The OpenMP* Common Core:
A hands on exploration

Tim Mattson

Intel Corp.

timothy.g.mattson@ intel.com

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

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

Use cooley … or even your own laptop (Apple or Linux … windows is difficult). For Apple laptops, use gcc, not clang

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

Warning: by default Xcode renames gcc to Apple's clang compiler. Use Homebrew to load a real, gcc compiler.

- On cooley …. An X86 cluster (Two 2.4 GHz Intel Haswell E5-2620 v3 processors per node with 6 cores per CPU, 12 cores total) with 384 GB RAM

  ssh <<login_name>>@cooley.alcf.anl.gov
- The OpenMP compiler

  Add the following line to ".soft.cooley" and then run the resoft command

  +intel-composer-xe

  icc –qopenmp –O3 << file names>>

Note: the gcc compiler works for OpenMP on Cooley:
gcc –fopenmp <<file names>>

- Copy the exercises to your home directory

  $ cp /projects/ATPESC2019/OMP_Exercises
- 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 ATPESC2019 –n 1 –t 30 -I

Note: this is a capital "I" (eye) not a lower case L

You can use theta as well, but the interactive shell runs on "the mom node". You need to use "aprun" to submit jobs.

Preliminaries: Part 1

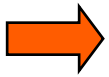
- Disclosures
  - The views expressed in this tutorial are those of the people delivering the tutorial.
    - We are not speaking for our employers.
    - We are not speaking for the OpenMP ARB
- We take these tutorials VERY seriously:
  - Help us improve … tell us how you would make this tutorial better.

Preliminaries: Part 2

- Our plan for the day .. Active learning!
  - We will mix short lectures with short exercises.
  - You will use your laptop to connect to a multiprocessor server.

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

Outline



- Introduction to OpenMP
- Creating Threads
- Synchronization
- Parallel Loops
- Data environment
- Memory model
- Irregular Parallelism and tasks
- Recap
- Beyond the common core:
  - Worksharing revisited
  - Synchronization: More than you ever wanted to know
  - Thread private data
  - Going deeper into tasks

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

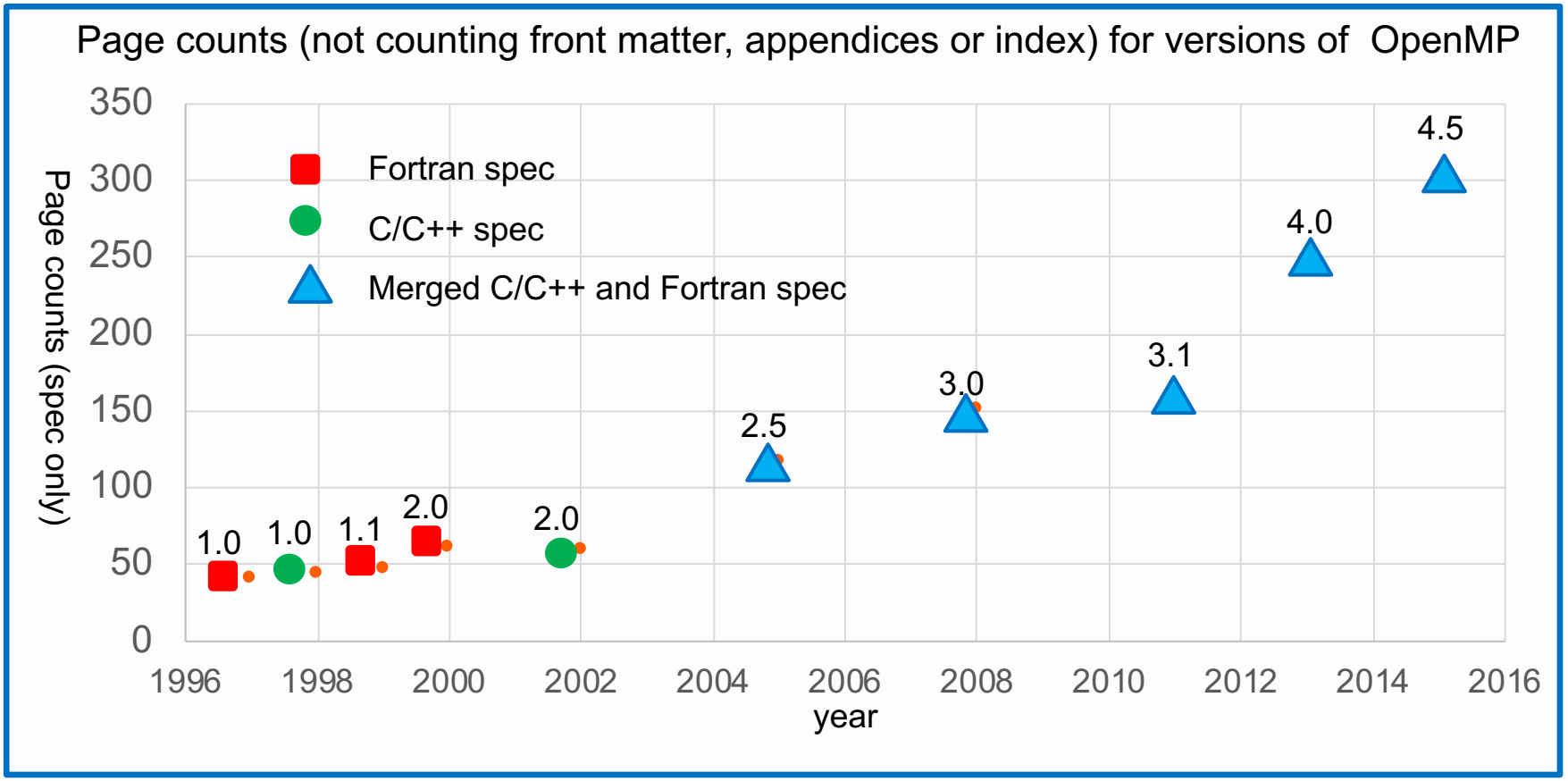
- A set of compiler directives and library routines for parallel application programmers
- Greatly simplifies writing multi-threaded (MT) programs in Fortran, C and C++
- Standardizes established SMP practice + vectorization and heterogeneous device programming

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

The growth of complexity in OpenMP

- OpenMP started out in 1997 as a simple interface for application programmers more versed in their area of science than computer science.

- The complexity has grown over the years! It has become overwhelming

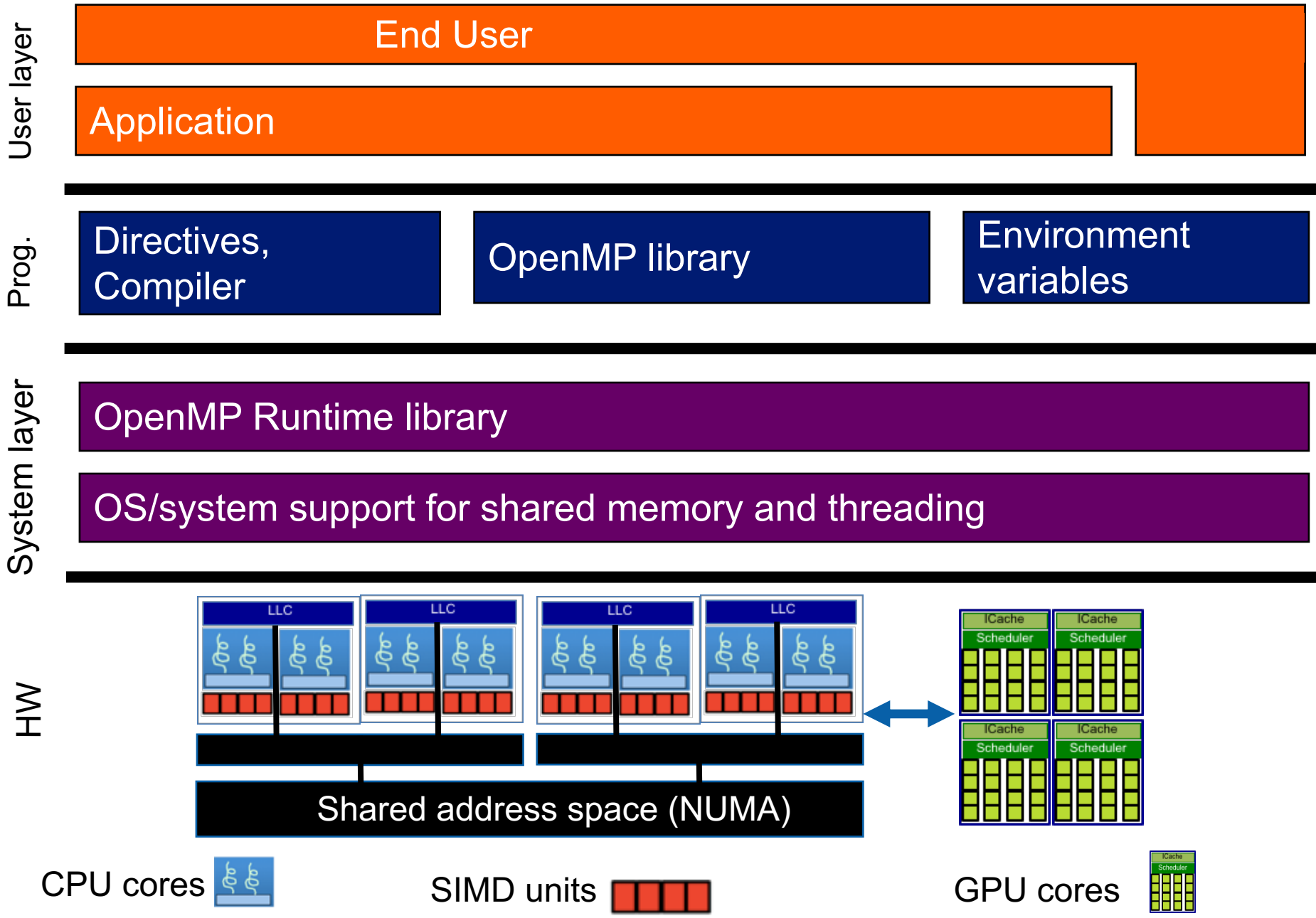


A new strategy: Start with the 19 constructs OpenMP programmers use all the time (**OpenMP Common Core**), then dive into the rest of the spec. as needed.

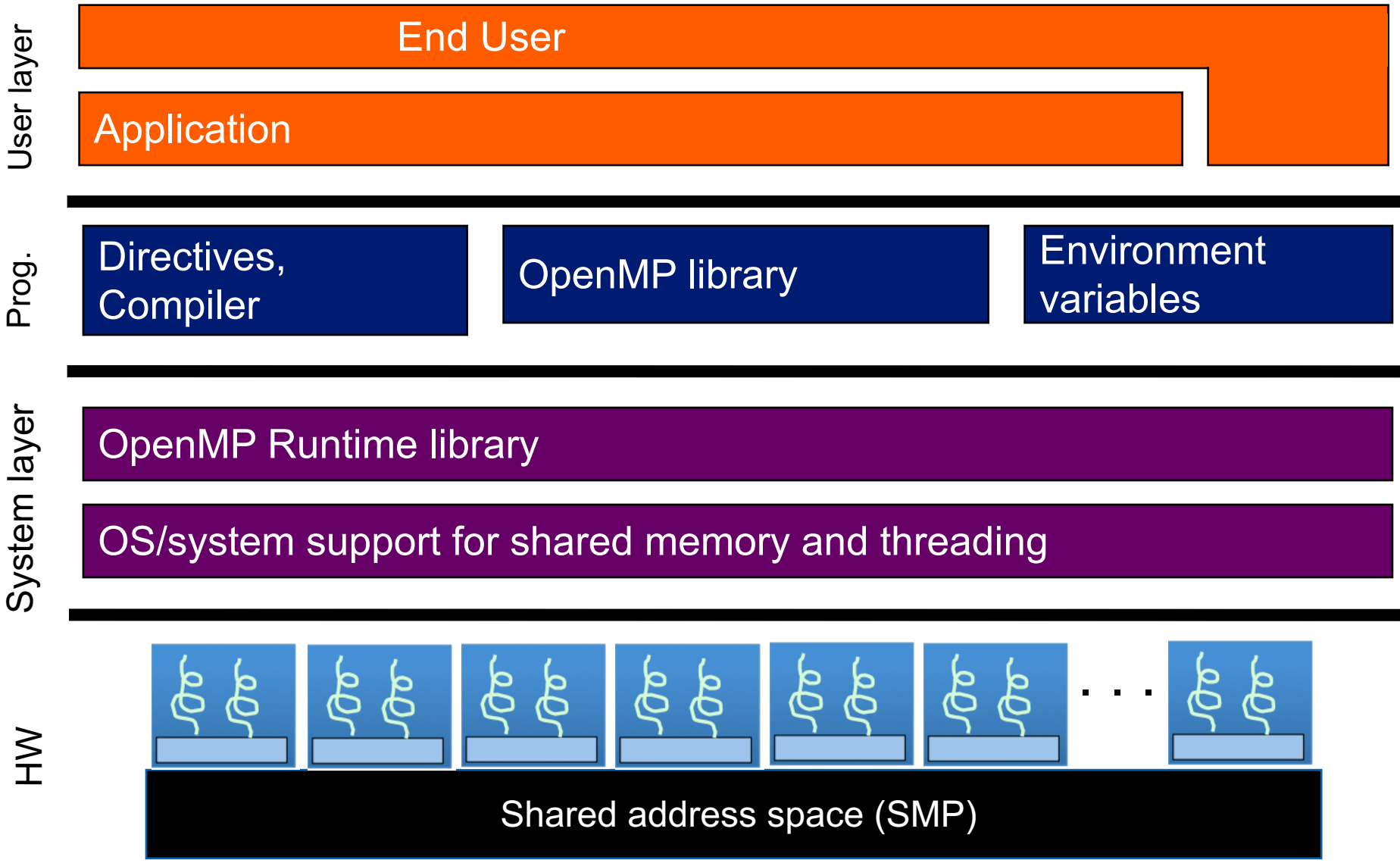
The OpenMP Common Core: Most OpenMP programs only use these 19 items

OpenMP pragma, function, or clause	Concepts
#pragma omp parallel	Parallel region, teams of threads, structured block, interleaved execution across threads
int omp_get_thread_num() int omp_get_num_threads()	Create threads with a parallel region and split up the work using the number of threads and thread ID
double omp_get_wtime()	Speedup and Amdahl's law. False Sharing and other performance issues
setenv OMP_NUM_THREADS N	Internal control variables. Setting the default number of threads with an environment variable
#pragma omp barrier #pragma omp critical	Synchronization and race conditions. Revisit interleaved execution.
#pragma omp for #pragma omp parallel for	Worksharing, parallel loops, loop carried dependencies
reduction(op:list)	Reductions of values across a team of threads
schedule(dynamic [,chunk]) schedule (static [,chunk])	Loop schedules, loop overheads and load balance
private(list), firstprivate(list), shared(list)	Data environment
nowait	Disabling implied barriers on workshare constructs, the high cost of barriers, and the flush concept (but not the flush directive)
#pragma omp single	Workshare with a single thread
#pragma omp task #pragma omp taskwait	Tasks including the data environment for tasks.

OpenMP basic definitions: Basic Solution stack



OpenMP basic definitions: Basic Solution stack



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

OpenMP basic syntax

- Most of the constructs in OpenMP are compiler directives.

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

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

Exercise, Part A: Hello world

Verify that your environment works

- Write a program that prints “hello world”.

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




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

}
```

Exercise, Part B: Hello world

Verify that your OpenMP environment works

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

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

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

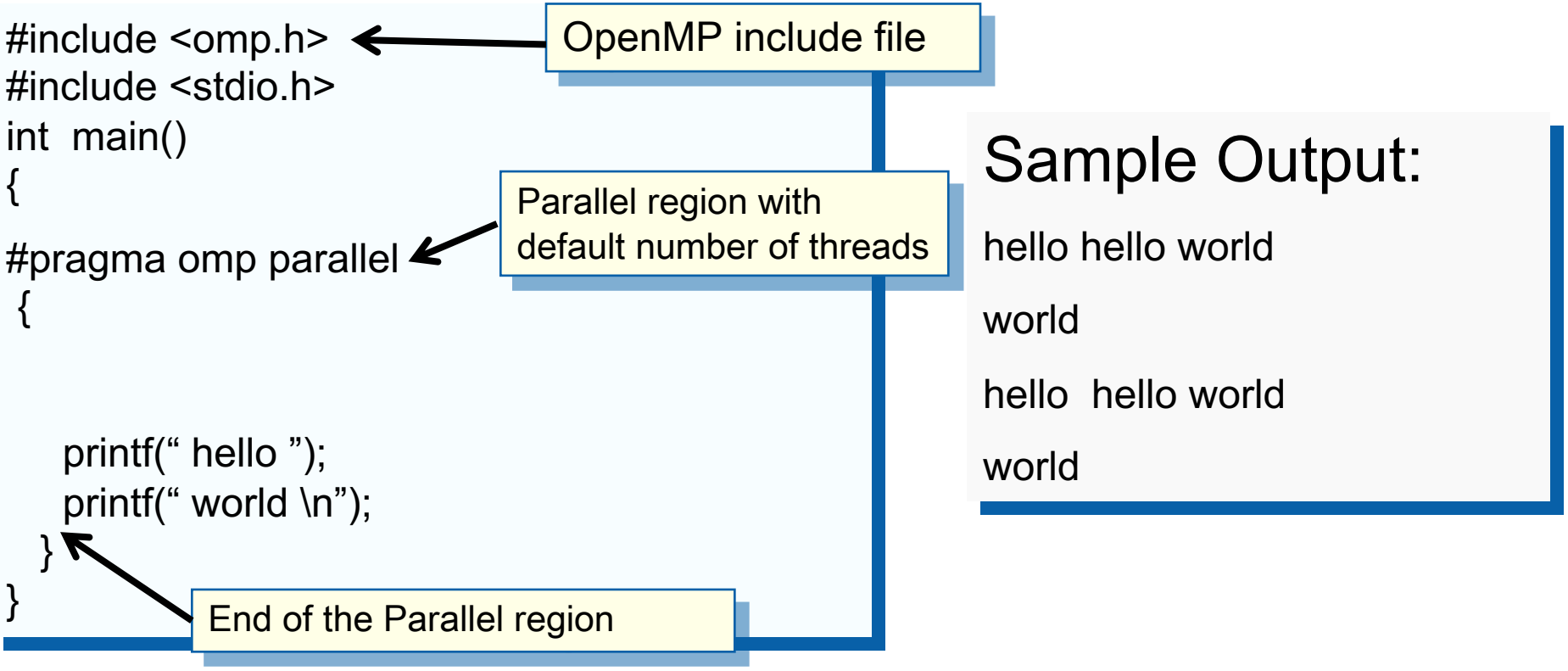
Switches for compiling and linking

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

Solution

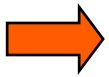
A multi-threaded “Hello world” program

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



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

Outline

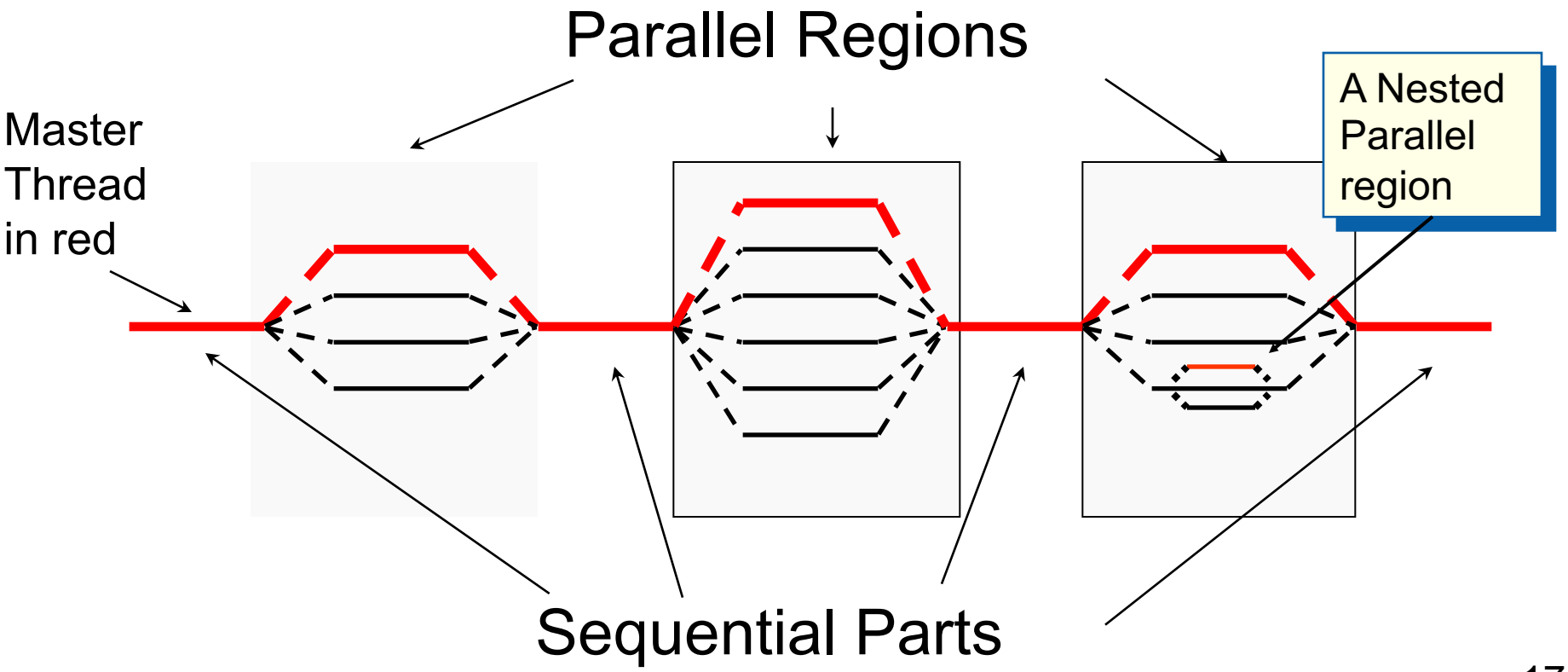


- Introduction to OpenMP
- Creating Threads
- Synchronization
- Parallel Loops
- Data environment
- Memory model
- Irregular Parallelism and tasks
- Recap
- Beyond the common core:
  - Worksharing revisited
  - Synchronization: More than you ever wanted to know
  - Thread private data
  - Going deeper into tasks

OpenMP programming model:

Fork-Join Parallelism:

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



Thread creation: Parallel regions

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

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

Runtime function to request a certain number of threads

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

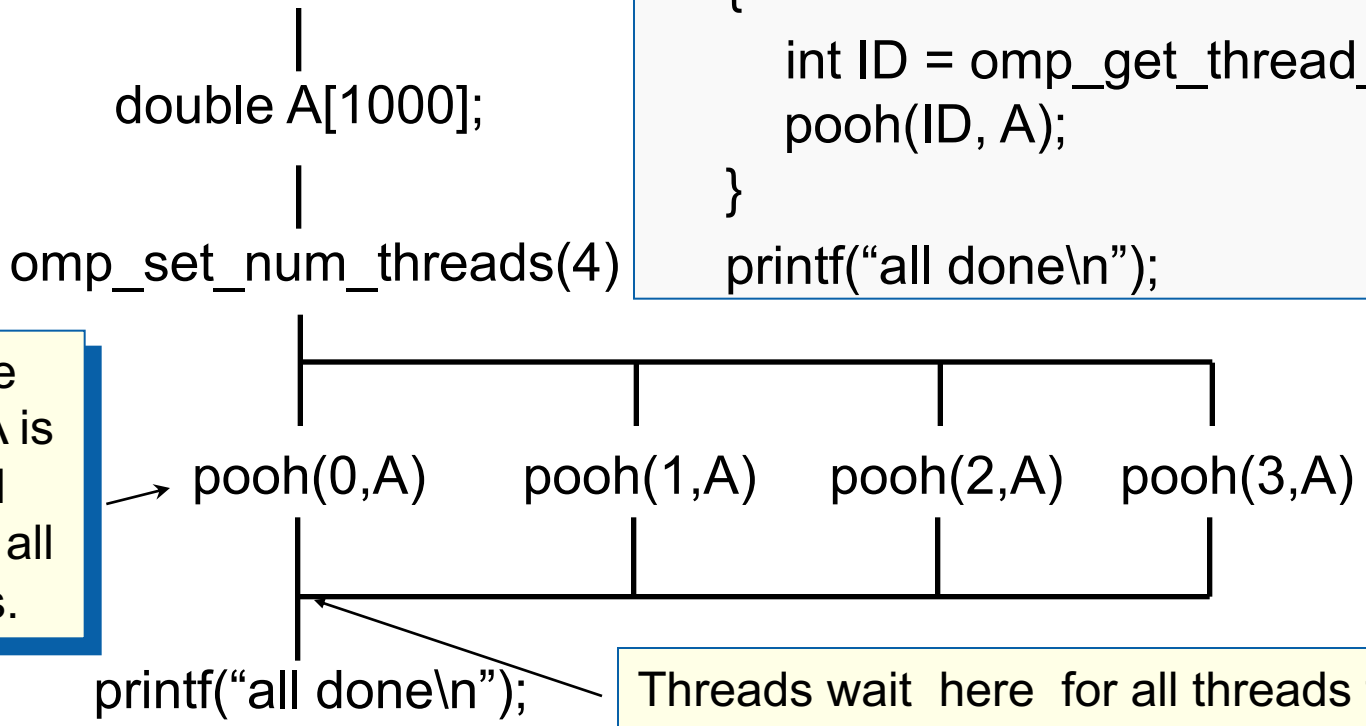
Runtime function returning a thread ID

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

Thread creation: Parallel regions example

- Each thread executes the same code redundantly.

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



A single copy of A is shared between all threads.

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

Thread creation: How many threads did you actually get?

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

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

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

Runtime function to request a certain number of threads

Runtime function to return actual number of threads in the team

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

Internal control variables & the number of threads

- There are a few ways to control the number of threads.
  - omp_set_num_threads(4)

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

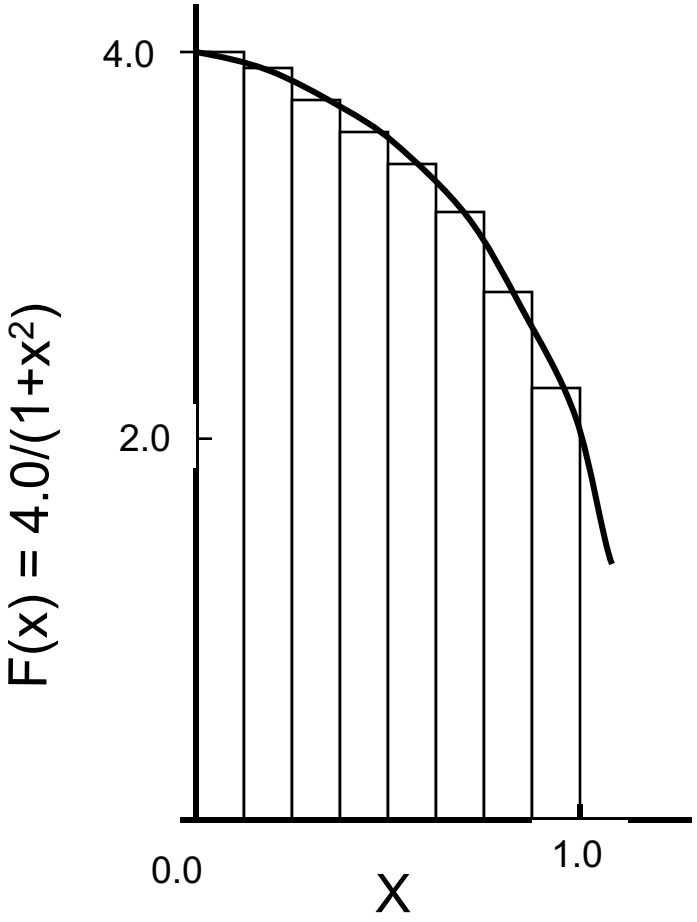
- To change this internal control variable without re-compilation.
  - When an OpenMP program starts up, it queries an environment variable OMP_NUM_THREADS and sets the appropriate internal control variable to the value of OMP_NUM_THREADS

- For example, to set the initial, default number of threads to request in OpenMP from my apple laptop

```
> export OMP_NUM_THREADS=12
```

An interesting problem to play with

Numerical integration



Mathematically, we know that:

$$\int_0^1 \frac{4.0}{(1+x^2)}\,dx = \pi$$

We can approximate the integral as a sum of rectangles:

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

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

Serial PI program

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

            step = 1.0/(double) num_steps;

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

See OMP_exercises/pi.c

Serial PI program

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

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

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

See OMP_exercises/pi.c

Exercise: the parallel Pi program

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

  #pragma omp parallel.

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

Number of threads in the team

Thread ID or rank

Time in Seconds since a fixed point in the past

Request a number of threads in the team

Hints: the Parallel Pi program

- Use a parallel construct:

  #pragma omp parallel

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

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

Example: A simple SPMD pi program

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

SPMD: Single Program Mulitple Data

- Run the same program on P processing elements where P can be arbitrarily large.
- Use the rank … an ID ranging from 0 to (P-1) … to select between a set of tasks and to manage any shared data structures.

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

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

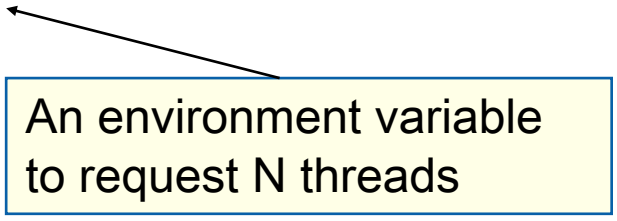
Internal control variables and the number of threads

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

    > **export OMP_NUM_THREADS=12**

Exercise

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



Results*

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

Example: A simple Parallel pi program

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

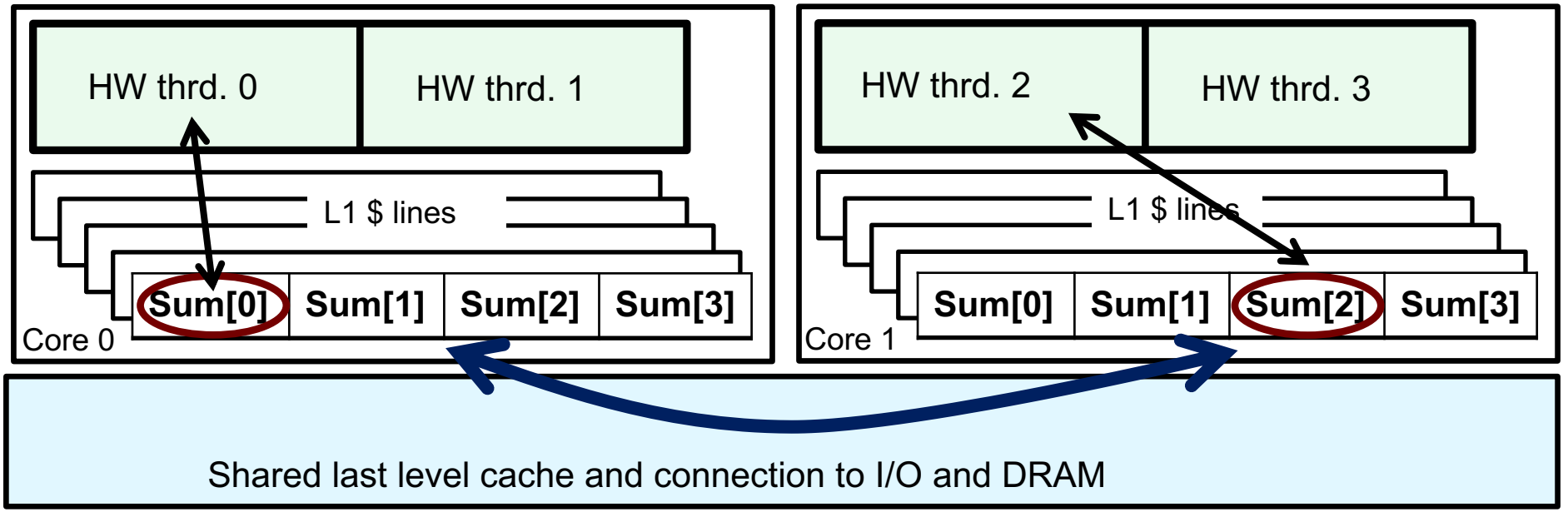
threads	1st SPMD*
1	1.86
2	1.03
3	1.08
4	0.97

*SPMD: Single Program Multiple Data

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

Why such poor scaling? False sharing

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



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

Example: Eliminate false sharing by padding the sum array

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

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

Results*: pi program padded accumulator

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

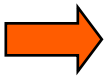
Example: eliminate False sharing by padding the sum array

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

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

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

Outline



- Introduction to OpenMP
- Creating Threads
- Synchronization
- Parallel Loops
- Data environment
- Memory model
- Irregular Parallelism and tasks
- Recap
- Beyond the common core:
  - Worksharing revisited
  - Synchronization: More than you ever wanted to know
  - Thread private data
  - Going deeper into tasks

Synchronization

- High level synchronization included in the common core (the full OpenMP specification has MANY more):
  - critical
  - barrier

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

Synchronization: critical

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

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

```
float  res;
#pragma omp parallel
{    float B;   int i, id, nthrds;
     id = omp_get_thread_num();
     nthrds = omp_get_num_threads();
      for(i=id;i<niters;i+=nthrds){
           B =  big_job(i);
#pragma omp critical
           res += consume (B);
     }
}
```

Synchronization: barrier

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

```
double Arr[8], Brr[8];          int numthrds;

omp_set_num_threads(8)

#pragma omp parallel

{    int id, nthrds;

     id = omp_get_thread_num();

     nthrds = omp_get_num_threads();

     if (id==0) numthrds = nthrds;

     Arr[id] = big_ugly_calc(id, nthrds);

#pragma omp barrier
     Brr[id] = really_big_and_ugly(id, nthrds, A);
}
```

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

Exercise

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

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

Pi program with false sharing*

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

Example: A simple Parallel pi program

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

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

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

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

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

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

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

No array, so no false sharing.

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

Results*: pi program critical section

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

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

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

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

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

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

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

Be careful where you put a critical section

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

Outline

- Introduction to OpenMP
- Creating Threads
- Synchronization
- Parallel Loops
- Data environment
- Memory model
- Irregular Parallelism and tasks
- Recap
- Beyond the common core:
  - Worksharing revisited
  - Synchronization: More than you ever wanted to know
  - Thread private data
  - Going deeper into tasks

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);
        }
}
```

Loop construct name:

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

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

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

Loop worksharing 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.
  - #pragma omp for schedule(dynamic, CHUNK)

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

Least work at runtime : scheduling done at compile-time

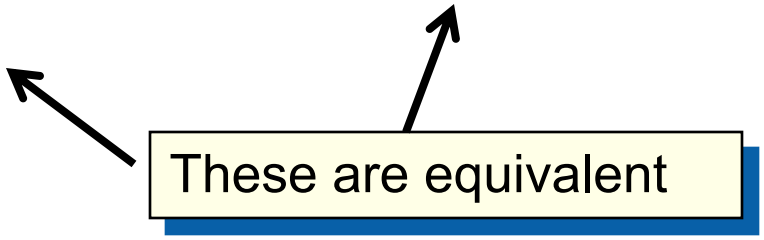
Most work at runtime : complex scheduling logic used at run-time

Combined parallel/worksharing construct

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

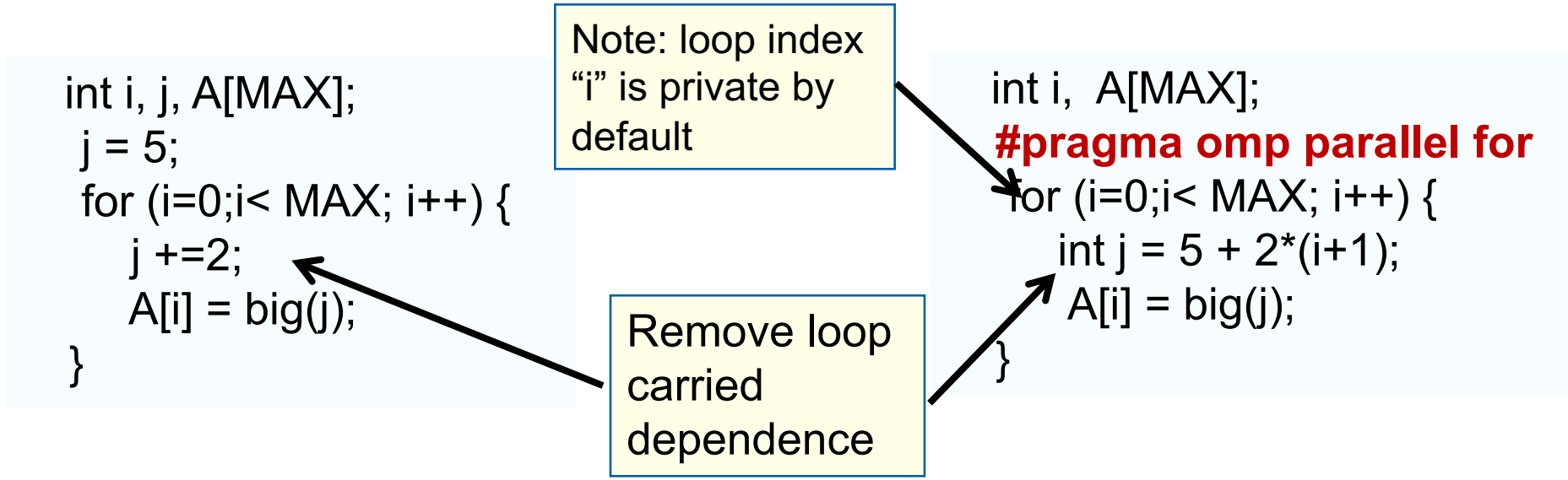
```
 double  res[MAX];  int i;
#pragma omp parallel
{
    #pragma omp for
    for (i=0;i< MAX; i++) {
        res[i] = huge();
    }
}
```

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



Working with loops

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



Reduction

- How do we handle this case?

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

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

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

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

Reduction

- OpenMP reduction clause:

  reduction (op : list)

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

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

OpenMP: Reduction operands/initial-values

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

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

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

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

Exercise: Pi with loops

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

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

Example: Pi with a loop and a reduction

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

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

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

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

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.

The nowait clause

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

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

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

implicit barrier at the end of a for worksharing construct

no implicit barrier due to nowait

implicit barrier at the end of a parallel region

Outline

- Introduction to OpenMP
- Creating Threads
- Synchronization
- Parallel Loops
- Data environment
- Memory model
- Irregular Parallelism and tasks
- Recap
- Beyond the common core:
  - Worksharing revisited
  - Synchronization: More than you ever wanted to know
  - Thread private data
  - Going deeper into tasks

Data environment: Default storage attributes

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

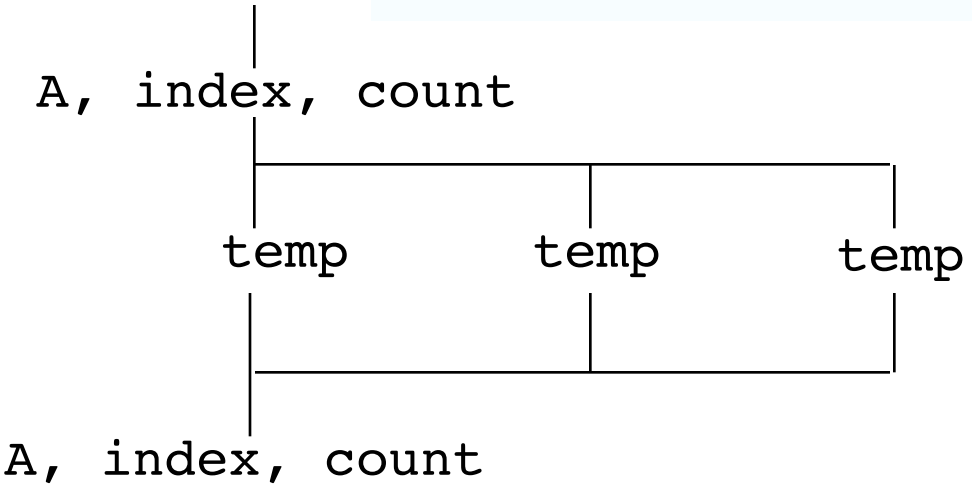
Data sharing: Examples

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

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

A, index and count are shared by all threads.

temp is local to each thread



Data sharing: Changing storage attributes

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

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

default() can only be used on parallel constructs

Data sharing: Private clause

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

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

tmp was not initialized

tmp is 0 here

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

Data sharing: Private and the original variable

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

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

tmp has unspecified value

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

unspecified which copy of tmp

Firstprivate clause

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

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

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

Data sharing:
A data environment test

- Consider this example of PRIVATE and FIRSTPRIVATE

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

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

Inside this parallel region ...

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

Following the parallel region ...

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

Data sharing: Default clause

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

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

```
#include <omp.h>
int main()
{
    int i, j=5;      double x=1.0, y=42.0;
    #pragma omp parallel for default(none) reduction(*:x)
    for (i=0;i<N;i++){
        for(j=0; j<3; j++)
            x+= foobar(i, j, y);
    }
    printf(" x is %f\n",(float)x);
}
```

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

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

Exercise: Mandelbrot set area

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

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 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 critical
           numoutside++;
           break;
         }
       }
}
```

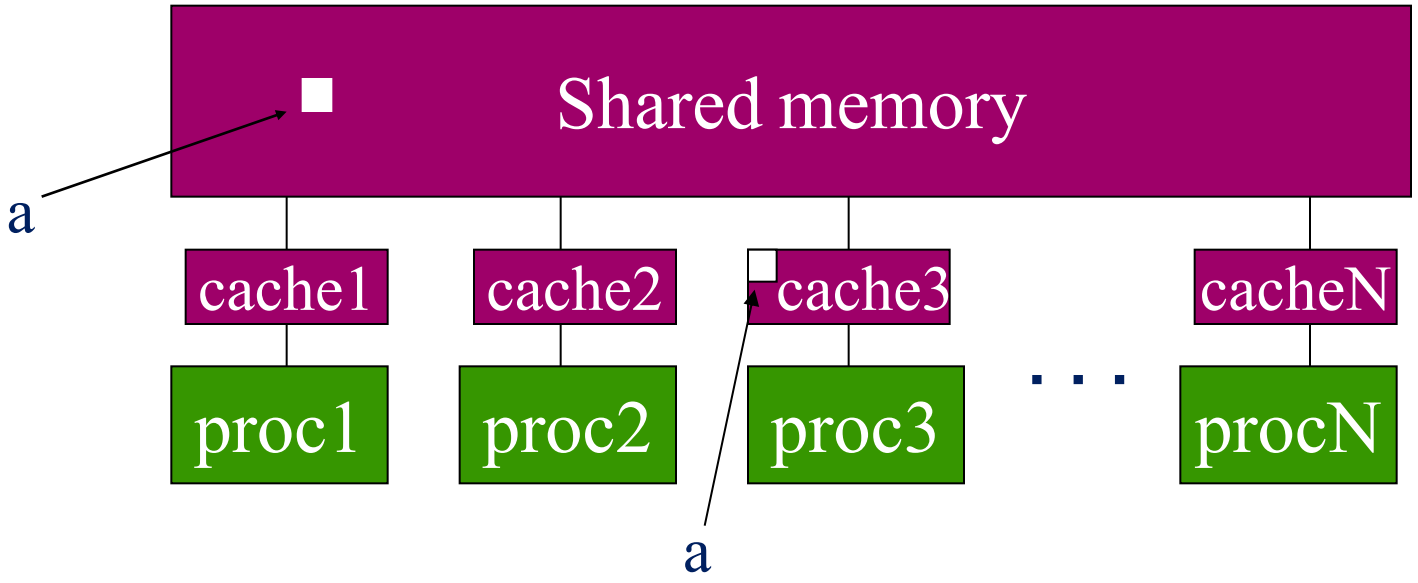
- eps was not initialized
- Protect updates of numoutside
- Which value of c does testpoint() see? Global or private?

Outline

- Introduction to OpenMP
- Creating Threads
- Synchronization
- Parallel Loops
- Data environment
- Memory model
- Irregular Parallelism and tasks
- Recap
- Beyond the common core:
  - Worksharing revisited
  - Synchronization: More than you ever wanted to know
  - Thread private data
  - Going deeper into tasks

OpenMP memory model

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



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

OpenMP and relaxed consistency

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

Flush operation

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

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

Synchronization: flush example

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

```
double A;

A = compute();

#pragma omp flush(A)

   // flush to memory to make sure other
   //  threads can pick up the right value
```

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

What is the BIG DEAL with flush?

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

Note: the flush operation does not actually synchronize different threads. It just ensures that a thread’s variables are made consistent with main memory

Flush and synchronization

- A flush operation is implied by OpenMP synchronizations, e.g.,
  - at entry/exit of parallel regions
  - at implicit and explicit barriers
  - at entry/exit of critical regions

  ….

  (but not at entry to worksharing regions)

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

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

Flush and synchronization

- A flush is implied where it makes sense:
  - at entry/exit of parallel regions
  - at implicit and explicit barriers
  - at entry/exit of critical regions

  ….

  (but not at entry to worksharing regions)

This means if you are mixing reads and writes of a variable across multiple threads, you cannot assume the reading threads see the results of the writes unless:

- the writing threads follow the writes with a construct that implies a flush.
- the reading threads proceed the reads with a construct that implies a flush.

If you use the common core of OpenMP … this isn't an issue. You should avoid writing code that depends on ordering reads/writes around flushes.

Outline

- Introduction to OpenMP
- Creating Threads
- Synchronization
- Parallel Loops
- Data environment
- Memory model
- Irregular Parallelism and tasks
- Recap
- Beyond the common core:
  - Worksharing revisited
  - Synchronization: More than you ever wanted to know
  - Thread private data
  - Going deeper into tasks

Irregular parallelism

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

- Example: Traversing Linked lists:

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

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

Exercise: traversing linked lists

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

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

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

Linked lists with OpenMP pre 3.0

- See the file solutions/Linked_notasks.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 with OpenMP pre 3.0

- See the file solutions/Linked_notasks.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

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

There has got to be a better way to do this

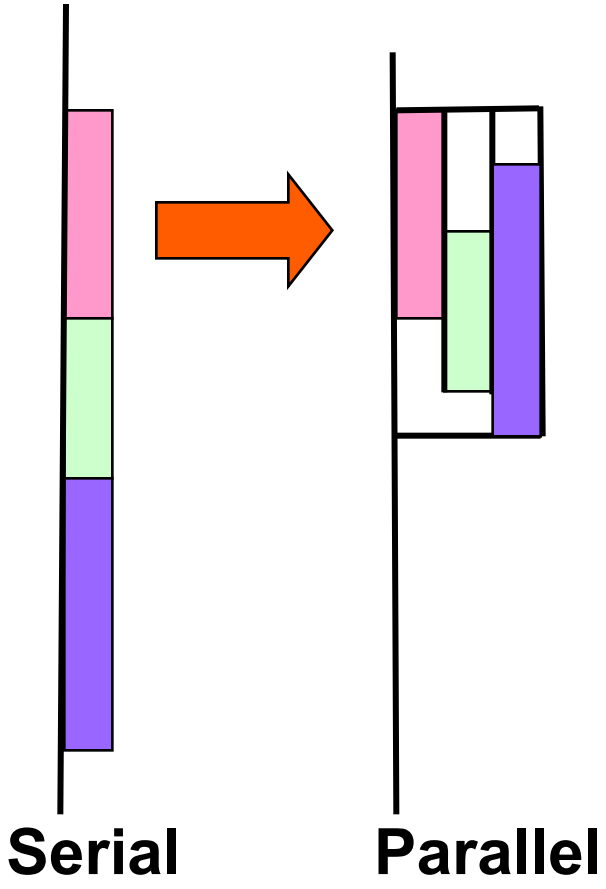
Process nodes in parallel with a for loop

	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

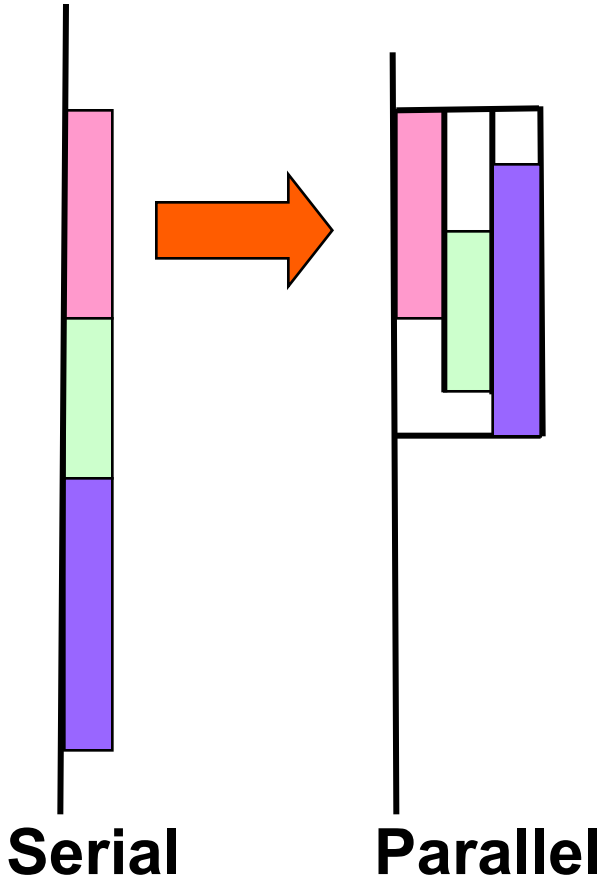
What are tasks?

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



What are tasks?

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



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

Single worksharing Construct

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

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

Task Directive

`#pragma omp task` *[clauses]*

structured-block

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

Create some threads

One Thread packages tasks

Tasks executed by some thread in some order

All tasks complete before this barrier is released

Exercise: Simple tasks

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

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

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

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

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

Racey cars: solution

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

Data scoping with tasks

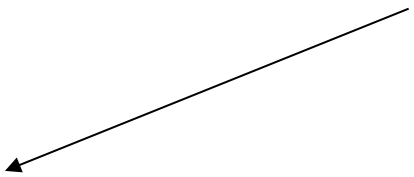
- Variables can be shared, private or firstprivate with respect to task

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

Data scoping defaults

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

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



A is shared
B is firstprivate
C is private

Exercise: traversing linked lists

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

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

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

Parallel linked list traversal

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

Only one thread packages tasks

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

When/where are tasks complete?

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

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

```
#pragma omp taskwait
```

  - Note: applies only to tasks generated in the current task, not to “descendants” .

Example

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

`fred()` and `daisy()` must complete before `billy()` starts

Example: Fibonacci numbers

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

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

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

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

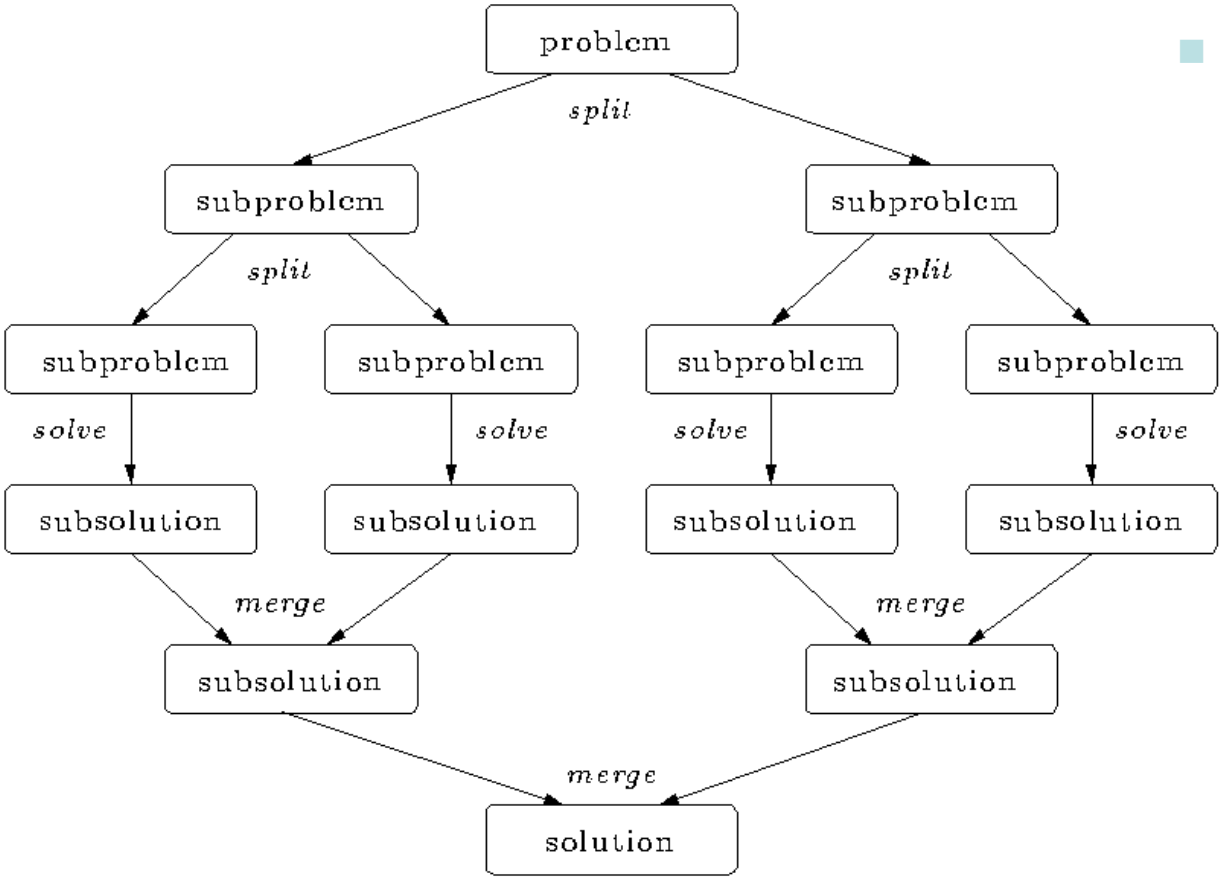
Parallel Fibonacci

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

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

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

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

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

Exercise: Pi with tasks

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

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

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

Program: OpenMP tasks

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

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

Results*: pi with tasks

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

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

Using tasks

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

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

Outline

- Introduction to OpenMP
- Creating Threads
- Synchronization
- Parallel Loops
- Data environment
- Memory model
- Irregular Parallelism and tasks
- Recap
- Beyond the common core:
  - Worksharing revisited
  - Synchronization: More than you ever wanted to know
  - Thread private data
  - Going deeper into tasks

The OpenMP Common Core: Most OpenMP programs only use these 20 items

OpenMP pragma, function, or clause	Concepts
#pragma omp parallel	Parallel region, teams of threads, structured block, interleaved execution across threads.
void omp_set_num_threads() int omp_get_thread_num() int omp_get_num_threads()	Default number of threads and internal control variables. SPMD pattern: Create threads with a parallel region and split up the work using the number of threads and the thread ID.
double omp_get_wtime()	Speedup and Amdahl's law. False sharing and other performance issues.
setenv OMP_NUM_THREADS N	Setting the internal control variable for the default number of threads with an environment variable
#pragma omp barrier #pragma omp critical	Synchronization and race conditions. Revisit interleaved execution.
#pragma omp for #pragma omp parallel for	Worksharing, parallel loops, loop carried dependencies.
reduction(op:list)	Reductions of values across a team of threads.
schedule (static [,chunk]) schedule(dynamic [,chunk])	Loop schedules, loop overheads, and load balance.
shared(list), private(list), firstprivate(list)	Data environment.
nowait	Disabling implied barriers on workshare constructs, the high cost of barriers, and the flush concept (but not the flush directive).
#pragma omp single	Workshare with a single thread.
#pragma omp task #pragma omp taskwait	Tasks including the data environment for tasks.

There is much more to OpenMP than the Common Core.

- Synchronization mechanisms
  - locks, flush and several forms of atomic
- Data management
  - lastprivate, threadprivate, default(private|shared)
- Fine grained task control
  - dependencies, tied vs. untied tasks, task groups, task loops …
- Vectorization constructs
  - simd, uniform, simdlen, inbranch vs. nobranch, ….
- Map work onto an attached device
  - target, teams distribute parallel for, target data …
- … and much more. The OpenMP 4.5 specification is over 350 pages!!!

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

The fundamental design patterns of parallel programming

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

Fork-join

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

Pthreads, OpenMP are based on this pattern.

SPMD: Single Program Mulitple Data

- Run the same program on P processing elements where P can be arbitrarily large.
- Use the rank … an ID ranging from 0 to (P-1) … to select between a set of tasks and to manage any shared data structures.

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

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

A simple SPMD pi program

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

Loop-level parallelism

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

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

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

Loop Level parallelism pi program

```
#include <omp.h>
static long num_steps = 100000;        double step;

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

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

i private by default

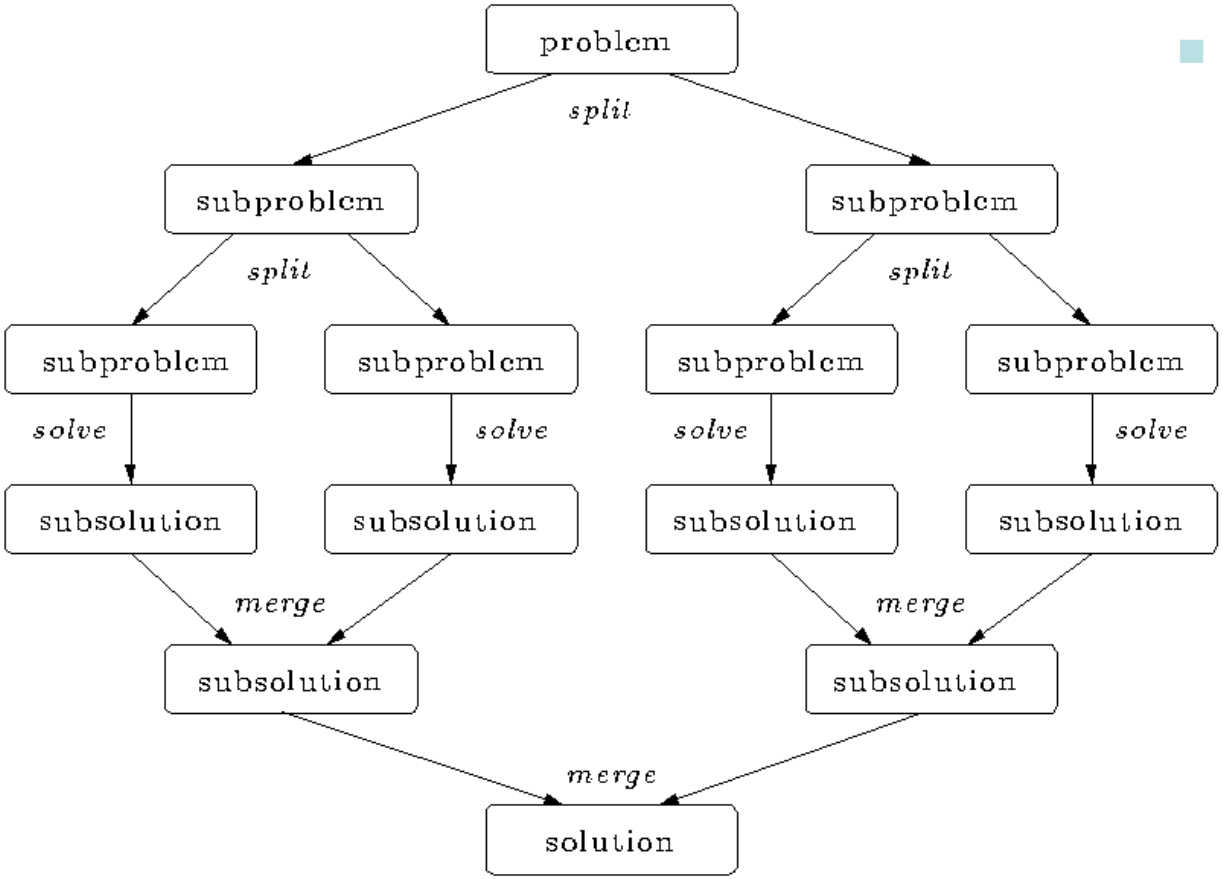
Note: we created a parallel program without changing any code and by adding 2 simple lines of text!

Divide and Conquer Pattern

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

Divide and conquer

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



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

Geometric Decomposition

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

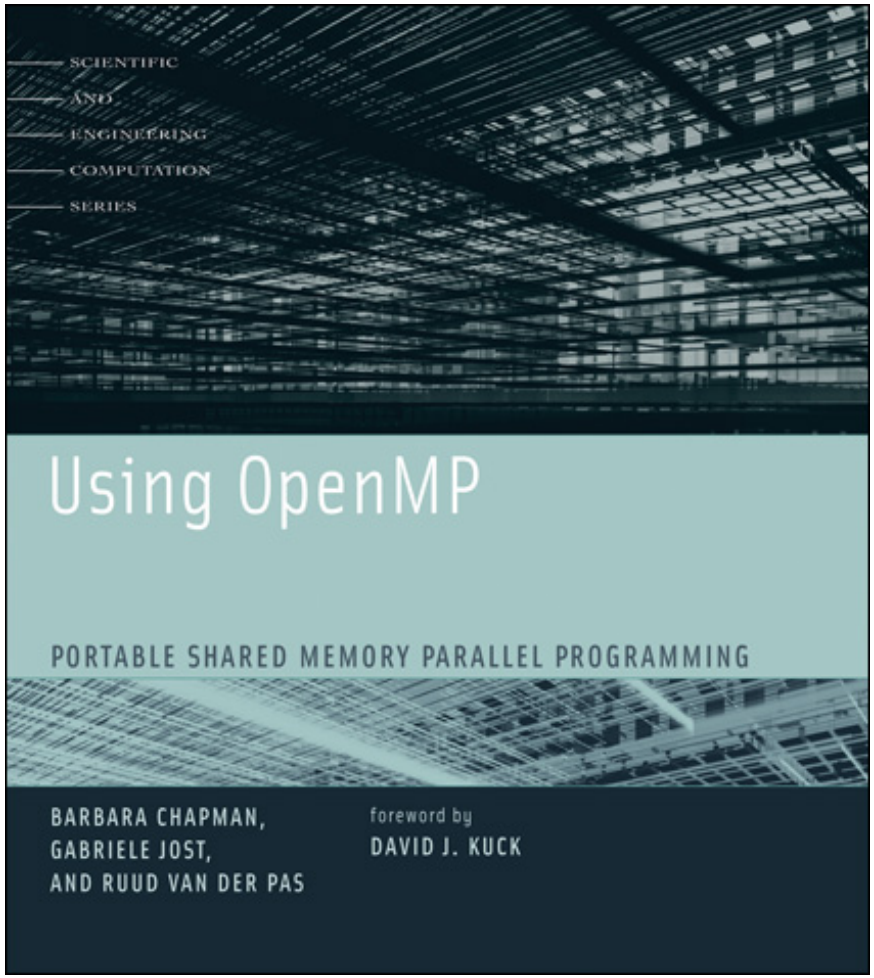
SIMT (Data-parallel/index-space)

We'll cover this when we discuss GPUs

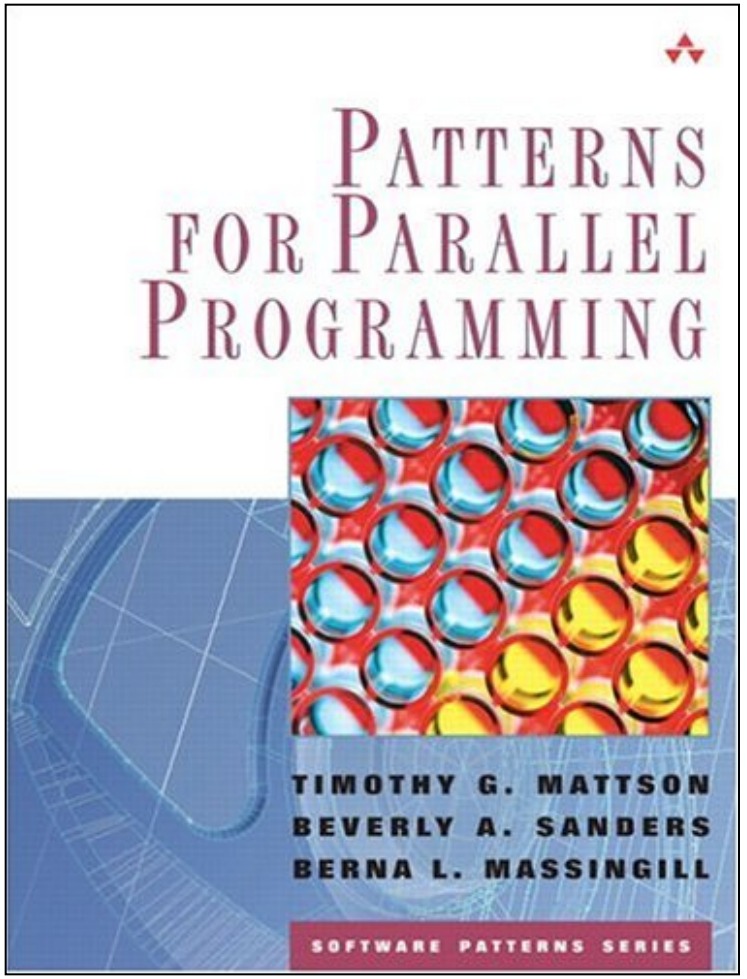
- Single instruction, multiple data:
  - Implement data parallel problems:
    - Define an abstract index space that appropriately spans the problem domain.
    - Data structures in the problem are aligned to this index space.
    - Tasks (e.g. work-items in OpenCL or "threads" in CUDA) operate on these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

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

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

Resources:

A great new book that covers OpenMP features beyond OpenMP 2.5

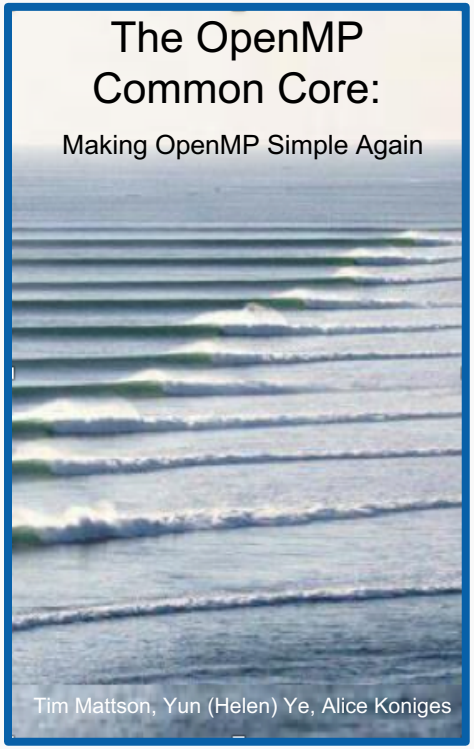
Visit the OpenMP booth and enter a drawing for a chance to win a copy of the book. Drawing Tues and Wed @ 4:30, Thurs @ 2:00. You must be present to win.

Resources:

- We wrote a book about the OpenMP Common Core and some of the key elements of OpenMP to master as you move beyond the common core.
- It's geared towards people learning OpenMP, but as one commentator put it … everyone at any skill level should read the memory model chapters.

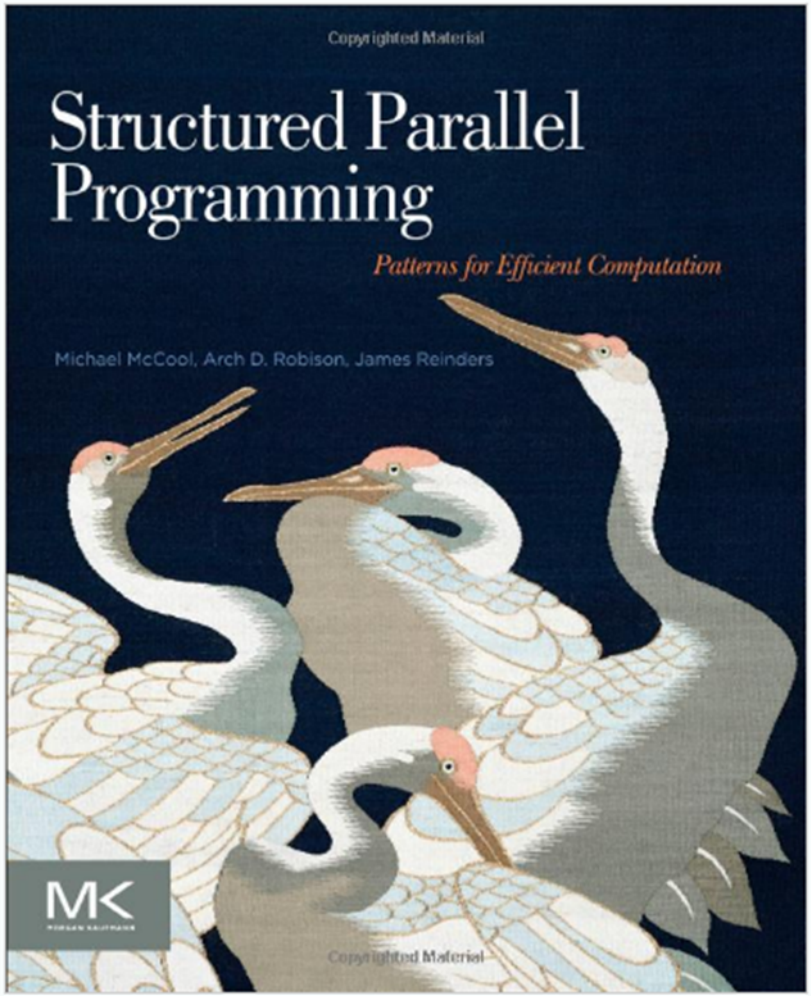
Focusses on shared memory programming of SMP computers … but briefly mentions NUMA, SIMD and GPUs.

And it's recent … coming out in Nov 2019 from MIT Press

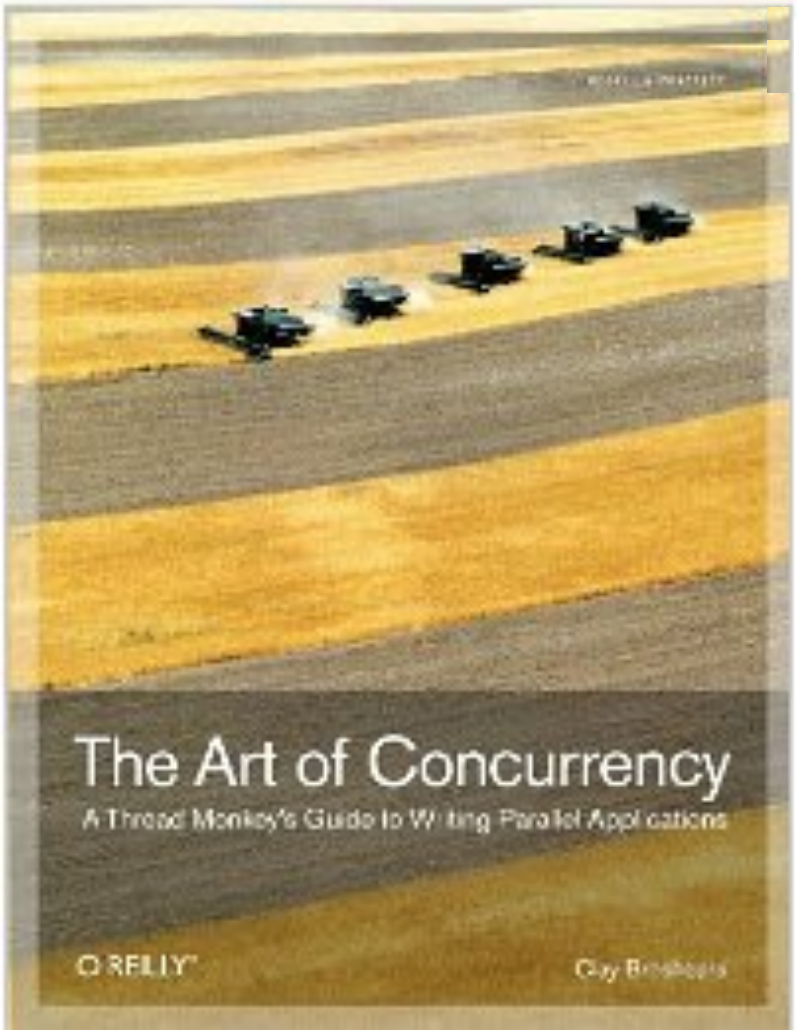


This is a guess at what the book will look like.

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)

Outline

- Introduction to OpenMP
- Creating Threads
- Quantifying Performance and Amdahl's law
- Synchronization
- Parallel Loops
- Data environment
- Memory model
- Irregular Parallelism and tasks
- Recap
- Beyond the common core:
  - Worksharing revisited
  - Synchronization: More than you ever wanted to know
  - Thread private data
  - Going deeper into tasks

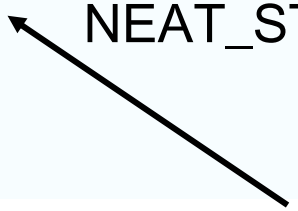
Outline

- Introduction to OpenMP
- Creating Threads
- Quantifying Performance and Amdahl's law
- Synchronization
- Parallel Loops
- Data environment
- Memory model
- Irregular Parallelism and tasks
- Recap
- Beyond the common core:
  - Worksharing revisited
  - Synchronization: More than you ever wanted to know
  - Thread private data
  - Going deeper into tasks

The loop worksharing constructs

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

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



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

Loop construct name:

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

Loop worksharing constructs: The schedule clause

- The schedule clause affects how loop iterations are mapped onto threads
  - **schedule(static [,chunk])**
    - Deal-out blocks of iterations of size “chunk” to each thread.
  - **schedule(dynamic[,chunk])**
    - Each thread grabs “chunk” iterations off a queue until all iterations have been handled.
  - **schedule(guided[,chunk])**
    - Threads dynamically grab blocks of iterations. The size of the block starts large and shrinks down to size “chunk” as the calculation proceeds.
  - **schedule(runtime)**
    - Schedule and chunk size taken from the OMP_SCHEDULE environment variable (or the runtime library).
  - **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

Nested loops

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

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

Number of loops to be parallelized, counting from the outside

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

Sections worksharing Construct

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

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

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

Array sections with reduce

```
#include <stdio.h>
#define N 100
void init(int n, float (*b)[N]);

int main(){
int i,j; float a[N], b[N][N]; init(N,b);
for(i=0; i<N; i++)  a[i]=0.0e0;

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

Works the same as the other reductions:

- A private array is formed for each thread.
- Initialized according to the identify for the operator
- element wise combination across threads at end of the construct.
- elementwise combination with original array at the end of the construct.

Outline

- Introduction to OpenMP
- Creating Threads
- Quantifying Performance and Amdahl's law
- Synchronization
- Parallel Loops
- Data environment
- Memory model
- Irregular Parallelism and tasks
- Recap
- Beyond the common core:
  - Worksharing revisited
  - Synchronization: More than you ever wanted to know
  - Thread private data
  - Going deeper into tasks

Synchronization

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

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

Covered earlier

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

- In your first Pi program, you probably used an array to create space for each thread to store its partial sum.
- You fixed this by using a critical section instead of updating the array (remember .. the array you created by promoting the scalar “sum” to an array).
- Use and atomic instead. Does the performance improve?

Parallel loop with ordered region

- An **ordered clause** on a loop worksharing construct
  - indicates that the loop contains an ordered region

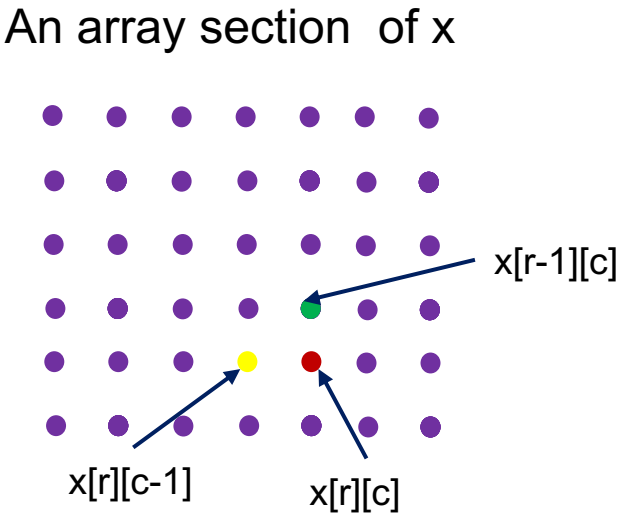
- The **ordered construct** defines an ordered region
  - The Statements in ordered region execute in iteration order

```
#pragma omp for ordered
    for (i=0; i<N; i++) {
      float res = work(i);
      #pragma omp ordered
      {
        printf("result for %d was %f\n", i, res);
        fflush(stdout);
      }
    }
```

Parallelizing nested loops

- Will these nested parallel loops execute correctly?

```
#pragma omp parallel for collapse(2)
for (r=1; r<N; r++) {
   for (c=1; c<N; c++) {

    x[r][c] += fn(x[r-1][c], x[r][c-1]);

  }
}
```



- Pattern of dependencies between elements of x prevent straightforward parallelization
- is there a way to manage the synchronization so we can parallelize this loop?

Ordered stand-alone directive

- Specifies cross-iteration dependencies in a doacross loop nest … i.e. loop level parallelism over nested loops with a regular pattern of synchronization to manage dependencies.

```
#pragma omp ordered depend(sink : vec)
#pragma omp ordered depend(source)
```

vec is a comma separated list of dependencies … one per loop involved in the dependencies

- **Depend** clauses specify the order the threads execute **ordered** regions.
  - The **sink** *dependence-type*
    - specifies a cross-iteration dependence, where the iteration vector *vec* indicates the iteration that satisfies the dependence.
  - The **source** *dependence-type*
    - specifies the cross-iteration dependences that arise from the current iteration.

Parallelizing DOACROSS loops

2 loops contribute to the pattern of dependencies … so the dependency relations for each depend(sink) is of length 2

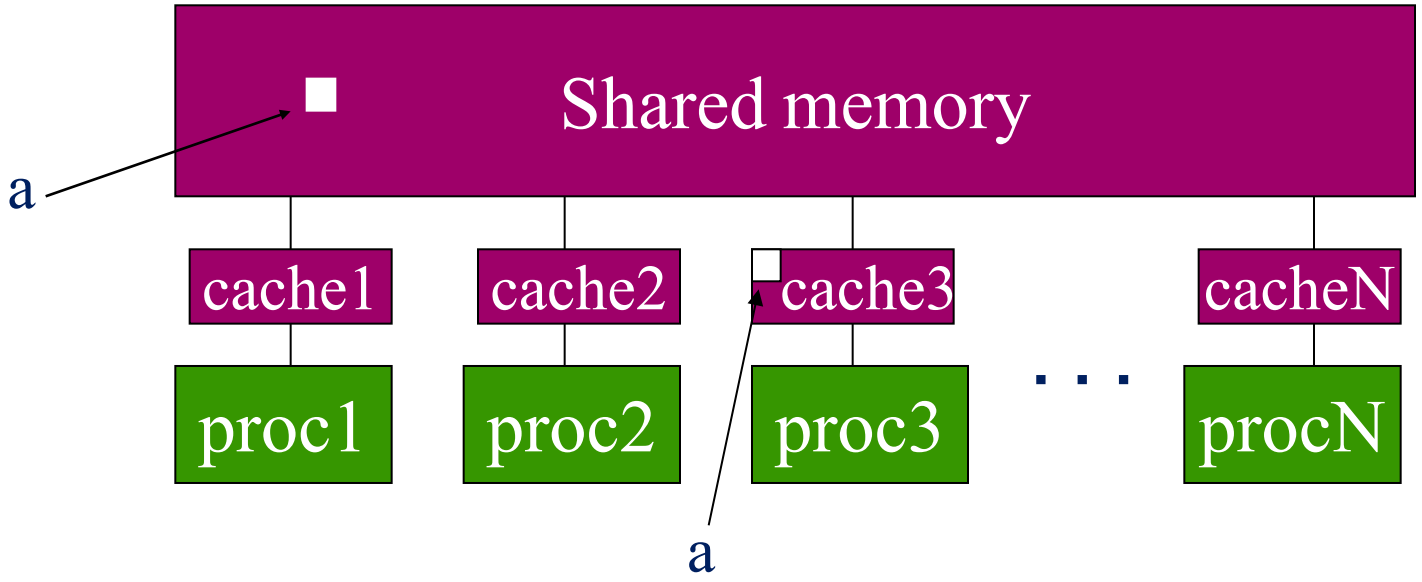
Threads wait here until x[r-1][c] and x[r][c-1] have been released

```
#pragma omp for ordered(2) collapse(2)
    for (r=1; r<N; r++) {
        for (c=1; c<N; c++) {
            // other parallel work ...
            #pragma omp ordered depend(sink:r-1,c) \
                                depend(sink:r,c-1)
                x[r][c] += fn(x[r-1][c], x[r][c-1]);
            #pragma omp ordered depend(source)
        }
    }
```

x[r][c] is complete and released for use by other threads

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.

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 synchronization in OpenMP

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

Exercise: Producer/consumer

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

        flag = 1;

      }
      #pragma omp section
      {

        while (flag == 0){

        }

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

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

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

Solution (try 1): Producer/consumer

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

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

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

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

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

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

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;}

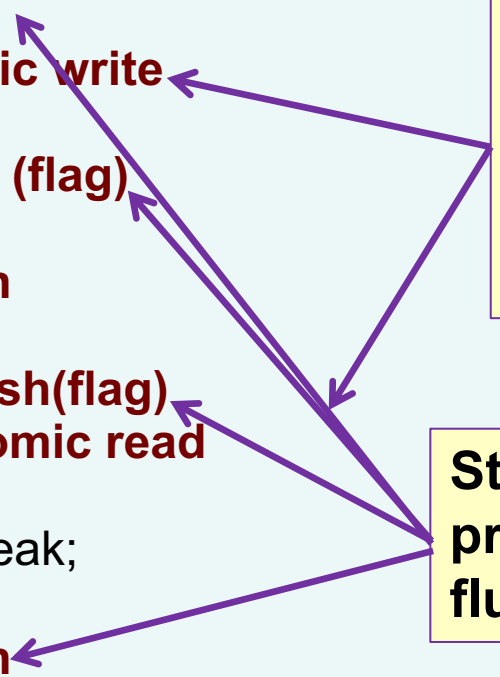
{v= x; --x;}

{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

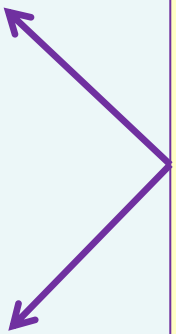
OpenMP 4.0 Atomic: Sequential consistency

- 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)

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)

Lock Example from Gafort (SpecOMP'2001)

- Genetic algorithm in Fortran
- Most "interesting" loop: shuffle the population.
  - Original loop is not parallel; performs pair-wise swap of an array element with another, randomly selected element. There are 40,000 elements.
  - Parallelization idea:
    - Perform the swaps in parallel
    - Need to prevent simultaneous access to same array element: use one lock per array element → 40,000 locks.

Parallel loop In shuffle.f of Gafort

Exclusive access to array elements. Ordered locking prevents deadlock.

```
!$OMP PARALLEL PRIVATE(rand, iother, itemp, temp, my_cpu_id)
      my_cpu_id = 1
!$    my_cpu_id = omp_get_thread_num() + 1
!$OMP DO
      DO j=1,npopsiz-1
        CALL ran3(1,rand,my_cpu_id,0)
        iother=j+1+DINT(DBLE(npopsiz-j)*rand)
!$      IF (j < iother) THEN
!$        CALL omp_set_lock(lck(j))
!$        CALL omp_set_lock(lck(iother))
!$      ELSE
!$        CALL omp_set_lock(lck(iother))
!$        CALL omp_set_lock(lck(j))
!$      END IF
        itemp(1:nchrome)=iparent(1:nchrome,iother)
        iparent(1:nchrome,iother)=iparent(1:nchrome,j)
        iparent(1:nchrome,j)=itemp(1:nchrome)
        temp=fitness(iother)
        fitness(iother)=fitness(j)
        fitness(j)=temp
!$      IF (j < iother) THEN
!$        CALL omp_unset_lock(lck(iother))
!$        CALL omp_unset_lock(lck(j))
!$      ELSE
!$        CALL omp_unset_lock(lck(j))
!$        CALL omp_unset_lock(lck(iother))
!$      END IF
      END DO
!$OMP END DO
!$OMP END PARALLEL
```

Exercise

- We provide a program in the file hist.c
- This program tests our random number generator by calling it many times and producing a histogram of the results.
- Parallelize this program.

```
omp_lock_t lck;
omp_init_lock(&lck);
omp_set_lock(&lck);
omp_unset_lock(&lck);
omp_destroy_lock(&lck);
```

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.

Outline

- Introduction to OpenMP
- Creating Threads
- Quantifying Performance and Amdahl's law
- Synchronization
- Parallel Loops
- Data environment
- Memory model
- Irregular Parallelism and tasks
- Recap
- Beyond the common core:
  - Worksharing revisited
  - Synchronization: More than you ever wanted to know
  - Thread private data
  - Going deeper into tasks

Data sharing: Threadprivate

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

A threadprivate example (C)

Use threadprivate to create a counter for each thread.

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

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

Data copying: Copyin

You initialize threadprivate data using a copyin clause.

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

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

!$OMP PARALLEL COPYIN(A)

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

!$OMP END PARALLEL

end
```

Data copying: Copyprivate

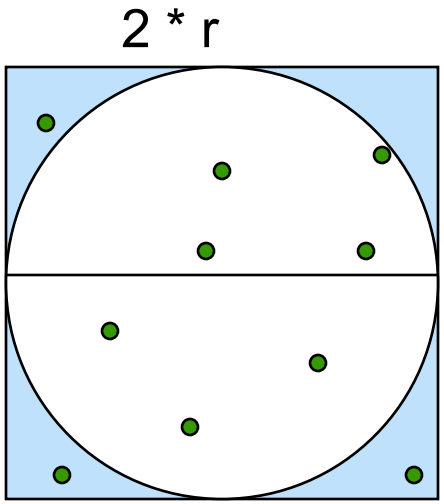
Used with a single region to broadcast values of privates from one member of a team to the rest of the team

```
#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).

Outline

- Introduction to OpenMP
- Creating Threads
- Quantifying Performance and Amdahl's law
- Synchronization
- Parallel Loops
- Data environment
- Memory model
- Irregular Parallelism and tasks
- Recap
- Beyond the common core:
  - Worksharing revisited
  - Synchronization: More than you ever wanted to know
  - Thread private data
  - Going deeper into tasks

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:
- 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

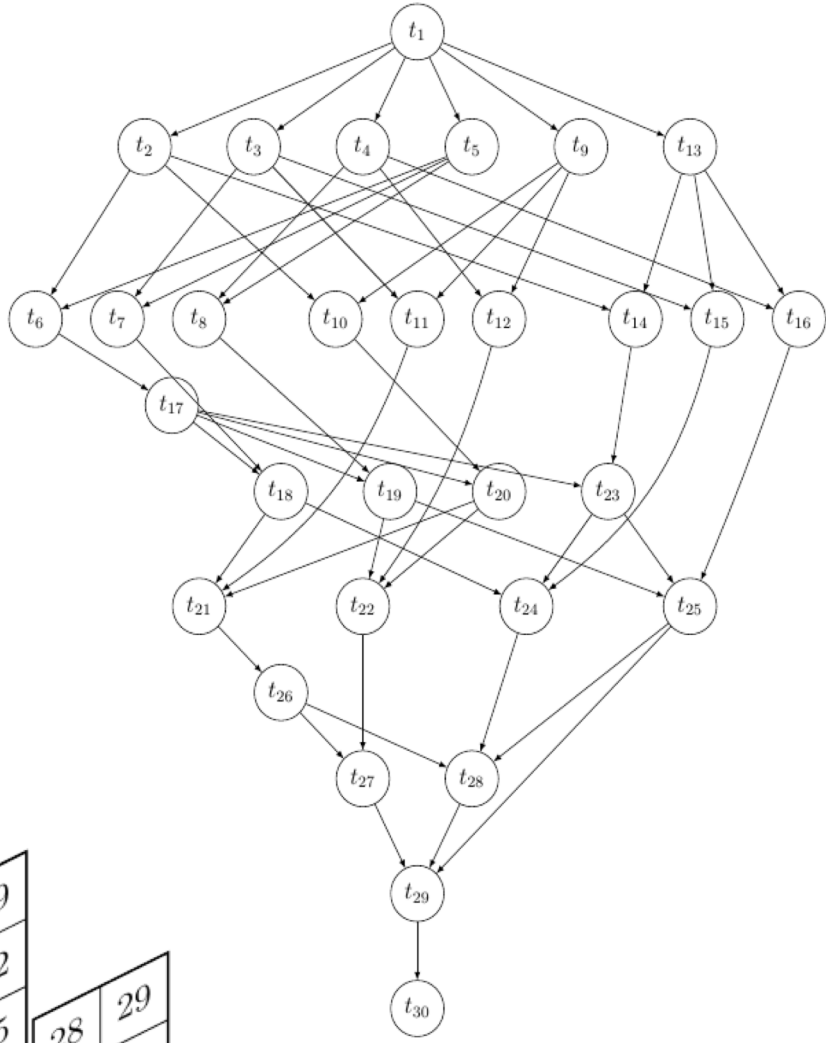
Task dependencies example

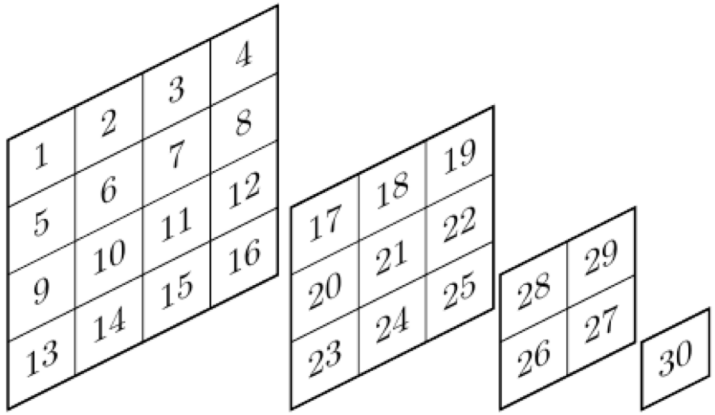
```
#pragma omp task depend (out:a)
   { ... } //writes a
#pragma omp task depend (out:b)
   { ... } //writes b
#pragma omp task depend (in:a,b)
   { ... } //reads a and b
```

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

LU Decomposition

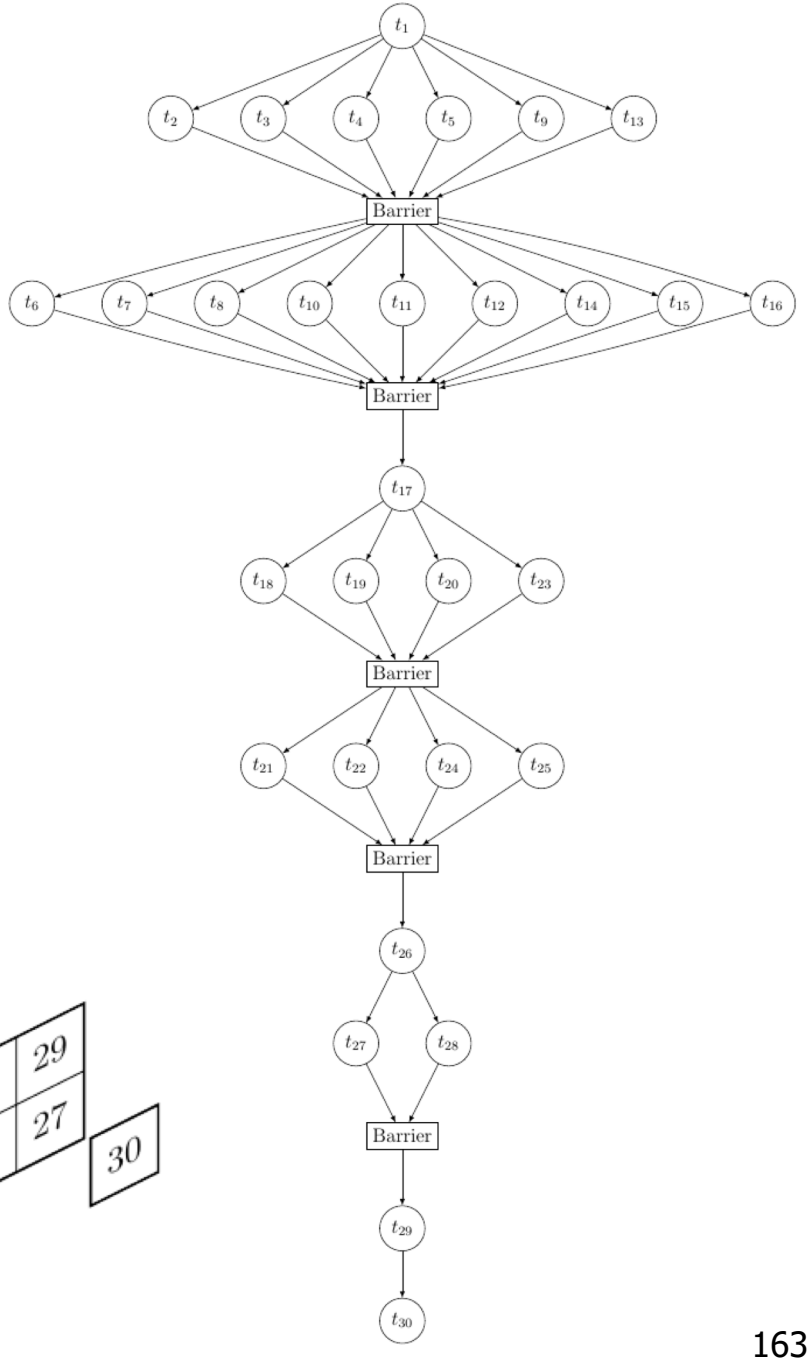
- The matrix is divided into NxN blocks, and a task operates on one block.
- Each iteration the working matrix gets one block smaller in each dimension, resulting in a task graph resembling the one to the right.

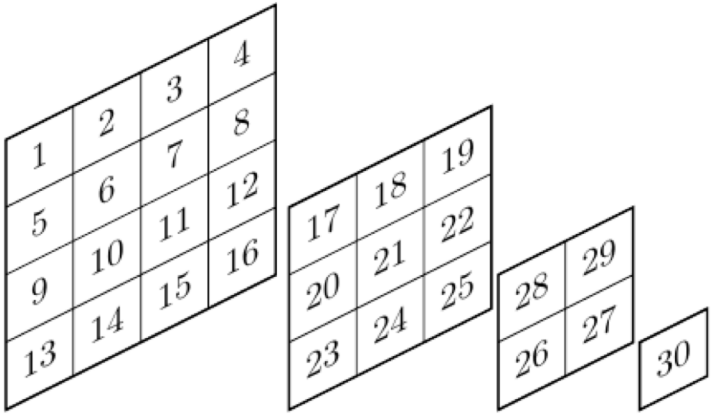




LU Decomposition

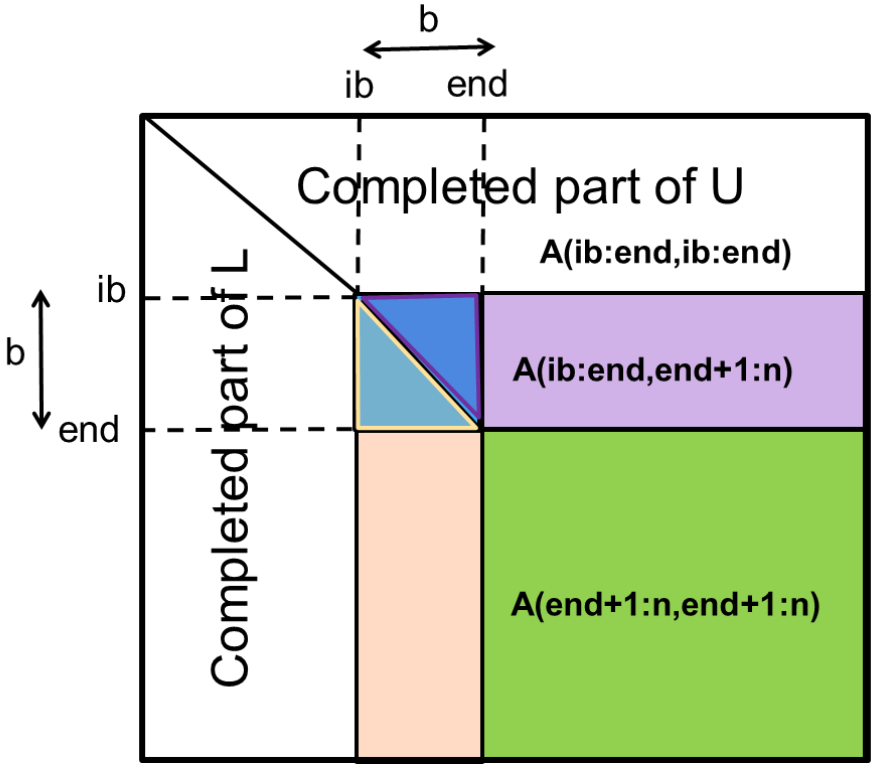
For Comparison, the tasking version without dependencies resembles a fork join programming model, similar to a worksharing version.

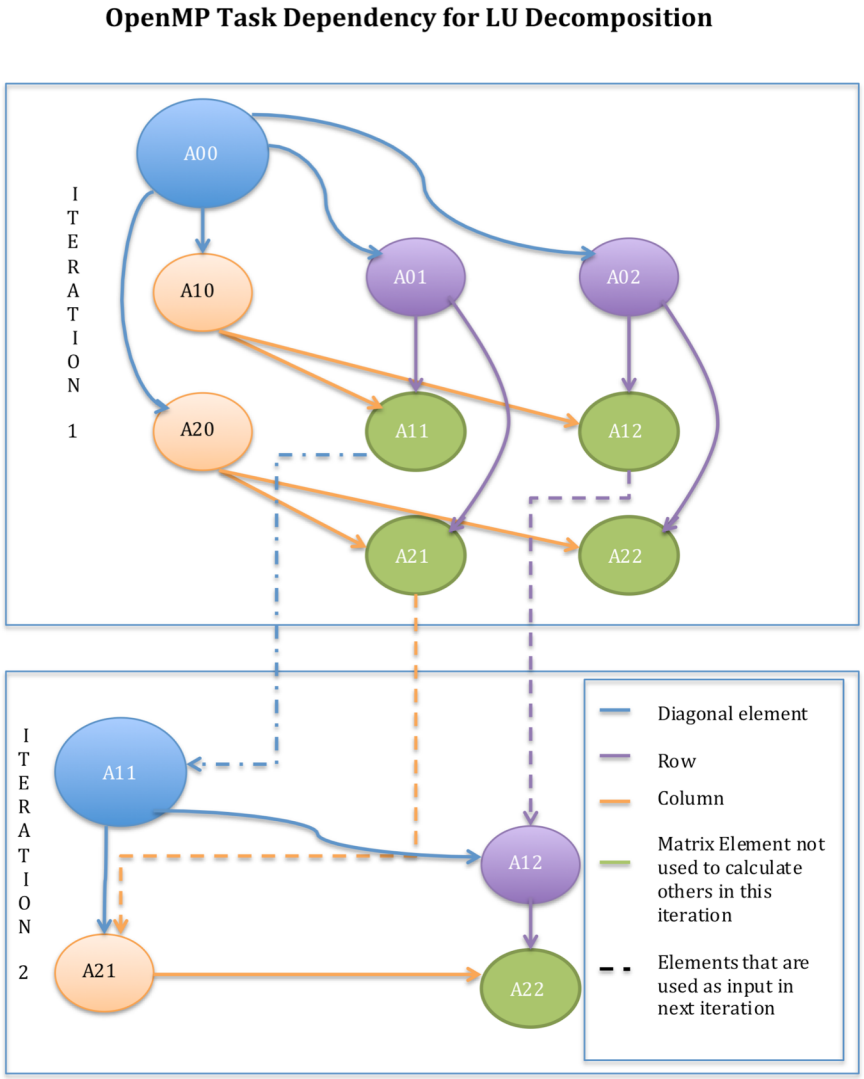




LU Decomposition

There are 4 different operations (diag, row, col, and inner), and the the dependencies between these operations are shown in the graph to the right.





Source: CS164 lecture Fall 2014, Kurt Keutzer of UC Berkeley

LU Decomposition

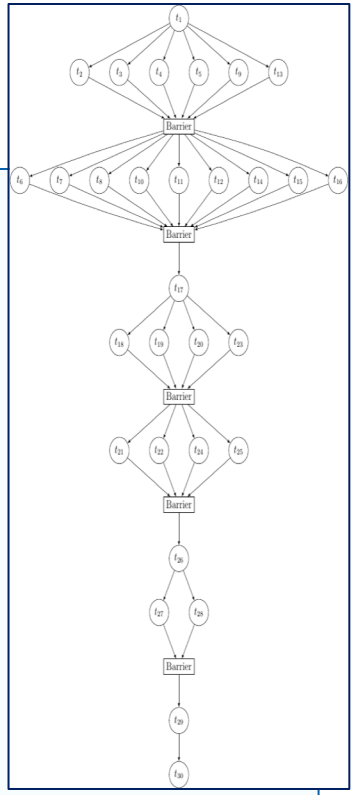
```
void diag_op(const block &B) {
    for(int i = 0; i < B.width; i++)
        for(int j = i+1; j < B.width; j++) {
            B.start[j*B.stride+i] /= B.start[i*B.stride+i];
            for(int k = i+1; k < B.width; k++)
                B.start[j*B.stride+k] -= B.start[j*B.stride+i] * B.start[i*B.stride+k];
}}
void col_op(const block &B1, const block &B2) {
    for(int i=0; i < B2.width; i++)
        for(int j=0; j < B1.height; j++) {
            B1.start[j*B1.stride+i] /= B2.start[i*B2.stride+i];
            for(int k = i+1; k < B2.width; k++)
                B1.start[j*B1.stride+k] += -B1.start[j*B1.stride+i] * B2.start[i*B2.stride+k];
}}
void row_op(const block &B1, const block &B2) {
    for(int i=0; i < B2.width; i++)
        for(int j=i+1; j < B2.width; j++)
            for(int k=0; k < B1.width; k++)
                B1.start[j*B1.stride+k] += -B2.start[j*B2.stride+i] * B1.start[i*B1.stride+k];
}
void inner_op(const block &B1, const block &B2, const block &B3) {
    for(int i=0; i < B3.width; i++)
        for(int j=0; j < B1.height; j++)
            for(int k=0; k < B2.width; k++)
                B1.start[j*B1.stride+k] += -B3.start[j*B3.stride+i] * B2.start[i*B2.stride+k];
}
```

LU Decomposition

- Each operation is put into a function, and the core logic (without tasks) is shown below

```
void LU(int num_blocks) {
    for(int i=0; i<num_blocks; i++) {
        diag_op( block_list[i][i] );
        for(int j=i+1; j<num_blocks; j++){
            row_op( block_list[i][j], block_list[i][i] );
            col_op( block_list[j][i], block_list[i][i] );
        }
        for(int j=i+1; j<num_blocks; j++) {
            for(int k=i+1; k<num_blocks; k++) {
                inner_op( block_list[j][k], block_list[i][k],
                                            block_list[j][i] );
            }
        }
    }
}
```

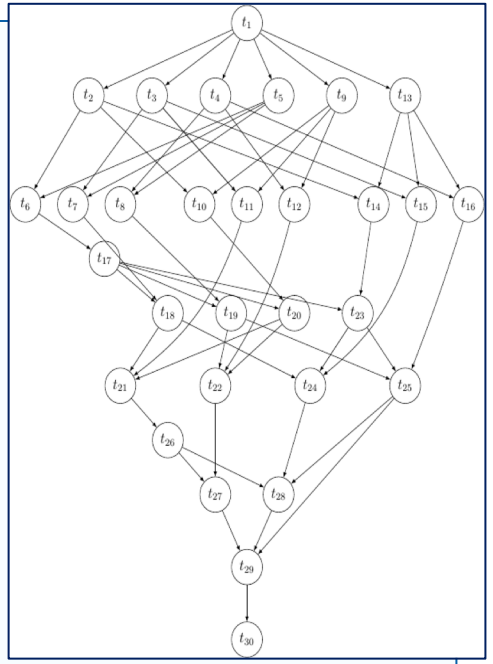
LU Decomposition: Taskwait




```
for(int i=0; i<num_blocks; i++) {
#pragma omp task diag_op( block_list[i][i] );
    #pragma omp taskwait
    for(int j=i+1; j<num_blocks; j++) {
#pragma omp task
        row_op( block_list[i][j], block_list[i][i] );
#pragma omp task
        col_op( block_list[j][i], block_list[i][i] );
    }
    #pragma omp taskwait
    for(int j=i+1; j<num_blocks; j++) {
        for(int k=i+1; k<num_blocks; k++) {
#pragma omp task depend(   in: block_list[i][k], block_list[j][i]) \
                 depend(inout: block_list[j][k])
            inner_op( block_list[j][k], block_list[i][k],
                                        block_list[j][i] );
        }
    }
    #pragma omp taskwait

}
#pragma omp taskwait
```

LU Decomposition: dependencies

```
for(int i=0; i<num_blocks; i++) {
#pragma omp task depend(inout: block_list[i][i])
    diag_op( block_list[i][i] );
    for(int j=i+1; j<num_blocks; j++) {
#pragma omp task depend(in   : block_list[i][i]) \
                 depend(inout: block_list[i][j])
        row_op( block_list[i][j], block_list[i][i] );
#pragma omp task depend(in   : block_list[i][i]) \
                 depend(inout: block_list[j][i])
        col_op( block_list[j][i], block_list[i][i] );
    }
    for(int j=i+1; j<num_blocks; j++) {
        for(int k=i+1; k<num_blocks; k++) {
#pragma omp task depend(   in: block_list[i][k], block_list[j][i]) \
                 depend(inout: block_list[j][k])
            inner_op( block_list[j][k], block_list[i][k],
                                        block_list[j][i] );
        }
    }
}
#pragma omp taskwait
```

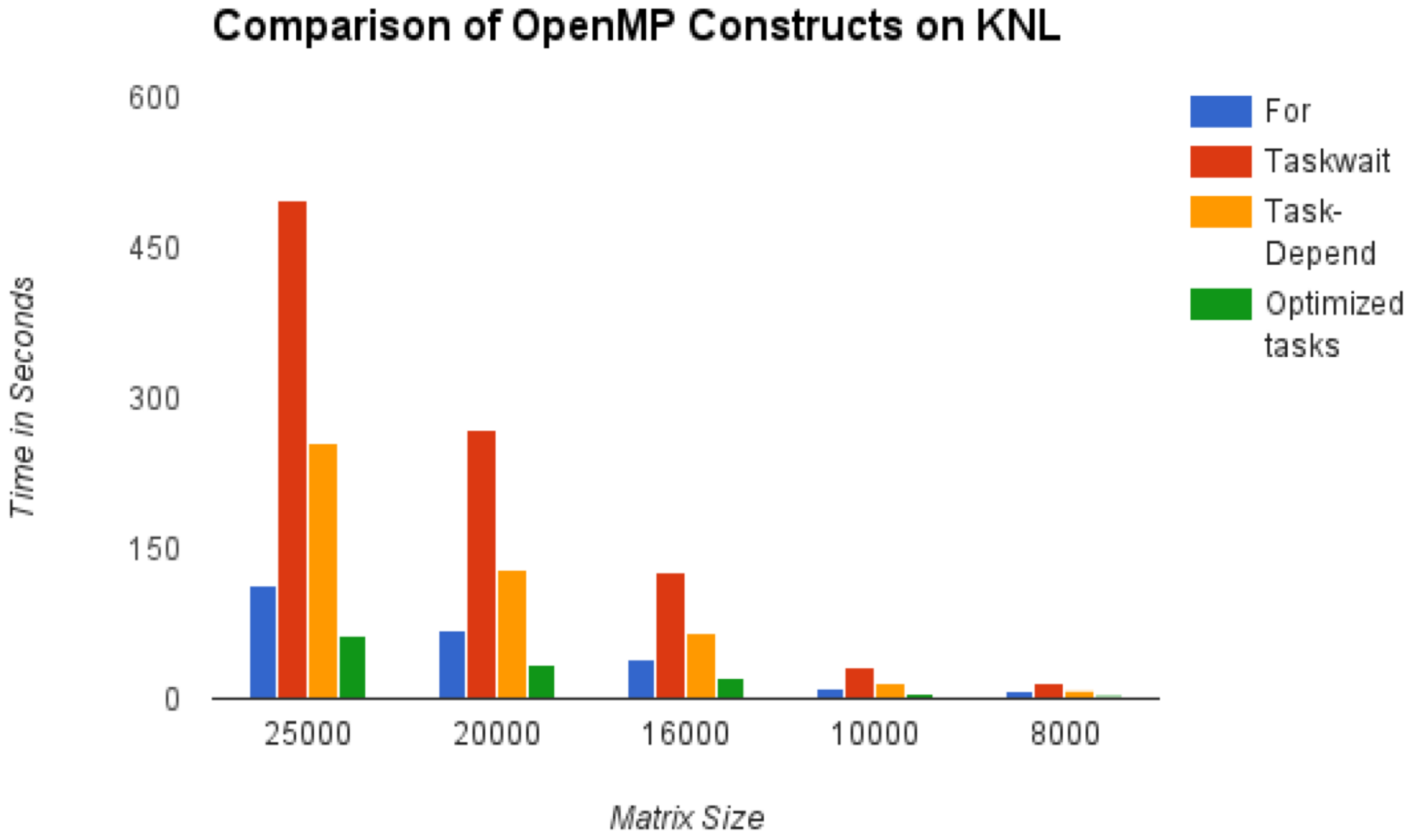


LU Decomposition: Cache Oblivious Algorithm

```
void rec_diag(int iter, int mat_size) {
    int half = mat_size/2;
    if(half == nesting_size_cutoff) {
#pragma omp task depend( inout: block_list[iter][iter])
        rec_diag (iter, half);
#pragma omp task depend( in: block_list[iter][iter]) \
                 depend( inout: block_list[iter][iter+half])
        rec_row  (iter, iter+half, half);
#pragma omp task depend( in: block_list[iter][iter]) \
                 depend( inout: block_list[iter+half][iter])
        rec_col  (iter, iter+half, half);
#pragma omp task depend( in: block_list[iter][iter+half],
block_list[iter+half][iter]) \
                 depend( inout: block_list[iter+half][iter+half])
        rec_inner(iter, iter+half, iter+half, half);
#pragma omp task depend( inout: block_list[iter+half][iter+half])
        rec_diag (iter+half, half);
    } else if(mat_size == 1) {
        diag_op(block_list[iter][iter]);
    } else {
        rec_diag (iter, half);
        rec_row  (iter, iter+half, half);
        rec_col  (iter, iter+half, half);
        rec_inner(iter, iter+half, iter+half, half);
        rec_diag (iter+half, half);
    }
}
```

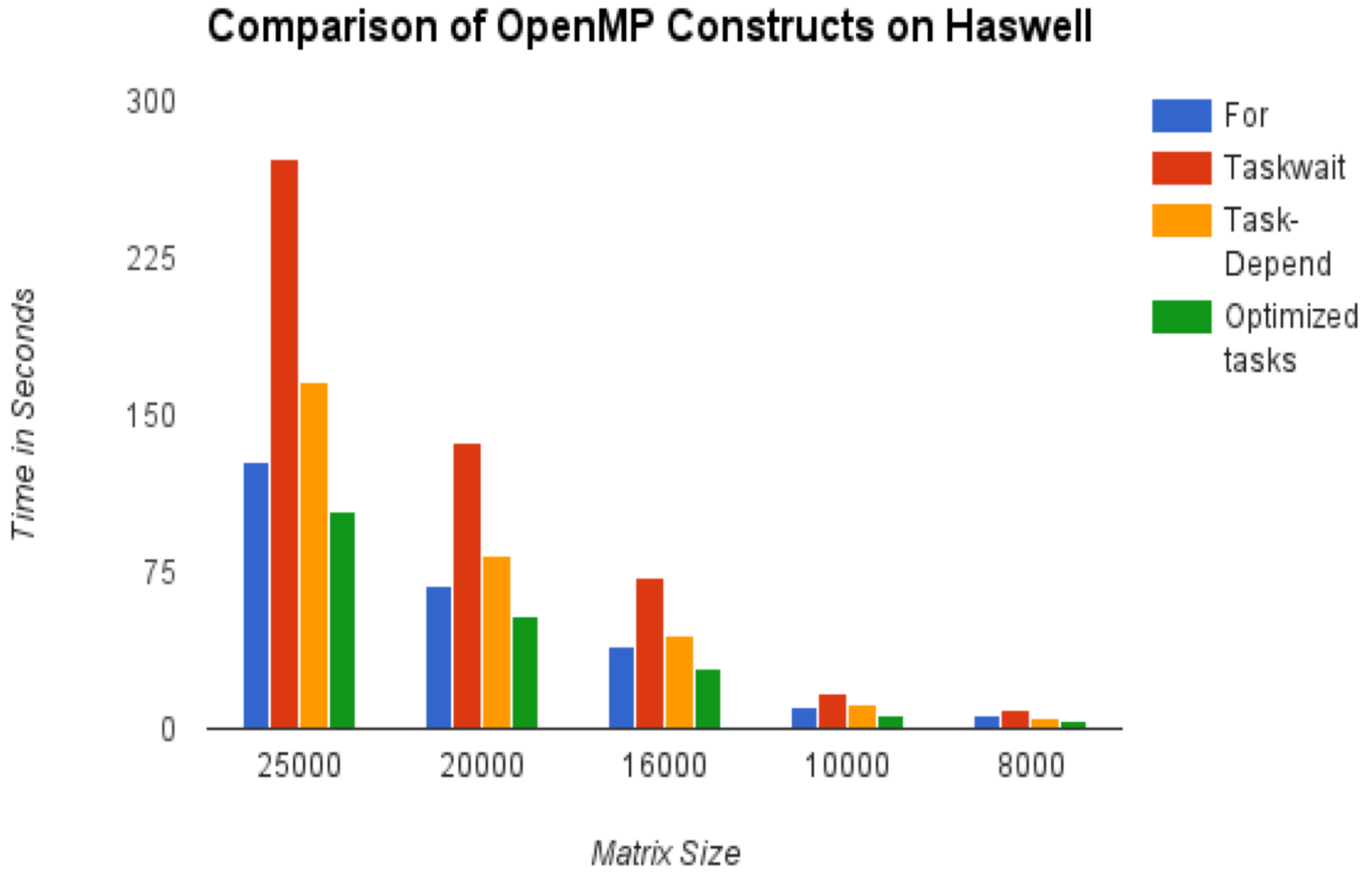
LU Decomposition: performance comparisons

The recursive cache oblivious version improves performance substantially.



LU Decomposition: Performance comparisons

The recursive cache oblivious version improves performance substantially.



Task definitions

- Task: a specific instance of executable code and its data environment.
- Task region: all the code encountered during the execution of a task.
- When a task construct is encountered by a thread, the generated task may be:
  - Deferred: executed by some thread independently of generating task.
  - Undeferred: completes execution before the generating task continues.
  - Included: Undeferred and executed by the thread that encounters the task construct.
- Tasks once started may suspend, wait, and restart.
  - Tied tasks: if a thread is suspended, the same thread will restart the thread at a later time.
  - Untied tasks: if a task is suspended, any thread in the binding team may restart the thread at a later time.

The task construct (OpenMP 4.5)

#pragma omp task *[clause[[,]clause]...]*
structured-block

Generates an explicit task

where *clause* is one of the following:

if(*[* **task :]***scalar-expression***)**

untied

default(shared | none)

private(*list***)**

firstprivate(*list***)**

shared(*list***)**

final(scalar-expression)

mergeable

depend(*dependence-type* **:** *list***)**

priority(*priority-value***)**

The evolution of the task construct

OpenMP 3.0

OpenMP 3.1

OpenMP 4.0

OpenMP 4.5

The task construct: the newer/rarely used clauses

untied	The created task, if suspended, can be executed by a different thread
final(scalar-expression)	If the scalar-expression is true, generated tasks are undeferred and execute immediately by the encountering thread.
mergeable	The task is mergable if it is undeferred and included (i.e. uses the parent tasks data environment).
priority(*priority-value*)	Gives a hint to the compiler to schedule tasks with a larger priority value (>0) before tasks with a lower value.

OpenMP 3.0 OpenMP 3.1 OpenMP 4.0 OpenMP 4.5

Waiting for tasks to complete

```
#pragma omp taskwait
```

OpenMP 3.0

Causes current task region to suspend and wait for completion of all the child tasks created before the taskwait to complete

- A standalone directive
- Defines a task scheduling point

```
#pragma omp taskgroup
        structured-block
```

OpenMP 4.0

A thread encounters the taskgroup construct. It executes the code in the structured block.
That thread suspends and waits at the end of the taskgroup region until all child tasks and any of their descendant tasks are complete.

Task switching

- Consider the following example ... Where the program may generate so many tasks that the internal data structures managing tasks overflow.

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

- Solution … Task switching; Threads can switch to other tasks at certain points called ***thread scheduling*** points.
- With Task switching, a thread can
  - Execute an already generated task … to “drain the task pool”
  - Execute the encountered task immediately (instead of deferring task execution for later)

Explicit task scheduling

#pragma omp taskyield

OpenMP 3.1

Tells the OpenMP runtime that the current task can be suspended in favor of execution of a different task

- A standalone directive
- Defines an explicit task scheduling point

A function that only one task at a time can execute (mutual exclusion)

```
#include <omp.h>
void something_useful ( void );
void mutual_excl_op( void );
void foo ( omp_lock_t * lock, int n )
{   for (int  i = 0; i < n; i++ )
    #pragma omp task
    {    something_useful();
         while ( !omp_test_lock(lock) ) {
                #pragma omp taskyield
          }
          mutual_excl_op();
          omp_unset_lock(lock);
     }
}
```

Grab a lock if you can, return if you can't

Tell the runtime it can suspend current task and schedule another

Release the lock that protected mutual_excl_op()

Task scheduling Points

- Task switching can only occur at Task Scheduling points.
- Task scheduling points happen …
  - After generation of an explicit task
  - After completion of a task region
  - In a taskyield region
  - In a taskwait region
  - At the end of a taskgropup or barrier
  - In and around regions associated with target constructs (not discussed here).
- At a task scheduling point, *any of* the following can happen for any tasks bound to the current team
  - Begin execution of a tied or untied task
  - Resume any suspended task (tied or untied)

Task Scheduling Details

- An included task is executed immediately after generation of the task
- Scheduling of new tied tasks is constrained by the set of task regions that are currently tied to the thread, and that are not suspended in a barrier region.
  - If this set is empty, any new tied task may be scheduled.
  - Otherwise, a new tied task may be scheduled only if it is a descendent task of every task in the set.
- A dependent task shall not be scheduled until its task dependences are fulfilled.
- When an explicit task is generated by a construct containing an if clause for which the expression evaluated to false, and the previous constraints are already met, the task is executed immediately after generation of the task.

Task Execution around task scheduling points

- Think of a task as a set of “task regions” between task scheduling points
- Each “task region” executes uninterrupted from start to end in the order they are encountered.
- A correct program must behave correctly and consistently with all conceivable scheduling sequences that are compatible with the rules above.
  - If multiple “task regions” between scheduling points modify values in threadprivate storage, a data race is produced and the state of threadprivate storage is not defined.
  - Lock acquire and release in different task regions may break program-order lock protocols and deadlock.

Conclusion

- We have covered the core features of OpenMP used for “traditional” regular applications on shared memory systems:
  - Thread creation and Synchronization
  - Manging the Data environment
  - Loop work share constructs
- We’ve covered :newer” features to support more irregular applications:
  - Tasks
  - Divide and conquer algorithms
- There are some additional features we did not cover at all:
  - Target directives for coprocessors and GPU’s
  - SIMD constructs for vectorization

OpenMP is growing to meet the needs of modern HPC platforms.

OpenMP and MPI together provide the lasting software foundation HPC programmers need to run on current and future platforms.

Don’t let “those other guys” with their pseudo-standards controlled by a single vendor fool you

Appendices

- Challenge Problems
- Challenge Problems: solutions
  - Monte Carlo PI and random number generators
  - Molecular dynamics
  - Matrix multiplication
  - Recursive matrix multiplication
- Mixing OpenMP and MPI
- Fortran and OpenMP
- Details on the cache oblivious LU example

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 random number generators
  2. Parallel molecular dynamics
  3. Optimizing matrix multiplication
  4. Recursive matrix multiplication algorithms

Challenge 1: Parallel Random number generators

- Go back to the monte Carlo pi program we discussed earlier when we covered threadprivate data.
- Make the parallel random number generators correct when used in parallel

Challenge 2: 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 2: 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 2 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 3: Matrix multiplication

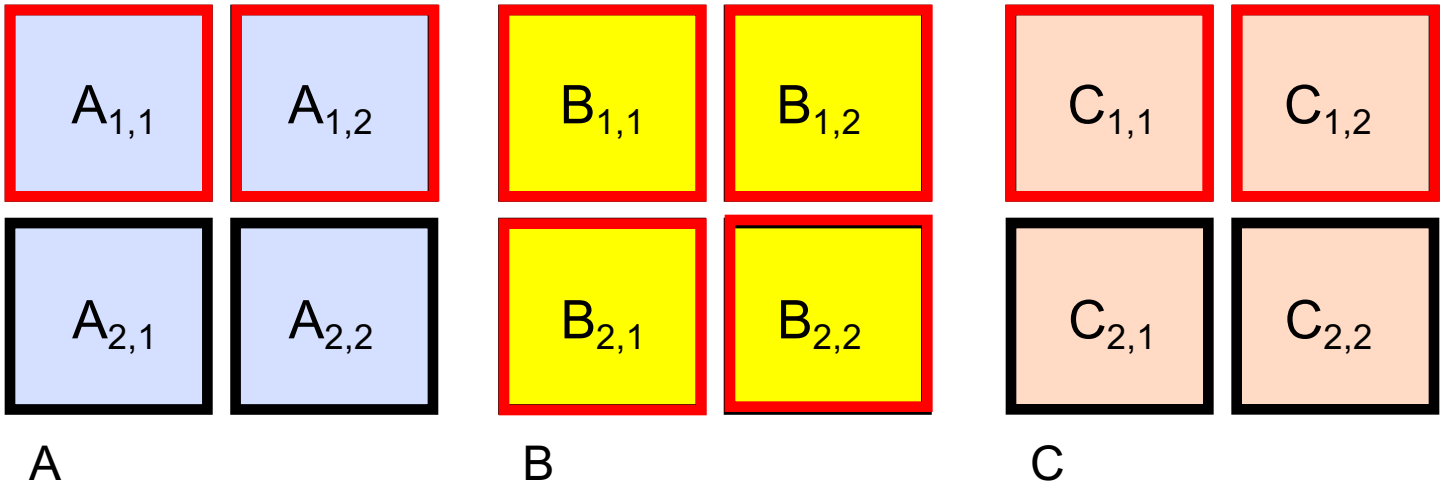
- Parallelize the matrix multiplication program in the file mm_testbed.c
- Can you optimize the program by playing with how the loops are scheduled?
- Try the following and see how they interact with the constructs in OpenMP
  - Alignment
  - Cache blocking
  - Loop unrolling
  - Vectorization
- Goal: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

Challenge 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

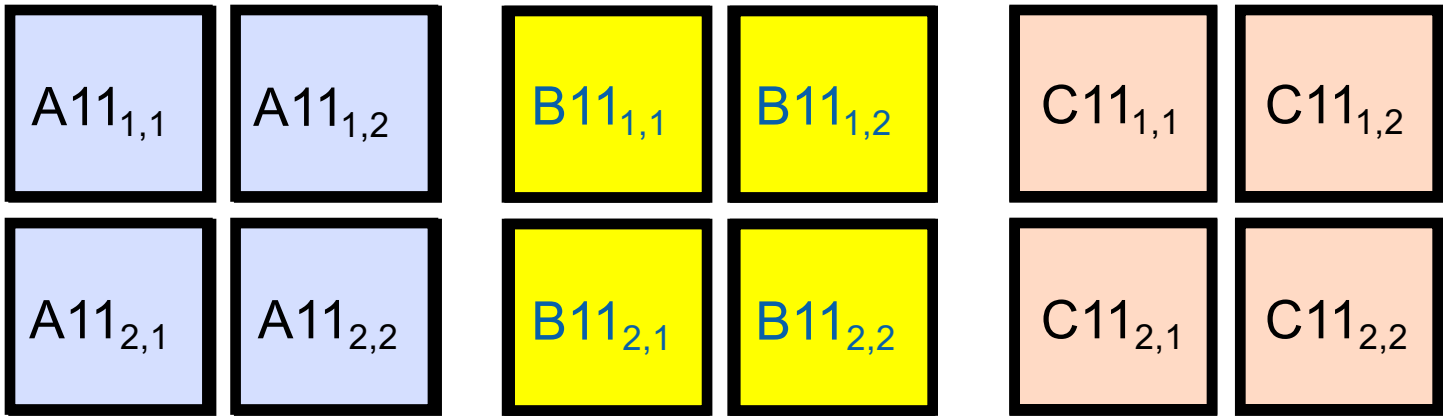


$$C_{1,1} = A_{1,1} \cdot B_{1,1} + A_{1,2} \cdot B_{2,1}$$
$$C_{2,1} = A_{2,1} \cdot B_{1,1} + A_{2,2} \cdot B_{2,1}$$

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

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

Appendices

- Challenge Problems
- Challenge Problems: solutions
  - Monte Carlo PI and random number generators
  - Molecular dynamics
  - Matrix multiplication
  - Recursive matrix multiplication
- Mixing OpenMP and MPI
- Fortran and OpenMP
- Details on the cache oblivious LU example

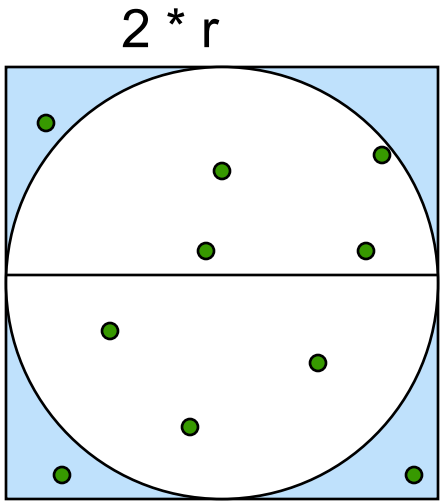
Computers and random numbers

- 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;
```

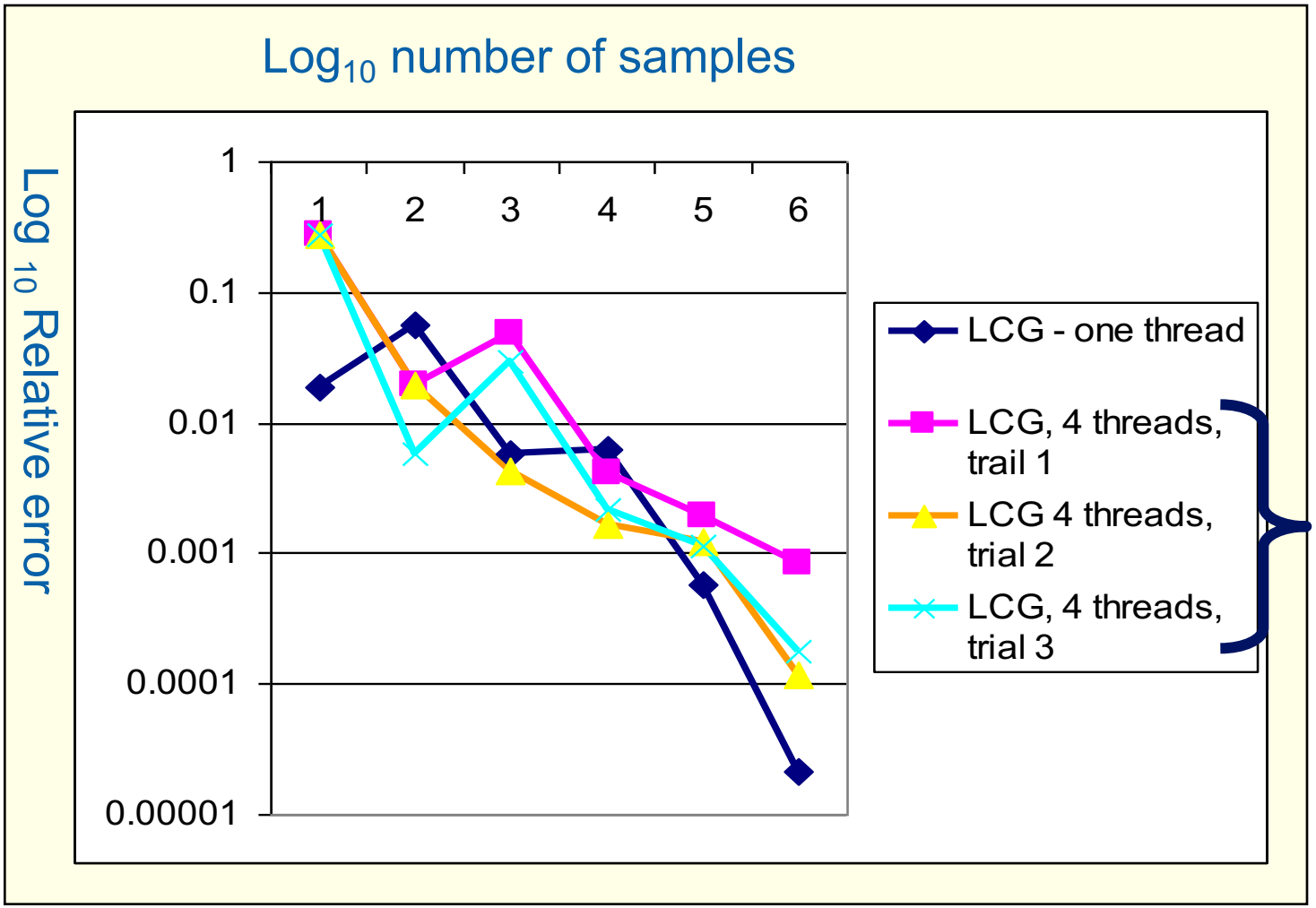
- If you pick the multiplier and addend correctly, LCG has a period of PMOD.
- Picking good LCG parameters is complicated, so look it up (Numerical Recipes is a good source). I used the following:
  - MULTIPLIER = 1366
  - ADDEND = 150889
  - PMOD = 714025

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.

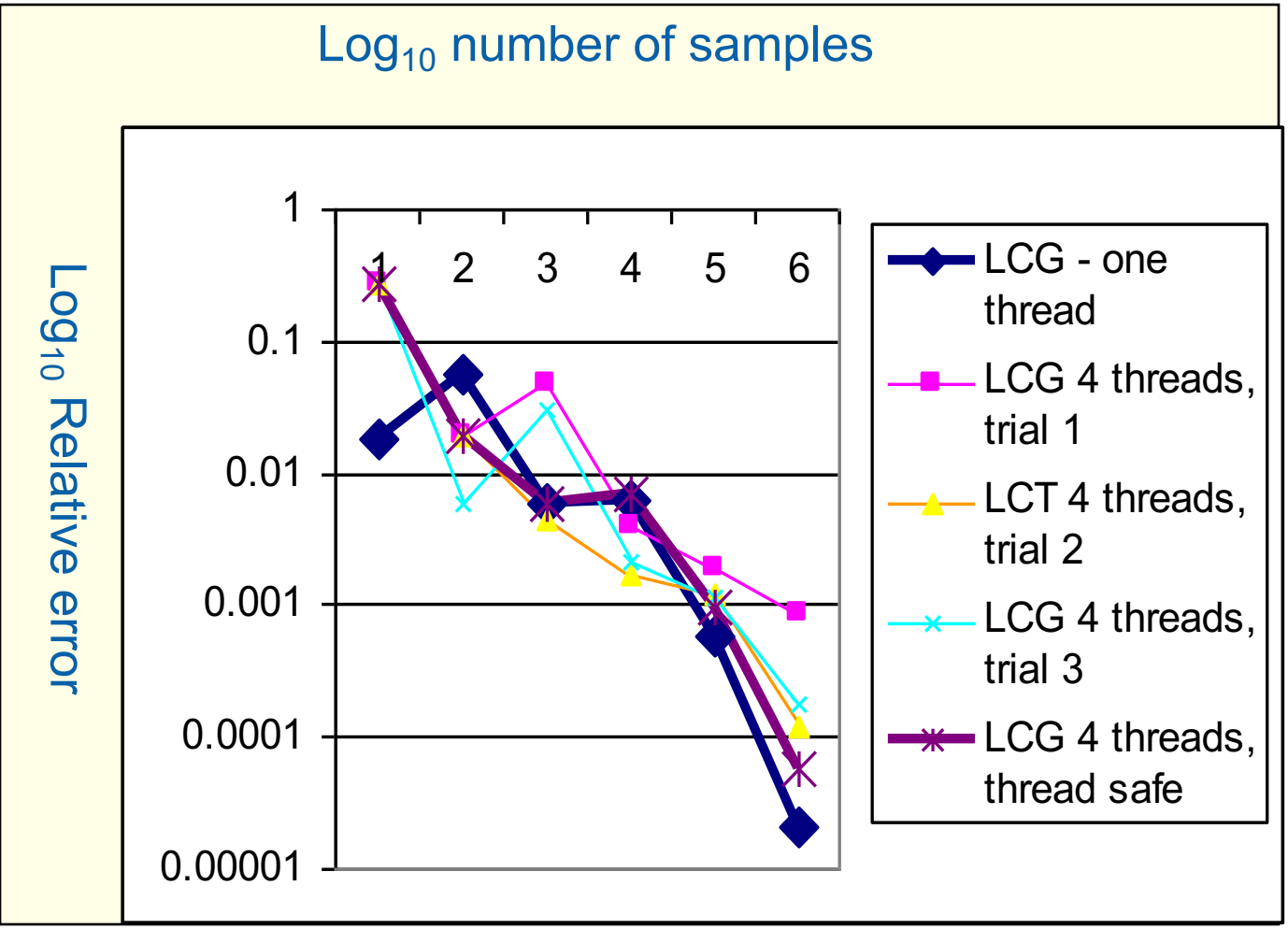
LCG code: threadsafe version

```
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 + ADDEND)% PMOD;
    random_last = random_next;

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

random_last carries state between random number computations,

To make the generator threadsafe, make random_last threadprivate so each thread has its own copy.

Thread safe random number generators



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

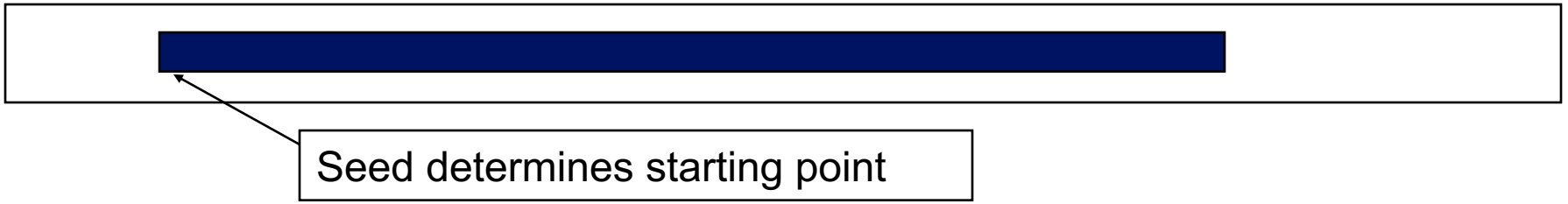
But for large number of samples, its quality is lower than the one thread result!

Why?

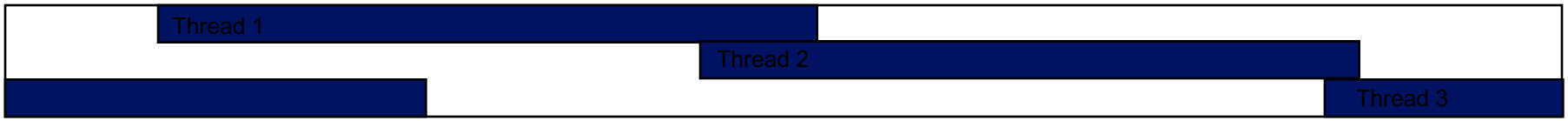
Pseudo Random Sequences

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

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



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



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

Parallel random number generators

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

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

Nice for debugging, but not really needed scientifically.

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

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

MKL Random number generators (RNG)

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

```
#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.

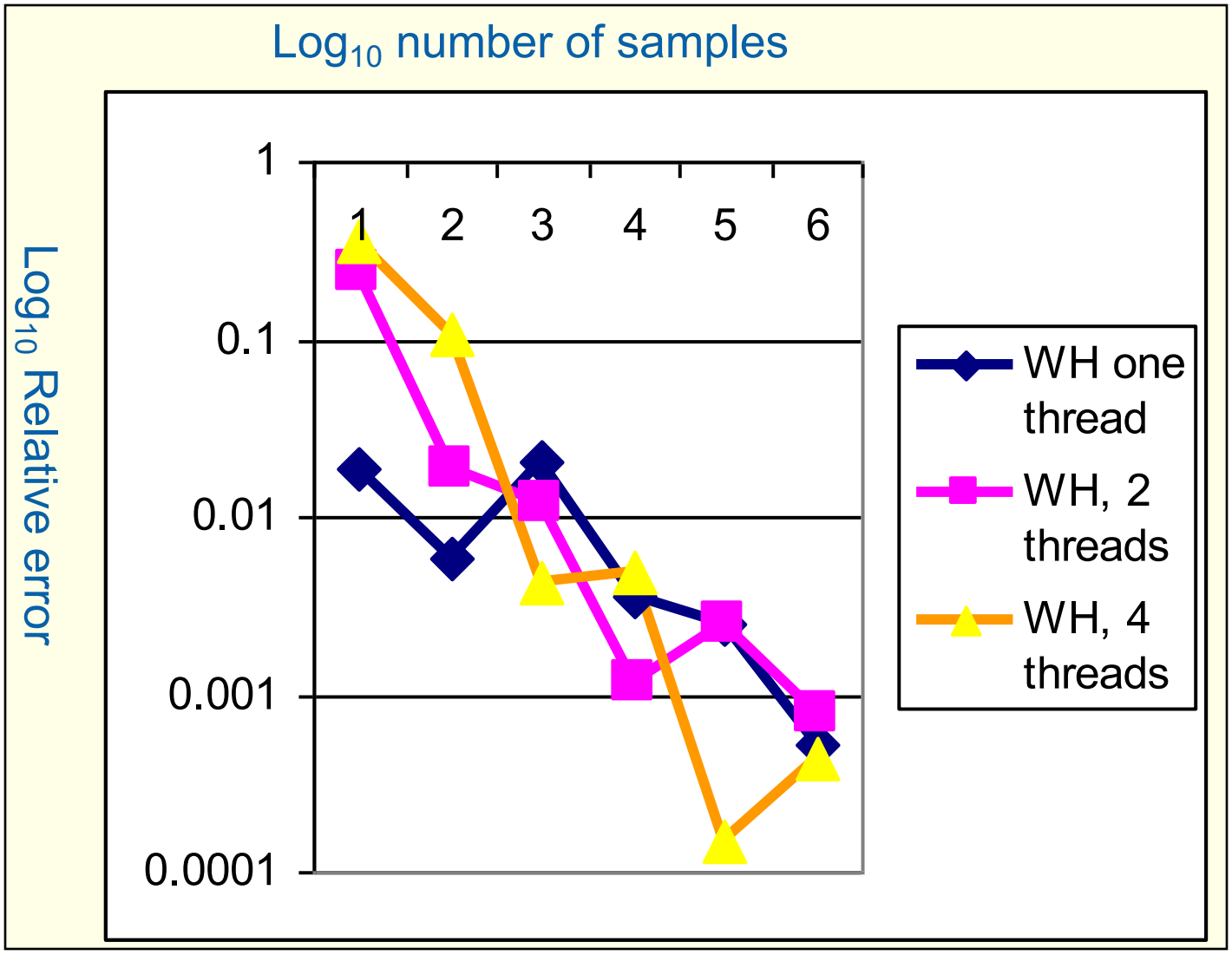
Delete the stream when you are done

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

- Challenge Problems
- Challenge Problems: solutions
  - Monte Carlo PI and random number generators
  - Molecular dynamics
  - Matrix multiplication
  - Recursive matrix multiplication
- Mixing OpenMP and MPI
- Fortran and OpenMP
- Details on the cache oblivious LU example

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

- Challenge Problems
- Challenge Problems: solutions
  - Monte Carlo PI and random number generators
  - Molecular dynamics
  - Matrix multiplication
  - Recursive matrix multiplication
- Mixing OpenMP and MPI
- Fortran and OpenMP
- Details on the cache oblivious LU example

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

- Challenge Problems
- Challenge Problems: solutions
  - Monte Carlo PI and random number generators
  - Molecular dynamics
  - Matrix multiplication
  - Recursive matrix multiplication
- Mixing OpenMP and MPI
- Fortran and OpenMP
- Details on the cache oblivious LU example

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

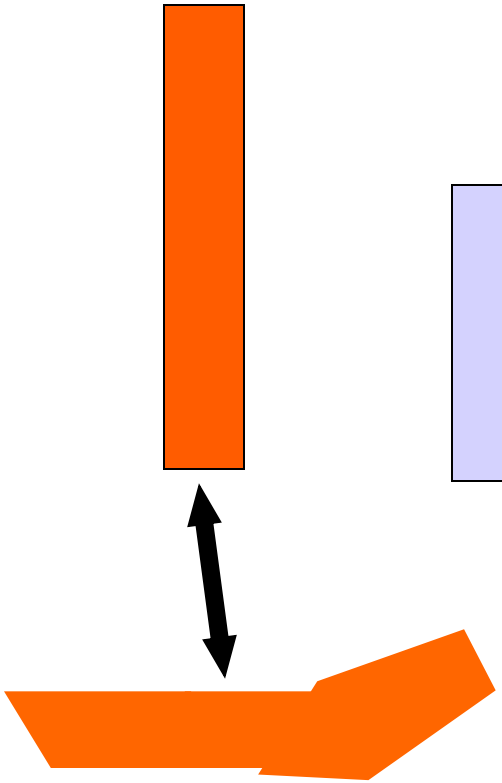
```
#define THRESHOLD 32768   // product size below which simple matmult code is called

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

- Challenge Problems
- Challenge Problems: solutions
  - Monte Carlo PI and random number generators
  - Molecular dynamics
  - Matrix multiplication
  - Linked lists
  - Recursive matrix multiplication
- Mixing OpenMP and MPI
- Fortran and OpenMP
- Details on the cache oblivious LU example

How do people mix MPI and OpenMP?

A sequential program working on a data set



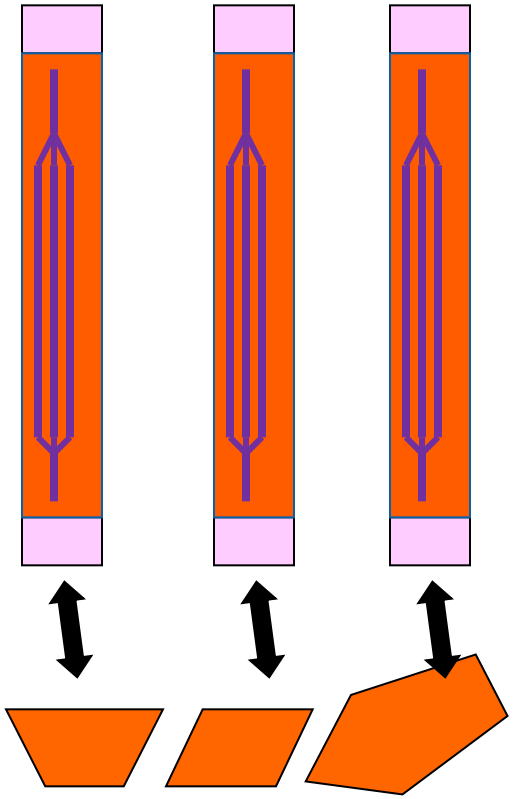
Replicate the program.

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

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

```
#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) ;
}
```

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

- Challenge Problems
- Challenge Problems: solutions
  - Monte Carlo PI and random number generators
  - Molecular dynamics
  - Matrix multiplication
  - Linked lists
  - Recursive matrix multiplication
- Mixing OpenMP and MPI
- Fortran and OpenMP
- Details on the cache oblivious LU example

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 PARALLEL
  - 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_lock_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

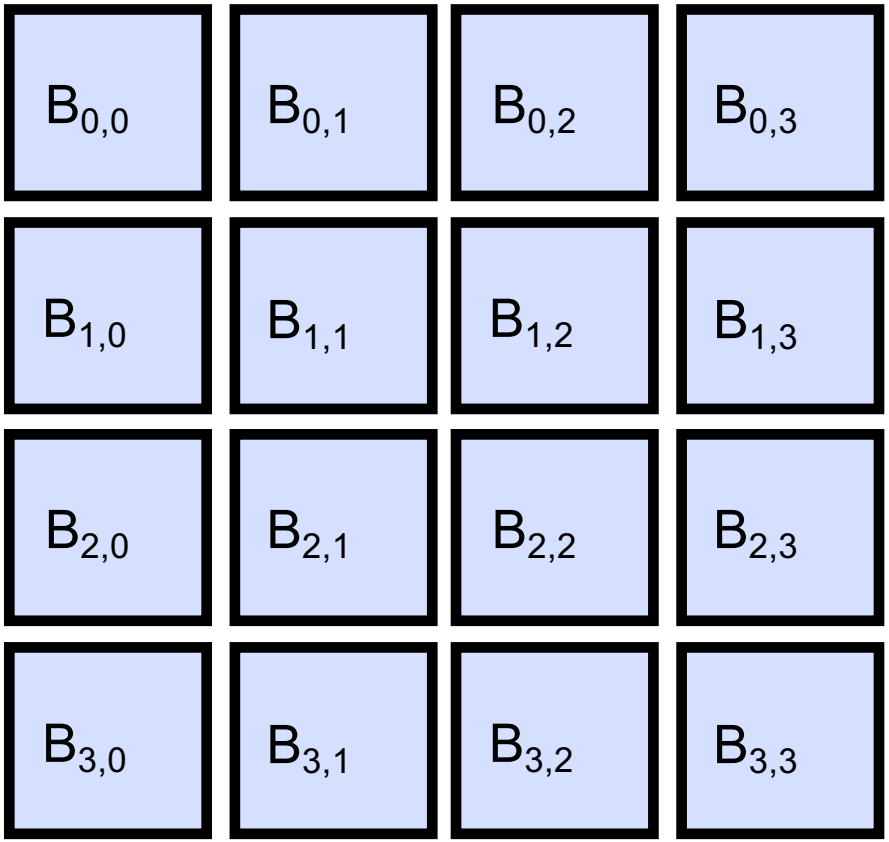
- Challenge Problems
- Challenge Problems: solutions
  - Monte Carlo PI and random number generators
  - Molecular dynamics
  - Matrix multiplication
  - Linked lists
  - Recursive matrix multiplication
- Mixing OpenMP and MPI
- Fortran and OpenMP
- Details on the cache oblivious LU example

LU Decomposition
Recursive Cache Oblivious Algorithm

- This approach forces the amount of work per task and the blocking size for the targeted cache to be the same.
- This becomes an issue on larger matrix sizes, and on architectures with smaller caches. Either the number of tasks gets very large and increases overhead, or the tasks don’t take advantage of Cache.
- A cache oblivious algorithm provides a way to control the number of tasks while still optimizing for one or more levels of cache within each task.

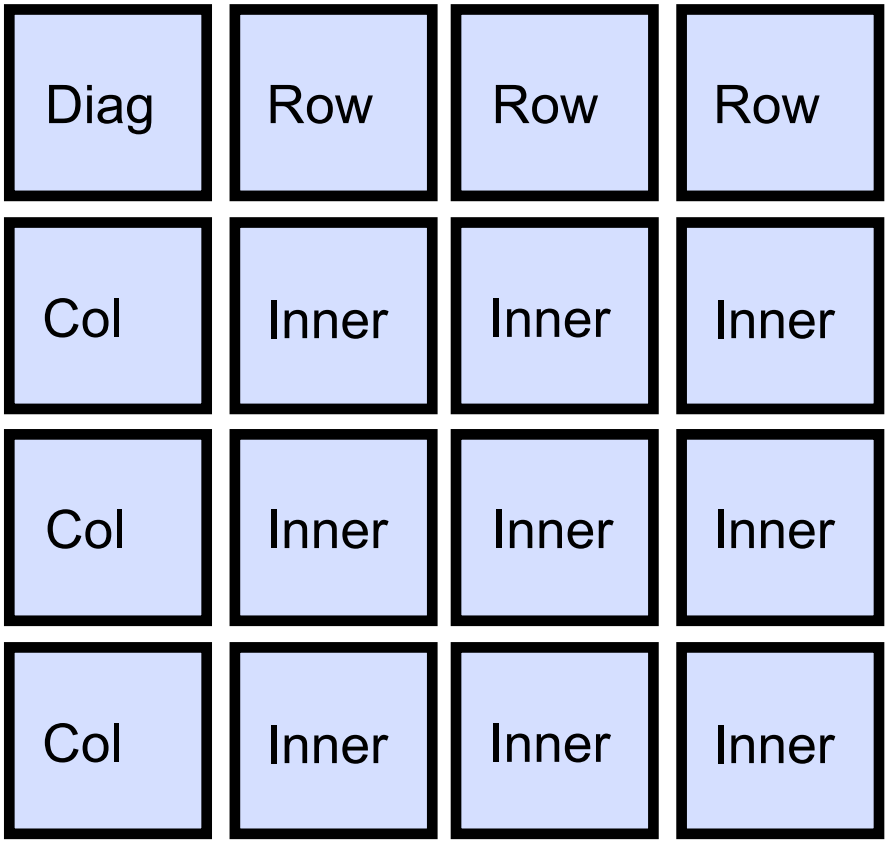
LU Decomposition
Recursive Cache Oblivious Algorithm

- To start with an example, take a matrix divided into 4x4 blocks



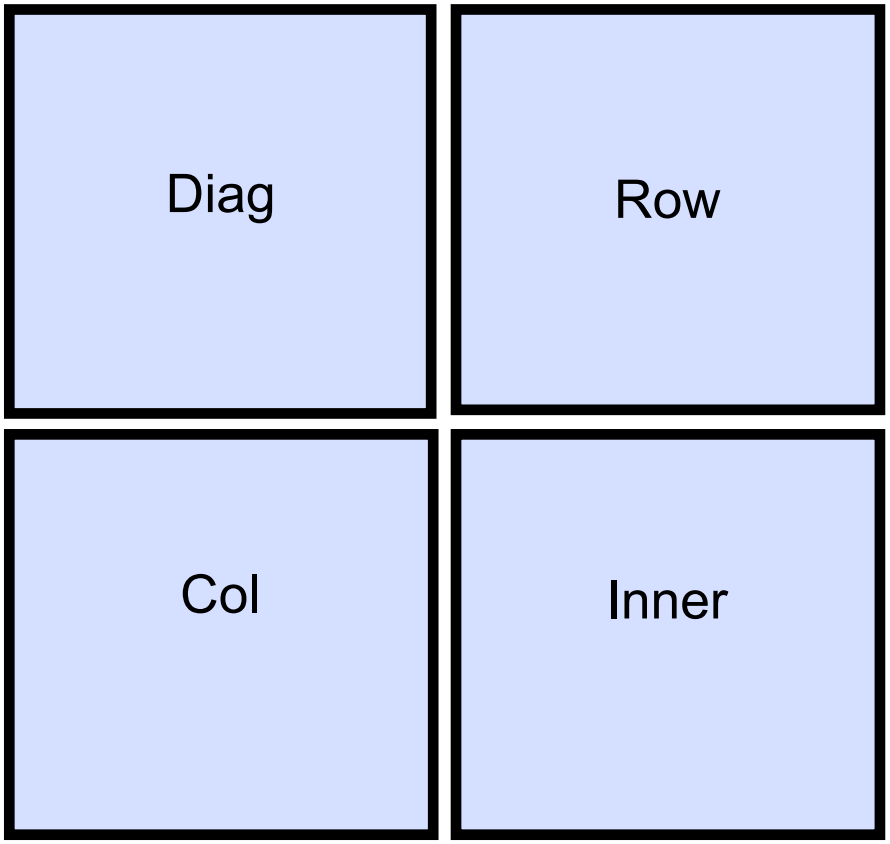
LU Decomposition
Recursive Cache Oblivious Algorithm

- The first version would go through the first iteration and create tasks for these blocks, then move on to the next iteration.



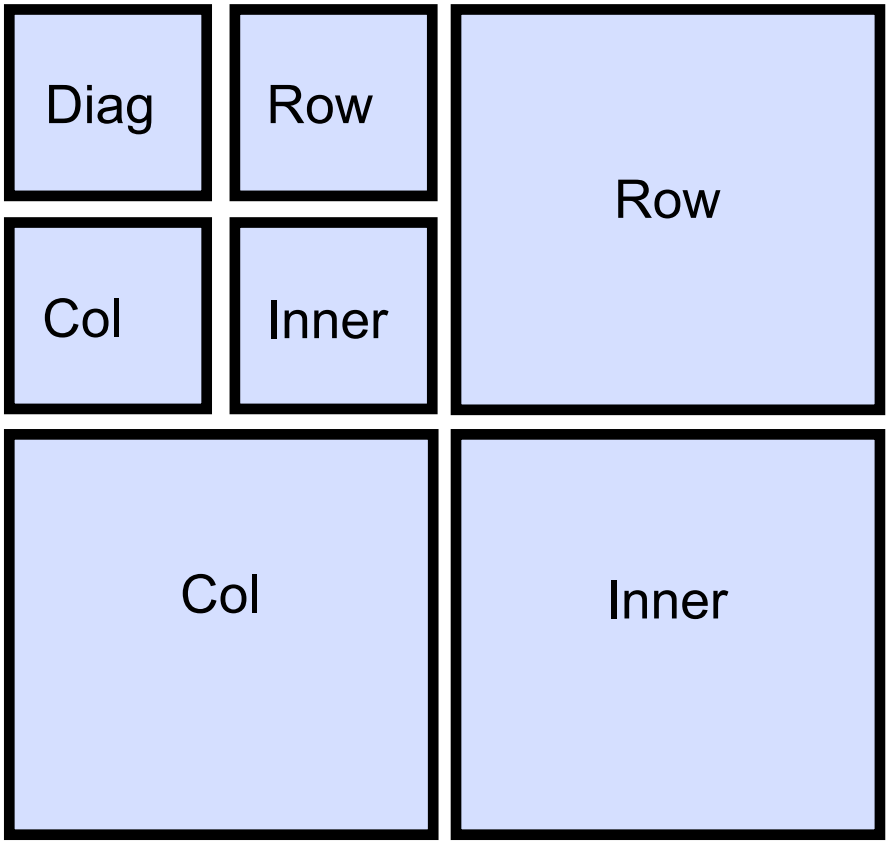
LU Decomposition
Recursive Cache Oblivious Algorithm

- The recursive version starts by calling Diag to divide the whole matrix into quadrants.
- Each of these quadrants is processed, and then Diag is called again on the output of Inner, which handles the second half of iterations.



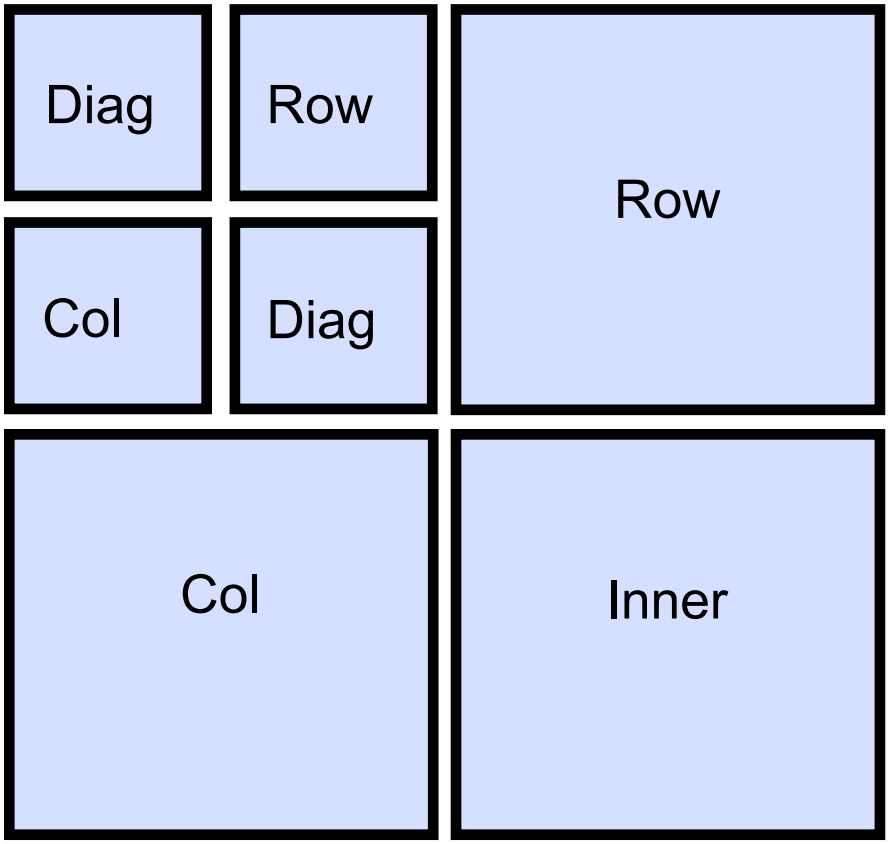
LU Decomposition
Recursive Cache Oblivious Algorithm

- Within diag, the blocks are processed as shown.



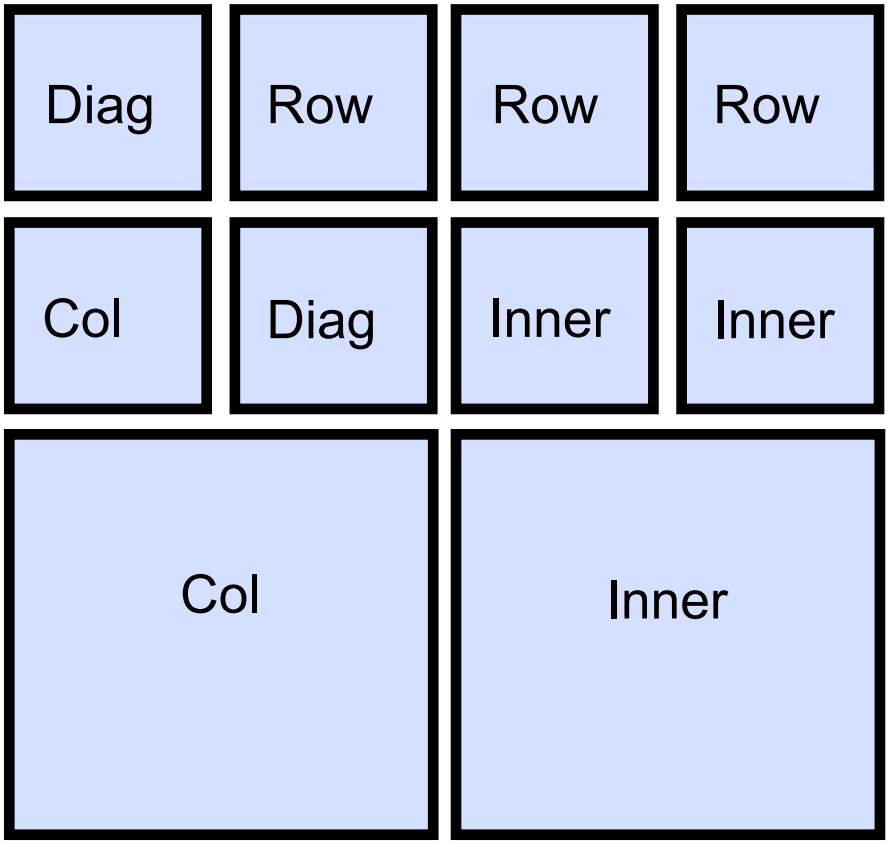
LU Decomposition
Recursive Cache Oblivious Algorithm

- Then, like mentioned earlier, diag is called again to handle the next iteration.



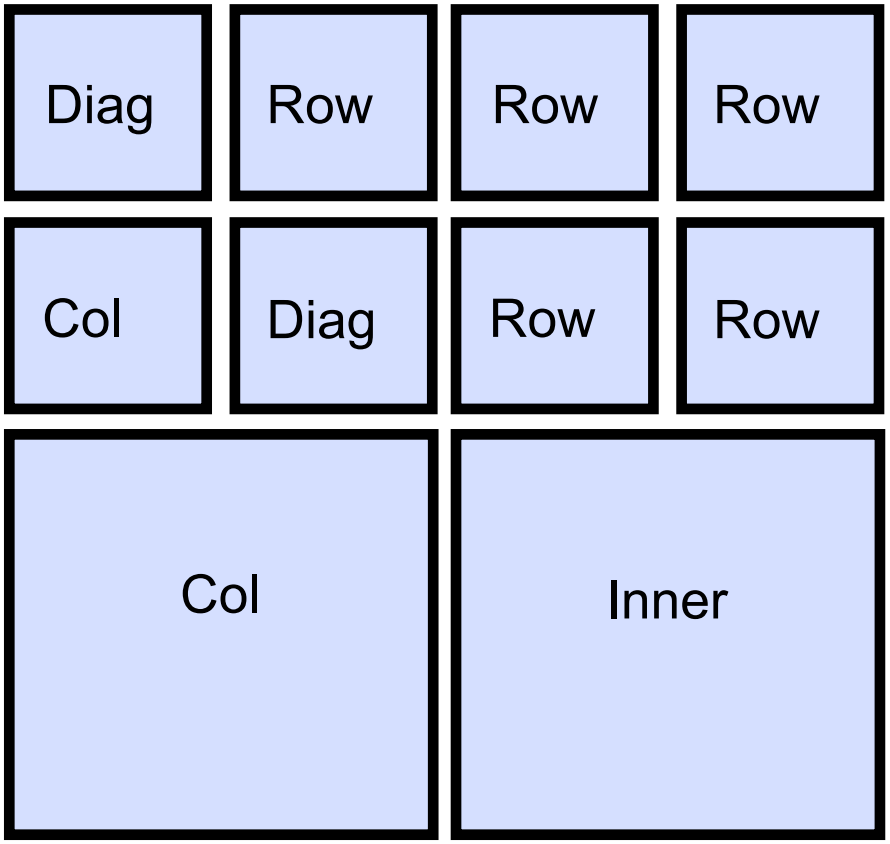
LU Decomposition
Recursive Cache Oblivious Algorithm

- Similarly, row and inner are called for the first iteration.



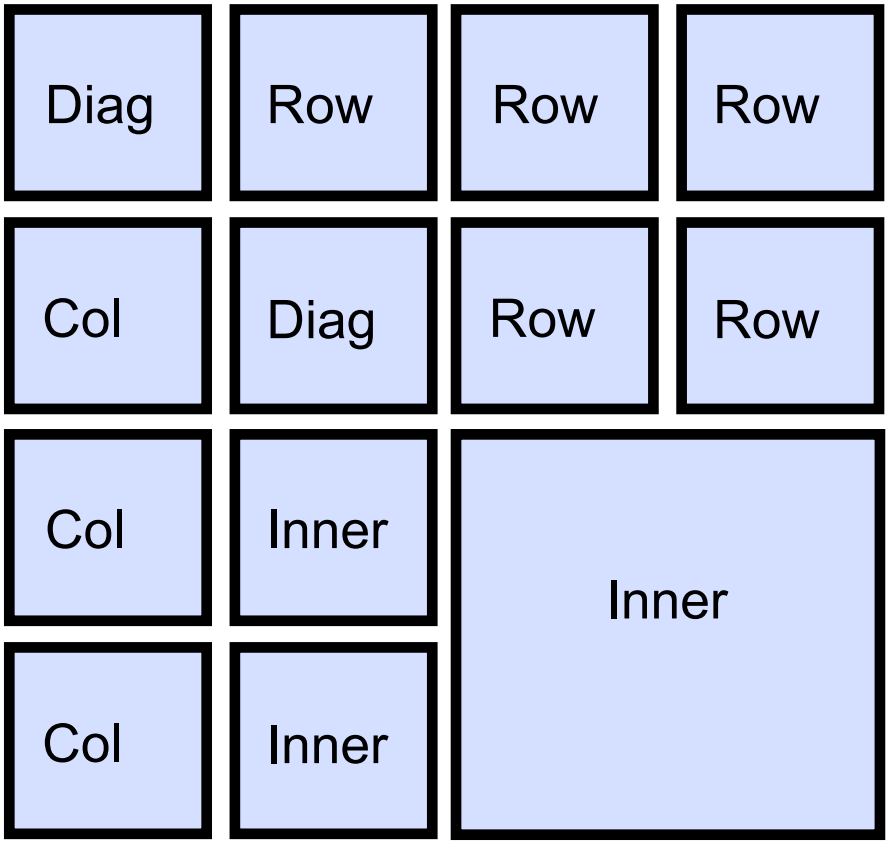
LU Decomposition
Recursive Cache Oblivious Algorithm

- Then row is called again for the second iteration.



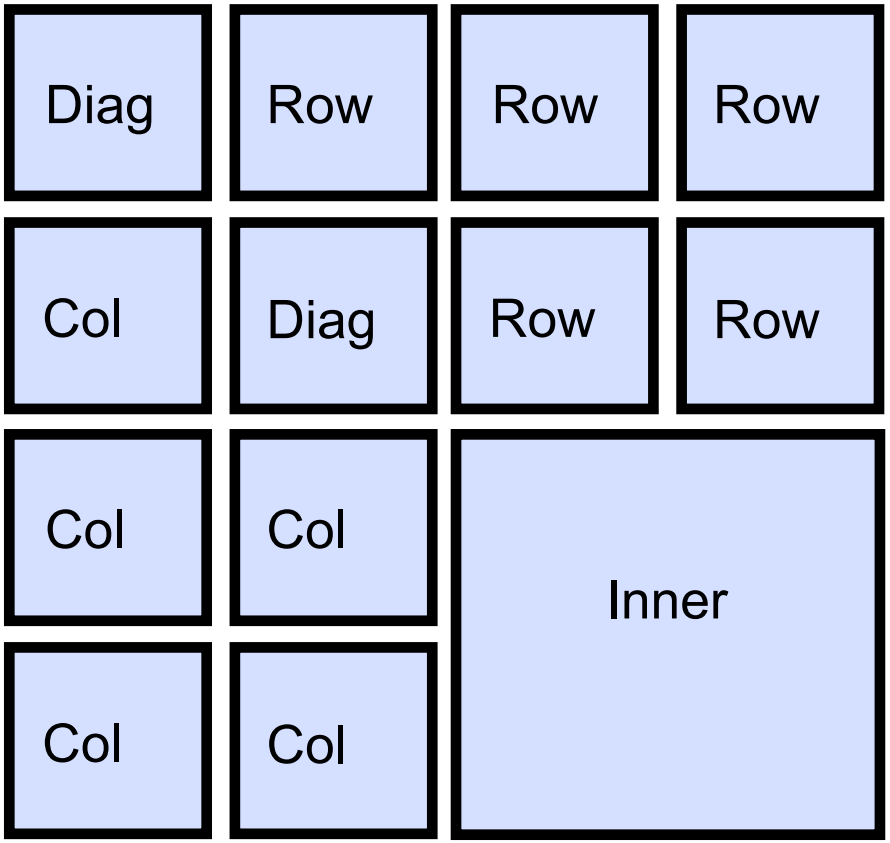
LU Decomposition
Recursive Cache Oblivious Algorithm

- Once the row quadrant is finished, the col quadrant is similarly processed.



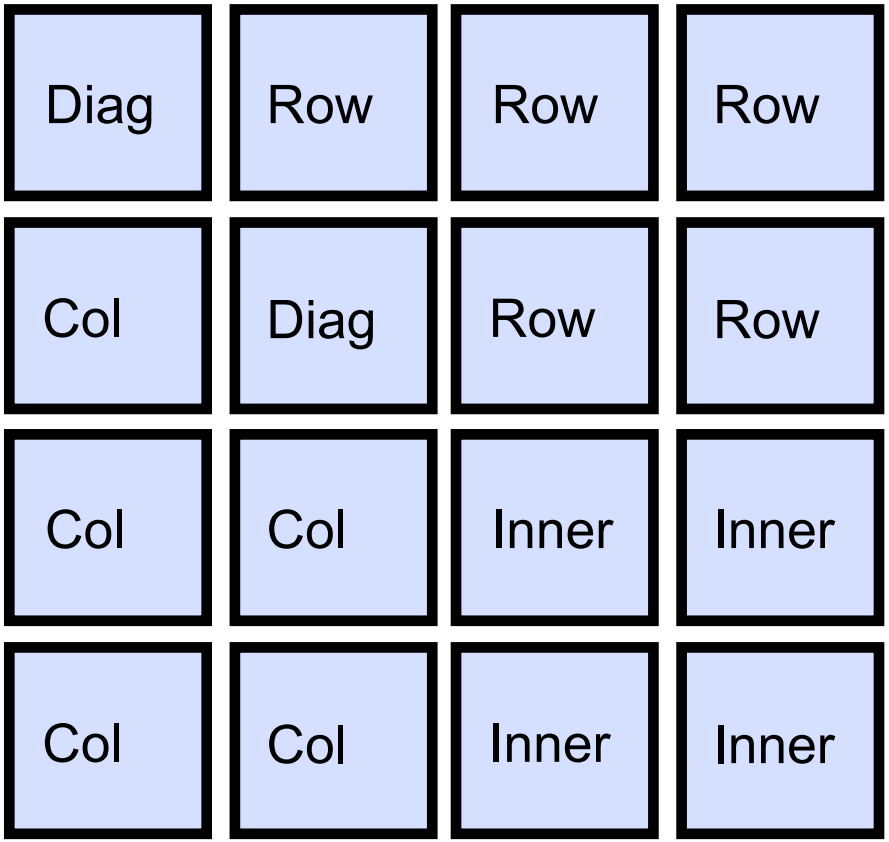
LU Decomposition
Recursive Cache Oblivious Algorithm

- And again, col is processed for the second iteration.



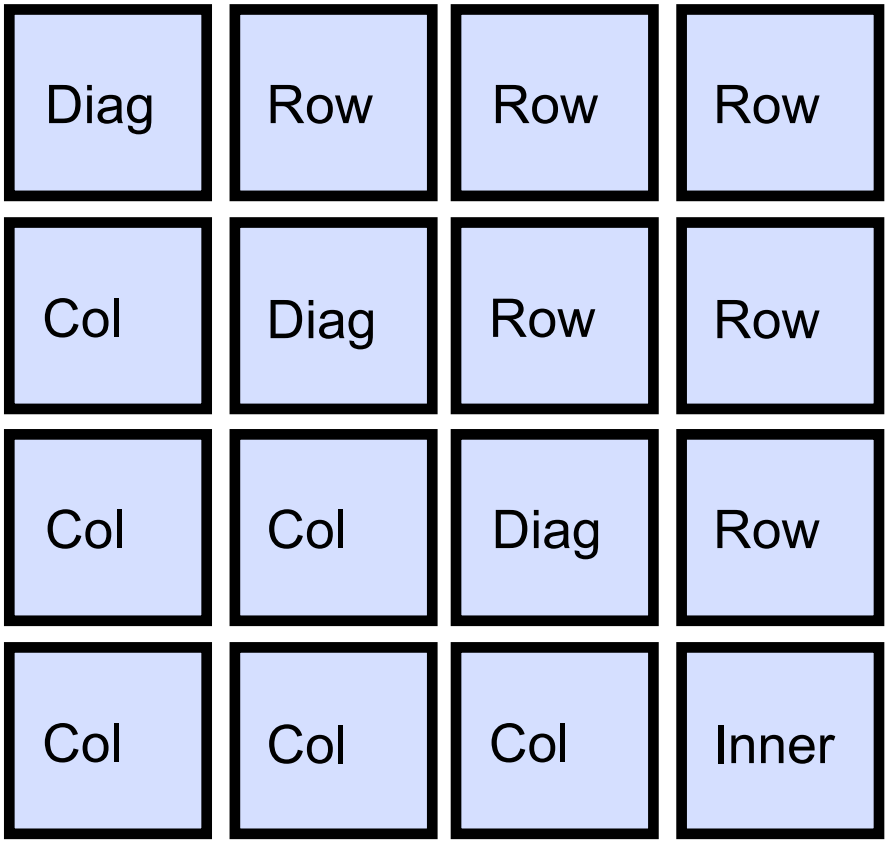
LU Decomposition
Recursive Cache Oblivious Algorithm

- Each of the blocks in inner is processed using row and column 0 for the first iteration. Then processed again using row and column 1 for the second iteration.



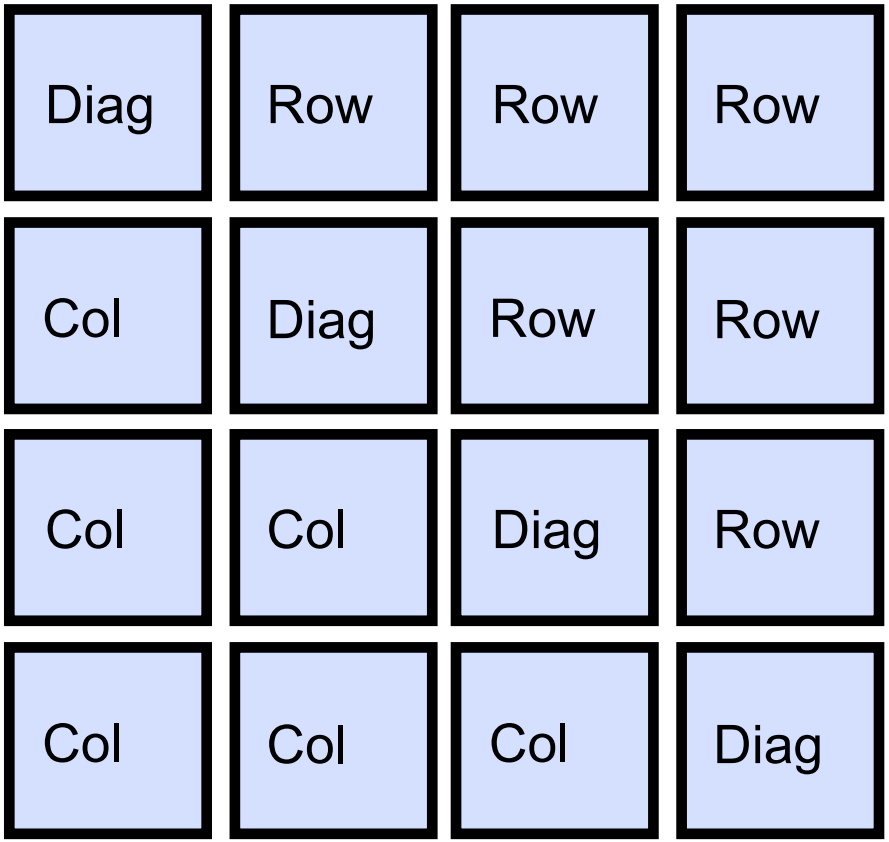
LU Decomposition
Recursive Cache Oblivious Algorithm

- Now the Inner quadrant is done and ready to be passed to diag, and perform what would be the third iteration.



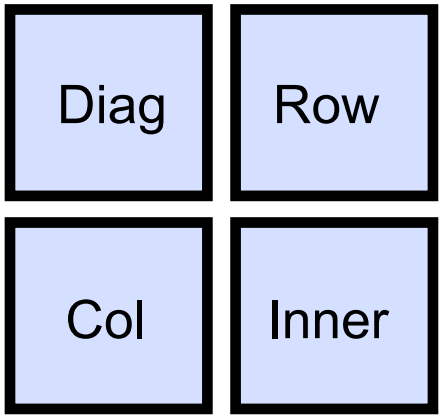
LU Decomposition
Recursive Cache Oblivious Algorithm

- And the final step is diag on the last block, for the fourth iteration.



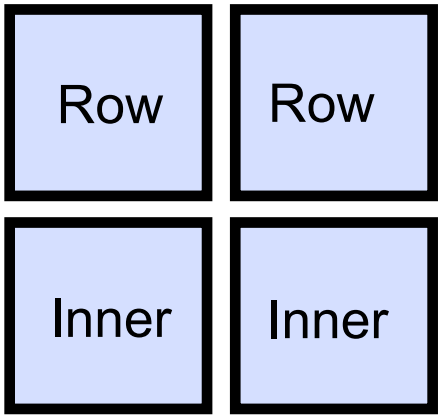
LU Decomposition
Recursive Cache Oblivious Algorithm

And now code for the serial version




```
void rec_diag(int iter, int mat_size) {
    int half = mat_size/2;
    if(mat_size == 1) {
        diag_op(block_list[iter][iter]);
    } else {
        rec_diag (iter, half);
        rec_row  (iter, iter+half, half);
        rec_col  (iter, iter+half, half);
        rec_inner(iter, iter+half, iter+half, half);
        rec_diag (iter+half, half);
    }
}
```

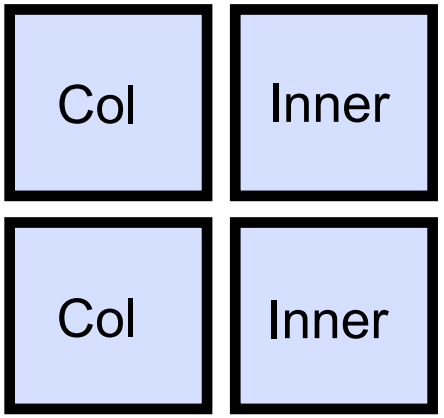
LU Decomposition
Recursive Cache Oblivious Algorithm



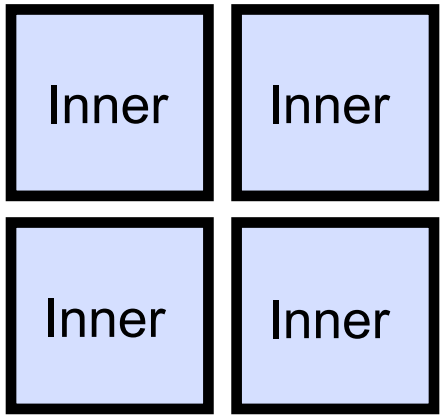

```
void rec_row(int iter, int i, int mat_size)
    int half= mat_size/2;
    if(mat_size == 1) {
        row_op(block_list[iter][i],
               block_list[iter][iter]);
    } else {
        //left side
        rec_row  ( iter, i, half);
        rec_inner( iter, iter+half, i, half);
        rec_row  ( iter+half, i, half);
        //right side
        rec_row  ( iter, i+half, half);
        rec_inner( iter, iter+half, i+half, half);
        rec_row  ( iter+half, i+half, half);
    }
}
```

LU Decomposition
Recursive Cache Oblivious Algorithm

```
void rec_col(int iter, int i, int mat_size)
    int half= mat_size/2;
    if(mat_size == 1) {
        col_op(block_list[i][iter],
               block_list[iter][iter]);
    } else {
        //top half
        rec_col   ( iter, i, half);
        rec_inner( iter, i, iter+half, half);
        rec_col   ( iter+half, i, half);
        //bottom half
        rec_col   ( iter, i+half, half);
        rec_inner( iter, i+half, iter+half, half);
        rec_col   ( iter+half, i+half, half);
    }
}
```



LU Decomposition
Recursive Cache Oblivious Algorithm




```
void rec_inner(int iter,
               int i, int j, int mat_size) {
    int half = mat_size/2;
    int offset_i = i+half;
    int offset_j = j+half;
if(mat_size == 1){
        inner_op(block_list[i][j],
                 block_list[iter][j],
                 block_list[i][iter]);
    } else {
        rec_inner( iter,        i,        j, half);
        rec_inner( iter,        i, offset_j, half);
        rec_inner( iter, offset_i,        j, half);
        rec_inner( iter, offset_i, offset_j, half);

        rec_inner( iter+half,        i,        j, half);
        rec_inner( iter+half,        i, offset_j, half);
        rec_inner( iter+half, offset_i,        j, half);
        rec_inner( iter+half, offset_i, offset_j, half);
    }
}
```

LU Decomposition
Recursive Cache Oblivious Algorithm

- Adding only tasking directives with depend the clause to this serial version would result in the program creating the same tasks as the previous version.
- In order to get the locality benefits of the cache oblivious algorithm, a cutoff is needed.

LU Decomposition
Recursive Cache Oblivious Algorithm

```
void rec_diag(int iter, int mat_size) {
    int half = mat_size/2;
    if(half == nesting_size_cutoff) {
#pragma omp task depend( inout: block_list[iter][iter])
        rec_diag (iter, half);
#pragma omp task depend( in: block_list[iter][iter]) \
                 depend( inout: block_list[iter][iter+half])
        rec_row  (iter, iter+half, half);
#pragma omp task depend( in: block_list[iter][iter]) \
                 depend( inout: block_list[iter+half][iter])
        rec_col  (iter, iter+half, half);
#pragma omp task depend( in: block_list[iter][iter+half],
block_list[iter+half][iter]) \
                 depend( inout: block_list[iter+half][iter+half])
        rec_inner(iter, iter+half, iter+half, half);
#pragma omp task depend( inout: block_list[iter+half][iter+half])
        rec_diag (iter+half, half);
    } else if(mat_size == 1) {
        diag_op(block_list[iter][iter]);
    } else {
        rec_diag (iter, half);
        rec_row  (iter, iter+half, half);
        rec_col  (iter, iter+half, half);
        rec_inner(iter, iter+half, iter+half, half);
        rec_diag (iter+half, half);
    }
}
```