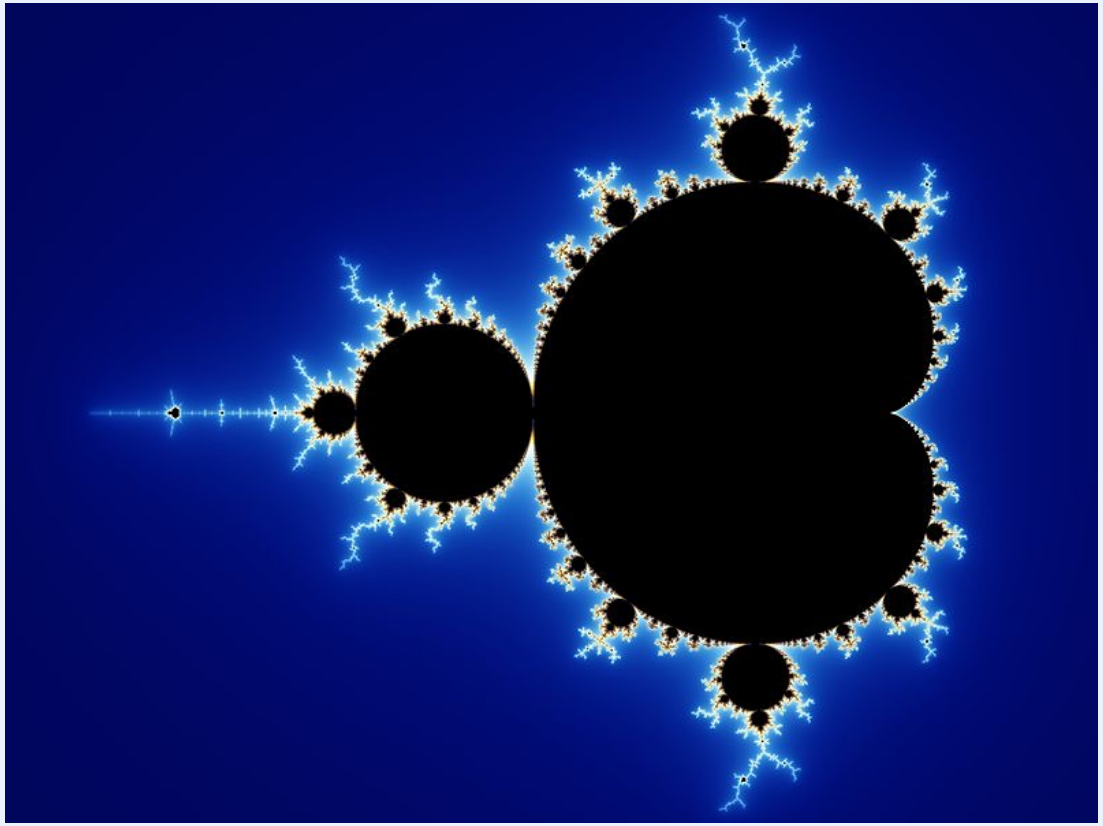


Image Source: Created by Wolfgang Beyer with the program Ultra Fractal 3. - Own work, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=321973

The Mandelbrot set … The points, c, for which the following iterative map converges

$$z_{n+1} = z_n^2 + c$$

With z_n and c as complex numbers and $z_0 = 0$.

This exercise come from Mark Bull of EPCC (at University of Edinburgh)

The Mandelbrot Set Area Program (original code)

```
#include <omp.h>
# define NPOINTS 1000
# define MXITR 1000
void testpoint(double, double);
Int numoutside = 0;
int main(){
   int i, j;
   int num=0;
   double C_real, C_imag;
   double area, error, eps  = 1.0e-5;
#pragma omp parallel for private(eps)
   for (i=0; i<NPOINTS; i++) {
     for (j=0; j<NPOINTS; j++) {
       C_real = -2.0+2.5*(double)(i)/(double)(NPOINTS)+eps;
       C_imag = 1.125*(double)(j)/(double)(NPOINTS)+eps;
       testpoint(C_real, C_imag);
     }
   }
area=2.0*2.5*1.125*(double)(NPOINTS*NPOINTS-
numoutside)/(double)(NPOINTS*NPOINTS);
   error=area/(double)NPOINTS;
}
```

```
void testpoint(double C_real, double C_imag){
       double zr, zi;
       int iter;
       double temp;

       zr=C_real;     zi=C_imag;
       int numoutside = 0;
       for (iter=0; iter<MXITR; iter++){
         temp = (zr*zr)-(zi*zi)+C_real;
         zi = zr*zi*2+C_imag;
         zr = temp;
         if ((zr*zr+zi*zi)>4.0) {
                  numoutside++;
         }
       }
       return 0;
}
```

The Mandelbrot Set Area Program

```
#include <omp.h>
# define NPOINTS 1000
# define MXITR 1000
void testpoint(double, double);
Int numoutside = 0;
int main(){
   int i, j;
   int num=0;
   double C_real, C_imag;
   double area, error, eps  = 1.0e-5;
#pragma omp parallel for private(j, C_real, C_imag)
   for (i=0; i<NPOINTS; i++) {
     for (j=0; j<NPOINTS; j++) {
       C_real = -2.0+2.5*(double)(i)/(double)(NPOINTS)+eps;
       C_imag = 1.125*(double)(j)/(double)(NPOINTS)+eps;
       testpoint(C_real, C_imag);
     }
   }
area=2.0*2.5*1.125*(double)(NPOINTS*NPOINTS-
numoutside)/(double)(NPOINTS*NPOINTS);
   error=area/(double)NPOINTS;
}
```

```
void testpoint(double C_real, double C_imag){
       double zr, zi;
       int iter;
       double temp;

       zr=C_real;     zi=C_imag;
       int numoutside = 0;
       for (iter=0; iter<MXITR; iter++){
         temp = (zr*zr)-(zi*zi)+C_real;
         zi = zr*zi*2+C_imag;
         zr = temp;
         if ((zr*zr+zi*zi)>4.0) {
           #pragma omp critical
                 numoutside++;
         }
       }
       return 0;
}
```

- eps was not initialized
- Data race on j, C_real, and C_imag
- Protect updates of numoutside

Data Sharing: Default clause

- **default(none)**: Forces you to define the storage attributes for variables that appear inside the static extent of the construct … if you fail the compiler will complain. Good programming practice!
- You can put the default clause on parallel and parallel + workshare constructs.

The static extent is the code in the compilation unit that contains the construct.

```
#include <omp.h>
int main()
{
    int i, j=5;      double x=0.0, y=42.0;
    #pragma omp parallel for default(none) reduction(*:x)
    for (i=0;i<N;i++){
       for(j=0; j<3; j++)
           x+= foobar(i, j, y);
    }
    printf(" x is %f\n",(float)x);
}
```

The compiler would complain about j and y, which is important since you don't want j to be shared

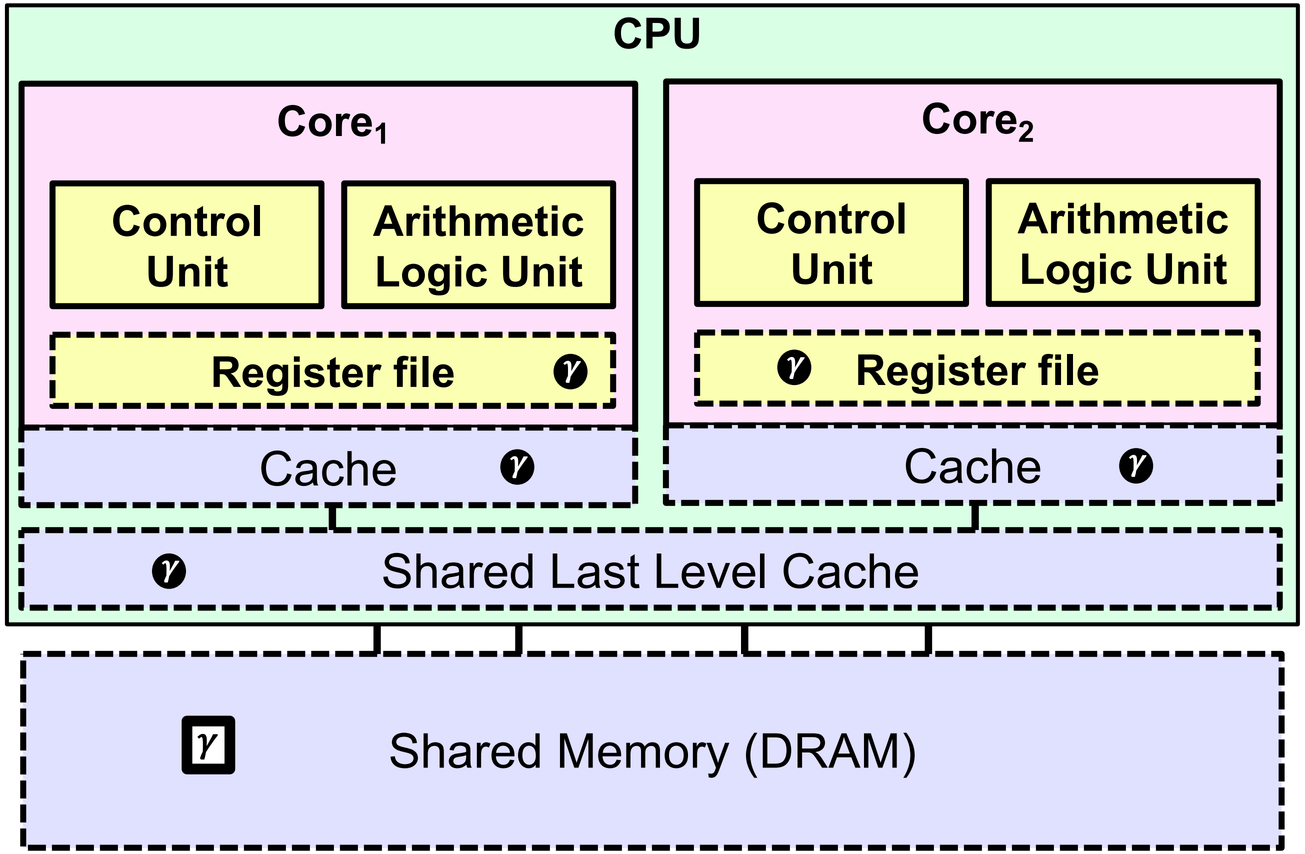
The full OpenMP specification has other versions of the default clause, but they are not used very often so we skip them in the common core

Outline

- Introduction to OpenMP
- Creating Threads
- Synchronization
- Parallel Loops
- Data Environment
- Memory Model
- Irregular Parallelism and Tasks
- OpenMP: Beyond the CPU
- Recap

Memory Models …

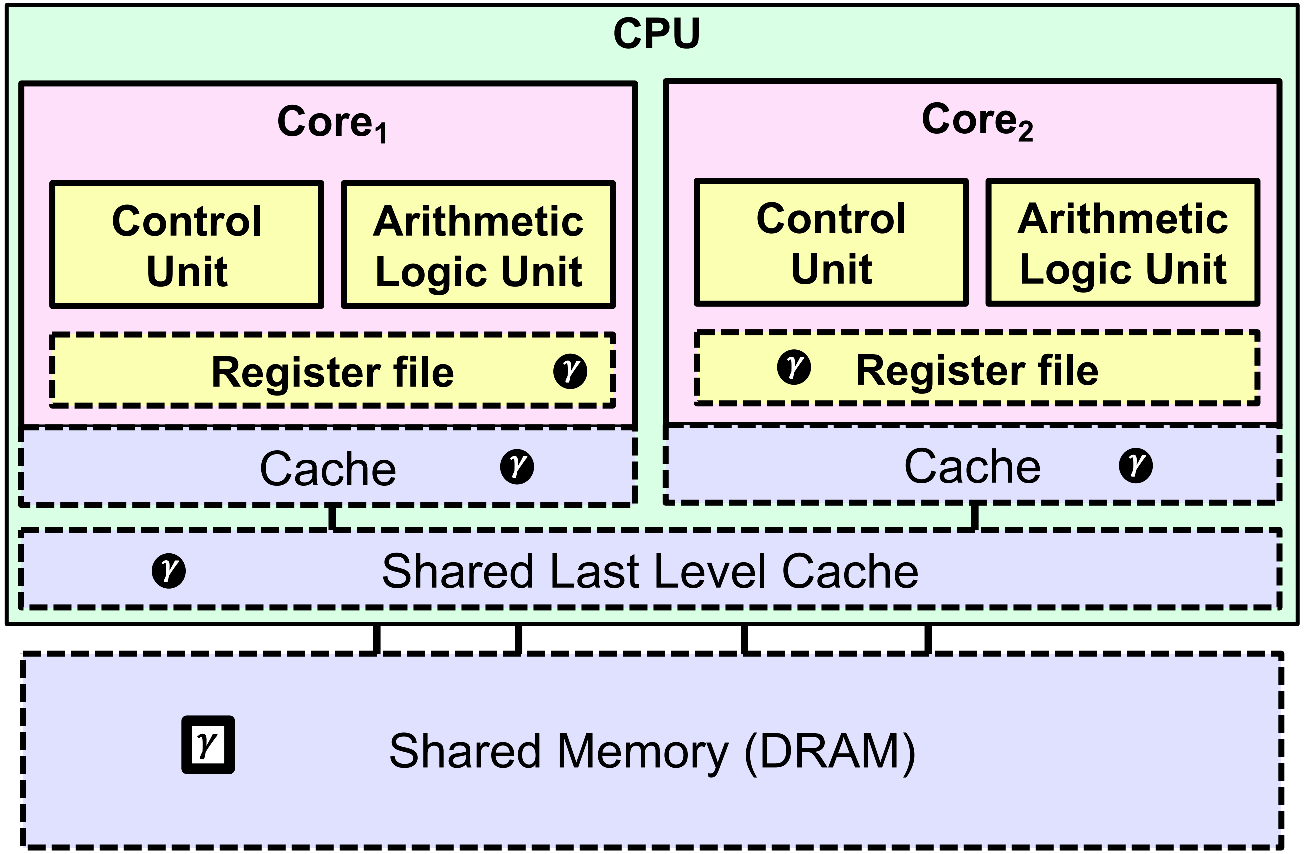
- A shared address space is a region of memory visible to the team of threads … multiple threads can read and write variables in the shared address space.
- Multiple copies of a variable (such as γ) may be present in memory, at various levels of cache, or in registers and they may ALL have different values.



- Which value of γ is the one a thread should see at any point in a computation?

Memory Models …

- A shared address space is a region of memory visible to the team of threads … multiple threads can read and write variables in the shared address space.
- Multiple copies of a variable (such as γ) may be present in memory, at various levels of cache, or in registers and they may ALL have different values.



A memory consistency model (or "**memory model**" for short) provides the rules needed to answer this question.

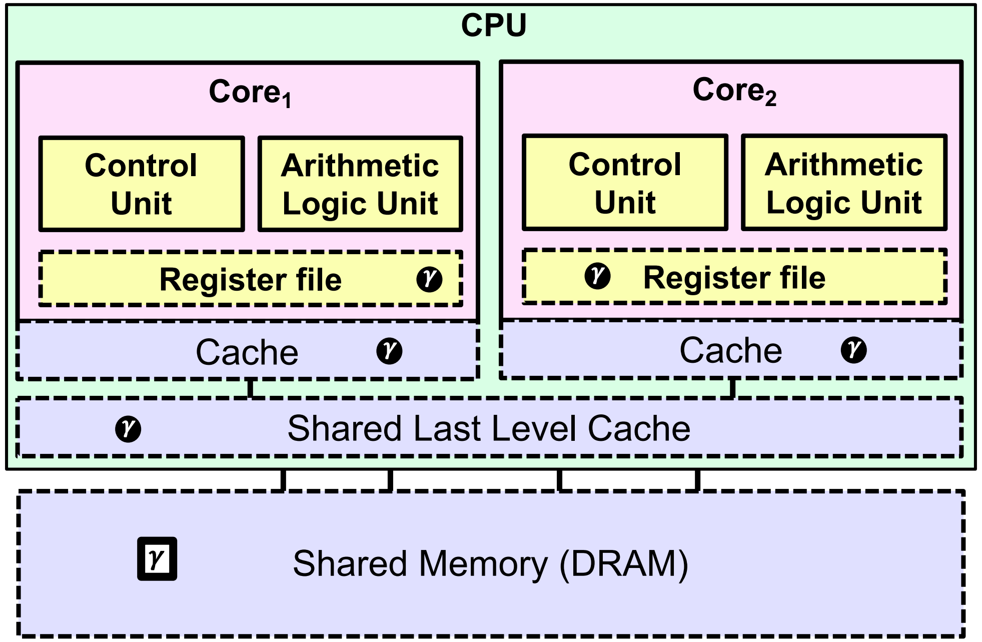
- Which value of γ is the one a thread should see at any point in a computation?

Memory Models …

- The fundamental issue is how do the values of variables across the memory hierarchy interact with the statements executed by two or more threads?
- Two options:

1. **Sequential Consistency**

- Threads execute and the associated loads/stores appear in some order defined by the semantically allowed interleaving of program statements.
- **All threads see the same interleaved order of loads and stores**



2. **Relaxed Consistency**

- Threads execute and the associated loads/stores appear in some order defined by the semantically allowed interleaving of program statements.
- **Threads may see different orders of loads and stores**

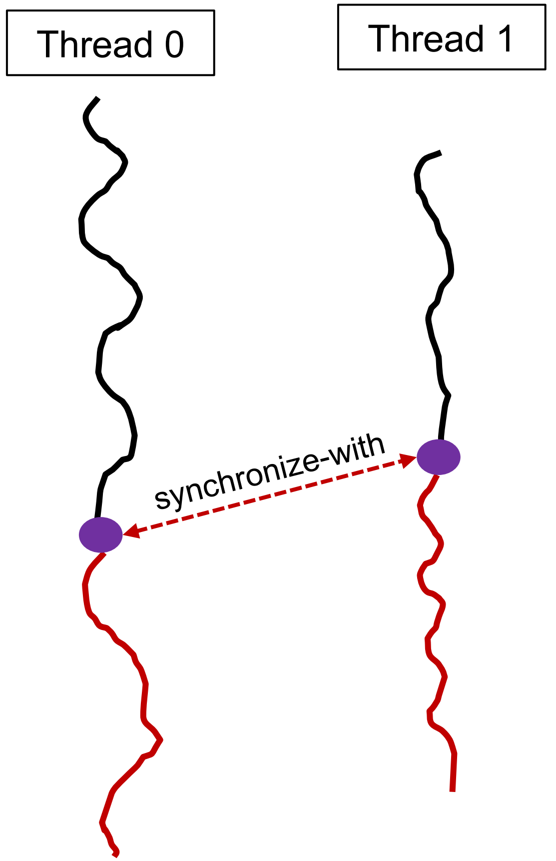
Most (if not all) multithreading programming models assume **relaxed consistency**. Maintaining sequential consistency across the full program-execution adds too much synchronization overhead.

Memory Models: *Happens-before* and *synchronized-with* relations

- Single thread execution:
  - Program order ... Loads and stores appear to occur in the order defined by the program's semantics. If you can't observe it, however, compilers can reorder instructions to maximize performance.

- Multithreaded execution … concurrency in action
  - The compiler doesn't understand instruction-ordering across threads … loads/stores to shared memory across threads can expose ambiguous orders of loads and stores
  - Instructions between threads are unordered except when specific ordering constraints are imposed, i.e., **synchronization**.
  - Synchronization lets us force that some instructions **happens-before** other instructions

- Two parts to synchronization:
  - A **synchronize-with** relationship exists at statements in 2 or more threads at which memory order constraints can be established.
  - **Memory order**: defines the view of loads/stores on either side of a synchronized-with operations.



Memory orders defined at the synchronize-with statements define happens-before relationships between Loads/stores in the black/red sections of threads 0 and 1.

Enforcing Memory Orders: the Flush Operation

- Flush defines a sequence point at which a thread is guaranteed to see a consistent view of memory*
  - Previous read/writes by this thread have completed and are visible to other threads
  - No subsequent read/writes by this thread have occurred

```
double A;

A = compute();

#pragma omp flush(A)

   // flush to memory to make sure other
   //  threads can see the updated value of A
```

- A flush on its own, however, is not enough. It only controls memory visibility from the perspective of the thread calling the flush.

- You must pair it with an operation to create a synchronized-with relation between threads.

- We've worked with collective synchronization operations that apply across the full team of threads (**critical** and **barrier**). They both imply the flush so you should *NEVER* need to call flush explicitly

- You can build custom synchronization protocols applying to any combination of pairs of threads … but that is seriously advanced multithreaded programming and should be avoided if at all possible

* This applies to the set of shared variables visible to a thread at the point the flush is encountered. We call this "**the flush set**"

Outline

- Introduction to OpenMP
- Creating Threads
- Synchronization
- Parallel Loops
- Data Environment
- Memory Model
- Irregular Parallelism and Tasks
- OpenMP: Beyond the CPU
- Recap

Irregular Parallelism

- Let's call a problem "irregular" when one or both of the following hold:
  - Data Structures are sparse or involve indirect memory references
  - Control structures are not basic for-loops

- Example: Traversing Linked lists:

```
p = listhead ;
while (p) {
  process(p);
  p=p->next;
}
```

- Using what we've learned so far, traversing a linked list in parallel using OpenMP is difficult.

Exercise: Traversing linked lists

- Consider the program linked.c
  - Traverses a linked list computing a sequence of Fibonacci numbers at each node.
- Parallelize this program selecting from the following list of constructs:

```
#pragma omp parallel
#pragma omp for
#pragma omp parallel for
#pragma omp for reduction(op:list)
#pragma omp critical
int omp_get_num_threads();
int omp_get_thread_num();
double omp_get_wtime();
schedule(static[,chunk]) or schedule(dynamic[,chunk])
private(), firstprivate(), default(none)
```

- Hint: Just worry about the while loop that is timed inside main(). You don't need to make any changes to the "list functions"

Linked Lists with OpenMP: My solution

- See the file solutions/linked_notasks.c

```
while (p != NULL) {
     p = p->next;
      count++;
}
struct node *parr = (struct node*) malloc(count*sizeof(struct node));
p = head;
for(i=0; i<count; i++) {
      parr[i] = p;
      p = p->next;
   }
#pragma omp parallel
{
     #pragma omp for schedule(static,1)
     for(i=0; i<count; i++)
        processwork(parr[i]);
}
```

Count number of items in the linked list

Copy pointer to each node into an array

Process nodes in parallel with a for loop

Linked Lists with OpenMP (without tasks)

- See the file solutions/linked_notasks.c

```
while (p != NULL) {
    p = p->next;
     count++;
}
struct node *parr = (struct node*) malloc(count*sizeof(struct node));
p = head;
for(i=0; i<count; i++) {
      parr[i] = p;
      p = p->next;
   }
#pragma omp parallel
{
    #pragma omp for schedule(static,1)
    for(i=0; i<count; i++)
       processwork(parr[i]);
}
```

Count number of items in the linked list

Copy pointer to each node into an array

Process nodes in parallel with a for loop

Number of threads	Schedule	
	Default	Static,1
1	48 seconds	45 seconds
2	39 seconds	28 seconds

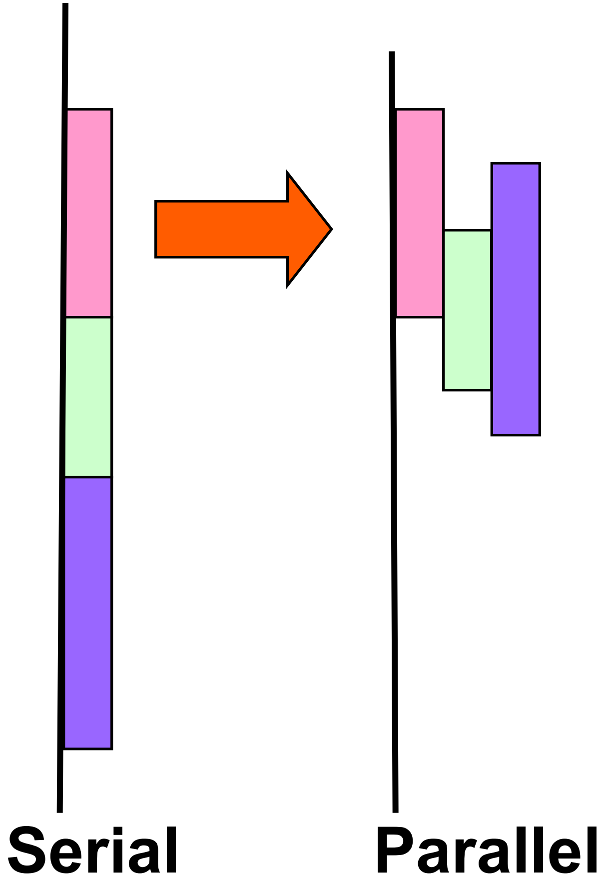
With so much code to add and three passes through the data, this is really ugly.

There has got to be a better way to do this

Results on an Intel dual core 1.83 GHz CPU, Intel IA-32 compiler 10.1 build 2

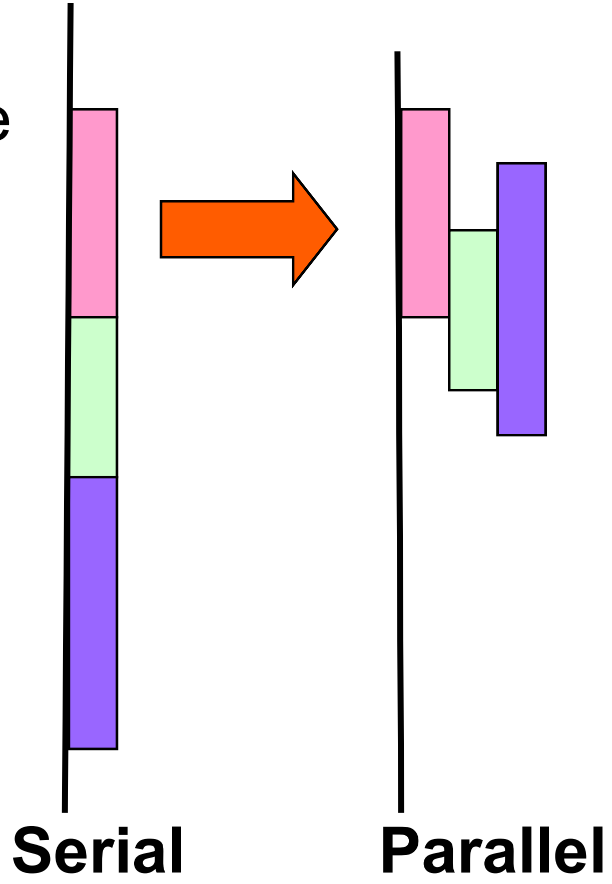
What are Tasks?

- Tasks are independent units of work
- Tasks are composed of:
  - code to execute
  - data to compute with
- Threads are assigned to perform the work of each task.
  - The thread that encounters the task construct may execute the task immediately.
  - The threads may defer execution until later



What are Tasks?

- The task construct includes a structured block of code
- Inside a parallel region, a thread encountering a task construct will package up the code block and its data for execution
- Tasks can be nested: i.e., a task may itself generate tasks.



A common Pattern is to have one thread create the tasks while the other threads wait at a barrier and execute the tasks

Single Worksharing Construct

- The **single** construct denotes a block of code that is executed by only one thread (not necessarily the primary* thread).
- A barrier is implied at the end of the single block (can remove the barrier with a *nowait* clause).

```
#pragma omp parallel
{
        do_many_things();
    #pragma omp single
        {    exchange_boundaries();   }
        do_many_other_things();

}
```

*This used to be called the "master thread". The term "master" has been deprecated in OpenMP 5.1 and replaced with the term "primary".

Task Directive

#pragma omp task *[clauses]*

structured-block

```
#pragma omp parallel
{
  #pragma omp single
   {
     #pragma omp task
        fred();
     #pragma omp task
        daisy();
     #pragma omp task
        billy();
   }
}
```

Create some threads

One Thread packages tasks

Tasks executed by some thread in some order

All tasks complete before this barrier is released

Exercise: Simple tasks

- Write a program using tasks that will "randomly" generate one of two strings:
  - "I think " "race" "car" "s are fun"
  - "I think " "car" "race" "s are fun"

- Hint: use tasks to print the indeterminate part of the output (i.e. the "race" or "car" parts).

- This is called a "Race Condition". It occurs when the result of a program depends on how the OS schedules the threads.

- NOTE: A "data race" is when threads "race to update a shared variable". They produce race conditions. Programs containing data races are undefined (in OpenMP but also ANSI standards C++'11 and beyond).

```
#pragma omp parallel
#pragma omp task
#pragma omp single
```

This exercise comes from Ruud van der Pas of Oracle

Racey Cars: Solution

```
#include <stdio.h>
#include <omp.h>
int main()
{  printf("I think");
   #pragma omp parallel
   {
      #pragma omp single
      {
         #pragma omp task
            printf(" car");
         #pragma omp task
            printf(" race");
      }
   }
   printf("s");
   printf(" are fun!\n");
}
```

Data Scoping with Tasks

- Variables can be shared, private or firstprivate with respect to task
- These concepts are a little bit different compared with threads:
  - If a variable is shared on a task construct, the references to it inside the construct are to the storage with that name at the point where the task was encountered
  - If a variable is private on a task construct, the references to it inside the construct are to new uninitialized storage that is created when the task is executed
  - If a variable is firstprivate on a construct, the references to it inside the construct are to new storage that is created and initialized with the value of the existing storage of that name when the task is encountered

Data Scoping Defaults

- The behavior you want for tasks is usually firstprivate, because the task may not be executed until later (and variables may have gone out of scope)
  - Variables that are private when the task construct is encountered are firstprivate by default

- Variables that are shared in all constructs starting from the innermost enclosing parallel construct are shared by default

```
#pragma omp parallel shared(A) private(B)
{
    ...
#pragma omp task
    {
        int C;
        compute(A, B, C);
    }
}
```

A is shared
B is firstprivate
C is private

Exercise: Traversing linked lists

- Consider the program linked.c
  - Traverses a linked list computing a sequence of Fibonacci numbers at each node.
- Parallelize this program selecting from the following list of constructs:

```
#pragma omp parallel
#pragma omp single
#pragma omp task
int omp_get_num_threads();
int omp_get_thread_num();
double omp_get_wtime();
private(), firstprivate()
```

- Hint: Just worry about the contents of main(). You don't need to make any changes to the "list functions"

Parallel Linked List Traversal

```
#pragma omp parallel
{
  #pragma omp single
   {
    p = listhead ;
    while (p) {
       #pragma omp task firstprivate(p)
             {
               process (p);
             }
       p=next (p) ;
    }
   }
}
```

Only one thread packages tasks

makes a copy of **p** when the task is packaged

When/Where are Tasks Complete?

- At thread barriers (explicit or implicit)
  - all tasks generated inside a region must complete at the next barrier encountered by the threads in that region. Common examples:
    - **Tasks generated inside a single construct**: all tasks complete before exiting the barrier on the single.
    - **Tasks generated inside a parallel region**: all tasks complete before exiting the barrier at the end of the parallel region.

- At taskwait directive
  - i.e. Wait until all tasks defined in the current task have completed.

    ```
    #pragma omp taskwait
    ```
  - Note: applies only to tasks generated in the current task, not to "descendants" .

Example

```
#pragma omp parallel
{
  #pragma omp single
   {
     #pragma omp task
        fred();
     #pragma omp task
        daisy();
     #pragma omp taskwait
     #pragma omp task
        billy();
   }
}
```

`fred()` and `daisy()` must complete before `billy()` starts, but this does not include tasks created inside **fred()** and **daisy()**

`All tasks` including those created inside **fred()** and **daisy()** must complete before exiting this barrier

Example

```
#pragma omp parallel
{
  #pragma omp single nowait
   {
     #pragma omp task
        fred();
     #pragma omp task
        daisy();
     #pragma omp taskwait
     #pragma omp task
        billy();
   }
}
```

The barrier at the end of the single is expensive and not needed since you get the barrier at the end of the parallel region.   So use nowait to turn it off.

`All tasks` including those created inside **fred()** and **daisy()** must complete before exiting this barrier

Example: Fibonacci numbers

```
int fib (int n)
{
   int x,y;
   if (n < 2) return n;

   x = fib(n-1);
   y = fib (n-2);
   return (x+y);
}

int main()
{
   int NW = 5000;
   fib(NW);
}
```

- $F_n = F_{n-1} + F_{n-2}$
- Inefficient $O(2^n)$ recursive implementation!

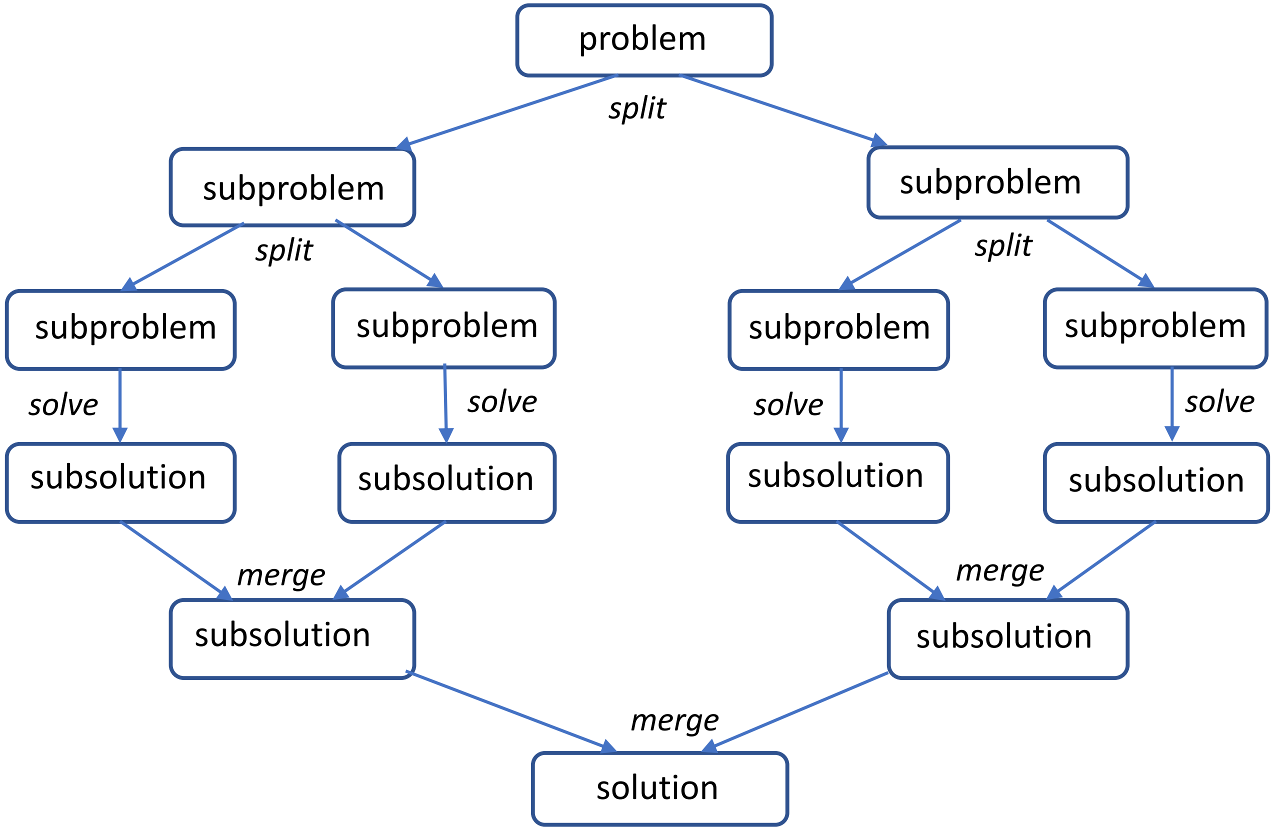
Parallel Fibonacci

```
int fib (int n)
{   int x,y;
    if (n < 2) return n;

#pragma omp task shared(x)
    x = fib(n-1);
#pragma omp task shared(y)
    y = fib (n-2);
#pragma omp taskwait
    return (x+y);
}

Int main()
{  int NW = 5000;
   #pragma omp parallel
   {
      #pragma omp single
           fib(NW);
   }
}
```

- Binary tree of tasks
- Traversed using a recursive function
- A task cannot complete until all tasks below it in the tree are complete (enforced with taskwait)
- `x,y` are local, and so by default they are private to current task
  - must be shared on child tasks so they don't create their own firstprivate copies at this level!

Divide and Conquer

- Split the problem into smaller sub-problems; continue until the sub-problems can be solved directly



- 3 Options for parallelism:
  - ☐ Do work as you split into sub-problems
  - ☐ Do work only at the leaves
  - ☐ Do work as you recombine

Exercise: PI with tasks

- Go back to the original pi.c program
  - Parallelize this program using OpenMP tasks

```
#pragma omp parallel
#pragma omp task
#pragma omp taskwait
#pragma omp single
double omp_get_wtime()
int omp_get_thread_num();
int omp_get_num_threads();
```

- Hint: first create a recursive pi program and verify that it works. **Think about the computation you want to do at the leaves. If you go all the way down to one iteration per leaf-node, won't you just swamp the system with tasks?**

Program: OpenMP tasks

```
include <omp.h>
static long num_steps = 100000000;
#define MIN_BLK  10000000
double pi_comp(int Nstart,int Nfinish,double step)
{   int i,iblk;
    double x, sum = 0.0,sum1, sum2;
    if (Nfinish-Nstart < MIN_BLK){
       for (i=Nstart;i< Nfinish; i++){
          x = (i+0.5)*step;
          sum = sum + 4.0/(1.0+x*x);
       }
    }
    else{
       iblk = Nfinish-Nstart;
       #pragma omp task shared(sum1)
           sum1 = pi_comp(Nstart,         Nfinish-iblk/2,step);
       #pragma omp task shared(sum2)
            sum2 = pi_comp(Nfinish-iblk/2, Nfinish,       step);
       #pragma omp taskwait
          sum = sum1 + sum2;
   }return sum;
}
```

```
int main ()
{
  int i;
  double step, pi, sum;
   step = 1.0/(double) num_steps;
   #pragma omp parallel
   {
      #pragma omp single
          sum =
             pi_comp(0,num_steps,step);
    }
     pi = step * sum;
}
```

Results*: Pi with tasks

threads	1st SPMD	SPMD critical	PI Loop	Pi tasks
1	1.86	1.87	1.91	1.87
2	1.03	1.00	1.02	1.00
3	1.08	0.68	0.80	0.76
4	0.97	0.53	0.68	0.52

*Intel compiler (icpc) with no optimization on Apple OS X 10.7.3 with a dual core (four HW thread) Intel® Core^{TM} i5 processor at 1.7 Ghz and 4 Gbyte DDR3 memory at 1.333 Ghz.

Using Tasks

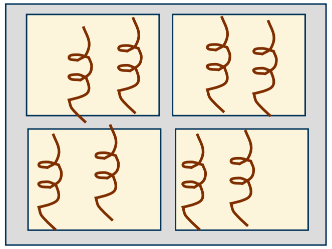
- Don’t use tasks for things already well supported by OpenMP
  - e.g. standard do/for loops
  - the overhead of using tasks is greater

- Don’t expect miracles from the runtime
  - best results usually obtained where the user controls the number and granularity of tasks

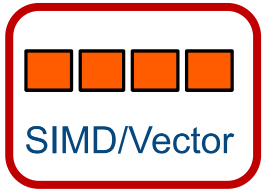
Outline

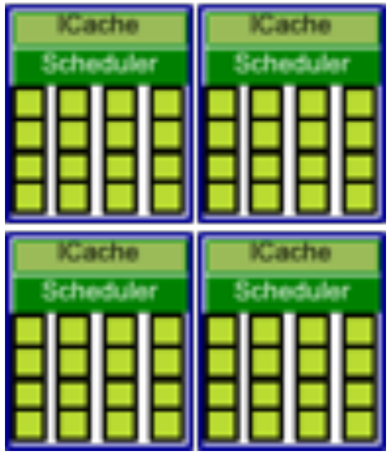
- Introduction to OpenMP
- Creating Threads
- Synchronization
- Parallel Loops
- Data Environment
- Memory Model
- Irregular Parallelism and Tasks
- OpenMP: Beyond the CPU
- Recap

Hardware is diverse

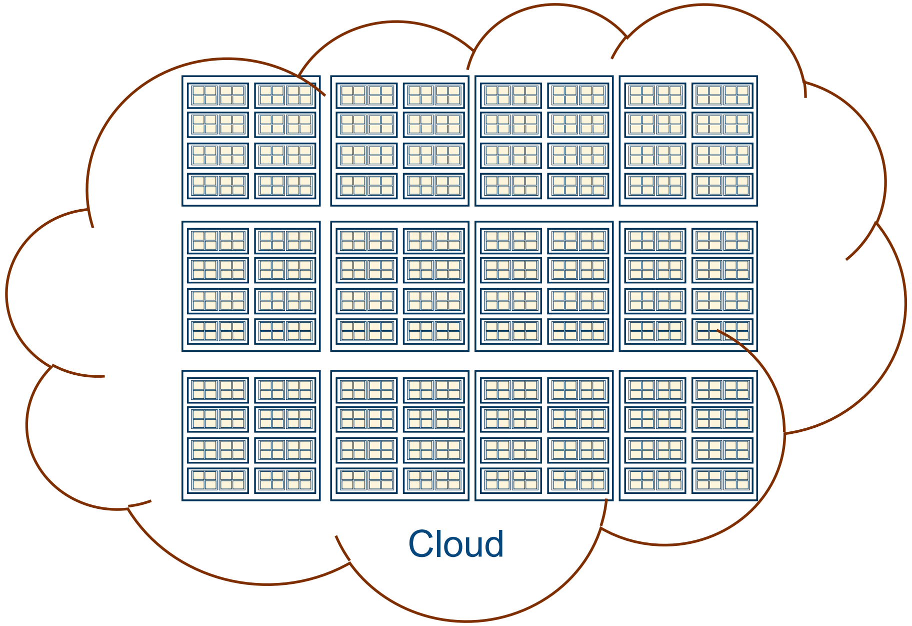


CPU

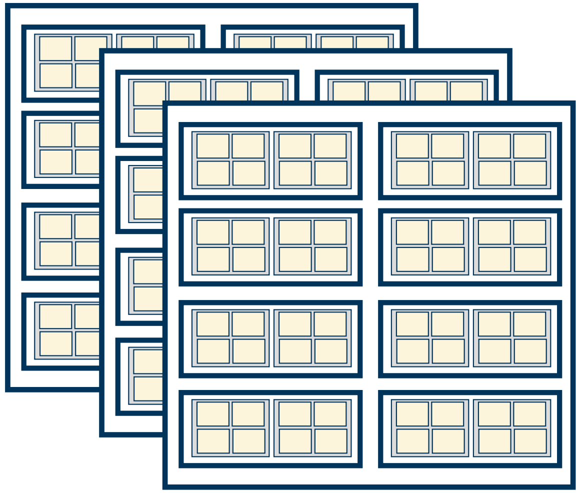




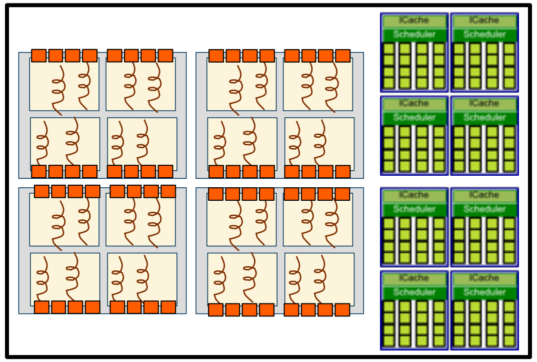
GPU



Cloud



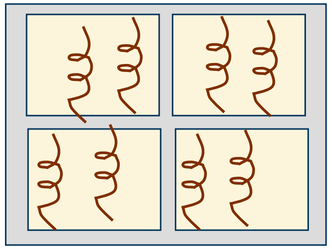
Cluster



Heterogeneous node

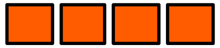
Hardware is diverse

You know all about programming with threads and a shared memories

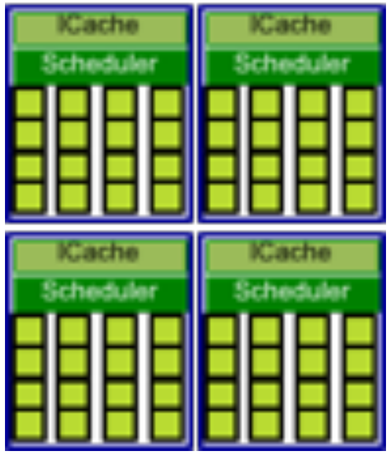


CPU

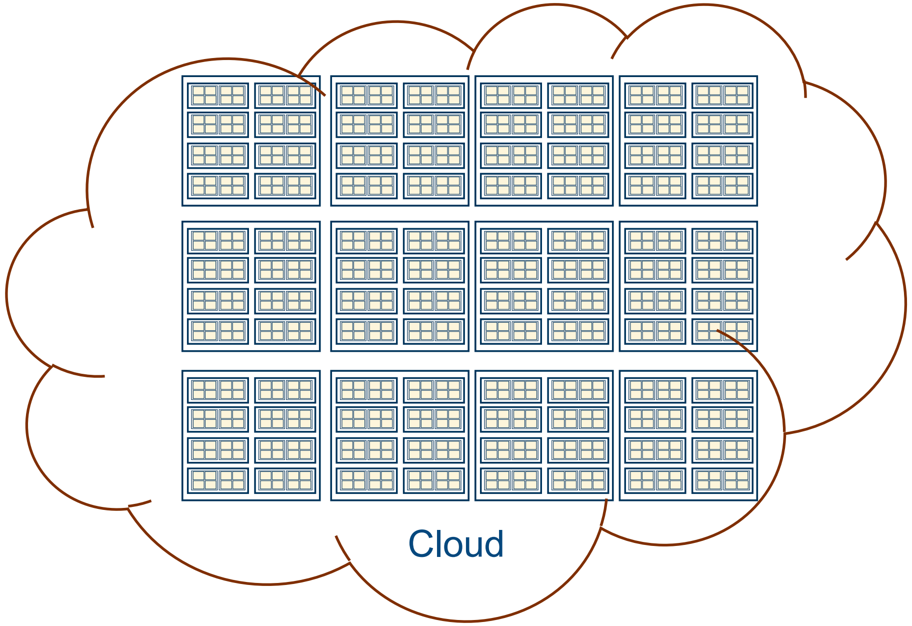
We'll leave vectorization to the compiler (use –O3)



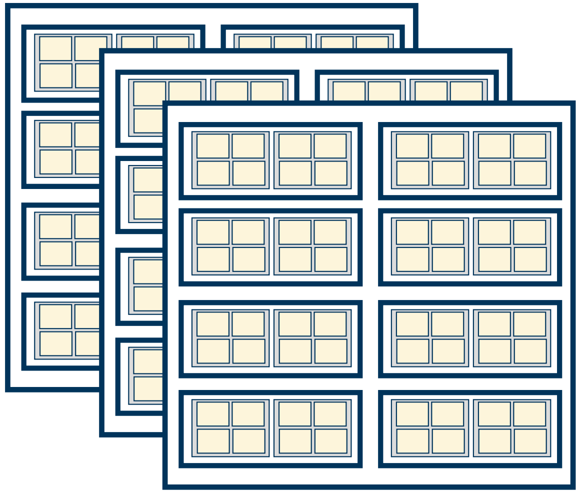
SIMD/Vector



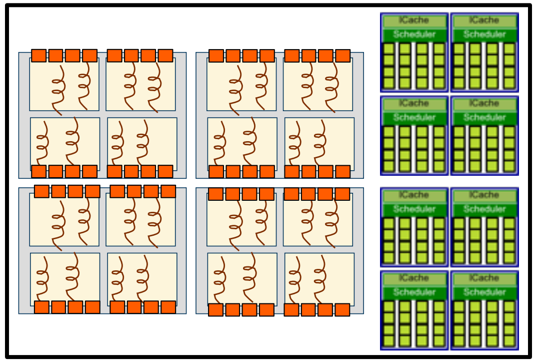
GPU



Cloud

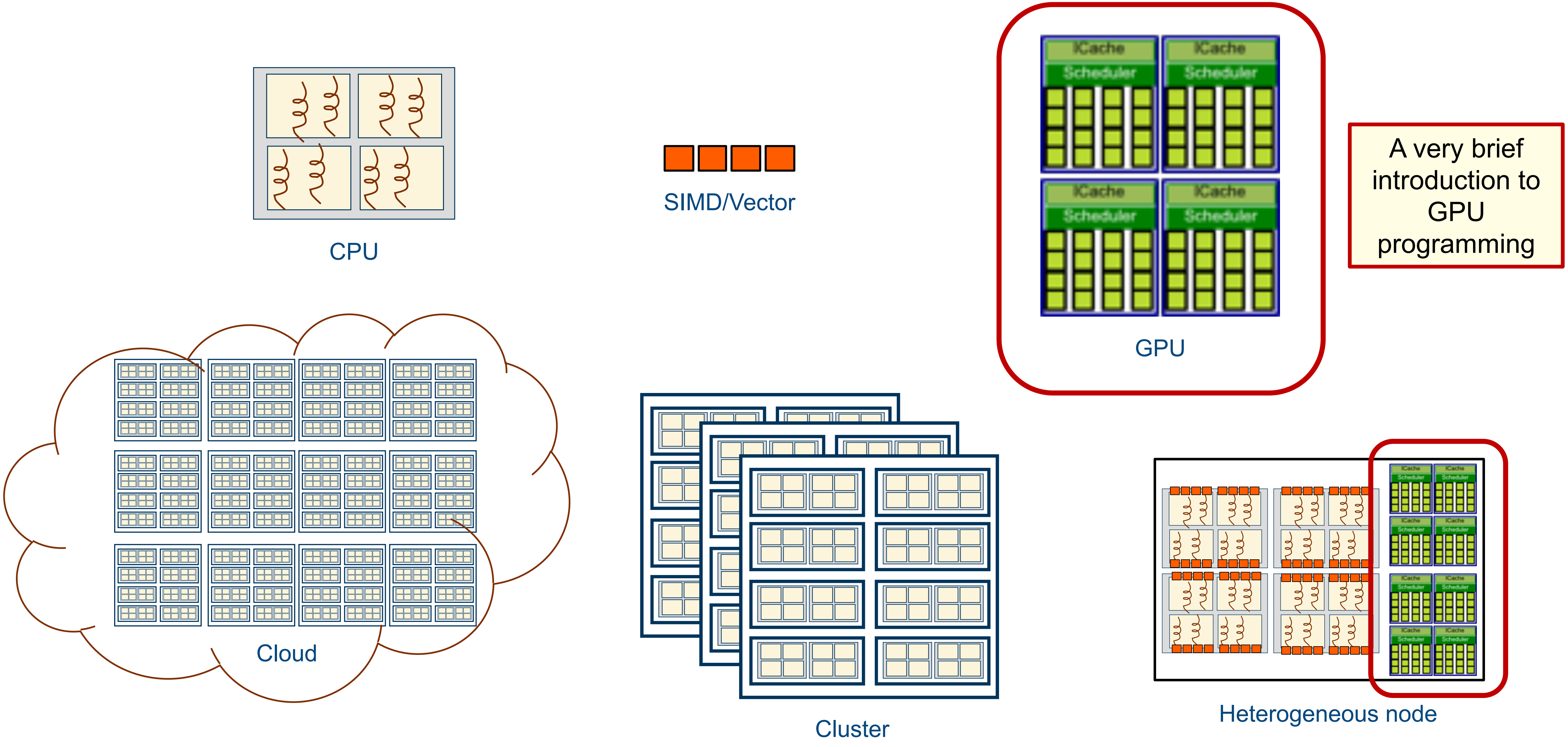


Cluster

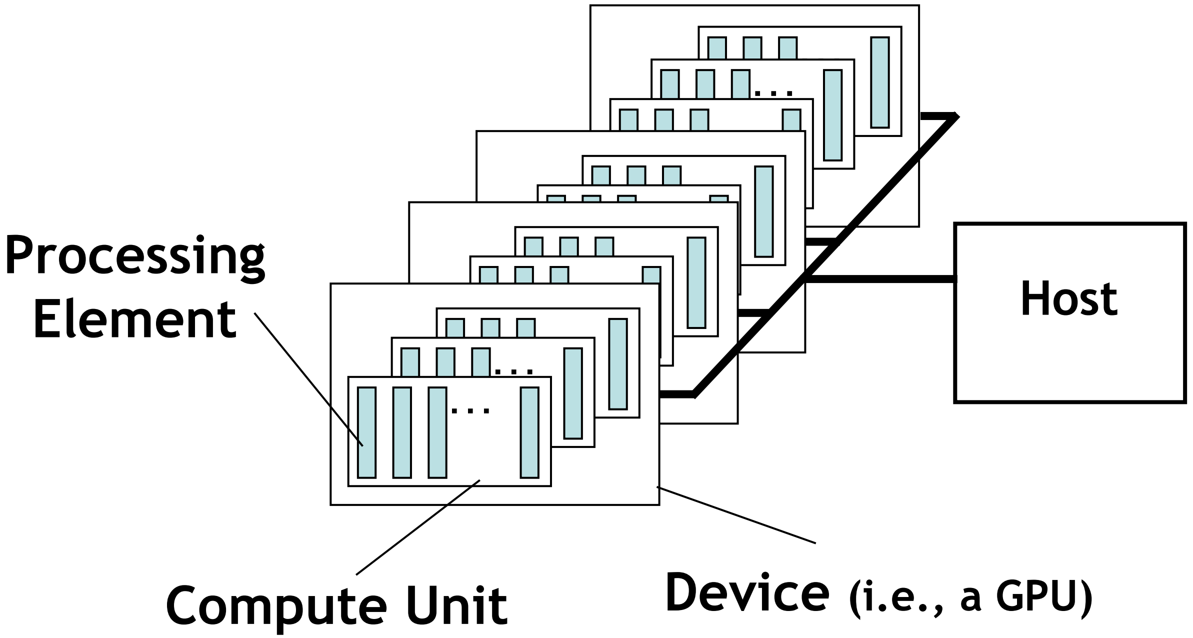


Heterogeneous node

Hardware is diverse

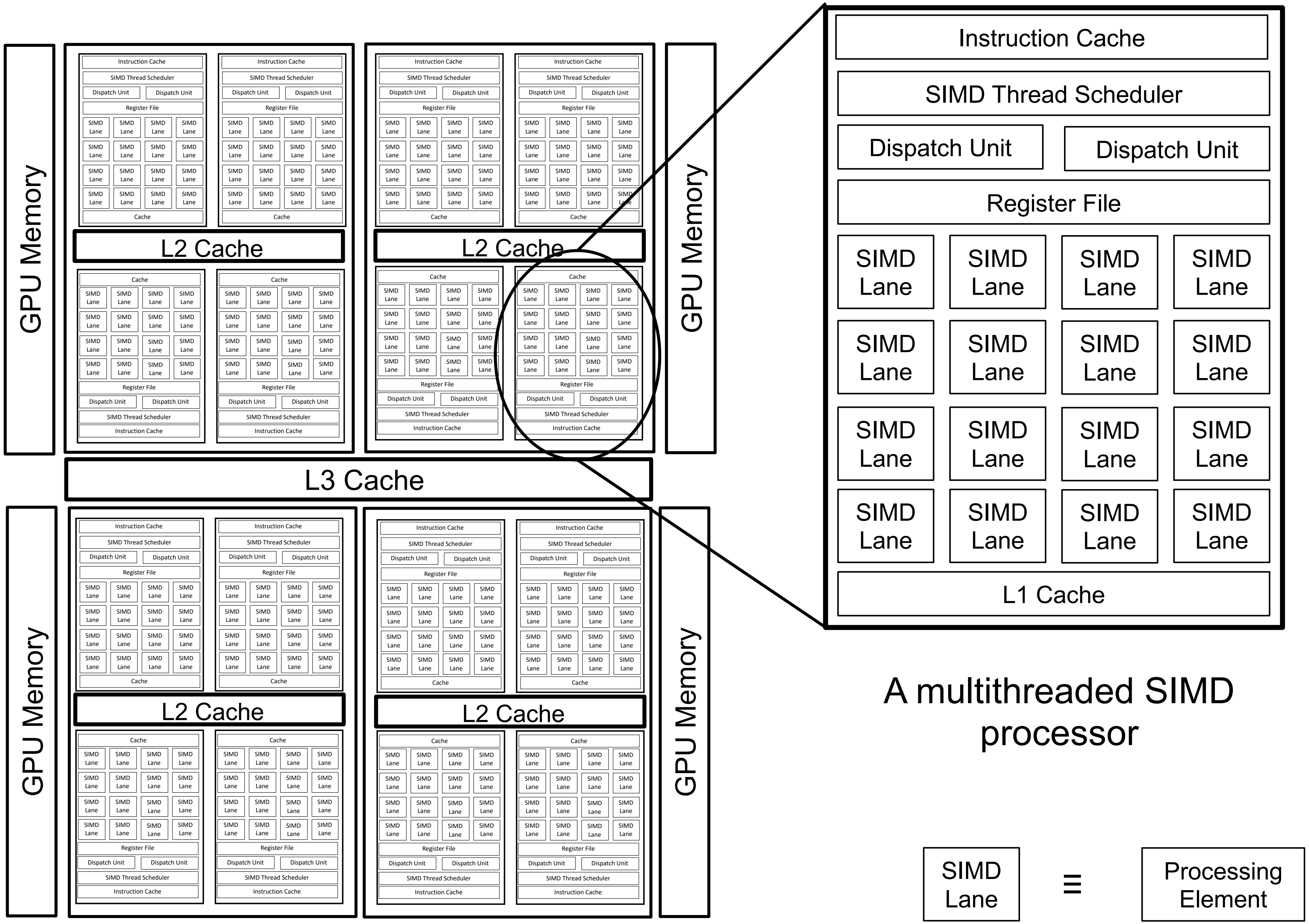


GPU Programming: A Host/Device Platform Model



- One ***Host*** and one or more ***Devices***
  - Each Device is composed of one or more Compute Units
  - Each Compute Unit is divided into one or more ***Processing Elements***
- Memory divided into ***host memory*** and ***device memory***

A Generic GPU (following Hennessey and Patterson)



A multithreaded SIMD processor

SIMD Lane ≡ Processing Element ≡ CUDA Core

Computer Architecture: A Quantitative Approach, John L. Hennessy and David A. Patterson.

The “BIG idea” Behind GPU programming

Traditional Loop based vector addition (vadd)

```
int main() {
   int N = . . . ;
   float *a, *b, *c;

   a* =(float *) malloc(N * sizeof(float));

   // ... allocate other arrays (b and c)
   // and fill with data

   for (int i=0;i<N; i++)
      c[i] = a[i] + b[i];

}
```

Data Parallel vadd with CUDA

```
// Compute sum of length-N vectors: C = A + B
void __global__
vecAdd (float* a, float* b, float* c, int N) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < N) c[i] = a[i] + b[i];
}

int main () {
    int N = ... ;
    float *a, *b, *c;
    cudaMalloc (&a,  sizeof(float) * N);
  // ... allocate other arrays (b and c)
  // and fill with data

  // Use thread blocks with 256 threads each
    vecAdd <<< (N+255)/256, 256 >>> (a, b, c, N);
}
```

Assume a GPU with unified shared memory … allocate on host, visible on device too

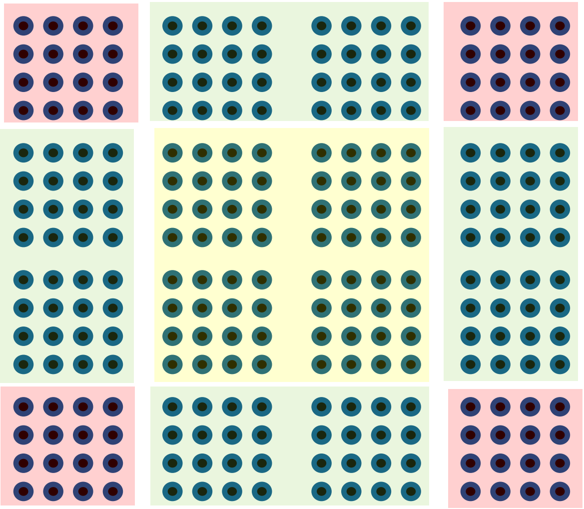
How do we execute code on a GPU: The SIMT model (Single Instruction Multiple Thread)

1. Write kernel code for the scalar work-items

```
// Compute sum of order-N matrices: C = A + B
void __global__
matAdd (float* a, float* b, float* c, int N) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    if (i < N && j<N) c[i][j] == a[i][j] + b[i][j];
}

int main () {
    int N = ... ;
    float *a, *b, *c;
    cudaMalloc (&a,  sizeof(float) * N);
  // ... allocate other arrays (b and c)
  // and fill with data

  // define threadBlocks and the Grid
   dim3 dimBlock(4,4);
   dim3 dimGrid(4,4);

  // Launch kernel on Grid
    matAdd <<< dimGrid,dimBlock>>> (a, b, c, N);
}
```

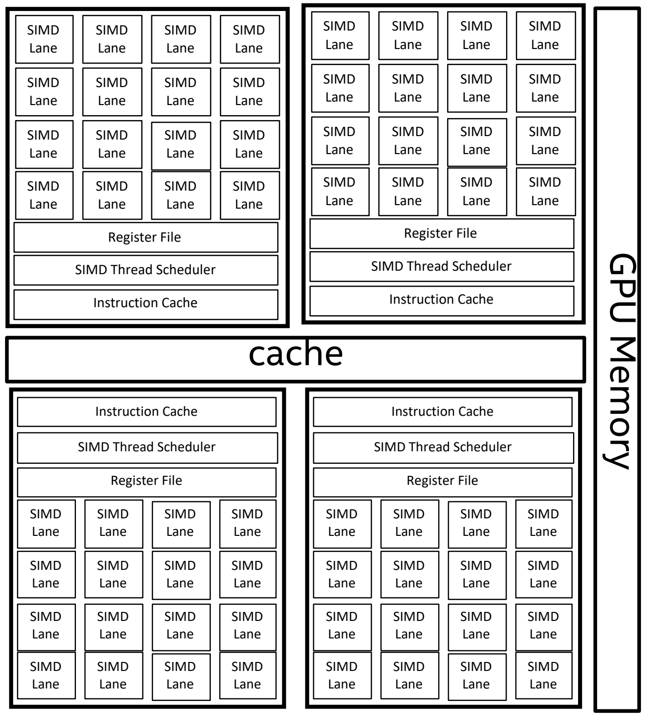
This is CUDA code

2. Map work-items onto an N dim index space.



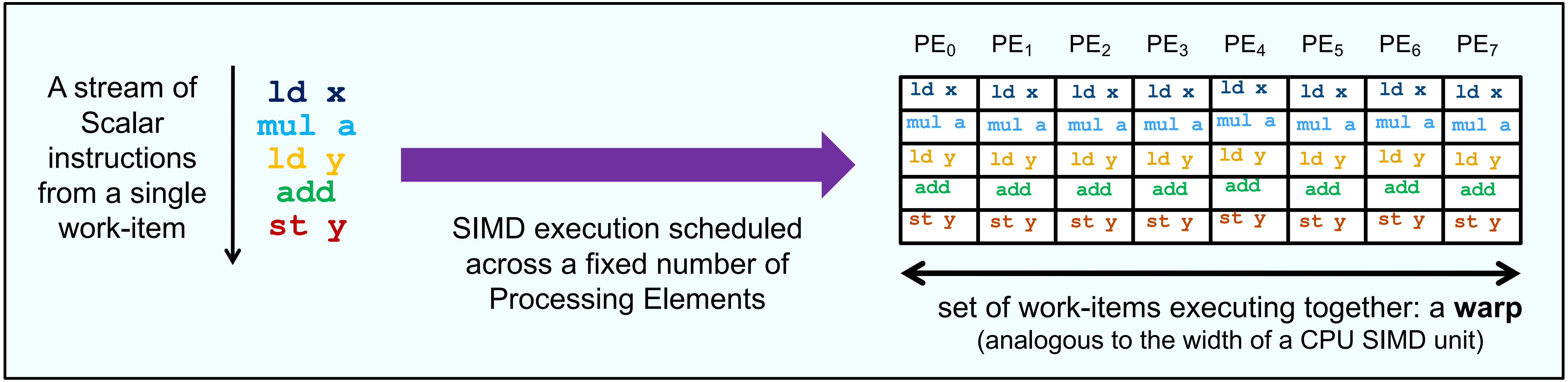
3. Map data structures onto the same index space

4. Run on hardware designed around the same SIMT execution model



SIMT: One instruction stream maps onto many Processing Elements

- SIMT model: Individual scalar instruction streams are grouped together for SIMD execution on hardware

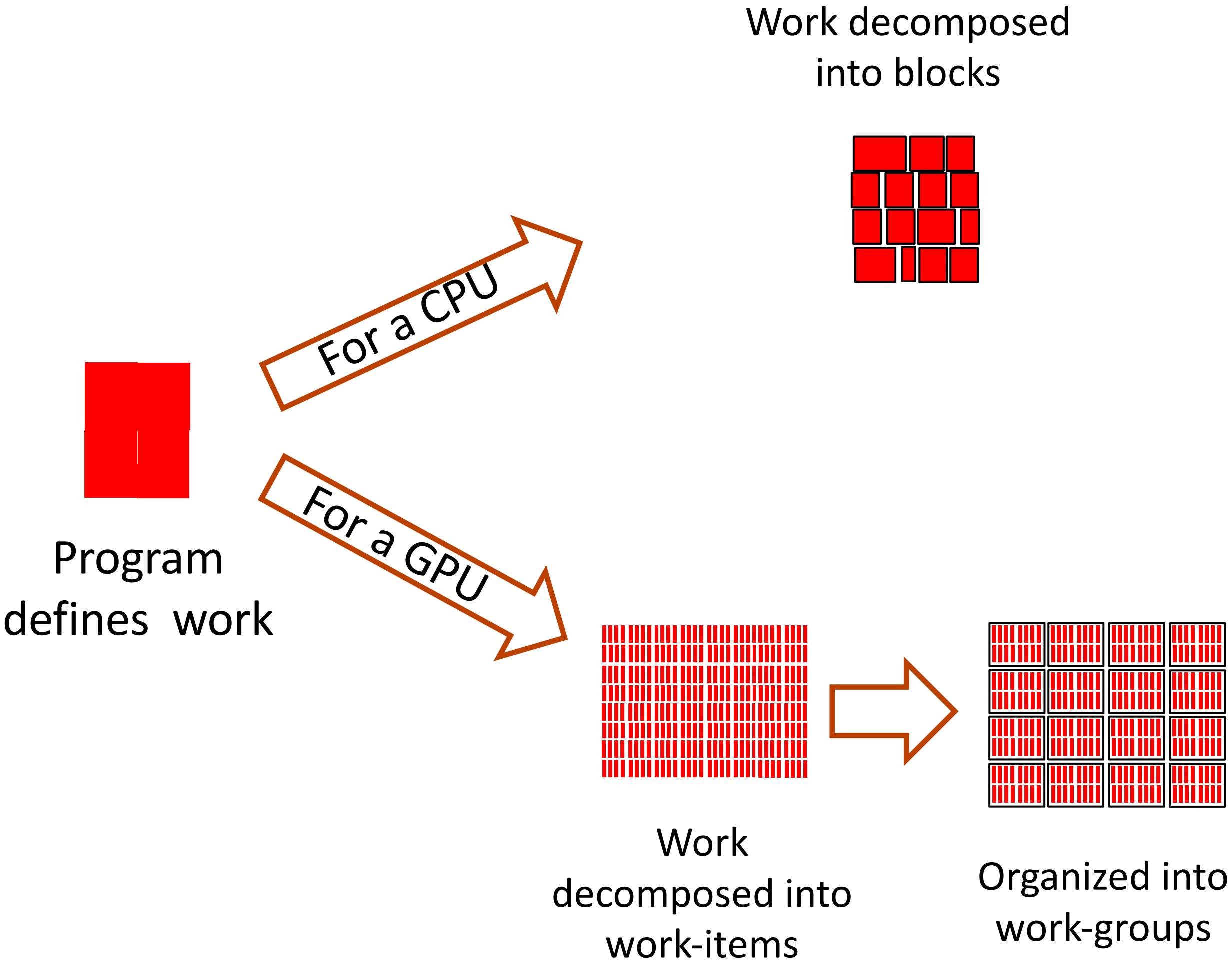


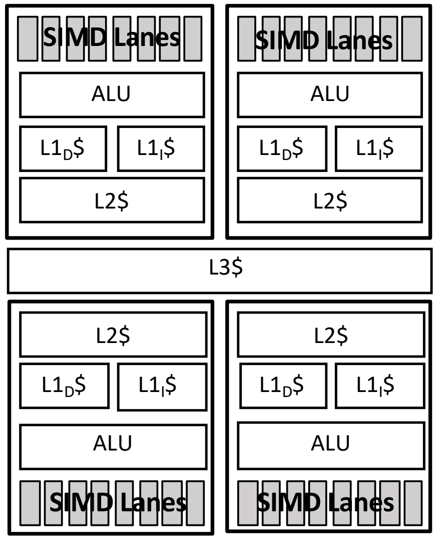
GPU nomenclature is really messed up. (sorry about that … we tried to unify around OpenCL but failed).

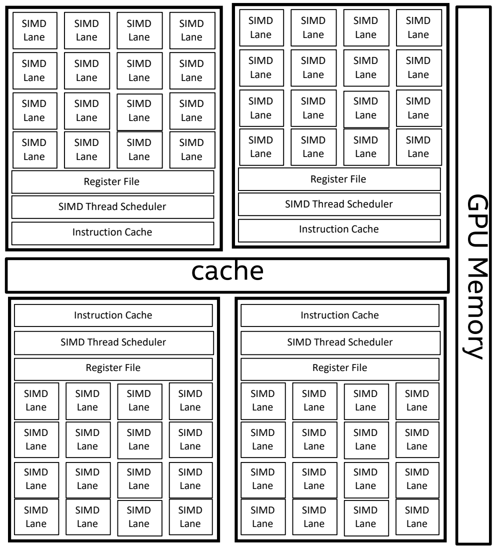
Instruction stream at finest grain	**Work-item**, CUDA Thread
Blocks for scheduling work-items	**work-group**, thread block
Execution width for work-items	Subgroup, **warp**
Finest grained processing element (PE) in a GPU	SIMD Lane, **Processing Element**, CUDA Core
Block of PEs driven by a single Instruction sequencer	multithreaded SIMD processor, compute unit, **Streaming multiprocessor**

These names are particularly awful since they conflict with established names from CPU Computing.

Executing a program on CPUs and GPUs

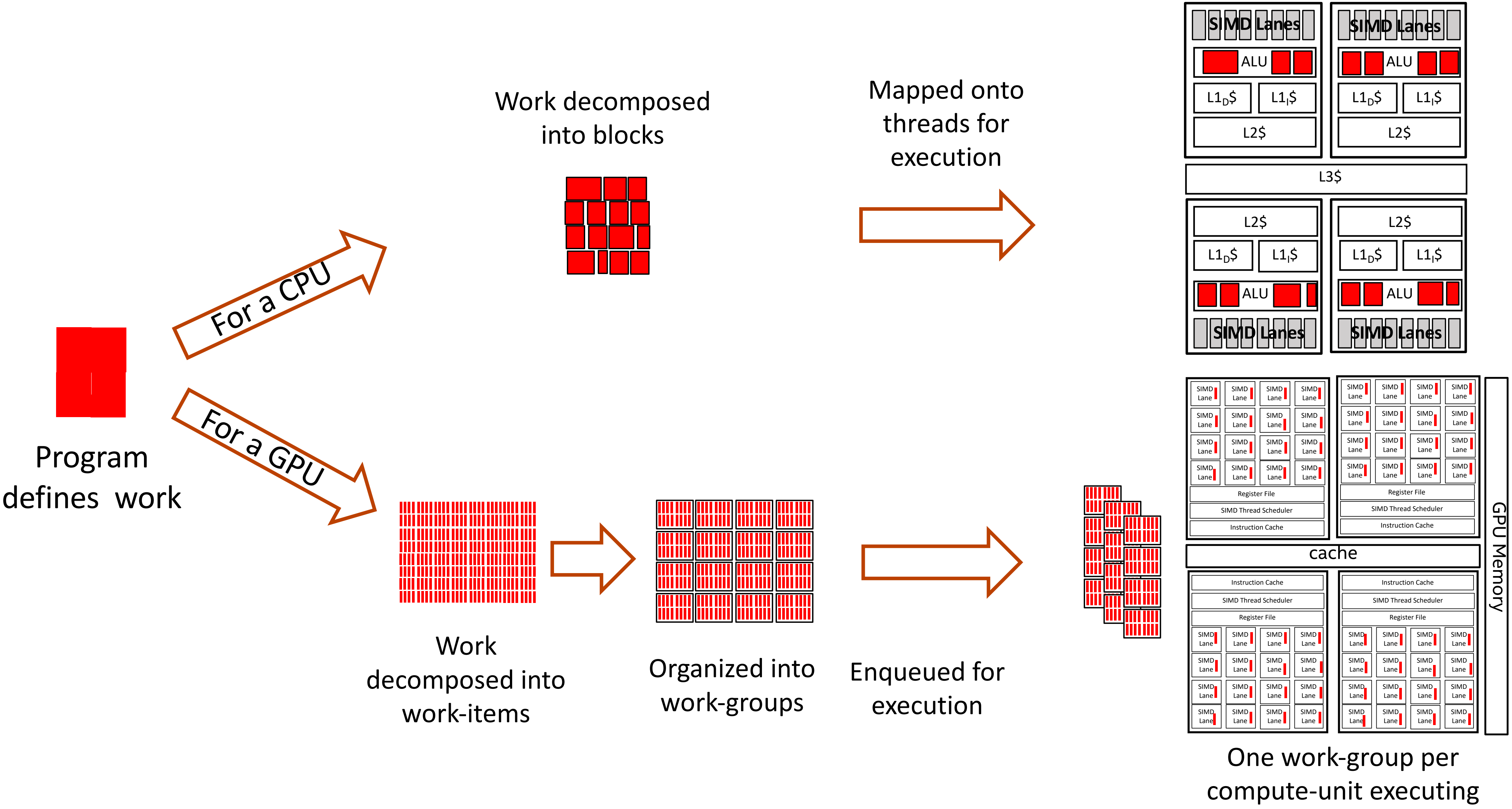




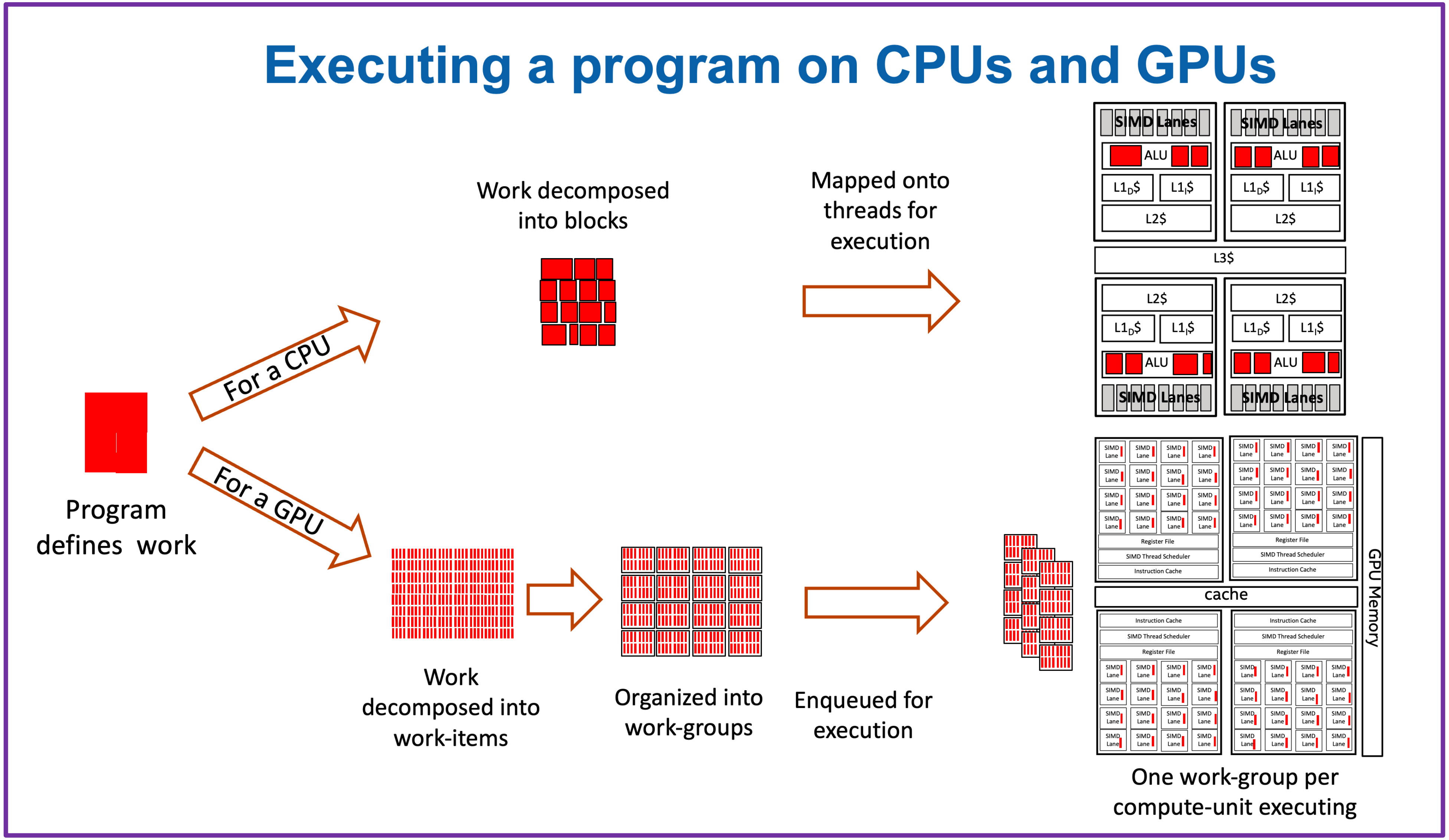


One work-group per compute-unit executing

Executing a program on CPUs and GPUs



CPU/GPU execution models

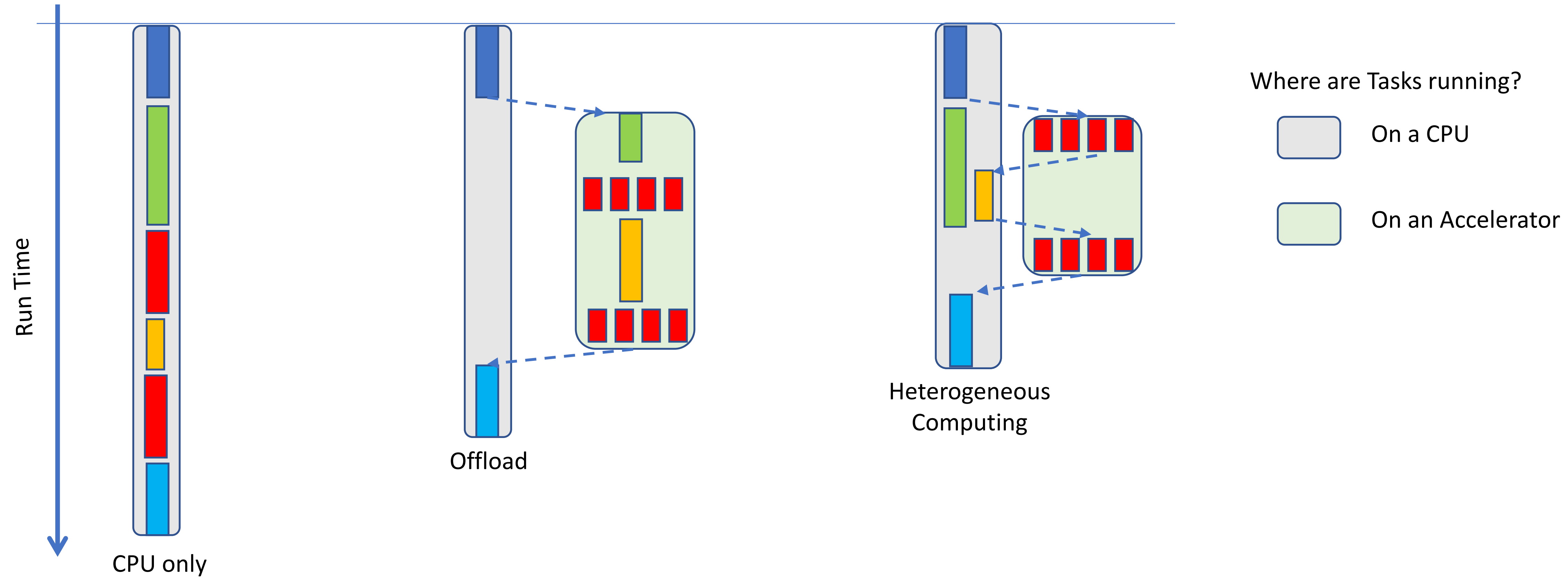


For a CPU, the threads are all active and able to make forward progress. Optimized for latency sensitive problems

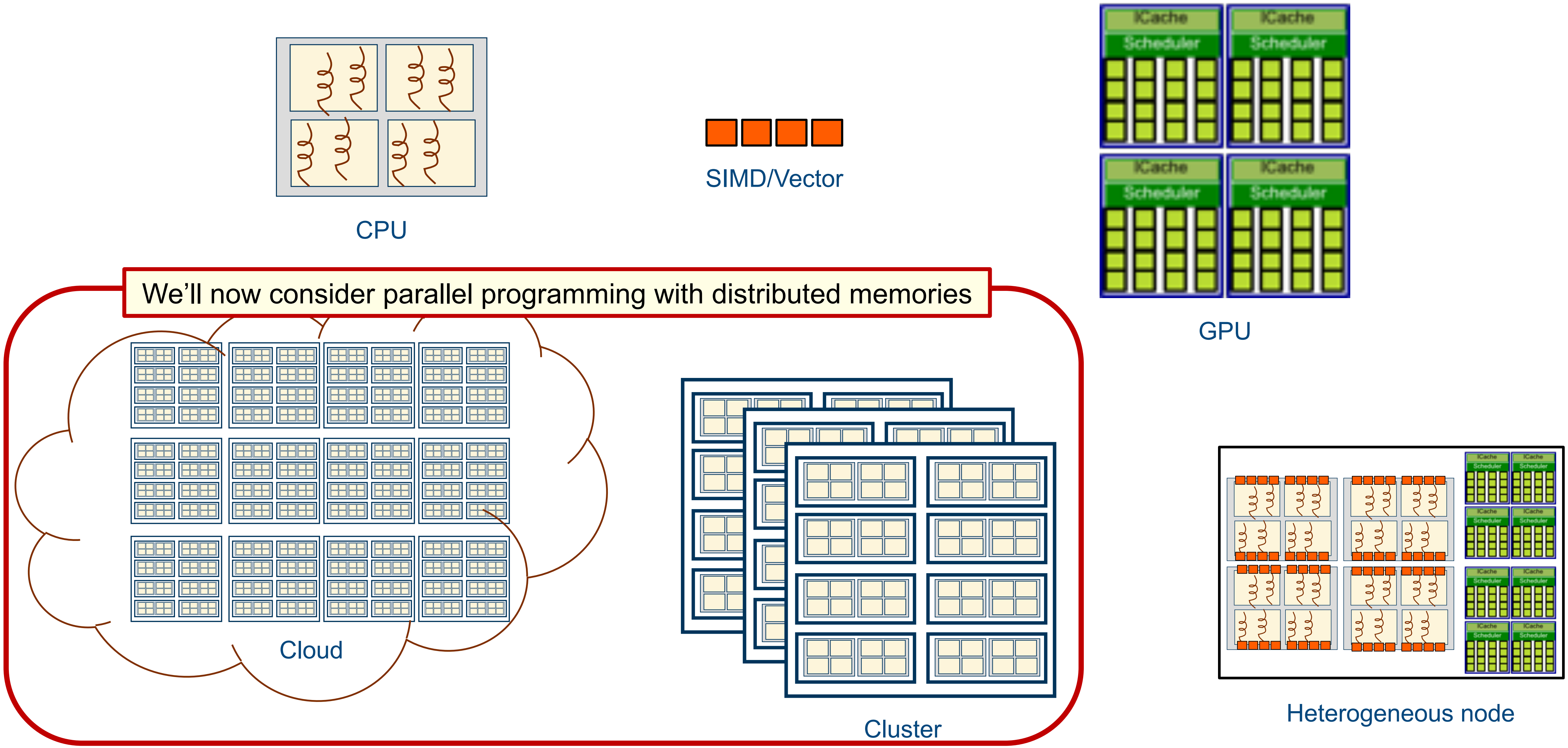
For a GPU, any given work-group might be in the queue waiting to execute. Optimized for high throughput workflows

No single processor is best at everything

- The idea that you should move everything to the GPU makes no sense
- **Heterogeneous Computing**: Run sub-problems in parallel on the hardware best suited to them.



Hardware is diverse … and its only getting worse!!!



Parallel API's: MPI … the Message Passing Interface

MPI_Type_contiguous

MPI_Bcast

MPI_Recv_init

MPI_Scan

MPI_Type_free

MPI_Group_size

MPI_Unpack

MPI_COMM_WORLD

MPI_Allgather

MPI_Errhandler_create

MPI_Group_compare

CSOMP_ORDERED

MPI_Sendrecv

MPI_Barrier

MPI_Startall

MPI_Start

MPI_Reduce

MPI_Test_cancelled

MPI_Pack

MPI_Recv

MPI_Bsend_init

omp_set_lock(lck)

MPI_Sendrecv_replace

MPI_Ssend

MPI_Waitall

MPI_Alltoallv

MPI_Send

MPI: An API for Writing Applications for Distributed memory systems

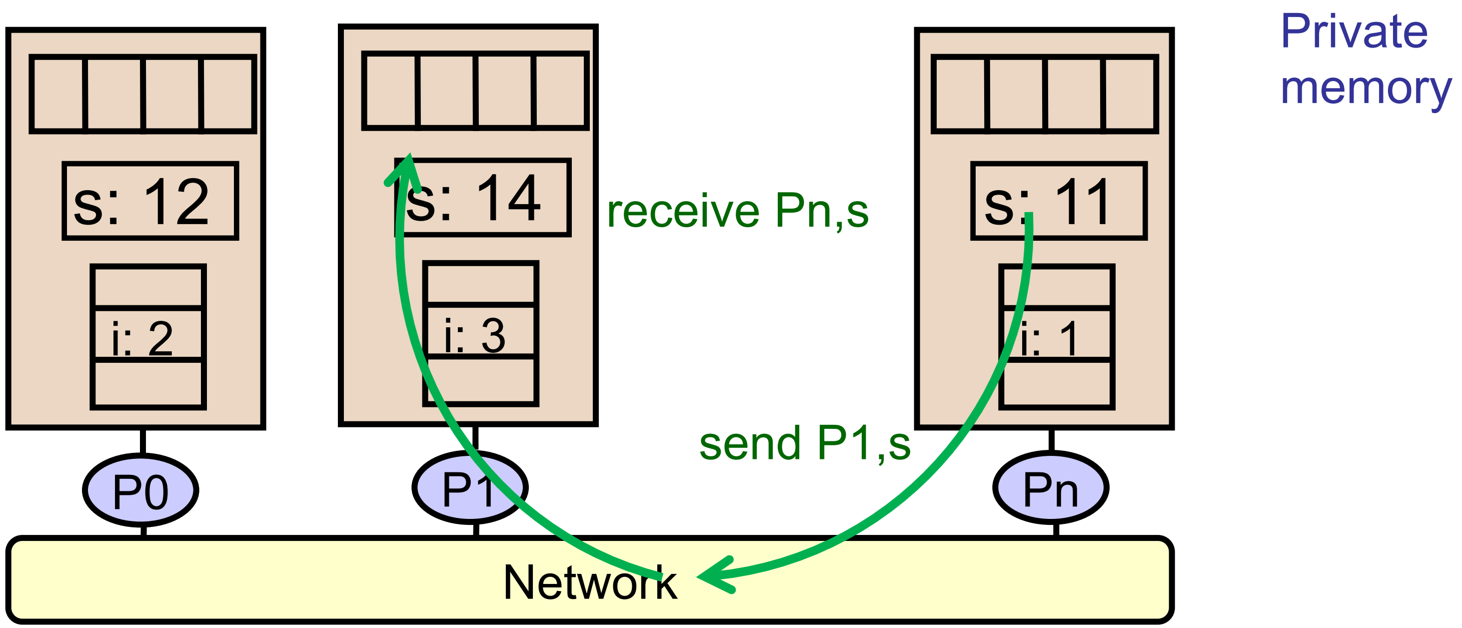
- A library of routines to coordinate the execution of multiple processes.
- Provides point to point and collective communication in Fortran, C and C++
- Unifies last 35+ years of cluster computing and MPP practice

MPP: Massively Parallel Processors

Programming Model: Message Passing

- Program consists of a collection of processes.
  - **Number of processes almost always fixed at program startup time**
  - **Local address space per node -- NO physically shared memory.**
  - **Logically shared data is partitioned over local processes.**

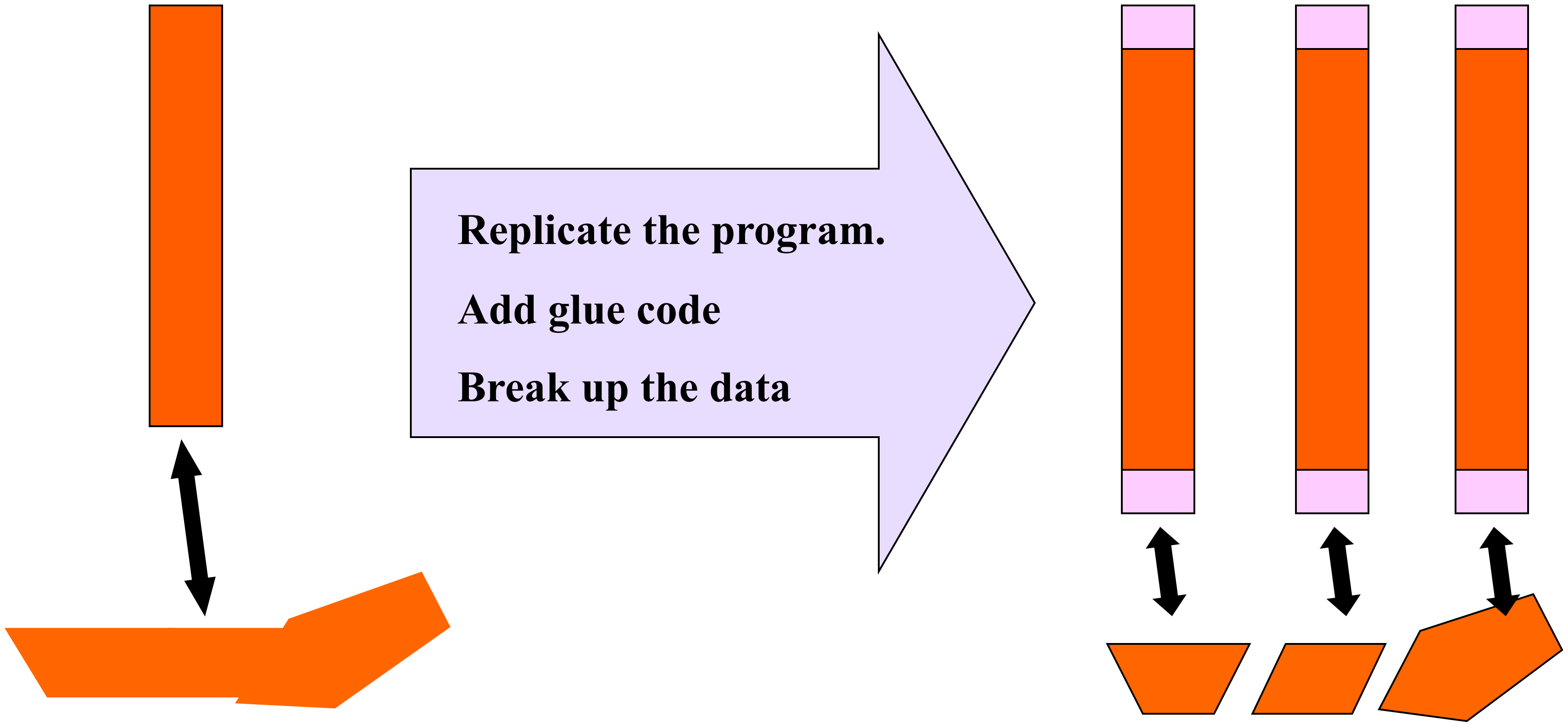
- Processes communicate by explicit send/receive pairs
  - **Synchronization is implicit by communication events.**
  - **MPI (Message Passing Interface) is the most commonly used API**



How do people use MPI? The SPMD Design Pattern

- A single program working on a decomposed data set.
- Use Node ID and numb of nodes to split up work between processes
- Coordination by passing messages.

A sequential program working on a data set

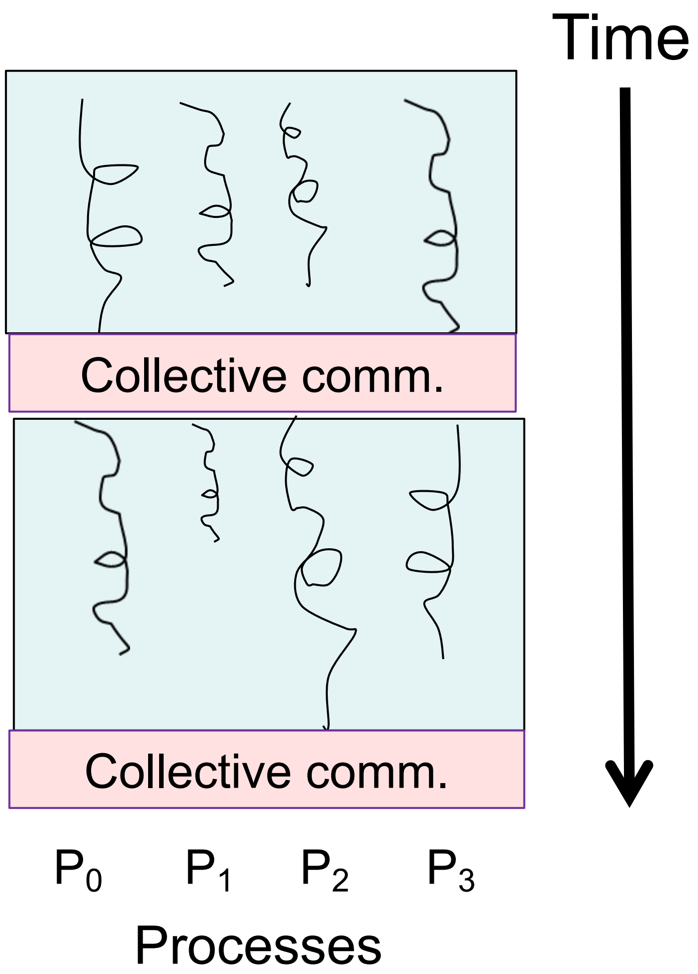


Bulk Synchronous Programming (BSP):

A common design pattern used with MPI Programs

BSP includes the map-reduce pattern commonly used in the cloud. It's easy to use and surprisingly useful

- Many MPI applications have few (if any) sends and receives. They use the following very common pattern:
  - Use the Single Program Multiple Data pattern
  - Each process maintains a local view of the global data
  - A problem broken down into phases each of which is composed of two subphases:
    - Compute on local view of data
    - Communicate to update global view on all processes (collective communication).
  - Continue phases until complete



This is a subset of the SPMD pattern sometimes referred to as the Bulk Synchronous pattern.

Example: finite difference methods

- Solve the heat diffusion equation in 1 D:
  - u(x,t) describes the temperature field
  - We set the heat diffusion constant to one
  - Boundary conditions, constant u at endpoints.

$$\frac{\partial^2 u}{\partial x^2} = \frac{\partial u}{\partial t}$$

- map onto a mesh with stepsize h and k

$$x_i = x_0 + ih \qquad t_i = t_0 + ik$$

- Central difference approximation for spatial derivative (at fixed time)

$$\frac{\partial^2 u}{\partial x^2} = \frac{u_{j+1} - 2u_j + u_{j-1}}{h^2}$$

- Time derivative at t = t^{n+1}

$$\frac{du}{dt} = \frac{u^{n+1} - u^n}{k}$$

Example: Explicit finite differences

- Combining time derivative expression using spatial derivative at t = tn

$$\frac{u_j^{n+1} - u_j^n}{k} = \frac{u_{j+1}^n - 2u_j^n + u_{j-1}^n}{h^2}$$

- Solve for u at time n+1 and step j

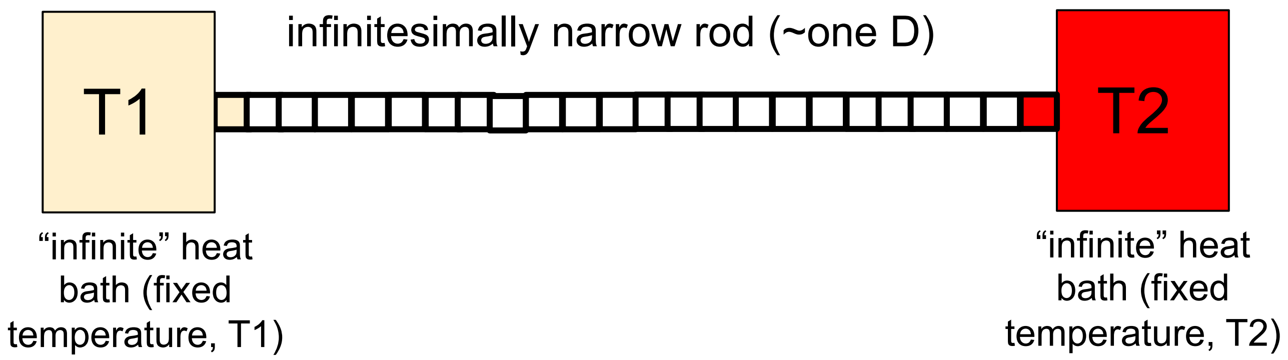
$$u_j^{n+1} = (1-2r)u_j^n + ru_{j-1}^n + ru_{j+1}^n \qquad r = k/h^2$$

- The solution at $t = t_{n+1}$ is determined explicitly from the solution at $t = t_n$ (assume u[t][0] = u[t][N] = Constant for all t).

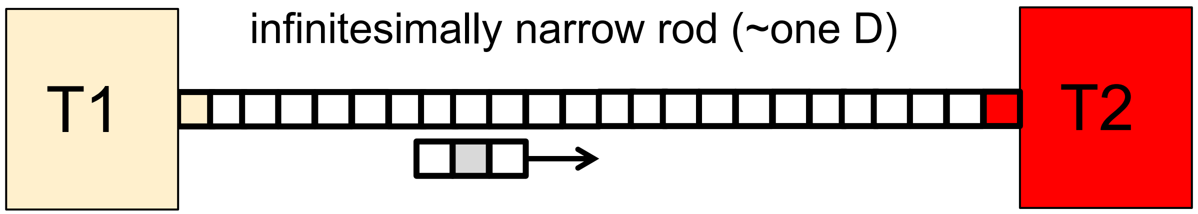
```
for (int t = 0; t < N_STEPS-1; ++t)
   for (int x = 1; x < N-1; ++x)
         u[t+1][x] = u[t][x] + r*(u[t][x+1] - 2*u[t][x] + u[t][x-1]);
```

- Explicit methods are easy to compute … each point updated based on nearest neighbors. Converges for r<1/2.

Heat Diffusion equation



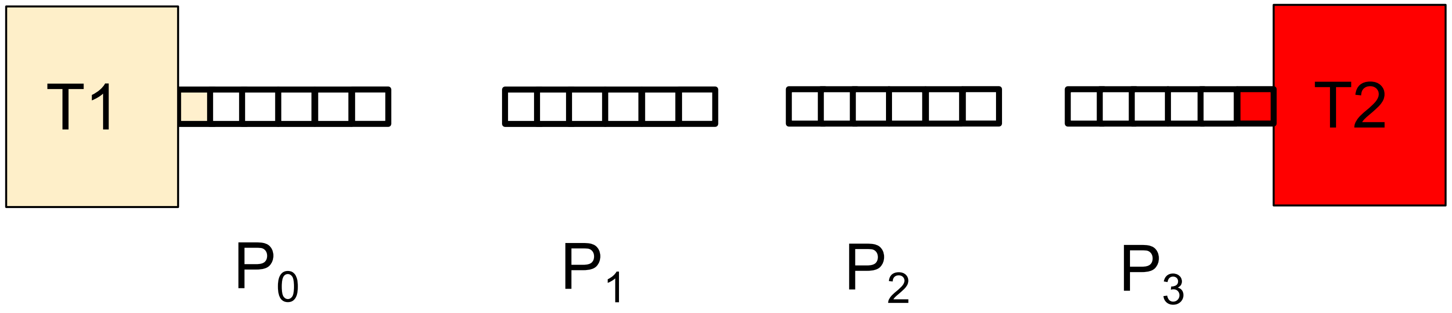
Heat Diffusion equation



Pictorially, you are sliding a three point "stencil" across the domain (u) and updating the center point at each stop.

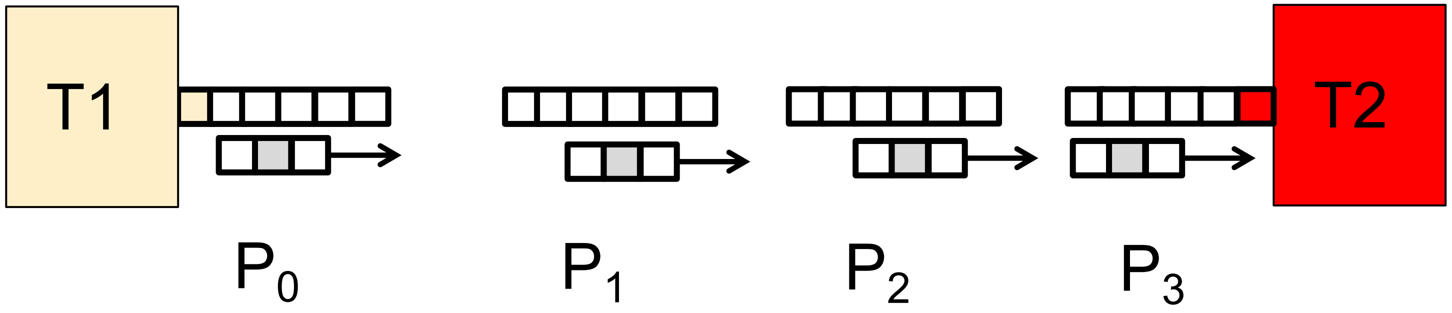
Heat Diffusion equation: in parallel

- Break it into chunks assigning one chunk to each process.



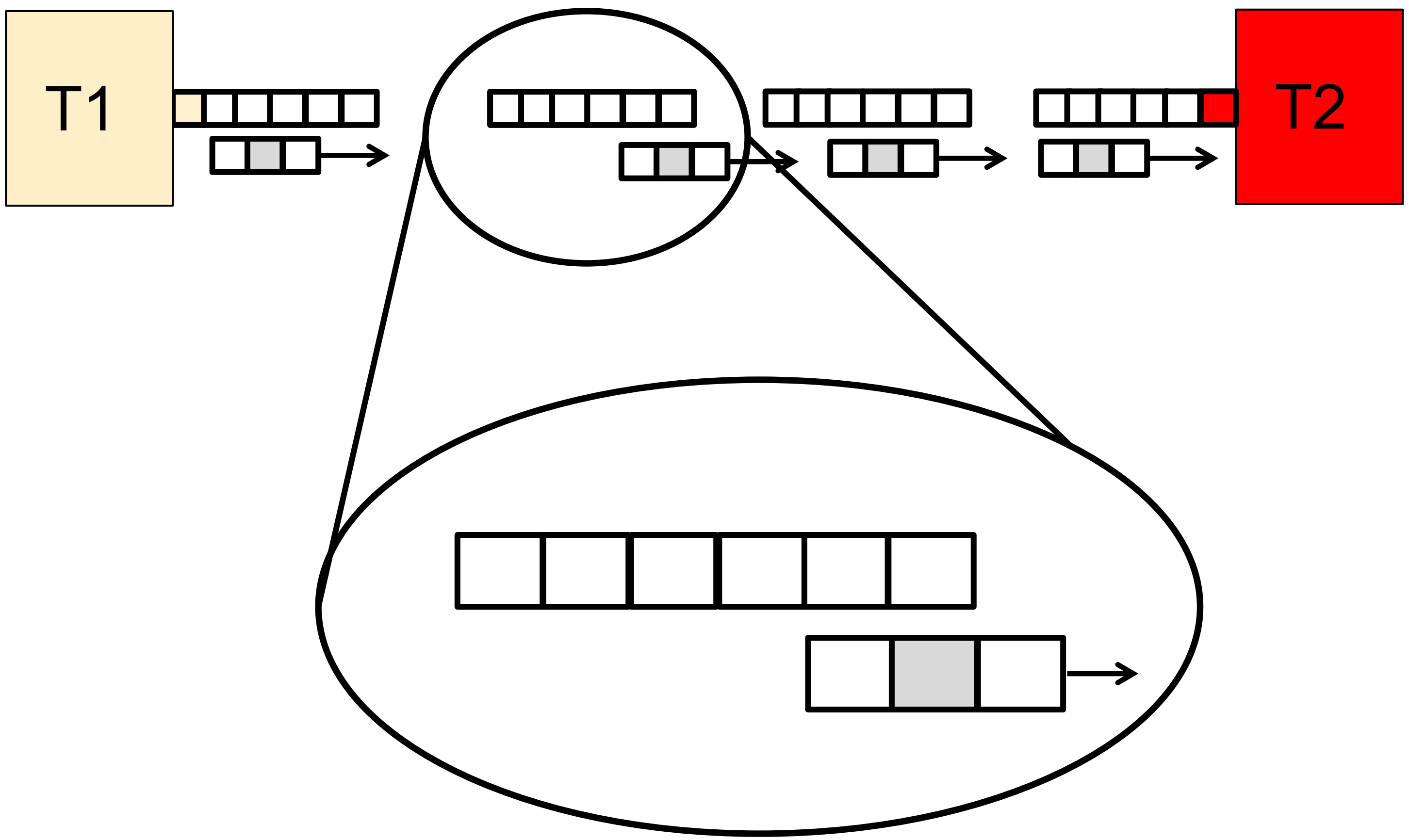
Heat Diffusion equation

- Each process works on its own chunk … sliding the stencil across the domain to updates its own data.



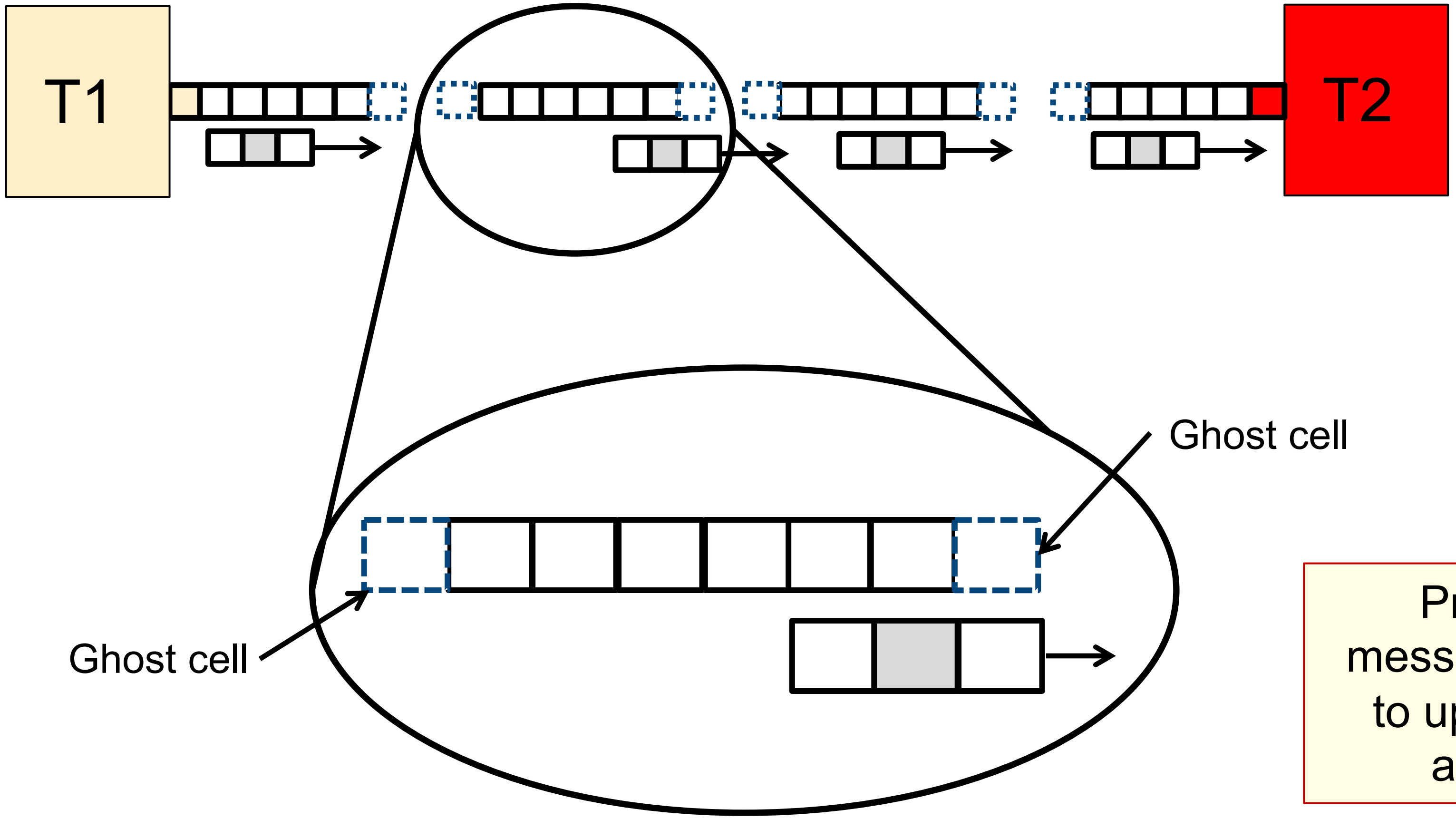
Heat Diffusion equation

- What about the ends of each chunk … where the stencil runs off the end and has missing values for the computation?



Heat Diffusion equation

- We add ghost cells to the ends of each chunk, update them with the required values from neighbor chunks at each time step … hence giving the stencil everything it needs on any given chunk to update all of its values.



Processes send messages to neighbors to update ghost cells after each step

Heat Diffusion MPI Example

```
MPI_Init (&argc, &argv);
MPI_Comm_size (MPI_COMM_WORLD, &P);
MPI_Comm_rank (MPI_COMM_WORLD, &myID);
double *u   = malloc (sizeof(double) * (2 + N/P))  // include "Ghost Cells" to hold
double *up1 = malloc (sizeof(double) * (2 + N/P)); // values from my neighbors

initialize_data(uk, ukp1, N, P);
for (int t = 0; t < N_STEPS; ++t){
  if (myID != 0)  MPI_Send (&u[1], 1, MPI_DOUBLE, myID-1, 0, MPI_COMM_WORLD);
  if (myID != P-1) MPI_Recv (&u[N/P+1], 1, MPI_DOUBLE, myID+1, 0, MPI_COMM_WORLD, &status);
  if (myID != P-1) MPI_Send (&u[N/P], 1, MPI_DOUBLE, myID+1, 0, MPI_COMM_WORLD);
  if (myID != 0)   MPI_Recv (&u[0], 1, MPI_DOUBLE, myID-1, 0,MPI_COMM_WORLD, &status);

  for (int x = 1; x <= N/P; ++x)
    up1[x] = u[x] + (k / (h*h)) * (u[x+1] - 2*u[x] + u[x-1]);
  if (myID != 0)
    up1[1] = u[1] + (k / (h*h)) * (u[1+1] - 2*u[1] + u[1-1]);
  if (myID != P-1)
    up1[N/P] = u[N/P] + (k/(h*h)) * (u[N/P+1] - 2*u[N/P] + u[N/P-1]);
  temp = up1; up1 = u; u = temp;

} // End of for (int t ...) loop

MPI_Finalize();
return 0;
```

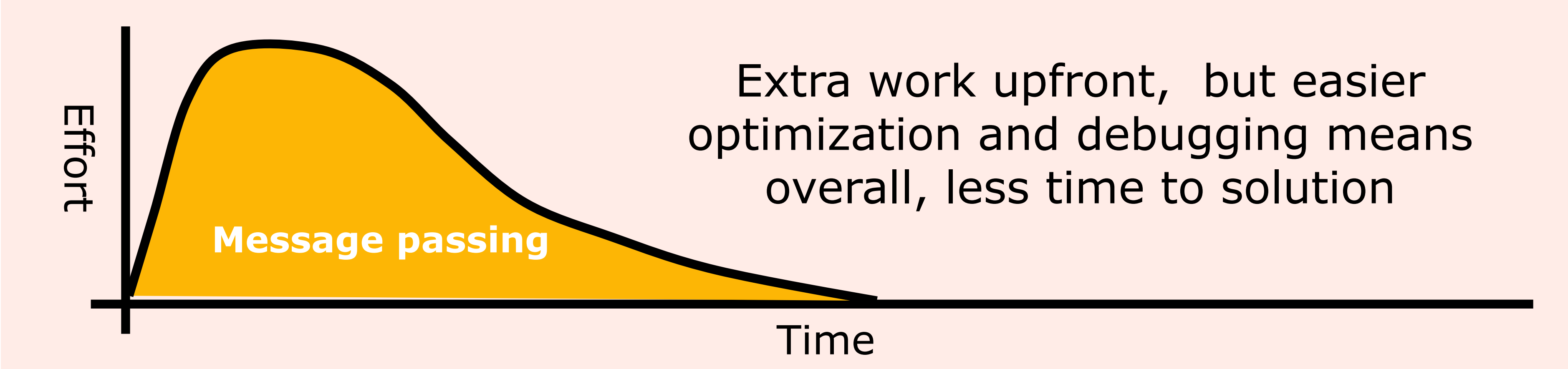
Exchange ghost cells

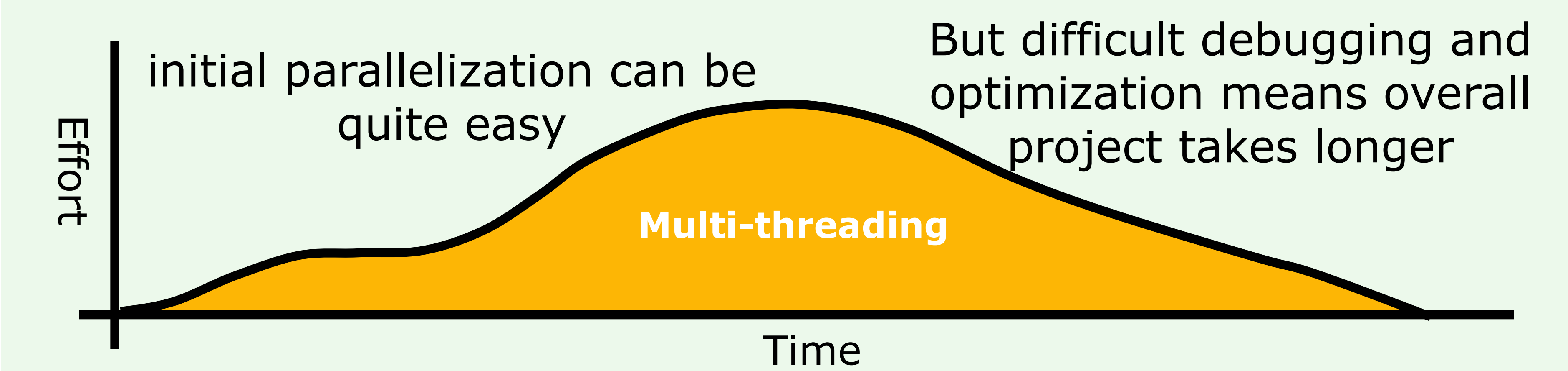
Do the computation, but keep track of boundary conditions

The code can get a bit ugly as each process figures out who to send to and who to receive from.

… but the concept is straightforward and eventually you get used to it.

Does a shared address space make programming easier?





Proving that a shared address space program using semaphores is race free is an NP-complete problem*

*P. N. Klein, H. Lu, and R. H. B. Netzer, Detecting Race Conditions in Parallel Programs that Use Semaphores, Algorithmica, vol. 35 pp. 321–345, 2003

Outline

- Introduction to OpenMP
- Creating Threads
- Synchronization
- Parallel Loops
- Data Environment
- Memory Model
- Irregular Parallelism and Tasks
- OpenMP: Beyond the CPU
- Recap

The OpenMP Common Core: Most OpenMP programs only use these 21 items

OpenMP pragma, function, or clause	Concepts
#pragma omp parallel	Parallel region, teams of threads, structured block, interleaved execution across threads.
void omp_set_thread_num() int omp_get_thread_num() int omp_get_num_threads()	Default number of threads and internal control variables. SPMD pattern: Create threads with a parallel region and split up the work using the number of threads and the thread ID.
double omp_get_wtime()	Speedup and Amdahl's law. False sharing and other performance issues.
setenv OMP_NUM_THREADS N	Setting the internal control variable for 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 (static [,chunk]) schedule(dynamic [,chunk])	Loop schedules, loop overheads, and load balance.
shared(list), private(list), firstprivate(list)	Data environment.
default(none)	Force explicit definition of each variable’s storage attribute
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.

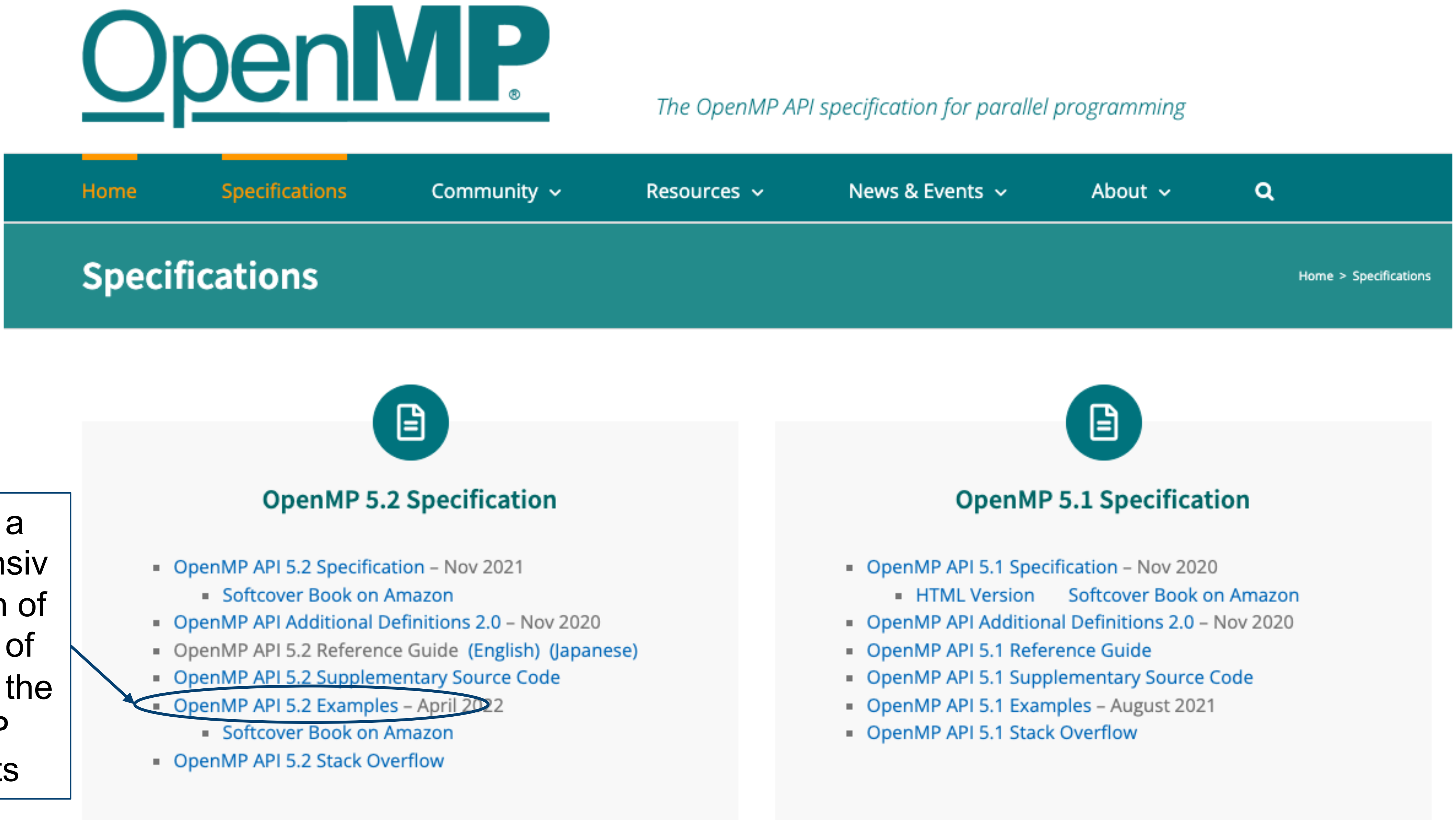
There is Much More to OpenMP than the Common Core

- Synchronization mechanisms
  - locks, synchronizing flushes and several forms of atomic
- Data environment
  - lastprivate, threadprivate, default(private|shared)
- Fine grained task control
  - dependencies, tied vs. untied tasks, task groups, task loops …
- Vectorization constructs
  - simd, uniform, simdlen, inbranch vs. nobranch, ….
- Map work onto an attached device (such as a GPU)
  - target, teams distribute parallel for, target data …
- … and much more. The OpenMP 5.0 specification is over 618 pages!!!

Don't become overwhelmed. Master the common core and move on to other constructs when you encounter problems that require them.

Resources

- www.openmp.org has a wealth of helpful resources



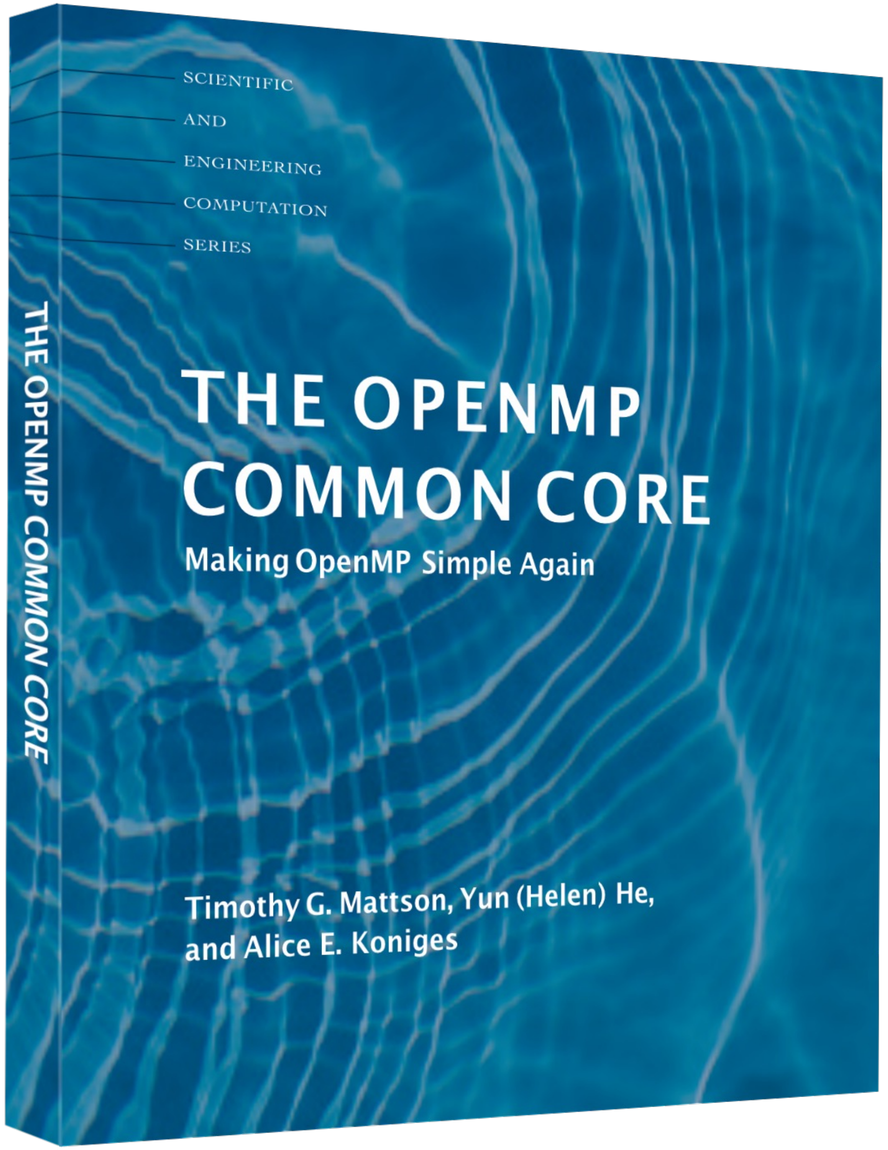
Including a comprehensive collection of examples of code using the OpenMP constructs

To learn OpenMP:

- An exciting new book that Covers the Common Core of OpenMP plus a few key features beyond the common core that people frequently use

- It's geared towards people learning OpenMP, but as one commentator put it … **everyone at any skill level should read the memory model chapters**.

- Available from MIT Press



www.ompcore.com for code samples and the Fortran supplement

Books about OpenMP

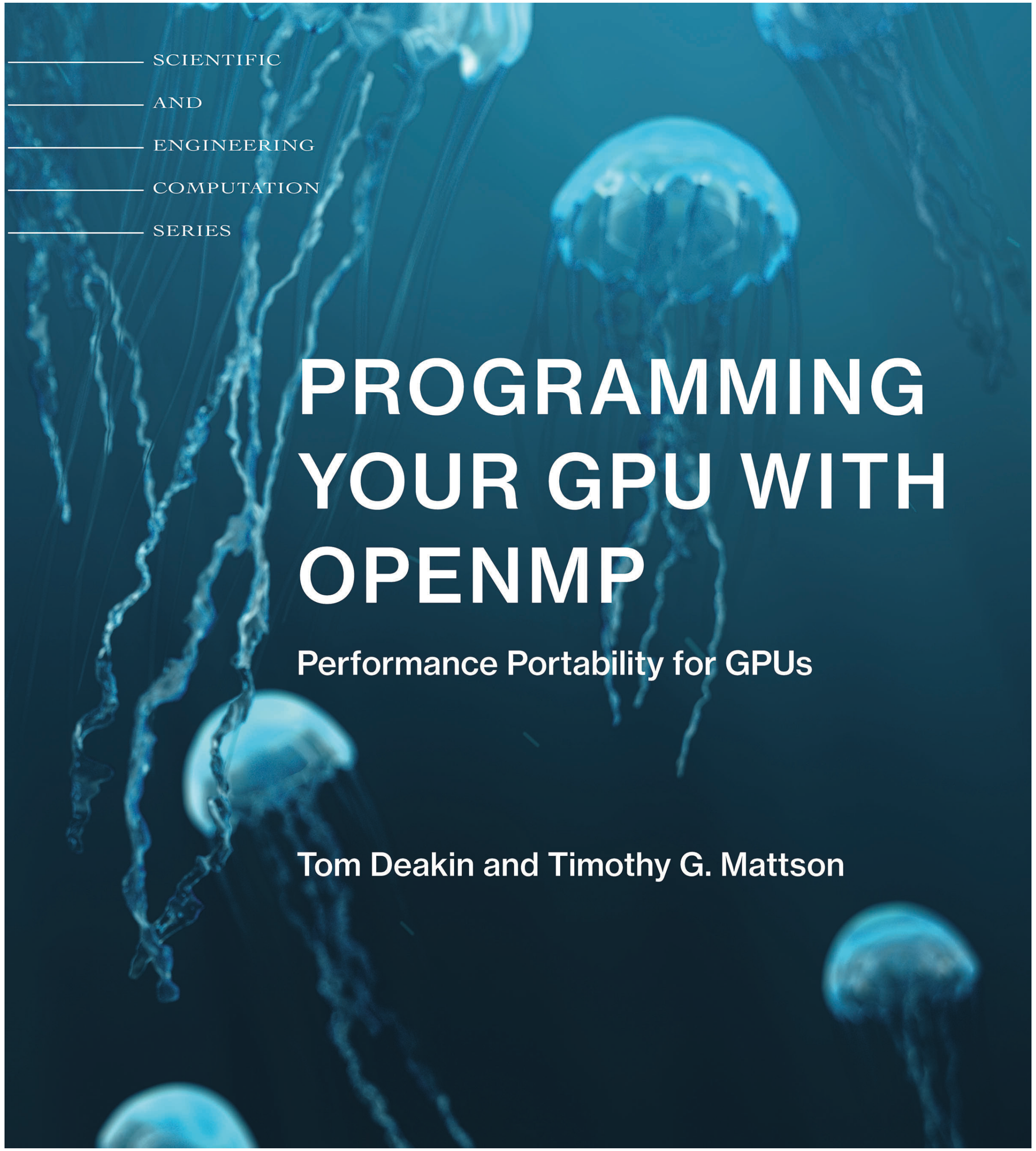
A great book that covers OpenMP features beyond OpenMP 2.5

Books about OpenMP

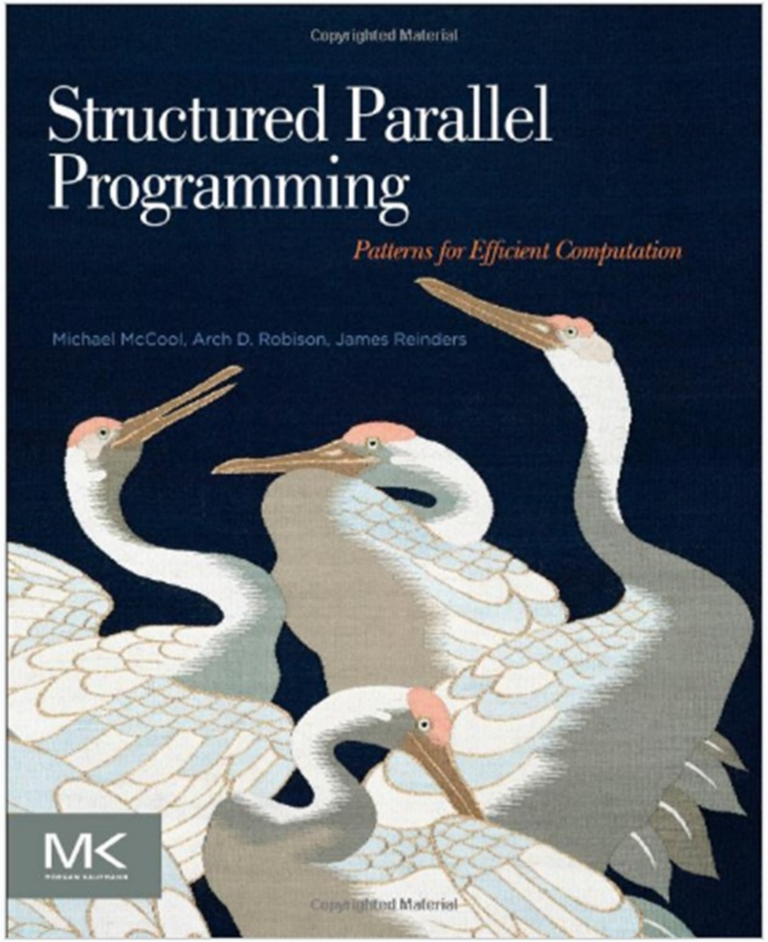
The latest book on OpenMP …

Released in November 2023.

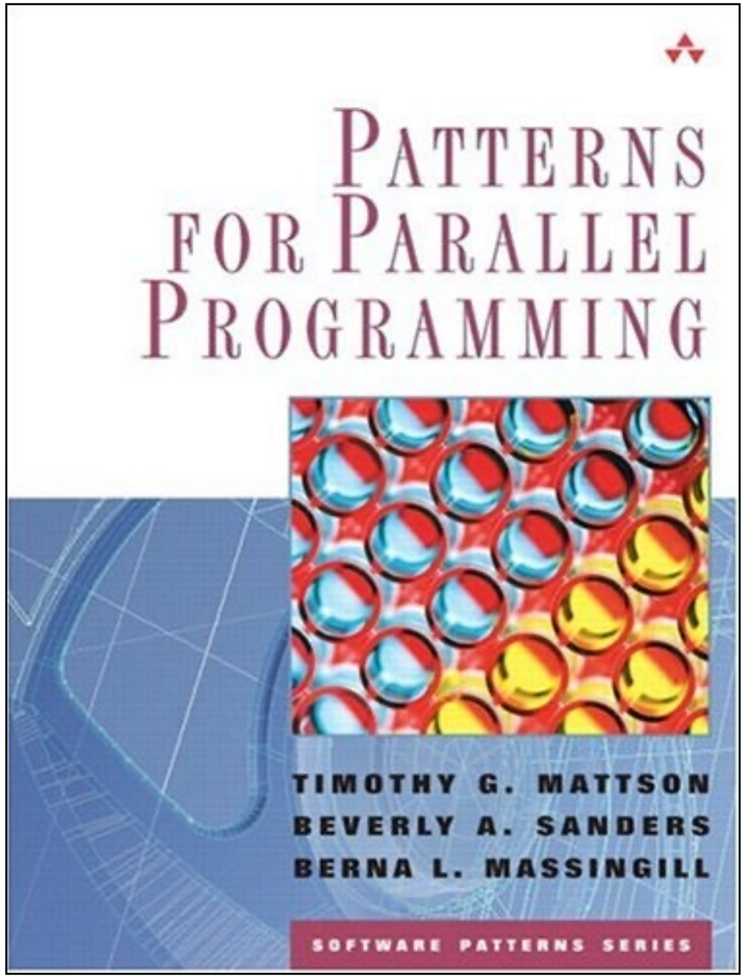
A book about how to use OpenMP to program a GPU.



Background references



A great book that explores key patterns with Cilk, TBB, OpenCL, and OpenMP (by McCool, Robison, and Reinders)



- A book about how to “think parallel” with examples in OpenMP, MPI and java