

An Introduction to Parallel Programming:

A hands-on Introduction (using OpenMP)

Tim Mattson
Human Learning Group
(retired from Intel Aug'2023)

With a lot of help from Helen He, Alice Koniges, David Eder, and many more

Download tutorial materials onto your laptop:
git clone <https://github.com/tgmattso/ParProgForPhys.git>

Introduction

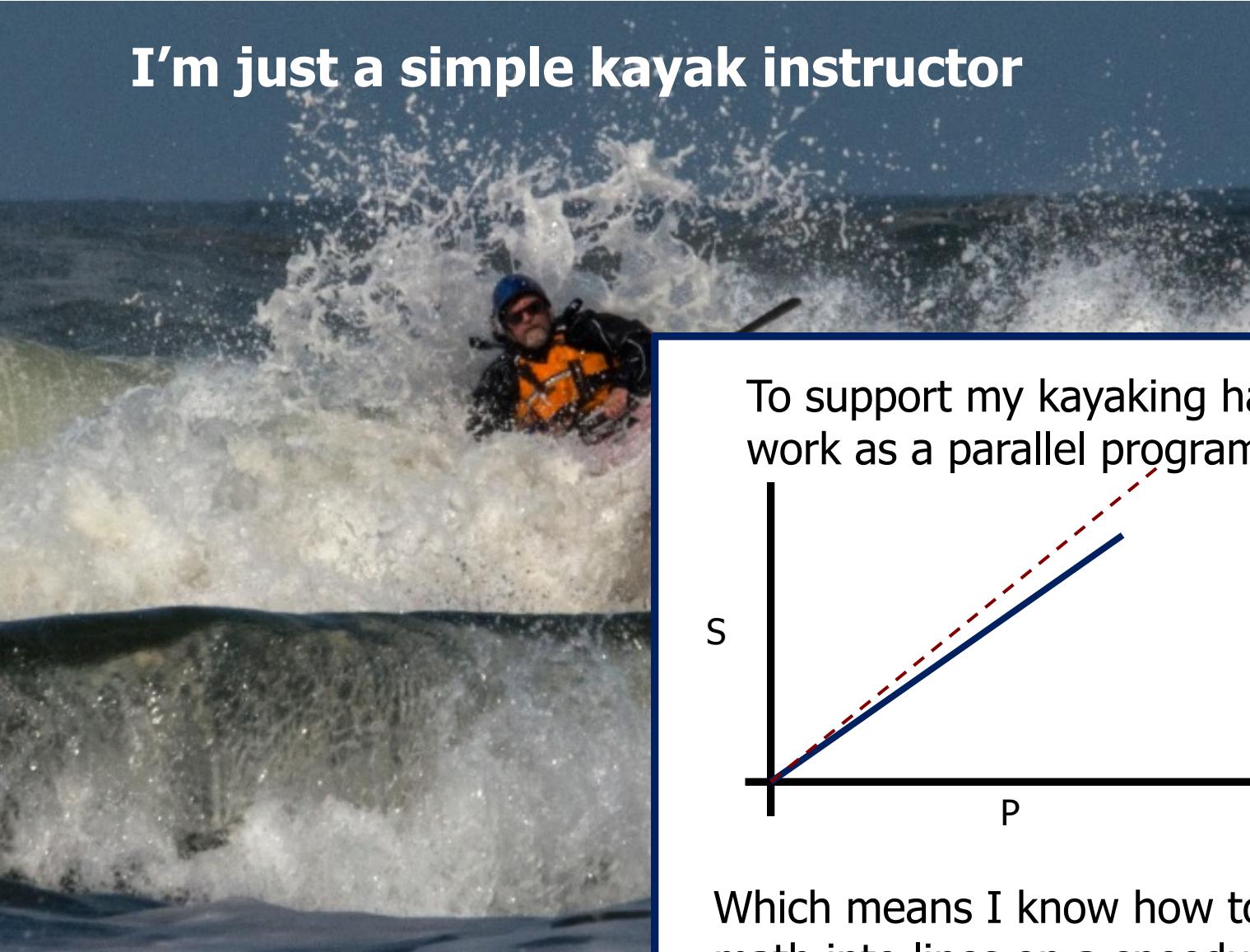
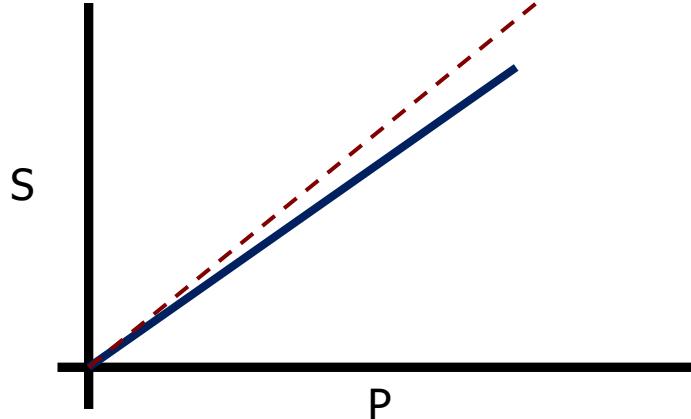


Photo © by Greg Clopton, 2014

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

Preliminaries: Part 1

- Disclosures
 - The views expressed in this tutorial are those of the people delivering the tutorial.
 - We are not speaking for the OpenMP ARB
- We take these tutorials VERY seriously:
 - Help us improve ... tell us how you would make this tutorial better.

Preliminaries: Part 2

- Our plan for the day .. Active learning!
 - We will mix short lectures with short exercises.
 - You will use your laptop to connect to a multiprocessor server.
- Please follow these simple rules
 - Do the exercises that we assign and then change things around and experiment.
 - Embrace active learning!
 - Don't cheat: Do Not look at the solutions before you complete an exercise ... even if you get really frustrated.

Download tutorial materials onto your laptop:
git clone <https://github.com/tgmattso/ParProgForPhys.git>

Use homebrew to install gnu compilers on your Apple laptop

I tested this on a new
(July 2023) MacBook
Air with an Apple M2
CPU

Warning: by default Xcode uses the name gcc for Apple's clang compiler.
Use Homebrew to load a real, gcc compiler.

- Go to the homebrew web site (brew.sh). Cut and paste the command near the top of the page to install homebrew (in /opt/homebrew):

```
/bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"
```

- Add /opt/homebrew/bin to your path. I did this by adding the following line to .zshrc

```
% export PATH=/opt/homebrew/bin:$PATH
```

- Install the latest gcc compiler

```
% brew install gcc
```

- This will install the compiler in /opt/homebrew/bin. Check /opt/homebrew/bin to see which gcc compiler was installed. In my case, it installed gcc-13
- Test the compiler (and the openmp option) with a simple hello world program

```
% gcc-13 --openmp hello.c
```

OpenMP Compilers on Apple Laptops: MacPorts

- To use OpenMP on your Apple laptop:
- Download Xcode. Be sure to choose the command line tools that match our OS.
- Download and use MacPorts to install the latest gnu compilers.

I have not tested this in a long time.
I greatly prefer homebrew.

But if you prefer MacPorts, this procedure
should work.

```
sudo port selfupdate
```

Update to latest version of
MacPorts

```
sudo port install gcc13
```

Grab version 13 gnu
compilers (5-10 mins)

```
port select --list gcc
```

List versions of gcc on your
system

```
sudo port select --set gcc mp-gcc13
```

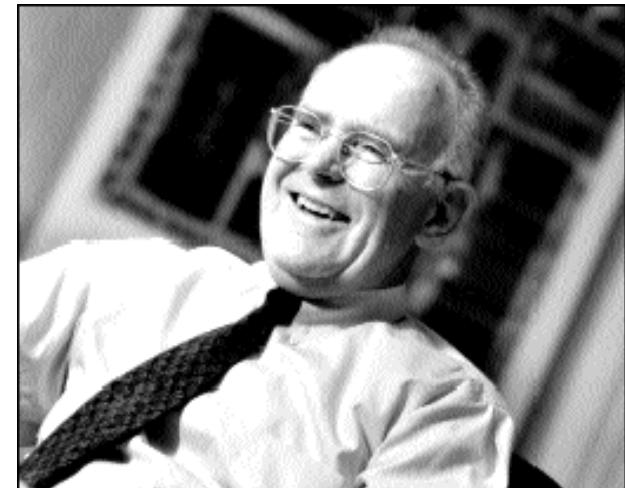
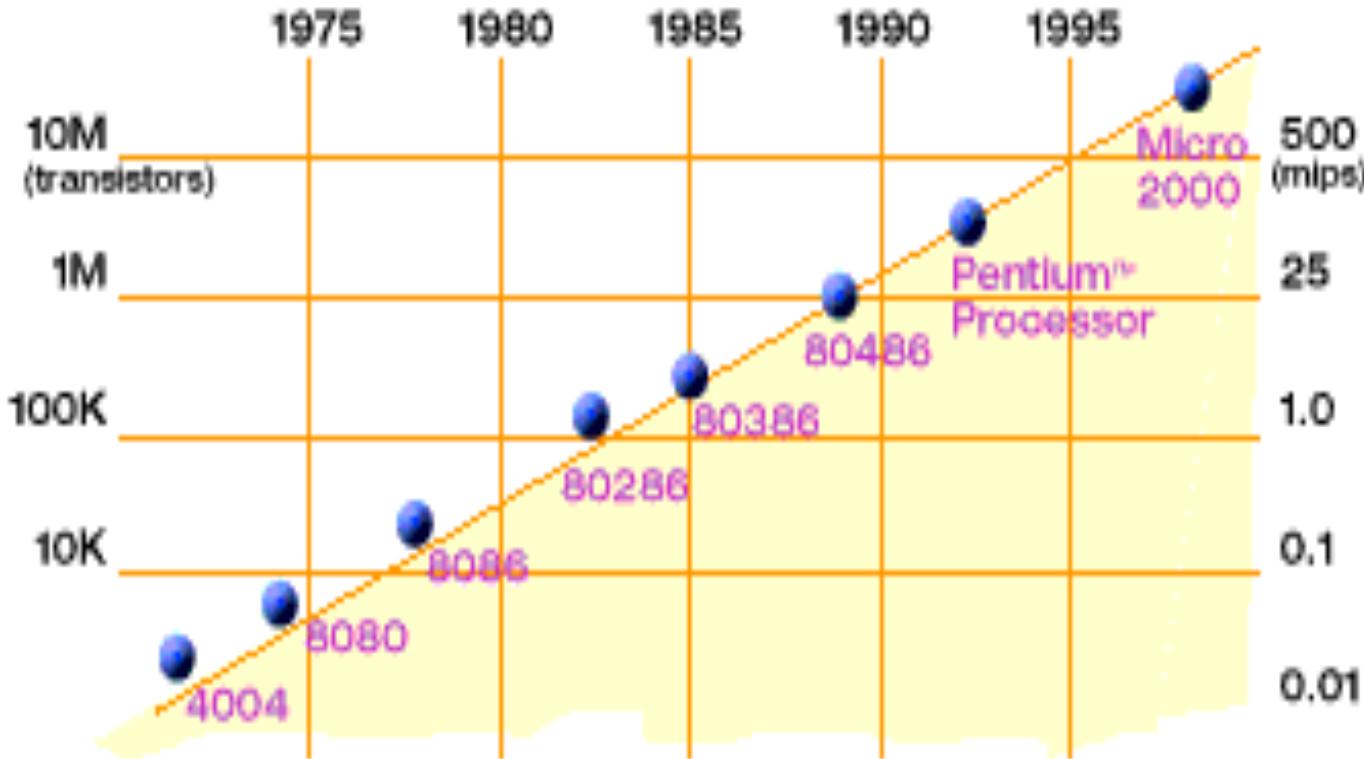
Select the mp enabled version of
the most recent gcc release

```
gcc -fopenmp hello.c
```

Test the installation with a simple
program

Download tutorial materials onto your laptop:
git clone <https://github.com/tgmattso/ParProgForPhys.git>

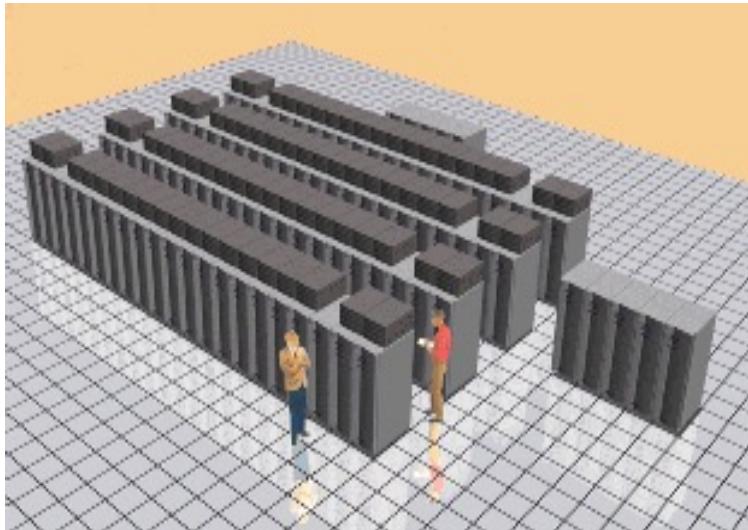
Moore's Law



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

Moore's Law: A personal perspective

First TeraScale* computer: 1997



Intel's ASCI Option Red

Intel's ASCI Red Supercomputer

9000 CPUs

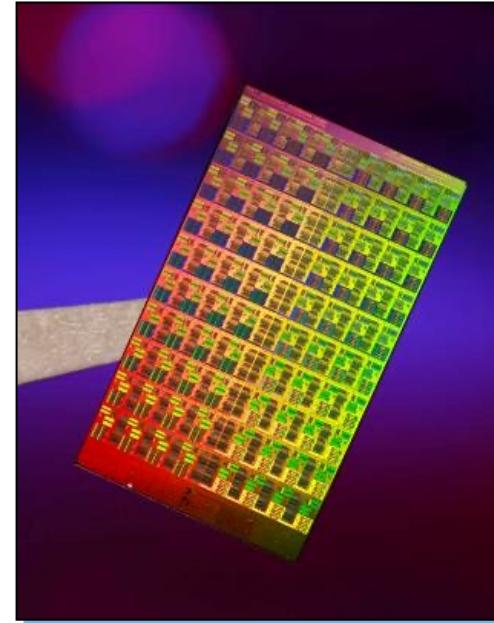
one megawatt of electricity.

1600 square feet of floor space.

*Double Precision TFLOPS running MP-Linpack

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

First TeraScale% chip: 2007



Intel's 80 core teraScale Chip

1 CPU

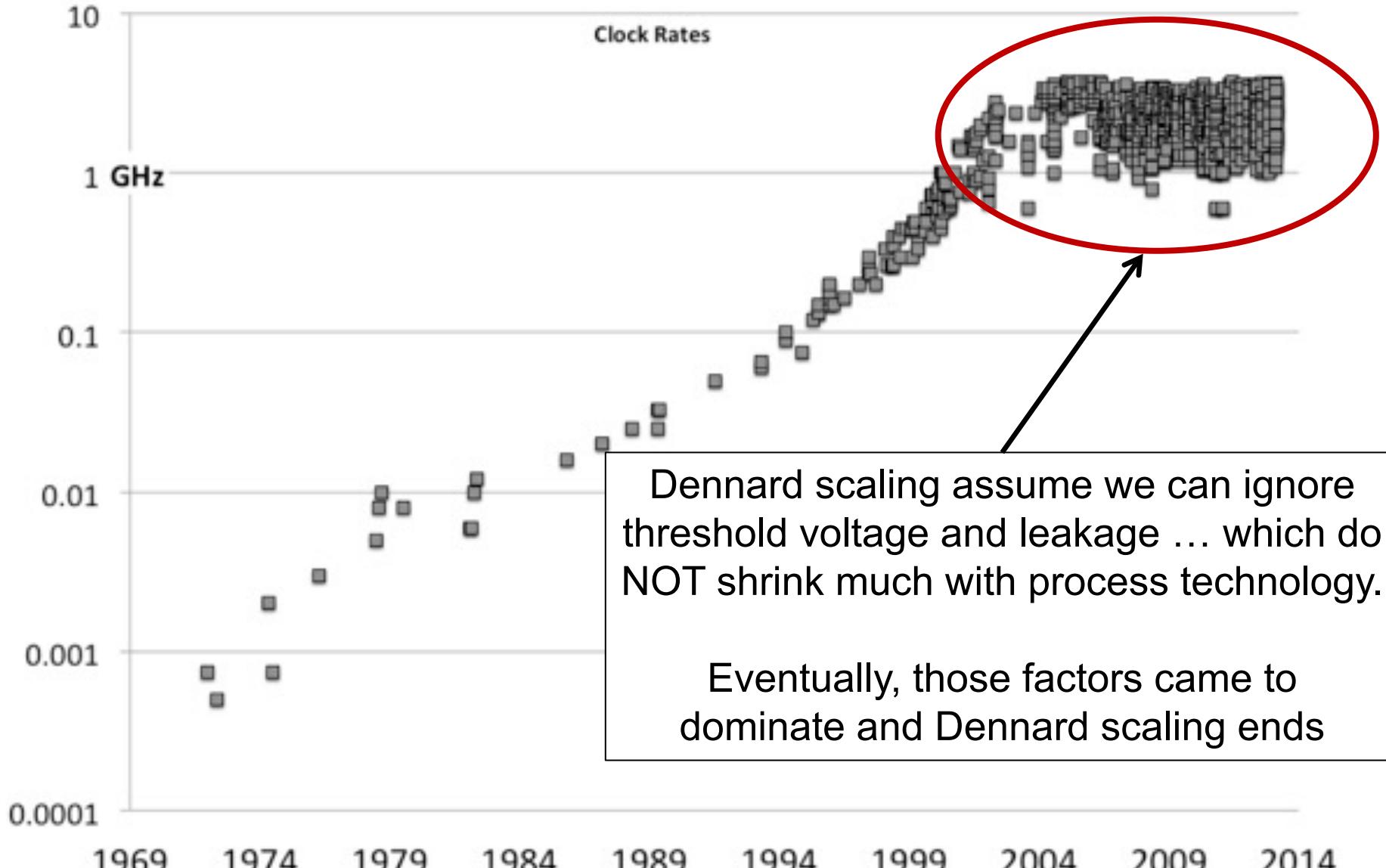
97 watt

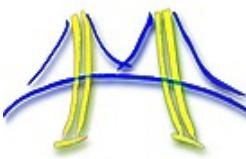
275 mm²

%Single Precision TFLOPS running stencil

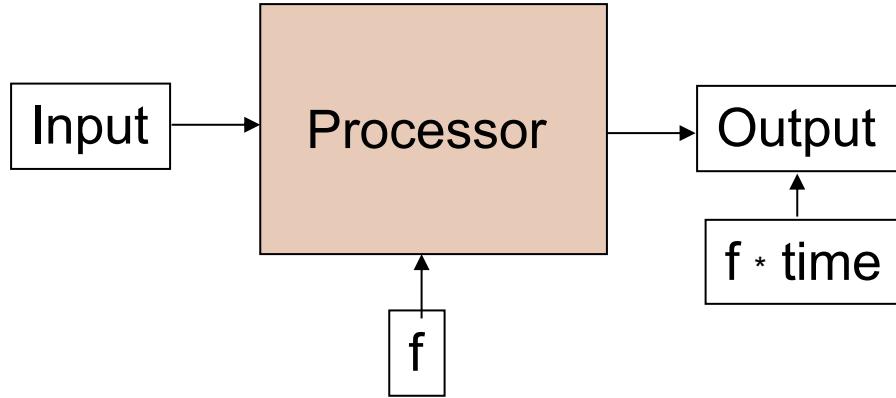
Programming Intel's 80 core terascale processor
SC08, Austin Texas, Nov. 2008, Tim Mattson,
Rob van der Wijngaart, Michael Frumkin

CPU Frequency (GHz) over time (years)





Consider power in a chip ...



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

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

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

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

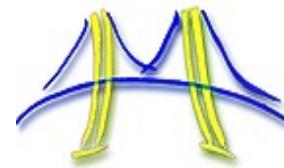
$$V * q = W.$$

But for a circuit $q = CV$ so

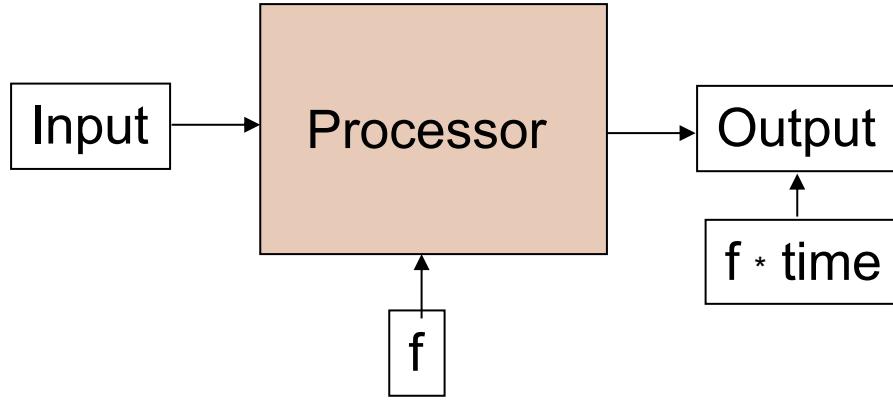
$$W = CV^2$$

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

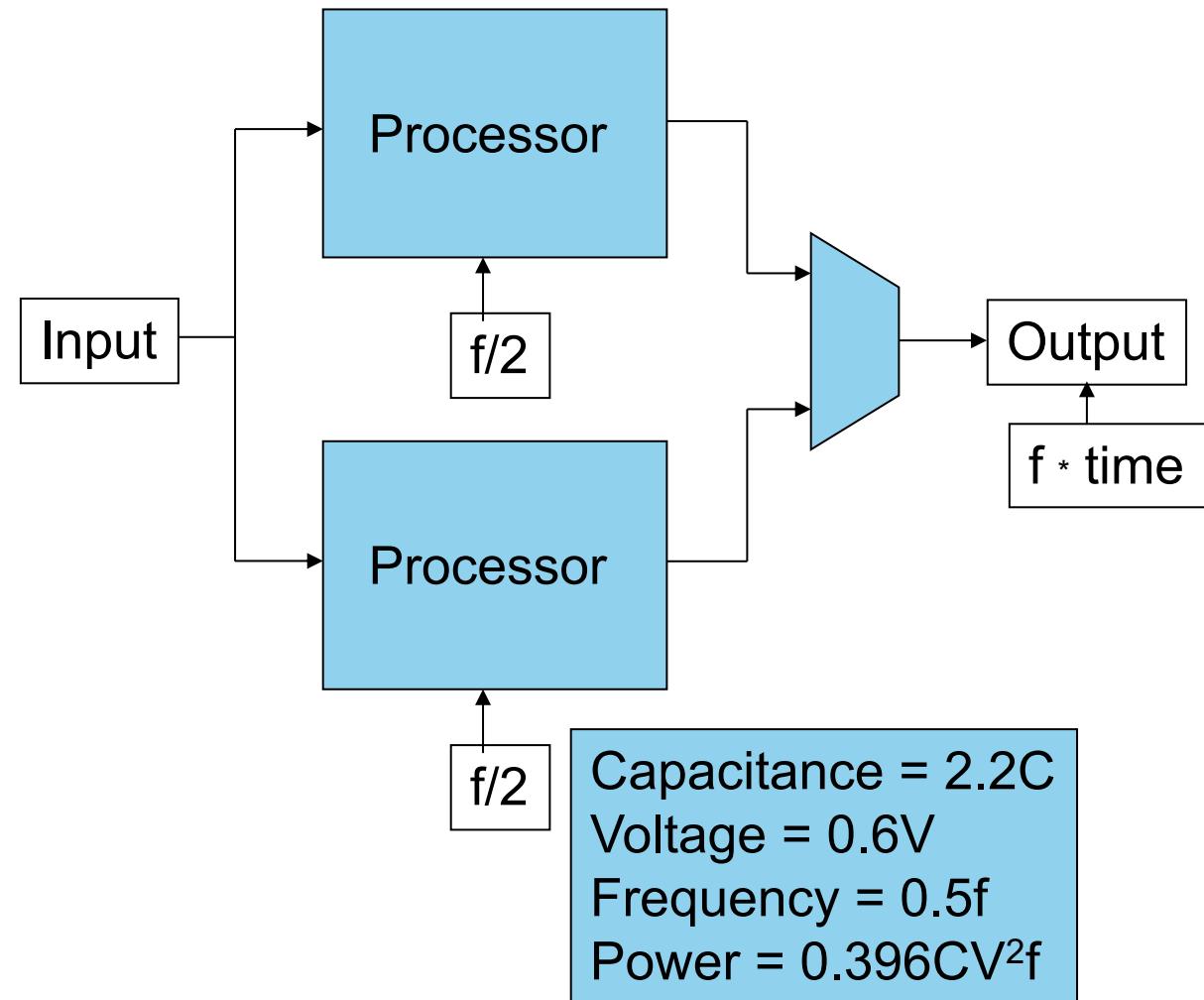
$$\text{Power} = W * F \rightarrow \text{Power} = CV^2f$$



... Reduce power by adding cores



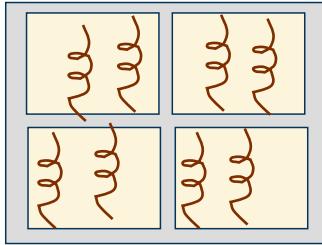
Capacitance = C
Voltage = V
Frequency = f
Power = CV^2f



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

For hardware ... parallelism is the path to performance

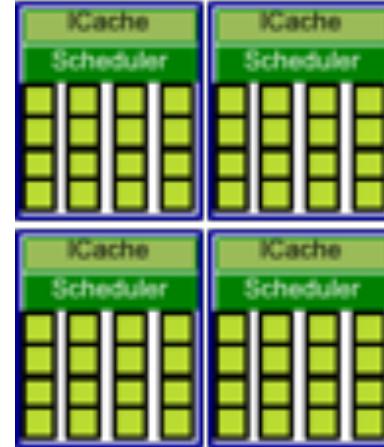
All hardware vendors are in the game ... parallelism is ubiquitous so if you care about getting the most from your hardware, you will need to create parallel software.



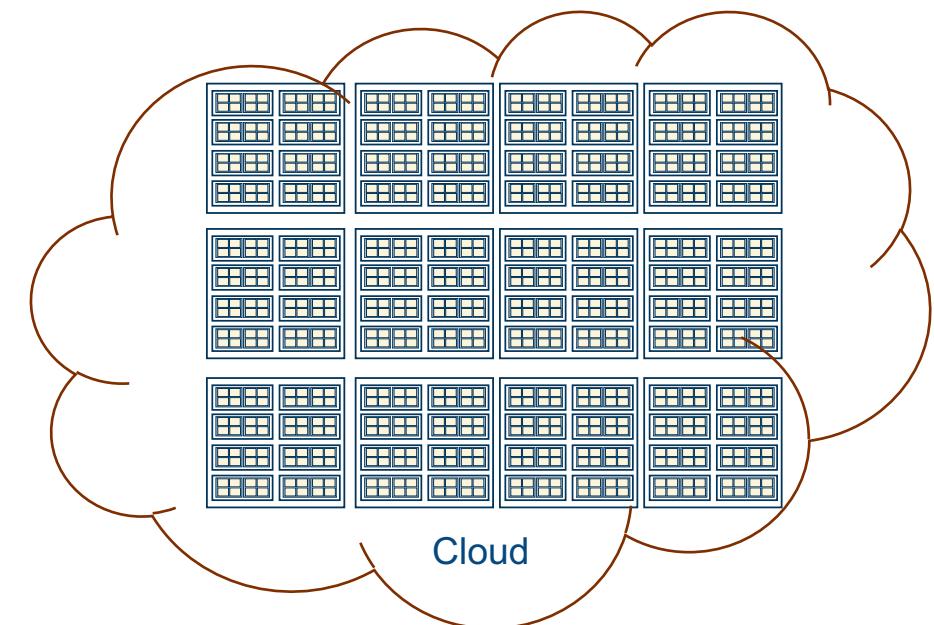
CPU



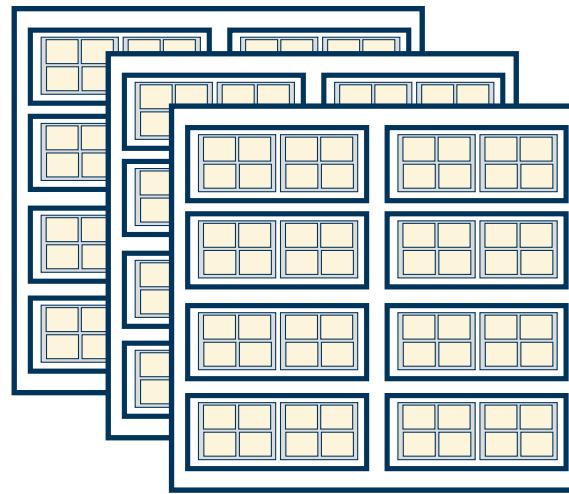
SIMD/Vector



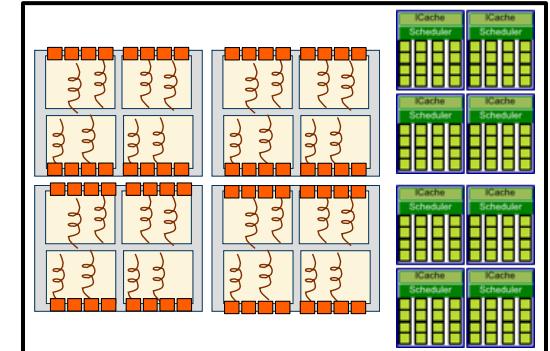
GPU



Cloud



Cluster



Heterogeneous node

The best way to master parallel computing ...

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

... and the simplest API for parallelism is?

Outline

- ➡ • Introduction to OpenMP
 - Creating Threads
 - Synchronization
 - Parallel Loops
 - Data Environment
 - Memory Model
 - Irregular Parallelism and Tasks
 - Worksharing Revisited
 - Synchronization Revisited: Options for Mutual exclusion
 - Threadprivate and the joys of “random” numbers
 - 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

cal

- 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)

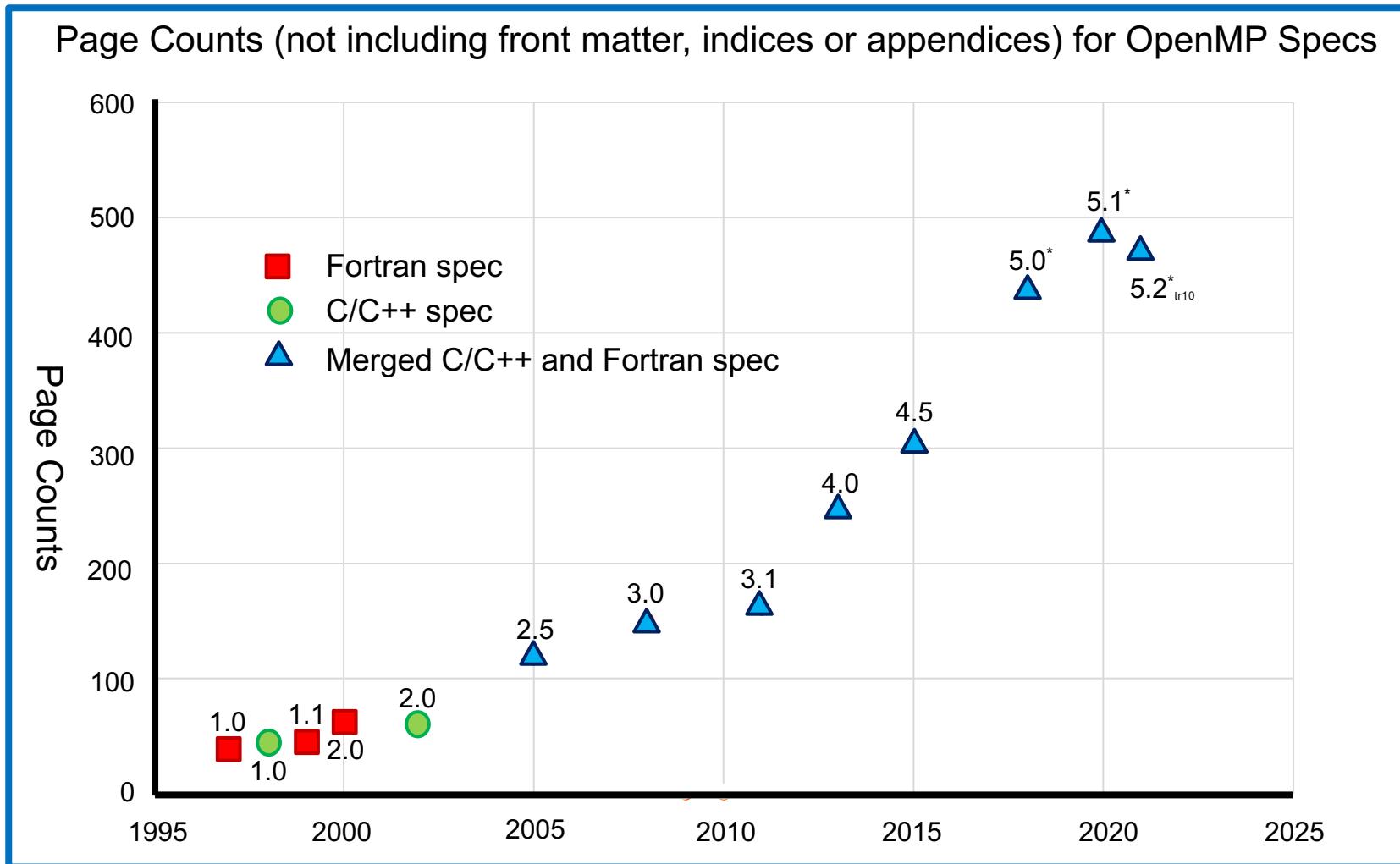
#pragma omp atomic seq_cst

Nthrds = OMP_GET_NUM_PROCS()

omp_set_lock(lck)

The Growth of Complexity in OpenMP

Our goal in 1997 ... A simple interface for application programmers

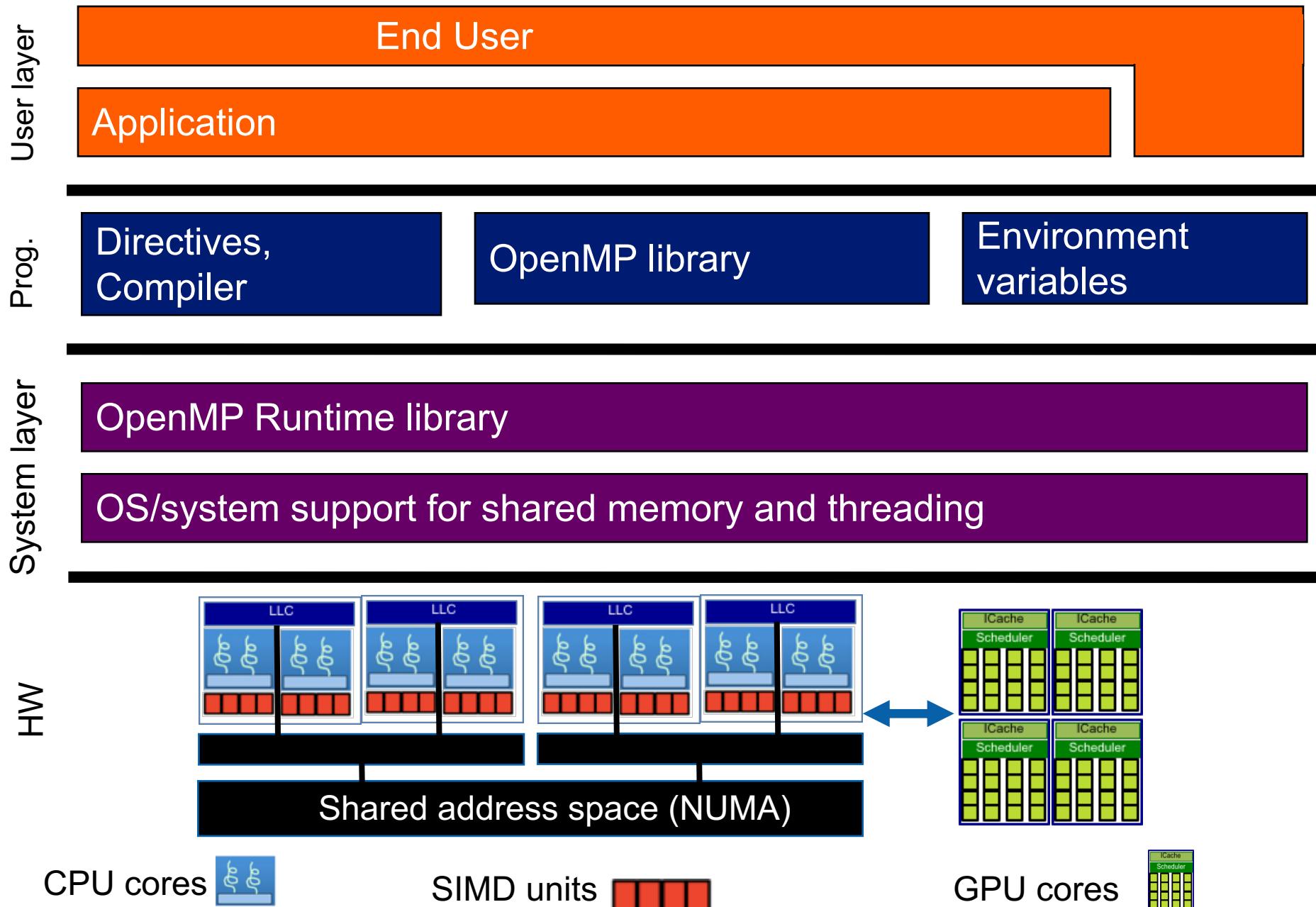


The full spec is overwhelming. We focus on the Common Core: the 21 items most people restrict themselves to

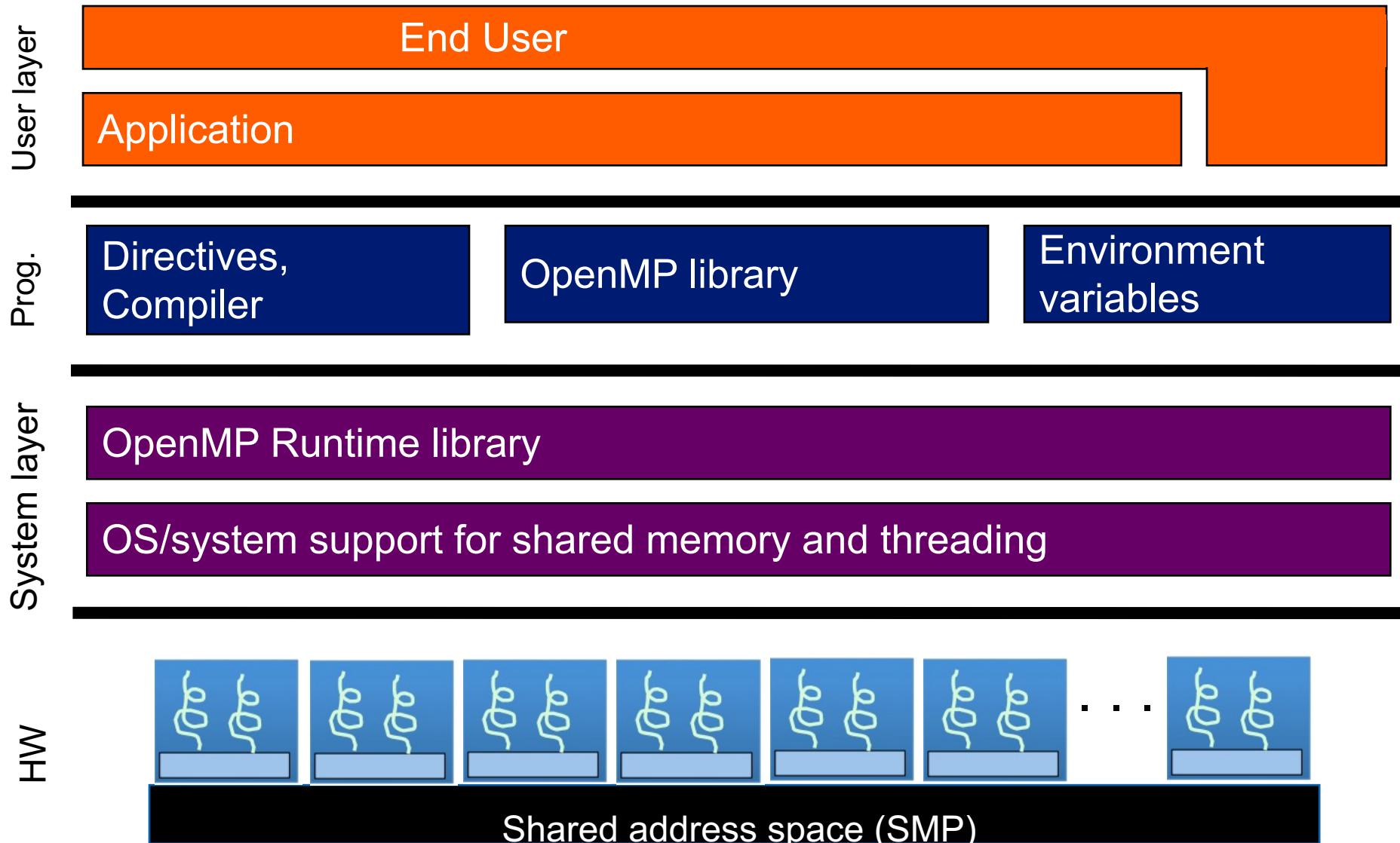
The OpenMP Common Core: Most OpenMP programs only use these 21 items

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	
<code>#pragma omp construct [clause [clause]...]</code>	<code>!\$OMP construct [clause [clause] ...]</code>
Example	
<code>#pragma omp parallel private(x)</code> { }	<code>!\$OMP PARALLEL PRIVATE(X)</code> <code>!\$OMP END PARALLEL</code>
Function prototypes and types:	
<code>#include <omp.h></code>	<code>use OMP_LIB</code>

- 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 -fopenmp	
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> ← OpenMP include file
#include <stdio.h>
int main()
{
#pragma omp parallel ← Parallel region with
    {                                default number of threads
        printf(" hello ");
        printf(" world \n");
    }                                End of the Parallel region
}
```

Sample Output:

hello hello world

world

hello hello world

world

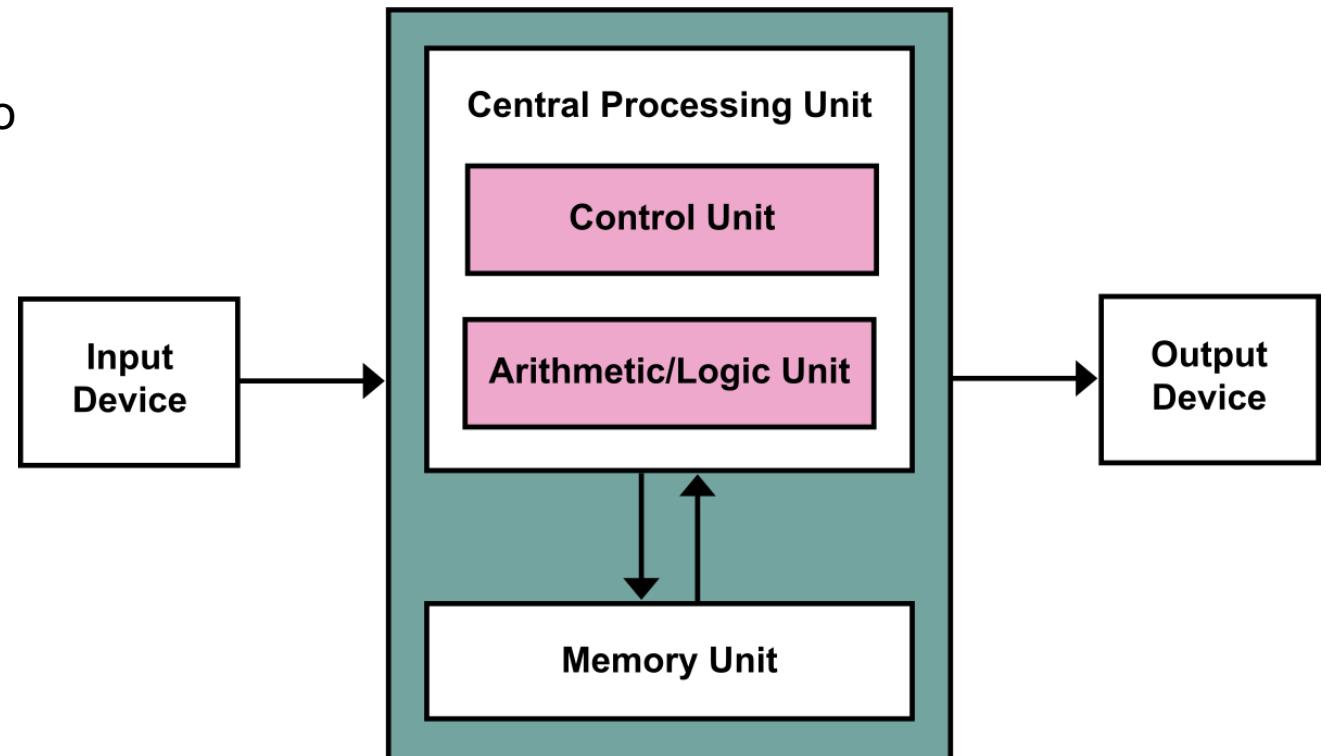
The statements are interleaved based on how the operating system schedules the threads

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:**

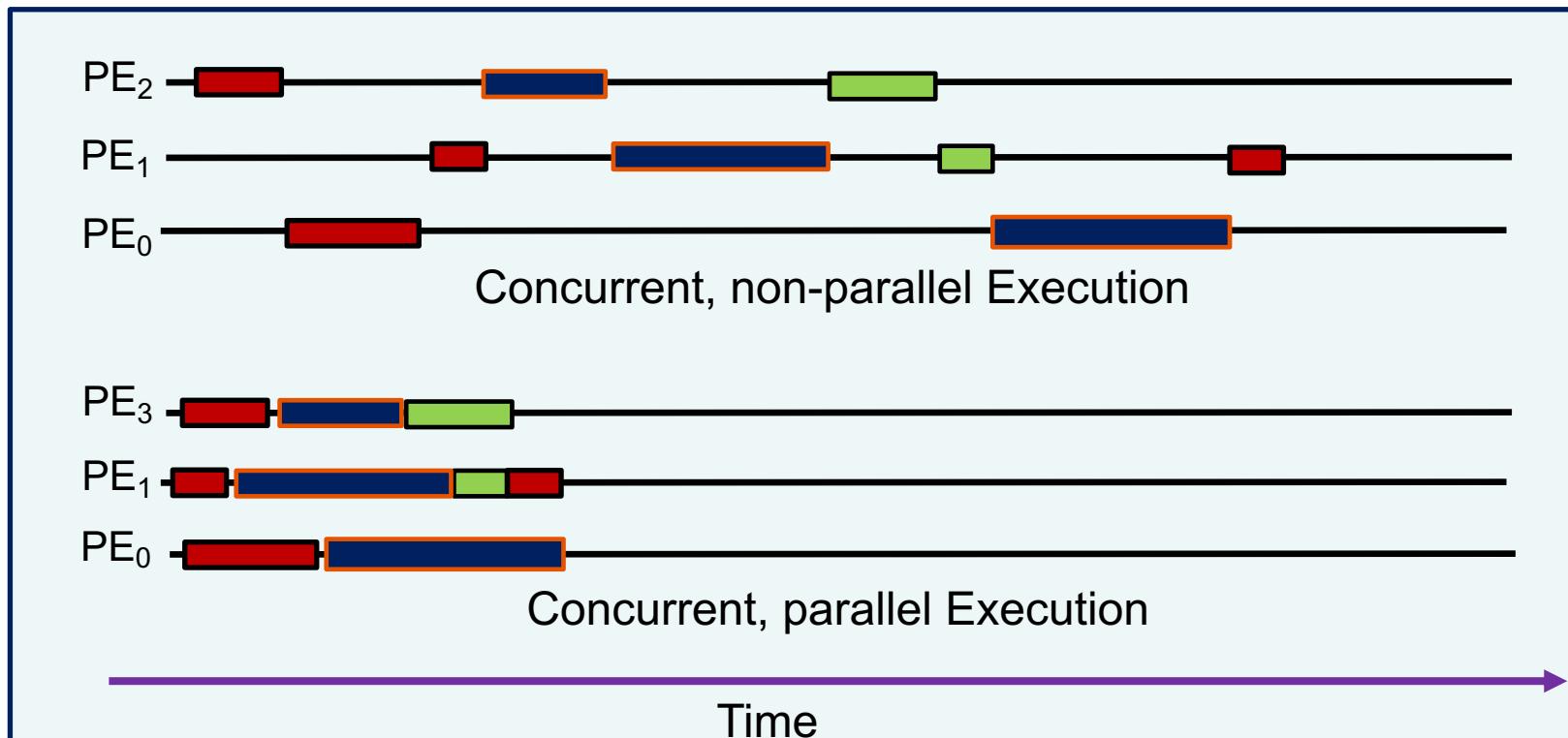
- A task that is available to be scheduled for execution. When the task is moving through its sequence of instructions, we say it is making **forward progress**

- **Fair scheduling:**

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

Concurrency vs. Parallelism

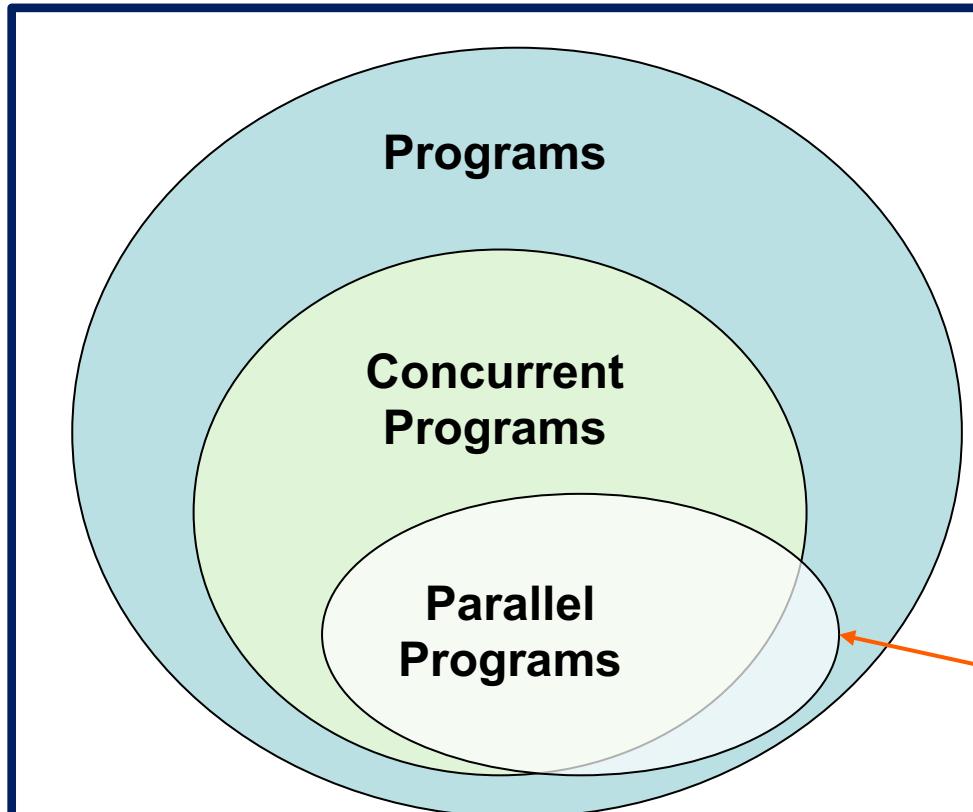
- Two important definitions:
 - Concurrency: A condition of a system in which multiple tasks are active and unordered. If **scheduled fairly**, they can be described as logically making **forward progress** at the same time.
 - Parallelism: A condition of a system in which multiple tasks are actually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



PE = Processing Element

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.

Outline

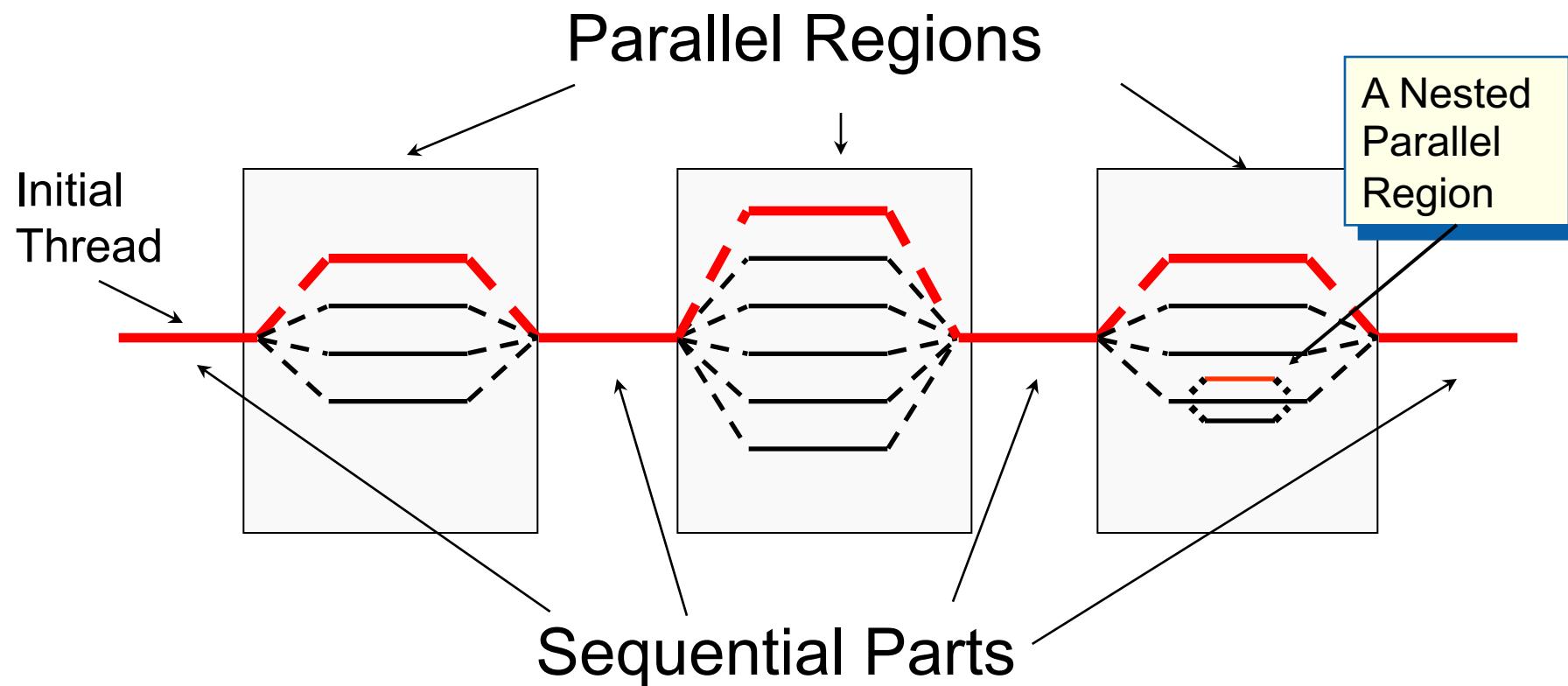
OpenMP®

- Introduction to OpenMP
- • Creating Threads
- Synchronization
- Parallel Loops
- Data Environment
- Memory Model
- Irregular Parallelism and Tasks
- Worksharing Revisited
- Synchronization Revisited: Options for Mutual exclusion
- Threadprivate and the joys of “random” numbers
- Recap

OpenMP Execution model:

Fork-Join Parallelism:

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



Thread Creation: Parallel Regions

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

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

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

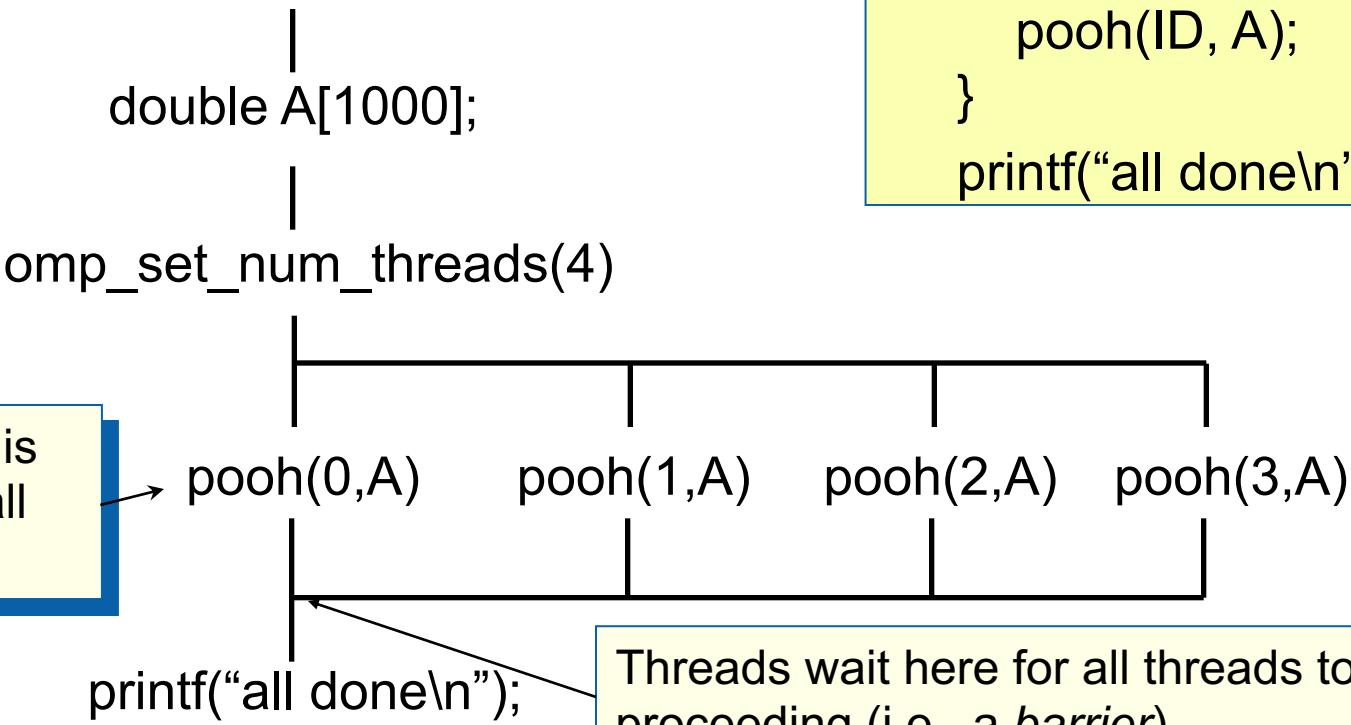
Runtime function to request a certain number of threads

Runtime function returning a thread ID

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

Thread Creation: Parallel Regions Example

- Each thread executes the same code redundantly.



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

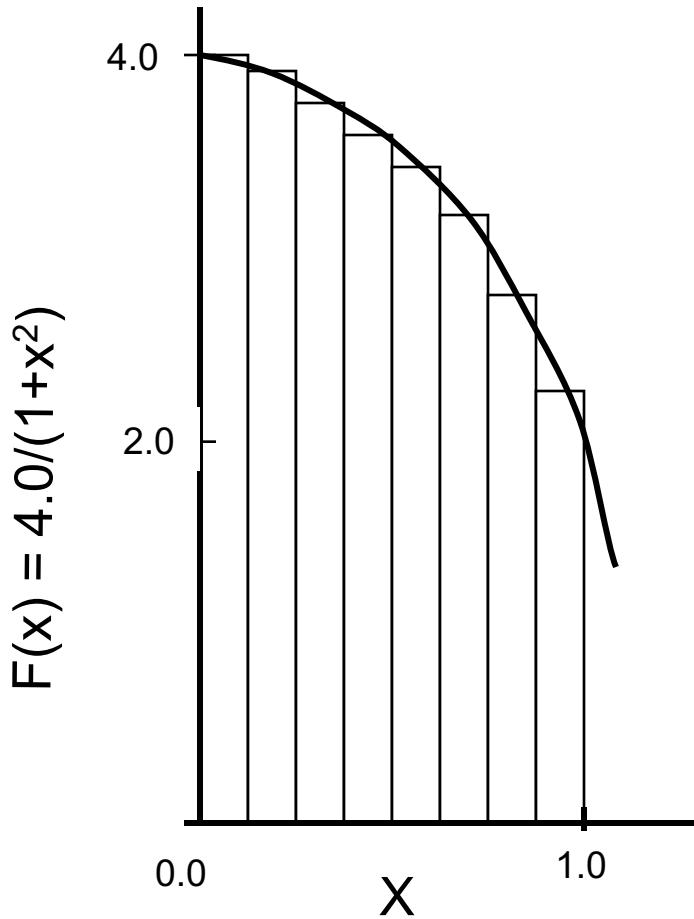
- Each thread calls `pooh(ID,A)` for $ID = 0$ to $nthrds-1$

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;
}
```

See ParProgForPhys/OMP_Exercises/pi.c

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

See [ParProgForPhys /OMP_Exercises/pi.c](#)

Exercise: the Parallel Pi Program

- Create a parallel version of the pi program using a parallel construct:
`#pragma omp parallel`
- Pay close attention to shared versus private variables.
- In addition to a parallel construct, you will need the runtime library routines

```
- int omp_get_num_threads(); ← Number of threads in the team  
- int omp_get_thread_num(); ← Thread ID or rank  
- double omp_get_wtime(); ← Time in seconds since a fixed point in the past  
- omp_set_num_threads();
```

Request a number of threads in the team

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

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.

Results*

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

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

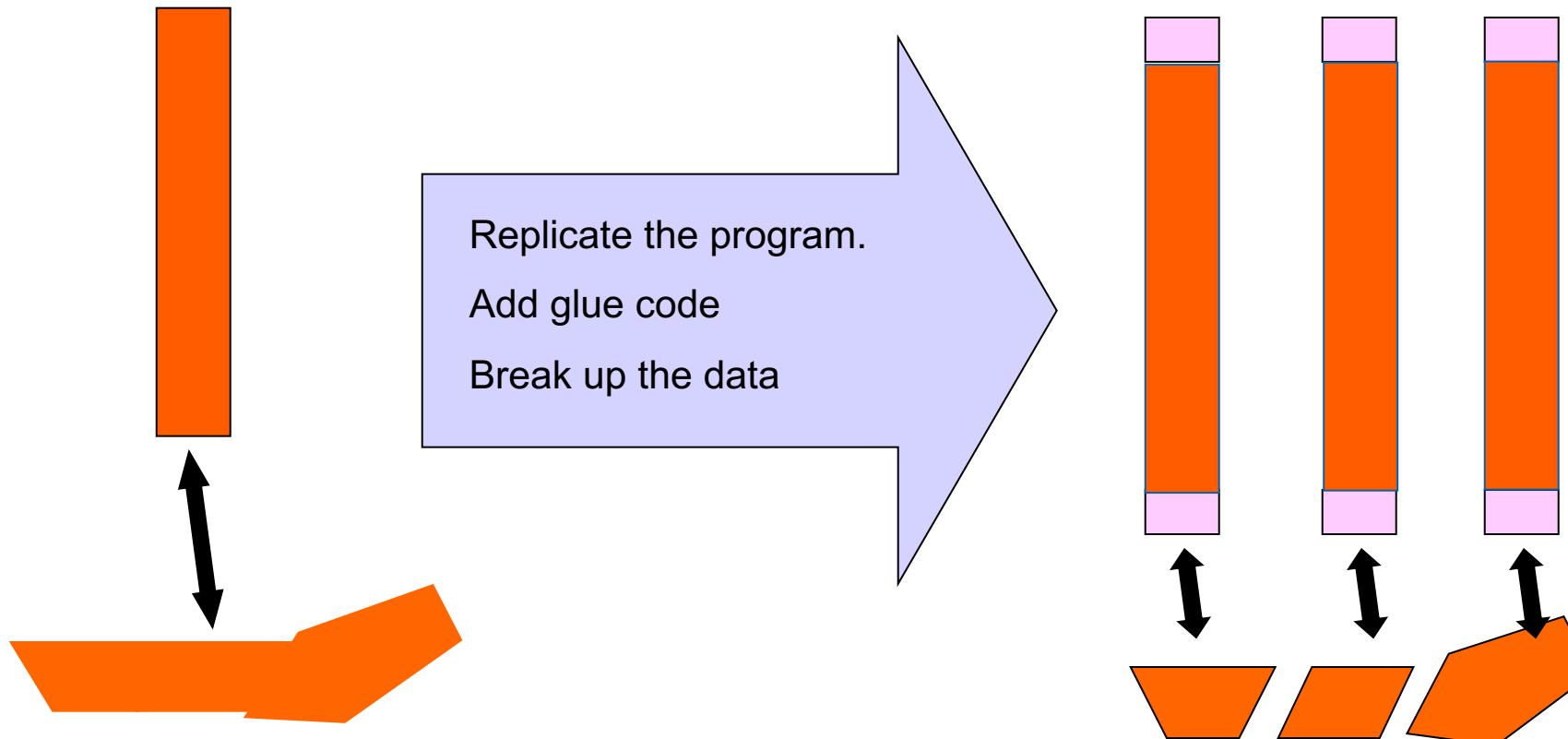
threads	1 st SPMD*
1	1.86
2	1.03
3	1.08
4	0.97

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

*SPMD: Single Program Multiple Data

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.

**A brief digression to talk about
performance issues 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

...

Consume Results

Compute T_N

Ideally Cut
runtime by $\sim 1/P$

(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.
- Perfect Linear Speedup: happens when no parallel overhead and algorithm is 100% parallel.
- Super-linear Speedup: typically due to cache effects ... i.e. as P grows, aggregate cache size grows so more of the problem fits in cache

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

$$S(P) = P$$

$$S(P) > P$$

Amdahl's Law

- What is the maximum speedup you can expect from a parallel program?
- Approximate the runtime as a part that can be sped up with additional processors and a part that is fundamentally serial.

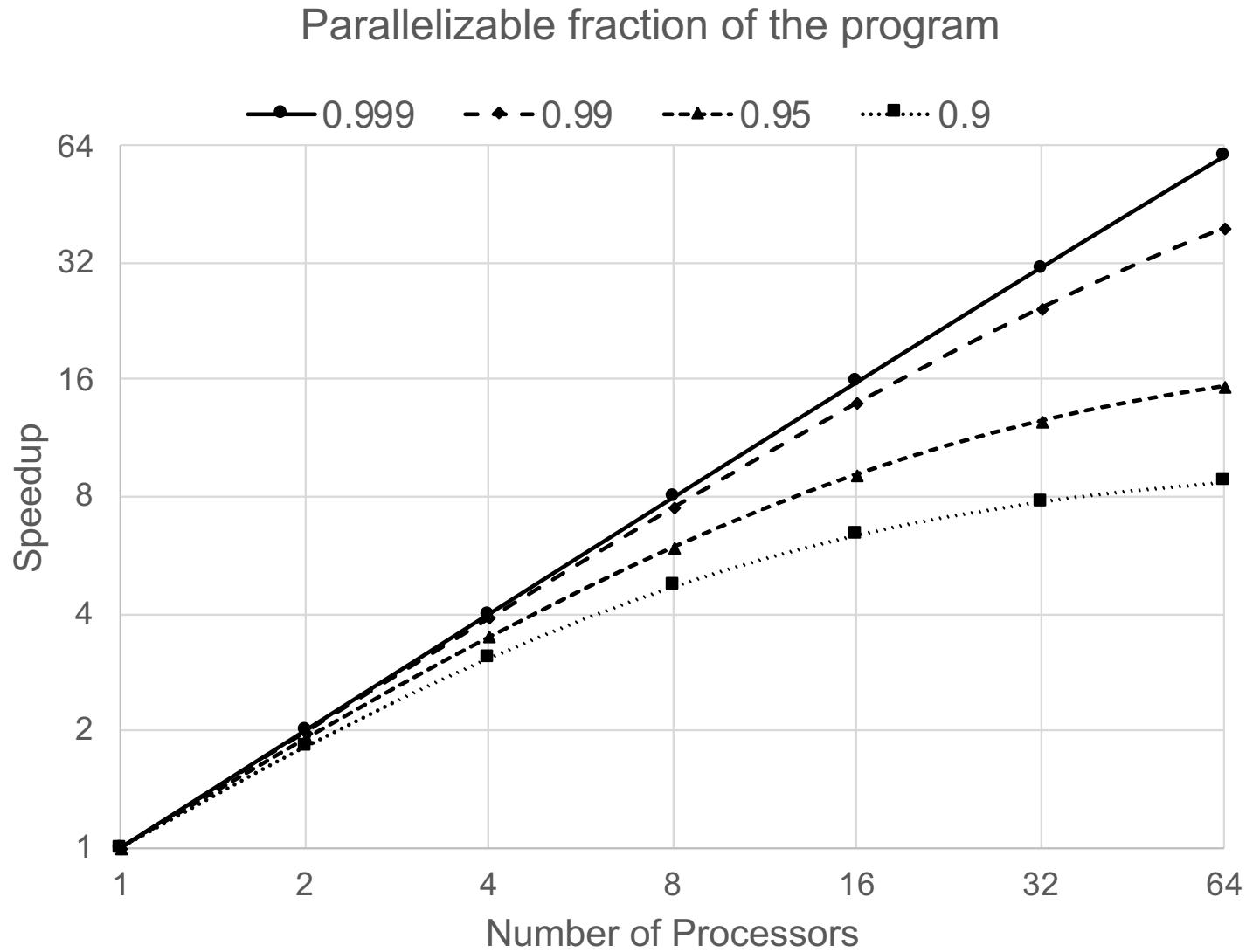
$$Time_{par}(P) = (serial_fraction + \frac{parallel_fraction}{P}) * Time_{seq}$$

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

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

- If you had an unlimited number of processors: $P \rightarrow \infty$
- The maximum possible speedup is: $S = \frac{1}{\alpha} \leftarrow \boxed{\text{Amdahl's Law}}$

Amdahl's Law



So now you should understand my silly introduction slide.

Introduction

I'm just a simple kayak instructor

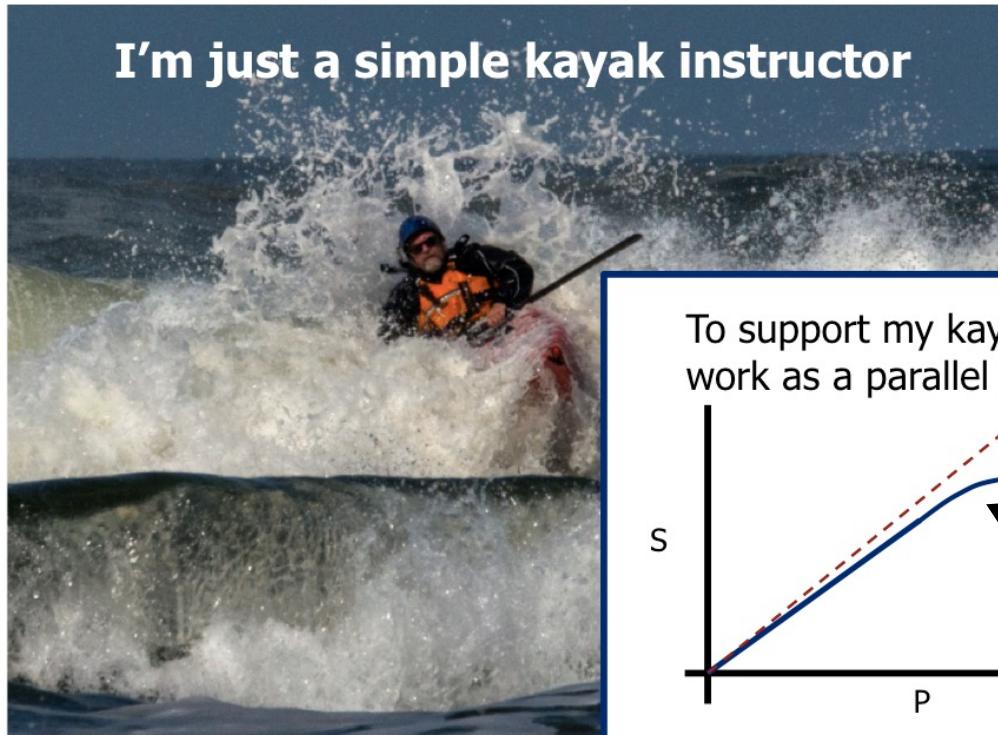
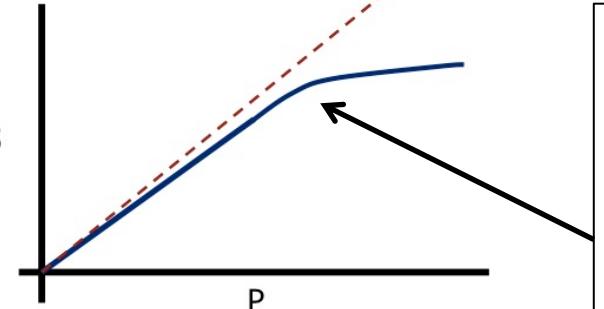


Photo © by Greg Clopton, 2014

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

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.

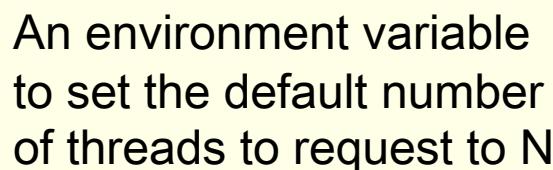
Internal control variables and how to control the number of threads in a team

- We've used the following construct to control the number of threads. (e.g. to request 12 threads):
 - `omp_set_num_threads(12)`
- What does `omp_set_num_threads()` actually do?
 - It resets an “internal control variable” the system queries to select the default number of threads to request on subsequent parallel constructs.
- Is there an easier way to change this internal control variable ... perhaps one that doesn't require re-compilation? Yes.
 - When an OpenMP program starts up, it queries an environment variable `OMP_NUM_THREADS` and sets the appropriate internal control variable to the value of **OMP_NUM_THREADS**
 - For example, to set the initial, default number of threads to request in OpenMP from my apple laptop
 - > **export OMP_NUM_THREADS=12**

Exercise

- Go back to your parallel pi program and explore how well it scales with the number of threads.
- Can you explain your performance with Amdahl's law? If not what else might be going on?

- `int omp_get_num_threads();`
- `int omp_get_thread_num();`
- `double omp_get_wtime();`
- `omp_set_num_threads();`
- `export OMP_NUM_THREADS = N`



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

Results*

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

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

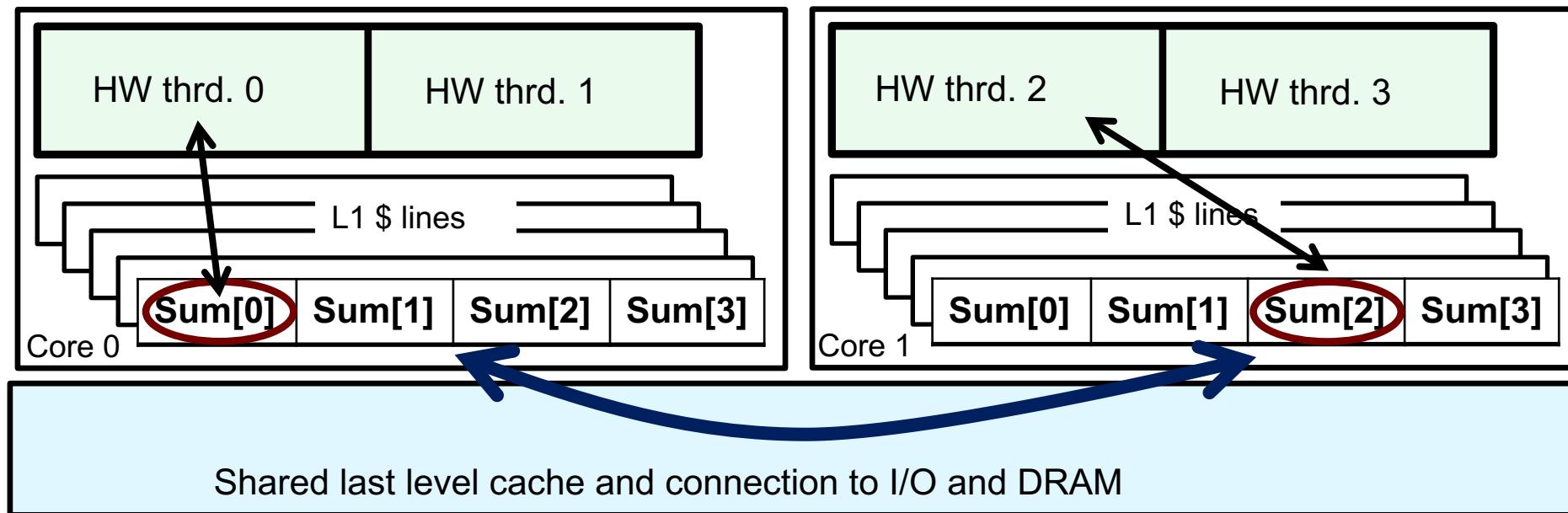
threads	1 st SPMD*
1	1.86
2	1.03
3	1.08
4	0.97

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

*SPMD: Single Program Multiple Data

Why Such Poor Scaling? False Sharing

- If independent data elements happen to sit on the same cache line, each update will cause the cache lines to “slosh back and forth” between threads ... This is called “**false sharing**”.



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

Example: Eliminate false sharing by padding the sum array

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

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

Results*: PI Program, Padded Accumulator

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

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

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

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

- Introduction to OpenMP
- Creating Threads
- • Synchronization
- Parallel Loops
- Data Environment
- Memory Model
- Irregular Parallelism and Tasks
- Worksharing Revisited
- Synchronization Revisited: Options for Mutual exclusion
- Threadprivate and the joys of “random” numbers
- 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 must reach before any threads are allowed to proceed.
- It is a “stand alone” pragma meaning it is not associated with user code ... it is an executable statement.

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

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



Exercise

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

```
int omp_get_num_threads();
int omp_get_thread_num();
double omp_get_wtime();
omp_set_num_threads();
#pragma parallel
#pragma 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	1 st 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; ← Create a scalar local to each
    id = omp_get_thread_num();                                thread to accumulate partial sums.
    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); ← No array, so no false sharing.
    }
    #pragma omp critical
    pi += sum * step; ← 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	1st SPMD padded	SPMD critical
1	1.86	1.86	1.87
2	1.03	1.01	1.00
3	1.08	0.69	0.68
4	0.97	0.53	0.53

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

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

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

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

Outline

OpenMP®

- Introduction to OpenMP
- Creating Threads
- Synchronization
- • Parallel Loops
- Data Environment
- Memory Model
- Irregular Parallelism and Tasks
- Worksharing Revisited
- Synchronization Revisited: Options for Mutual exclusion
- Threadprivate and the joys of “random” numbers
- 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);
```

```
}
```

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 construct name:

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

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];}
```

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	Least work at runtime : scheduling done at compile-time
DYNAMIC	Unpredictable, highly variable work per iteration	Most work at runtime : complex scheduling logic used at run-time

Combined Parallel/Worksharing Construct

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

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

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

These are equivalent

Working with loops

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

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

Note: loop index
“i” is private by
default

Remove loop
carried
dependence

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

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
-	0
min	Largest pos. number
max	Most neg. number

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

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

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

Exercise: PI with loops

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

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

Example: PI with a loop and a reduction

```
#include <omp.h>
static long num_steps = 100000;      double step;
void main ()
{   int i;           double x, pi, sum = 0.0;
    step = 1.0/(double) num_steps;
    #pragma omp parallel
    {
        double x;           ← Create a scalar local to each thread to hold
        #pragma omp for reduction(+:sum)
        for (i=0;i< num_steps; i++){
            x = (i+0.5)*step;
            sum = sum + 4.0/(1.0+x*x), ← Break up loop iterations
        }                                and assign them to
    }                                    threads ... setting up a
    pi = step * sum;                   reduction into sum.
}                                     Note ... the loop index is
                                   local to a thread by default.
```

Example: PI with a loop and a reduction

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

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

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

Results*: PI with a loop and a reduction

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

Example: Pi with a

```
#include <omp.h>
static long num_steps = 100000000;
void main ()
{
    int i;      double x, pi, sum;
    step = 1.0/(double) num_steps;
    #pragma omp parallel
    {
        double x;
        #pragma omp for reduction(+:sum)
        for (i=0;i< num_steps; i++){
            x = (i+0.5)*step;
            sum = sum + 4.0/(1.0+x*x);
        }
    }
    pi = step * sum;
}
```

threads	1 st SPMD	1 st 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.

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

OpenMP®

- Introduction to OpenMP
- Creating Threads
- Synchronization
- Parallel Loops
- • Data Environment
- Memory Model
- Irregular Parallelism and Tasks
- Worksharing Revisited
- Synchronization Revisited: Options for Mutual exclusion
- Threadprivate and the joys of “random” numbers
- 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.

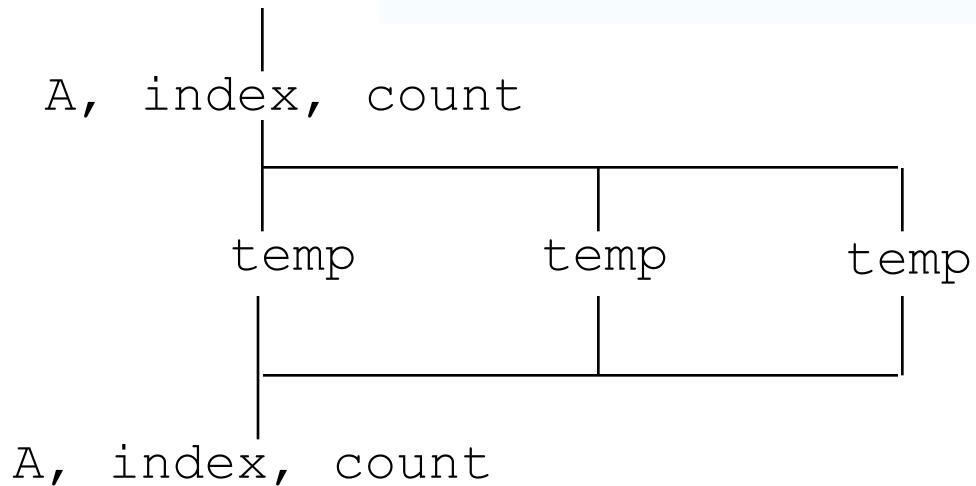
Data Sharing: Examples

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

A, index and count are shared by all threads.

temp is local to each thread

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



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

```
void wrong() {  
    int tmp = 0;  
#pragma omp parallel for private(tmp)  
    for (int j = 0; j < 1000; ++j)  
        tmp += j;  
    printf("%d\n", tmp);  
}
```

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

tmp was not
initialized

tmp 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

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

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.

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

```
#pragma omp parallel private (list)
#pragma omp parallel shared (list)
#pragma omp parallel firstprivate (list)
#pragma omp parallel default(none)
#pragma omp for reduction(op:list)
```

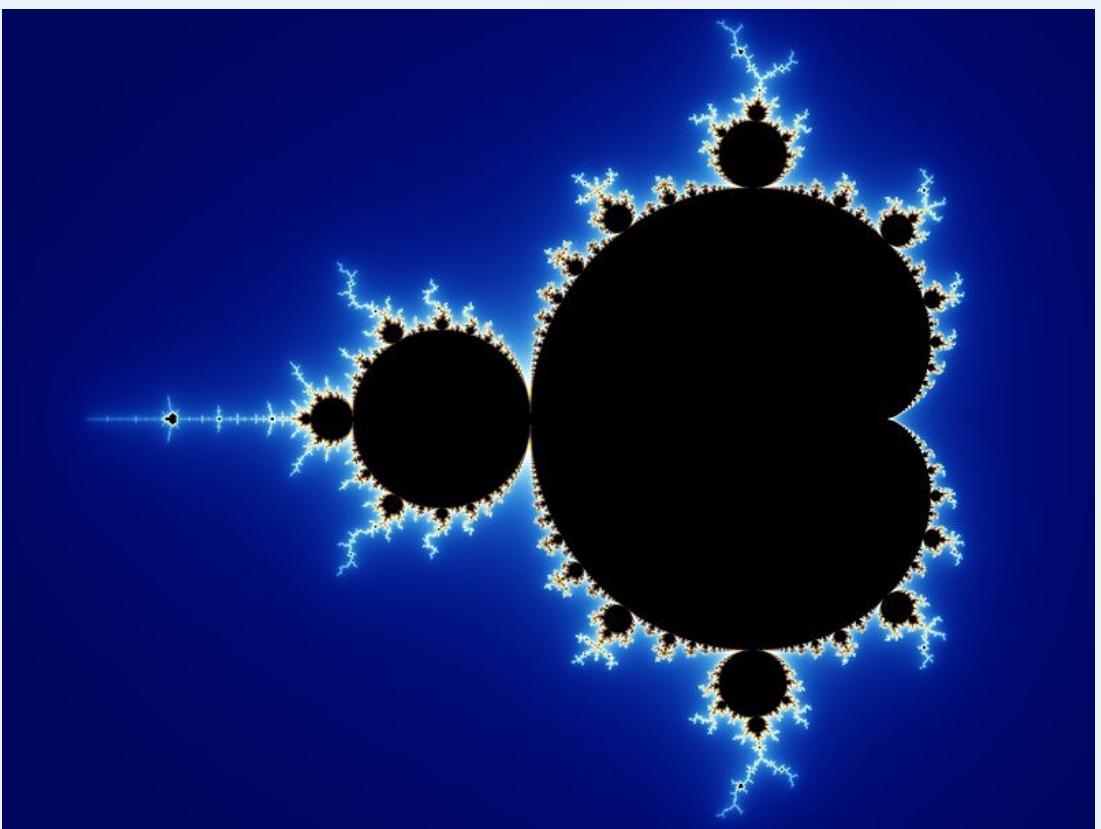


Image Source: Created by Wolfgang Beyer with the program Ultra Fractal 3. - Own work, CC BY-SA 3.0, <https://commons.wikimedia.org/w/index.php?curid=321973>

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 $z_0 = 0$.

The Mandelbrot Set Area Program (original code)

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

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

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

The Mandelbrot Set Area Program

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

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

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

- `eps` was not initialized ... OK to leave it shared
- Make `j`, `C_real`, and `C_imag` private
- Protect updates of `numoutside`

Data Sharing: Private and the original variable

- The original variable's value is unspecified if it is referenced outside of the construct
 - Implementations may reference the original variable or a copy a dangerous programming practice!
 - For example, consider what would happen if the compiler inlined work()?

```
int tmp;  
void danger() {  
    tmp = 0;  
#pragma omp parallel private(tmp)  
    work();  
    printf("%d\n", tmp);  
}
```

tmp has unspecified value

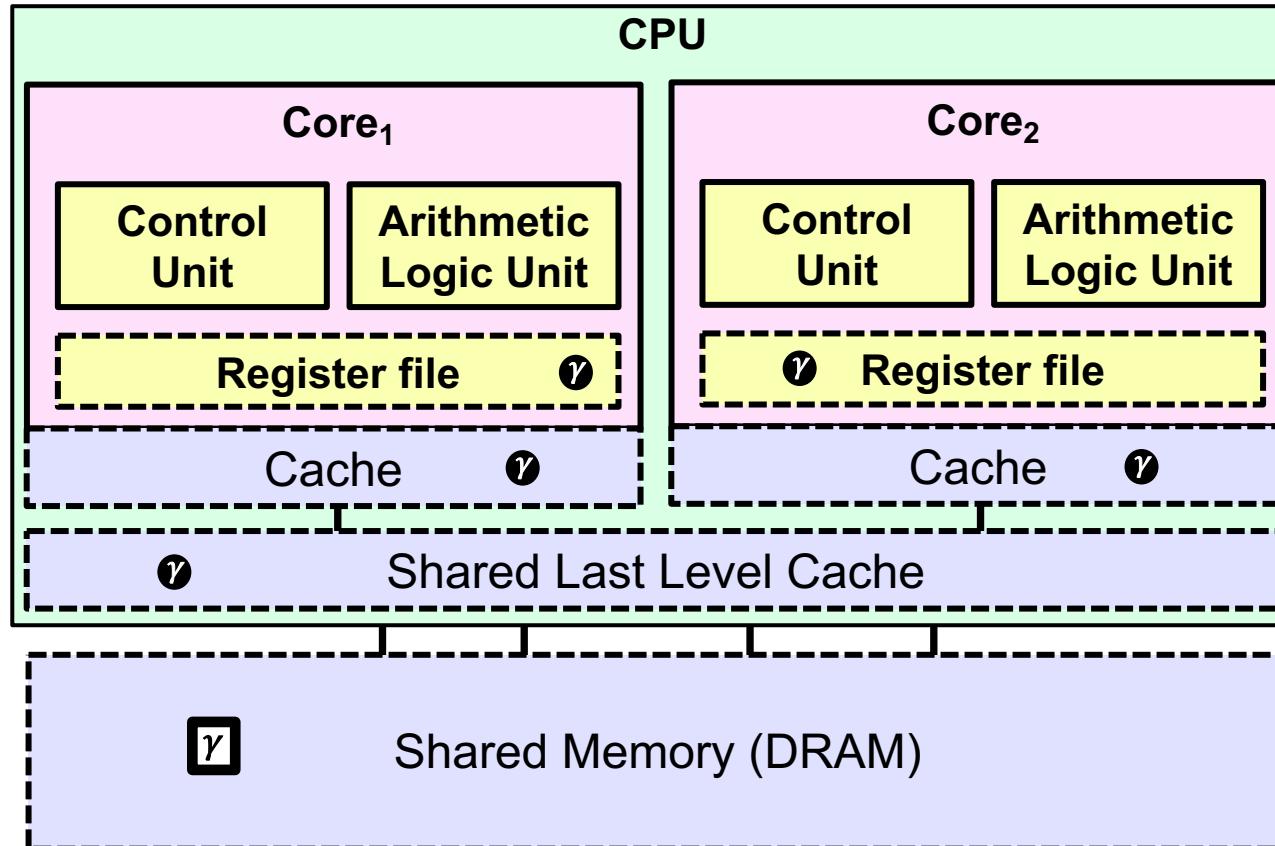
```
extern int tmp;  
void work() {  
    tmp = 5;  
}
```

unspecified which
copy of tmp

- Introduction to OpenMP
- Creating Threads
- Synchronization
- Parallel Loops
- Data Environment
- • Memory Model
- Irregular Parallelism and Tasks
- Worksharing Revisited
- Synchronization Revisited: Options for Mutual exclusion
- Threadprivate and the joys of “random” numbers
- Recap

Memory Models ...

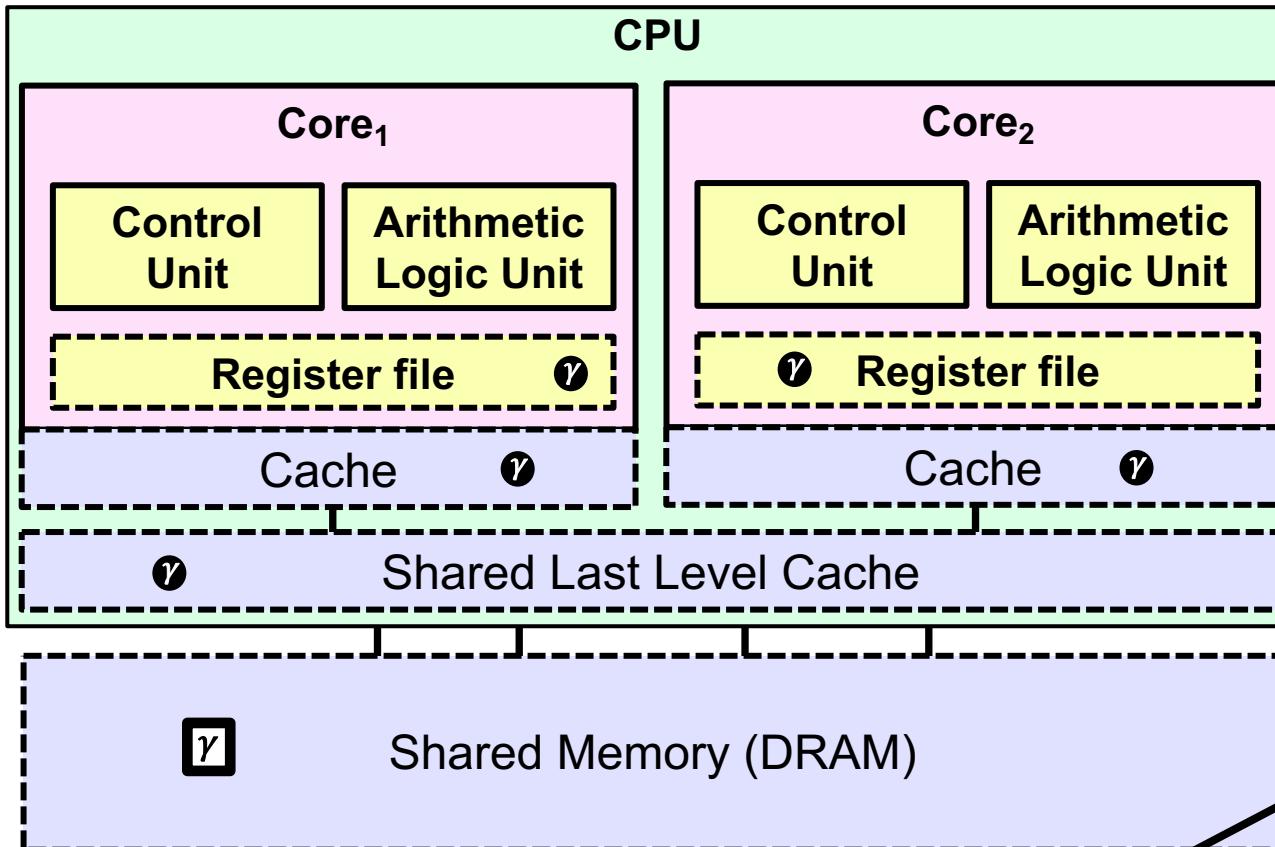
- Programming models for Multithreading support shared memory.
- All threads share an address space ... but consider the variable γ



- Multiple copies of a variable (such as γ) may be present at various levels of cache, or in registers and they may ALL have different values.
- So which value of γ is the one a thread should see at any point in a computation?

Memory Models ...

- Programming models for Multithreading support shared memory.
- All threads share an address space ... but consider the variable γ



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

- Multiple copies of a variable (such as γ) may be present at various levels of cache, or in registers and they may ALL have different values.
- So which value of γ is the one a thread should see at any point in a computation?

OpenMP and Relaxed Consistency

- Most (if not all) multithreading programming models (including OpenMP) supports a **relaxed-consistency** memory model
 - Threads can maintain a **temporary view** of shared memory that is not consistent with that of other threads
 - These temporary views are made consistent only at certain points in the program
 - The operation that enforces consistency is called the **flush operation***

*Note: in OpenMP 5.0 the name for the flush described here was changed to a "strong flush". This was done so we could distinguish the traditional OpenMP flush (the strong flush) from the new synchronizing flushes (acquire flush and release flush).

Flush Operation

- 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
- A flush operation is analogous to a **fence** in other shared memory APIs

* 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**”

Flush Example

- Flush forces data to be updated in memory so other threads see the most recent value*

```
double A;  
A = compute();  
#pragma omp flush(A)  
    // flush to memory to make sure other  
    // threads can pick up the right value
```

Note: OpenMP's flush is analogous to a fence in other shared memory APIs

* If you pass a list of variables to the flush directive, then that list is “**the flush set**”

What is the BIG DEAL with Flush?

- Compilers routinely reorder instructions implementing a program
 - Can better exploit the functional units, keep the machine busy, hide memory latencies, etc.
- Compilers generally cannot move instructions:
 - Past a barrier
 - Past a flush on all variables
- But it can move them past a flush with a list of variables so long as those variables are not accessed
- Keeping track of consistency when flushes are used can be confusing ... especially if “flush(list)” is used.

Warning: the flush operation (a strong flush) does not actually synchronize different threads. It just ensures that a thread's variables are made consistent with main memory

Flush and Synchronization

- A flush operation is implied by OpenMP synchronizations, e.g.,
 - at entry/exit of parallel regions
 - at implicit and explicit barriers
 - at entry/exit of critical regions
 -
- (but not on entry to worksharing regions)

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.

- Introduction to OpenMP
- Creating Threads
- Synchronization
- Parallel Loops
- Data Environment
- Memory Model
- • Irregular Parallelism and Tasks
- Worksharing Revisited
- Synchronization Revisited: Options for Mutual exclusion
- Threadprivate and the joys of “random” numbers
- 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 (without tasks)

- See the file solutions/linked_notasks.c

```
while (p != NULL) {
    p = p->next;
    count++;
}

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

Count number of items in the linked list

Copy pointer to each node into an array

Process nodes in parallel with a for loop

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

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

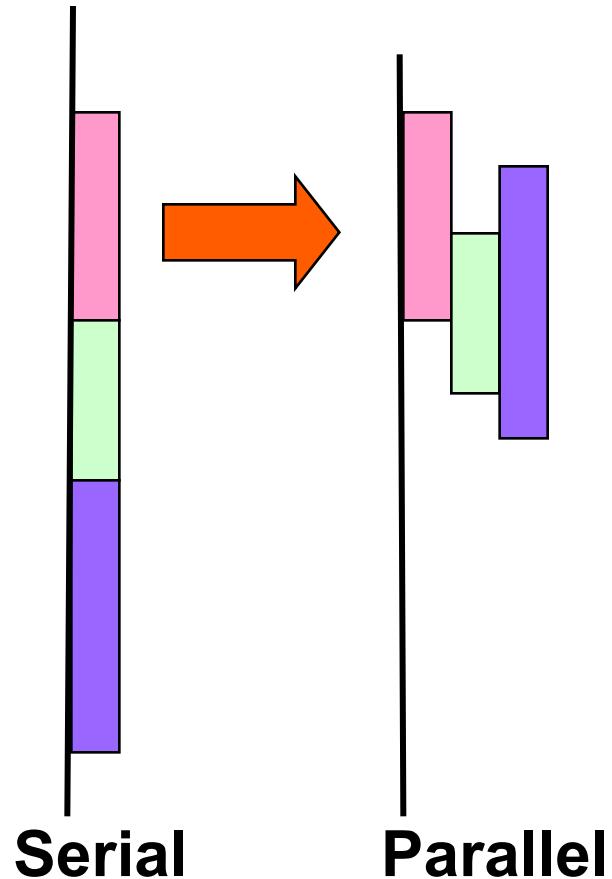
With so much code to add and three passes through the data, this is really ugly.

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

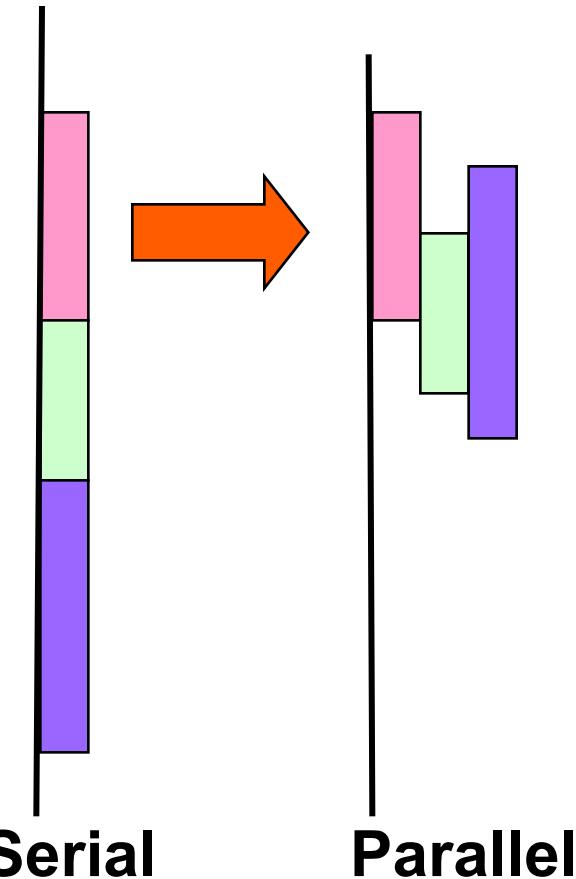
What are Tasks?

- 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 ← Create some threads
{
```

```
    #pragma omp single ← One Thread
    {                                packages tasks
```

```
        #pragma omp task
            fred();
```

```
        #pragma omp task
            daisy();
```

```
        #pragma omp task
            billy();
```

```
}
```

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
```

Racey Cars: Solution

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

Data Scoping with Tasks

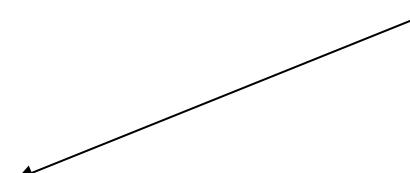
- Variables can be shared, private or firstprivate with respect to task
- These concepts are a little bit different compared with threads:
 - If a variable is **shared** on a task construct, the references to it inside the construct are to the storage with that name at the point where the task was encountered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

Data Scoping Defaults

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

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

A is shared
B is firstprivate
C is private



Exercise: Traversing linked lists

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

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

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

Parallel Linked List Traversal

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

Only one thread packages tasks

makes a copy of p
when the task is
packaged

When/Where are Tasks Complete?

- At thread barriers (explicit or implicit)
 - all tasks generated inside a region must complete at the next barrier encountered by the threads in that region. Common examples:
 - **Tasks generated inside a single construct:** all tasks complete before exiting the barrier on the single.
 - **Tasks generated inside a parallel region:** all tasks complete before exiting the barrier at the end of the parallel region.
- At taskwait directive
 - i.e. Wait until all tasks defined in the current task have completed.
`#pragma omp taskwait`
 - Note: applies only to tasks generated in the current task, not to “descendants” .

Example

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

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

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

Example

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

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

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

Example: Fibonacci numbers

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

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

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

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

Parallel Fibonacci

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

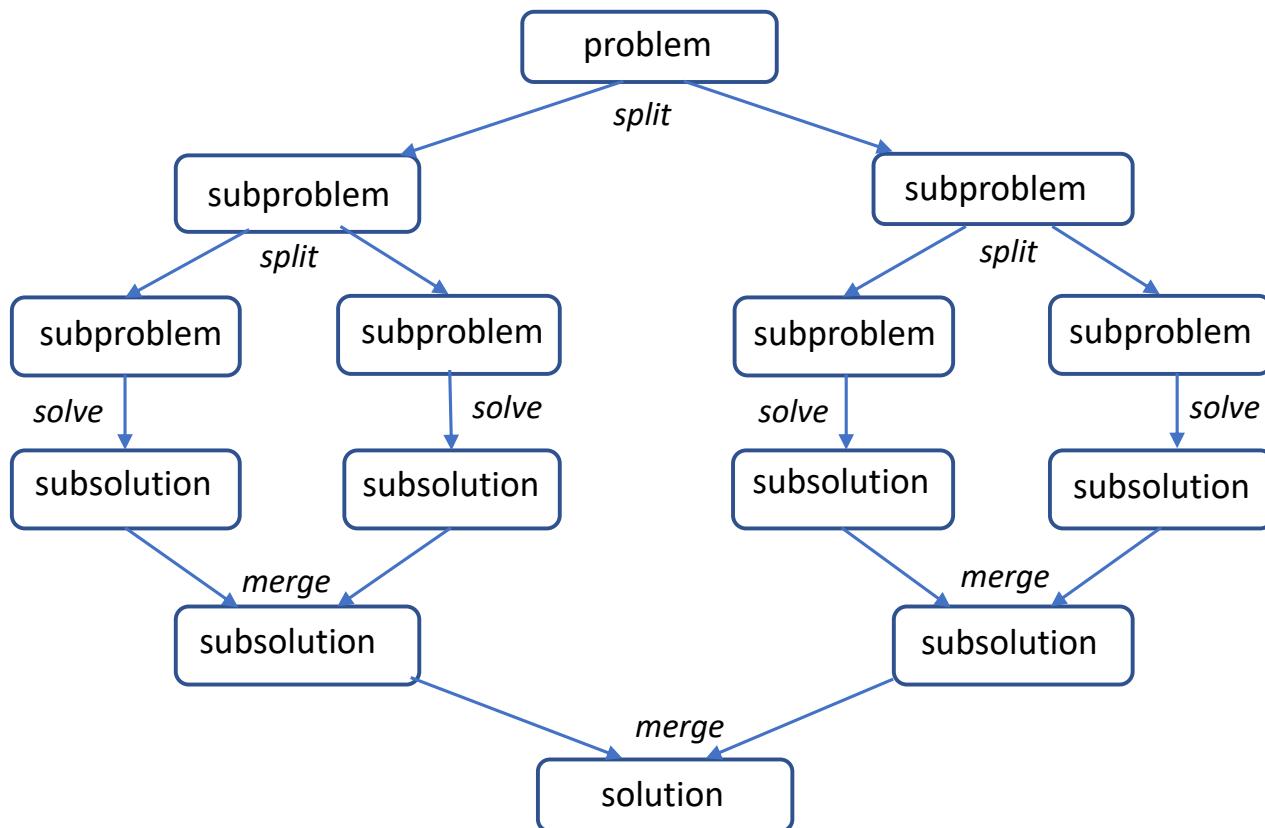
#pragma omp task shared(x)
    x = fib(n-1);
#pragma omp task shared(y)
    y = fib (n-2);
#pragma omp taskwait
    return (x+y);
}

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

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

Divide and Conquer

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



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

Exercise: PI with tasks

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

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

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

Program: OpenMP tasks

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

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

Results*: Pi with tasks

threads	1 st 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

- Introduction to OpenMP
- Creating Threads
- Synchronization
- Parallel Loops
- Data Environment
- Memory Model
- Irregular Parallelism and Tasks
- ➡ • Worksharing Revisited
- Synchronization Revisited: Options for Mutual exclusion
- Threadprivate and the joys of “random” numbers
- Recap

The Loop Worksharing Constructs

- 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);
    }
}
```

The variable I is made “private” to each thread by default. You could do this explicitly with a “private(I)” clause

Loop construct name:

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

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.
 - **schedule(guided[,chunk])**
 - Threads dynamically grab blocks of iterations. The size of the block starts large and shrinks down to size “chunk” as the calculation proceeds.
 - **schedule(runtime)**
 - Schedule and chunk size taken from the OMP_SCHEDULE environment variable (or the runtime library) ... vary schedule without a recompile!
 - **Schedule(auto)**
 - Schedule is left up to the runtime to choose (does not have to be any of the above).

OpenMP 4.5 added modifiers monotonic, nonmonotonic and simd.

Loop Worksharing Constructs: The schedule clause

Schedule Clause	When To Use	
STATIC	Pre-determined and predictable by the programmer	Least work at runtime : scheduling done at compile-time
DYNAMIC	Unpredictable, highly variable work per iteration	Most work at runtime : complex scheduling logic used at run-time
GUIDED	Special case of dynamic to reduce scheduling overhead	
AUTO	When the runtime can “learn” from previous executions of the same loop	

Nested Loops

- For perfectly nested rectangular loops we can parallelize multiple loops in the nest with the collapse clause:

```
#pragma omp parallel for collapse(2)
```

```
for (int i=0; i<N; i++) {  
    for (int j=0; j<M; j++) {  
        . . . .  
    }  
}
```

Number of loops
to be
parallelized,
counting from
the outside

- Will form a single loop of length NxM and then parallelize that.
- Useful if N is O(no. of threads) so parallelizing the outer loop makes balancing the load difficult.

Sections Worksharing Construct

- The *Sections* worksharing construct gives a different structured block to each thread.

```
#pragma omp parallel
{
    #pragma omp sections
    {
        #pragma omp section
            x_calculation();
        #pragma omp section
            y_calculation();
        #pragma omp section
            z_calculation();
    }
}
```

By default, there is a barrier at the end of the “omp sections”. Use the “nowait” clause to turn off the barrier.

Array Sections with Reduce

```
#include <stdio.h>
#define N 100
void init(int n, float (*b)[N]);
int main(){
    int i,j; float a[N], b[N][N]; init(N,b);
    for(i=0; i<N; i++) a[i]=0.0e0;
```

Works the same as any other reduce ... a private array is formed for each thread, element wise combination across threads and then with original array at the end

```
#pragma omp parallel for reduction(+:a[0:N]) private(j)
for(i=0; i<N; i++){
    for(j=0; j<N; j++){
        a[j] += b[i][j];
    }
}
printf(" a[0] a[N-1]: %f %f\n", a[0], a[N-1]);
return 0;
```

Exercise

- Go back to your parallel mandel.c program.
- Using what we've learned in this block of slides can you improve the runtime?

Optimizing mandel.c

```
wtime = omp_get_wtime();
#pragma omp parallel for collapse(2) schedule(runtime) firstprivate(eps) private(j,c)
for (i=0; i<NPOINTS; i++) {
    for (j=0; j<NPOINTS; j++) {
        c.r = -2.0+2.5*(double)(i)/(double)(NPOINTS)+eps;
        c.i = 1.125*(double)(j)/(double)(NPOINTS)+eps;
        testpoint(c);
    }
}
wtime = omp_get_wtime() - wtime;
```

```
$ export OMP_SCHEDULE="dynamic,100"
$ ./mandel_par
```

default schedule	0.48 secs
schedule(dynamic,100)	0.39 secs
collapse(2) schedule(dynamic,100)	0.34 secs

Four threads on a dual core Apple laptop (Macbook air ... 2.2 Ghz Intel Core i7 with 8 GB memory)
and the gcc version 9.1. Times are the minimum time from three runs

- Introduction to OpenMP
- Creating Threads
- Synchronization
- Parallel Loops
- Data Environment
- Memory Model
- Irregular Parallelism and Tasks
- Worksharing Revisited
- • Synchronization Revisited: Options for Mutual exclusion
- Threadprivate and the joys of “random” numbers
- Recap

Synchronization

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

- High level synchronization included in the common core:

- critical
- barrier

Covered earlier

- Other, more advanced, synchronization operations:

- atomic
- ordered
- flush
- locks (both simple and nested)

Covered in this section

Synchronization: Atomic

- Atomic provides mutual exclusion but only applies to the update of a memory location (the update of X in the following example)

```
#pragma omp parallel
```

```
{
```

```
    double B;
```

```
    B = DOIT();
```

```
#pragma omp atomic
```

```
    X += big_ugly(B);
```

```
}
```

Synchronization: Atomic

- Atomic provides mutual exclusion but only applies to the update of a memory location (the update of X in the following example)

```
#pragma omp parallel  
{  
    double B, tmp;  
    B = DOIT();  
    tmp = big_ugly(B);  
#pragma omp atomic  
    X += tmp;  
}
```

Atomic only protects the
read/update of X

The OpenMP 3.1 Atomics (1 of 2)

- Atomic was expanded to cover the full range of common scenarios where you need to protect a memory operation so it occurs atomically:

```
# pragma omp atomic [read | write | update | capture]
```

- Atomic can protect loads

```
# pragma omp atomic read
```

```
v = x;
```

- Atomic can protect stores

```
# pragma omp atomic write
```

```
x = expr;
```

- Atomic can protect updates to a storage location (this is the default behavior ... i.e. when you don't provide a clause)

```
# pragma omp atomic update
```

```
x++; or ++x; or x--; or -x; or
```

```
x binop= expr; or x = x binop expr;
```

This is the
original OpenMP
atomic

The OpenMP 3.1 Atomics (2 of 2)

- Atomic can protect the assignment of a value (its capture) AND an associated update operation:

```
# pragma omp atomic capture  
statement or structured block
```

- Where the statement is one of the following forms:

v = x++; **v = ++x;** **v = x--;** **v = -x;** **v = x binop expr;**

- Where the structured block is one of the following forms:

{v = x; x binop = expr;}

{v=x; x=x binop expr;}

{v = x; x++;}

{++x; v=x:}

{v = x; x--;}

{--x; v = x;}

{x binop = expr; v = x;}

{X = x binop expr; v = x;}

{v=x; ++x:}

{x++; v = x;}

{v = x; --x;}

{x--; v = x;}

The capture semantics in atomic were added to map onto common hardware supported atomic operations and to support modern lock free algorithms

Synchronization: Lock Routines

- Simple Lock routines:
 - A simple lock is available if it is unset.
 - `omp_init_lock()`, `omp_set_lock()`,
`omp_unset_lock()`, `omp_test_lock()`, `omp_destroy_lock()`
- Nested Locks
 - A nested lock is available if it is unset or if it is set but owned by the thread executing the nested lock function
 - `omp_init_nest_lock()`, `omp_set_nest_lock()`, `omp_unset_nest_lock()`,
`omp_test_nest_lock()`, `omp_destroy_nest_lock()`

A lock implies a memory fence (a “flush”) of all thread visible variables

Note: a thread always accesses the most recent copy of the lock, so you don't need to use a flush on the lock variable.

Locks with hints were added in OpenMP 4.5 to suggest a lock strategy based on intended use (e.g. contended, uncontended, speculative, unspeculative)

Synchronization: Simple Locks Example

- Count odds and evens in an input array(x) of N random values.

```
int i, ix, even_count = 0, odd_count = 0;  
omp_lock_t odd_lck, even_lck;  
omp_init_lock(&odd_lck);  
omp_init_lock(&even_lck);
```

One lock per case ... even and odd

```
#pragma omp parallel for private(ix) shared(even_count, odd_count)  
for(i=0; i<N; i++){  
    ix = (int) x[i]; //truncate to int
```

```
    if((int) x[i])%2 == 0 {  
        omp_set_lock(&even_lck);  
        even_count++;  
        omp_unset_lock(&even_lck);  
    }  
    else{  
        omp_set_lock(&odd_lck);  
        odd_count++;  
        omp_unset_lock(&odd_lck);  
    }  
}  
omp_destroy_lock(&odd_lck);  
omp_destroy_lock(&even_lck);  
}
```

Enforce mutual exclusion updates,
but in parallel for each case.

Free-up storage when done.

Exercise

- In the file hist.c, we provide a program that generates a large array of random numbers and then generates a histogram of values.
- This is a "quick and informal" way to test a random number generator ... if all goes well the bins of the histogram should be the same size.
- Parallelize the filling of the histogram
You must assure that your program is race free and gets the same result as the sequential program.
- Using everything we've covered today, **manage updates to shared data in multiple ways**. Try to minimize the time to generate the histogram.
- Time ONLY the assignment to the histogram. Can you beat the sequential time?

```
#define num_trials 1000000 // number of x values
#define num_buckets 50      // number of buckets in histogram
static long xlow        = 0.0;      // low end of x range
static long xhi         = 100.0;    // High end of x range

int main (){
    double x[num_trials];    // array used to assign counters in the histogram
    long hist[num_buckets]; // the histogram
    double bucket_width;    // the width of each bucket in the histogram
    double time;

    seed(xlow, xhi); // seed the random number generator over range of x
    bucket_width = (xhi-xlow)/(double)num_buckets;

    // fill the array. << code not shown >>

    // initialize the histogram << code not shown >>

    // Assign x values to the right histogram bucket
    time = omp_get_wtime();
    for(int i=0;i<num_trials;i++){

        long ival = (long) (x[i] - xlow)/bucket_width;
        hist[ival]++;
    }

    time = omp_get_wtime() - time;

    // compute statistics and output results << code not shown >>
    return 0;
}
```

Only focus
on this part of
the program

Histogram Program: Critical section

- A critical section means that only one thread at a time can update a histogram bin ... but this effectively serializes the loops and adds huge overhead as the runtime manages all the threads waiting for their turn for the update.

```
#pragma omp parallel for
for(i=0;i<NVALS;i++){
    ival = (int) x[i];
    #pragma omp critical
        hist[ival]++;
}
```

Easy to write and
correct, but terrible
performance

Histogram program: one lock per histogram bin

- Example: conflicts are rare, but to play it safe, we must assure mutual exclusion for updates to histogram elements.

```
#pragma omp parallel for
for(i=0;i<NBUCKETS; i++){
    omp_init_lock(&hist_locks[i]);
    hist[i] = 0;
}

#pragma omp parallel for
for(i=0;i<NVALS;i++){
    ival = (int) x[i];
    omp_set_lock(&hist_locks[ival]);
    hist[ival]++;
    omp_unset_lock(&hist_locks[ival]);
}

#pragma omp parallel for
for(i=0;i<NBUCKETS; i++)
    omp_destroy_lock(&hist_locks[i]);
```

One lock per element of hist

Enforce mutual exclusion on update to hist array

Free-up storage when done.

Histogram program: reduction with an array

- We can give each thread a copy of the histogram, they can fill them in parallel, and then combine them when done

```
#pragma omp parallel for reduction(+:hist[0:Nbins])
for(i=0;i<NVALS;i++){
    ival = (int) x[i];
    hist[ival]++;
}
```

Easy to write and correct, Uses a lot of memory on the stack, but its fast ... sometimes faster than the serial method.

sequential	0.0019 secs
critical	0.079 secs
Locks per bin	0.029 secs
Reduction, replicated histogram array	0.00097 secs

1000000 random values in X sorted into 50 bins. Four threads on a dual core Apple laptop (Macbook air ... 2.2 Ghz Intel Core i7 with 8 GB memory) and the gcc version 9.1. Times are for the above loop only (we do not time set-up for locks, destruction of locks or anything else)

Outline

OpenMP®

- Introduction to OpenMP
- Creating Threads
- Synchronization
- Parallel Loops
- Data Environment
- Memory Model
- Irregular Parallelism and Tasks
- Worksharing Revisited
- Synchronization Revisited: Options for Mutual exclusion
- ➡ • Threadprivate and the joys of “random” numbers
- Recap

Data Sharing: Threadprivate

- Makes global data private to a thread
 - Fortran: **COMMON** blocks
 - C: File scope and static variables, static class members
- Different from making them **PRIVATE**
 - with **PRIVATE** global variables are masked.
 - **THREADPRIVATE** preserves global scope within each thread
- Threadprivate variables can be initialized using **COPYIN** or at time of definition (using language-defined initialization capabilities)

A Threadprivate Example (C)

Use `threadprivate` to create a counter for each thread.

```
int counter = 0;  
#pragma omp threadprivate(counter)  
  
int increment_counter()  
{  
    counter++;  
    return (counter);  
}
```

Data Copying: Copyin

You initialize threadprivate data using a copyin clause.

```
parameter (N=1000)
common/buf/A(N)
!$OMP THREADPRIVATE(/buf/)

!$ Initialize the A array
call init_data(N,A)

!$OMP PARALLEL COPYIN(A)
... Now each thread sees threadprivate array A initialized
... to the global value set in the subroutine init_data()

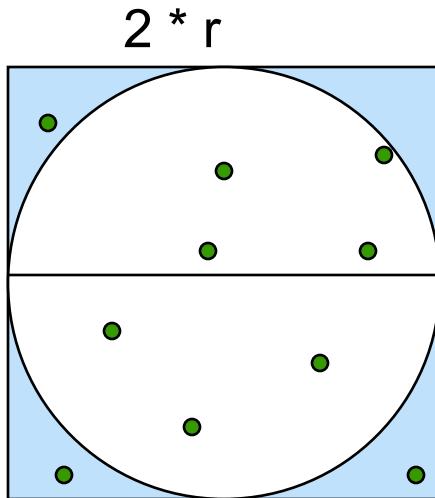
!$OMP END PARALLEL

end
```

Exercise: Monte Carlo Calculations

Using random numbers to solve tough problems

- Sample a problem domain to estimate areas, compute probabilities, find optimal values, etc.
- Example: Computing π with a digital dart board:



$N = 10$	$\pi = 2.8$
$N=100$	$\pi = 3.16$
$N= 1000$	$\pi = 3.148$

- Throw darts at the circle/square.
- Chance of falling in circle is proportional to ratio of areas:
$$A_c = r^2 * \pi$$
$$A_s = (2 * r) * (2 * r) = 4 * r^2$$
$$P = A_c / A_s = \pi / 4$$
- Compute π by randomly choosing points; π is four times the fraction that falls in the circle

Exercise: Monte Carlo pi (cont)

- We provide three files for this exercise
 - pi_mc.c: the Monte Carlo method pi program
 - random.c: a simple random number generator
 - random.h: include file for random number generator
- Create a parallel version of this program.
- Run it multiple times with varying numbers of threads.
- Is the program working correctly? Is there anything wrong?

Parallel Programmers love Monte Carlo algorithms

```
#include "omp.h"
static long num_trials = 10000;
int main ()
{
    long i;    long Ncirc = 0;    double pi, x, y;
    double r = 1.0; // radius of circle. Side of square is 2*r
    seed(0,-r, r); // The circle and square are centered at the origin
    #pragma omp parallel for private (x, y) reduction (+:Ncirc)
    for(i=0;i<num_trials; i++)
    {
        x = random();      y = random();
        if ( x*x + y*y ) <= r*r) Ncirc++;
    }

    pi = 4.0 * ((double)Ncirc/(double)num_trials);
    printf("\n %d trials, pi is %f \n",num_trials, pi);
}
```

Embarrassingly parallel: the parallelism is so easy its embarrassing.

Add two lines and you have a parallel program.

Random Numbers: Linear Congruential Generator (LCG)

- LCG: Easy to write, cheap to compute, portable, OK quality

```
random_next = (MULTIPLIER * random_last + ADDEND)% PMOD;  
random_last = random_next;
```

- If you pick the multiplier and addend correctly, LCG has a period of PMOD.
- Picking good LCG parameters is complicated, so look it up (Numerical Recipes is a good source). I used the following:
 - ◆ MULTIPLIER = 1366
 - ◆ ADDEND = 150889
 - ◆ PMOD = 714025

LCG code

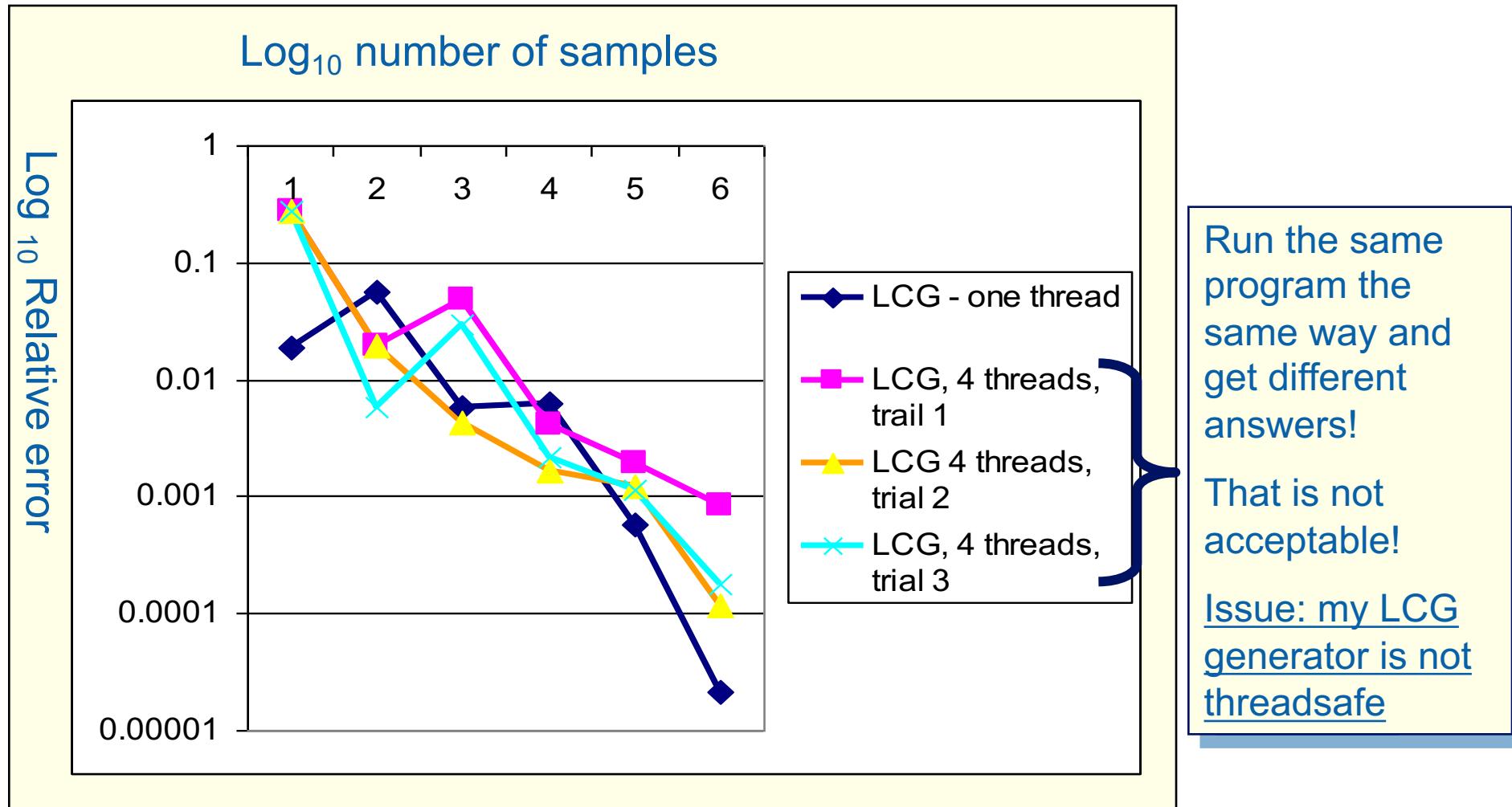
```
static long MULTIPLIER = 1366;
static long ADDEND    = 150889;
static long PMOD      = 714025;
long random_last = 0;
double random ()
{
    long random_next;

    random_next = (MULTIPLIER * random_last + ADDEND)% PMOD;
    random_last = random_next;

    return ((double)random_next/(double)PMOD);
}
```

Seed the pseudo random sequence by setting random_last

Running the PI_MC program with LCG generator



Program written using the Intel C/C++ compiler (10.0.659.2005) in Microsoft Visual studio 2005 (8.0.50727.42) and running on a dual-core laptop (Intel T2400 @ 1.83 Ghz with 2 GB RAM) running Microsoft Windows XP.

Exercise: Monte Carlo pi (cont)

- Create a threadsafe version of the monte carlo pi program
- Do not change the interfaces to functions in random.c
 - This is an exercise in modular software ... why should a user of your parallel random number generator have to know any details of the generator or make any changes to how the generator is called?
 - The random number generator must be thread-safe
- Verify that the program is thread safe by running multiple times for a fixed number of threads.
- Any concerns with the program behavior?

LCG code: threadsafe version

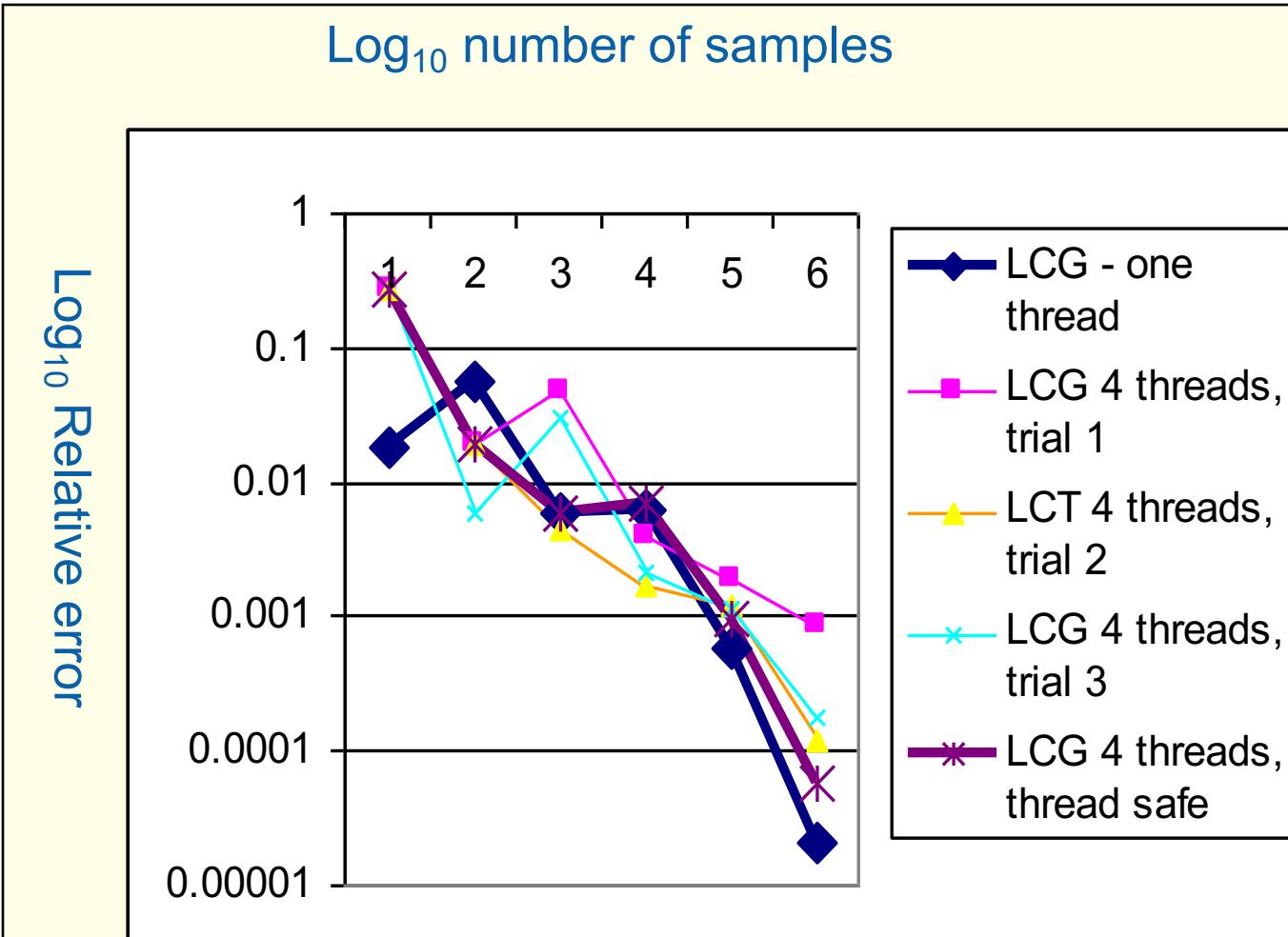
```
static long MULTIPLIER = 1366;
static long ADDEND    = 150889;
static long PMOD      = 714025;
long random_last = 0;
#pragma omp threadprivate(random_last)
double random ()
{
    long random_next;

    random_next = (MULTIPLIER * random_last + ADDEND)% PMOD;
    random_last = random_next;

    return ((double)random_next/(double)PMOD);
}
```

random_last carries state between random number computations, To make the generator threadsafe, make random_last threadprivate so each thread has its own copy.

Thread Safe Random Number Generators



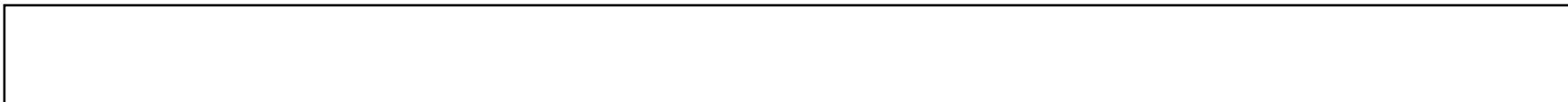
Thread safe version gives the same answer each time you run the program.

But for large number of samples, its quality is lower than the one thread result!

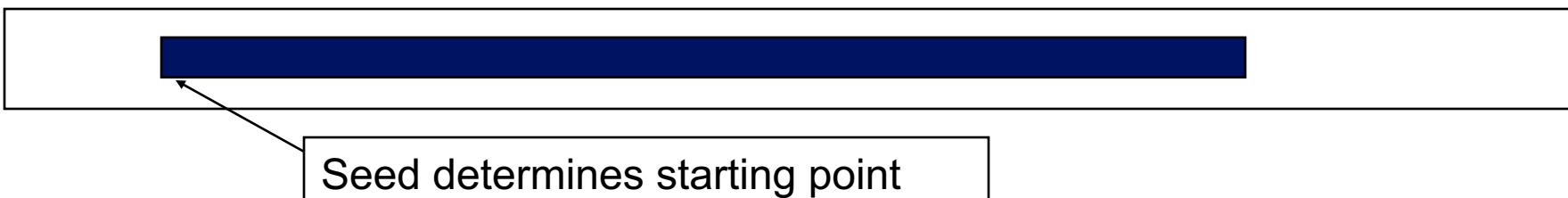
Why?

Pseudo Random Sequences

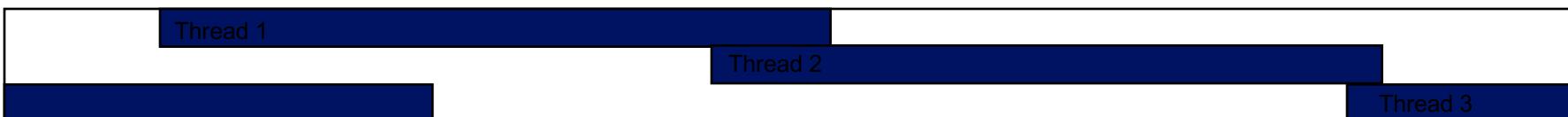
- Random number Generators (RNGs) define a sequence of pseudo-random numbers of length equal to the period of the RNG



- In a typical problem, you grab a subsequence of the RNG range



- Grab arbitrary seeds and you may generate overlapping sequences
 - ◆ E.g. three sequences ... last one wraps at the end of the RNG period.



- Overlapping sequences = over-sampling and bad statistics ... lower quality or even wrong answers!

Parallel random number generators

- Multiple threads cooperate to generate and use random numbers.
- Solutions:
 - Replicate and Pray
 - Give each thread a separate, independent generator
 - Have one thread generate all the numbers.
 - Leapfrog ... deal out sequence values “round robin” as if dealing a deck of cards.
 - Block method ... pick your seed so each threads gets a distinct contiguous block.
- Other than “replicate and pray”, these are difficult to implement. Be smart ... get a math library that does it right.

If done right, can generate the same sequence regardless of the number of threads ...

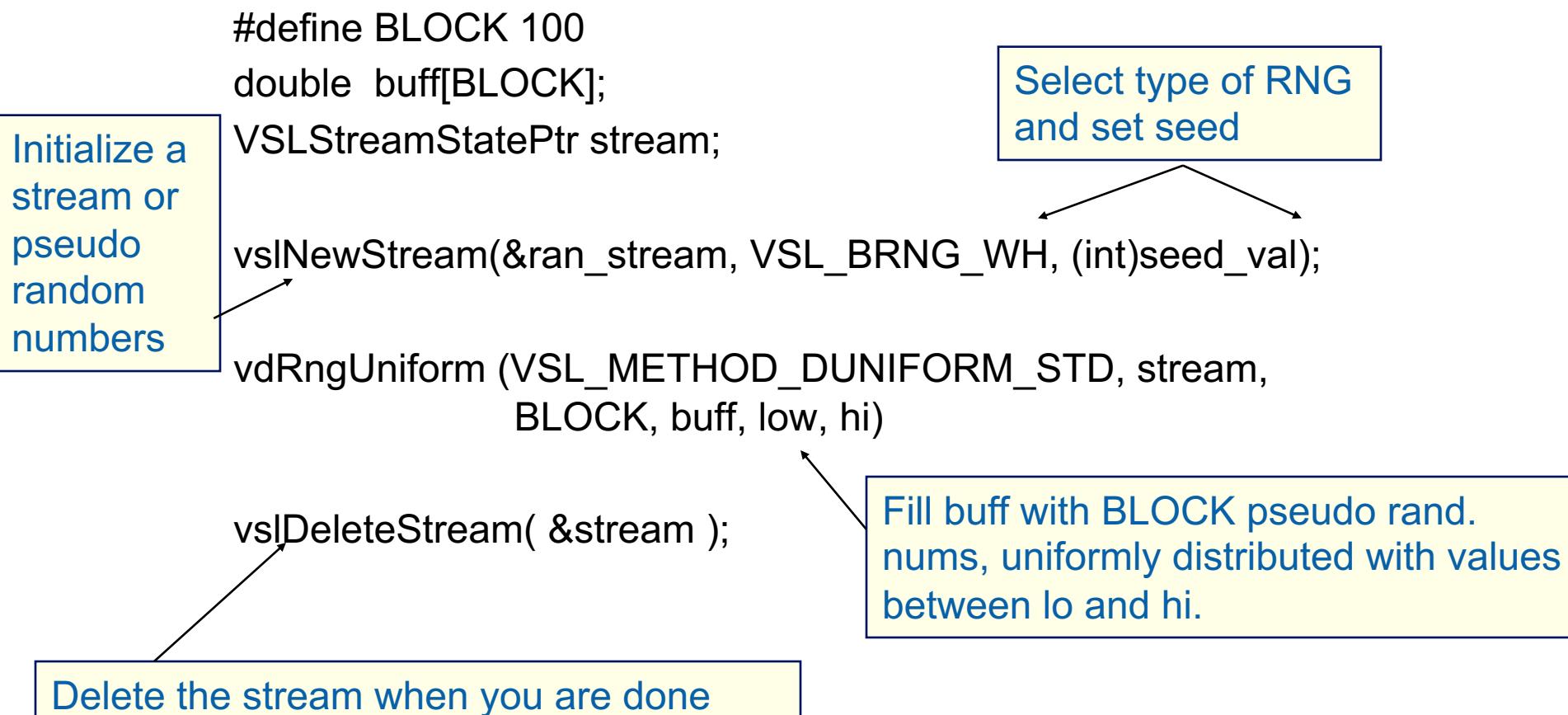
Nice for debugging, but not really needed scientifically.

Intel's Math kernel Library supports a wide range of parallel random number generators.

For an open alternative, the state of the art is the Scalable Parallel Random Number Generators Library (SPRNG): <http://www.sprng.org/> from Michael Mascagni's group at Florida State University.

MKL Random Number Generators (RNG)

- MKL includes several families of RNGs in its vector statistics library.
- Specialized to efficiently generate vectors of random numbers

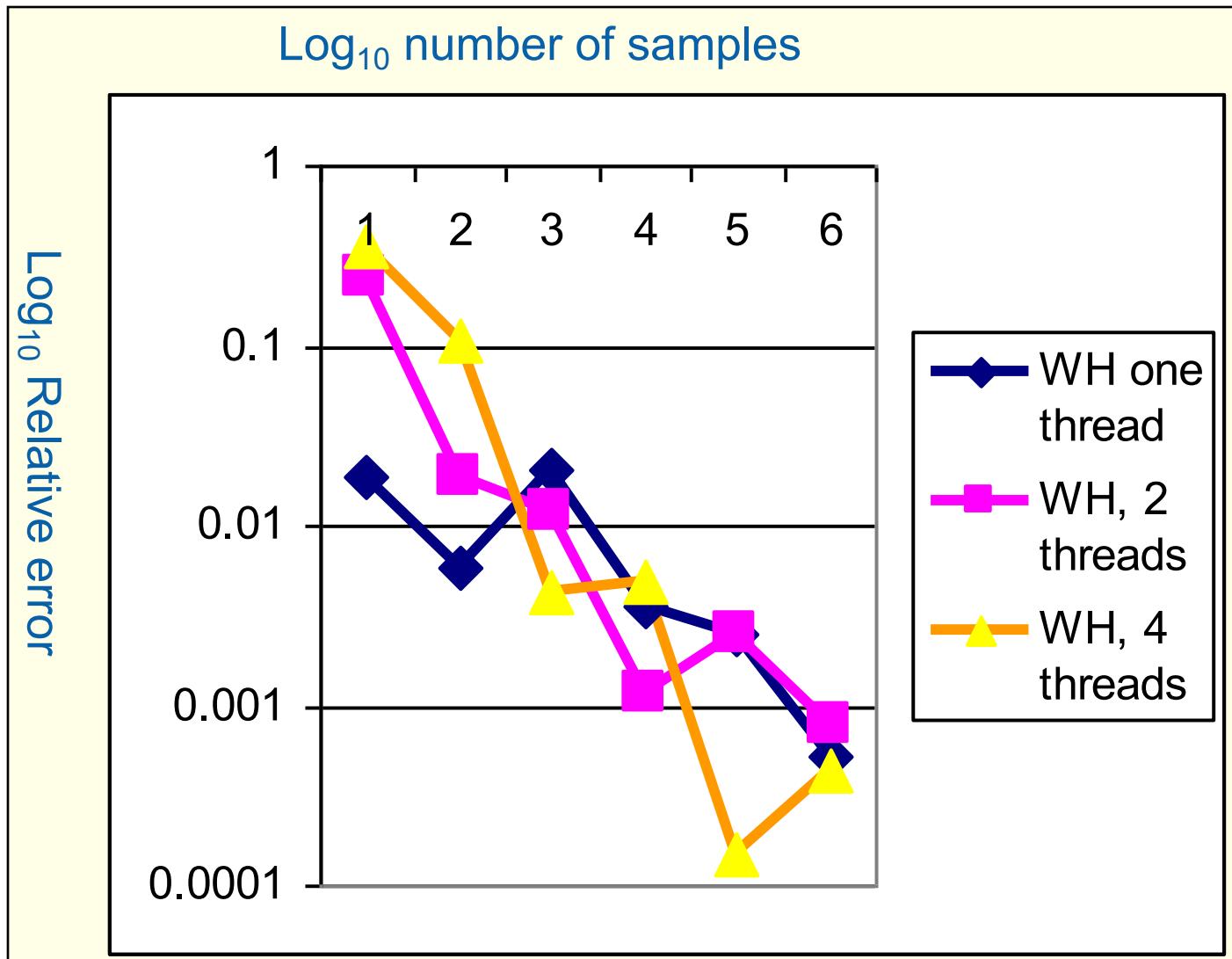


Wichmann-Hill Generators (WH)

- WH is a family of 273 parameter sets each defining a non-overlapping and independent RNG.
- Easy to use, just make each stream threadprivate and initiate RNG stream so each thread gets a unique WG RNG.

```
VSLStreamStatePtr stream;  
#pragma omp threadprivate(stream)  
  
....  
  
vslNewStream(&ran_stream, VSL_BRNG_WH+Thrd_ID, (int)seed);
```

Independent Generator for each thread



Notice that once you get beyond the high error, small sample count range, adding threads doesn't decrease quality of random sampling.

Leap Frog Method

- Interleave samples in the sequence of pseudo random numbers:
 - Thread i starts at the i^{th} number in the sequence
 - Stride through sequence, stride length = number of threads.
- Result ... the same sequence of values regardless of the number of threads.

```
#pragma omp single
{  nthreads = omp_get_num_threads();
   iseed = PMOD/MULTIPLIER;    // just pick a seed
   pseed[0] = iseed;
   mult_n = MULTIPLIER;
   for (i = 1; i < nthreads; ++i)
   {
      iseed = (unsigned long long)((MULTIPLIER * iseed) % PMOD);
      pseed[i] = iseed;
      mult_n = (mult_n * MULTIPLIER) % PMOD;
   }
}
random_last = (unsigned long long) pseed[id];
```

One thread computes offsets and strided multiplier

LCG with Addend = 0 just to keep things simple

Each thread stores offset starting point into its threadprivate “last random” value

Same sequence with many threads.

- We can use the leapfrog method to generate the same answer for any number of threads

Steps	One thread	2 threads	4 threads
1000	3.156	3.156	3.156
10000	3.1168	3.1168	3.1168
100000	3.13964	3.13964	3.13964
1000000	3.140348	3.140348	3.140348
10000000	3.141658	3.141658	3.141658

Used the MKL library with two generator streams per computation: one for the x values (WH) and one for the y values (WH+1). Also used the leapfrog method to deal out iterations among threads.

- Introduction to OpenMP
- Creating Threads
- Synchronization
- Parallel Loops
- Data Environment
- Memory Model
- Irregular Parallelism and Tasks
- Worksharing Revisited
- Synchronization Revisited: Options for Mutual exclusion
- Threadprivate and the joys of “random” numbers
- • Recap

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

OpenMP pragma, function, or clause	Concepts
#pragma omp parallel	Parallel region, teams of threads, structured block, interleaved execution across threads.
void omp_set_thread_num() int omp_get_thread_num() int omp_get_num_threads()	Default number of threads and internal control variables. SPMD pattern: Create threads with a parallel region and split up the work using the number of threads and the thread ID.
double omp_get_wtime()	Speedup and Amdahl's law. False sharing and other performance issues.
setenv OMP_NUM_THREADS N	Setting the internal control variable for the default number of threads with an environment variable
#pragma omp barrier #pragma omp critical	Synchronization and race conditions. Revisit interleaved execution.
#pragma omp for #pragma omp parallel for	Worksharing, parallel loops, loop carried dependencies.
reduction(op:list)	Reductions of values across a team of threads.
schedule (static [,chunk]) schedule(dynamic [,chunk])	Loop schedules, loop overheads, and load balance.
shared(list), private(list), firstprivate(list)	Data environment.
default(None)	Force explicit definition of each variable's storage attribute
nowait	Disabling implied barriers on workshare constructs, the high cost of barriers, and the flush concept (but not the flush directive).
#pragma omp single	Workshare with a single thread.
#pragma omp task #pragma omp taskwait	Tasks including the data environment for tasks.

There is Much More to OpenMP than the Common Core

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

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

OpenMP Organizations

- OpenMP Architecture Review Board (ARB) URL, the “owner” of the OpenMP specification:

www.openmp.org

- OpenMP User’s Group (cOMPunity) URL:

www.community.org

Get involved, join the ARB and cOMPunity.

Help define the future of OpenMP

Resources

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

The screenshot shows the OpenMP website's "Specifications" page. At the top, there's a large teal header with the OpenMP logo and the tagline "The OpenMP API specification for parallel programming". Below the header is a navigation bar with links for Home, Specifications, Community, Resources, News & Events, About, and a search icon. The main content area has a teal background and features two large cards. The left card is for the "OpenMP 5.2 Specification" and the right card is for the "OpenMP 5.1 Specification". Each card has a circular icon with a document symbol, the specification name, and a list of related links. A callout bubble from the text box on the left points to the "OpenMP API 5.2 Examples" link in the 5.2 specification card.

The OpenMP API specification for parallel programming

Home Specifications Community ▾ Resources ▾ News & Events ▾ About ▾

Specifications

Home > Specifications

OpenMP 5.2 Specification

- OpenMP API 5.2 Specification – Nov 2021
 - Softcover Book on Amazon
- OpenMP API Additional Definitions 2.0 – Nov 2020
- OpenMP API 5.2 Reference Guide (English) (Japanese)
- OpenMP API 5.2 Supplementary Source Code
- OpenMP API 5.2 Examples – April 2022
 - Softcover Book on Amazon
- OpenMP API 5.2 Stack Overflow

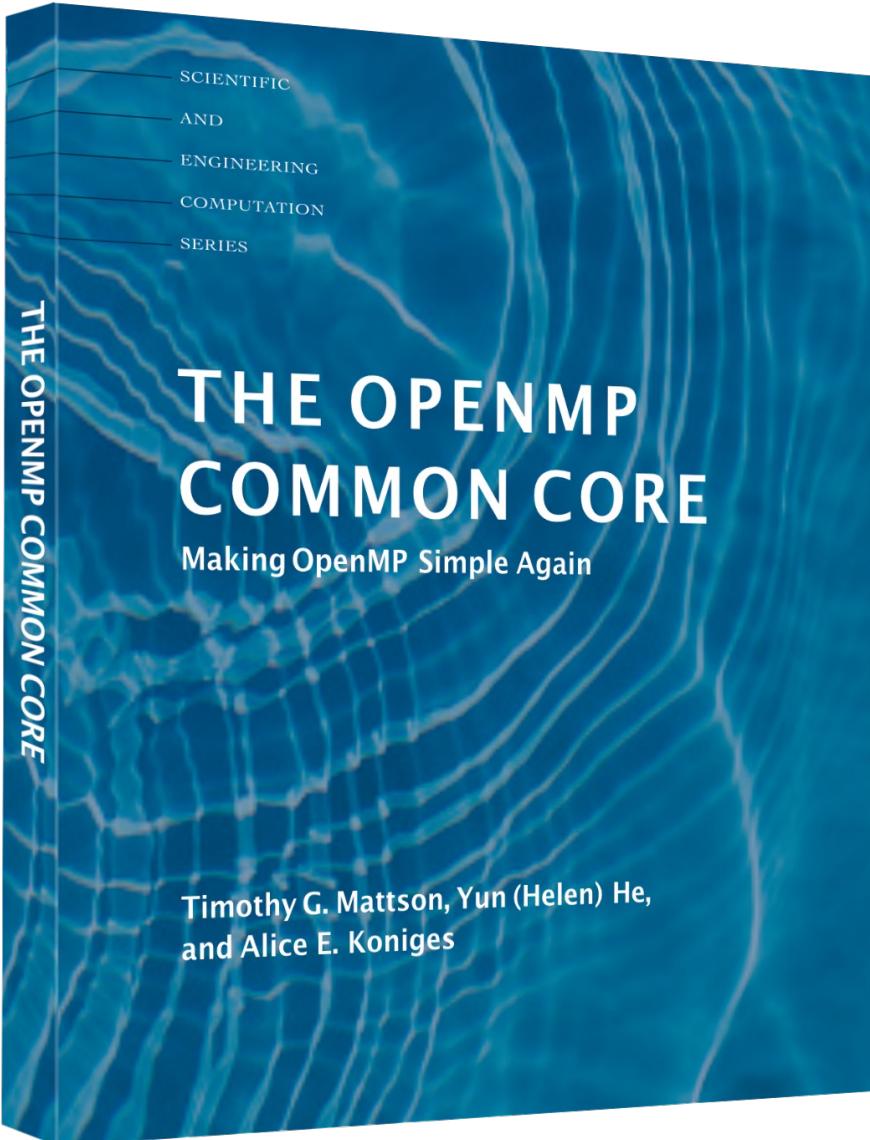
OpenMP 5.1 Specification

- OpenMP API 5.1 Specification – Nov 2020
 - HTML Version
 - Softcover Book on Amazon
- OpenMP API Additional Definitions 2.0 – Nov 2020
- OpenMP API 5.1 Reference Guide
- OpenMP API 5.1 Supplementary Source Code
- OpenMP API 5.1 Examples – August 2021
- OpenMP API 5.1 Stack Overflow

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

To learn OpenMP:

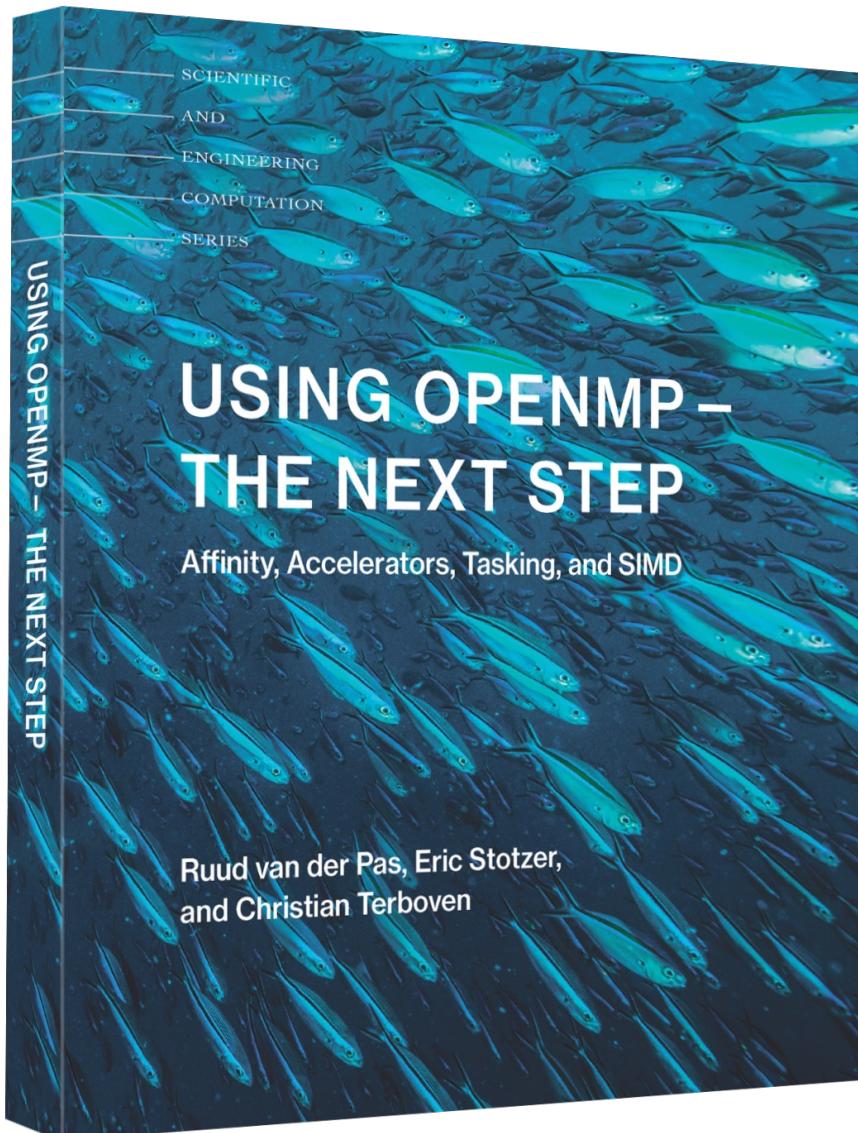
- An exciting new book that Covers the Common Core of OpenMP plus a few key features beyond the common core that people frequently use
- It's geared towards people learning OpenMP, but as one commentator put it ... **everyone at any skill level should read the memory model chapters.**
- Available from 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

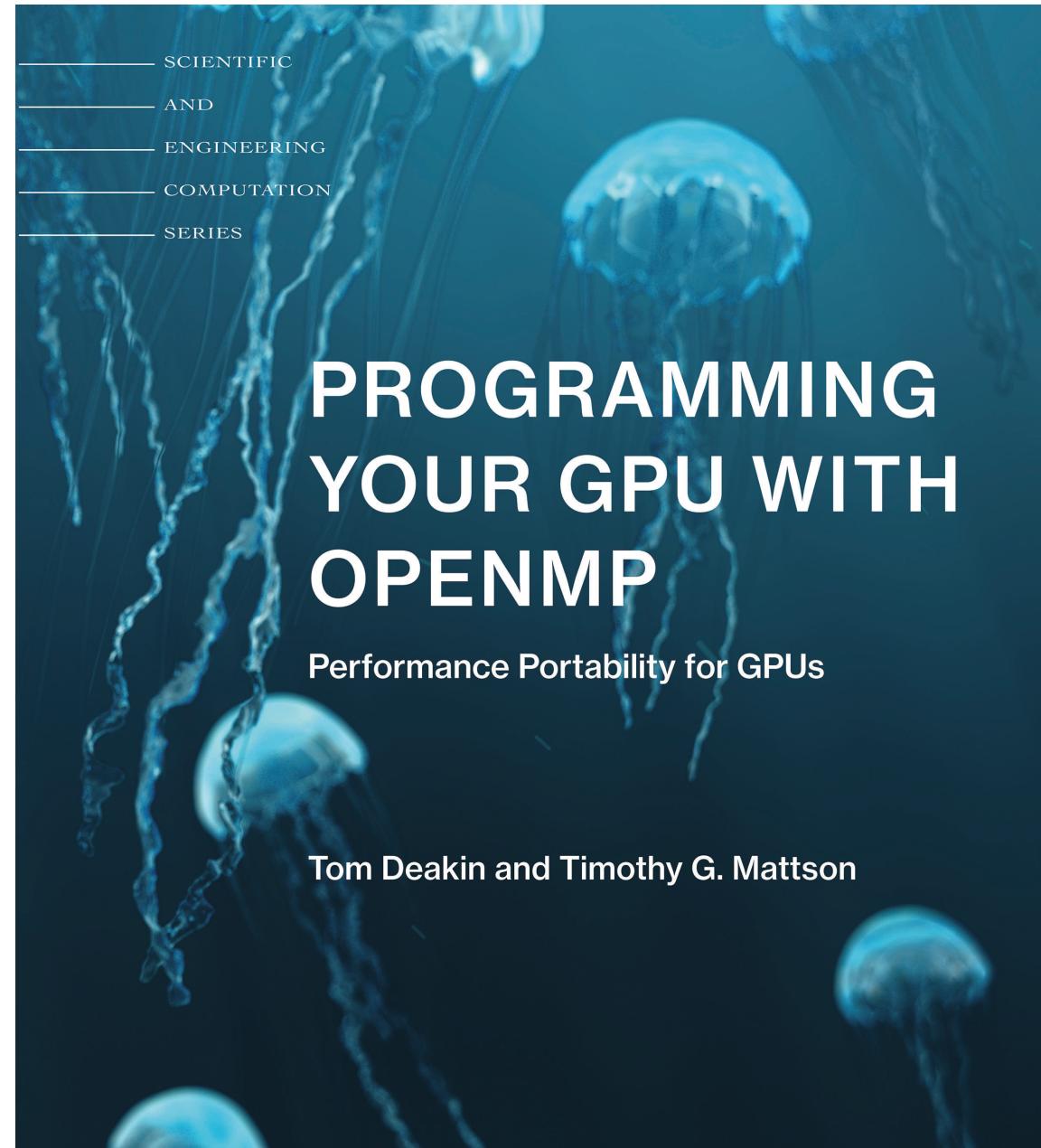


Books about OpenMP

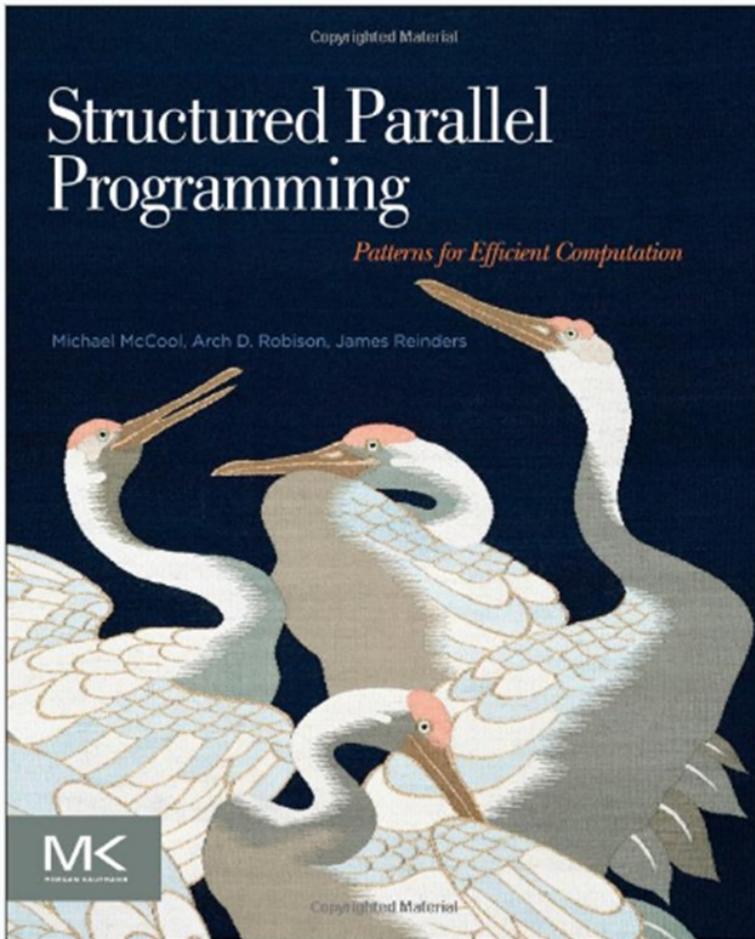
The latest book on OpenMP ...

Came out in early November 2023.

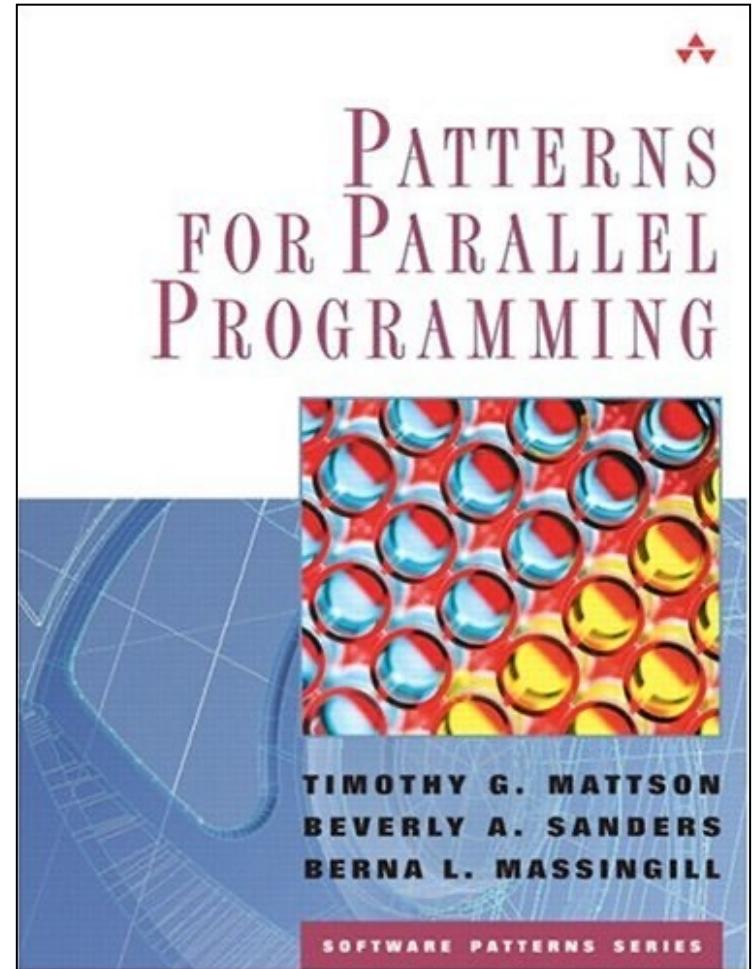
A book about how to use OpenMP to program a GPU.



Background references



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



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

Extras: One short extra section

- ➡ • Fortran and OpenMP

Fortran and OpenMP

- We were careful to design the OpenMP constructs so they cleanly map onto C, C++ and Fortran.
- There are a few syntactic differences that once understood, will allow you to move back and forth between languages.
- In the specification, language specific notes are included when each construct is defined.

OpenMP:

Some syntax details for Fortran programmers

- Most of the constructs in OpenMP are compiler directives.
 - For Fortran, the directives take one of the forms:

`C$OMP construct [clause [clause]...]`

`!$OMP construct [clause [clause]...]`

`*$OMP construct [clause [clause]...]`

- The OpenMP include file and lib module

`use omp_lib`

`include "omp_lib.h"`

OpenMP: Structured Blocks (Fortran)

- Most OpenMP constructs apply to structured blocks.
- Structured block: a block of code with one point of entry at the top and one point of exit at the bottom.
- The only “branches” allowed are STOP statements in Fortran and exit() in C/C++.

```
C$OMP PARALLEL  
10 wrk(id) = garbage(id)  
    res(id) = wrk(id)**2  
    if(conv(res(id)) goto 10  
C$OMP END PARALLEL  
print *,id
```

A structured block

```
C$OMP PARALLEL  
10 wrk(id) = garbage(id)  
30 res(id)=wrk(id)**2  
    if(conv(res(id))goto 20  
    go to 10  
C$OMP END PARALLEL  
    if(not_DONE) goto 30  
20 print *, id
```

Not A structured block

OpenMP:

Structured Block Boundaries

- In Fortran: a block is a single statement or a group of statements between directive/end-directive pairs.

```
C$OMP PARALLEL  
10  wrk(id) = garbage(id)  
     res(id) = wrk(id)**2  
     if(conv(res(id))) goto 10  
C$OMP END PARALLEL
```

```
C$OMP PARALLEL DO  
do I=1,N  
    res(I)=bigComp(I)  
end do  
C$OMP END PARALLEL DO
```

- The “construct/end construct” pairs is done anywhere a structured block appears in Fortran. Some examples:
 - DO ... END DO
 - PARALLEL ... END PARREL
 - CRITICAL ... END CRITICAL
 - SECTION ... END SECTION
 - SECTIONS ... END SECTIONS
 - SINGLE ... END SINGLE
 - MASTER ... END MASTER

Runtime Library Routines

- The include file or module defines parameters
 - Integer parameter `omp_loci_kind`
 - Integer parameter `omp_nest_lock_kind`
 - Integer parameter `omp_sched_kind`
 - Integer parameter `openmp_version`
 - With value that matches C's `_OPENMP` macro
- Fortran interfaces are similar to those used with C
 - Subroutine `omp_set_num_threads(num_threads)`
 - Integer function `omp_get_num_threads()`
 - Integer function `omp_get_thread_num()`
 - Subroutine `omp_init_lock(svar)`
 - Integer(kind=omp_lock_kind) svar
 - Subroutine `omp_destroy_lock(svar)`
 - Subroutine `omp_set_lock(svar)`
 - Subroutine `omp_unset_lock(svar)`

Nersc perlmutter

- To submit to the reservation, add the --reservation=<reservation-name> flag to your submission, either within the script by adding a line to the Slurm command preamble:
- `#SBATCH --reservation=<reservation-name>`
- OR on the command line when you submit the job:
- `sbatch --reservation=<reservation-name> ./myscript.sl`
- **To use the reserved nodes and share with multiple users**, add these flags to salloc or sbatch:
`-C gpu -q shared -A ntrain5 --reservation=<reservation_name> -N 1 -c 32 -G 1 -t 60:00`
- Please notice the **-q shared** and **-c 32 -G1** options. It will request 1/4 of the node's CPU and 1 GPU.
- Users can run either CPU or GPU jobs in this allocation.