OpenMP* and the fundamental design patterns of parallel programming

Tim Mattson

Intel Corp.

timothy.g.mattson@ intel.com

All materials are on github. To download them:

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

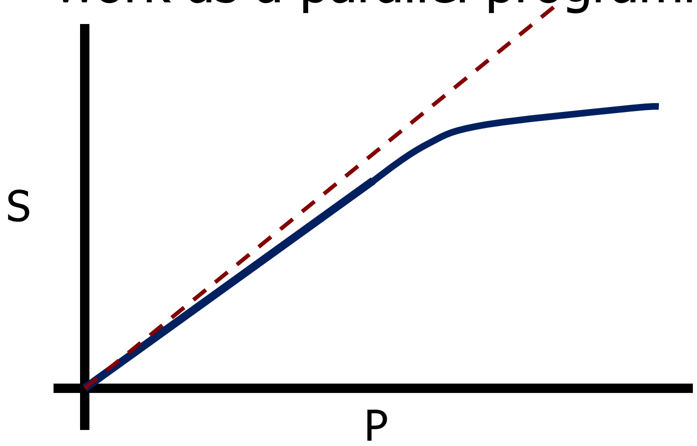
* The name “OpenMP” is the property of the OpenMP Architecture Review Board.

Introduction

I'm just a simple kayak instructor

Photo © by Greg Clopton, 2014

To support my kayaking habit I work as a parallel programmer



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

Disclaimer & Optimization Notice

INFORMATION IN THIS DOCUMENT IS PROVIDED “AS IS”. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO THIS INFORMATION INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

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.

All rights reserved. Intel, the Intel logo, Xeon, Xeon Phi, VTune, and Cilk are trademarks of Intel Corporation in the U.S. and other countries.
*Other names and brands may be claimed as the property of others.

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: 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.

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 –fopenmp << file names>>
  ```

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

- Copy the exercises to your home directory

  ```
  $ cp -r /grand/ATPESC2021/track-2b-omp  .
  ```

- 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 ATPESC2021 –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: Systems for exercises, Theta

# compile - Use cc, CC, ftn. Default compilers are Intel

```
CC -qopenmp program.C
```

# Start interactive job on one node

```
qsub–I –n 1 –t 30 –q ATPESC2021 –A ATPESC2021
```

# run on one node and ask for use of 4 threads

```
aprun -n 1 -N 1 -d 4 -j 1 -cc depth –e OMP_NUMTHREADS=4 ./a.out
```

# In batch mode, put the above in a jobscript file (job) and type:

```
qsub –n 1 –t 30 –q ATPESC2021 –A ATPESC2021 ./job_script
```

aprun options

- -n total_number_of_ranks
- -N ranks_per_node
- -d depth[number of cpus(hyperthreads) per rank]
- -cc depth   [Note: depthis a keyword]
- -j hyperthreads[cpus(hyperthreads) per compute unit (core)]

Set to one for our OpenMP exercises

Set to the number of threads available to your job

Additional examples please see https://gitlab.com/alcf/training/-/tree/master/GettingStarted/theta/omp

Also see JaeHyuk's talk from Sunday evening: http://press3.mcs.anl.gov/atpesc/files/2021/08/ATPESC-2021-Track-0-Talk-2-Kwack-Quick-Start.pdf

Preliminaries: Systems for exercises, Ascent

```
#compile
    xlC_r++ -O2 -qsmp -qoffload main.c
```

```
# start an interactive job
    bsub -W 2:00 -nnodes 1 -P GEN139 -Is $SHELL
```

```
# run the program
    jsrun -n 1 -a 1 -g 1 a.out
```

Example program build and run (both C and Fortran)

https://github.com/vlkale/OpenMP-tutorial/tree/master/offload-101

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 Revisited: Options for Mutual exclusion
  - Thread Affinity and Data Locality
  - Thread Private Data
  - Memory models and point-to-point Synchronization
  - Programming your GPU with OpenMP

OpenMP* overview:

```
C$OMP FLUSH
#pragma omp critical
C$OMP THREADPRIVATE(/ABC/)
CALL OMP SET NUM THREADS(10)
C$OM
C$OM
C$(
C
#p
ED
```

OpenMP: An API for Writing Multithreaded Applications

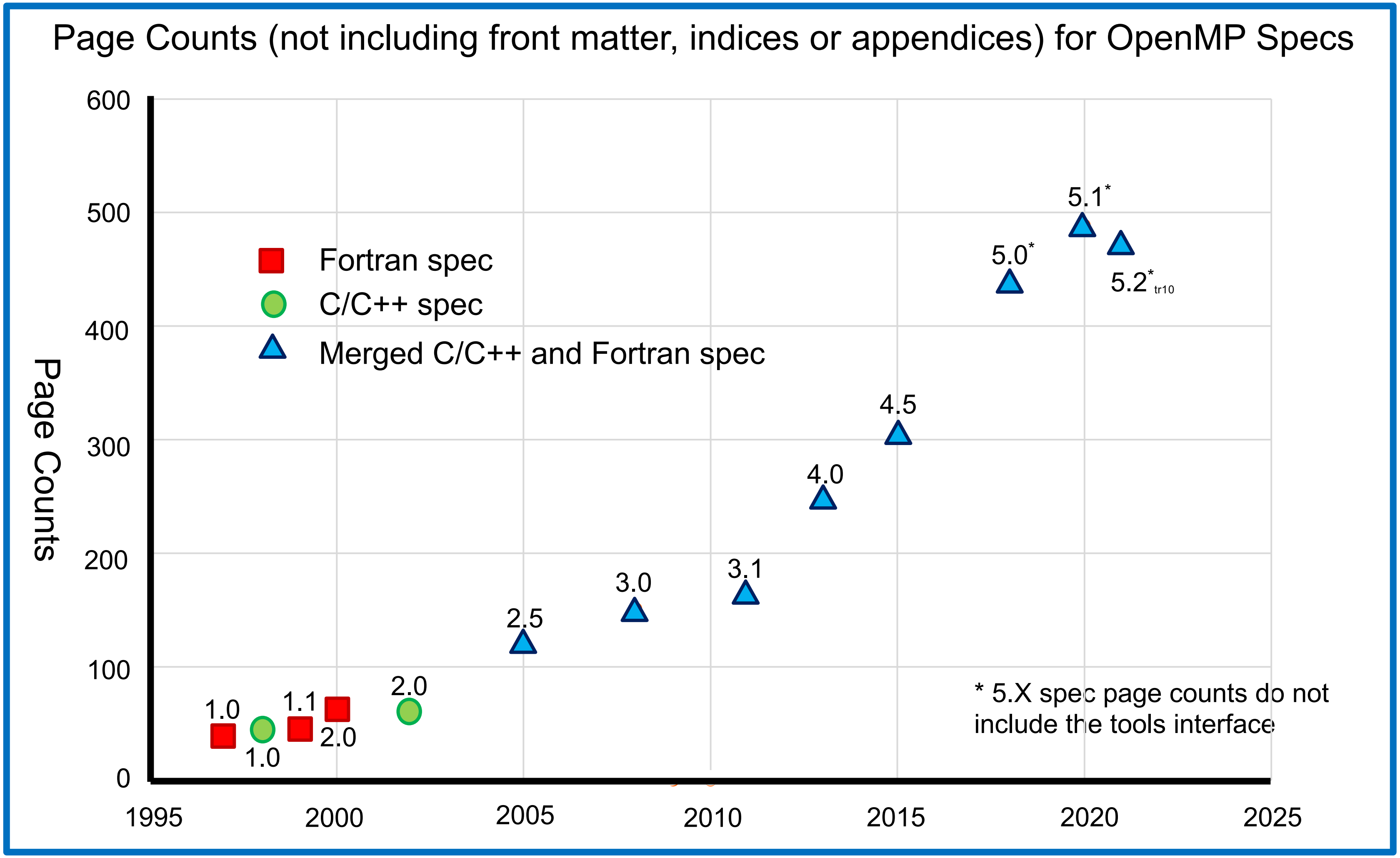
- A set of compiler directives and library routines for parallel application programmers
- Greatly simplifies writing multi-threaded (MT) programs in Fortran, C and C++
- Standardizes established SMP practice + vectorization and heterogeneous device programming

```
C$OMP PARALLEL COPYIN(/blk/)
C$OMP DO lastprivate(XX)
Nthrds = OMP_GET_NUM_PROCS()
omp_set_lock(lck)
```

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

The growth of complexity in OpenMP

- Our goal in 1997 … A simple interface for application programmers …



Page Counts (not including front matter, indices or appendices) for OpenMP Specs

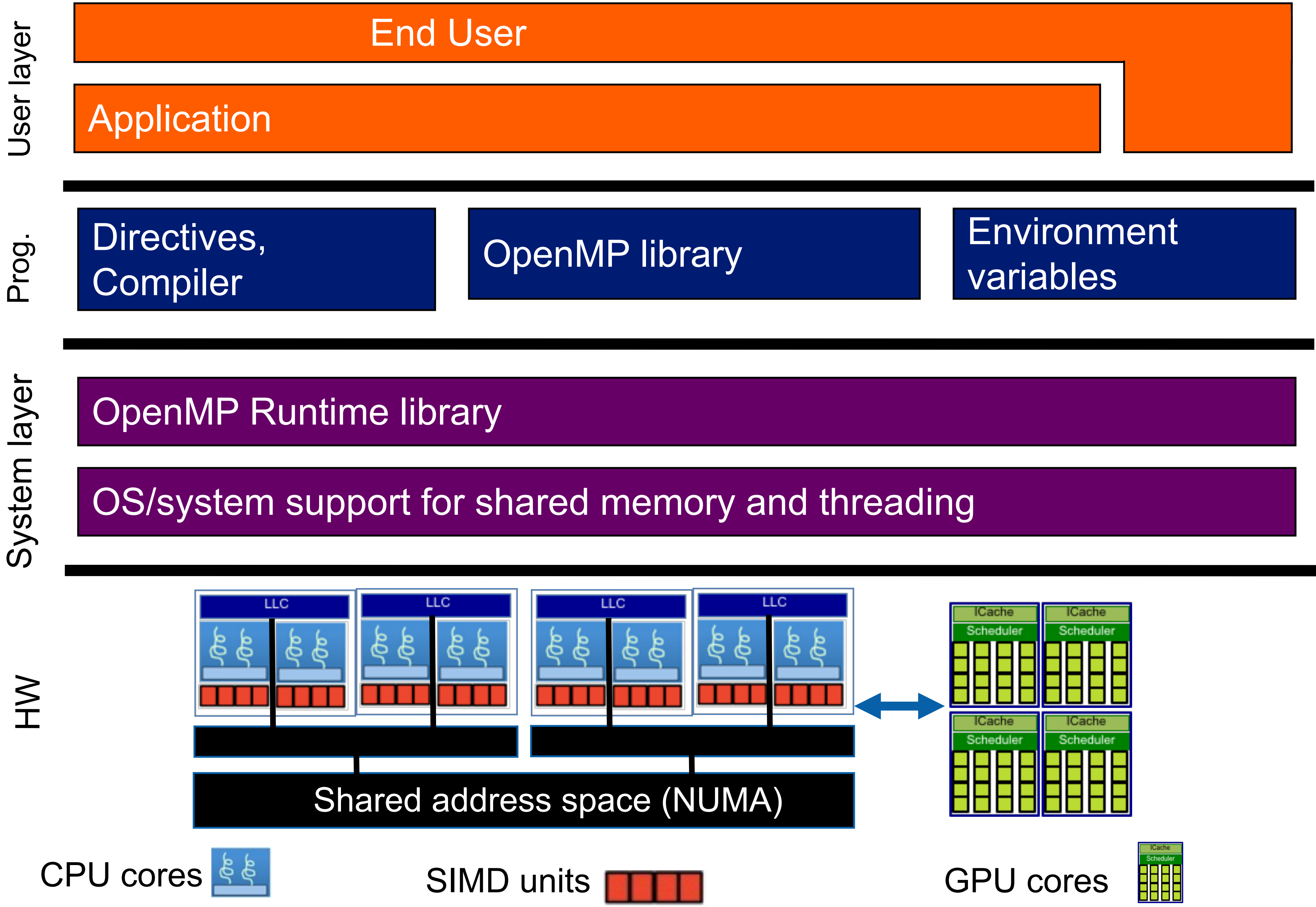
* 5.X spec page counts do not include the tools interface

- Today (In 2021) … OpenMP has become a full-featured interface for expert programmers.

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

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

OpenMP basic definitions: Basic Solution stack



User layer

End User

Application

Prog.

Directives, Compiler

OpenMP library

Environment variables

System layer

OpenMP Runtime library

OS/system support for shared memory and threading

HW

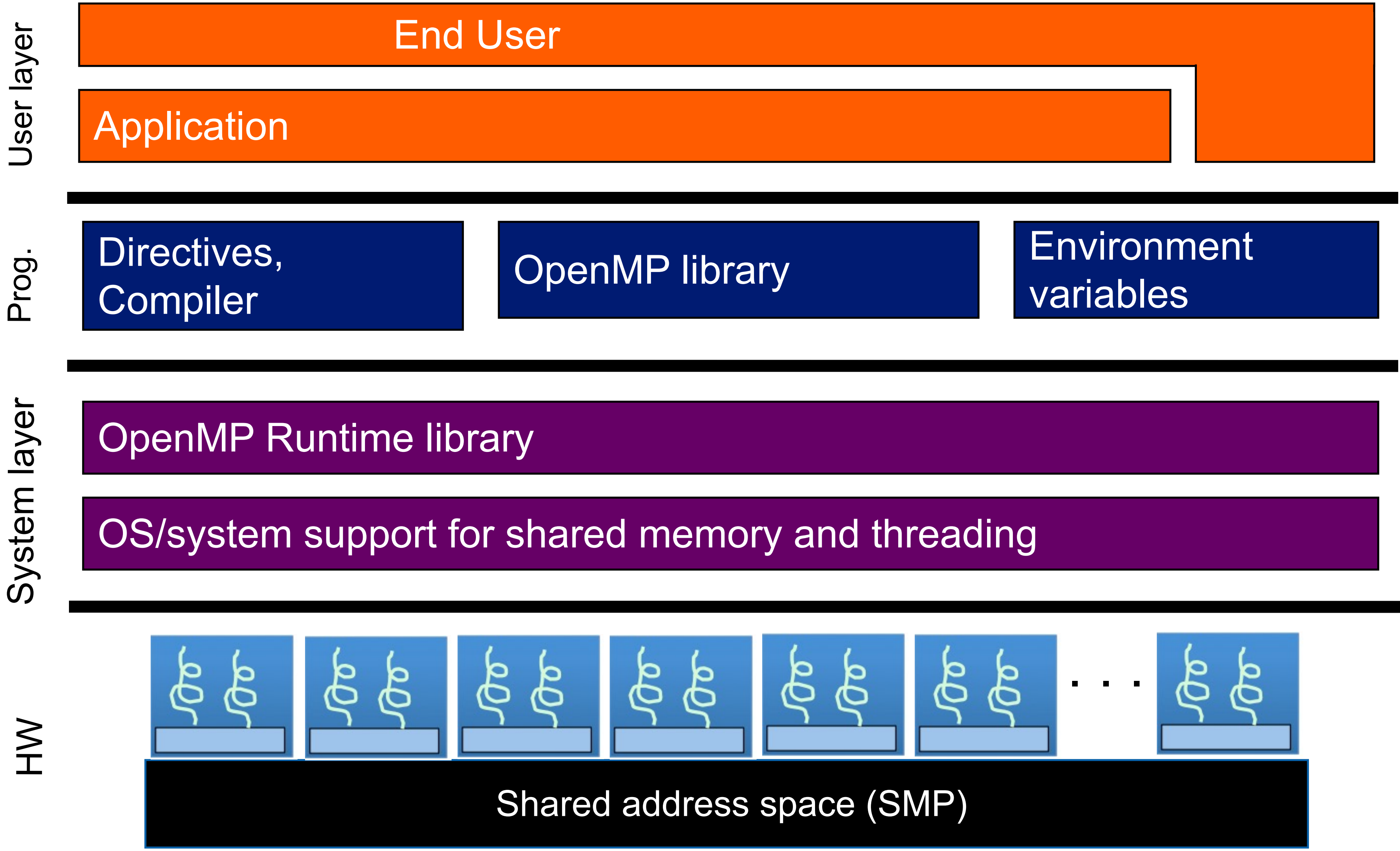
Shared address space (NUMA)

CPU cores

SIMD units

GPU cores

OpenMP basic definitions: Basic Solution stack



User layer

End User

Application

Prog.

Directives, Compiler

OpenMP library

Environment variables

System layer

OpenMP Runtime library

OS/system support for shared memory and threading

HW

. . .

Shared address space (SMP)

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.

<table>
<tr><th>C and C++</th><th>Fortran</th></tr>
<tr><td colspan="2">Compiler directives</td></tr>
<tr><td>#pragma omp construct [clause [clause]…]</td><td>!$OMP construct [clause [clause] …]</td></tr>
<tr><td colspan="2">Example</td></tr>
<tr><td>#pragma omp parallel private(x)
{
}</td><td>!$OMP PARALLEL
!$OMP END PARALLEL</td></tr>
<tr><td colspan="2">Function prototypes and types:</td></tr>
<tr><td>#include <omp.h></td><td>use OMP_LIB</td></tr>
</table>

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

Exercise, Part A: Hello world

Verify that your environment works

- Write a program that prints “hello world”.

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




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

}
```

Exercise, Part B: Hello world

Verify that your OpenMP environment works

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

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


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

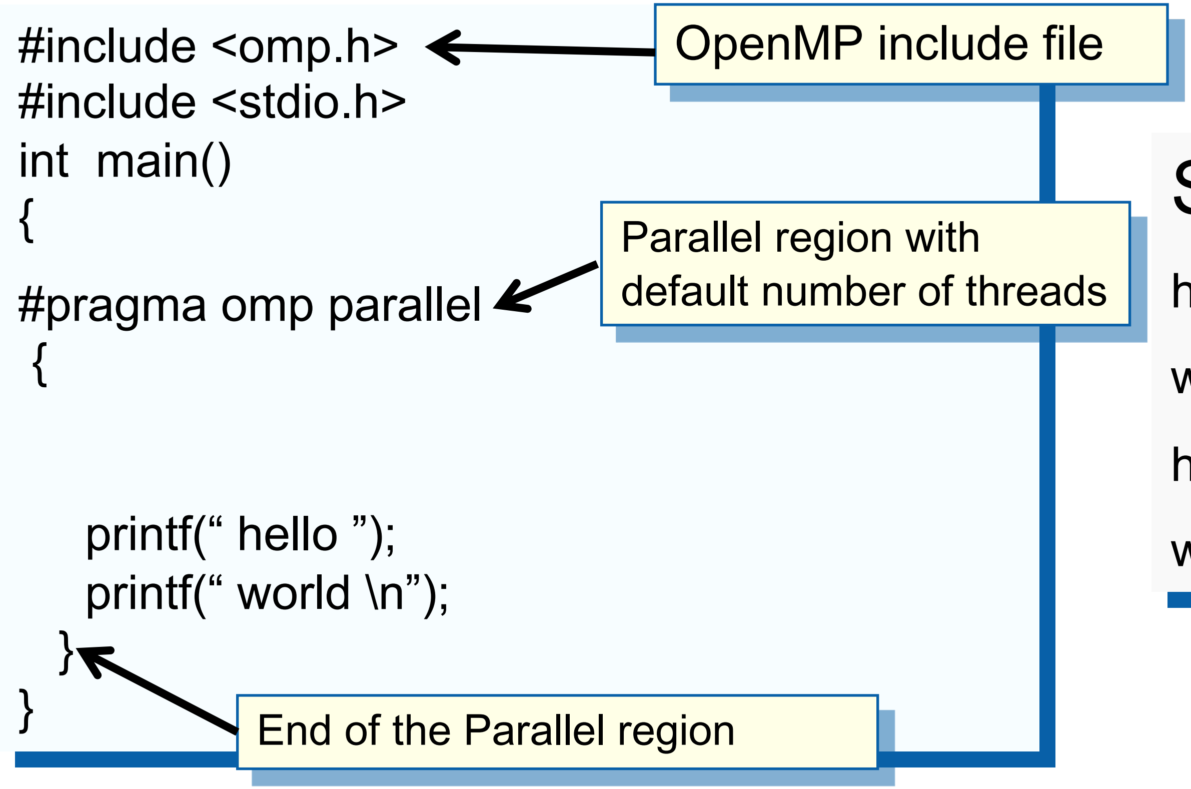
Switches for compiling and linking

gcc –fopenmp	**Gnu (Linux, OSX)**
CC -qopenmp	**On Theta**
icc –fopenmp	**Intel (Linux, OSX)**

Solution

A multi-threaded “Hello world” program

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



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


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

OpenMP include file

Parallel region with default number of threads

End of the Parallel region

Sample Output:

```
hello hello world
world
hello  hello world
world
```

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

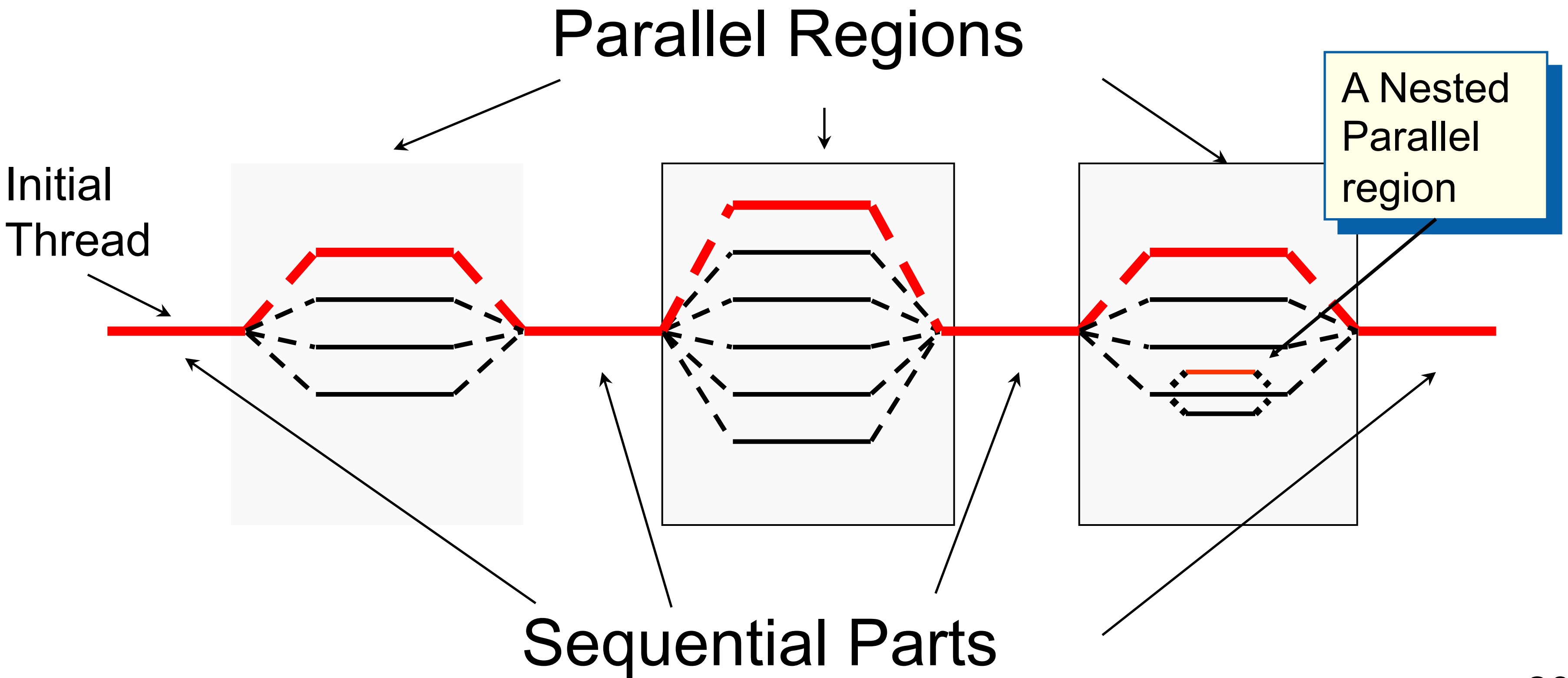
Outline

- Introduction to OpenMP
- Creating Threads
- Synchronization
- Parallel Loops
- Data environment
- Memory model
- Irregular Parallelism and tasks
- Recap
- Beyond the common core:
  - Worksharing Revisited
  - Synchronization Revisited: Options for Mutual exclusion
  - Thread Affinity and Data Locality
  - Thread Private Data
  - Memory models and point-to-point Synchronization
  - Programming your GPU with OpenMP

OpenMP programming model:

Fork-Join Parallelism:

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



Thread creation: Parallel regions

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

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

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

Runtime function to request a certain number of threads

Runtime function returning a thread ID

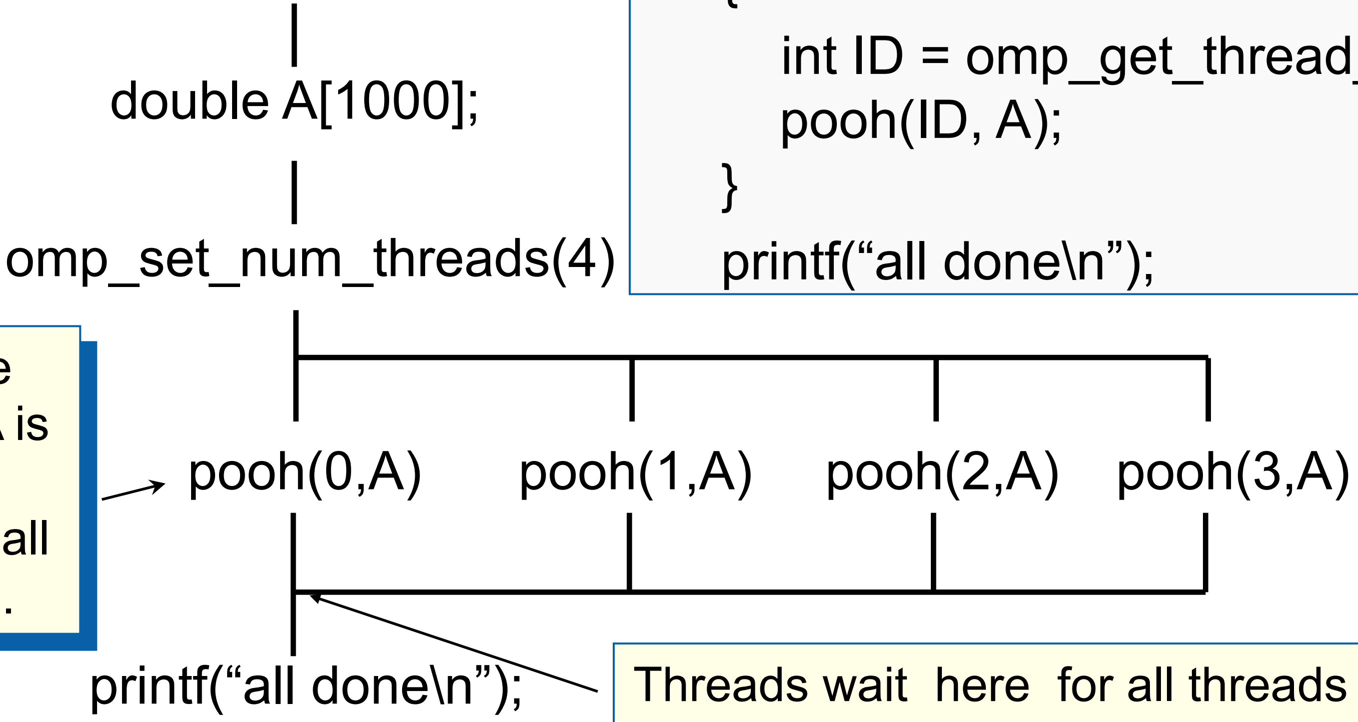
- Each thread calls pooh(ID,A) for `ID = 0 to 3`

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

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*)

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

Thread creation: How many threads did you actually get?

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

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

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

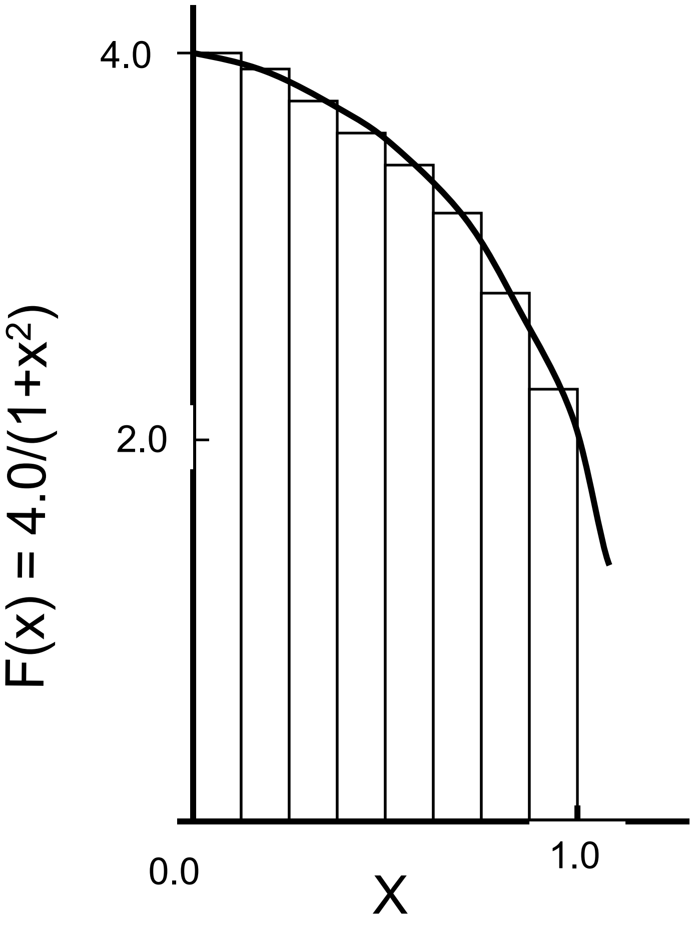
Runtime function to request a certain number of threads

Runtime function to return actual number of threads in the team

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

An interesting problem to play with

Numerical integration



Mathematically, we know that:

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

We can approximate the integral as a sum of rectangles:

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

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

Serial PI program

```
static long num_steps = 100000;
double step;
int main ()
{           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;
}
```

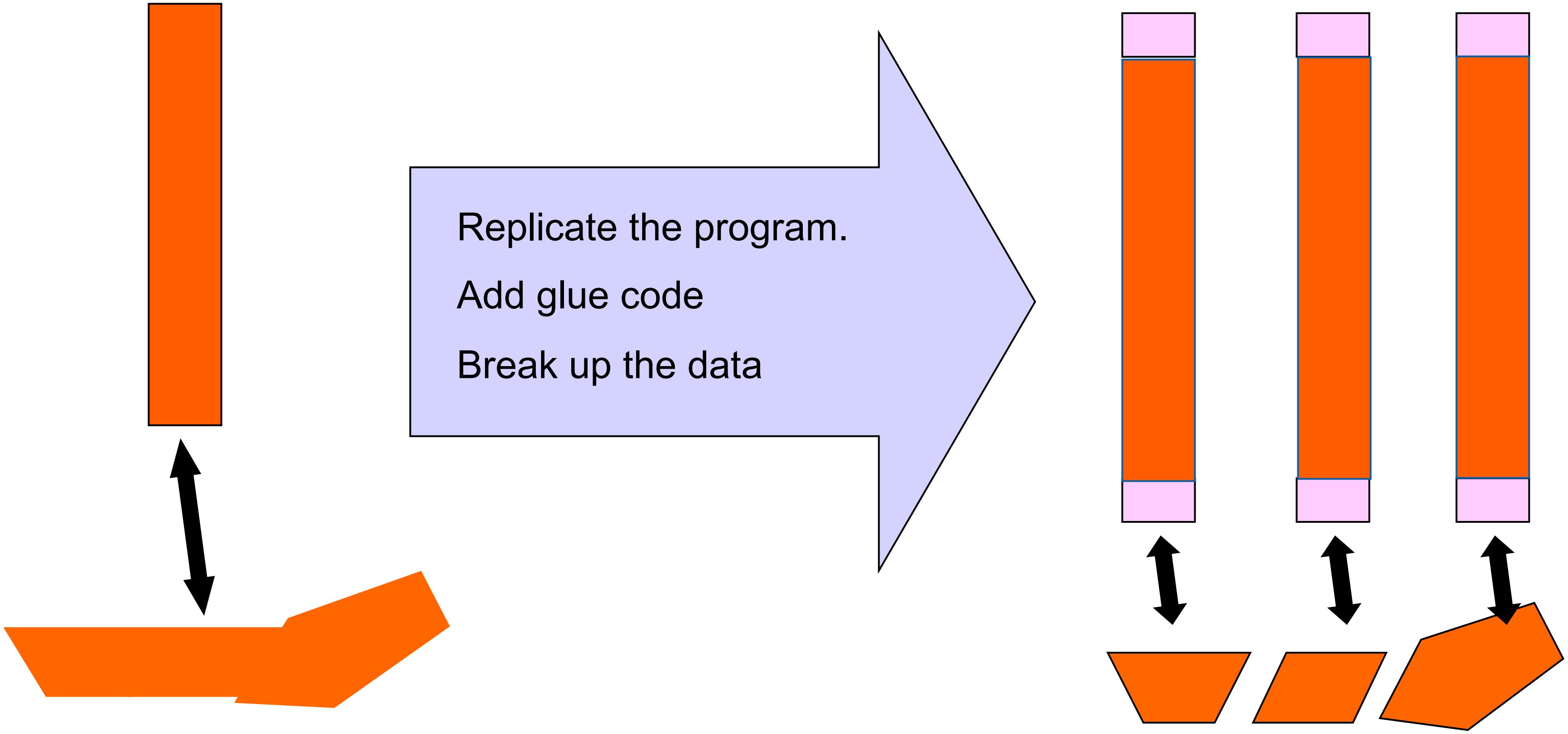
Promote scalar to an array dimensioned by number of threads to avoid race condition.

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.



Replicate the program.

Add glue code

Break up the data

- Use the rank … an ID ranging from 0 to (P-1) … to select between a set of tasks and to manage any shared data structures.

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

Pi program in MPI … using the SPMD pattern

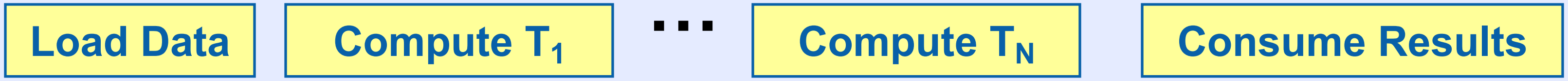
```
#include <mpi.h>
void main (int argc, char *argv[])
{
        int i, my_id, numprocs;  double x, pi, step, sum = 0.0 ;
        step = 1.0/(double) num_steps ;
        MPI_Init(&argc, &argv) ;
        MPI_Comm_Rank(MPI_COMM_WORLD, &my_id) ;
        MPI_Comm_Size(MPI_COMM_WORLD, &numprocs) ;
        my_steps = num_steps/numprocs ;
        int istart = my_id*my_steps;
        int iend = (my_id+1)*my_steps;
        if (my_id = numprocs-1) iend = num_steps;

        for (i=istarts; i<iend; 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) ;
       MPI_finalize();
}
```

Sum values in “sum” from each process and place it in “pi” on process 0

A brief digression to talk about performance issues in parallel programs

Consider performance of parallel programs

Compute N independent tasks on one processor



Load Data

Compute T_1

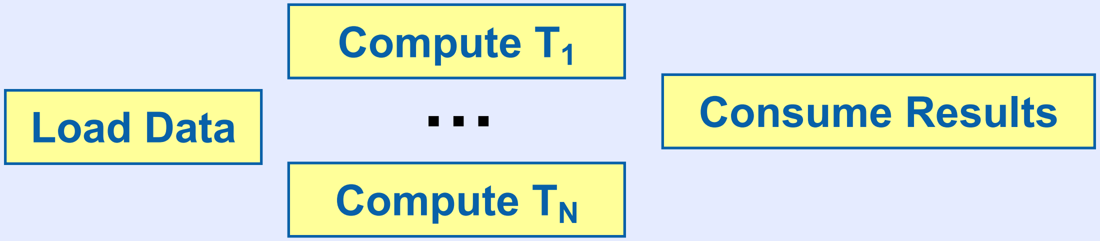
…

Compute T_N

Consume Results

$$\text{Time}_{\text{seq}}(1) = T_{\text{load}} + N{*}T_{\text{task}} + T_{\text{consume}}$$

Compute N independent tasks with P processors



Load Data

Compute T_1

…

Compute T_N

Consume Results

Ideally Cut runtime by ~1/P

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

$$\text{Time}_{\text{par}}(P) = T_{\text{load}} + (N/P){*}T_{\text{task}} + T_{\text{consume}}$$

Talking about performance

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

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

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

$$S(P) = P$$

- Super-linear Speedup: typically due to cache effects … i.e. as P grows, aggregate cache size grows so more of the problem fits in cache

$$S(P) > P$$

Amdahl's Law

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

$$Time_{par}(P) = (serial\_fraction + \frac{parallel\_fraction}{P}) * Time_{seq}$$

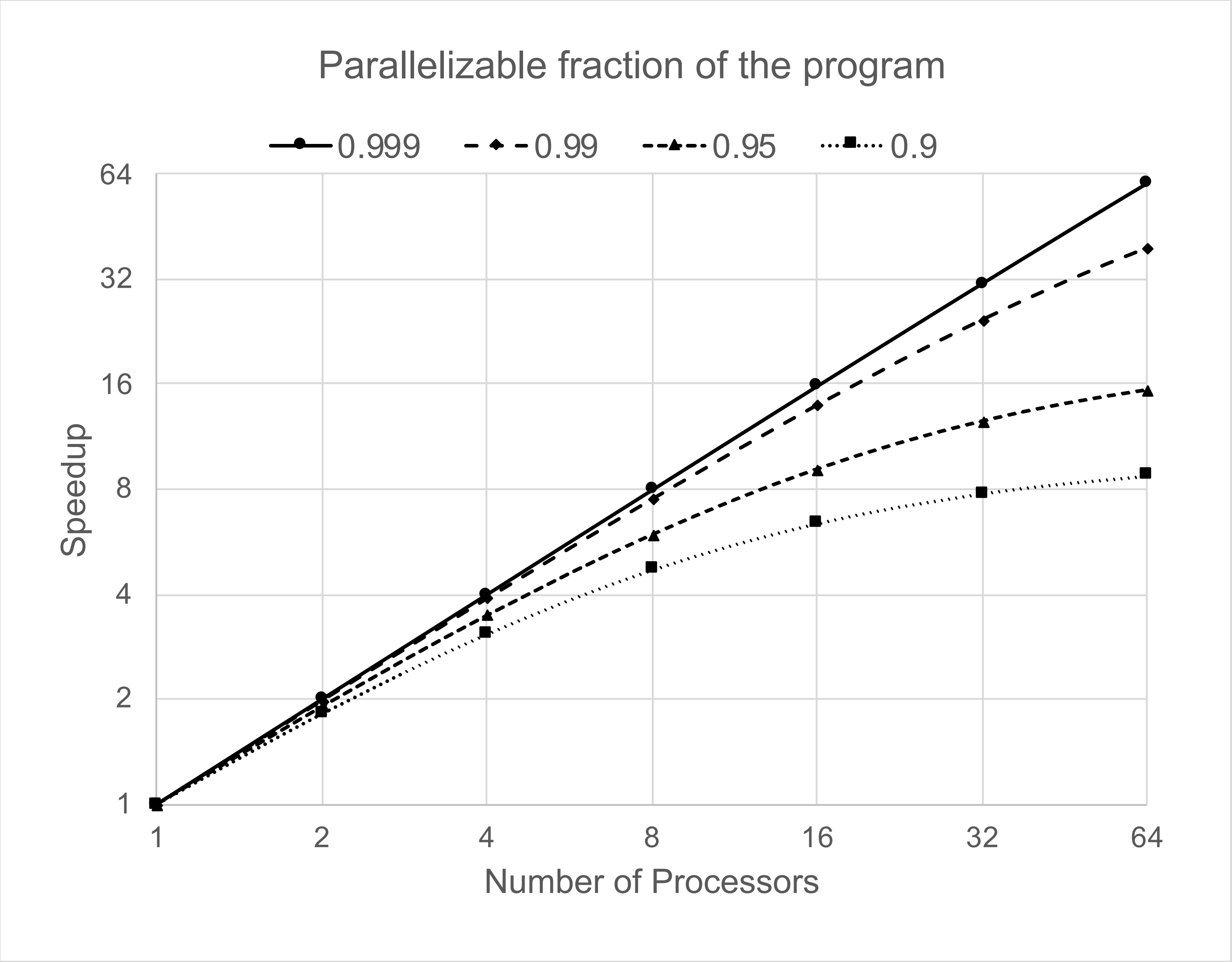
- If serial_fraction is α and parallel_fraction is (1- α) then the speedup is:

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

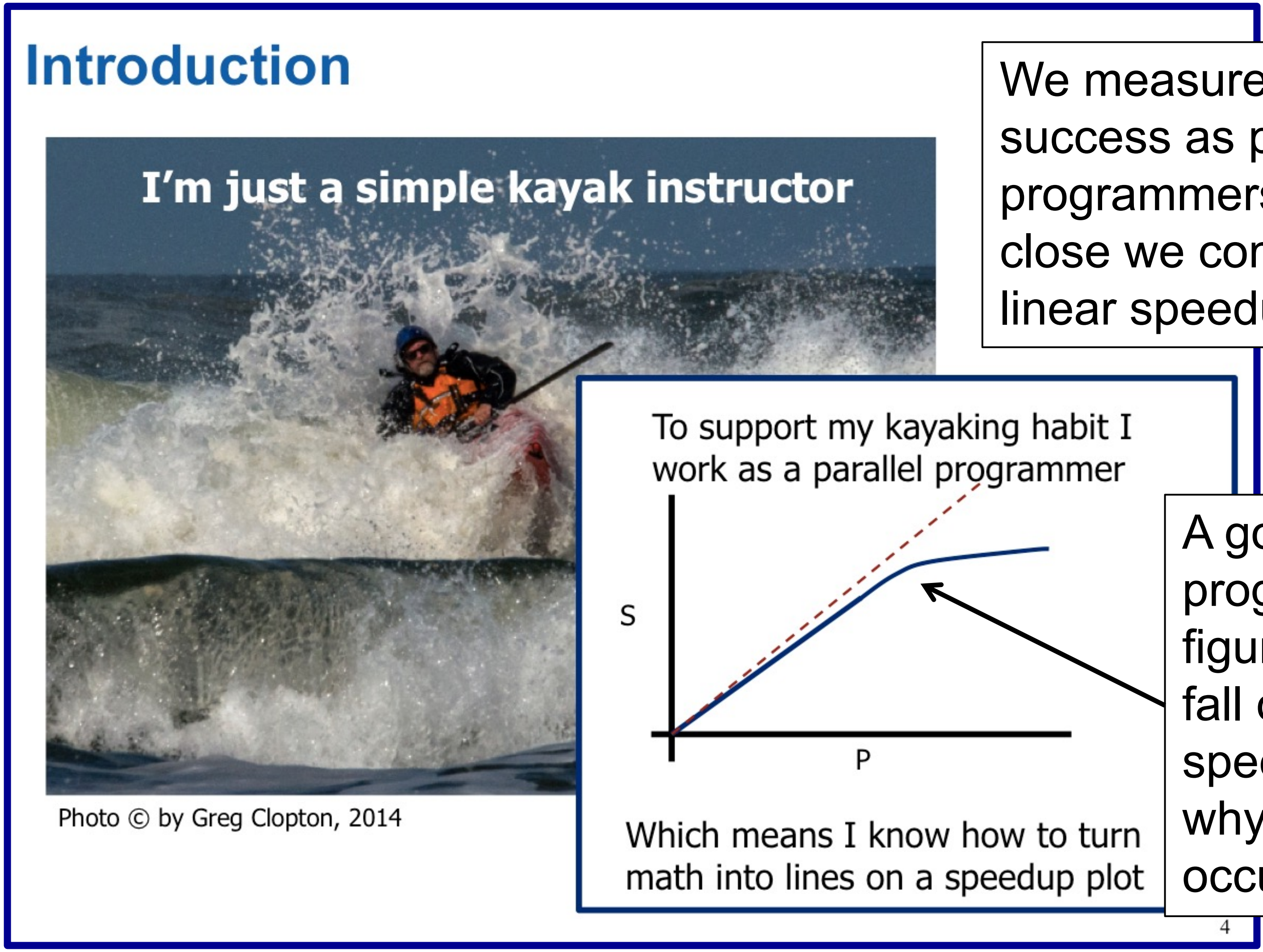
- If you had an unlimited number of processors: $P \rightarrow \infty$

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

Amdahl’s Law


Parallelizable fraction of the program
0.999
0.99
0.95
0.9
Speedup
64
32
16
8
4
2
1
1
2
4
8
16
32
64
Number of Processors

So now you should understand my silly introduction slide.



Introduction

I'm just a simple kayak instructor

Photo © by Greg Clopton, 2014

To support my kayaking habit I work as a parallel programmer

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

4

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

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

Now that you understand how to think about parallel performance, lets get back to OpenMP

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

- We've used the following construct to control the number of threads. (e.g. to request 12 threads):
  - omp_set_num_threads(12)

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

- Is there an easier way to change this internal control variable … perhaps one that doesn't require re-compilation? Yes.
  - When an OpenMP program starts up, it queries an environment variable OMP_NUM_THREADS and sets the appropriate internal control variable to the value of **OMP_NUM_THREADS**
  - For example, to set the initial, default number of threads to request in OpenMP from my apple laptop

```
> export OMP_NUM_THREADS=12
```

Exercise

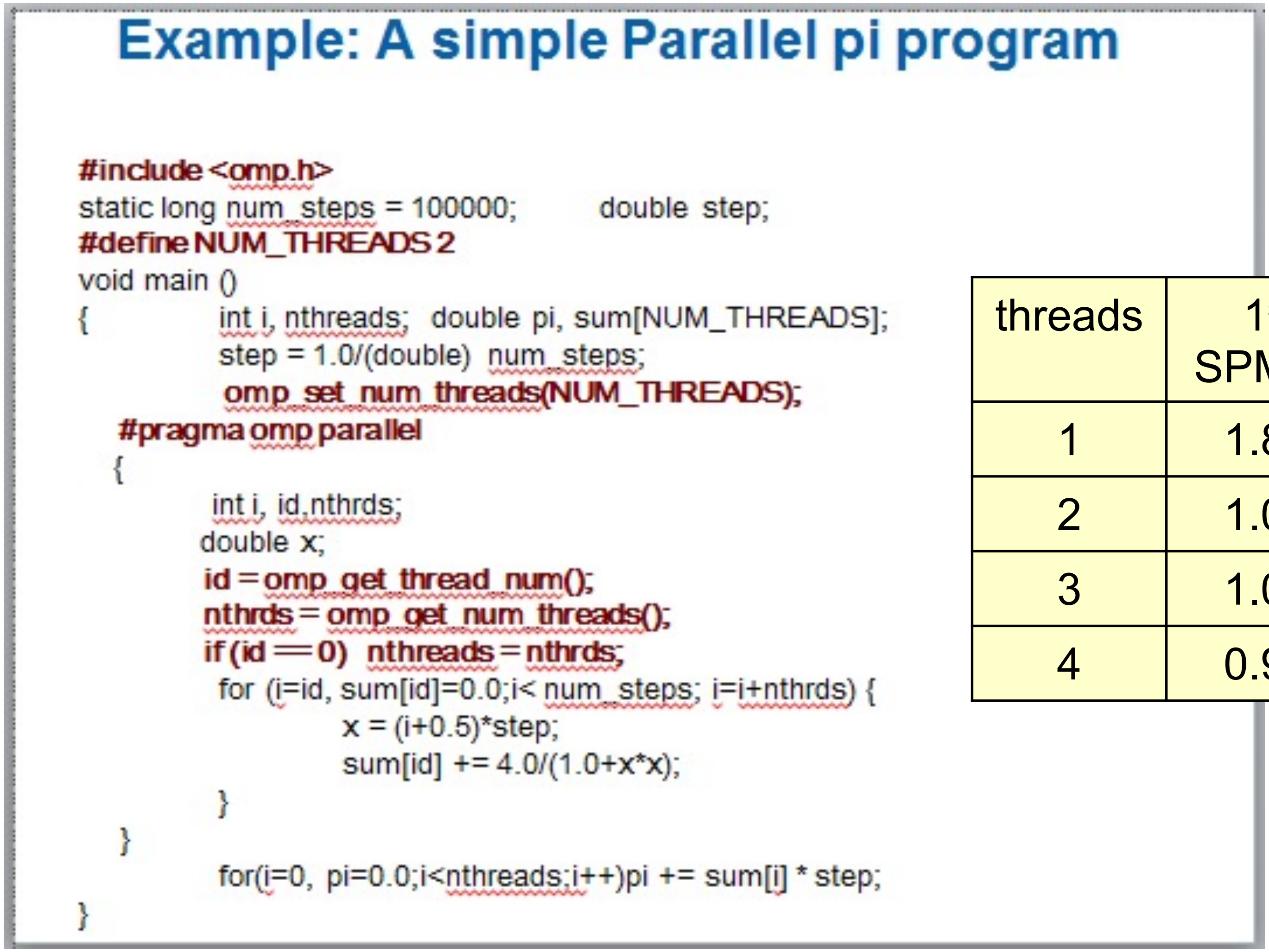
- Go back to your parallel pi program and explore how well it scales with the number of threads.
- Can you explain your performance with Amdahl's law? If not what else might be going on?

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

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

Results*

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



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;
}
```

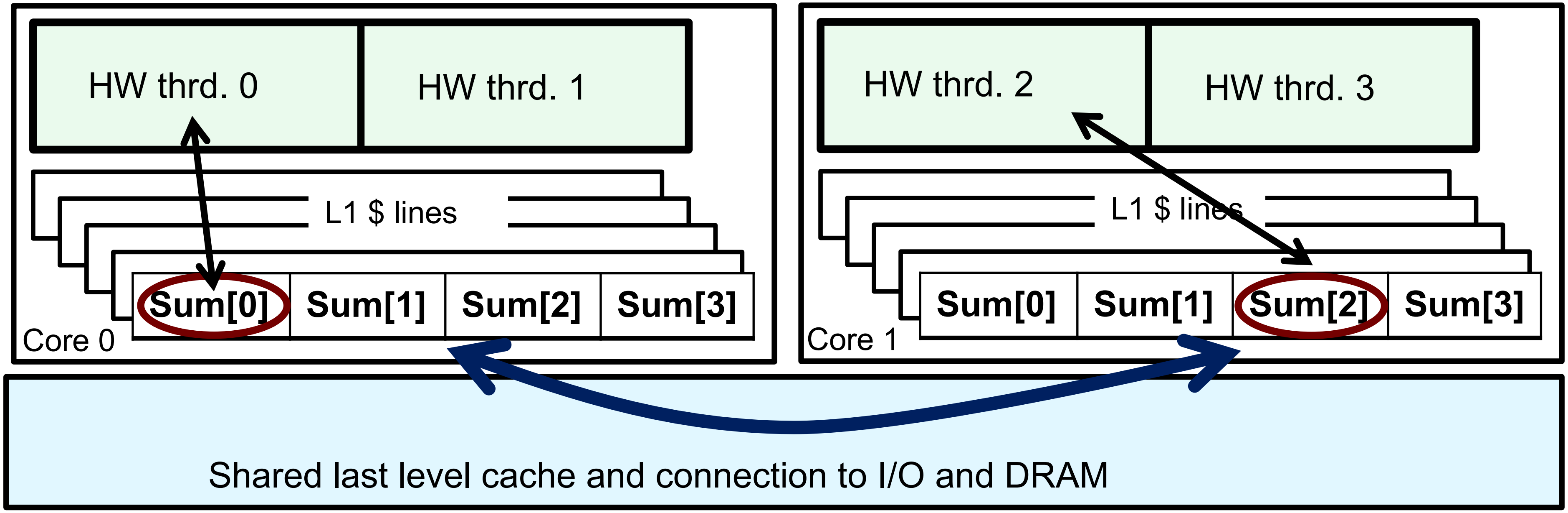
threads	1st SPMD*
1	1.86
2	1.03
3	1.08
4	0.97

*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^{TM} 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.

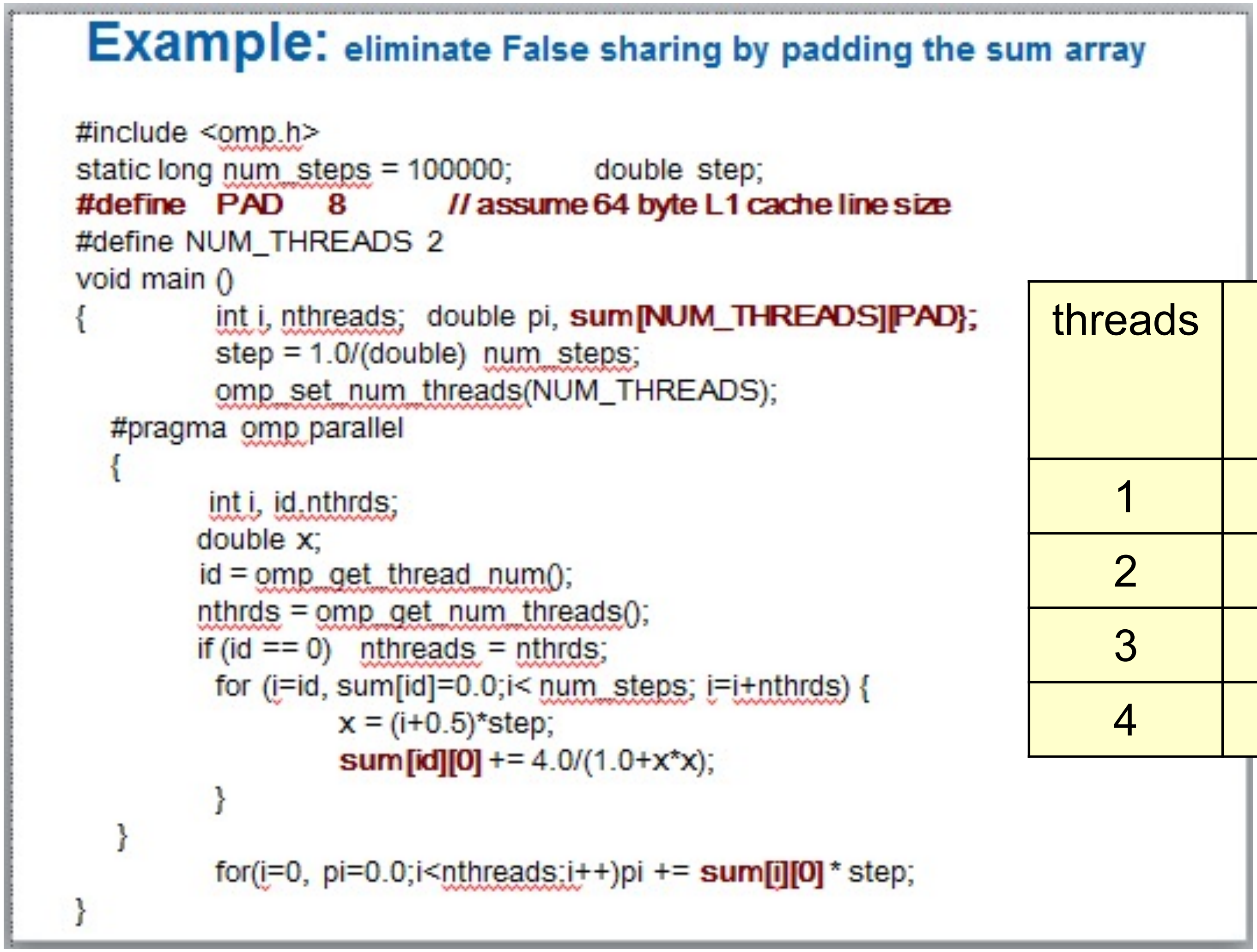
Example: Eliminate false sharing by padding the sum array

```
#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]=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[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 Revisited: Options for Mutual exclusion
  - Thread Affinity and Data Locality
  - Thread Private Data
  - Memory models and point-to-point Synchronization
  - Programming your GPU with OpenMP

Synchronization

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

- High level synchronization included in the common core:
  - critical
  - barrier

- Other, more advanced, synchronization operations:
  - atomic
  - ordered
  - flush
  - locks (both simple and nested)

Synchronization: critical

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

```
float  res;
#pragma omp parallel
{     float B;   int i, id, nthrds;
      id = omp_get_thread_num();
      nthrds = omp_get_num_threads();
      B =  big_SPMD_job(id, nthrds);
      #pragma omp critical
             res += consume (B);

}
```

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

Synchronization: barrier

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

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

omp_set_num_threads(8)

#pragma omp parallel

{   int id, nthrds;

     id = omp_get_thread_num();

     nthrds = omp_get_num_threads();

     if (id==0) numthrds = nthrds;

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

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

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

Exercise

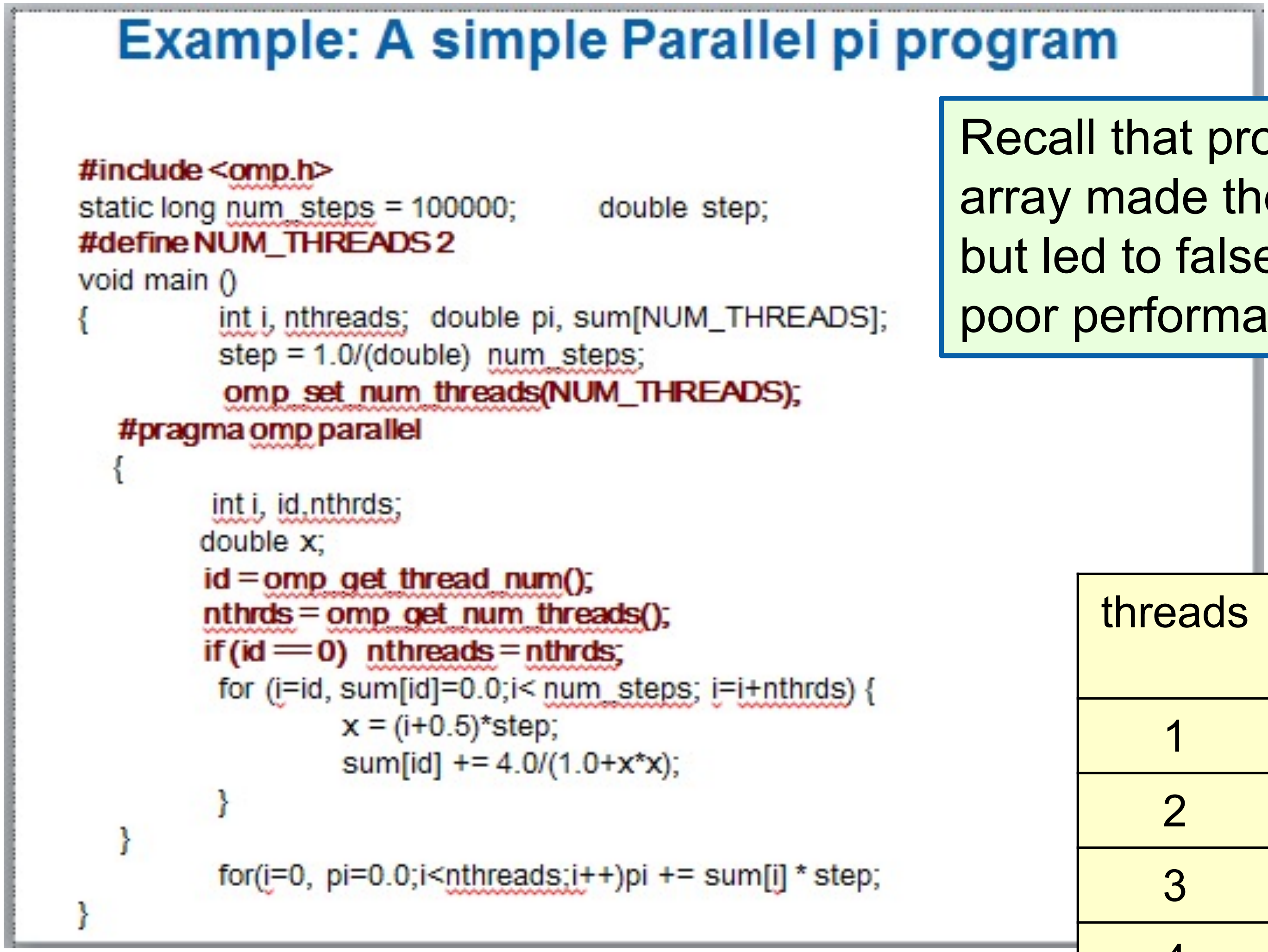
- In your first Pi program, you probably used an array to create space for each thread to store its partial sum.
- If array elements happen to share a cache line, this leads to false sharing.
  - Non-shared data in the same cache line so each update invalidates the cache line … in essence "sloshing independent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

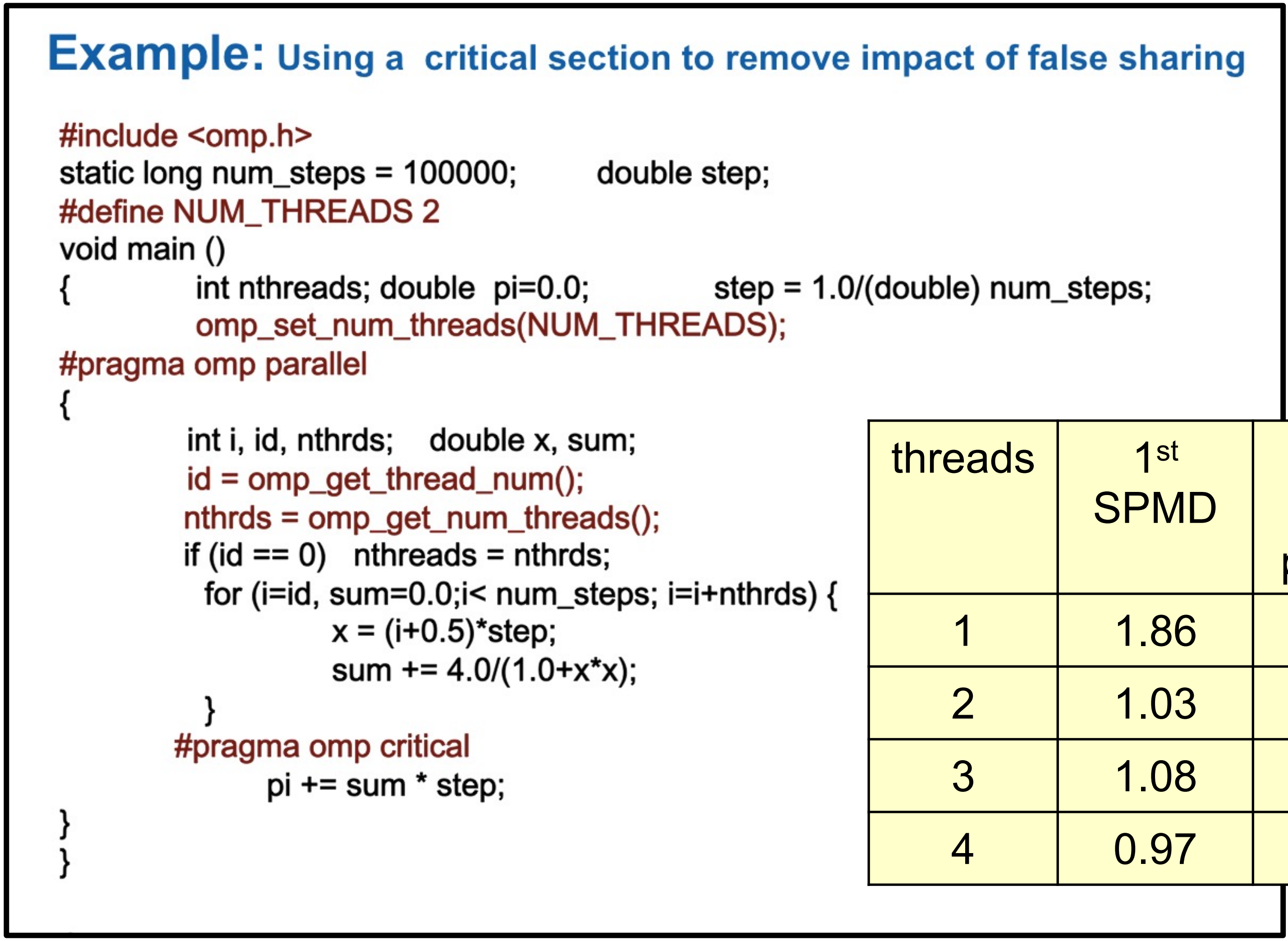
Create a scalar local to each thread to accumulate partial sums.

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^{TM} 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 Revisited: Options for Mutual exclusion
  - Thread Affinity and Data Locality
  - Thread Private Data
  - Memory models and point-to-point Synchronization
  - Programming your GPU with OpenMP

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

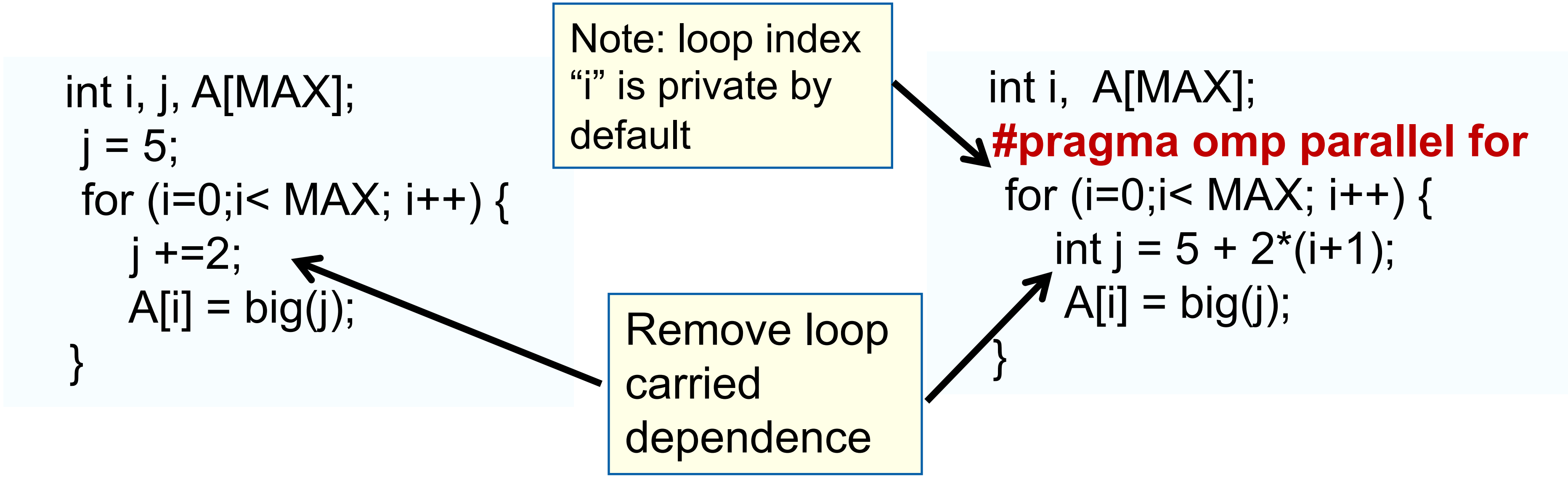
```
 double  res[MAX];  int i;
#pragma omp parallel
{
   #pragma omp for
   for (i=0;i< MAX; i++) {
       res[i] = huge();
   }
}
```

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

These are equivalent

Working with loops

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



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

Note: loop index "i" is private by default

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

Remove loop carried dependence

Reduction

- How do we handle this case?

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

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

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

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

Reduction

- OpenMP reduction clause:

  reduction (op : list)

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

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

OpenMP: Reduction operands/initial-values

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

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

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

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

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

Exercise: Pi with loops

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

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

Example: Pi with a loop and a reduction

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

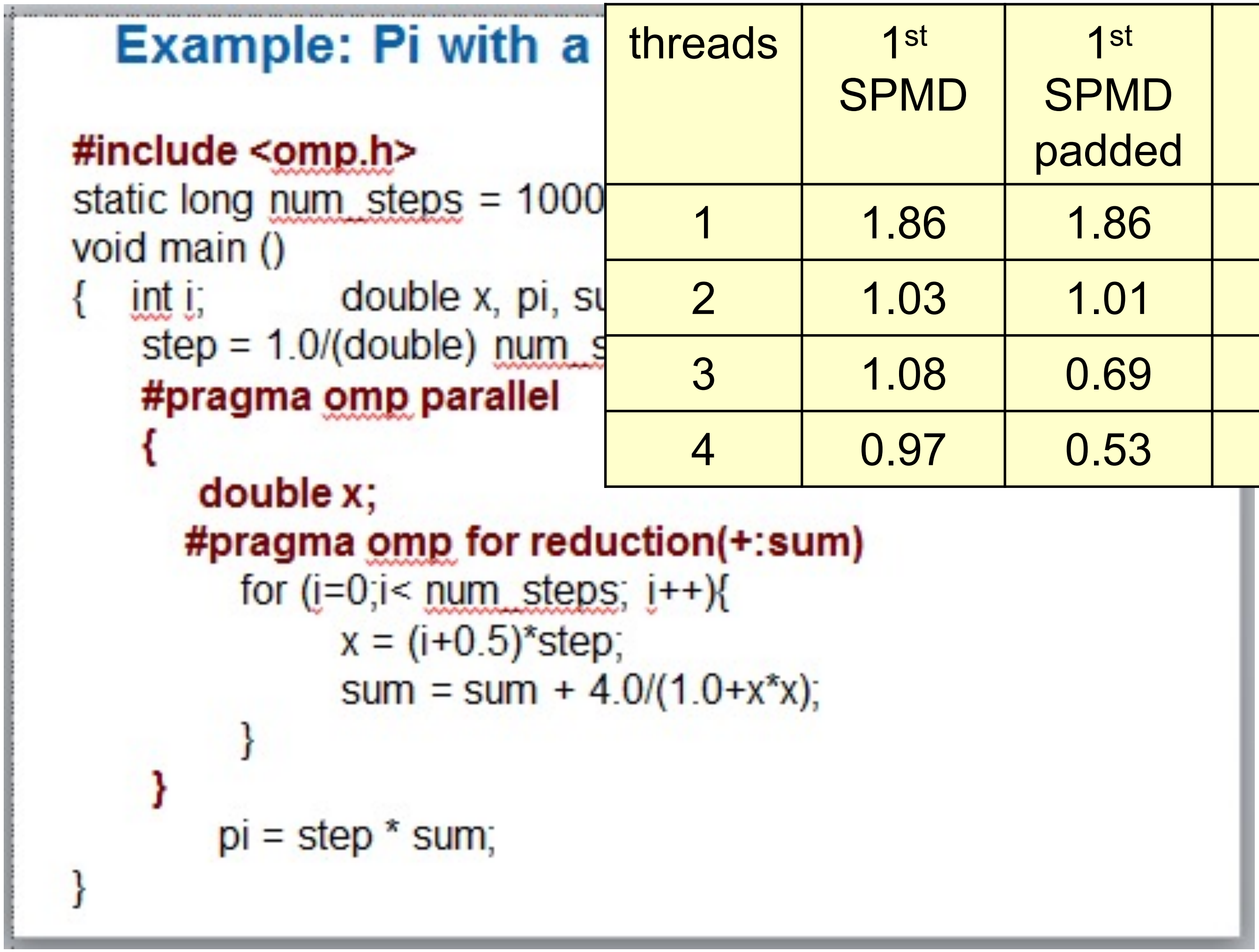
Example: Pi with a loop and a reduction

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

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

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

Results*: pi with a loop and a reduction

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



Example: Pi with a

```
#include <omp.h>
static long num_steps = 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^{TM} 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 Revisited: Options for Mutual exclusion
  - Thread Affinity and Data Locality
  - Thread Private Data
  - Memory models and point-to-point Synchronization
  - Programming your GPU with OpenMP

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.

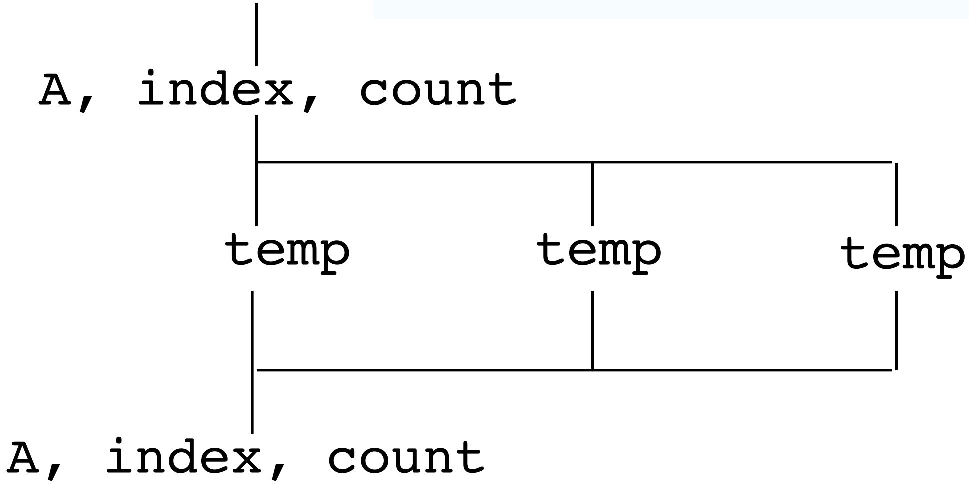
Data sharing: Examples

```
  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



A, index, count

temp temp temp

A, index, count

Data sharing:
Changing storage attributes

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

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

default() can only be used on parallel constructs

Data sharing: Private clause

- private(var) creates a new local copy of var for each thread.

```
int N = 1000;
extern void init_arrays(int N, double *A, double *B, double *C);



void example () {
    int i, j;
    double A[N][N], B[N][N], C[N][N];
    init_arrays(N, *A, *B, *C);


    #pragma omp parallel for private(j)
    for (i = 0; i < 1000; i++)
          for( j = 0; j<1000; j++)
                C[i][j] = A[i][j] + B[i][j];
}
```

OpenMP makes the loop control index on the parallel loop (i) private by default … but not for the second loop (j)

Data sharing: Private clause

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

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

tmp was not initialized

tmp is 0 here

When you need to refer to 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);
}
```

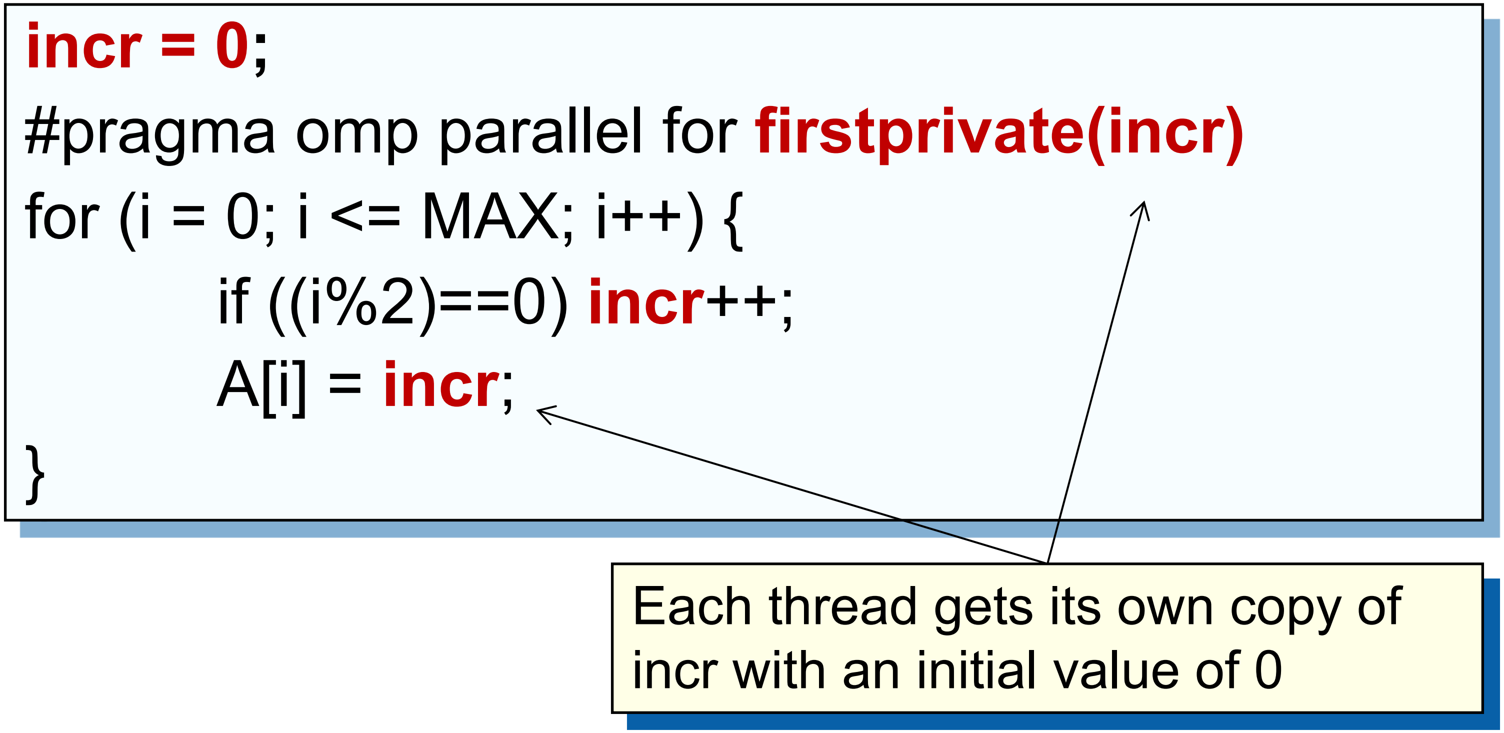
tmp has unspecified value

```
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.

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

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?

This exercises come from Mark Bull of EPCC (at University of Edinburgh)

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;
         }
       }
}
```

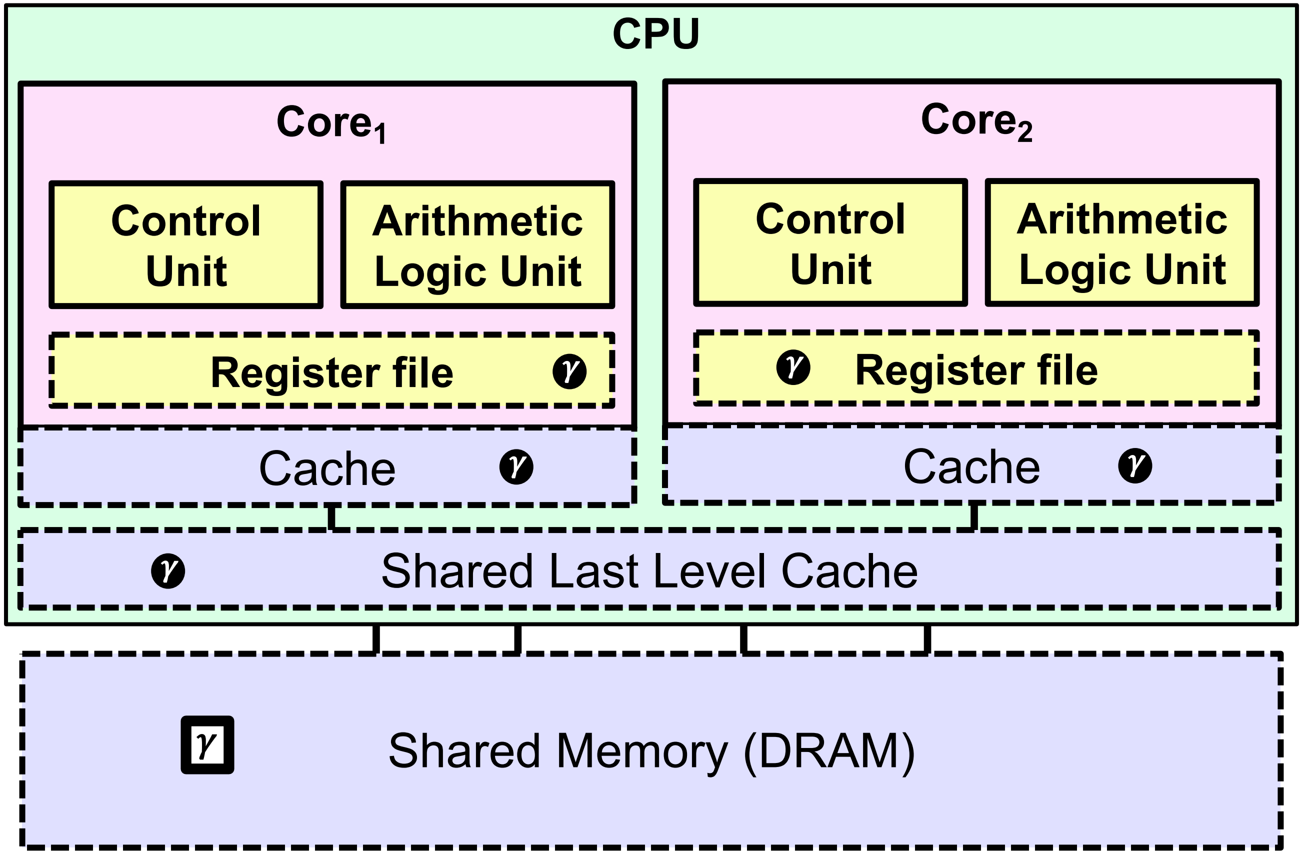
- eps was not initialized
- Protect updates of numoutside
- Which value of c does testpoint() see? Global or private?

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 Revisited: Options for Mutual exclusion
  - Thread Affinity and Data Locality
  - Thread Private Data
  - Memory models and point-to-point Synchronization
  - Programming your GPU with OpenMP

Memory models …

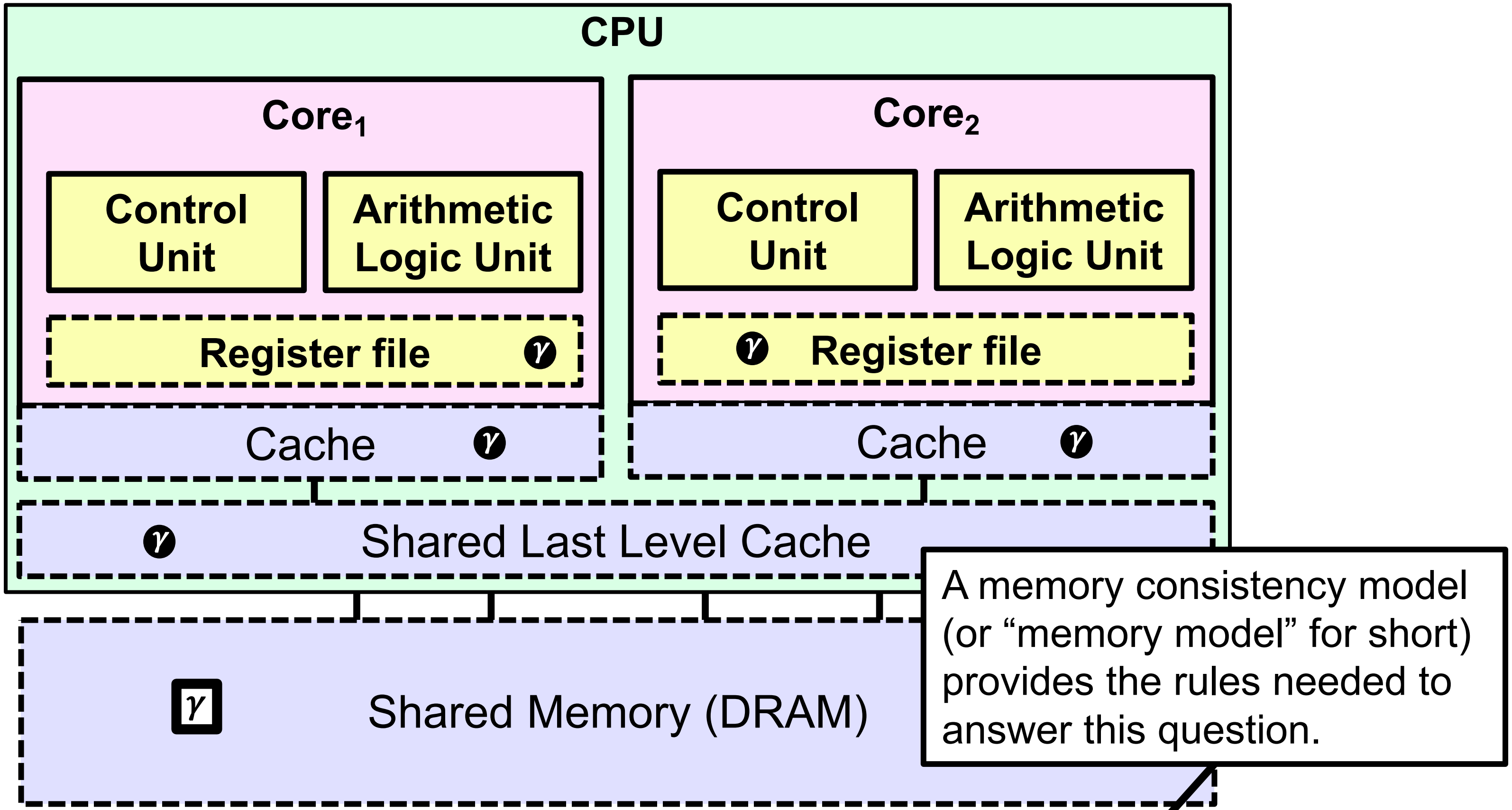
- Programming models for Multithreading support shared memory.
- All threads share an address space … but consider the variable γ



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

Memory models …

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



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

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

OpenMP and relaxed consistency

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

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

Flush operation

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

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

* This applies to the set of shared variables visible to a thread at the point the flush is encountered. We call this "**the flush set**"

flush example

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

```
double A;

A = compute();

#pragma omp flush(A)

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

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

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

What is the BIG DEAL with flush?

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

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

Flush and synchronization

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

  ….

  (but not on entry to worksharing regions)

WARNING:
If you find 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.

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 Revisited: Options for Mutual exclusion
  - Thread Affinity and Data Locality
  - Thread Private Data
  - Memory models and point-to-point Synchronization
  - Programming your GPU with OpenMP

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 (without tasks)

- 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

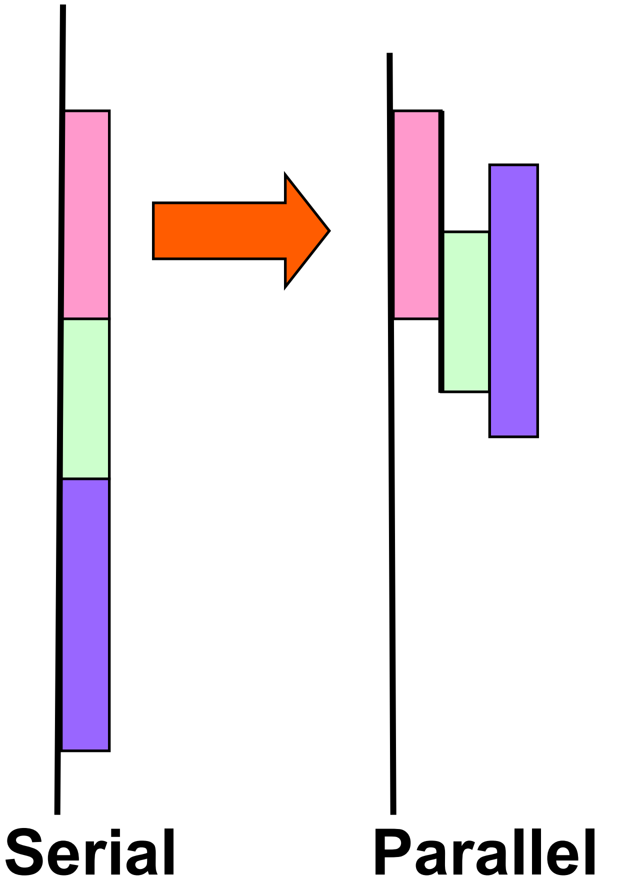
Process nodes in parallel with a for loop

	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

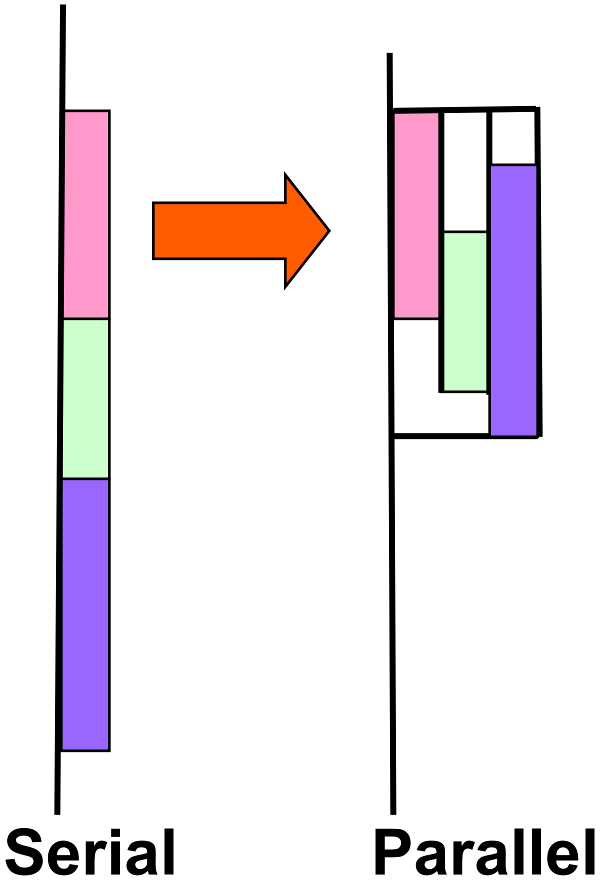
What are tasks?

- 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 primary* thread).
- A barrier is implied at the end of the single block (can remove the barrier with a *nowait* clause).

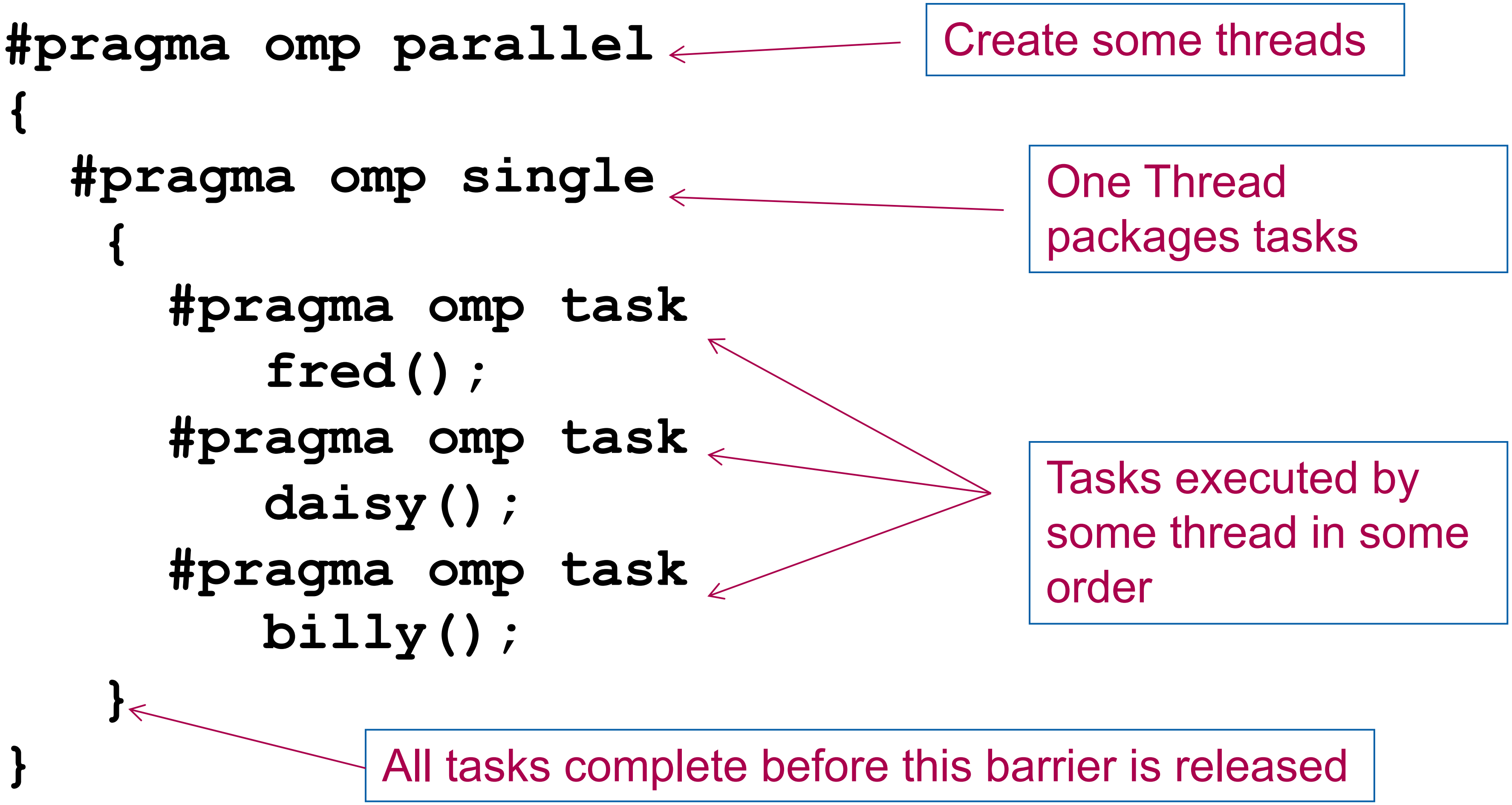
```
#pragma omp parallel
{
        do_many_things();
    #pragma omp single
        {    exchange_boundaries();   }
        do_many_other_things();

}
```

*the term “master” has been deprecated in OpenMP 5.1 and replaced with the term “primary”.

Task Directive

#pragma omp task *[clauses]*

structured-block



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

Create some threads

One Thread packages tasks

Tasks executed by some thread in some order

All tasks complete before this barrier is released

Exercise: Simple tasks

- Write a program using tasks that will “randomly” generate one of two strings:
  - “I think “ “race” “car” “s are fun”
  - “I think “ “car” “race” “s are fun”
- Hint: use tasks to print the indeterminate part of the output (i.e. the “race” or “car” parts).
- This is called a “Race Condition”. It occurs when the result of a program depends on how the OS schedules the threads.
- NOTE: A “data race” is when threads “race to update a shared variable”. They produce race conditions. Programs containing data races are undefined (in OpenMP but also ANSI standards C++’11 and beyond).

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

This exercise comes from Ruud van der Pas of Oracle

Racey cars: solution

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

Data scoping with tasks

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

- These concepts are a little bit different compared with threads:
  - If a variable is shared on a task construct, the references to it inside the construct are to the storage with that name at the point where the task was encountered

  - If a variable is private on a task construct, the references to it inside the construct are to new uninitialized storage that is created when the task is executed

  - If a variable is firstprivate on a construct, the references to it inside the construct are to new storage that is created and initialized with the value of the existing storage of that name when the task is encountered

Data scoping defaults

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

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

A is shared
B is firstprivate
C is private

Exercise: traversing linked lists

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

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

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

Parallel linked list traversal

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

Only one thread packages tasks

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

When/where are tasks complete?

- At thread barriers (explicit or implicit)
  - all tasks generated inside a region must complete at the next barrier encountered by the threads in that region. Common examples:
    - **Tasks generated inside a single construct**: all tasks complete before exiting the barrier on the single.
    - **Tasks generated inside a parallel region**: all tasks complete before exiting the barrier at the end of the parallel region.
- At taskwait directive
  - i.e. Wait until all tasks defined in the current task have completed.

```
#pragma omp taskwait
```

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

Example

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

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

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

Example

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

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

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

Example: Fibonacci numbers

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

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

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

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

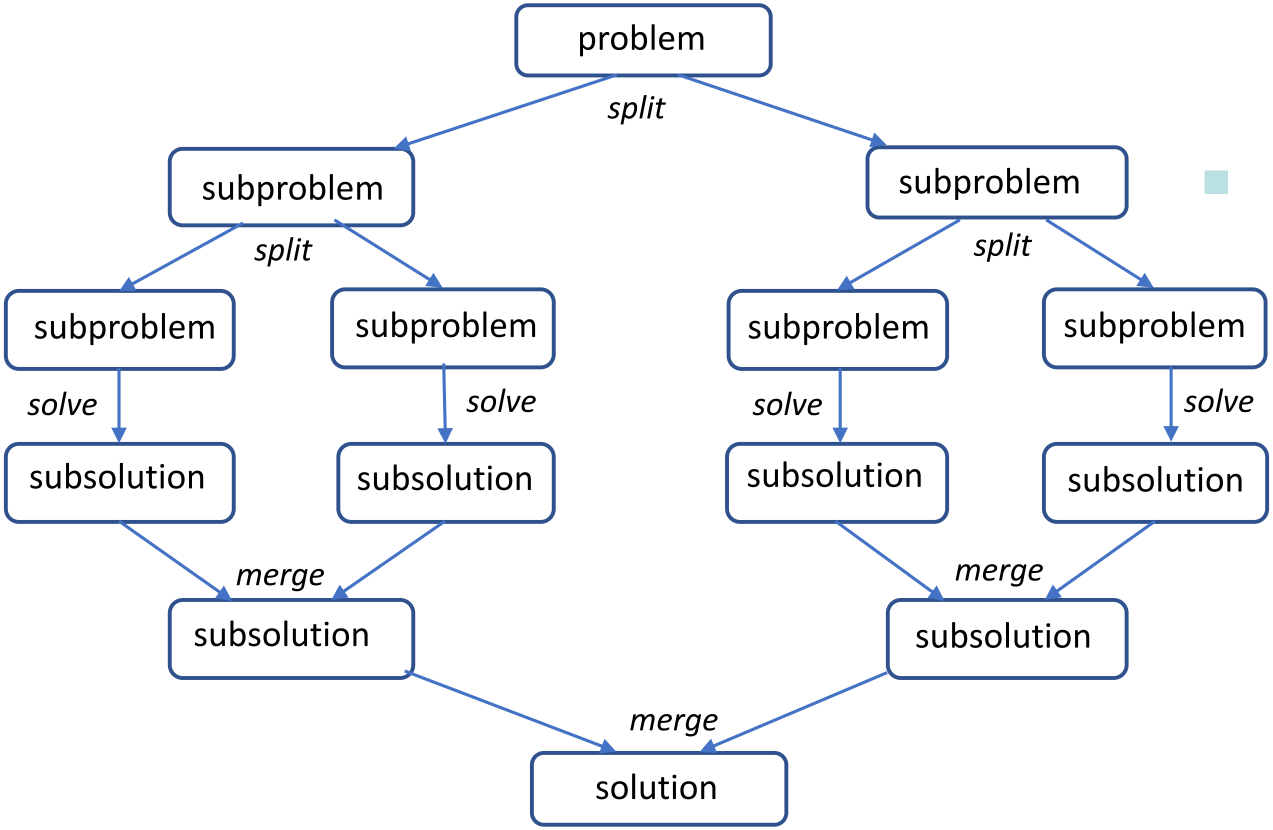
Parallel Fibonacci

```
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



problem

split

subproblem

subproblem

split

split

subproblem

subproblem

subproblem

subproblem

solve

solve

solve

solve

subsolution

subsolution

subsolution

subsolution

merge

merge

subsolution

subsolution

merge

solution

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

Exercise: Pi with tasks

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

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

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

Program: OpenMP tasks

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

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

Results*: pi with tasks

threads	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^{TM} 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 Revisited: Options for Mutual exclusion
  - Thread Affinity and Data Locality
  - Thread Private Data
  - Memory models and point-to-point Synchronization
  - Programming your GPU with OpenMP

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

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

There is much more to OpenMP than the Common Core.

- Synchronization mechanisms
  - locks, 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 5.0 specification is over 600 pages (not counting tools interface)!!!

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.

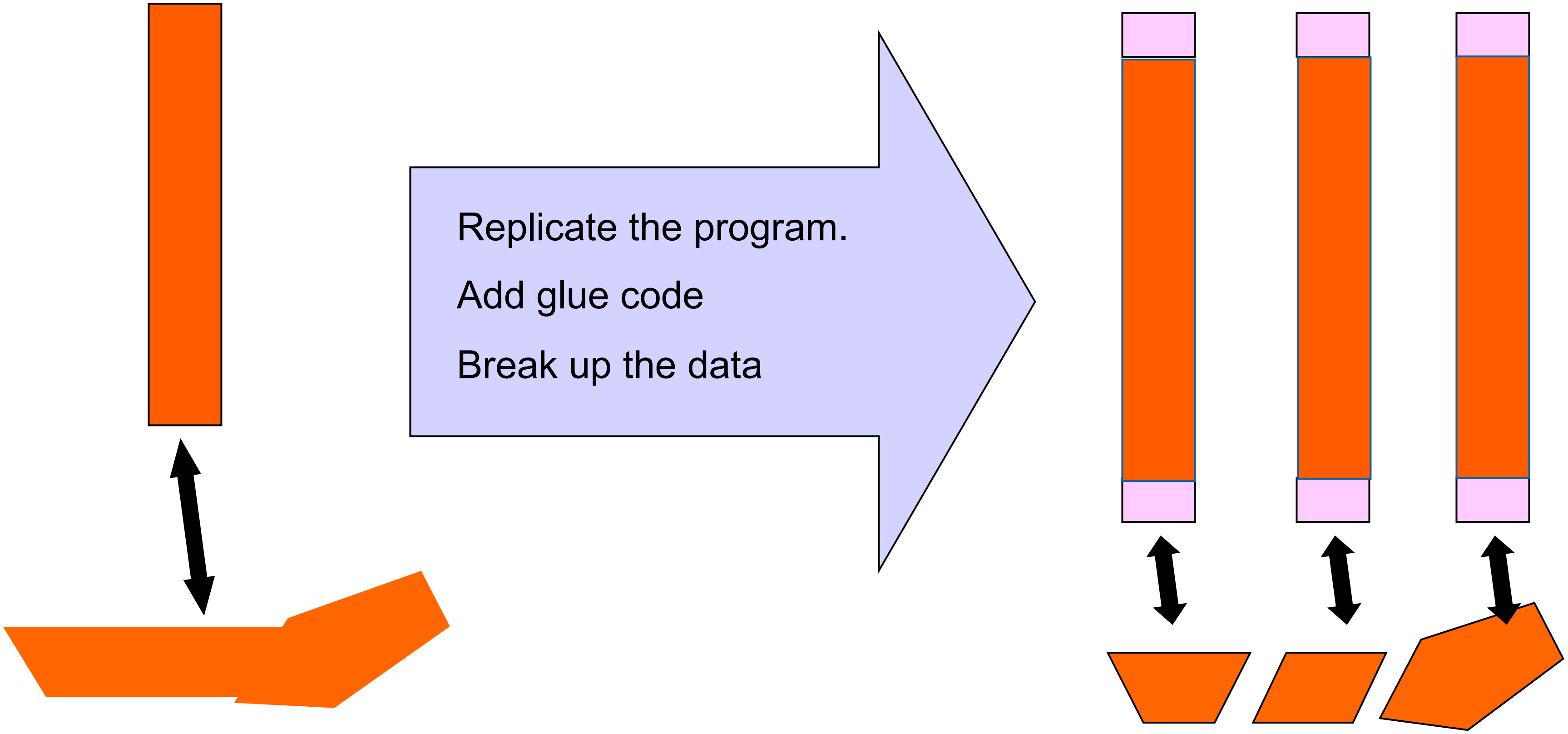
Fork-join

- 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.



Replicate the program.

Add glue code

Break up the data

- Use the rank … an ID ranging from 0 to (P-1) … to select between a set of tasks and to manage any shared data structures.

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

A 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

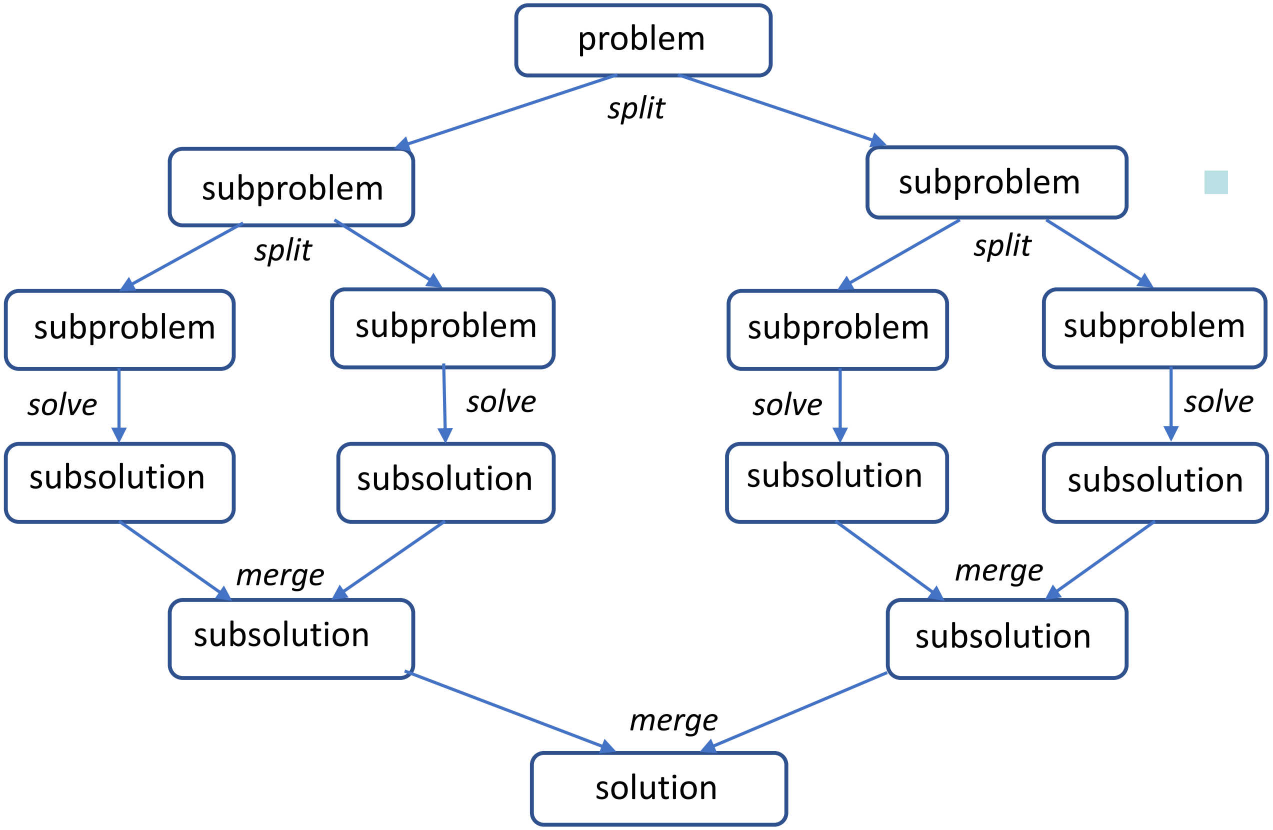
Note: we created a parallel program without changing any code and by adding 2 simple lines of text!

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

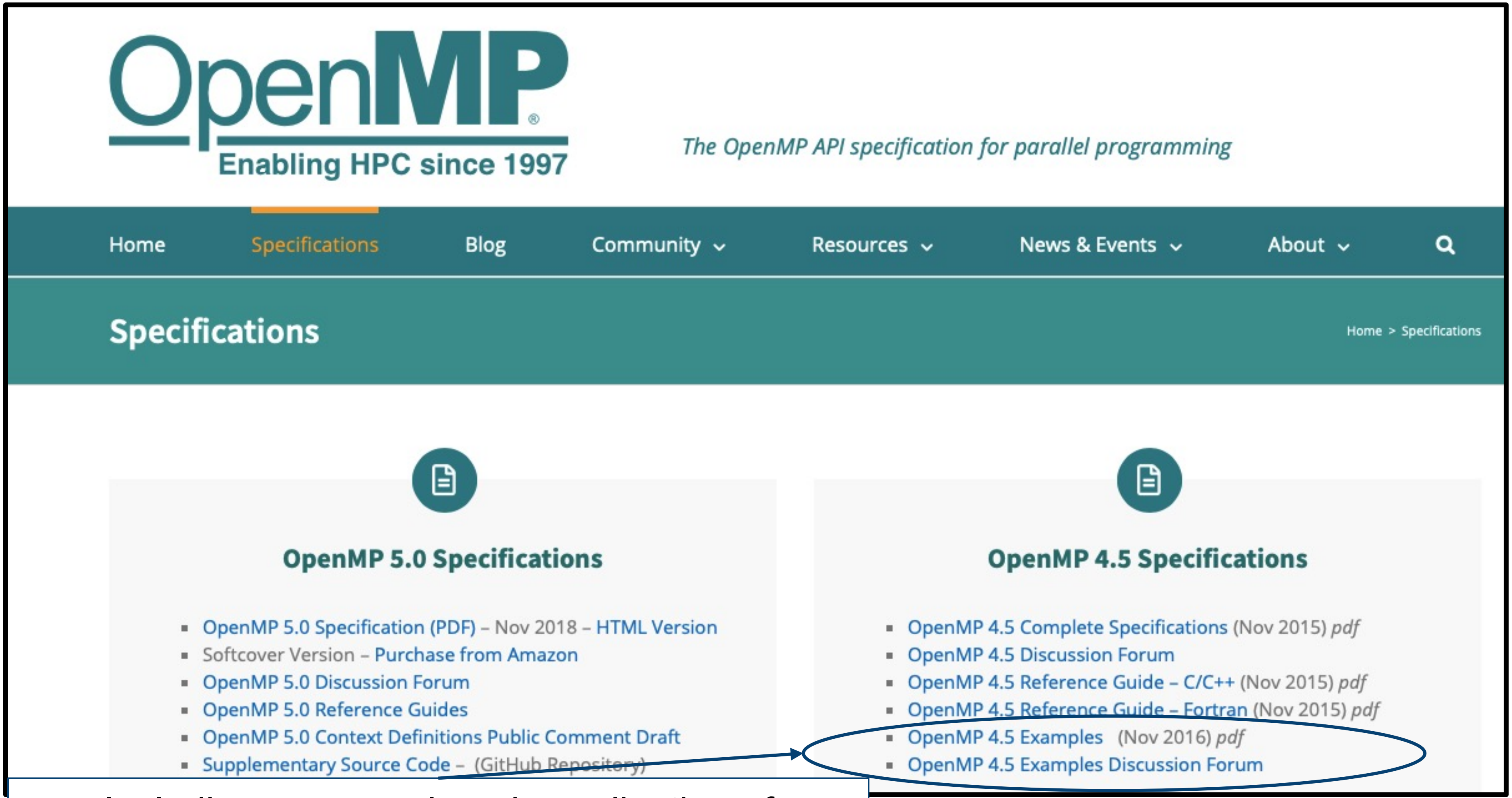
Program: OpenMP tasks

```
#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;
}
```

Resources

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



The OpenMP API specification for parallel programming

Specifications

OpenMP 5.0 Specifications

- OpenMP 5.0 Specification (PDF) – Nov 2018 – HTML Version
- Softcover Version – Purchase from Amazon
- OpenMP 5.0 Discussion Forum
- OpenMP 5.0 Reference Guides
- OpenMP 5.0 Context Definitions Public Comment Draft
- Supplementary Source Code – (GitHub Repository)

OpenMP 4.5 Specifications

- OpenMP 4.5 Complete Specifications (Nov 2015) *pdf*
- OpenMP 4.5 Discussion Forum
- OpenMP 4.5 Reference Guide – C/C++ (Nov 2015) *pdf*
- OpenMP 4.5 Reference Guide – Fortran (Nov 2015) *pdf*
- OpenMP 4.5 Examples (Nov 2016) *pdf*
- OpenMP 4.5 Examples Discussion Forum

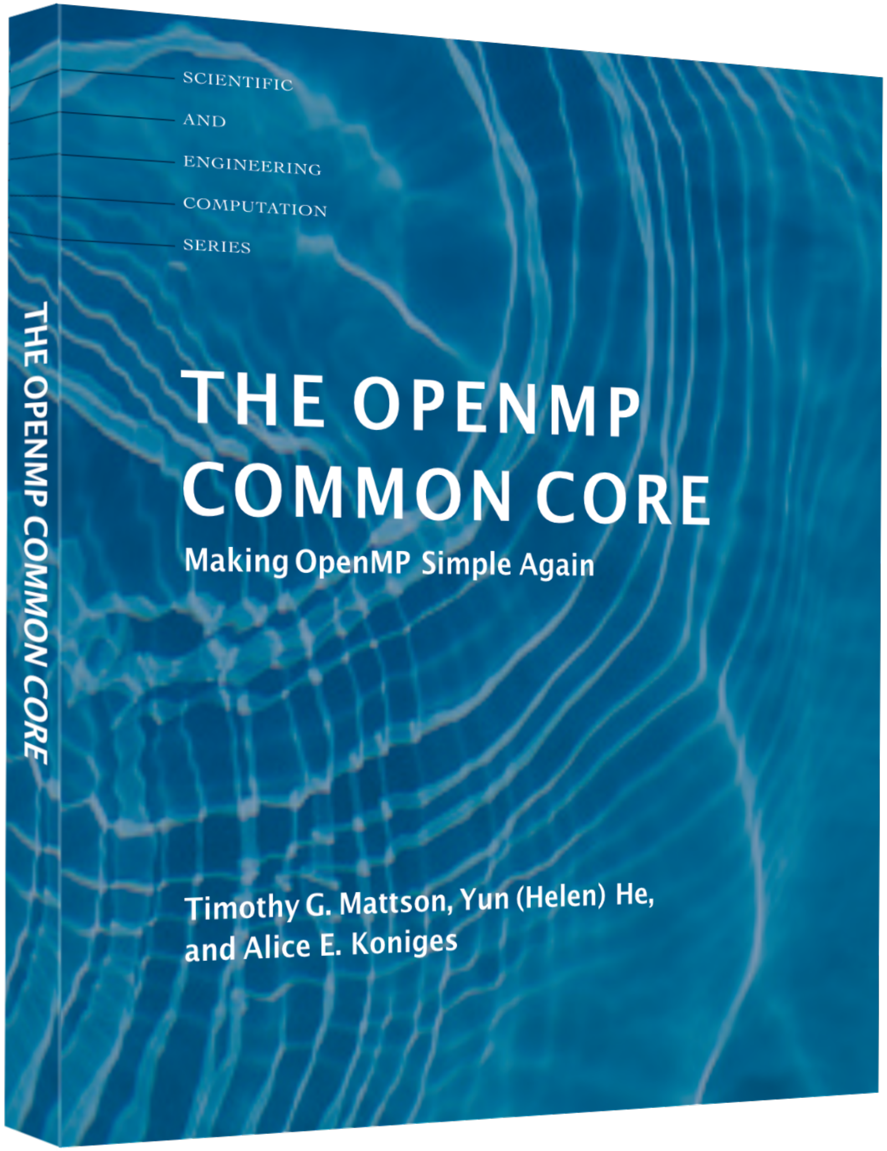
Including a comprehensive collection of examples of code using the OpenMP constructs

To learn OpenMP:

- An exciting new book that Covers the Common Core of OpenMP plus a few key features beyond the common core that people frequently use

- It's geared towards people learning OpenMP, but as one commentator put it … **everyone at any skill level should read the memory model chapters**.

- Available from MIT Press in November of 2019

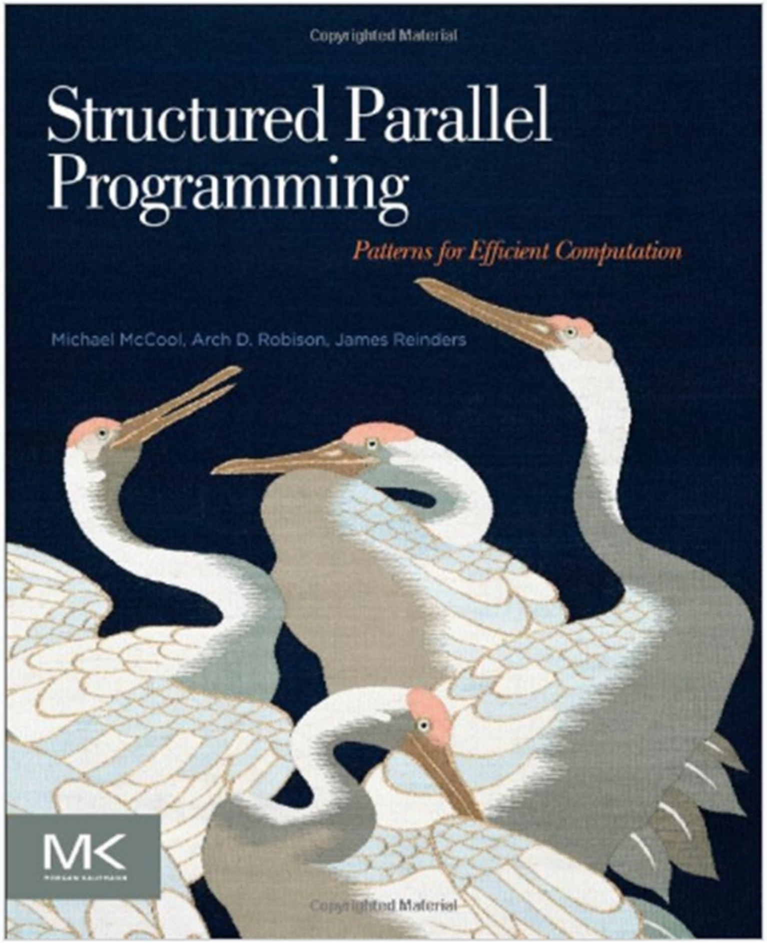


Books about OpenMP

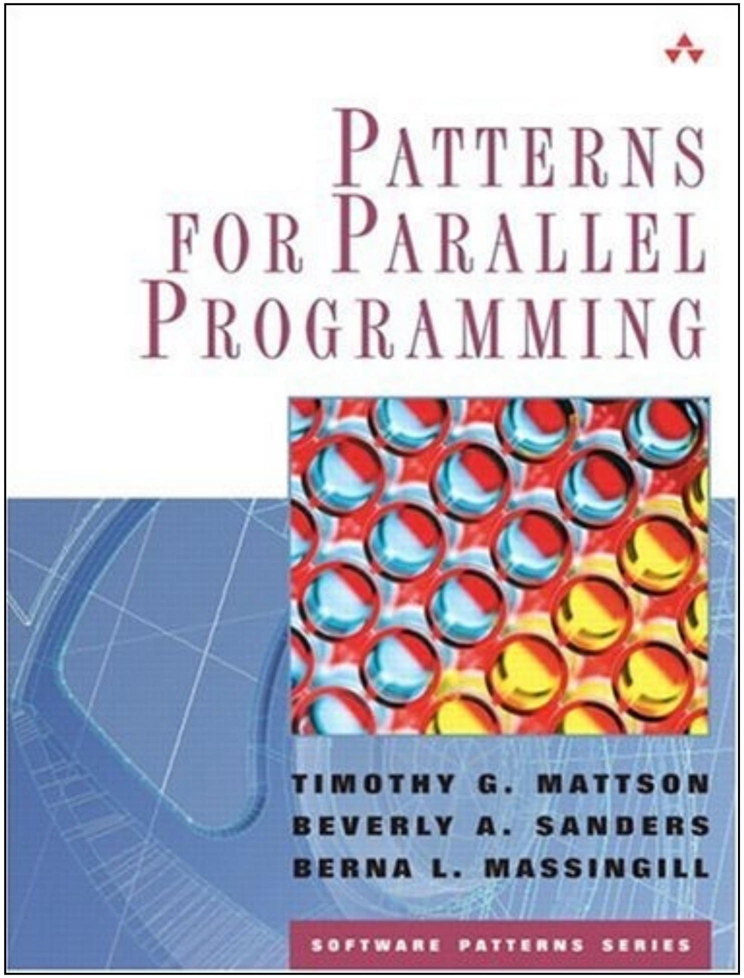
A great new book that covers OpenMP features beyond OpenMP 2.5

SCIENTIFIC
AND
ENGINEERING
COMPUTATION
SERIES
USING OPENMP – THE NEXT STEP
Affinity, Accelerators, Tasking, and SIMD
Ruud van der Pas, Eric Stotzer, and Christian Terboven
USING OPENMP – THE NEXT STEP

Background references


Copyrighted Material
Structured Parallel Programming
Patterns for Efficient Computation
Michael McCool, Arch D. Robison, James Reinders
MK
Copyrighted Material

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



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

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 Revisited: Options for Mutual exclusion
  - Thread Affinity and Data Locality
  - Thread Private Data
  - Memory models and point-to-point Synchronization
  - Programming your GPU with OpenMP

The Loop Worksharing Constructs

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

```
#pragma omp parallel

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

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

Loop construct name:

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

Loop Worksharing Constructs:
The Schedule Clause

- The schedule clause affects how loop iterations are mapped onto threads
  - **schedule(static [,chunk])**
    - Deal-out blocks of iterations of size "chunk" to each thread.
  - **schedule(dynamic[,chunk])**
    - Each thread grabs "chunk" iterations off a queue until all iterations have been handled.
  - **schedule(guided[,chunk])**
    - Threads dynamically grab blocks of iterations. The size of the block starts large and shrinks down to size "chunk" as the calculation proceeds.
  - **schedule(runtime)**
    - Schedule and chunk size taken from the OMP_SCHEDULE environment variable (or the runtime library) … vary schedule without a recompile!
  - **Schedule(auto)**
    - Schedule is left up to the runtime to choose (does not have to be any of the above).

OpenMP 4.5 added modifiers monotonic, nonmontonic and simd.

Loop Worksharing 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 any other reduce … a private array is formed for each thread, element wise combination across threads and then with original array at the end

Exercise

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

Optimizing mandel.c

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

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

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

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

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 Revisited: Options for Mutual exclusion
  - Thread Affinity and Data Locality
  - Thread Private Data
  - Memory models and point-to-point Synchronization
  - Programming your GPU with OpenMP

Synchronization

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

- High level synchronization included in the common core:
  - critical
  - barrier

Covered earlier

- Other, more advanced, synchronization operations:
  - atomic
  - ordered
  - flush
  - locks (both simple and nested)

Covered in this section

Synchronization: Atomic

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

```
#pragma omp parallel
{
        double B;
        B =  DOIT();


 #pragma omp atomic
             X +=  big_ugly(B);
}
```

Synchronization: Atomic

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

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

Atomic only protects the read/update of X

The OpenMP 3.1 Atomics (1 of 2)

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

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

- Atomic can protect loads

```
# pragma omp atomic read
      v = x;
```

- Atomic can protect stores

```
# pragma omp atomic write
      x = expr;
```

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

```
# pragma omp atomic update
      x++;  or ++x;  or x--;  or –x;  or
      x binop= expr; or x = x binop expr;
```

This is the original OpenMP atomic

The OpenMP 3.1 Atomics (2 of 2)

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

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

- Where the statement is one of the following forms:

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

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

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

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

Synchronization: Lock Routines

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

- Simple Lock routines:
  - A simple lock is available if it is unset.
    - omp_init_lock(), omp_set_lock(), omp_unset_lock(), omp_test_lock(), omp_destroy_lock()

- Nested Locks
  - A nested lock is available if it is unset or if it is set but owned by the thread executing the nested lock function
    - **omp_init_nest_lock(), omp_set_nest_lock(), omp_unset_nest_lock(), omp_test_nest_lock(), omp_destroy_nest_lock()**

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

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

Synchronization: Simple Locks Example

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

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

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

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

One lock per case … even and odd

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

Free-up storage when done.

Exercise

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

Histogram program: critical section

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

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

Easy to write and correct, but terrible performance

Histogram program: one lock per histogram bin

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

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

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

One lock per element of hist

Enforce mutual exclusion on update to hist array

Free-up storage when done.

Histogram program: reduction with an array

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

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

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

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

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

Sometimes when working with multiple interacting locks, you have to pay attention to the locking orders

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.

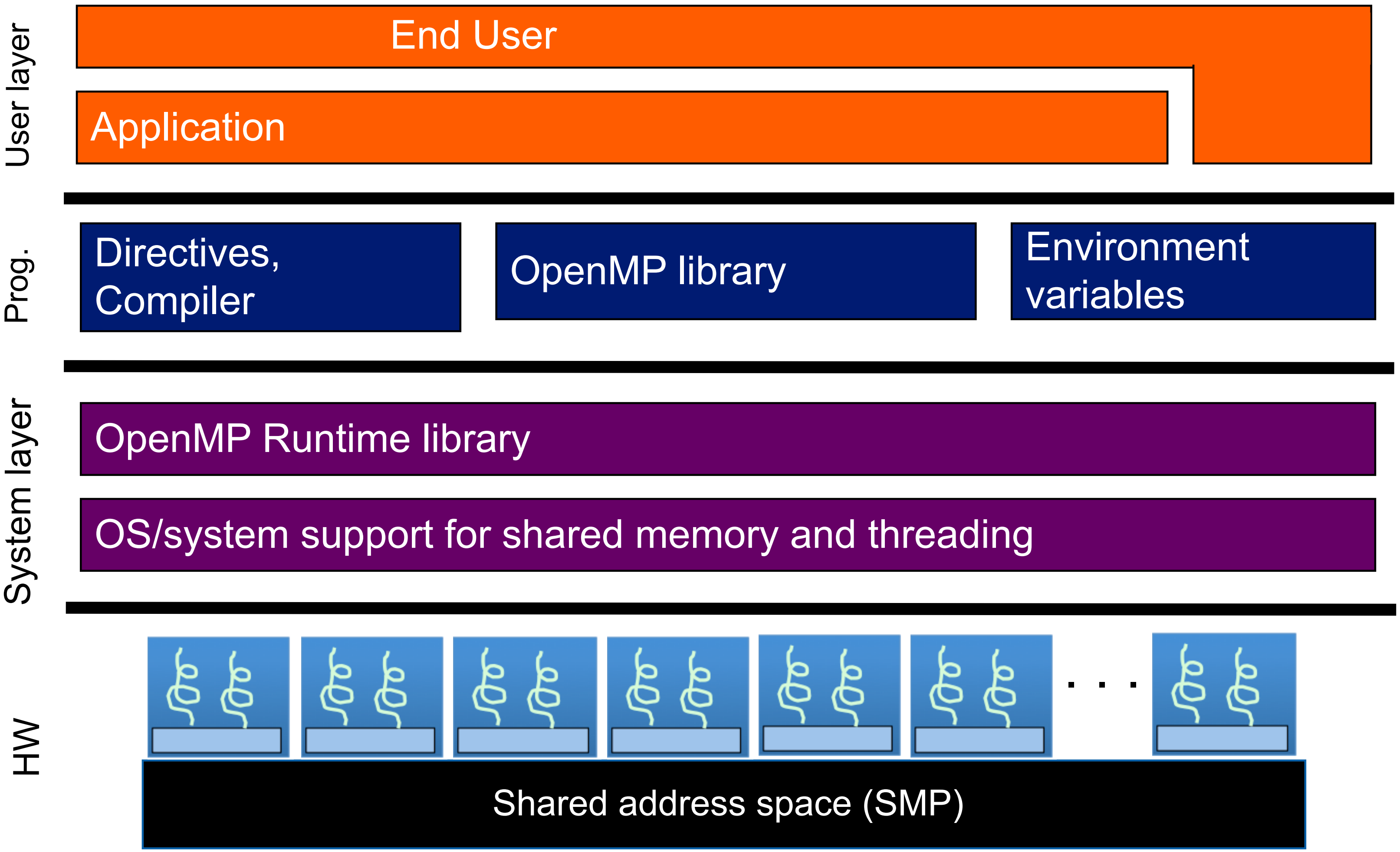
Ordered locking prevents deadlock.

```
!$OMP PARALLEL PRIVATE(rand, iother, itemp, temp, my_cpu_id)
      my_cpu_id = 1
!$    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
```

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 Revisited: Options for Mutual exclusion
  - Thread Affinity and Data Locality
  - Thread Private Data
  - Memory models and point-to-point Synchronization
  - Programming your GPU with OpenMP

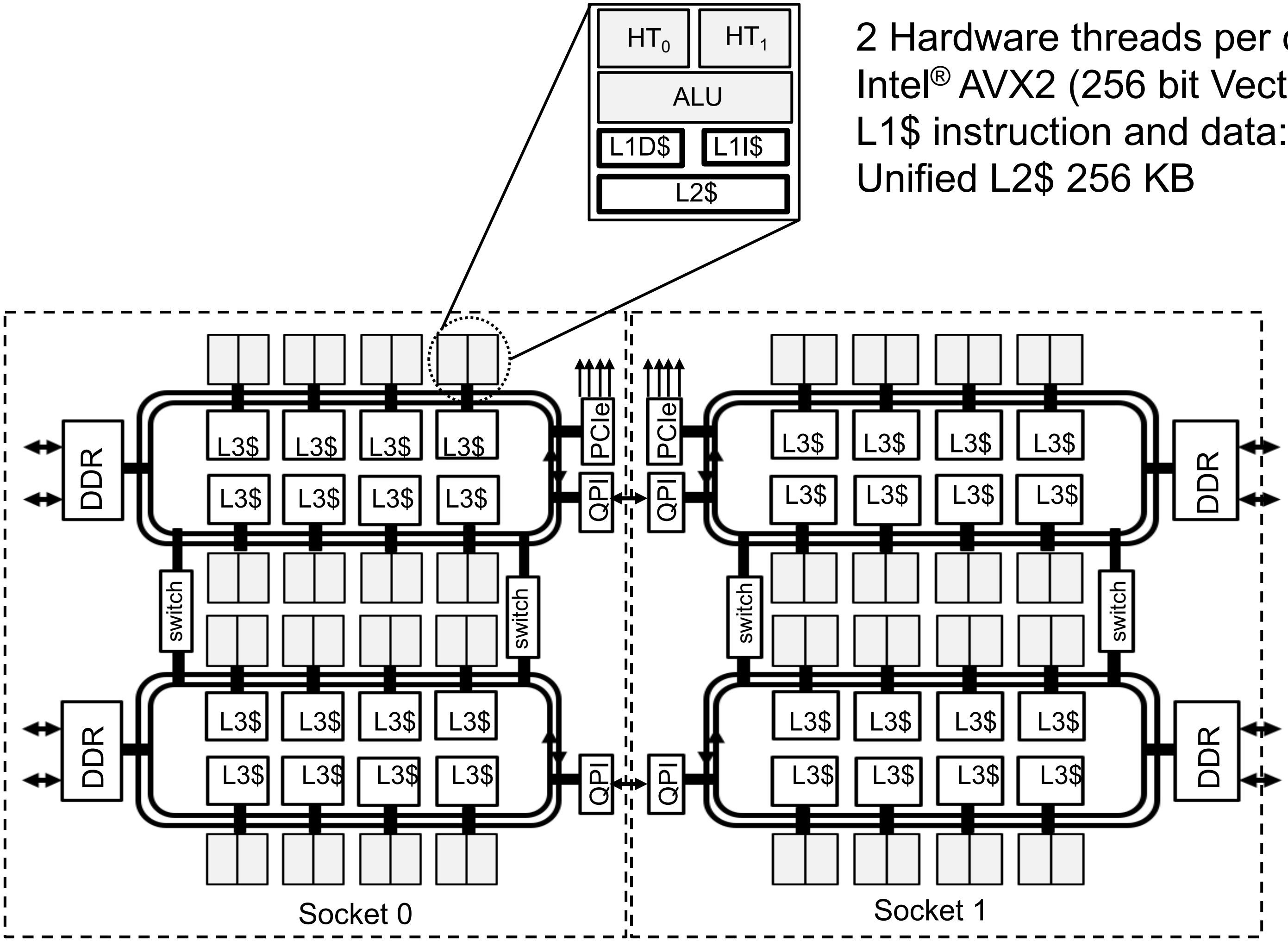
OpenMP basic definitions: Basic Solution stack



In learning OpenMP, you consider a Symmetric Multiprocessor (SMP) ….
i.e. lots of threads with "**equal cost access**" to memory

A Typical CPU Node in an HPC System

2 Intel® Xeon™ E5-2698 v3 CPUs (Haswell) per node (launched Q3'14)



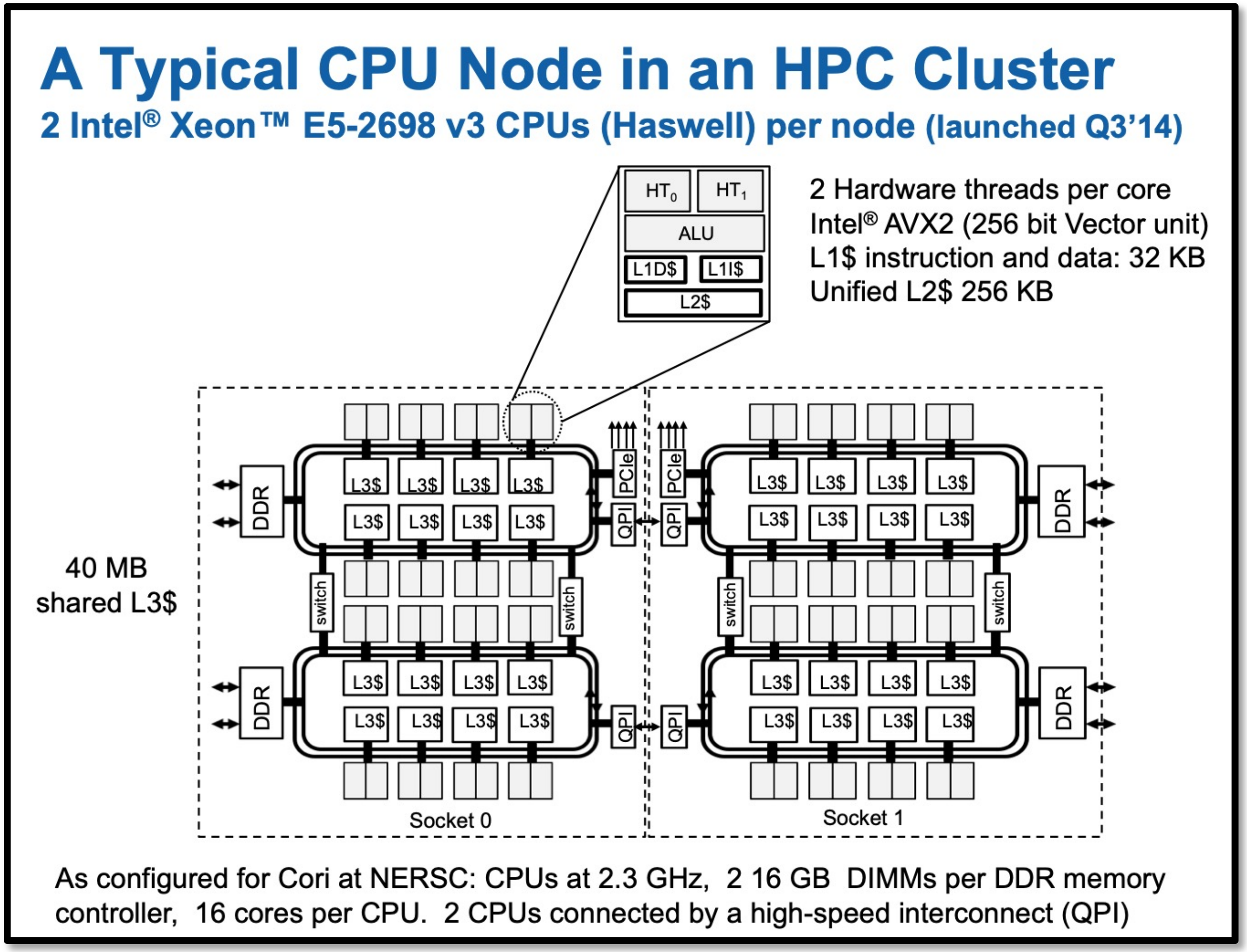
2 Hardware threads per core
Intel® AVX2 (256 bit Vector unit)
L1$ instruction and data: 32 KB
Unified L2$ 256 KB

40 MB
shared L3$

As configured for Cori at NERSC: CPUs at 2.3 GHz, 2 16 GB DIMMs per DDR memory controller, 16 cores per CPU. 2 CPUs connected by a high-speed interconnect (QPI)

Does this look like an SMP node to you?

There may be a single address space, but there are multiple levels of non-uniformity to the memory. This is a **N**on-**U**niform **M**emory **A**rchitecture (NUMA)



A Typical CPU Node in an HPC Cluster

2 Intel® Xeon™ E5-2698 v3 CPUs (Haswell) per node (launched Q3'14)

2 Hardware threads per core
Intel® AVX2 (256 bit Vector unit)
L1$ instruction and data: 32 KB
Unified L2$ 256 KB

40 MB
shared L3$

As configured for Cori at NERSC: CPUs at 2.3 GHz, 2 16 GB DIMMs per DDR memory controller, 16 cores per CPU. 2 CPUs connected by a high-speed interconnect (QPI)

Even a single CPU is properly considered a NUMA architecture

Exploring your NUMA world: numactl

- numactl is a Linux command to control the NUMA policy

- You can use it to learn about the NUMA features of your system:

- On Cori at NERSC (Two 16 core “Haswell” Intel® Xeon™ CPUs per node)

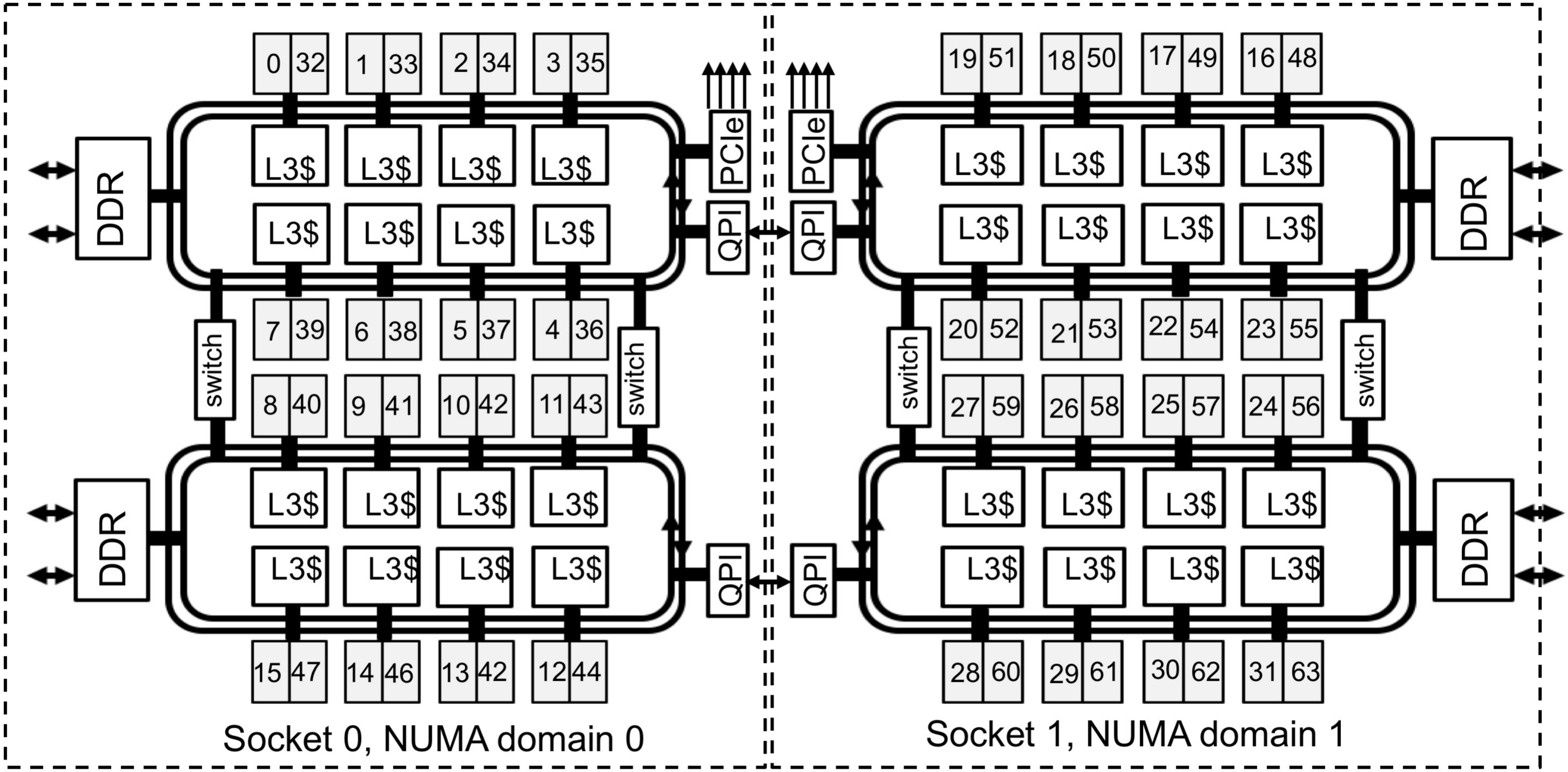
```
% numactl -H
available: 2 nodes (0-1)
node 0 cpus: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47
node 0 size: 64430 MB
node 0 free: 63002 MB
node 1 cpus: 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 48 49 50 51 52 53 54 55 56 57 58 59 60 61
62 63
node 1 size: 64635 MB
node 1 free: 63395 MB
node distances:node   0   1
0:  10  21
1:  21  10
```

Shows relative costs …. In this case, there’s a factor of two in the cost of the local (on CPU) DRAM vs going to the other socket

On some systems, this information is found with the Linux command: **lscpu**

Exploring your NUMA world: NUMACTL

- numactl shows you how the OS processor-numbers map onto the physical cores of the chip:



2 Intel® Xeon™ E5-2698 v3 CPUs (Haswell) per node (launched Q3'14)

Writing NUMA-aware OpenMP code

- ➡ Memory Affinity
  - Maximize reuse of data in the cache hierarchy
  - Maximize reuse of data in memory pages

- Control the places where threads are mapped
  - Place threads onto cores to optimize performance
  - Keep threads working on similar data close to each other
  - Maximize utilization of memory controllers by spreading threads out

- Processor binding … Disable thread migration
  - By Default, an OS migrates threads to maximize utilization of resources on the chip.
  - To Optimize for NUMA, we need to turn off thread migration … bind threads to a processor/core

Memory Affinity: “First Touch” Policy

Step 1.1 Initialization

by primary* thread only

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

Step 1.2 Initialization

by all threads

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

Step 2 Compute

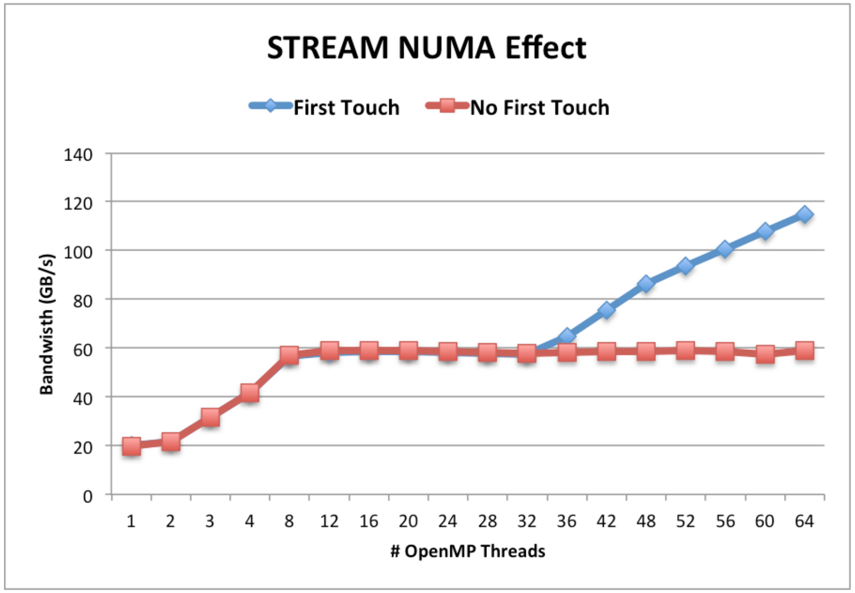
```
#pragma omp parallel for
for (j=0; j<VectorSize; j++) {
a[j]=b[j]+d*c[j];}
```

Memory affinity is defined at initialization.

Memory will be local to the thread which initializes it.

This is called **first touch** policy.

Red: step 1.1 + step 2. No First Touch
Blue: step 1.2 + step 2. First Touch



*the term “master” has been deprecated in OpenMP 5.1 and replaced with the term “primary”.

*OMP_PROC_BIND=close   to be described later

Writing NUMA-aware OpenMP code

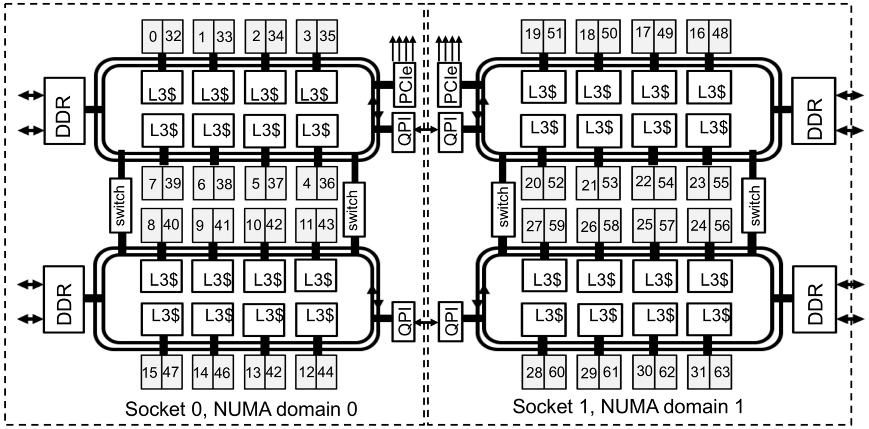
- Memory Affinity
  - Maximize reuse of data in the cache hierarchy
  - Maximize reuse of data in memory pages

- ➡ Control the places where threads can run
  - Keep threads working on similar data close to each other
  - Maximize utilization of memory controllers by spreading threads out

- Processor binding … Disable thread migration
  - By Default, an OS migrates threads to maximize utilization of resources on the chip.
  - To OPlace threads onto cores to optimize performance
  - ptimize for NUMA, we need to turn off thread migration … bind threads to a processor/core

The concept of places

- The Operating System assigns logical CPU IDs to hardware threads.
- Recall … the linux command *numactl –H* returns those numbers.

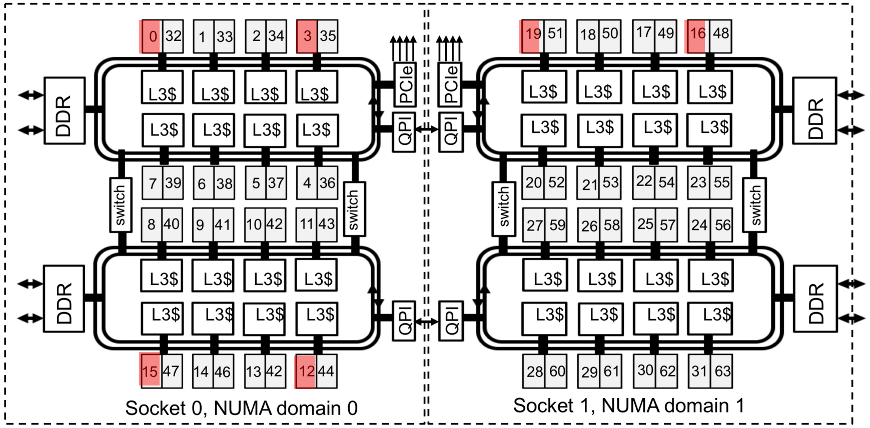


- A place: numbers between { }:

  export OMP_PLACES=“{0,1,2,3}”

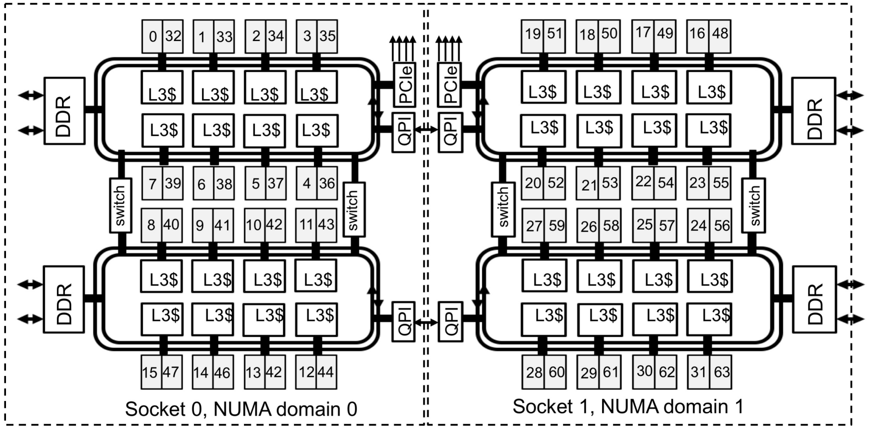
- A place defines where threads can run

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

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

}
```



The concept of places

- The Operating System assigns logical CPU IDs to hardware threads.
- Recall … the linux command *numactl –H* returns those numbers.



- Set with an environment variable:

  export OMP_PLACES="{0,1,2,3}"

- Can also specify with {lower-bound:length:stride}

Default Stride is 1

OMP_PLACES="{0,1,2,3}" → OMP_PLACES="{0:4:1}" → OMP_PACES="{0:4}"

- Can define multiple places:

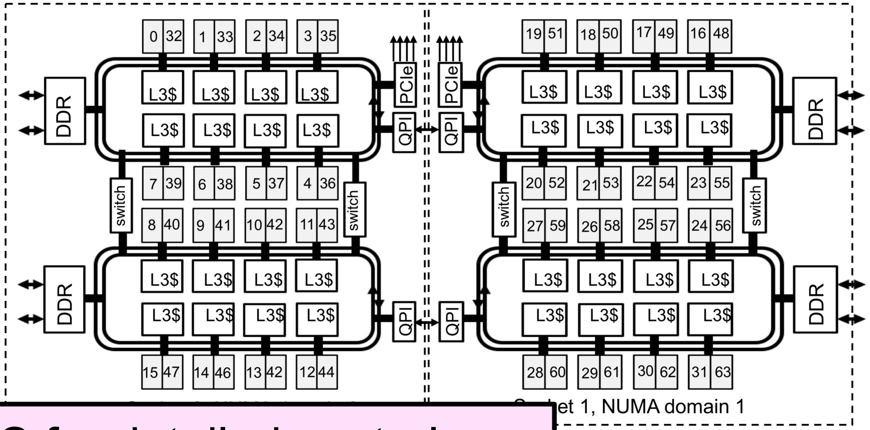
OMP_PLACES="{0,1,2,3},{4,6,8},{9,10,11,12}"

OMP_PLACES="{0,4},{4,3:2},{9:4}"

These are equivalent

The concept of places

- The Operating System assigns logical CPU IDs to hardware threads.
- Recall … the linux command *numactl –H* returns those numbers.



et 1, NUMA domain 1

- Set with an environment variable:

  expo

  1

- Can a

  OMP_ 4}"

- Can

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

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

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

OMP_PLACES="{0,1,2,3},{4,6,8},{9,10,11,12}"

OMP_PLACES="{0,4},{4,3:2},{9:4}"

These are equivalent

OMP_PLACES:

- OMP_PLACES can use the following abstract names:
  - threads: each place corresponds to a single hardware thread on the target machine.
  - cores: each place corresponds to a single core (having one or more hardware threads) on the target machine.
  - sockets: each place corresponds to a single socket (consisting of one or more cores) on the target machine.
- Examples:
  - export OMP_PLACES=threads
  - export OMP_PLACES=cores

Writing NUMA-aware OpenMP code

- Memory Affinity
  - Maximize reuse of data in the cache hierarchy
  - Maximize reuse of data in memory pages

- Control the places where threads can run
  - Keep threads working on similar data close to each other
  - Maximize utilization of memory controllers by spreading threads out

- Processor binding … Disable thread migration
  - By Default, an OS migrates threads to maximize utilization of resources on the chip.
  - To OPlace threads onto cores to optimize performance
  - ptimize for NUMA, we need to turn off thread migration … bind threads to a processor/core

Processor binding

- Control with the environment variable OMP_PROC_BIND
- The following values are recognized
  - **true**: Thread affinity enabled … threads stay put once they are placed on a system.
  - **false**: thread affinity disabled … the OS can migrate threads at will
  - **primary***: threads are assigned to the same processor/core as the primary thread of the team.
  - **close**: threads assigned "round robin" to places incremented by one starting from the place where the primary thread of the team is located.
  - **spread**: threads are spread out evenly among the available places
- The values **primary**, **close** and **spread** imply the value **true**

- Example:

```
export OMP_PROC_BIND=close
```

*the term "master" has been deprecated in OpenMP 5.1 and replaced with the term "primary".

Processor binding: Example

- Consider a CPU with 4 cores, 2 hyperthreads per core, and OMP_NUM_THREADS=4

- close: Bind threads as close to each other as possible

Node	Core 0		Core 1		Core 2		Core 3	
	HT1	HT2	HT1	HT2	HT1	HT2	HT1	HT2
Thread	0	1	2	3				

- spread: Bind threads as far apart as possible.

Node	Core 0		Core 1		Core 2		Core 3	
	HT1	HT2	HT1	HT2	HT1	HT2	HT1	HT2
Thread	0		1		2		3	

OMP_PROC_BIND Choices for STREAM

OMP_NUM_THREADS=32
OMP_PLACES=threads

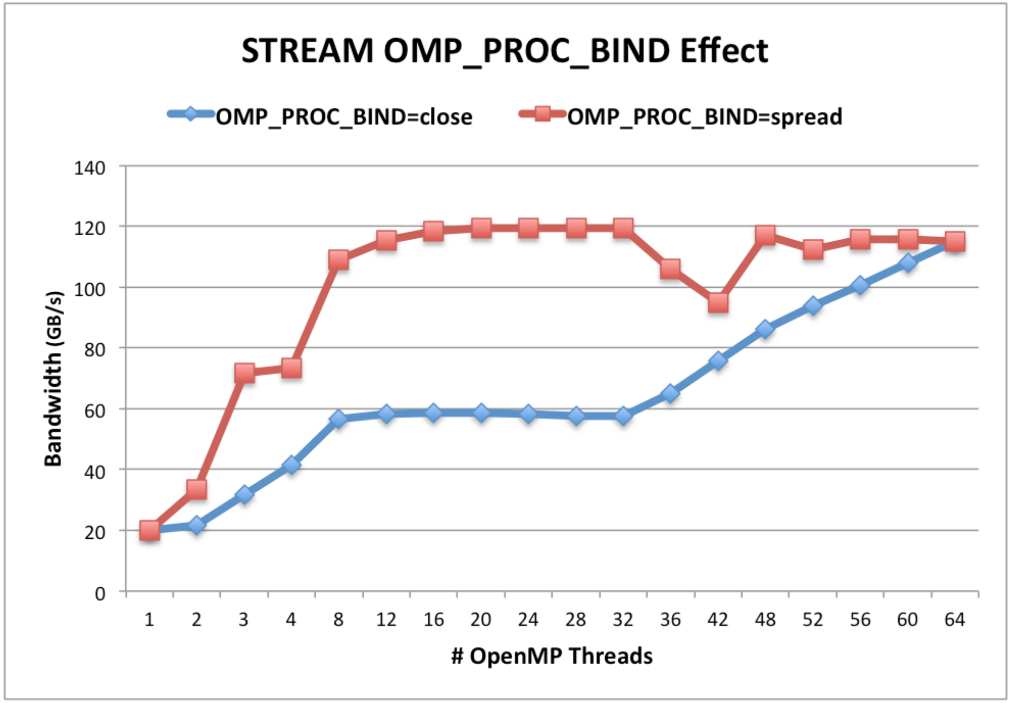
OMP_PROC_BIND=close

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

OMP_PROC_BIND=spread

Threads 0 to 31 bind to CPUs 0,1,2,… to 31. Both sockets and memory are used to maximize memory bandwidth.

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



Affinity Clauses for OpenMP Parallel Construct

- The “num_threads” and “proc_bind” clauses can be used
  - The values set with these clauses take precedence over values set by runtime environment variables
- Helps code portability
- Examples:
  - C/C++:

    ```
    #pragma omp parallel num_threads(2) proc_bind(spread)
    ```

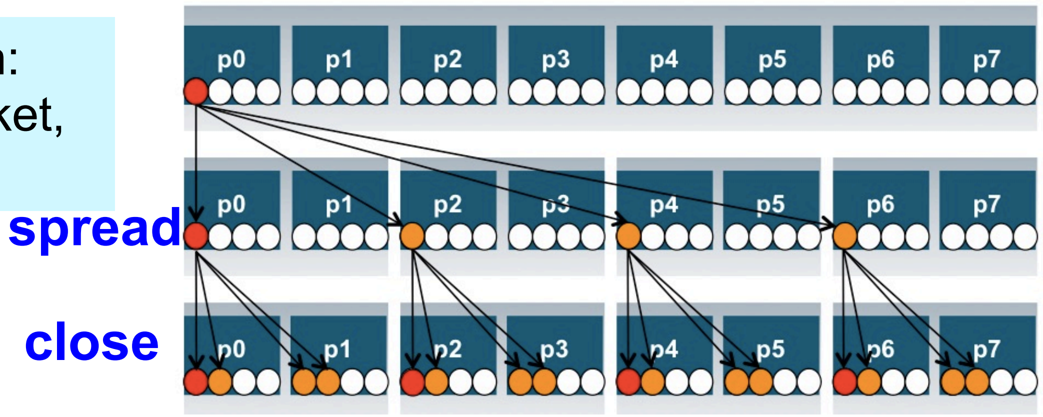
  - Fortran:

    ```
    !$omp parallel num_threads (2) proc_bind (spread)
    ...
    !$omp end parallel
    ```

Process and Thread Affinity in Nested OpenMP

- A combination of OpenMP environment variables and run time flags are needed for different compilers and different batch schedulers on different systems.

Illustration of a system with:
2 sockets, 4 cores per socket,
4 hyper-threads per core



Example: Use Intel compiler with SLURM on Cori Haswell:

```
export OMP_NESTED=true
export OMP_MAX_ACTIVE_LEVELS=2
export  OMP_NUM_THREADS=4,4
export OMP_PROC_BIND=spread,close
export OMP_PLACES=threads
srun -n 4 -c 16 –cpu_bind=cores ./nested.intel.cori
```

- Use num_threads clause in source codes to set threads for nested regions.
- For most other non-nested regions, use OMP_NUM_THREADS environment variable for simplicity and flexibility.

A NUMA Case study

Benchmarking … I must control everything!

- Goal: To compare different programming systems applied to the same problem:
  - We must control everything we can to make sure any observed differences are due to the different programming systems.

- We need to know exactly which cores we are using and how thread IDs map onto cores … so we can understand data detailed memory movement and make sure it’s the same between the different test cases.

Step 1: Know your system

- My system did not have numactl or Hwloc. So I went with my third option …. lscpu (note: I'm only showing a subset of the actual output):

```
$ lscpu
Architecture:             x86_64
CPU op-mode(s):           32-bit, 64-bit
Byte Order:               Little Endian
Address sizes:            46 bits physical, 48 bits virtual
CPU(s):                   72
On-line CPU(s) list:      0-71
Thread(s) per core:       2
Core(s) per socket:       18
Socket(s):                2
NUMA node(s):             2
Vendor ID:                GenuineIntel
CPU family:               6
Model:                    63
Model name:               Intel(R) Xeon(R) CPU E5-2699 v3 @ 2.30GHz
Stepping:                 2
CPU MHz:                  1197.539
CPU max MHz:              3600.0000
CPU min MHz:              1200.0000
L1d cache:                1.1 MiB
L1i cache:                1.1 MiB
L2 cache:                 9 MiB
L3 cache:                 90 MiB
NUMA node0 CPU(s):        0-17,36-53
NUMA node1 CPU(s):        18-35,54-71
```

SMT enabled … two HW threads per core

2 CPUs (sockets) with 18 physical cores per CPU

Note: a HW thread is a CPU (or core) as far as the OS is concerned. These two lines show you the numbering of these "cores".

Step 1: Know your system

- My system did not have numactl or Hwloc. So I went with my third option …. lscpu (note: I'm only showing a subset of the actual output):

```
$ lscpu
Architecture:              x86_64
CPU op-mode(s):             32-bit, 64-bit
Byte Order:                Little Endian
Address sizes:             46 bits physical, 48 bits virtual
CPU(s):                    72
On-line CPU(s) list:       0-71
Thread(s) per core:        2
Core(s) per socket:        18
Socket(s):                 2
NUMA node(s):              2
Vendor ID:                 GenuineIntel
CPU family:                6
Model:                     63
Model name:                Intel(R) Xeon(R) CPU E5-2699 v3 @ 2.30GHz
Stepping:                  2
CPU MHz:                   1197.539
CPU max MHz:                3600.0000
CPU min MHz:                1200.0000
L1d cache:                  1.1 MiB
L1i cache:                  1.1 MiB
L2 cache:                   9 MiB
L3 cache:                   90 MiB
NUMA node0 CPU(s):          0-17,36-53
NUMA node1 CPU(s):          18-35,54-71
```

SMT enabled … two HW threads per core

2 CPUs (sockets) with 18 physical cores per CPU

The numbering of these "cores" (in 2 sockets).

0/36	1/37	2/38	3/39	4/40	5/41	6/42	7/43	8/44
9/45	10/46	11/47	12/48	13/49	14/50	15/51	16/52	17/53

18/54	19/55	20/56	21/57	22/58	23/59	24/60	25/61	26/62
27/63	28/64	29/65	30/66	31/67	32/68	33/69	34/70	35/71

Setup a runscript (so you can reproduce the computations later)

```
#! /usr/bin/env bash
#  Run script for DGEMM with C and OpenMP

# Define shared parameters for the calculations we will run
BLOCK=0
ORDER=1000
ITERS=5

# setup environment for the intel compilers
source /opt/intel/compilers_and_libraries_2020.4.304/linux/bin/compilervars.sh -arch intel64

# Setup display of mapping from OpenMP threads to "hardware" threads.
export OMP_DISPLAY_AFFINITY=true
export OMP_AFFINITY_FORMAT="Thrd Lev=%3L, thrd_num=%5n,  thrd_aff=%15A"

# Enable explicit affinity control.
export OMP_PLACES="{0},{1},{2},{3},{4},{5},{6},{7},{8},{9},{10},{11},{12},{13},{14},{15},{16}"
export OMP_PROC_BIND=close

./dgemm 8  $ITERS $ORDER $BLOCK
./dgemm 16 $ITERS $ORDER $BLOCK
```

A NUMA case study: Results

```
Parallel Research Kernels version 2.17
OpenMP Dense matrix-matrix multiplication
Thrd Lev=1  , thrd_num=0    ,  thrd_aff=0
Thrd Lev=1  , thrd_num=4    ,  thrd_aff=4
Thrd Lev=1  , thrd_num=3    ,  thrd_aff=3
Thrd Lev=1  , thrd_num=5    ,  thrd_aff=5
Thrd Lev=1  , thrd_num=1    ,  thrd_aff=1
Thrd Lev=1  , thrd_num=2    ,  thrd_aff=2
Thrd Lev=1  , thrd_num=6    ,  thrd_aff=6
Thrd Lev=1  , thrd_num=7    ,  thrd_aff=7
Matrix order          = 1000
Number of threads     = 8
Rate : 21650.601956  +/- 1589.413250 MFlops/s
```

Notice the one-to-one mapping of thread ID onto hardware thread.

Normally, this is going too far, but for benchmarking, this is a handy trick.

```
Parallel Research Kernels version 2.17
OpenMP Dense matrix-matrix multiplication
Thrd Lev=1  , thrd_num=0    ,  thrd_aff=0
Thrd Lev=1  , thrd_num=13   ,  thrd_aff=13
Thrd Lev=1  , thrd_num=4    ,  thrd_aff=4
Thrd Lev=1  , thrd_num=11   ,  thrd_aff=11
Thrd Lev=1  , thrd_num=10   ,  thrd_aff=10
Thrd Lev=1  , thrd_num=8    ,  thrd_aff=8
Thrd Lev=1  , thrd_num=9    ,  thrd_aff=9
Thrd Lev=1  , thrd_num=1    ,  thrd_aff=1
Thrd Lev=1  , thrd_num=3    ,  thrd_aff=3
Thrd Lev=1  , thrd_num=2    ,  thrd_aff=2
Thrd Lev=1  , thrd_num=12   ,  thrd_aff=12
Thrd Lev=1  , thrd_num=7    ,  thrd_aff=7
Thrd Lev=1  , thrd_num=6    ,  thrd_aff=6
Thrd Lev=1  , thrd_num=5    ,  thrd_aff=5
Thrd Lev=1  , thrd_num=14   ,  thrd_aff=14
Thrd Lev=1  , thrd_num=15   ,  thrd_aff=15
Matrix order          = 1000
Number of threads     = 16
Rate : 38765.867067  +/- 3303.460980 MFlops/s
```

Summary for Thread Affinity and Data Locality

- Achieving best data locality, and optimal process and thread affinity is crucial in getting good performance with OpenMP, yet it is not straightforward to do so.
  - Understand the node architecture with tools such as “numactl -H” first.
  - Always use simple examples with the same settings for your real application to verify first.
- Exploit first touch data policy, optimize code for cache locality.
- Pay special attention to avoid false sharing.
- Put threads far apart (spread) may improve aggregated memory bandwidth and available cache size for your application, but may also increase synchronization overhead. And putting threads “close” have the reverse impact as “spread”.
- For nested OpenMP, set OMP_PROC_BIND=spread,close is generally recommended.

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 Revisited: Options for Mutual exclusion
    - Thread Affinity and Data Locality
    - Thread Private Data
    - Memory models and point-to-point Synchronization
    - Programming your GPU with OpenMP

Data Sharing: Threadprivate

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

A Threadprivate Example (C)

Use threadprivate to create a counter for each thread.

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

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

Data Copying: Copyin

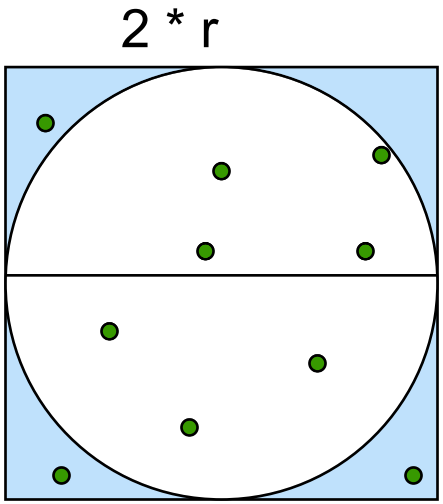
You initialize threadprivate data using a copyin clause.

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

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

!$OMP PARALLEL COPYIN(A)

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

!$OMP END PARALLEL

end
```

Exercise: Monte Carlo Calculations

Using random numbers to solve tough problems

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



N= 10	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.
- Run it multiple times with varying numbers of threads.
- Is the program working correctly? Is there anything wrong?

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;
```

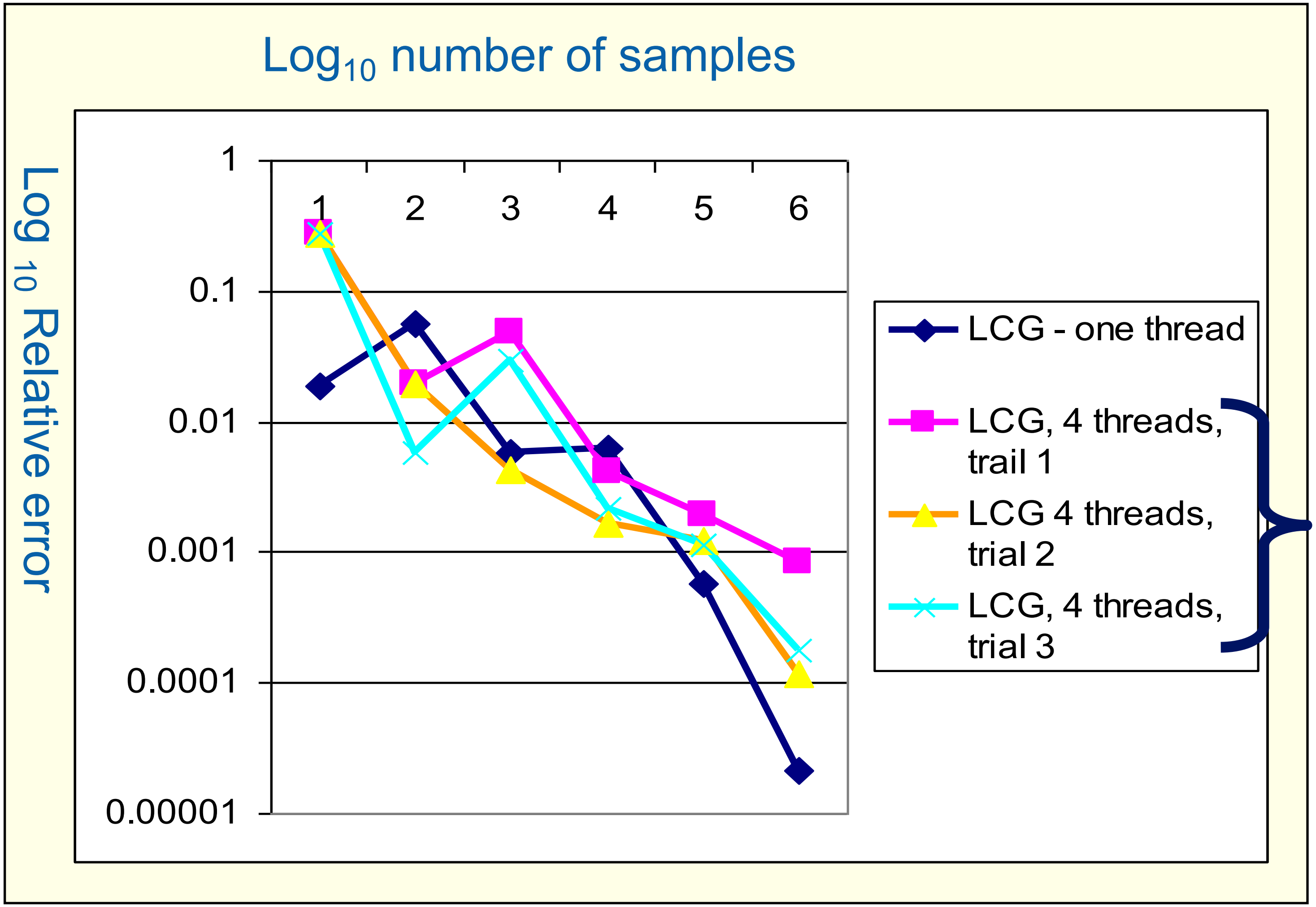
- If you pick the multiplier and addend correctly, LCG has a period of PMOD.
- Picking good LCG parameters is complicated, so look it up (Numerical Recipes is a good source). I used the following:
  - MULTIPLIER = 1366
  - ADDEND = 150889
  - PMOD = 714025

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.

Exercise: Monte Carlo pi (cont)

- Create a threadsafe version of the monte carlo pi program

- Do not change the interfaces to functions in random.c
  - This is an exercise in modular software … why should a user of your parallel random number generator have to know any details of the generator or make any changes to how the generator is called?
  - The random number generator must be thread-safe

- Verify that the program is thread safe by running multiple times for a fixed number of threads.

- Any concerns with the program behavior?

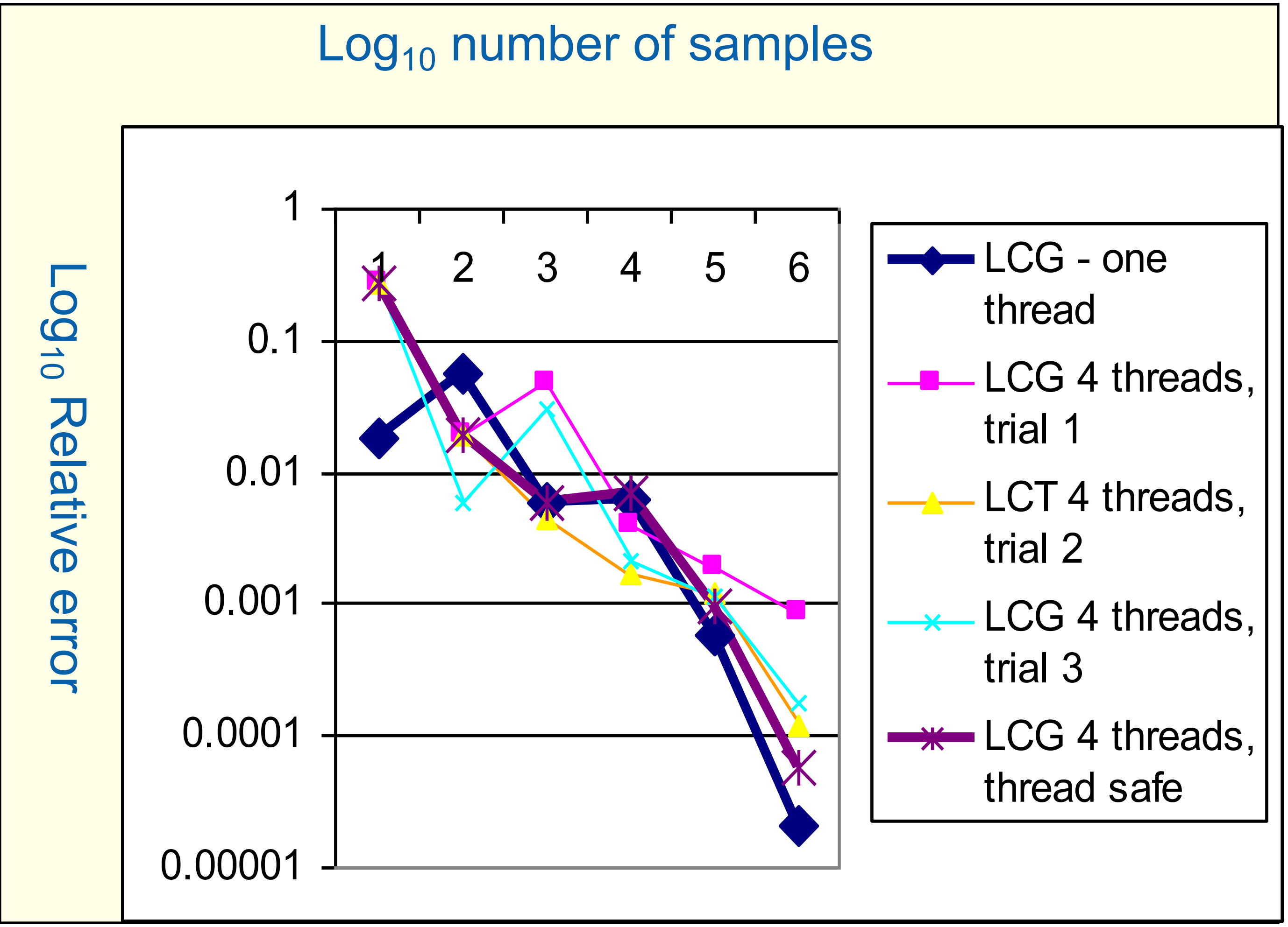
LCG code: threadsafe version

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

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

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

random_last carries state between random number computations,

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

Thread safe random number generators



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

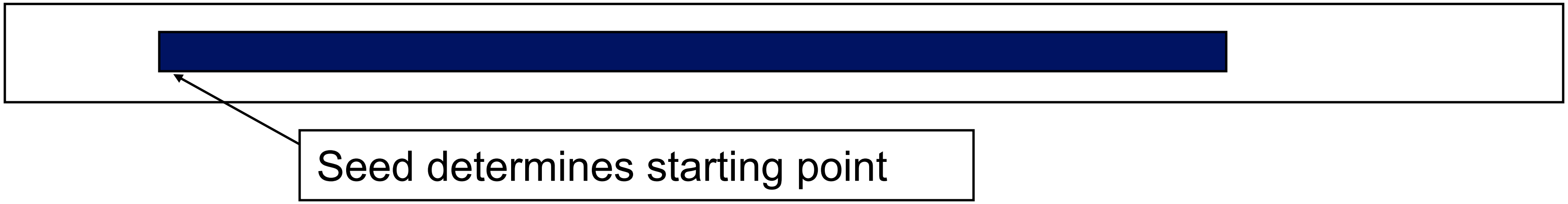
But for large number of samples, its quality is lower than the one thread result!

Why?

Pseudo Random Sequences

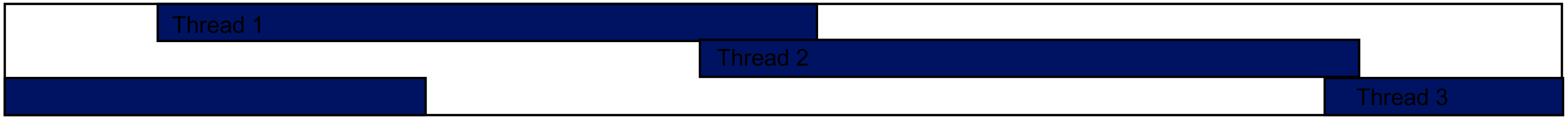
- Random number Generators (RNGs) define a sequence of pseudo-random numbers of length equal to the period of the RNG

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



Seed determines starting point

- 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;
```

Select type of RNG and set seed

Initialize a stream or pseudo random numbers

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

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

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

```
vslDeleteStream( &stream );
```

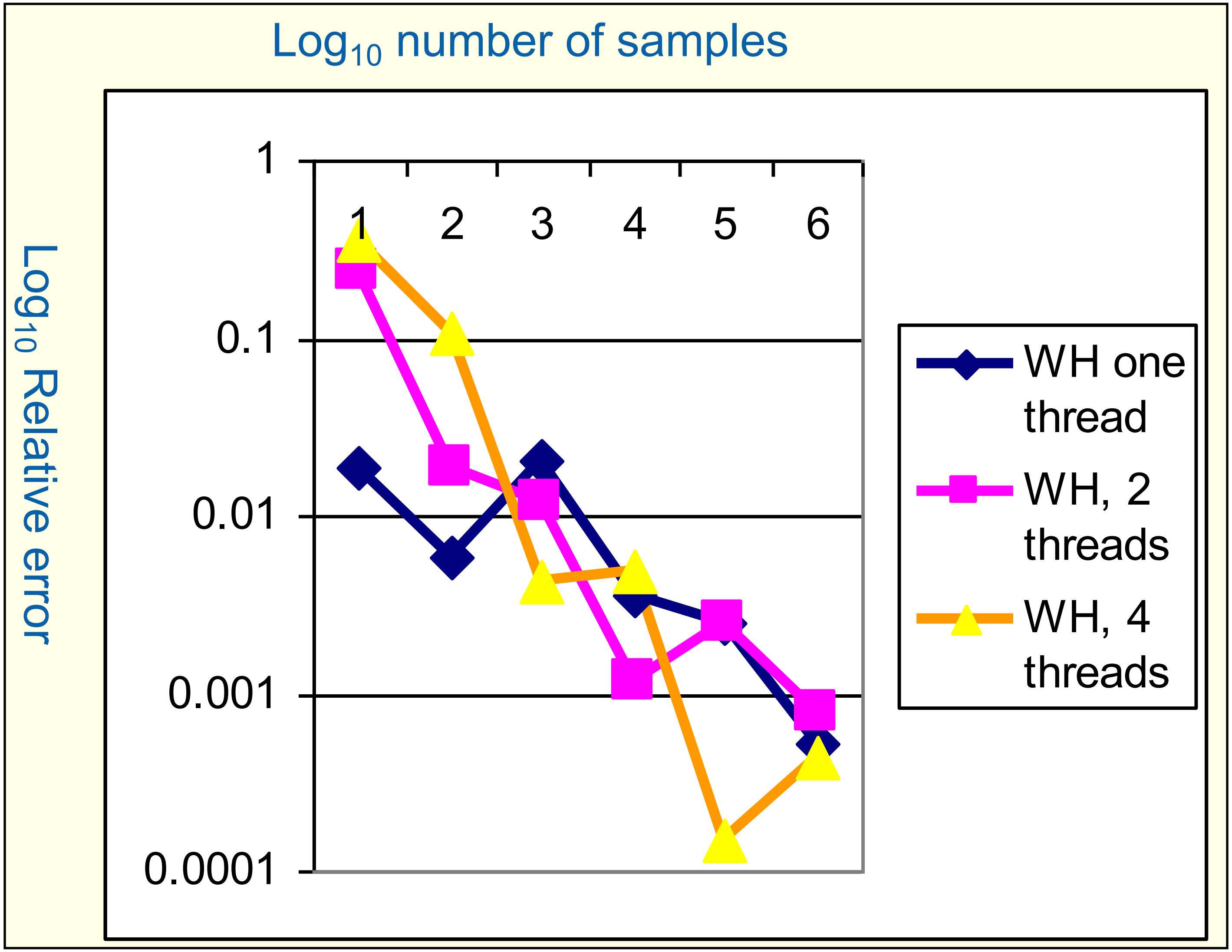
Delete the stream when you are done

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 ith 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

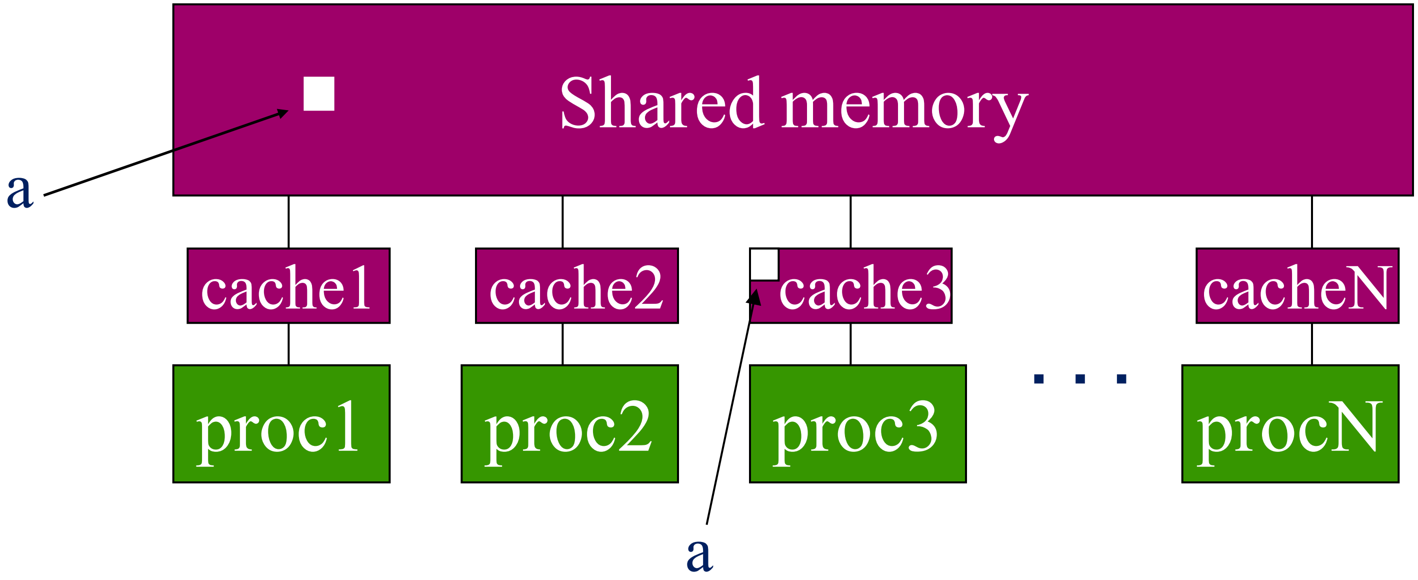
Used the MKL library with two generator streams per computation: one for the x values (WH) and one for the y values (WH+1). Also used the leapfrog method to deal out iterations among threads.

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 Revisited: Options for Mutual exclusion
  - Thread Affinity and Data Locality
  - Thread Private Data
  - ➡ Memory models and point-to-point Synchronization
  - Programming your GPU with OpenMP

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 primary* regions)

*the term “master” has been deprecated in OpenMP 5.1 and replaced with the term “primary”.

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

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

Atomics and Synchronization Flags

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

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

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

OpenMP 4.0 Atomic: Sequential Consistency

4.0

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

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

Atomics and Synchronization Flags (4.0)

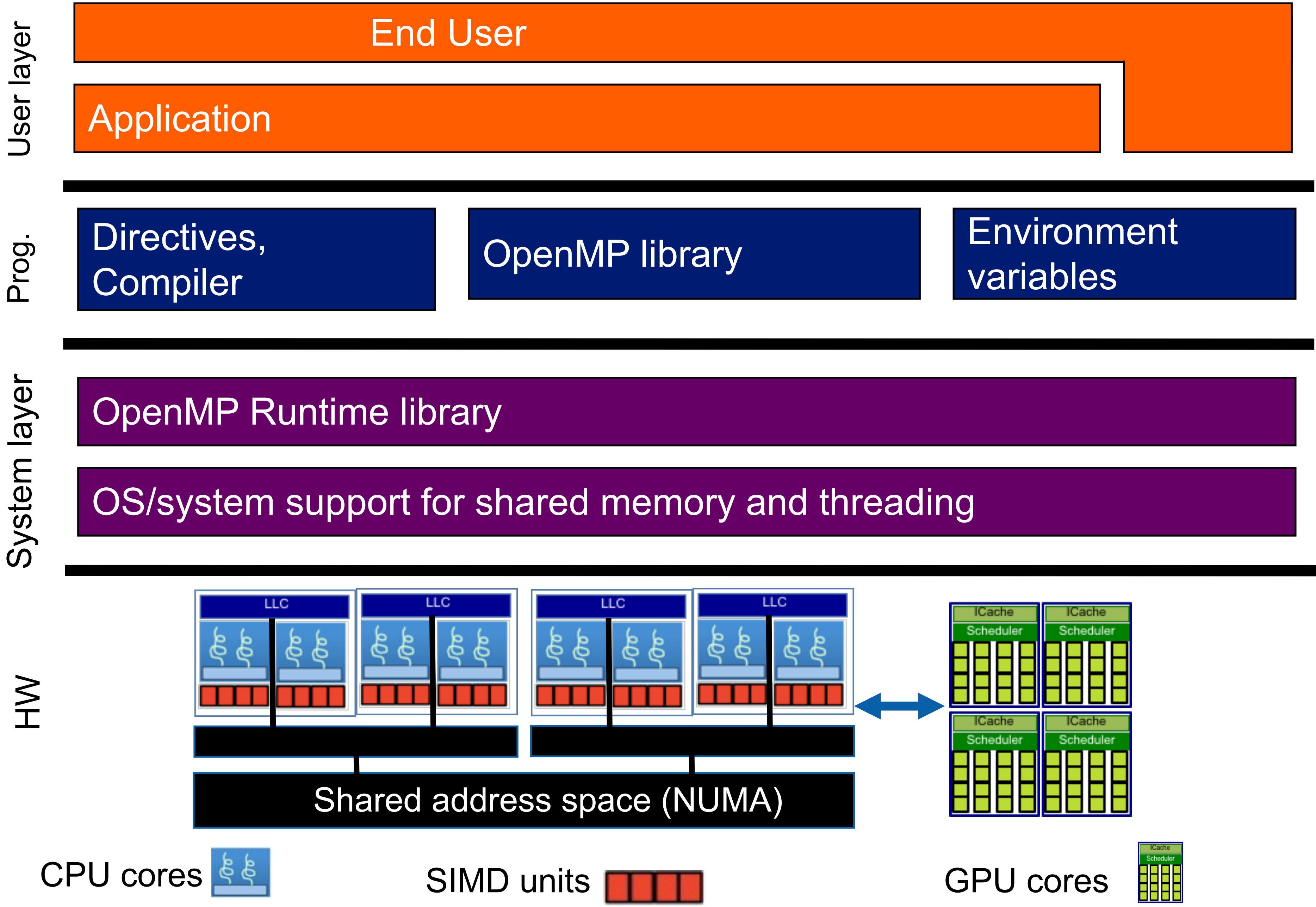
```
int main()
{   double *A, sum, runtime;
    int numthreads, flag = 0, flg_tmp;
    A = (double *)malloc(N*sizeof(double));
    #pragma omp parallel sections
    {
       #pragma omp section
        {  fill_rand(N, A);

           #pragma omp atomic write seq_cst
                    flag = 1;

        }
        #pragma omp section
        {  while (1){

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

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

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

Outline

- Introduction to OpenMP
- Creating Threads
- Synchronization
- Parallel Loops
- Data Environment
- Memory Model
- Irregular Parallelism and Tasks
- Recap
- Beyond the Common Core:
  - Worksharing Revisited
  - Synchronization Revisited: Options for Mutual exclusion
  - Thread Affinity and Data Locality
  - Thread Private Data
  - Memory models and point-to-point Synchronization
  - Programming your GPU with OpenMP

OpenMP basic definitions: Basic Solution stack



User layer

End User

Application

Prog.

Directives, Compiler

OpenMP library

Environment variables

System layer

OpenMP Runtime library

OS/system support for shared memory and threading

HW

Shared address space (NUMA)

CPU cores

SIMD units

GPU cores

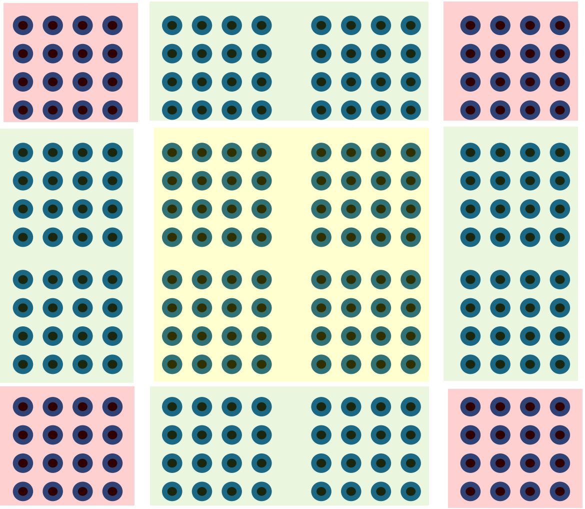
How do we execute code on a GPU: The SIMT model (Single Instruction Multiple Thread)

1. Turn source code into a scalar work-item

```
extern void reduce(  __local  float*,   __global float*);

__kernel void pi(  const int niters, float  step_size,
        __local  float*  l_sums,  __global float*  p_sums)
{
   int n_wrk_items  = get_local_size(0);
   int loc_id       = get_local_id(0);
   int grp_id   = get_group_id(0);
   float x, accum = 0.0f;    int i,istart,iend;

   istart =   (grp_id * n_wrk_items   + loc_id) * niters;
   iend   = istart+niters;

   for(i= istart; i<iend; i++){
       x = (i+0.5f)*step_size;    accum += 4.0f/(1.0f+x*x); }

   l_sums[local_id] = accum;
   barrier(CLK_LOCAL_MEM_FENCE);
   reduce(l_sums, p_sums);
}
```

This is OpenCL kernel code … the sort of code the OpenMP compiler generates on your behalf

2. Map work-items onto an an N dim index space.



3. Map data structures onto the same index space

4. Run on hardware designed around the same SIMT execution model

PCI Express 3.0 Host Interface
GigaThread Engine
SMX
L2 Cache
Memory Controller

Third Party names are the property of their owners

How do we execute code on a GPU: OpenCL and CUDA nomenclature

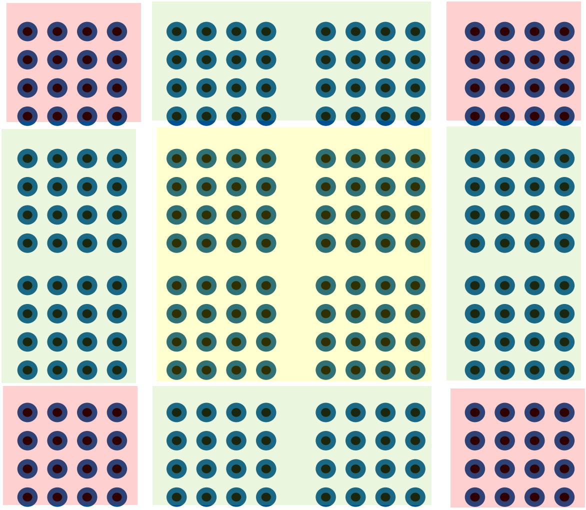
Turn source code into a scalar **work-item** (a CUDA **thread**)

```
extern void reduce(  __local  float*,   __global float*);

__kernel void pi(  const int niters, float  step_size,
        __local  float* l_sums,  __global float*  p_sums)
{
   int n_wrk_items  = get_local_size(0);
   int loc_id       = get_local_id(0);
   int grp_id   = get_group_id(0);
   float x, accum = 0.0f;    int i,istart,iend;

   istart =   (grp_id * n_wrk_items   + loc_id) * niters;
   iend   = istart+niters;

   for(i= istart; i<iend; i++){
       x = (i+0.5f)*step_size;    accum += 4.0f/(1.0f+x*x); }

   l_sums[local_id] = accum;
   barrier(CLK_LOCAL_MEM_FENCE);
   reduce(l_sums, p_sums);
}
```

This code defines a **kernel**

Submit a kernel to an OpenCL **command queue** or a CUDA **stream**

Organize work-items into **work-groups** and map onto an an N dim index space. Cuda calls a work-group a **thread-block**

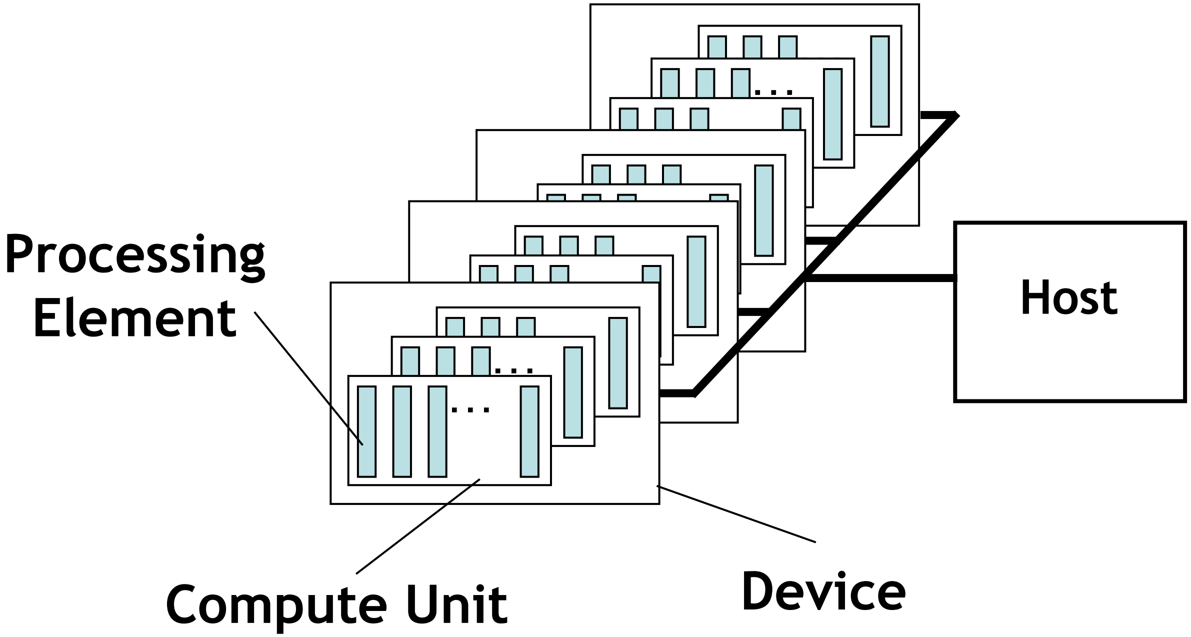


OpenCL index space is called an **NDRange.** CUDA calls this a **Grid**

It's called SIMT, but GPUs are really vector-architectures with a block of work-items executing together (a **subgroup** in OpenCL or a **warp** with Cuda)

Third Party names are the property of their owners

A Generic Host/Device Platform Model



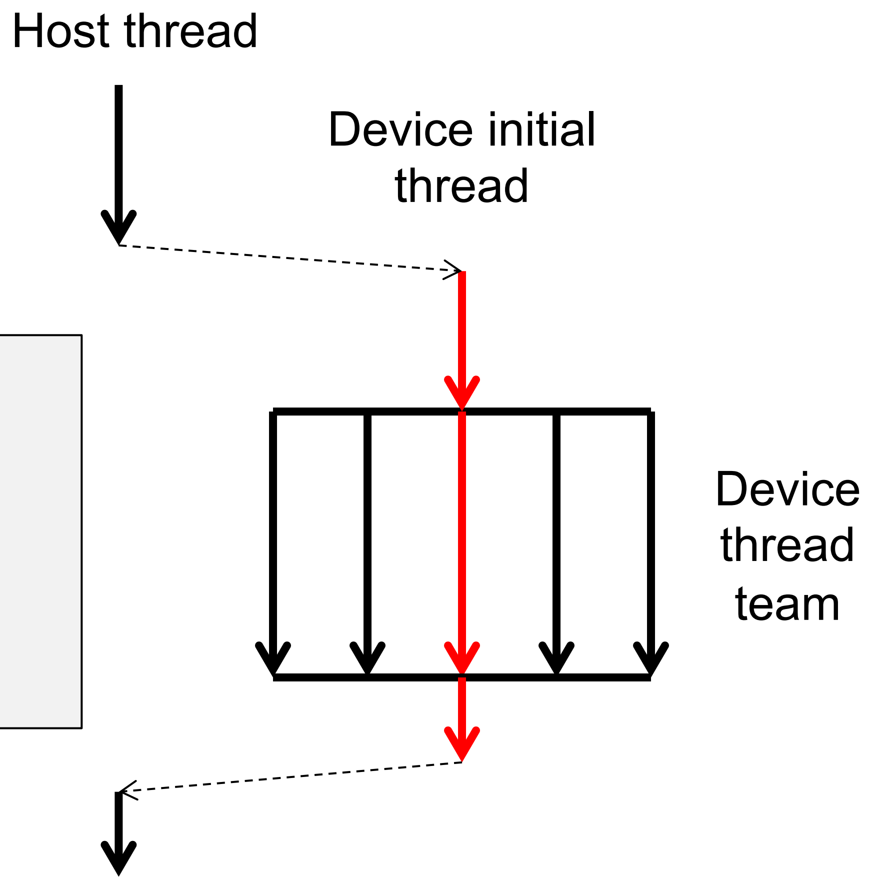
- One ***Host*** and one or more ***Devices***
  - Each Device is composed of one or more ***Compute Units***
  - Each Compute Unit is divided into one or more ***Processing Elements***
- Memory divided into ***host memory*** and ***device memory***

Third party names are the property of their owners.

Accelerated workshare v1.0

- **Simply add a target construct**

```
#pragma omp target
#pragma omp parallel for
 for (i=0;i<N;i++)
   …
```



– Transfer control of execution to a **SINGLE** device thread
– Only one team of threads workshares the loop

The target data environment

- Remember: distinct memory spaces on host and device.

- OpenMP uses a combination of *implicit* and *explicit* memory movement.

- Data may move between the host and the device in well defined places:
  - Firstly, at the beginning and end of a **target** region:

    ```
    #pragma omp target
    { // Data may move here
       …
    } // and here
    ```

  - We'll discuss the other places later…

Default Data Mapping: implicit movement with a target region

- Scalar variables:
  - Examples:
    - int N; double x;
  - OpenMP implicitly maps scalar variables as **firstprivate**
    - A new value per work-item initialized with the original value (in OpenCL nomenclature, the firstprivate goes in private memory).
  - The variable *is not* copied back to the host at the end of the target region.
  - OpenMP target regions for GPUs execute with CUDA/OpenCL, and a firstprivate scalar can be launched as a parameter to a kernel function without the overhead of setting up a variable in device memory.

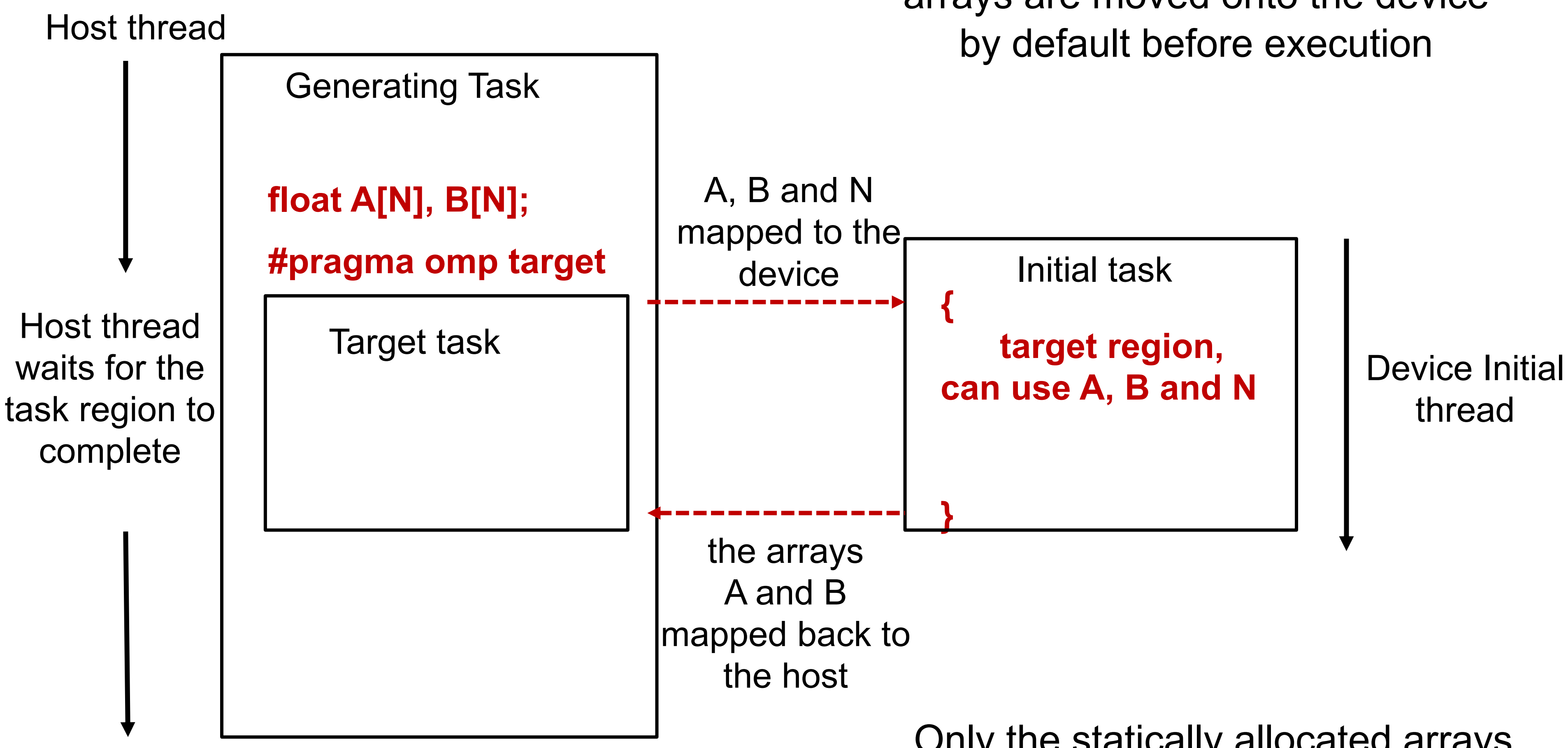
Default Data Mapping:
implicit movement with a target region

- Non-scalar variables:
  - Must have a *complete type*.
  - Example: fixed sized (stack) array:
    - double A[1000];
  - Copied to the device at the start of the **target** region, *and* copied back at the end. In OpenCL nomenclature, these are placed in device global memory.
  - A new value is created in the target region and initialized with the original data, but it is shared between threads on the device. Data is copied back to the host at the end of the target region.
  - OpenMP calls this mapping **tofrom**

Default Data Mapping:
implicit movement with a target region

- Pointers and their data:
  - *Example: arrays allocated on the heap*
    - double *A = malloc(sizeof(double)*1000);
  - The pointer value will be mapped.
  - But the data it points to ***will not*** be mapped by default*.*

The target data environment



Scalars and statically allocated arrays are moved onto the device by default before execution

Host thread

Generating Task

float A[N], B[N];

#pragma omp target

Target task

Host thread waits for the task region to complete

A, B and N mapped to the device

Initial task

{

target region, can use A, B and N

}

Device Initial thread

the arrays A and B mapped back to the host

Only the statically allocated arrays are moved back to the host after the target region completes

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

Default Data Sharing: example

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

  #pragma omp target
  {

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

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

    }

  } // end of target region
}
```

1. Variables created in host memory.

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

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

4. Execution on the device.

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

Explicit Data Sharing

- Previously, we described the rules for *implicit* data movement.

- We *explicitly* control the movement of data using the **map** clause.

- Data allocated on the heap needs to explicitly copied to/from the device:

```
int main(void) {
   int  ii=0, N = 1024;
   int* A = malloc(sizeof(int)*N);

   #pragma omp target
   {
     // N, ii and A all exist here
     // The data that A points to (*A , A[ii]) DOES NOT exist here!
   }
}
```

Controlling data movement

Data movement defined from the *host* perspective.

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

- The various forms of the map clause
  - **map(to:list)**: On entering the region, variables in the list are initialized on the device using the original values from the host (host to device copy).
  - **map(from:list)**: At the end of the target region, the values from variables in the list are copied into the original variables (device to host copy). On entering the region, initial value of the variable is not initialized.
  - **map(tofrom:list)**: the effect of both a map-to and a map-from (host to device copy at start of region, device to host copy at end)
  - **map(alloc:list)**: On entering the region, data is allocated and uninitialized on the device.
  - **map(list)**: equivalent to **map(tofrom:list)**.
- For pointers you must use array section notation ..
  - **map(to:a[0:N]).** Notation is **A[lower-bound : length]**

Moving arrays with the map clause

```
int main(void) {
   int  N = 1024;
   int* A = malloc(sizeof(int)*N);

   #pragma omp target map(A[0:N])
   {
     // N, ii and A all exist here
     // The data that A points to DOES exist here!
   }
}
```

Default mapping
map(tofrom: A[0:N])

Copy at start and end of **target** region.

teams and distribute constructs

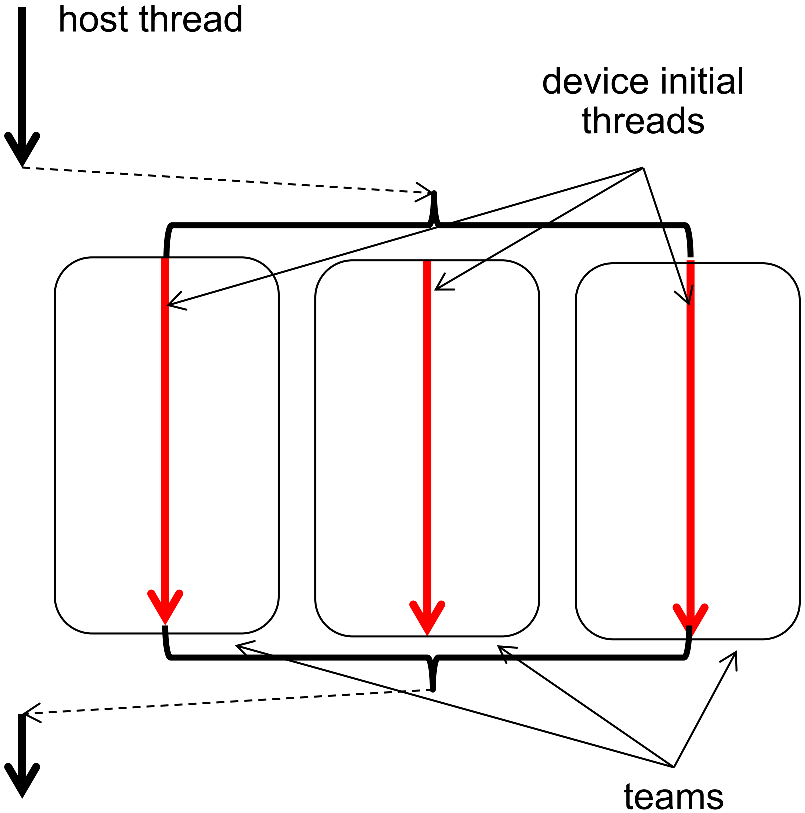
- The **teams** construct
  - Similar to the **parallel** construct
  - It starts a league of thread teams
  - Each team in the league starts as one initial thread – a team of one
  - Threads in different teams cannot synchronize with each other
  - The construct must be "perfectly" nested in a **target** construct

- The **distribute** construct
  - Similar to the **for** construct
  - Loop iterations are workshared across the initial threads in a league
  - No implicit barrier at the end of the construct
  - **dist_schedule(*kind[, chunk_size]*)**
    - If specified, scheduling kind must be static
    - Chunks are distributed in round-robin fashion in chunks of size ***chunk_size***
    - If no chunk size specified, chunks are of (almost) equal size; each team receives at least one chunk

Accelerated workshare v2.0

- teams construct
- distribute construct

```
#pragma omp target
#pragma omp teams
#pragma omp distribute
 for (i=0;i<N;i++)
    …
```

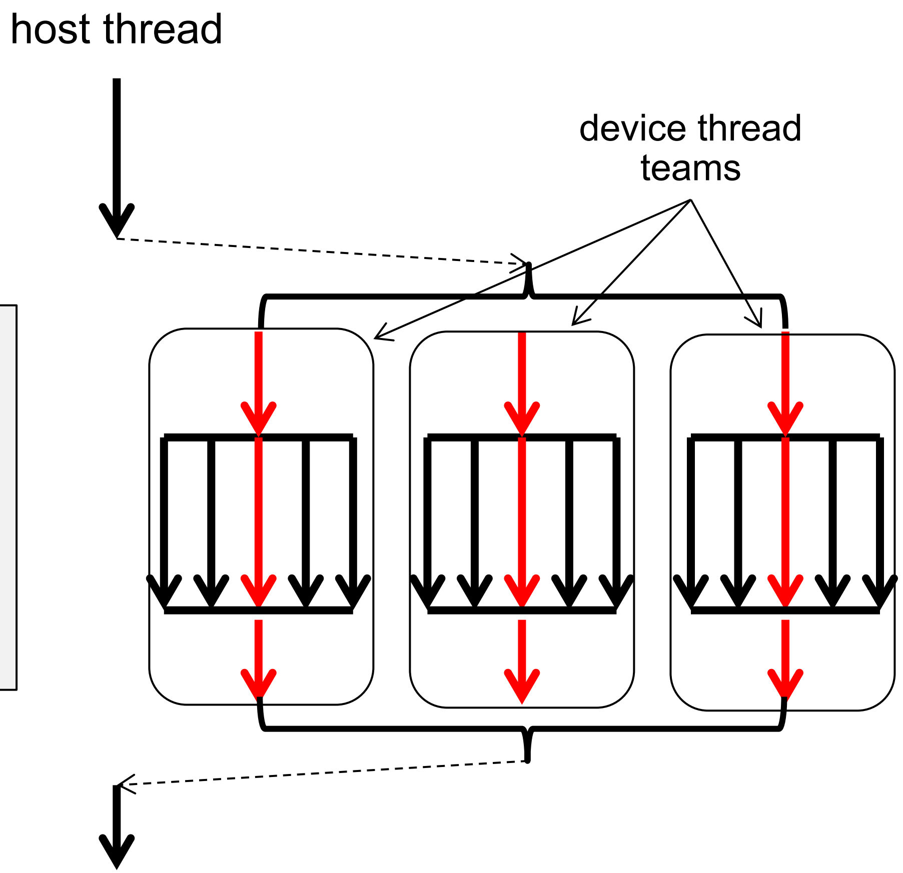


- Transfer execution control to MULTIPLE device initial threads
- Workshare loop iterations across the initial threads.

Accelerate workshare v3.0

- teams distribute
- parallel for simd

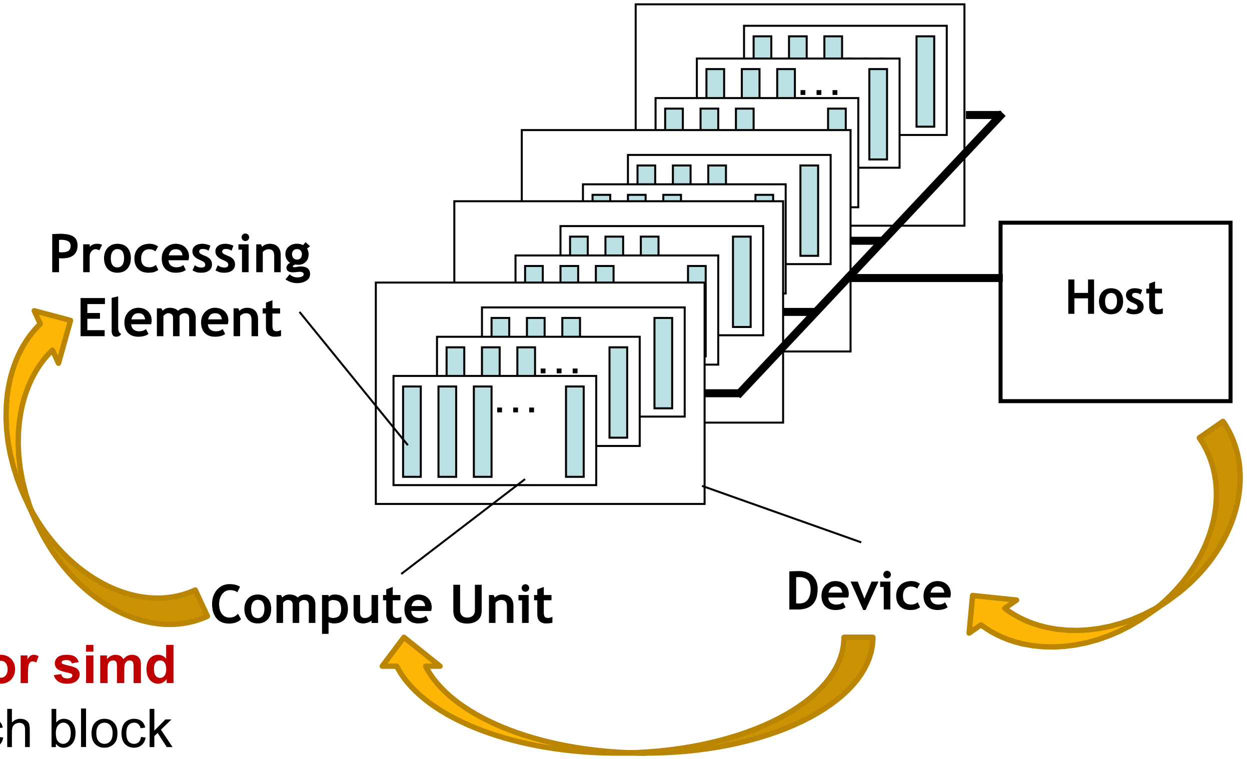
```
#pragma omp target
#pragma omp teams distribute
 for (i=0;i<N;i++)
 #pragma omp parallel for simd
 for (j=0;j<M;i++)
    …
```



- Transfer execution control to MULTIPLE device initial threads
  - Workshare loop iterations across the initial threads (teams distribute)
- Each initial thread becomes the primary* thread in a thread team
  - Workshare loop iterations across the threads in a team (parallel for)

*the term “master” has been deprecated in OpenMP 5.1 and replaced with the term “primary”.

Our host/device Platform Model and OpenMP



Target construct to get onto a device

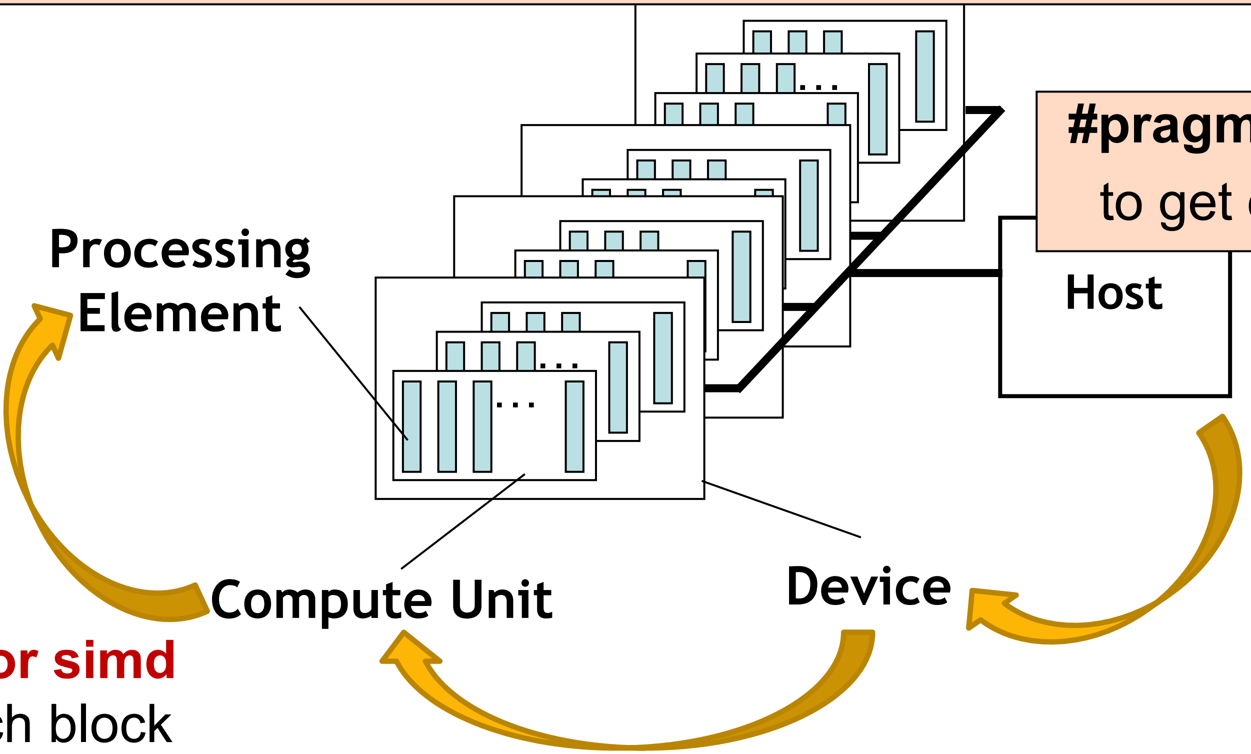
Teams construct to create a league of teams with one team of threads on each compute unit.

Parallel for simd to run each block of loop iterations on the processing elements

Distribute construct to assign blocks of loop iterations to teams.

Our host/device Platform Model and OpenMP

Typical usage ... let the compiler do what’s best for the device:



#pragma omp target

to get on the device

Target construct to get onto a device

Parallel for simd to run each block of loop iterations on the pr

elem

Teams construct to create a

#pragma omp teams distribute parallel for simd

to assign work to the device processing elements

Distribute construct to assign blocks of loop iterations to teams.

Our running example: Jacobi solver

- An iterative method to solve a system of linear equations
  - Given a matrix A and a vector b find the vector x such that $Ax=b$
- The basic algorithm:
  - Write A as a lower triangular (L), upper triangular (U) and diagonal matrix

    $$Ax = (L+D+U)x = b$$

  - Carry out multiplications and rearrange

    $$Dx=b-(L+U)x \quad \rightarrow \quad x = (b-(L+U)x)/D$$

  - Iteratively compute a new x using the x from the previous iteration

    $$X_{new} = (b-(L+U)x_{old})/D$$

- Advantage: we can easily test if the answer is correct by multiplying our final x by A and comparing to b
- Disadvantage: It takes many iterations and only works for diagonally dominant matrices

Jacobi Solver

Iteratively update xnew until the value stabilizes (i.e. change less than a preset TOL)

```
<<< allocate and initialize the matrix A >>>
<<< and vectors x1, x2 and b             >>>

while((conv > TOL) && (iters<MAX_ITERS))
  {
    iters++;

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

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

       // swap pointers for next
       // iteration
       TYPE* tmp = xold;
       xold = xnew;
       xnew = tmp;

    } // end while loop
```

Jacobi Solver (Par Targ, 1/2)

```
 while((conv > TOL) && (iters<MAX_ITERS))
  {
    iters++;
#pragma omp target map(tofrom:xnew[0:Ndim],xold[0:Ndim]) \
                        map(to:A[0:Ndim*Ndim], b[0:Ndim])
#pragma omp teams distribute parallel for simd private(i,j)
for (i=0; i<Ndim; i++){
        xnew[i] = (TYPE) 0.0;
        for (j=0; j<Ndim;j++){
            if(i!=j)
              xnew[i]+= A[i*Ndim + j]*xold[j];
        }
        xnew[i] = (b[i]-xnew[i])/A[i*Ndim+i];
    }
```

Jacobi Solver (Par Targ, 2/2)

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

This worked but the performance was awful. Why?

System	Implementation	Ndim = 4096
NVIDA® K20X™ GPU	Target dir per loop	131.94 secs

Cray® XC40™ Supercomputer running Cray® Compiling Environment 8.5.3.
Intel® Xeon ® CPU E5-2697 v2 @ 2.70GHz with 32 GB DDR3. NVIDIA® Tesla® K20X, 6GB.

Data movement dominates!!!

```
while((conv > TOLERANCE) && (iters<MAX_ITERS))
  { iters++;
    xnew = iters % s ? x2 : x1;
    xold   = iters % s ? x1 : x2;

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

Typically over 4000 iterations!

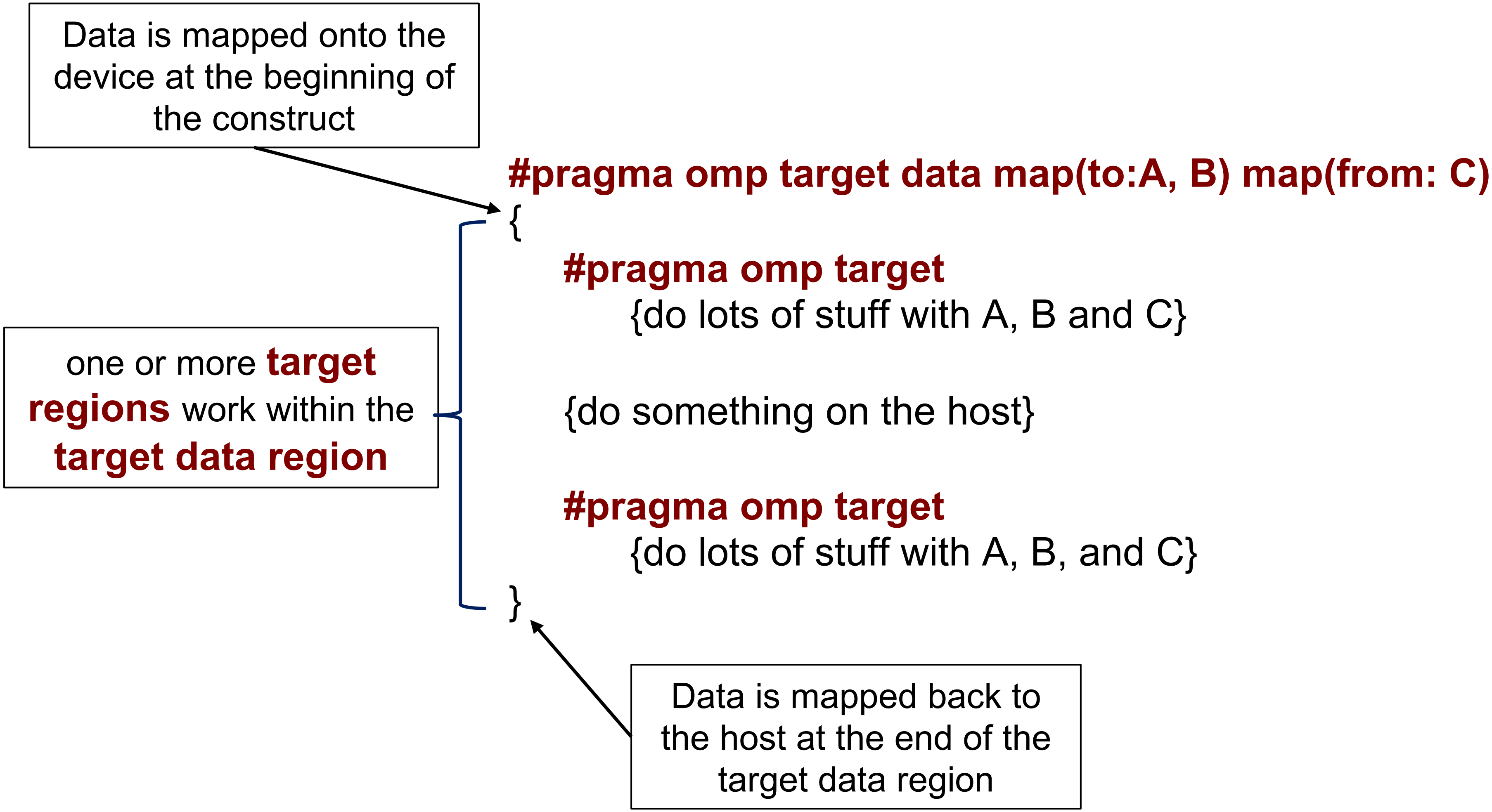
For each iteration, copy to device
(3*Ndim+Ndim^2)*sizeof(TYPE) bytes

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

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

Target data directive

- The **target data** construct creates a target data region … use **map** clauses for explicit data management



Data is mapped onto the device at the beginning of the construct

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

    {do something on the host}

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

one or more **target regions** work within the **target data region**

Data is mapped back to the host at the end of the target data region

Jacobi Solver (Par Targ Data, 1/2)

```
 #pragma omp target data map(tofrom:x1[0:Ndim],x2[0:Ndim]) \
                      map(to:A[0:Ndim*Ndim], b[0:Ndim] ,Ndim)
while((conv > TOL) && (iters<MAX_ITERS))
  {  iters++;

#pragma omp target
#pragma omp teams distribute parallel for simd private(j) firstprivate(xnew,xold)
     for (i=0; i<Ndim; i++){
        xnew[i] = (TYPE) 0.0;
        for (j=0; j<Ndim;j++){
           if(i!=j)
             xnew[i]+= A[i*Ndim + j]*xold[j];
        }
        xnew[i] = (b[i]-xnew[i])/A[i*Ndim+i];
     }
```

Jacobi Solver (Par Targ Data, 2/2)

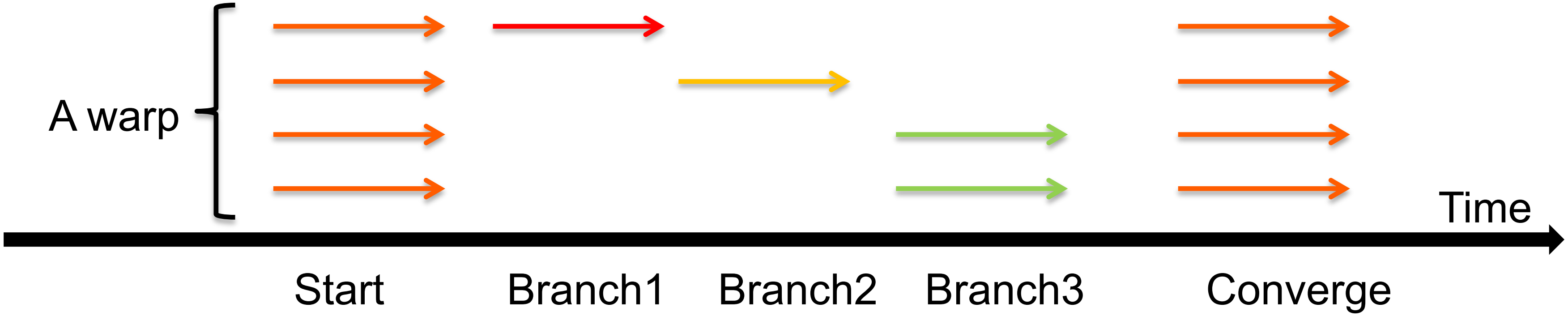
```
// test convergence
conv = 0.0;
#pragma omp target map(tofrom: conv)
#pragma omp teams distribute parallel for simd  \
                    private(tmp) firstprivate(xnew,xold)  reduction(+:conv)
   for (i=0; i<Ndim; i++){
      tmp  = xnew[i]-xold[i];
      conv += tmp*tmp;
   }
// end target region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

   TYPE* tmp = xold;
   xold = xnew;
   xnew = tmp;
} // end while loop
```

System	Implementation	Ndim = 4096
NVIDA® K20X™ GPU	Target dir per loop	131.94 secs
	Above plus target data region	18.37 secs

Third party names are the property of their owners.

Single Instruction Multiple Data

- Individual work-items of a warp start together at the same program address
- Each work-item has its own instruction address counter and register state
  - Each work-item is free to branch and execute independently
  - Supports the SPMD pattern.
- Branch behavior
  - Each branch will be executed serially
  - Work-items not following the current branch will be disabled



Branching

Conditional execution

```
// Only evaluate expression
// if condition is met
if (a > b)
{
  acc += (a - b*c);
}
```

Selection and masking

```
// Always evaluate expression
// and mask result
temp = (a - b*c);
mask = (a > b ? 1.f : 0.f);
acc += (mask * temp);
```

Coalescence

- Coalesce - to combine into one
- Coalesced memory accesses are key for high bandwidth
- Simply, it means, if thread *i* accesses memory location *n* then thread *i+1* accesses memory location *n+1*
- In practice, it’s not quite as strict…

```
for (int id = 0; id < size; id++)
{
  // ideal
    float val1 = memA[id];

  // still pretty good
    const int c = 3;
    float val2 = memA[id + c];

  // stride size is not so good
    float val3 = memA[c*id];

  // terrible
    const int loc =
      some_strange_func(id);

    float val4 = memA[loc];
}
```

Jacobi Solver (Targ Data/branchless/coalesced mem, 1/2)

```
  #pragma omp target data map(tofrom:x1[0:Ndim],x2[0:Ndim]) \
                     map(to:A[0:Ndim*Ndim], b[0:Ndim] ,Ndim)
while((conv > TOL) && (iters<MAX_ITERS))
  {  iters++;
#pragma omp target
     #pragma omp teams distribute parallel for simd private(j)
    for (i=0; i<Ndim; i++){
       xnew[i] = (TYPE) 0.0;
       for (j=0; j<Ndim;j++){
           xnew[i]+= (A[j*Ndim + i]*xold[j])*((TYPE)(i != j));
       }
       xnew[i] = (b[i]-xnew[i])/A[i*Ndim+i];
    }
```

We replaced the original code with a poor memory access pattern

    xnew[i]+= (A[i*Ndim + j]*xold[j])

With the more efficient

    xnew[i]+= (A[j*Ndim + i]*xold[j])

Jacobi Solver (Targ Data/branchless/coalesced mem, 2/2)

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

System	Implementation	Ndim = 4096
NVIDA® K20X™ GPU	Target dir per loop	131.94 secs
	Above plus target data region	18.37 secs
	Above plus reduced branching	13.74 secs
	Above plus improved mem access	7.64 secs

Cray® XC40™ Supercomputer running Cray® Compiling Environment 8.5.3.
Intel® Xeon ® CPU E5-2697 v2 @ 2.70GHz with 32 GB DDR3. NVIDIA® Tesla® K20X, 6GB.

Third party names are the property of their owners.

Appendices

- Challenge Problems
- Challenge Problems: solutions
  - 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: Molecular dynamics

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

Challenge 1: MD (cont.)

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

Challenge 1 MD: (cont.)

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

Challenge 2: Matrix multiplication

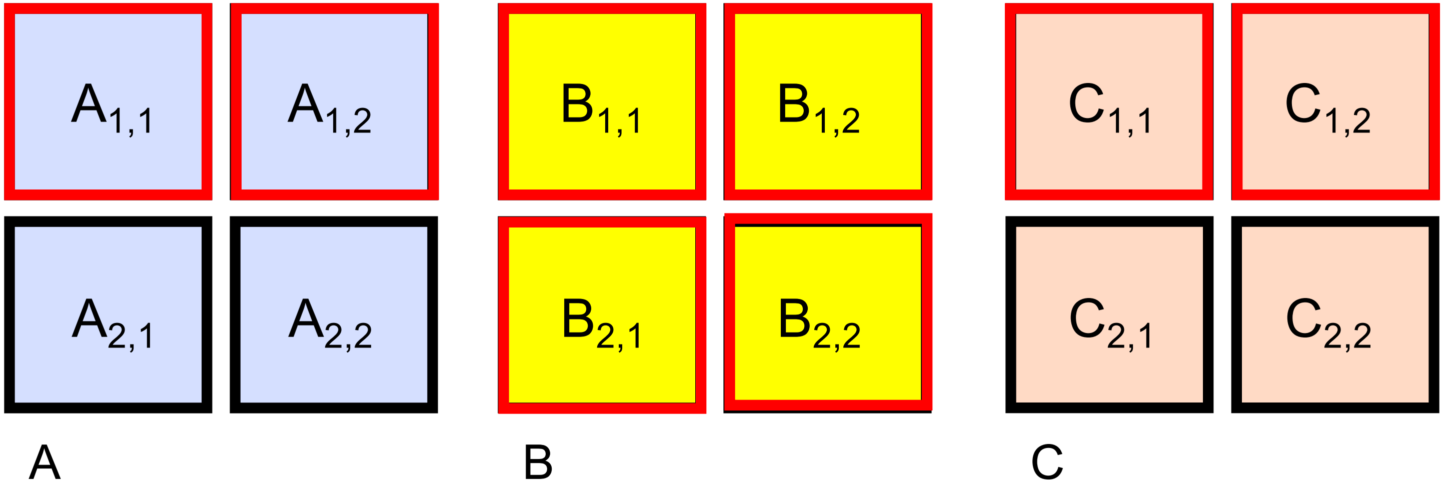
- Parallelize the matrix multiplication program in the file mm_testbed.c
- Can you optimize the program by playing with how the loops are scheduled?
- Try the following and see how they interact with the constructs in OpenMP
  - Alignment
  - Cache blocking
  - Loop unrolling
  - Vectorization
- Goal: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

Challenge 3: 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 3: Recursive matrix multiplication

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



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

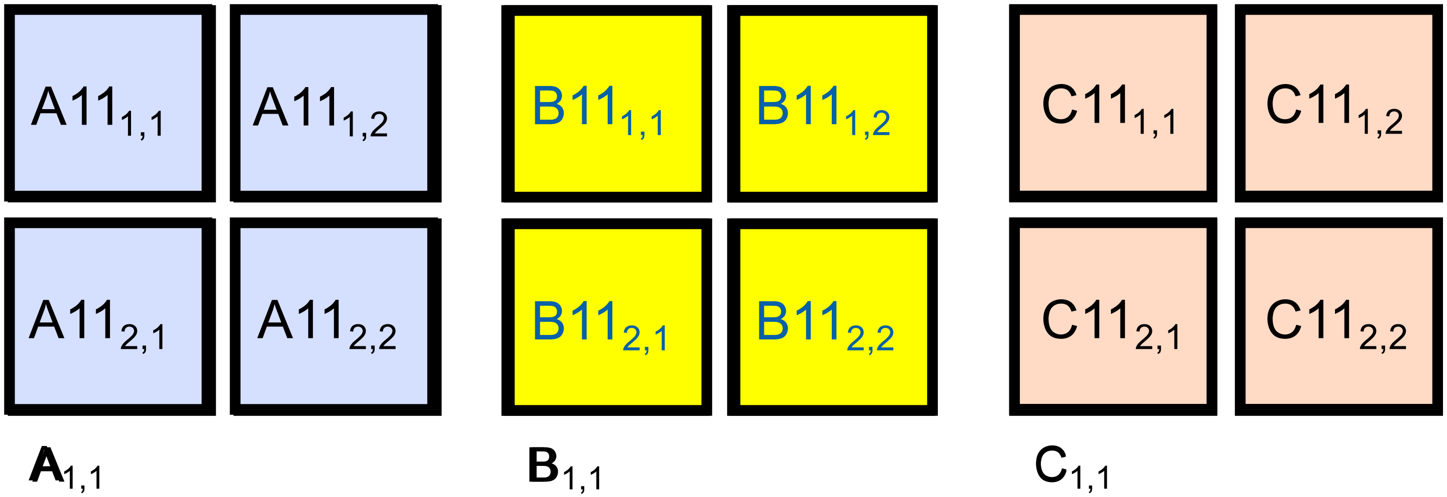
$$C_{2,1} = A_{2,1} \cdot B_{1,1} + A_{2,2} \cdot B_{2,1}$$

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

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

Challenge 3: 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



$A11_{1,1}$ $A11_{1,2}$
$A11_{2,1}$ $A11_{2,2}$

$A_{1,1}$

$B11_{1,1}$ $B11_{1,2}$
$B11_{2,1}$ $B11_{2,2}$

$B_{1,1}$

$C11_{1,1}$ $C11_{1,2}$
$C11_{2,1}$ $C11_{2,2}$

$C_{1,1}$

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

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

Challenge 3: Recursive matrix multiplication
Recursively multiply submatrices

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

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

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

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

- 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
  - 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
  - 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
  - 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

   }
}
```

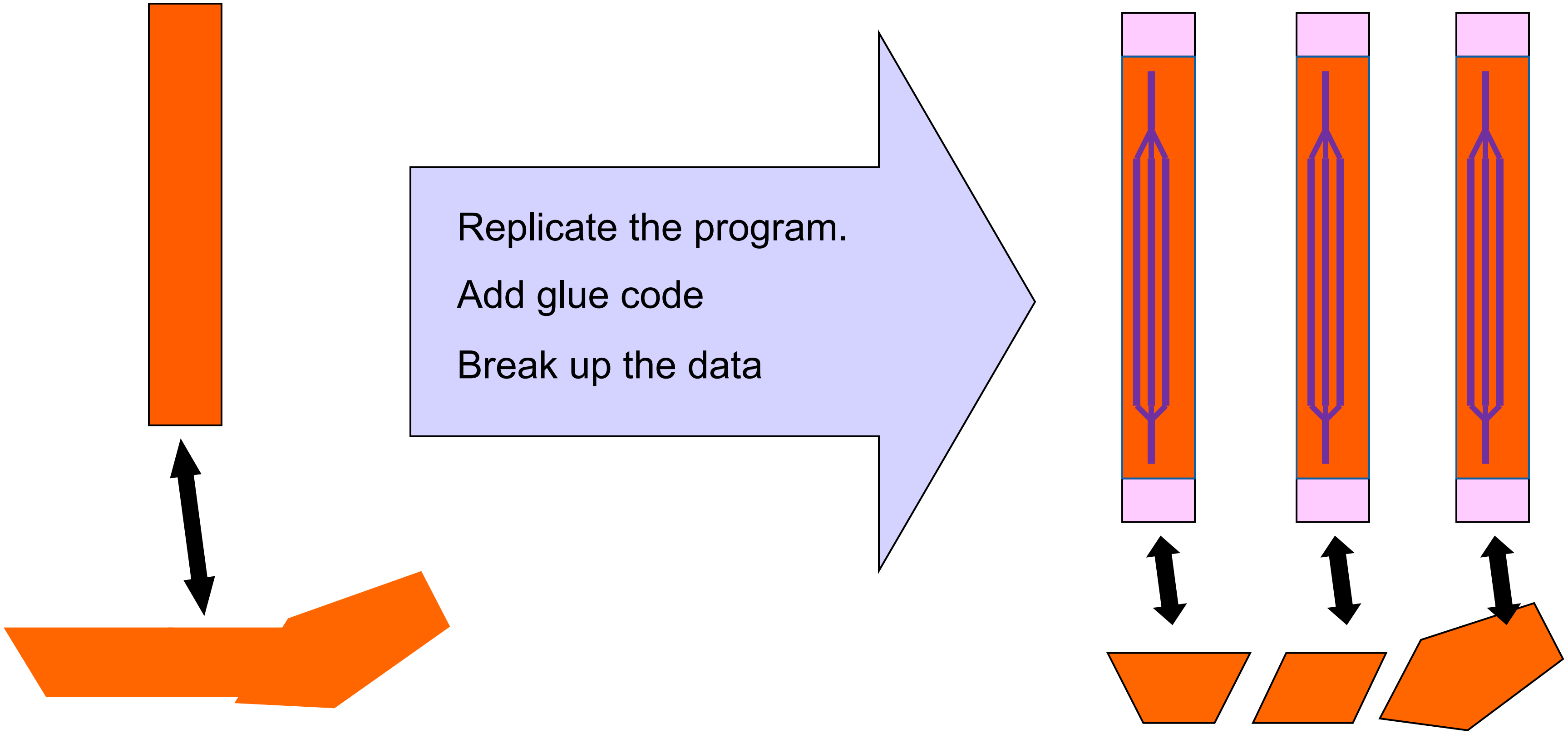
Appendices

- Challenge Problems
- Challenge Problems: solutions
  - 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

- Create the MPI program with its data decomposition.
- Use OpenMP inside each MPI process.



Replicate the program.

Add glue code

Break up the data

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 primary* 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.

*the term "master" has been deprecated in OpenMP 5.1 and replaced with the term "primary".

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 primary*) (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.

*the term “master” has been deprecated in OpenMP 5.1 and replaced with the term “primary”.

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.

*the master constructs was deprecated in OpenMP 5.1 to be replaced with the masked construct

Safe Mixing of MPI and OpenMP

Protect MPI calls inside a parallel region

Using the new masked directive in place of master

```
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 masked filter(0)
{
     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 masked filter(0)
   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.

*the term “master” has been deprecated in OpenMP 5.1 and replaced with the term “primary”.

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
  - 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
  - MASKED … END MASKED

*the master construct was deprecated in OpenMP 5.1 and replaced with masked construct

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

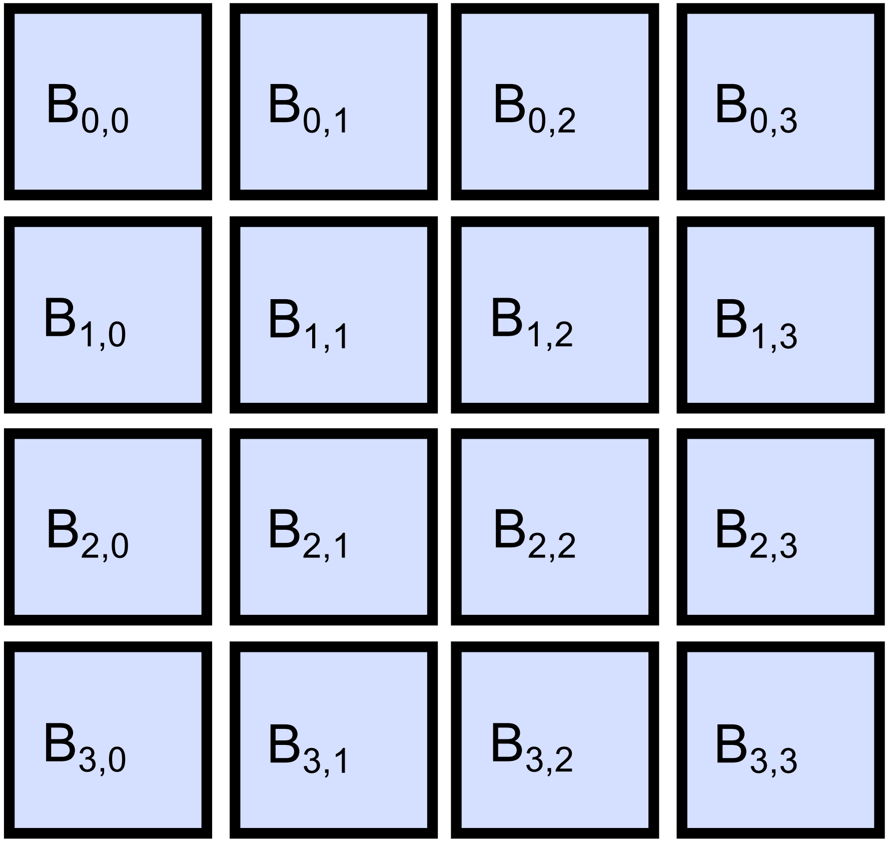
- Challenge Problems
- Challenge Problems: solutions
  - Molecular dynamics
  - Matrix multiplication
  - Linked lists
  - Recursive matrix multiplication
- Mixing OpenMP and MPI
- Fortran and OpenMP
- ➡ Details on the cache oblivious LU example

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

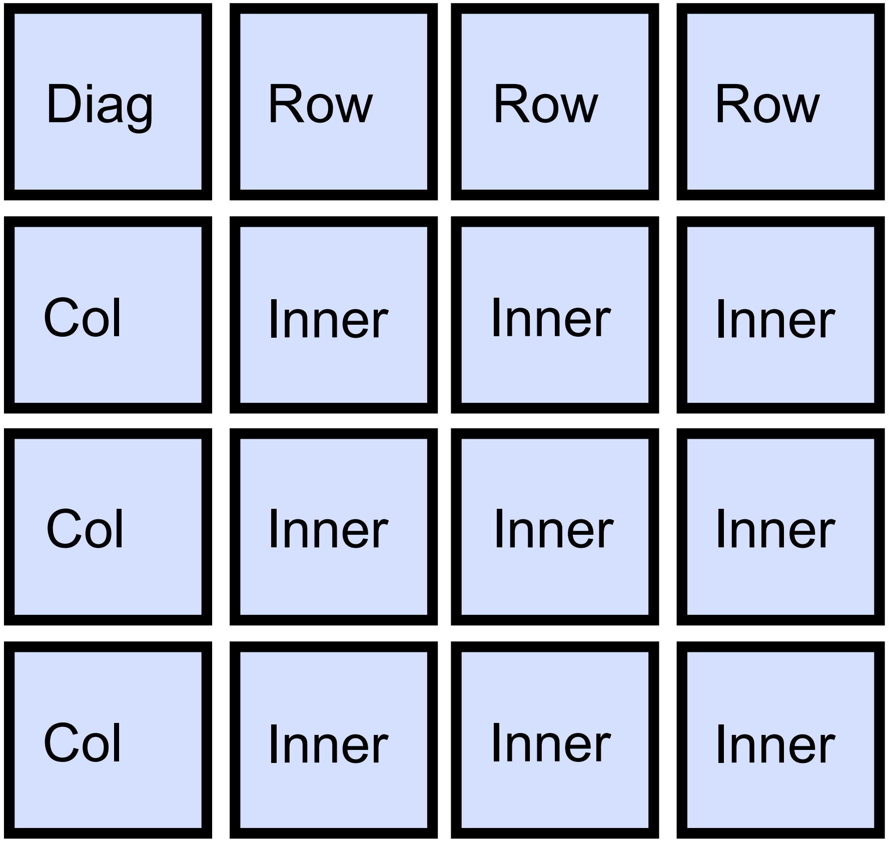
- To start with an example, take a matrix divided into 4x4 blocks



$B_{0,0}$	$B_{0,1}$	$B_{0,2}$	$B_{0,3}$
$B_{1,0}$	$B_{1,1}$	$B_{1,2}$	$B_{1,3}$
$B_{2,0}$	$B_{2,1}$	$B_{2,2}$	$B_{2,3}$
$B_{3,0}$	$B_{3,1}$	$B_{3,2}$	$B_{3,3}$

LU Decomposition
Recursive Cache Oblivious Algorithm

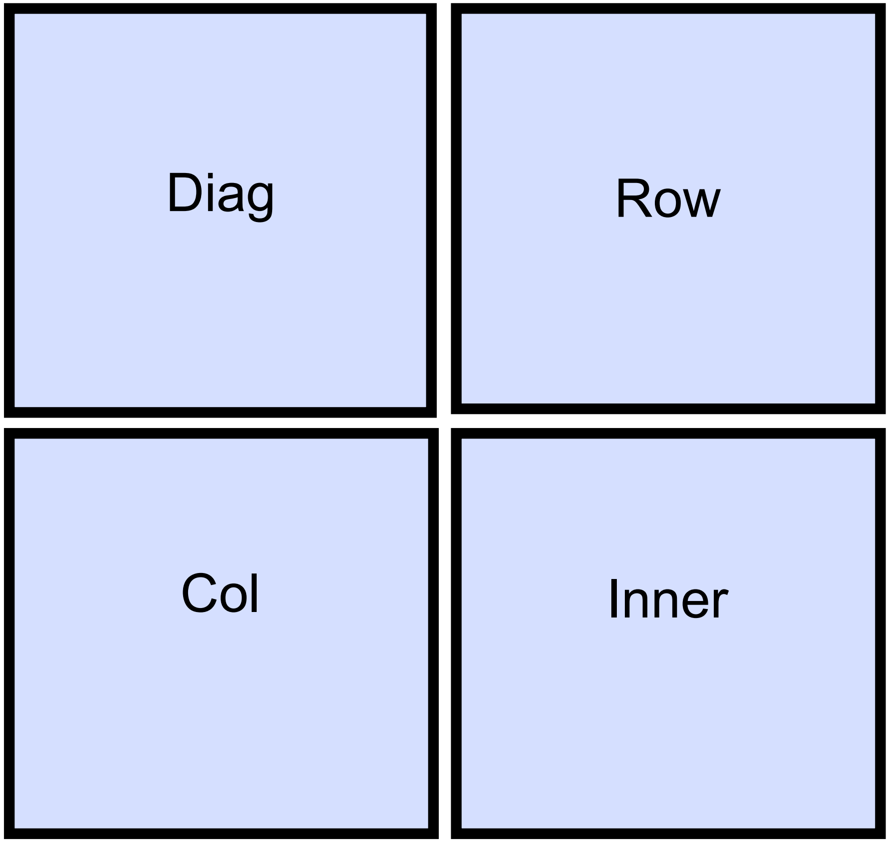
- The first version would go through the first iteration and create tasks for these blocks, then move on to the next iteration.



Diag	Row	Row	Row
Col	Inner	Inner	Inner
Col	Inner	Inner	Inner
Col	Inner	Inner	Inner

LU Decomposition
Recursive Cache Oblivious Algorithm

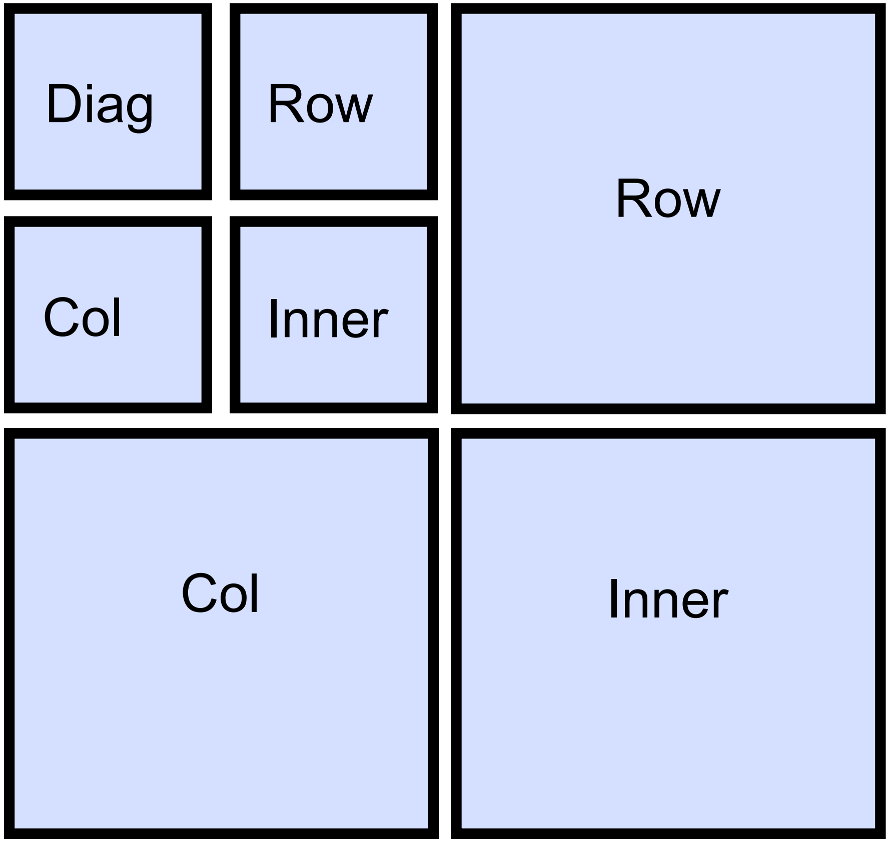
- The recursive version starts by calling Diag to divide the whole matrix into quadrants.
- Each of these quadrants is processed, and then Diag is called again on the output of Inner, which handles the second half of iterations.



Diag
Row
Col
Inner

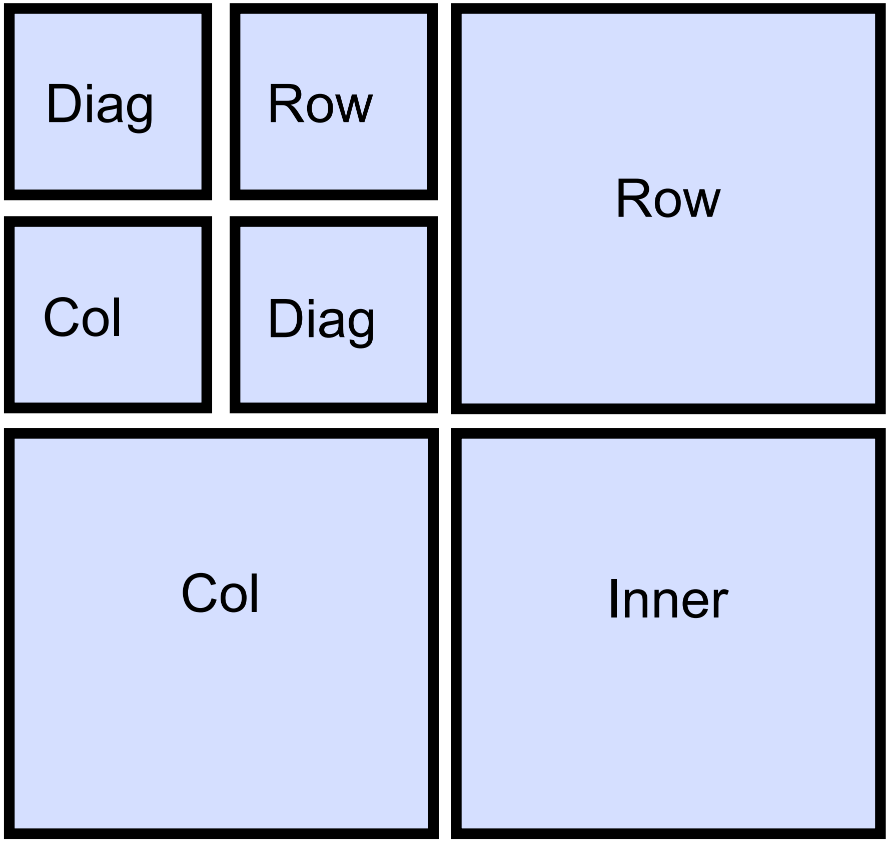
LU Decomposition
Recursive Cache Oblivious Algorithm

- Within diag, the blocks are processed as shown.



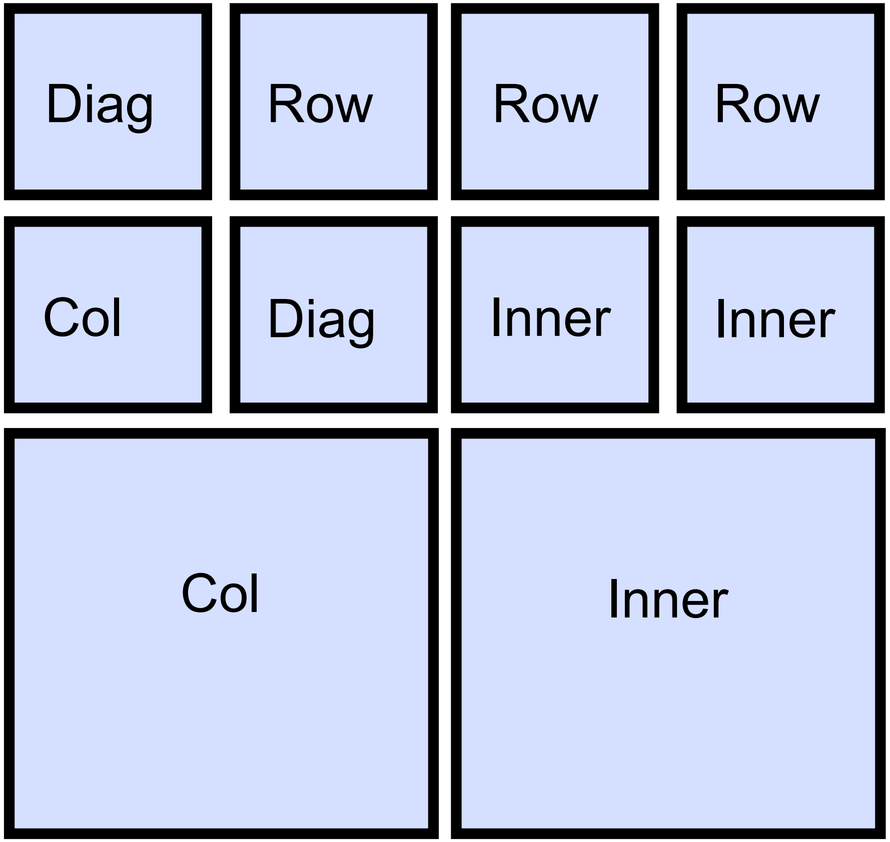
LU Decomposition
Recursive Cache Oblivious Algorithm

- 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.



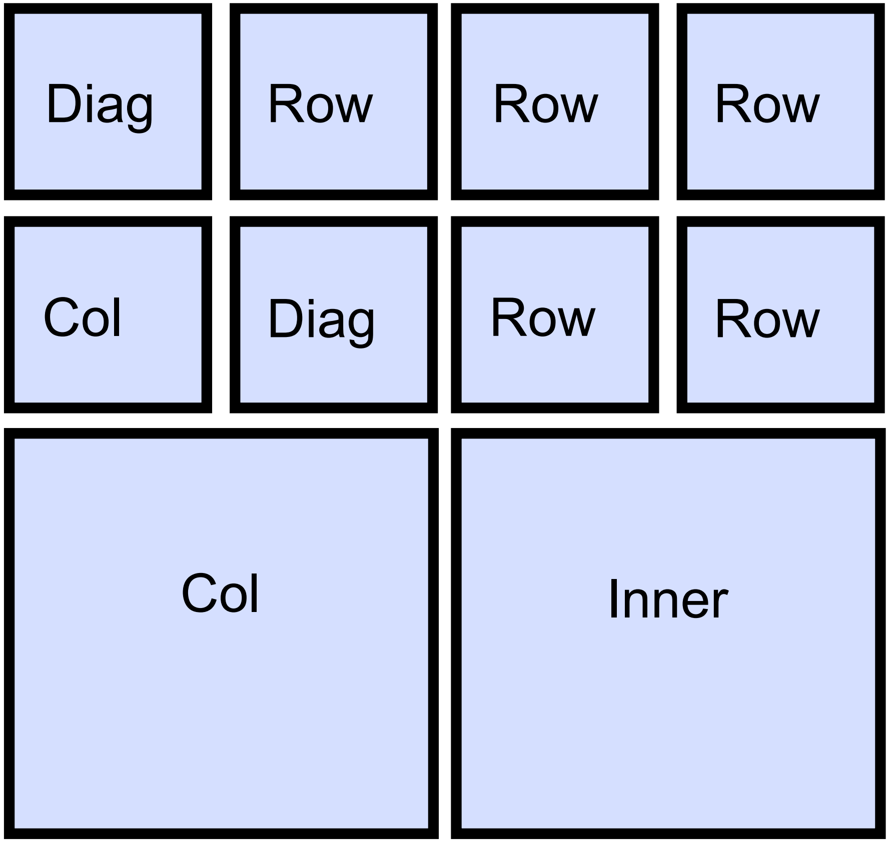
Diag Row Row Row

Col Diag Inner Inner

Col Inner

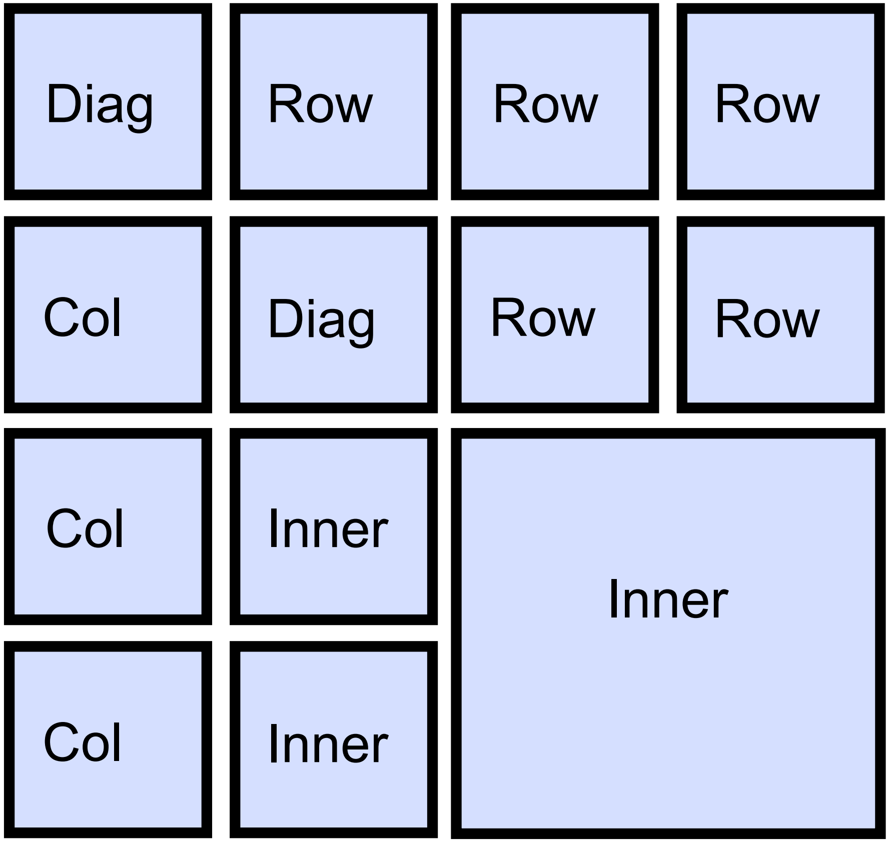
LU Decomposition
Recursive Cache Oblivious Algorithm

- 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.



Diag Row Row Row

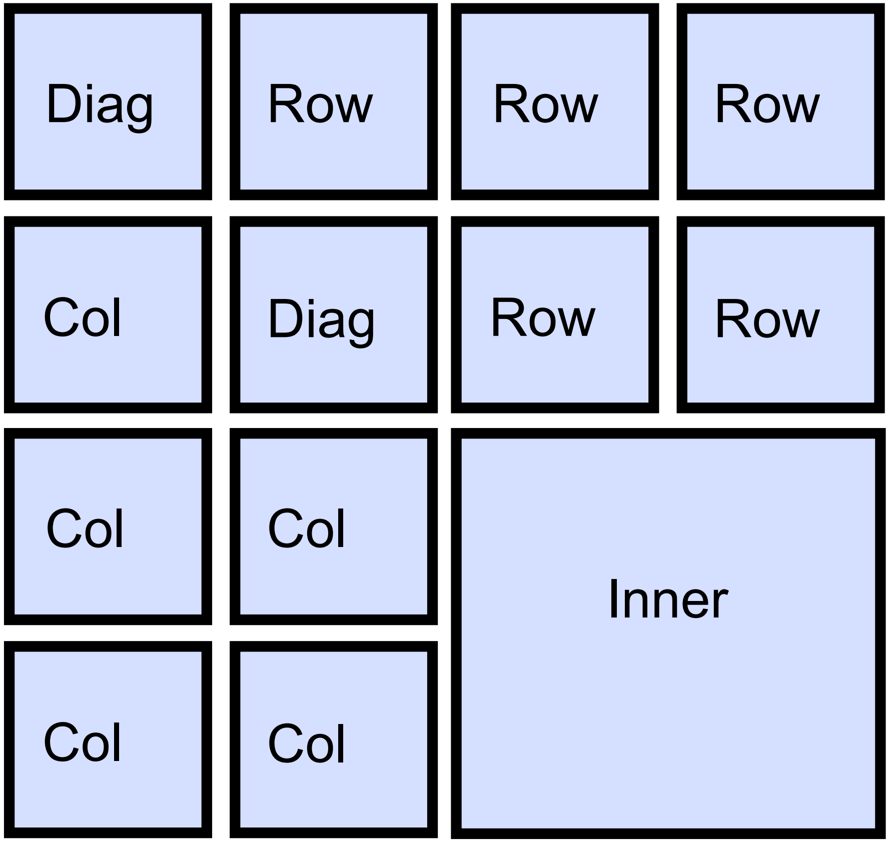
Col Diag Row Row

Col Inner Inner

Col Inner

LU Decomposition
Recursive Cache Oblivious Algorithm

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



Diag Row Row Row

Col Diag Row Row

Col Col Inner

Col Col

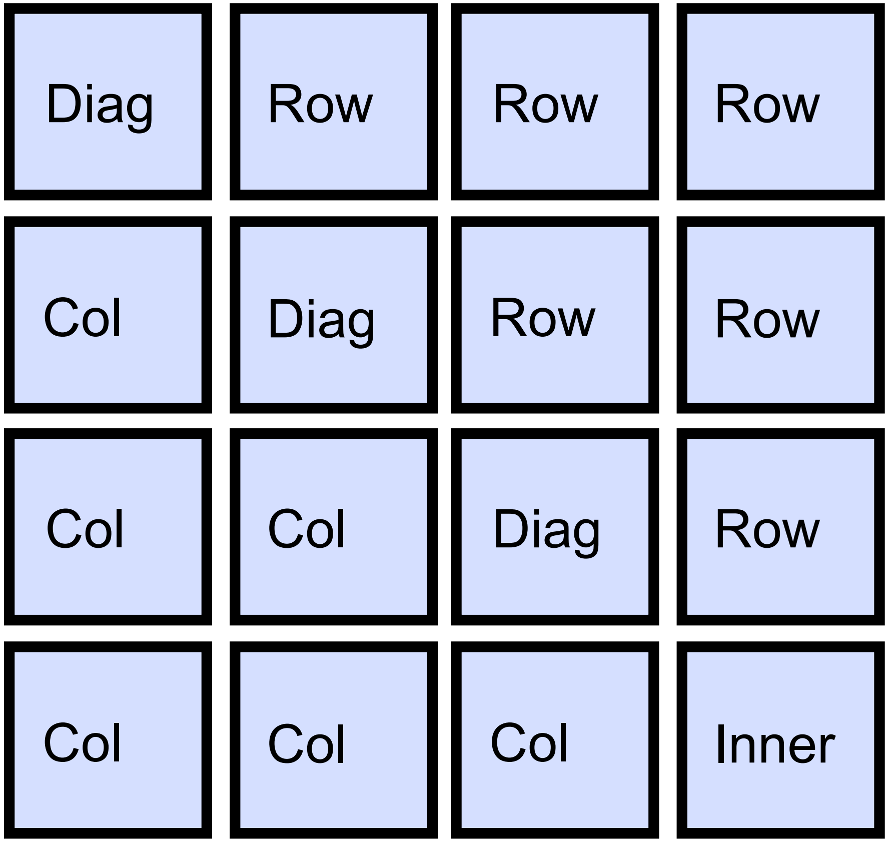
LU Decomposition
Recursive Cache Oblivious Algorithm

- 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.

Diag	Row	Row	Row
Col	Diag	Row	Row
Col	Col	Inner	Inner
Col	Col	Inner	Inner

LU Decomposition
Recursive Cache Oblivious Algorithm

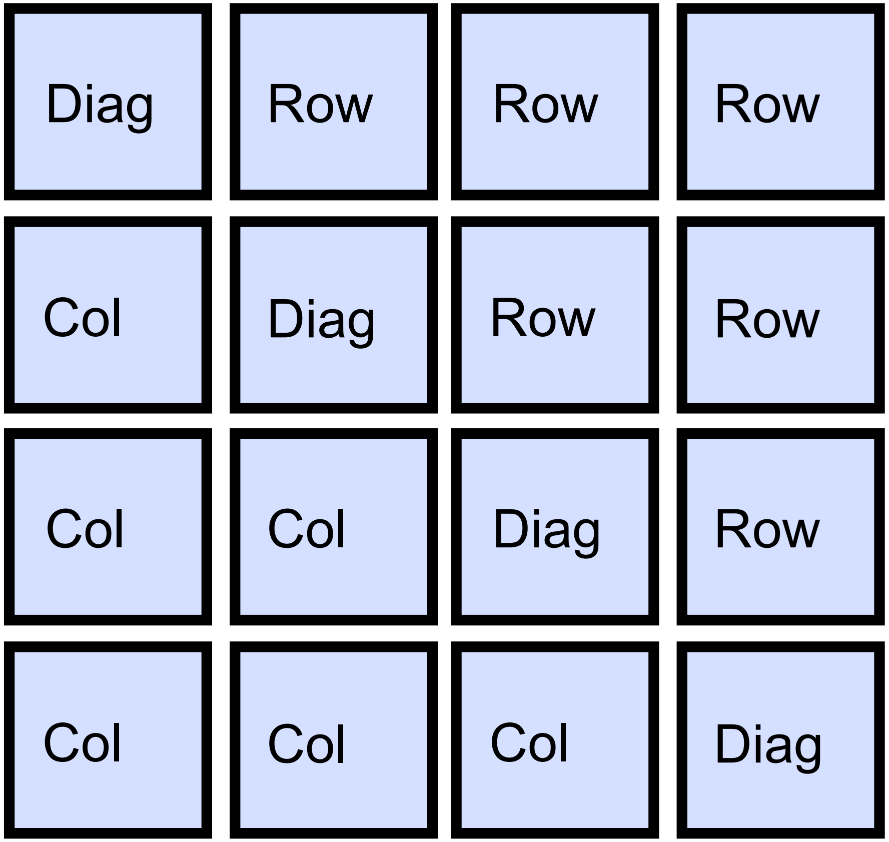
- Now the Inner quadrant is done and ready to be passed to diag, and perform what would be the third iteration.



Diag	Row	Row	Row
Col	Diag	Row	Row
Col	Col	Diag	Row
Col	Col	Col	Inner

LU Decomposition
Recursive Cache Oblivious Algorithm

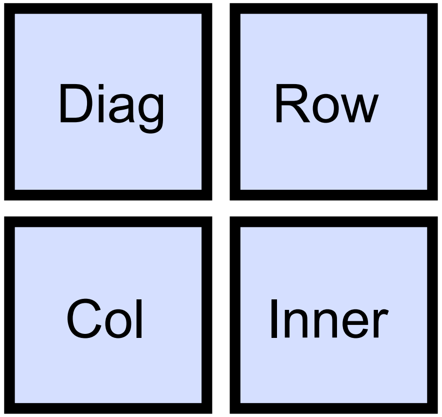
- And the final step is diag on the last block, for the fourth iteration.



Diag	Row	Row	Row
Col	Diag	Row	Row
Col	Col	Diag	Row
Col	Col	Col	Diag

LU Decomposition
Recursive Cache Oblivious Algorithm

And now code for the serial version



Diag

Row

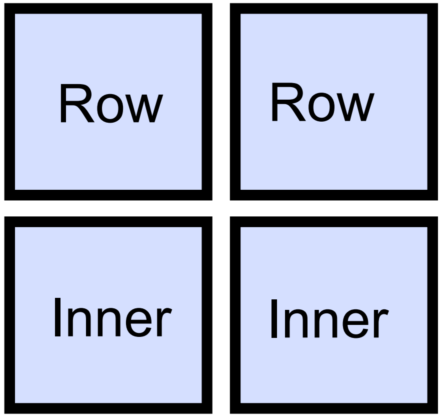
Col

Inner

```
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);
    }
}
```

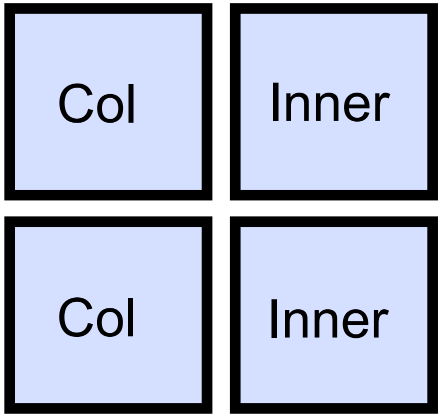


Row Row

Inner Inner

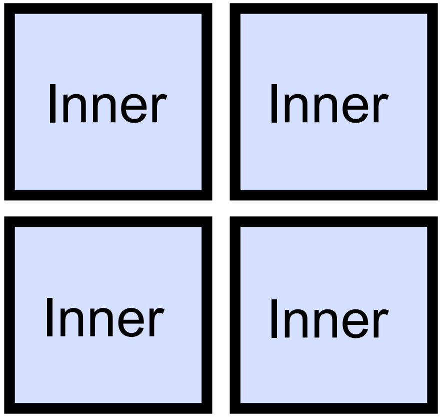
LU Decomposition
Recursive Cache Oblivious Algorithm

```
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);
    }
}
```