The OpenMP* Common Core:
A hands-on Introduction

Tim Mattson
Intel Corp.

Yun (Helen) He
Berkeley Lab

Alice Koniges
Univ. of Hawai'i

David Eder
Univ. of Hawai'i

Download tutorial materials onto your laptop:
git clone https://github.com/tgmattso/OmpCommonCore.git

Outline

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

OpenMP* Overview

```
C$OMP FLUSH
#pragma omp critical
#pragma omp single
C$OMP THREADPRIVATE(/ABC/)
C$OMP ATOMIC
CALL OMP_SET_NUM_THREADS(10)
```

OpenMP: An API for Writing Parallel Applications

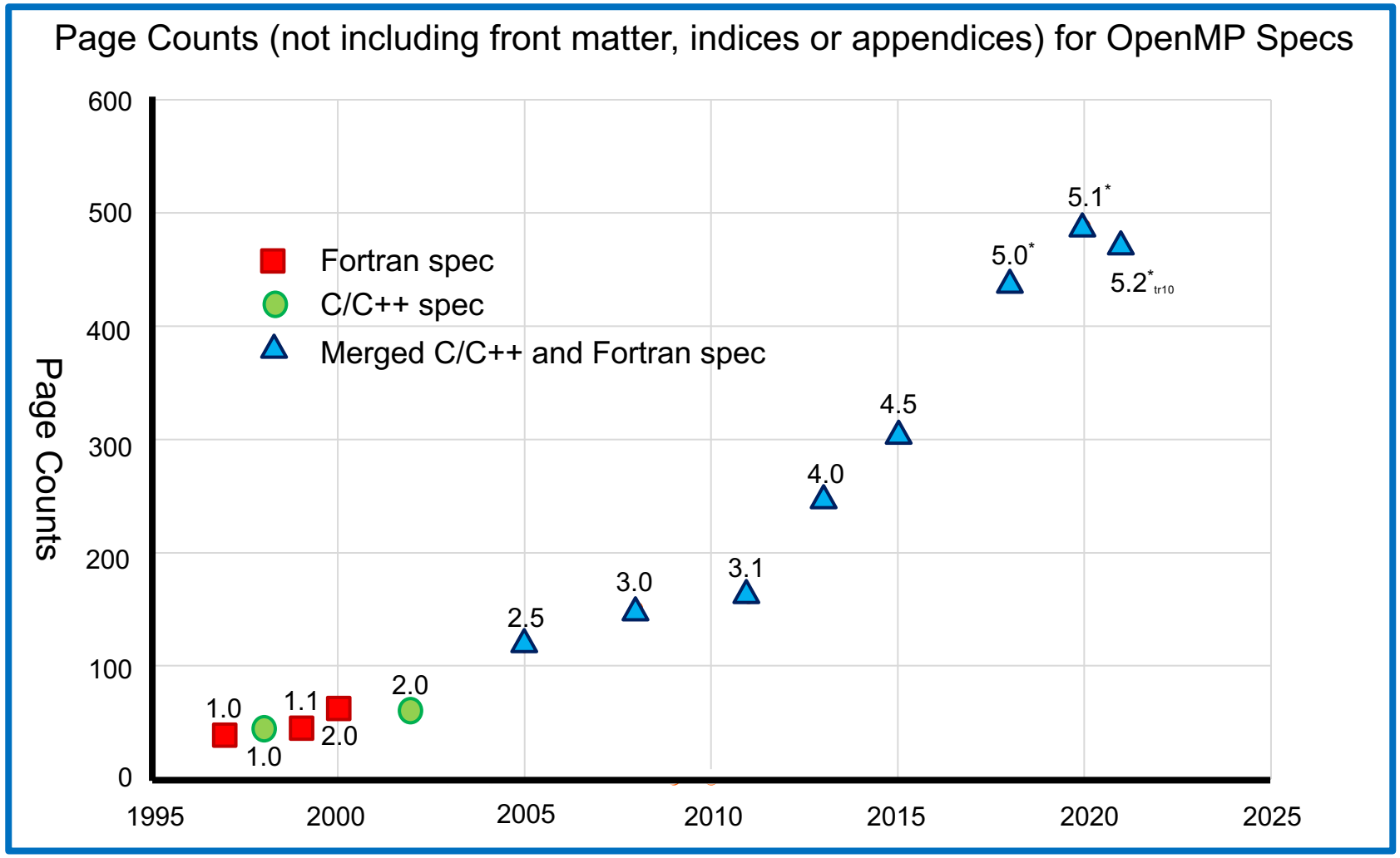
- A set of compiler directives and library routines for parallel application programmers
- Greatly simplifies writing multi-threaded (MT) programs in Fortran, C and C++
- Also supports non-uniform memories, vectorization and GPU programming

```
#pragma omp parallel for private(A, B)
C$OMP PARALLEL  REDUCTION (+: A, B)
C$OMP PARALLEL COPYIN(/blk/)
C$OMP DO lastprivate(XX)
#pragma omp atomic seq_cst
Nthrds = OMP_GET_NUM_PROCS()
omp_set_lock(lck)
```

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

The Growth of Complexity in OpenMP

Our goal in 1997 … A simple interface for application programmers



The full spec is overwhelming. We focus on the Common Core: the 21 items most people restrict themselves to

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

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

OpenMP Basic Definitions: Basic Solution Stack

User layer

End User

Application

Prog.

Directives, Compiler

OpenMP library

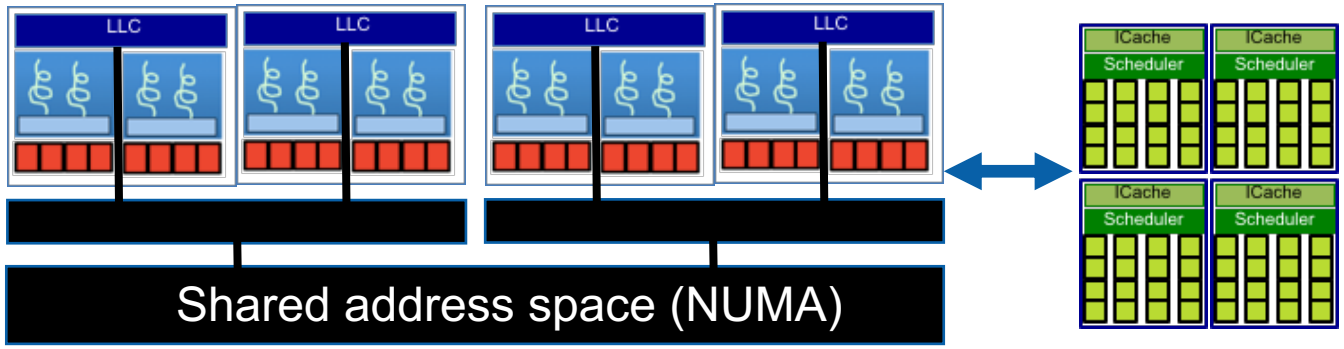
Environment variables

System layer

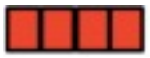
OpenMP Runtime library

OS/system support for shared memory and threading

HW

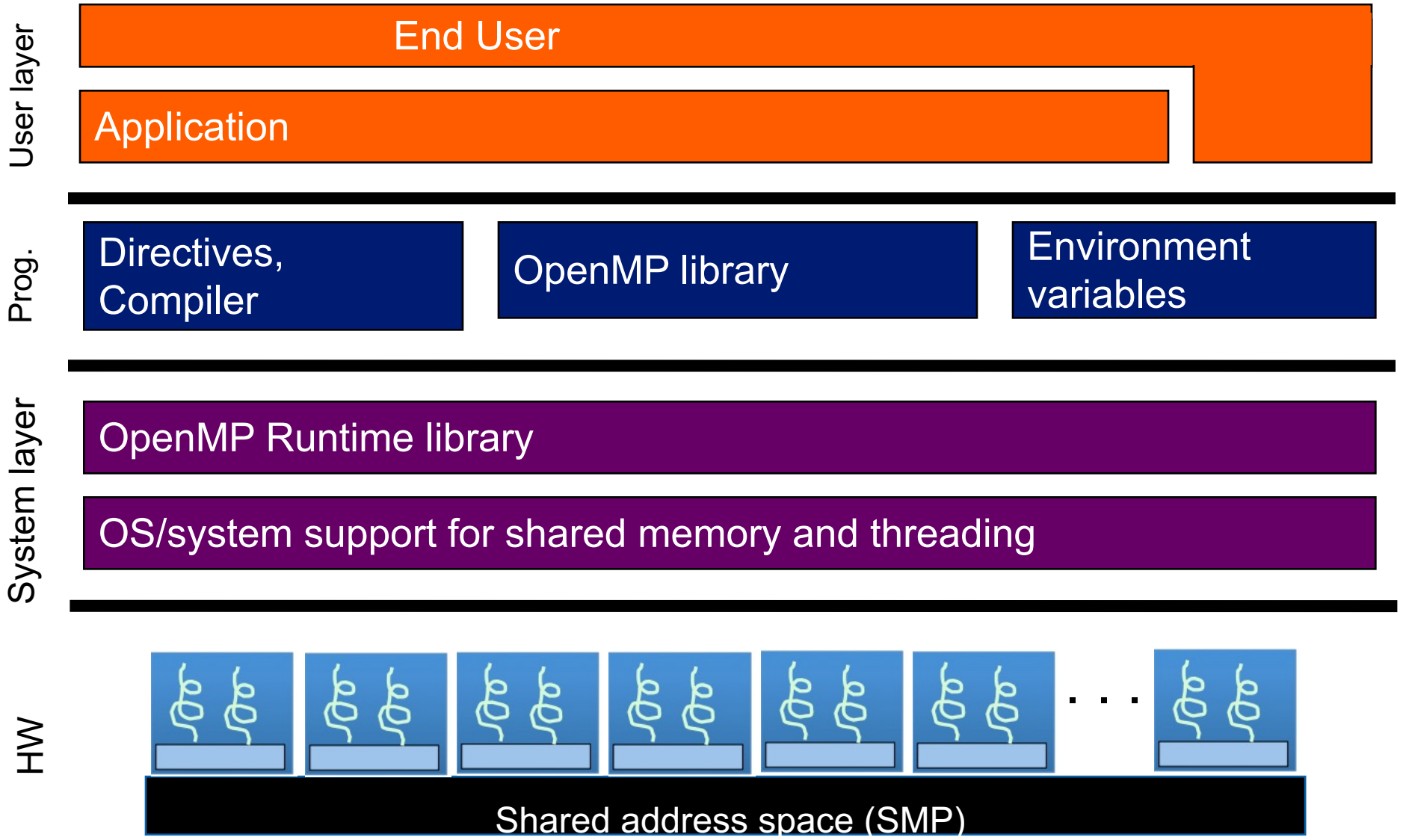


CPU cores 

SIMD units 

GPU cores 

OpenMP Basic Definitions: Basic Solution Stack



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

OpenMP Basic Syntax

- Most of the constructs in OpenMP are compiler directives.

C and C++	Fortran
Compiler directives	
#pragma omp construct [clause [clause]…]	***!$OMP construct [clause [clause] …]***
Example	
#pragma omp parallel private(x) ***{*** ***}***	***!$OMP PARALLEL PRIVATE(X)*** ***!$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/OpenMPCommonCore.git
```

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




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

}
```

- To download the slides:

https://github.com/tgmattso/OmpCommonCore.git

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_Common_Core.git

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


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

Switches for compiling and linking

gcc -fopenmp	**Gnu (Linux, OSX)**
cc -qopenmp	**Intel (Linux@NERSC)**
icl /Qopenmp	**Intel (windows)**
icc -fopenmp	**Intel (Linux, OSX)**

Solution

A Multi-Threaded “Hello World” Program

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

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

#pragma omp parallel
 {



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

OpenMP include file

Parallel region with default number of threads

End of the Parallel region

Sample Output:

hello hello world

world

hello hello world

world

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

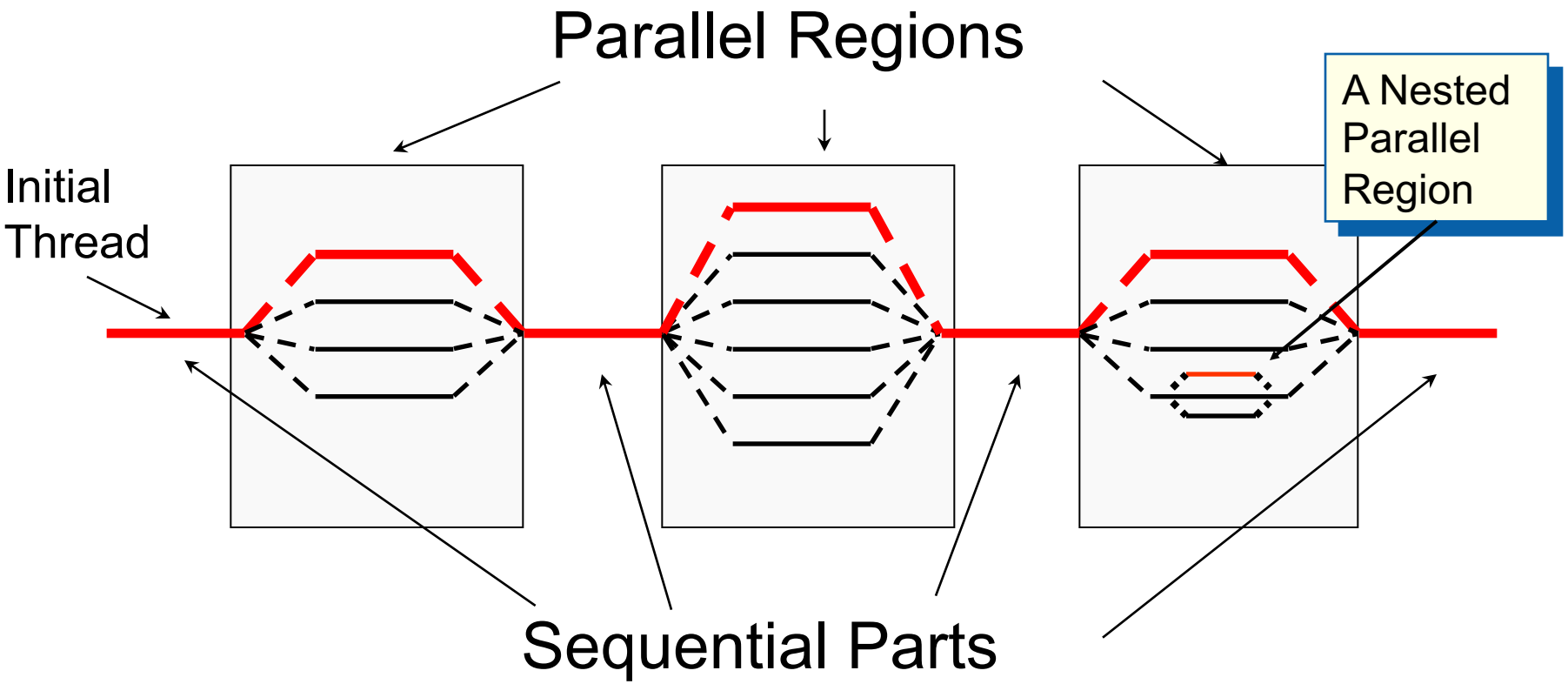
Outline

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

OpenMP Execution model:

Fork-Join Parallelism:

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



Thread Creation: Parallel Regions

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

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

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

Runtime function to request a certain number of threads

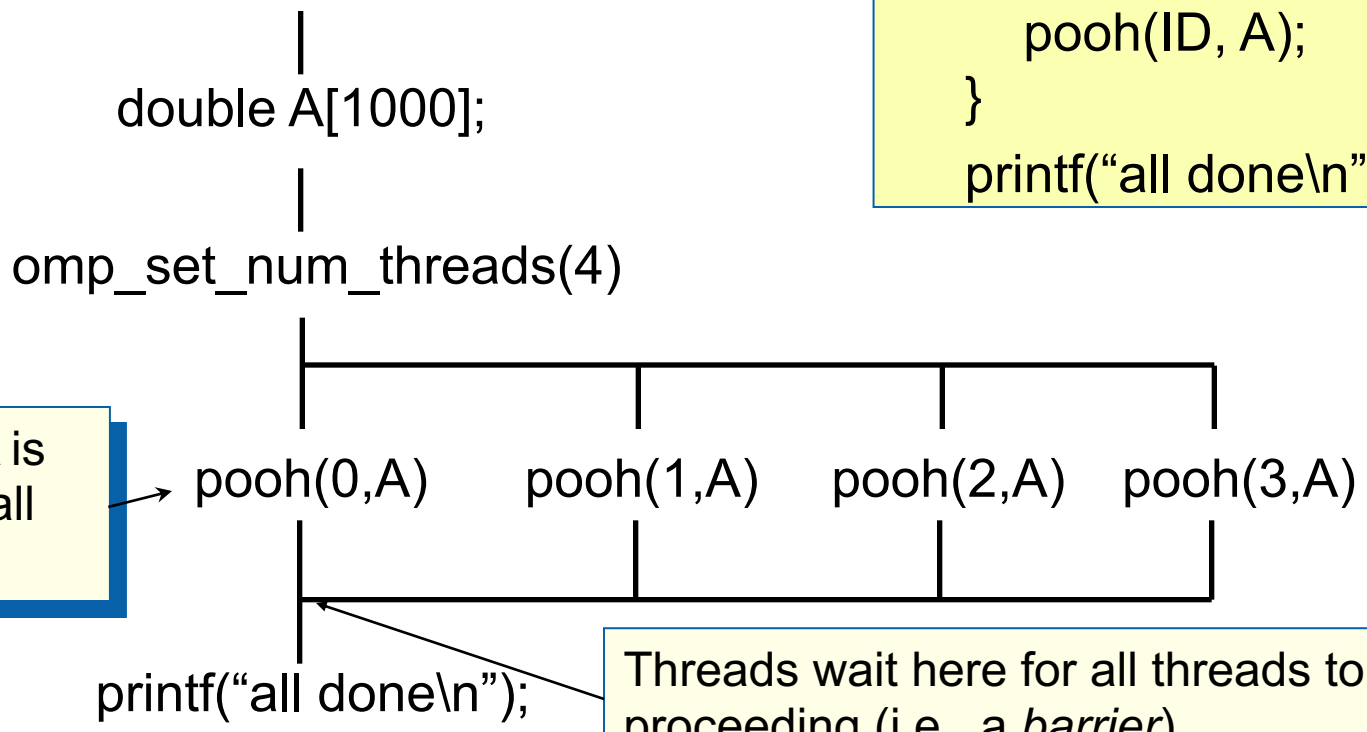
Runtime function returning a thread ID

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

Thread Creation: Parallel Regions Example

- Each thread executes the same code redundantly.

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



A single copy of A is shared between all threads.

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

Thread creation: How many threads did you actually get?

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

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

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

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

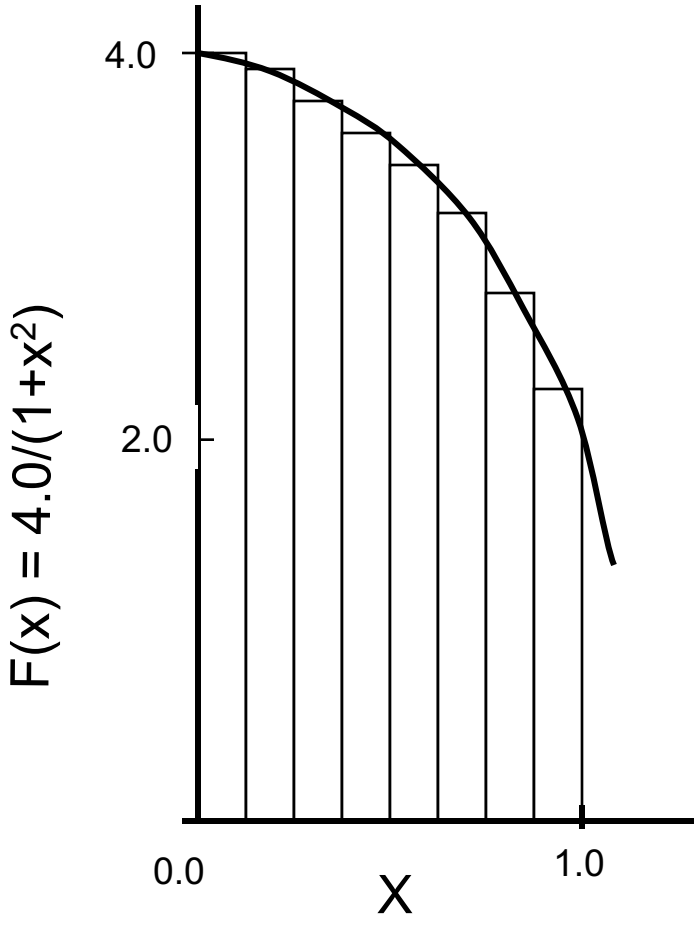
Runtime function to request a certain number of threads

Runtime function to return actual number of threads in the team

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

An Interesting Problem to Play With

Numerical Integration



Mathematically, we know that:

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

We can approximate the integral as a sum of rectangles:

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

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

Serial PI Program

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

            step = 1.0/(double) num_steps;

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

See ParProgCourse/Exercises/OpenMP/pi.c

Serial PI Program

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

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

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

See ParProgCourse/Exercises/OpenMP/pi.c

Exercise: the Parallel Pi Program

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

  #pragma omp parallel

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

Number of threads in the team

Thread ID or rank

Time in seconds since a fixed point in the past

Request a number of threads in the team

Hints: the Parallel Pi Program

- Use a parallel construct:

```
#pragma omp parallel
```

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

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

Example: A simple SPMD pi program

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

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

Only one thread should copy the number of threads to the global value to make sure multiple threads writing to the same address don't conflict.

This is a common trick in SPMD programs to create a cyclic distribution of loop iterations

Results*

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

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

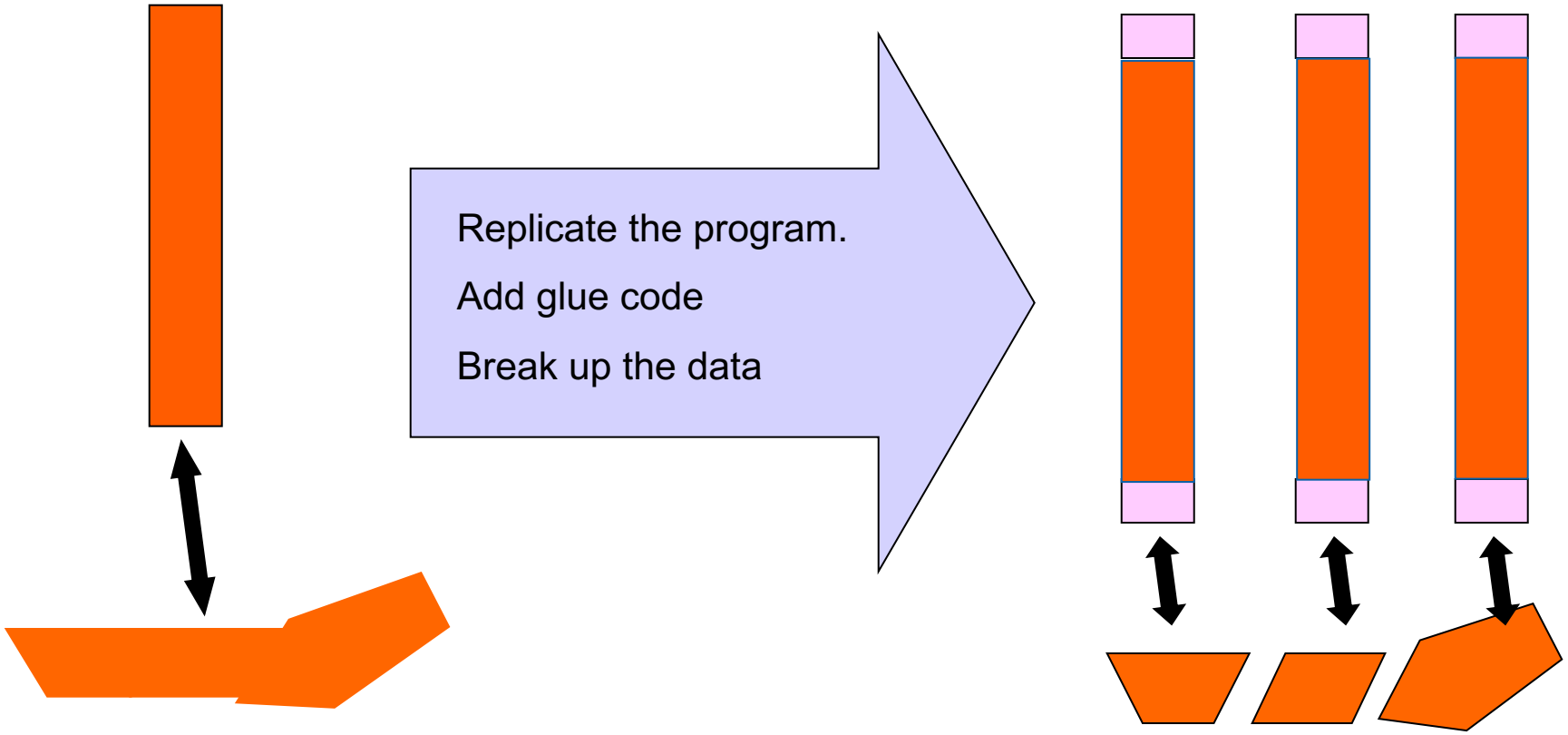
threads	1st SPMD*
1	1.86
2	1.03
3	1.08
4	0.97

Intel compiler (icpc) with 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.

*SPMD: Single Program Multiple Data

SPMD: Single Program Mulitple Data

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

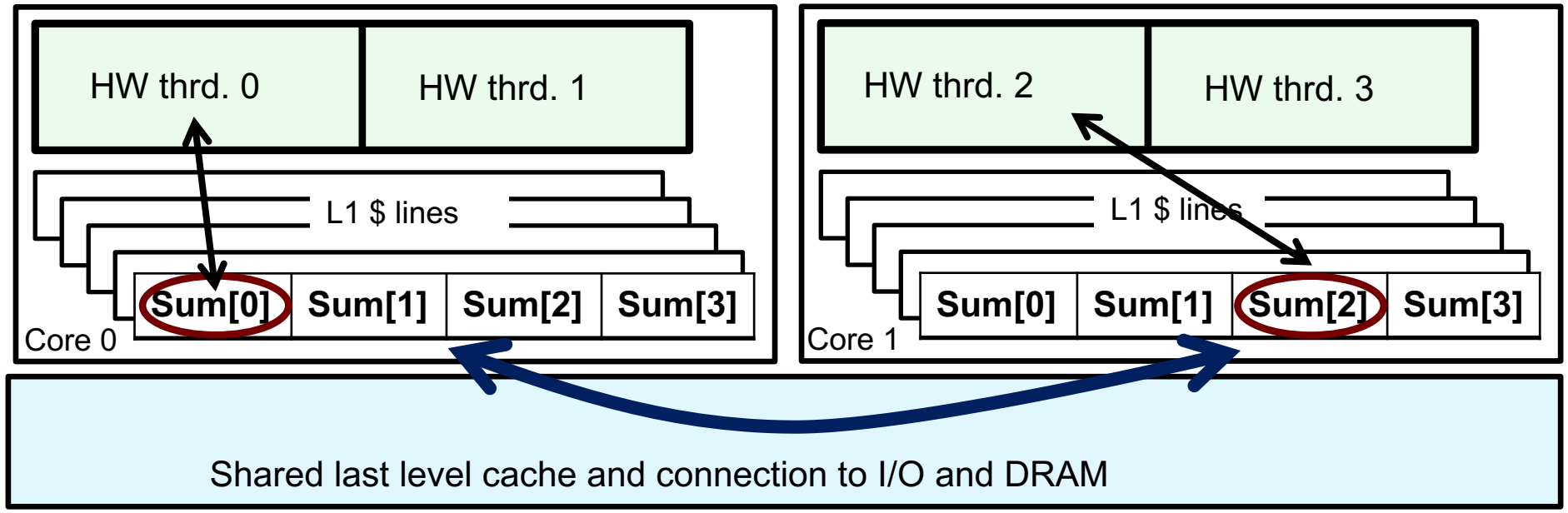


- Use the rank … an ID ranging from 0 to (P-1) … to select between a set of tasks and to manage any shared data structures.

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"**.



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

Example: Eliminate false sharing by padding the sum array

```
#include <omp.h>
static long num_steps = 100000;        double step;
#define NUM_THREADS 2
#define PAD 8        // assume 64 byte L1 cache line size
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.

```
#include <omp.h>
static long num_steps = 100000;        double step;
#define NUM_THREADS 2
#define PAD 8        // assume 64 byte L1 cache line size
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.

Four Ways to Request a Number of Threads for a Parallel Region

1. The system has an Internal Control Variable (ICV) that defines the default number of threads to request for a parallel region.

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

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

4. 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)

Order of Precedence for setting the requested number of threads: (4) > (3) > (2) > (1)

Outline

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

Synchronization

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

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

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

Synchronization: critical

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

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

}
```

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

Synchronization: barrier

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

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

omp_set_num_threads(8)

#pragma omp parallel

{    int id, nthrds;

     id = omp_get_thread_num();

     nthrds = omp_get_num_threads();

     if (id==0) numthrds = nthrds;

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

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

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

PI Program with False Sharing

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

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

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

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

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

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

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

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

No array, so no false sharing.

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

Results*: pi program critical section

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

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

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

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

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

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

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

Synchronization

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

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

Covered earlier

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

Covered in this section

Synchronization: Atomic

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

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

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

Synchronization: Atomic

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

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

Atomic only protects the read/update of X

The OpenMP 3.1 Atomics (1 of 2)

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

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

- Atomic can protect loads

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

- Atomic can protect stores

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

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

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

This is the original OpenMP atomic

The OpenMP 3.1 Atomics (2 of 2)

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

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

- Where the statement is one of the following forms:

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

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

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

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

Synchronization: Lock Routines

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

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

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

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

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

Synchronization: Simple Locks Example

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

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

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

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

One lock per case … even and odd

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

Free-up storage when done.

Exercise

- In the file hist.c, we provide a program that generates a large array of random numbers and then generates a histogram of values.

- This is a ”quick and informal” way to test a random number generator … if all goes well the bins of the histogram should be the same size.

- Parallelize the filling of the histogram You must assure that your program is race free and gets the same result as the sequential program.

- Using everything we’ve covered today, manage updates to shared data in two different ways. Try to minimize the time to generate the histogram.

- Time ONLY the assignment to the histogram. Can you beat the sequential time?

Histogram Program: Critical section

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

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

Easy to write and correct, but terrible performance

Histogram program: one lock per histogram bin

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

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

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

One lock per element of hist

Enforce mutual exclusion on update to hist array

Free-up storage when done.

Histogram program: reduction with an array

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

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

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

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

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

Outline

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

The Loop Worksharing Construct

- 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 Construct

A motivating example

Sequential code

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

OpenMP parallel region

(SPMD Pattern)

```
#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)-1;
        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.
- Example:
  - #pragma omp for schedule(dynamic, 10)

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

Loop Worksharing Constructs: The schedule clause

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

OpenMP 4.5 added modifiers monotonic, nonmontonic and simd.

Loop Worksharing Constructs: The schedule clause

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

Least work at runtime : scheduling done at compile-time

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

Combined Parallel/Worksharing Construct

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

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

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

These are equivalent

Working with loops

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

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

Note: loop index "i" is private by default

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

- How do we handle this case?

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

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

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

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

Reduction

- OpenMP reduction clause:

  reduction (op : list)

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

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

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

OpenMP: Reduction operands/initial-values

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

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

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

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

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

Example: PI with a loop and a reduction

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

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

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

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

Example: PI with a loop and a reduction

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

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

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

Results*: PI with a loop and a reduction

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

Example: Pi with a

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

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

*Intel compiler (icpc) with no optimization on Apple OS X 10.7.3 with a dual core (four HW thread) Intel® Core™ 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 it's 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

The Loop Worksharing Constructs

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

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

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

Loop construct name:

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

Nested Loops

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

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

Number of loops to be parallelized, counting from the outside

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

Sections Worksharing Construct

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

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

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

Array Sections with Reduce

```
#include <stdio.h>
#define N 100
void init(int n, float (*b)[N]);
int main(){
int i,j; float a[N], b[N][N]; init(N,b);
for(i=0; i<N; i++)  a[i]=0.0e0;

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

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

Exercise

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

Optimizing mandel.c

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

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

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

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