An Introduction to Parallel Programming with OpenMP

Tim Mattson

An Introduction to me

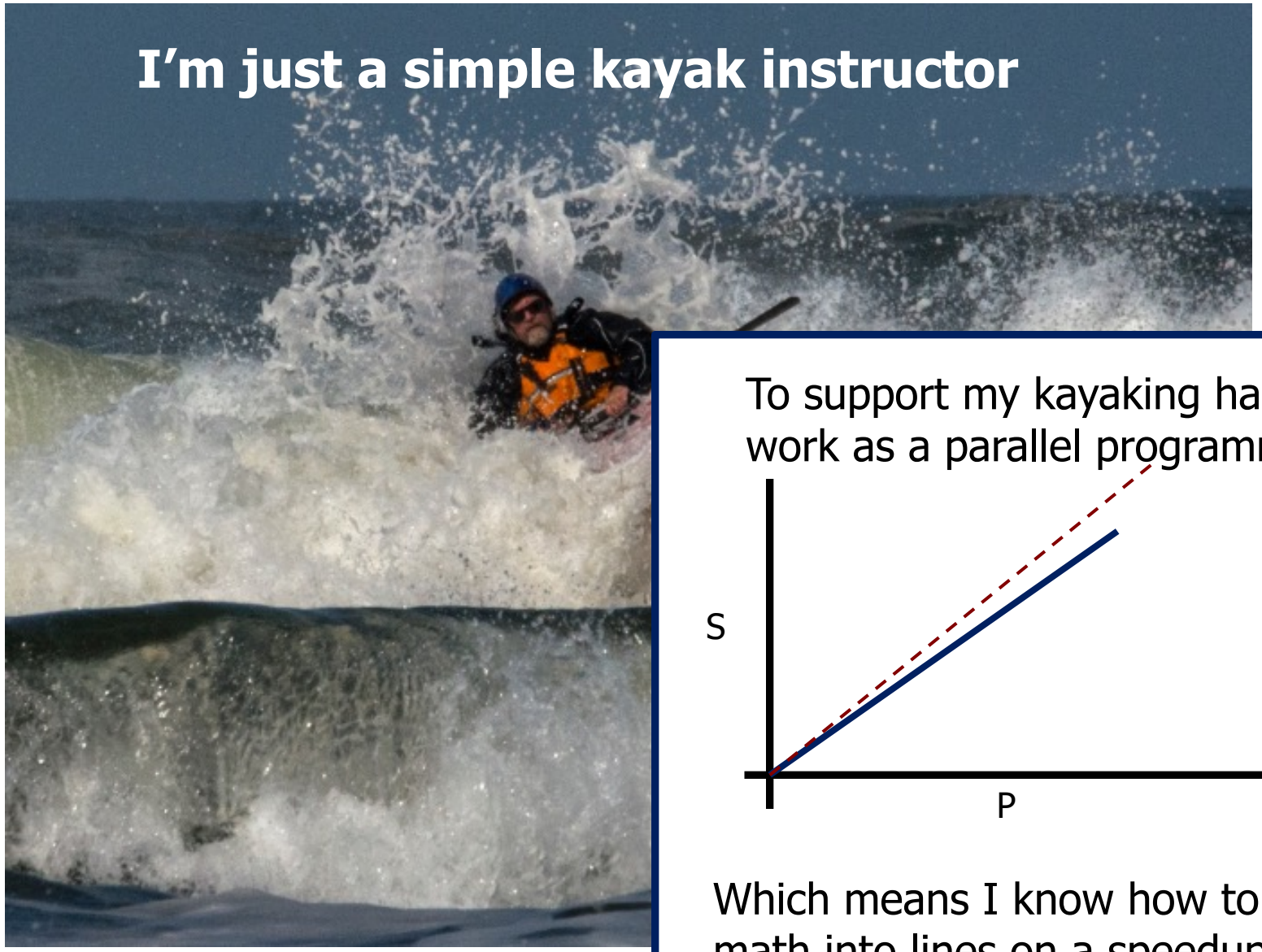
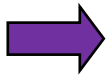


Photo © by Greg Clopton, 2014

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

Outline



- Introduction to OpenMP
- Creating Threads
- Synchronization
- Parallel Loops
- Data Environment
- Irregular Parallelism and Tasks
- NUMA systems and GPUs
- 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)

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

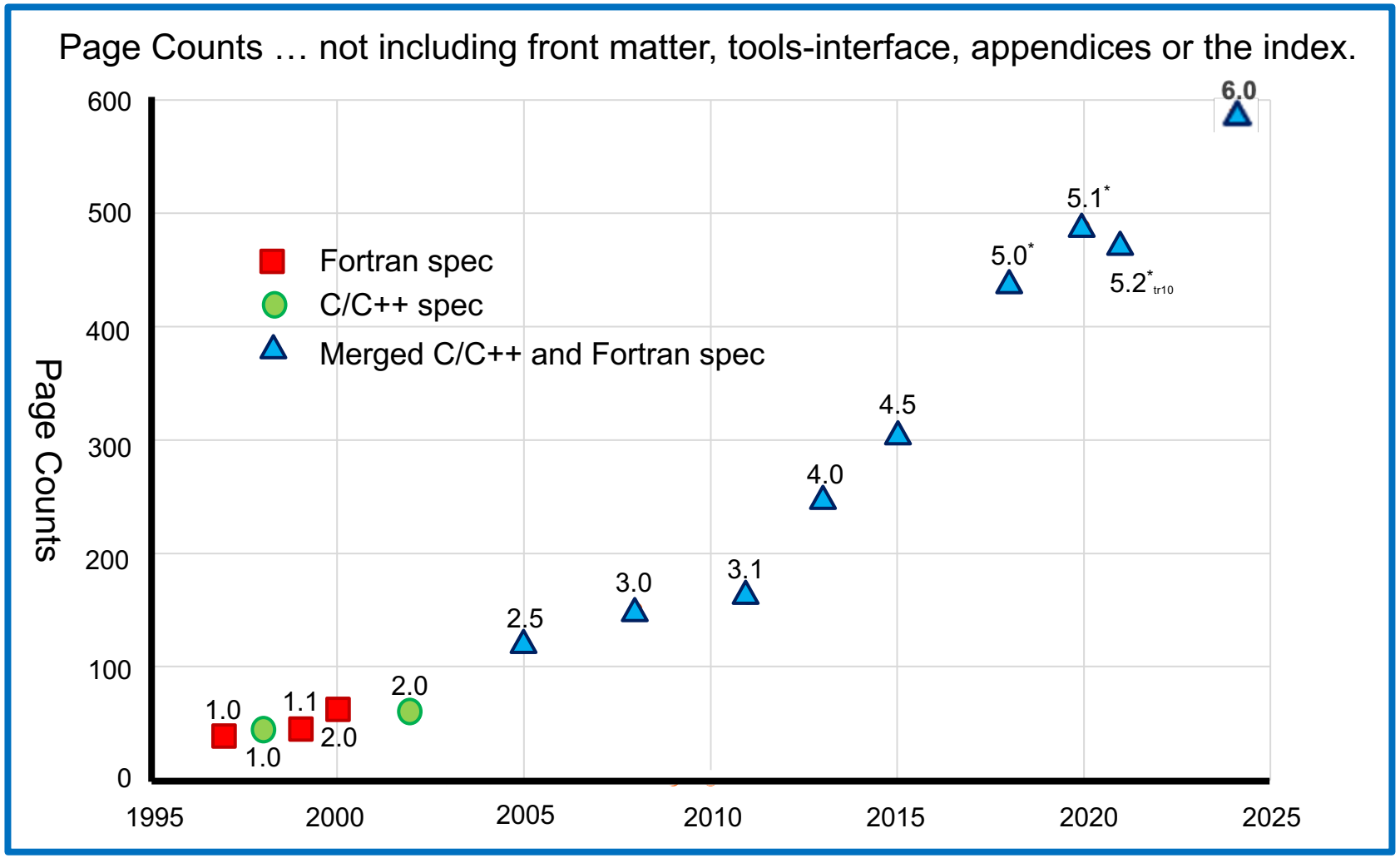
Nthrds = OMP_GET_NUM_PROCS()

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

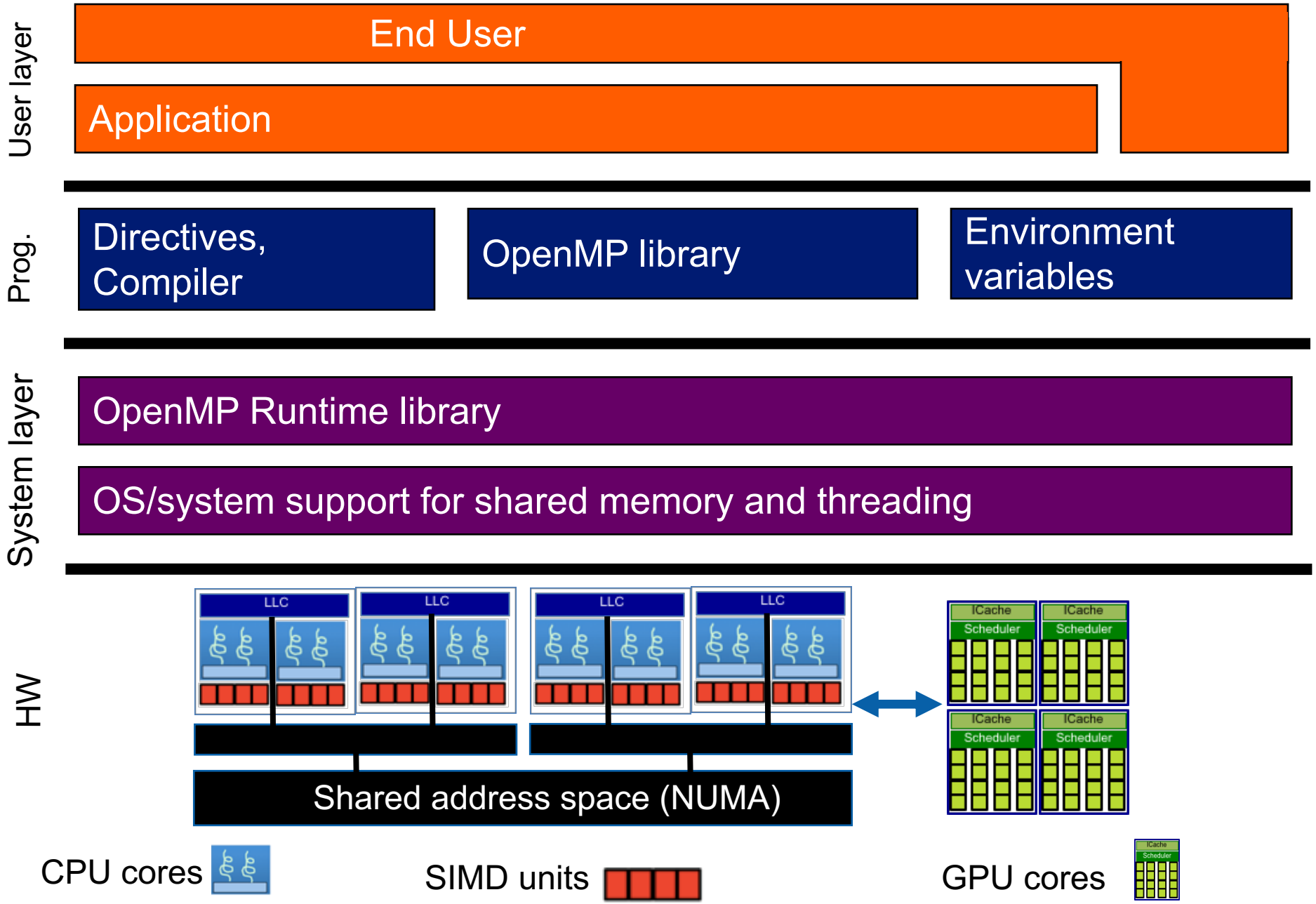


The OpenMP specification is so long and complex that few (if any) humans understand the full document

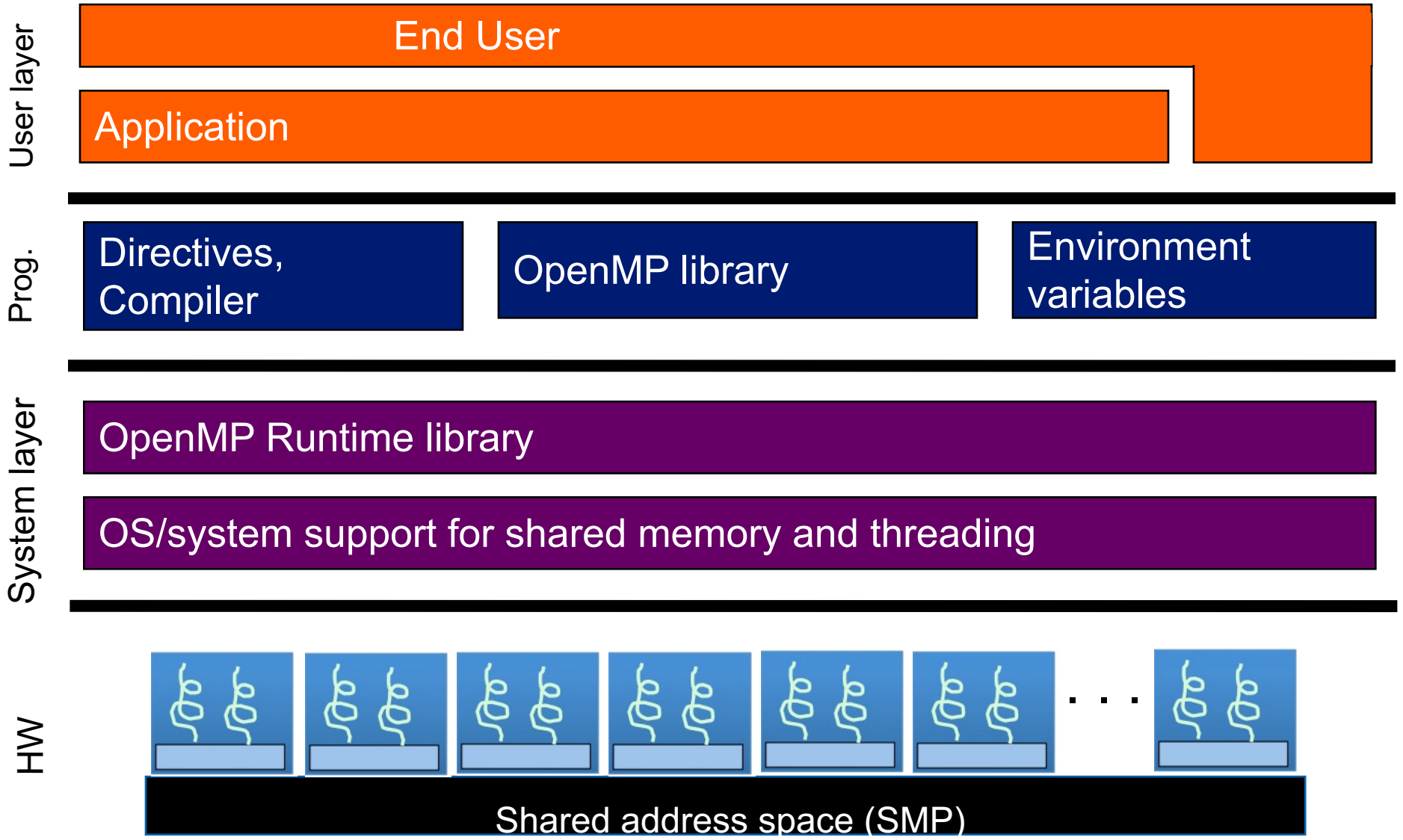
The OpenMP Common Core: Most OpenMP programs only use these 21 items

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”);

}
```

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

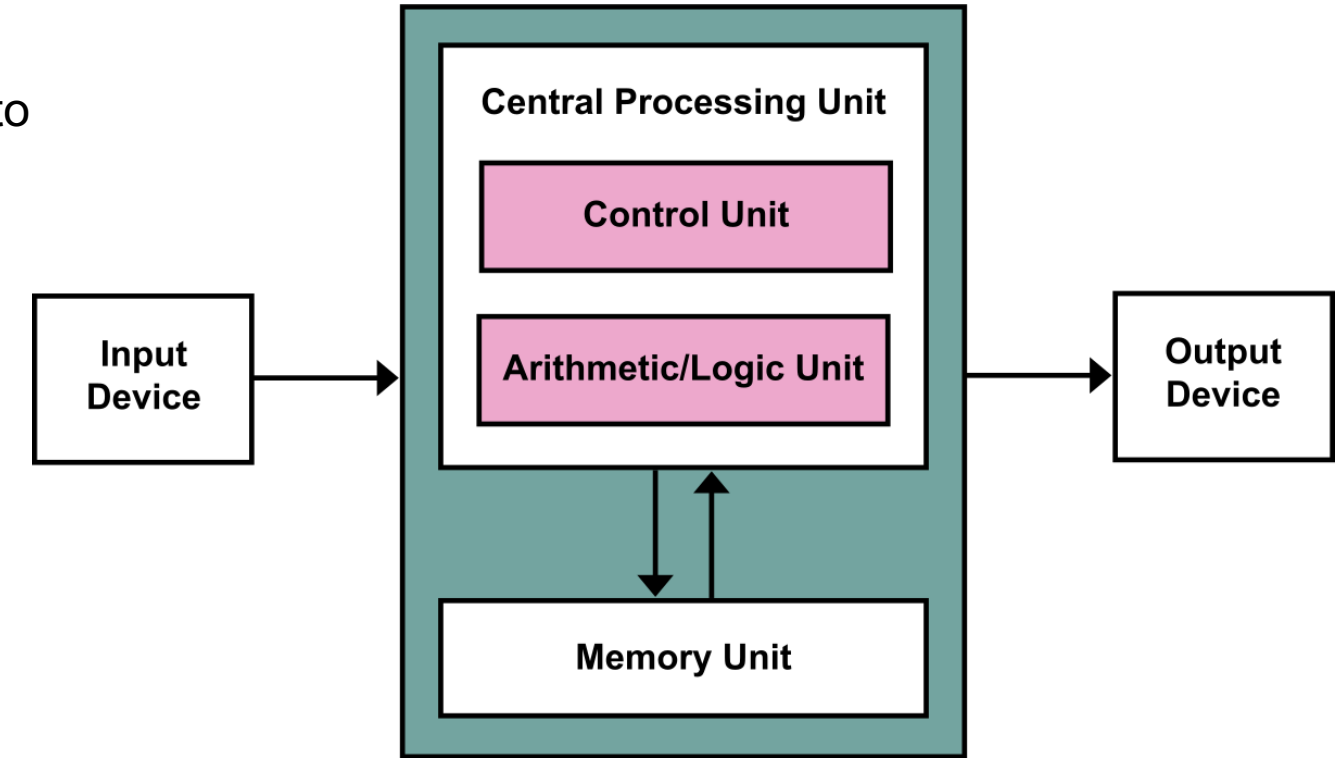
world

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).



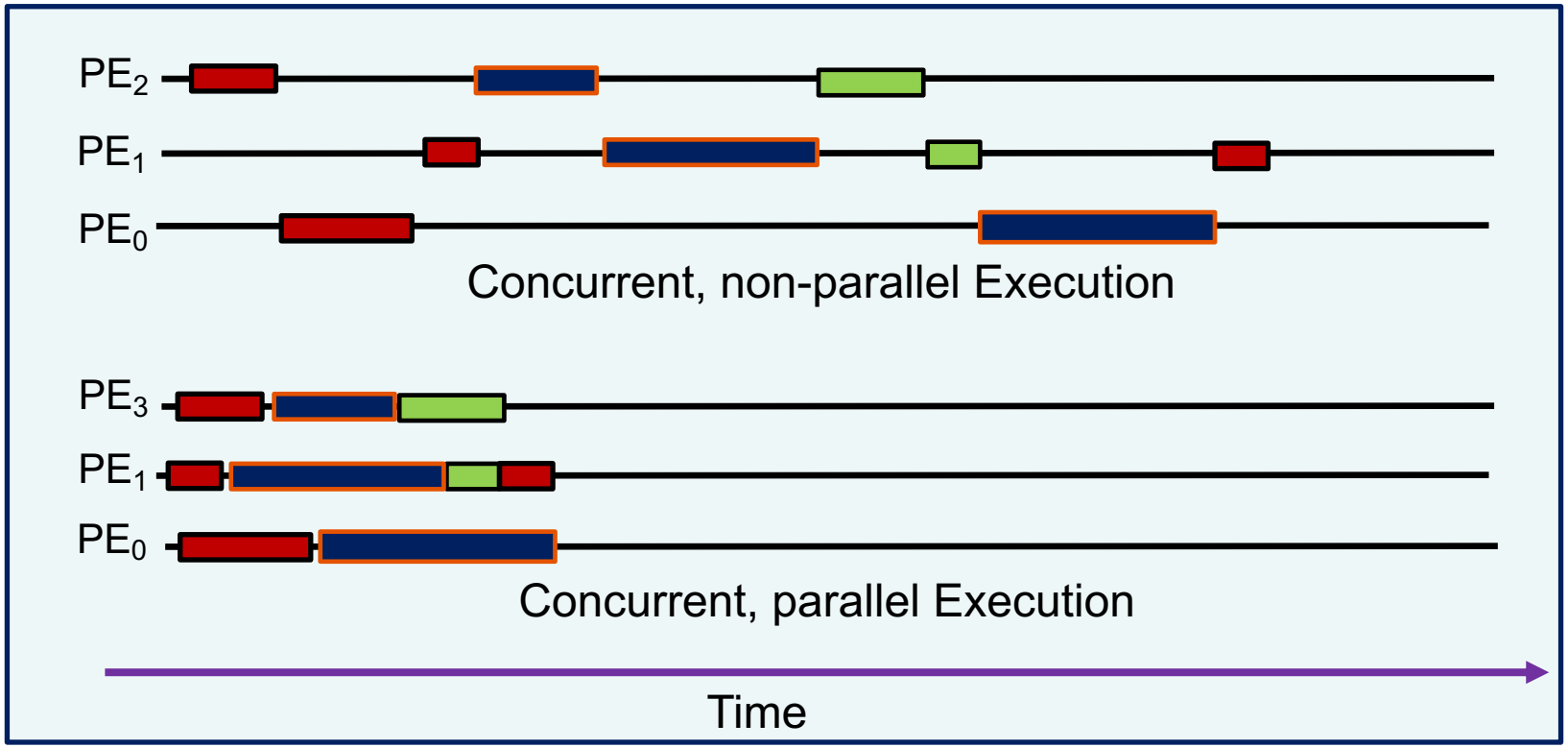
- **Task:**
  - A sequence of instructions plus a data environment. A program is composed of one or more tasks.

- **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.

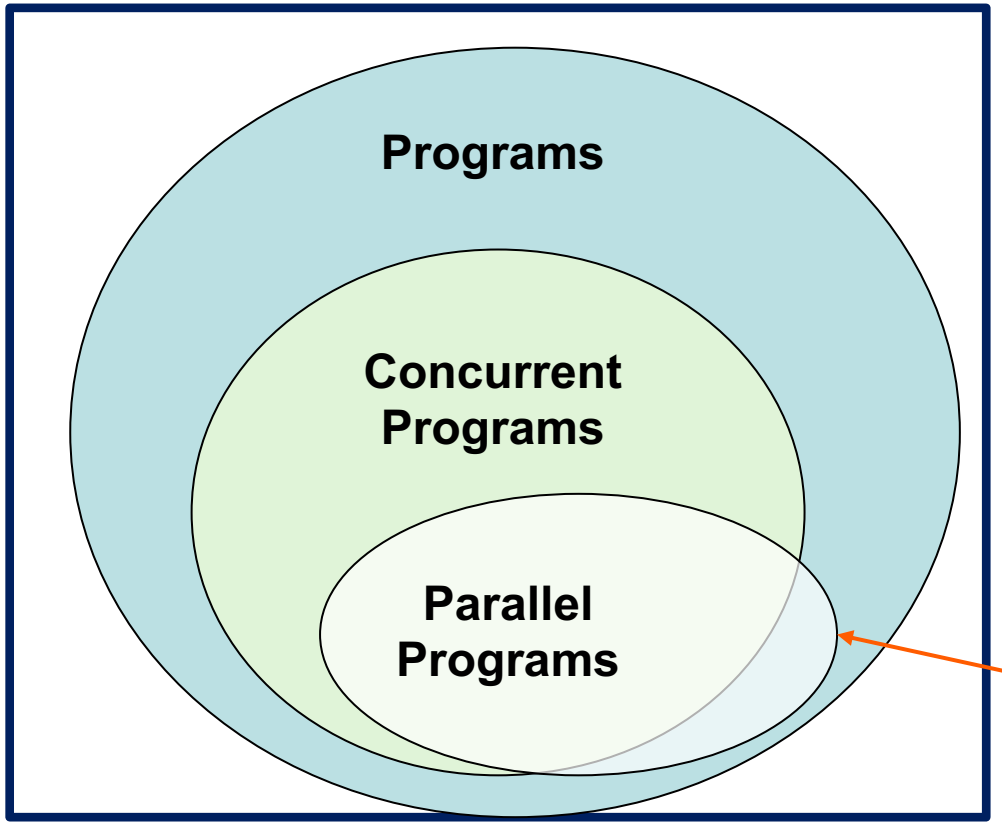


PE = Processing Element

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:
  - 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.



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

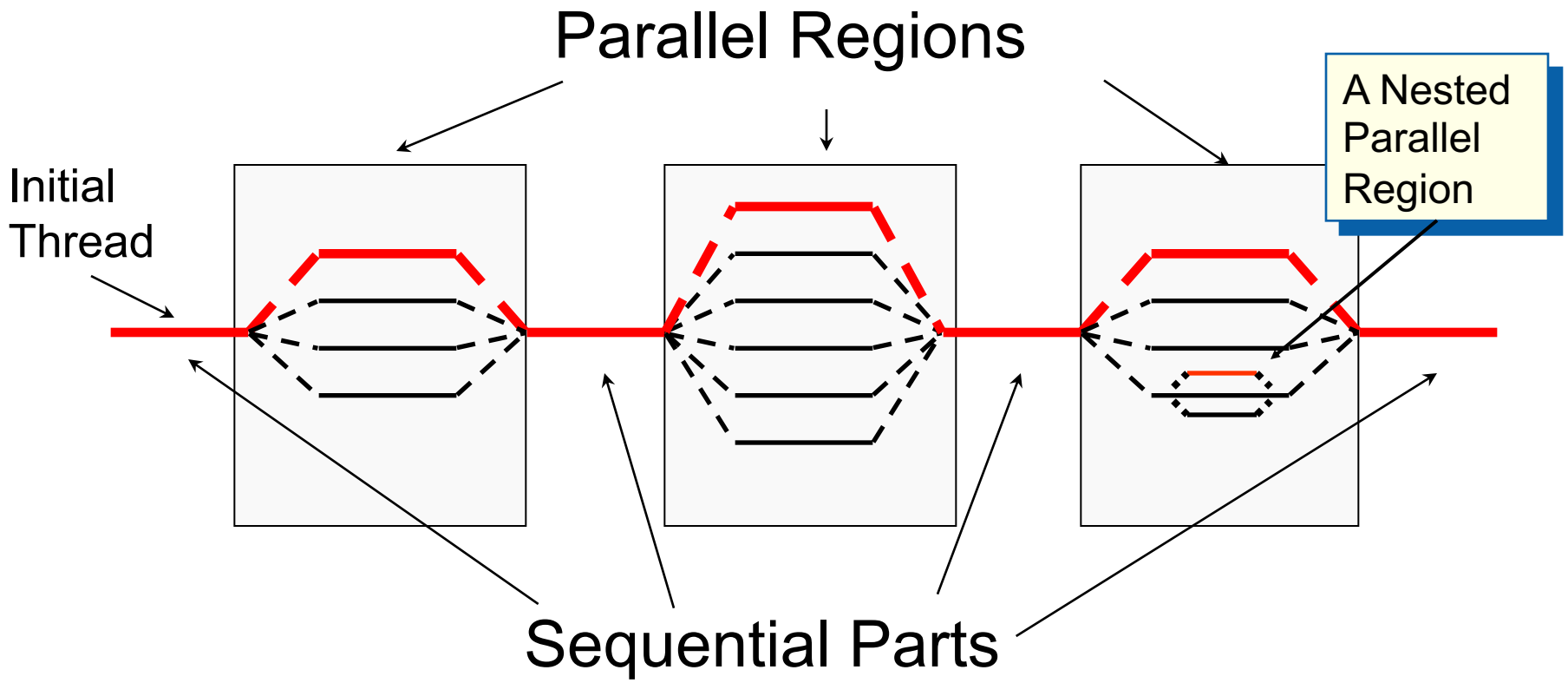
Outline

- Introduction to OpenMP
- Creating Threads
- Synchronization
- Parallel Loops
- Data Environment
- Irregular Parallelism and Tasks
- NUMA systems and GPUs
- Recap

OpenMP Execution model:

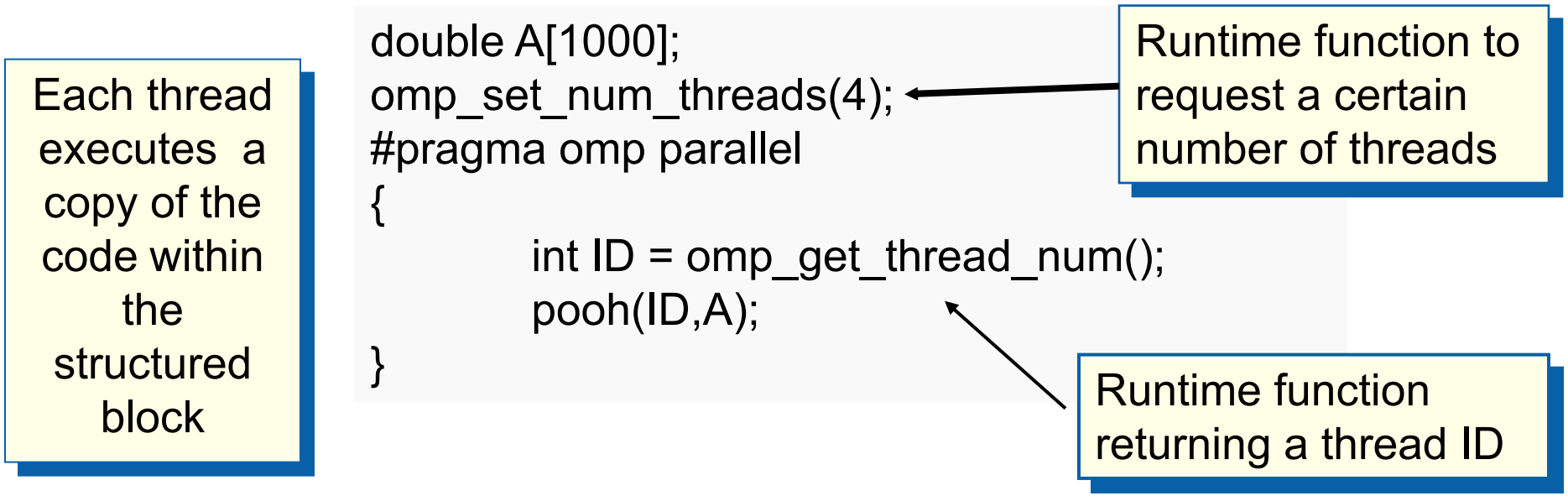
Fork-Join Parallelism:

- 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:

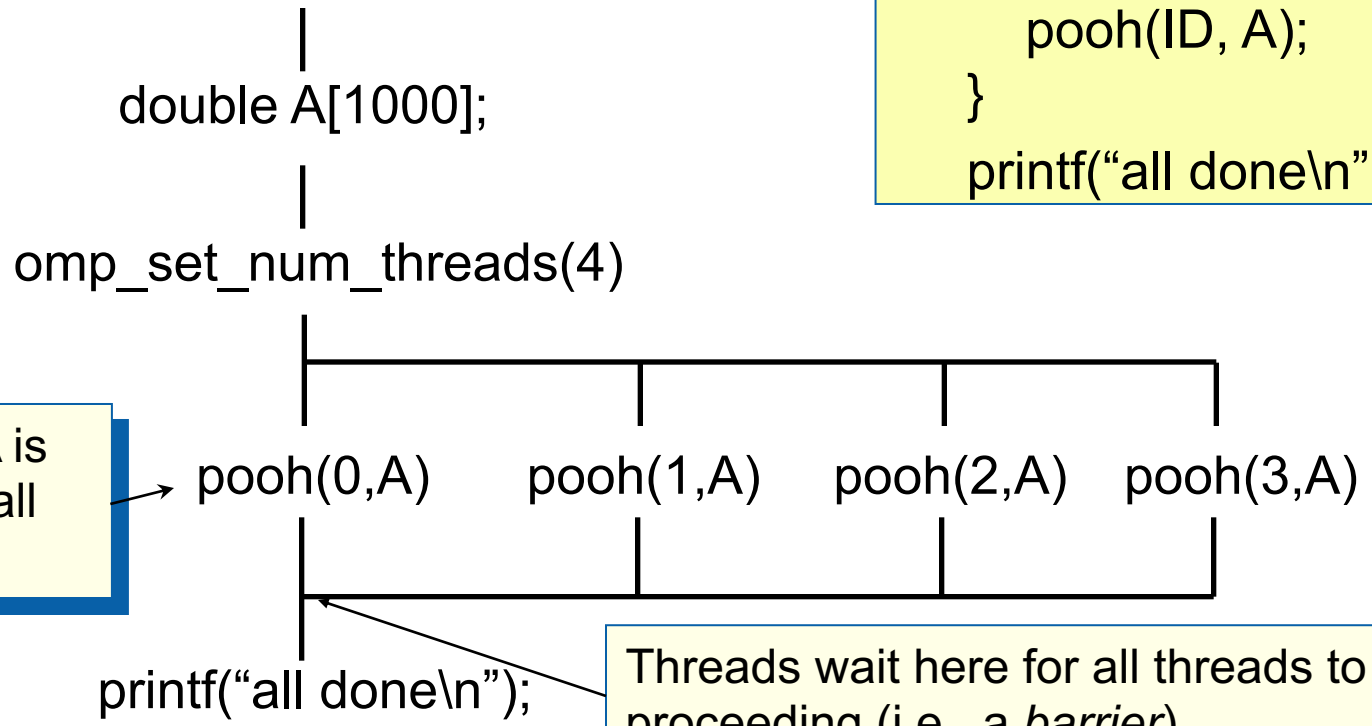


- Each thread calls pooh(ID,A) for ID = 0 to 3

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");
```



A single copy of A is shared between all threads.

Threads wait here for all threads to finish before proceeding (i.e., a *barrier*)

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);
}
```

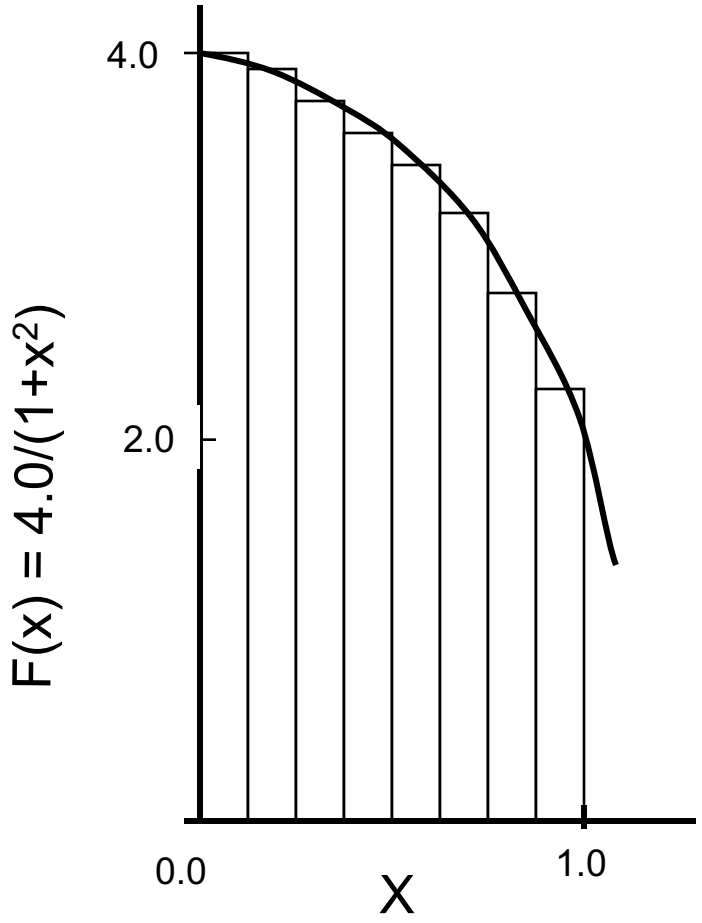
Runtime function to request a certain number of threads

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();
  - int omp_get_thread_num();
  - double omp_get_wtime();
  - omp_set_num_threads();

Number of threads in the team

Thread ID or rank

Time in seconds since a fixed point in the past

Request a number of threads in the team

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

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.

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

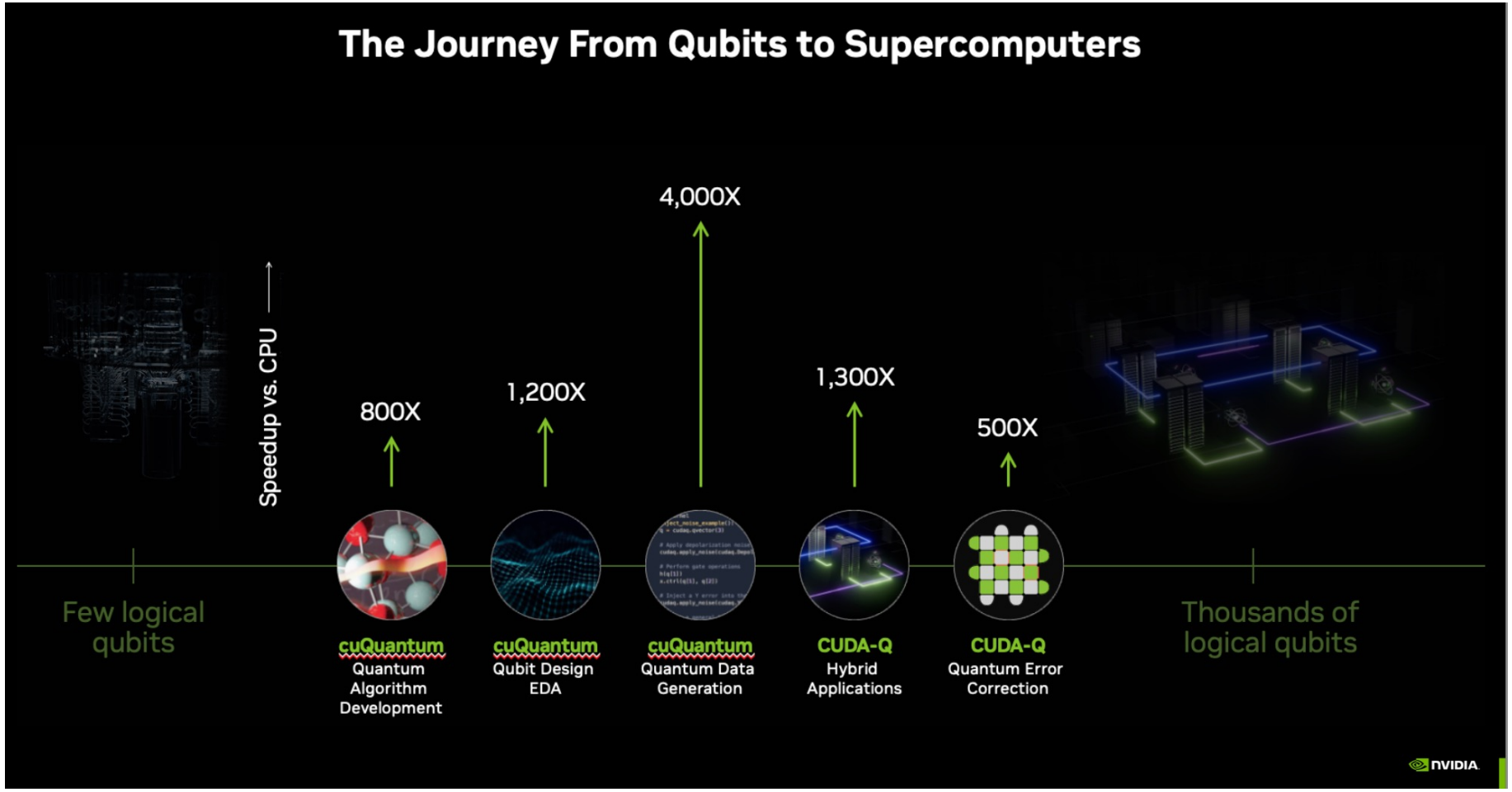
- Perfect Linear Speedup: happens when no parallel overhead and algorithm is 100% parallel.

$$S(P) = P$$

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

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

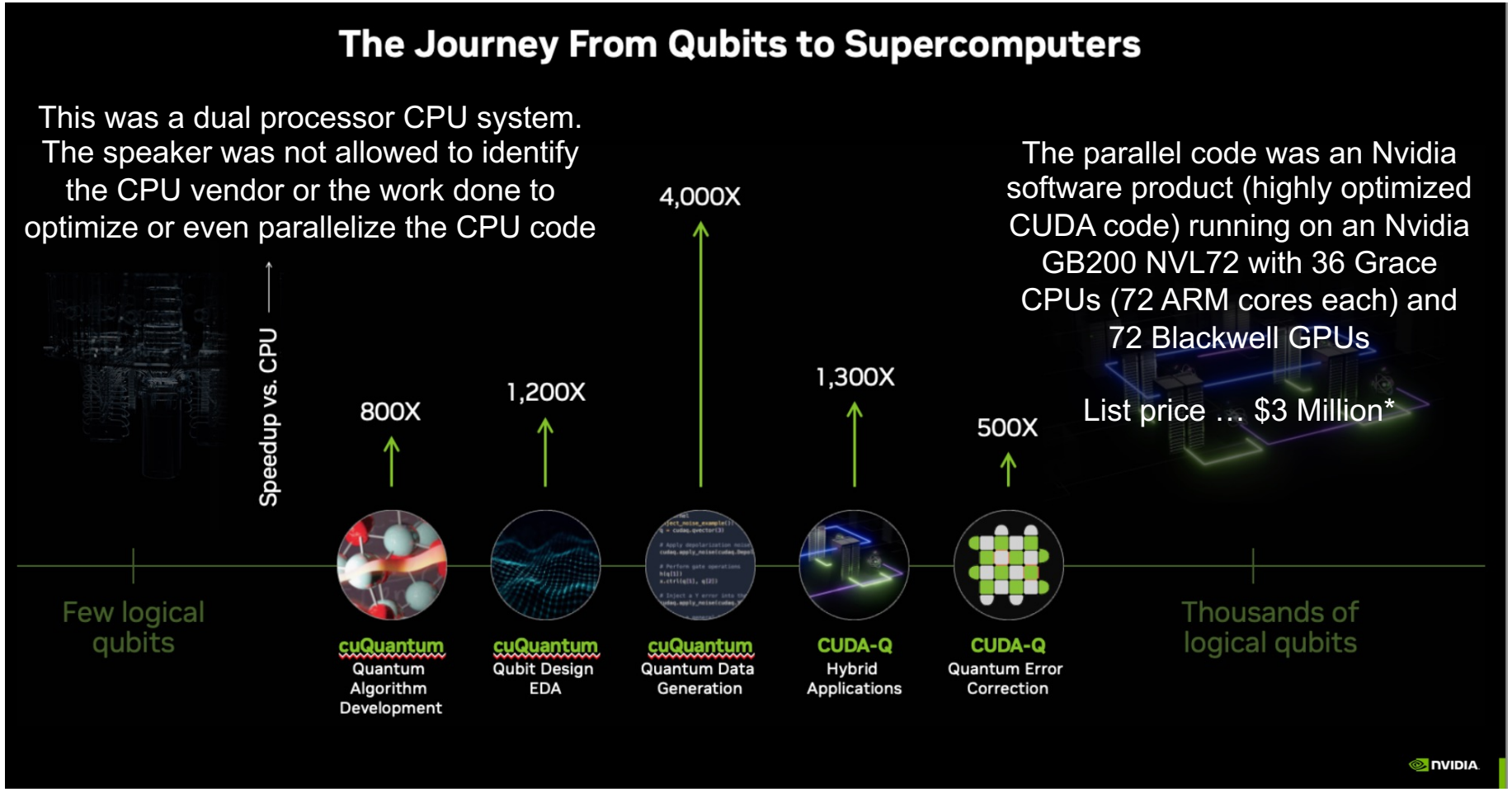
Speedups as disinformation



A slide from an Nvidia talk at ATPESC'25 about their quantum computing product

Reported speedups without defining what they were comparing against … included a verbal comment that this showed if you really care about performance, you must use a GPU

Speedups as disinformation



These Speedups are blatant disinformation… it's demeaning to show such data.

The sad thing is, the Nvidia product is **excellent**. In a fair comparison, they'd still come out on top. Why they need to resort to such misleading statements is baffling.

*https://www.datacenterdynamics.com/en/news/lambda-partners-with-pegatron-to-deploy-nvidia-gb200-nvl72-rack/

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:

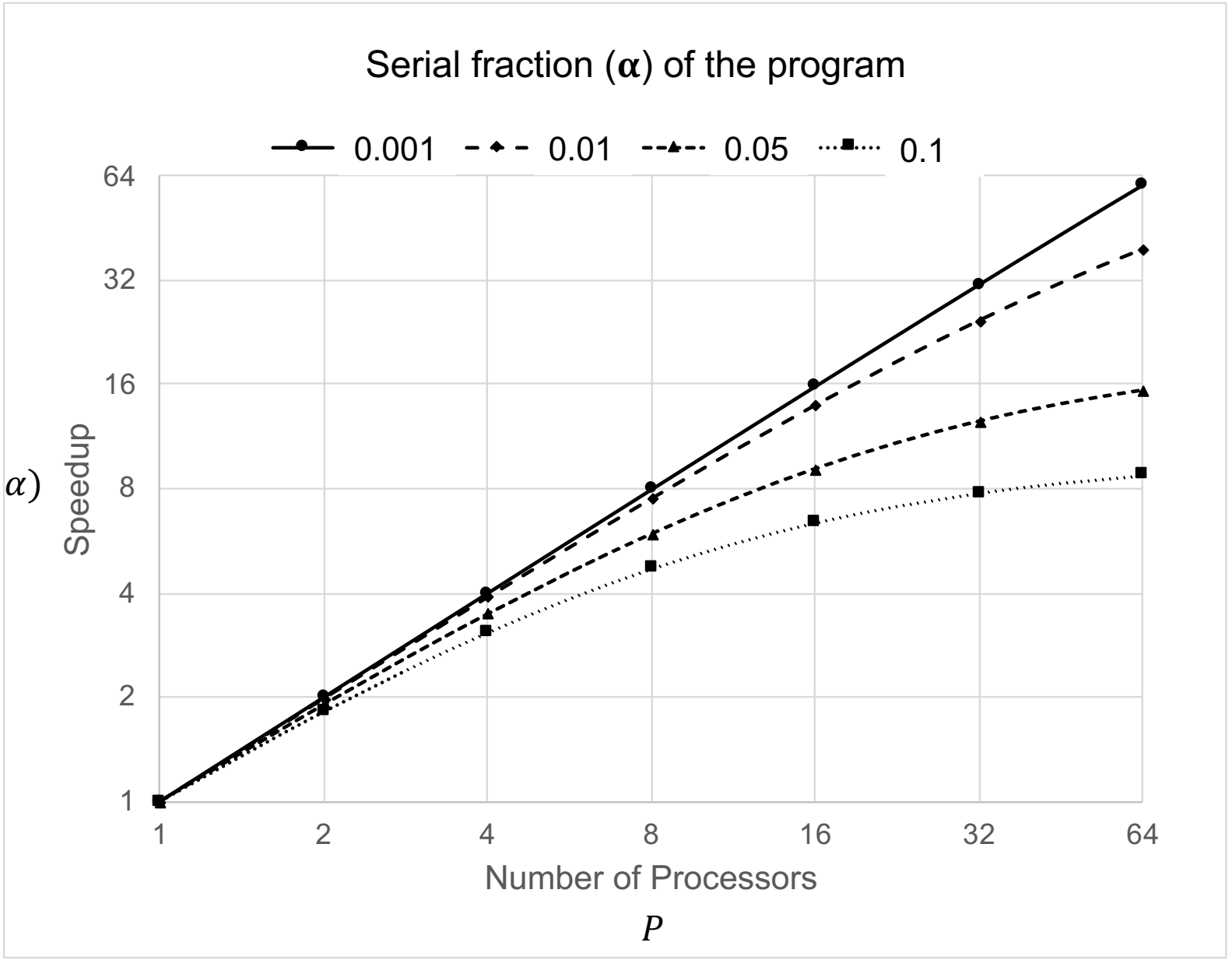
$$S(P) = \frac{Time_{seq}}{Time_{par}(P)} = \frac{Time_{seq}}{(\alpha + \frac{1-\alpha}{P}) * Time_{seq}} = \frac{1}{\alpha + \frac{1-\alpha}{P}}$$

$$S(P,\alpha) = \frac{1}{\alpha - \frac{1-\alpha}{P}}$$

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

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

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



$S(P, \alpha)$

$$S(P, \alpha) = \frac{1}{\alpha - \frac{1 - \alpha}{P}}$$

So now you should understand my silly introduction slide.

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

Introduction

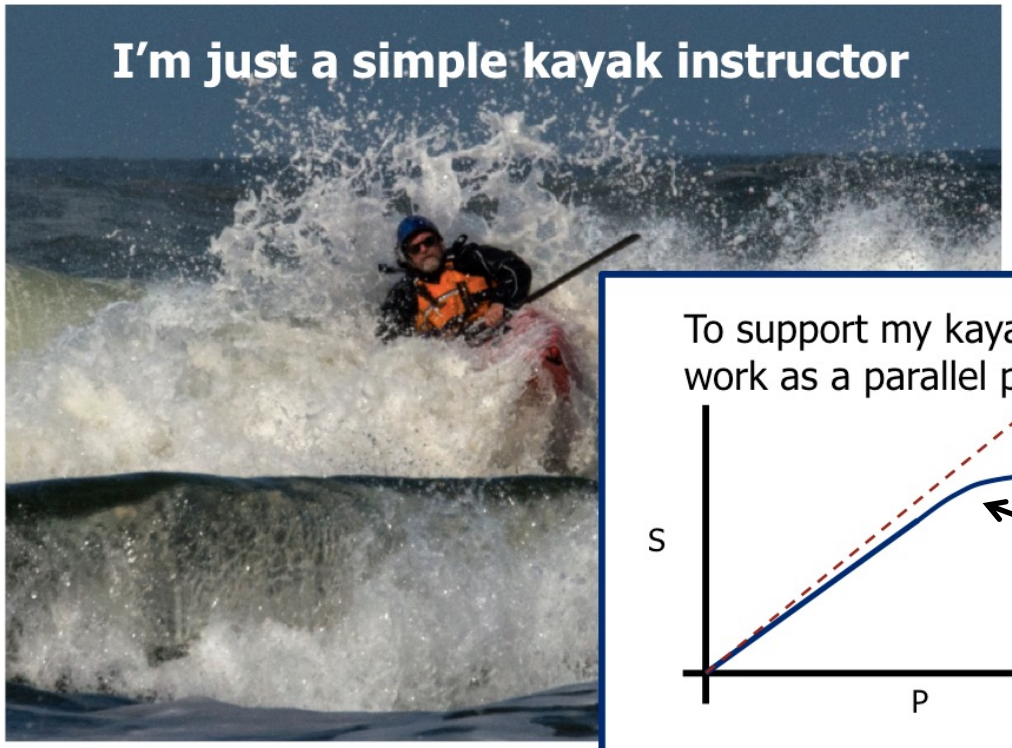


Photo © by Greg Clopton, 2014

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

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

4

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)

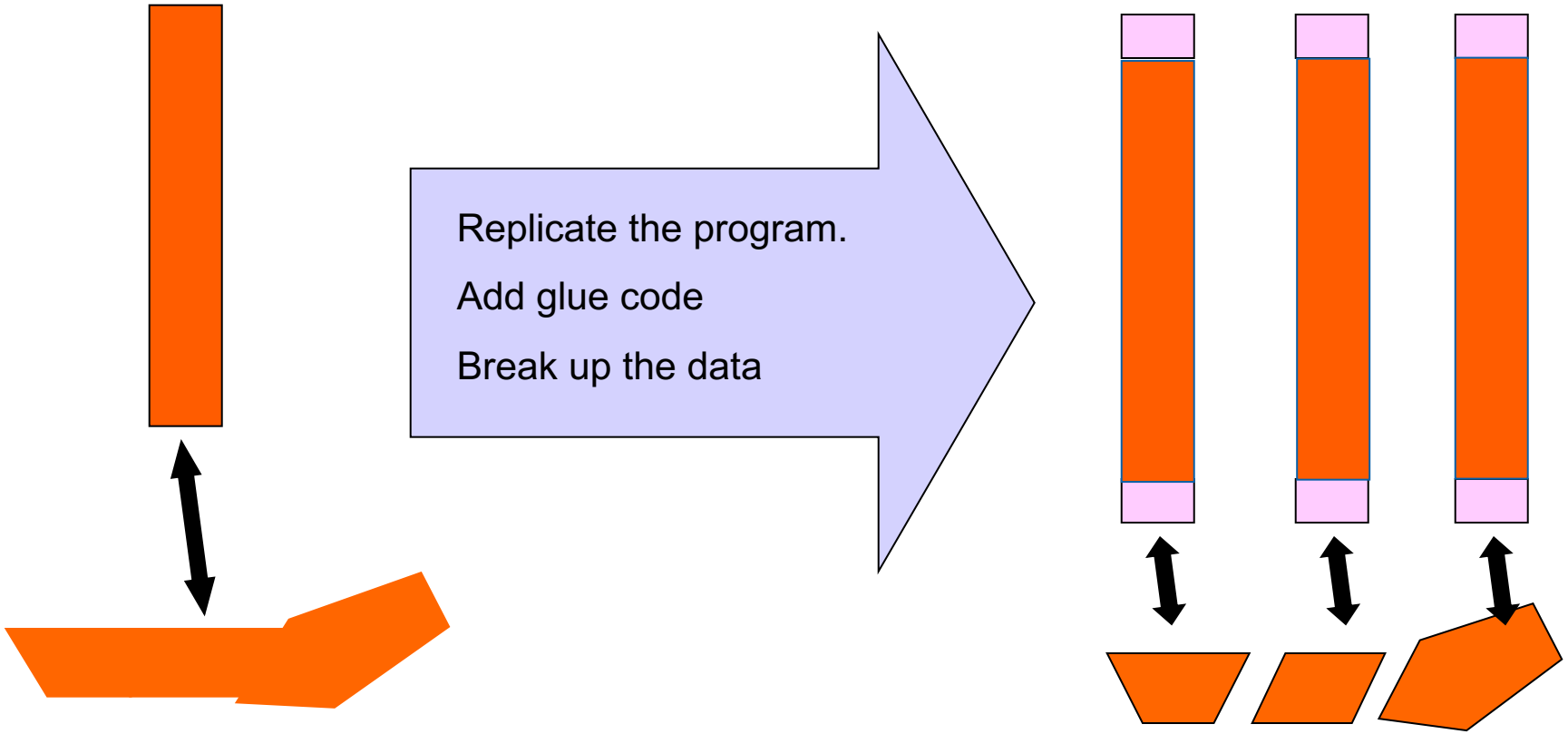
- What does omp_set_num_threads() actually do?
  - It **resets** an "**internal control variable**" the system queries to select the default number of threads to request on subsequent parallel constructs.

- 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**

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.

Outline

- Introduction to OpenMP
- Creating Threads
- Synchronization
- Parallel Loops
- Data Environment
- Irregular Parallelism and Tasks
- NUMA systems and GPUs
- 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.

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

```
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);

}
```

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.

You will learn more about this important concept in the lecture on memory

- If array elements happen to share a cache line, this leads to false sharing.
  - Non-shared data in the same cache line so updates invalidate 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™ 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	1st SPMD	SPMD critical
1	1.86	1.87
2	1.03	1.00
3	1.08	0.68
4	0.97	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.

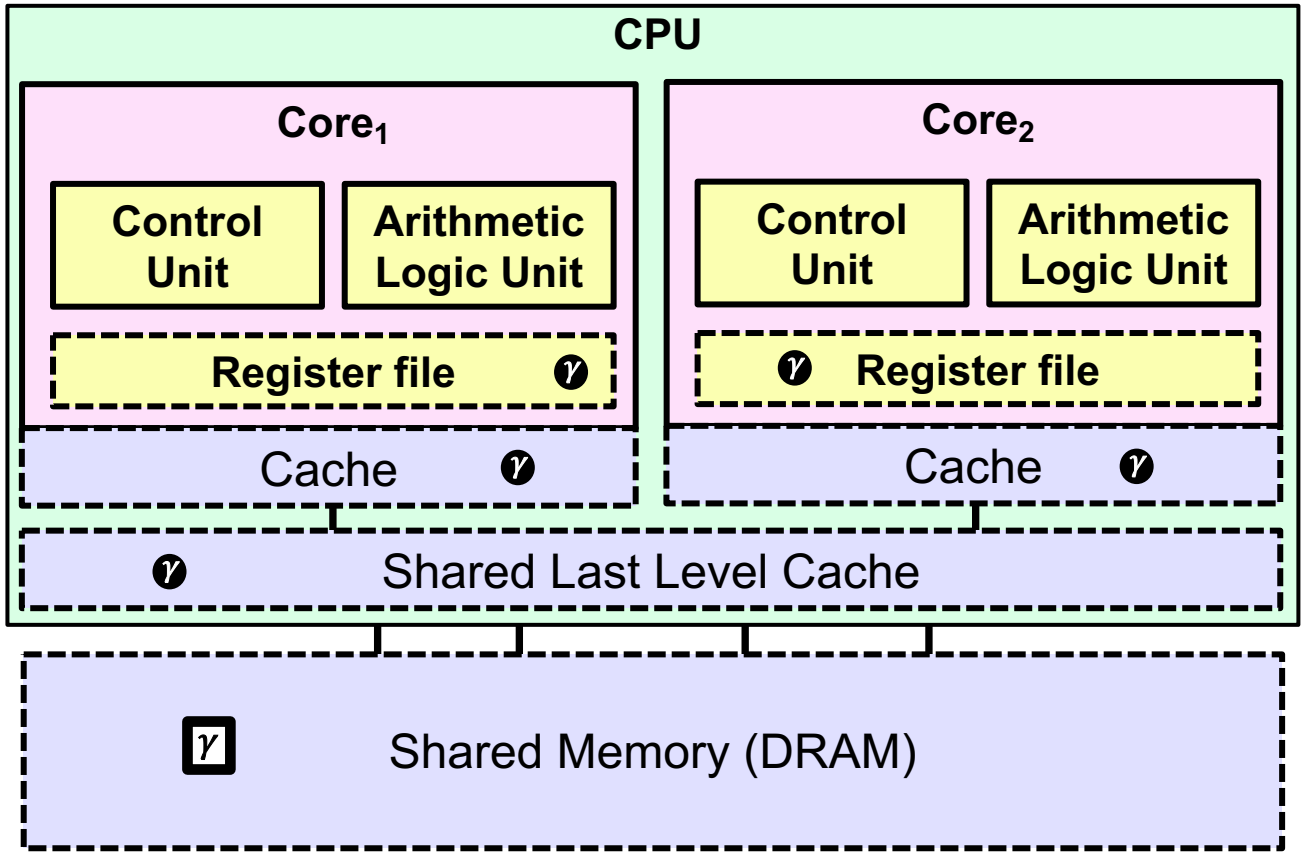
Example: Using a critical section to remove impact of false sharing

```
#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?

Memory Models …

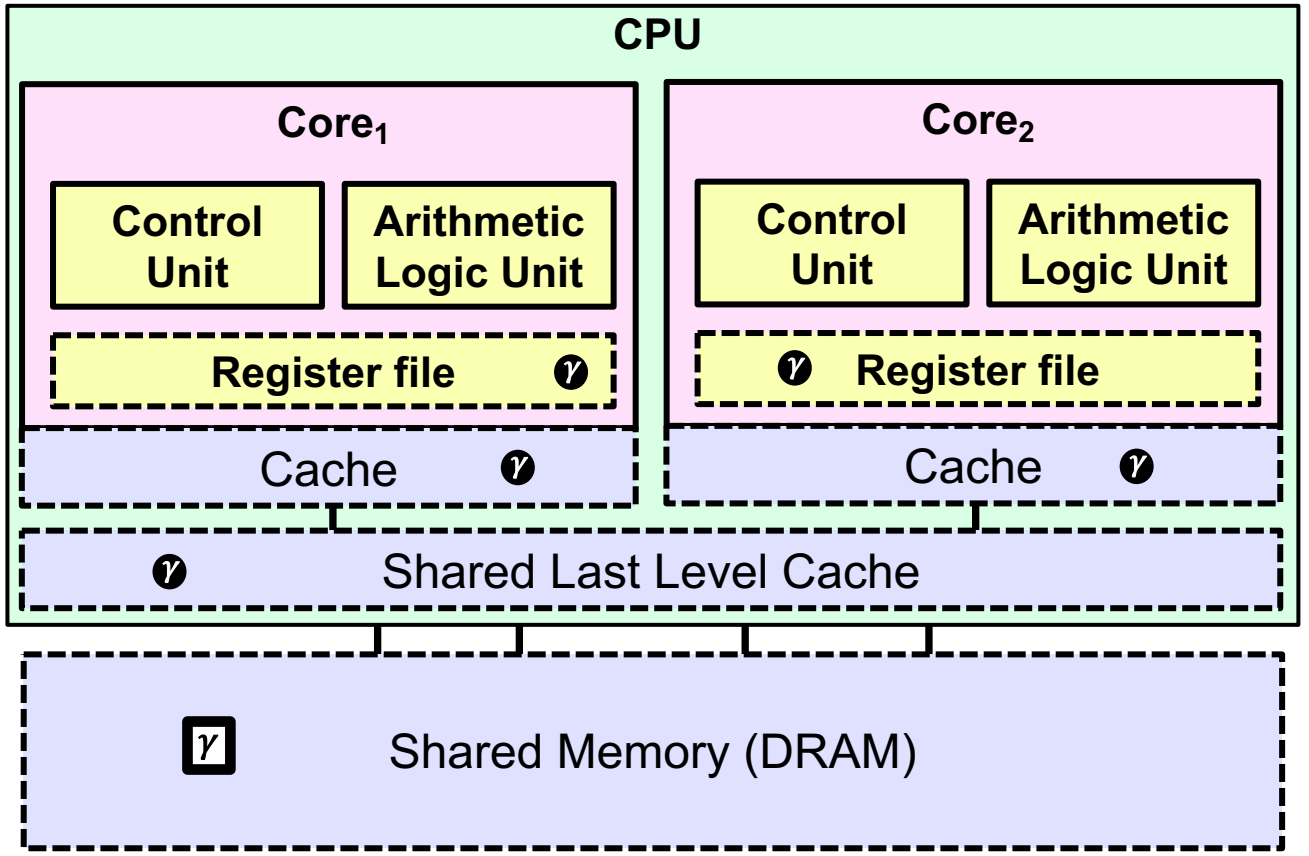
- A shared address space is a region of memory visible to the team of threads … multiple threads can read and write variables in the shared address space.
- Multiple copies of a variable (such as γ) may be present in memory, at various levels of cache, or in registers and they may ALL have different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

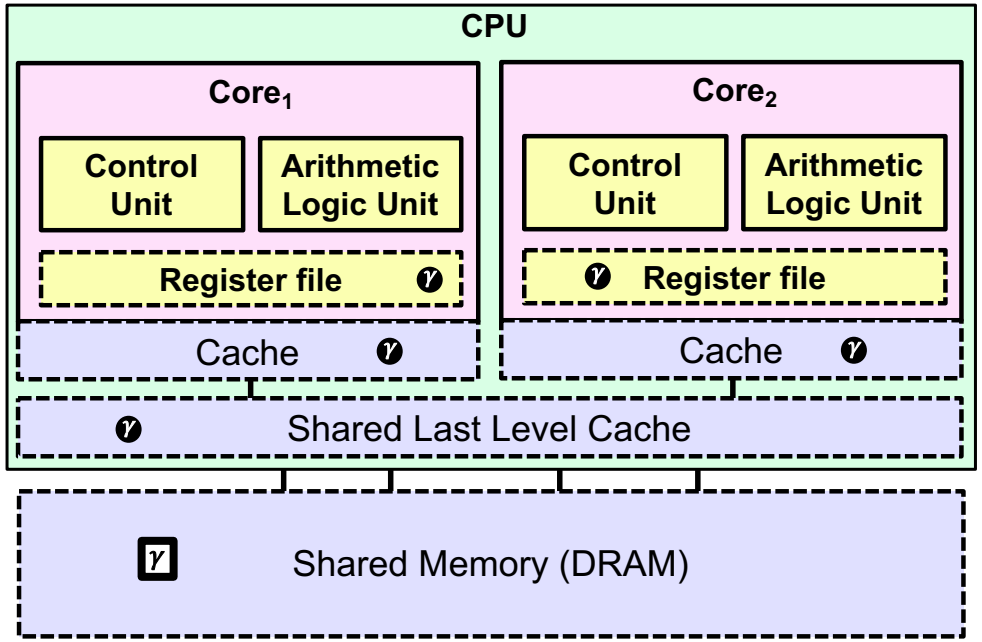
Memory Models …

- 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**

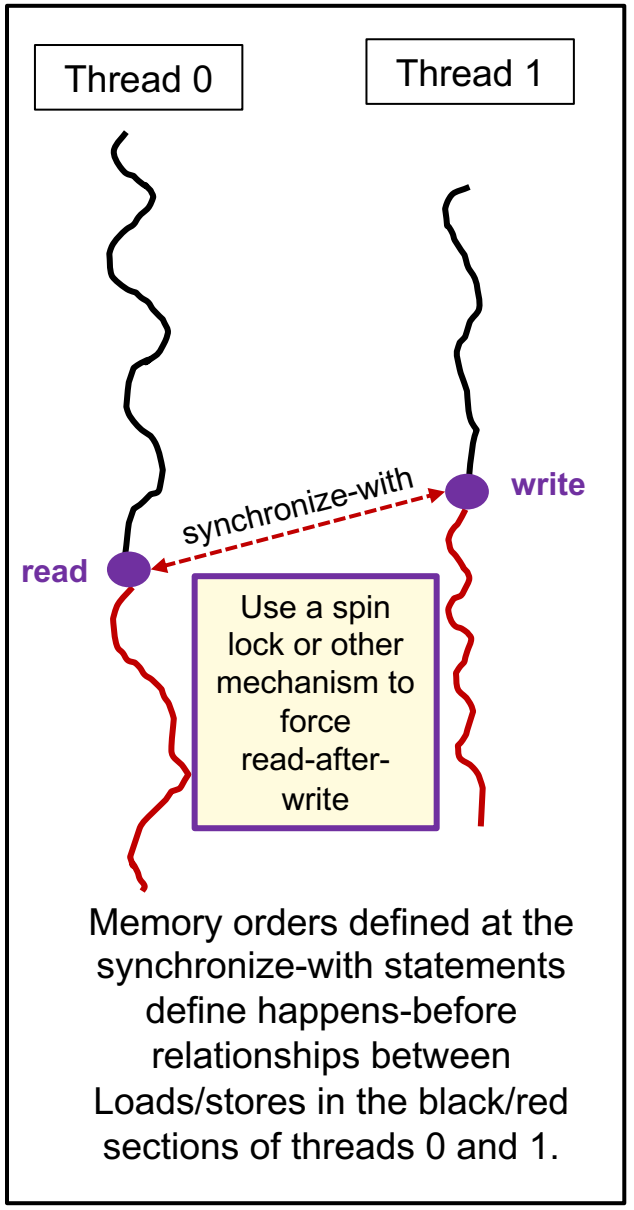
Most (if not all) multithreading programming models assume **relaxed consistency**. Maintaining sequential consistency across the full program-execution adds too much synchronization overhead.

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.



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**"

Keep it simple … let OpenMP take care of Flushes for you

- A flush operation is implied by OpenMP constructs …
  - at entry/exit of parallel regions
  - at implicit and explicit barriers
  - at entry/exit of critical regions

This has not been a detailed discussion of the full OpenMP memory model. The goal was to explain how memory models work and to understand the subset of features people commonly use.

- OpenMP programs that:
  - Do not use non-sequentially consistent atomic constructs;
  - Do not rely on the accuracy of a false result from omp_test_lock and omp_test_nest_lock; and
  - Correctly avoid data races

… behave as though operations on shared variables were simply interleaved in an order consistent with the order in which they are performed by each thread. The relaxed consistency model is invisible for such programs, and any explicit flushes in such programs are redundant.

WARNING:
If you find yourself wanting to write code with explicit flushes, stop and get help. It is very difficult to manage flushes on your own. Even experts often get them wrong.

This is why we defined OpenMP constructs to automatically apply flushes most places where you really need them.

Outline

- Introduction to OpenMP
- Creating Threads
- Synchronization
- Parallel Loops
- Data Environment
- Irregular Parallelism and Tasks
- NUMA systems and GPUs
- 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

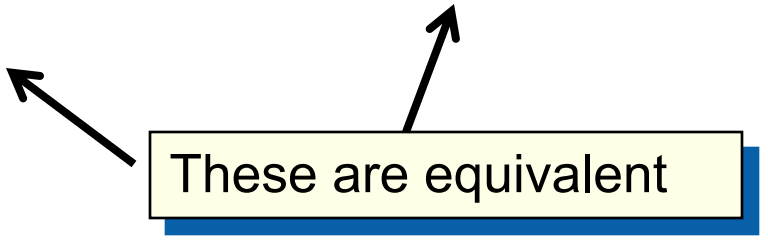
```
#pragma omp parallel
#pragma omp for
        for(i=0;i<N;i++)   { a[i] = a[i] + b[i];}
```

Combined Parallel/Worksharing Construct

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

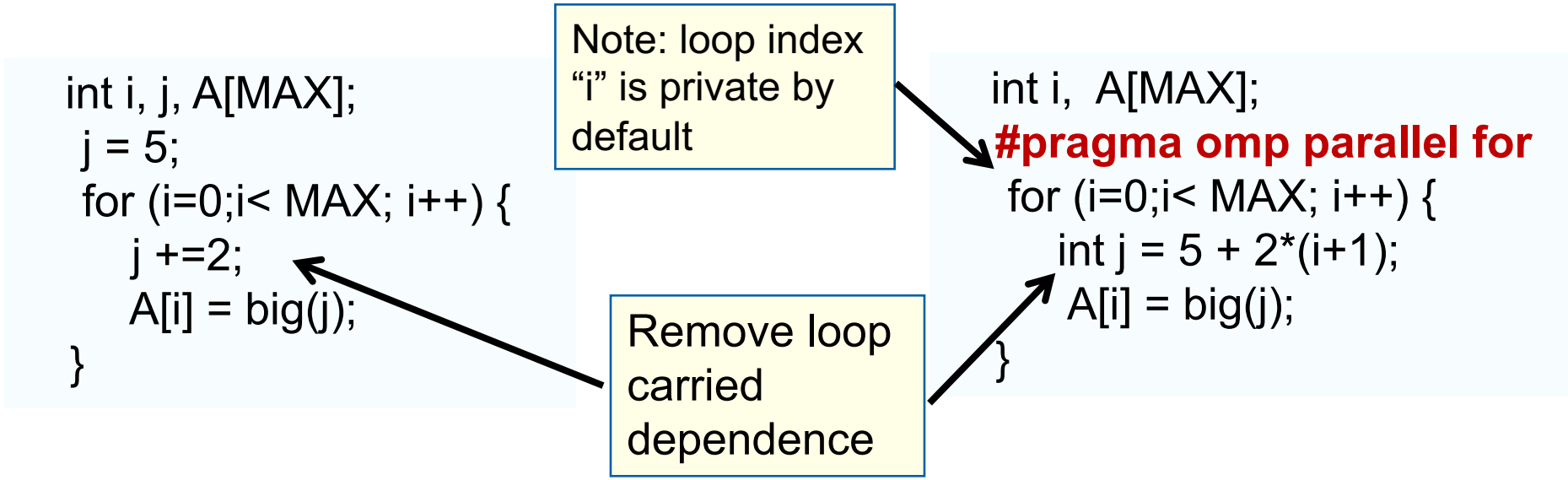
```
 double  res[MAX];  int i;
#pragma omp parallel
{
    #pragma omp for
    for (i=0;i< MAX; i++) {
        res[i] = huge();
    }
}
```

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



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



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>
void main ()
{
    long num_steps = 100000;
    double pi, sum = 0.0;
    double 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;
}
```

Results*: PI with a loop and a reduction

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

Example: PI with a loop and a reduction

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

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™ i5 processor at 1.7 Ghz and 4 Gbyte DDR3 memory at 1.333 Ghz.

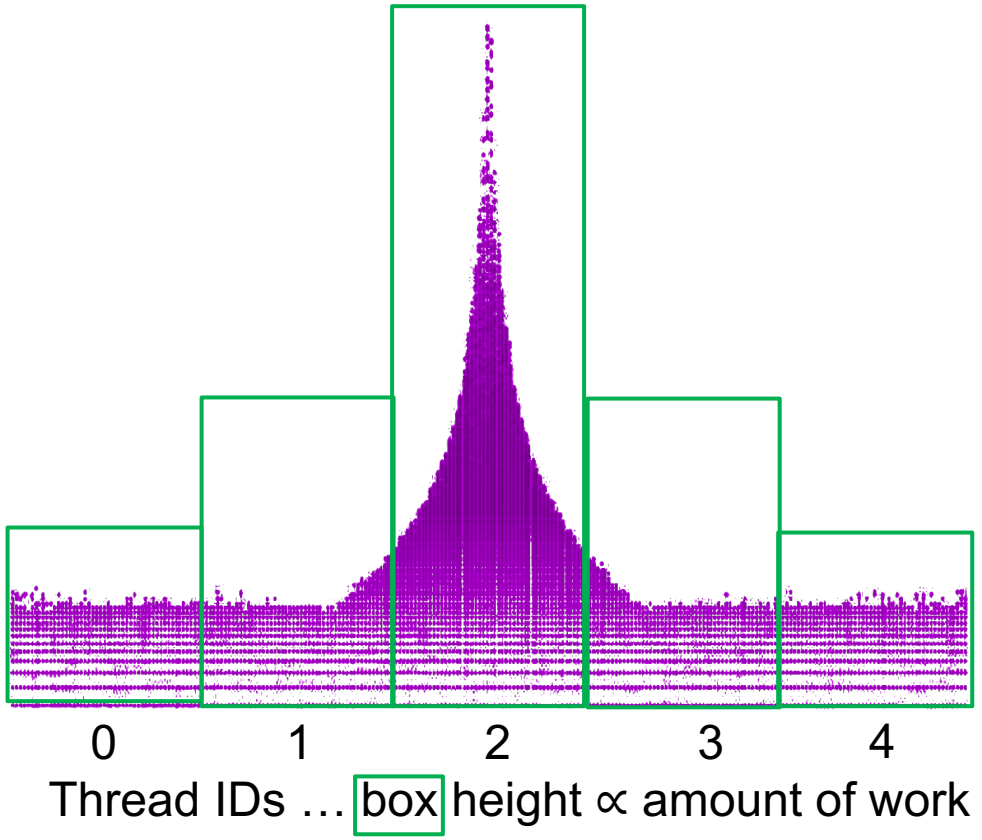
…. Let's pause a moment and consider one of the fundamental issues EVERY parallel programmer must grapple with

Load Balancing

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

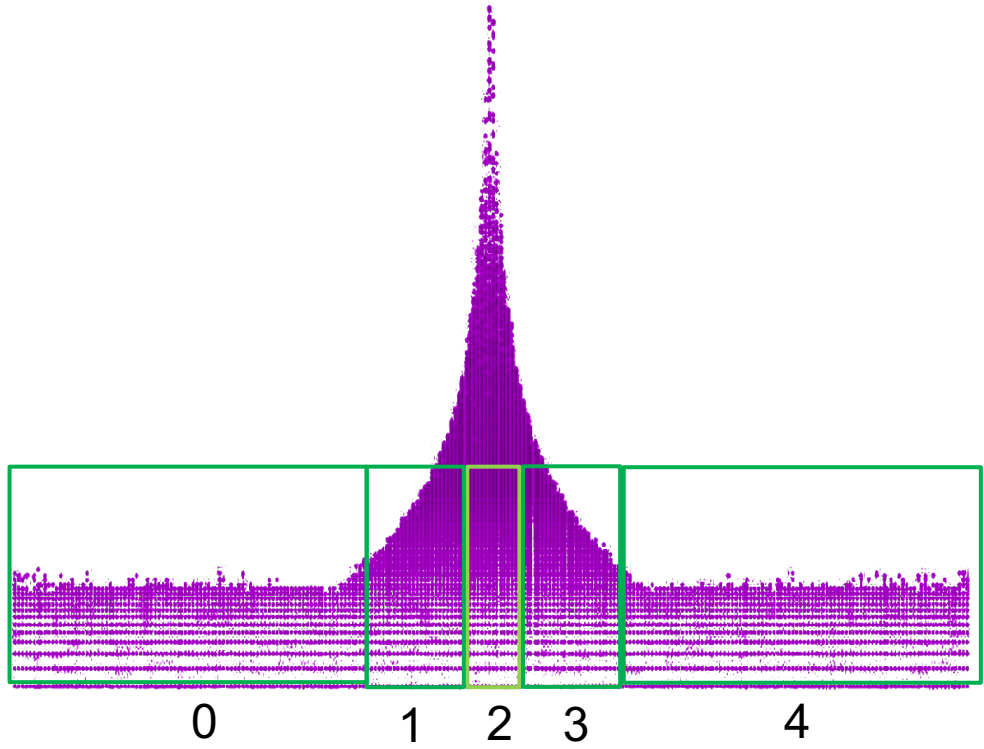
- Example: Partition a problem into equal sized chunks but for work that is unevenly distributed spatially.
  - Thread 2 has MUCH more work. The uneven distribution of work will limit performance.

- 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.

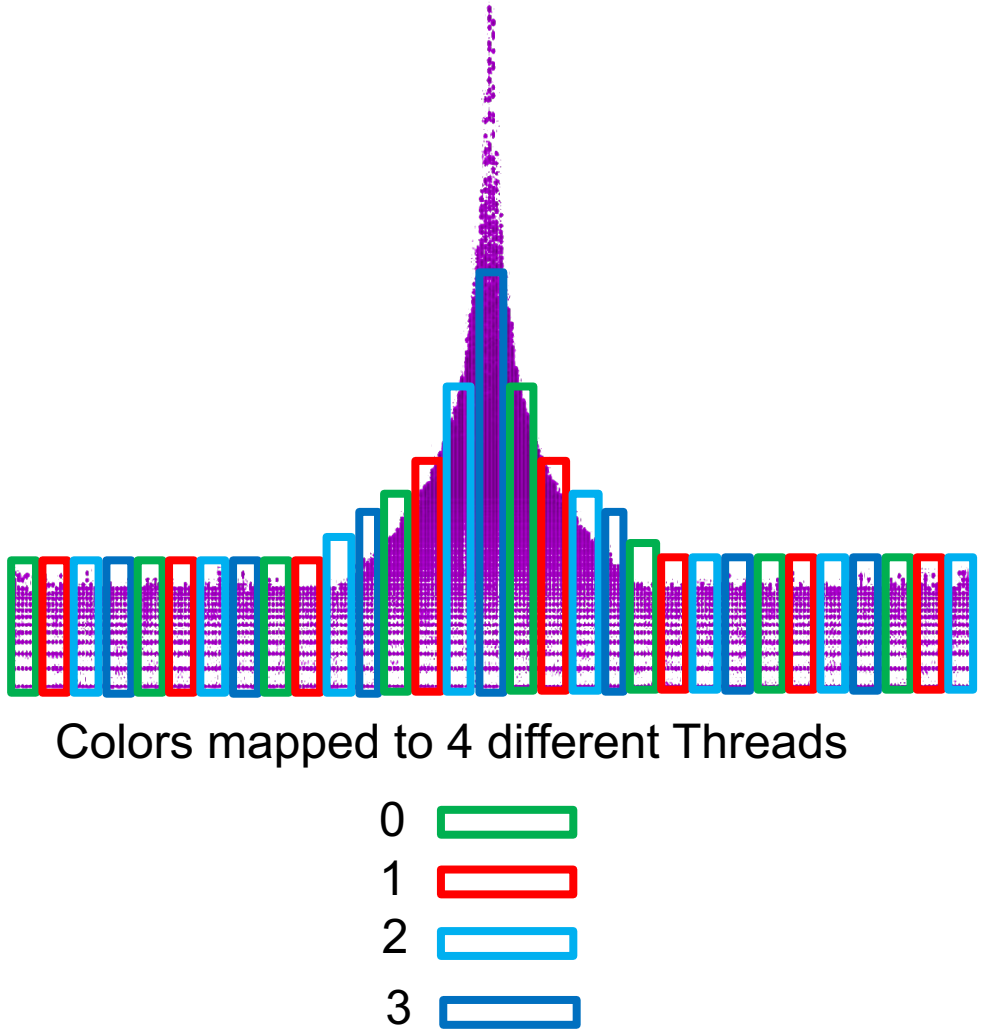


Thread IDs … box height ∝ amount of work

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

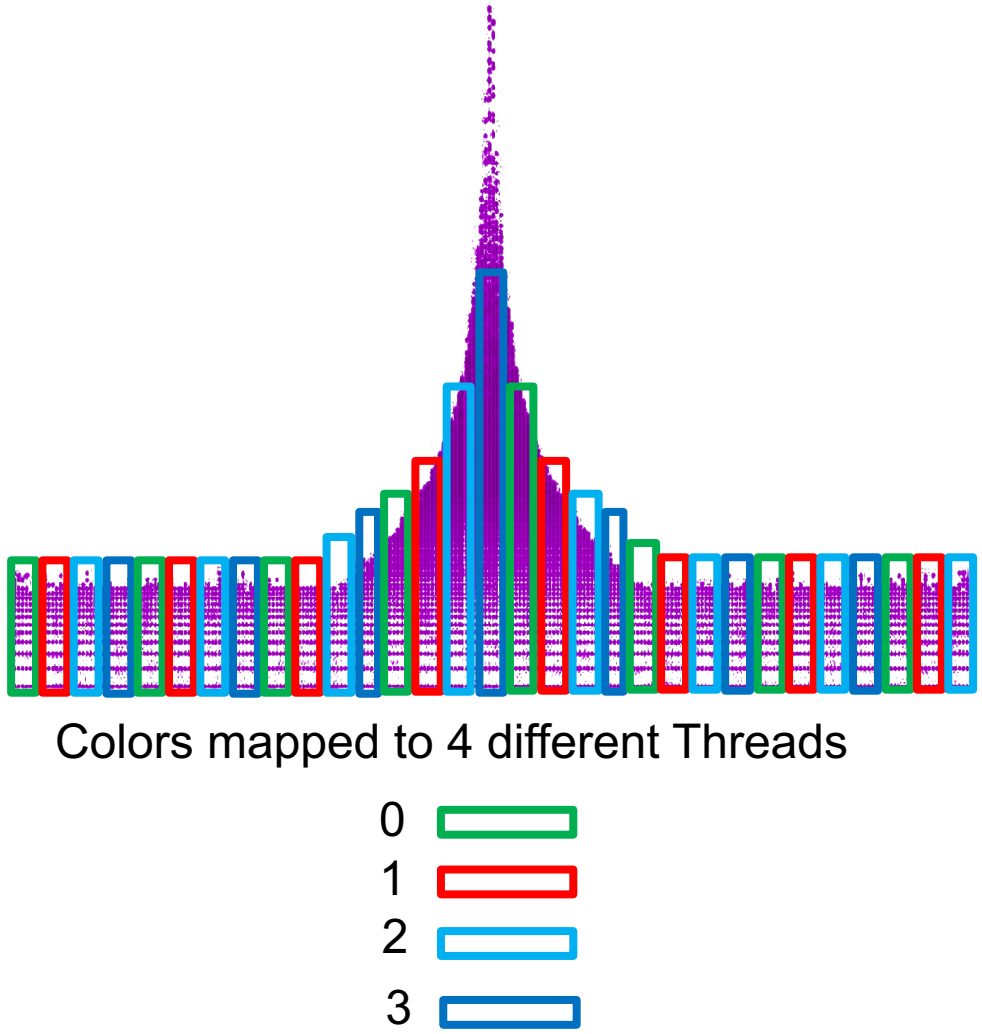


Load Balancing

- 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



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**

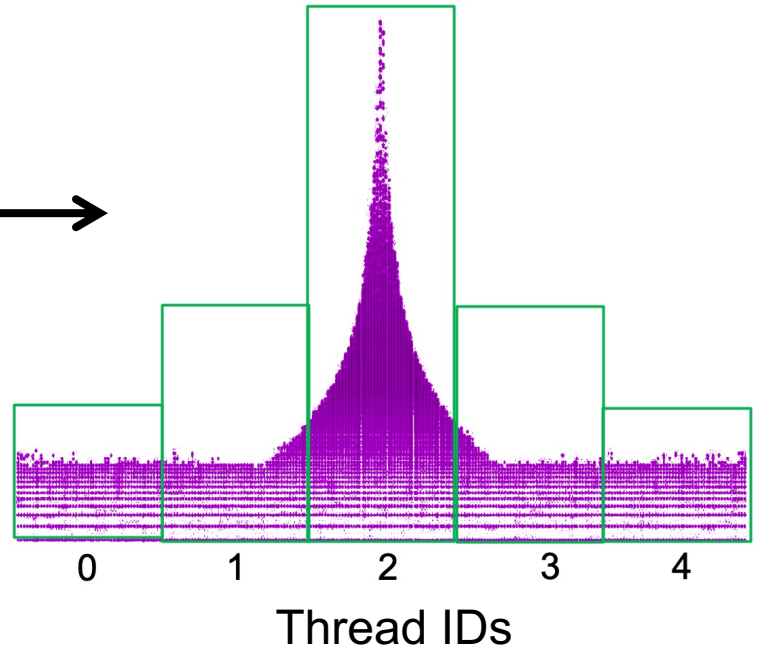
Least work at runtime : scheduling done at compile-time

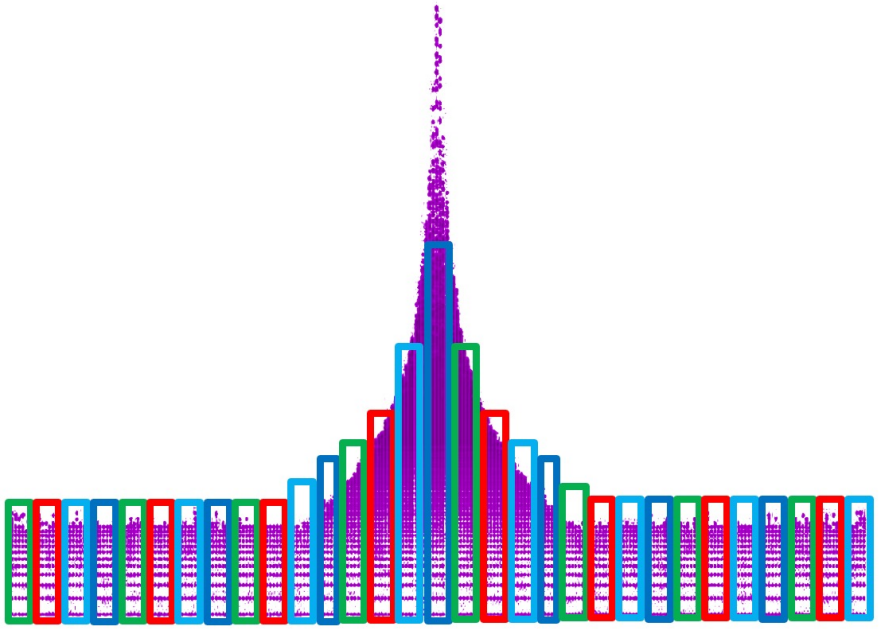
Most work at runtime : complex scheduling logic used at run-time

Loop Worksharing Constructs: The schedule clause

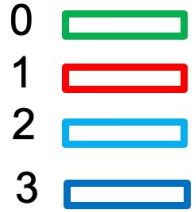
- The schedule clause … most common cases:

#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

implicit barrier at the end of a parallel region

no implicit barrier due to nowait

Outline

- Introduction to OpenMP
- Creating Threads
- Synchronization
- Parallel Loops
- Data Environment
- Irregular Parallelism and Tasks
- NUMA systems and GPUs
- 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.

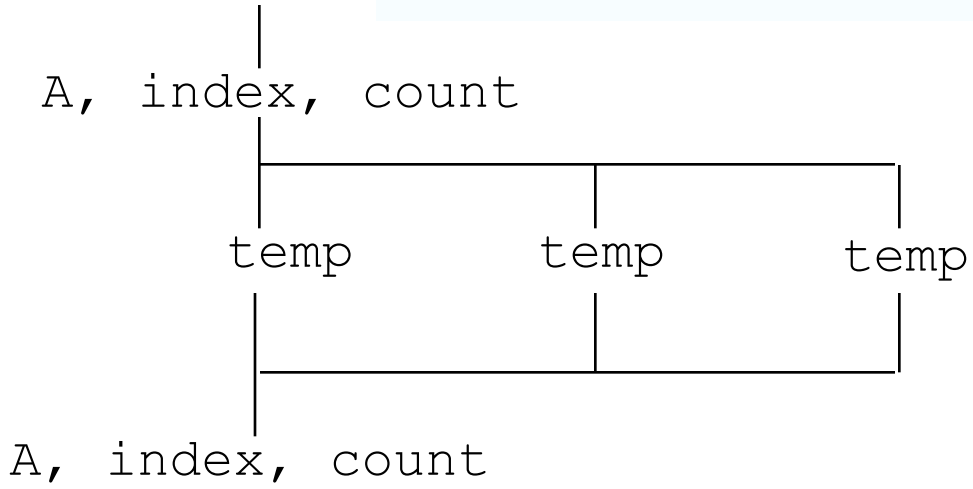
Data Sharing: Examples

```
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

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

```
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

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.

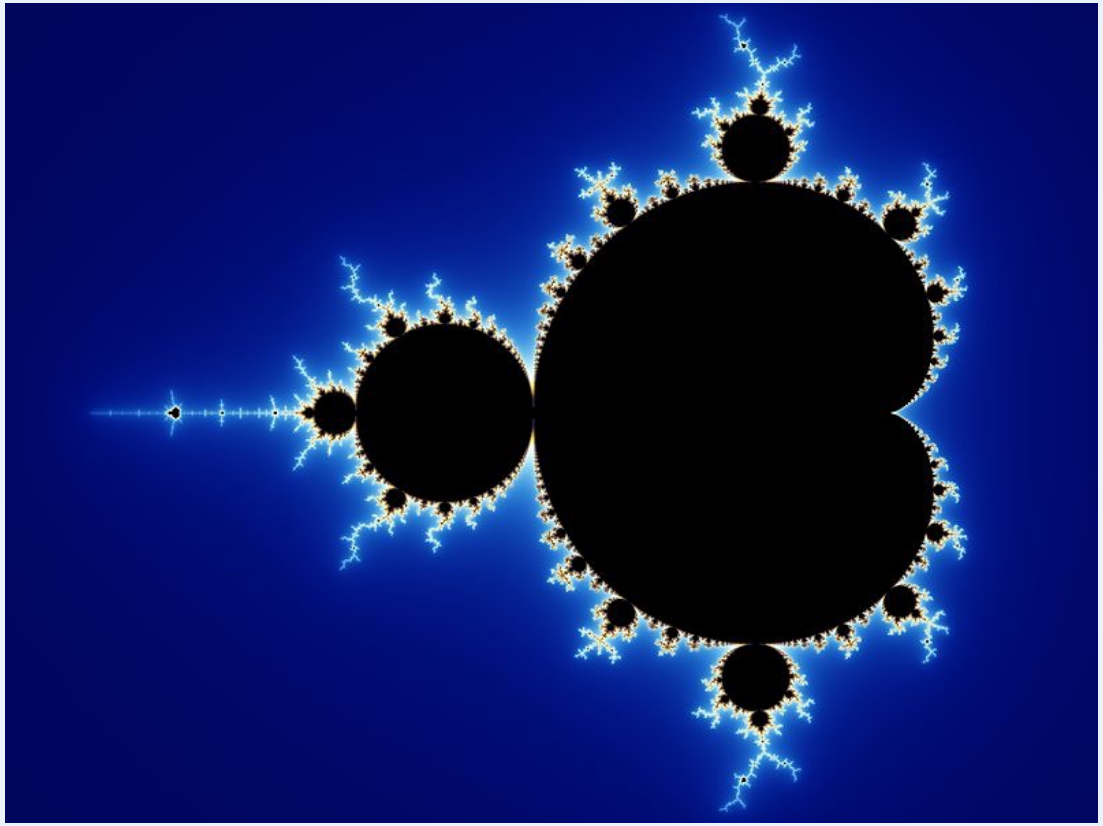


Image Source: Created by Wolfgang Beyer with the program Ultra Fractal 3. - Own work, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=321973

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;
      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++;
              break;  // exit the loop
        }
      }
      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;
      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++;
          break;  // exit the loop
        }
      }
      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
- Irregular Parallelism and Tasks
- NUMA systems and GPUs
- 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

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

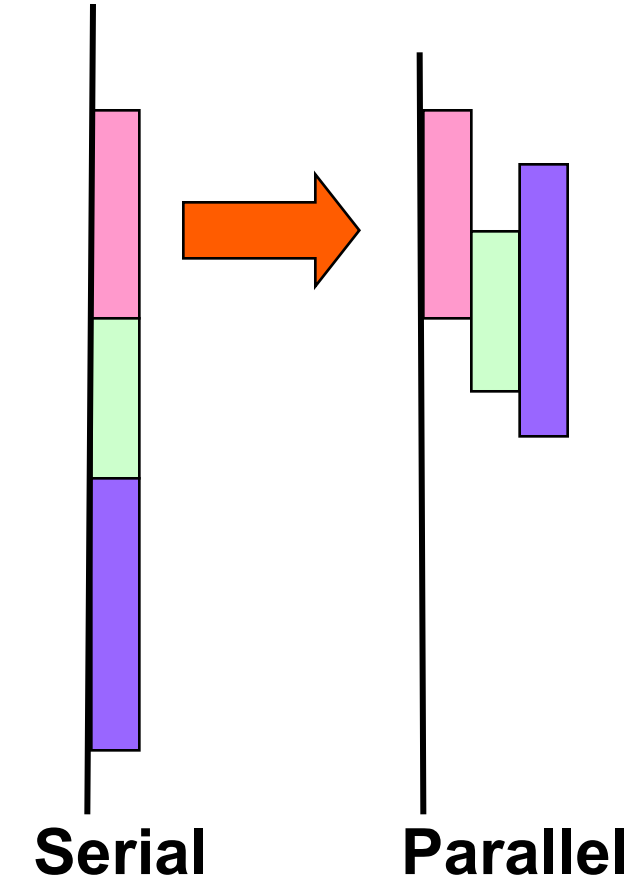
There has got to be a better way to do this

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

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

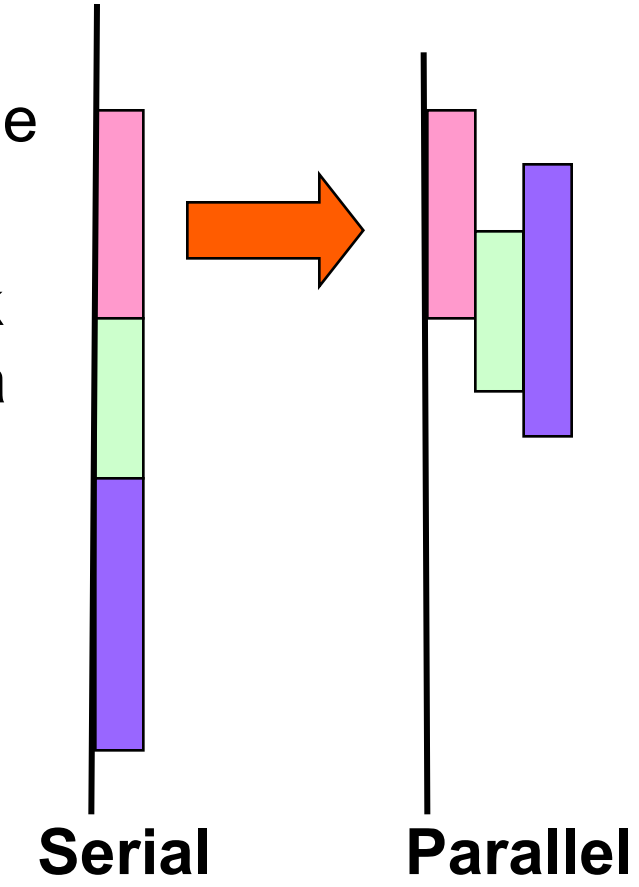
What are Tasks?

- 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!

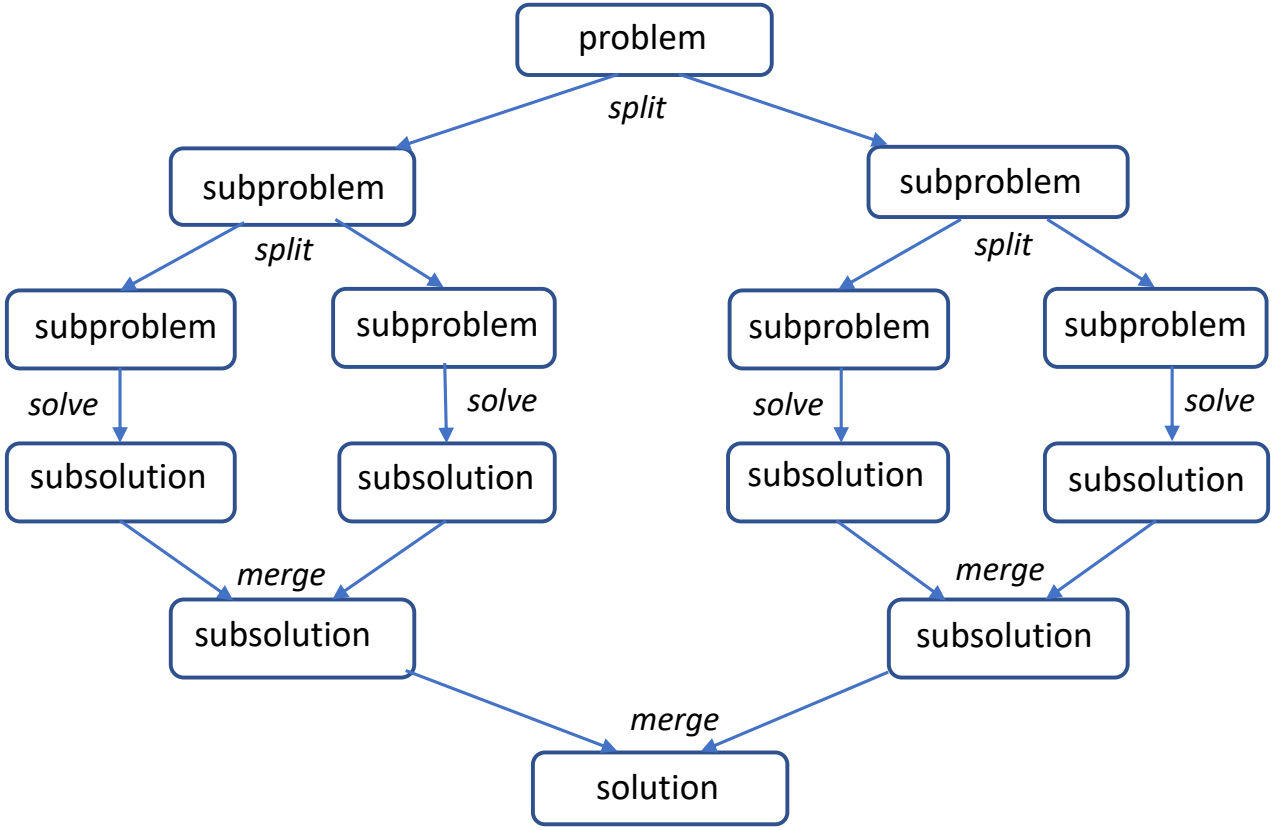
Parallel Fibonacci

```
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?**

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™ 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

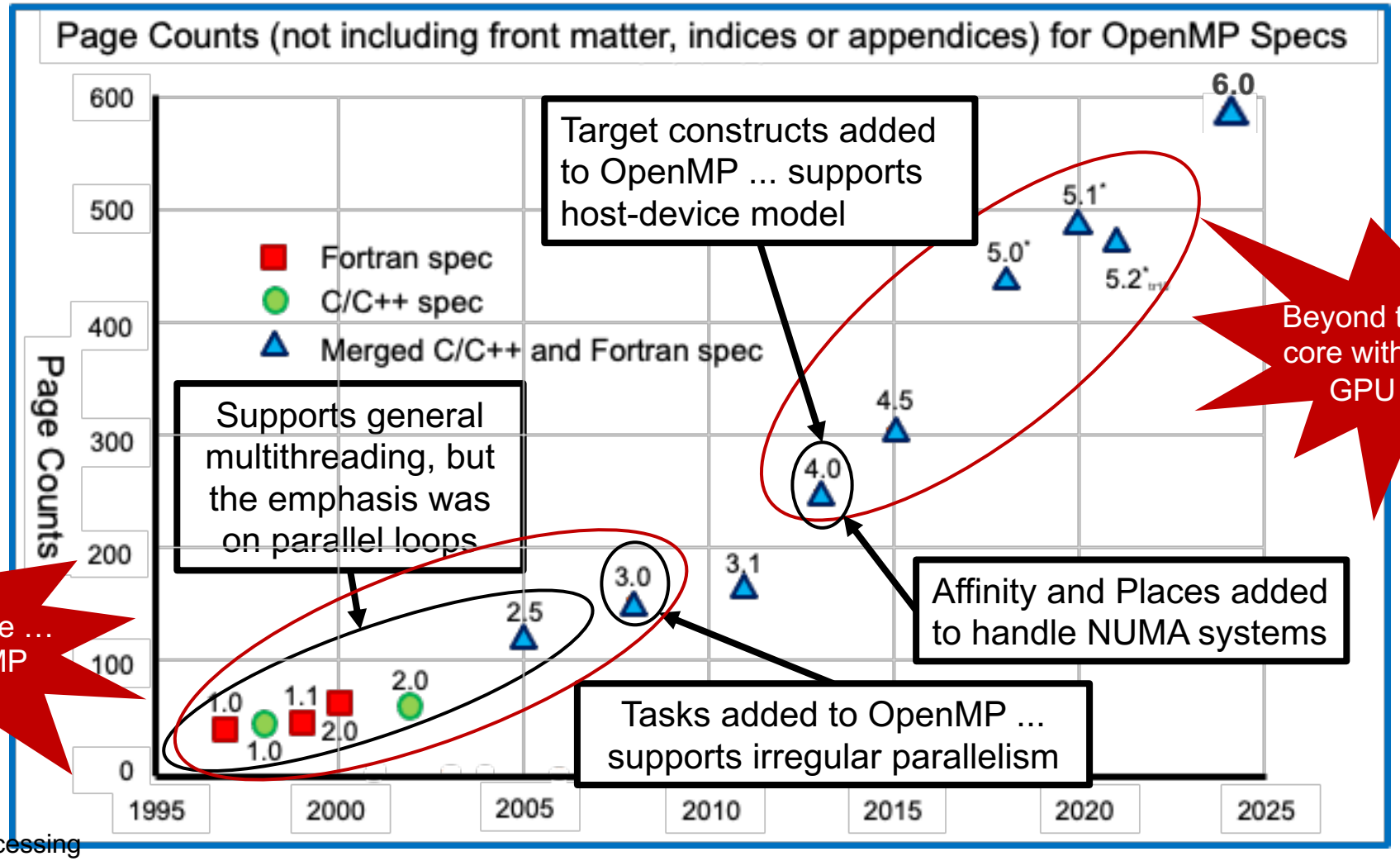
- Don’t expect miracles from the runtime
  - best results usually obtained where the user controls the number and granularity of tasks

Outline

- Introduction to OpenMP
- Creating Threads
- Synchronization
- Parallel Loops
- Data Environment
- Irregular Parallelism and Tasks
- NUMA systems and GPUs
- Recap

The growth of complexity in OpenMP

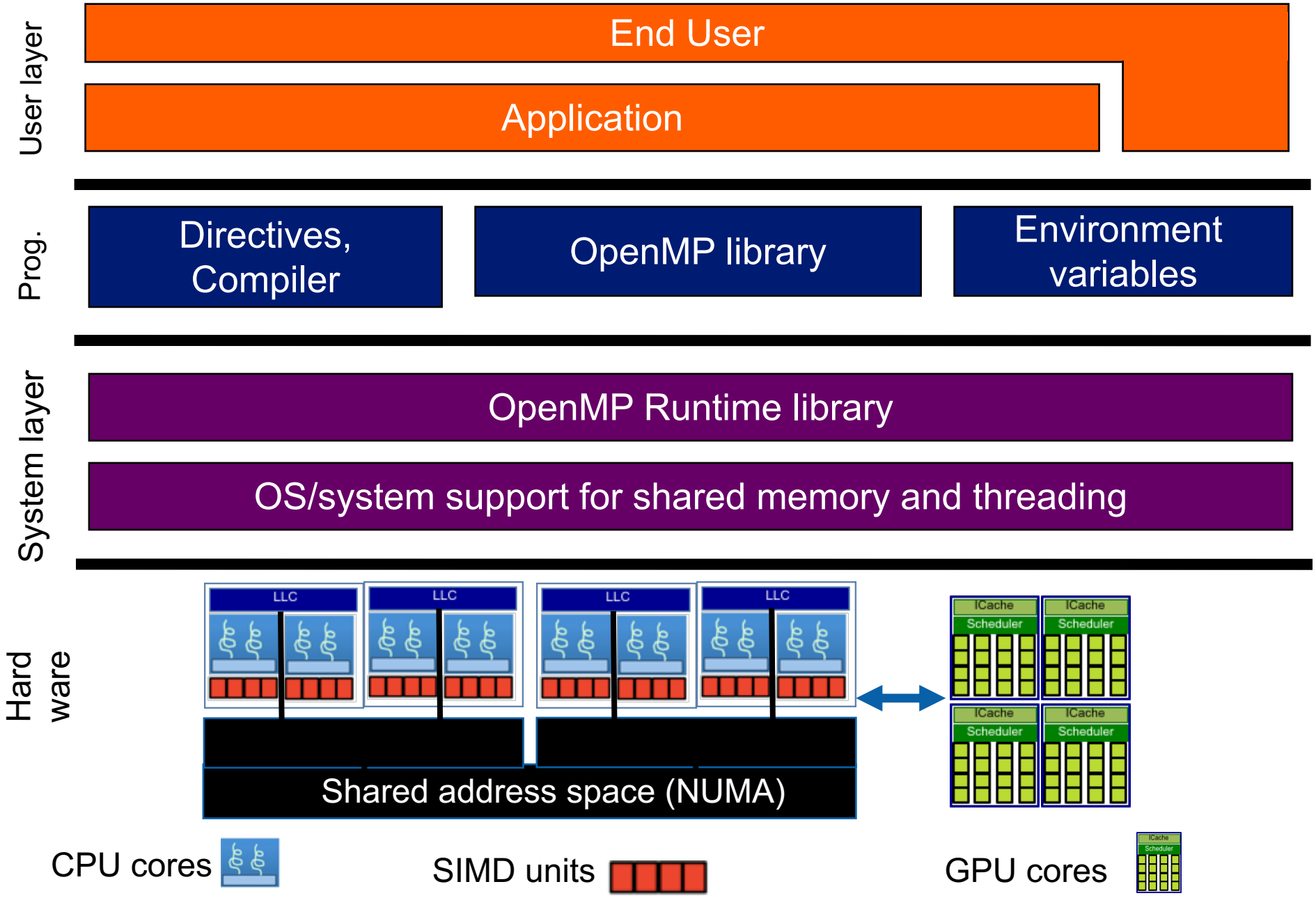
- OpenMP started out in 1997 as a simple interface for the application programmers more versed in their area of science than computer science.
- The complexity has grown considerably over the years!



SMP: symmetric multiprocessing

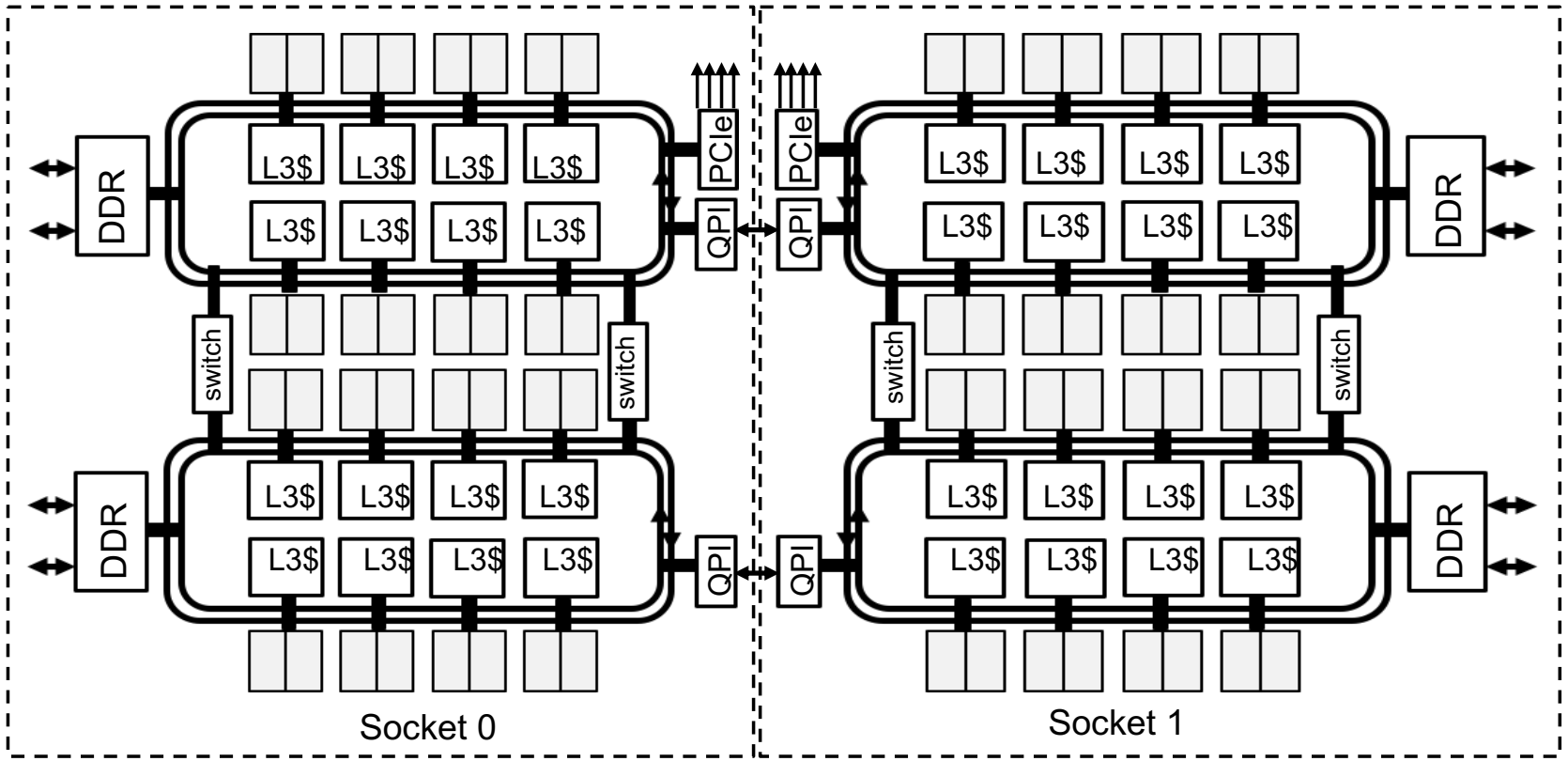
NUMA: non-uniform memory architecture

OpenMP basic definitions: the solution stack



NUMA Systems: You must optimize code for their complex memory subsystems

- A floating-point operation takes O(~1 ns).
  - L1 Cache ~1.5 ns
  - L2 Cache reference ~5 ns
  - L3 Cache reference ~25 ns
  - Near memory DRAM access ~100ns
  - Near memory DRAM access ~100ns
  - Far memory DRAM access ~200 ns
- The key to performance is to minimize memory movement …. get the memory movement right and the “rest” is easy

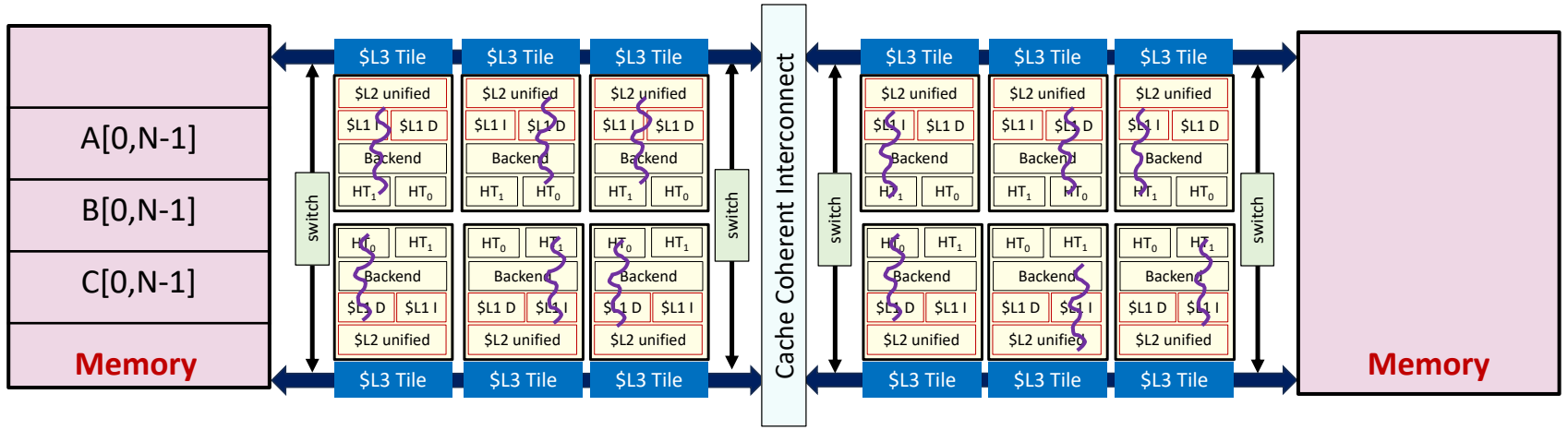


ns: nanosecond

Dual Socket node with Intel® Xeon™ E5-2698v3 CPUs

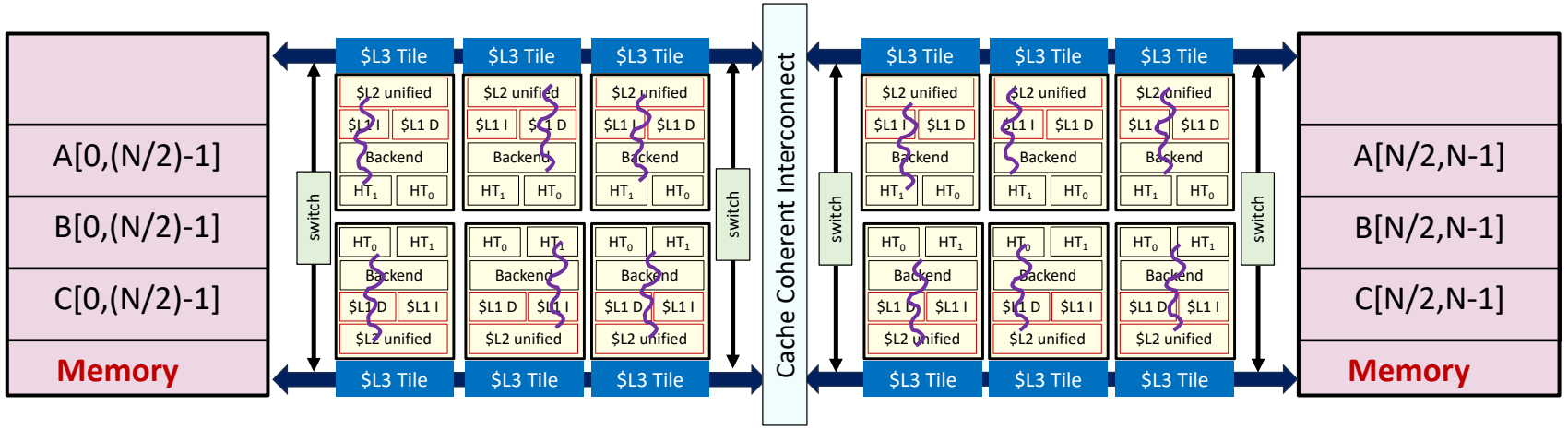
Example: use all available memory

- Stream memory bandwidth benchmark running on a two socket Intel® Xeon™ X5675 with 12 threads on 12 cores



3 arrays in one NUMA domain

copy	scale	add	triad
18.8 GB/s	18.5 GB/s	18.1 GB/s	18.2 GB/s



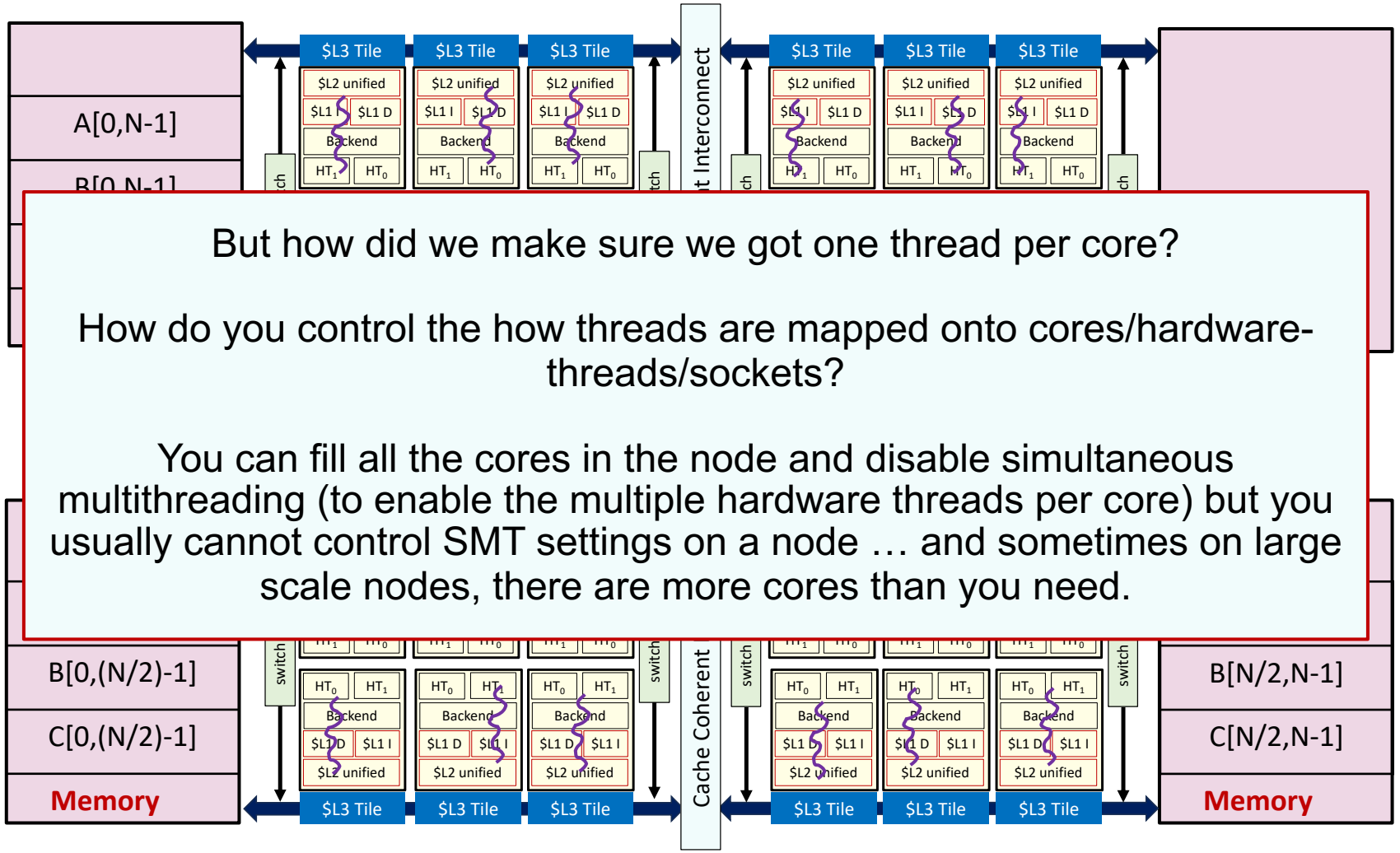
Arrays split between both NUMA domains

copy	scale	add	triad
41.3 GB/s	39.3 GB/s	40.3 GB/s	40.4 GB/s

Based on content from Christian Terboven (RWTH/AACHEN) and Michael Klemm (OpenMP)

Example: use all available memory

- Stream memory bandwidth benchmark running on a two socket Intel® Xeon™ X5675 with 12 threads on 12 cores



But how did we make sure we got one thread per core?

How do you control the how threads are mapped onto cores/hardware-threads/sockets?

You can fill all the cores in the node and disable simultaneous multithreading (to enable the multiple hardware threads per core) but you usually cannot control SMT settings on a node … and sometimes on large scale nodes, there are more cores than you need.

3 arrays in one NUMA domain

copy	scale	add	triad
18.8 GB/s	18.5 GB/s	18.1 GB/s	18.2 GB/s

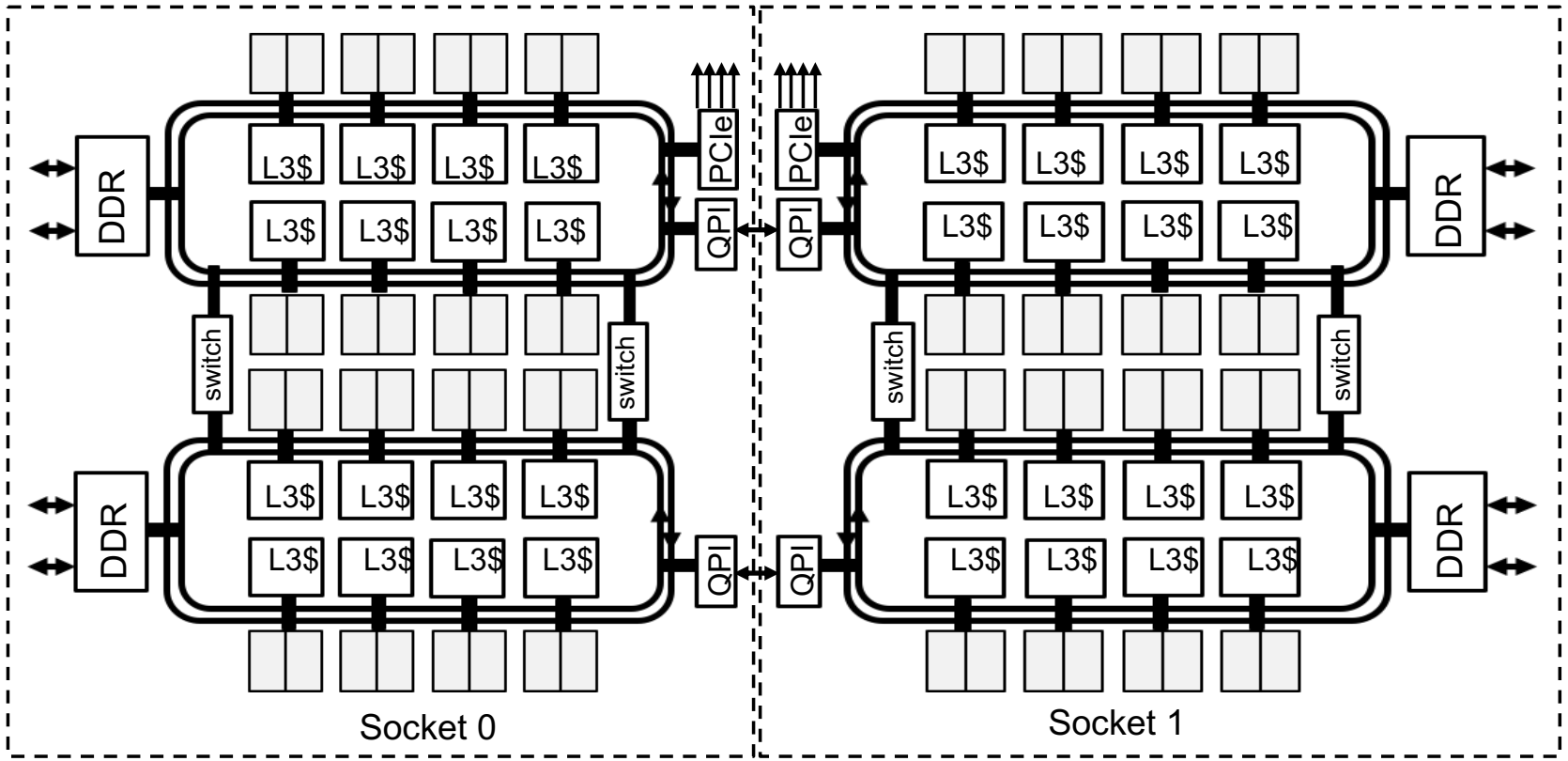
Arrays split between both NUMA domains

copy	scale	add	triad
41.3 GB/s	39.3 GB/s	40.3 GB/s	40.4 GB/s

Based on content from Christian Terboven (RWTH/AACHEN) and Michael Klemm (OpenMP)

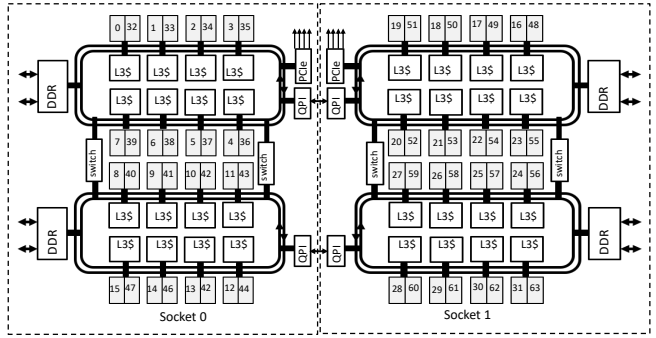
NUMA nodes and the places we can put threads

- OpenMP defines the concept of places on a NUMA node where threads can execute.
- The idea is to map the OS defined virtual cores onto places visible to OpenMP for threads assignment
- The first step is to understand the OS defined virtual cores (also known as virtual processing units or PUs)



Discover the OS view of virtual cores

- Portable Hardware Locality tools …. hwloc-ls, lstopo, Numactl and others depending on the system. Generates text or graphical output depending on how the tools are configured on your system.



Machine (126GB total)

NUMANode P#0 (63GB)

Package P#0

L3 (40MB)

L2 (256KB)	L2 (256KB)	L2 (256KB)	L2 (256KB)	L2 (256KB)	L2 (256KB)	L2 (256KB)	L2 (256KB)	L2 (256KB)	L2 (256KB)	L2 (256KB)	L2 (256KB)	L2 (256KB)	L2 (256KB)	L2 (256KB)	L2 (256KB)
L1d (32KB)	L1d (32KB)	L1d (32KB)	L1d (32KB)	L1d (32KB)	L1d (32KB)	L1d (32KB)	L1d (32KB)	L1d (32KB)	L1d (32KB)	L1d (32KB)	L1d (32KB)	L1d (32KB)	L1d (32KB)	L1d (32KB)	L1d (32KB)
L1i (32KB)	L1i (32KB)	L1i (32KB)	L1i (32KB)	L1i (32KB)	L1i (32KB)	L1i (32KB)	L1i (32KB)	L1i (32KB)	L1i (32KB)	L1i (32KB)	L1i (32KB)	L1i (32KB)	L1i (32KB)	L1i (32KB)	L1i (32KB)
Core P#0	Core P#1	Core P#2	Core P#3	Core P#4	Core P#5	Core P#6	Core P#7	Core P#8	Core P#9	Core P#10	Core P#11	Core P#12	Core P#13	Core P#14	Core P#15
PU P#0	PU P#1	PU P#2	PU P#3	PU P#4	PU P#5	PU P#6	PU P#7	PU P#8	PU P#9	PU P#10	PU P#11	PU P#12	PU P#13	PU P#14	PU P#15
PU P#32	PU P#33	PU P#34	PU P#35	PU P#36	PU P#37	PU P#38	PU P#39	PU P#40	PU P#41	PU P#42	PU P#43	PU P#44	PU P#45	PU P#46	PU P#47

NUMANode P#1 (63GB)

Package P#1

L3 (40MB)

L2 (256KB)	L2 (256KB)	L2 (256KB)	L2 (256KB)	L2 (256KB)	L2 (256KB)	L2 (256KB)	L2 (256KB)	L2 (256KB)	L2 (256KB)	L2 (256KB)	L2 (256KB)	L2 (256KB)	L2 (256KB)	L2 (256KB)	L2 (256KB)
L1d (32KB)	L1d (32KB)	L1d (32KB)	L1d (32KB)	L1d (32KB)	L1d (32KB)	L1d (32KB)	L1d (32KB)	L1d (32KB)	L1d (32KB)	L1d (32KB)	L1d (32KB)	L1d (32KB)	L1d (32KB)	L1d (32KB)	L1d (32KB)
L1i (32KB)	L1i (32KB)	L1i (32KB)	L1i (32KB)	L1i (32KB)	L1i (32KB)	L1i (32KB)	L1i (32KB)	L1i (32KB)	L1i (32KB)	L1i (32KB)	L1i (32KB)	L1i (32KB)	L1i (32KB)	L1i (32KB)	L1i (32KB)
Core P#0	Core P#1	Core P#2	Core P#3	Core P#4	Core P#5	Core P#6	Core P#7	Core P#8	Core P#9	Core P#10	Core P#11	Core P#12	Core P#13	Core P#14	Core P#15
PU P#16	PU P#17	PU P#18	PU P#19	PU P#20	PU P#21	PU P#22	PU P#23	PU P#24	PU P#25	PU P#26	PU P#27	PU P#28	PU P#29	PU P#30	PU P#31
PU P#48	PU P#49	PU P#50	PU P#51	PU P#52	PU P#53	PU P#54	PU P#55	PU P#56	PU P#57	PU P#58	PU P#59	PU P#60	PU P#61	PU P#62	PU P#63

Graphical output for a dual Socket node with Intel® Xeon™ E5-2698v3 CPUs

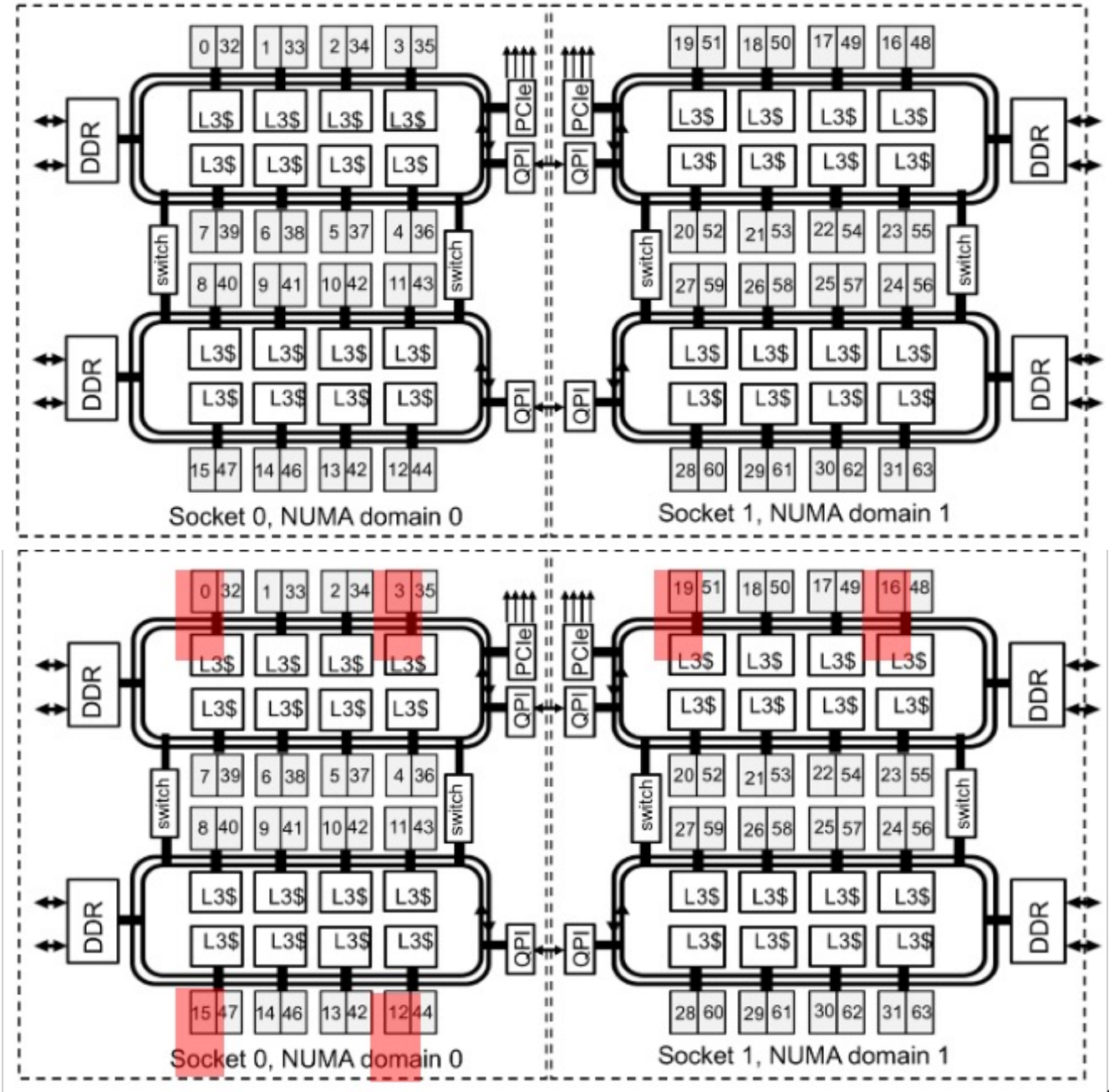
PU: processor unit. The smallest physical execution unit that hwloc recognizes.

Based on content from
Yun (Helen) He from NERSC)

Using OMP_PLACES to select where to run code

- After using a tool to learn the logical core IDs (processor units or PUs) use environment variables to define where threads can be scheduled to execute.

```
> export OMP_PLACES="{0, 3, 15, 12, 19,  16, 28, 31}"
> export NUM_THREADS= 6

#pragma omp parallel
{
   // do a bunch of cool stuff

}
```



Using OMP_PLACES to select where to run code

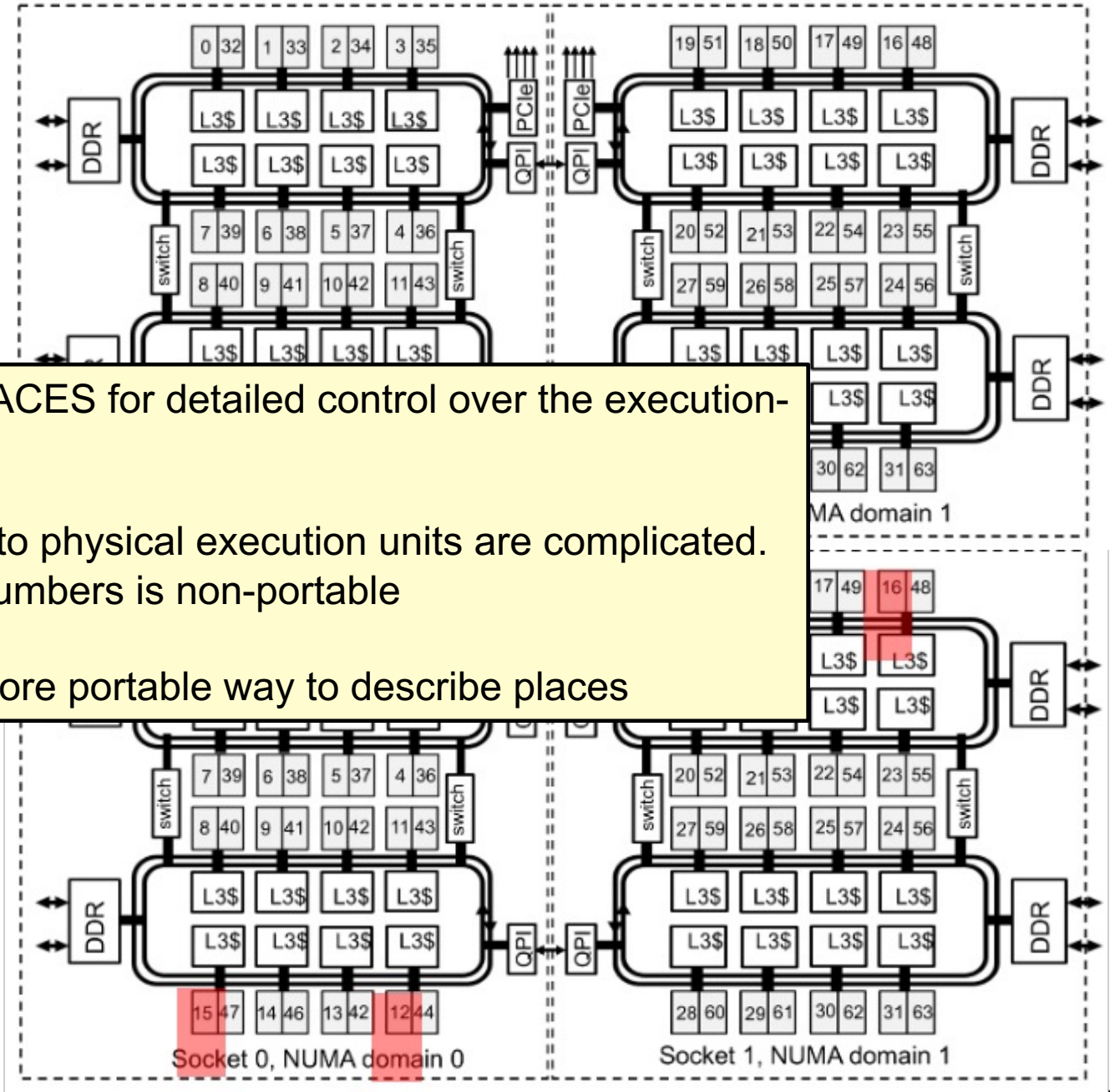
- After using a tool to learn the logical core IDs (processor units or PUs) use environment variables to define where threads can be scheduled to execute.

```
> export OMP_PLACES="{0, 3, 15, 12, 19,  16, 28, 31}"
> export NUM_THR

#pragma omp parall
{
    // do a bunch of c

}
```

Programmers can use OMP_PLACES for detailed control over the execution-units threads utilize. BUT …

- The rules for mapping onto physical execution units are complicated.
- PLACES expressed as numbers is non-portable

There has to be an easier and more portable way to describe places



Hardware Abstraction: OMP_PLACES

- OMP_PLACES environment variable
  - controls thread allocation
  - defines a series of places to which the threads are assigned

- It can be an abstract name or a specific list
  - threads: each place corresponds to a single hardware thread
  - cores: each place corresponds to a single core (which may have one or more hardware threads)
  - sockets: each place corresponds to a single socket (consisting of one or more cores)
  - a list with explicit place values of CPU ids, such as:
    - export OMP_PLACES=“ {0:4:2},{1:4:2}” (equivalent to “{0,2,4,6},{1,3,5,7}”)

- Examples:
  - export OMP_PLACES=threads
  - export OMP_PLACES=cores

Thread Affinity … mapping threads to places

Thread affinity to places: OMP_PROC_BIND

- Controls thread affinity within and between OpenMP places
- Allowed values:
  - true: the runtime will not move threads around between processors
  - false: the runtime may move threads around between processors
  - close: bind threads close to the primary* thread
  - spread: bind threads as evenly distributed as possible (i.e., spread them out)
  - primary: bind threads to the same place as the primary thread
- The values **primary***, **close,** and **spread** imply the value **true**

Examples:

```
export OMP_PROC_BIND=spread
```

*Primary thread: this is the thread with ID=0 that encountered the parallel construct and created the team of threads

Thread affinity to places: OMP_PROC_BIND

- Controls thread affinity within and between OpenMP places
- Allowed values:
  - true: the runtime will not move threads around between processors
  - false: the runtime may move threads around between processors
  - close: bind threads close to the primary* thread
  - spread: bind threads as evenly distributed (spreaded) as possible
  - primary: bind threads to the same place as the primary thread
- The values **primary***, **close,** and **spread** imply the value **true**

Example … using clauses on a parallel construct:

```
#pragma omp parallel num_threads(4) proc_bind(spread)
```

*Primary thread: this is the thread with ID=0 that encountered the parallel construct and created the team of threads

Examples: OMP_PROC_BIND

Consider 4 cores total, 2 hardware threads per core, 4 OpenMP threads

- none: no affinity setting
- close: Bind threads as close to each other as possible

Node	Core 0		Core 1		Core 2		Core 3	
	HT_0	HT_1	HT_0	HT_1	HT_0	HT_1	HT_0	HT_1
	PU 0	PU 1	PU 2	PU 3	PU 4	PU 5	PU 6	PU 7
Thread	0	1	2	3				

- spread: Bind threads as far apart as possible

Node	Core 0		Core 1		Core 2		Core 3	
	HT_0	HT_1	HT_0	HT_1	HT_0	HT_1	HT_0	HT_1
	PU 0	PU 1	PU 2	PU 3	PU 4	PU 5	PU 6	PU 7
Thread	0		1		2		3	

- primary: bind threads to the same place as the primary thread

We define places explicitly with the IDs of the OS virtual cores (the PUs).

We do not control where the initial thread is placed. We will assume it is placed on HT1 or Core 0.

For this example, we have 4 place partitions.

OMP_PLACES={0,1},{2,3},{4,5},{6,7}

With close, threads placed in consecutive locations

With spread, threads placed in first place of each partition

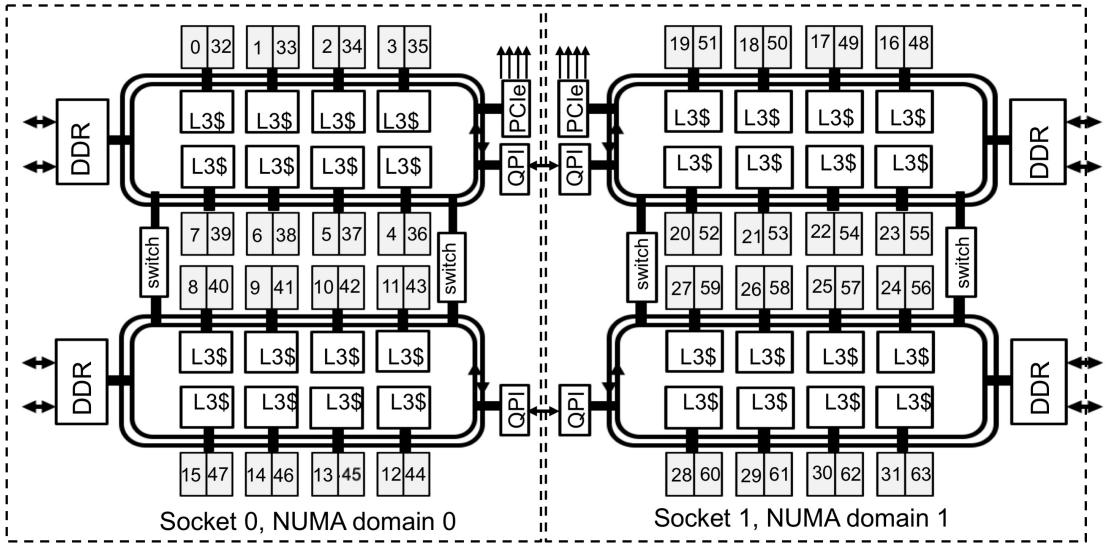
PU: processor unit. The smallest physical execution unit that hwloc recognizes.

OMP_PROC_BIND Choices for STREAM Benchmark

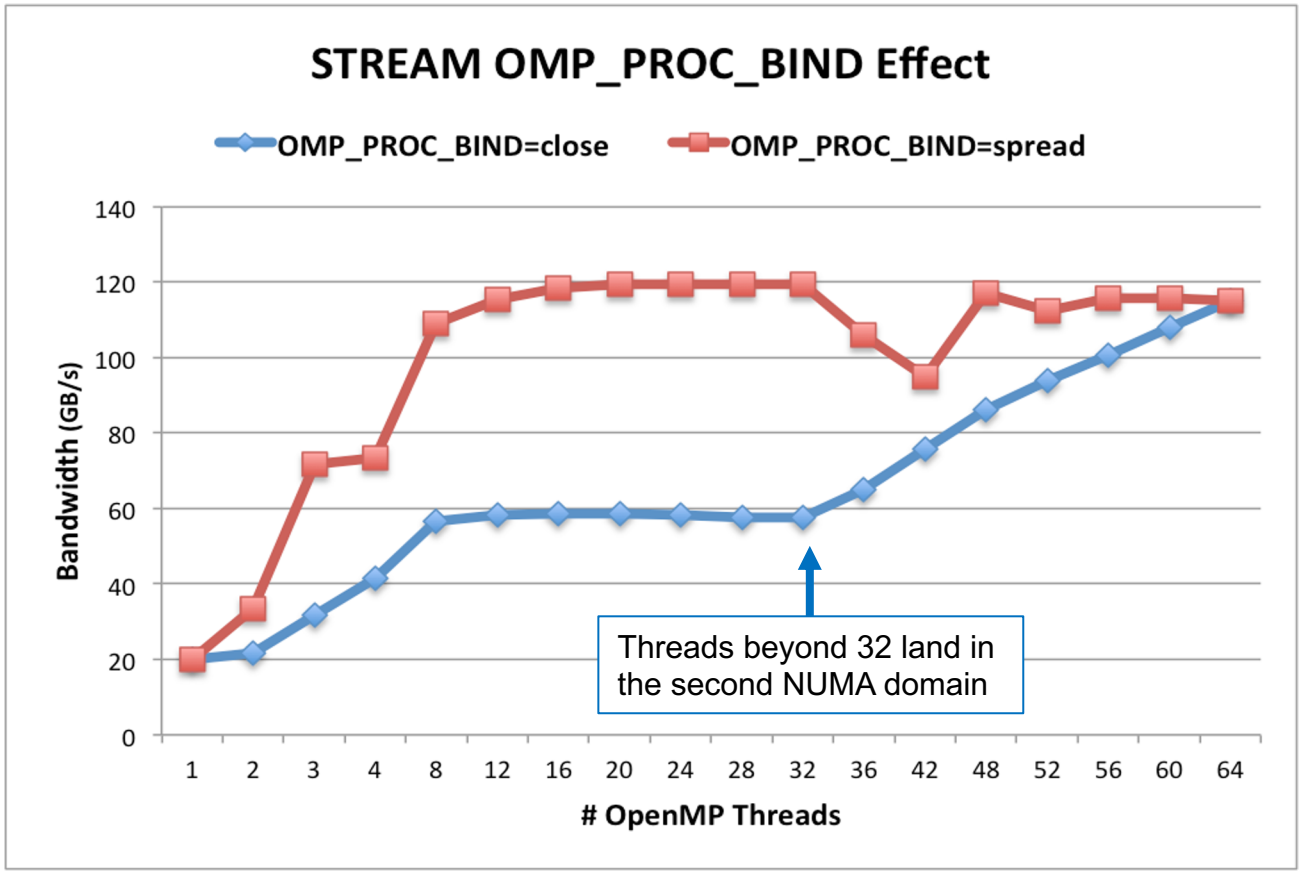
OMP_NUM_THREADS=32
OMP_PLACES=threads

OMP_PROC_BIND=close
Threads 0 to 31 bind to cores (0,32),(1,33),(2,34),…(15,47). All threads are in the first socket. The second socket is idle. Not optimal.

OMP_PROC_BIND=spread
Threads 0 to 31 bind to cores 0,1,2,… to 31. Both sockets and memory are used to maximize memory bandwidth.



Blue: OMP_PROC_BIND=close
Red: OMP_PROC_BIND=spread
Both with First Touch



Stream is a well known memory bandwidth benchmark based on simple vector operations on huge vectors

Based on content from Yun (Helen) He from NERSC)

Aligning memory to threads … First touch

Memory Affinity: Exploiting "First Touch" page mapping policy

Step 1.1 Initialization by primary thread only

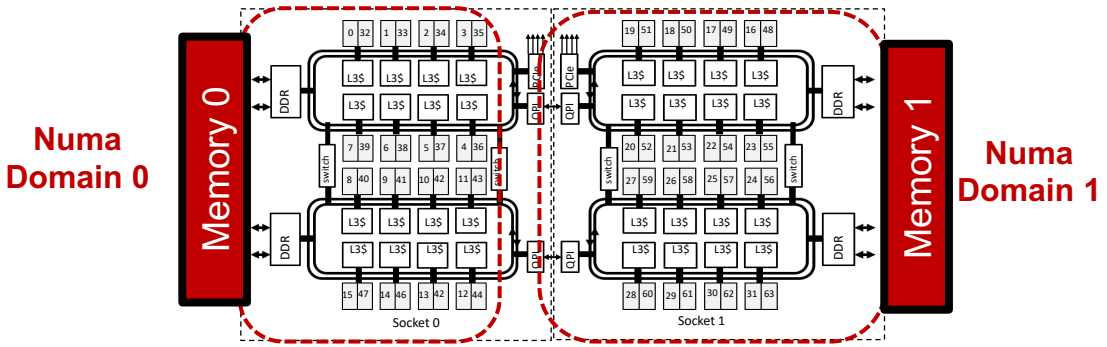
```
for (j=0; j<VectorSize; j++) {
      a[j] = 1.0; b[j] = 2.0; c[j] = 0.0;}
```

Step 1.2 Initialization by all threads

```
#pragma omp parallel for
for (j=0; j<VectorSize; j++) {
      a[j] = 1.0; b[j] = 2.0; c[j] = 0.0;}
```

Step 2 Compute

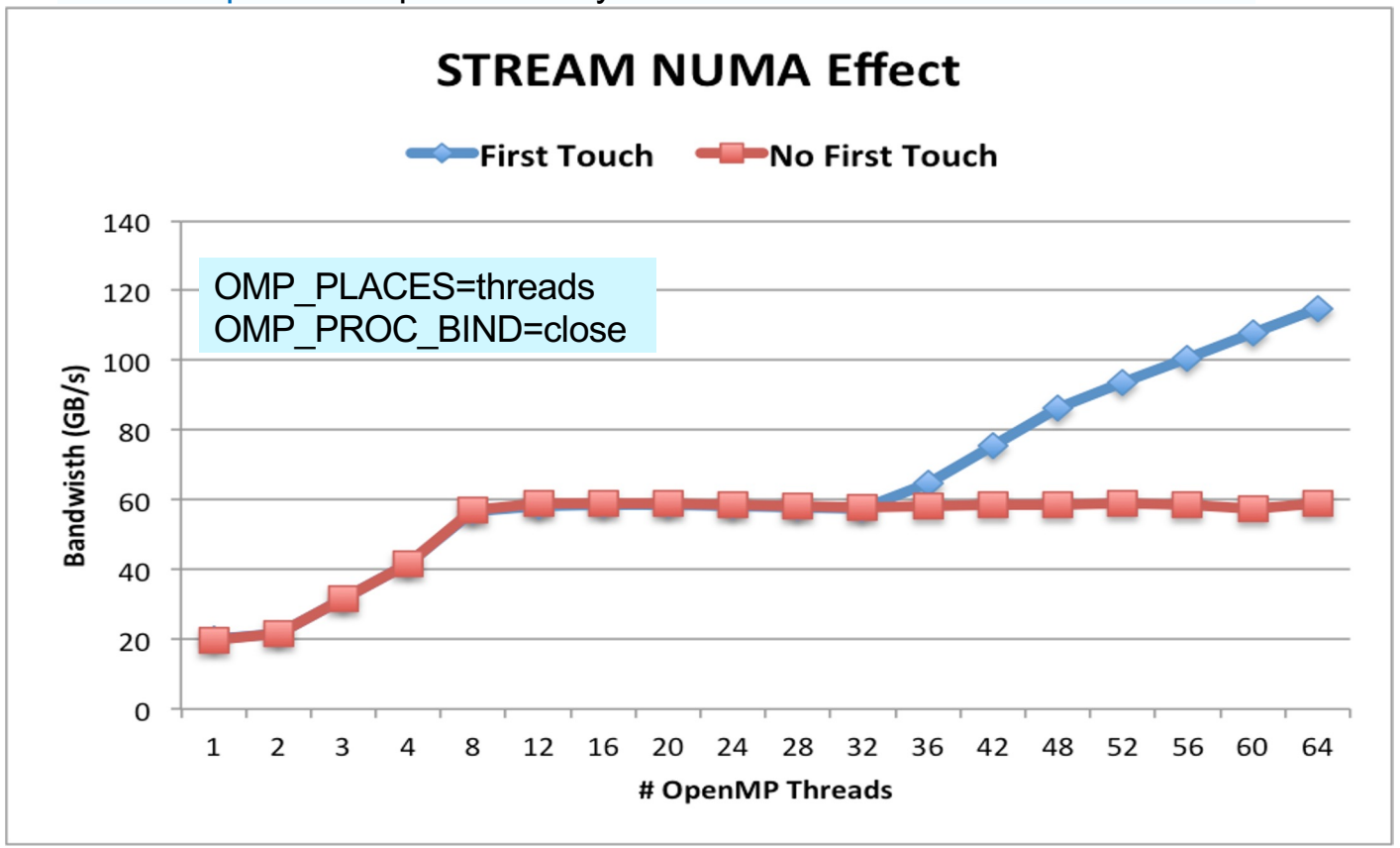
```
#pragma omp parallel for
for (j=0; j<VectorSize; j++)
      a[j]=b[j]+d*c[j];
```



A dual Socket node with Intel® Xeon™ E5-2698v3 CPUs

- The OS maps pages of memory based on a **first touch** policy.
- Hence, Affinity to memory is not defined when memory is allocated … it is defined when the memory is initialized.
- The result is memory is local to the thread which initializes it.

Red: step 1.1 + step 2. Memory from Numa Domain 0 only
Blue: step 1.2 + step 2. Memory used from both NUMA domains



Based on content from Yun (Helen) He from NERSC)

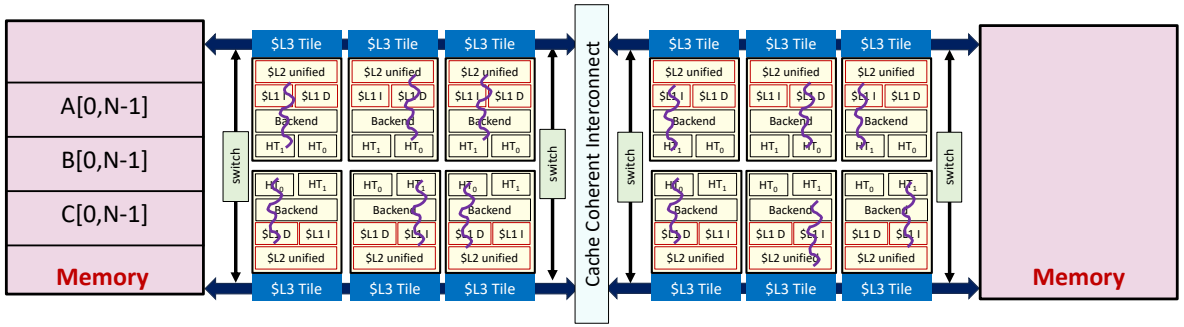
Example: working with the First Touch Policy

Rember this slide?

Example: use all available memory

- Stream memory bandwidth benchmark running on a two socket Intel® Xeon™ X5675 with 12 threads on 12 cores

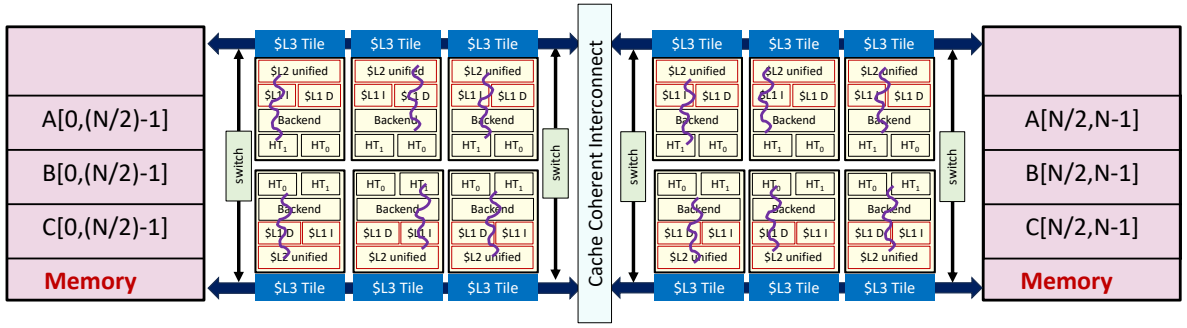
Arrays A, B, and C initialized on primary thread



3 arrays in one NUMA domain

copy	scale	add	triad
18.8 GB/s	18.5 GB/s	18.1 GB/s	18.2 GB/s

Arrays A, B, and C initialized in parallel



Arrays split between both NUMA domains

copy	scale	add	triad
41.3 GB/s	39.3 GB/s	40.3 GB/s	40.4 GB/s

Based on content from Christian Terboven (RWTH/AACHEN) and Michael Klemm (OpenMP)

But its not just any “in parallel”. You want to initialize the arrays with the same “parallel for schedule” that will be used when the threads do the computations with A, B, and C

Nested parallelism

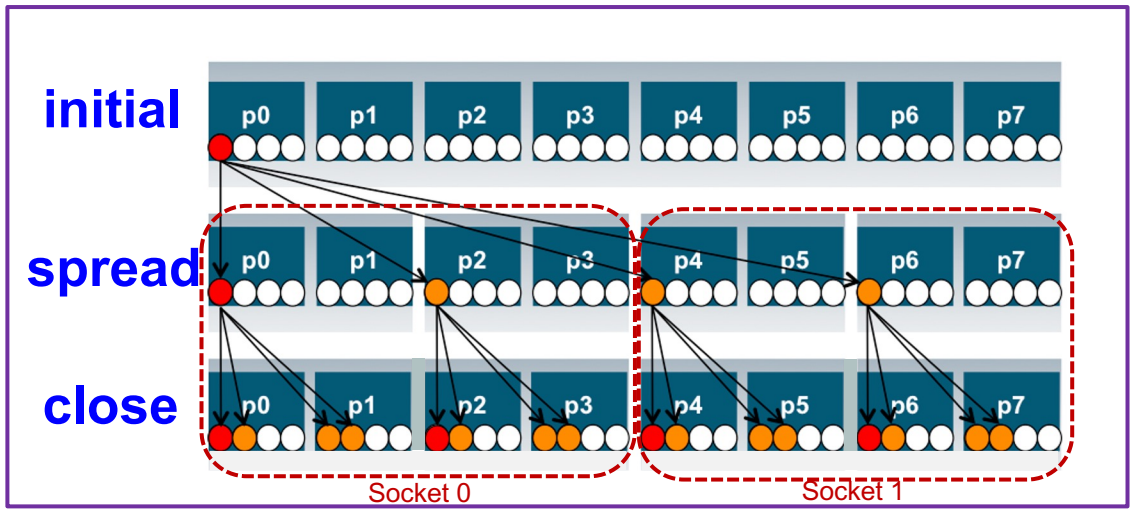
Process and Thread Affinity in Nested OpenMP

Consider a program with nested parallel regions

```
#pragma omp parallel
    #pragma omp parallel
```

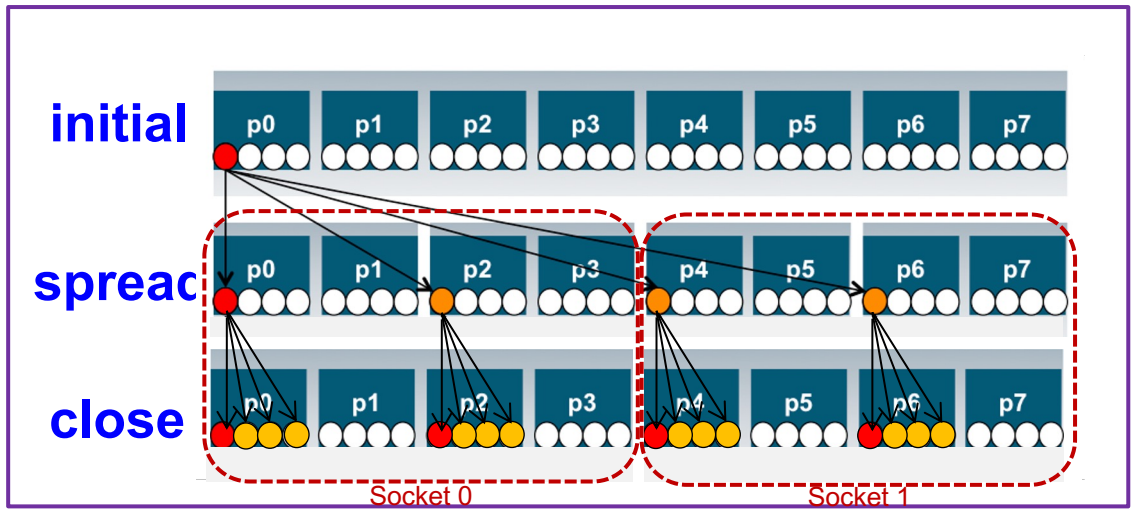
Running on a system with 2 sockets, 4 cores per socket, 4 hardware-threads per core

```
export OMP_MAX_ACTIVE_LEVELS=2
export OMP_NUM_THREADS=4,4
export OMP_PLACES=cores
export OMP_PROC_BIND=spread,close
./a.out
```



Cyclic distribution between “close” cores

```
export OMP_MAX_ACTIVE_LEVELS=2
export OMP_NUM_THREADS=4,4
export OMP_PLACES=threads
export OMP_PROC_BIND=spread,close
./a.out
```



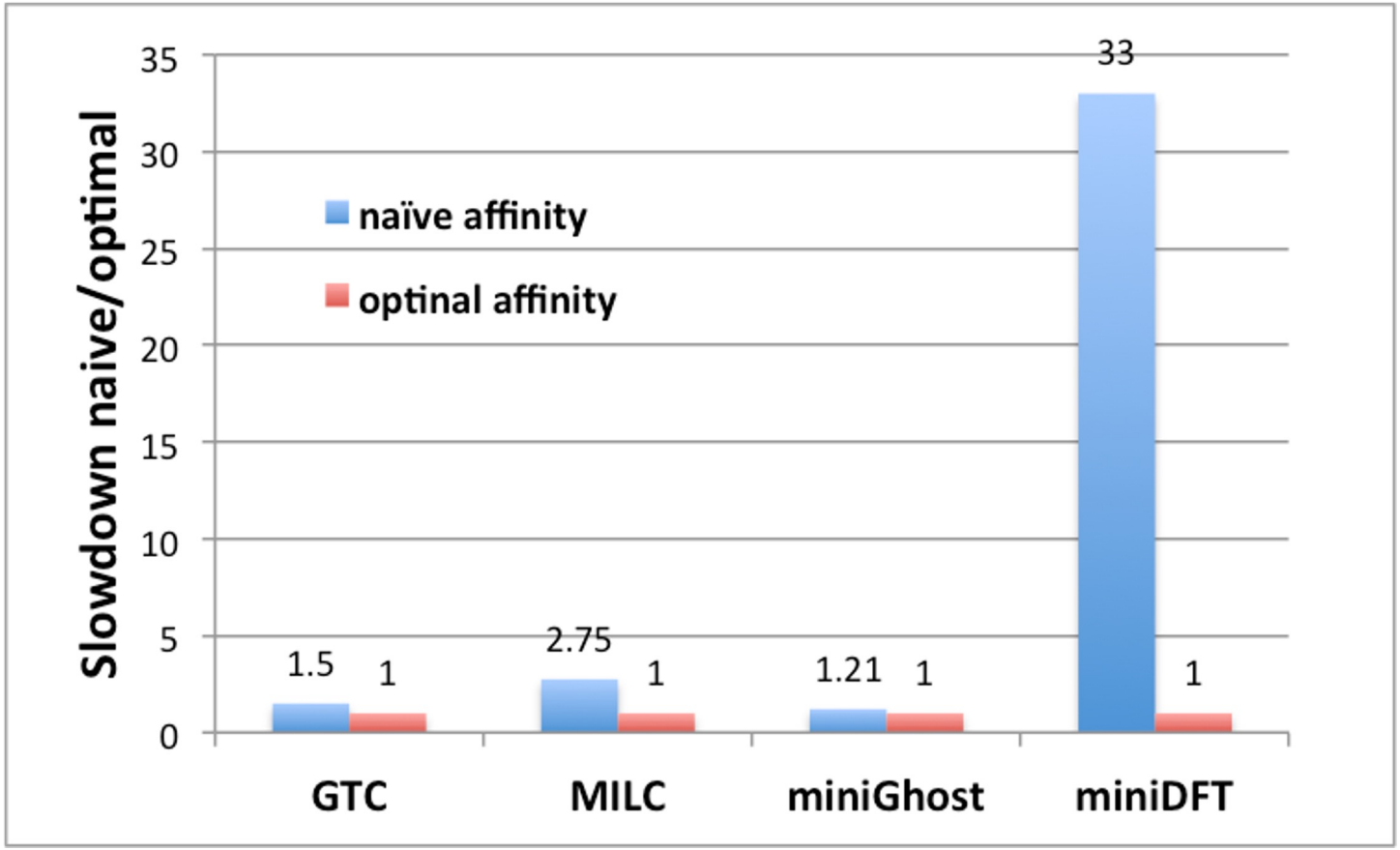
Distribution across four hardware threads

Based on content from Yun (Helen) He from NERSC)

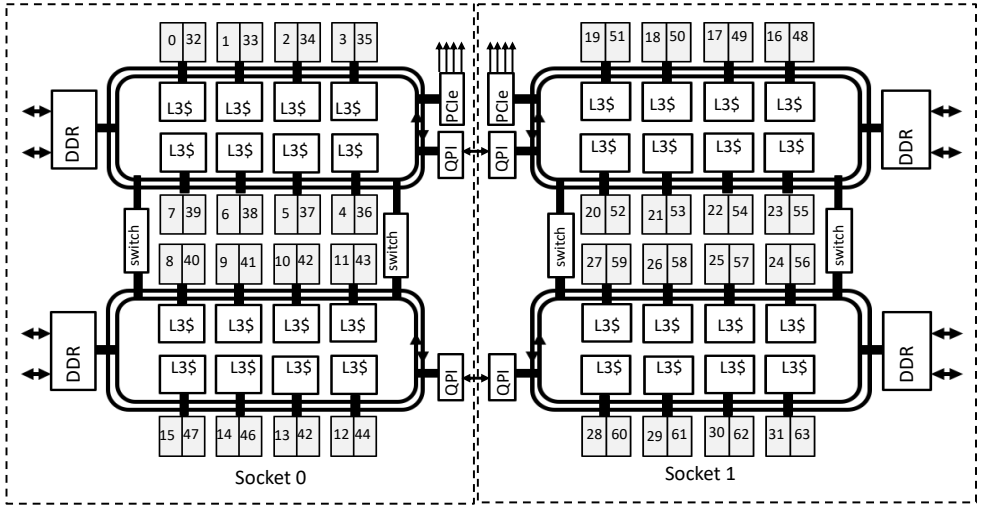
Wrapping up our discussion of taking NUMA features of a system into account in your multithreaded programs …

Getting the affinity right can have serious impacts on performance

Application Benchmark Performance for a number of benchmarks at NERSC



Lower is better



Results running on the Cori system at NERSE which has dual Socket nodes with Intel® Xeon™ E5-2698v3 CPUs

Based on content from Yun (Helen) He from NERSC (National Energy Research Supercomputing Center)

Finding the best strategy for thread affinity

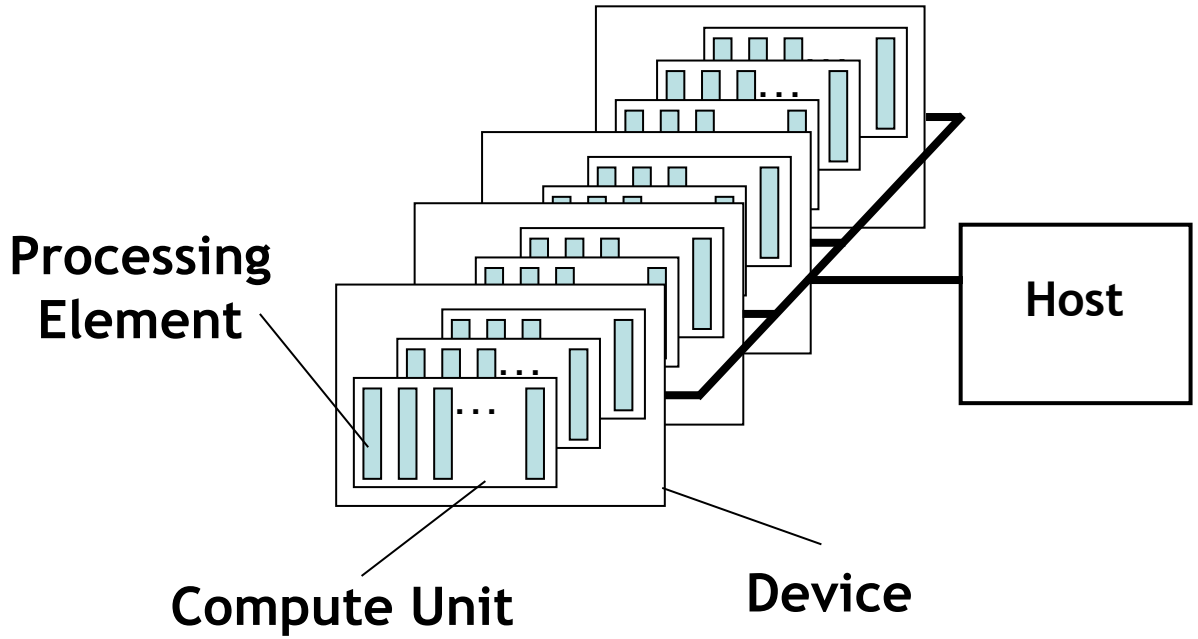
- Experiment to find the best combinations of OMP_PLACES and OMP_PROC_BIND.
  - Using the environment variables makes it easy to try many options

- The best approach depends on the system but also on the features of an application
  - Putting threads for apart … on different sockets
    - May improve aggregate memory bandwidth available to an application
    - May improve combined cache size for the application
    - May increase synchronization overhead
  - Putting threads close together … on adjacent cores that may share some caches
    - May reduce synchronization overhead
    - May decrease memory bandwidth and total cache size

- Vendors have their own constructs for controlling NUMA features of a system.
  - Avoid vendor-specific constructs if you can … use portable OMP_PLACES and OMP_PROC_BIND

Introduction to GPU programming

A Generic Host/Device Platform Model



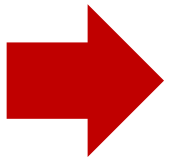
- One ***Host*** and one or more ***Devices***
  - Each Device is composed of one or more Compute Units
  - Each Compute Unit is divided into one or more ***Processing Elements***

- Memory divided into ***host memory*** and ***device memory***

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

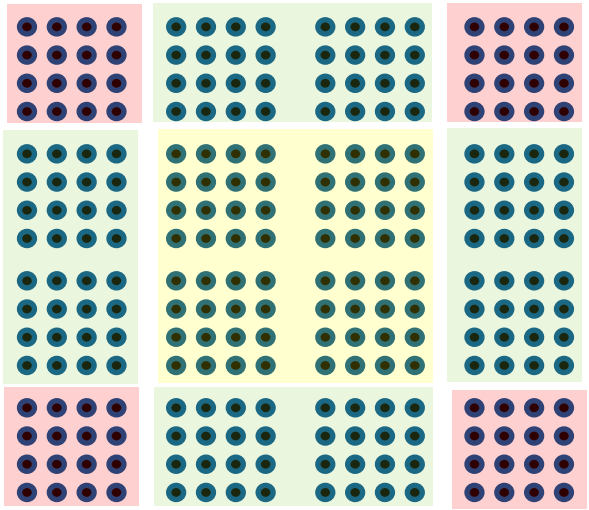
How do we execute code on a GPU: The SIMT model (Single Instruction Multiple Thread)

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.

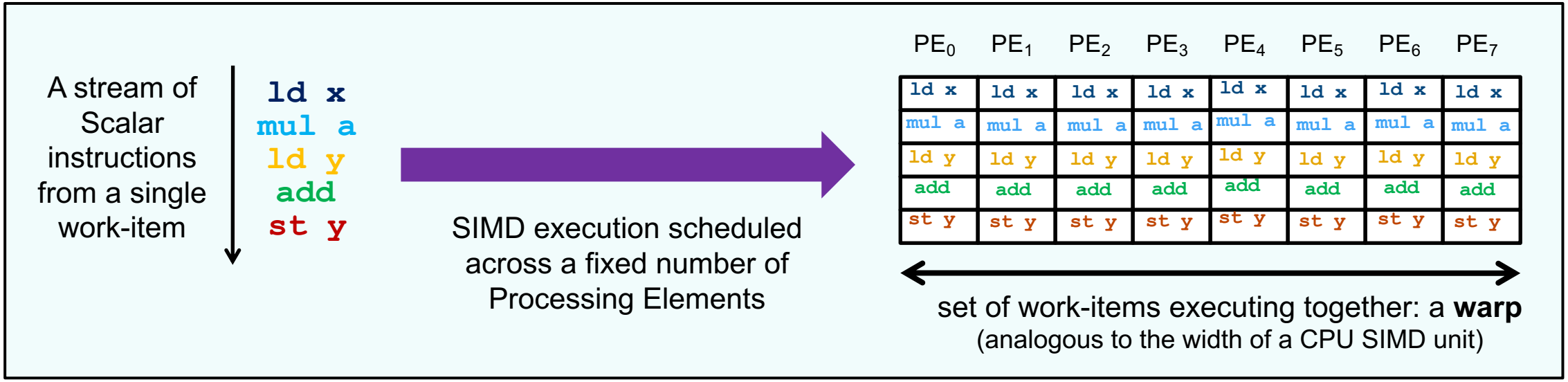


3. Map data structures onto the same index space

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

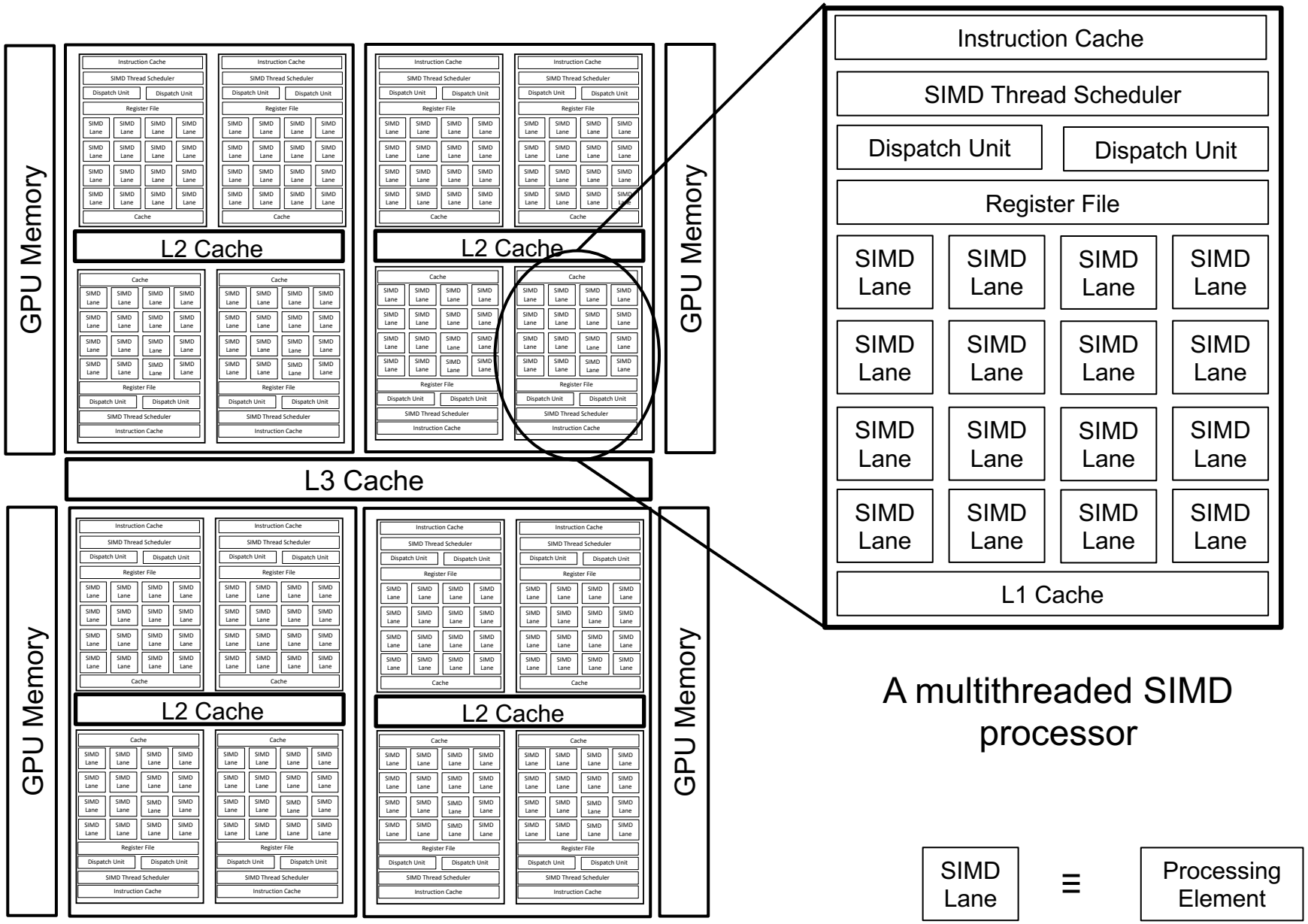


GPU nomenclature is really messed up. (sorry about that … we tried to unify around OpenCL but failed).

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.

A Generic GPU (following Hennessey and Patterson)



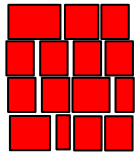
A multithreaded SIMD processor

SIMD Lane ≡ Processing Element ≡ CUDA Core

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

Executing a program on CPUs and GPUs

Work decomposed into blocks



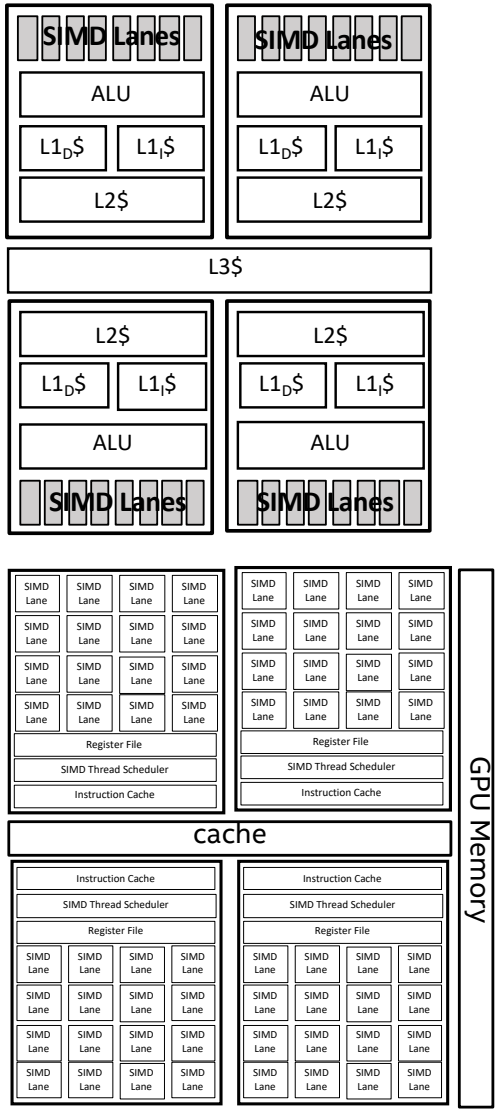
For a CPU

For a GPU

Program defines work

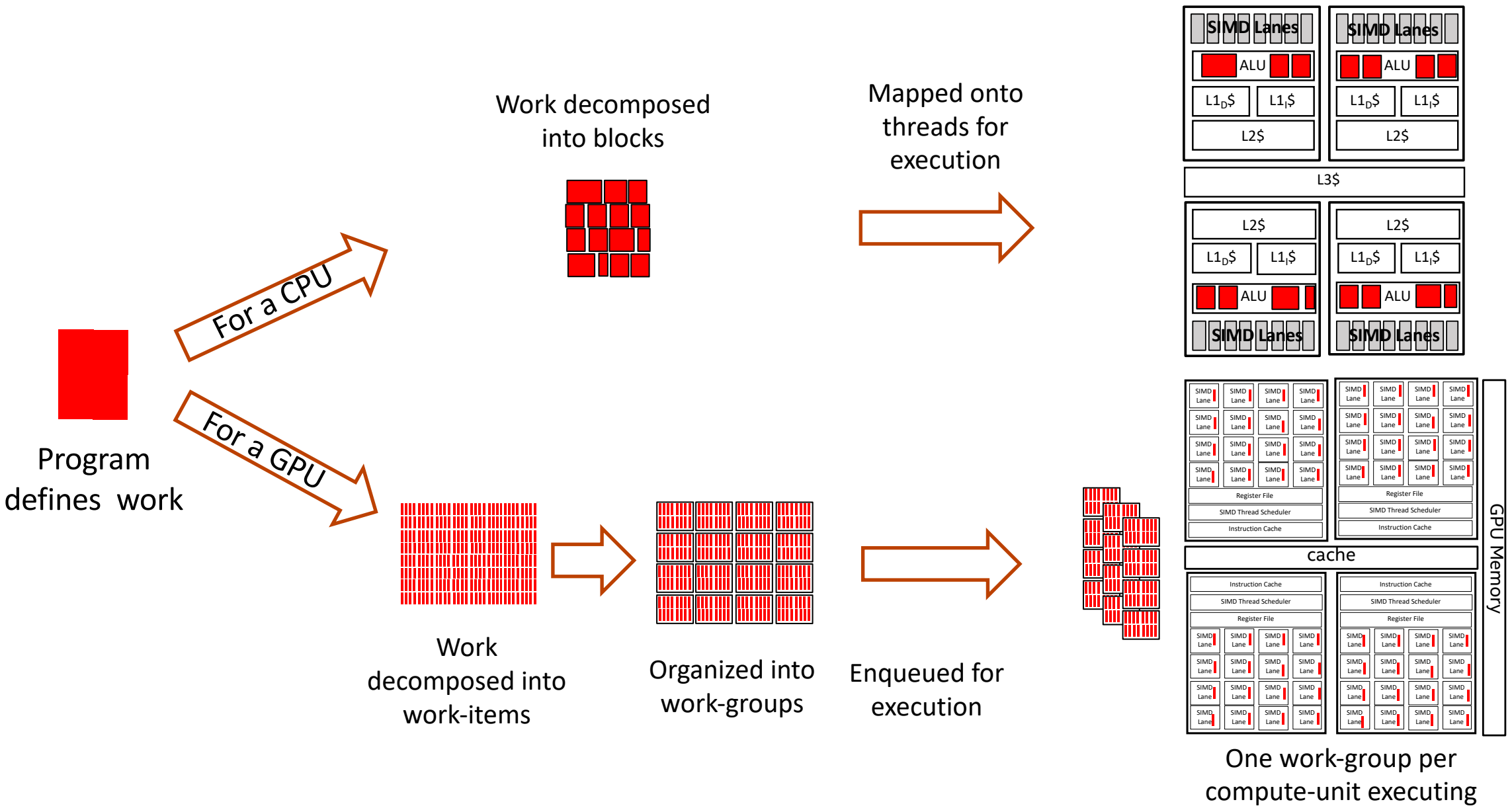
Work decomposed into work-items

Organized into work-groups



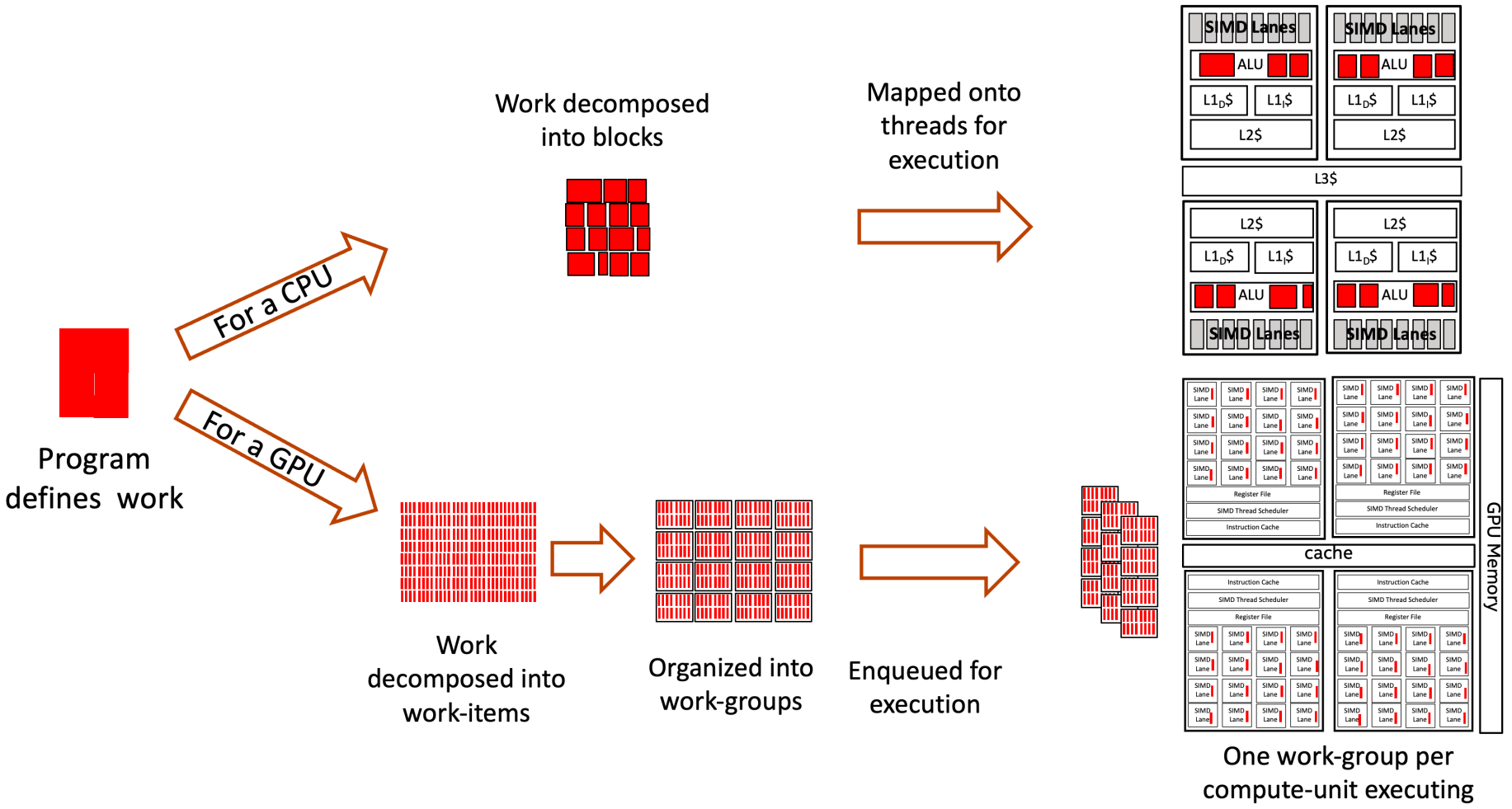
One work-group per compute-unit executing

Executing a program on CPUs and GPUs



CPU/GPU execution models

Executing a program on CPUs and GPUs

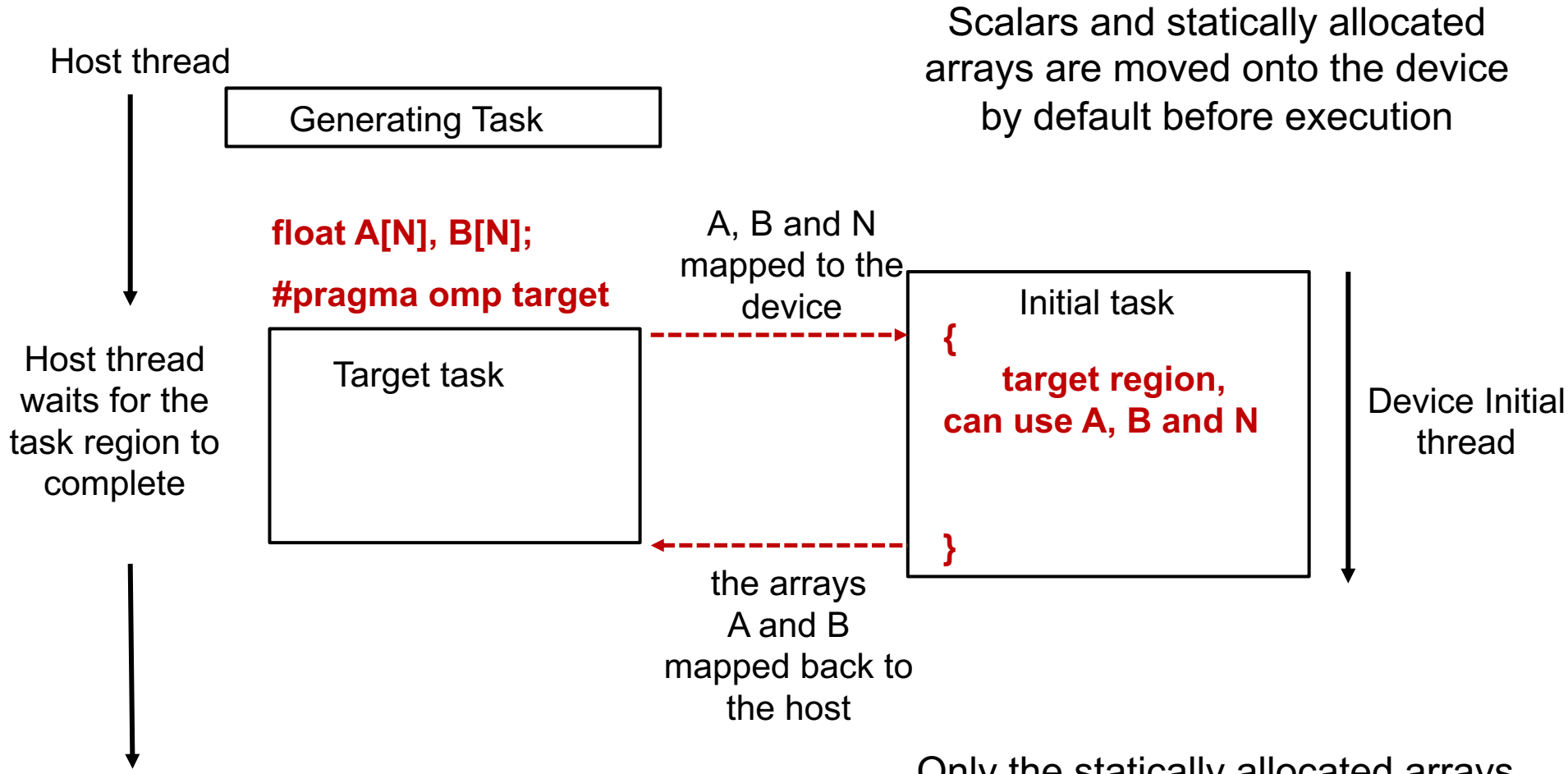


For a CPU, the threads are all active and able to make forward progress.

For a GPU, any given work-group might be in the queue waiting to execute.

Programming a GPU with OpenMP

Running code on the GPU:
The target construct and default data movement



Based on figure 6.4 in **Using OpenMP – The Next Step** by van der Pas, Stotzer and Terboven, MIT Press, 2017

Default Data Sharing: example

```
int main(void) {
  int N = 1024;
  double A[N], B[N];

  #pragma omp target
  {

    for (int ii = 0; ii < N; ++ii) {

      A[ii] = A[ii] + B[ii];

    }

  } // end of target region
}
```

1. Variables created in host memory.

2. Scalar **N** and stack arrays **A** and **B** are copied *to* device memory. Execution transferred to device.

3. **ii** is **private** on the device as it's declared within the target region

4. Execution on the device.

5. stack arrays **A** and **B** are copied *from* device memory back to the host. Host resumes execution.

Now let's run code in parallel on the device

```
int main(void) {
  int N = 1024;
  double A[N], B[N];

  #pragma omp target
  {
    #pragma omp loop
    for (int ii = 0; ii < N; ++ii) {

      A[ii] = A[ii] + B[ii];

    }

  } // end of target region
}
```

The loop construct tells the compiler:

"this loop will execute correctly if the loop iterations run in any order. You can safely run them concurrently. And the loop-body doesn't contain any OpenMP constructs. So do whatever you can to make the code run fast"

The loop construct is a declarative construct. You tell the compiler what you want done but you DO NOT tell it how to "do it". This is new for OpenMP

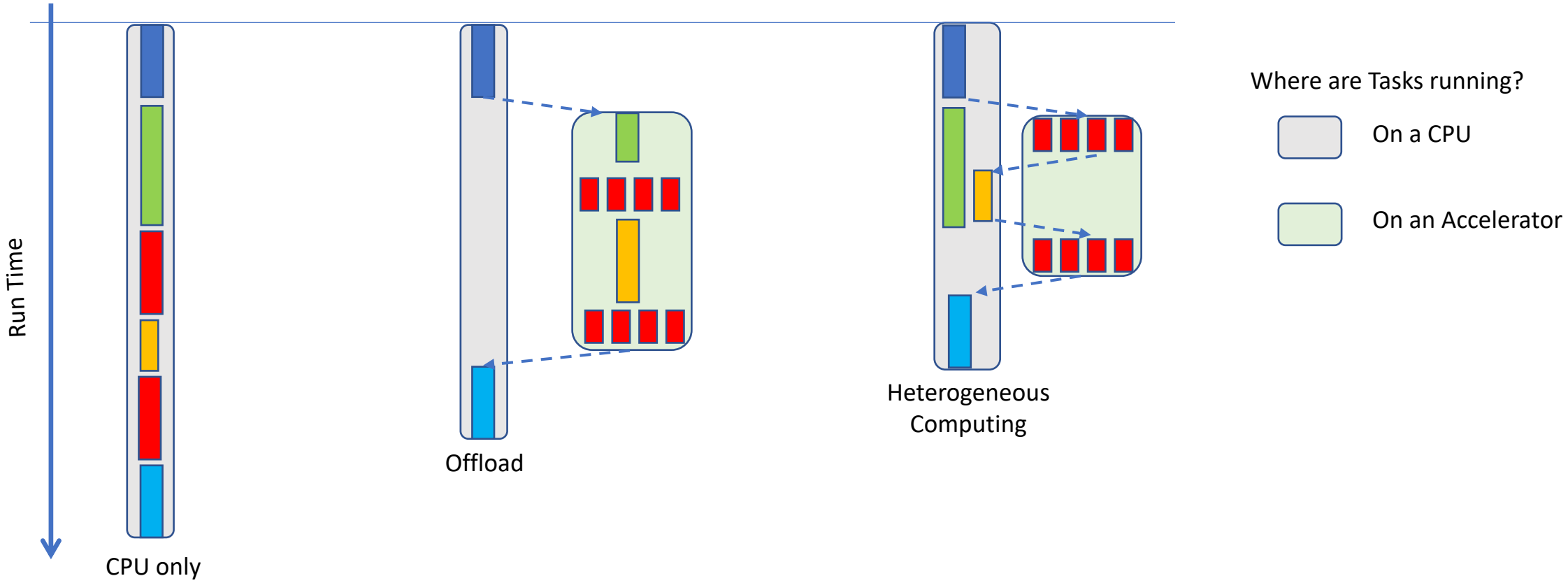
Solution: Simple vector add in OpenMP on GPU

```
int main()
{
    float a[N], b[N], c[N], res[N];
    int err=0;

   // fill the arrays
   #pragma omp parallel for
   for (int i=0; i<N; i++){
      a[i] = (float)i;
      b[i] = 2.0*(float)i;
      c[i] = 0.0;
      res[i] = i + 2*i;
   }

   // add two vectors
   #pragma omp target
   #pragma omp loop
   for (int i=0; i<N; i++){
      c[i] = a[i] + b[i];
   }

   // test results
   #pragma omp parallel for reduction(+:err)
   for(int i=0;i<N;i++){
      float val = c[i] - res[i];
      val = val*val;
      if(val>TOL) err++;
   }
   printf("vectors added with %d errors\n", err);
   return 0;
}
```

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.



5-point stencil: the heat program

- The heat equation models changes in temperature over time.

$$\frac{\partial u}{\partial t} - \alpha \nabla^2 u = 0$$

- We'll solve this numerically on a computer using an explicit **finite difference** discretisation.
- $u = u(t, x, y)$ is a function of space and time.
- Partial differentials are approximated using diamond difference formulae:

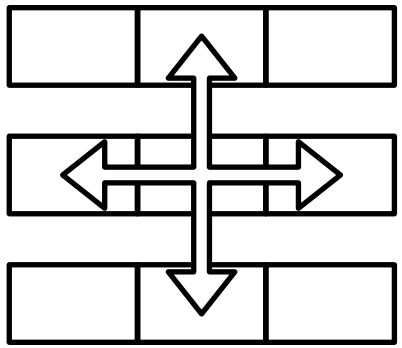
$$\frac{\partial u}{\partial t} \approx \frac{u(t+1, x, y) - u(t, x, y)}{dt}$$

$$\frac{\partial^2 u}{\partial x^2} \approx \frac{u(t, x+1, y) - 2u(t, x, y) + u(t, x-1, y)}{dx^2}$$

  - Forward finite difference in time, central finite difference in space.

5-point stencil: the heat program

- Given an initial value of u, and any boundary conditions, we can calculate the value of u at time t+1 given the value at time t.
- Each update requires values from the north, south, east and west neighbours only:



- Computation is essentially a weighted average of each cell and its neighbouring cells.
- If on a boundary, look up a boundary condition instead.

Serial CPU code

Heat diffusion problem: 5-point stencil code

```
const double r = alpha * dt / (dx * dx);
 const double r2 = 1.0 - 4.0*r;

 // malloc and initialize u_tmp and u (code not shown)

 for (int t = 0; t < nsteps; ++t) {

    for (int i = 0; i < n; ++i) {
      for (int j = 0; j < n; ++j) {
        u_tmp[i+j*n] =  r2 * u[i+j*n]          +
            r * ((i < n-1) ? u[i+1+j*n]   : 0.0) +
            r * ((i > 0)   ? u[i-1+j*n]   : 0.0) +
            r * ((j < n-1) ? u[i+(j+1)*n] : 0.0) +
            r * ((j > 0)   ? u[i+(j-1)*n] : 0.0);
      }
    }
    // Pointer swap to get ready for next step
    tmp = u;
    u = u_tmp;
    u_tmp = tmp;
 }
}
```

Loop over time steps

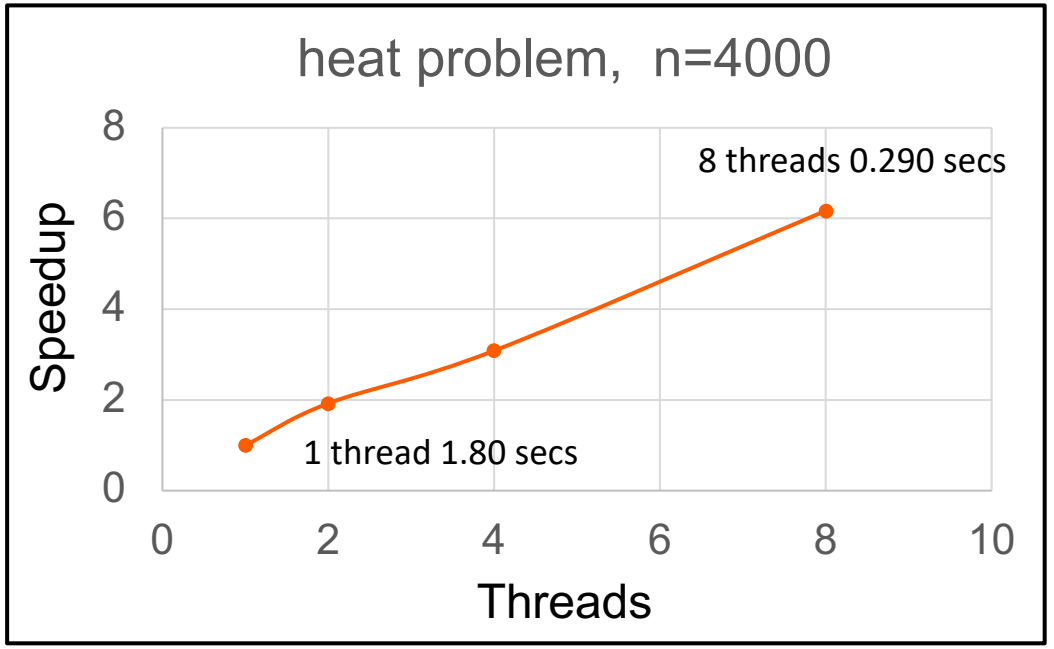
Loop over NxN spatial domain

Update the 5-point stencil. Boundary conditions on the edges of the domain are fixed at zero.

Heat diffusion problem: 5-point stencil code

Parallel CPU code, n=4000

```
const double r = alpha * dt / (dx * dx);
const double r2 = 1.0 - 4.0*r;

// malloc and initialize u_tmp and u (code not shown)

for (int t = 0; t < nsteps; ++t) {

   #pragma omp parallel for collapse(2)
   for (int i = 0; i < n; ++i) {
     for (int j = 0; j < n; ++j) {
       u_tmp[i+j*n] =  r2 * u[i+j*n]          +
           r * ((i < n-1) ? u[i+1+j*n]   : 0.0) +
           r * ((i > 0)   ? u[i-1+j*n]   : 0.0) +
           r * ((j < n-1) ? u[i+(j+1)*n] : 0.0) +
           r * ((j > 0)   ? u[i+(j-1)*n] : 0.0);
     }
    }
    // Pointer swap to get ready for next step
    tmp = u;
    u = u_tmp;
    u_tmp = tmp;
  }
}
```



Intel® Xeon™ Gold 5218 @ 2.3 Ghz, 8 cores.
Nvidia HPC Toolkit compiler nvc –fast –fopenmp heat.c

Heat diffusion problem: 5-point stencil code

Parallel GPU code, n=4000

```
const double r = alpha * dt / (dx * dx);
const double r2 = 1.0 - 4.0*r;

// malloc and initialize u_tmp and u (code not shown)

for (int t = 0; t < nsteps; ++t) {
   #pragma omp target map(tofrom: u[0:n*n], u_tmp[0:n*n])
   #pragma omp loop
   for (int i = 0; i < n; ++i) {
     for (int j = 0; j < n; ++j) {
       u_tmp[i+j*n] =  r2 * u[i+j*n]          +
           r * ((i < n-1) ? u[i+1+j*n]   : 0.0) +
           r * ((i > 0)   ? u[i-1+j*n]   : 0.0) +
           r * ((j < n-1) ? u[i+(j+1)*n] : 0.0) +
           r * ((j > 0)   ? u[i+(j-1)*n] : 0.0);
     }
    }
    // Pointer swap to get ready for next step
    tmp = u;
    u = u_tmp;
    u_tmp = tmp;
  }
}
```

When you map pointers between the host and the device, OpenMP remembers the address.

Swapped addresses on the hosts swaps addresses on the device

```
GPU Solver time = 1.40 secs
```

This isn't much better than the runtime for a single CPU (1.8 secs) and worse than 8 cores on a CPU (0.29 secs).

Why is the performance so bad?

```
NVIDIA T4 GPU, 16 Gbyte, Turing Arch.
Nvidia HPC Toolkit compiler
nvc -fast -mp=gpu -gpu=cc75 heat.c
```

Parallel GPU code, n=4000

Heat diffusion problem: 5-point stencil code

```
const double r = alpha * dt / (dx * dx);
const double r2 = 1.0 - 4.0*r;

// malloc and initialize u_tmp and u (code not shown)

for (int t = 0; t < nsteps; ++t) {
   #pragma omp target map(tofrom: u[0:n*n], u_tmp[0:n*n])
   #pragma omp loop
   for (int i = 0; i < n; ++i) {
     for (int j = 0; j < n; ++j) {
       u_tmp[i+j*n] =  r2 * u[i+j*n]          +
           r * ((i < n-1) ? u[i+1+j*n]   : 0.0) +
           r * ((i > 0)   ? u[i-1+j*n]   : 0.0) +
           r * ((j < n-1) ? u[i+(j+1)*n] : 0.0) +
           r * ((j > 0)   ? u[i+(j-1)*n] : 0.0);
     }
   }
   // Pointer swap to get ready for next step
   tmp = u;
   u = u_tmp;
   u_tmp = tmp;
}
}
```

With a runtime of 1.4 secs (worse than the CPU time) we see that Data Movement dominates performance.

At the beginning of each iteration, copy $(2*N^2)*sizeof(TYPE)$ bytes to the device

We need to create a **data region** on the GPU that is distinct from the target region.

That way, we can keep the data on the device between target constructs

At the end of each iteration, copy $(2*N^2)*sizeof(TYPE)$ bytes from the device

Target enter/exit data constructs

- Create a data region on the target device (a device data environment) with two standalone directives:

```
#pragma omp target enter data map(…)
#pragma omp target exit data map(…)
```

- The **target enter data** maps variables to the device data environment.

- The **target exit data** unmaps variables from the device data environment.

- Once created, subsequent **target** regions inherit the existing data environment.

Target enter/exit data example

```
void init_array(int *A, int *B, int N) {
   for (int i = 0; i < N; ++i) { A[i] = i;  B[i]=2*I;}

  #pragma omp target enter data map(to: A[0:N], B[0:N])
}

int main(void) {

   int N = 1024;
   int *A = malloc(sizeof(int) * N);
   int *B = malloc(sizeof(int) * N);
   init_array(A, B, N);

   #pragma omp target
   #pragma omp loop
   for (int i = 0; i < N; ++i)
      A[i] = A[i] * B[i];

 #pragma omp target exit data map(from: A[0:N])
}
```

Heat diffusion problem: 5-point stencil code

Parallel GPU code, n=4000

```
  const double r = alpha * dt / (dx * dx);
  const double r2 = 1.0 - 4.0*r;
  // malloc and initialize u_tmp and u (code not shown)
  #pragma omp target enter data map(to: u[0:n*n], u_tmp[0:n*n])

  for (int t = 0; t < nsteps; ++t) {
     #pragma omp target
     #pragma omp loop
     for (int i = 0; i < n; ++i) {
       for (int j = 0; j < n; ++j) {
         u_tmp[i+j*n] =  r2 * u[i+j*n]          +
             r * ((i < n-1) ? u[i+1+j*n]   : 0.0) +
             r * ((i > 0)   ? u[i-1+j*n]   : 0.0) +
             r * ((j < n-1) ? u[i+(j+1)*n] : 0.0) +
             r * ((j > 0)   ? u[i+(j-1)*n] : 0.0);
       }
     }
     // Pointer swap to get ready for next step
     tmp = u;
     u = u_tmp;
     u_tmp = tmp;
  }
 #pragma omp target exit data map(from: u[0:n*n])
}
```

Create a data region and map indicated data on entry

GPU Solver time* = **0.42** secs

This is a general principal … if you want performance, you must optimize data movement.

*includes time for target enter/exit data

Exit the data region and map indicated data

NVIDIA T4 GPU, 16 Gbyte, Turing Arch.
Nvidia HPC Toolkit compiler
nvc -fast -mp=gpu -gpu=cc75 heat.c

Heat diffusion problem: 5-point stencil code

Parallel CPU results, n=4000

```
const double r = alpha * dt / (dx * dx);
const double r2 = 1.0 - 4.0*r;
// malloc and initialize u_tmp and u (code not shown)


for (int t = 0; t < nsteps; ++t) {

    #pragma omp parallel for
    for (int i = 0; i < n; ++i) {
      for (int j = 0; j < n; ++j) {
        u_tmp[i+j*n] =  r2 * u[i+j*n]          +
            r * ((i < n-1) ? u[i+1+j*n]   : 0.0) +
            r * ((i > 0)   ? u[i-1+j*n]   : 0.0) +
            r * ((j < n-1) ? u[i+(j+1)*n] : 0.0) +
            r * ((j > 0)   ? u[i+(j-1)*n] : 0.0);
      }
     }
     // Pointer swap for  next step
     tmp = u;
     u = u_tmp;
     u_tmp = tmp;
  }

}
```

This is the ij loop order.

Let's optimize the CPU code as well

CPU

Num threads	ij loop order
1	1.512849
2	0.776229
4	0.400822
8	0.227317

Intel® Xeon™ Gold 5218 @ 2.3 Ghz, 8 cores.
Nvidia HPC Toolkit compiler nvc –fast –fopenmp heat.c

All times in seconds

Heat diffusion problem: 5-point stencil code

Parallel CPU results, n=4000

```
const double r = alpha * dt / (dx * dx);
const double r2 = 1.0 - 4.0*r;
// malloc and initialize u_tmp and u (code not shown)


for (int t = 0; t < nsteps; ++t) {

    #pragma omp parallel for
    for (int j = 0; j < n; ++j) {
      for (int i = 0; i < n; ++i) {
        u_tmp[i+j*n] =  r2 * u[i+j*n]          +
            r * ((i < n-1) ? u[i+1+j*n]   : 0.0) +
            r * ((i > 0)   ? u[i-1+j*n]   : 0.0) +
            r * ((j < n-1) ? u[i+(j+1)*n] : 0.0) +
            r * ((j > 0)   ? u[i+(j-1)*n] : 0.0);
      }
    }
    // Pointer swap for  next step
    tmp = u;
    u = u_tmp;
    u_tmp = tmp;
}

}
```

This is the ji loop order. Swap these loops to get the ij order.

Make j the outermost loop so adjacent loop iterations access adjacent memory locations.

CPU

Num threads	ij loop order	ji loop order
1	1.512849	0.262260
2	0.776229	0.132453
4	0.400822	0.064220
8	0.227317	0.046586

Intel® Xeon™ Gold 5218 @ 2.3 Ghz, 8 cores.
Nvidia HPC Toolkit compiler nvc –fast –fopenmp heat.c

All times in seconds

This is particularly important on a GPU … you want memory coalesced with the GPUs processing elements (PE) … i.e., elements of u accessed by PE_i should be adjacent to the elements of u accessed by PE_{i+1}

Heat diffusion problem: 5-point stencil code

Parallel CPU and GPU results, n=4000

```
const double r = alpha * dt / (dx * dx);
const double r2 = 1.0 - 4.0*r;
// malloc and initialize u_tmp and u (code not shown)
#pragma omp target enter data map(to: u[0:n*n], u_tmp[0:

for (int t = 0; t < nsteps; ++t) {
    #pragma omp target
    #pragma omp loop
    for (int j = 0; j < n; ++j) {
      for (int i = 0; i < n; ++i) {
        u_tmp[i+j*n] =  r2 * u[i+j*n]          +
            r * ((i < n-1) ? u[i+1+j*n]   : 0.0) +
            r * ((i > 0)   ? u[i-1+j*n]   : 0.0) +
            r * ((j < n-1) ? u[i+(j+1)*n] : 0.0) +
            r * ((j > 0)   ? u[i+(j-1)*n] : 0.0);
      }
    }
    // Pointer swap
    tmp = u;
    u = u_tmp;
    u_tmp = tmp;
  }
 #pragma omp target exit data map(from: u[0:n*n])
}
```

This is the ji loop order.

Memory coalescence is important for CPUs and GPUs.

Note: collapse(2) did not help on the GPU or the CPU

CPU

Num threads	ij loop order	ji loop order
1	1.512849	0.262260
2	0.776229	0.132453
4	0.400822	0.064220
8	0.227317	0.046586

Intel® Xeon™ Gold 5218 @ 2.3 Ghz, 8 cores.
Nvidia HPC Toolkit compiler nvc –fast –fopenmp heat.c

All times in seconds

GPU

ij without timing enter and exit data	ij loop order	ji without timing enter and exit data	ji loop order
0.056830	0.417887	0.020123	0.358905

NVIDIA T4 GPU, 16 Gbyte, Turing Arch.
Nvidia HPC Toolkit compiler. nvc -fast -mp=gpu heat.c

Outline

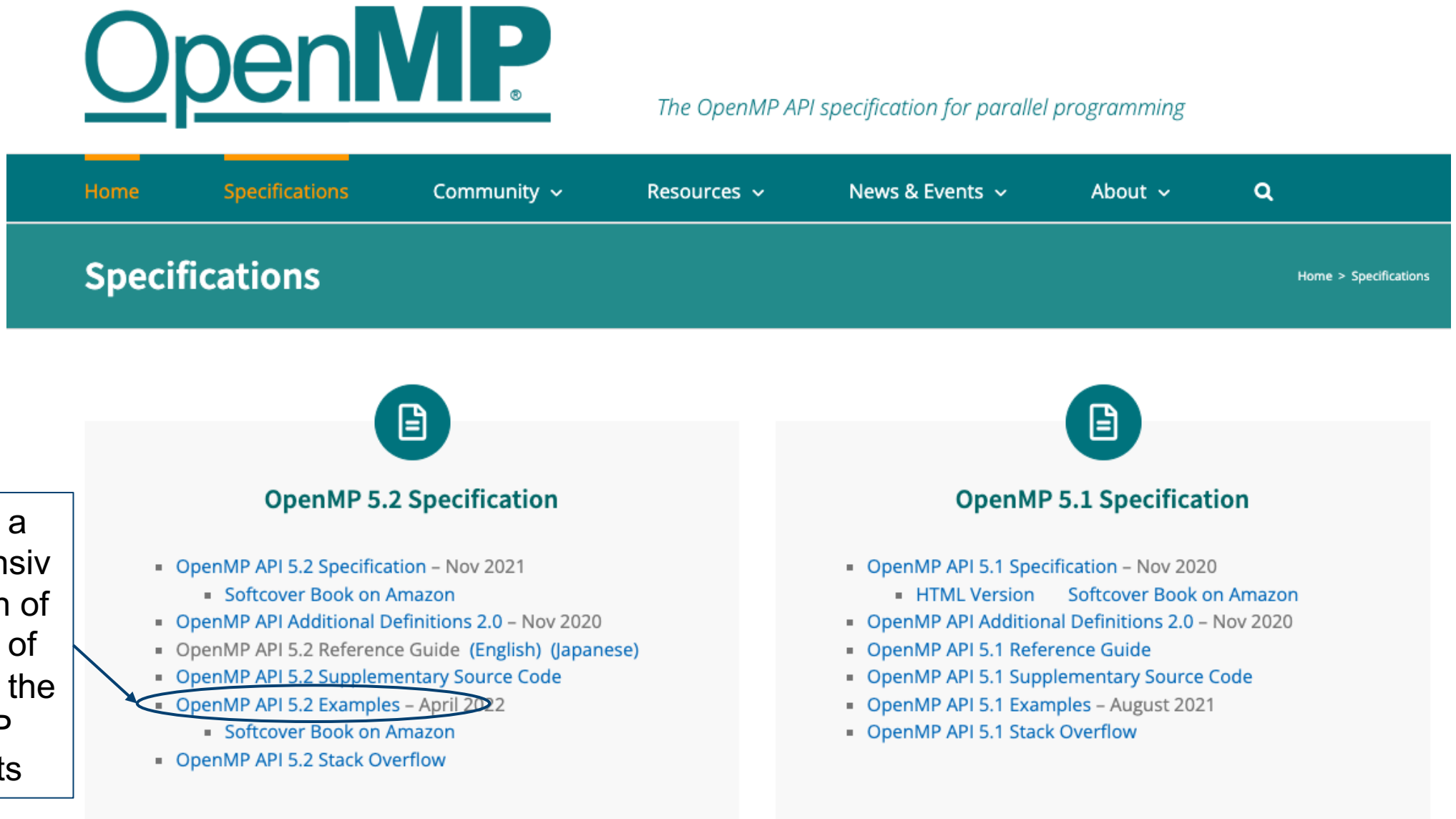
- Introduction to OpenMP
- Creating Threads
- Synchronization
- Parallel Loops
- Data Environment
- Memory Model
- Irregular Parallelism and Tasks
- Recap

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.

Resources

- The OpenMP Architecture review Board (ARB) has a wealth of helpful resources on its web site: www.openmp.org



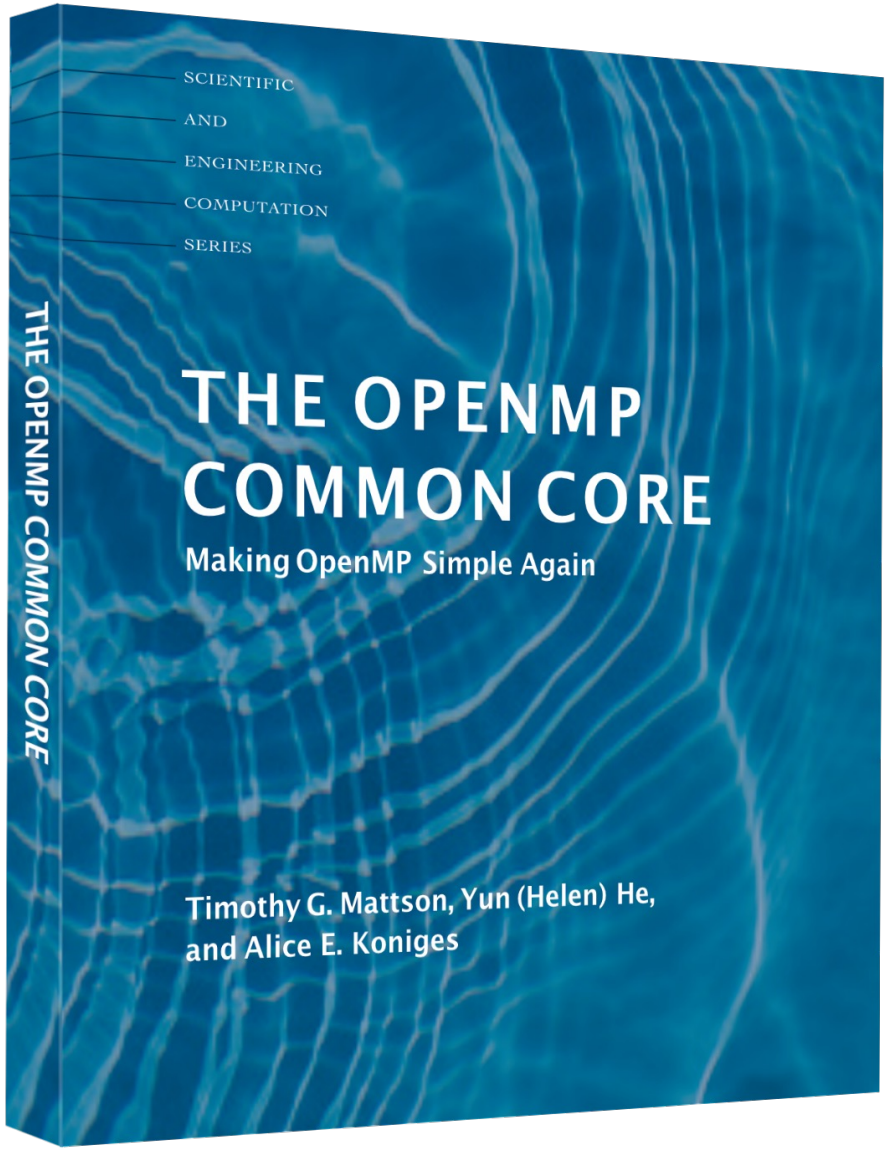
Including a comprehensive collection of examples of code using the OpenMP constructs

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

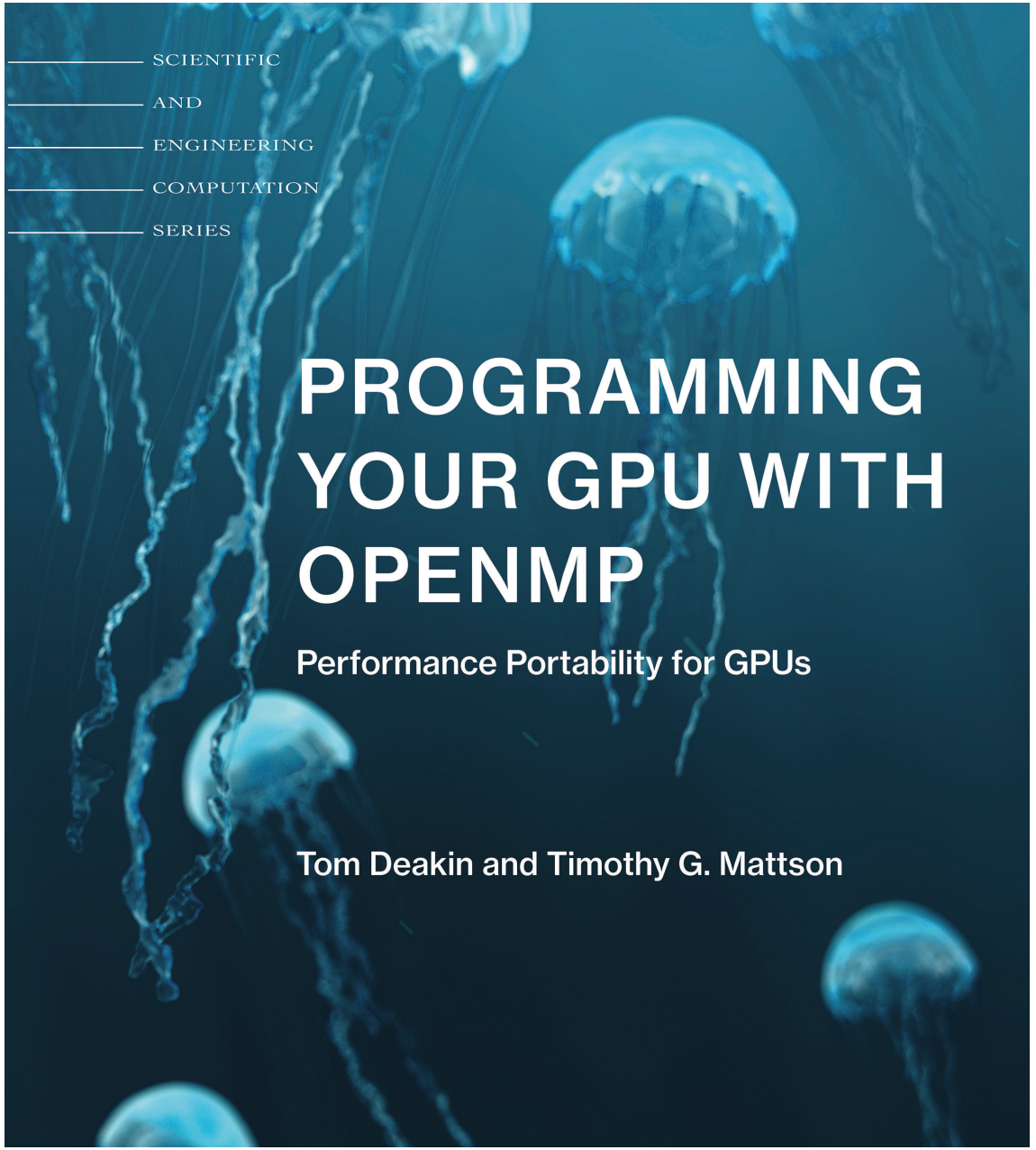
A great book that covers OpenMP features beyond OpenMP 2.5

Books about OpenMP

The latest book on OpenMP …

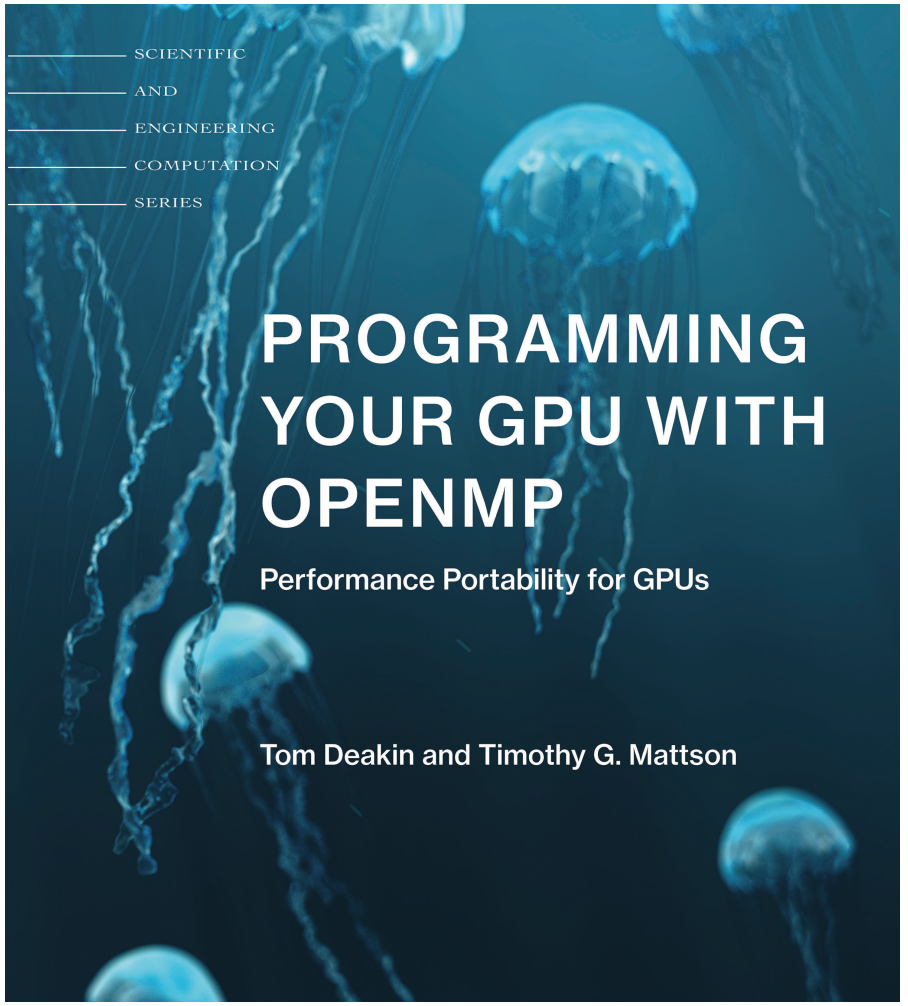
Released in November 2023.

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



GPU programming with OpenMP

- There is much more … which you can learn about from our book
  - Loop is a descriptive construct … you leave all the details to the runtime. Always start with Loop plus enter-data/exit-data since often that is all you need
  - OpenMP includes constructs for detailed control of the GPU so you can do programing akin to that with CUDA. I do not recommend this. You maximize portability if you let the runtime system handle mapping code onto hardware details for you. But if you want to control local memories, you may have no choice.
  - The interop constructs let you call functions native to a particular GPU (such as BLAS) from inside the OpenMP program. They are a bit complicated to work with. See our book to learn more.



Learn all the details of GPU programming with OpenMP (up to version 5.2) . Released in November 2023

Exercises to play with during consolidation

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?**

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)
```

You saw my solution to this problem (without using tasks). Try and come up with some additional solutions. There are many ways to do this, so get creative.

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