
# A “Hands-on” Introduction to OpenMP*

**Tim Mattson**

**Intel Corp.**

**timothy.g.mattson@intel.com**

Acknowledgements: J. Mark Bull (EPCC), Mike Pearce (Intel), Larry Meadows (Intel), Barbara Chapman (SBU), Bronis de Supinski (LLNL), and many others have contributed to these slides over the years.





1

# Disclaimer & Optimization Notice



Performance tests and ratings are measured using specific computer systems and/or components and reflect the approximate performance of Intel products as measured by those tests. Any difference in system hardware or software design or configuration may affect actual performance. Buyers should consult other sources of information to evaluate the performance of systems or components they are considering purchasing. For more information on performance tests and on the performance of Intel products, reference www.intel.com/software/products.



## Optimization Notice

Intel’s compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

# Preliminaries: Systems for exercises

- Blue Gene
  ssh <<login_name>>@vesta.aclf.anl.gov
- The OpenMP compiler
  Uncomment the line in .soft then run the resoft command
  +mpiwrapper-xl
  xlc++_r –qsmp=omp  << file names>>

- X86 cluster
  ssh <<login_name>>@cooley.aclf.anl.gov
- The OpenMP compiler
  Add the line to ".soft.cooley" and then run the resoft command
  +intel-composer-xe
  icc –qopenmp –O3 << file names>>

Use either system or even your laptop if you wish

- Copy the exercises to your home directory
  $ cp /projects/ATPESC2016/openmp
- You can just run on the login nodes or use qsub (to get good timing numbers)
- To get a single node for 30 minutes in interactive mode
  qsub –A ATPESC2016 –n 1 –t 30 -Ik

# 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.

# Plan

| | Module | Concepts | Exercises | |
|---|---|---|---|---|
| 8:30 | OpenMP core concepts | • Intro to OpenMP<br>• Creating threads | • Hello_world<br>• Pi_spmd | |
| 10:30 | Working with threads | • Synchronization<br>• Parallel loops<br>• Single, master, and more | • Pi_spmd_final<br>• Pi_loop | 10 AM Break |
| 1:00 | Managing data and tasks | • Data Environment<br>• tasks | • Mandelbrot set area<br>• Racy tasks<br>• Recursive pi | Noon Lunch |
| 3:30 | Understanding shared memory | • Memory Model<br>• Threadprivate | • Monte Carlo pi | 3 PM Break |
| | OpenMP beyond SMP | • SIMD<br>• Devices and OpenMP | • Jaobi Solver | |

… Plus a set of “challenge problems” for the evening program.

# Plan

| | Module | Concepts | Exercises | |
|---|---|---|---|---|
| 8:30 | OpenMP core concepts | • Intro to OpenMP<br>• Creating threads | • Hello_world<br>• Pi_spmd | |
| 10:30 | Working with threads | • Synchronization<br>• Parallel loops<br>• Single, master, and more | • Pi_spmd_final<br>• Pi_loop | 10 AM Break |
| 1:00 | Managing data and tasks | • Data Environment<br>• tasks | • Mandelbrot set area<br>• Racy tasks<br>• Recursive pi | Noon Lunch |
| 3:30 | Understanding shared memory | • Memory Model<br>• Threadprivate | • Monte Carlo pi | 3 PM Break |
| | OpenMP beyond SMP | • SIMD<br>• Devices and OpenMP | • Jaobi Solver | |

… Plus a set of “challenge problems” for the evening program.

# 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()

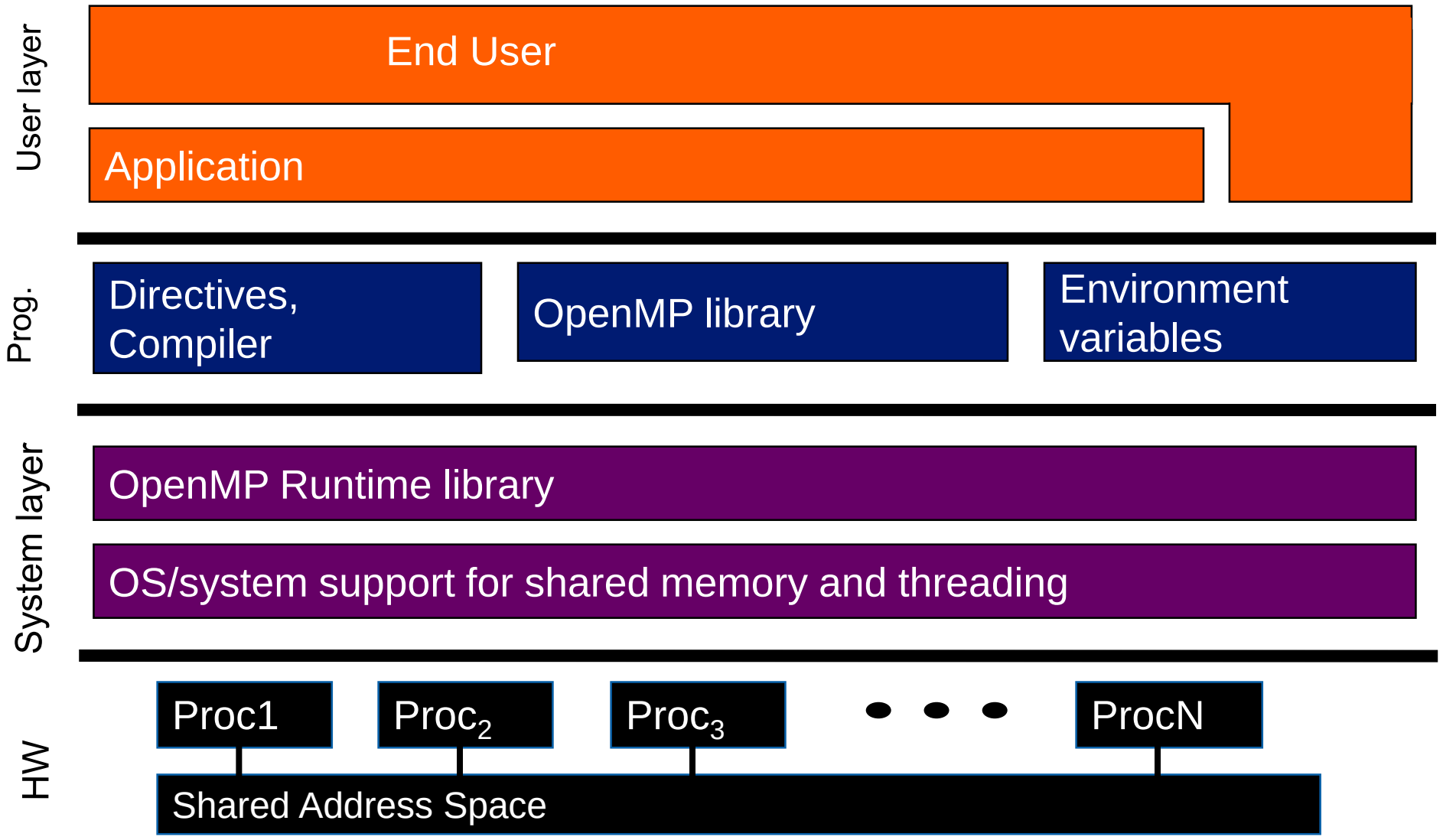
omp_set_lock(lck)

## *OpenMP: An API for Writing Multithreaded Applications*

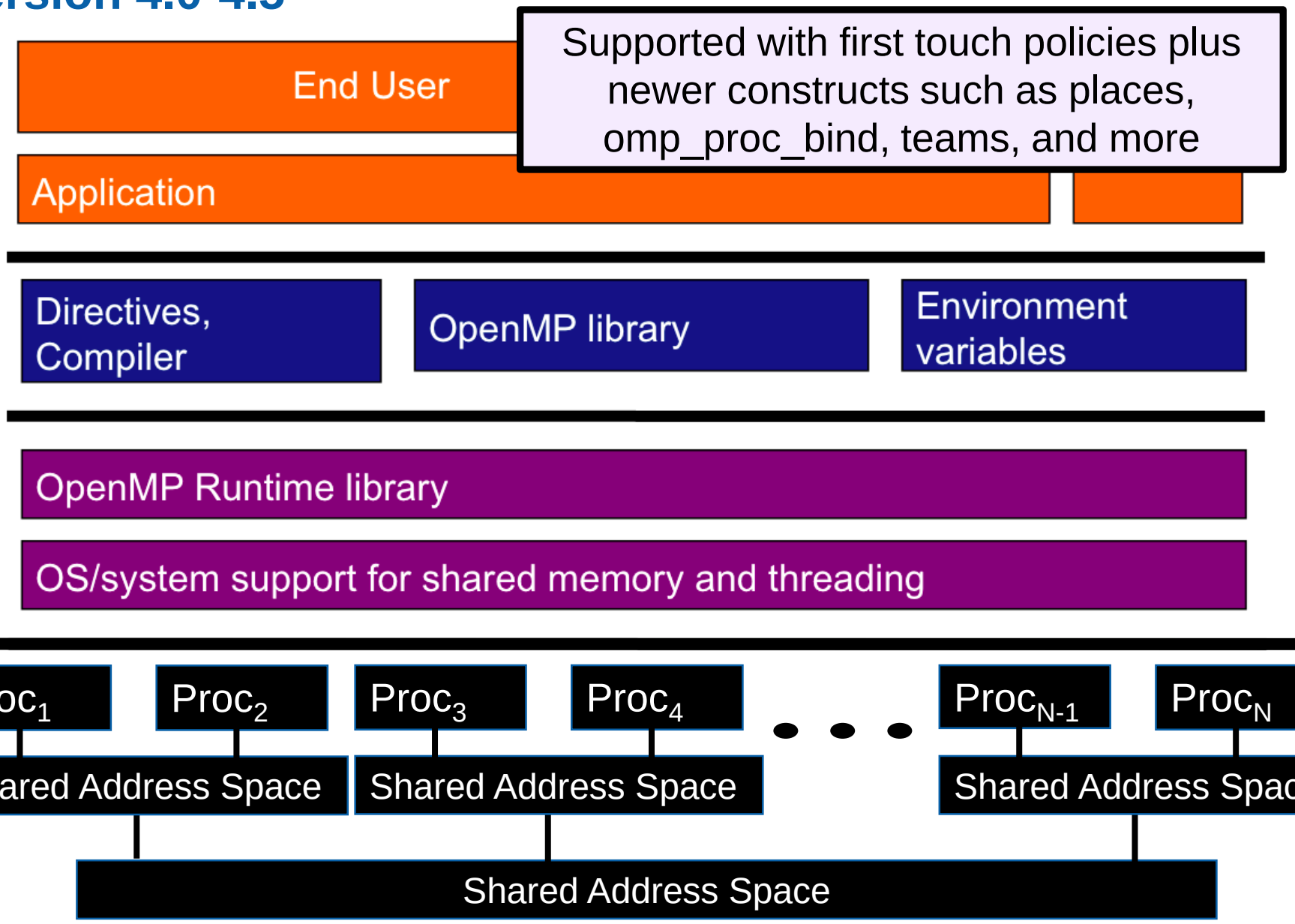
- A set of compiler directives and library routines for parallel application programmers
- Greatly simplifies writing multi-threaded (MT) programs in Fortran, C and C++
- Standardizes established SMP practice + vectorization and heterogeneous device programming

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

# OpenMP basic definitions: Basic Solution stack
## Versions 1.0 to 3.1

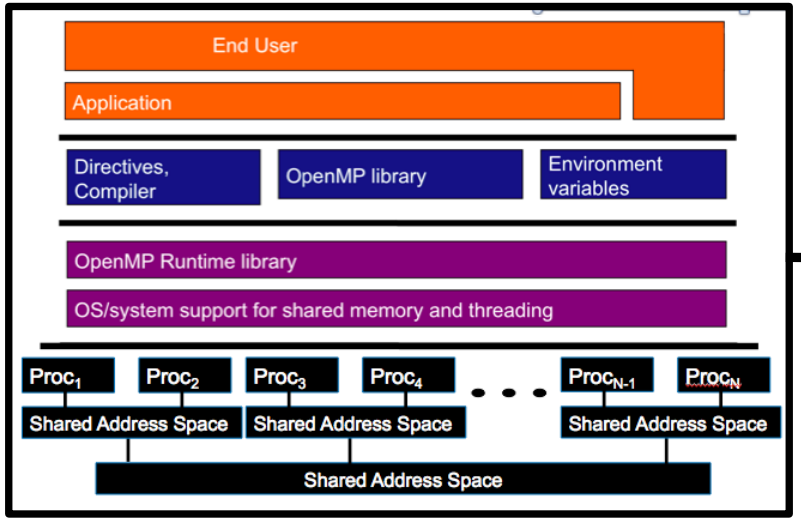


# OpenMP basic definitions: NUMA Solution stack
## Version 4.0-4.5

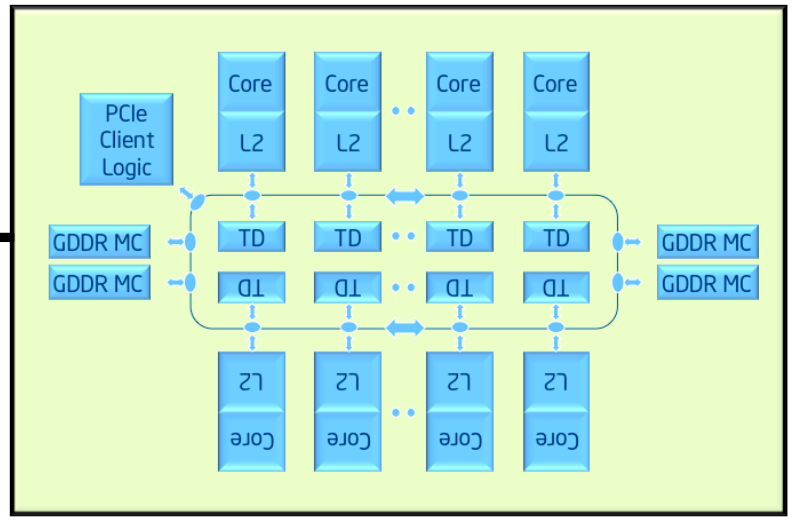


# OpenMP basic definitions: Target solution stack
## Version 4.0-4.5

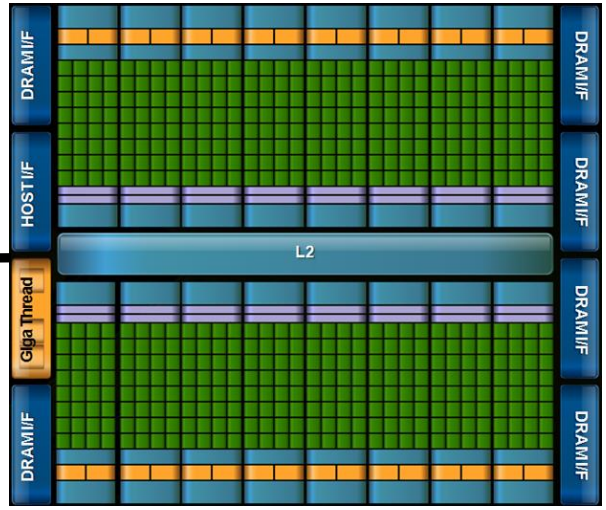
Supported (since OpenMP 4.0) with target, teams, distribute, and other constructs




Host




Target Device: Intel® Xeon Phi™ coprocessor




Target Device: GPU

# OpenMP core syntax

- Most of the constructs in OpenMP are compiler directives.

  ***#pragma omp construct [clause [clause]…]***

  - Example

    ***#pragma omp parallel num_threads(4)***

- Function prototypes and types in the file:

  ***#include <omp.h>***

  ***use omp_lib***

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

# Exercise 1, Part A: Hello world

## Verify that your environment works

- Write a program that prints “hello world”.

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



    int ID = 0;

    printf(“ hello(%d) ”, ID);
    printf(“ world(%d) \n”, ID);

}
```

# 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
 {
    int ID = 0;
    printf(“ hello(%d) ”, ID);
    printf(“ world(%d) \n”, ID);
 }
}
```

**Switches for compiling and linking**

**gcc -fopenmp  Linux, OSX**

**pgcc -mp  pgi**

**icl /Qopenmp  intel (windows)**

**icc –qopenmp  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
 {

    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

Runtime library function to return a thread ID.

End of the Parallel region

Sample Output:

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

world(0)

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

world(2)

# OpenMP overview:
## How do threads interact?

- OpenMP is a multi-threading, shared address model
  - Threads communicate by sharing variables.

- Unintended sharing of data causes race conditions:
  - Race condition: when the program's outcome changes as the threads are scheduled differently.

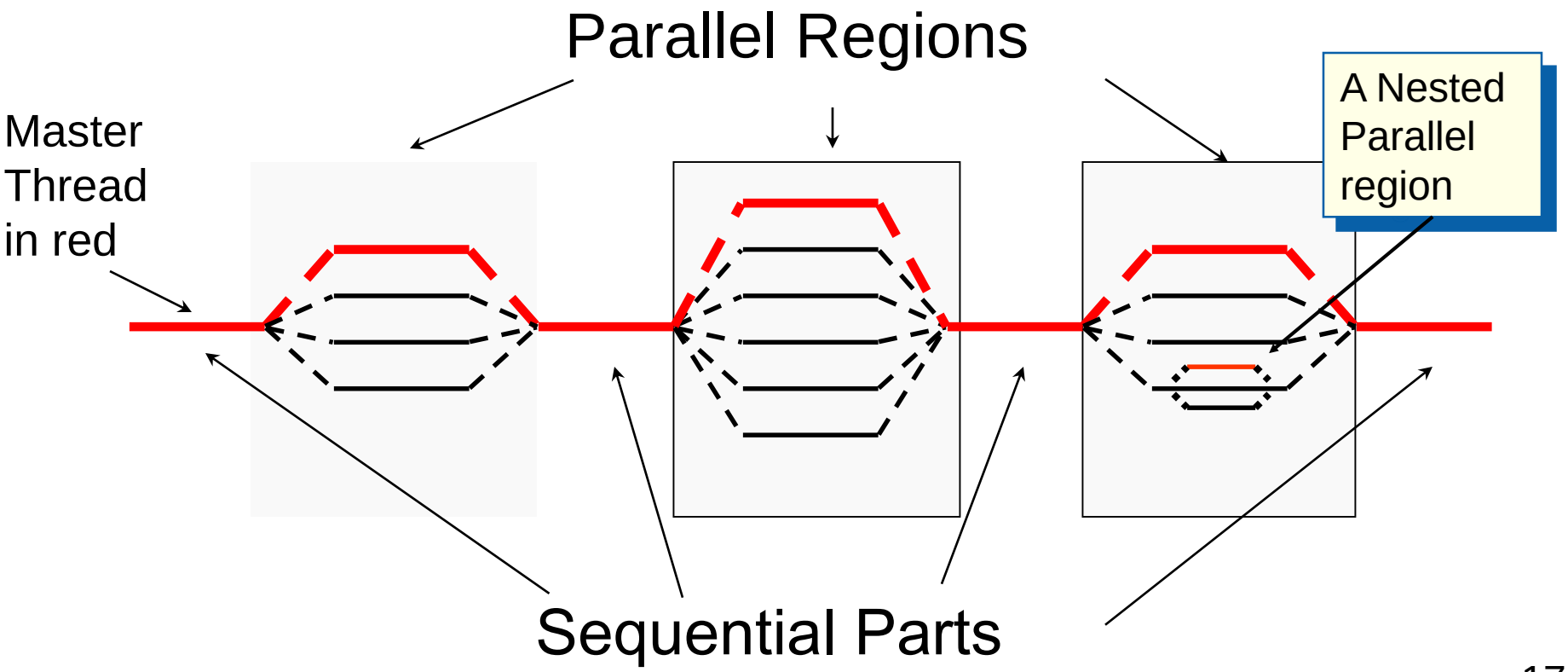
- To control race conditions:
  - Use synchronization to protect data conflicts.

- Synchronization is expensive so:
  - Change how data is accessed to minimize the need for synchronization

# 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

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

```
double A[1000];

#pragma omp parallel num_threads(4)
{
        int ID = omp_get_thread_num();
        pooh(ID,A);
}
```

clause to request a certain number of threads

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

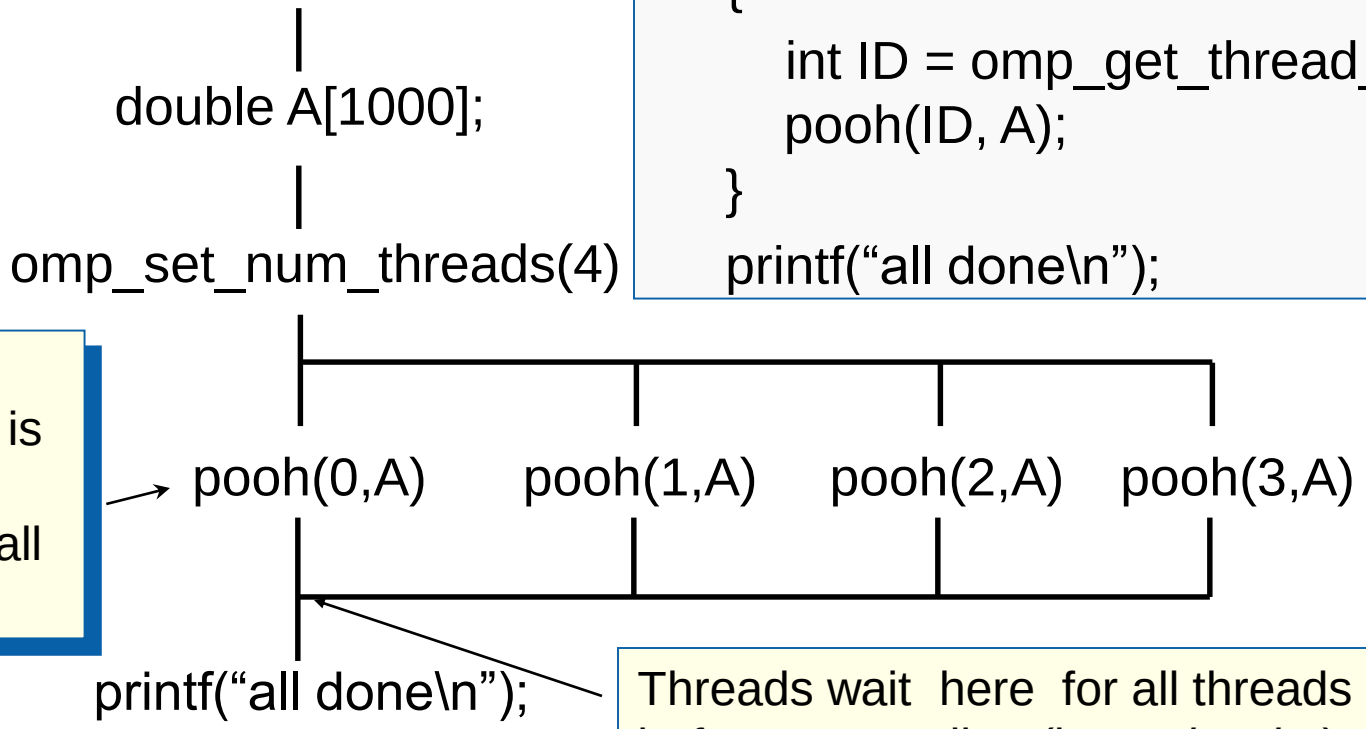
Runtime function returning a thread ID

- 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");
```

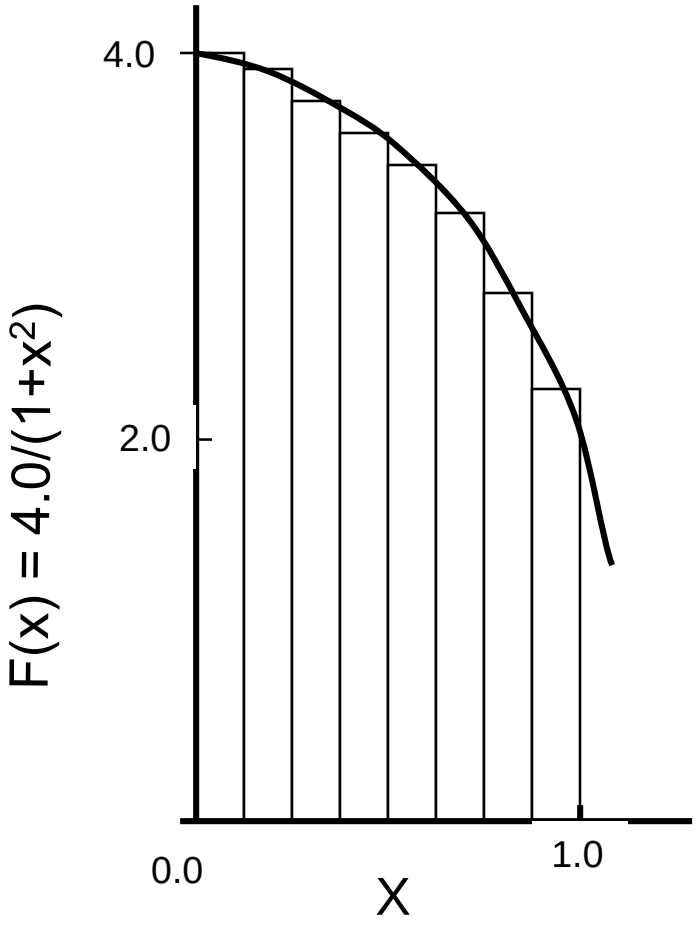



A single copy of A is shared between all threads.

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

# Exercises 2-4,6:

## 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 $\Delta x$ and height $F(x_i)$ at the middle of interval i.

# Exercises 2-4,6: 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

# 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*: The SPMD pattern

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

## Example: A simple Parallel pi program

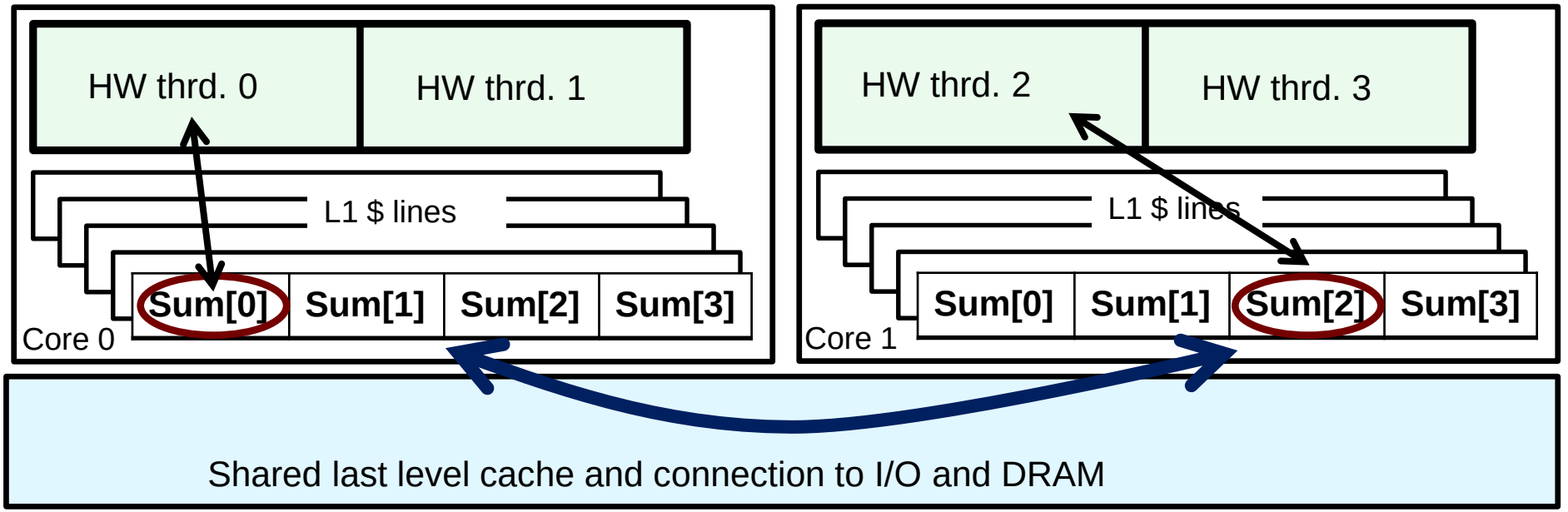
```
#include <omp.h>
static long num_steps = 100000;        double step;
#define NUM_THREADS 2
void main ()
{          int i, nthreads;  double pi, sum[NUM_THREADS];
           step = 1.0/(double) num_steps;
           omp_set_num_threads(NUM_THREADS);
   #pragma omp parallel
   {
          int i, id,nthrds;
          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 (icc) 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.

# Plan

| | Module | Concepts | Exercises | |
|---|---|---|---|---|
| 8:30 | OpenMP core concepts | • Intro to OpenMP<br>• Creating threads | • Hello_world<br>• Pi_spmd | |
| 10:30 | Working with threads | • Synchronization<br>• Parallel loops<br>• Single, master, and more | • Pi_spmd_final<br>• Pi_loop | 10 AM Break |
| 1:00 | Managing data and tasks | • Data Environment<br>• tasks | • Mandelbrot set area<br>• Racy tasks<br>• Recursive pi | Noon Lunch |
| 3:30 | Understanding shared memory | • Memory Model<br>• Threadprivate | • Monte Carlo pi | 3 PM Break |
| | OpenMP beyond SMP | • SIMD<br>• Devices and OpenMP | • Jaobi Solver | |

… Plus a set of “challenge problems” for the evening program.

# Synchronization

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

- High level synchronization:
  - critical
  - atomic
  - barrier
  - ordered
- Low level synchronization
  - flush
  - locks (both simple and nested)

Discussed later

# 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);
      }
}
```



hints were added to critical in OpenMP 4.5 to suggest a locking  strategy based on intended use  of the critical construct (e.g. contended, uncontended, speculative,, unspeculative)

# 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)

# Exercise 3

- 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?

# Example: Using an atomic 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);
              }
               sum = sum*step;
          #pragma omp atomic
                    pi += sum ;
}
}
```

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 so updates don’t conflict

# Plan

| | Module | Concepts | Exercises | |
|---|---|---|---|---|
| 8:30 | OpenMP core concepts | • Intro to OpenMP<br>• Creating threads | • Hello_world<br>• Pi_spmd | |
| 10:30 | Working with threads | • Synchronization<br>• Parallel loops<br>• Single, master, and more | • Pi_spmd_final<br>• Pi_loop | 10 AM Break |
| 1:00 | Managing data and tasks | • Data Environment<br>• tasks | • Mandelbrot set area<br>• Racy tasks<br>• Recursive pi | Noon Lunch |
| 3:30 | Understanding shared memory | • Memory Model<br>• Threadprivate | • Monte Carlo pi | 3 PM Break |
| | OpenMP beyond SMP | • SIMD<br>• Devices and OpenMP | • Jaobi Solver | |

… Plus a set of “challenge problems” for the evening program.

# Alternatives to SPMD

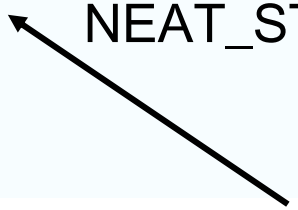
- A parallel construct by itself creates an SPMD or "Single Program Multiple Data" program … i.e., each thread redundantly executes the same code.
- How do you split up pathways through the code between threads within a team?
  - Worksharing constructs
    - Loop construct
    - Sections/section constructs
    - Single construct
  - Task constructs

Discussed later

# 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(guided[,chunk])
    - Threads dynamically grab blocks of iterations. The size of the block starts large and shrinks down to size “chunk” as the calculation proceeds.
  - schedule(runtime)
    - Schedule and chunk size taken from the OMP_SCHEDULE environment variable (or the runtime library).
  - schedule(auto)
    - Schedule is left up to the runtime to choose (does not have to be any of the above).

OpenMP 4.5 added modifiers monotonic, nonmontonic and simd.

# loop work-sharing constructs:
## The schedule clause

| Schedule Clause | When To Use |
|---|---|
| **STATIC** | **Pre-determined and predictable by the programmer** |
| **DYNAMIC** | **Unpredictable, highly variable work per iteration** |
| **GUIDED** | **Special case of dynamic to reduce scheduling overhead** |
| **AUTO** | **When the runtime can "learn" from previous executions of the same loop** |

Least work at runtime : scheduling done at compile-time

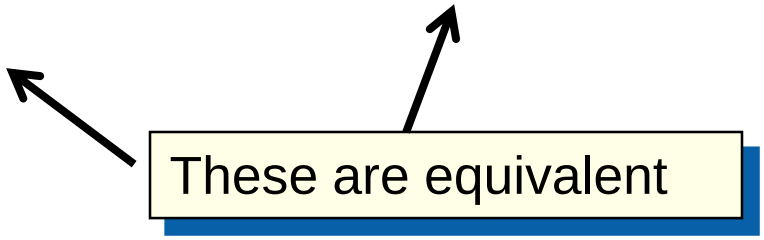
Most work at runtime : complex scheduling logic used at run-time

# Combined parallel/worksharing construct

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

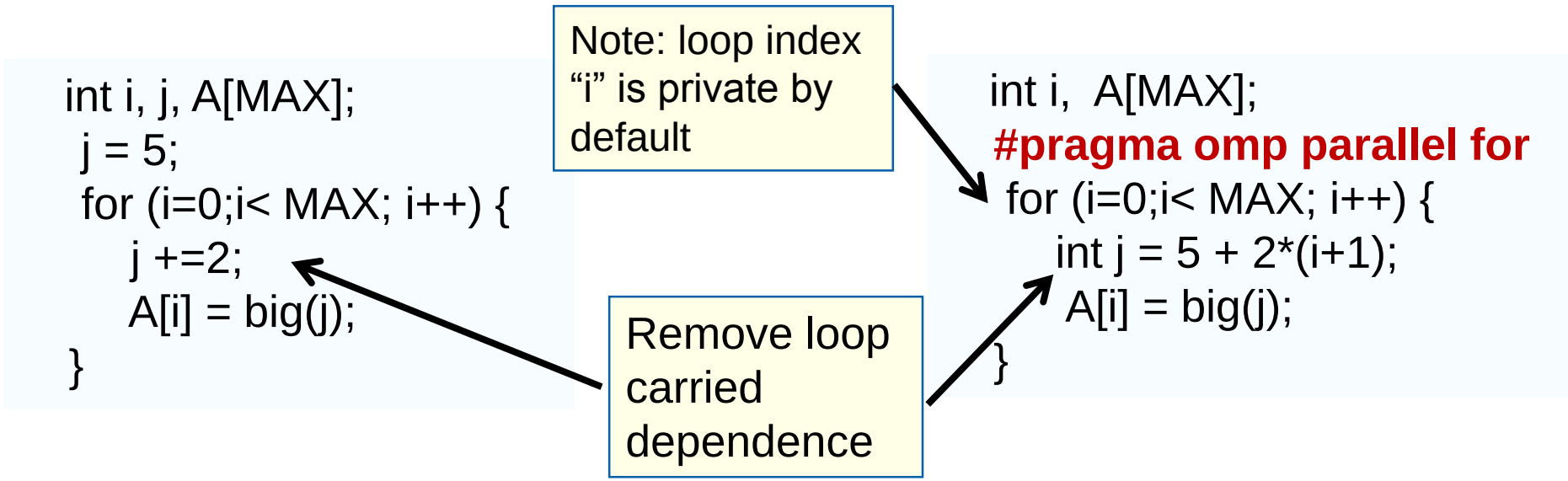
```
 double  res[MAX];  int i;
#pragma omp parallel
{
    #pragma omp for
    for (i=0;i< MAX; i++) {
        res[i] = huge();
    }
}
```

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



# Working with loops

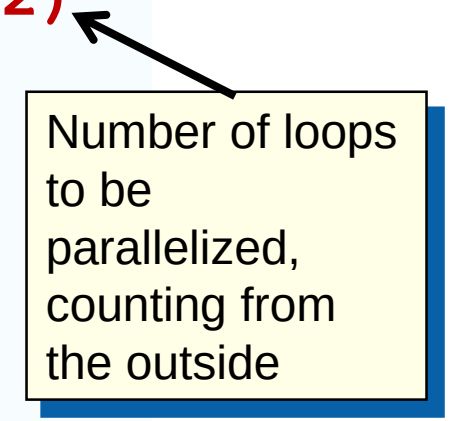
- Basic approach
  - Find compute intensive loops
  - Make the loop iterations independent ... So they can safely execute in any order without loop-carried dependencies
  - Place the appropriate OpenMP directive and test



# Nested loops

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

```
#pragma omp parallel for collapse(2)
for (int i=0; i<N; i++) {
  for (int j=0; j<M; j++) {
          .....
  }
}
```




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

# Reduction

- How do we handle this case?

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

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

# Reduction

- OpenMP reduction clause:

  reduction (op : list)

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

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

# OpenMP: Reduction operands/initial-values

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

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

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

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

OpenMP 4.0 added user defined reductions (discussed later).

# Exercise 4: 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.

# Plan

| | Module | Concepts | Exercises | |
|---|---|---|---|---|
| 8:30 | OpenMP core concepts | • Intro to OpenMP<br>• Creating threads | • Hello_world<br>• Pi_spmd | |
| | | | | 10 AM Break |
| 10:30 | Working with threads | • Synchronization<br>• Parallel loops<br>• Single, master, and more | • Pi_spmd_final<br>• Pi_loop | |
| | | | | Noon Lunch |
| 1:00 | Managing data and tasks | • Data Environment<br>• tasks | • Mandelbrot set area<br>• Racy tasks<br>• Recursive pi | |
| | | | | 3 PM Break |
| 3:30 | Understanding shared memory | • Memory Model<br>• Threadprivate | • Monte Carlo pi | |
| | OpenMP beyond SMP | • SIMD<br>• Devices and OpenMP | • Jaobi Solver | |

… Plus a set of “challenge problems” for the evening program.

# Synchronization: Barrier

- 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

# 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();
}
```

# Master construct

- The master construct denotes a structured block that is only executed by the master thread.
- The other threads just skip it (no synchronization is implied).

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

# Sections worksharing construct

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

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

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

# Synchronization: Lock routines

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

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

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

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

# Synchronization: Simple locks

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

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

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

One lock per element of hist

Enforce mutual exclusion on update to hist array

Free-up storage when done.

# Runtime library routines

- Runtime environment routines:
  - Modify/Check the number of threads
    - omp_set_num_threads(), omp_get_num_threads(), omp_get_thread_num(), omp_get_max_threads()
  - Are we in an active parallel region?
    - omp_in_parallel()
  - Do you want the system to vary the number of threads dynamically from one parallel construct to another?
    - omp_set_dynamic(), omp_get_dynamic();
  - How many processors in the system?
    - omp_get_num_procs()

…plus a few less commonly used routines.

# Runtime Library routines

- To use a known, fixed number of threads in a program, (1) tell the system that you don't want dynamic adjustment of the number of threads, (2) set the number of threads, then (3) save the number you got.

```
#include <omp.h>
void main()
{   int num_threads;
     omp_set_dynamic( 0 );
     omp_set_num_threads( omp_get_num_procs() );
   #pragma omp parallel
   {     int id= omp_get_thread_num();
         #pragma omp single
            num_threads = omp_get_num_threads();
         do_lots_of_stuff(id);
    }
}
```

Disable dynamic adjustment of the number of threads.

Request as many threads as you have processors.

Protect this op since Memory stores are not atomic

Even in this case, the system may give you fewer threads than requested. If the precise # of threads matters, test for it and respond accordingly.

# Environment Variables

- Set the default number of threads to use.
  - OMP_NUM_THREADS *int_literal*
- Control how “omp for schedule(RUNTIME)” loop iterations are scheduled.
  - OMP_SCHEDULE “schedule[, chunk_size]”
- Process binding is enabled if this variable is true … i.e., if true the runtime will not move threads around between processors.
  - OMP_PROC_BIND true | false

… Plus several less commonly used environment variables.

# Plan

| | Module | Concepts | Exercises | |
|---|---|---|---|---|
| 8:30 | OpenMP core concepts | • Intro to OpenMP<br>• Creating threads | • Hello_world<br>• Pi_spmd | |
| 10:30 | Working with threads | • Synchronization<br>• Parallel loops<br>• Single, master, and more | • Pi_spmd_final<br>• Pi_loop | 10 AM Break |
| 1:00 | Managing data and tasks | • Data Environment<br>• tasks | • Mandelbrot set area<br>• Racy tasks<br>• Recursive pi | Noon Lunch |
| 3:30 | Understanding shared memory | • Memory Model<br>• Threadprivate | • Monte Carlo pi | 3 PM Break |
| | OpenMP beyond SMP | • SIMD<br>• Devices and OpenMP | • Jaobi Solver | |

… Plus a set of “challenge problems” for the evening program.

# Data environment: Default sharing 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.

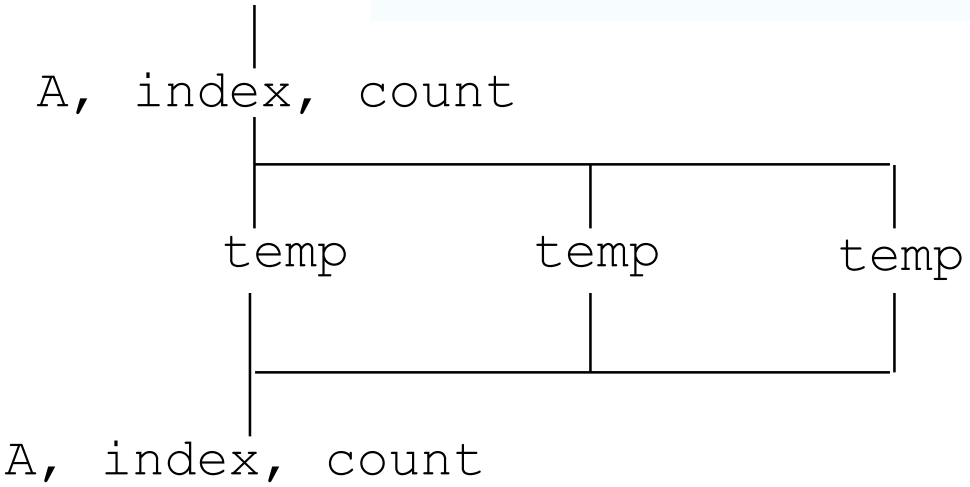
# Data sharing: Examples

```
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 sharing attributes

- One can selectively change sharing 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.

- The final value of a private variable inside a parallel loop can be transmitted to the shared variable outside the loop with:
  - lastprivate(list)
- The default attributes can be overridden with:
  - default (private| shared| none)

default(private) *iin Fortran only*

***All data clauses apply to parallel, worksharing, and task constructs except "shared", which only applies to parallel and task 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);
}
```

Nomenclature: The version of tmp prior to the construct is called the "**original**" variable

tmp was not initialized

tmp reverts to the value of the original variable after the construct (0 in this case)

# 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

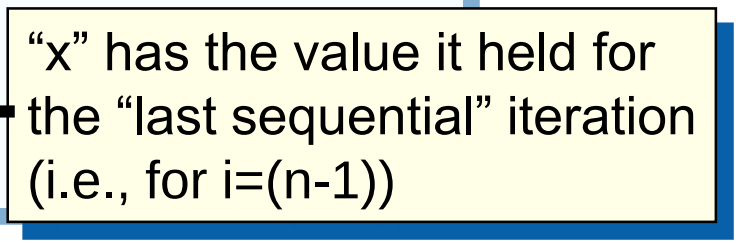
```
incr = 0;
#pragma omp parallel for firstprivate(incr)
for (i = 0; i <= MAX; i++) {
        if ((i%2)==0) incr++;
        A[i] = incr;
}
```

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

# Lastprivate clause

- Variables update a shared variable using value from the (logically) last iteration
- C++ objects are updated as if by assignment

```
void sq2(int n, double *lastterm)
{
    double x; int i;
    #pragma omp parallel for lastprivate(x)
    for (i = 0; i < n; i++){
        x = a[i]*a[i] + b[i]*b[i];
        b[i] = sqrt(x);
     }
     *lastterm = x;
}
```



# 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

- The default storage attribute is **default(shared)** (so no need to use it)
  - Exception: #pragma omp task
- To change default: **default(private)**
  - *each* variable in the construct is made private as if specified in a private clause
  - mostly saves typing
- **default(none)**: *no* default for variables in static extent. Must list storage attribute for each variable in static extent. Good programming practice!

Only the Fortran API supports default(private).

C/C++ only has default(shared) or default(none).

# Data sharing: Default clause example

```
      itotal = 1000
C$OMP PARALLEL PRIVATE(np, each)
      np = omp_get_num_threads()
      each = itotal/np
      ……
C$OMP END PARALLEL
```

These two code fragments are equivalent

```
      itotal = 1000
C$OMP PARALLEL DEFAULT(PRIVATE) SHARED(itotal)
      np = omp_get_num_threads()
      each = itotal/np
      ……
C$OMP END PARALLEL
```

# Exercise 5: 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?

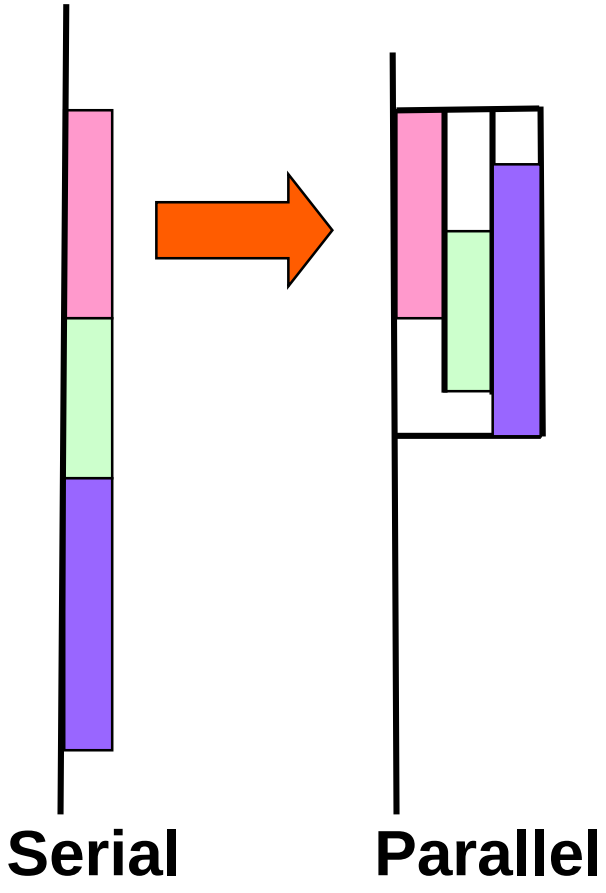
# Plan

| | Module | Concepts | Exercises | |
|---|---|---|---|---|
| 8:30 | OpenMP core concepts | • Intro to OpenMP<br>• Creating threads | • Hello_world<br>• Pi_spmd | |
| 10:30 | Working with threads | • Synchronization<br>• Parallel loops<br>• Single, master, and more | • Pi_spmd_final<br>• Pi_loop | 10 AM Break |
| 1:00 | Managing data and tasks | • Data Environment<br>• tasks | • Mandelbrot set area<br>• Racy tasks<br>• Recursive pi | Noon Lunch |
| 3:30 | Understanding shared memory | • Memory Model<br>• Threadprivate | • Monte Carlo pi | 3 PM Break |
| | OpenMP beyond SMP | • SIMD<br>• Devices and OpenMP | • Jaobi Solver | |

… Plus a set of “challenge problems” for the evening program.

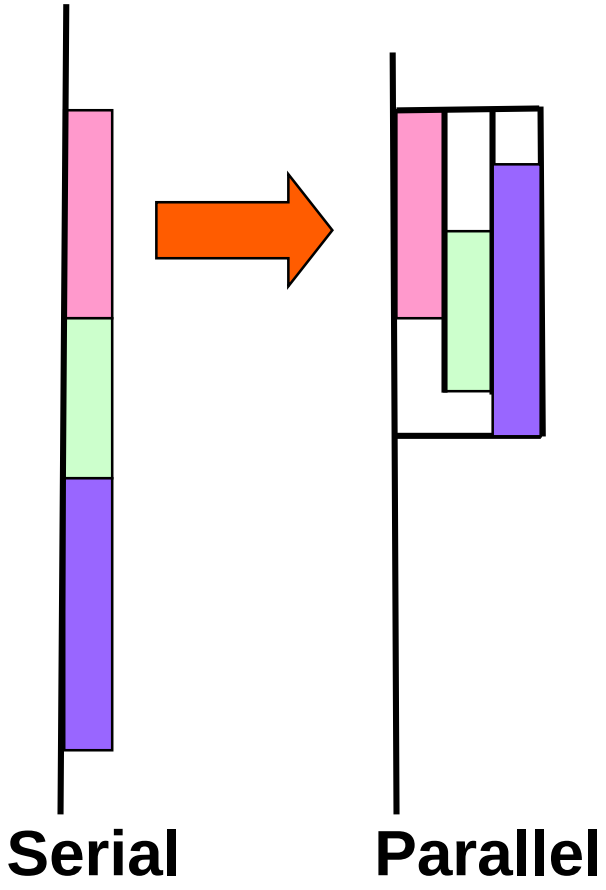
# What are tasks?

- 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.



# Task Directive

**`#pragma omp task`** *[clauses]*

*structured-block*

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

Create some threads

Thread 0 packages tasks

Tasks executed by some thread in some order

All tasks complete before this barrier is released

# Exercise 5: 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 master
#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.

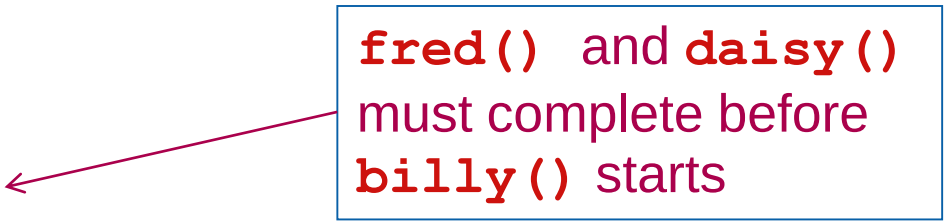
    **`#pragma omp taskwait`**
  - Note: applies only to tasks generated in the current task, not to “descendants” .

- At the end of a taskgroup region
  - **`#pragma omp taskgroup`**

    *structured-block*
  - wait until all tasks created within the taskgroup have completed … applies to all “descendants”

# Example

```
#pragma omp parallel
{
  #pragma omp master
   {
     #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 master
   {
    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 >
```

# Parallel pointer chasing on multiple lists

```
#pragma omp parallel
{
    #pragma omp for private(p)
    for ( int i =0; i <numlists; i++) {
        p = listheads[i] ;
        while (p ) {
        #pragma omp task firstprivate(p)
            {
              process(p);
            }
        p=next(p);
        }
    }
}
```

All threads package tasks

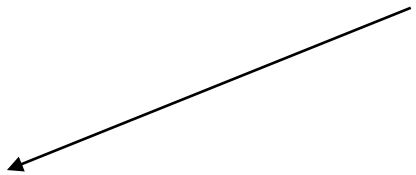
# Data scoping with tasks

- 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 master
           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 6: 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 master
#pragma omp single
double omp_get_wtime()
int omp_get_thread_num();
int omp_get_num_threads();
```

# Task switching

- Certain constructs define task scheduling points … for example:
  - Generation and completion of a Task, Taskwait, implicit or explicit barriers, target data-region constructs,
- When a thread encounters a task scheduling point, it is allowed to suspend the current task and execute another (called *task switching*)
- It can then return to the original task and resume

# Task switching

```
#pragma omp single
{
  for (i=0; i<ONEZILLION; i++)
    #pragma omp task
      process(item[i]);
}
```

- Risk of generating too many tasks
- Generating task will have to suspend for a while
- With task switching, the executing thread can:
  - execute an already generated task (draining the "*task pool*")
  - execute the encountered task

# Task dependencies

!$omp task depend (*type* : *list*)

where *type* is in, out or inout and *list* is a list of variables.

- list may contain subarrays: OpenMP 4.0 includes a syntax for C/C++
- in: the generated task will be a dependent task of all previously generated sibling tasks that reference at least one of the list items in an out or inout clause
- out or inout: the generated task will be a dependent task of all previously generated sibling tasks that reference at least one of the list items in an in, out or inout clause

# Task dependencies example

```
#pragma omp task depend (out:a)
   { ... } //writes a
#pragma omp task depend (out:b)
   { ... } //writes b
#pragma omp task depend (in:a,b)
   { ... } //reads a and b
```

- The first two tasks can execute in parallel
- The third task cannot start until the first two are complete

# Controlling tasks

- Two things can happen with a task:
  - *included* (executed now by the thread that encounters them)
  - *deferred* (executed by some thread independently of generating task)
    - *undeferred* (completes execution before the generating task continues)

- The task construct can take an **`if(expr)`** clause, which if the expression evaluates to false, means the task will be undeferred

- The task construct can take a **`final(expr)`** clause, which if the expression evaluates to true, means any tasks generated inside this task will be included

- The task construct can take a **`mergeable`** clause, which indicates it can be safely executed by reusing its parent data environment; most useful if used in conjunction with **`final`**

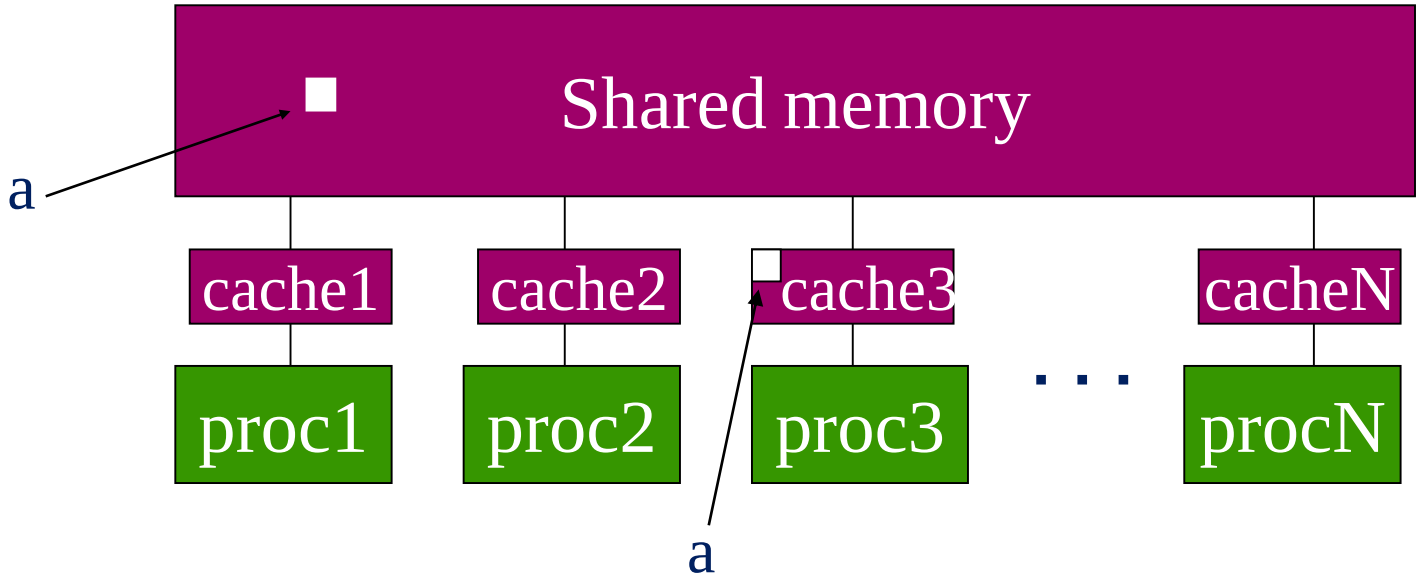
# Plan

| | Module | Concepts | Exercises | |
|---|---|---|---|---|
| 8:30 | OpenMP core concepts | • Intro to OpenMP<br>• Creating threads | • Hello_world<br>• Pi_spmd | 10 AM Break |
| 10:30 | Working with threads | • Synchronization<br>• Parallel loops<br>• Single, master, and more | • Pi_spmd_final<br>• Pi_loop | Noon Lunch |
| 1:00 | Managing data and tasks | • Data Environment<br>• tasks | • Mandelbrot set area<br>• Racy tasks<br>• Recursive pi | 3 PM Break |
| 3:30 | Understanding shared memory | • Memory Model<br>• Threadprivate | • Prod/cons<br>• Monte Carlo pi | |
| | OpenMP beyond SMP | • SIMD<br>• Devices and OpenMP | • Jaobi Solver | |

… Plus a set of “challenge problems” for the evening program.

# OpenMP memory model

- OpenMP supports a shared memory model
- All threads share an address space, where variable can be stored or retrieved:




- Threads maintain their own temporary view of memory as well … the details of which are not defined in OpenMP but this temporary view typically resides in caches, registers, write-buffers, etc.

# 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 enforces a consistent view of memory.

- For variables visible to other threads and associated with the flush operation (the **flush-set**)
  - The compiler can't move loads/stores of the flush-set around a flush:
    - All previous read/writes of the flush-set by this thread have completed
    - No subsequent read/writes of the flush-set by this thread have occurred
  - Variables in the flush set are moved from temporary storage to shared memory.
  - Reads of variables in the flush set following the flush are loaded from shared memory.

IMPORTANT POINT: The flush makes the calling threads temporary view match the view in shared memory. Flush by itself does not force synchronization.

# Memory consistency: flush example

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

Flush without a list: flush set is all thread visible variables

Flush with a list: flush set is the list of variables

```
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

# 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
  - whenever a lock is set or unset

  ….

  (but not at entry to worksharing regions or entry/exit of master regions)

# 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

The problem is this program technically has a race … on the store and later load of flag

# The OpenMP 3.1 atomics (1 of 2)

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

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

- Atomic can protect loads

  **# pragma omp atomic read**

  **v = x;**

- Atomic can protect stores

  **# pragma omp atomic write**

  **x = expr;**

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

  **# pragma omp atomic update**

  **x++; or ++x; or x--; or –x; or**

  **x binop= expr; or x = x binop expr;**

This is the original OpenMP atomic

# The OpenMP 3.1 atomics (2 of 2)

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

  # pragma omp atomic capture

  statement or structured block

- Where the statement is one of the following forms:

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

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

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

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

**{v = x; x++;}**

**{++x; v=x:}**

**{v = x; x--;}**

**{--x; v = x;}**

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

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

**{v=x; ++x:}**

**{x++; v = x;}**

**{v= x; --x;}**

**{x--; v = x;}**

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

# 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**

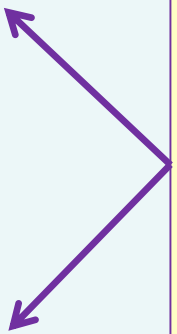
# OpenMP 4.0 Atomic: Sequential consistency


- 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)**

# Plan

| | Module | Concepts | Exercises | |
|---|---|---|---|---|
| 8:30 | OpenMP core concepts | • Intro to OpenMP<br>• Creating threads | • Hello_world<br>• Pi_spmd | |
| 10:30 | Working with threads | • Synchronization<br>• Parallel loops<br>• Single, master, and more | • Pi_spmd_final<br>• Pi_loop | 10 AM Break |
| 1:00 | Managing data and tasks | • Data Environment<br>• tasks | • Mandelbrot set area<br>• Racy tasks<br>• Recursive pi | Noon Lunch |
| 3:30 | Understanding shared memory | • Memory Model<br>• Threadprivate | • Prod/cons<br>• Monte Carlo pi | 3 PM Break |
| | OpenMP beyond SMP | • SIMD<br>• Devices and OpenMP | • Jaobi Solver | |

… Plus a set of “challenge problems” for the evening program.

# Data sharing: Threadprivate

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

# A threadprivate example (C)

Use threadprivate to create a counter for each thread.

```
int counter = 0;
#pragma omp threadprivate(counter)

int increment_counter()
{
    counter++;
    return (counter);
}
```

# Data copying: Copyin

You initialize threadprivate data using a copyin clause.

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

C Initialize the A array
      call init_data(N,A)

!$OMP PARALLEL COPYIN(A)

 … Now each thread sees threadprivate array A initialized
 … to the global value set in the subroutine init_data()

!$OMP END PARALLEL

end
```

# Data copying: Copyprivate

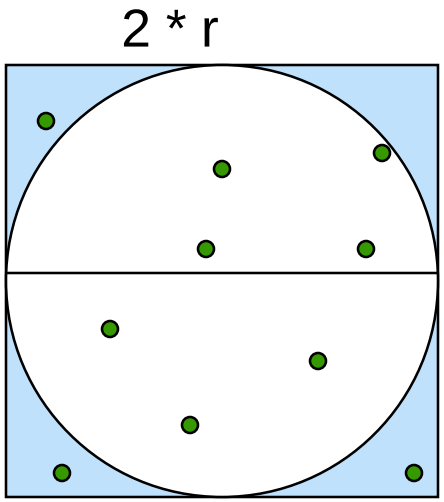
Used with a single region to broadcast values of privates from one member of a team to the rest of the team

```
#include <omp.h>
void input_parameters (int, int); // fetch values of input parameters
void do_work(int, int);

void main()
{
   int Nsize, choice;

   #pragma omp parallel private (Nsize, choice)
   {
      #pragma omp single copyprivate (Nsize, choice)
            input_parameters (*Nsize, *choice);

      do_work(Nsize, choice);
   }
}
```

# Exercise: Monte Carlo calculations

**Using random numbers to solve tough problems**

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




| | |
|---|---|
| N= 10 | 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

# Exercise: Monte Carlo pi (cont)

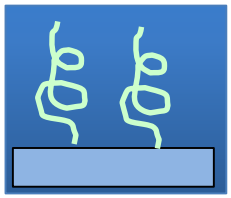
- We provide three files for this exercise
  - pi_mc.c: the Monte Carlo method pi program
  - random.c: a simple random number generator
  - random.h: include file for random number generator
- Create a parallel version of this program without changing the interfaces to functions in random.c
  - This 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).

# Plan

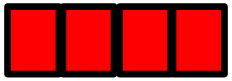
| | Module | Concepts | Exercises | |
|---|---|---|---|---|
| 8:30 | OpenMP core concepts | • Intro to OpenMP<br>• Creating threads | • Hello_world<br>• Pi_spmd | |
| 10:30 | Working with threads | • Synchronization<br>• Parallel loops<br>• Single, master, and more | • Pi_spmd_final<br>• Pi_loop | 10 AM Break |
| 1:00 | Managing data and tasks | • Data Environment<br>• tasks | • Mandelbrot set area<br>• Racy tasks<br>• Recursive pi | Noon Lunch |
| 3:30 | Understanding shared memory | • Memory Model<br>• Threadprivate | • Monte Carlo pi | 3 PM Break |
| | OpenMP beyond SMP | • SIMD<br>• Devices and OpenMP | • Jaobi Solver | |

… Plus a set of “challenge problems” for the evening program.

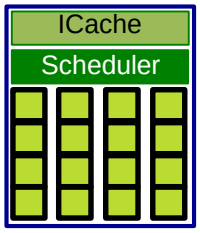
# Hardware Diversity: Basic Building Blocks



CPU Core: one or more hardware threads sharing an address space. Optimized for low latencies.

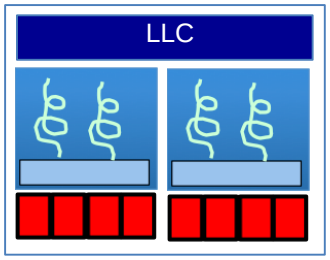


SIMD: Single Instruction Multiple Data.
Vector registers/instructions with 128 to 512 bits so a single stream of instructions drives multiple data elements.

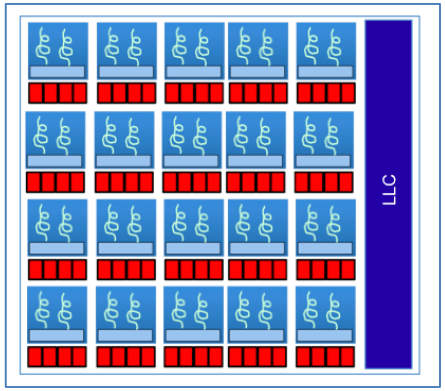



SIMT: Single Instruction Multiple Threads.
A single stream of instructions drives many threads. More threads than functional units. Over subscription to hide latencies. Optimized for throughput.

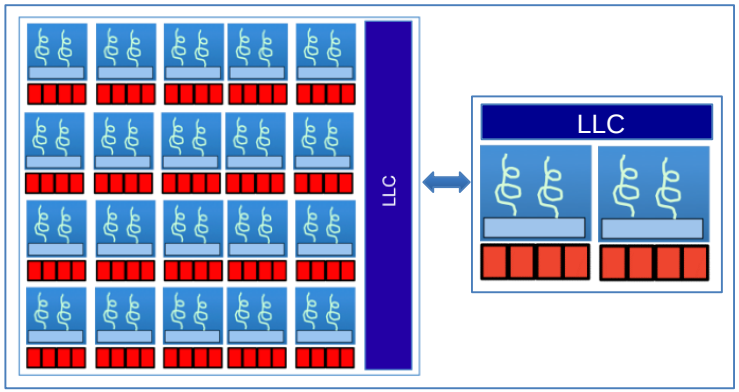
# Hardware Diversity: Combining building blocks to construct nodes



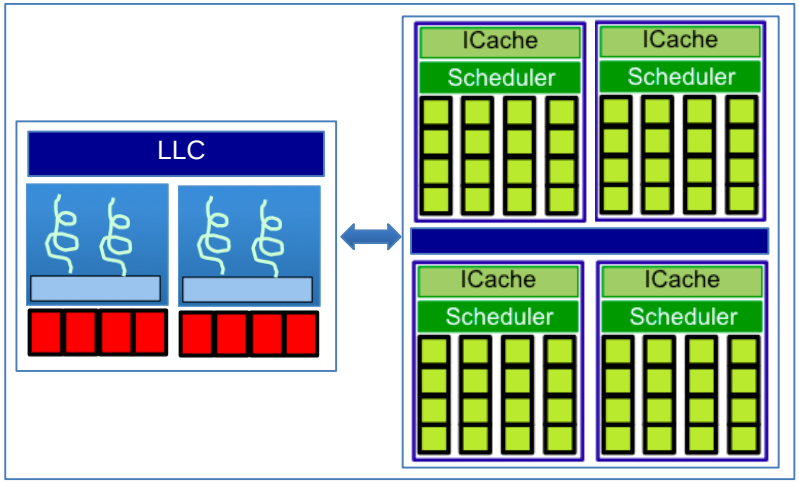

Multicore CPU



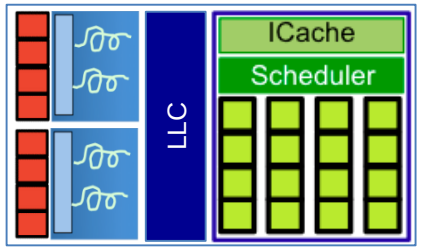

Manycore CPU




Heterogeneous:
CPU+manycore CPU

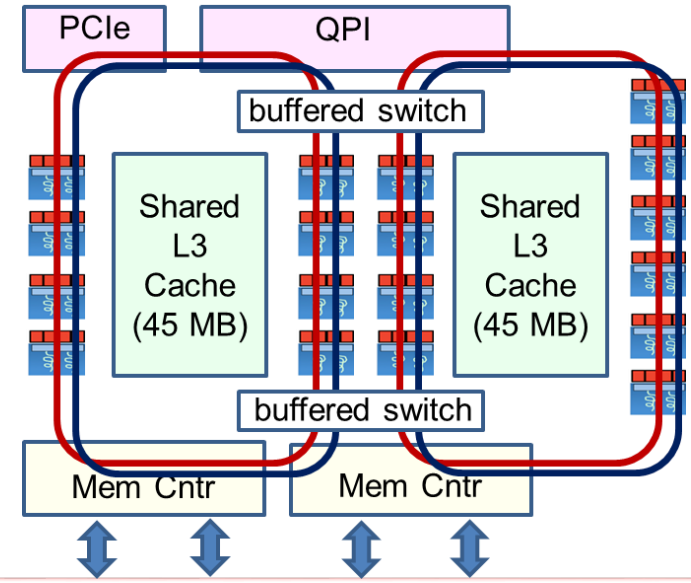



Heterogeneous: CPU+GPU



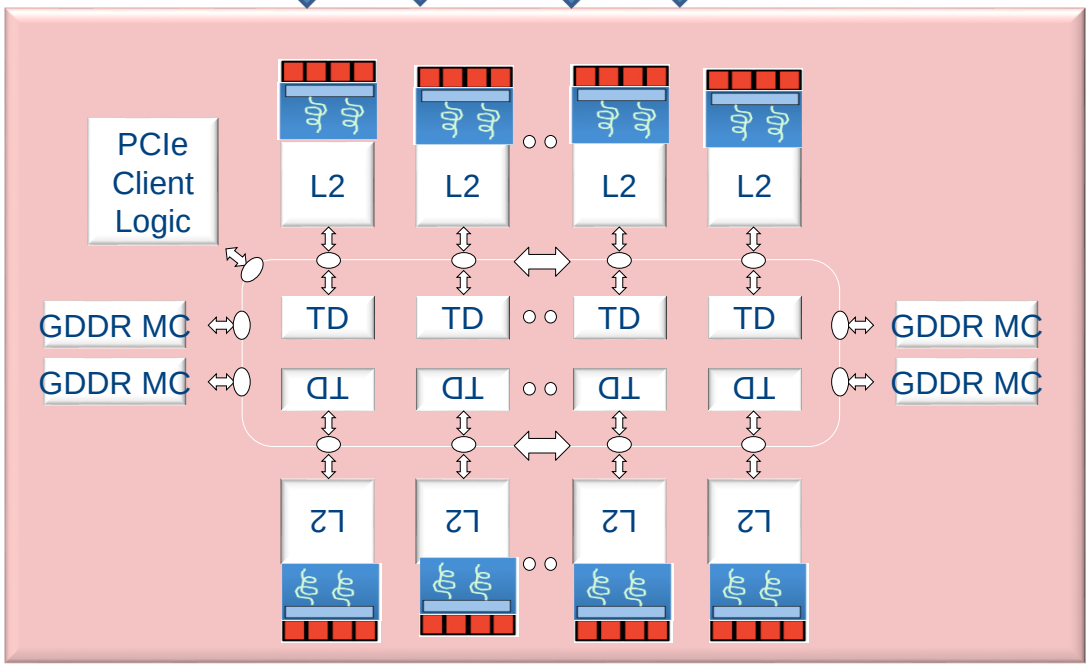

Heterogeneous:
Integrated CPU+GPU

# Hardware diversity: CPUs




Intel® Xeon® processor
E7 v3 series (Haswell or HSW)

- 18 cores
- 36 Hardware threads
- 256 bit wide vector units

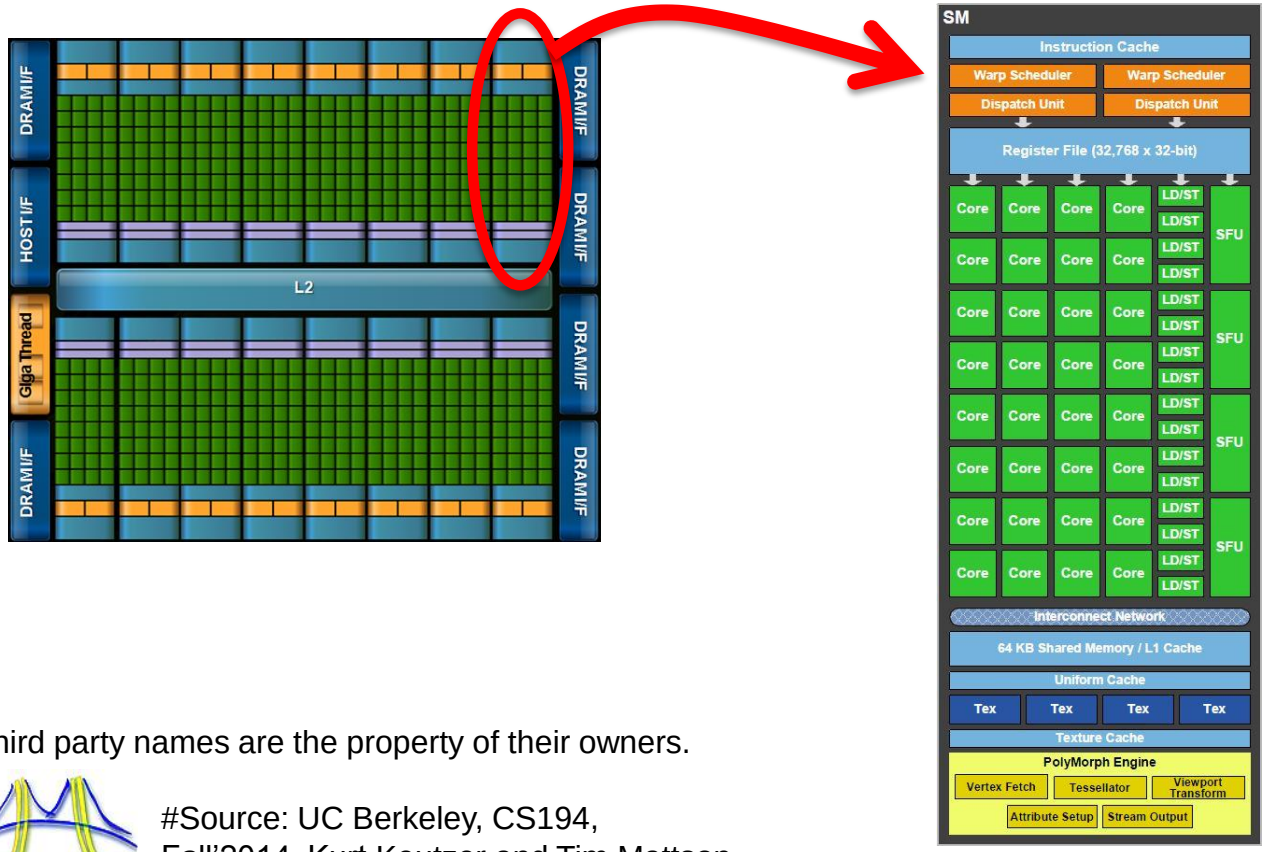



Intel® Xeon Phi™ coprocessor
(Knights Corner)

- 61 cores
- 244 Hardware threads
- 512 bit wide vector units

# Hardware diversity: GPUs

- Nvidia® GPUs are a collection of “Streaming Multiprocessors” (SM)
  - Each SM is analogous to a core of a Multi-Core CPU
- Each SM is a collection of SIMD execution pipelines that share control logic, register file, and L1 Cache#



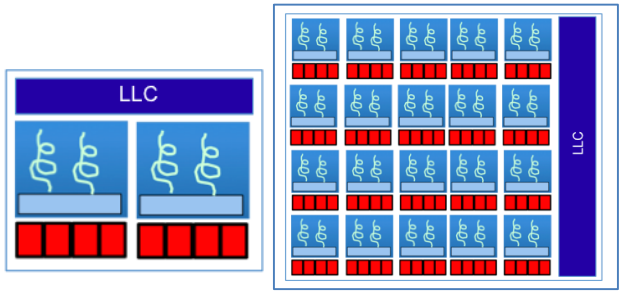

For example: an NVIDIA Tesla C2050 (Fermi) GPU with 3GB of memory and 14 streaming multiprocessor cores*.



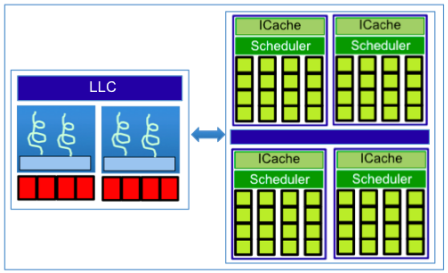
#Source: UC Berkeley, CS194, Fall’2014, Kurt Keutzer and Tim Mattson

*Source: http://www.nersc.gov/users/computational-systems/dirac/node-and-gpu-configuration/

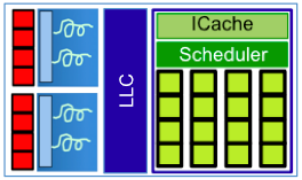
# Hardware Diversity: programming models




OpenMP, OpenCL, pthreads, MPI, TBB, Cilk, C++'11…

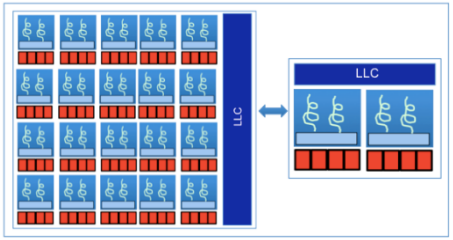



OpenMP, OpenCL, CUDA, OpenACC



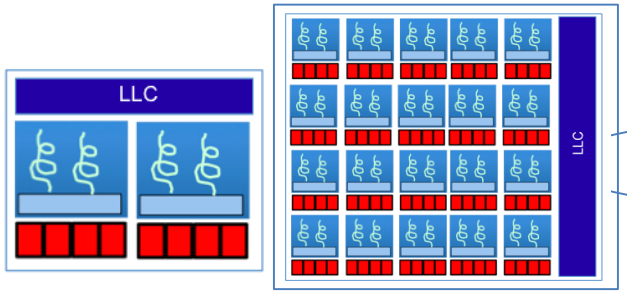

OpenMP, OpenCL,

Do you notice a trend?

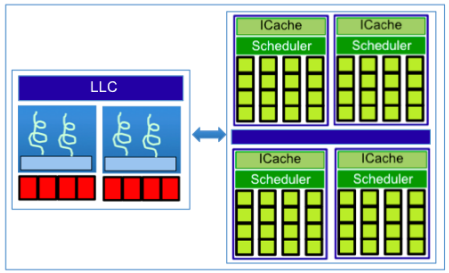



OpenMP, OpenCL, pthreads, TBB, Cilk, C++'11…

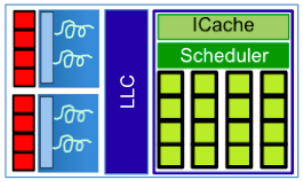
# Hardware Diversity: programming models




OpenMP, OpenCL, pthreads, MPI, TBB, Cilk, C++'11…

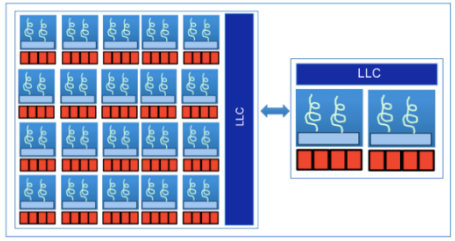



OpenMP, OpenCL, CUDA, OpenACC



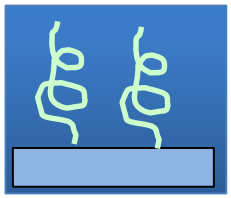

OpenMP, OpenCL,

If you want to support the diversity of nodes in HPC from a single source-code base, you have only two choices: OpenMP and OpenCL



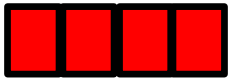

OpenMP, OpenCL, pthreads, TBB, Cilk, C++'11…

# Hardware Diversity: Basic Building Blocks

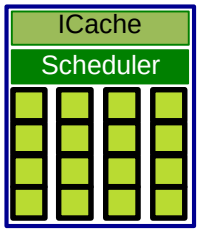


CPU Core: one or more hardware threads sharing an address space. Optimized for low latencies.

Let's start with the SIMD construct for dealing with vector units

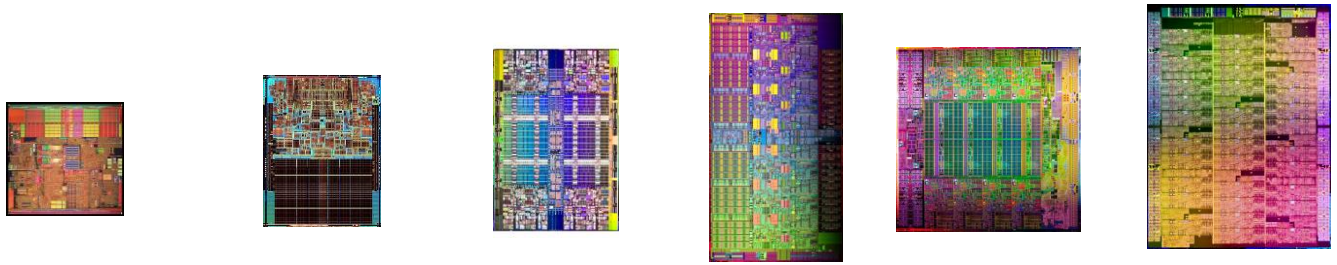


SIMD: Single Instruction Multiple Data.
Vector registers/instructions with 128 to 512 bits so a single stream of instructions drives multiple data elements.




SIMT: Single Instruction Multiple Threads.
A single stream of instructions drives many threads. More threads than functional units. Over subscription to hide latencies. Optimized for throughput.

# Evolution of Hardware (Intel)

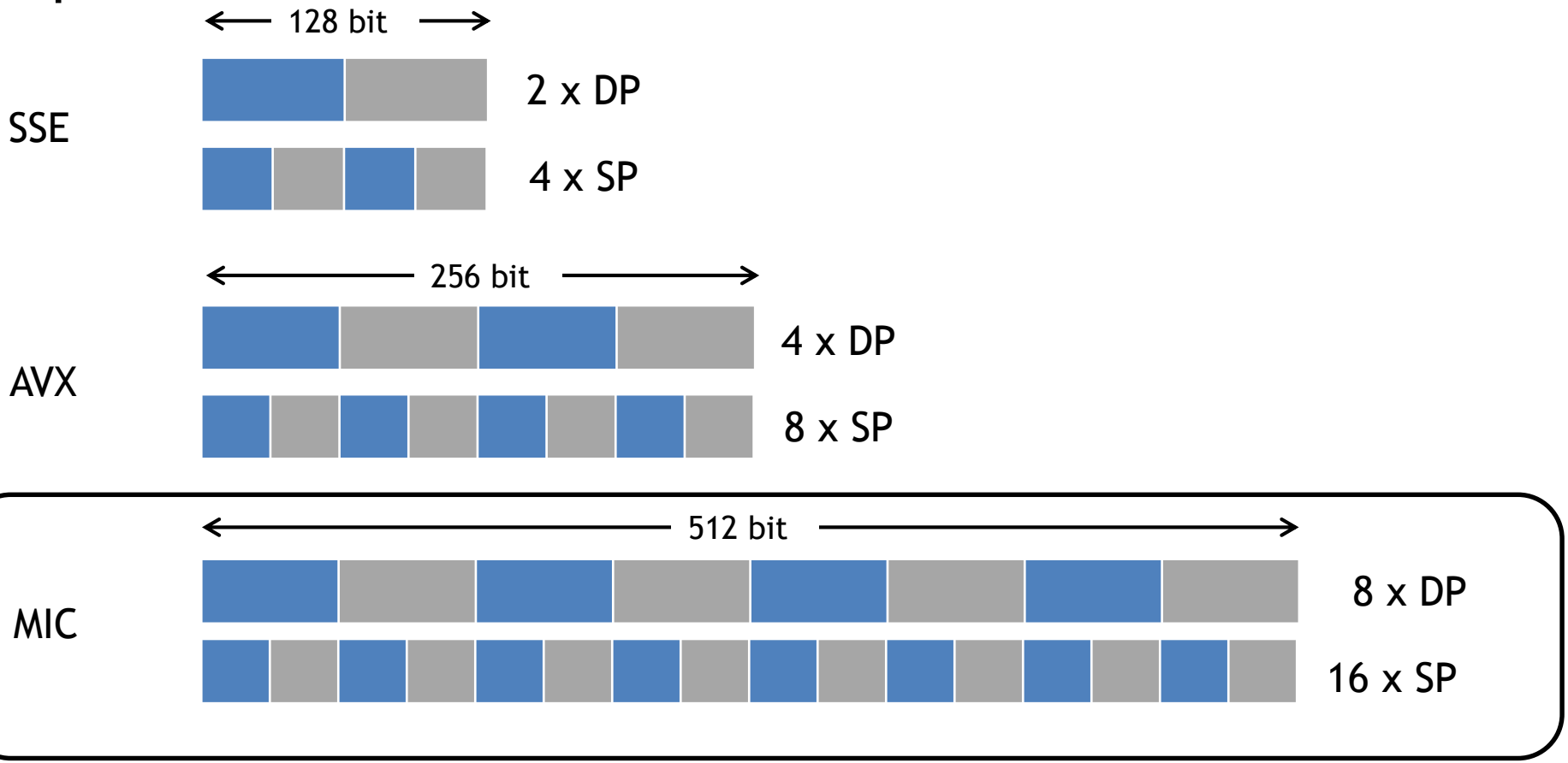


*Images not intended to reflect actual die sizes*

| | 64-bit Intel® Xeon® processor | Intel® Xeon® processor 5100 series | Intel® Xeon® processor 5500 series | Intel® Xeon® processor 5600 series | Intel® Xeon® processor E5-2600 series | Intel® Xeon Phi™ Co-processor 5110P |
|---|---|---|---|---|---|---|
| Frequency | 3.6GHz | 3.0GHz | 3.2GHz | 3.3GHz | 2.7GHz | 1053MHz |
| Core(s) | 1 | 2 | 4 | 6 | 8 | 60 |
| Thread(s) | 2 | 2 | 8 | 12 | 16 | 240 |
| SIMD width | 128 (2 clock) | 128 (1 clock) | 128 (1 clock) | 128 (1 clock) | 256 (1 clock) | 512 (1 clock) |

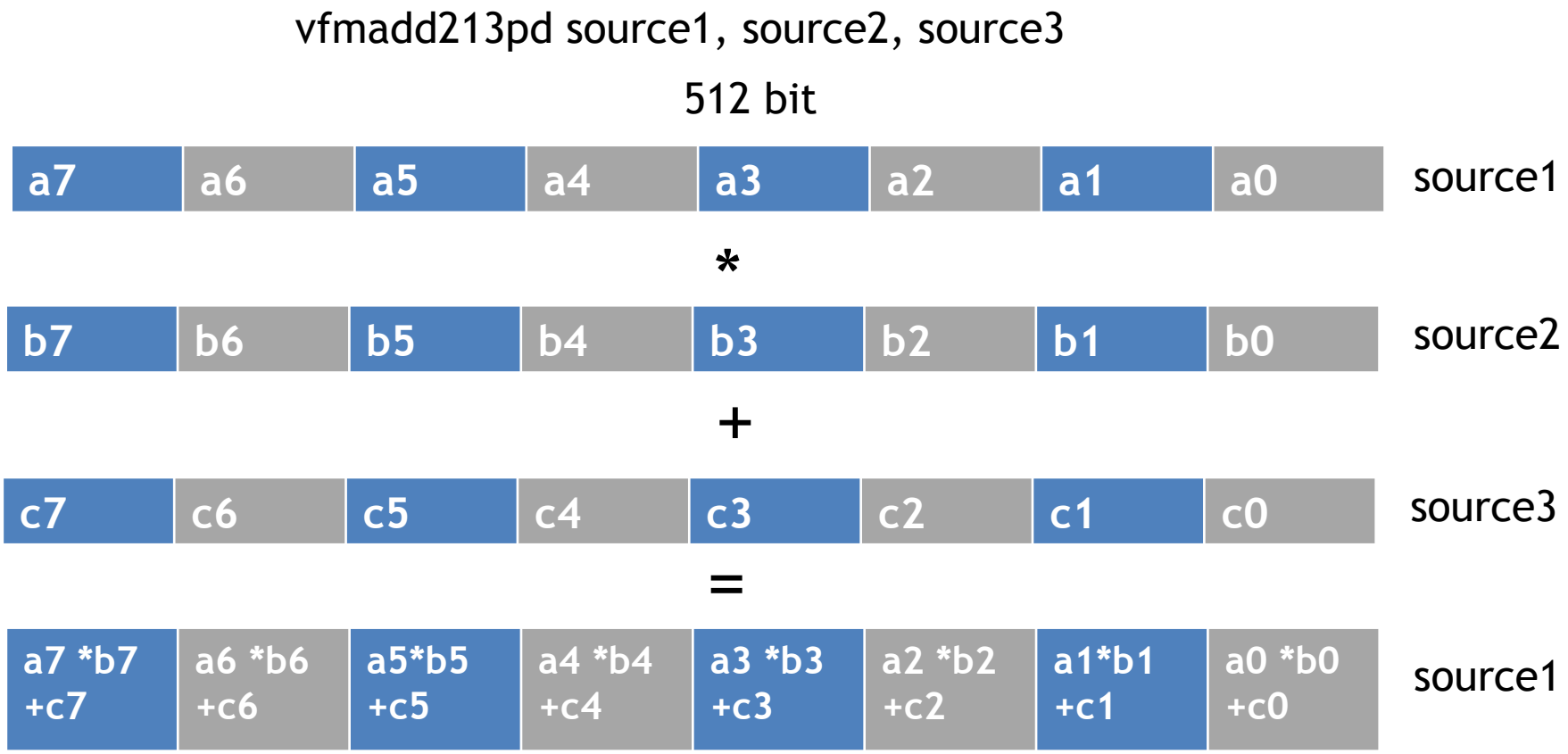
# SIMD on Intel® Architecture

- Width of SIMD registers has been growing in the past:



# More Powerful SIMD Units

- SIMD instructions become more powerful
- One example is Intel® Xeon Phi™ Coprocessor



# Auto-vectorization

- Auto vectorization only helps in some cases
  - → Increased complexity of instructions make hard for the compiler to select proper instructions
  - → Code pattern needs to be recognized by the compiler
  - → Precision requirements often inhibit SIMD code gen
- Example: Intel® Composer XE

  `-vec` (automatically enabled with **-O3**)

  `-vec-report`

  `-opt-report`

# Why Auto-vectorizers Fail

- Data dependencies
- Other potential reasons
  - → Alignment
  - → Function calls in loop block
  - → Complex control flow / conditional branches
  - → Loop not “countable”
    - → E.g. upper bound not a runtime constant
  - → Mixed data types
  - → Non-unit stride between elements
  - → Loop body too complex (register pressure)
  - → Vectorization seems inefficient
- Many more … but less likely to occur

# Data Dependencies

- Suppose two statements S1 and S2
- S2 depends on S1, iff S1 must execute before S2
  - → Control-flow dependence
  - → Data dependence
  - → Dependencies can be carried over between loop iterations
- Important flavors of data dependencies

FLOW

```
s1: a = 40

    b = 21

s2: c = a + 2
```

ANTI

```
    b = 40

s1: a = b + 1

s2: b = 21
```

# In a Time Before OpenMP 4.0

- Support required vendor-specific extensions
  - → Programming models (e.g., Intel® Cilk Plus)
  - → Compiler pragmas (e.g., `#pragma vector`)
  - → Low-constructs (e.g., `_mm_add_pd()`)

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

You need to trust your compiler to do the "right" thing.

# SIMD Loop Construct

- Vectorize a loop nest
  - → Cut loop into chunks that fit a SIMD vector register
  - → No parallelization of the loop body

- Syntax (C/C++)

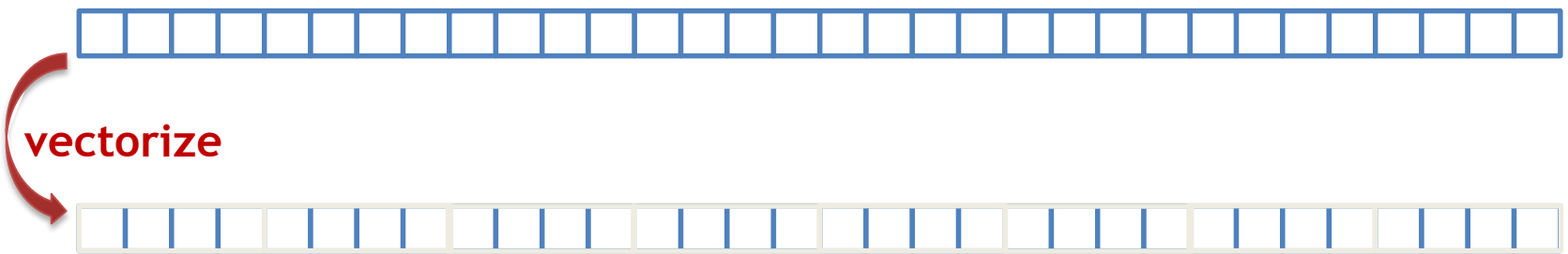
```
#pragma omp simd [clause[[,] clause],…]
for-loops
```

- Syntax (Fortran)

```
!$omp simd [clause[[,] clause],…]
do-loops
```

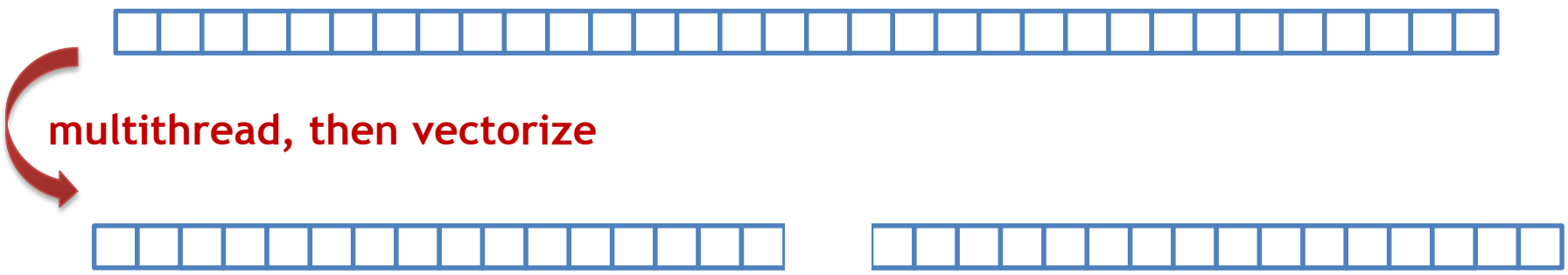
# Example

```
void sprod(float *a, float *b, int n) {
  float sum = 0.0f;
#pragma omp simd reduction(+:sum)
  for (int k=0; k<n; k++)
    sum += a[k] * b[k];
  return sum;
}
```



# Example: threads plus SIMD

```
void sprod(float *a, float *b, int n) {
  float sum = 0.0f;
#pragma omp parallel for simd reduction(+:sum)
  for (int k=0; k<n; k++)
    sum += a[k] * b[k];
  return sum;
}
```



# Data Sharing Clauses

- `private(`*`var-list`*`):`
  Uninitialized vectors for variables in *var-list*


- `firstprivate(`*`var-list`*`):`
  Initialized vectors for variables in *var-list*


- `reduction(`*`op`*`:`*`var-list`*`):`
  Create private variables for *var-list* and apply reduction operator *op* at the end of the construct

# SIMD Loop Clauses

- `safelen (length)`
  - → Maximum number of iterations that can run concurrently without breaking a dependence
  - → in practice, maximum vector length
- `linear (list[:linear-step])`
  - → The variable's value is in relationship with the iteration number
    - → $x_i = x_{orig} + i * \text{linear-step}$
- `aligned (list[:alignment])`
  - → Specifies that the list items have a given alignment
  - → Default is alignment for the architecture
- `collapse (n)`

# SIMD Function Vectorization

```
float min(float a, float b) {
    return a < b ? a : b;
}


float distsq(float x, float y) {
    return (x - y) * (x - y);
}

void example() {
#pragma omp parallel for simd
    for (i=0; i<N; i++) {
        d[i] = min(distsq(a[i], b[i]), c[i]);
}   }
```

# SIMD Function Vectorization

- Declare one or more functions to be compiled for the target device

- Syntax (C/C++):

```
#pragma omp declare simd [clause[[,] clause],…]
[#pragma omp declare simd [clause[[,] clause],…]]
[…]
function-definitions-or-declaration
```

- Syntax (Fortran):

```
!$omp declare simd (proc-name-list)
```

# SIMD Function Vectorization

```
#pragma omp declare simd
float min(float a, float b) {
    return a < b ? a : b;
}

#pragma omp declare simd
float distsq(float x, float y) {
    return (x - y) * (x - y);
}

void example() {
#pragma omp parallel for simd
    for (i=0; i<N; i++) {
        d[i] = min(distsq(a[i], b[i]), c[i]);
}   }
```

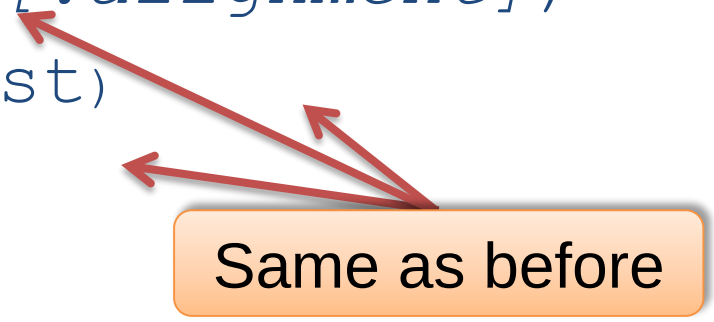
```
vec8 min_v(vec8 a, vec8 b) {
    return a < b ? a : b;
}
```

```
vec8 distsq_v(vec8 x, vec8 y)
    return (x - y) * (x - y);
}
```

```
vd = min_v(distsq_v(va, vb, vc)
```

# SIMD Function Vectorization

- `simdlen (`*`length`*`)`
  - → generate function to support a given vector length
- `uniform (`*`argument-list`*`)`
  - → argument has a constant value between the iterations of a given loop
- `inbranch`
  - → function always called from inside an if statement
- `notinbranch`
  - → function never called from inside an if statement
- `linear (`*`argument-list[:linear-step]`*`)`
- `aligned (`*`argument-list[:alignment]`*`)`
- `reduction (`*`operator`*`:list)`



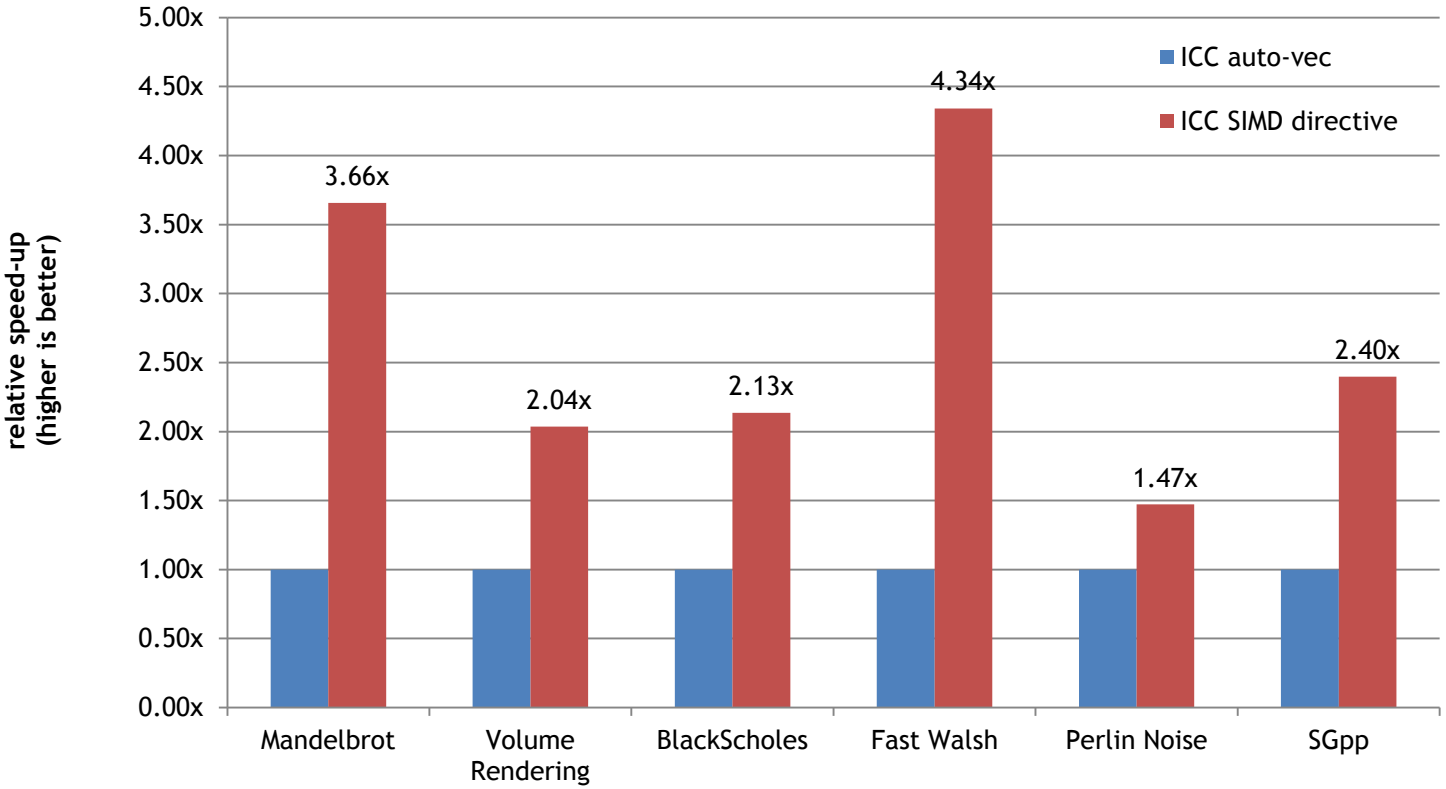
# inbranch & notinbranch

```
#pragma omp declare simd inbranch
float do_stuff(float x) {
    /* do something */
    return x * 2.0;
}

void example() {
#pragma omp simd
    for (int i = 0; i < N; i++)
        if (a[i] < 0.0)
            b[i] = do_stuff(a[i]);
}
```

```
vec8 do_stuff_v(vec8 x, mask m) {
    /* do something */
    vmulpd x{m}, 2.0, tmp
    return tmp;
}
```

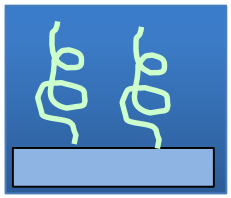
```
for (int i = 0; i < N; i+=8) {
    vcmp_lt &a[i], 0.0, mask
    b[i] = do_stuff_v(&a[i], mask);
}
```

# SIMD Constructs & Performance

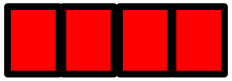



M.Klemm, A.Duran, X.Tian, H.Saito, D.Caballero, and X.Martorell. Extending OpenMP with Vector Constructs for Modern Multicore SIMD Architectures. In Proc. of the Intl. Workshop on OpenMP, pages 59-72, Rome, Italy, June 2012. LNCS 7312.

# Hardware Diversity: Basic Building Blocks

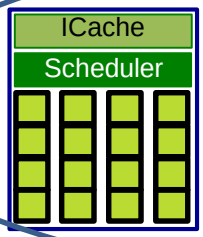


CPU Core: one or more hardware threads sharing an address space. Optimized for low latencies.



SIMD: Single Instruction Multiple Data.
Vector registers/instructions with 128 to 512 bits so a single stream of instructions drives multiple data elements.

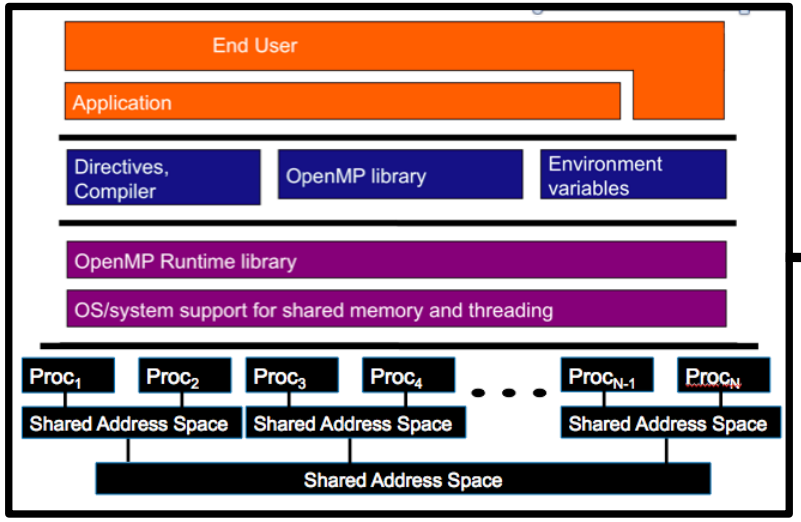
How to program a GPU with OpenMP



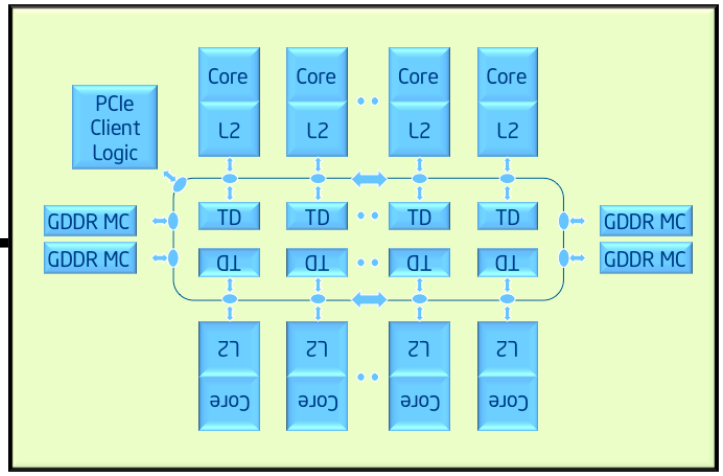

SIMT: Single Instruction Multiple Threads.
A single stream of instructions drives many threads. More threads than functional units. Over subscription to hide latencies. Optimized for throughput.

# OpenMP basic definitions: Target solution stack

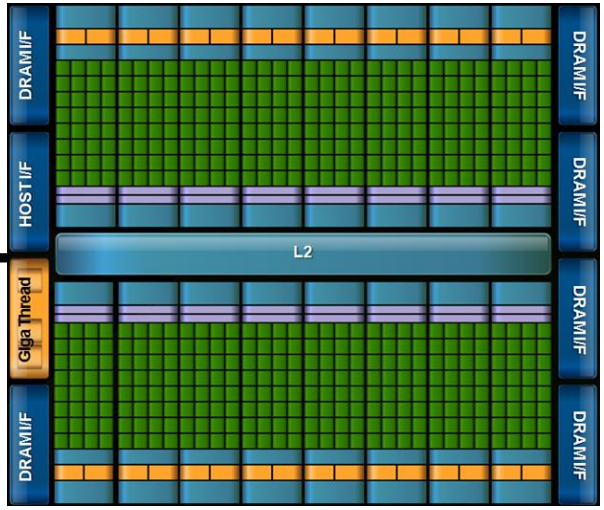
Supported (since OpenMP 4.0) with target, teams, distribute, and other constructs




Host



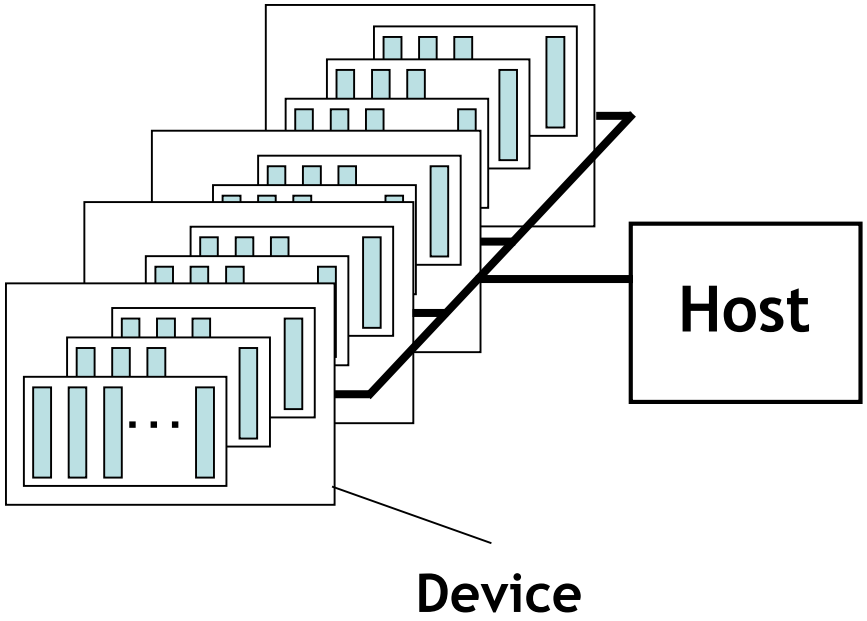

Target Device: Intel® Xeon Phi™ processor




Target Device: GPU

# The OpenMP device programming model

- OpenMP uses a host/device model
  - The host is where the initial thread of the program begins execution
  - Zero or more devices are connected to the host




```
#include <omp.h>
#include <stdio.h>
int main()
{
    printf("There are %d devices\n",
            omp_get_num_devices());
}
```

# Target directive

- The target construct offloads a code region to a device.

```
#pragma omp target
{….}  // a structured block of code
```

- An initial thread running on the device executes the code in the code block.

```
#pragma omp target
{
    #pragma omp parallel for
        {do lots of stuf}
}
```

# Target directive

- The target construct offloads a code region to a device.

```
#pragma omp target device(1)
{….}  // a structured block of code
```

Optional clause to select some device other than the default device.

- An initial thread running on the device executes the code in the code block.

```
#pragma omp target
{
    #pragma omp parallel for
        {do lots of stuf}
}
```

# The target data environment

- The target clause creates a data environment on the device:

```
int i, a[N], b[N], c[N];
#pragma omp target
```

Original variables on the host: N, i, a, b, c …

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

Are mapped onto the corresponding variables on the device: N, i, a, b, c …

- Originals variables copied into corresponding variables before the initial thread begins execution on the device.
- Corresponding variables copied into original variables when the target code region completes

# Controlling data movement

```
int i, a[N], b[N], c[N];
#pragma omp target map(to:a,b) map(tofrom:c)
```

Data movement can be explicitly controlled with the map clause

- The various forms of the map clause
  - **map(to:list)**: *read-only* data on the device. Variables in the list are initialized on the device using the original values from the host.
  - **map(from:list)**: *write-only* data on the device: initial value of the variable is not initialized. At the end of the target region, the values from variables in the list are copied into the original variables.
  - **map(tofrom:list)**: the effect of both a map-to and a map-from
  - **map(alloc:list)**: data is allocated and uninitialized on the device.
  - **map(list)**: equivalent to map(tofrom:list).
- For pointers you must use array notation ..
  - Map(to:a[0:N])

# Exercise

- Start with the provided serial Jacobi solver.
- Use the target data construct to create a data region. Manage data movement with map clauses to minimize data movement.
  - #pragma omp target
  - #pragma omp target data
  - #pragma omp target map(to:list) map(from:list) map(tofrom:list)
  - int omp_get_num_devices();
  - #pragma omp parallel for reduction(+:var) private(list)

# Jacobi Solver (serial 1/2)

```
while((conv > TOL) && (iters<MAX_ITERS))
  {
    iters++;
    xtmp  = xnew;    // don't copy arrays.
    xnew  = xold;    // just swap pointers.
    xold  = xtmp;

    for (i=0; i<Ndim; i++){
        xnew[i] = (TYPE) 0.0;
        for (j=0; j<Ndim;j++){
            if(i!=j)
              xnew[i]+= A[i*Ndim + j]*xold[j];
        }
        xnew[i] = (b[i]-xnew[i])/A[i*Ndim+i];
    }
```

# Jacobi Solver (serial 2/2)

```
    //
    // test convergence
    //
    conv = 0.0;
    for (i=0; i<Ndim; i++){
        tmp  = xnew[i]-xold[i];
        conv += tmp*tmp;
    }
    conv = sqrt((double)conv);

} \\ end while loop
```

# Jacobi Solver (Par Targ, 1/2)

```
while((conv > TOL) && (iters<MAX_ITERS))
  {
    iters++;
    xtmp  = xnew;    // don't copy arrays.
    xnew  = xold;    // just swap pointers.
    xold  = xtmp;
#pragma omp target map(tofrom:xnew[0:Ndim],xold[0:Ndim]) \
                   map(to:A[0:Ndim*Ndim], b[0:Ndim] ,Ndim)
     #pragma omp parallel for private(i,j)
    for (i=0; i<Ndim; i++){
        xnew[i] = (TYPE) 0.0;
        for (j=0; j<Ndim;j++){
            if(i!=j)
              xnew[i]+= A[i*Ndim + j]*xold[j];
        }
        xnew[i] = (b[i]-xnew[i])/A[i*Ndim+i];
    }
```

# Jacobi Solver (Par Targ, 2/2)

```
    //
    // test convergence
    //
    conv = 0.0;
#pragma omp target map(to:xnew[0:Ndim],xold[0:Ndim]) \
                        map(to:Ndim) map(tofrom:conv)
      #pragma omp parallel for private(i,tmp) reduction(+:conv)
    for (i=0; i<Ndim; i++){
        tmp  = xnew[i]-xold[i];
        conv += tmp*tmp;
    }
    conv = sqrt((double)conv);

} \\ end while loop
```

# Jacobi Solver (Par Targ, 2/2)

```
    //
    // test convergence
    //
    conv = 0.0;
#pragma omp target map(to:xnew[0:Ndim],xold[0:Ndim]) \
                    map(to:Ndim) map(tofrom:conv)
  #pragma omp parallel for private(i,tmp) reduction(+:conv)
    for (i=0; i<Ndim; i++){
        tmp  = xnew[i]-xold[i];
        conv += tmp*tmp;
    }
    conv = sqrt((double)conv);

} \\ end while loop
```

This worked but the performance was awful. Why?

| System | Implementation | Ndim = 1000 | Ndim = 4096 |
|---|---|---|---|
| Intel® Xeon Phi™ co-processor (knights corner) | Target dir per loop | 134 seconds | Did not finish (> 40 minutes) |
| | Native OMP | 3.2 seconds | 5.3 seconds |

# Data movement dominates!!!

```
while((conv > TOLERANCE) && (iters<MAX_ITERS))
  { iters++;
    xtmp  = xnew;    // don't copy arrays.
    xnew  = xold;    // just swap pointers.
    xold  = xtmp;

    #pragma omp target map(tofrom:xnew[0:Ndim],xold[0:Ndim]) \
                       map(to:A[0:Ndim*Ndim], b[0:Ndim] ,Ndim)
     #pragma omp parallel for private(i,j)
     for (i=0; i<Ndim; i++){
        xnew[i] = (TYPE) 0.0;
        for (j=0; j<Ndim;j++){
           if(i!=j)
             xnew[i]+= A[i*Ndim + j]*xold[j];
        }
        xnew[i] = (b[i]-xnew[i])/A[i*Ndim+i];
     }
// test convergence
    conv = 0.0;
    #pragma omp target map(to:xnew[0:Ndim],xold[0:Ndim]) \
                           map(to:Ndim) map(tofrom:conv)
     #pragma omp parallel for private(i,tmp) reduction(+:conv)
     for (i=0; i<Ndim; i++){
        tmp  = xnew[i]-xold[i];
        conv += tmp*tmp;
     }
    conv = sqrt((double)conv);
  }
```

Typically over 4000 iterations!

For each iteration, copy to device
$(3*Ndim+Ndim^2)*sizeof(TYPE)$ bytes

For each iteration, copy from device
2*Ndim*sizeof(TYPE) bytes

For each iteration, copy to device
2*Ndim*sizeof(TYPE) bytes

# Target data directive

- The **target data** construct creates a target data region.
- You use the map clauses for explicit data management

```
#pragma omp target data map(to: A,B) map(from: C)
{….}  // a structured block of code
```

- Data copied into the device data environment at the beginning of the directive and at the end
- Inside the **target data** region, multiple **target** regions can work with the single data region

```
#pragma omp target data map(to: A,B) map(from: C)
{
    #pragma omp target
        {do lots of stuff with A, B and C}
    {do something on the host}
    #pragma omp target
        {do lots of stuff with A, B, and C}
}
```

# Target update directive

- You can update data between target regions with the target update directive.

```
#pragma omp target data map(to: A,B) map(from: C)
{
    #pragma omp target
        {do lots of stuf with A, B and C}

    #pragma omp  update from(A)

    host_do_something_with(A)

    #pragma omp update to(A)

    #pragma omp target
        {do lots of stuff with A, B, and C}
}
```

Copy A from the device onto the host.

Copy A on the host to A on the device. t

# Jacobi Solver (Par Targ Data, 1/2)

```
#pragma omp target data map(tofrom:xnew[0:Ndim],xold[0:Ndim]) \
                        map(to:A[0:Ndim*Ndim], b[0:Ndim] ,Ndim)
while((conv > TOL) && (iters<MAX_ITERS))
  {  iters++;
    xtmp  = xnew;    // don't copy arrays.
    xnew  = xold;    // just swap pointers.
    xold  = xtmp;
 #pragma update to(xnew[0:Ndim], xold[0:Ndim])
 #pragma omp target
      #pragma omp parallel for private(i,j)
    for (i=0; i<Ndim; i++){
        xnew[i] = (TYPE) 0.0;
        for (j=0; j<Ndim;j++){
            if(i!=j)
              xnew[i]+= A[i*Ndim + j]*xold[j];
        }
        xnew[i] = (b[i]-xnew[i])/A[i*Ndim+i];
    }
```

# Jacobi Solver (Par Targ Data, 2/2)

```
    //
    // test convergence
    //
    conv = 0.0;
#pragma omp update to(conv)
#pragma omp target
    #pragma omp parallel for private(i,tmp) reduction(+:conv)
    for (i=0; i<Ndim; i++){
        tmp  = xnew[i]-xold[i];
        conv += tmp*tmp;
    }
#pragma omp update from (conv)
    conv = sqrt((double)conv);

} \\ end while loop
```

# Jacobi Solver Results: summary

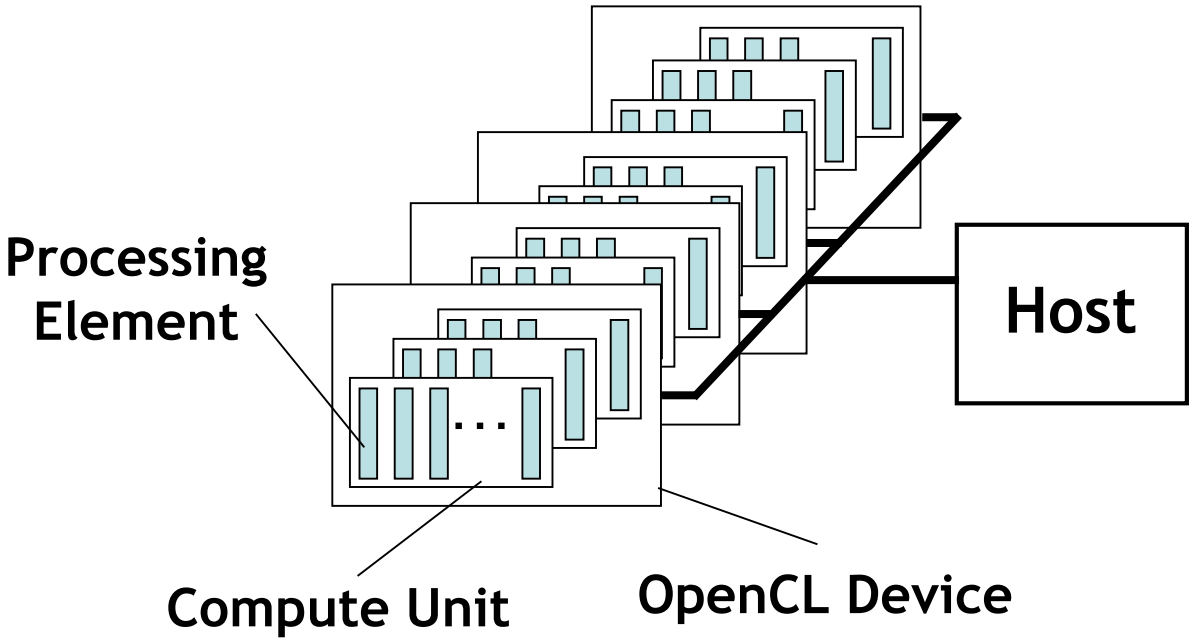
| System | Implementation | Ndim = 1024 | Ndim = 4096 |
|---|---|---|---|
| Intel® Xeon™ processor | parfor | 0.55 seconds | 21 seconds |
| | par_for | 0.36 seconds | 21 seconds |
| Intel® Xeon Phi™ co-processor (knights corner) | Target dir per loop | 134 seconds | Did not finish (> 40 minutes) |
| | Data region + target per loop | 3.4 seconds | 12.2 seconds |
| | Native par_for | 3.2 seconds | 5.3 seconds |
| | OpenCL Best | 0.97 seconds | 9.8 seconds |

Source: Tom Deakin and James Prices, University of Bristol, UK. All results with the Intel icc compiler. Compiler options -03.

# Mapping onto more complex devices

- So far, we have just “off-loaded” OpenMP code onto a general purpose CPU device that supports OpenMP multithreaded parallelism.
- How would we map OpenMP 4.0 onto a more specialized, throughput oriented device such as a GPU?

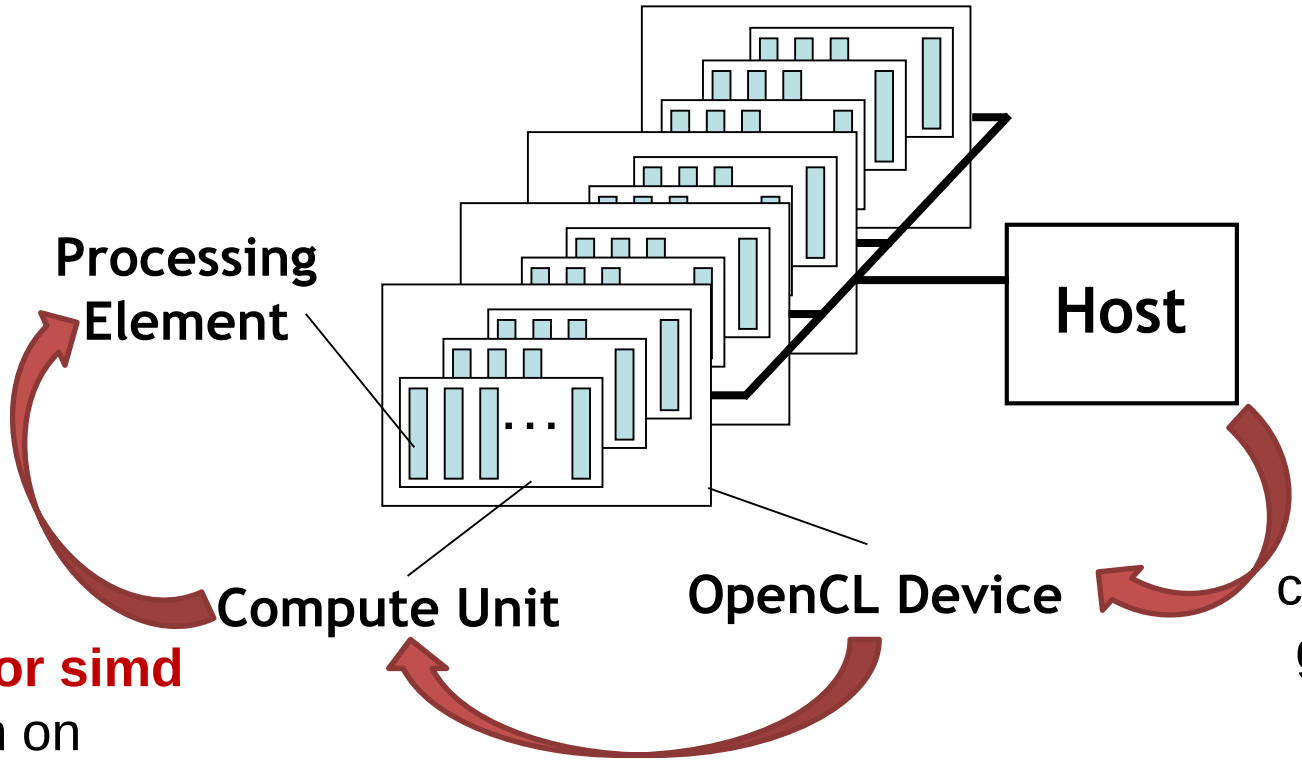
# OpenCL Platform Model




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

*the name OpenCL is the property of the Khronos Group

# OpenCL Platform Model and OpenMP




**Target** construct to get onto a device

**Parallel for simd** to run on processing elements + vector units

**Teams** construct to create a league of teams with one team of threads on each compute unit.

**Distribute** clause to assign work-groups to teams.

# Consider the familiar VADD example

```
#include<omp.h>
#include<stdio.h>
#define N  1024
int main()
{
   float a[N], b[N], c[N];
   int i;

// initialize a, b and c ….

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

// Test results, report results …

}
```

We will explore how to map this code onto Many-core processors (GPU and CPU) using the OpenMP constructs:

- target
- teams
- distribute

# 2 Constructs to control devices

- **teams** construct creates a league of thread teams:

  #pragma omp teams
- Supports the clauses:
  - num_teams(int) … the number of teams in the league
  - thread_limit(int) … max number of threads per team
  - Plus private(), firstprivate() and reduction()

- **distribute** construct distributes iterations of following loops to the master thread of each team in a league:

  #pragma omp distribute

  //immediately following for loop(s)
- Supports the clauses:
  - dist_schedule(static [, chunk] … the number of teams in the league.
  - collapse(int) … combine n closely nested loop into one before distributing.
  - Plus private(), firstprivate() and reduction()

# Vadd: OpenMP to OpenCL connection

```
#pragma omp target map(to:a,b) map(tofrom:c)

#pragma omp teams num_teams(NCU) thread_limit(NPE)

#pragma omp distribute
 for (ib=0;ib<N; ib=ib+wrk_grp_sz)

   #pragma omp parallel for simd
   for (i=ib;  i<ib+wrk_grp_sz; i++)
     c[i] += a[i] + b[i];
```

Offload to a device.

Describe a device … NCU compute units & NPE proc. elements per compute unit

Distribute work-groups to compute units

The body of this loop are the Individual work-items in a work-group

# Vadd: OpenMP to OpenCL connection

```
 int blksz=32, ib, Nblk;
 Nblk = N/blksz;
#pragma omp target map(to:a,b) map(tofrom:c)
    #pragma omp teams num_teams(NCU) thread_limit(NPE)

    #pragma omp distribute
     for (ib=0;ib<Nblk;ib++){
        int ibeg=ib*blksz;
        int iend=(ib+1)*blksz;
        if(ib==(Nblk-1))iend=N;

        #pragma omp parallel for simd
        for (i=ibeg; i<iend; i++)
           c[i] += a[i] + b[i];
    }
```

You can include any work-group wide code you want .. For example to explicitly control how iterations map onto work items in a work-group.

# Vadd: OpenMP to OpenCL connection

```
// A more compact way to write the VADD code, letting the runtime
//  worry about work-group details

#pragma omp target map(to:a,b) map(tofrom:c)
 #pragma omp teams distribute parallel for
      for (i=0; i<N; i++)
              c[i] += a[i] + b[i];
```

In many cases, you might be better off to just distribute the parallel loops to the league of teams and leave it to the runtime system to manage the details.  This would be more portable code as well.

# What about OpenACC?

- OpenACC is an Nvidia owned and driven solution to pragma driven programming of GPUs (not Open in the way OpenMP is).
- It started inside the OpenMP effort, but they pulled out and created their own competing standard (not a nice thing to do).
- It is focused on the GPU alone … ignoring the fact that what one really needs is a single source code base that handles CPU, GPU and Xeon-Phi-like manycore processors

# Jacobi iteration: OpenACC (GPU)

```
#pragma acc data copy(A), create(Anew)
while (err>tol && iter < iter_max){
   err = 0.0;
   #pragma acc parallel loop reduction(max:err)
   for(int j=1; j< n-1; j++){
      for(int i=1; i<M-1; i++){
         Anew[j][i] = 0.25* (A[j][i+1] + A[j][i-1]+
                             A[j-1][i] + A[j+1][i]);
         err = max(err,abs(Anew[j][i] - A[j][i]));
       }
    }
    #pragma acc parallel loop
    for(int j=1; j< n-1; j++){
      for(int i=1; i<M-1; i++){
         A[j][i] = Anew[j]i];
       }
    }
    iter ++;
}
```

Create a data region on the GPU. Copy A once onto the GPU, and create Anew on the device (no copy from host)

Copy A back out to host … but only once

Source: based on Mark Harris of NVIDIA®, "Getting Started with OpenACC", GPU technology Conf., 2012
The name "OpenACC" is the property of Nvidia.

# Jacobi iteration: OpenMP accelerator directives

```
#pragma omp target data map(A, Anew)
while (err>tol && iter < iter_max){
   err = 0.0;
   #pragma target
   #pragma omp parallel for reduction(max:err)
   for(int j=1; j< n-1; j++){
      for(int i=1; i<M-1; i++){
         Anew[j][i] = 0.25* (A[j][i+1] + A[j][i-1]+
                             A[j-1][i] + A[j+1][i]);
         err = max(err,abs(Anew[j][i] - A[j][i]));
       }
    }
    #pragma omp target
    #pragma omp parallel for
    for(int j=1; j< n-1; j++){
      for(int i=1; i<M-1; i++){
         A[j][i] = Anew[j]i];
       }
    }
    iter ++;
}
```

Create a data region on the GPU. Map A and Anew onto the target device

Uses existing OpenMP constructs such as parallel and for

Copy A back out to host … but only once

# OpenMP vs. OpenACC

- Ignore the misinformation you hear “out there”.
- The two approach have shared roots (based on pioneering work of Michael Wolf … then of PGI)
- You can construct exceptions, but for the most part, if you can express something in OpenACC, you can do so with OpenMP.
- So why not go with the open Standard that truely works across platforms?

The name “OpenACC” is the property of Nvidia.

# Plan

| | Module | Concepts | Exercises | |
|---|---|---|---|---|
| 8:30 | OpenMP core concepts | • Intro to OpenMP<br>• Creating threads | • Hello_world<br>• Pi_spmd | |
| 10:30 | Working with threads | • Synchronization<br>• Parallel loops<br>• Single, master, and more | • Pi_spmd_final<br>• Pi_loop | 10 AM Break |
| 1:00 | Managing data and tasks | • Data Environment<br>• tasks | • Mandelbrot set area<br>• Racy tasks<br>• Recursive pi | Noon Lunch |
| 3:30 | Understanding shared memory | • Memory Model<br>• Threadprivate | • Monte Carlo pi | 3 PM Break |
| | OpenMP beyond SMP | • SIMD<br>• Devices and OpenMP | • Jaobi Solver | |

**… Plus a set of "challenge problems" for the evening program.**

# 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. Optimizing matrix multiplication
  3. Traversing linked lists in different ways
  4. 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: 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
  - Alignment
  - Cache blocking
  - Loop unrolling
  - Vectorization
- Goal: Can you approach the peak performance of the computer?

# Challenge 3: Traversing linked lists

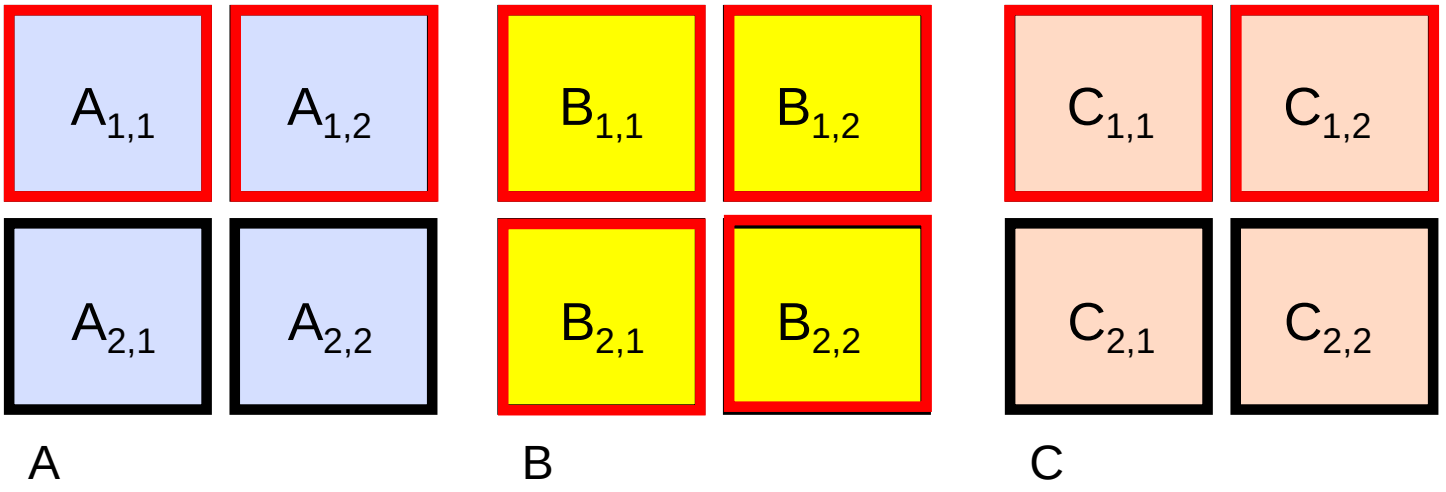
- Consider the program linked.c
  - Traverses a linked list, computing a sequence of Fibonacci numbers at each node
- Parallelize this program two different ways
  1. Use OpenMP tasks
  2. Use anything you choose in OpenMP *other than* tasks.
- The second approach (no tasks) can be difficult and may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

# Challenge 4: 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 4: Recursive matrix multiplication

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

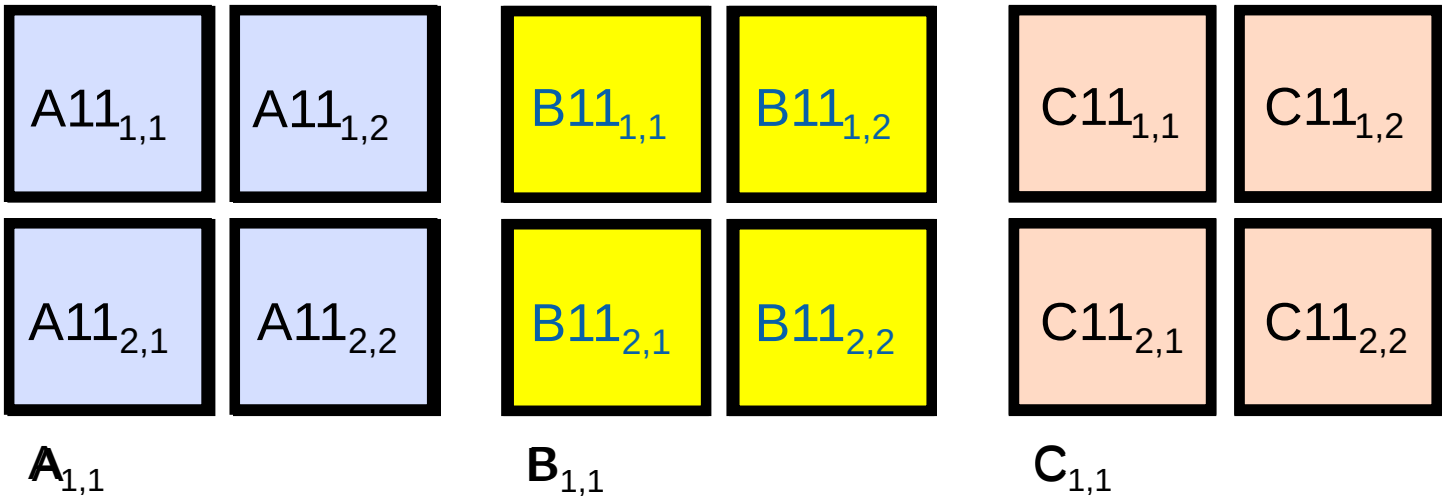



$$C_{1,1} = A_{1,1} \cdot B_{1,1} + A_{1,2} \cdot B_{2,1}$$
$$C_{2,1} = A_{2,1} \cdot B_{1,1} + A_{2,2} \cdot B_{2,1}$$

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

# Challenge 4: 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 4: 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

# Conclusion

- We have now covered the core features of the OpenMP specification
  - We've left off some minor details, but we've covered all major topics … remaining content you can pick up on your own
- Download the spec to learn more … the spec is filled with examples to support your continuing education
  - www.openmp.org
- Get involved:
  - Get your organization to join the OpenMP ARB
  - Work with us through cOMPunity

# Appendices

- Sources for additional information
- OpenMP History
- Solutions to exercises
  - Hello world
  - Simple SPMD Pi program
  - SPMD Pi without false sharing
  - Loop level Pi
  - Mandelbrot Set area
  - Racy tasks
  - Recursive pi program
  - Exercise: Monte Carlo pi and random numbers
  - Jacobi solver
- Challenge Problems
  - Molecular dynamics
  - Matrix multiplication
  - Linked lists
  - 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:

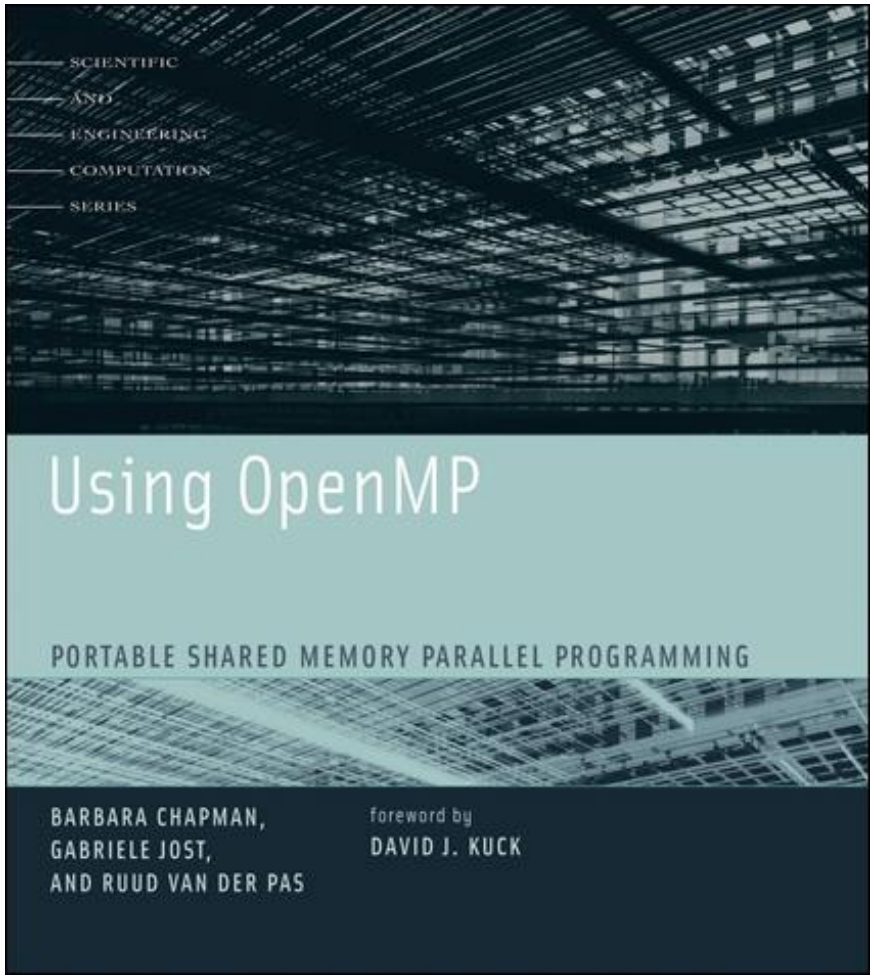
  www.openmp.org

- OpenMP User's Group (cOMPunity) URL:

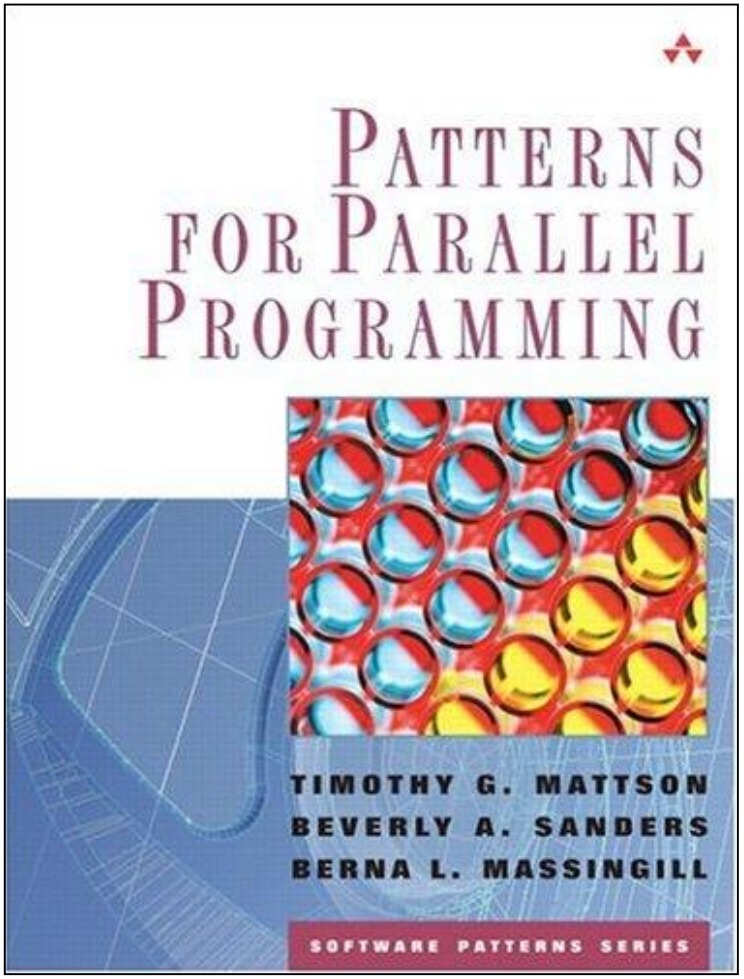
  www.compunity.org

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

# Books about OpenMP

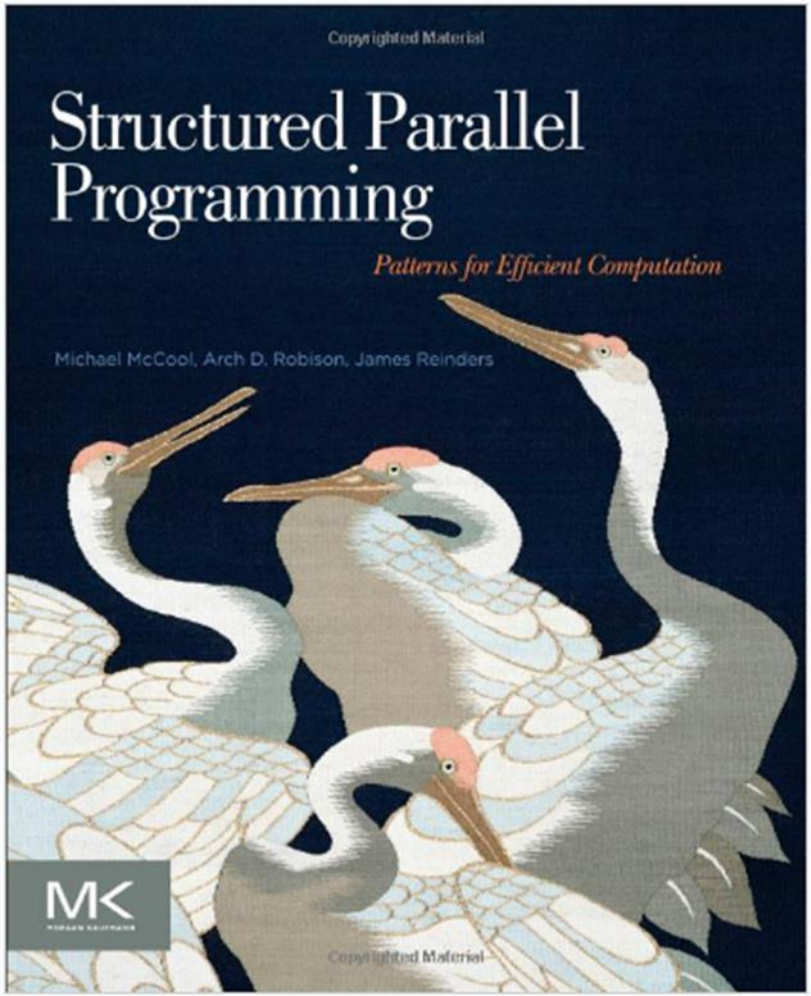



- A book about OpenMP by a team of authors at the forefront of OpenMP's evolution.

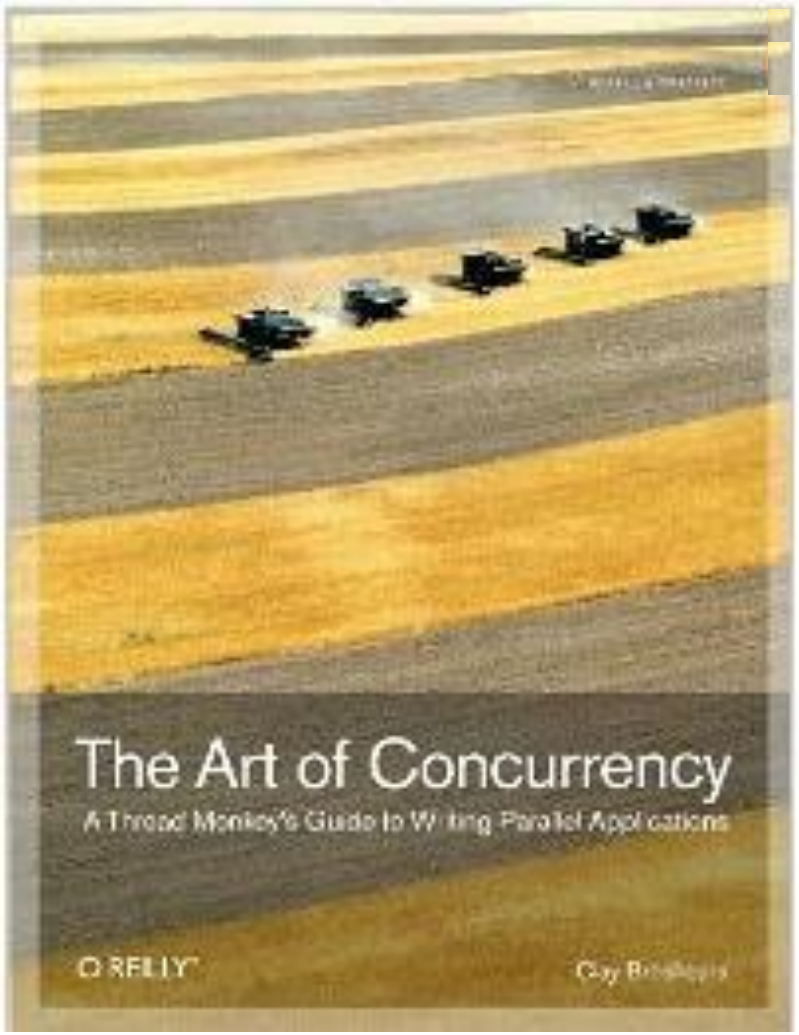



- A book about how to "think parallel" with examples in OpenMP, MPI and java

# 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

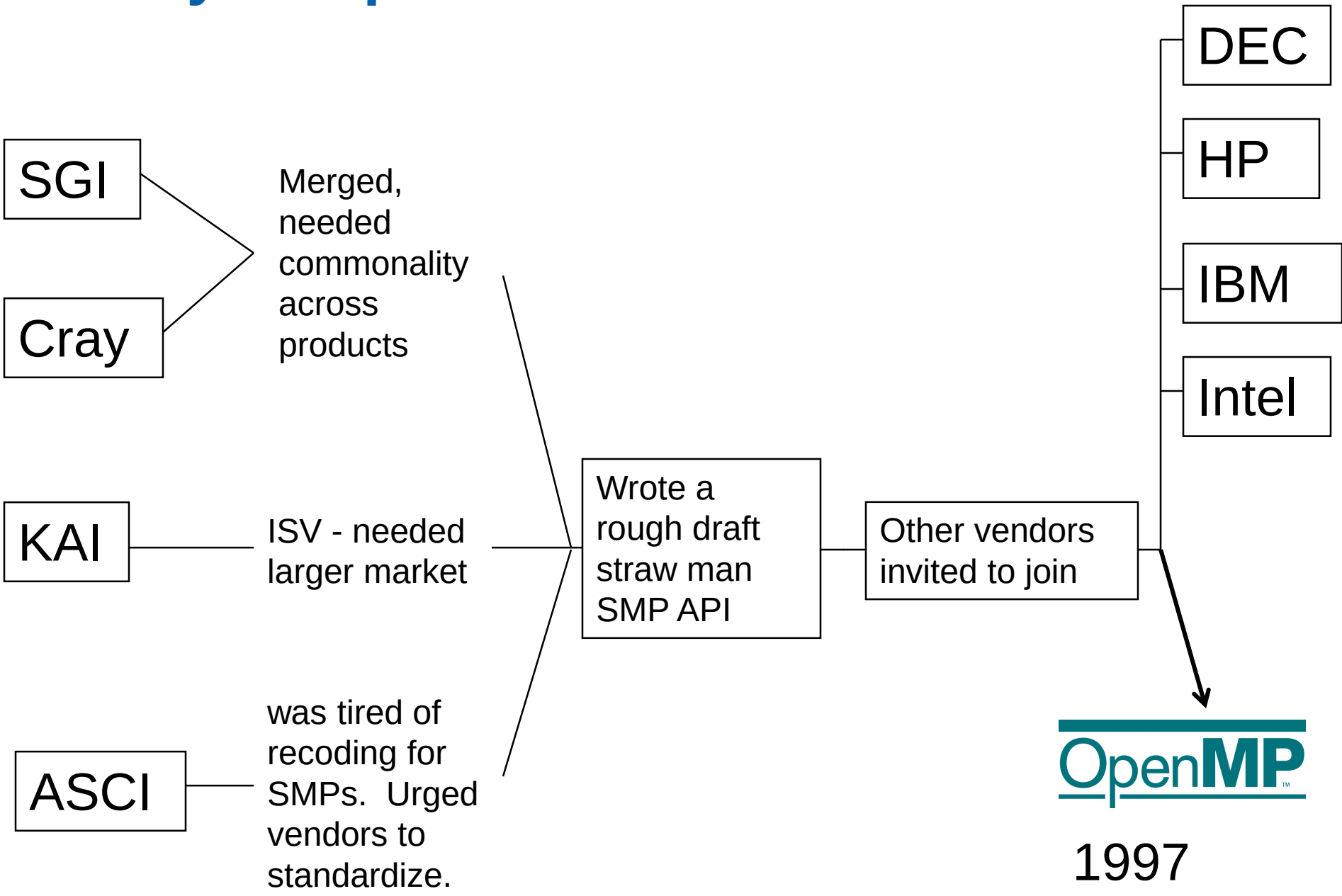
- Sources for additional information
- OpenMP History
- Solutions to exercises
  - Exercise 1: hello world
  - Exercise 2: Simple SPMD pi program
  - Exercise 3: SPMD pi without false sharing
  - Exercise 4: Loop level pi
  - Exercise 5: Mandelbrot Set area
  - Exercise 6: Recursive pi program
- Challenge 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

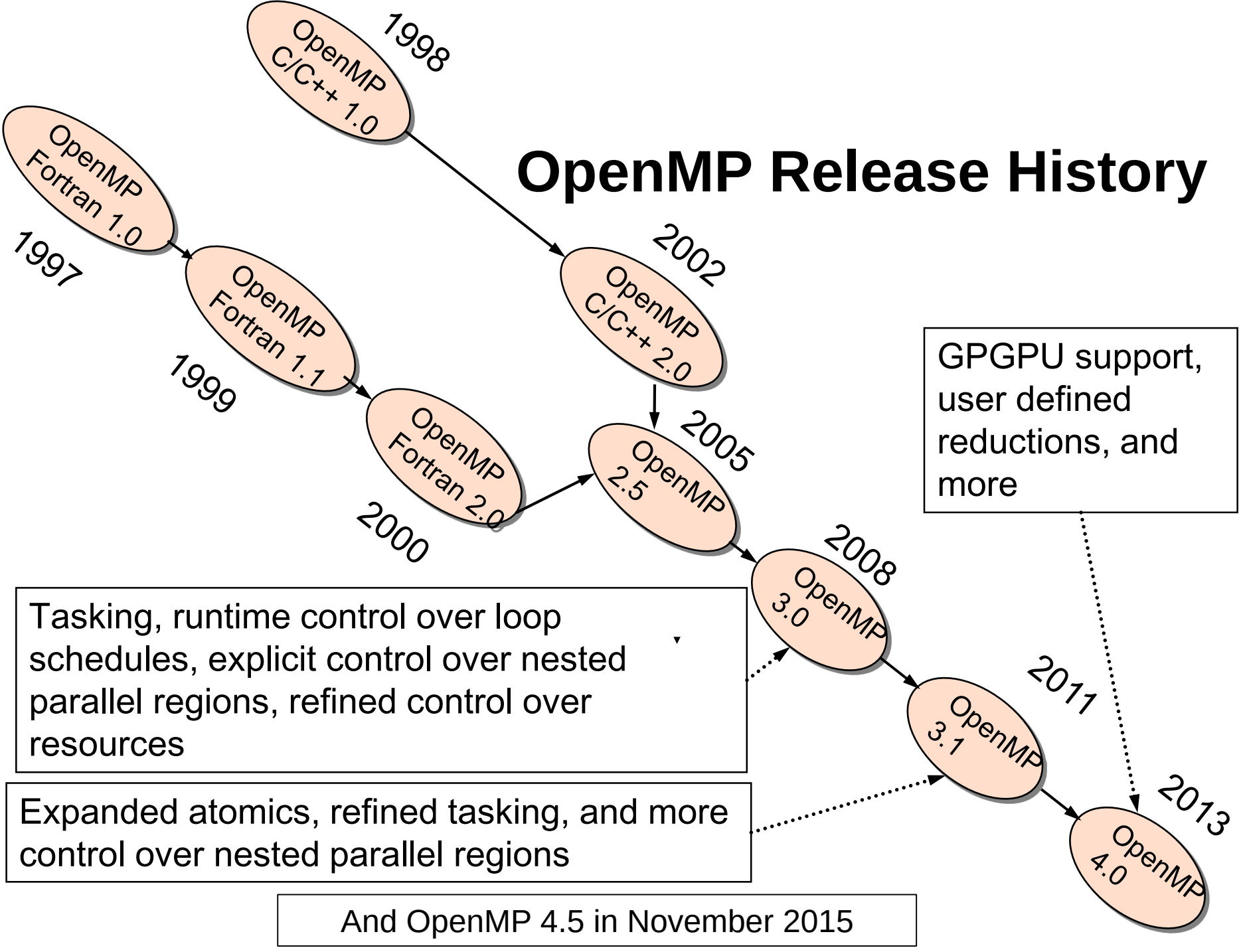
- OpenMP based upon SMP directive standardization efforts PCF and aborted ANSI X3H5 – late 80's
  - Nobody fully implemented either standard
  - Only a couple of partial implementations
- Vendors considered proprietary API's to be a competitive feature:
  - Every vendor had proprietary directives sets
  - 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



# OpenMP 4.0 ratified July 2013

- End of a long road? A brief rest stop along the way…
- Addresses several major open issues for OpenMP
- Do not break existing code unnecessarily
- Includes 106 passed tickets
  - Focused on major tickets initially
  - Builds on two comment drafts (“RC1” and “RC2”)
  - Many small tickets after RC2, a few large ones

# Overview of major 4.0 additions

- Device constructs
- SIMD constructs
- Cancellation
- Task dependences and task groups
- Thread affinity control
- User-defined reductions
- Initial support for Fortran 2003
- Support for array sections (including in C and C++)
- Sequentially consistent atomics
- Display of initial OpenMP internal control variables

# OpenMP 4.0 provides support for a wide range of devices

- Use `target` directive to offload a region

```
#pragma omp target [clause [[,] clause] …]
```

- Creates new data environment from enclosing device data environment
- Clauses support data movement and conditional offloading
  - `device` supports offload to a device other than default
  - Does not assume copies are made – memory may be shared with host
    - Does not copy if present in enclosing device data environment
  - `if` supports running on host if amount of work is small
- Other constructs support device data environment
  - `target data` places `map` list items in device data environment
  - `target update` ensures variable is consistent in host and device

# Several other device constructs support full-featured code

- Use `target declare` directive to create device versions

```
#pragma omp declare target
```

  - Can be applied to functions and global variables
  - Required for UDRs that use functions and execute on device
- `teams` directive creates multiple teams in a `target` region

```
#pragma omp teams [clause [[,] clause] …]
```

  - Work across teams only synchronized at end of `target` region
  - Useful for GPUs (corresponds to thread blocks)
- Use `distribute` directive to run loop across multiple teams

```
#pragma omp distribute [clause [[,] clause] …]
```

- Several combined/composite constructs simplify device use

# Example: OpenMP support for devices

## Jacobi iteration

```
#pragma omp target data map(A, Anew)
while (err>tol && iter < iter_max){
  err = 0.0;
#pragma omp target teams distribute parallel for reduction(max:err)
  for(int j=1; j< n-1; j++){
     for(int i=1; i<M-1; i++){
        Anew[j][i] = 0.25* (A[j][i+1] + A[j][i-1]+
                            A[j-1][i] + A[j+1][i]);
        err = max(err,abs(Anew[j][i] - A[j][i]));
       }
    }
    #pragma omp target teams distribute parallel for
    for(int j=1; j< n-1; j++){
      for(int i=1; i<M-1; i++){
         A[j][i] = Anew[j]i];
       }
    }
    iter ++;
}
```

Create a data region on the device. Map A and Anew onto the target device

Copy A back out to host … but only once

The "target teams" construct tells the compiler to pick the number of teams … which translates to thread blocks for CUDA.

# OpenMP 4.0 provides portable SIMD constructs

- Use `simd` directive to indicate a loop should be SIMDized

```
#pragma omp simd [clause [[,] clause] …]
```

- Execute iterations of following loop in SIMD chunks
  - Region binds to the current task, so loop is not divided across threads
  - SIMD chunk is set of iterations executed concurrently by a SIMD lanes
- Creates a new data environment
- Clauses control data environment, how loop is partitioned
  - `safelen(`*`length`*`)` limits the number of iterations in a SIMD chunk
  - `linear` lists variables with a linear relationship to the iteration space
  - `aligned` specifies byte alignments of a list of variables
  - `private`, `lastprivate`, `reduction`, `collapse` – usual meanings

# The `declare simd` construct generates SIMD functions

```
#pragma omp declare simd notinbranch
float min (float a, float b) {
   return a < b ? a : b;  }

#pragma omp declare simd notinbranch
float distsq (float x, float y) {
   return (x – y) * (x – y);  }
```

Notinbranch tells the compiler you can assume this function will not be called inside a branch statement .. i.e. all vector lanes will execute this function

- Compile library and use functions in a SIMD loop

```
void minex (float *a, float *b, float *c, float *d) {
  #pragma omp parallel for simd
  for (i = 0; i < N; i++)
    d[i] = min (distsq(a[i], b[i]), c[i]);
}
```

  - Creates implicit tasks of `parallel` region
  - Divides loop into SIMD chunks
  - Schedules SIMD chunks across implicit tasks
  - Loop is fully SIMDized by using SIMD versions of functions

# A simple UDR example

- Declare the reduction operator

```
#pragma omp declare reduction (merge : std::vector<int> :
   omp_out.insert(omp_out.end(), omp_in.begin(), omp_in.end()))
```

- Use the reduction operator in a `reduction` clause

```
void schedule (std::vector<int> &v, std::vector<int> &filtered) {
  #pragma omp parallel for reduction (merge : filtered)
  for (std:vector<int>::iterator it = v.begin(); it < v.end();
it++)
    if ( filter(*it) )  filtered.push_back(*it);
}
```

- Private copies created for a reduction are initialized to the identity that was specified for the operator and type
  - Default identity defined if `identity` clause not present
- Compiler uses `combiner` to combine private copies
  - `omp_out` refers to private copy that holds combined value
  - `omp_in` refers to the other private copy

# OpenMP 4.0 includes initial support for Fortran 2003

- Added to list of base language versions
- Have a list of unsupported Fortran 2003 features
  - List initially included 24 items (some big, some small)
  - List has been reduced to 14 items
  - List in specification reflects approximate OpenMP Next priority
  - Priorities determined by importance and difficulty
- Plan: Reduce list and ideally provide full support in 5.0
  - Many small changes throughout; Support:
    - Procedure pointers
    - Renaming operators on the `USE` statement
    - `ASSOCIATE` construct
    - `VOLATILE` attribute
    - Structure constructors
  - Will support Fortran 2003 object-oriented features next
    - The biggest issue
    - Considering concurrent reexamination of C++ support

# Plan for OpenMP specifications

- OpenMP Tools Interface Technical Report
  - Released in March 2014
  - Working towards adoption in 5.0
- TR3: Initial OpenMP 4.5 Comment Draft
  - Changes adopted in time frame of SC14
  - Provided clear guidance to begin 4.1 implementations
- Final OpenMP 4.5 Comment Draft: Released Late Last Month
- OpenMP 4.5
  - Clarifications, refinements and minor extensions to existing specification
  - Major focus is device construct refinements
  - Do not break existing code
  - Released by SC15
- OpenMP 5.0
  - Address several major open issues for OpenMP
  - Expect less significant advance than 4.0 from 3.1/3.0
  - Do not break existing code unnecessarily
  - Targeting release for SC15 (somewhat ambitious)

# OpenMP 4.5 included many refinements

- 92 tickets have been passed
  - Many refinements to device support
  - Reflects improved efficiency due to LaTex conversion
- Many clarifications and minor enhancements
  - Handled several items from Fortran 2003 list
  - SIMD and tasking extensions and refinements
  - Reductions for C/C++ arrays and templates
  - Runtime routines to support cancelation and affinity
- Some new features are being added
  - Support for DOACROSS loops
  - Can divide loop into tasks with `taskloop` construct

# TR3 (initial OpenMP 4.1 comment draft) refines device constructs

- Adds flush to several device constructs
- Supports unstructured data movement
- Can now require update/assignment for map (`always`)
- Improves asynchronous execution
  - In 4.0, could have a `task` region with only a `target` region
  - `target` and other device regions are now tasks
    - By default, undeferred
    - Can use `nowait` and `depend` clauses
- Many clarifications and minor corrections

# Final OpenMP 4.1 comment draft further refines device constructs

- `memcpy` API to support manual mapping
- Device pointers (provides interoperability with CUDA and OpenCL libraries)
- Mapping structure elements
- Tweaks to device environment support, including:
  - Default for scalar variables: `firstprivate`
  - `link` clause for `declare target` construct
- New combined constructs
- Other miscellaneous usability features

# More significant topics are being considered for OpenMP 5.0

- Updates to support latest C/C++ standards
- More tasking advances (support for event loops)
- General error model
- Continued improvements to device support
- Performance and debugging tools support
- Interoperability and composability
- Locality and affinity
- Transactional memory
- Additional looping constructs and refinements

# 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
  - Hello world
  - Simple SPMD Pi program
  - SPMD Pi without false sharing
  - Loop level Pi
  - Mandelbrot Set area
  - Racy tasks
  - Recursive pi program
  - Exercise: Monte Carlo pi and random numbers
  - Jacobi solver
- Challenge Problems
  - Molecular dynamics
  - Matrix multiplication
  - Linked lists
  - Recursive matrix multiplication
- Fortran and OpenMP
- Mixing OpenMP and MPI
- Compiler notes

# Hello world Exercise: 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

Sample Output:

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

world(0)

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

world(2)

Runtime library function to return a thread ID.

End of the Parallel region

# Appendices

- Sources for additional information
- OpenMP History
- Solutions to exercises
  - Hello world
  - Simple SPMD Pi program
  - SPMD Pi without false sharing
  - Loop level Pi
  - Mandelbrot Set area
  - Racy tasks
  - Recursive pi program
  - Exercise: Monte Carlo pi and random numbers
  - Jacobi solver
- Challenge Problems
  - Molecular dynamics
  - Matrix multiplication
  - Linked lists
  - 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.

# Solution: 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
  - Hello world
  - Simple SPMD Pi program
  - SPMD Pi without false sharing
  - Loop level Pi
  - Mandelbrot Set area
  - Racy tasks
  - Recursive pi program
  - Exercise: Monte Carlo pi and random numbers
  - Jacobi solver
- Challenge Problems
  - Molecular dynamics
  - Matrix multiplication
  - Linked lists
  - 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.

# Solution: 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
  - Hello world
  - Simple SPMD Pi program
  - SPMD Pi without false sharing
  - Loop level Pi
  - Mandelbrot Set area
  - Racy tasks
  - Recursive pi program
  - Exercise: Monte Carlo pi and random numbers
  - Jacobi solver
- Challenge Problems
  - Molecular dynamics
  - Matrix multiplication
  - Linked lists
  - Recursive matrix multiplication
- Fortran and OpenMP
- Mixing OpenMP and MPI
- Compiler notes

# Loopy Pi: 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;
}
```

# Loopy pi: 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
  - Hello world
  - Simple SPMD Pi program
  - SPMD Pi without false sharing
  - Loop level Pi
  - Mandelbrot Set area
  - Racy tasks
  - Recursive pi program
  - Exercise: Monte Carlo pi and random numbers
  - Jacobi solver
- Challenge Problems
  - Molecular dynamics
  - Matrix multiplication
  - Linked lists
  - Recursive matrix multiplication
- Fortran and OpenMP
- Mixing OpenMP and MPI
- Compiler notes

# Solution: 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!!!!

# Solution: 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.

# Solution: 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
  - Hello world
  - Simple SPMD Pi program
  - SPMD Pi without false sharing
  - Loop level Pi
  - Mandelbrot Set area
  - Racy tasks
  - Recursive pi program
  - Exercise: Monte Carlo pi and random numbers
  - Jacobi solver
- Challenge Problems
  - Molecular dynamics
  - Matrix multiplication
  - Linked lists
  - Recursive matrix multiplication
- Fortran and OpenMP
- Mixing OpenMP and MPI
- Compiler notes

# Racy Tasks

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

Create a team of threads

One thread creates task, others wait at the corresponding barrier … ready to execute tasks

All tasks complete before moving past the barrier associated with the single

# Appendices

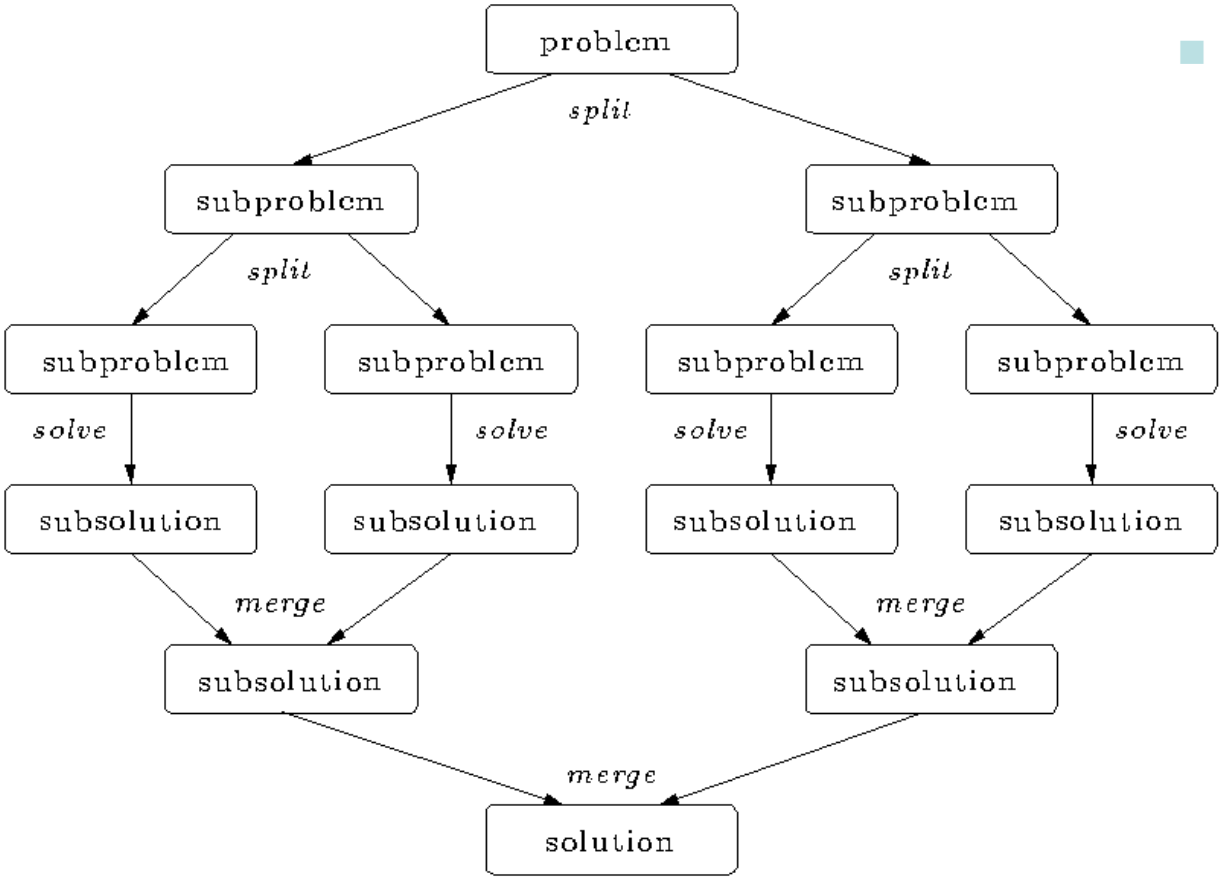
- Sources for additional information
- OpenMP History
- Solutions to exercises
  - Hello world
  - Simple SPMD Pi program
  - SPMD Pi without false sharing
  - Loop level Pi
  - Mandelbrot Set area
  - Racy tasks
  - Recursive pi program
  - Exercise: Monte Carlo pi and random numbers
  - Jacobi solver
- Challenge Problems
  - Molecular dynamics
  - Matrix multiplication
  - Linked lists
  - 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
  - Hello world
  - Simple SPMD Pi program
  - SPMD Pi without false sharing
  - Loop level Pi
  - Mandelbrot Set area
  - Racy tasks
  - Recursive pi program
  - Exercise: Monte Carlo pi and random numbers
  - Jacobi solver
- Challenge Problems
  - Molecular dynamics
  - Matrix multiplication
  - Linked lists
  - Recursive matrix multiplication
- Fortran and OpenMP
- Mixing OpenMP and MPI
- Compiler notes

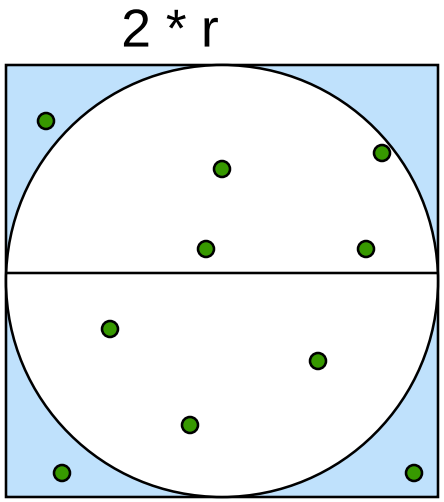
# Computers and random numbers

- 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;
```

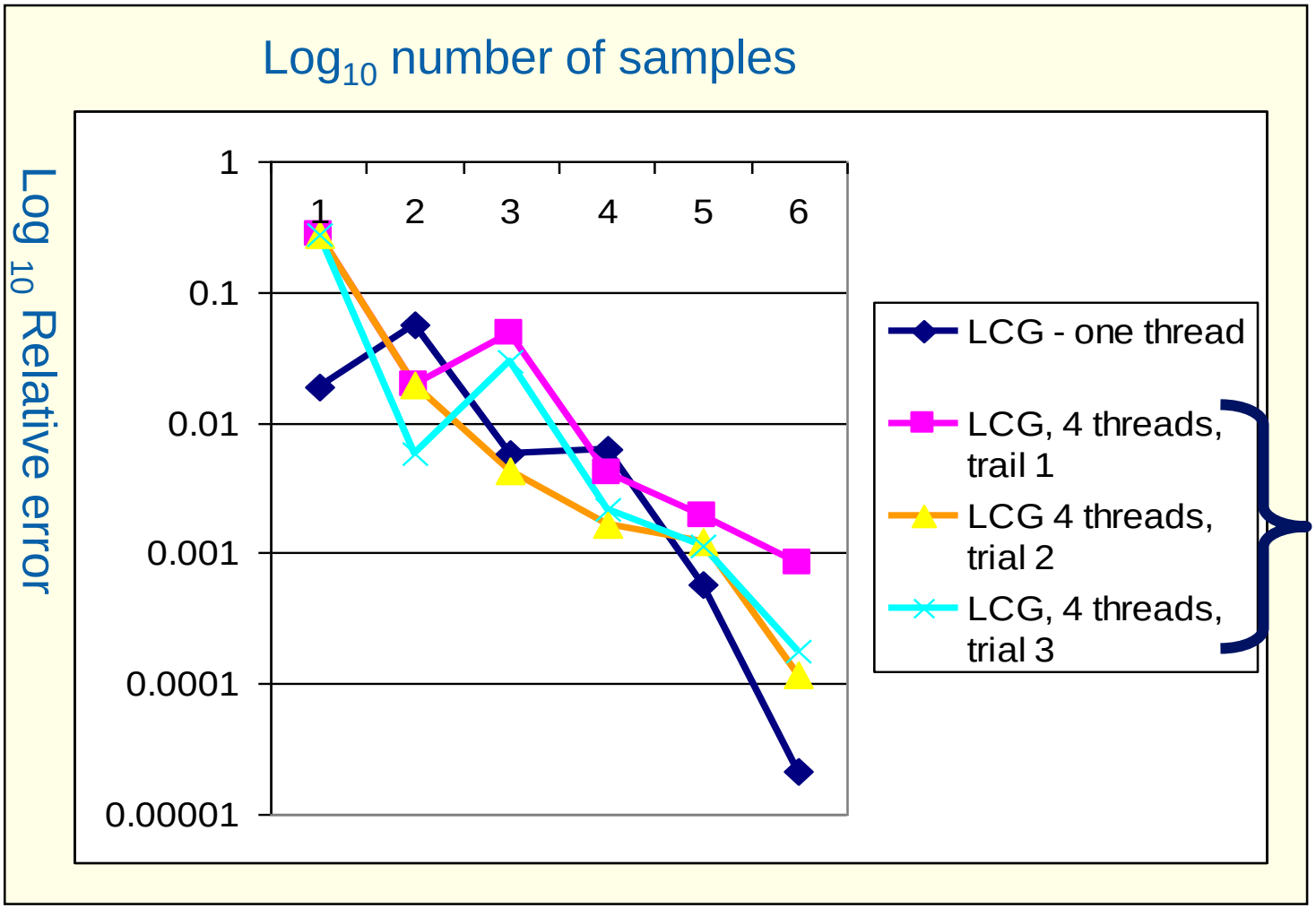
- If you pick the multiplier and addend correctly, LCG has a period of PMOD.
- Picking good LCG parameters is complicated, so look it up (Numerical Recipes is a good source). I used the following:
  - MULTIPLIER = 1366
  - ADDEND = 150889
  - PMOD = 714025

# 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.

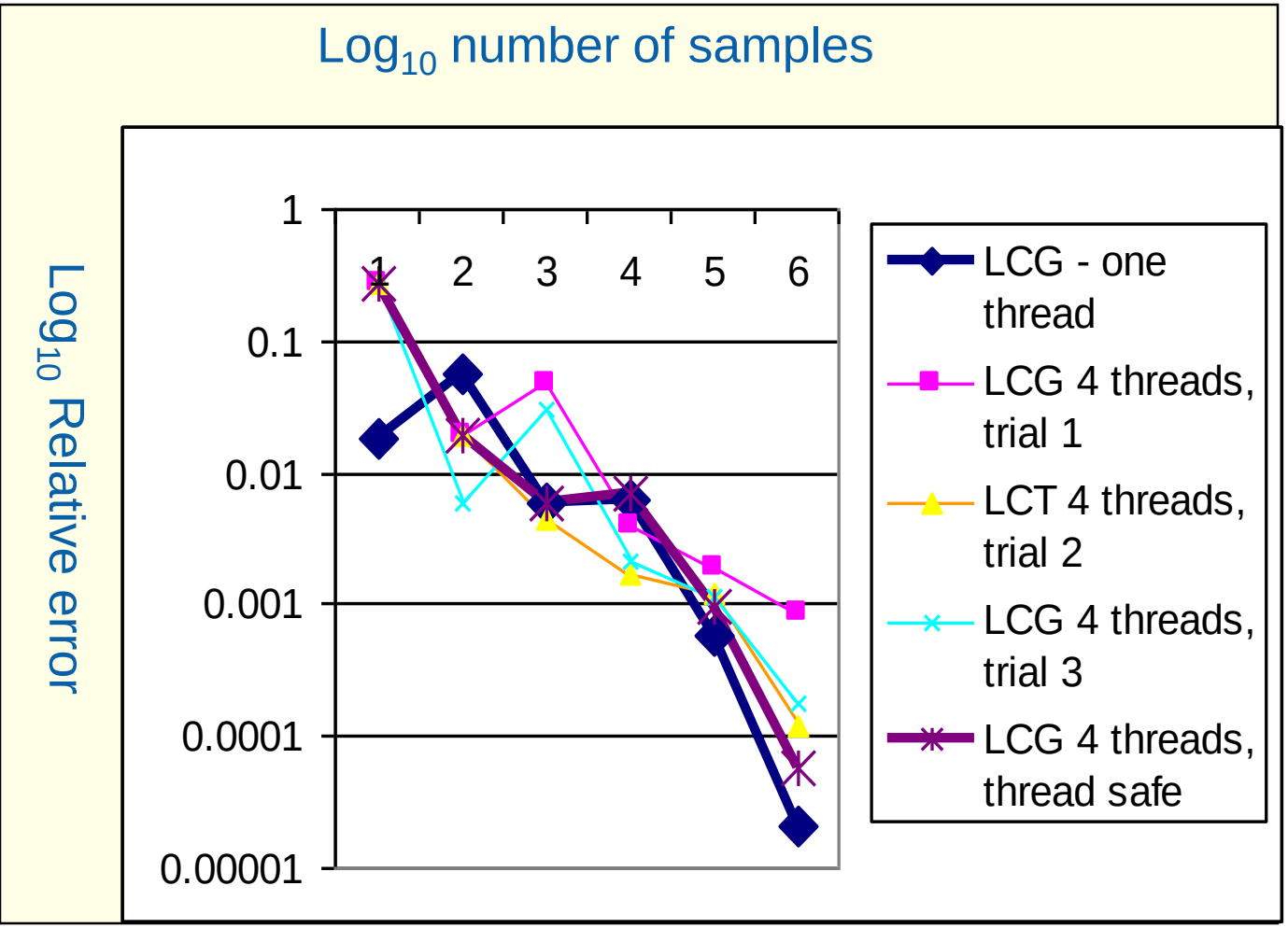
# LCG code: threadsafe version

```
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.

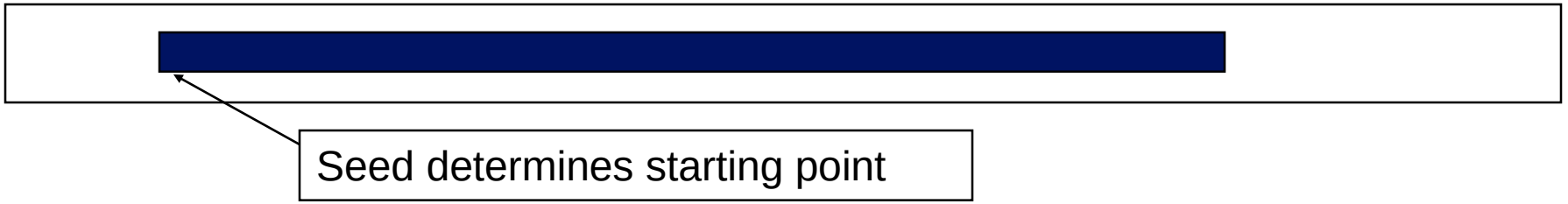
But for large number of samples, its quality is lower than the one thread result!

Why?

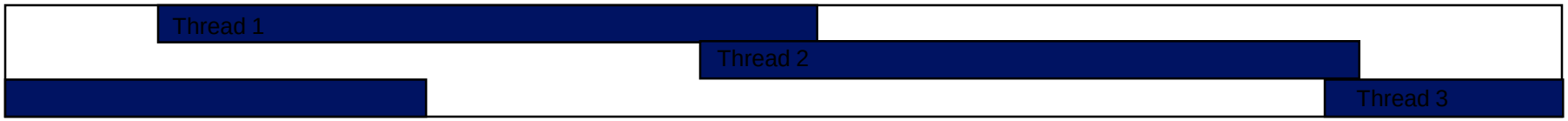
# Pseudo Random Sequences

- 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.

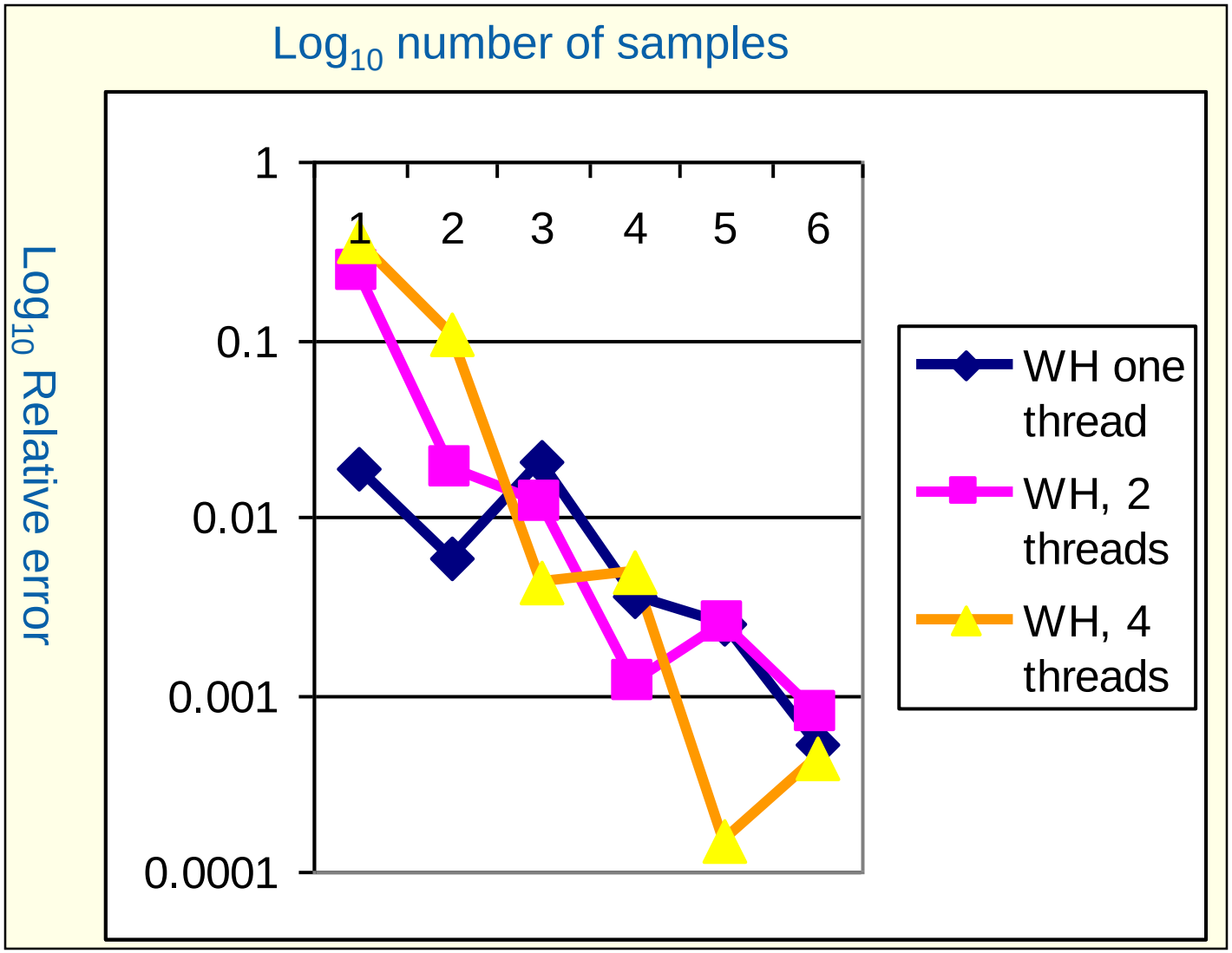
Delete the stream when you are done

# 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
  - Hello world
  - Simple SPMD Pi program
  - SPMD Pi without false sharing
  - Loop level Pi
  - Mandelbrot Set area
  - Racy tasks
  - Recursive pi program
  - Exercise: Monte Carlo pi and random numbers
  - Jacobi solver
- Challenge Problems
  - Molecular dynamics
  - Matrix multiplication
  - Linked lists
  - Recursive matrix multiplication
- Fortran and OpenMP
- Mixing OpenMP and MPI
- Compiler notes

# Appendices

- Sources for additional information
- OpenMP History
- Solutions to exercises
  - Hello world
  - Simple SPMD Pi program
  - SPMD Pi without false sharing
  - Loop level Pi
  - Mandelbrot Set area
  - Racy tasks
  - Recursive pi program
  - Exercise: Monte Carlo pi and random numbers
  - Jacobi solver
- Challenge Problems
  - Molecular dynamics
  - Matrix multiplication
  - Linked lists
  - Recursive matrix multiplication
- Fortran and OpenMP
- Mixing OpenMP and MPI
- Compiler notes

# Molecular dynamics: Solution

Compiler will warn you if you have missed some variables

```
#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) {
     ……
```

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

# Molecular dynamics : 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

# Molecular dynamics : 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

# Molecular dynamics : 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

# Molecular dynamics : 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
  - Hello world
  - Simple SPMD Pi program
  - SPMD Pi without false sharing
  - Loop level Pi
  - Mandelbrot Set area
  - Racy tasks
  - Recursive pi program
  - Exercise: Monte Carlo pi and random numbers
  - Jacobi solver
- Challenge Problems
  - Molecular dynamics
  - Matrix multiplication
  - Linked lists
  - Recursive matrix multiplication
- Fortran and OpenMP
- Mixing OpenMP and MPI
- Compiler notes

# Challenge: 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

There is much more that can be done. This is really just the first and most simple step

```
#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
  - Hello world
  - Simple SPMD Pi program
  - SPMD Pi without false sharing
  - Loop level Pi
  - Mandelbrot Set area
  - Racy tasks
  - Recursive pi program
  - Exercise: Monte Carlo pi and random numbers
  - Jacobi solver
- Challenge Problems
  - Molecular dynamics
  - Matrix multiplication
  - Linked lists
  - Recursive matrix multiplication
- Fortran and OpenMP
- Mixing OpenMP and MPI
- Compiler notes

# Exercise: traversing linked lists

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

- 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

# Exercise: traversing linked lists

- Consider the program linked.c
  - Traverses a linked list computing a sequence of Fibonacci numbers at each node.
- Parallelize this program 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
  - Hello world
  - Simple SPMD Pi program
  - SPMD Pi without false sharing
  - Loop level Pi
  - Mandelbrot Set area
  - Racy tasks
  - Recursive pi program
  - Exercise: Monte Carlo pi and random numbers
  - Jacobi solver
- Challenge Problems
  - Molecular dynamics
  - Matrix multiplication
  - Linked lists
  - 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
  - Hello world
  - Simple SPMD Pi program
  - SPMD Pi without false sharing
  - Loop level Pi
  - Mandelbrot Set area
  - Racy tasks
  - Recursive pi program
  - Exercise: Monte Carlo pi and random numbers
  - Jacobi solver
- Challenge Problems
  - Molecular dynamics
  - Matrix multiplication
  - Linked lists
  - 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

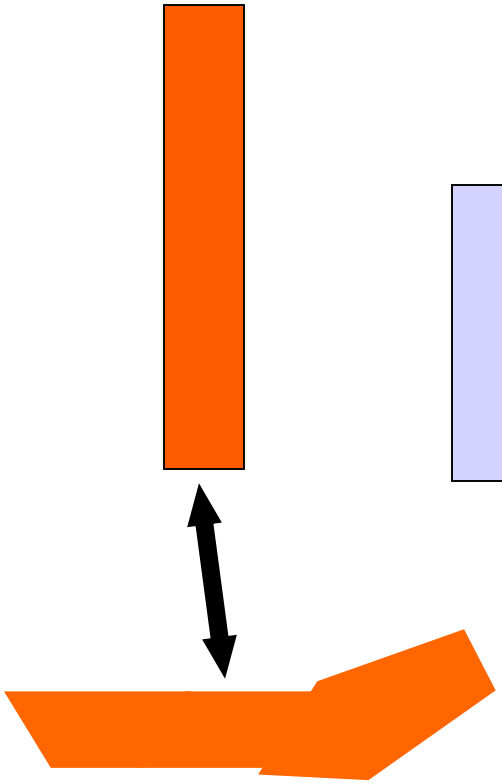
- The include file or module defines parameters
  - Integer parameter omp_locl_kind
  - Integer parameter omp_nest_lock_kind
  - Integer parameter omp_sched_kind
  - Integer parameter openmp_version
    - With value that matches C's _OPEMMP macro
- Fortran interfaces are similar to those used with C
  - 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
  - Hello world
  - Simple SPMD Pi program
  - SPMD Pi without false sharing
  - Loop level Pi
  - Mandelbrot Set area
  - Racy tasks
  - Recursive pi program
  - Exercise: Monte Carlo pi and random numbers
  - Jacobi solver
- Challenge Problems
  - Molecular dynamics
  - Matrix multiplication
  - Linked lists
  - Recursive matrix multiplication
- Fortran and OpenMP
- Mixing OpenMP and MPI
- Compiler notes

# How do people mix MPI and OpenMP?

A sequential program working on a data set



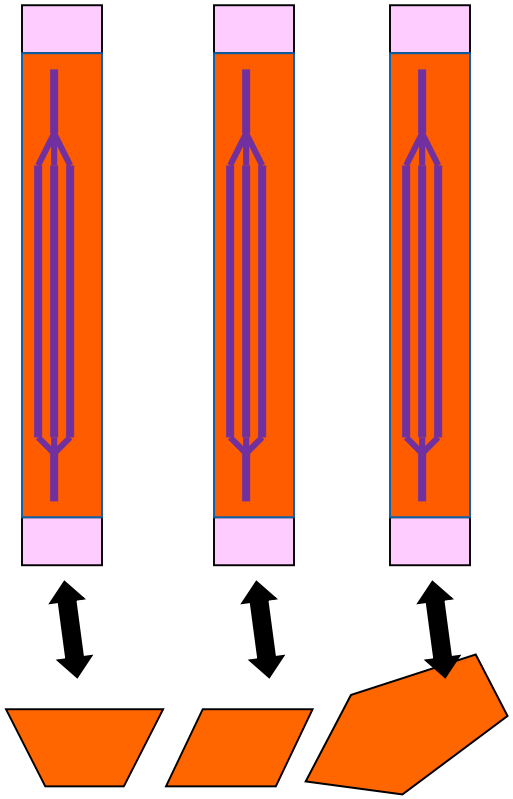
Replicate the program.

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
  - Hello world
  - Simple SPMD Pi program
  - SPMD Pi without false sharing
  - Loop level Pi
  - Mandelbrot Set area
  - Racy tasks
  - Recursive pi program
  - Exercise: Monte Carlo pi and random numbers
  - Jacobi solver
- Challenge Problems
  - Molecular dynamics
  - Matrix multiplication
  - Linked lists
  - 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: OSX and Linux

- OSX and icc:
  - > icc -qopenmp foo.c
  - > export OMP_NUM_THREADS=4
  - > ./a.out
  - > -Fa to generate assembly,
  - > -qopemp-simd … to use vectors but not threads (hence no threads overhead)

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

# Gnu compilers on Apple laptops

- The default compilers on apple systems (included with xcode) based on clang do not always support OpenMP.
- The gnu compilers closely track the latest OpenMP standards. To load them onto an apple laptop (example shown for gcc5):
  - Download xcode with command line tools (from Apple) and macports (from macports.org)
  - sudo port install gcc5
  - sudo port select –set gcc mp-gcc5
  - gcc –fopenmp <<file names>>
- A copy of the OpenMP exercises are on github
  - git clone https://github.com/tgmattso/OpenMP_Exercises.git

# 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)