



The OpenMP* Common Core: A hands on exploration

Barbara Chapman

Stony Brook University

Barbara.chapman@stonybrook.edu

Alice Koniges and Helen He

LBL

AEKoniges@lbl.gov

Larry Meadows and Tim Mattson

Intel

Lawrence.f.meadows@intel.com

Many others have contributed to these slides. The first version of the “Common Core” slides were created by Tim Mattson, Intel Corp.

About The Presenters

- **Barbara Chapman** is a Professor at Stony Brook University. She has been involved with OpenMP since 2000.
- **Alice Koniges** is a computer scientist and physicist at Berkeley Lab's Computational Research Division. She represents Berkeley on the OpenMP and MPI standards committees.

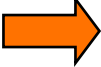
Preliminaries: Part 1

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

Preliminaries: Part 2

- Our plan for the day .. Active learning!
 - We will mix short lectures with short exercises.
 - You will use your laptop to connect to a multiprocessor server.
- Please follow these simple rules
 - Do the exercises that we assign and then change things around and experiment.
 - Embrace active learning!
 - Don't cheat: Do Not look at the solutions before you complete an exercise ... even if you get really frustrated.

Outline

- 
- Introduction to OpenMP
 - Creating Threads
 - Synchronization
 - Parallel Loops
 - Data environment
 - Memory model

OpenMP* overview:

C\$OMP FLUSH

#pragma omp critical

C\$OMP THREADPRIVATE (/ABC/)

CALL OMP_SET_NUM_THREADS(10)

OpenMP: An API for Writing Multithreaded Applications

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

C\$OMP PARALLEL COPYIN (/blk/)

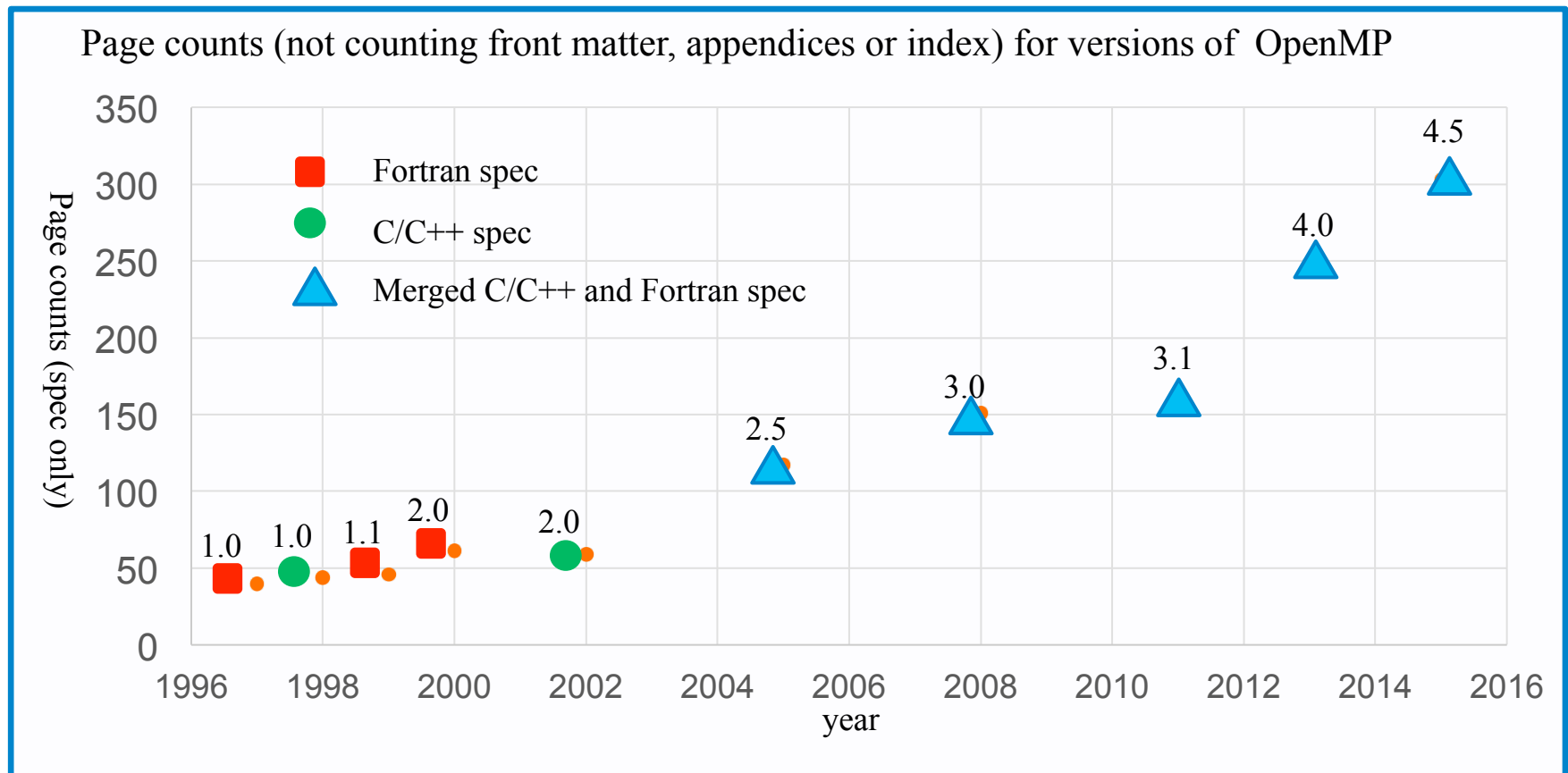
C\$OMP DO lastprivate (XX)

Nthrds = OMP_GET_NUM_PROCS()

omp_set_lock(lck)

The 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!



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”

Resources

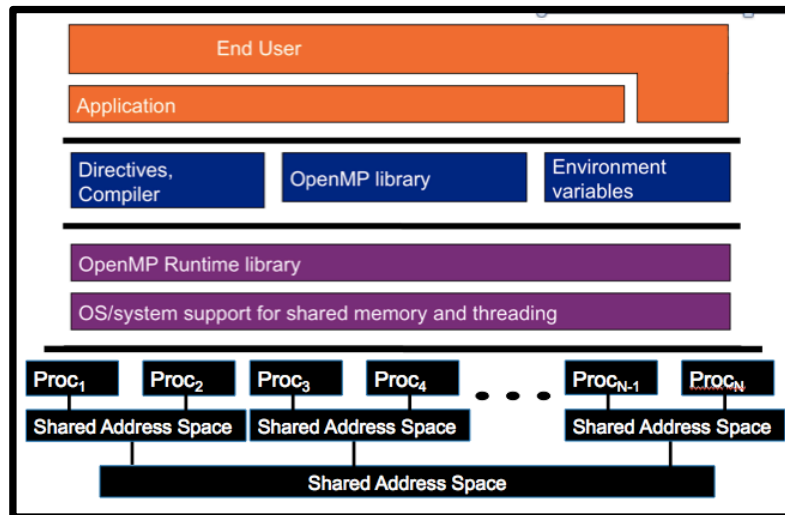
<http://www.openmp.org>

- We can only give an overview today
 - **We won't cover all features**
- Lots of information available at ARB's website
 - Specifications, technical reports, **summary cards** for downloading
 - Tutorials and publications; links to other tutorials
- Tutorials also at:
 - Supercomputing conferences
 - Annual OpenMPCon, IWOMP workshop
 - Some user sites, e.g. NERSC



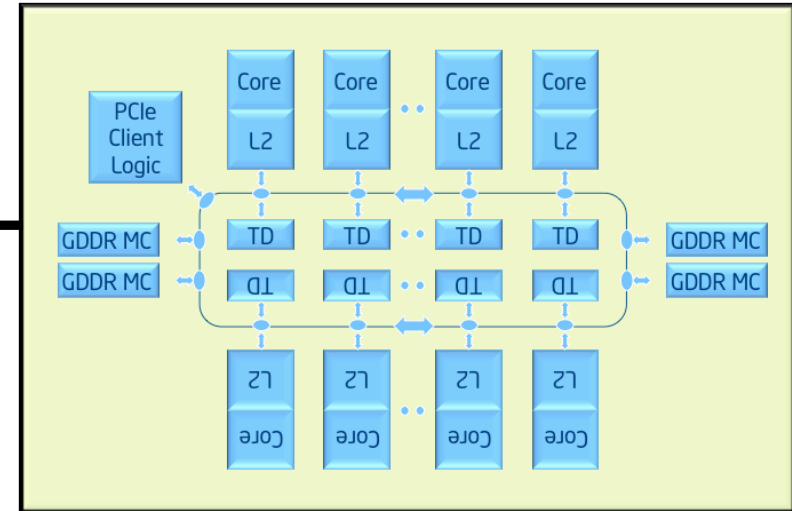
Where Does OpenMP Run?

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

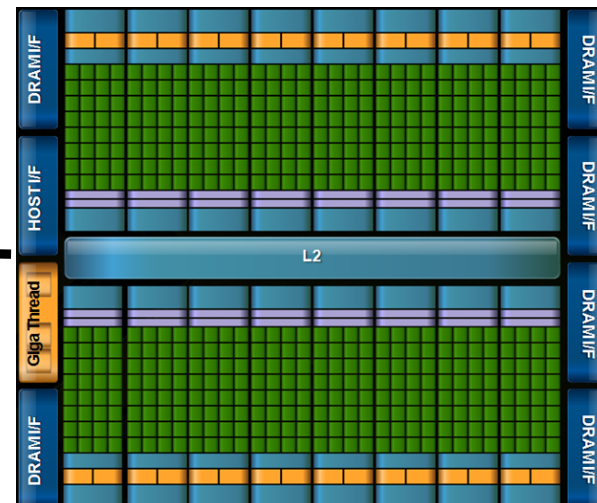


Host

OpenMP 4.5



Target Device: Intel® Xeon Phi™ coprocessor



Target Device: GPU

How Does OpenMP Work?

- Teams of OpenMP threads are created to perform the computation in a code
 - **Work is divided** among the threads, which run on the different cores
 - The threads collaborate **by sharing variables**
 - Threads **synchronize** to order accesses and prevent data corruption
 - **Structured programming** is encouraged to reduce likelihood of bugs
- Most Fortran/C/C++ compilers implement OpenMP
 - Use compiler “flag”, sometimes a specific **optimization level**
- Alternatives:
 - MPI
 - POSIX thread library is lower level
 - Automatic parallelization is higher level (user does nothing)
 - But usually successful on simple codes only

Programming in Pthreads vs. OpenMP

```
#include <pthread.h>
#define DEFAULT_NUM_THREADS 4

/* encapsulate multiple args to a thread */
typedef struct args {
    int id;          /* this thread's number */
} args_t;

/* function that is run inside each thread */
void *do_hello_world(void *arg)
{
    args_t *ap = (args_t *) arg; /* unpack incoming args */
    printf("Hello from thread %d\n", ap->id); /* ACTUAL WORK */
    return NULL;
}

int main(int argc, char *argv[])
{
    int i, num_threads = DEFAULT_NUM_THREADS;
    pthread_t *thread_pool;
    args_t *thread_args;

    if (argc > 1) {
        num_threads = atoi(argv[1]);
        if (num_threads < 0) {
            num_threads = DEFAULT_NUM_THREADS;
        }
    }

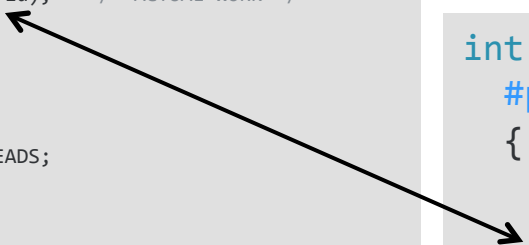
    thread_pool = (pthread_t *) malloc(num_threads *
                                      sizeof(*thread_pool));
    thread_args = (args_t *) malloc(num_threads *
                                    sizeof(*thread_args));

    /* create and run threads: pass id of thread to each */
    for (i = 0; i < num_threads; i += 1) {
        thread_args[i].id = i;
        pthread_create(&thread_pool[i], NULL, do_hello_world,
                      (void *) &thread_args[i]);
    }

    /* wait for all threads to finish */
    for (i = 0; i < num_threads; i += 1) {
        pthread_join(thread_pool[i], NULL);
    }

    free(thread_args);
    free(thread_pool);
    return 0;
}
```

```
int main(int argc, char *argv[]) {
    #pragma omp parallel
    {
        int ID = omp_get_thread_num();
        printf("hello from thread %d\n", ID);
    }
    return 0;
}
```

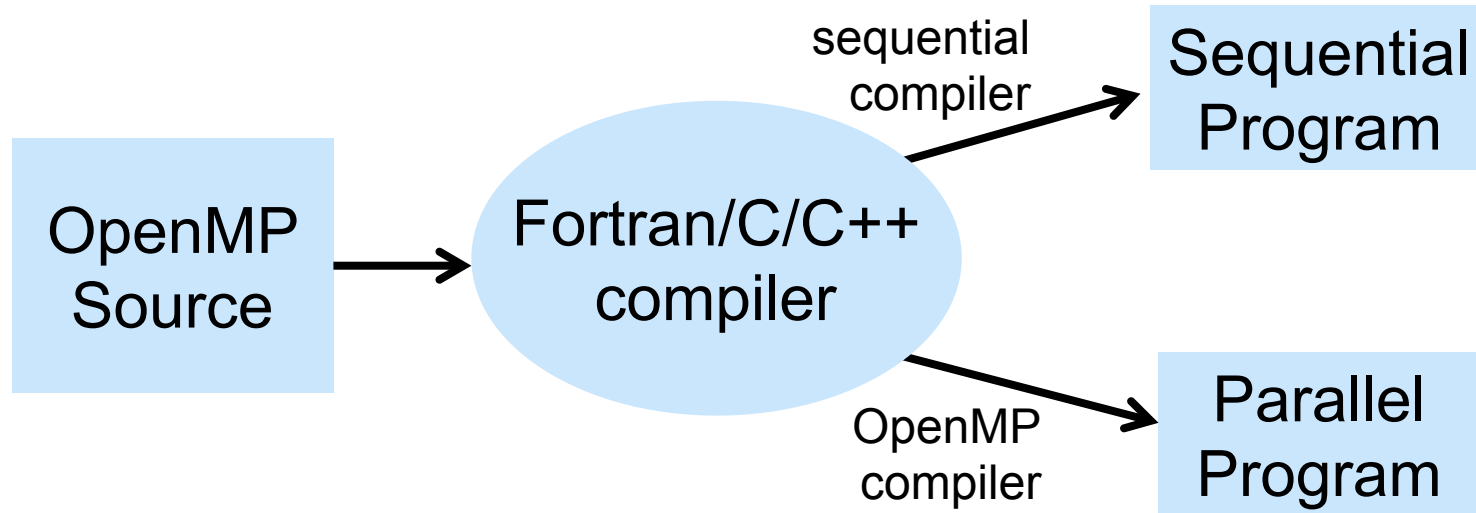


What Does the User Have to Do?

- Starting point is most often MPI or sequential program code
- Application developer must decide how the work can be divided up among multiple threads
 - Identify parallelism and needed synchronization
 - Getting this right is the **user's responsibility!**
 - Insert OpenMP constructs that represent the strategy
- Getting good performance requires an understanding of implications of chosen strategy
 - Translation introduces overheads
 - Data access pattern might affect performance
- **Sometimes, non-trivial rewriting of code is needed to accomplish desired results**

User makes strategic decisions; compiler figures out details

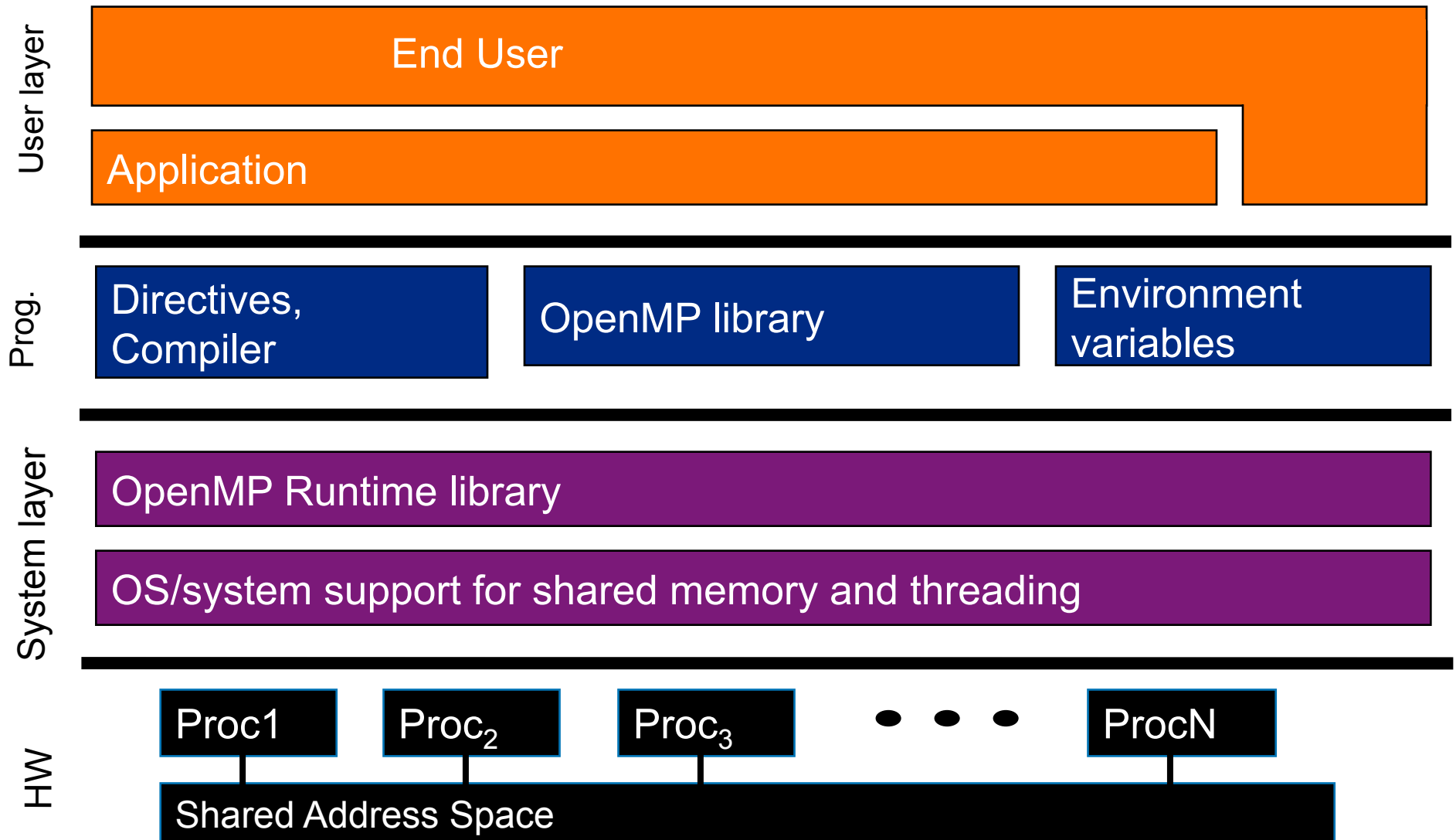
OpenMP Usage



Info on compiler used in training

Compiler Name	Compiler Version	OpenMP version	OpenMP flag	C/C++/Fortran compiler
GNU Compiler Collection (gcc) [cori, bluewaters, Edison, stampede 2]	6.3.0	4.5	-fopenmp	gcc, g++, gfortran
Intel Compilers [cori, bluewaters, Edison, stampede 2]	17.0.X	4.5	-qopenmp	icc, icpc, ifort

OpenMP basic definitions: Basic Solution stack



OpenMP basic syntax

- Most of the constructs in OpenMP are compiler directives.

#pragma omp construct [clause [clause]...]

- Example

#pragma omp parallel private(x)

- Function prototypes and types in the file:

#include <omp.h>

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

Exercise, Part A: Hello world

Verify that your environment works

- Write a program that prints “hello world”.

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

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

}
```


Exercise, Part B: Hello world

Verify that your OpenMP environment works

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

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

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

    }
}
```

Switches for compiling and linking

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

Solution

A multi-threaded “Hello world” program

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

```
#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

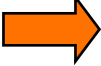
End of the Parallel region

Sample Output:

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

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

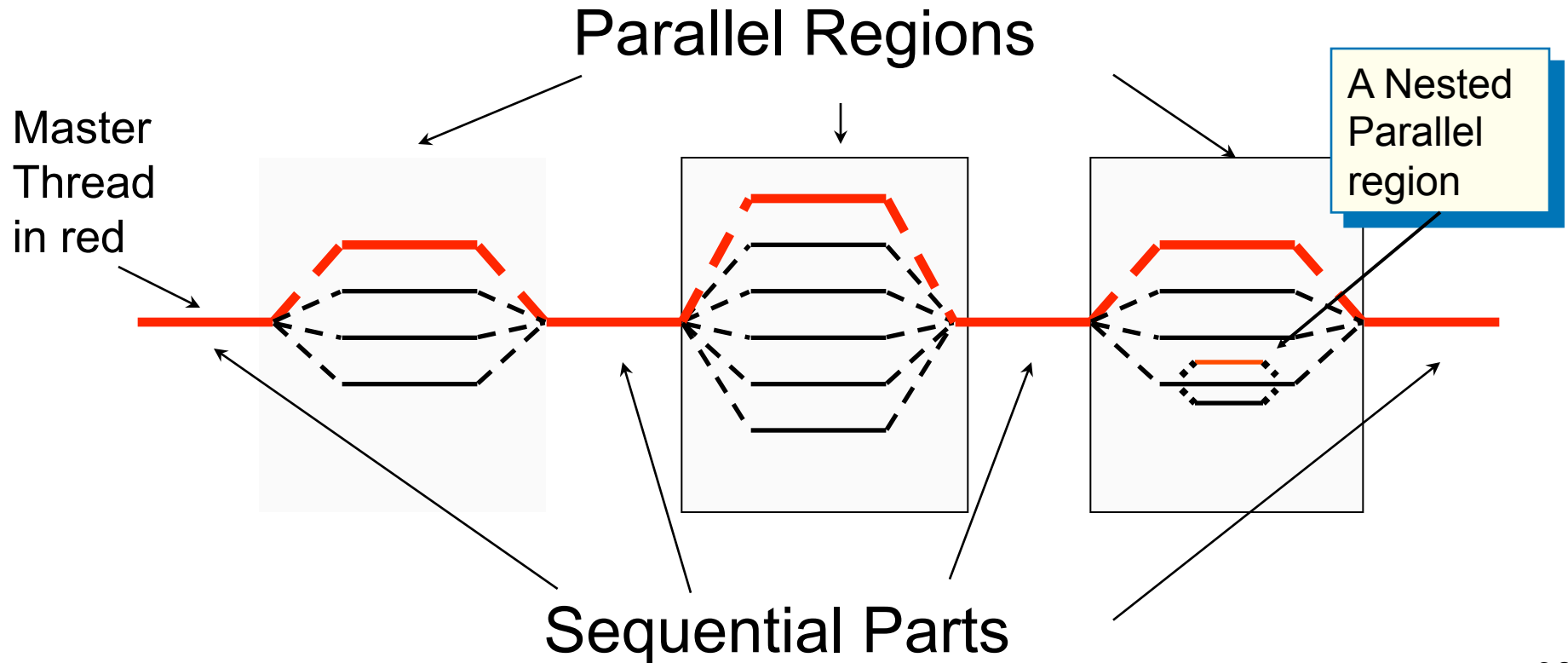
Outline

- 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

OpenMP programming model:

Fork-Join Parallelism:

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



Thread creation: Parallel regions

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

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

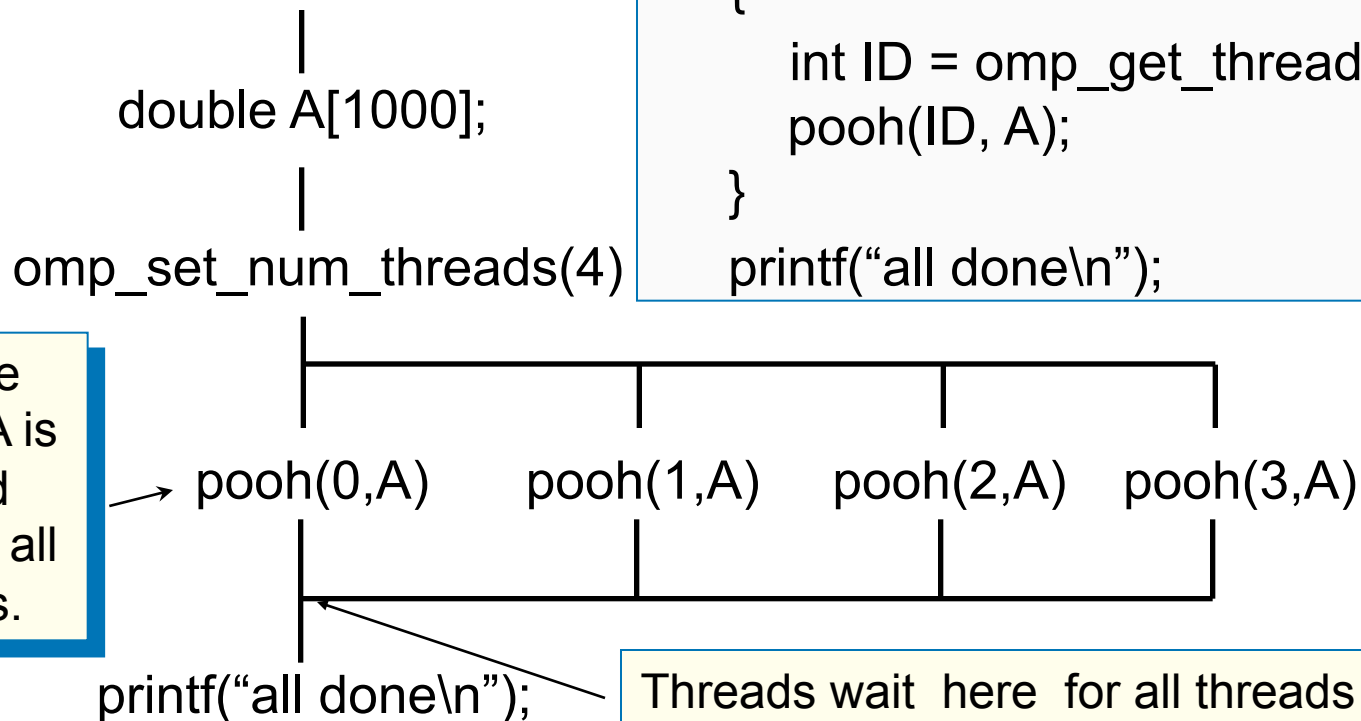
Runtime function returning a thread ID

- 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?

- 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`

Internal control variables & the number of threads

- There are a few ways to control the number of threads.
 - `omp_set_num_threads(4)`
- What does `omp_set_num_threads()` actually do?
 - It **resets** an “**internal control variable**” the system queries to select the default number of threads to request on subsequent parallel constructs.
- Is there an easier way to change this internal control variable ... perhaps one that doesn't require re-compilation? Yes.
 - When an OpenMP program starts up, it queries an environment variable `OMP_NUM_THREADS` and sets the appropriate internal control variable to the value of `OMP_NUM_THREADS`
- For example, to set the initial, default number of threads to request in OpenMP from my apple laptop
 - > `export OMP_NUM_THREADS=12`

Performance Tips

- Experiment to find the best number of threads on your system
- Put as much code as possible inside parallel regions
 - Amdahl's law: **If 1/s of the program is sequential, then you cannot ever get a speedup better than s**
 - So if 1% of a program is serial, speedup is limited to 100, no matter how many processors it is computed on
- Have large parallel regions
 - Minimize overheads: starting and stopping threads, executing barriers, moving data into cache
 - Directives can be “orphaned”; procedure calls inside regions are fine
- Run-time routines are your friend
 - Usually very efficient and allow maximum control over thread behavior
- Barriers are expensive
 - With large numbers of threads, they can be slow
 - Depends in part on HW and on implementation quality
 - Some threads might have to wait a long time if load not balanced

An interesting problem to play with

Numerical integration

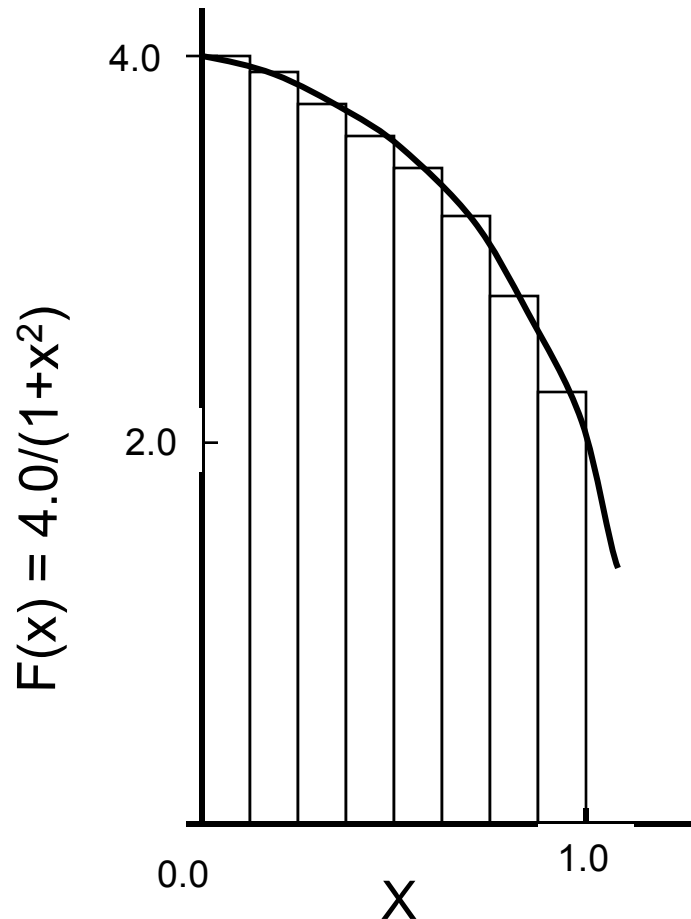
Mathematically, we know that:

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

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;
}
```

Serial PI program

```
#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

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

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();`

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;
    }
}
```

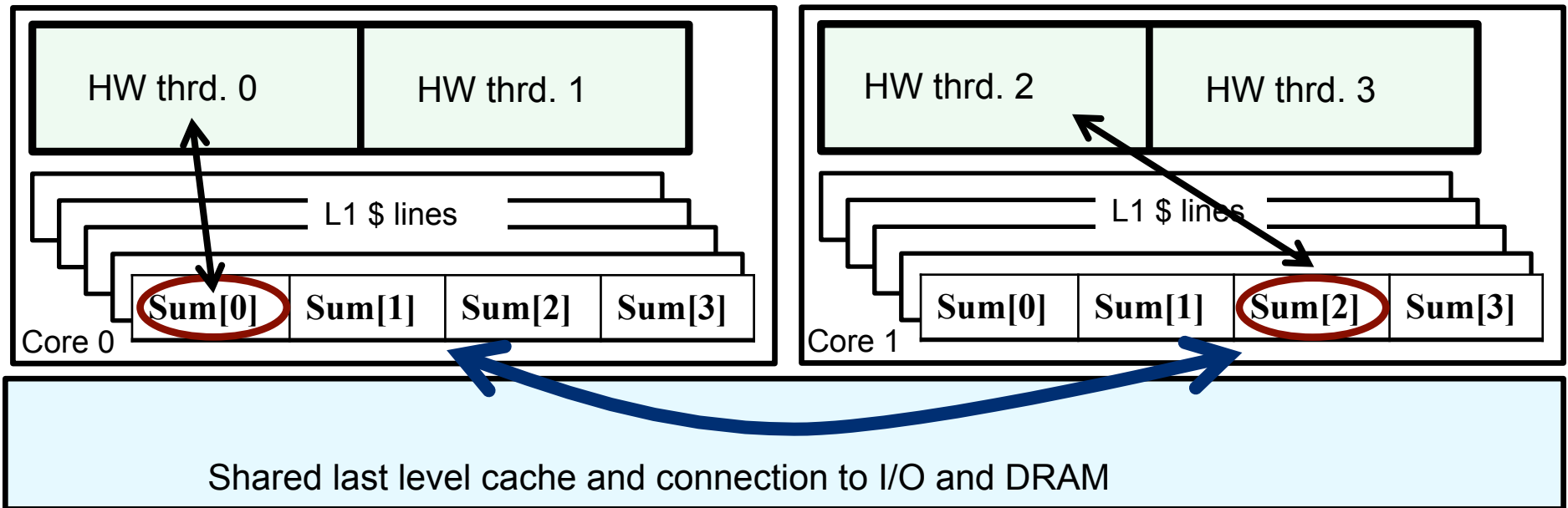
threads	1 st SPMD*
1	1.86
2	1.03
3	1.08
4	0.97

*SPMD: Single Program Multiple Data

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

Why such poor scaling? False sharing

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

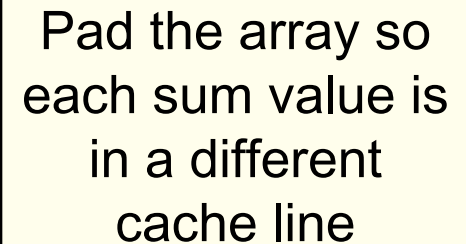


- If you promote scalars to an array to support creation of an SPMD program, the array elements are contiguous in memory and hence share cache lines ... Results in poor scalability.
- Solution: Pad arrays so elements you use are on distinct cache lines.

Example: Eliminate false sharing by padding the sum array

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

    for(i=0, pi=0.0; i<nthreads; i++) pi += sum[i][0] * step;
}
```



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

Results*: pi program padded accumulator

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

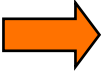
Example: eliminate False sharing by padding the sum array

```
#include <omp.h>
static long num_steps = 100000;    double step;
#define PAD 8    // assume 64 byte L1 cache line size
#define NUM_THREADS 2
void main ()
{
    int i, nthrds; 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) nthrds = 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<nthrds; i++) pi += sum[i][0] * step;
}
```

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

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

Outline

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

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 must reach before any threads are allowed to proceed.
- It is a “stand alone” pragma meaning it is not associated with user code ... it is an executable statement.

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


omp_set_num_threads(8)

#pragma omp parallel
{  int id, nthrds;

    id = omp_get_thread_num();
    nthrds = omp_get_num_threads();
    if (id==0) numthrds = nthrds;
    Arr[id] = big_ugly_calc(id, nthrds);

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

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



Exercise

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

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	1 st SPMD
1	1.86
2	1.03
3	1.08
4	0.97

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

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

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

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

No array, so no false sharing.

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

Results*: pi program critical section

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

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

```
#include <omp.h>
static long num_steps = 100000;    double step;
#define NUM_THREADS 2
void main ()
{
    int nthreads; double pi=0.0;    step = 1.0/(double) num_steps;
    omp_set_num_threads(NUM_THREADS);
#pragma omp parallel
{
    int i, id, nthrds;  double x, sum;
    id = omp_get_thread_num();
    nthrds = omp_get_num_threads();
    if (id == 0) nthrds = 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	1 st SPMD	1 st SPMD padded	SPMD critical
1	1.86	1.86	1.87
2	1.03	1.01	1.00
3	1.08	0.69	0.68
4	0.97	0.53	0.53

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

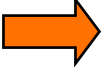
Example: Using a critical section to remove impact of false sharing

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

Be careful where you
put a critical section

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

Outline

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

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

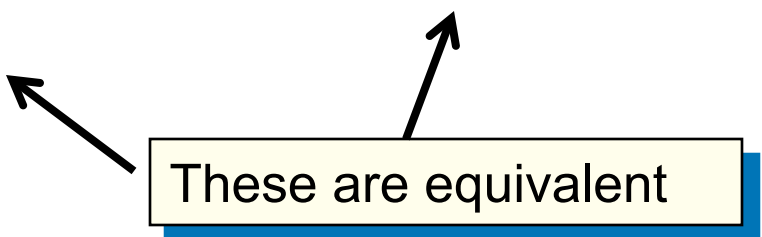
Most work at runtime :
complex scheduling logic used at run-time

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

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.

Exercise: Pi with loops

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

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

Example: Pi with a loop and a reduction

```
#include <omp.h>
```

```
static long num_steps = 100000;    double step;
```

```
void main ()
```

```
{  int i;        double x, pi, sum = 0.0;
```

```
    step = 1.0/(double) num_steps;
```

```
    #pragma omp parallel
```

```
    {
```

```
        double x;
```

```
        #pragma omp for reduction(+:sum)
```

```
        for (i=0;i< num_steps; i++){
```

```
            x = (i+0.5)*step;
```

```
            sum = sum + 4.0/(1.0+x*x),
```

```
        }
```

```
    }
```

```
    pi = step * sum;
```

```
}
```

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

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

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

Results*: pi with a loop and a reduction

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

Example: Pi with a

```
#include <omp.h>
static long num_steps = 100000000;
void main ()
{ int i; double x, pi, sum;
  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;
}
```

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

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

The nowait clause

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

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

```
#pragma omp parallel
```

```
{
```

```
    int id=omp_get_thread_num();
```

```
    A[id] = big_calc1(id);
```

```
#pragma omp barrier
```

```
#pragma omp for
```

```
    for(i=0;i<N;i++){C[i]=big_calc3(i,A);}
```


```
#pragma omp for nowait
```

```
    for(i=0;i<N;i++){ B[i]=big_calc2(C, i); }
```

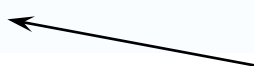
```
    A[id] = big_calc4(id);
```

```
}
```

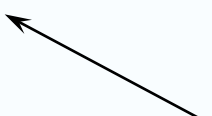
implicit barrier at the end of a for
worksharing construct



implicit barrier at the end
of a parallel region



no implicit barrier
due to nowait



Limitations of Parallel For / Do

```
#pragma omp parallel
{
    ...
    while (my_pointer != NULL) {
        do_independent_work(my_pointer);
        my_pointer = my_pointer->next;
    } // End of while loop
    ...
}
```

To use a for or do construct, loops must be countable.


To parallelize this loop, it is necessary to first count the number of iterations and then rewrite it as a *for* loop.

Or we can use tasks. More on this later...

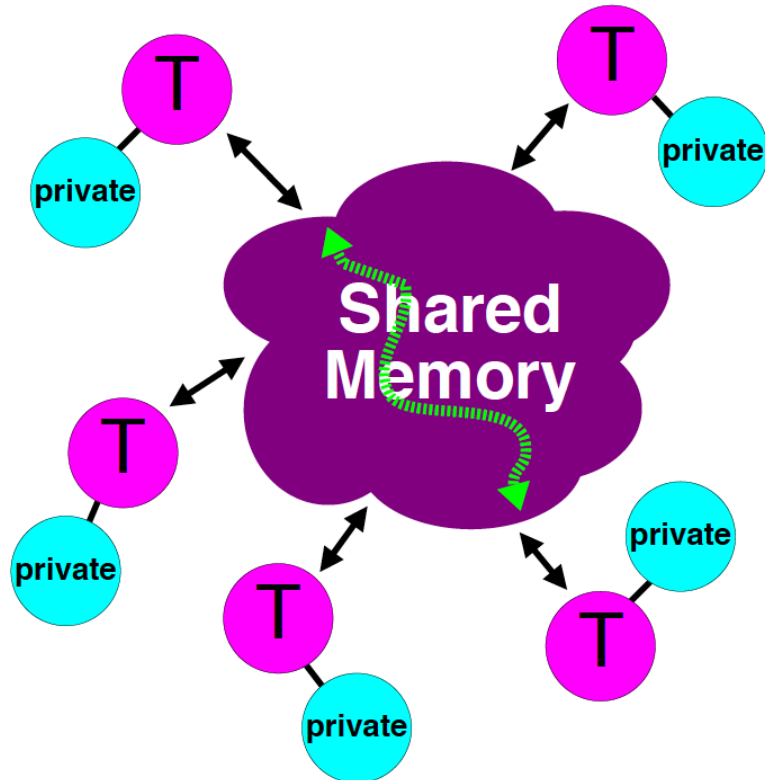
Performance Tips

- Is there enough work to amortize overheads?
 - May not be worthwhile for very small loops (if clause can control this)
 - Might be overcome by choosing different loop, rewriting loop nest or collapsing loop nest
- Best choice of schedule might change with system, problem size
 - Experimentation may be needed
- Minimize synchronization
 - Use nowait where possible
- Locality
 - Most large systems are NUMA
 - Be prepared to modify your loop nests
 - Change loop order to get better cache behavior
- If performance is bad, look for false sharing
 - We talk about this in part 2 of the tutorial
 - Occurs frequently, performance degradation can be catastrophic

Outline

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

OpenMP Memory Model



- All threads access the same, globally shared memory
- Data can be shared or private
 - **Shared** – only one instance of data
 - Threads can access data simultaneously
 - Changes are visible to all threads
 - Not necessarily immediately
 - **Private** - Each thread has copy of data
 - No other thread can access it
 - Changes only visible to the thread owning the data
- OpenMP has **relaxed-consistency** shared memory model
 - Threads may have a *temporary* view of shared memory that is not consistent with that of other threads
 - These temporary views are made consistent at certain places in code

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.

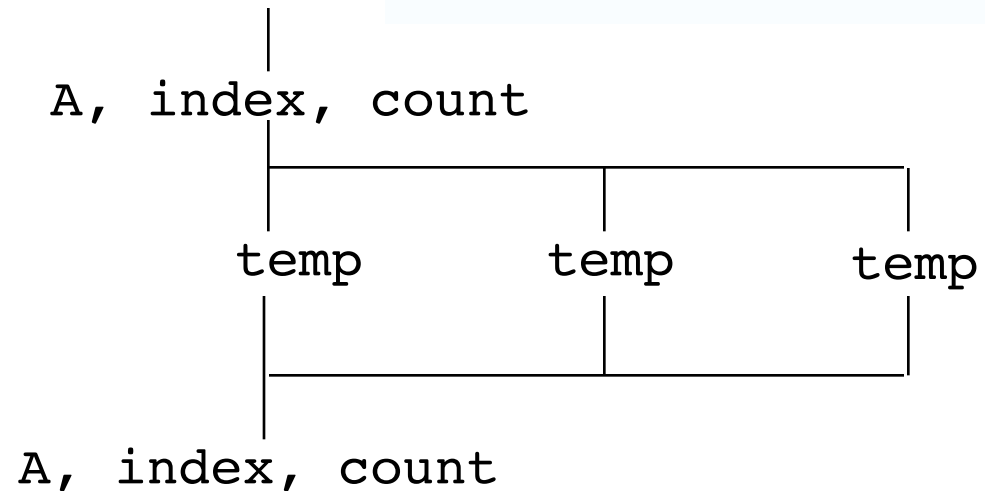
Data sharing: Examples

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

A, index and count are shared by all threads.

temp is local to each thread

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

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

tmp is 0 here

tmp was not initialized

Data sharing: Private clause

When is the original variable valid?

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

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

tmp has unspecified value

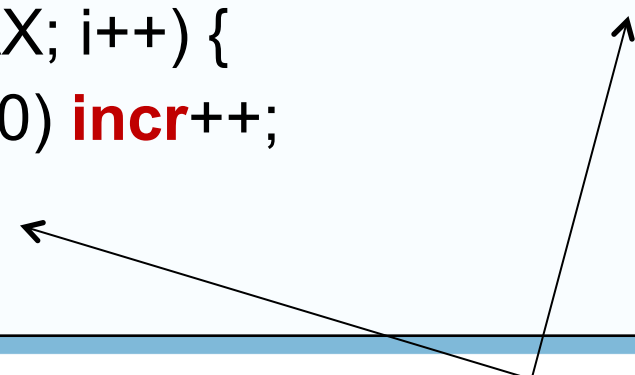
```
extern int tmp;  
void work() {  
    tmp = 5;  
}
```

unspecified which
copy of tmp

Firstprivate clause

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

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



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

Data sharing:

A data environment test

- Consider this example of PRIVATE and FIRSTPRIVATE

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

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

Inside this parallel region ...

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

Following the parallel region ...

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

Data sharing: Default clause

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

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

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

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

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

Performance and Correctness Tips

- There is one version of shared data
 - Keeping data shared reduces overall memory consumption
- Private data is stored locally, so use of private variables can increase efficiency
 - Avoids false sharing
 - May make it easier to parallelize loops
 - But private data is no longer available after parallel regions ends
- It is an error if multiple threads update the same variable at the same time (a data race)
- It is a good idea to use “default none” while testing code
- Putting code into a subroutine / function can make it easier to write code with many private variables
 - Local / automatic data in a procedure is private by default

Exercise: Mandelbrot set area

- The supplied program (mandel.c) computes the area of a Mandelbrot set.
- The program has been parallelized with OpenMP, but we were lazy and didn't do it right.
- Find and fix the errors (hint ... the problem is with the data environment).
- Once you have a working version, try to optimize the program.
 - Try different schedules on the parallel loop.
 - Try different mechanisms to support mutual exclusion ... do the efficiencies change?

The Mandelbrot area program

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

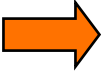
int main(){
    int i, j;
    double area, error, eps = 1.0e-5;
#pragma omp parallel for default(shared) private(c, j) \
firstprivate(eps)
    for (i=0; i<NPOINTS; i++) {
        for (j=0; j<NPOINTS; j++) {
            c.r = -2.0+2.5*(double)(i)/(double)(NPOINTS)+eps;
            c.i = 1.125*(double)(j)/(double)(NPOINTS)+eps;
testpoint(c);
        }
    }
    area=2.0*2.5*1.125*(double)(NPOINTS*NPOINTS-
numoutside)/(double)(NPOINTS*NPOINTS);
    error=area/(double)NPOINTS;
}
```

```
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;
        }
    }
}
```

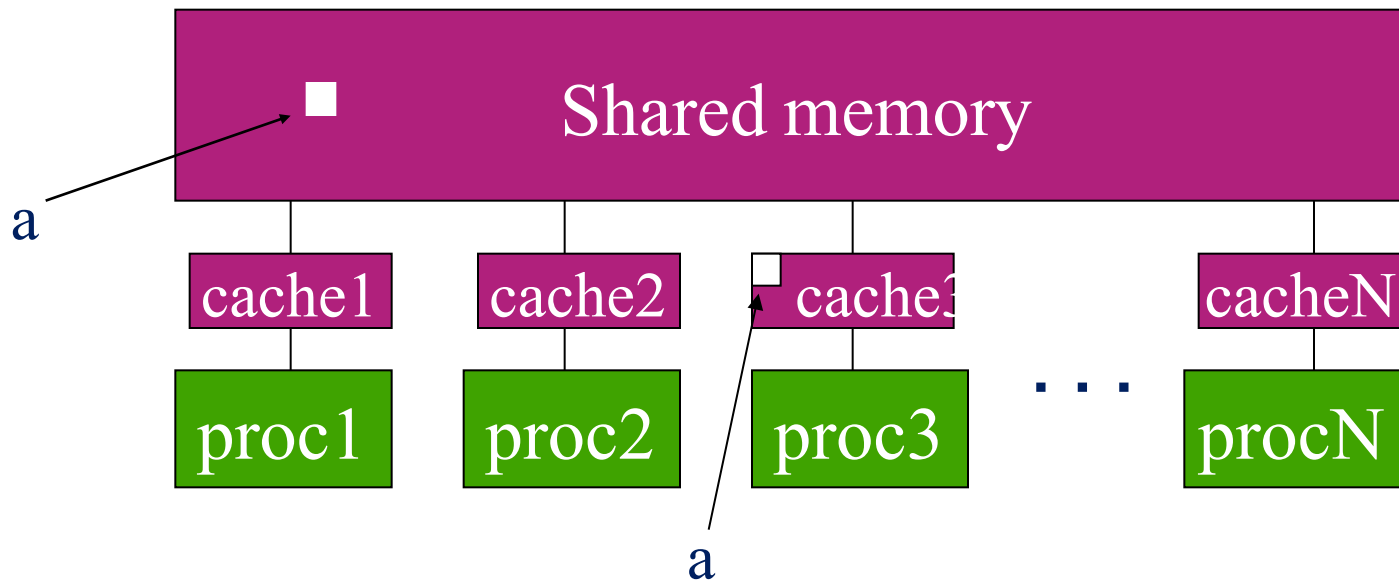
- eps was not initialized
- Protect updates of numoutside
- Which value of c does testpoint() see? Global or private?

Outline

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

OpenMP memory model

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



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

OpenMP and relaxed consistency

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

Flush operation

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

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.

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();
}
```

The OpenMP Common Core: Most OpenMP programs only use these 16 constructs

OMP Construct	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. The flush concept (but not the concept)
#pragma omp single	Workshare with a single thread
#pragma omp task #pragma omp taskwait	tasks including the data environment for tasks.