A brief Introduction to parallel programming

Tim Mattson

Intel Corp.

timothy.g.mattson@ intel.com

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

1

Introduction

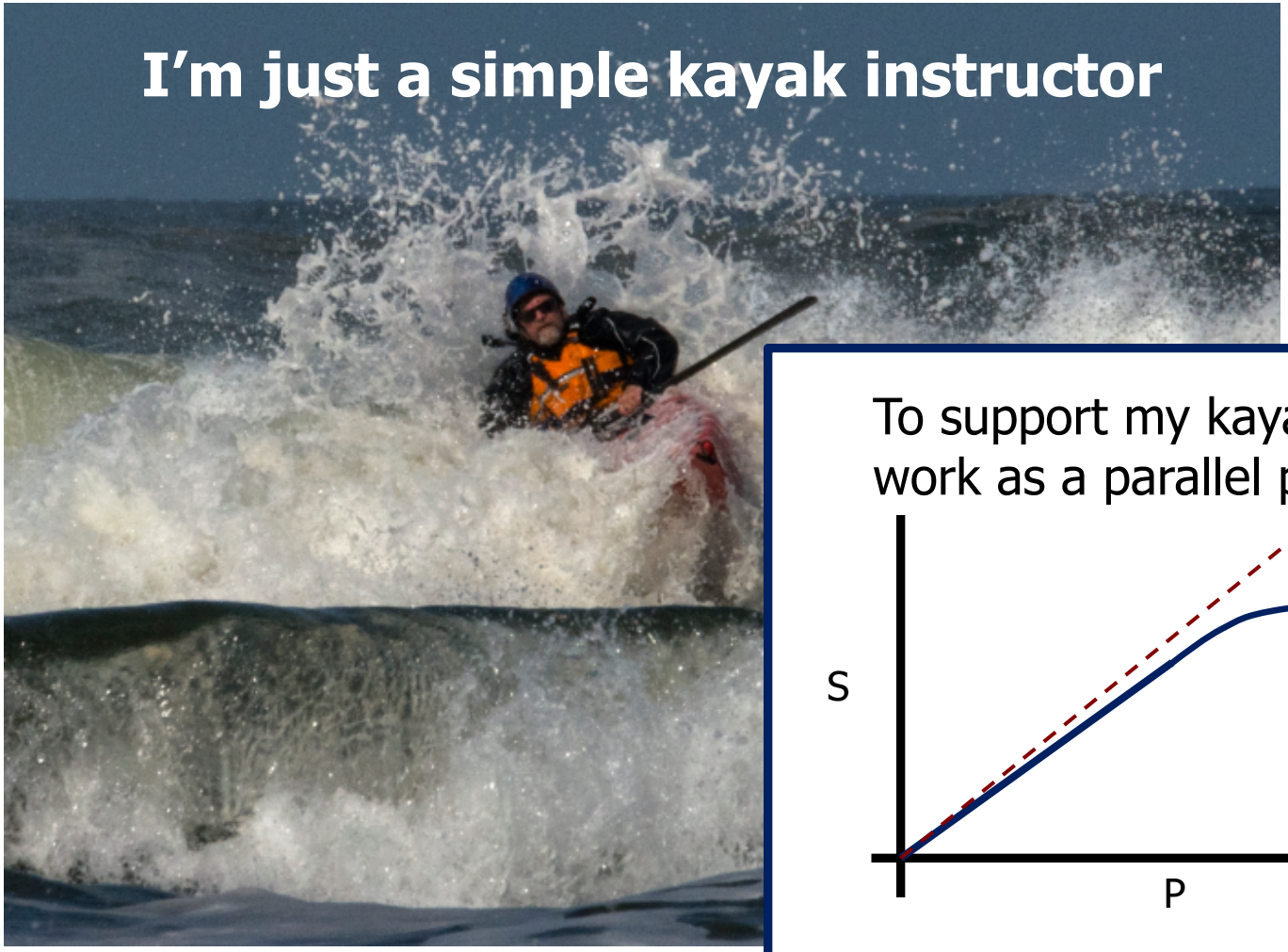


Photo © by Greg Clopton, 2014

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

Concurrency vs. Parallelism

- Two important definitions:
  - Concurrency: A condition of a system in which multiple tasks are ***logically*** active at one time.
  - Parallelism: A condition of a system in which multiple tasks are **actually** active at one time.

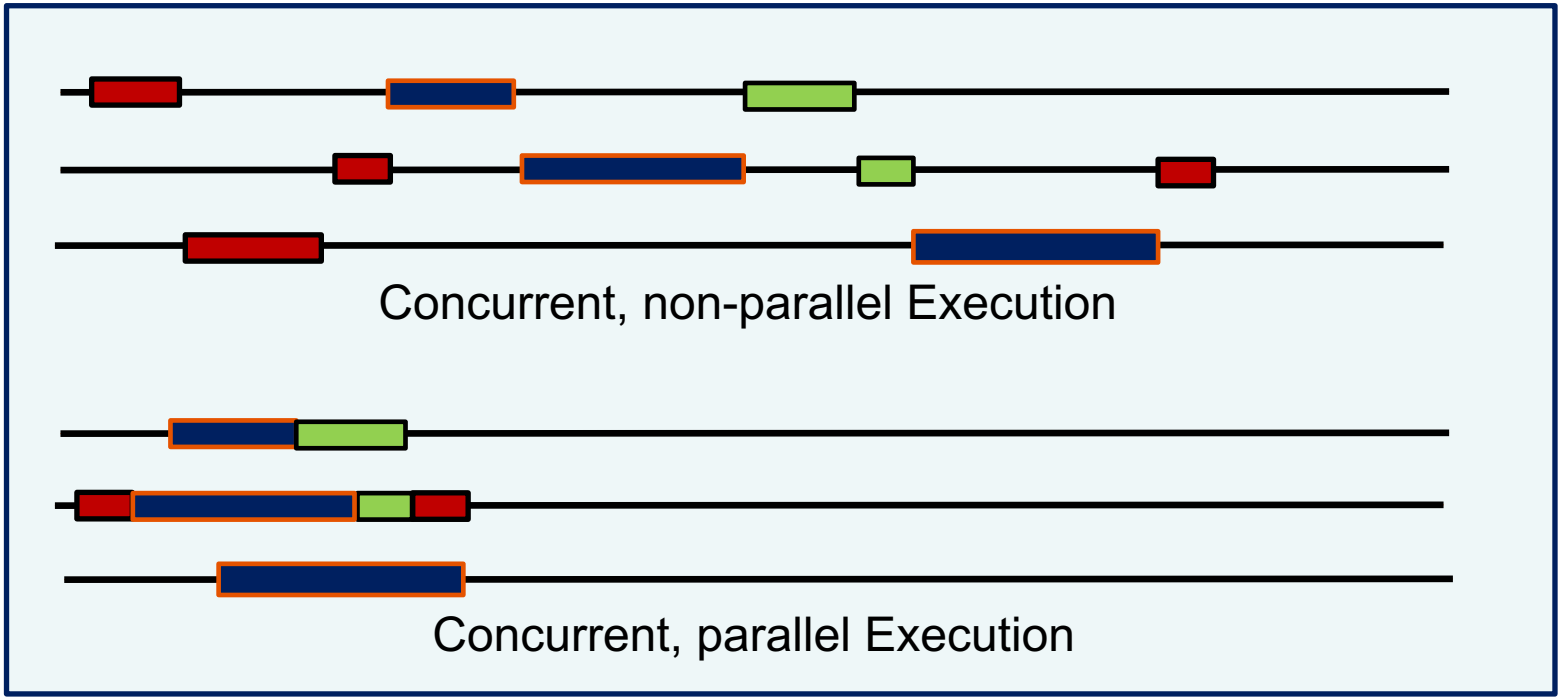
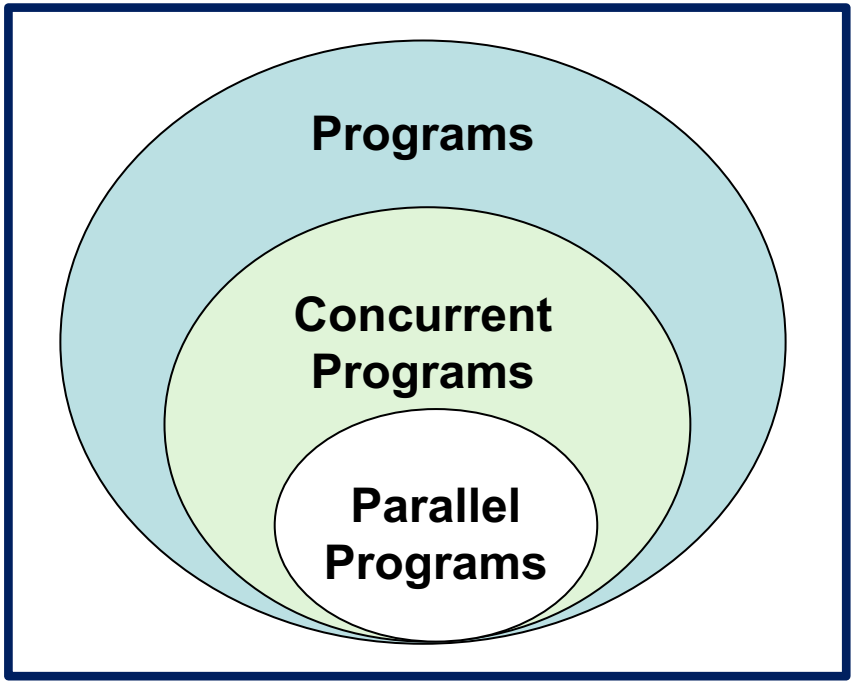


Figure from "An Introduction to Concurrency in Programming Languages" by J. Sottile, Timothy G. Mattson, and Craig E Rasmussen, 2010

Concurrency vs. Parallelism

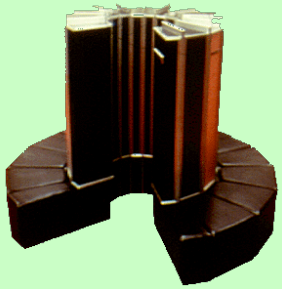
- Two important definitions:
  - Concurrency: A condition of a system in which multiple tasks are ***logically*** active at one time.
  - Parallelism: A condition of a system in which multiple tasks are **actually** active at one time.



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

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

Parallel computing: It's old



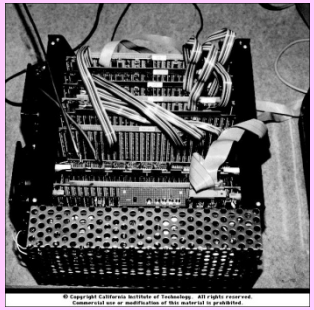
Cray 1 (1976)

Cray 2 (1985)

Cray C-90 (1991)

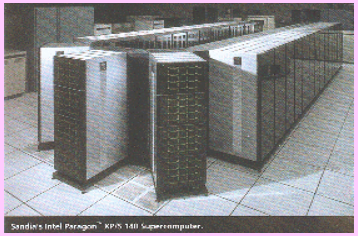
Vector Computers

SMP computers

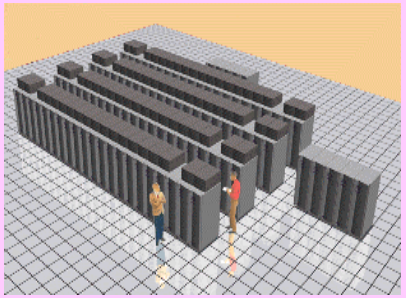


Cosmic cube (1983)

Massively Parallel Processors (MPP)



Paragon (1993)



ASCI Red (1997)

Clusters (late 80's)

Cluster Computers

Linux PC Clusters
(~1995)

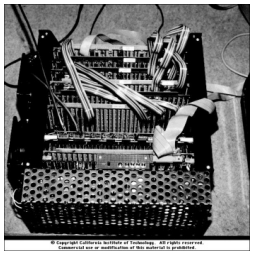
Late 70's

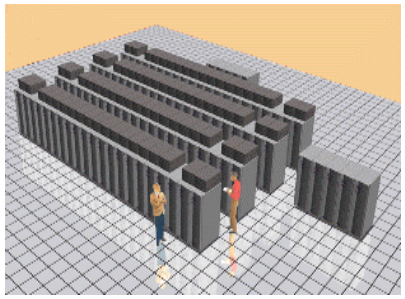
Late 80's

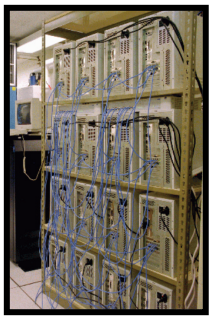
Late 90's

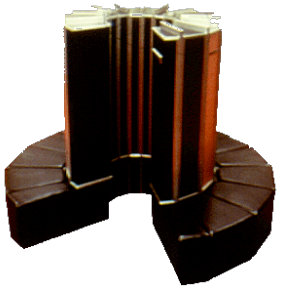
The Parallel Programming Problem

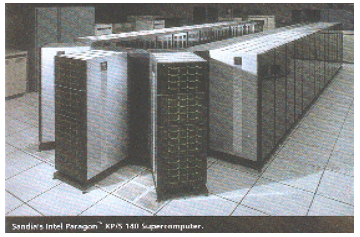
Even with all this cool hardware …











And some really beautiful architectural dead-ends …

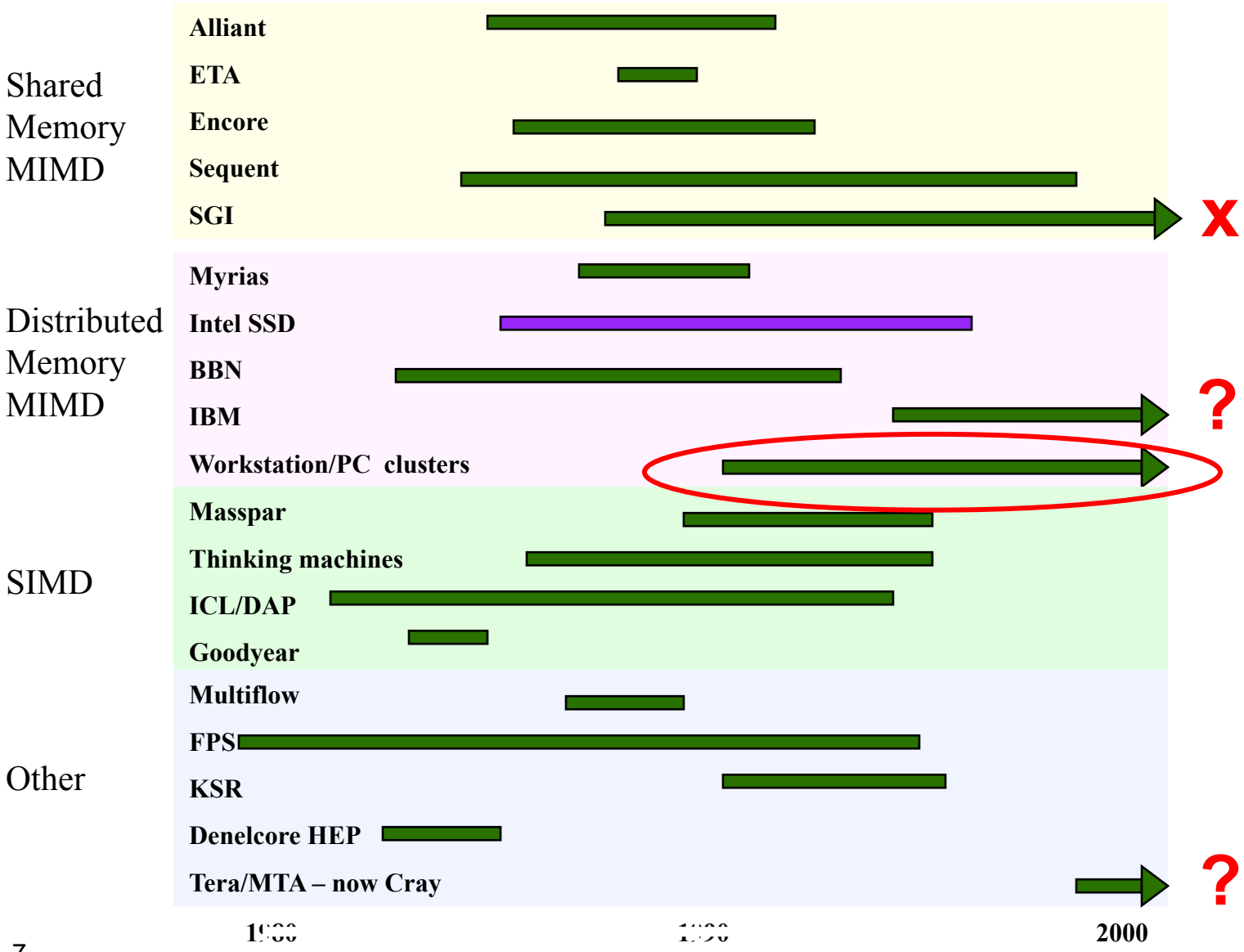
The Connection Machine

Very Few People Wrote parallel software.

And hardware without Software is useless

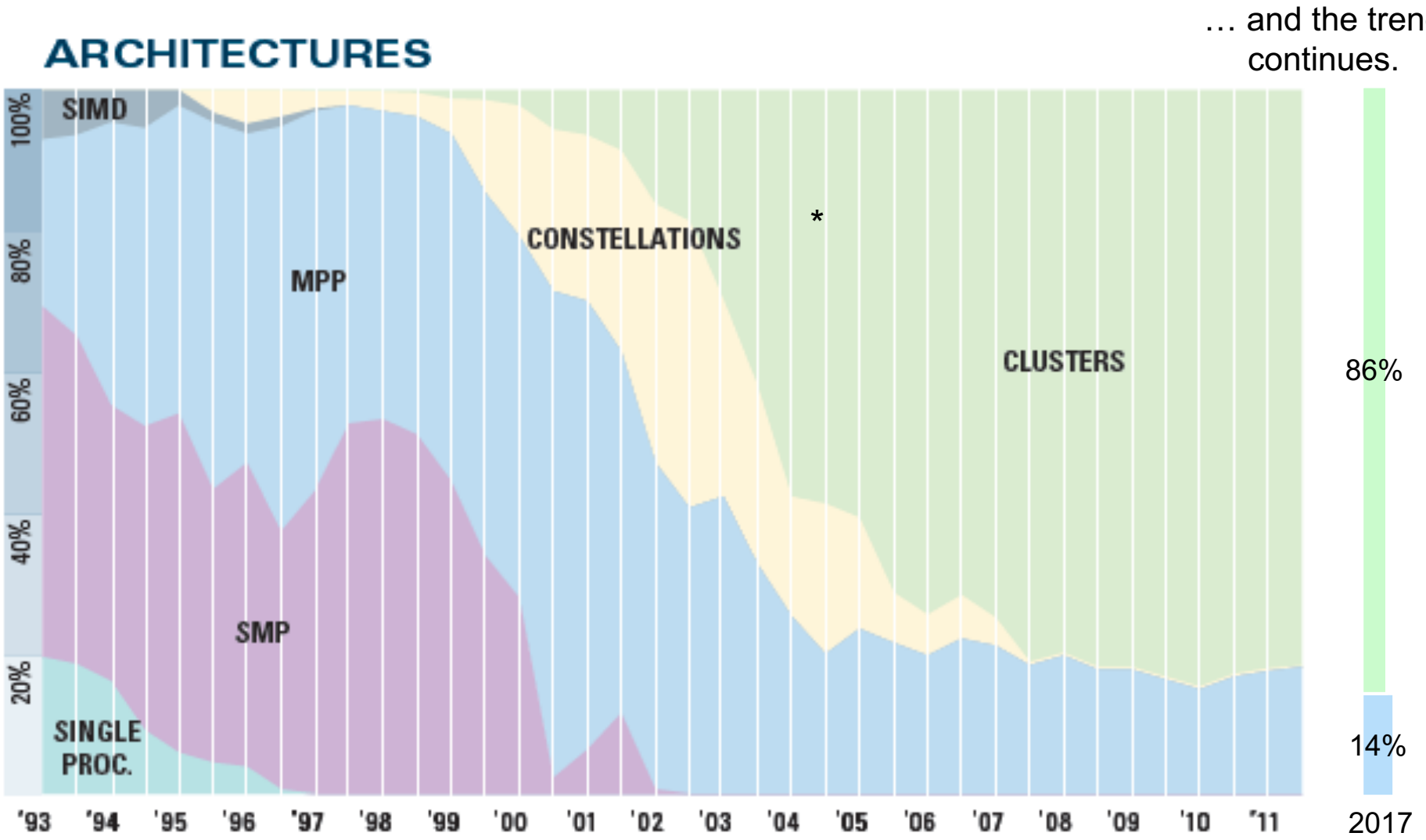
The Dead Architecture Society

A lack of application software cratered the market, ending the Golden age of parallel computing ended.



Top 500 list: System Architecture

… and the trend continues.



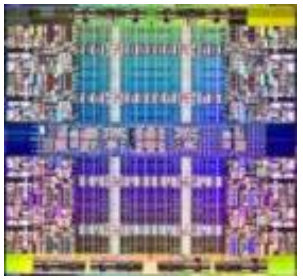
*Constellation: A cluster for which the number of processors on a node is greater than the number of nodes in the cluster. I've never seen anyone use this term outside of the top500 list.

Parallel computing: 2015-2020

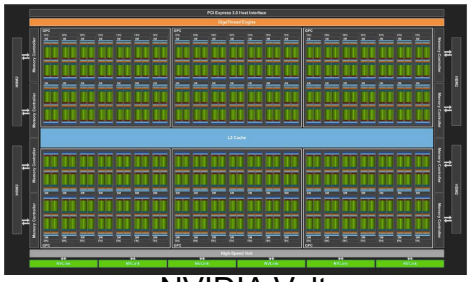
- Clusters with MPI

source: https://www.indiamart.com/nanohub/computer-servers.html

- Nodes in the cluster:
  - multicore CPU with OpenMP
  - GPU with CUDA*, OpenCL, OpenMP, OpenACC*
  - On the fringe, but interesting:
    - FPGA
    - Specialized accelerators



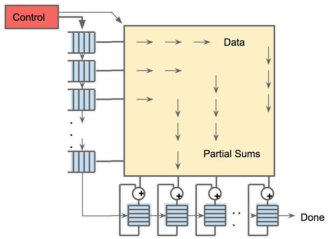
Intel® Haswell



NVIDIA Volta

Altera® (now intel) FPGA

Google TPU



*Proprietary solutions … avoid using if possible

Our agenda

- We will cover the full breadth of modern parallel programming on major classes of parallel hardware
- We'll spend most of our time learning the fundamental parallel programming concepts using OpenMP.
- Why OpenMP?
  - It's the easiest to use and requires the minimum learning overhead
  - Most key parallel design patterns can be learned with OpenMP.
  - You all have processors that can use OpenMP (multicore CPUs)
- We'll close with lectures covering
  - MPI and Clusters
  - OpenCL and GPUs
  - Specialized hardware: FPGA and fixed-function-processors

An Introduction to Parallel Programming using OpenMP

Tim Mattson

Senior Principal Engineer

Intel labs

timothy.g.mattson@ intel.com

11

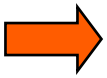
Preliminaries: Part 1

- Disclosures
  - The views expressed in this tutorial are those of the people delivering the tutorial.
    - We are not speaking for our employers.
    - 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.

Outline



- Introduction to OpenMP
- Creating Threads
- Quantifying Performance and Amdahl's law
- Synchronization
- Parallel Loops
- Data environment
- Memory model
- Irregular Parallelism and tasks
- Recap and the fundamental design patterns of parallel programming

OpenMP* overview:

C$OMP FLUSH

#pragma omp critical

C$OMP THREADPRIVATE(/ABC/)

CALL OMP_SET_NUM_THREADS(10)

C$OMP PARALLEL COPYIN(/blk/)

C$OMP DO lastprivate(XX)

Nthrds = OMP_GET_NUM_PROCS()

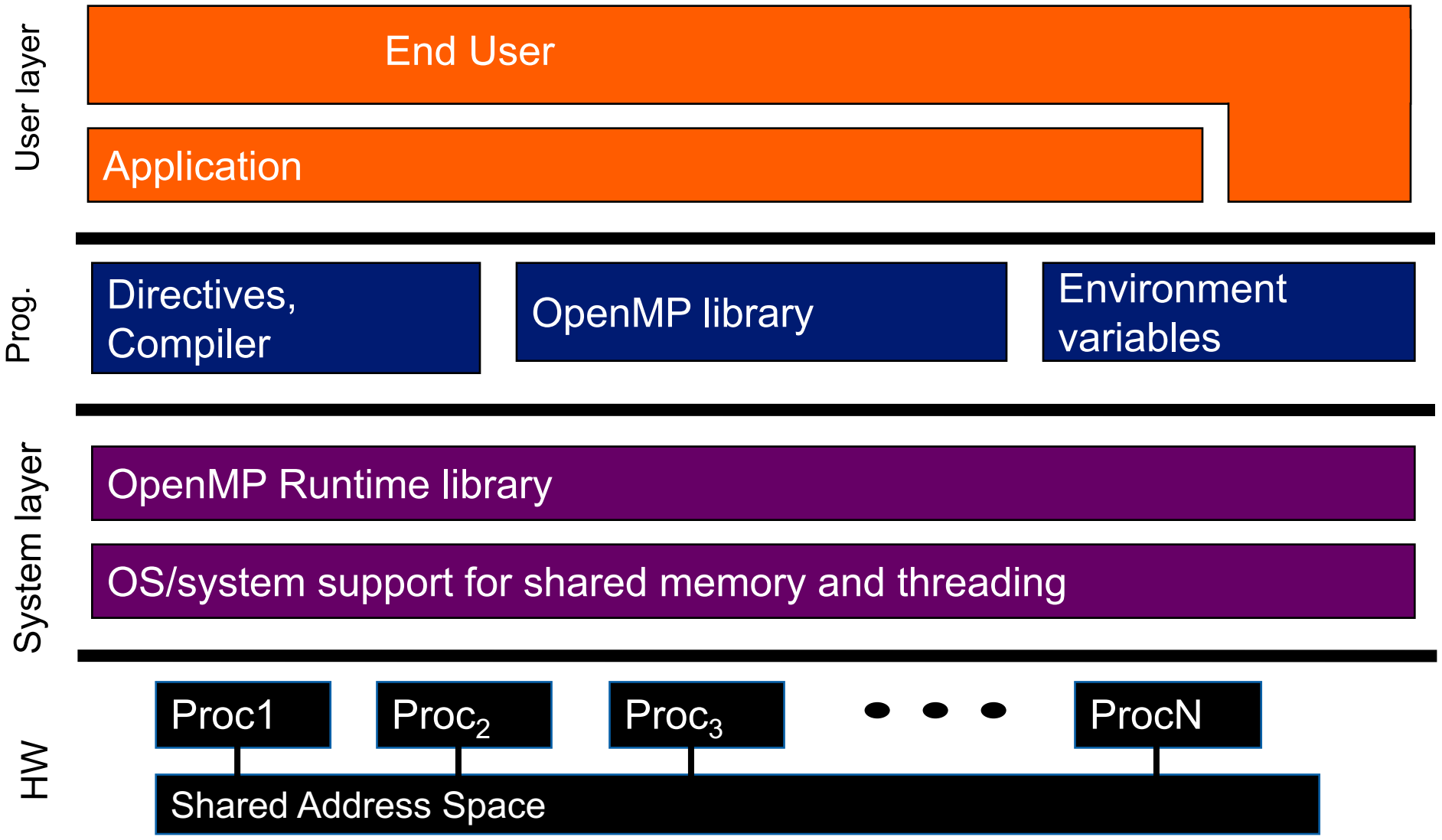
omp_set_lock(lck)

OpenMP: An API for Writing Multithreaded Applications

- A set of compiler directives and library routines for parallel application programmers
- Greatly simplifies writing multi-threaded (MT) programs in Fortran, C and C++
- Standardizes established SMP practice + vectorization and heterogeneous device programming

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

OpenMP basic definitions: Basic Solution stack



OpenMP core syntax

- Most of the constructs in OpenMP are compiler directives.

  #pragma omp construct [clause [clause]…]

  - Example

    #pragma omp parallel private(x)

- Function prototypes and types in the file:

  #include <omp.h>

- 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 1, 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 1, 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)**
pgcc -mp pgi	**PGI (Linux)**
icl /Qopenmp	**Intel (windows)**
icc –fopenmp	**Intel (Linux, OSX)**

Exercise 1: Solution

A multi-threaded “Hello world” program

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

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


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

OpenMP include file

Parallel region with default number of threads

End of the Parallel region

Sample Output:

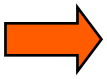
hello hello world

world

hello hello world

world

Outline

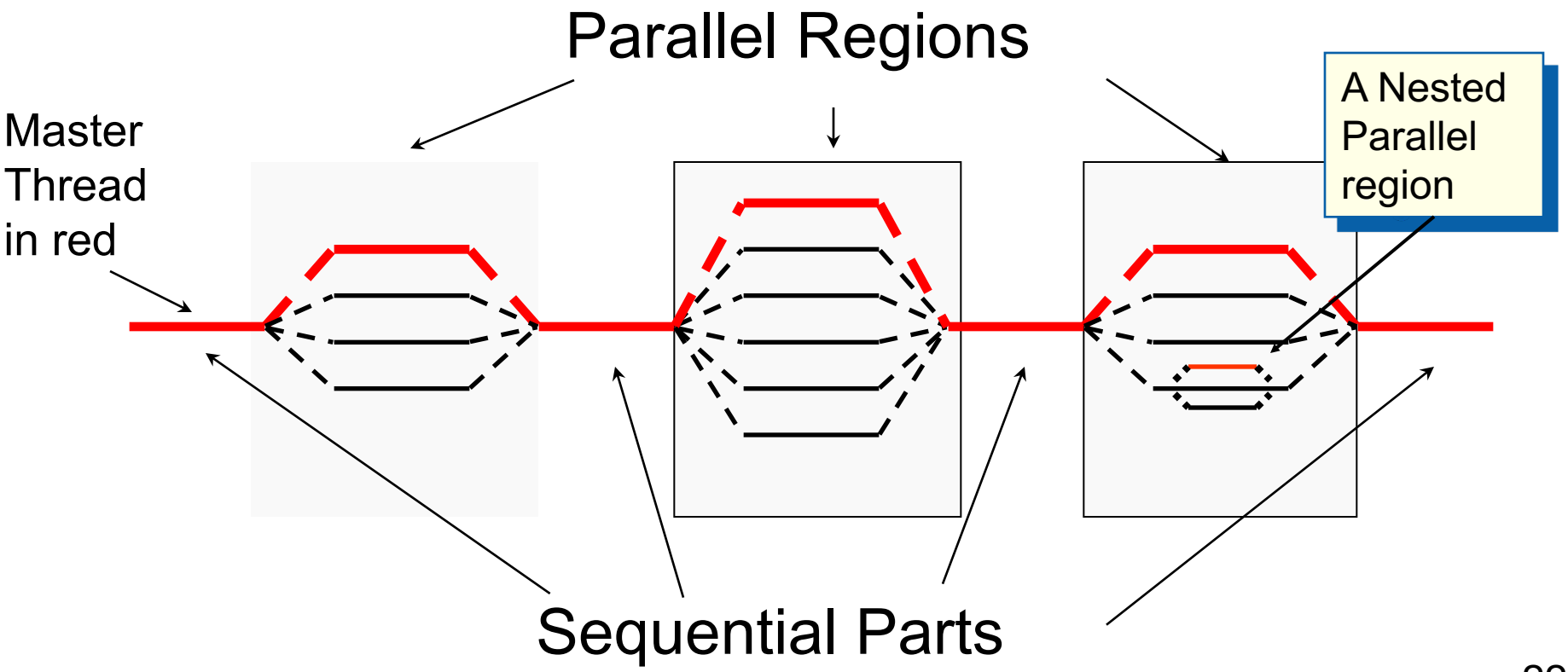


- Introduction to OpenMP
- Creating Threads
- Quantifying Performance and Amdahl's law
- Synchronization
- Parallel Loops
- Data environment
- Memory model
- Irregular Parallelism and tasks
- Recap and the fundamental design patterns of parallel programming

OpenMP programming model:

Fork-Join Parallelism:

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



Thread creation: Parallel regions

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

```
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

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

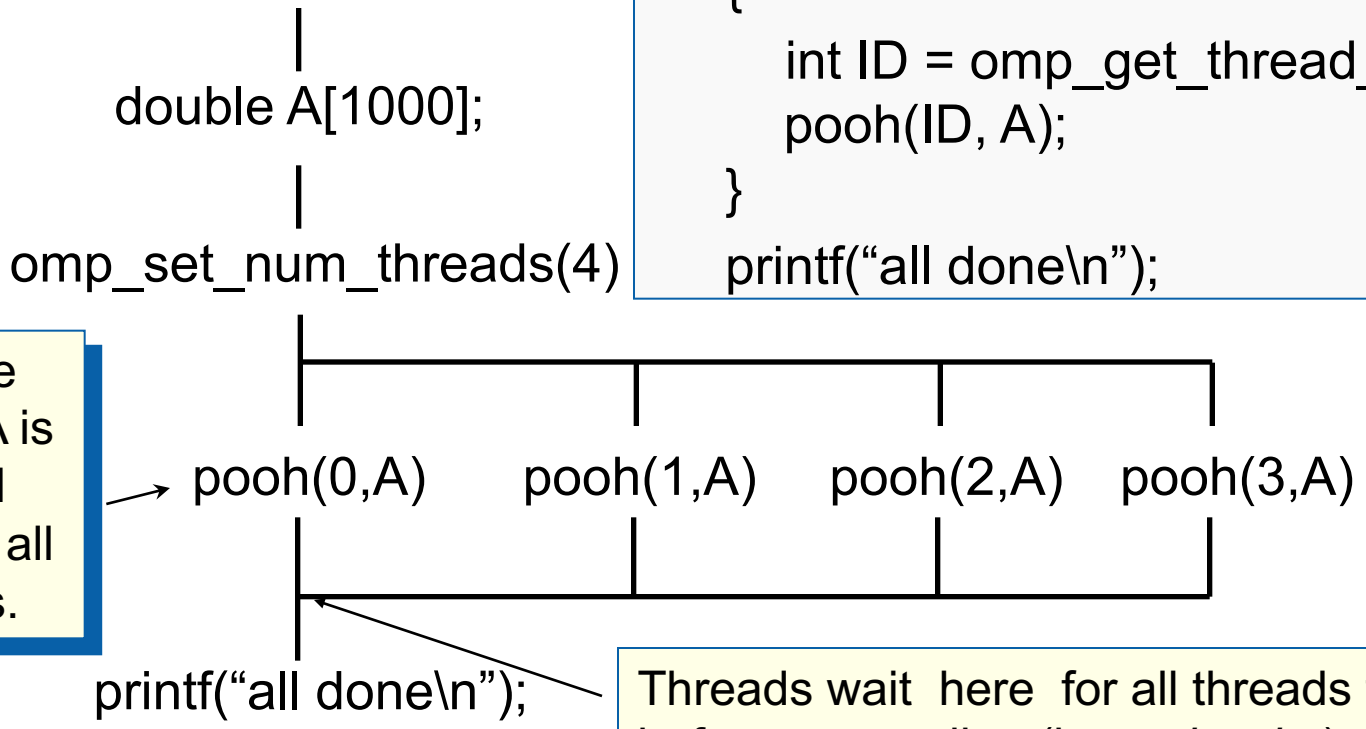
Runtime function returning a thread ID

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

Thread creation: Parallel regions example

- Each thread executes the same code redundantly.

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



A single copy of A is shared between all threads.

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

Thread creation: How many threads did you actually get?

- You create a team threads in OpenMP* with the parallel construct.
- You can request a number of threads with omp_set_num_threads()
- But is the number of threads requested the number you actually get?
  - NO! An implementation can silently decide to give you a team with fewer threads.
  - Once a team of threads is established … the system will not reduce the size of the team.

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);
}
```

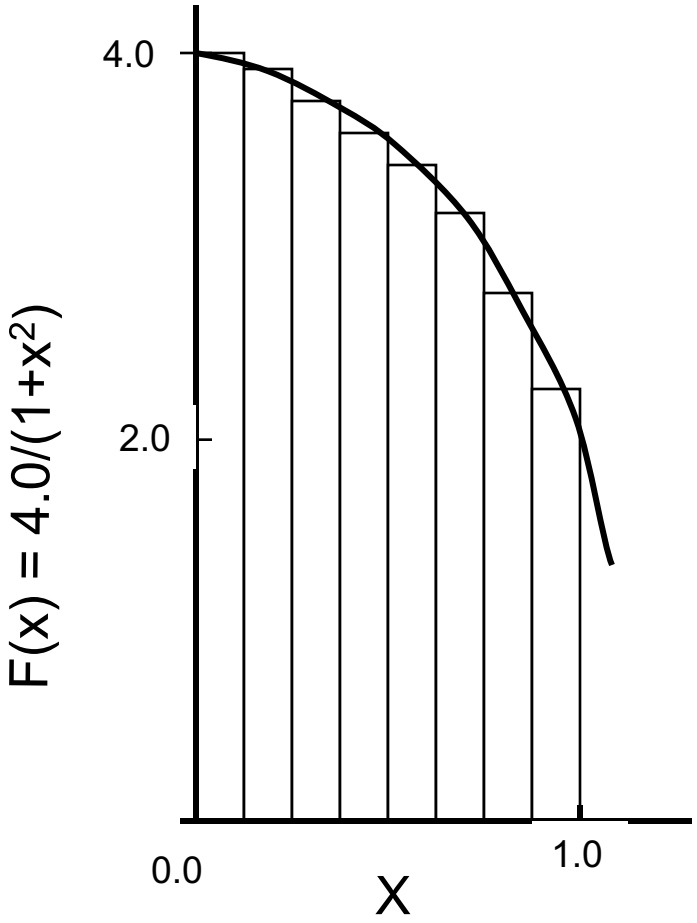
Runtime function to request a certain number of threads

Runtime function to return actual number of threads in the team

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

Our recurring problem in the Exercises:

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 rectangles:

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

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 ()
{           int i;      double x, pi, sum = 0.0;

            step = 1.0/(double) num_steps;

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

See OMP_exercises/pi.c

Serial PI program

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

           step = 1.0/(double) num_steps;
           double tdata = omp_get_wtime();
           for (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 OMP_exercises/pi.c

Exercise 2

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

  #pragma omp parallel.

- Pay close attention to shared versus private variables.
- In addition to a parallel construct, you will need the runtime library routines
  - int omp_get_num_threads();
  - int omp_get_thread_num();
  - double omp_get_wtime();
  - omp_set_num_threads();

Number of threads in the team

Thread ID or rank

Time in Seconds since a fixed point in the past

Request a number of threads in the team

Exercise 2 (hints)

- 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();

Results*

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

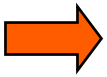
Example: A simple Parallel pi program

```
#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,nthrds;
          double x;
          id = omp_get_thread_num();
          nthrds = omp_get_num_threads();
          if (id == 0) nthreads = nthrds;
           for (i=id, sum[id]=0.0;i< num_steps; i=i+nthrds) {
                    x = (i+0.5)*step;
                    sum[id] += 4.0/(1.0+x*x);
           }
   }
           for(i=0, pi=0.0;i<nthreads;i++)pi += sum[i] * step;
}
```

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

*Intel compiler (icpc) with 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.

Outline



- Introduction to OpenMP
- Creating Threads
- Quantifying Performance and Amdahl's law
- Synchronization
- Parallel Loops
- Data environment
- Memory model
- Irregular Parallelism and tasks
- Recap and the fundamental design patterns of parallel programming

Consider performance of parallel programs

Compute N independent tasks on one processor

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

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

Compute N independent tasks with P processors

Load Data

Compute T_1

…

Compute T_N

Consume Results

Ideally Cut runtime by ~1/P

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

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

Talking about performance

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

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

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

$$S(P) = P$$

- 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

$$S(P) > P$$

So now you should understand my silly introduction slide.

Introduction

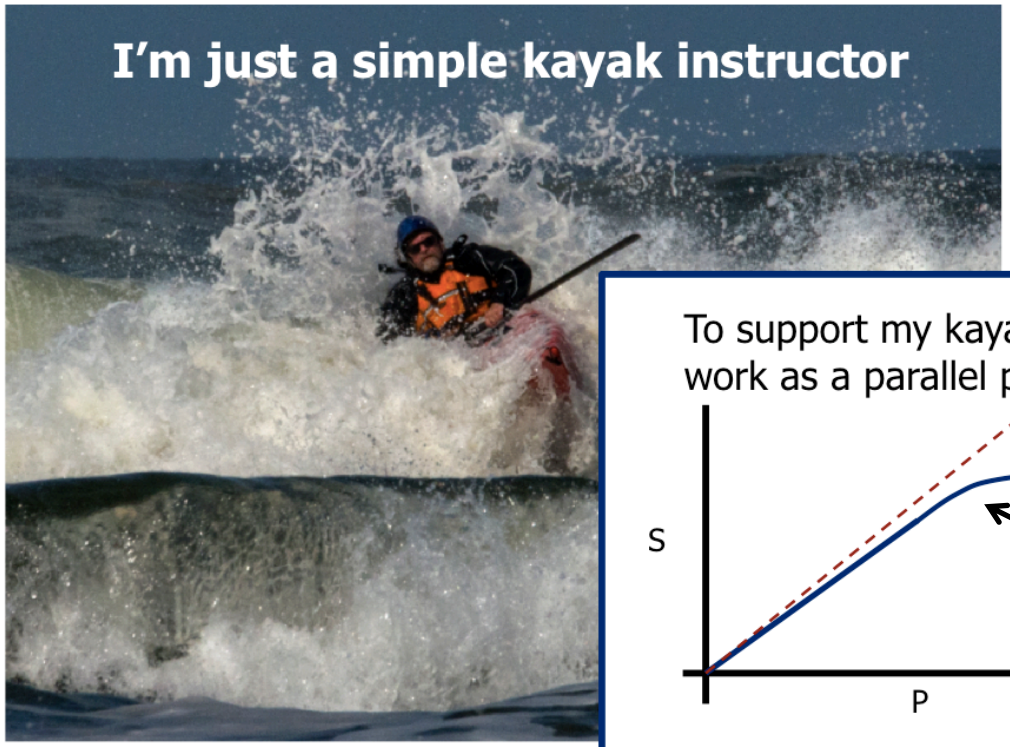


Photo © by Greg Clopton, 2014

To support my kayaking habit I work as a parallel programmer

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

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

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

4

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 serial_fraction is α and parallel_fraction is (1- α) then the speedup is:

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

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

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

Internal control variables and the number of threads

- There are a few ways to control the number of threads.
- We've used the following construct (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 3

- Go back to your parallel pi program and explore how well it scales with (1) the number of threads and (2) the number of steps in the integration.
- 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 request N threads

Results*

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

Example: A simple Parallel pi program

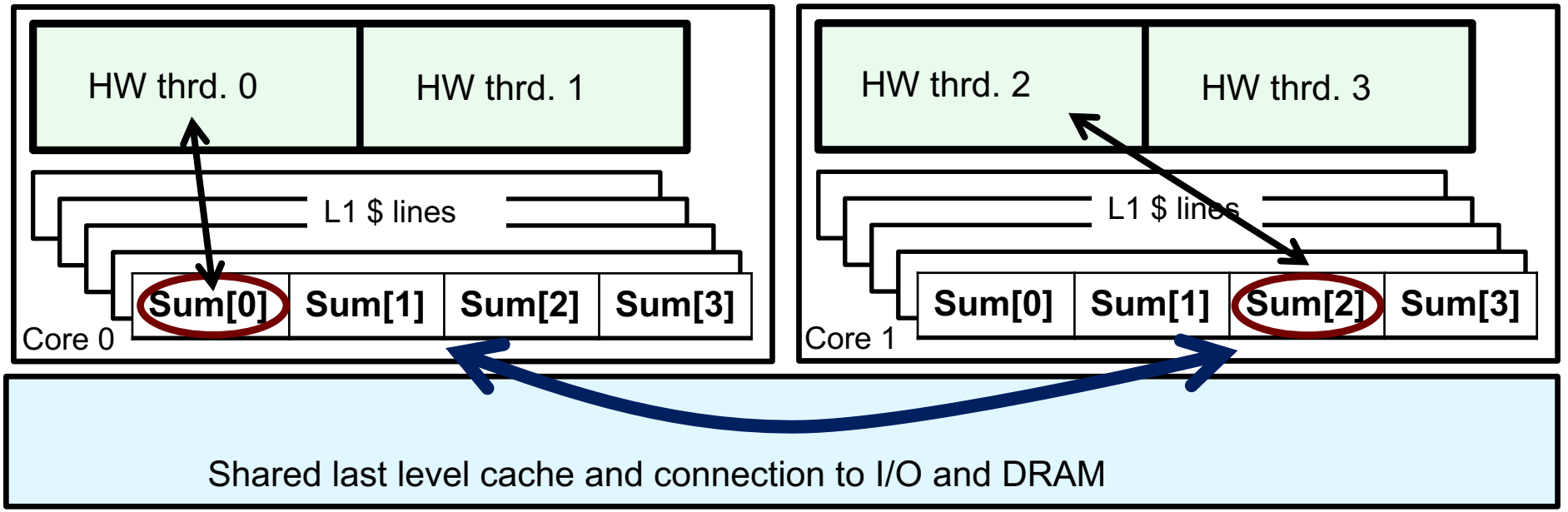
```
#include <omp.h>
static long num_steps = 100000;        double step;
#define NUM_THREADS 2
void main ()
{          int i, nthreads;  double pi, sum[NUM_THREADS];
           step = 1.0/(double) num_steps;
           omp_set_num_threads(NUM_THREADS);
   #pragma omp parallel
   {
          int i, id,nthrds;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

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

*Intel compiler (icpc) with 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.

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    PAD      8          // assume 64 byte L1 cache line size
#define NUM_THREADS 2
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.

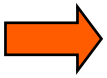
Example: eliminate False sharing by padding the sum array

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

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

*Intel compiler (icpc) with 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.

Outline



- Introduction to OpenMP
- Creating Threads
- Quantifying Performance and Amdahl's law
- Synchronization
- Parallel Loops
- Data environment
- Memory model
- Irregular Parallelism and tasks
- Recap and the fundamental design patterns of parallel programming

Synchronization

- High level synchronization:
  - critical
  - barrier

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

Synchronization: critical

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

Threads wait their turn – only one 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();
      for(i=id;i<niters;i+=nthrds){
           B =  big_job(i);
#pragma omp critical
           res += consume (B);
     }
}
```

Synchronization: barrier

- Barrier: a point in a program all threads much reach before any threads are allowed to proceed.

```
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, A);
}
```

Onr Thread saves a copy of nthrds into a global variable, numthrds

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

Exercise 5

- In exercise 2, 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” from exercise 2 to avoid false sharing due to the sum array.

Pi program with false sharing*

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

Example: A simple Parallel 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,nthrds;
          double x;
          id = omp_get_thread_num();
          nthrds = omp_get_num_threads();
          if (id == 0) nthreads = nthrds;
           for (i=id, sum[id]=0.0;i< num_steps; i=i+nthrds) {
                    x = (i+0.5)*step;
                    sum[id] += 4.0/(1.0+x*x);
           }
   }
           for(i=0, pi=0.0;i<nthreads;i++)pi += sum[i] * step;
}
```

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

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

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

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

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

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

No array, so no false sharing.

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

Results*: pi program critical section

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

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

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

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;
            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+nthreads){
                        x = (i+0.5)*step;
                       #pragma omp critical
                              pi += 4.0/(1.0+x*x);
              }
}
pi *= step;
}
```

Be careful where you put a critical section

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

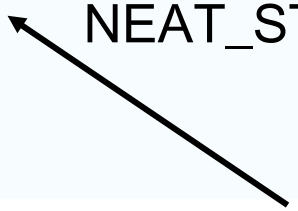
Outline

- Introduction to OpenMP
- Creating Threads
- Quantifying Performance and Amdahl's law
- Synchronization
- Parallel Loops
- Data environment
- Memory model
- Irregular Parallelism and tasks
- Recap and the fundamental design patterns of parallel programming

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
A motivating example

Sequential code

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

OpenMP parallel region

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

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.

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

Least work at runtime : scheduling done at compile-time

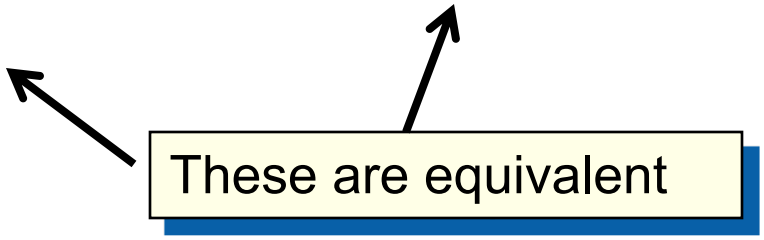
Most work at runtime : complex scheduling logic used at run-time

Combined parallel/worksharing construct

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

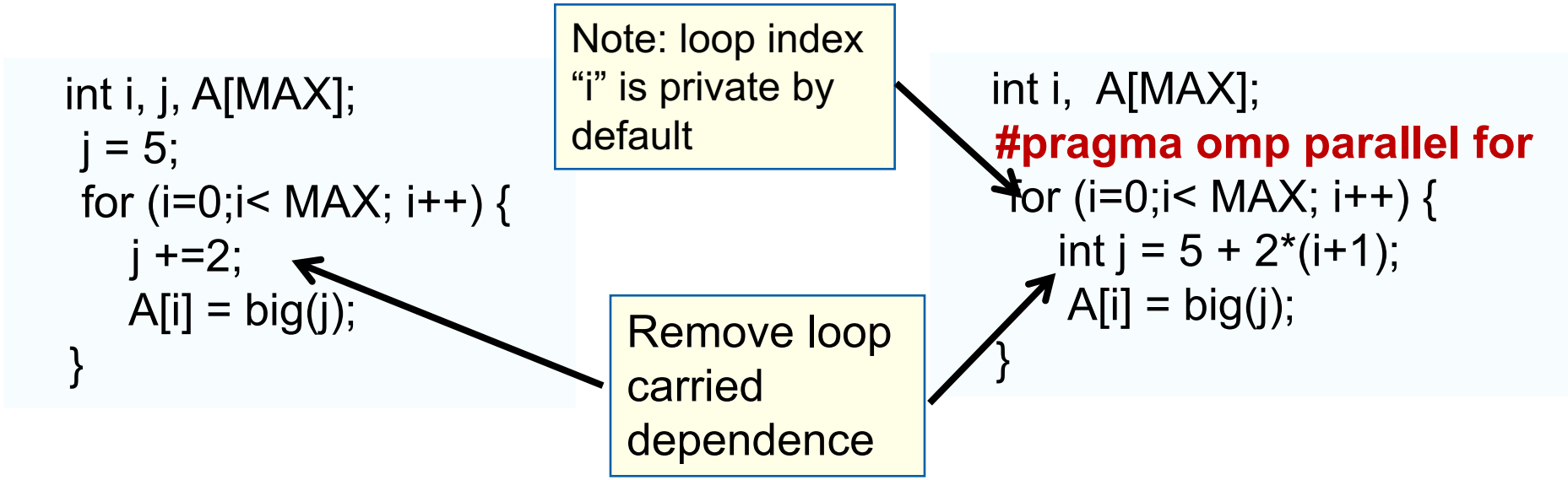
```
 double  res[MAX];  int i;
#pragma omp parallel
{
    #pragma omp for
    for (i=0;i< MAX; i++) {
        res[i] = huge();
    }
}
```

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



Working with loops

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



Reduction

- How do we handle this case?

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

- We are combining values into a single accumulation variable (ave) … there is a true dependence between loop iterations that can't be trivially removed
- This is a very common situation … it is called a "reduction".
- Support for reduction operations is included in most parallel programming environments.

Reduction

- OpenMP reduction clause:

  reduction (op : list)

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

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

OpenMP: Reduction operands/initial-values

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

Operator	Initial value
+	0
*	1
-	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.

Exercise 6: 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.

Example: Pi with a loop and a reduction

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

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

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

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

Results*: pi with a loop and a reduction

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

Example: Pi with a

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

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

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

Barriers, for loops and the nowait clause

- Barrier: Each thread waits until all threads arrive.

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

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

implicit barrier at the end of a for worksharing construct

implicit barrier at the end of a parallel region

no implicit barrier due to nowait

Outline

- Introduction to OpenMP
- Creating Threads
- Quantifying Performance and Amdahl's law
- Synchronization
- Parallel Loops
- Data environment
- Memory model
- Irregular Parallelism and tasks
- Recap and the fundamental design patterns of parallel programming

Data environment: Default storage attributes

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

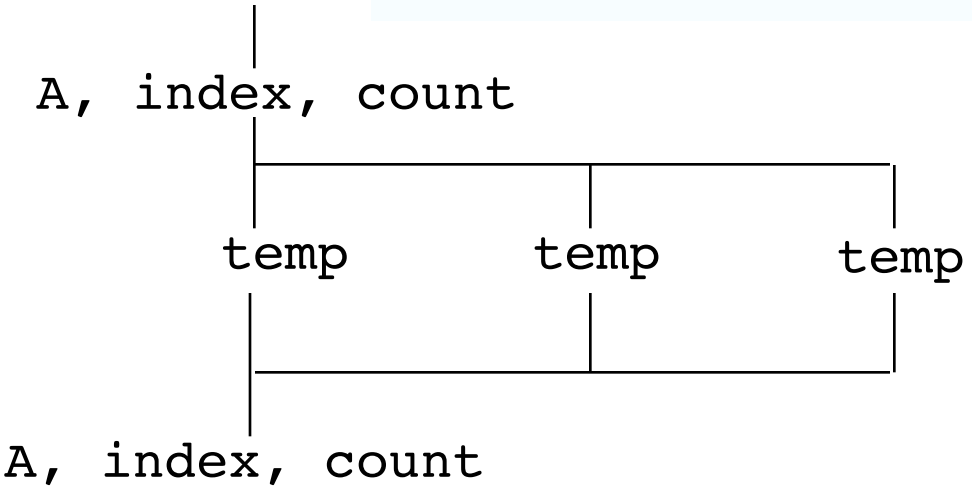
Data sharing: Examples

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

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

A, index and count are shared by all threads.

temp is local to each thread



Data sharing: Changing storage attributes

- One can selectively change storage attributes for constructs using the following clauses* (note: list is a comma-separated list of variables)
  - shared(list)
  - private(list)
  - firstprivate(list)

All the clauses on this page apply to the OpenMP construct NOT to the entire region.

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

***All data clauses apply to parallel constructs and worksharing constructs except "shared", which only applies to parallel constructs**

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);
}
```

tmp was not initialized

tmp is 0 here

Data sharing: Private clause When is the original variable valid?

- 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

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)**: *no* default for variables in static extent. Must list storage attribute for each variable in static extent. Good programming practice!

Exercise 7: Mandelbrot set area

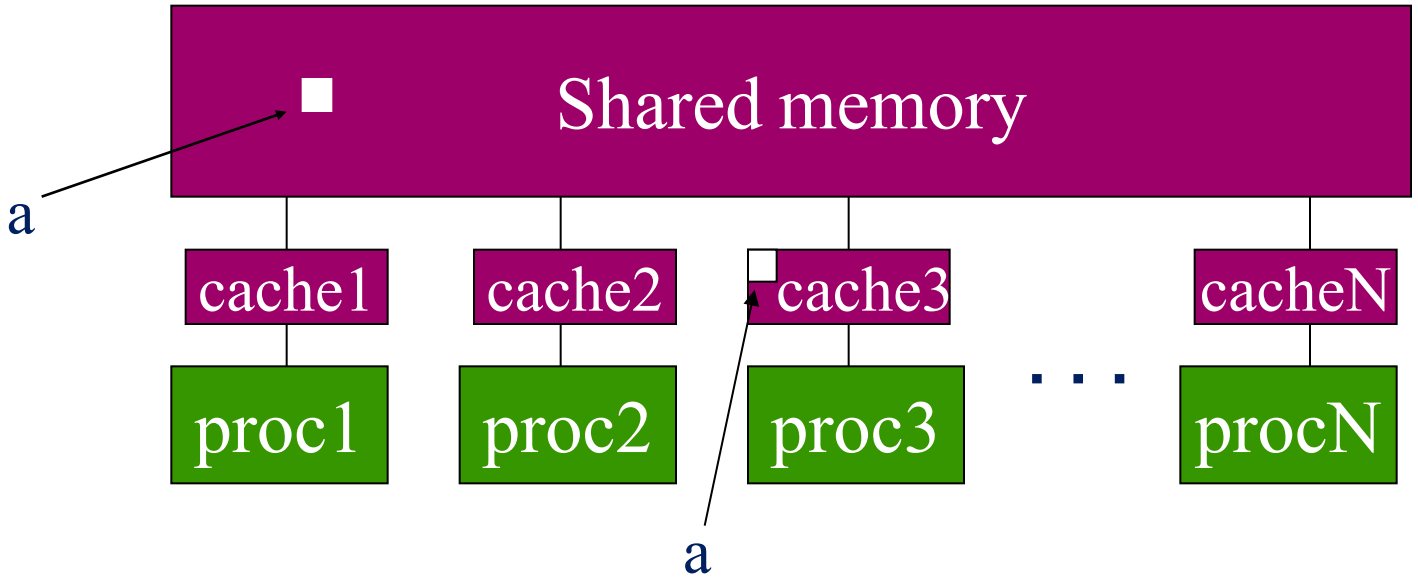
- The supplied program (mandel.c) computes the area of a Mandelbrot set.
- The program has been parallelized with OpenMP, but we were lazy and didn't do it right.
- Find and fix the errors (hint … the problem is with the data environment).
- Once you have a working version, try to optimize the program.
  - Try different schedules on the parallel loop.
  - Try different mechanisms to support mutual exclusion … do the efficiencies change?

Outline

- Introduction to OpenMP
- Creating Threads
- Quantifying Performance and Amdahl's law
- Synchronization
- Parallel Loops
- Data environment
- Memory model
- Irregular Parallelism and tasks
- Recap and the fundamental design patterns of parallel programming

OpenMP memory model

- OpenMP supports a shared memory model
- All threads share an address space, but it can get complicated:



- Multiple copies of data may be present in various levels of cache, or in registers

OpenMP and relaxed consistency

- OpenMP supports a **relaxed-consistency** shared 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**

Flush operation

- Defines a sequence point at which a thread is guaranteed to see a consistent view of memory
  - All 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

Synchronization: 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

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.
- Compiler 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.

Note: the flush operation 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 at entry to worksharing regions)

WARNING:
If you find your self 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.

Example: prod_cons.c

- Parallelize a producer/consumer program
  - One thread produces values that another thread consumes.

```
int main()
{
  double *A, sum, runtime;     int flag = 0;

  A = (double *) malloc(N*sizeof(double));

  runtime = omp_get_wtime();

  fill_rand(N, A);        // Producer: fill an array of data

  sum = Sum_array(N, A);  // Consumer: sum the array

  runtime = omp_get_wtime() - runtime;

  printf(" In %lf secs, The sum is %lf \n",runtime,sum);
}
```

- Often used with a stream of produced values to implement "pipeline parallelism"
- The key is to implement pairwise synchronization between threads

Pairwise synchronizaion in OpenMP

- OpenMP lacks synchronization constructs that work between pairs of threads.
- When needed, you have to build it yourself.
- Pairwise synchronization
  - Use a shared flag variable
  - Reader spins waiting for the new flag value
  - Use flushes to force updates to and from memory

Exercise: Producer/consumer

```
int main()
{
   double *A, sum, runtime;     int numthreads, flag = 0;
   A = (double *)malloc(N*sizeof(double));
   #pragma omp parallel sections
   {
     #pragma omp section
      {
        fill_rand(N, A);

        flag = 1;

      }
      #pragma omp section
      {

        while (flag == 0){

        }

        sum = Sum_array(N, A);
      }
    }
}
```

Put the flushes in the right places to make this program race-free.

Do you need any other synchronization constructs to make this work?

Solution (try 1): Producer/consumer

```
int main()
{
   double *A, sum, runtime;     int numthreads, flag = 0;
   A = (double *)malloc(N*sizeof(double));
   #pragma omp parallel sections
   {
     #pragma omp section
      {
        fill_rand(N, A);
        #pragma omp flush
        flag = 1;
        #pragma omp flush (flag)
      }
      #pragma omp section
      {
        #pragma omp flush (flag)
        while (flag == 0){
            #pragma omp flush (flag)
        }
        #pragma omp flush
        sum = Sum_array(N, A);
      }
    }
}
```

Use flag to Signal when the "produced" value is ready

Flush forces refresh to memory; guarantees that the other thread sees the new value of A

Flush needed on both "reader" and "writer" sides of the communication

Notice you must put the flush inside the while loop to make sure the updated flag variable is seen

This program works with the x86 memory model (loads and stores use relaxed atomics), but it technically has a race … on the store and later load of flag

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

Additional forms of atomic were added in 3.1 (discussed later)

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;**

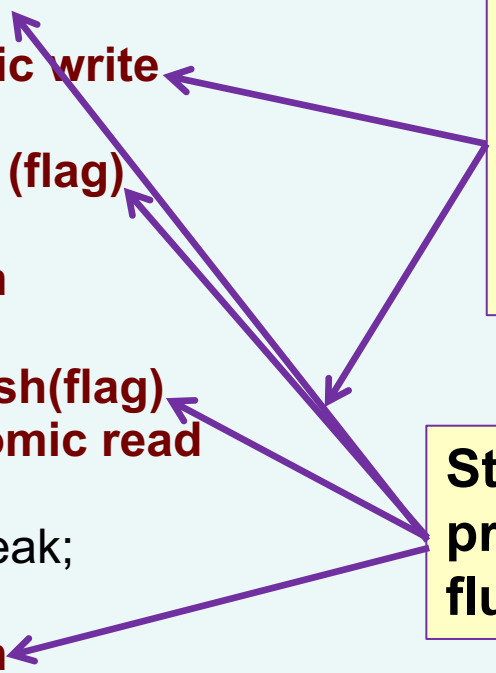
- Where the structured block is one of the following forms:

{v = x; x binop = expr;}	**{x binop = expr; v = x;}**
{v=x; x=x binop expr;}	**{X = x binop expr; v = x;}**
{v = x; x++;}	**{v=x; ++x:}**
{++x; v=x:}	**{x++; v = x;}**
{v = x; x--;}	**{v= x; --x;}**
{--x; v = 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

Atomics and synchronization flags

```
int main()
{   double *A, sum, runtime;
    int numthreads, flag = 0, flg_tmp;
    A = (double *)malloc(N*sizeof(double));
    #pragma omp parallel sections
    {
      #pragma omp section
       {  fill_rand(N, A);
          #pragma omp flush
          #pragma omp atomic write
                  flag = 1;
          #pragma omp flush (flag)
       }
       #pragma omp section
       {  while (1){
             #pragma omp flush(flag)
             #pragma omp atomic read
                    flg_tmp= flag;
              if (flg_tmp==1) break;
          }
          #pragma omp flush
          sum = Sum_array(N, A);
       }
     }
}
```



This program is truly race free … the reads and writes of flag are protected so the two threads cannot conflict

Still painful and error prone due to all of the flushes that are required

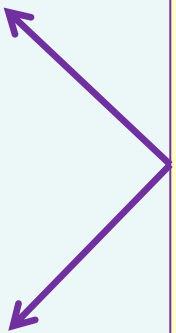
OpenMP 4.0 Atomic: Sequential consistency

- Sequential consistency:
  - The order of loads and stores in a race-free program appear in some interleaved order and all threads in the team see this same order.
- OpenMP 4.0 added an optional clause to atomics
  - #pragma omp atomic [read | write | update | capture] [**seq_cst**]
- In more pragmatic terms:
  - If the seq_cst clause is included, OpenMP adds a flush without an argument list to the atomic operation so you don't need to.
- In terms of the C++'11 memory model:
  - Use of the seq_cst clause makes atomics follow the sequentially consistent memory order.
  - Leaving off the seq_cst clause makes the atomics relaxed.

Advice to programmers: save yourself a world of hurt … let OpenMP take care of your flushes for you whenever possible … use seq_cst

Atomics and synchronization flags (4.0)

```
int main()
{   double *A, sum, runtime;
    int numthreads, flag = 0, flg_tmp;
    A = (double *)malloc(N*sizeof(double));
    #pragma omp parallel sections
    {
      #pragma omp section
       {  fill_rand(N, A);

          #pragma omp atomic write seq_cst
                  flag = 1;

       }
       #pragma omp section
       {  while (1){

             #pragma omp atomic read seq_cst
                   flg_tmp= flag;
              if (flg_tmp==1) break;
           }

           sum = Sum_array(N, A);
       }
     }
}
```



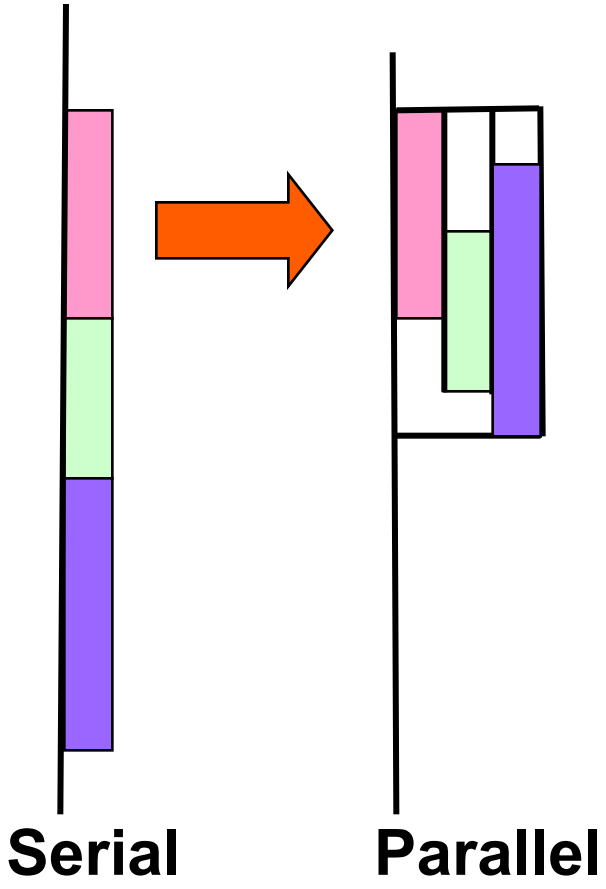
This program is truly race free … the reads and writes of flag are protected so the two threads cannot conflict – and you do not use any explicit flush constructs (OpenMP does them for you)

Outline

- Introduction to OpenMP
- Creating Threads
- Quantifying Performance and Amdahl's law
- Synchronization
- Parallel Loops
- Data environment
- Memory model
- Irregular Parallelism and tasks
- Recap and the fundamental design patterns of parallel programming

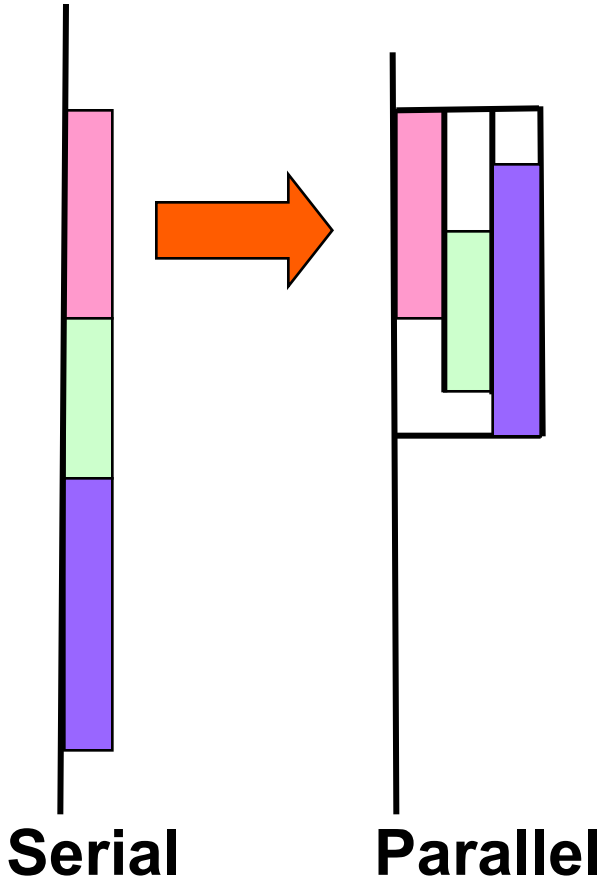
What are tasks?

- 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 master 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();

}
```

Task Directive

```
#pragma omp task [clauses]
structured-block
```

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

Create some threads

One Thread packages tasks

Tasks executed by some thread in some order

All tasks complete before this barrier is released

Exercise 8: Simple tasks

- Write a program using tasks that will “randomly” generate one of two strings:
  - I think race cars are fun
  - I think car races 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
```

When/where are tasks complete?

- At thread barriers (explicit or implicit)
  - applies to all tasks generated in the current parallel region up to the barrier

- 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 taskwait
     #pragma omp task
        billy();
   }
}
```

Linked list traversal

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

- Classic linked list traversal
- Do some work on each item in the list
- Assume that items can be processed independently
- Cannot use an OpenMP loop directive

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

Parallel linked list traversal

Thread 0:

```
p = listhead ;
while (p) {
< package up task >
   p=next (p) ;
}

while (tasks_to_do){
  < execute task >
}

< barrier >
```

Other threads:

```
while (tasks_to_do) {
< execute task >
}

< barrier >
```

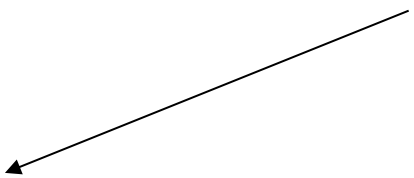
Data scoping with tasks

- 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

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 $O(n^2)$ 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!

Using tasks

- Getting the data attribute scoping right can be quite tricky
  - default scoping rules different from other constructs
  - as usual, using `default(none)` is a good idea

- 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

Exercise 9: Pi with tasks

- Consider the program Pi_recur.c. This program implements a recursive algorithm version of the program for computing pi
  - 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();
```

Outline

- Introduction to OpenMP
- Creating Threads
- Quantifying Performance and Amdahl's law
- Synchronization
- Parallel Loops
- Data environment
- Memory model
- Irregular Parallelism and tasks
- Recap and the fundamental design patterns of parallel programming

The OpenMP Common Core: Most OpenMP programs only use these 16 constructs

OMP Construct	Concepts
#pragma omp parallel	parallel region, teams structured block, interleaved execution across threads
int omp_get_thread_num() int omp_get_num_threads()	The SPMD Pattern.
double omp_get_wtime()	Speedup and Amdahl's law. False Sharing and other performance issues
setenv OMP_NUM_THREADS N	internal control variables
#pragma omp barrier #pragma omp critical	Synchronization and race conditions. Revisit interleaved execution. Teach the concept of flush but not the construct.
#pragma omp for	worksharing, parallel loops, loop carried dependencies
reduction(op:list)	reductions of values across a team of threads
schedule(dynamic [,chunk]) schedule (static [,chunk])	Loop schedules, loop overheads and load balance
private(list), firstprivate(list), shared(list)	Data environment
nowait	synchronization overhead, the high cost of barriers
#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, flush and several forms of atomic
- Data management
  - 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
  - target, teams distribute parallel for, target data …
- … and much more. The OpenMP 4.5 specification is over 350 pages!!!

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

The fundamental design patterns of parallel programming

- People learn best by mapping new information onto an existing conceptual framework.
- We have created a conceptual framework for parallel programming and defined it in terms of parallel design patterns:
  - https://patterns.eecs.berkeley.edu/
- If you know the patterns and how to see them in your parallel algorithms, it is much easier to learn new programing models.
- A few parallel programming design patterns are so commonly used, knowledge of the is almost unviersal among parallel programmers.

Fork-join

- Use when:
  - Target platform has a shared address space
  - Dynamic task parallelism
- Particularly useful when you have a serial program to transform incrementally into a parallel program
- Solution:
  1. A computation begins and ends as a single thread.
  2. When concurrent tasks are desired, additional threads are forked.
  3. The thread carries out the indicated task,
  4. The set of threads recombine (join)

Pthreads, OpenMP are based on this pattern.

SPMD: Single Program Mulitple 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.

This pattern is very general and has been used to support most (if not all) the algorithm strategy patterns.

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

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,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;
}
```

Loop-level parallelism

- Collections of tasks are defined as iterations of one or more loops.
- Loop iterations are divided between a collection of processing elements to compute tasks concurrently. Key elements:
  - identify compute intensive loops
  - Expose concurrency by removing/managing loop carried dependencies
  - Exploit concurrency for parallel execution usually using a parallel loop construct/directive.

```
#pragma parallel for shared(Results) schedule(dynamic)
For(i=0;i<N;i++){
        Do_work(i, Results);
}
```

This design pattern is also heavily used with data parallel design patterns. OpenMP programmers commonly use this pattern.

Loop Level parallelism pi program

```
#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 for private(x) 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;
}
```

For good OpenMP implementations, reduction is more scalable than critical.

i private by default

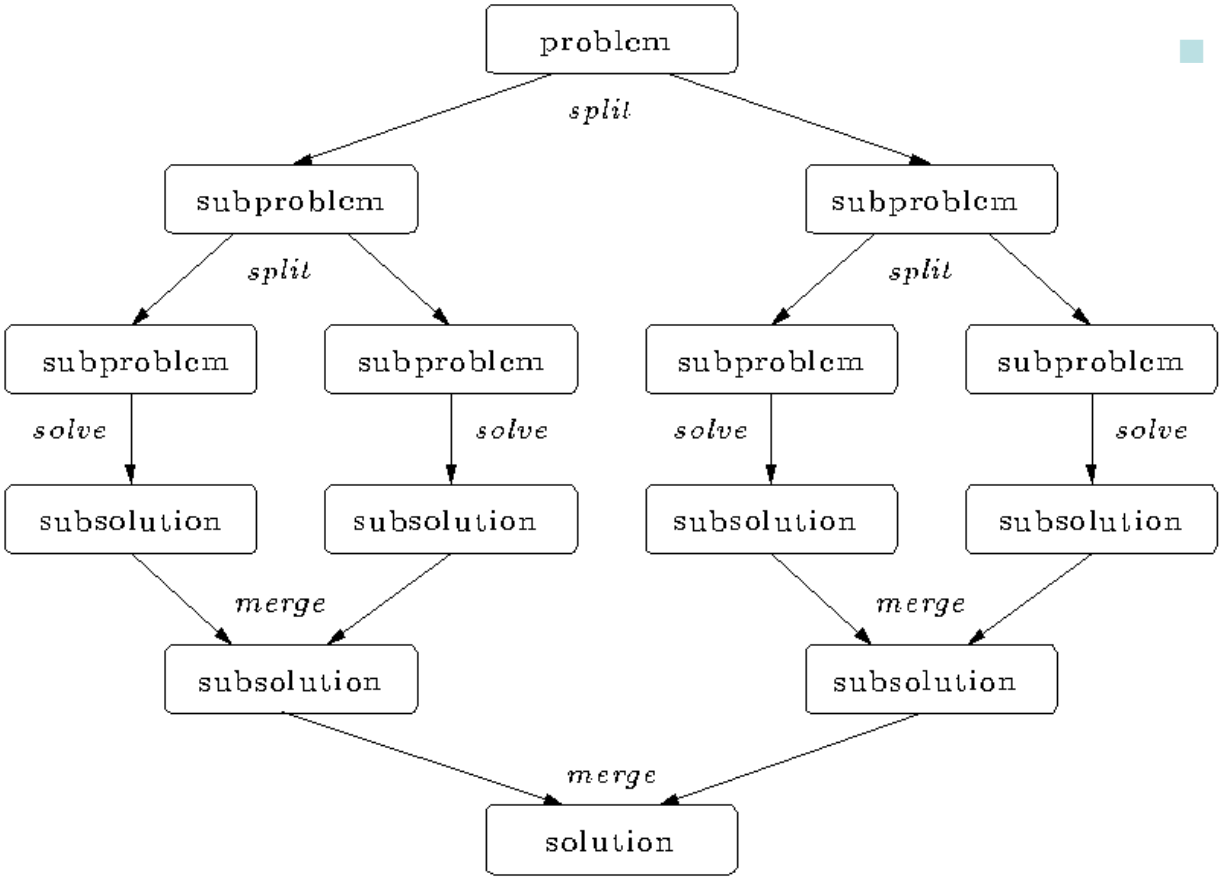
Note: we created a parallel program without changing any code and by adding 2 simple lines of text!

Divide and Conquer Pattern

- Use when:
  - A problem includes a method to divide the problem into subproblems and a way to recombine solutions of subproblems into a global solution.
- Solution
  - Define a split operation
  - Continue to split the problem until subproblems are small enough to solve directly.
  - Recombine solutions to subproblems to solve original global problem.
- Note:
  - Computing may occur at each phase (split, leaves, recombine).

Divide and conquer

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



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

Program: OpenMP tasks (divide and conquer pattern)

```
#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;
}
```

Geometric Decomposition

We'll cover this when we discuss MPI

- Use when:
  - The problem is organized around a central data structure that can be decomposed into smaller segments (chunks) that can be updated concurrently.
- Solution
  - Typically, the data structure is updated iteratively where a new value for one chunk depends on neighboring chunks.
  - The computation breaks down into three components: (1) exchange boundary data, (2) update the interiors or each chunk, and (3) update boundary regions. The optimal size of the chunks is dictated by the properties of the memory hierarchy.
- Note:
  - This pattern is often used with the Structured Mesh and linear algebra computational patterns.

SIMT (Data-parallel/index-space)

We'll cover this when we discuss GPUs

- Single instruction, multiple data:
  - Implement data parallel problems:
    - Define an abstract index space that appropriately spans the problem domain.
    - Data structures in the problem are aligned to this index space.
    - Tasks (e.g. work-items in OpenCL or "threads" in CUDA) operate on these data structures for each point in the index space.
- This approach was popularized for graphics applications where the index space mapped onto the pixels in an image. In the last ~5 years, It's been extended to General Purpose GPU (GPGPU) programming.

Note: This is basically a fine grained extreme form of the SPMD pattern.

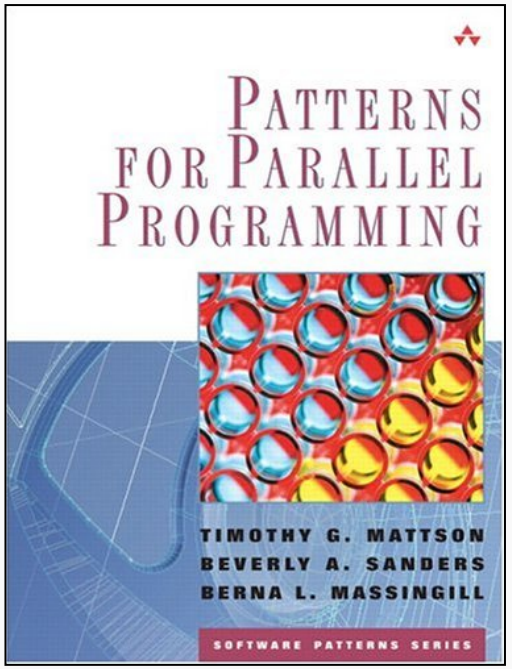
“Patterns make you smarter”

Prof. Kurt Keutzer of UC Berkeley

- Keep the design patterns relevant to you in mind as you consider different programing models.
- It helps organize your understanding of the models and make informed judgements as you choose the model to use.

I guess I should put in a plug for my book.

It’s a bit old (2004 … predates GPGPU) but for clusters and multithreading it’s still relevant (and selling well).



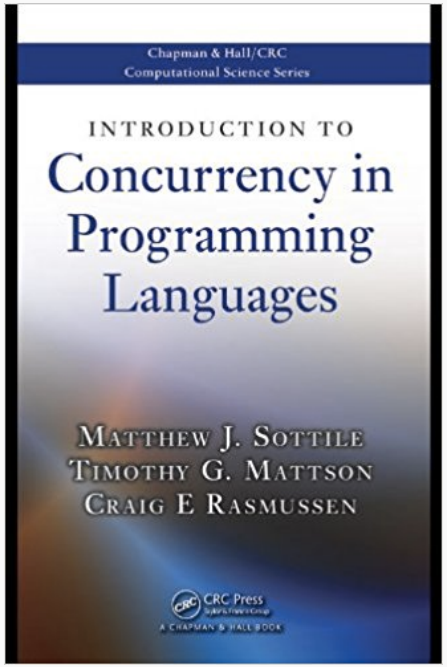
“Patterns make you smarter”

Prof. Kurt Keutzer of UC Berkeley

- Keep the design patterns relevant to you in mind as you consider different programing models.
- It helps organize your understanding of the models and make informed judgements as you choose the model to use.

… well, and maybe my other book

Much newer (2010) but still needs updating on GPUs



Backup materials

- Challenge problems
- Appendices

Challenge problems

- Long term retention of acquired skills is best supported by “random practice”.
  - i.e., a set of exercises where you must draw on multiple facets of the skills you are learning.
- To support “Random Practice” we have assembled a set of “challenge problems”
  1. Parallel molecular dynamics
  2. Monte Carlo “pi” program and parallel random number generators
  3. Optimizing matrix multiplication
  4. Traversing linked lists in different ways
  5. Recursive matrix multiplication algorithms

Challenge 1: Molecular dynamics

- The code supplied is a simple molecular dynamics simulation of the melting of solid argon
- Computation is dominated by the calculation of force pairs in subroutine `forces` (in forces.c)
- Parallelise this routine using a parallel for construct and atomics; think carefully about which variables should be SHARED, PRIVATE or REDUCTION variables
- Experiment with different schedule kinds

Challenge 1: MD (cont.)

- Once you have a working version, move the parallel region out to encompass the iteration loop in main.c
  - Code other than the forces loop must be executed by a single thread (or workshared).
  - How does the data sharing change?
- The atomics are a bottleneck on most systems.
  - This can be avoided by introducing a temporary array for the force accumulation, with an extra dimension indexed by thread number
  - Which thread(s) should do the final accumulation into f?

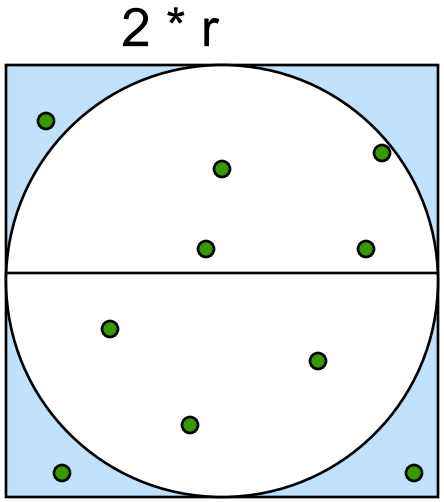
Challenge 1 MD: (cont.)

- Another option is to use locks
  - Declare an array of locks
  - Associate each lock with some subset of the particles
  - Any thread that updates the force on a particle must hold the corresponding lock
  - Try to avoid unnecessary acquires/releases
  - What is the best number of particles per lock?

Challenge 2: 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	π = 2.8
N=100	π = 3.16
N= 1000	π = 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

Challenge 2: 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 without changing 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.
- Extra Credit:
  - Make your random number generator numerically correct (non-overlapping sequences of pseudo-random numbers).

Challenge 3: Matrix multiplication

- Parallelize the matrix multiplication program in the file matmul.c
- Can you optimize the program by playing with how the loops are scheduled?
- Try the following and see how they interact with the constructs in OpenMP
  - Cache blocking
  - Loop unrolling
  - Vectorization
- Goal: Can you approach the peak performance of the computer?

Challenge 4: Traversing linked lists

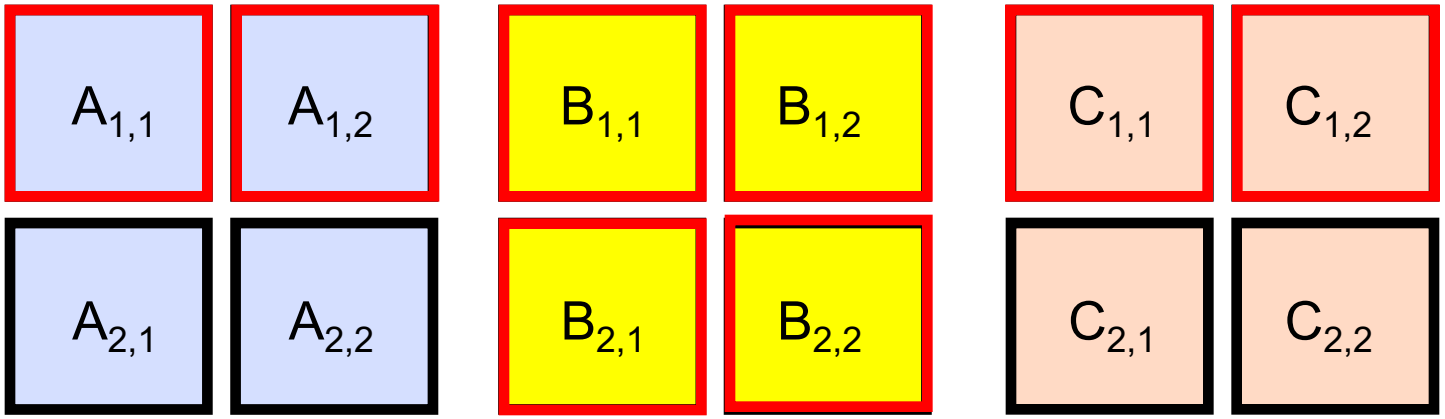
- Consider the program linked.c
  - Traverses a linked list, computing a sequence of Fibonacci numbers at each node
- Parallelize this program two different ways
  1. Use OpenMP tasks
  2. Use anything you choose in OpenMP *other than* tasks.
- The second approach (no tasks) can be difficult and may take considerable creativity in how you approach the problem (why its such a pedagogically valuable problem)

Challenge 5: Recursive matrix multiplication

- The following three slides explain how to use a recursive algorithm to multiply a pair of matrices
- Source code implementing this algorithm is provided in the file matmul_recur.c
- Parallelize this program using OpenMP tasks

Challenge 5: Recursive matrix multiplication

- Quarter each input matrix and output matrix
- Treat each submatrix as a single element and multiply
- 8 submatrix multiplications, 4 additions



$C_{1,1} = A_{1,1} \cdot B_{1,1} + A_{1,2} \cdot B_{2,1}$

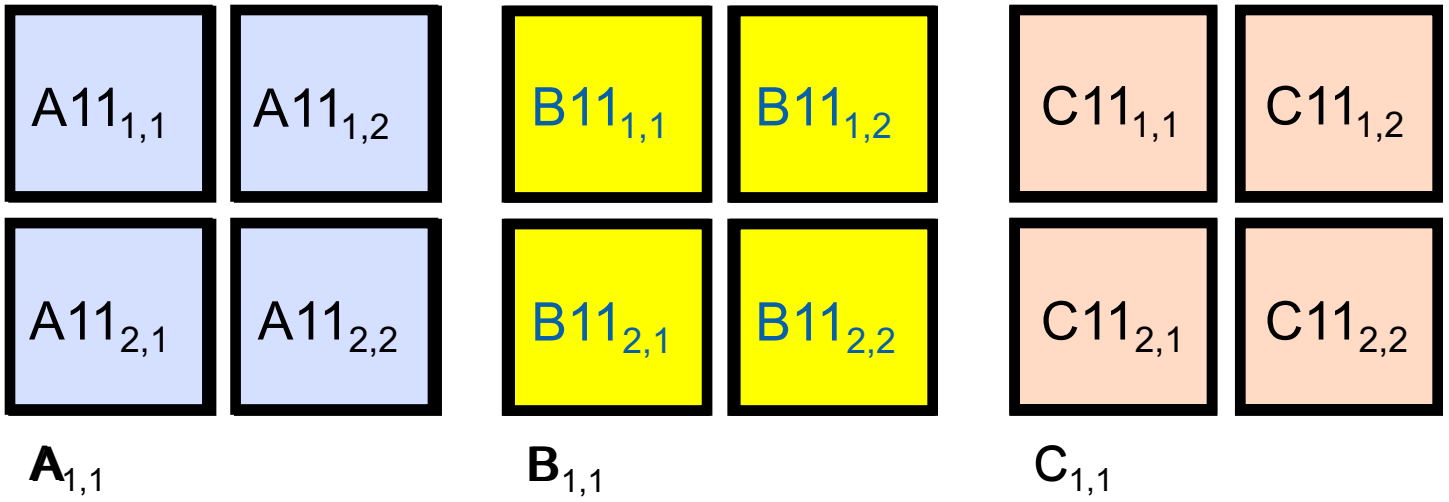
$C_{1,2} = A_{1,1} \cdot B_{1,2} + A_{1,2} \cdot B_{2,2}$

$C_{2,1} = A_{2,1} \cdot B_{1,1} + A_{2,2} \cdot B_{2,1}$

$C_{2,2} = A_{2,1} \cdot B_{1,2} + A_{2,2} \cdot B_{2,2}$

Challenge 5: Recursive matrix multiplication
How to multiply submatrices?

- Use the same routine that is computing the full matrix multiplication
  - Quarter each input submatrix and output submatrix
  - Treat each sub-submatrix as a single element and multiply



$C_{1,1} = A_{1,1} \cdot B_{1,1} + A_{1,2} \cdot B_{2,1}$

$C11_{1,1} = A11_{1,1} \cdot B11_{1,1} + A11_{1,2} \cdot B11_{2,1} +$
$A12_{1,1} \cdot B21_{1,1} + A12_{1,2} \cdot B21_{2,1}$

Challenge 5: Recursive matrix multiplication
Recursively multiply submatrices

$$C_{1,1} = A_{1,1} \cdot B_{1,1} + A_{1,2} \cdot B_{2,1} \qquad C_{1,2} = A_{1,1} \cdot B_{1,2} + A_{1,2} \cdot B_{2,2}$$

$$C_{2,1} = A_{2,1} \cdot B_{1,1} + A_{2,2} \cdot B_{2,1} \qquad C_{2,2} = A_{2,1} \cdot B_{1,2} + A_{2,2} \cdot B_{2,2}$$

- Need range of indices to define each submatrix to be used

```
void matmultrec(int mf, int ml, int nf, int nl, int pf, int pl,
                double **A, double **B, double **C)
{// Dimensions: A[mf..ml][pf..pl]  B[pf..pl][nf..nl]  C[mf..ml][nf..nl]

// C11 += A11*B11
 matmultrec(mf, mf+(ml-mf)/2, nf, nf+(nl-nf)/2, pf, pf+(pl-pf)/2, A,B,C);
// C11 += A12*B21
 matmultrec(mf, mf+(ml-mf)/2, nf, nf+(nl-nf)/2, pf+(pl-pf)/2, pl, A,B,C);
   . . .
}
```

- Also need stopping criteria for recursion

Appendices

- Sources for additional information
- OpenMP History
- Solutions to exercises
  - Exercise 1: hello world
  - Exercise 2: Simple SPMD Pi program
  - Exercise 3: SPMD Pi without false sharing
  - Exercise 4: Loop level Pi
  - Exercise 5: Mandelbrot Set area
  - Exercise 6: Recursive pi program
- Challenge Problems
  - Challenge 1: Molecular dynamics
  - Challenge 2: Monte Carlo pi and random numbers
  - Challenge 3: Matrix multiplication
  - Challenge 4: Linked lists
  - Challenge 5: Recursive matrix multiplication
- Fortran and OpenMP
- Mixing OpenMP and MPI
- Compiler notes

OpenMP organizations

- OpenMP architecture review board URL, the "owner" of the OpenMP specification:

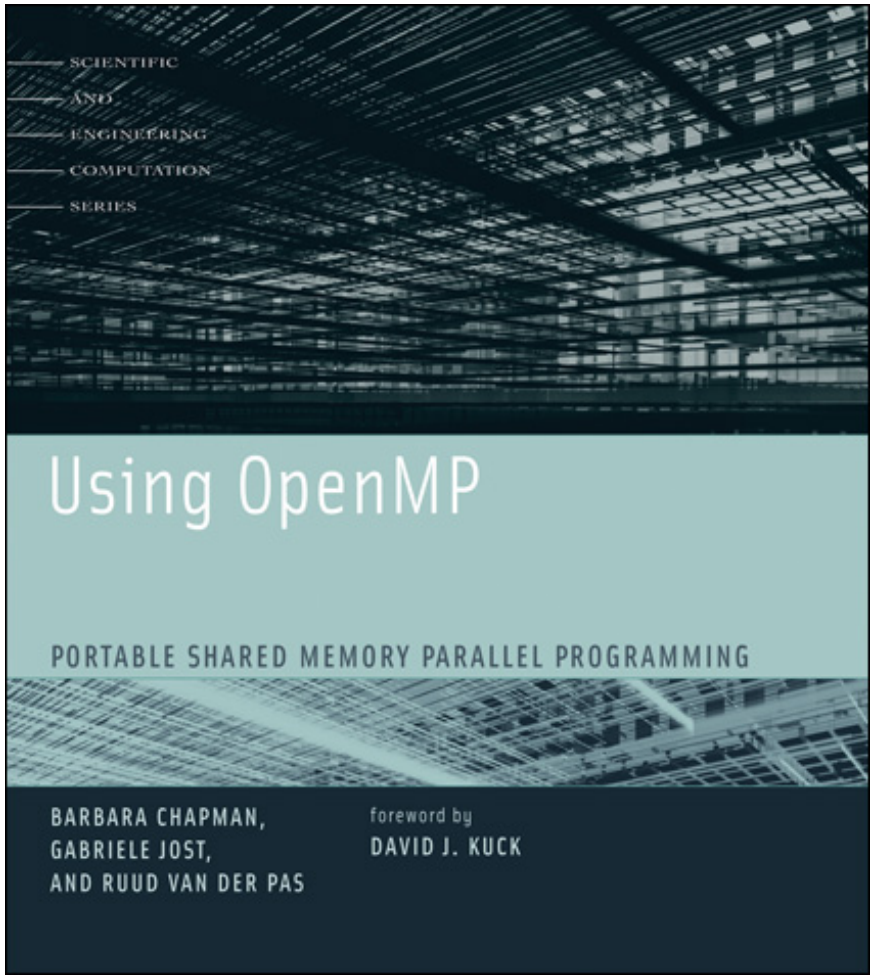
  www.openmp.org

- OpenMP User's Group (cOMPunity) URL:

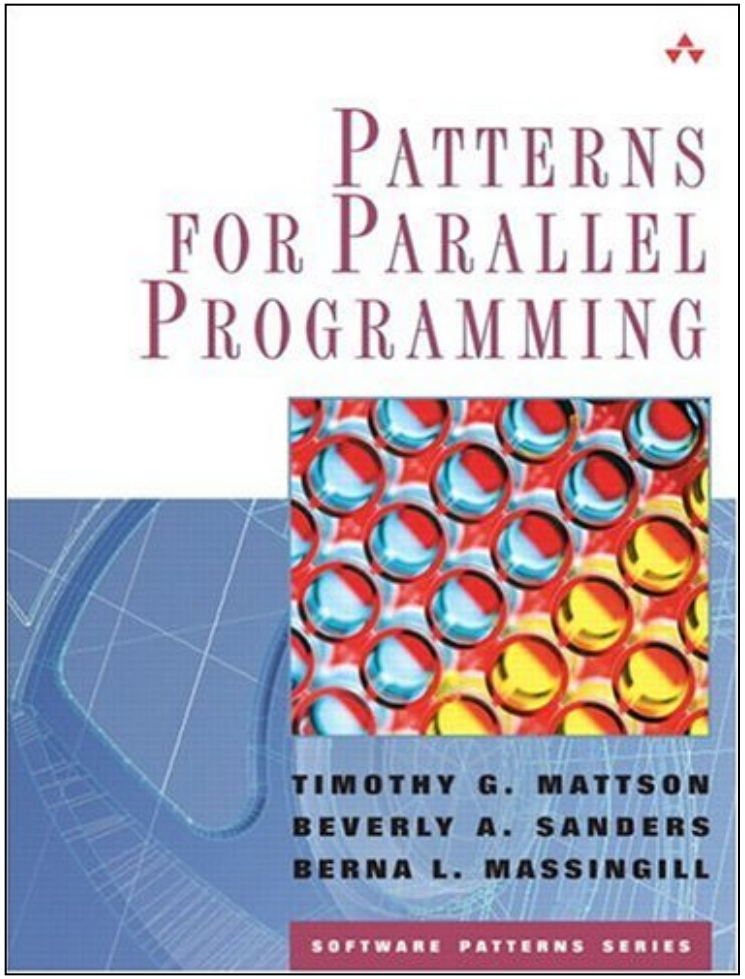
  www.compunity.org

Get involved, join cOMPunity and help define the future of OpenMP

Books about OpenMP

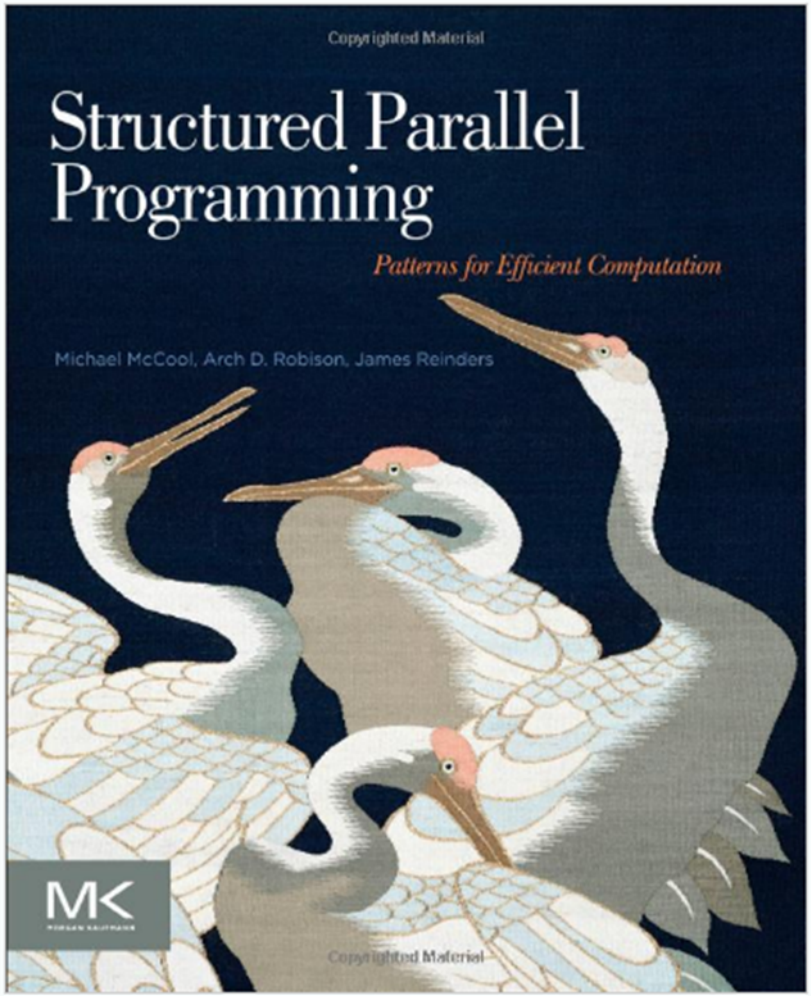


- A book about OpenMP by a team of authors at the forefront of OpenMP's evolution.

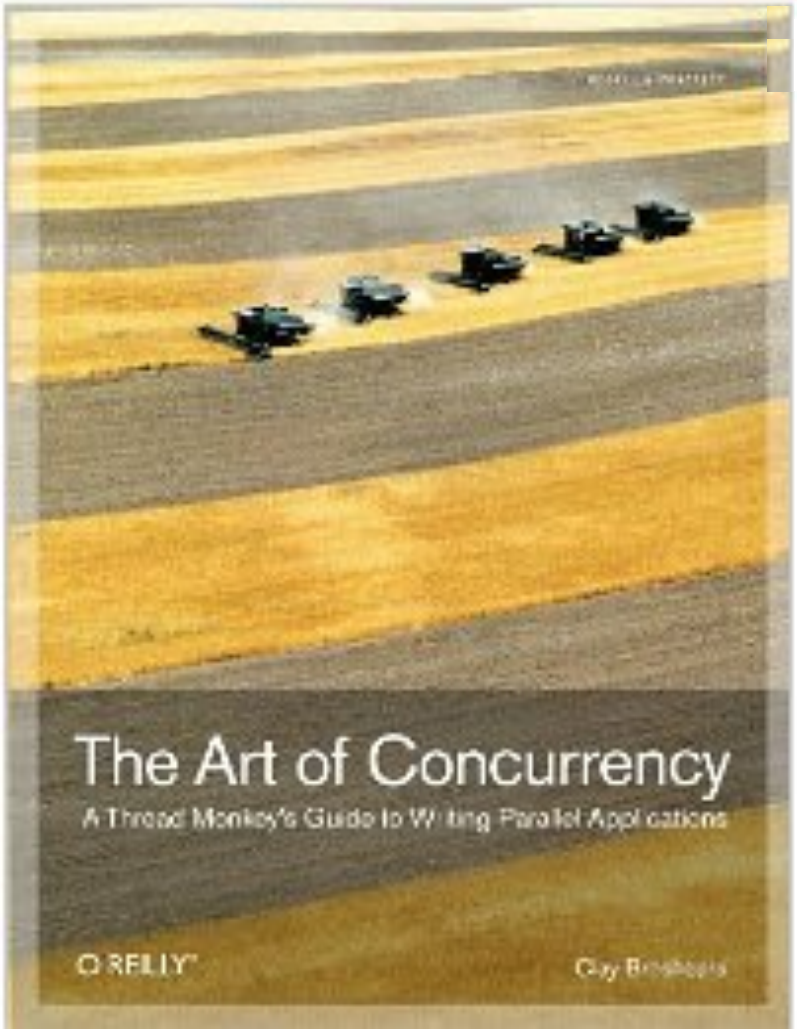


- A book about how to "think parallel" with examples in OpenMP, MPI and java

Background references



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



An excellent introduction and overview of multithreaded programming in general (by Clay Breshears)

OpenMP Papers

- Sosa CP, Scalmani C, Gomperts R, Frisch MJ. Ab initio quantum chemistry on a ccNUMA architecture using OpenMP. III. Parallel Computing, vol.26, no.7-8, July 2000, pp.843-56. Publisher: Elsevier, Netherlands.
- Couturier R, Chipot C. Parallel molecular dynamics using OPENMP on a shared memory machine. Computer Physics Communications, vol.124, no.1, Jan. 2000, pp.49-59. Publisher: Elsevier, Netherlands.
- Bentz J., Kendall R., "Parallelization of General Matrix Multiply Routines Using OpenMP", Shared Memory Parallel Programming with OpenMP, Lecture notes in Computer Science, Vol. 3349, P. 1, 2005
- Bova SW, Breshearsz CP, Cuicchi CE, Demirbilek Z, Gabb HA. Dual-level parallel analysis of harbor wave response using MPI and OpenMP. International Journal of High Performance Computing Applications, vol.14, no.1, Spring 2000, pp.49-64. Publisher: Sage Science Press, USA.
- Ayguade E, Martorell X, Labarta J, Gonzalez M, Navarro N. Exploiting multiple levels of parallelism in OpenMP: a case study. Proceedings of the 1999 International Conference on Parallel Processing. IEEE Comput. Soc. 1999, pp.172-80. Los Alamitos, CA, USA.
- Bova SW, Breshears CP, Cuicchi C, Demirbilek Z, Gabb H. Nesting OpenMP in an MPI application. Proceedings of the ISCA 12th International Conference. Parallel and Distributed Systems. ISCA. 1999, pp.566-71. Cary, NC, USA.

OpenMP Papers (continued)

- Jost G., Labarta J., Gimenez J., What Multilevel Parallel Programs do when you are not watching: a Performance analysis case study comparing MPI/OpenMP, MLP, and Nested OpenMP, Shared Memory Parallel Programming with OpenMP, Lecture notes in Computer Science, Vol. 3349, P. 29, 2005
- Gonzalez M, Serra A, Martorell X, Oliver J, Ayguade E, Labarta J, Navarro N. Applying interposition techniques for performance analysis of OPENMP parallel applications. Proceedings 14th International Parallel and Distributed Processing Symposium. IPDPS 2000. IEEE Comput. Soc. 2000, pp.235-40.
- Chapman B, Mehrotra P, Zima H. Enhancing OpenMP with features for locality control. Proceedings of Eighth ECMWF Workshop on the Use of Parallel Processors in Meteorology. Towards Teracomputing. World Scientific Publishing. 1999, pp.301-13. Singapore.
- Steve W. Bova, Clay P. Breshears, Henry Gabb, Rudolf Eigenmann, Greg Gaertner, Bob Kuhn, Bill Magro, Stefano Salvini. Parallel Programming with Message Passing and Directives; SIAM News, Volume 32, No 9, Nov. 1999.
- Cappello F, Richard O, Etiemble D. Performance of the NAS benchmarks on a cluster of SMP PCs using a parallelization of the MPI programs with OpenMP. Lecture Notes in Computer Science Vol.1662. Springer-Verlag. 1999, pp.339-50.
- Liu Z., Huang L., Chapman B., Weng T., Efficient Implementationi of OpenMP for Clusters with Implicit Data Distribution, Shared Memory Parallel Programming with OpenMP, Lecture notes in Computer Science, Vol. 3349, P. 121, 2005

OpenMP Papers (continued)

- B. Chapman, F. Bregier, A. Patil, A. Prabhakar, “Achieving performance under OpenMP on ccNUMA and software distributed shared memory systems,” *Concurrency and Computation: Practice and Experience.* 14(8-9): 713-739, 2002.
- J. M. Bull and M. E. Kambites. JOMP: an OpenMP-like interface for Java. Proceedings of the ACM 2000 conference on Java Grande, 2000, Pages 44 - 53.
- L. Adhianto and B. Chapman, “Performance modeling of communication and computation in hybrid MPI and OpenMP applications, Simulation Modeling Practice and Theory, vol 15, p. 481-491, 2007.
- Shah S, Haab G, Petersen P, Throop J. Flexible control structures for parallelism in OpenMP; Concurrency: Practice and Experience, 2000; 12:1219-1239. Publisher John Wiley & Sons, Ltd.
- Mattson, T.G., How Good is OpenMP? Scientific Programming, Vol. 11, Number 2, p.81-93, 2003.
- Duran A., Silvera R., Corbalan J., Labarta J., “Runtime Adjustment of Parallel Nested Loops”, Shared Memory Parallel Programming with OpenMP, Lecture notes in Computer Science, Vol. 3349, P. 137, 2005

Appendices

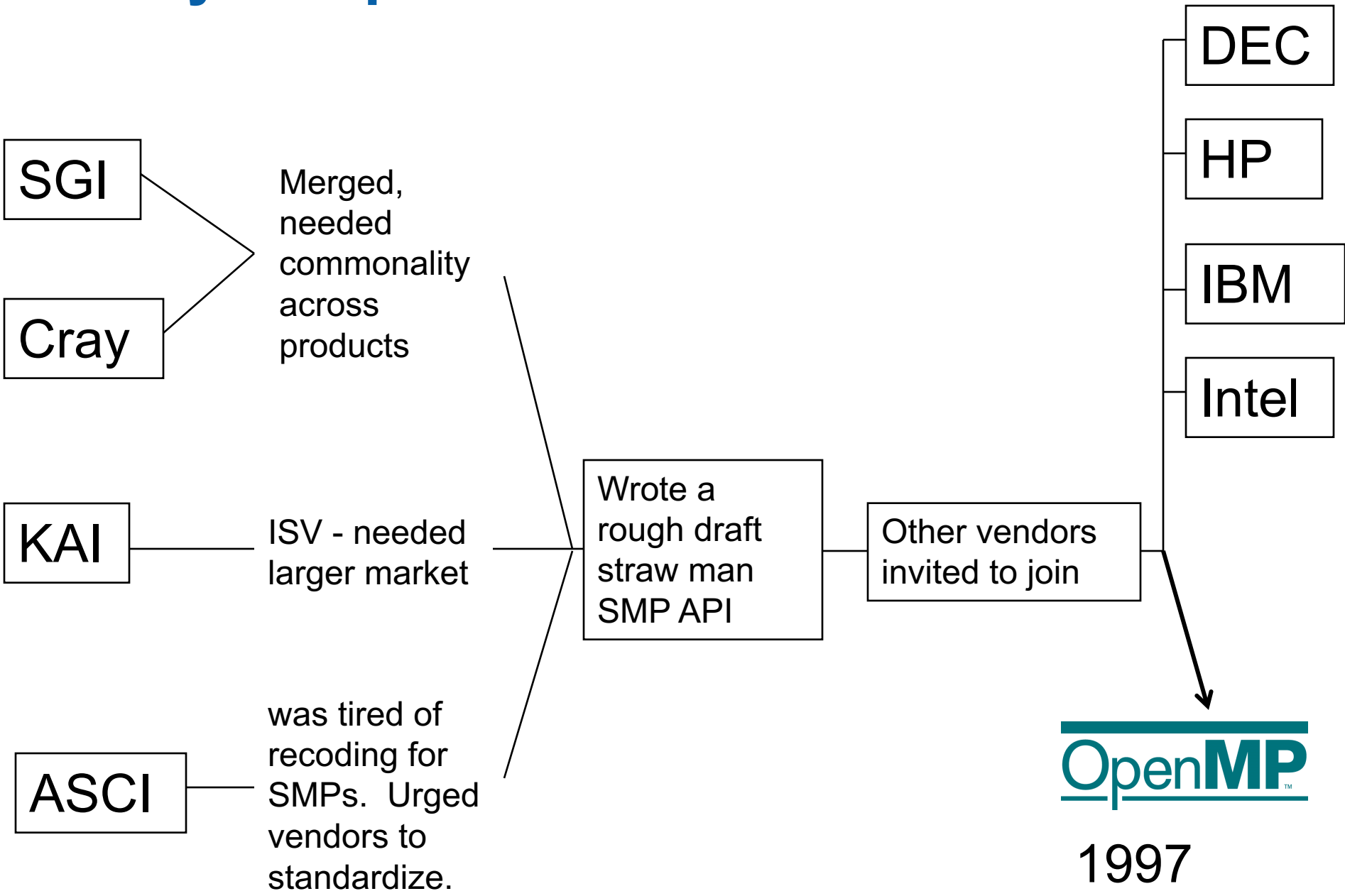
- Sources for additional information
- OpenMP History
- Solutions to exercises
  - Exercise 1: hello world
  - Exercise 2: Simple SPMD pi program
  - Exercise 3: SPMD pi without false sharing
  - Exercise 4: Loop level pi
  - Exercise 5: Mandelbrot Set area
  - Exercise 6: Recursive pi program
- Challenge Problems
  - Challenge 1: Molecular dynamics
  - Challenge 2: Monte Carlo pi and random numbers
  - Challenge 3: Matrix multiplication
  - Challenge 4: Linked lists
  - Challenge 5: Recursive matrix multiplication
- Fortran and OpenMP
- Mixing OpenMP and MPI
- Compiler notes

OpenMP pre-history

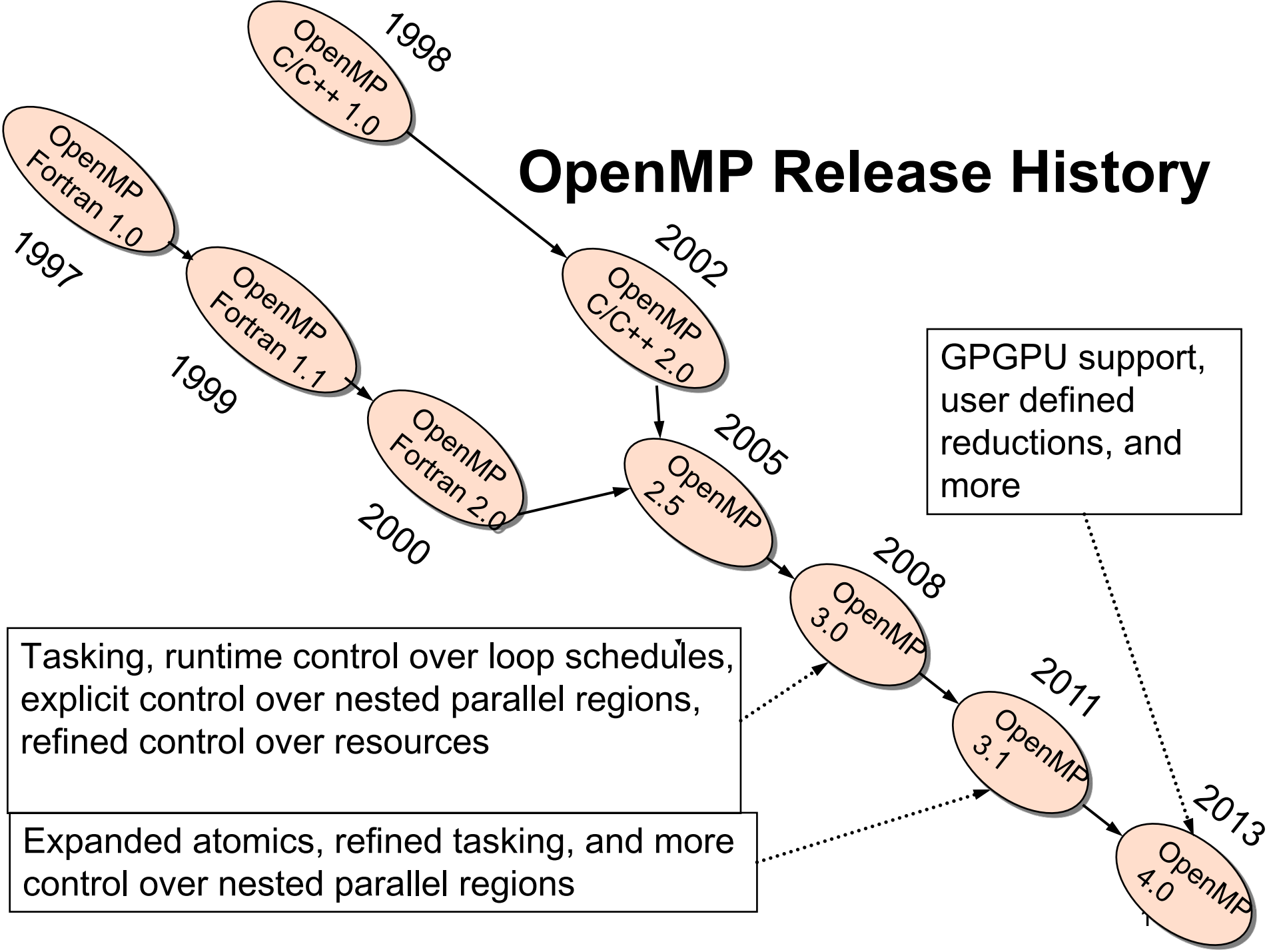
- OpenMP based upon SMP directive standardization efforts PCF and aborted ANSI X3H5 – late 80's
  - Nobody fully implemented either standard
  - Only a couple of partial implementations
- Vendors considered proprietary API's to be a competitive feature:
  - Every vendor had proprietary directives sets
  - Even KAP, a "portable" multi-platform parallelization tool used different directives on each platform

PCF – Parallel computing forum KAP – parallelization tool from KAI.

History of OpenMP



OpenMP Release History



Appendices

- Sources for Additional information
- OpenMP History
- Solutions to exercises
  - Exercise 1: hello world
  - Exercise 2: Simple SPMD Pi program
  - Exercise 3: SPMD Pi without false sharing
  - Exercise 4: Loop level Pi
  - Exercise 5: Mandelbrot Set area
  - Exercise 6: Recursive pi program
- Challenge Problems
  - Challenge 1: Molecular dynamics
  - Challenge 2: Monte Carlo pi and random numbers
  - Challenge 3: Matrix multiplication
  - Challenge 4: linked lists
  - Challenge 5: Recursive matrix multiplication
- Fortran and OpenMP
- Mixing OpenMP and MPI
- Compiler Notes

Appendices

- Sources for Additional information
- OpenMP History
- Solutions to exercises
  - Exercise 1: hello world
  - Exercise 2: Simple SPMD Pi program
  - Exercise 3: SPMD Pi without false sharing
  - Exercise 4: Loop level Pi
  - Exercise 5: Mandelbrot Set area
  - Exercise 6: Recursive pi program
- Challenge Problems
  - Challenge 1: Molecular dynamics
  - Challenge 2: Monte Carlo pi and random numbers
  - Challenge 3: Matrix multiplication
  - Challenge 4: linked lists
  - Challenge 5: Recursive matrix multiplication
- Fortran and OpenMP
- Mixing OpenMP and MPI
- Compiler Notes

Exercise 1: Solution

A multi-threaded “Hello world” program

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

```
#include “omp.h”
void main()
{
#pragma omp parallel
 {
    int ID = omp_get_thread_num();
    printf(“ hello(%d) ”, ID);
    printf(“ world(%d) \n”, ID);
  }
}
```

OpenMP include file

Parallel region with default number of threads

End of the Parallel region

Runtime library function to return a thread ID.

Sample Output:

hello(1) hello(0) world(1)

world(0)

hello (3) hello(2) world(3)

world(2)

Appendices

- Sources for Additional information
- OpenMP History
- Solutions to exercises
  - Exercise 1: hello world
  - Exercise 2: Simple SPMD Pi program
  - Exercise 3: SPMD Pi without false sharing
  - Exercise 4: Loop level Pi
  - Exercise 5: Mandelbrot Set area
  - Exercise 6: Recursive pi program
- Challenge Problems
  - Challenge 1: Molecular dynamics
  - Challenge 2: Monte Carlo pi and random numbers
  - Challenge 3: Matrix multiplication
  - Challenge 4: linked lists
  - Challenge 5: Recursive matrix multiplication
- Fortran and OpenMP
- Mixing OpenMP and MPI
- Compiler Notes

The SPMD pattern

- The most common approach for parallel algorithms is the SPMD or Single Program Multiple Data pattern.
- Each thread runs the same program (Single Program), but using the thread ID, they operate on different data (Multiple Data) or take slightly different paths through the code.
- In OpenMP this means:
  - A parallel region “near the top of the code”.
  - Pick up thread ID and num_threads.
  - Use them to split up loops and select different blocks of data to work on.

Exercise 2: 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,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;
}
```

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

Appendices

- Sources for Additional information
- OpenMP History
- Solutions to exercises
  - Exercise 1: hello world
  - Exercise 2: Simple SPMD Pi program
  - Exercise 3: SPMD Pi without false sharing
  - Exercise 4: Loop level Pi
  - Exercise 5: Mandelbrot Set area
  - Exercise 6: Recursive pi program
- Challenge Problems
  - Challenge 1: molecular dynamics
  - Challenge 2: Monte Carlo pi and random numbers
  - Challenge 3: Matrix multiplication
  - Challenge 4: linked lists
  - Challenge 5: Recursive matrix multiplication
- Fortran and OpenMP
- Mixing OpenMP and MPI
- Compiler Notes

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.
  - Result … poor scalability
- Solution:
  - When updates to an item are frequent, work with local copies of data instead of an array indexed by the thread ID.
  - Pad arrays so elements you use are on distinct cache lines.

Exercise 3: SPMD pi without false sharing

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

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

No array, so no false sharing.

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

Appendices

- Sources for Additional information
- OpenMP History
- Solutions to exercises
  - Exercise 1: hello world
  - Exercise 2: Simple SPMD Pi program
  - Exercise 3: SPMD Pi without false sharing
  - Exercise 4: Loop level Pi
  - Exercise 5: Mandelbrot Set area
  - Exercise 6: Recursive pi program
- Challenge Problems
  - Challenge 1: molecular dynamics
  - Challenge 2: Monte Carlo pi and random numbers
  - Challenge 3: Matrix multiplication
  - Challenge 4: linked lists
  - Challenge 5: Recursive matrix multiplication
- Fortran and OpenMP
- Mixing OpenMP and MPI
- Compiler Notes

Exercise 4: Solution

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

Exercise 4: Solution

```
#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 for private(x) 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;
}
```

For good OpenMP implementations, reduction is more scalable than critical.

i private by default

Note: we created a parallel program without changing any code and by adding 2 simple lines of text!

Appendices

- Sources for Additional information
- OpenMP History
- Solutions to exercises
  - Exercise 1: hello world
  - Exercise 2: Simple SPMD Pi program
  - Exercise 3: SPMD Pi without false sharing
  - Exercise 4: Loop level Pi
  - Exercise 5: Mandelbrot Set area
  - Exercise 6: Recursive pi program
- Challenge Problems
  - Challenge 1: molecular dynamics
  - Challenge 2: Monte Carlo pi and random numbers
  - Challenge 3: Matrix multiplication
  - Challenge 4: linked lists
  - Challenge 5: Recursive matrix multiplication
- Fortran and OpenMP
- Mixing OpenMP and MPI
- Compiler Notes

Exercise 5: The Mandelbrot area program

```
#include <omp.h>
# define NPOINTS 1000
# define MXITR 1000
void testpoint(void);
struct d_complex{
   double r;     double i;
};
struct d_complex c;
int numoutside = 0;

int main(){
   int i, j;
   double area, error, eps  = 1.0e-5;
#pragma omp parallel for default(shared) \
                              private(c,eps)
   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();
     }
   }
area=2.0*2.5*1.125*(double)(NPOINTS*NPOINTS-
numoutside)/(double)(NPOINTS*NPOINTS);
   error=area/(double)NPOINTS;
}
```

```
void testpoint(void){
struct d_complex z;
       int iter;
       double temp;

       z=c;
       for (iter=0; iter<MXITR; iter++){
         temp = (z.r*z.r)-(z.i*z.i)+c.r;
         z.i = z.r*z.i*2+c.i;
         z.r = temp;
         if ((z.r*z.r+z.i*z.i)>4.0) {
           numoutside++;
           break;
         }
       }
}
```

When I run this program, I get a different incorrect answer each time I run it … there is a race condition!!!!

Exercise 5: Area of a Mandelbrot set

- Solution is in the file mandel_par.c
- Errors:
  - Eps is private but uninitialized. Two solutions
    - It's read-only so you can make it shared.
    - Make it firstprivate
  - The loop index variable j is shared by default; make it private
  - The variable c has global scope so "testpoint" may pick up the global value rather than the private value in the loop; solution … pass c as an arg to testpoint
  - Updates to "numoutside" are a race; protect with an atomic.

Debugging parallel programs

- Find tools that work with your environment and learn to use them; a good parallel debugger can make a huge difference
- But parallel debuggers are not portable and you will assuredly need to debug “by hand” at some point
- There are tricks to help you; the most important is to use the default(none) pragma

```
#pragma omp parallel for default(none) private(c, eps)
  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();
    }
  }
}
```

Using default(none) generates a compiler error that j is unspecified.

Exercise 5: The Mandelbrot area program

```
#include <omp.h>
# define NPOINTS 1000
# define MXITR 1000
struct d_complex{
   double r;     double i;
};
void testpoint(struct d_complex);
struct d_complex c;
int numoutside = 0;

int main(){
   int i, j;
   double area, error, eps  = 1.0e-5;
#pragma omp parallel for default(shared) private(c, j) \
    firstpriivate(eps)
   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);
     }
   }
area=2.0*2.5*1.125*(double)(NPOINTS*NPOINTS-
numoutside)/(double)(NPOINTS*NPOINTS);
   error=area/(double)NPOINTS;
}
```

```
void testpoint(struct  d_complex c){
struct d_complex z;
       int iter;
       double temp;

       z=c;
       for (iter=0; iter<MXITR; iter++){
         temp = (z.r*z.r)-(z.i*z.i)+c.r;
         z.i = z.r*z.i*2+c.i;
         z.r = temp;
         if ((z.r*z.r+z.i*z.i)>4.0) {
         #pragma omp atomic
           numoutside++;
           break;
         }
       }
}
```

Other errors found using a debugger or by inspection:

- eps was not initialized
- Protect updates of numoutside
- Which value of c die testpoint() see? Global or private?

Appendices

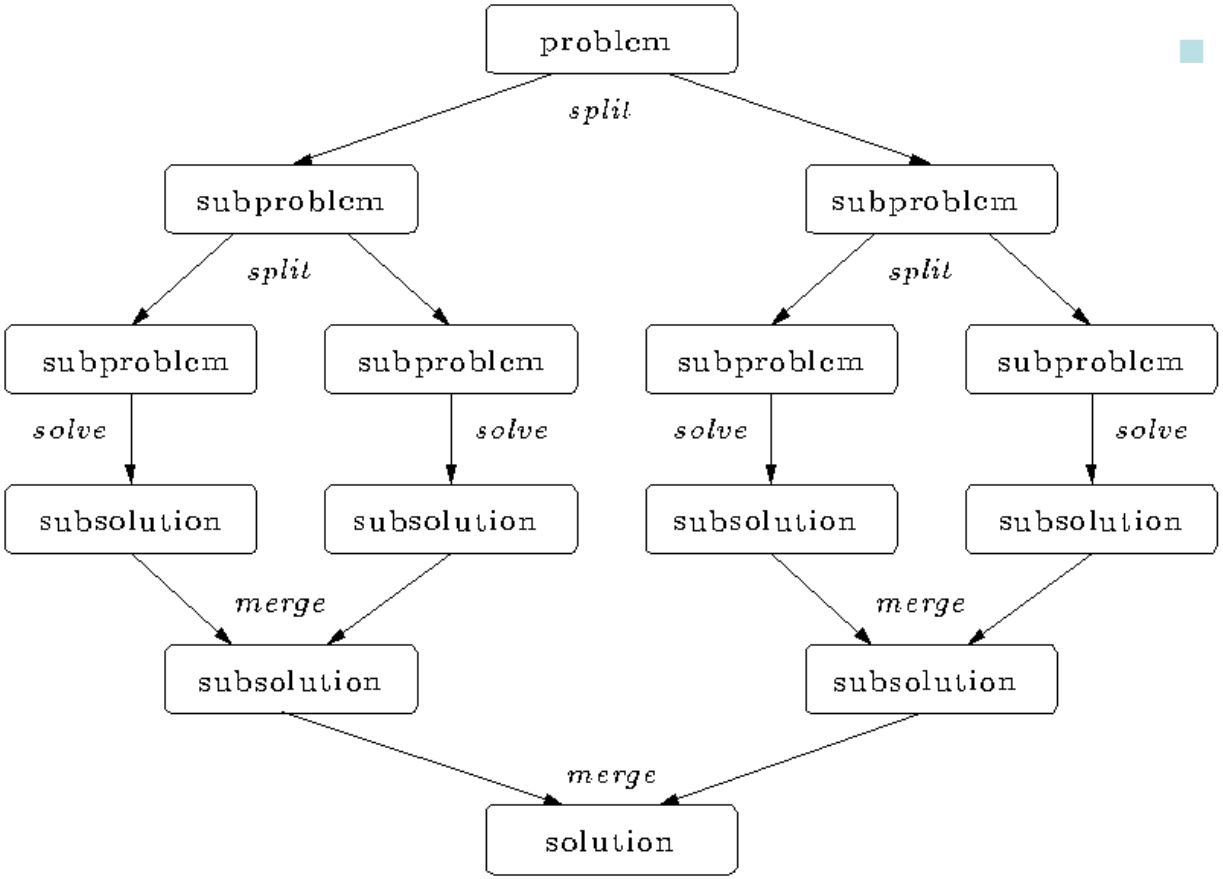
- Sources for Additional information
- OpenMP History
- Solutions to exercises
  - Exercise 1: hello world
  - Exercise 2: Simple SPMD Pi program
  - Exercise 3: SPMD Pi without false sharing
  - Exercise 4: Loop level Pi
  - Exercise 5: Mandelbrot Set area
  - Exercise 6: Recursive pi program
- Challenge Problems
  - Challenge 1: molecular dynamics
  - Challenge 2: Monte Carlo pi and random numbers
  - Challenge 3: Matrix multiplication
  - Challenge 4: linked lists
  - Challenge 5: Recursive matrix multiplication
- Fortran and OpenMP
- Mixing OpenMP and MPI
- Compiler Notes

Divide and conquer pattern

- Use when:
  - A problem includes a method to divide into subproblems and a way to recombine solutions of subproblems into a global solution
- Solution
  - Define a split operation
  - Continue to split the problem until subproblems are small enough to solve directly
  - Recombine solutions to subproblems to solve original global problem
- Note:
  - Computing may occur at each phase (split, leaves, recombine)

Divide and conquer

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



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

Program: OpenMP tasks (divide and conquer pattern)

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

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

Results*: pi with tasks

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

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

Appendices

- Sources for Additional information
- OpenMP History
- Solutions to exercises
  - Exercise 1: hello world
  - Exercise 2: Simple SPMD Pi program
  - Exercise 3: SPMD Pi without false sharing
  - Exercise 4: Loop level Pi
  - Exercise 5: Mandelbrot Set area
  - Exercise 6: Recursive pi program
- Challenge Problems
  - Challenge 1: molecular dynamics
  - Challenge 2: Monte Carlo pi and random numbers
  - Challenge 3: Matrix multiplication
  - Challenge 4: linked lists
  - Challenge 5: Recursive matrix multiplication
- Fortran and OpenMP
- Mixing OpenMP and MPI
- Compiler Notes

Appendices

- Sources for Additional information
- OpenMP History
- Solutions to exercises
  - Exercise 1: hello world
  - Exercise 2: Simple SPMD Pi program
  - Exercise 3: SPMD Pi without false sharing
  - Exercise 4: Loop level Pi
  - Exercise 5: Mandelbrot Set area
  - Exercise 6: Recursive pi program
- Challenge Problems
  - Challenge 1: molecular dynamics
  - Challenge 2: Monte Carlo pi and random numbers
  - Challenge 3: Matrix multiplication
  - Challenge 4: linked lists
  - Challenge 5: Recursive matrix multiplication
- Fortran and OpenMP
- Mixing OpenMP and MPI
- Compiler Notes

Challenge 1: Solution

```
#pragma omp parallel for default (none) \
    shared(x,f,npart,rcoff,side) \
    reduction(+:epot,vir) \
    schedule (static,32)
    for (int i=0; i<npart*3; i+=3) {
    ……
```

Compiler will warn you if you have missed some variables

Loop is not well load balanced: best schedule has to be found by experiment.

Challenge 1: Solution (cont.)

```
.....
#pragma omp atomic
        f[j]    -= forcex;
#pragma omp atomic
        f[j+1]  -= forcey;
#pragma omp atomic
        f[j+2]  -= forcez;
      }
    }
#pragma omp atomic
    f[i]    += fxi;
#pragma omp atomic
    f[i+1]   += fyi;
#pragma omp atomic
    f[i+2]   += fzi;
  }
 }
```

All updates to f must be atomic

Challenge 1: With orphaning

```
#pragma omp single
{
   vir    = 0.0;
   epot   = 0.0;
}
#pragma omp for reduction(+:epot,vir)  schedule (static,32)
    for (int i=0; i<npart*3; i+=3) {
……
```

Implicit barrier needed to avoid race condition with update of reduction variables at end of the for construct

All variables which used to be shared here are now implicitly determined

Challenge 1: With array reduction

```
      ftemp[myid][j]   -= forcex;
      ftemp[myid][j+1]  -= forcey;
      ftemp[myid][j+2]  -= forcez;
    }
  }
   ftemp[myid][i]        += fxi;
   ftemp[myid][i+1]       += fyi;
   ftemp[myid][i+2]       += fzi;
}
```

Replace atomics with accumulation into array with extra dimension

Challenge 1: With array reduction

```
….
#pragma omp for
   for(int i=0;i<(npart*3);i++){
         for(int id=0;id<nthreads;id++){
            f[i] += ftemp[id][i];
           ftemp[id][i] = 0.0;
      }
   }
```

Reduction can be done in parallel

Zero ftemp for next time round

Appendices

- Sources for Additional information
- OpenMP History
- Solutions to exercises
  - Exercise 1: hello world
  - Exercise 2: Simple SPMD Pi program
  - Exercise 3: SPMD Pi without false sharing
  - Exercise 4: Loop level Pi
  - Exercise 5: Mandelbrot Set area
  - Exercise 6: Recursive pi program
- Challenge Problems
  - Challenge 1: molecular dynamics
  - Challenge 2: Monte Carlo pi and random numbers
  - Challenge 3: Matrix multiplication
  - Challenge 4: linked lists
  - Challenge 5: Recursive matrix multiplication
- Fortran and OpenMP
- Mixing OpenMP and MPI
- Compiler Notes

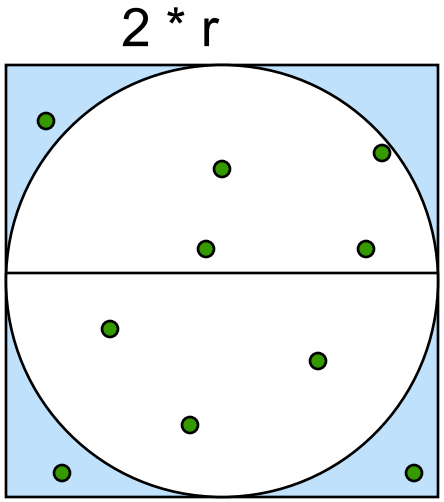
Computers and random numbers

- We use “dice” to make random numbers:
  - Given previous values, you cannot predict the next value.
  - There are no patterns in the series … and it goes on forever.
- Computers are deterministic machines … set an initial state, run a sequence of predefined instructions, and you get a deterministic answer
  - By design, computers are not random and cannot produce random numbers.
- However, with some very clever programming, we can make “pseudo random” numbers that are as random as you need them to be … but only if you are very careful.
- Why do I care? Random numbers drive statistical methods used in countless applications:
  - Sample a large space of alternatives to find statistically good answers (Monte Carlo methods).

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	π = 2.8
N=100	π = 3.16
N= 1000	π = 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, count the fraction that falls in the circle, compute pi.

Parallel Programmers love Monte Carlo algorithms

Embarrassingly parallel: the parallelism is so easy its embarrassing.

Add two lines and you have a parallel program.

```
#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 squrare 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);
}
```

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;
```

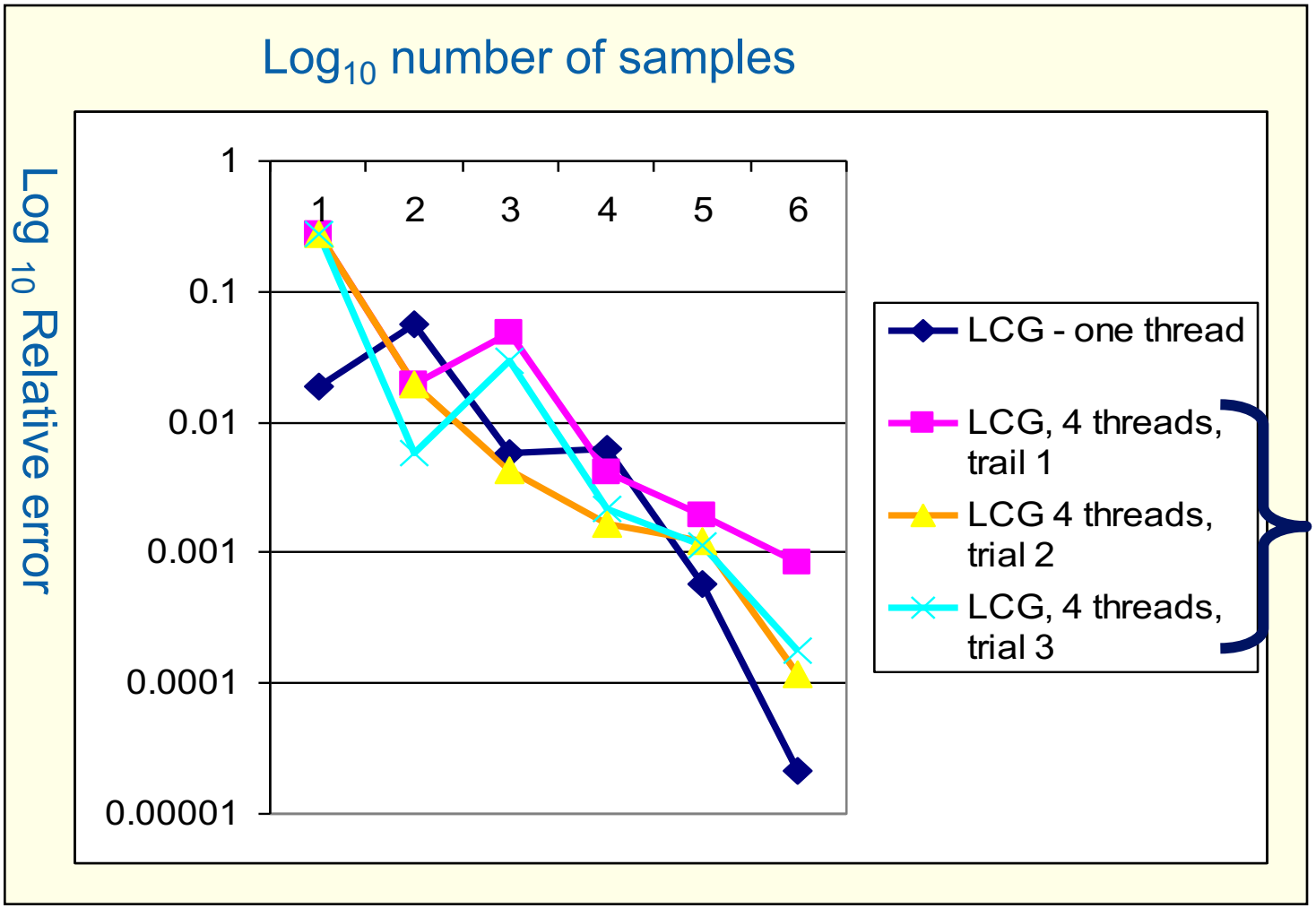
- If you pick the multiplier and addend correctly, LCG has a period of PMOD.
- Picking good LCG parameters is complicated, so look it up (Numerical Recipes is a good source). I used the following:
  - MULTIPLIER = 1366
  - ADDEND = 150889
  - PMOD = 714025

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



Run the same program the same way and get different answers!

That is not acceptable!

Issue: my LCG generator is not threadsafe

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.

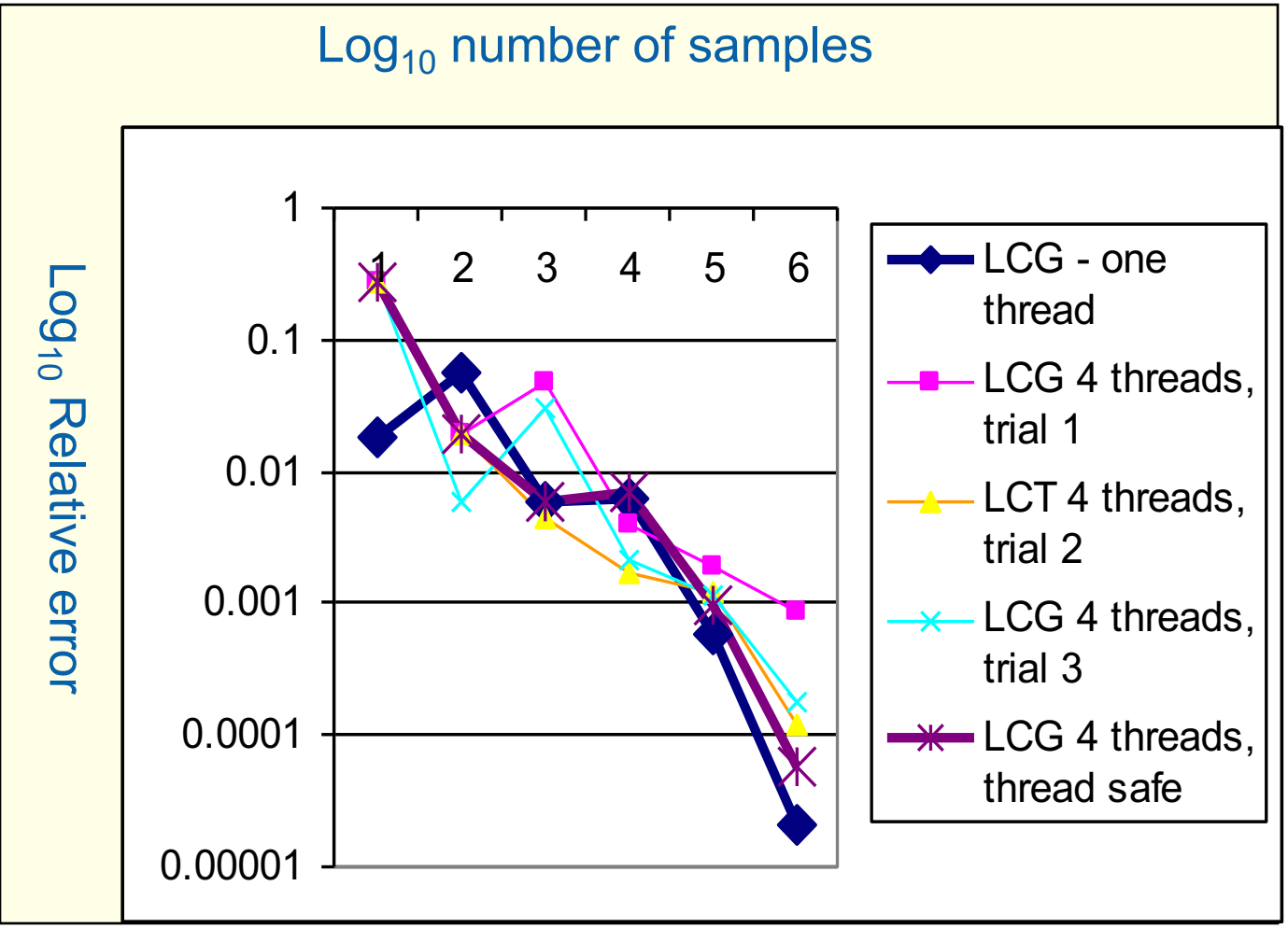
LCG code: threadsafe version

```
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 + AD
    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.

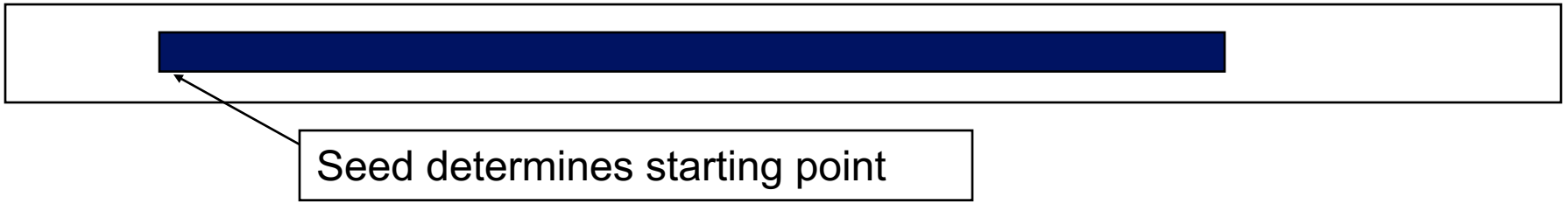
But for large number of samples, its quality is lower than the one thread result!

Why?

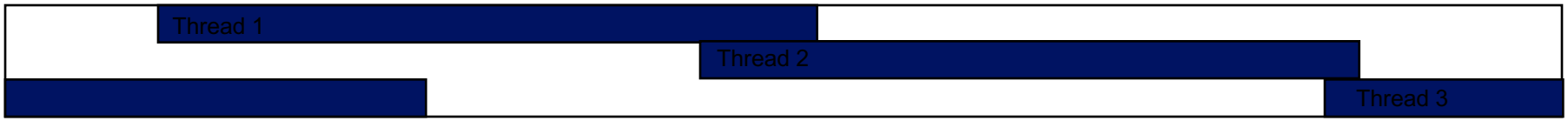
Pseudo Random Sequences

- 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 … buy 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 all of these methods.

MKL Random number generators (RNG)

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

```
#define BLOCK 100
double  buff[BLOCK];
VSLStreamStatePtr stream;

vslNewStream(&ran_stream, VSL_BRNG_WH, (int)seed_val);

vdRngUniform (VSL_METHOD_DUNIFORM_STD, stream,
                    BLOCK, buff, low, hi)

vslDeleteStream( &stream );
```

Initialize a stream or pseudo random numbers

Select type of RNG and set seed

Fill buff with BLOCK pseudo rand. nums, uniformly distributed with values between lo and hi.

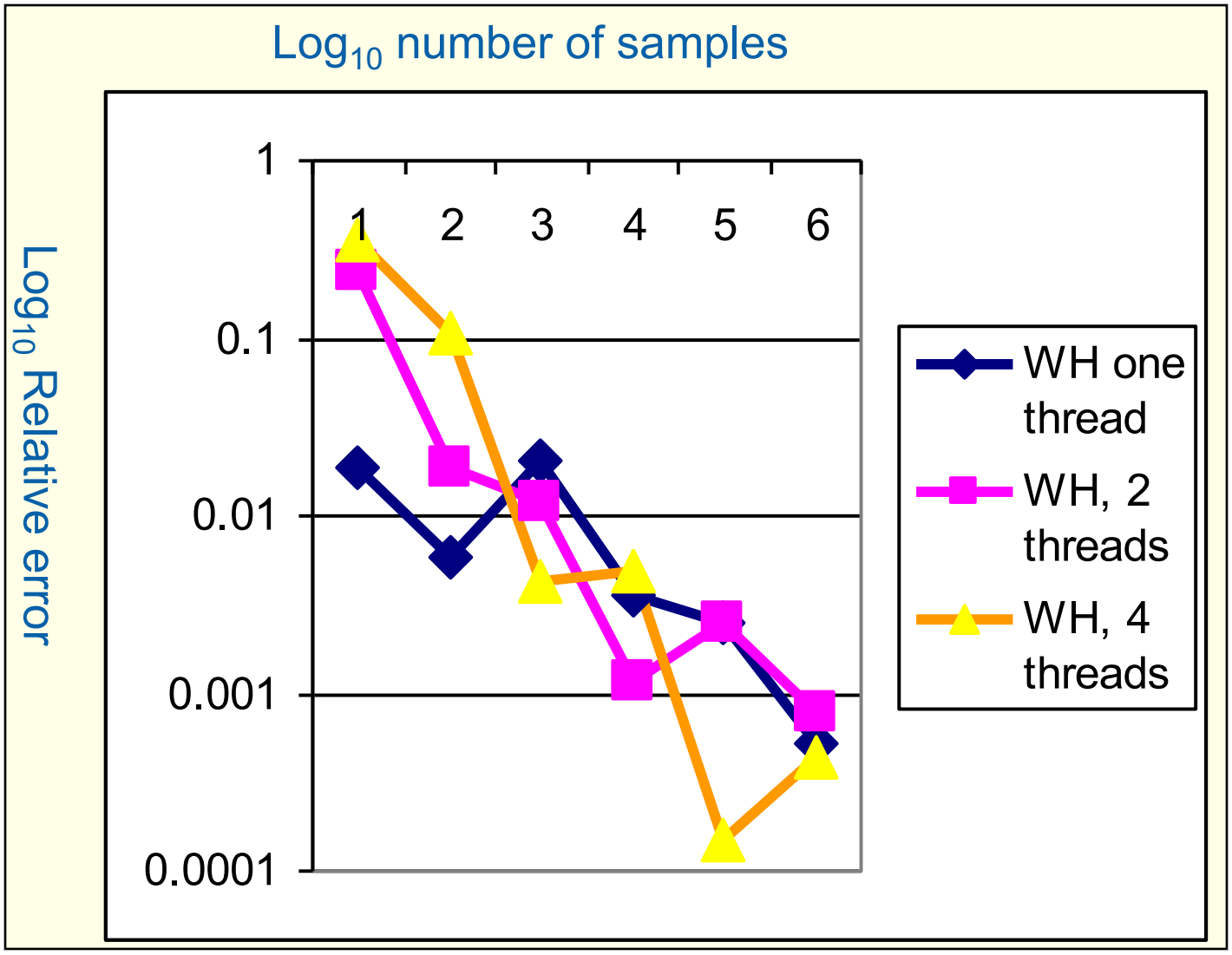
Delete the stream when you are done

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.

Appendices

- Sources for Additional information
- OpenMP History
- Solutions to exercises
  - Exercise 1: hello world
  - Exercise 2: Simple SPMD Pi program
  - Exercise 3: SPMD Pi without false sharing
  - Exercise 4: Loop level Pi
  - Exercise 5: Mandelbrot Set area
  - Exercise 6: Recursive pi program
- Challenge Problems
  - Challenge 1: molecular dynamics
  - Challenge 2: Monte Carlo Pi and random numbers
  - Challenge 3: Matrix multiplication
  - Challenge 4: linked lists
  - Challenge 5: Recursive matrix multiplication
- Fortran and OpenMP
- Mixing OpenMP and MPI
- Compiler Notes

Challenge 3: Matrix Multiplication

- Parallelize the matrix multiplication program in the file matmul.c
- Can you optimize the program by playing with how the loops are scheduled?
- Try the following and see how they interact with the constructs in OpenMP
  - Cache blocking
  - Loop unrolling
  - Vectorization
- Goal: Can you approach the peak performance of the computer?

Matrix multiplication

```
#pragma omp parallel for private(tmp, i, j, k)
  for (i=0; i<Ndim; i++){
        for (j=0; j<Mdim; j++){
              tmp = 0.0;
              for(k=0;k<Pdim;k++){
                    /* C(i,j) = sum(over k) A(i,k) * B(k,j) */
                    tmp += *(A+(i*Ndim+k)) *  *(B+(k*Pdim+j));
              }
              *(C+(i*Ndim+j)) = tmp;
        }
  }
```

- On a dual core laptop
  - 13.2 seconds 153 Mflops one thread
  - 7.5 seconds 270 Mflops two threads

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

Appendices

- Sources for Additional information
- OpenMP History
- Solutions to exercises
  - Exercise 1: hello world
  - Exercise 2: Simple SPMD Pi program
  - Exercise 3: SPMD Pi without false sharing
  - Exercise 4: Loop level Pi
  - Exercise 5: Mandelbrot Set area
  - Exercise 6: Recursive pi program
- Challenge Problems
  - Challenge 1: molecular dynamics
  - Challenge 2: Monte Carlo Pi and random numbers
  - Challenge 3: Matrix multiplication
  - Challenge 4: linked lists
  - Challenge 5: Recursive matrix multiplication
- Fortran and OpenMP
- Mixing OpenMP and MPI
- Compiler Notes

Challenge 4: traversing linked lists

- Consider the program linked.c
  - Traverses a linked list computing a sequence of Fibonacci numbers at each node.
- Parallelize this program two different ways
  1. Use OpenMP tasks
  2. Use anything you choose in OpenMP *other than* tasks.
- The second approach (no tasks) can be difficult and may take considerable creativity in how you approach the problem (hence why its such a pedagogically valuable problem).

Linked lists with tasks (OpenMP 3)

- See the file Linked_omp3_tasks.c

```
#pragma omp parallel
{
  #pragma omp single
  {
     p=head;
    while (p) {
      #pragma omp task firstprivate(p)
            processwork(p);
       p = p->next;
    }
  }
}
```

Creates a task with its own copy of “p” initialized to the value of “p” when the task is defined

Challenge 4: traversing linked lists

- Consider the program linked.c
  - Traverses a linked list computing a sequence of Fibonacci numbers at each node.
- Parallelize this program two different ways
  1. Use OpenMP tasks
  2. Use anything you choose in OpenMP *other than* tasks.
- The second approach (no tasks) can be difficult and may take considerable creativity in how you approach the problem (hence why its such a pedagogically valuable problem).

Linked lists without tasks

- See the file Linked_omp25.c

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

	Default schedule	Static,1
One Thread	48 seconds	45 seconds
Two Threads	39 seconds	28 seconds

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

Linked lists without tasks: C++ STL

- See the file Linked_cpp.cpp

```
std::vector<node *> nodelist;
for (p = head; p != NULL; p = p->next)
    nodelist.push_back(p);
```

Copy pointer to each node into an array

```
int j = (int)nodelist.size();
#pragma omp parallel for schedule(static,1)
    for (int i = 0; i < j; ++i)
        processwork(nodelist[i]);
```

Count number of items in the linked list

Process nodes in parallel with a for loop

	C++, default sched.	C++, (static,1)	C, (static,1)
One Thread	37 seconds	49 seconds	45 seconds
Two Threads	47 seconds	32 seconds	28 seconds

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

Appendices

- Sources for Additional information
- OpenMP History
- Solutions to exercises
  - Exercise 1: hello world
  - Exercise 2: Simple SPMD Pi program
  - Exercise 3: SPMD Pi without false sharing
  - Exercise 4: Loop level Pi
  - Exercise 5: Mandelbrot Set area
  - Exercise 6: Recursive pi program
- Challenge Problems
  - Challenge 1: molecular dynamics
  - Challenge 2: Monte Carlo Pi and random numbers
  - Challenge 3: Matrix multiplication
  - Challenge 4: linked lists
  - Challenge 5: Recursive matrix multiplication
- Fortran and OpenMP
- Mixing OpenMP and MPI
- Compiler Notes

Recursive matrix multiplication

- Could be executed in parallel as 4 tasks
  - Each task executes the two calls for the same output submatrix of C
- However, the same number of multiplication operations needed

```
#define THRESHOLD 32768   // product size below which simple matmult code is called

void matmultrec(int mf, int ml, int nf, int nl, int pf, int pl,
                double **A, double **B, double **C)

// Dimensions: A[mf..ml][pf..pl]    B[pf..pl][nf..nl]   C[mf..ml][nf..nl]

{
   if ((ml-mf)*(nl-nf)*(pl-pf) < THRESHOLD)
      matmult (mf, ml, nf, nl, pf, pl, A, B, C);
   else
   {
#pragma omp task firstprivate(mf,ml,nf,nl,pf,pl)
{
      matmultrec(mf, mf+(ml-mf)/2, nf, nf+(nl-nf)/2, pf, pf+(pl-pf)/2, A, B, C);  // C11 += A11*B11
      matmultrec(mf, mf+(ml-mf)/2, nf, nf+(nl-nf)/2, pf+(pl-pf)/2, pl, A, B, C);  // C11 += A12*B21
}
#pragma omp task firstprivate(mf,ml,nf,nl,pf,pl)
{
      matmultrec(mf, mf+(ml-mf)/2, nf+(nl-nf)/2, nl, pf, pf+(pl-pf)/2, A, B, C);  // C12 += A11*B12
      matmultrec(mf, mf+(ml-mf)/2, nf+(nl-nf)/2, nl, pf+(pl-pf)/2, pl, A, B, C);  // C12 += A12*B22
}
#pragma omp task firstprivate(mf,ml,nf,nl,pf,pl)
{
     matmultrec(mf+(ml-mf)/2, ml, nf, nf+(nl-nf)/2, pf, pf+(pl-pf)/2, A, B, C);  // C21 += A21*B11
     matmultrec(mf+(ml-mf)/2, ml, nf, nf+(nl-nf)/2, pf+(pl-pf)/2, pl, A, B, C);  // C21 += A22*B21
}
#pragma omp task firstprivate(mf,ml,nf,nl,pf,pl)
{
     matmultrec(mf+(ml-mf)/2, ml, nf+(nl-nf)/2, nl, pf, pf+(pl-pf)/2, A, B, C);  // C22 += A21*B12
     matmultrec(mf+(ml-mf)/2, ml, nf+(nl-nf)/2, nl, pf+(pl-pf)/2, pl, A, B, C);  // C22 += A22*B22
}
#pragma omp taskwait

   }
}
```

Appendices

- Sources for Additional information
- OpenMP History
- Solutions to exercises
  - Exercise 1: hello world
  - Exercise 2: Simple SPMD Pi program
  - Exercise 3: SPMD Pi without false sharing
  - Exercise 4: Loop level Pi
  - Exercise 5: Mandelbrot Set area
  - Exercise 6: Recursive pi program
- Challenge Problems
  - Challenge 1: molecular dynamics
  - Challenge 2: Monte Carlo Pi and random numbers
  - Challenge 3: Matrix multiplication
  - Challenge 4: linked lists
  - Challenge 5: Recursive matrix multiplication
- Fortran and OpenMP
- Mixing OpenMP and MPI
- Compiler Notes

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
  - CRICITAL … END CRITICAL
  - SECTION … END SECTION
  - SECTIONS … END SECTIONS
  - SINGLE … END SINGLE
  - MASTER … END MASTER

Runtime library routines

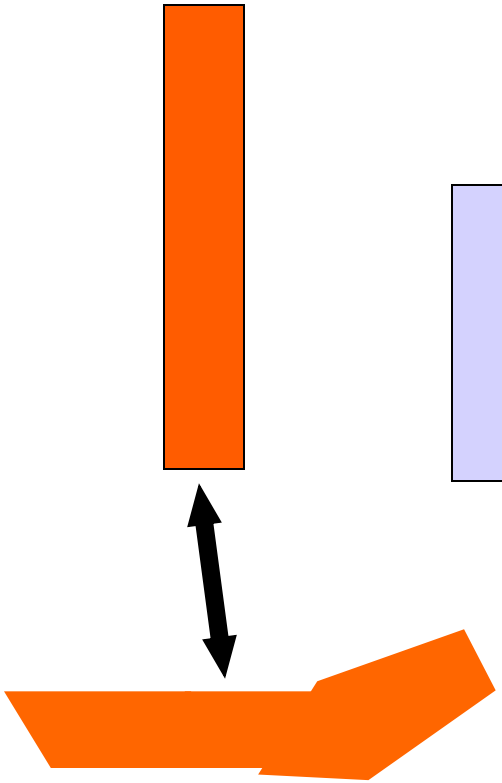
- The include file or module defines parameters
  - Integer parameter omp_locl_kind
  - Integer parameter omp_nest_lock_kind
  - Integer parameter omp_sched_kind
  - Integer parameter openmp_version
    - With value that matches C's _OPEMMP macro
- Fortran interfaces are similar to those used with C
  - 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)

Appendices

- Sources for Additional information
- OpenMP History
- Solutions to exercises
  - Exercise 1: hello world
  - Exercise 2: Simple SPMD Pi program
  - Exercise 3: SPMD Pi without false sharing
  - Exercise 4: Loop level Pi
  - Exercise 5: Mandelbrot Set area
  - Exercise 6: Recursive pi program
- Challenge Problems
  - Challenge 1: molecular dynamics
  - Challenge 2: Monte Carlo Pi and random numbers
  - Challenge 3: Matrix multiplication
  - Challenge 4: linked lists
  - Challenge 5: Recursive matrix multiplication
- Flush, memory models and OpenMP: producer consumer
- Fortran and OpenMP
- Mixing OpenMP and MPI
- Compiler Notes

How do people mix MPI and OpenMP?

A sequential program working on a data set



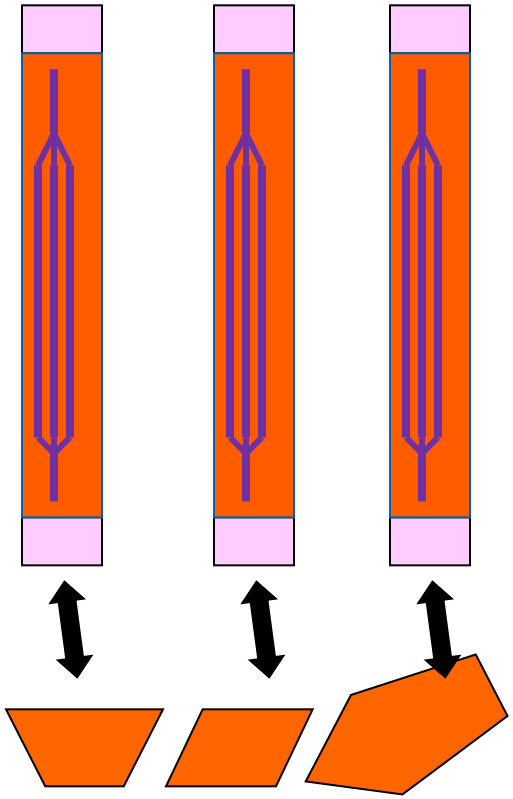
Replicate the program.

Add glue code

Break up the data

•Create the MPI program with its data decomposition.

• Use OpenMP inside each MPI process.



Pi program with MPI and OpenMP

```
#include <mpi.h>
#include “omp.h”
void main (int argc, char *argv[])
{
        int i, my_id, numprocs;  double x, pi, step, sum = 0.0 ;
        step = 1.0/(double) num_steps ;
        MPI_Init(&argc, &argv) ;
        MPI_Comm_Rank(MPI_COMM_WORLD, &my_id) ;
        MPI_Comm_Size(MPI_COMM_WORLD, &numprocs) ;
        my_steps = num_steps/numprocs ;
#pragma omp parallel for reduction(+:sum) private(x)
        for (i=my_id*my_steps; i<(m_id+1)*my_steps ; i++)
        {
                x = (i+0.5)*step;
                sum += 4.0/(1.0+x*x);
        }
        sum *= step ;
        MPI_Reduce(&sum, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,
                MPI_COMM_WORLD) ;
}
```

Get the MPI part done first, then add OpenMP pragma where it makes sense to do so

Key issues when mixing OpenMP and MPI

1. Messages are sent to a process not to a particular thread.
   - Not all MPIs are threadsafe. MPI 2.0 defines threading modes:
     - MPI_Thread_Single: no support for multiple threads
     - MPI_Thread_Funneled: Mult threads, only master calls MPI
     - MPI_Thread_Serialized: Mult threads each calling MPI, but they do it one at a time.
     - MPI_Thread_Multiple: Multiple threads without any restrictions
   - Request and test thread modes with the function:

     MPI_init_thread(desired_mode, delivered_mode, ierr)
2. Environment variables are not propagated by mpirun. You'll need to broadcast OpenMP parameters and set them with the library routines.

Dangerous Mixing of MPI and OpenMP

- The following will work only if MPI_Thread_Multiple is supported … a level of support I wouldn't depend on.

```
MPI_Comm_Rank(MPI_COMM_WORLD, &mpi_id) ;
#pragma omp parallel
{
    int tag, swap_neigh, stat, omp_id = omp_thread_num();
    long buffer [BUFF_SIZE], incoming [BUFF_SIZE];
    big_ugly_calc1(omp_id, mpi_id, buffer);
                                                    // Finds MPI id and tag so
    neighbor(omp_id, mpi_id, &swap_neigh, &tag);  // messages don't conflict

    MPI_Send (buffer,   BUFF_SIZE, MPI_LONG, swap_neigh,
              tag, MPI_COMM_WORLD);
    MPI_Recv (incoming, buffer_count, MPI_LONG, swap_neigh,
              tag,  MPI_COMM_WORLD, &stat);

    big_ugly_calc2(omp_id, mpi_id, incoming, buffer);
#pragma critical
   consume(buffer, omp_id, mpi_id);
}
```

Messages and threads

- Keep message passing and threaded sections of your program separate:
  - Setup message passing outside OpenMP parallel regions (MPI_Thread_funneled)
  - Surround with appropriate directives (e.g. critical section or master) (MPI_Thread_Serialized)
  - For certain applications depending on how it is designed it may not matter which thread handles a message. (MPI_Thread_Multiple)
    - Beware of race conditions though if two threads are probing on the same message and then racing to receive it.

Safe Mixing of MPI and OpenMP

Put MPI in sequential regions

```
MPI_Init(&argc, &argv) ;      MPI_Comm_Rank(MPI_COMM_WORLD, &mpi_id) ;

// a whole bunch of initializations

#pragma omp parallel for
for (I=0;I<N;I++) {
    U[I] =  big_calc(I);
}

    MPI_Send (U,   BUFF_SIZE, MPI_DOUBLE, swap_neigh,
                tag, MPI_COMM_WORLD);
    MPI_Recv (incoming, buffer_count, MPI_DOUBLE, swap_neigh,
                tag,  MPI_COMM_WORLD, &stat);

#pragma omp parallel for
for (I=0;I<N;I++) {
    U[I] =  other_big_calc(I, incoming);
}

consume(U, mpi_id);
```

Technically Requires MPI_Thread_funneled, but I have never had a problem with this approach … even with pre-MPI-2.0 libraries.

Safe Mixing of MPI and OpenMP

Protect MPI calls inside a parallel region

```
MPI_Init(&argc, &argv) ;      MPI_Comm_Rank(MPI_COMM_WORLD, &mpi_id) ;

// a whole bunch of initializations

#pragma omp parallel
{
#pragma omp for
   for (I=0;I<N;I++)    U[I] =  big_calc(I);

#pragma master
{
    MPI_Send (U,   BUFF_SIZE, MPI_DOUBLE, neigh, tag,  MPI_COMM_WORLD);
    MPI_Recv (incoming, count, MPI_DOUBLE, neigh,  tag,  MPI_COMM_WORLD,
                                                                  &stat);
}
#pragma omp barrier
#pragma omp for
   for (I=0;I<N;I++)   U[I] =  other_big_calc(I, incoming);

#pragma omp master
   consume(U, mpi_id);
}
```

Technically Requires MPI_Thread_funneled, but I have never had a problem with this approach … even with pre-MPI-2.0 libraries.

Hybrid OpenMP/MPI works, but is it worth it?

- Literature* is mixed on the hybrid model: sometimes its better, sometimes MPI alone is best.
- There is potential for benefit to the hybrid model
  - MPI algorithms often require replicated data making them less memory efficient.
  - Fewer total MPI communicating agents means fewer messages and less overhead from message conflicts.
  - Algorithms with good cache efficiency should benefit from shared caches of multi-threaded programs.
  - The model maps perfectly with clusters of SMP nodes.
- But really, it's a case by case basis and to large extent depends on the particular application.

*L. Adhianto and Chapman, 2007

Appendices

- Sources for Additional information
- OpenMP History
- Solutions to exercises
  - Exercise 1: hello world
  - Exercise 2: Simple SPMD Pi program
  - Exercise 3: SPMD Pi without false sharing
  - Exercise 4: Loop level Pi
  - Exercise 5: Mandelbrot Set area
  - Exercise 6: Recursive pi program
- Challenge Problems
  - Challenge 1: molecular dynamics
  - Challenge 2: Monte Carlo Pi and random numbers
  - Challenge 3: Matrix multiplication
  - Challenge 4: linked lists
  - Challenge 5: Recursive matrix multiplication
- Fortran and OpenMP
- Mixing OpenMP and MPI
- Compiler Notes

Compiler notes: Intel on Windows

- Intel compiler:
  - Launch SW dev environment … on my laptop I use:
    - start/intel software development tools/intel C++ compiler 11.0/C+ build environment for 32 bit apps
  - cd to the directory that holds your source code
  - Build software for program foo.c
    - icl /Qopenmp foo.c
  - Set number of threads environment variable
    - set OMP_NUM_THREADS=4
  - Run your program
    - foo.exe

To get rid of the pwd on the prompt, type

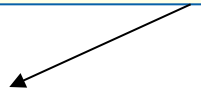
prompt = %

Compiler notes: Visual Studio

- Start “new project”
- Select win 32 console project
  - Set name and path
  - On the next panel, Click “next” instead of finish so you can select an empty project on the following panel.
  - Drag and drop your source file into the source folder on the visual studio solution explorer
  - Activate OpenMP
    - Go to project properties/configuration properties/C.C++/language … and activate OpenMP
- Set number of threads inside the program
- Build the project
- Run “without debug” from the debug menu.

Compiler notes: Other

for the Bash shell



- Linux and OS X with gcc:
  - > gcc -fopenmp foo.c
  - > export OMP_NUM_THREADS=4
  - > ./a.out
- Linux and OS X with PGI:
  - > pgcc -mp foo.c
  - > export OMP_NUM_THREADS=4
  - > ./a.out

OpenMP constructs

Where variable list is a comma separated list of variables

- #pragma omp parallel
- #pragma omp for
- #pragma omp critical
- #pragma omp atomic
- #pragma omp barrier
- Data environment clauses
  - private (variable_list)
  - firstprivate (variable_list)
  - lastprivate (variable_list)
  - reduction(+:variable_list)

Print the value of the macro

_OPENMP

And its value will be

yyyymm

For the year and month of the spec the implementation used

- Tasks (remember … private data is made firstprivate by default)
  - pragma omp task
  - pragma omp taskwait
- #pragma threadprivate(variable_list)

OMP Construct	Concepts
#pragma omp parallel	parallel region, teams structured block, interleaved execution across threads
omp_get_wtime()	Speedup, Amdahl's law and other performance issues
omp_get_num_threads()	The SPMD Pattern. False sharing
omp_get_thread_num	
Barrier	synchronization and race conditions. Revisit interleaved execution. Teach the concept of flush but not the construct.
Critical	
#pragma omp for	worksharing, parallel loops, loop carried dependencies
schedule(dynamic [,chunk])	Loop schedules, loop overheads and load balance
schedule (static [,chunk])	
private	Data environment
firstprivate	
shared	
reduction	
nowait	synchronization overhead, the high cost of barriers
#pragma omp single	
OMP_NUM_THREADS	internal control variables
Features just beyond the common core	
atomic, flush, locks	Details on synchronization including spin locks and locking protocals
#pragma omp task	tasks including the data environment for tasks.
OMP_PLACES	NUMA issues
OMP_Proc_BIND	

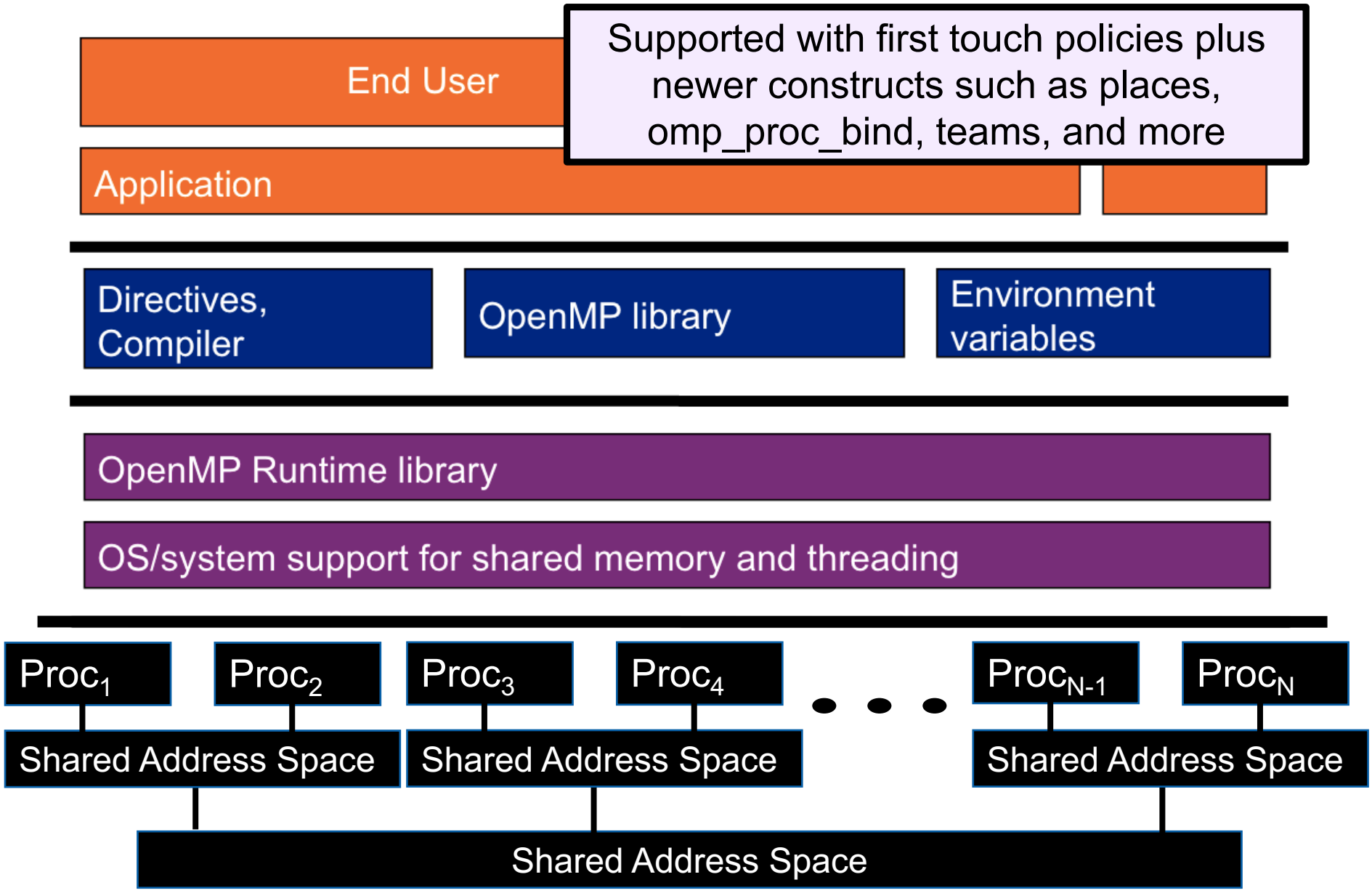
OMP Construct	Concepts	Exercises
#pragma omp parallel	parallel region, teams structured block, interleaved execution across threads	hello world
int omp_get_thread_num() int omp_get_num_threads()	The SPMD Pattern.	pi SPMD
double omp_get_wtime()	Speedup and Amdahl's law. False Sharing and other performance issues	
setenv OMP_NUM_THREADS N	internal control variables	pi SPMD on many threads
#pragma omp barrier #pragma omp critical	Synchronization and race conditions. Revisit interleaved execution. Teach the concept of flush but not the construct.	PI SPMD without sum array
#pragma omp for	worksharing, parallel loops, loop carried dependencies	pi par loop
reduction(op:list)	reductions of values across a team of threads	pi par loop
schedule(dynamic [,chunk]) schedule (static [,chunk])	Loop schedules, loop overheads and load balance	
private(list), firstprivate(list), shared(list)	Data environment	Mandelbrot area
nowait	synchronization overhead, the high cost of barriers	
#pragma omp single		
#pragma omp task #pragma omp taskwait	tasks including the data environment for tasks.	linked lists

Advertised Plan for the course

(in four units ranging from 90 minutes to 2 hours)

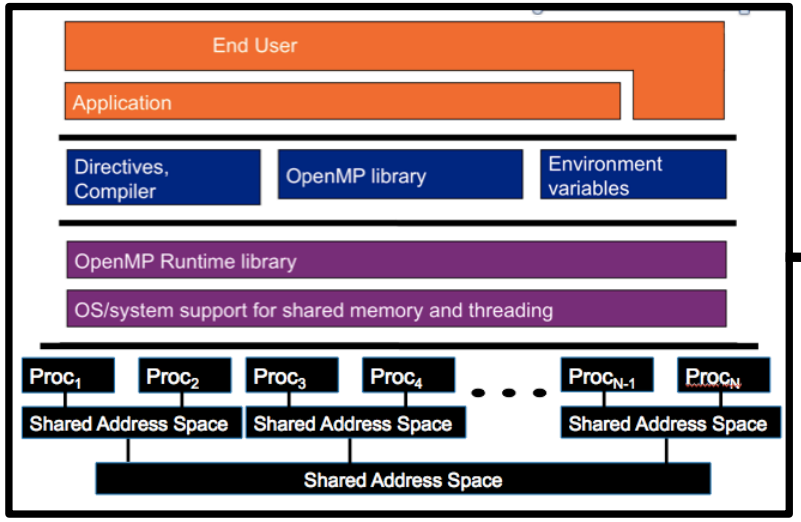
- An introduction to parallel computing with OpenMP.
  - We start with a discussion of the historical roots of parallel computing and how they appear in a modern context. We'll then use OpenMP and a series of hands-on exercises to explore the fundamental concepts behind parallel programming.
- The OpenMP Common Core
  - We will explore through hands-on exercises the common core of OpenMP; that is, the features of the API that most OpenMP programmers use in all their parallel programs. This will provide a foundation of understanding you can build on as you explore the more advanced features of OpenMP
- Working with OpenMP
  - We'll explore more complex OpenMP problems and get a feel for how to work with OpenMP with real applications.
- The world beyond OpenMP
  - Parallel programming is hard. There is no way to avoid that reality. We can mitigate these difficulties by focusing on the fundamental design patterns from which most parallel algorithms are constructed. Once mastered, these patterns make it much easier to understand how your problems map onto other parallel programming models. Hence for our last session on parallel programming, we'll review these essential design patterns as seen in OpenMP, and then show how they appear in cluster computing (with MPI) and GPGPU computing (with OpenCL … and a bit of CUDA).

OpenMP basic definitions: NUMA Solution stack

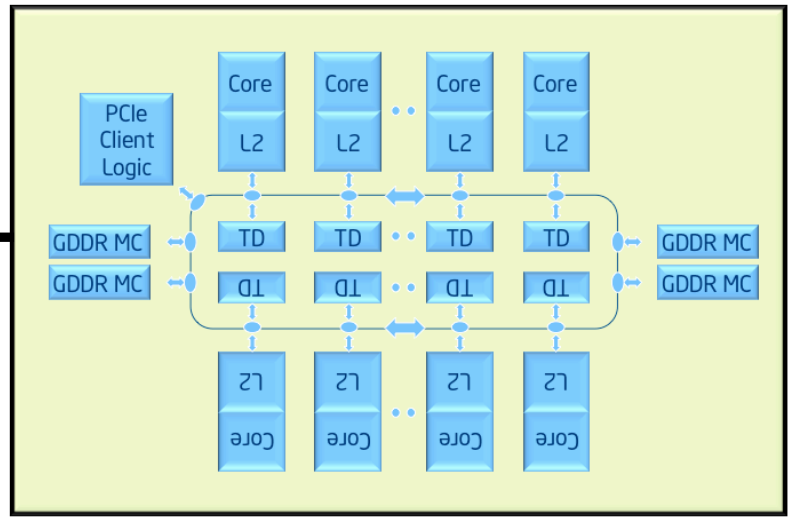


OpenMP basic definitions: Target solution stack

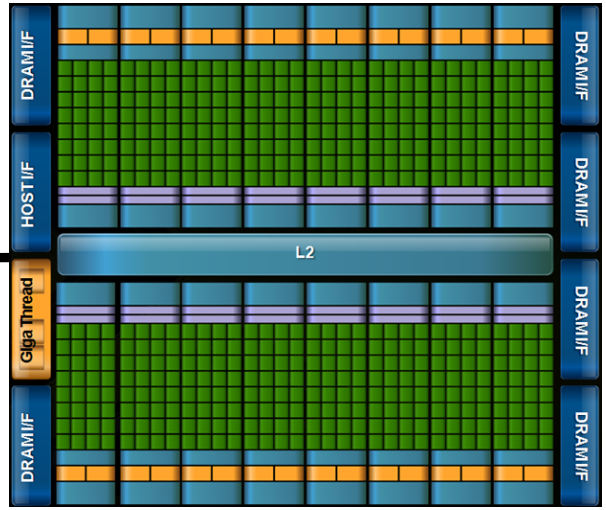
Supported (since OpenMP 4.0) with target, teams, distribute, and other constructs



Host



Target Device: Intel® Xeon Phi™ coprocessor



Target Device: GPU

What if the problem size grows

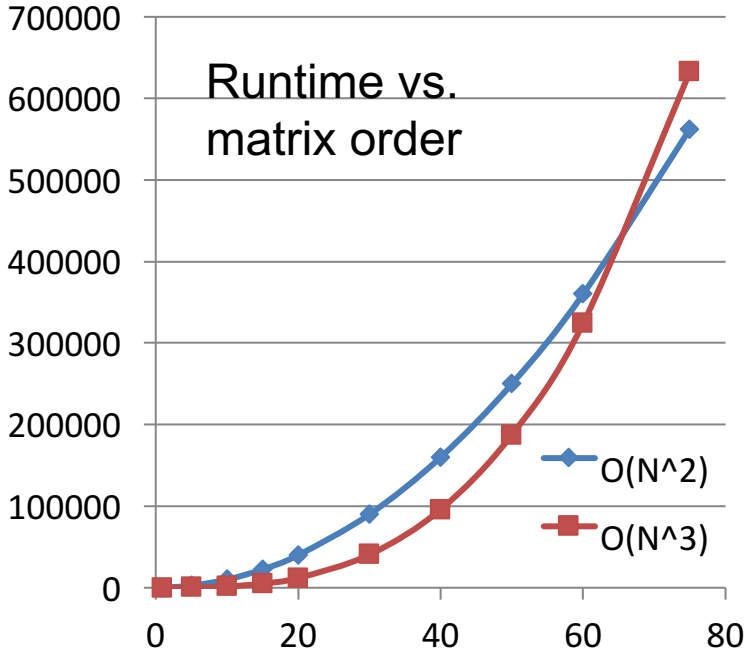
- Consider the dense linear algebra problems.
- A key feature of many of these operations between matrices (such as LU factorization or matrix multiplication) … work scales as the cube of the order of the matrix.
- Assume we can parallelize the linear algebra operation ($O(N^3)$) but not the loading of the matrices from memory ($O(N^2)$). How does the serial fraction vary with matrix order (assume loading from memory is much slower than a floating point op).

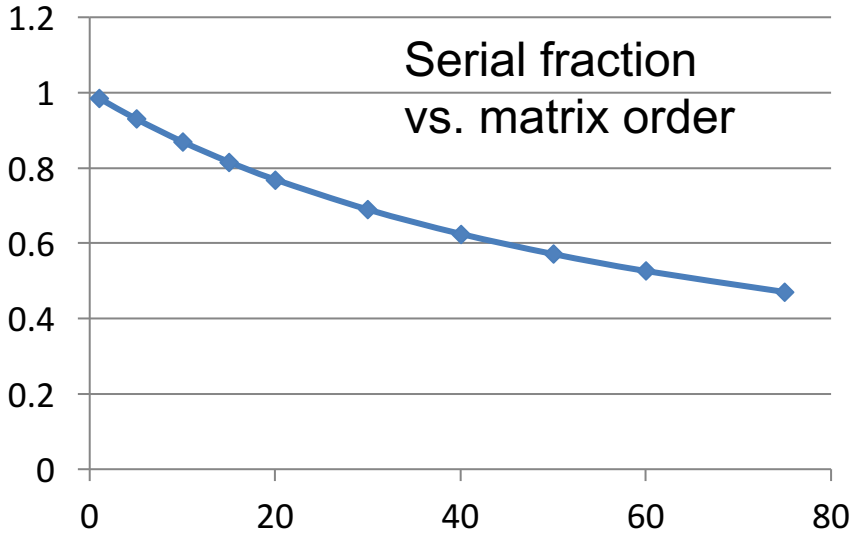
What would plots of runtime vs. problem size look like for the N squared and N cubed terms?

What would plots of serial fraction vs. problem size look like for the N squared and N cubed terms?

What if the problem size grows

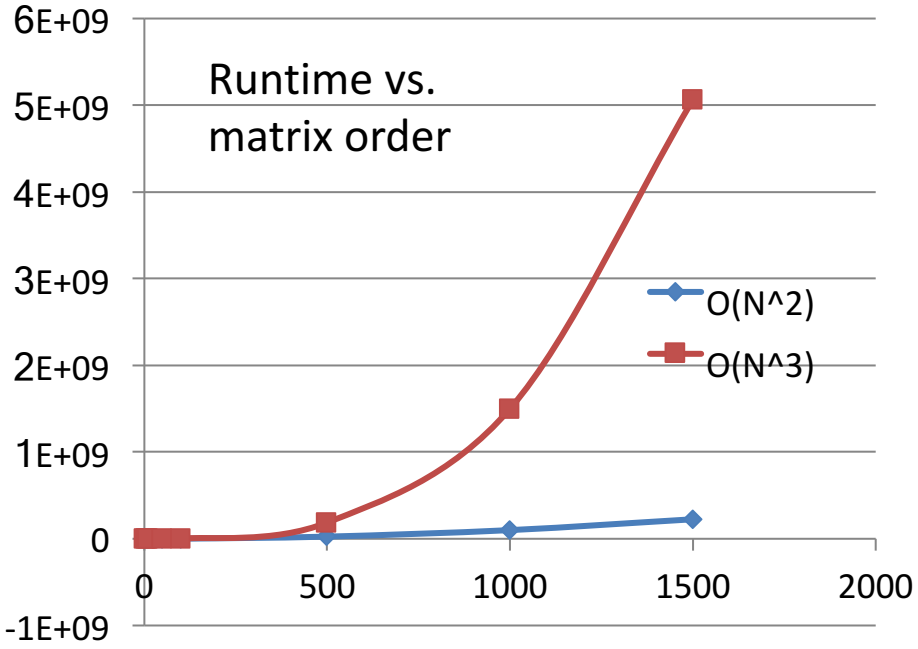
- Consider the dense linear algebra problems.
- A key feature of many of these operations between matrices (such as LU factorization or matrix multiplication) … work scales as the cube of the order of the matrix.
- Assume we can parallelize the linear algebra operation ($O(N^3)$) but not the loading of the matrices from memory ($O(N^2)$). How does the serial fraction vary with matrix order (assume loading from memory is much slower than a floating point op).

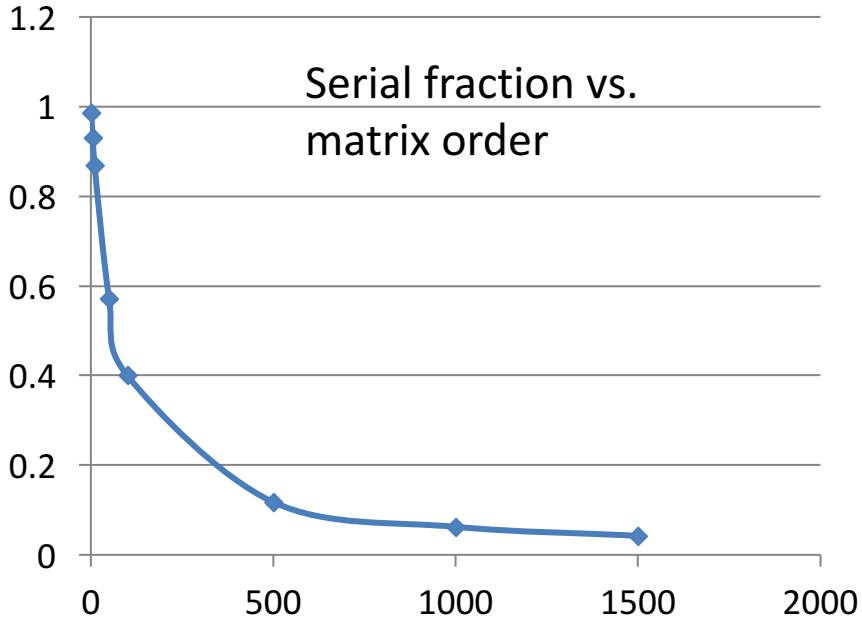




What if the problem size grows

- Consider the dense linear algebra problems.
- A key feature of many of these operations between matrices (such as LU factorization or matrix multiplication) … work scales as the cube of the order of the matrix.
- Assume we can parallelize the linear algebra operation ($O(N^3)$) but not the loading of the matrices from memory ($O(N^2)$). How does the serial fraction vary with matrix order (assume loading from memory is much slower than a floating point op).





For much larger Matrix orders …

Weak Scaling: a response to Amdhal

- Gary Montry and John Gustafson (1988, Sandia National Laboratories) observed that for many problems the serial fraction of a function of the problem size (N) decreases:

$$S(P,N) = \frac{T_{seq}(1)}{(\alpha(N) + \frac{1-\alpha(N)}{P}) * T_{seq}(1)} \qquad \lim_{N \to N_{large}} \alpha(N) = 0$$

$$S(P, N_{large}) \to P$$

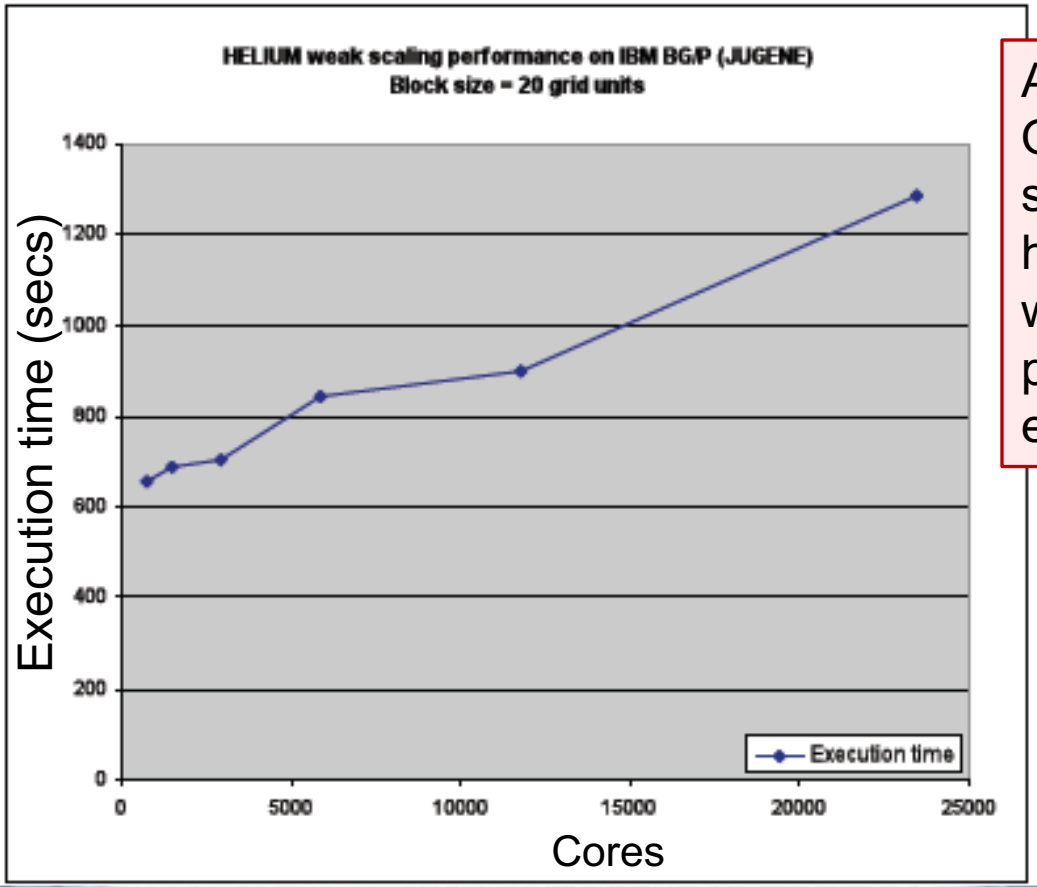
- In other words … if parallelizable computations asymptotically dominate the runtime, then you can increase a problem size until limitations due to Amdahl's law can be ignored. This is an easier form of scalability for a programmer to meet … so its called "weak scaling":
  - **Weak Scaling:** Performance of an application when the problem size increases with the number of processors (fixed size problem per node)

Example of weak scaling

HELIUM Weak Scaling Performance on BG/P

epcc

- Local block size fixed to 20 grid units



A time dependent Quantum simulation of helium atoms with 20 grid units per processing element.

IBM Blue Gene P, 0.85 GHz, PowerPC 450, 4-way processors

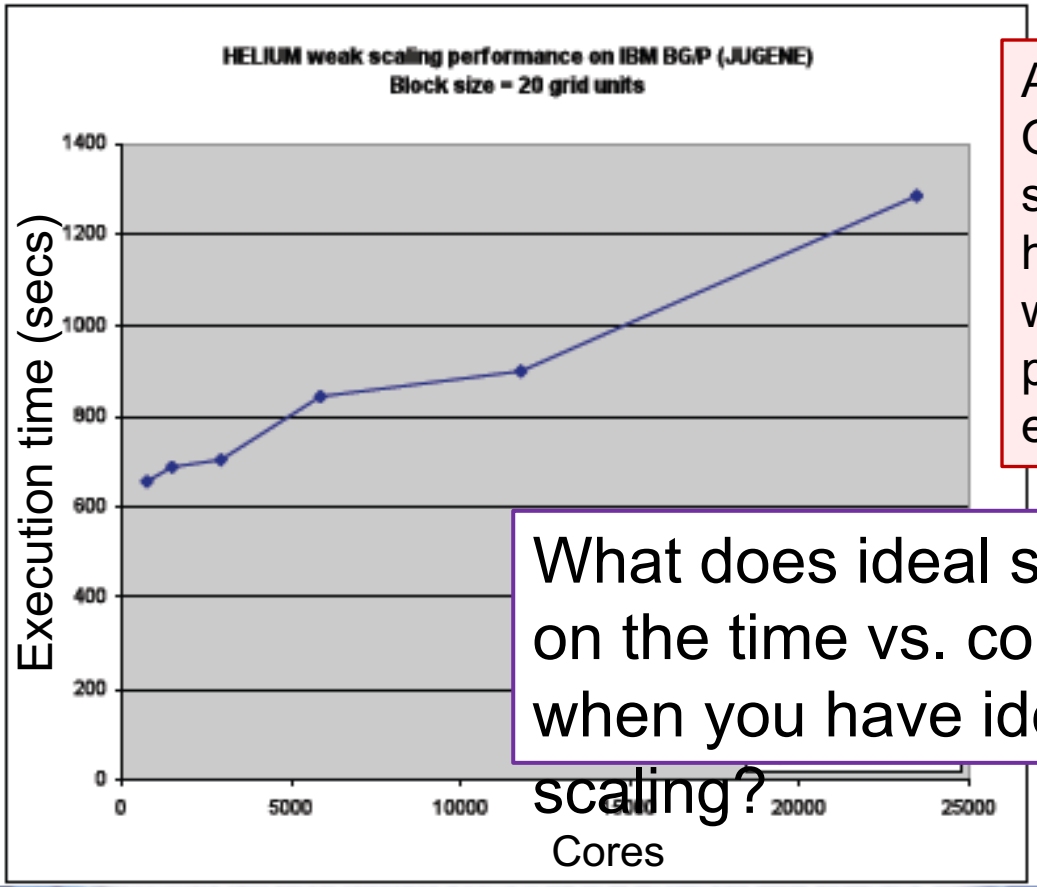
http://www.spscicomp.org/ScicomP16/presentations/PRACE_ScicomP.pdf

Example of weak scaling

HELIUM Weak Scaling Performance on BG/P

epcc

- Local block size fixed to 20 grid units



A time dependent Quantum simulation of helium atoms with 20 grid units per processing element.

What does ideal scaling look on the time vs. cores plot when you have ideal weak scaling?

IBM Blue Gene P, 0.85 GHz, PowerPC 450, 4-way processors

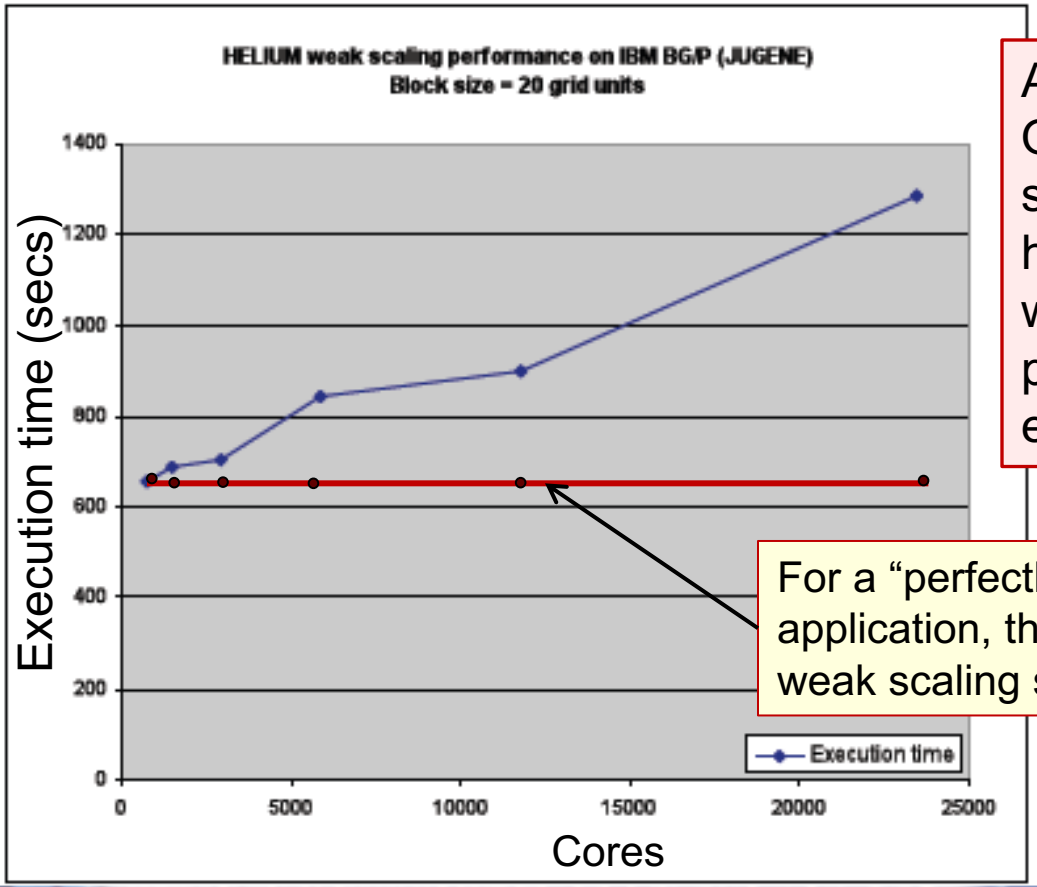
http://www.spscicomp.org/ScicomP16/presentations/PRACE_ScicomP.pdf

Example of weak scaling

HELIUM Weak Scaling Performance on BG/P

epcc

- Local block size fixed to 20 grid units



A time dependent Quantum simulation of helium atoms with 20 grid units per processing element.

For a "perfectly scalable" application, the trend line for weak scaling should be flat.

IBM Blue Gene P, 0.85 GHz, PowerPC 450, 4-way processors

http://www.spscicomp.org/ScicomP16/presentations/PRACE_ScicomP.pdf