The OpenMP* Common Core:
A hands on exploration

Tim Mattson

Intel Corp.

timothy.g.mattson@ intel.com

Alice Koniges

Berkeley Lab

AEKoniges@lbl.gov

Yun (Helen) He

Berkeley Lab

yhe@lbl.gov

Barbara Chapman

Stony Brook University

Barbara.chapman@stonybrook.edu

1

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

Preliminaries: Part 1

- Disclosures
  - The views expressed in this tutorial are those of the people delivering the tutorial.
    - We are <u>not</u> speaking for our employers.
    - We are <u>not</u> 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!
  - <u>Don't cheat</u>: Do Not look at the solutions before you complete an exercise … even if you get really frustrated.

Outline

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

OpenMP* overview:

C$OMP FLUSH

#pragma omp critical

C$OMP THREADPRIVATE(/ABC/)

CALL OMP_SET_NUM_THREADS(10)

C$OM

C$OM

C$O

C

#p

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

ED

C$OMP PARALLEL COPYIN(/blk/)

C$OMP DO lastprivate(XX)

Nthrds = OMP_GET_NUM_PROCS()

omp_set_lock(lck)

The growth of complexity in OpenMP

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

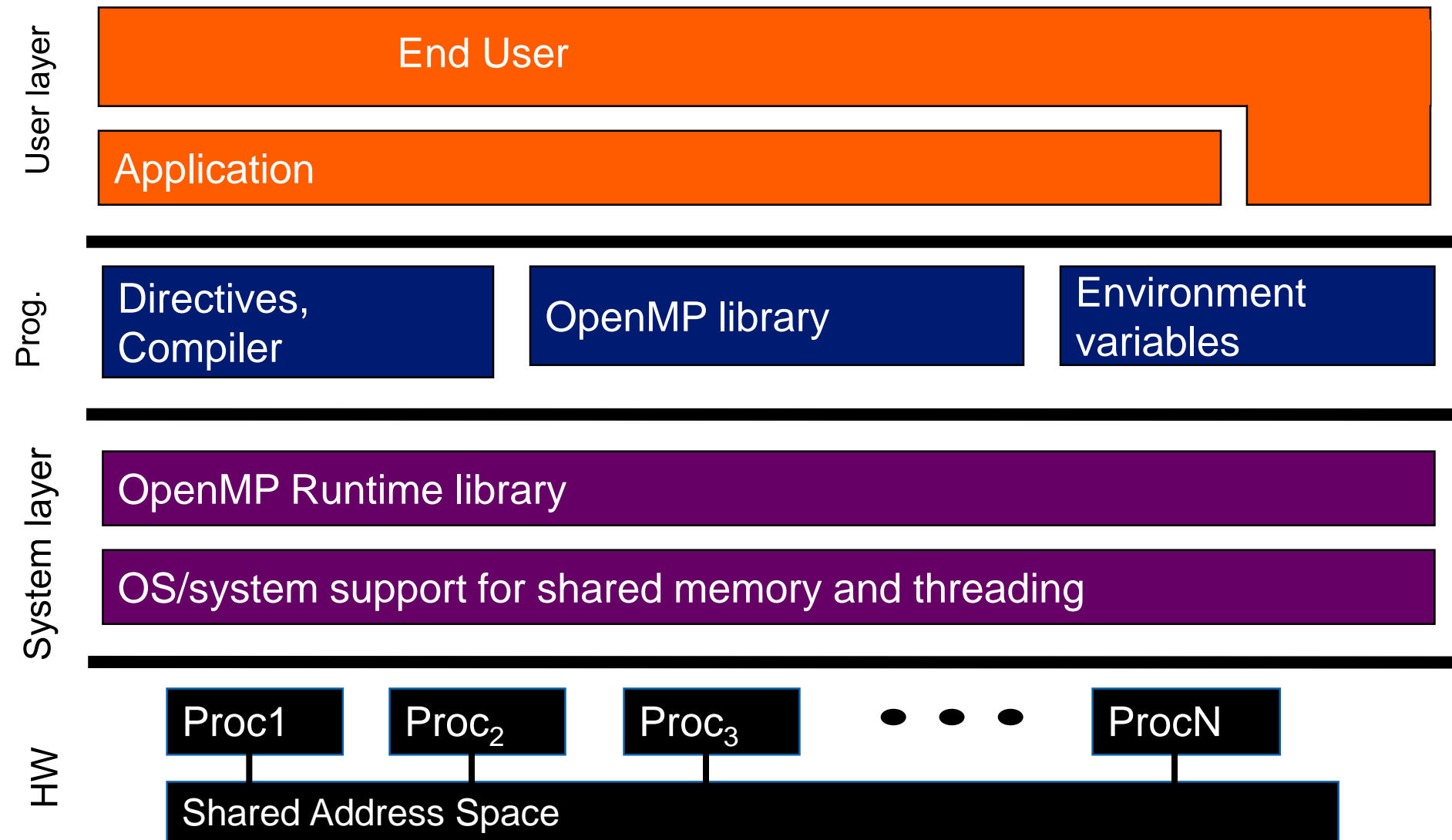- The complexity has grown considerably over the years!

Page counts (not counting front matter, appendices or index) for versions of OpenMP
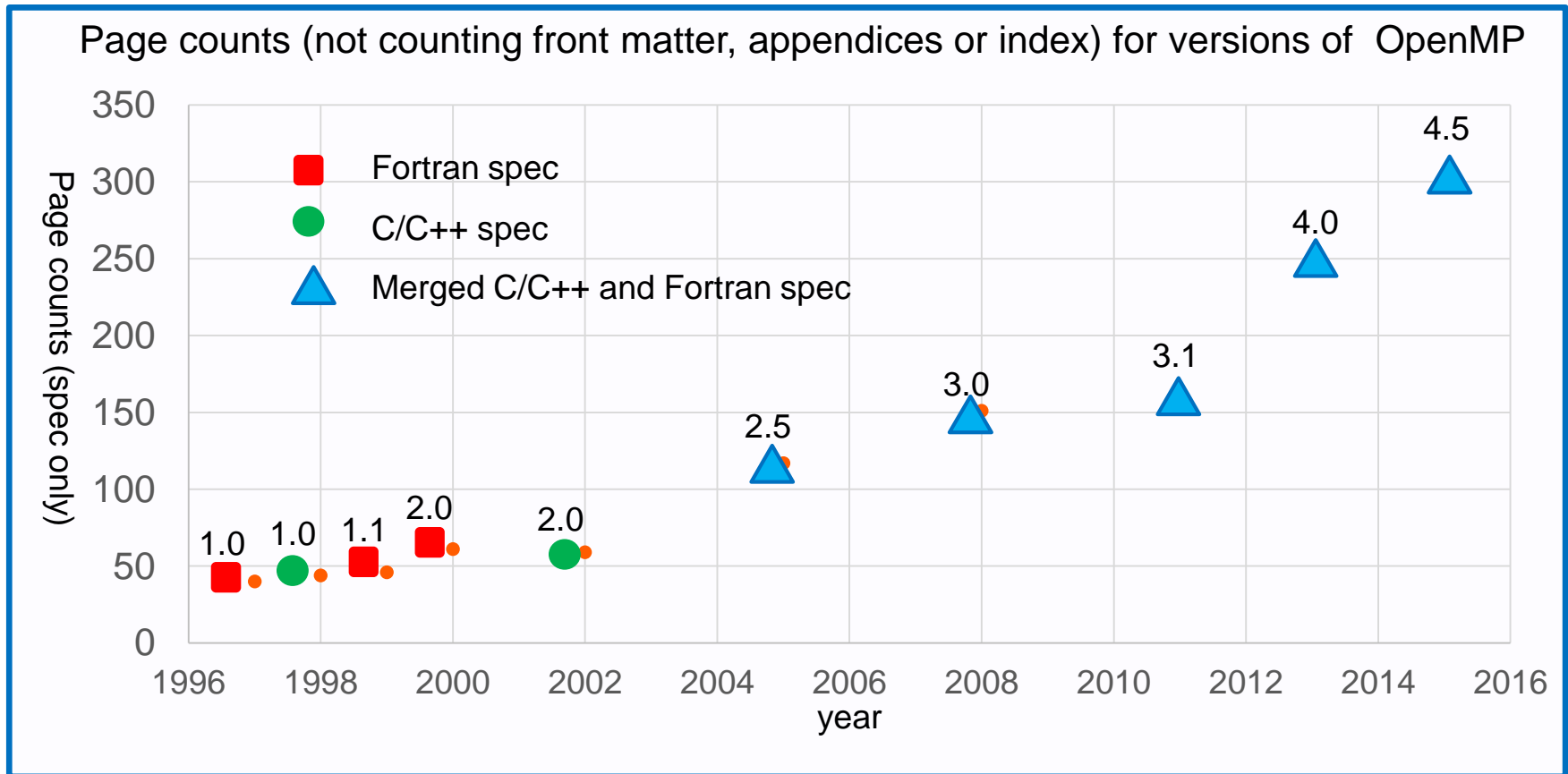
The complexity of the full spec is overwhelming, so we focus on the 16 constructs most OpenMP programmers restrict themselves to … the so called "OpenMP Common Core"

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

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

OpenMP basic definitions: Basic Solution stack

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

Proc1

$Proc_2$

$Proc_3$

• • •

ProcN

Shared Address Space

OpenMP basic syntax

- Most of the constructs in OpenMP are compiler directives.

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

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

Exercise, Part A: Hello world
Verify that your environment works

- Write a program that prints "hello world".

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

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




    printf(" hello ");
    printf(" world \n");

}
```

- For detailed NERSC instructions and to download the slides:

http://www.nersc.gov/users/software/programming-models/openmp/sc17-openmp/

Exercise, Part B: Hello world
Verify that your OpenMP environment works

- Write a multithreaded program that prints "hello world".

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

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

  #pragma omp parallel

  {



    printf(" hello ");
    printf(" world \n");

  }

}
```

Switches for compiling and linking

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

- For detailed NERSC instructions and to download the slides:

http://www.nersc.gov/users/software/programming-models/openmp/sc17-openmp/

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: More than you ever wanted to know
  - Thread private data
  - Thread affinity and data locality

OpenMP programming model:

Fork-Join Parallelism:

◆ Master thread spawns a team of threads as needed.

◆ Parallelism added incrementally until performance goals are met, i.e., the sequential program evolves into a parallel program.

Parallel Regions

Master Thread in red

A Nested Parallel region

Sequential Parts
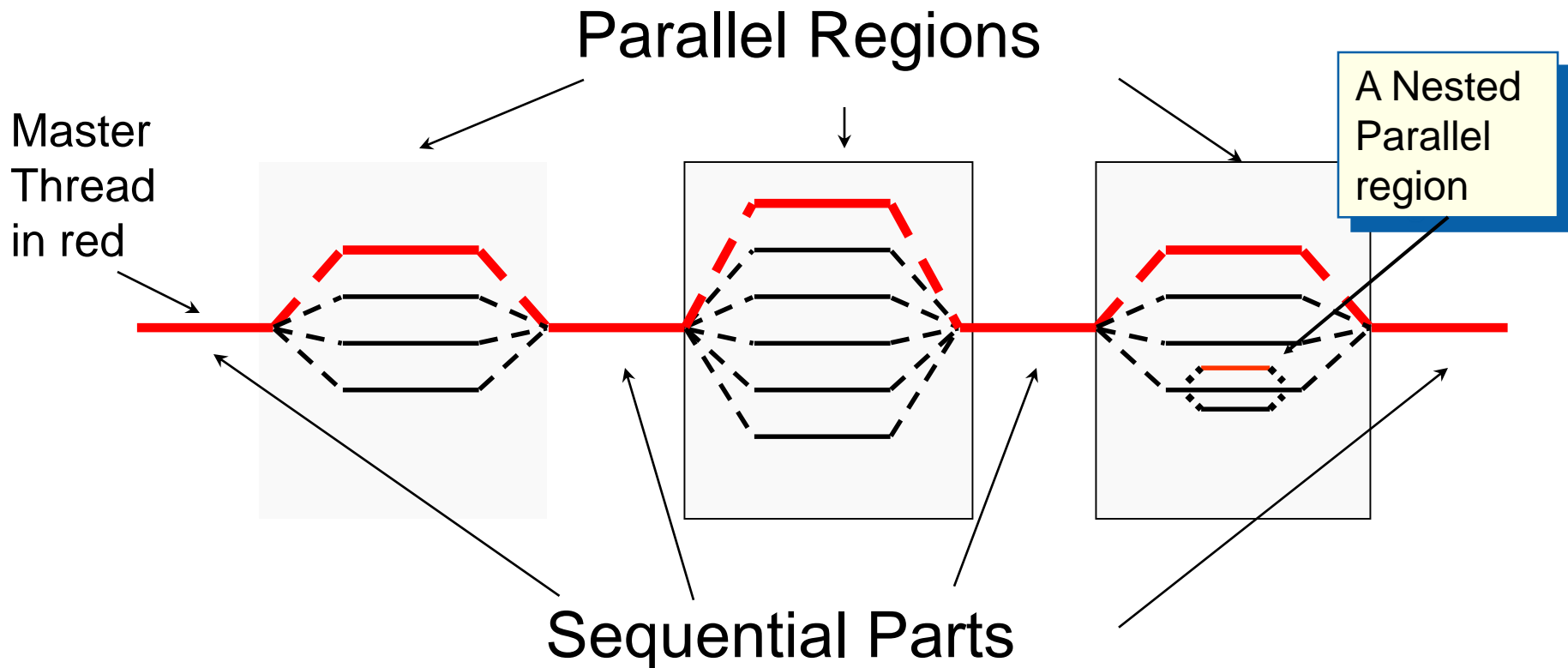
Thread creation: Parallel regions

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

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

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

Runtime function to request a certain number of threads

Runtime function returning a thread ID

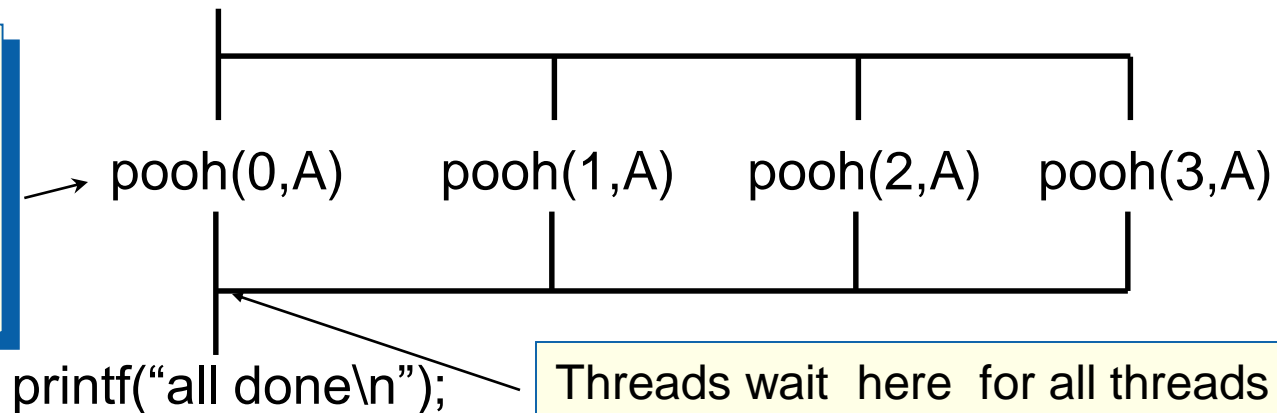● Each thread calls pooh(ID,A) for `ID = 0 to 3`

Thread creation: Parallel regions example

- Each thread executes the same code redundantly.

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

double A[1000];

omp_set_num_threads(4)

A single copy of A is shared between all threads.

pooh(0,A)   pooh(1,A)   pooh(2,A)   pooh(3,A)

printf("all done\n");

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

Thread creation: How many threads did you actually get?

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

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

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

        int nthrds = omp_get_num_threads();
        pooh(ID,A);
}
```

Runtime function to request a certain number of threads

Runtime function to return actual number of threads in the team

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

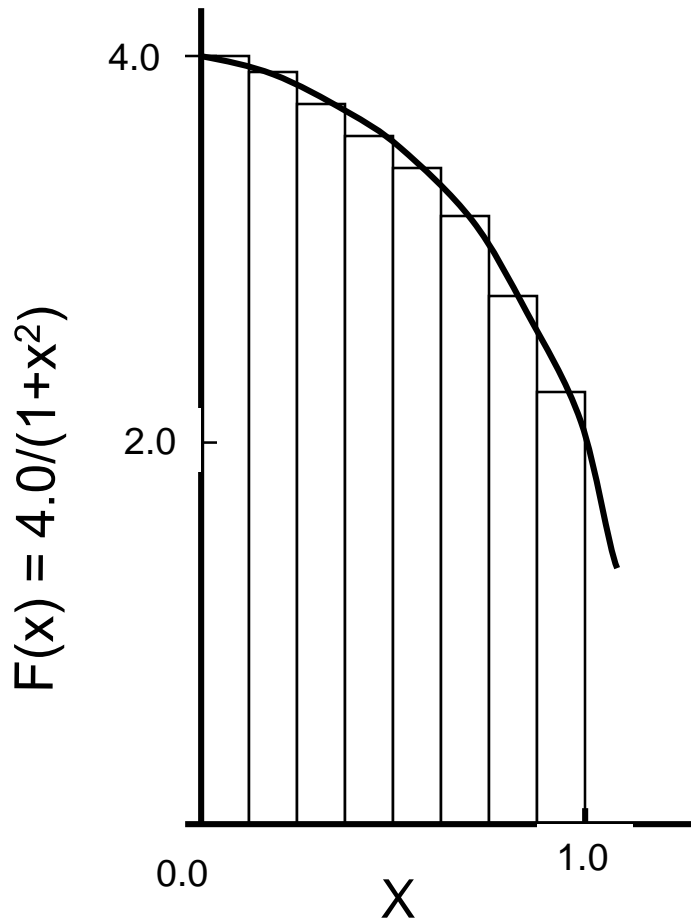# An interesting problem to play with
Numerical integration

Mathematically, we know that:

$$\int_{0}^{1} \frac{4.0}{(1+x^2)} \, dx = \pi$$

We can approximate the integral as a sum of rectangles:

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

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

Serial PI program

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

           step = 1.0/(double) num_steps;

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

See OMP_exercises/pi.c

Serial PI program

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

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

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

See OMP_exercises/pi.c

Exercise: the parallel Pi program

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

  #pragma omp parallel.

- Pay close attention to shared versus private variables.

- In addition to a parallel construct, you will need the runtime library routines

  - int omp_get_num_threads(); → Number of threads in the team

  - int omp_get_thread_num(); → Thread ID or rank

  - double omp_get_wtime(); → Time in Seconds since a fixed point in the past

  - omp_set_num_threads(); → Request a number of threads in the team

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

http://www.nersc.gov/users/software/programming-models/openmp/sc17-openmp/

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();

22

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 default optimization level (O2) on Apple OS X 10.7.3 with a dual core
(four HW thread) Intel® Core™ i5 processor at 1.7 Ghz and 4 Gbyte DDR3 memory at 1.333 Ghz.23
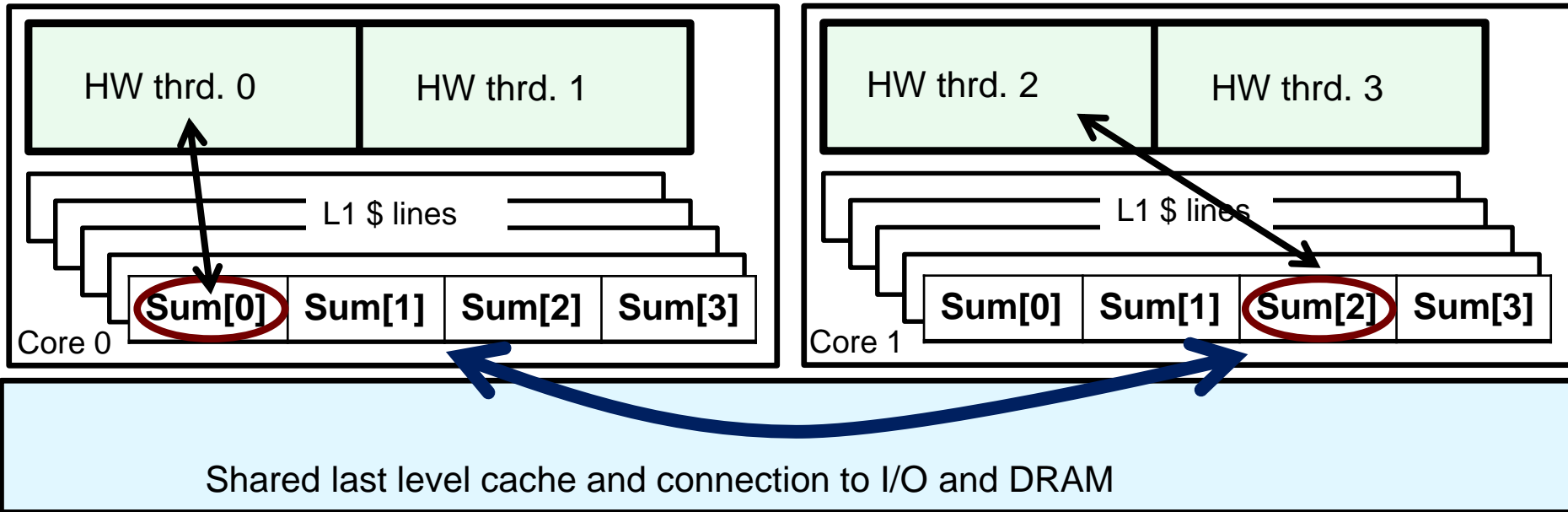
SPMD: Single Program Mulitple Data

- Run the same program on P processing elements where P can be arbitrarily large.

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

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

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

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"**.

HW thrd. 0	HW thrd. 1

L1 $ lines

Sum[0]	**Sum[1]**	**Sum[2]**	**Sum[3]**

Core 0

HW thrd. 2	HW thrd. 3

L1 $ lines

Sum[0]	**Sum[1]**	**Sum[2]**	**Sum[3]**

Core 1

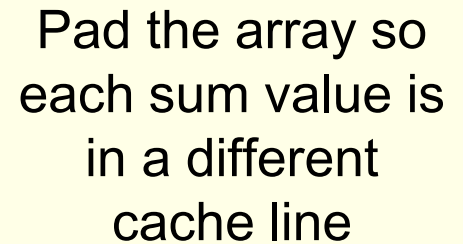Shared last level cache and connection to I/O and DRAM

- 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

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

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

Results*: pi program padded accumulator

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

Example: eliminate False sharing by padding the sum array

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

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

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

Changing the Number of Threads

- Inside the OpenMP runtime is an Internal Control Variable (ICV) for the default number of threads requested by a parallel construct.

- The system has an implementation defined value for that ICV

- When an OpenMP program starts up, it queries an environment variable OMP_NUM_THREADS and sets the appropriate <u>internal control variable</u> to the value of OMP_NUM_THREADS
  - For example, to set the default number of threads on my apple laptop
    - ➢ export OMP_NUM_THREADS=12

- The omp_set_num_threads() runtime function overrides the value from the environment and resets the ICV to a new value.

- A clause on the parallel construct requests a number of threads for that parallel region, but it does not change the ICV
  - #pragma omp parallel num_threads(4)

Outline

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

Synchronization

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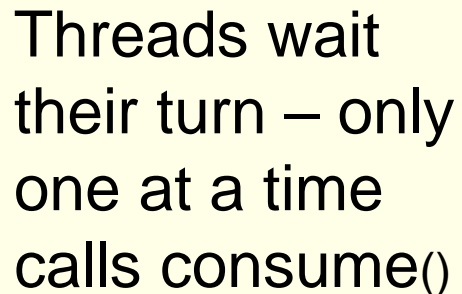 - critical
  - barrier

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

Synchronization: critical

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

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

```
float  res;

#pragma omp parallel

{     float B;   int i, id, nthrds;

      id = omp_get_thread_num();

      nthrds = omp_get_num_threads();

       for(i=id;i<niters;i+=nthrds){

            B =  big_job(i);

#pragma omp critical
            res += consume (B);

      }
}
```

Synchronization: barrier

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

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

omp_set_num_threads(8)

#pragma omp parallel

{    int id, nthrds;

        id = omp_get_thread_num();

        nthrds = omp_get_num_threads();

        if (id==0) numthrds = nthrds;

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

#pragma omp barrier
        Brr[id] = really_big_and_ugly(id, nthrds, 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.
  - #pragma omp critical
  - #pragma omp parallel
  - omp_set_num_threads()
  - omp_get_num_threads()
  - omp_get_thread_num()
  - export OMP_NUM_THREADS=42

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 default optimization level (O2) on Apple OS X 10.7.3 with a dual core (four HW thread) Intel® Core™ i5 processor at 1.7 Ghz and 4 Gbyte DDR3 memory at 1.333 Ghz. 34

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

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

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

Be careful where you put a critical section

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

Outline

- Introduction to OpenMP
- Creating Threads
- Synchronization
➡ - Parallel Loops
- Data environment
- Memory model
- Irregular Parallelism and tasks
- Recap
- Beyond the common core:
  − Worksharing revisited
  − Synchronization: More than you ever wanted to know
  − Threadprivate data
  − Thread affinity and data locality

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.

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

42

Working with loops

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

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

Note: loop index "i" is private by default

Remove loop carried dependence

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

Reduction

⑩ How do we handle this case?

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

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

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

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

Reduction

- OpenMP reduction clause:

  reduction (op : list)

- Inside a parallel or a work-sharing construct:

  − A local copy of each list variable is made and initialized depending on the "op" (e.g. 0 for "+").

  − Updates occur on the local copy.

  − Local copies are reduced into a single value and combined with the original global value.

- The variables in "list" must be shared in the enclosing parallel region.

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

OpenMP: Reduction operands/initial-values

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

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

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

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

Exercise: Pi with loops and a reduction

- 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();
```

Remember: OpenMP makes the loop control index in a loop workshare construct private for you … you don't need to do this yourself

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.

48

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

The nowait clause

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

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

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

implicit barrier at the end of a for worksharing construct

no implicit barrier due to nowait

implicit barrier at the end of a parallel region

Outline

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

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
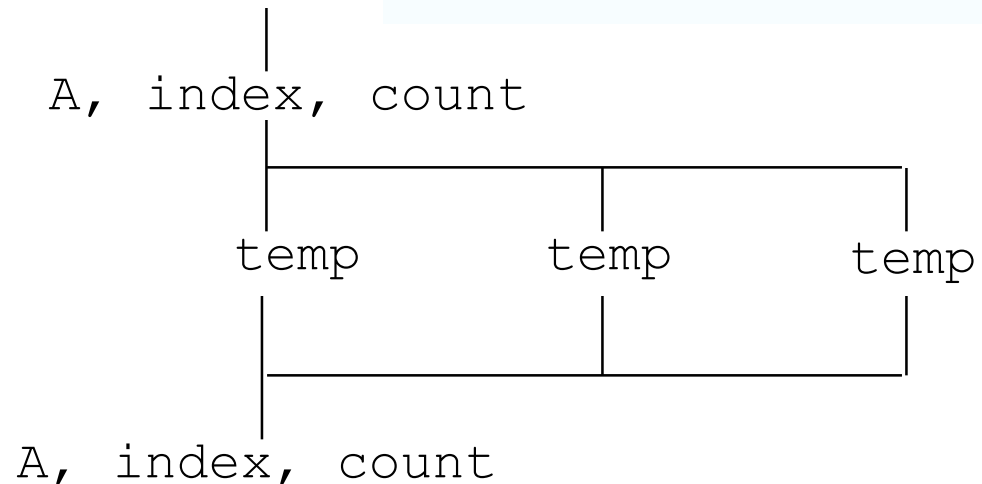
Data sharing: Changing storage attributes

- 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 clauses apply to the OpenMP construct NOT to the entire region.

- 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 be used on parallel constructs

Data sharing: Private clause

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

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

tmp was not initialized

tmp is 0 here

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

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

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

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

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

tmp has unspecified value

unspecified which copy of tmp

Firstprivate clause

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

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

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

Data sharing:
A data environment test

- Consider this example of PRIVATE and FIRSTPRIVATE

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

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

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

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

Data sharing: Default clause

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

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

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

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

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

Mandelbrot Set
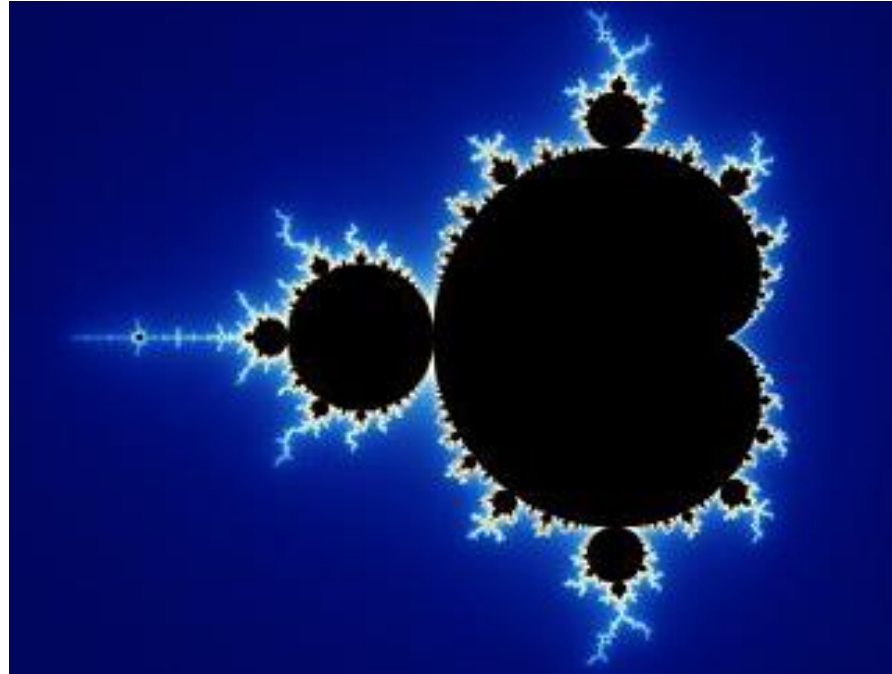
> **For each point c in the complex plane,**
> **is $z = z*z + c$ bounded?**

Black is inside set. Other colors indicate how quickly it crossed threshold.

Exercise: Mandelbrot set area

- The supplied program (mandel.c) computes the area of a Mandelbrot set.

- The program has been parallelized with OpenMP, but we were lazy and didn't do it right.

- Find and fix the errors (hint … the problem is with the data environment).

- Once you have a working version, try to optimize the program.
  - Try different schedules on the parallel loop.
  - Try different mechanisms to support mutual exclusion … do the efficiencies change?

The Mandelbrot area program

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

int main(){
   int i, j;
   double area, error, eps  = 1.0e-5;
#pragma omp parallel for default(shared) private(c, j) \
    firstprivate(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;
}
```
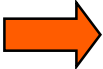
```c
void testpoint(struct  d_complex c){
struct d_complex z;
    int iter;
    double temp;

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

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

Outline

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

OpenMP memory model

- OpenMP supports a shared memory model
- All threads share an address space, but it can get complicated:
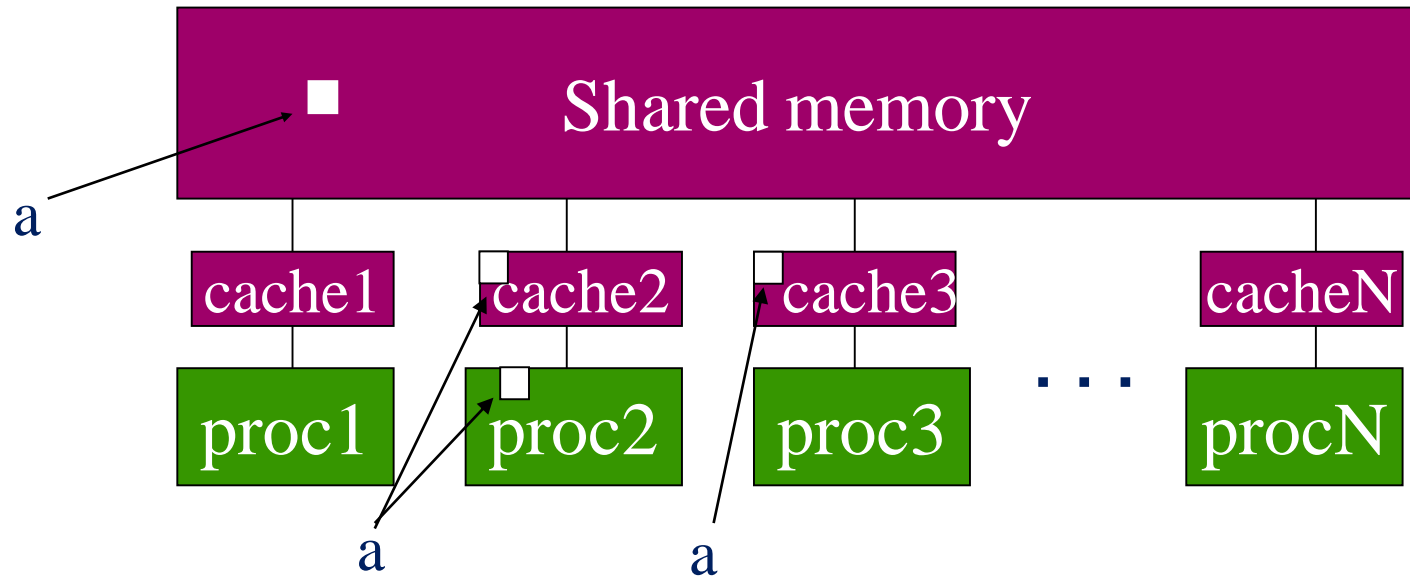
- Multiple copies of data may be present in memory, various levels of cache, or in registers

OpenMP and relaxed consistency

- OpenMP supports a **relaxed-consistency** shared memory model
  - Threads can maintain a **temporary view** of shared memory that is not consistent with that of other threads

  - These temporary views are made consistent only at certain points in the program

  - The operation that enforces consistency is called the **flush operation**

Flush operation

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

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

Flush and synchronization

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

    ….

    (but not at entry to worksharing regions)

> This means if you are mixing reads and writes of a variable across multiple threads, you cannot assume the reading threads see the results of the writes unless:
>
> - the writing threads follow the writes with a construct that implies a flush.
> - the reading threads precede the reads with a construct that implies a flush.
>
> This is a rare event … or putting this another way, you should avoid writing code that depends on ordering reads/writes around flushes.

Outline

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

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

Serial
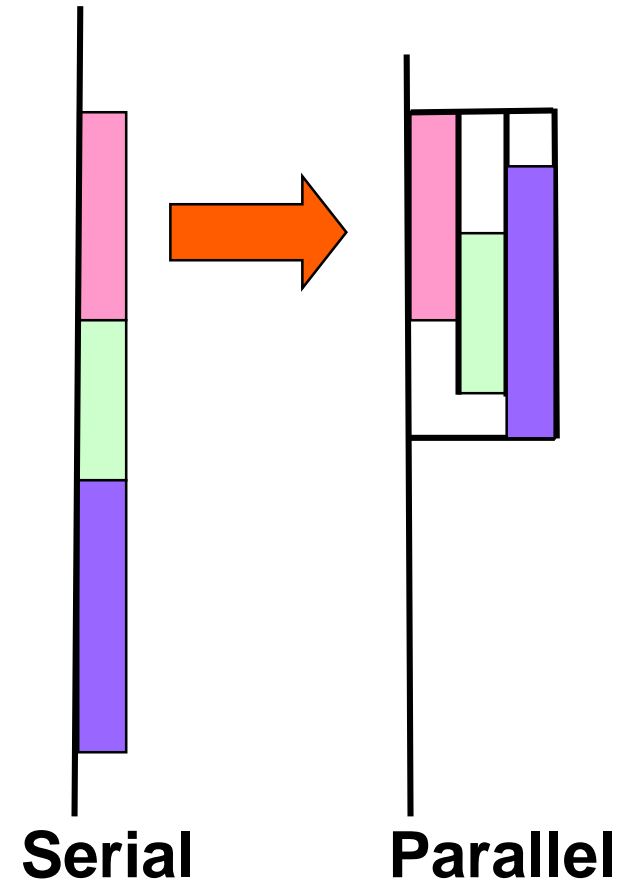
Parallel

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.

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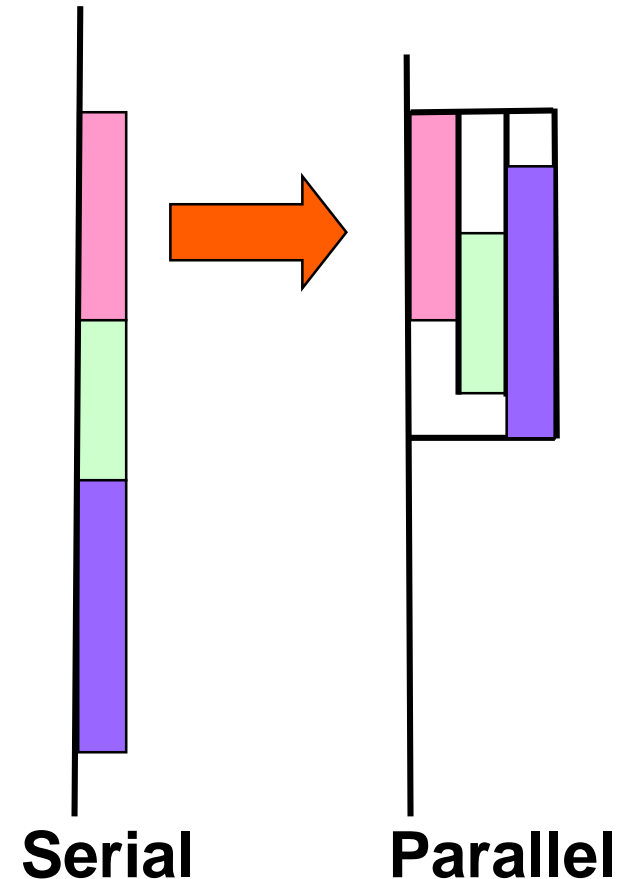
A common Pattern is to have one thread create the tasks while the other threads wait at a barrier and execute the tasks

70

Single worksharing Construct

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

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

}
```

Task Directive

`#pragma omp task` *[clauses]*

structured-block

```
#pragma omp parallel          ← — — — Create some threads
{
    #pragma omp single        ← — — — One Thread
     {                                 packages tasks
        #pragma omp task
            fred();
        #pragma omp task      ← — — — Tasks executed by
            daisy();                  some thread in some
        #pragma omp task      ← — — — order
            billy();
     }
} ←                            All tasks complete before this barrier is released
```

Exercise: Simple tasks

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

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

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

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

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

Racey cars: solution

```c
#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");
}
```

When/where are tasks complete?

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

- At taskwait directive
  - i.e. Wait until all tasks defined in the current task have completed.
    ```
    #pragma omp taskwait
    ```
  - Note: applies only to tasks generated in the current task, not to "descendants" .

Example

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

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

Linked list traversal

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

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

Parallel linked list traversal

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

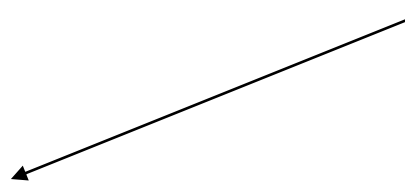makes a copy of **p** when the task is packaged

Data scoping with tasks

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

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

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

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

Data scoping defaults

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

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

A is shared
B is firstprivate
C is private

Example: Fibonacci numbers

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

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

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

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

Parallel Fibonacci

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

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

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

- Binary tree of tasks

- Traversed using a recursive function

- A task cannot complete until all tasks below it in the tree are complete (enforced with taskwait)

- **x,y** are local, and so by default they are private to current task
    - must be shared on child tasks so they don't create their own firstprivate copies at this level!

Divide and conquer

• Split the problem into smaller sub-problems; continue until the sub-problems can be solve directly
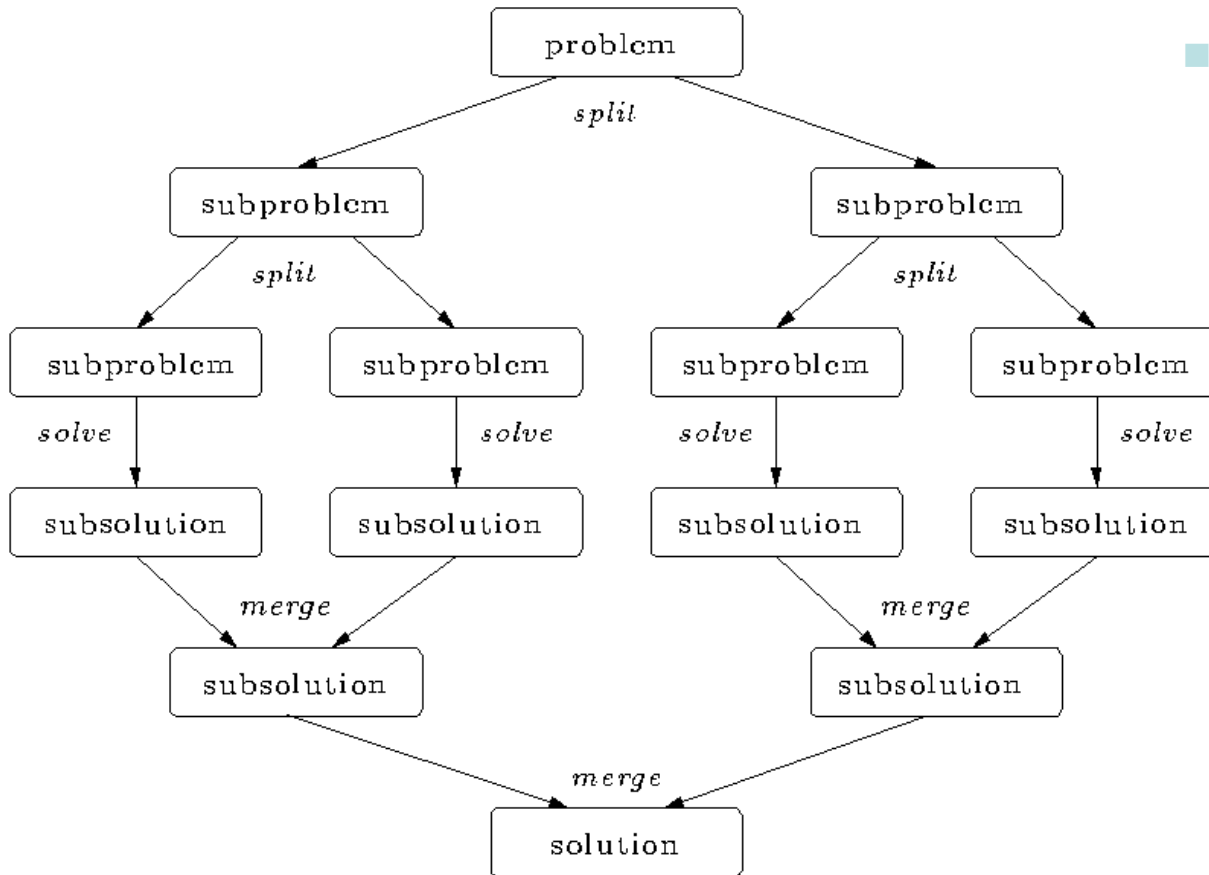
■ 3 Options:

  ☐ Do work as you split into sub-problems

  ☐ Do work only at the leaves

  ☐ Do work as you recombine

Exercise: Pi with tasks

- Consider the program Pi_recur.c. This program uses a recursive algorithm in integrate the function in the pi 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();

Program: OpenMP tasks

```c
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;
}
```

```c
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;
}
```
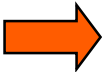
Results*: pi with tasks

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

Using tasks

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

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

Outline

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

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

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

There is much more to OpenMP than the Common Core.

- Synchronization mechanisms
  - locks, flush and several forms of atomic

- Data environment
  - lastprivate, threadprivate, default(private|shared)

- Fine grained task control
  - dependencies, tied vs. untied tasks, task groups, task loops …

- Vectorization constructs
  - simd, uniform, simdlen, inbranch vs. nobranch, ….

- Map work onto an attached device
  - target, teams distribute parallel for, target data …

- … and much more. The OpenMP 4.5 specification is over 350 pages!!!

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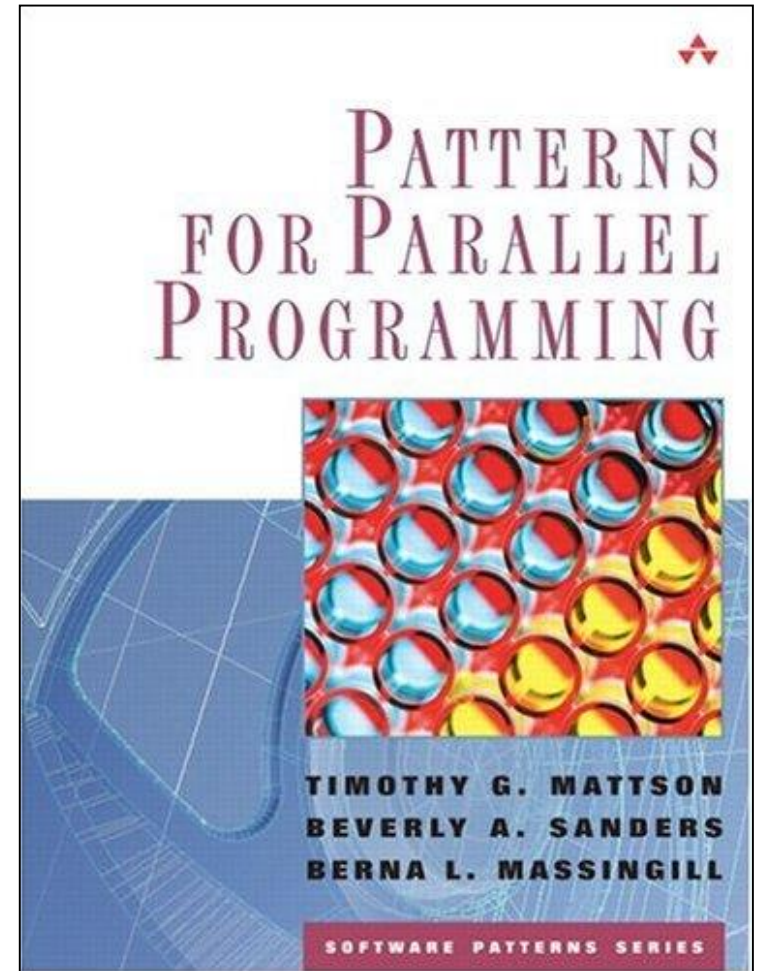
OpenMP organizations

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

  www.openmp.org

- OpenMP User's Group (cOMPunity) URL:

  www.compunity.org

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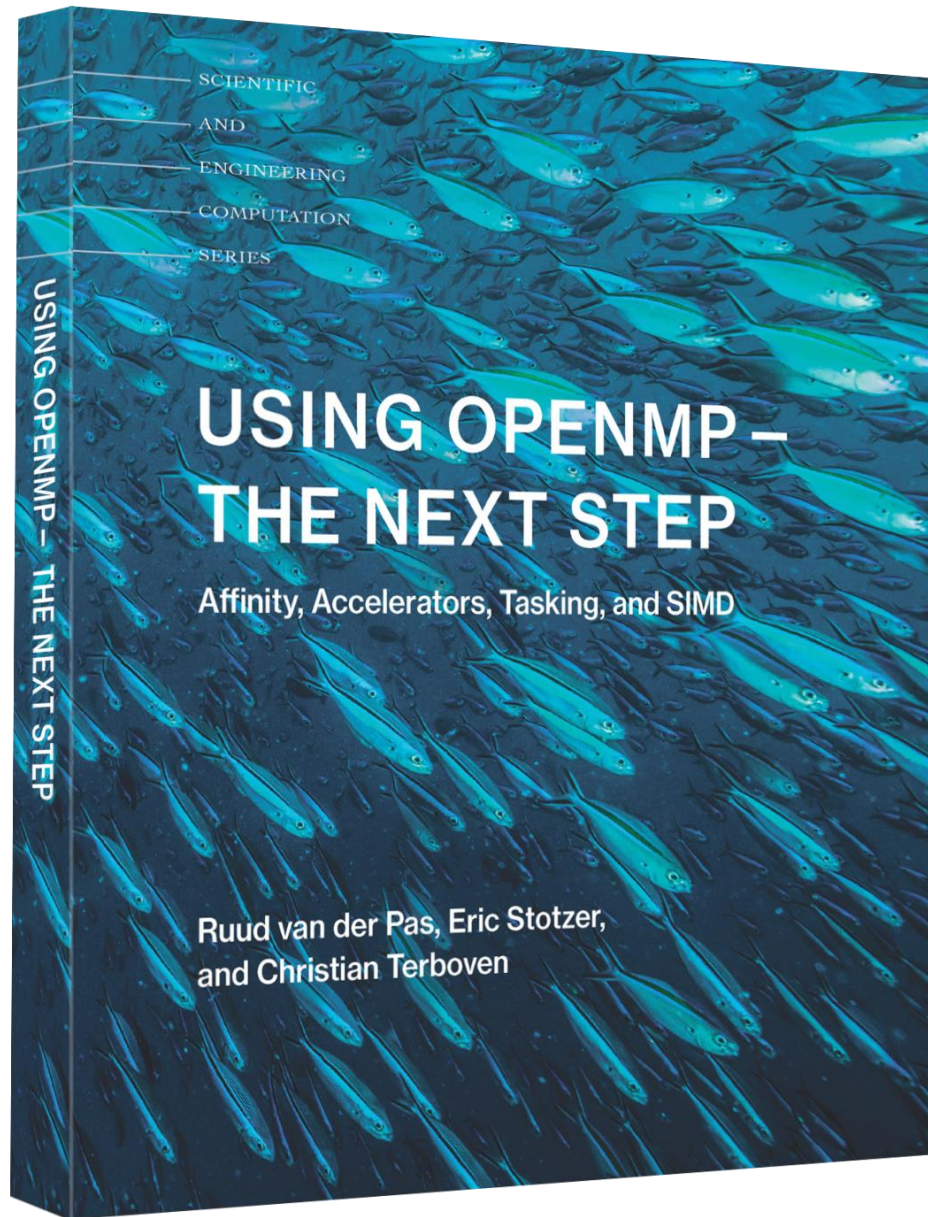
Books about OpenMP

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

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

Resources:

A great new book that covers OpenMP features beyond OpenMP 2.5

SCIENTIFIC AND ENGINEERING COMPUTATION SERIES

USING OPENMP – THE NEXT STEP

USING OPENMP – THE NEXT STEP

Affinity, Accelerators, Tasking, and SIMD

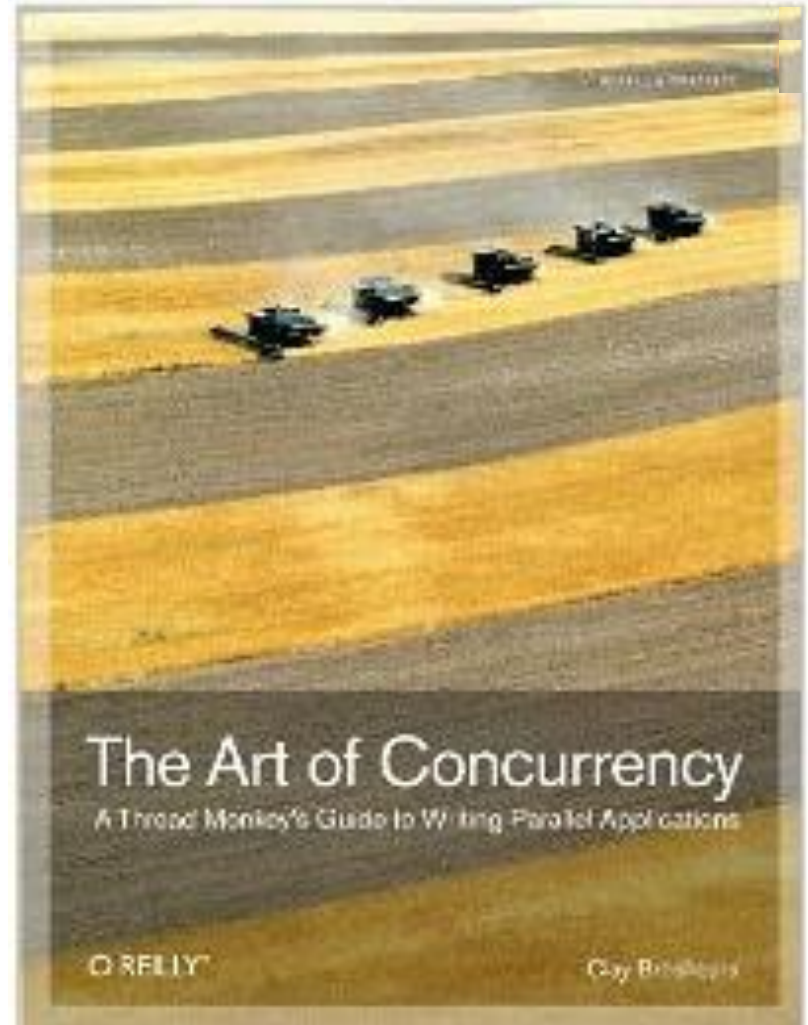Ruud van der Pas, Eric Stotzer, and Christian Terboven

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

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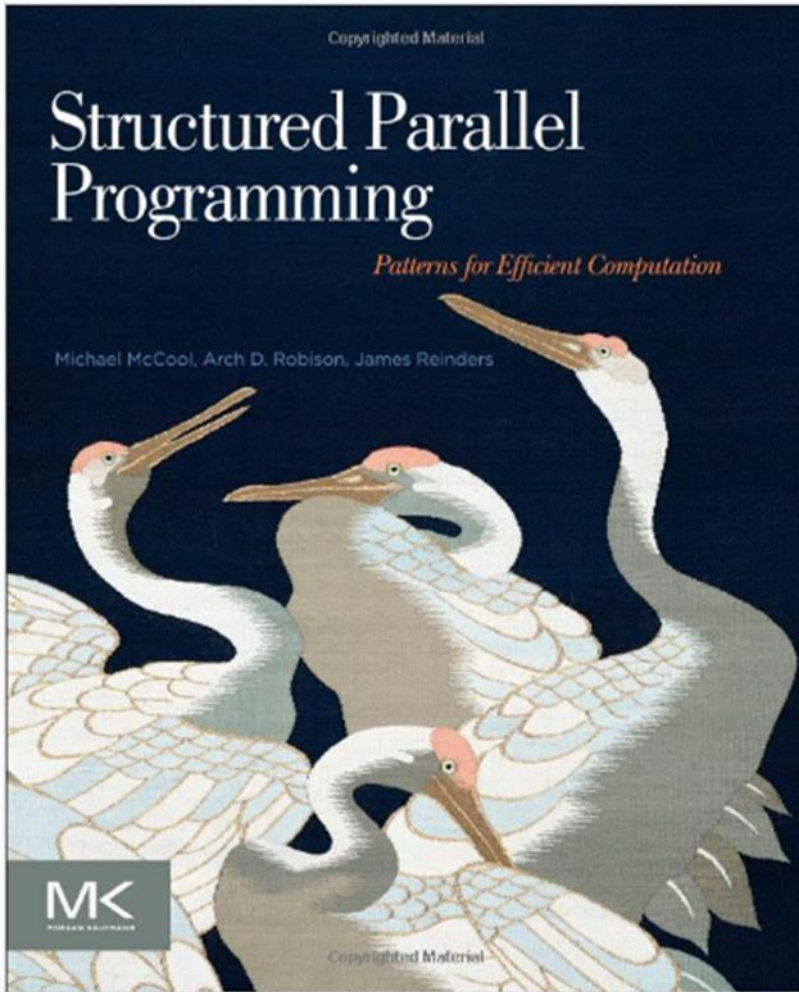
A great book that explores key patterns with Cilk, TBB, OpenCL, and OpenMP (by McCool, Robison, and Reinders)

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

Please tell our SC tutorial overlords how amazingly GREAT this tutorial is!!!!

Online feedback forms available through the following URL or QR code

Evaluation site URL: http://bit.ly/sc17-eval

Outline

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

Outline

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

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

Loop worksharing constructs:
The schedule clause

- The schedule clause affects how loop iterations are mapped onto threads
  - schedule(static [,chunk])
    - Deal-out blocks of iterations of size "chunk" to each thread.
  - schedule(dynamic[,chunk])
    - Each thread grabs "chunk" iterations off a queue until all iterations have been handled.
  - schedule(guided[,chunk])
    - Threads dynamically grab blocks of iterations. The size of the block starts large and shrinks down to size "chunk" as the calculation proceeds.
  - schedule(runtime)
    - Schedule and chunk size taken from the OMP_SCHEDULE environment variable (or the runtime library).
  - schedule(auto)
    - Schedule is left up to the runtime to choose (does not have to be any of the above).

OpenMP 4.5 added modifiers monotonic, nonmontonic and simd.

loop work-sharing constructs:
The schedule clause

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

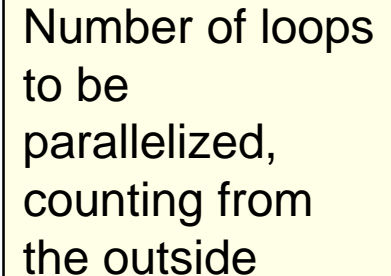Least work at runtime : scheduling done at compile-time

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

Nested loops

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

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

Number of loops to be parallelized, counting from the outside

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

101

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

```c
#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

Outline

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

Synchronization

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

- High level synchronization:
  - **critical**
  - **barrier**

  Covered earlier

  - atomic
  - ordered
- Low level synchronization
  - flush
  - locks (both simple and nested)

Synchronization: atomic

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

```
#pragma omp parallel
{
        double B;

        B =  DOIT();


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

}
```

Synchronization: atomic

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

```
#pragma omp parallel
{
        double B, tmp;

        B =  DOIT();

        tmp = big_ugly(B);

  #pragma omp atomic
            X +=  tmp;

}
```

Atomic only protects the read/update of X

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

Exercise

- In your first Pi program, you probably used an array to create space for each thread to store its partial sum.

- You fixed this by using a critical section instead of updating the array (remember .. the array you created by promoting the scalar "sum" to an array).

- Use and atomic instead.   Does the performance improve?

Parallel loop with ordered region

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

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

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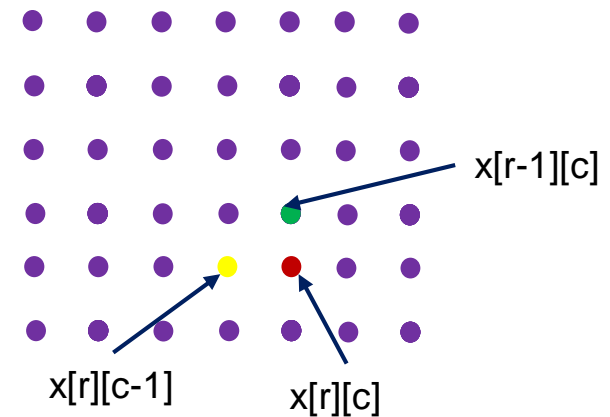
Parallelizing nested loops

- Will these nested parallel loops execute correctly?

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

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

   }
}
```

An array section of x

x[r-1][c]

x[r][c-1]   x[r][c]

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

Ordered stand-alone directive

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

#pragma omp ordered depend(sink : vec)

#pragma omp ordered depend(source)

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

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

Parallelizing DOACROSS loops

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

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

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

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

OpenMP memory model

- OpenMP supports a shared memory model

- All threads share an address space, where variable can be stored or retrieved:
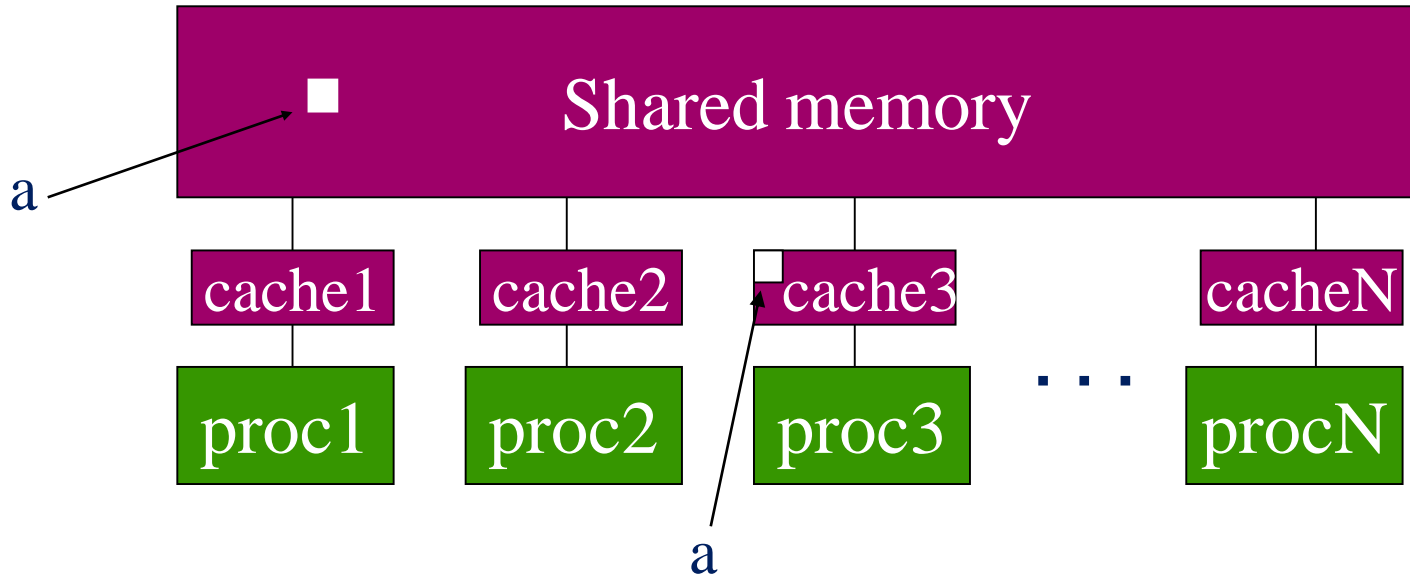
- Threads maintain their own temporary view of memory as well … the details of which are not defined in OpenMP but this temporary view typically resides in caches, registers, write-buffers, etc.

Flush operation

- Defines a sequence point at which a thread enforces a consistent view of memory.

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

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

Memory consistency: flush example

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

```
double A;

A = compute();

#pragma omp flush(A)

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

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

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

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

Flush and synchronization

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

  ….

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

Example: prod_cons.c

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

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

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

  runtime = omp_get_wtime();

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

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

  runtime = omp_get_wtime() - runtime;

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

  - Often used with a stream of produced values to implement "pipeline parallelism"

  - The key is to implement pairwise synchronization between threads

Pairwise synchronization in OpenMP

- OpenMP lacks synchronization constructs that work between pairs of threads.

- When needed, you have to build it yourself.

- Pairwise synchronization
  - Use a shared flag variable
  - Reader spins waiting for the new flag value
  - Use flushes to force updates to and from memory

Exercise: Producer/consumer

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

            flag = 1;

        }
        #pragma omp section
        {

            while (flag == 0){

            }

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

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

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

119

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

123

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)

125

Synchronization: Lock routines

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

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

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

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

Synchronization: Simple locks

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

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

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

One lock per element of hist

Enforce mutual exclusion on update to hist array

Free-up storage when done.

127

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 $\rightarrow$ 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
```

129

Exercise

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

Outline

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

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.

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

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

133

Data copying: Copyin

You initialize threadprivate data using a copyin clause.

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

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

!$OMP PARALLEL COPYIN(A)

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

!$OMP END PARALLEL

end
```

Data copying: Copyprivate

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

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

void main()
{
  int Nsize, choice;

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

      do_work(Nsize, choice);
  }
}
```
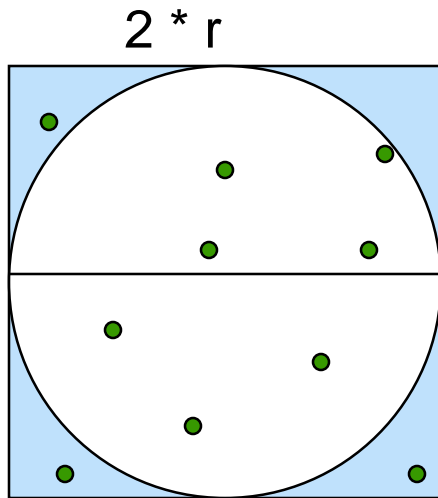
Exercise: Monte Carlo calculations

Using random numbers to solve tough problems

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

2 * r

$N = 10 \quad\quad \pi = 2.8$

$N = 100 \quad\quad \pi = 3.16$

$N = 1000 \quad\quad \pi = 3.148$

- Throw darts at the circle/square.
- Chance of falling in circle is proportional to ratio of areas:

$$A_c = r^2 * \pi$$

$$A_s = (2*r) * (2*r) = 4 * r^2$$

$$P = A_c/A_s = \pi/4$$

- Compute π by randomly choosing points; π is four times the fraction that falls in the circle

Exercise: Monte Carlo pi (cont)

- We provide three files for this exercise
  - pi_mc.c: the Monte Carlo method pi program
  - random.c: a simple random number generator
  - random.h: include file for random number generator
- Create a parallel version of this program without changing the interfaces to functions in random.c
  - This is an exercise in modular software … why should a user of your parallel random number generator have to know any details of the generator or make any changes to how the generator is called?
  - The random number generator must be thread-safe.
- Extra Credit:
  - Make your random number generator numerically correct (non-overlapping sequences of pseudo-random numbers).

Outline

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

Thread Affinity and Data Locality

- Affinity
  - Process Affinity: bind processes (MPI tasks, etc.) to CPUs
  - Thread Affinity: further binding threads to CPUs that are allocated to their parent process

- Data Locality
  - Memory Locality: allocate memory as close as possible to the core on which the task that requested the memory is running
  - Cache Locality: use data in cache as much as possible

- Correct process, thread and memory affinity is the basis for getting optimal performance.

Memory Locality

- Most systems today are Non-Uniform Memory Access (NUMA)
- Example, the Intel® Xeon Phi™ processor

Over 6 TF SP peak
Full Xeon ISA compatibility through AVX-512

Up to 16GB high-bandwidth on-package memory (MCDRAM)
Exposed as NUMA node
>400 GB/s sustained BW

2x 512b VPU per core
(Vector Processing Units)

Up to 72 cores (36 tiles)
2D mesh architecture

6 channels DDR4
Up to 384GB
~90 GB/s

Up to 72 cores

Tile

2 VPU | HUB | 2 VPU
Core | 1MB L2 | Core

On-package
2 ports OPA
Integrated Fabric

Based on Intel® Atom™ processor with many HPC enhancements
Deep out-of-order buffers
Gather/scatter in hardware
Improved branch prediction
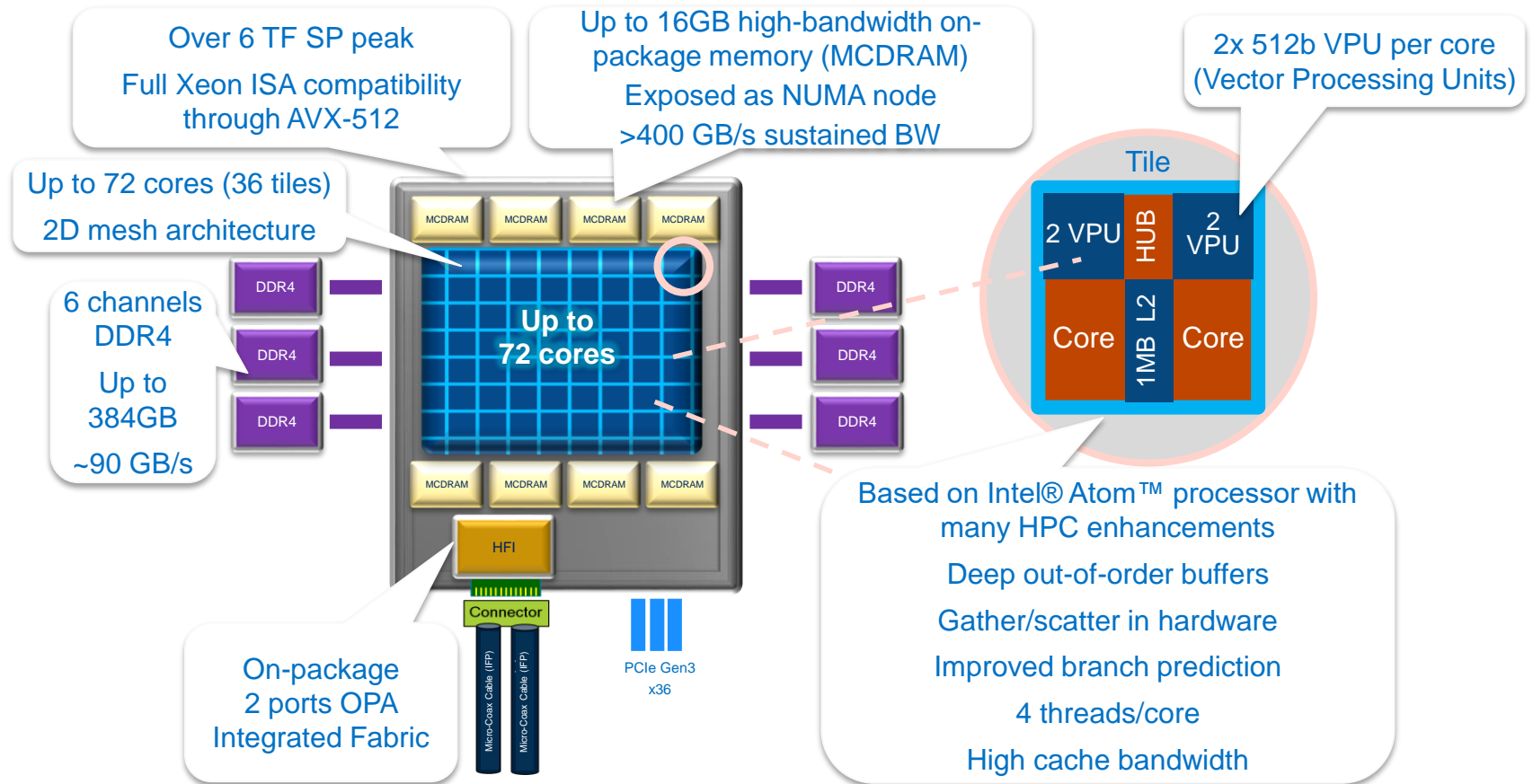4 threads/core
High cache bandwidth

Diagram is for conceptual purposes only and only illustrates a CPU and memory – it is not to scale and does not include all functional areas of the CPU, nor does it represent actual component layout.

Memory Locality

- Memory access in different NUMA domains are different
  - Accessing memory in remote NUMA is slower than accessing memory in local NUMA
  - Accessing High Bandwidth Memory on KNL* is faster than DDR

- OpenMP does not explicitly map data across shared memories

- Memory locality is important since it impacts both memory and intra-node performance

*KNL: Intel® Xeon Phi™ processor 7250 with 68 cores @ 1.4 Ghz …
the "bootable" version that sits in a socket, not a co-processor

Example Compute Nodes (Intel Haswell*)

Cori Phase1 Compute Node
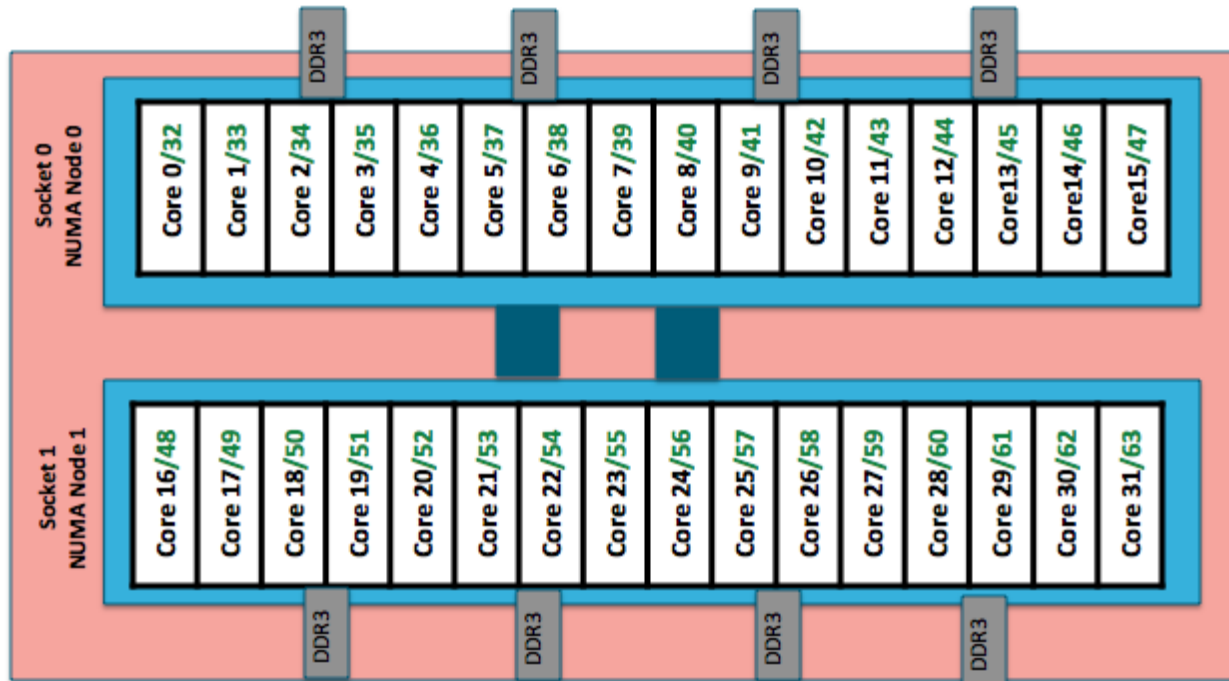
- **An Intel Haswell node has 32 cores (64 CPUs), 128 MB DDR memory.**

- **2 NUMA domains per node, 16 cores per NUMA domain. 2 hardware threads (CPUs) per core.**

- **Memory bandwidth is non-homogeneous among NUMA domains.**
  - **CPUs 0-15, 32-47 are closer to memory in NUMA domain 0, farther to memory in NUMA domain 1.**
  - **CPUs 16-31, 48-64 are closer to memory in NUMA domain 1, farther to memory in NUMA domain 0.**

*Haswell: 16-core **Intel® Xeon™ Processor E5-2698 v3 at 2.3 GHz**

Tools to Check Compute Node Information (1)

- **numactl:** controls NUMA policy for processes or shared memory
  - − **numactl -H:** provides NUMA info of the CPUs

% numactl –H

```
% 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
```
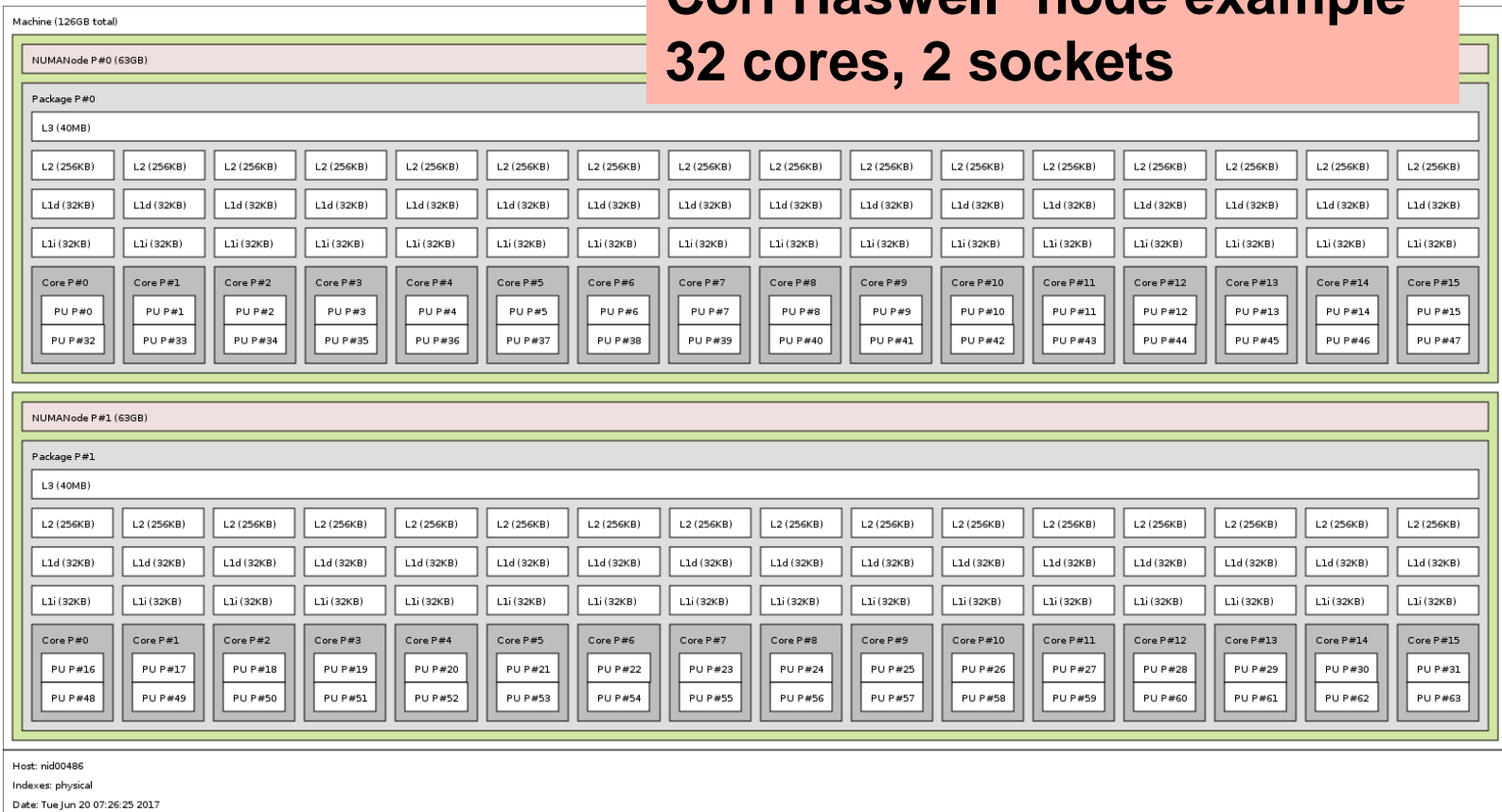
Tools to Check Compute Node Information (2)

- **Portable Hardware Locality (hwloc)**
  - **hwloc-ls:** provides a graphical representation of the system topology, NUMA nodes, cache info, and the mapping of procs.

% hwloc-ls

Cori Haswell* node example 32 cores, 2 sockets

*Haswell: 16-core **Intel® Xeon™ Processor E5-2698 v3 at 2.3 GHz**

144

Memory Affinity: "First Touch" Memory

Step 1.1 Initialization by master 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];}

OMP_PROC_BIND=close

Memory affinity is not defined when memory was allocated, instead it will be 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

145
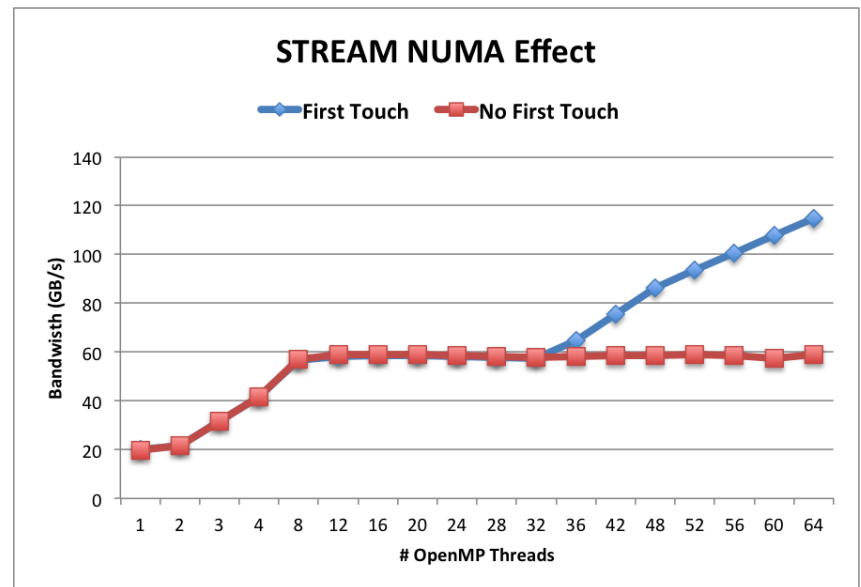
"Perfect Touch" is Hard

- Hard to do "perfect touch" for real applications.
- General recommendation is to use number of threads fewer than number of CPUs per NUMA domain.
- In the previous example, 16 cores (32 CPUs) per NUMA domain. Sample run options:
  - 2 MPI tasks, 1 MPI task per NUMA domain, with 32 OpenMP threads (if using hyperthreads) or 16 OpenMP threads (if not using hyperthreads) per MPI task
  - 4 MPI tasks, 2 MPI tasks per NUMA domain, with 16 OpenMP threads (if using hyperthreads) or 8 OpenMP threads (if not using hyperthreads) per MPI task
  - …

Runtime Environment Variable: OMP_PROC_BIND

- Controls thread affinity within and between OpenMP places
- OpenMP 3.1 only has OMP_PROC_BIND, either TRUE or FALSE.
  - If true, the runtime will not move threads around between processors.
- OpenMP 4.0 still allows the above. Added options:
  - close: bind threads close to the master thread
  - spread: bind threads as evenly distributed (spreaded) as possible
  - master: bind threads to the same place as the master thread
- Examples:
  - OMP_PROC_BIND=spread
  - OMP_PROC_BIND=spread,close (for nested levels)

Runtime Environment Variable: OMP_PROC_BIND (2)

- Use 4 cores total, 2 hyperthreads per core, and OMP_NUM_THREADS=4 an example
- none: no affinity setting.
- 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	

- master: bind threads to the same place as the master thread

Runtime Environment Variable: OMP_PLACES (1)

- OpenMP 4.0 added OMP_PLACES environment variable
  - To control thread allocation
  - defines a series of places to which the threads are assigned

- OMP_PLACES can be
  - 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.
  - A list with explicit CPU ids (see next slide)

- Examples:
  - export OMP_PLACES=threads
  - export OMP_PLACES=cores

Runtime Environment Variable: OMP_PLACES (2)

- OMP_PLACES can also be
  - A list with explicit place values of CPU ids, such as:
    - "{0,1,2,3},{4,5,6,7},{8,9,10,11},{12,13,14,15}"
    - "{0:4},{4:4},{8:4},{12:4}" (default stride is 1)
    - Format: {lower-bound:length:stride}. Thus, specifying {0:3:2} is the same as specifying {0,2,4}

- Examples:
  - export OMP_PLACES=" {0:4:2},{1:4:2}" (which is equivalent to "{0,2,4,6},{1,3,5,7}")
  - export OMP_PLACES="{0:8:1}" (which is equivalent to "{0,1,2,3,4,5,6,7}"

Other Runtime Environment Variables for Affinity Support

- OMP_NUM_THREADS

- OMP_THREAD_LIMIT

- OMP_NESTED

- OMP_MAX_ACTIVE_LEVELS

- Names are upper case, values are case insensitive

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

STREAM OMP_PROC_BIND Effect
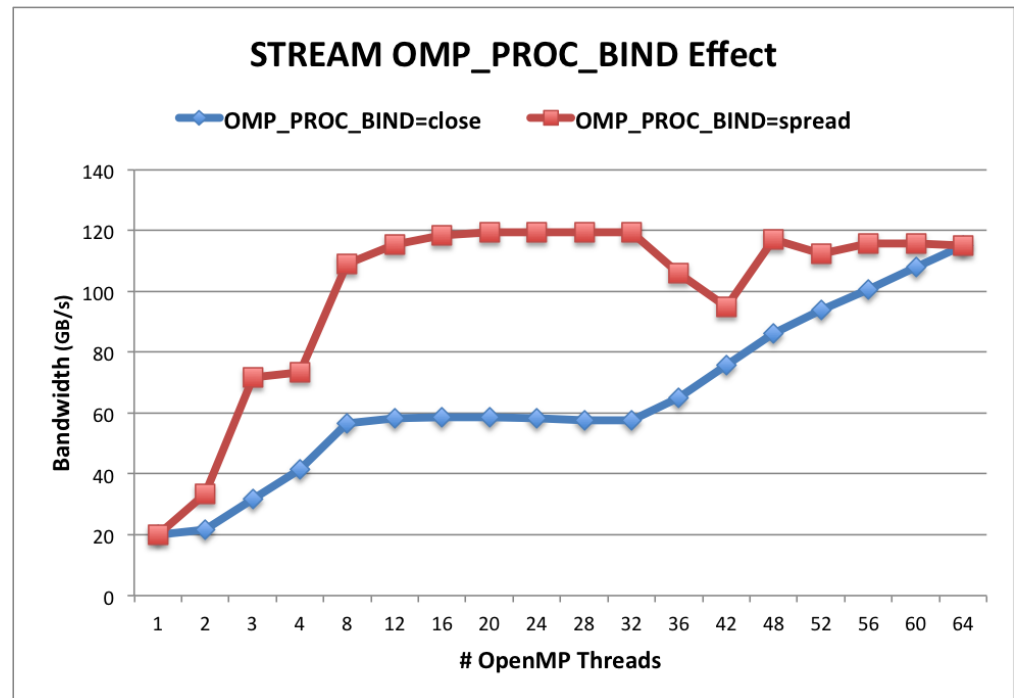
Affinity Clauses for OpenMP Parallel Construct

- 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

Runtime APIs for Thread Affinity Support

- OpenMP 4.5 added runtime functions to determine the effect of thread affinity clauses
- Query functions for OpenMP thread affinity were added
  - **omp_get_num_places**: returns the number of places
  - **omp_get_place_num_procs**: returns number of processors in the given place
  - **omp_get_place_proc_ids**: returns the ids of the processors in the given place
  - **omp_get_place_num:** returns the place number of the place to which the current thread is bound
  - **omp_get_partition_num_places:** returns the number of places in the current partition
  - **omp_get_partition_place_nums:** returns the list of place numbers corresponding to the places in the current partition

Other Runtime APIs for Thread Affinity Support

- omp_get_nested, omp_set_nested
- omp_get_thread_limit
- omp_get_level
- omp_get_active_level
- omp_get_max_active_levels, omp_set_max_active_levels
- omp_get_proc_bind, omp_set_proc_bind
- omp_get_num_threads, omp_set_num_threads
- omp_get_max_threads

Exercise: "First Touch" with STREAM benchmark

- STREAM benchmark codes: stream.c, stream.f

- Check the source codes to see if "first touch" is implemented

- With "first touch" on (stream.c) and off (stream_nft.c), experiment with different OMP_NUM_THREADS and OMP_PROC_BIND settings to understand how "first touch" and OMP_PROC_BIND choices affect STREAM memory bandwidth results (look at the Best Rate for Triad in the output).

- Compare your results with the two STREAM plots shown earlier in this slide deck.

Sample Nested OpenMP Program

```c
#include <omp.h>
#include <stdio.h>
void report_num_threads(int level)
{
    #pragma omp single {
        printf("Level %d: number of threads in the
team: %d\n", level, omp_get_num_threads());
    }
}
int main()
{
    omp_set_dynamic(0);
    #pragma omp parallel num_threads(2) {
        report_num_threads(1);
        #pragma omp parallel num_threads(2) {
            report_num_threads(2);
            #pragma omp parallel num_threads(2) {
                report_num_threads(3);
            }
        }
    }
    return(0);
}
```

% a.out
Level 1: number of threads in the team: 2
Level 2: number of threads in the team: 1
Level 3: number of threads in the team: 1
Level 2: number of threads in the team: 1
Level 3: number of threads in the team: 1

% export OMP_NESTED=true
% export OMP_MAX_ACTIVE_LEVELS=3
% a.out
Level 1: number of threads in the team: 2
Level 2: number of threads in the team: 2
Level 2: number of threads in the team: 2
Level 3: number of threads in the team: 2
Level 3: number of threads in the team: 2
Level 3: number of threads in the team: 2
Level 3: number of threads in the team: 2

Level 0: P0
Level 1: P0 P1
Level 2: P0 P2; P1 P3
Level 3: P0 P4; P2 P5; P1 P6; P3 P7

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
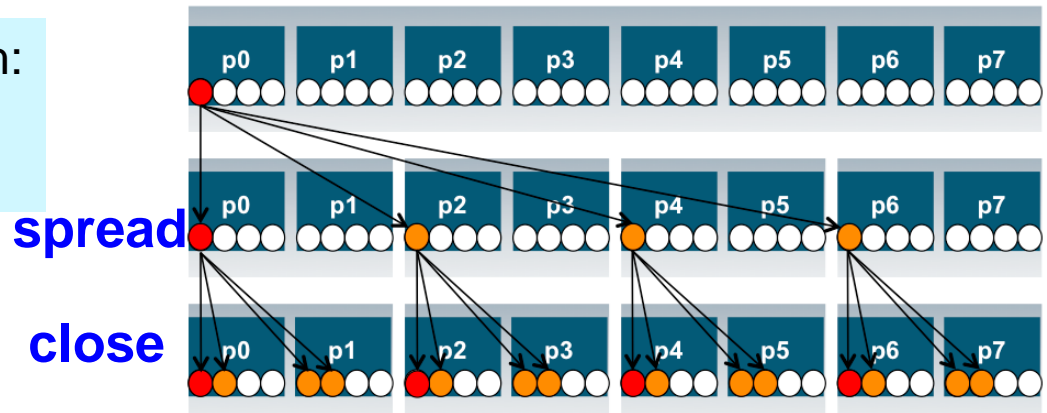
spread

close

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.
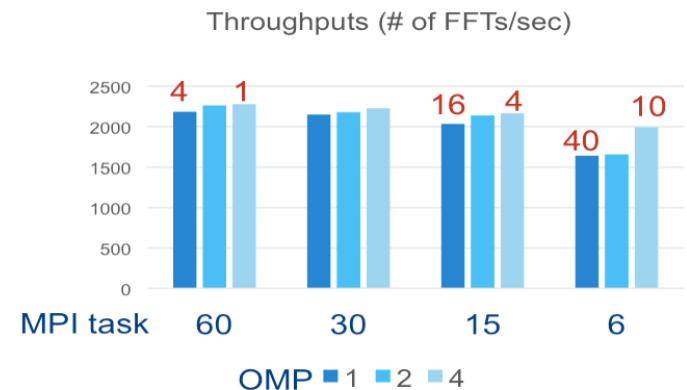
When to Use Nested OpenMP

- Beneficial to use nested OpenMP to allow more fine-grained thread parallelism.
- Some application teams are exploring with nested OpenMP to allow more fine-grained thread parallelism.
  - Hybrid MPI/OpenMP not using node fully packed
  - Top level OpenMP loop does not use all available threads
  - Multiple levels of OpenMP loops are not easily collapsed
  - Certain computational intensive kernels could use more threads
  - MKL can use extra cores with nested OpenMP
- Nested level can be arbitrarily deep.

Use Multiple Threads in MKL

- By Default, in OpenMP parallel regions, only 1 thread will be used for MKL calls.
  - MKL_DYNAMICS is true by default
- Nested OpenMP can be used to enable multiple threads for MKL calls. Treat MKL as a nested inner OpenMP region.
- Sample settings

```
export OMP_NESTED=true
export OMP_PLACES=cores
export OMP_PROC_BIND=sprad,close
export OMP_NUM_THREADS=6,4
export MKL_DYNAMICS=false
export OMP_MAX_ACTIVE_LEVELS=2
```

Throughputs (# of FFTs/sec)

$$N_{MKL} = 240/(N_{MPI} * OMP)$$

FFT3D on KNC, Ng=64^3 *example*

Courtesy of Jeongnim Kim, Intel

*KNC: Intel® Xeon Phi™ processor (Knights Corner) … the first generation co-processor version of the chip.

Example Compute Nodes (Cori KNL*)

- **A Cori KNL node has 68 cores/272 CPUs, 96 GB DDR memory, 16 GB high bandwidth on package memory (MCDRAM).**
- **Three cluster modes, all-to-all, quadrant, sub-NUMA clustering, are available at boot time to configure the KNL mesh interconnect.**

Arrangement of Hardware Threads for 68 Core KNL

Core #	0	1	2	3	...	16	17	18	...	33	34	35	...	50	51	52	...	65	66	67
HW Thread #	0	1	2	3	...	16	17	18	...	33	34	35	...	50	51	52	...	65	66	67
	68	69	70	71	...	84	85	86	...	101	102	103	...	118	119	120	...	133	134	135
	136	137	138	139	...	152	153	154	...	169	170	171	...	186	187	188	...	201	202	203
	204	205	206	207	...	220	221	222	...	237	238	239	...	254	255	256	...	269	270	271

- A quad,cache node has only 1 NUMA node with all CPUs on the NUMA node 0 (DDR memory). The MCDRAM is hidden from the "numactl –H" result since it is a cache.
- A quad,flat node has only 2 NUMA nodes with all CPUs on the NUMA node 0 (DDR memory). And NUMA node 1 has MCDRAM only.
- A snc2,flat node has 4 NUMA domains with DDR memory and all CPUs on NUMA nodes 0 and 1. (NUMA node 0 has physical cores 0 to 33 and all corresponding hyperthreads, and NUMA node 1 has physical cores 34 to 67 and all corresponding hyperthreads). NUMA nodes 2 and 3 have MCDRAM only.

*KNL: Intel® Xeon Phi™ processor 7250 with 68 cores @ 1.4 GHz

Intel KNL Quad,Flat Node Example

Cori KNL quad,flat node example
68 cores (272 CPUs)

% 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 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36
37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73
74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100 101 102 103 104 105 106 107
108 109 110 111 112 113 114 115 116 117 118 119 120 121 122 123 124 125 126 127 128 129 130 131 132 133 134
135 136 137 138 139 140 141 142 143 144 145 146 147 148 149 150 151 152 153 154 155 156 157 158 159 160 161
162 163 164 165 166 167 168 169 170 171 172 173 174 175 176 177 178 179 180 181 182 183 184 185 186 187 188
189 190 191 192 193 194 195 196 197 198 199 200 201 202 203 204 205 206 207 208 209 210 211 212 213 214 215
216 217 218 219 220 221 222 223 224 225 226 227 228 229 230 231 232 233 234 235 236 237 238 239 240 241 242
243 244 245 246 247 248 249 250 251 252 253 254 255 256 257 258 259 260 261 262 263 264 265 266 267 268 269
270 271
node 0 size: 96723 MB
node 0 free: 93924 MB
node 1 cpus:
node 1 size: 16157 MB
node 1 free: 16088 MB
node distances:
node   0  1
  0:  10 31
  1:  31 10
```

- The quad,flat mode has only 2 NUMA nodes with all CPUs on the NUMA node 0 (DDR memory).
- And NUMA node 1 has MCDRAM (high bandwidth memory).

Exercise: Affinity Choices and Results

- Pure OpenMP code: xthi-omp.c

- Nested OpenMP code: xthi-nested-omp.c

- Sample output:

  % ./xthi-omp |sort –k4n

  Hello from thread 0, on nid00011. (core affinity = 0)

  Hello from thread 1, on nid00011. (core affinity = 4)

  Hello from thread 2, on nid00011. (core affinity = 8) ...

- Experiment with different OMP_NUM_THREADS, OMP_PROC_BIND, and OMP_PLACES settings to check thread affinity on different compute node architectures (for example, Cori Haswell and KNL).

Essential runtime settings for KNL MCDRAM Memory Affinity

- In quad, cache mode, no special setting is needed to use MCDRAM

- In quad,flat mode, using quad,flat as an example
  - NUMA node 1 is MCDRAM

- Enforced memory mapping to MCDRAM
  - If using >16 GB, malloc will fail
  - Use "**numactl -m 1 ./myapp**" as the executable (instead of "./myapp")

- Preferred memory mapping to MCDRAM:
  - If using >16 GB, malloc will spill to DDR
  - Use "**numactl -p 1 ./myapp**" as the executable (instead of "./myapp")

164

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.
- Use numactl -m or -p option to explicitly request memory allocation in specific NUMA domain (for example: high bandwidth memory in KNL)

*KNL: Intel® Xeon Phi™ processor 7250 with 68 cores @ 1.4 GHz

Appendices

➡ • Challenge Problems
• Challenge Problems: solutions
  – Monte Carlo PI and random number generators
  – Molecular dynamics
  – Matrix multiplication
  – Linked lists
  – Recursive matrix multiplication
• Fortran and OpenMP

Challenge problems

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

Challenge 1: Molecular dynamics

- The code supplied is a simple molecular dynamics simulation of the melting of solid argon

- Computation is dominated by the calculation of force pairs in subroutine `forces` (in forces.c)

- Parallelise this routine using a parallel for construct and atomics; think carefully about which variables should be SHARED, PRIVATE or REDUCTION variables

- Experiment with different schedule kinds

Challenge 1: MD (cont.)

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

Challenge 1 MD: (cont.)

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

Challenge 2: Matrix multiplication

- Parallelize the matrix multiplication program in the file matmul.c

- Can you optimize the program by playing with how the loops are scheduled?

- Try the following and see how they interact with the constructs in OpenMP
  - Alignment
  - Cache blocking
  - Loop unrolling
  - Vectorization

- Goal: Can you approach the peak performance of the computer?

Challenge 3: Traversing linked lists

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

- Parallelize this program two different ways
  1. Use OpenMP tasks
  2. Use anything you choose in OpenMP *other than* tasks.

- The second approach (no tasks) can be difficult and may take considerable creativity in how you approach the problem (why its such a pedagogically valuable problem)

Challenge 4: Recursive matrix multiplication

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

Challenge 4: Recursive matrix multiplication

- Quarter each input matrix and output matrix
- Treat each submatrix as a single element and multiply
- 8 submatrix multiplications, 4 additions
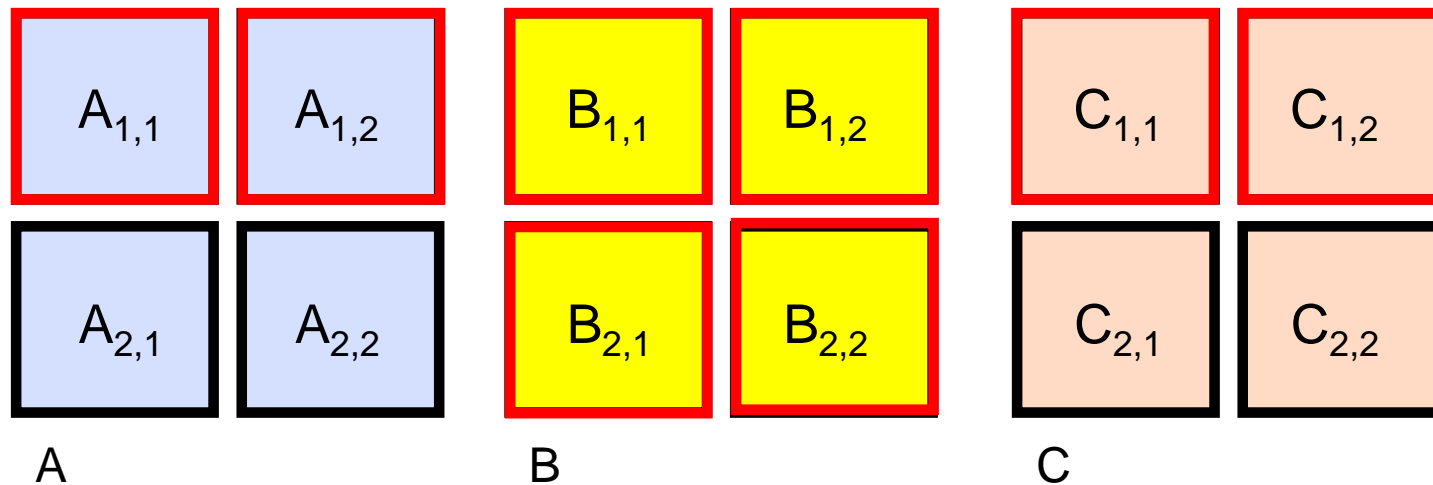
$$C_{1,1} = A_{1,1} \cdot B_{1,1} + A_{1,2} \cdot B_{2,1}$$
$$C_{2,1} = A_{2,1} \cdot B_{1,1} + A_{2,2} \cdot B_{2,1}$$

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

Challenge 4: Recursive matrix multiplication How to multiply submatrices?

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

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

$\mathbf{A}_{1,1}$

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

$\mathbf{B}_{1,1}$

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

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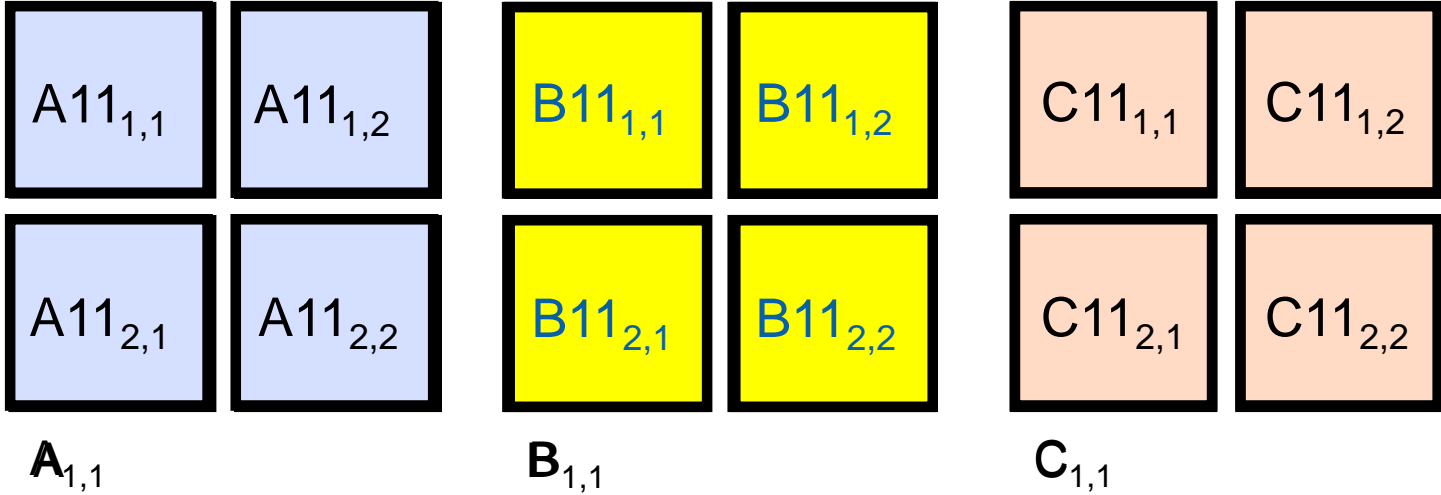
$C_{1,1} = A_{1,1} \cdot B_{1,1} + A_{1,2} \cdot B_{2,1}$

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

Challenge 4: Recursive matrix multiplication Recursively multiply submatrices

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

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

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

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

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

- Also need stopping criteria for recursion

Appendices

- Challenge Problems
- Challenge Problems: solutions
  - → Monte Carlo PI and random number generators
  - Molecular dynamics
  - Matrix multiplication
  - Linked lists
  - Recursive matrix multiplication
- Fortran and OpenMP

Computers and random numbers

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

Monte Carlo Calculations

Using Random numbers to solve tough problems

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

2 * r
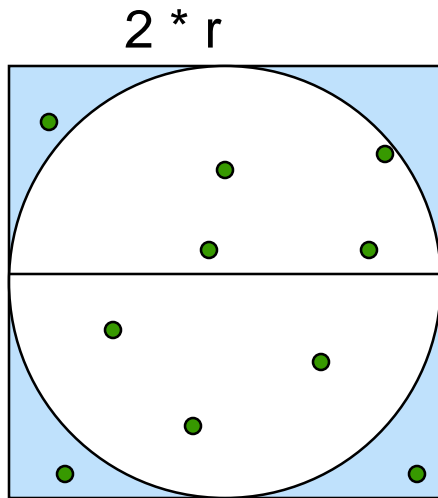
- Throw darts at the circle/square.
- Chance of falling in circle is proportional to ratio of areas:

$$A_c = r^2 * \pi$$

$$A_s = (2*r) * (2*r) = 4 * r^2$$

$$P = A_c/A_s = \pi /4$$

- Compute π by randomly choosing points, count the fraction that falls in the circle, compute pi.

N= 10	π = 2.8
N=100	π = 3.16
N= 1000	π = 3.148

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:

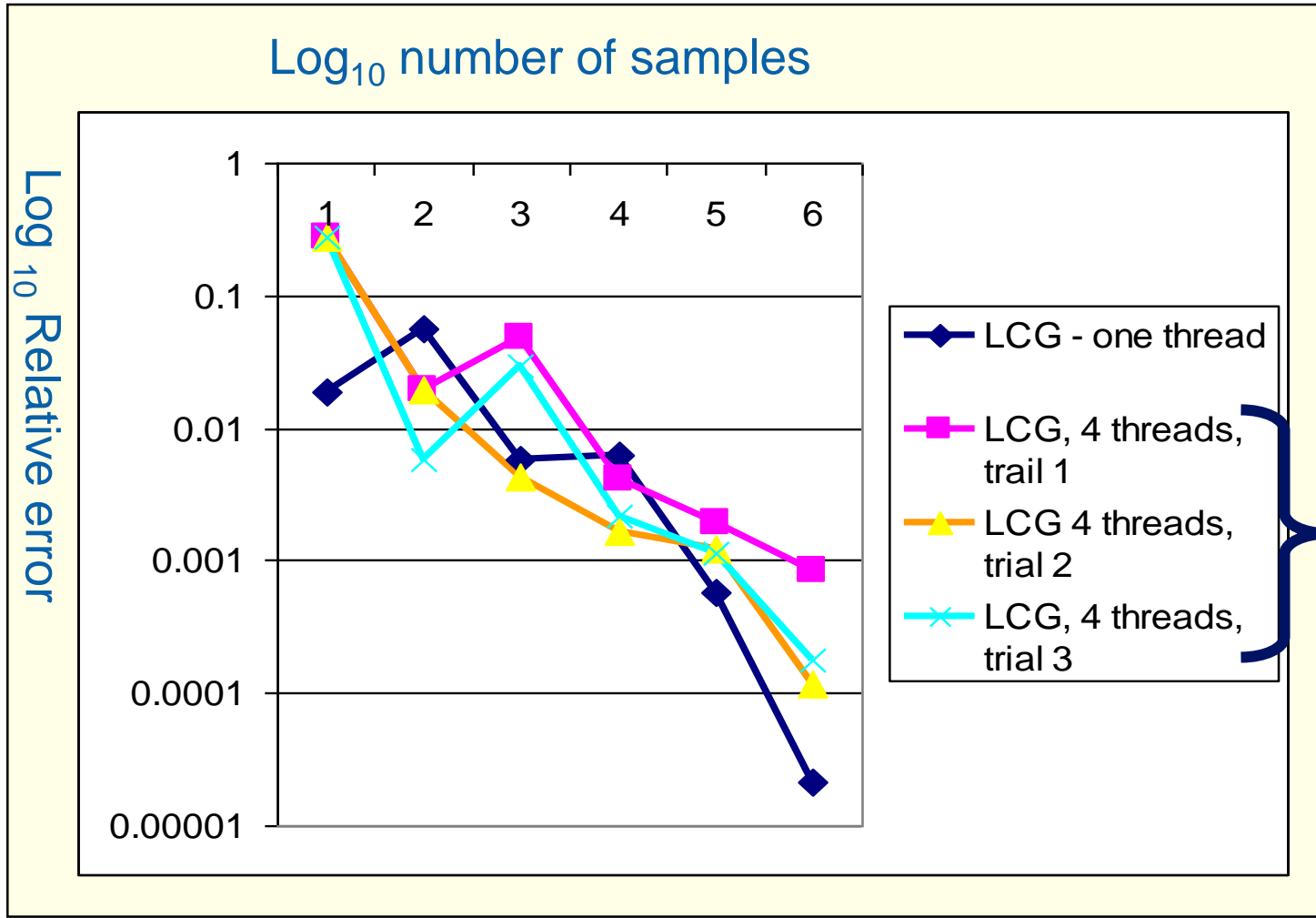   ◆ MULTIPLIER = 1366

    ◆ 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.

183

LCG code: threadsafe version

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

   random_next = (MULTIPLIER  * random_last + AD
   random_last = random_next;

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

random_last carries state between random number computations,

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

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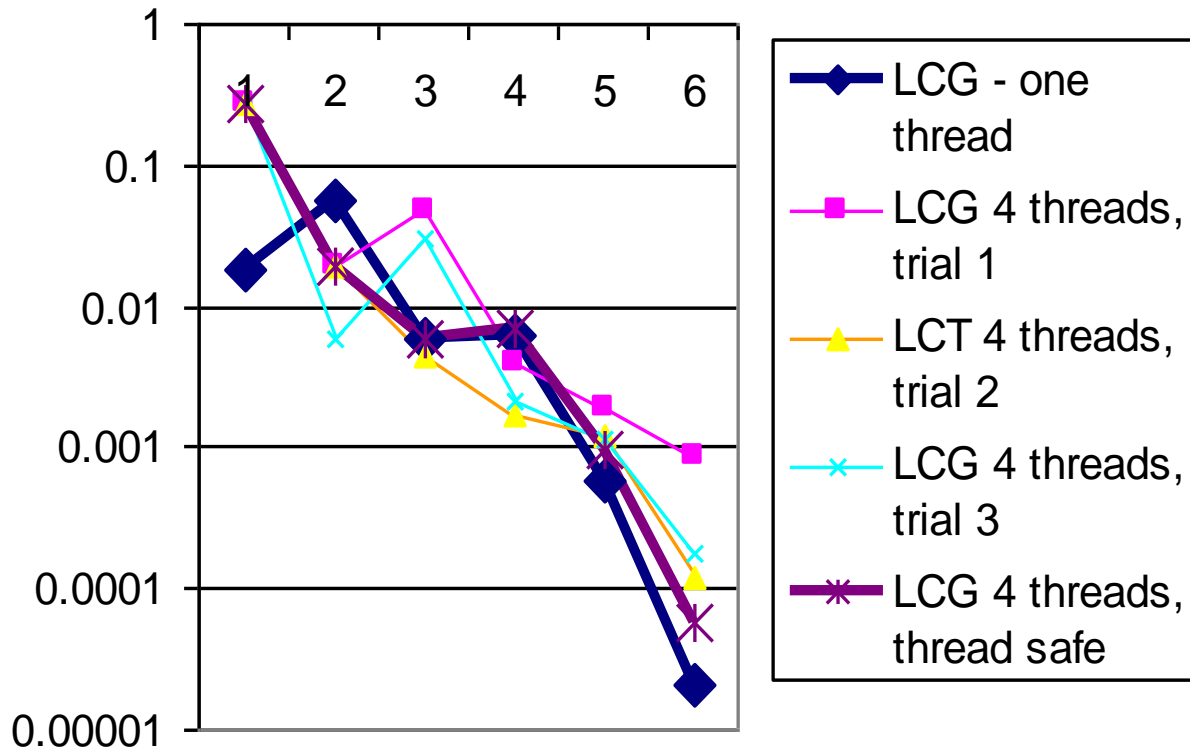
Thread safe version gives the same answer each time you run the program.

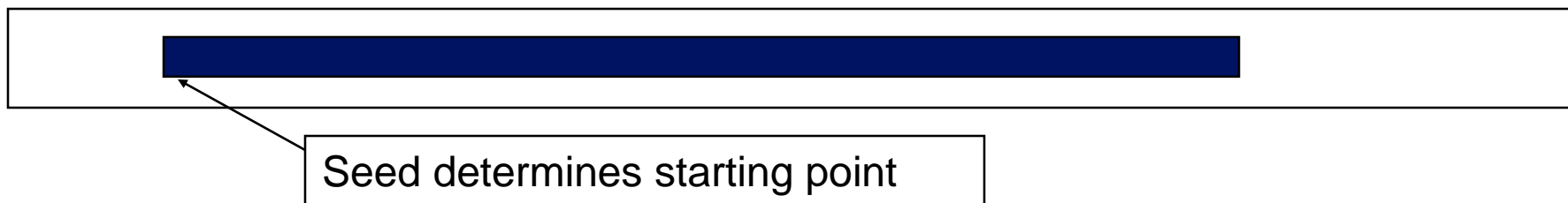But for large number of samples, its quality is lower than the one thread result!

Why?

185

Pseudo Random Sequences

- Random number Generators (RNGs) define a sequence of pseudo-random numbers of length equal to the period of the RNG
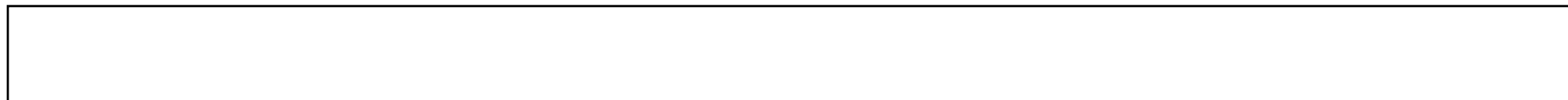
  - In a typical problem, you grab a subsequence of the RNG range

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.

Thread 1
Thread 2
Thread 3
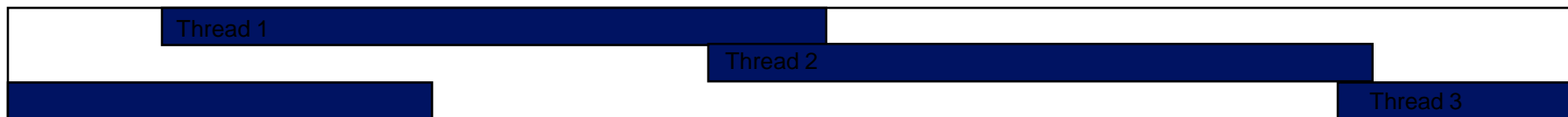
  - Overlapping sequences = over-sampling and bad statistics … lower quality or even wrong answers!

Parallel random number generators

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

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

Nice for debugging, but not really needed scientifically.

Intel's Math kernel Library supports all of these methods.

MKL Random number generators (RNG)

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

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

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

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

vslDeleteStream( &stream );
```

Initialize a stream or pseudo random numbers

Select type of RNG and set seed

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

Delete the stream when you are done

Wichmann-Hill generators (WH)

- WH is a family of 273 parameter sets each defining a non-overlapping and independent RNG.

- Easy to use, just make each stream threadprivate and initiate RNG stream so each thread gets a unique WG RNG.

```
VSLStreamStatePtr stream;

#pragma omp threadprivate(stream)

                     …

vslNewStream(&ran_stream, VSL_BRNG_WH+Thrd_ID, (int)seed);
```

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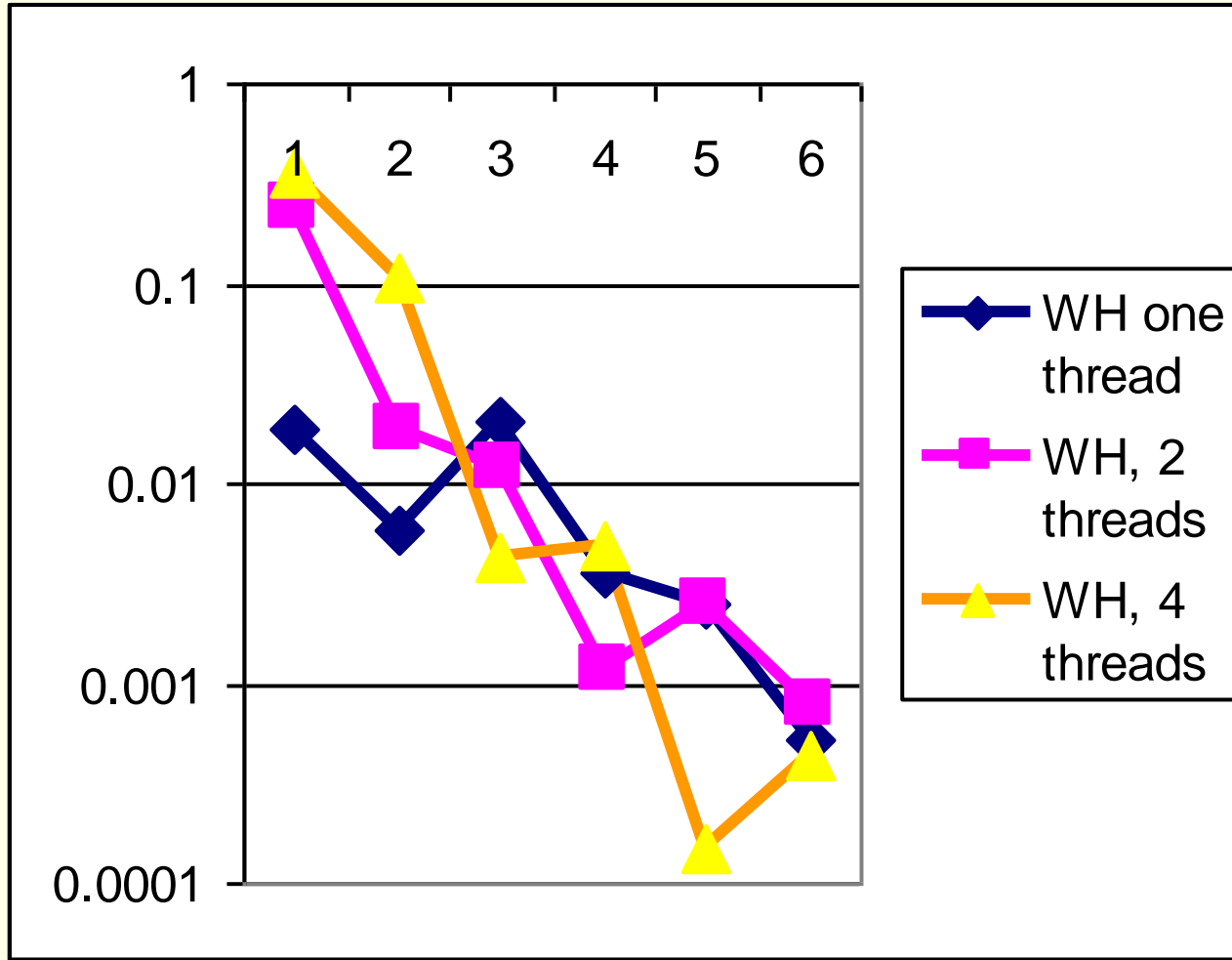
Notice that once you get beyond the high error, small sample count range, adding threads doesn't decrease quality of random sampling.

Leap Frog method

- Interleave samples in the sequence of pseudo random numbers:
    - Thread i starts at the 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.

Appendices

- Challenge Problems
- Challenge Problems: solutions
  - Monte Carlo PI and random number generators
  - Molecular dynamics
  - Matrix multiplication
  - Linked lists
  - Recursive matrix multiplication
- Fortran and OpenMP

Molecular dynamics: Solution

> Compiler will warn you if you have missed some variables

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

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

Molecular dynamics : Solution (cont.)

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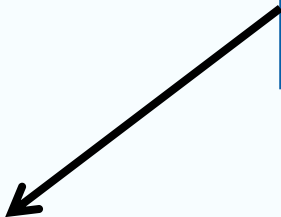
All updates to f must be atomic

Molecular dynamics : With orphaning

```
#pragma omp single
{
    vir    = 0.0;
    epot   = 0.0;
}
#pragma omp for reduction(+:epot,vir)  schedule (static,32)
    for (int i=0; i<npart*3; i+=3) {
.........
```
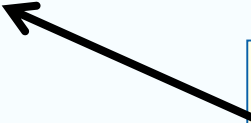
Implicit barrier needed to avoid race condition with update of reduction variables at end of the for construct

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

Molecular dynamics : With array reduction

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

Replace atomics with accumulation into array with extra dimension

Molecular dynamics : With array reduction

….

#pragma omp for

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

Reduction can be done in parallel

Zero ftemp for next time round

Appendices

- Challenge Problems
- Challenge Problems: solutions
  - Monte Carlo PI and random number generators
  - Molecular dynamics
  - → Matrix multiplication
  - Linked lists
  - Recursive matrix multiplication
- Fortran and OpenMP

Challenge: Matrix Multiplication

- Parallelize the matrix multiplication program in the file matmul.c

- Can you optimize the program by playing with how the loops are scheduled?

- Try the following and see how they interact with the constructs in OpenMP
  - Cache blocking
  - Loop unrolling
  - Vectorization

- Goal: Can you approach the peak performance of the computer?

Matrix multiplication

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

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

- On a dual core laptop

    - 13.2 seconds 153 Mflops one thread

    - 7.5 seconds 270 Mflops two threads

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

Appendices

- Challenge Problems
- Challenge Problems: solutions
  - Monte Carlo PI and random number generators
  - Molecular dynamics
  - Matrix multiplication
  - Linked lists
  - Recursive matrix multiplication
- Fortran and OpenMP

Exercise: traversing linked lists

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

Linked lists with tasks

- See the file Linked_omp3_tasks.c

```
#pragma omp parallel
{

   #pragma omp single
  {

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

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

Exercise: traversing linked lists

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

Linked lists without tasks

- See the file Linked_omp25.c

```
while (p != NULL) {
    p = p->next;
     count++;
}
p = head;
for(i=0; i<count; i++) {
    parr[i] = p;
    p = p->next;
  }
#pragma omp parallel
{
    #pragma omp for schedule(static,1)
    for(i=0; i<count; i++)
      processwork(parr[i]);
}
```

> Count number of items in the linked list

> Copy pointer to each node into an array

> Process nodes in parallel with a for loop

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

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

Linked lists without tasks: C++ STL

- See the file Linked_cpp.cpp

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

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

Copy pointer to each node into an array

Count number of items in the linked list

Process nodes in parallel with a for loop

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

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

Appendices

- Challenge Problems
- Challenge Problems: solutions
    - Monte Carlo PI and random number generators
    - Molecular dynamics
    - Matrix multiplication
    - Linked lists
    → - Recursive matrix multiplication
- Fortran and OpenMP

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

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

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

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

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

   }
}
```

Appendices

- Challenge Problems
- Challenge Problems: solutions
  - Monte Carlo PI and random number generators
  - Molecular dynamics
  - Matrix multiplication
  - Linked lists
  - Recursive matrix multiplication
- Fortran and OpenMP

Fortran and OpenMP

- We were careful to design the OpenMP constructs so they cleanly map onto C, C++ and Fortran.

- There are a few syntactic differences that once understood, will allow you to move back and forth between languages.

- In the specification, language specific notes are included when each construct is defined.

OpenMP:
Some syntax details for Fortran programmers

- Most of the constructs in OpenMP are compiler directives.
  - For Fortran, the directives take one of the forms:
    C$OMP *construct [clause [clause]…]*
    !$OMP *construct [clause [clause]…]*
    *$OMP *construct [clause [clause]…]*
- The OpenMP include file and lib module
    use omp_lib
    Include omp_lib.h

OpenMP:
Structured blocks (Fortran)

– Most OpenMP constructs apply to structured blocks.

– Structured block: a block of code with one point of entry at the top and one point of exit at the bottom.

– The only "branches" allowed are STOP statements in Fortran and exit() in C/C++.

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

A structured block

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

Not A structured block

OpenMP:
Structured Block Boundaries

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

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

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

- The "construct/end construct" pairs is done anywhere a structured block appears in Fortran. Some examples:

  - DO … END DO
  - PARALLEL … END PARREL
  - CRICITAL … END CRITICAL
  - SECTION … END SECTION

  - SECTIONS … END SECTIONS
  - SINGLE … END SINGLE
  - MASTER … END MASTER

Runtime library routines

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

OpenMP compilers on Apple laptops: MacPorts

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

```
sudo port selfupdate
```
Update to latest version of MacPorts

```
sudo port install gcc6
```
Grab version 6 gnu compilers (5-10 mins)

```
port select --list gcc
```
List versions of gcc on your system

```
sudo port select –set gcc mp-gcc6
```
Select the mp enabled version of the most recent gcc release

```
gcc –fopenmp hello.c
```
Test the installation with a simple program

OpenMP compilers on Apple laptops: Homebrew

- An alternate way to use OpenMP on your Apple laptop:
- Install Homebrew. If Hombrew is already installed, skip to the install gcc section.

```
echo $SHELL
```

Check that you are running bash shell for ruby. Use the ruby to install homebrew.

```
/usr/bin/ruby -e "$(curl -fsSL
https://raw.githubusercontent.com/Homebrew/install/master/install)"
```

```
brew install gcc --without-multilib
```

Install a homebrew version of gcc without multilib, and locate it

```
which gcc-7
```

In my case, hombrew installed a new version of gcc called it gcc-7

```
gcc-7 -fopenmp hello_par.c
./a.out
export OMP_NUM_THREADS=8
./a.out
```

Test the installation with a simple program

Slides and exercises at:
http://www.nersc.gov/users/software/programming-models/openmp/sc17-openmp