



# A “Hands-on” Introduction to OpenMP\*



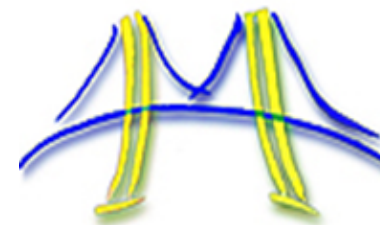
# Disclaimer

**READ THIS ... its very important**

- The views expressed in this talk are those of the speakers and not their employer.
- This is an academic style talk and does not address details of any particular Intel product. You will learn nothing about Intel products from this presentation.
- This was a team effort, but if we say anything really stupid, it's our fault ... don't blame our collaborators.

# Acknowledgements

- This course is based on a long series of tutorials presented at Supercomputing conferences. The following people helped prepare this content:
  - J. Mark Bull (the University of Edinburgh)
  - Rudi Eigenmann (Purdue University)
  - Barbara Chapman (University of Houston)
  - Larry Meadows, Sanjiv Shah, and Clay Breshears (Intel Corp).
- Some slides are based on a course I teach with Kurt Keutzer of UC Berkeley. The course is called “CS194: Architecting parallel applications with design patterns”. These slides are marked with the UC Berkeley ParLab logo:



# Introduction

- OpenMP is one of the most common parallel programming models in use today.
- It is relatively easy to use which makes a great language to start with when learning to write parallel software.
- Assumptions:
  - We assume you know C. OpenMP supports Fortran and C++, but we will restrict ourselves to C.
  - We assume you are new to parallel programming.
  - We assume you have access to a compiler that supports OpenMP (more on that later).

# Preliminaries:

- Our plan ... Active learning!
  - We will mix short lectures with short exercises.
- Download exercises and reference materials.
- Please follow these simple rules
  - Do the exercises 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.

# Agenda

- 
- Getting started with OpenMP
  - Working with threads
  - Synchronization in OpenMP
  - Loop and single worksharing constructs
  - OpenMP Data Environment
  - OpenMP tasks
  - Closing Comments

# OpenMP\* Overview:

```
C$OMP FLUSH
```

```
#pragma omp critical
```

```
C$OMP THREADPRIVATE (/ABC/)
```

```
CALL OMP SET NUM THREADS (10)
```

```
C$OM
```

## *OpenMP: An API for Writing Multithreaded Applications*

```
C$OM
```

- 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 last 20 years of SMP practice

```
C$O
```

```
C
```

```
#p:
```

```
C$OMP PARALLEL COPYIN (/blk/)
```

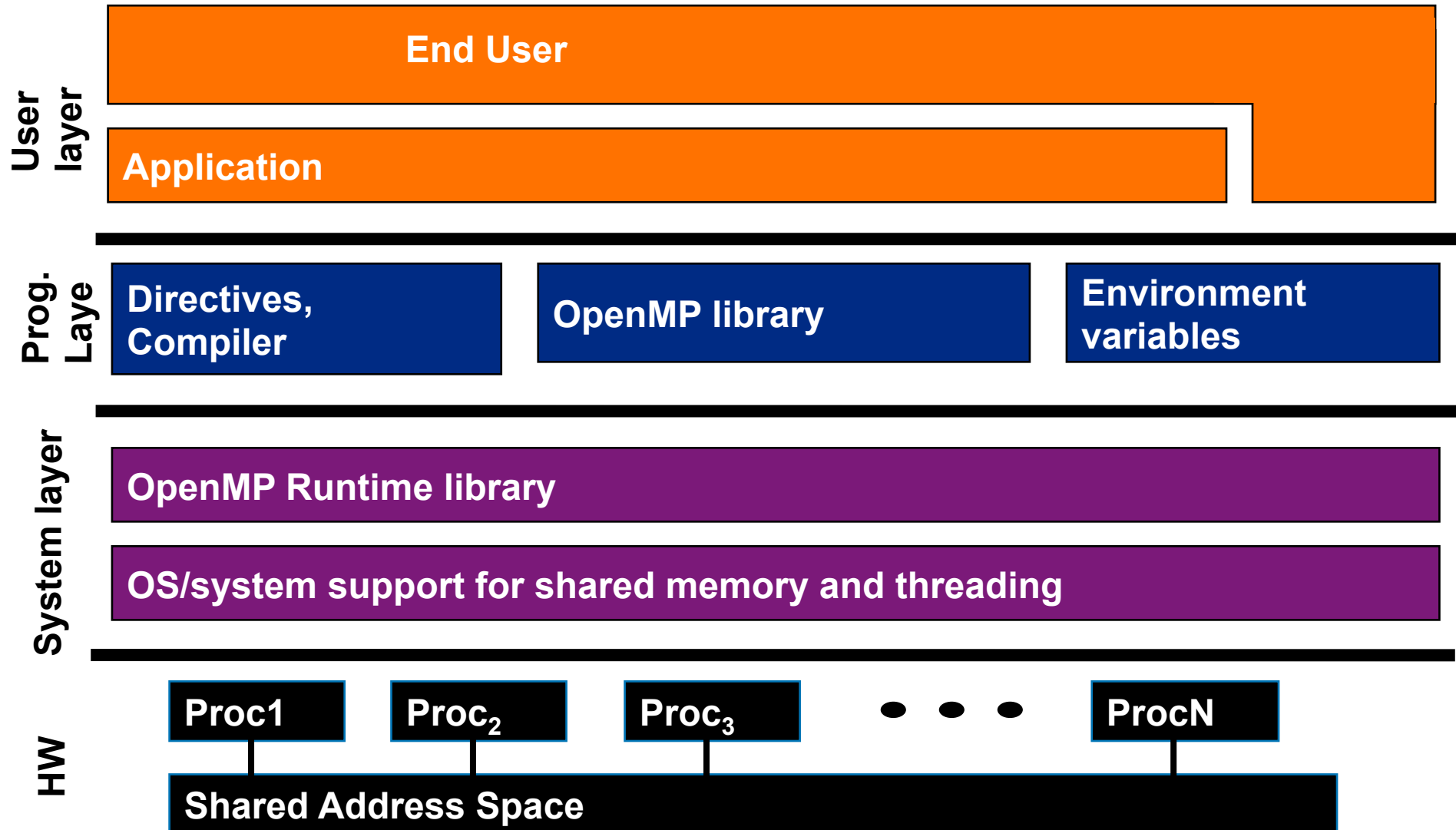
```
C$OMP DO lastprivate (XX)
```

```
Nthrds = OMP_GET_NUM_PROCS()
```

```
omp_set_lock(lck)
```

```
ED
```

# OpenMP Basic Defs: Solution Stack





# OpenMP core syntax

- Most of the constructs in OpenMP are compiler directives.

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

- Example

**`#pragma omp parallel num_threads(4)`**

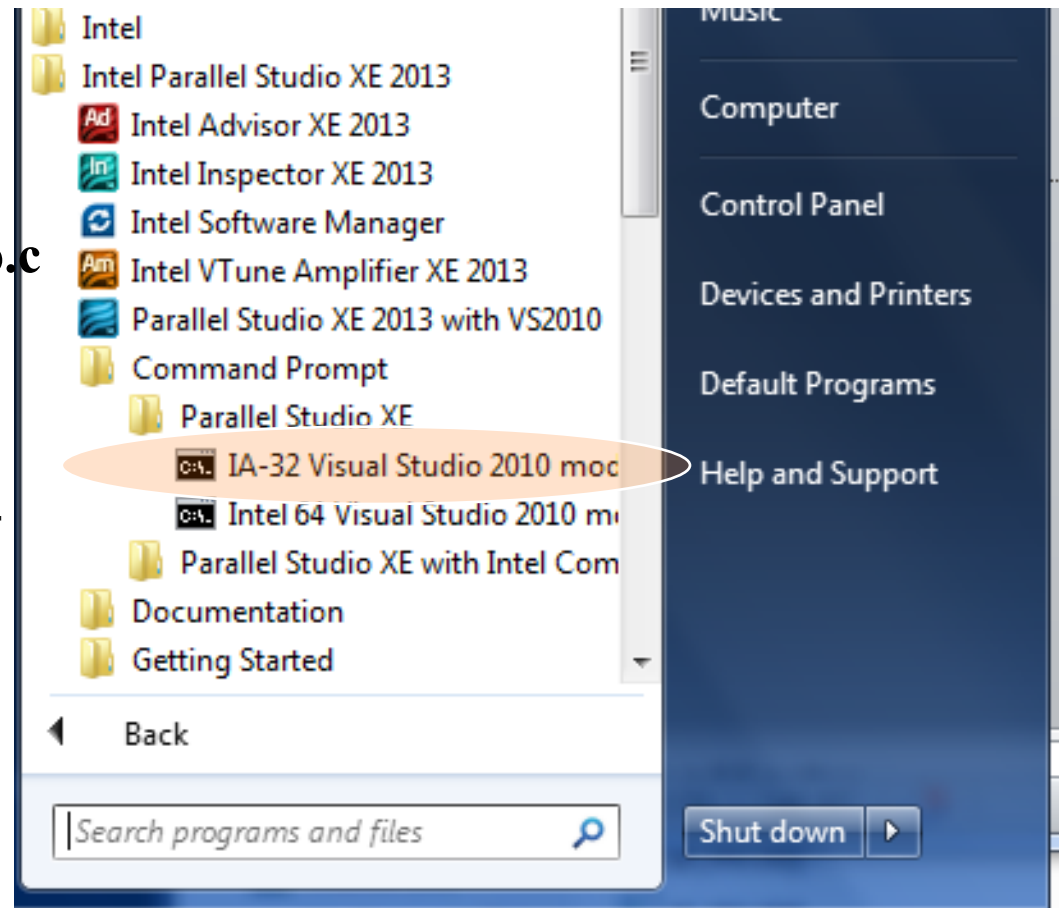
- Function prototypes and types in the file:

**`#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.

# Compiler notes: Intel on Windows

- Launch SW dev environment
- cd to the directory that holds your source code
- **Build software for program foo.c**
  - ◆ `icl /Qopenmp foo.c`
- **Set number of threads environment variable**
  - ◆ `set OMP_NUM_THREADS=4`
- **Run your program**
  - ◆ `foo.exe`



# Compiler notes: Visual Studio

- Start “new project”
- Select win 32 console project
  - Set name and path
  - On the next panel, Click “next” instead of finish so you can select an empty project on the following panel.
  - Drag and drop your source file into the source folder on the visual studio solution explorer
  - Activate OpenMP
    - Go to project properties/configuration properties/C.C++/language ... and activate OpenMP
- Set number of threads inside the program
- Build the project
- Run “without debug” from the debug menu.

# Compiler notes: Other

- Linux and OS X with gcc:

- > gcc -fopenmp foo.c

- > export OMP\_NUM\_THREADS=4

- > ./a.out

for the Bash shell



- Linux and OS X with PGI:

- > pgcc -mp foo.c

- > export OMP\_NUM\_THREADS=4

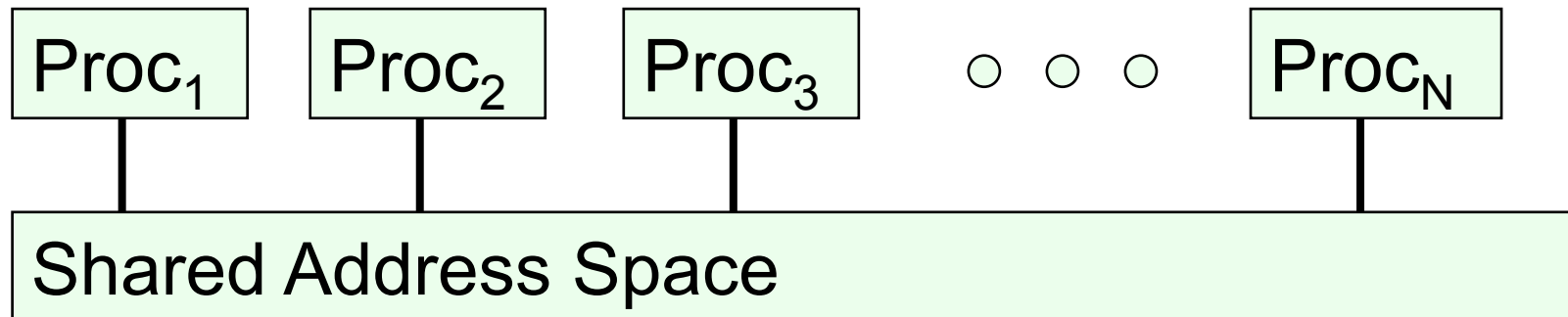
- > ./a.out

# Shared memory Computers

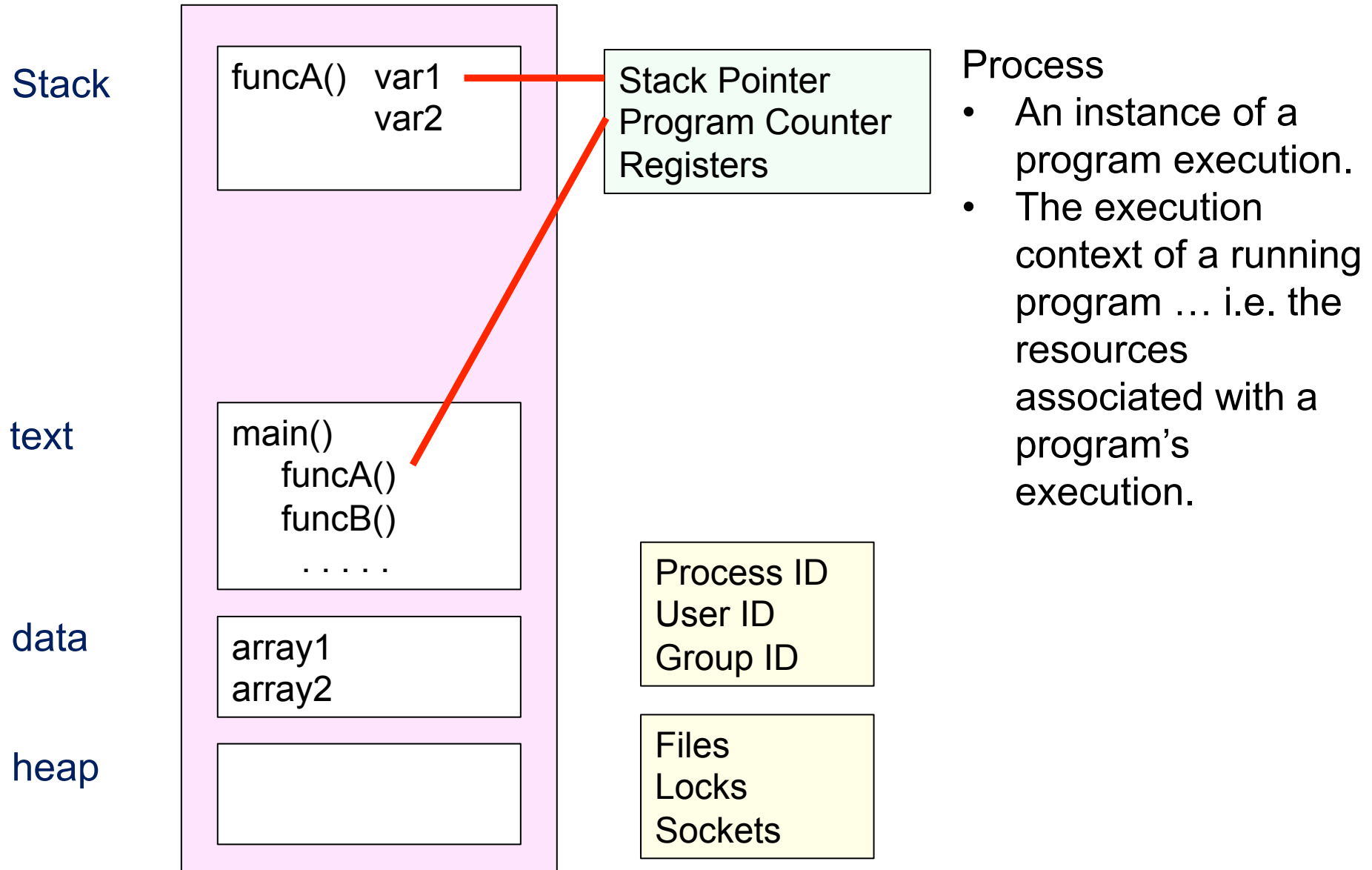
- **Shared memory computer** : any computer composed of multiple processing elements that share an address space.

Two Classes:

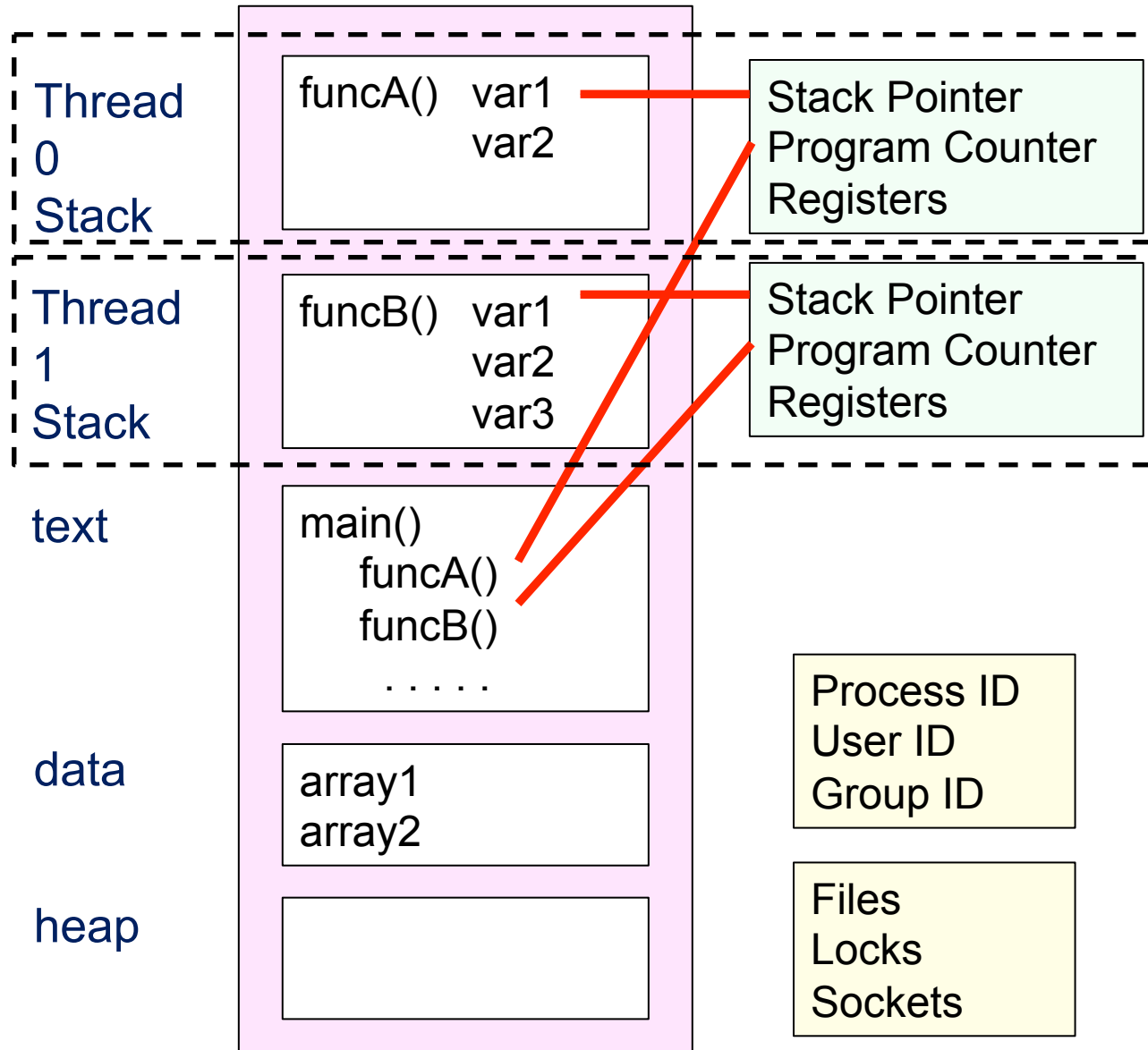
- **Symmetric multiprocessor (SMP)**: a shared address space with “equal-time” access for each processor, and the OS treats every processor the same way.
- **Non Uniform address space multiprocessor (NUMA)**: different memory regions have different access costs ... think of memory segmented into “Near” and “Far” memory.



# Programming shared memory computers



# Programming shared memory computers

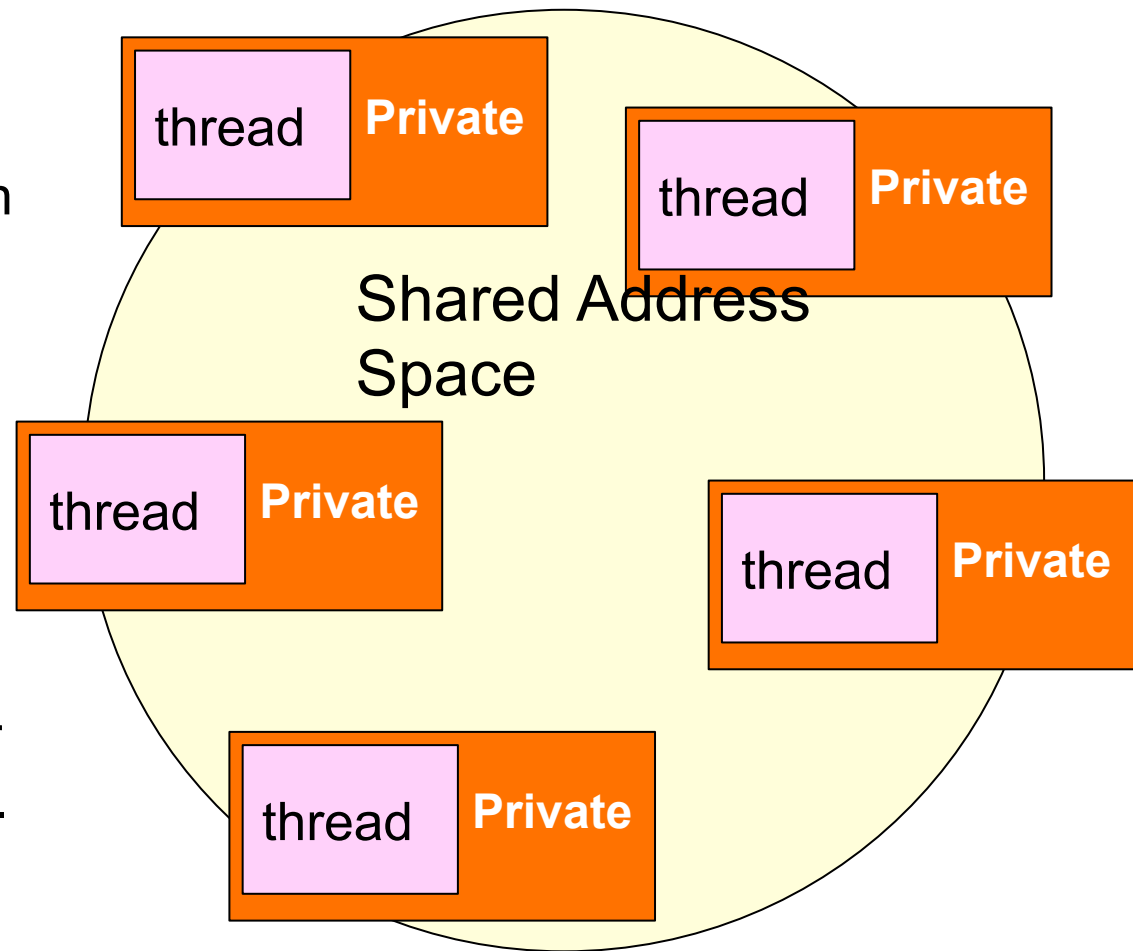


## Threads:

- Threads are "light weight processes"
- Threads share Process state among multiple threads ... this greatly reduces the cost of switching context.

# A shared memory program

- An instance of a program:
  - One process and lots of threads.
  - Threads interact through reads/writes to a shared address space.
  - OS scheduler decides when to run which threads ... interleaved for fairness.
  - Synchronization to assure every legal order results in correct results.





# OpenMP Overview:

## How do threads interact?

- OpenMP is a multi-threading, shared address model.
  - Threads communicate by sharing variables.
- Unintended sharing of data causes race conditions:
  - race condition: when the program's outcome changes as the threads are scheduled differently.
- To control race conditions:
  - Use synchronization to protect data conflicts.
- Synchronization is expensive so:
  - Change how data is accessed to minimize the need for synchronization.

# Exercise 1, Part A: Hello world

## Verify that your environment works

- Write a program that prints “hello world”.

```
int main()
{

    int ID = 0;

    printf(" hello(%d) ", ID);
    printf(" world(%d) \n", ID);

}
```

# Exercise 1, Part B: Hello world

## Verify that your OpenMP environment works

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

```
#include <omp.h>
int main()
{
    #pragma omp parallel
    {
        int ID = 0;

        printf(" hello(%d) ", ID);
        printf(" world(%d) \n", ID);
    }
}
```

Linux and OS X

gcc -fopenmp

PGI Linux

pgcc -mp

Intel windows

icl /Qopenmp

Intel Linux and OS X

icpc -openmp

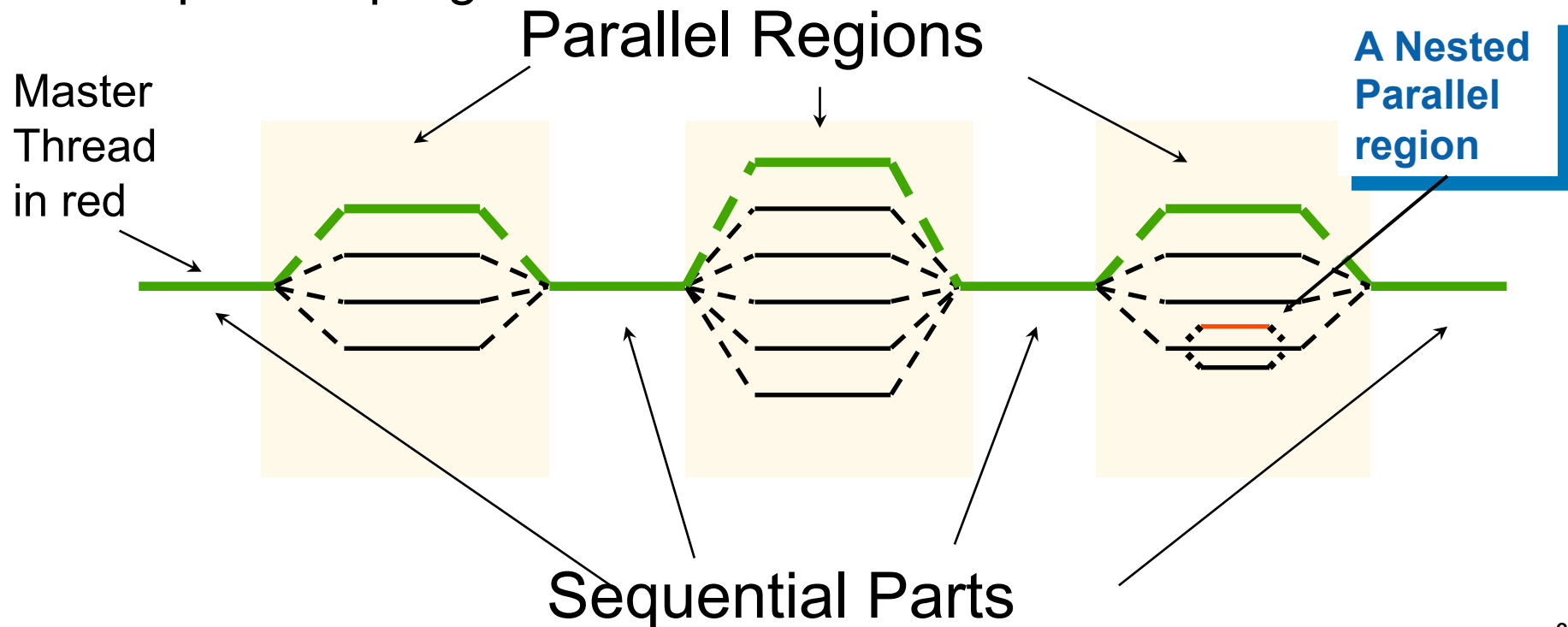
# Agenda

- Getting started with OpenMP
-  • Working with threads
- Synchronization in OpenMP
- Loop and single worksharing constructs
- OpenMP Data Environment
- OpenMP tasks
- Closing Comments

# 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

- 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];
```

```
#pragma omp parallel num_threads(4)
{
    int ID = omp_get_thread_num();
    pooh(ID,A);
}
```

clause to request a certain number of threads

Runtime function returning a thread ID

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

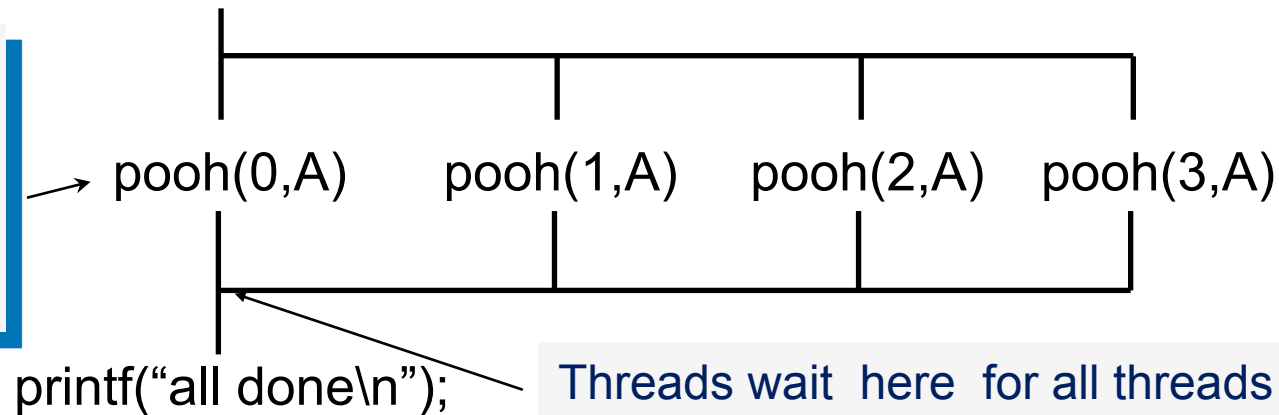
# Thread Creation: Parallel Regions

- Each thread executes the same code redundantly.

```
double A[1000];  
#pragma omp parallel num_threads(4)  
{  
    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.



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



# OpenMP: what the compiler does

```
#pragma omp parallel num_threads(4)
{
    foobar ();
}
```

- The OpenMP compiler generates code logically analogous to that on the right of this slide, given an OpenMP pragma such as that on the top-left
- All known OpenMP implementations use a thread pool so full cost of threads creation and destruction is not incurred for each parallel region.
- Only three threads are created because the last parallel section will be invoked from the parent thread.

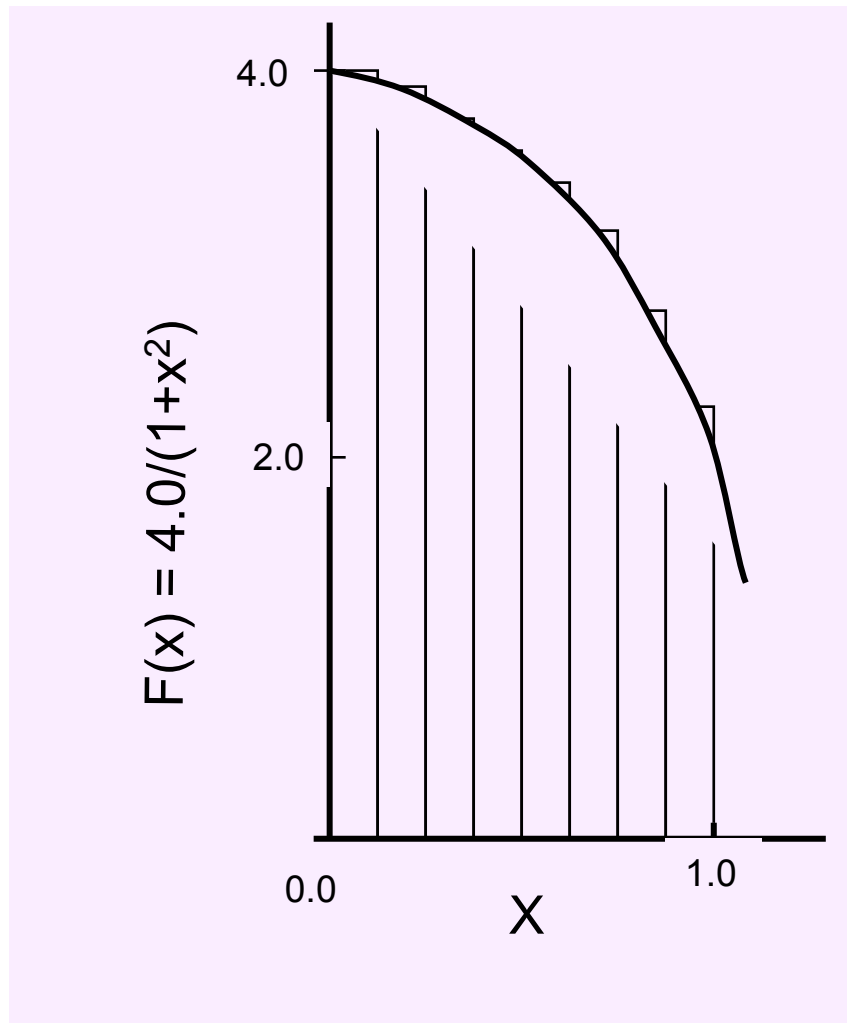
```
void thunk ()
{
    foobar ();
}

pthread_t tid[4];
for (int i = 1; i < 4; ++i)
    pthread_create (
        &tid[i], 0, thunk, 0);
thunk();

for (int i = 1; i < 4; ++i)
    pthread_join (tid[i]);
```

## Exercises 2 to 4:

### 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  $\Delta x$  and height  $F(x_i)$  at the middle of interval  $i$ .

# Exercises 2 to 4: 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;
}
```

## Exercise 2

- 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();`

Number of threads in the team

Thread ID or rank

Time in Seconds since a fixed point in the past

# 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;
}
```

# 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;
```

```
}
```

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

# Algorithm strategy:

## The SPMD (Single Program Multiple Data) design pattern

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

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

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

# 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, nthrds; 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) nthrds = 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<nthrds; i++) pi += sum[i] * step;
    }
}
```

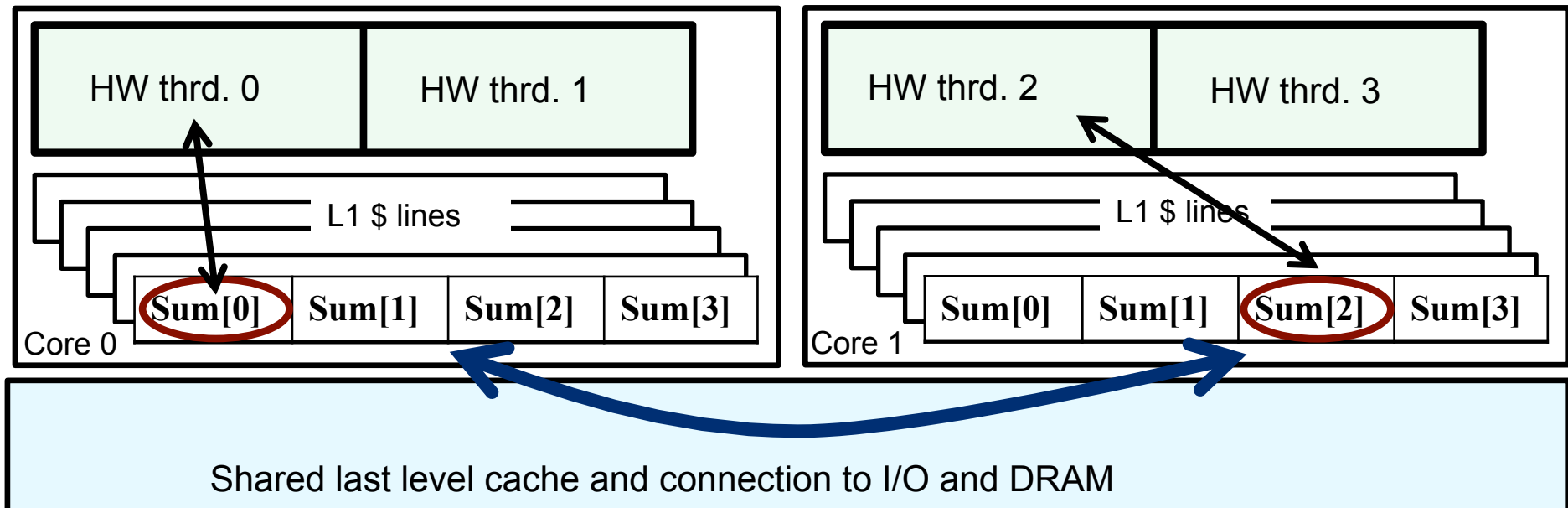
threads	1 <sup>st</sup> SPMD
1	1.86
2	1.03
3	1.08
4	0.97

\*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.



# 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 <sup>st</sup> SPMD	1 <sup>st</sup> 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.

# Do we really need to pad our arrays?

- Padding arrays requires deep knowledge of the cache architecture. Move to a machine with different sized cache lines and your software performance falls apart.
- There has got to be a better way to deal with false sharing.

# Agenda

- Getting started with OpenMP
- Working with threads
-  • Synchronization in OpenMP
- Loop and single worksharing constructs
- OpenMP Data Environment
- OpenMP tasks
- Closing Comments

# OpenMP Overview:

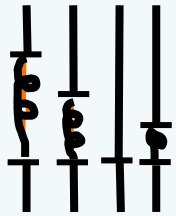
## How do threads interact?

Recall our high level overview of OpenMP?

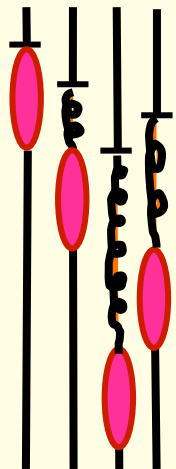
- OpenMP is a multi-threading, shared address model.
  - Threads communicate by sharing variables.
- Unintended sharing of data causes race conditions:
  - race condition: when the program's outcome changes as the threads are scheduled differently.
- To control race conditions:
  - Use synchronization to protect data conflicts.
- Synchronization is expensive so.
  - Change how data is accessed to minimize the need for synchronization.

# Synchronization:

- Synchronization: bringing one or more threads to a well defined and known point in their execution.
- The two most common forms of synchronization are:



Barrier: each thread wait at the barrier until all threads arrive.



Mutual exclusion: Define a block of code that only one thread at a time can execute.

# Synchronization

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

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



# Synchronization: Barrier

- Barrier: Each thread waits until all threads arrive.


```
#pragma omp parallel
{
    int id=omp_get_thread_num();
    A[id] = big_calc1(id);
#pragma omp barrier

    B[id] = big_calc2(id, A);
}
```

# 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: Atomic (basic form)

- **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 tmp, B;

    B = DOIT();

    tmp = big_ugly(B);

    #pragma omp atomic
    X += tmp;

}
```

The statement inside the atomic must be one of the following forms:

- $x \text{ binop} = \text{expr}$
- $x++$
- $++x$
- $x--$
- $--x$

X is an lvalue of scalar type and binop is a non-overloaded built in operator.

Additional forms of atomic were added in OpenMP 3.1. We will discuss these later.

## Exercise 3

- In exercise 2, 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” from exercise 2 to avoid false sharing due to the sum array.

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

# 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, nthrds; 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) nthrds = 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<nthrds; 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 <sup>st</sup> 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 ()
{
    double pi;    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;
    id = omp_get_thread_num();
    nthrds = omp_get_num_threads();
    for (i=id, sum=0.0; i< num_steps; i=i+nthreads){
        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 ()
{
    double pi;    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;
        id = omp_get_thread_num();
        nthrds = omp_get_num_threads();
        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 <sup>st</sup> SPMD	1 <sup>st</sup> 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 ()
{
    double pi;    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;
    id = omp_get_thread_num();
    nthrds = omp_get_num_threads();
    for (i=id, sum=0.0; i< num_steps; i=i+nthrds){
        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?



## Example: Using an atomic to remove impact of false sharing

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

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 so updates don’t conflict

# Agenda

- Getting started with OpenMP
- Working with threads
- Synchronization in OpenMP
-  • Loop and single worksharing constructs
- OpenMP Data Environment
- OpenMP tasks
- Closing Comments

# SPMD vs. worksharing

- A parallel construct by itself creates an SPMD or “Single Program Multiple Data” program ... i.e., each thread redundantly executes the same code.
- How do you split up pathways through the code between threads within a team?
  - This is called worksharing
    - Loop construct
    - Sections/section constructs
    - Single construct
    - Task construct

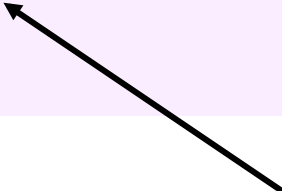
# The loop worksharing Constructs

- 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

## 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(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).

# 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

# 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

A diagram consisting of two arrows pointing from two separate code snippets towards a central text box. The first arrow originates from the left code block and points towards the bottom-left corner of the text box. The second arrow originates from the right code block and points towards the top-right corner of the text box. The text box itself is rectangular with a blue border and a blue shadow, containing the text "These are equivalent".



# 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.

# 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();
}
```

## Exercise 4: 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 reduction(+:var)
#pragma omp critical
int omp_get_num_threads();
int omp_get_thread_num();
double omp_get_wtime();
```

# 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;
}
```

# 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 <sup>st</sup> SPMD	1 <sup>st</sup> 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.



# Agenda

- Getting started with OpenMP
- Working with threads
- Synchronization in OpenMP
- Loop and single worksharing constructs
-  • OpenMP Data Environment
- OpenMP tasks
- Closing Comments

# 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.

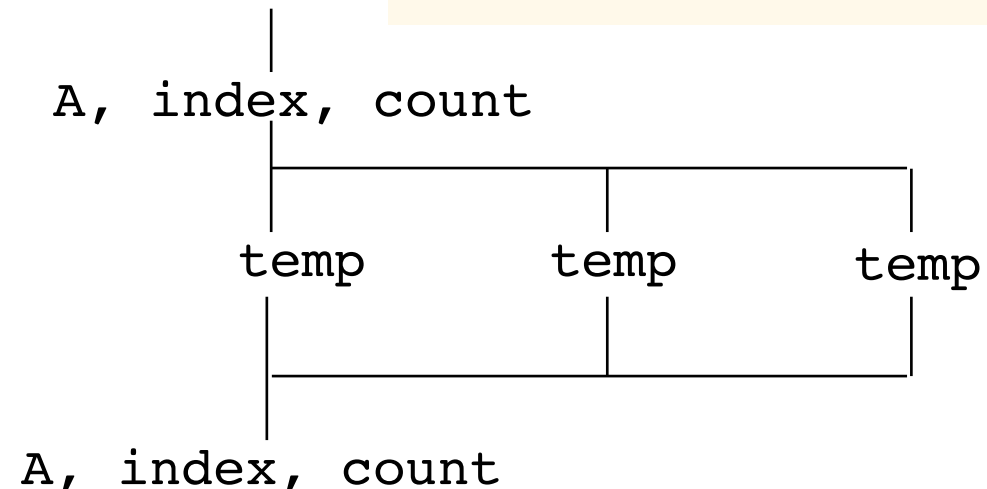
# Data sharing: Examples

```
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\*
  - SHARED
  - PRIVATE
  - FIRSTPRIVATE
- The final value of a private inside a parallel loop can be transmitted to the shared variable outside the loop with:
  - LASTPRIVATE
- The default attributes can be overridden with:
  - DEFAULT (PRIVATE | SHARED | NONE)

**All the clauses on this page apply to the OpenMP construct NOT to the entire region.**

*DEFAULT(PRIVATE) is Fortran only*

\*All data clauses apply to parallel constructs and worksharing constructs except “shared” which only applies to 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

# Firstprivate Clause

- Variables initialized from 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

# Example: Pi program ... minimal changes

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

```
static long num_steps = 100000;
```

```
double step;
```

```
void main ()
```

```
{    int i;    double x, pi, sum = 0.0;  
    step = 1.0/(double) num_steps;
```

```
#pragma omp parallel for private(x) reduction(+:sum)
```

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

i private by default

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

```
}
```

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

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

# Agenda

- Getting started with OpenMP
- Working with threads
- Synchronization in OpenMP
- Loop and single worksharing constructs
- OpenMP Data Environment
- • OpenMP tasks
- Closing Comments



# Consider simple list traversal

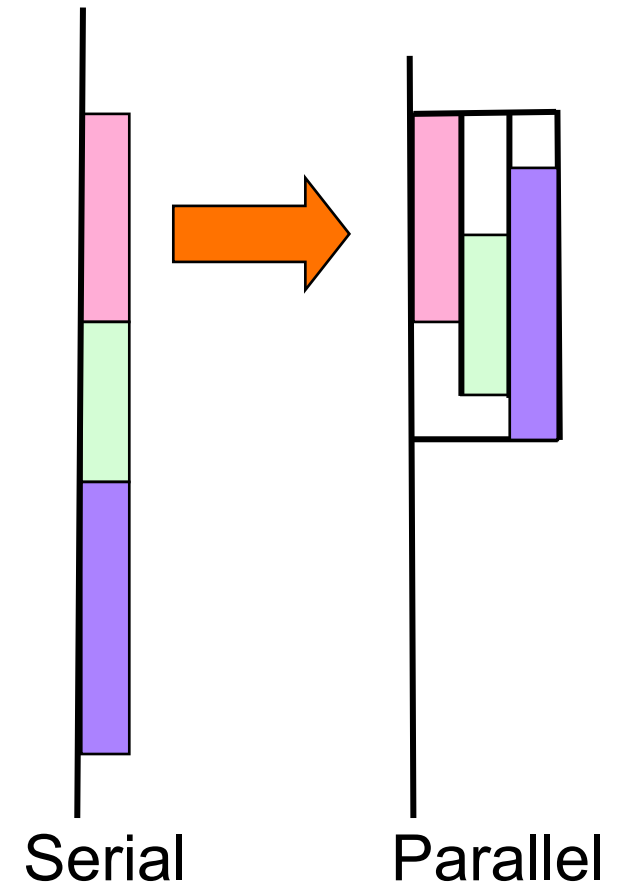
- Given what we've covered about OpenMP, how would you process this loop in Parallel?

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

- Remember, the loop worksharing construct only works with loops for which the number of loop iterations can be represented by a closed-form expression at compiler time. While loops are not covered.

# OpenMP Tasks

- Tasks are independent units of work.
- Tasks are composed of:
  - **code** to execute
  - **data** environment
  - **internal control variables (ICV)**
- Threads perform the work of each task.
- The runtime system decides when tasks are executed
  - Tasks may be deferred
  - Tasks may be executed immediately



# Task Construct – Explicit Tasks

1. Create a team of threads.

```
#pragma omp parallel  
{
```

2. One thread executes the **single** construct  
... other threads wait at the implied barrier at the end of the single construct

```
#pragma omp single  
{
```

3. The “single” thread creates a task with its own value for the pointer p

```
node * p = head;  
while (p) {
```

```
#pragma omp task firstprivate(p)  
    process(p);
```

```
    p = p->next;
```

```
}
```

```
}  
}
```

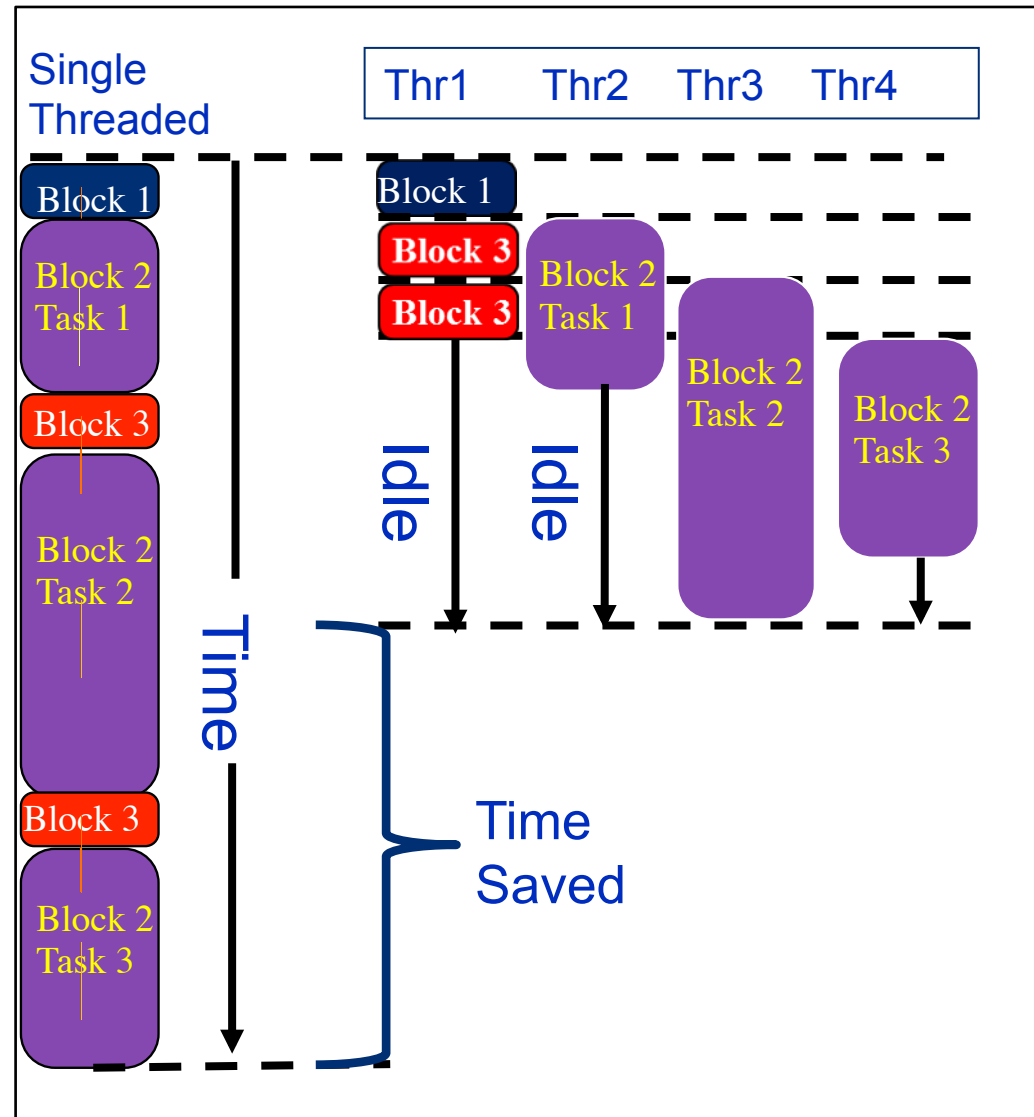
4. Threads waiting at the barrier execute tasks.

Execution moves beyond the barrier once all the tasks are complete

# Execution of tasks

Have potential to parallelize irregular patterns and recursive function calls

```
#pragma omp parallel
{
    #pragma omp single
    { //block 1
        node * p = head;
        while (p) { // block 2
            #pragma omp task
            process(p);
            p = p->next; //block 3
        }
    }
}
```



# When are tasks guaranteed to complete

- Tasks are guaranteed to be complete at thread barriers:

`#pragma omp barrier`

- or task barriers

`#pragma omp taskwait`

```
#pragma omp parallel  
{
```

```
    #pragma omp task  
    foo();
```

```
    #pragma omp barrier  
    #pragma omp single
```

```
{
```

```
    #pragma omp task  
    bar();
```

```
}
```

```
}
```

Multiple foo tasks created here – one for each thread

All foo tasks guaranteed to be completed here

One bar task created here

bar task guaranteed to be completed here

# Data Scoping with tasks: Fibonacci example.

This is an instance of the divide and conquer design pattern

```
int fib ( int n )  
{  
  
    int x,y;  
    if ( n < 2 ) return n;  
    #pragma omp task  
    x = fib(n-1);  
    #pragma omp task  
    y = fib(n-2);  
    #pragma omp taskwait  
    return x+y  
}
```

n is private in both tasks

x is a private variable  
y is a private variable

What's wrong here?

A task's private variables are  
undefined outside the task

# Data Scoping with tasks: Fibonacci example.

```
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;  
}
```

n is private in both tasks

x & y are shared  
Good solution  
we need both values to  
compute the sum

## Data Scoping with tasks: List Traversal example

```
List m1; //my_list
Element *e;
#pragma omp parallel
#pragma omp single
{
    for(e=m1->first;e;e=e->next)
#pragma omp task
    process(e);
}
```

What's wrong here?

Possible data race !  
Shared variable e  
updated by multiple tasks



## Data Scoping with tasks: List Traversal example

```
List m1; //my_list
Element *e;
#pragma omp parallel
#pragma omp single
{
    for(e=m1->first;e;e=e->next)
#pragma omp task firstprivate(e)
    process(e);
}
```

Good solution – e is  
firstprivate

## Exercise 5: tasks in OpenMP

- Start with your pi program.
- Parallelize this program using tasks.



# OpenMP PI Program:

## Loop level parallelism pattern

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

    pi = sum * step;
}
```

# Results\*: pi with tasks

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

Program: OpenMP tasks (divide and conquer pattern)				
<pre>#include &lt;omp.h&gt; 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 &lt; MIN_BLK){     for (i=Nstart;i&lt; 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; }</pre>				
<pre>int main () {   int i;   double step, pi, sum;   step = 1.0/(double) num_steps;   #pragma omp parallel   {</pre>				
threads	1 <sup>st</sup> SPMD	SPMD critical	PI Loop	Pi tasks
1	1.86	1.87	1.91	1.87
2	1.03	1.00	1.02	1.00
3	1.08	0.68	0.80	0.76
4	0.97	0.53	0.68	0.52

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

# Agenda

- Getting started with OpenMP
- Working with threads
- Synchronization in OpenMP
- Loop and single worksharing constructs
- OpenMP Data Environment
- OpenMP tasks
- ➔ • Closing Comments

# Summary

- We have now covered the most commonly used features of OpenMP.
- To close, let's consider some of the key parallel design patterns we've discussed..

# SPMD: Single Program Multiple Data

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

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

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

## OpenMP Pi program: SPMD pattern

```
#include <omp.h>
void main (int argc, char *argv[])
{
    int i, pi=0.0, step, sum = 0.0;
    step = 1.0/(double) num_steps ;
    #pragma omp parallel firstprivate(sum) private(x, i)
    {
        int id = omp_get_thread_num();
        int numprocs = omp_get_num_threads();
        int step1 = id *num_steps/numprocs ;
        int stepN = (id+1)*num_steps/numprocs;
        if (stepN != num_steps) stepN = num_steps;
        for (i=step1; i<stepN; i++)
        {
            x = (i+0.5)*step;
            sum += 4.0/(1.0+x*x);
        }
        #pragma omp critical
            pi += sum *step ;
    }
}
```



# Loop parallelism

- Collections of tasks are defined as iterations of one or more loops.
- Loop iterations are divided between a collection of processing elements to compute tasks in parallel.

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

This design pattern is heavily used with data parallel design patterns.

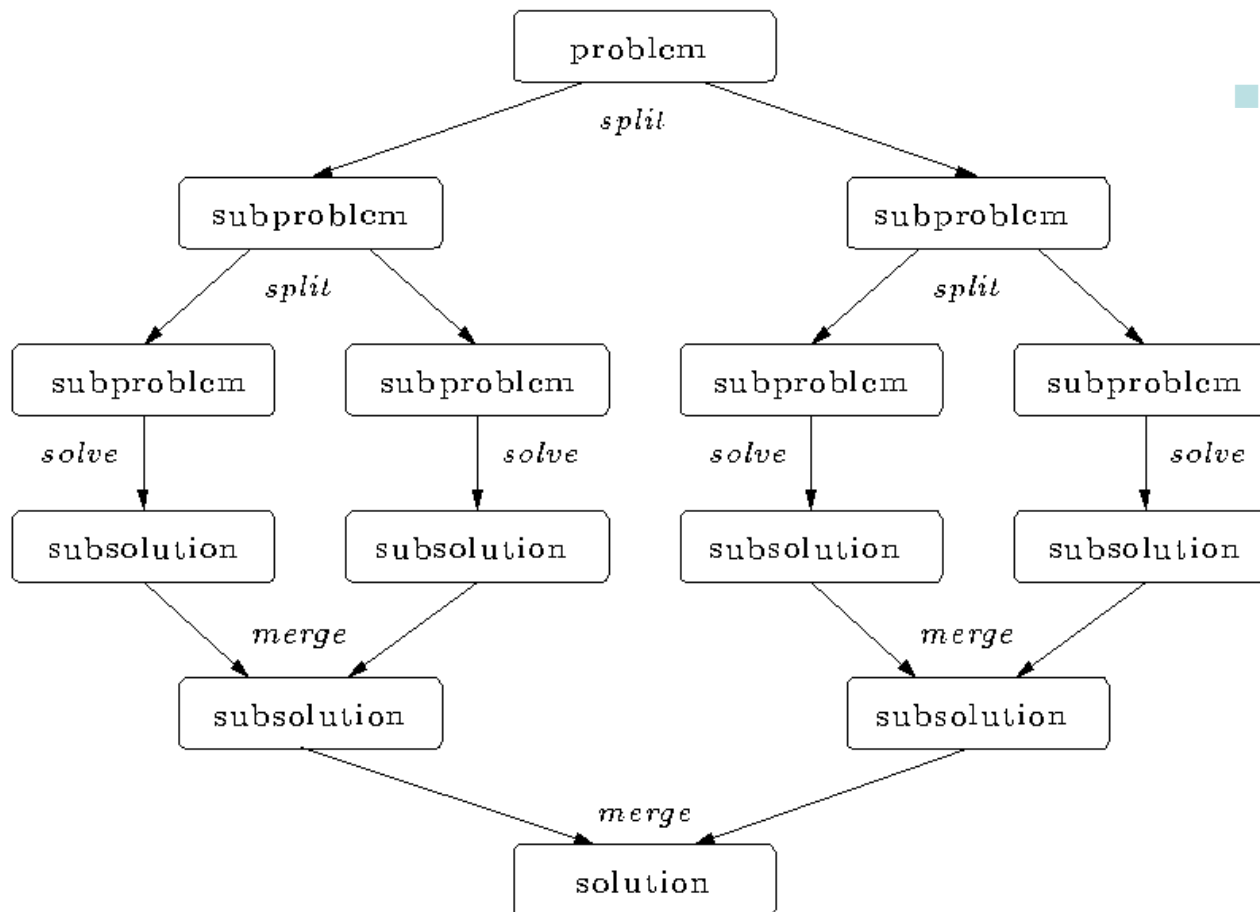
OpenMP programmers commonly use this pattern.

# Divide and Conquer Pattern

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

# Divide and conquer

- Split the problem into smaller sub-problems. Continue until the sub-problems can be solve directly.



## ■ 3 Options:

- Do work as you split into sub-problems.
- Do work only at the leaves.
- Do work as you recombine.

## Program: OpenMP tasks (divide and conquer pattern)

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

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

# Learning more about OpenMP:

## OpenMP Organizations

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

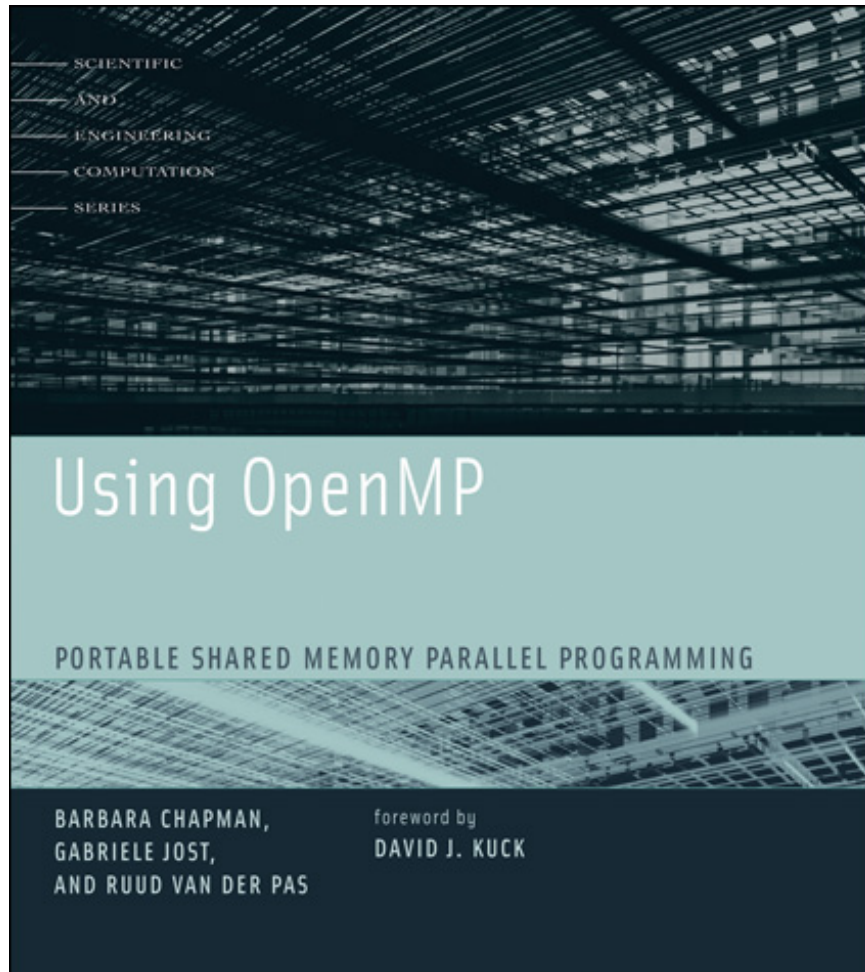
**[www.openmp.org](http://www.openmp.org)**

- OpenMP User’s Group (cOMPunity) URL:

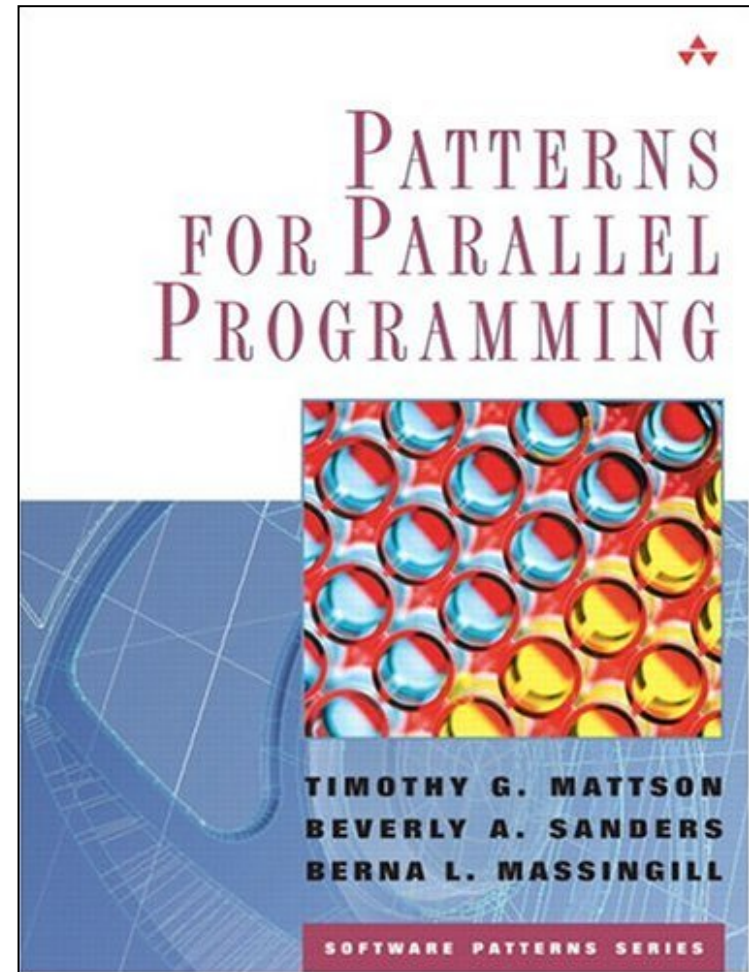
**[www.compunity.org](http://www.compunity.org)**

Get involved, join compunity and help  
define the future of OpenMP

# Books about OpenMP



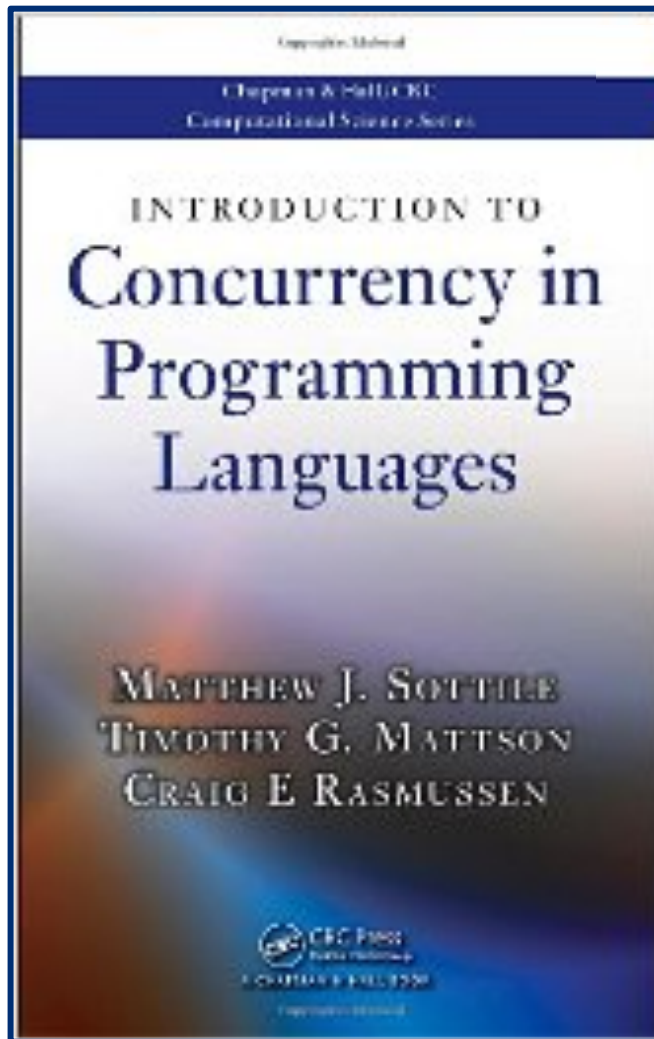
An excellent book about using OpenMP ... though out of date (OpenMP 2.5)



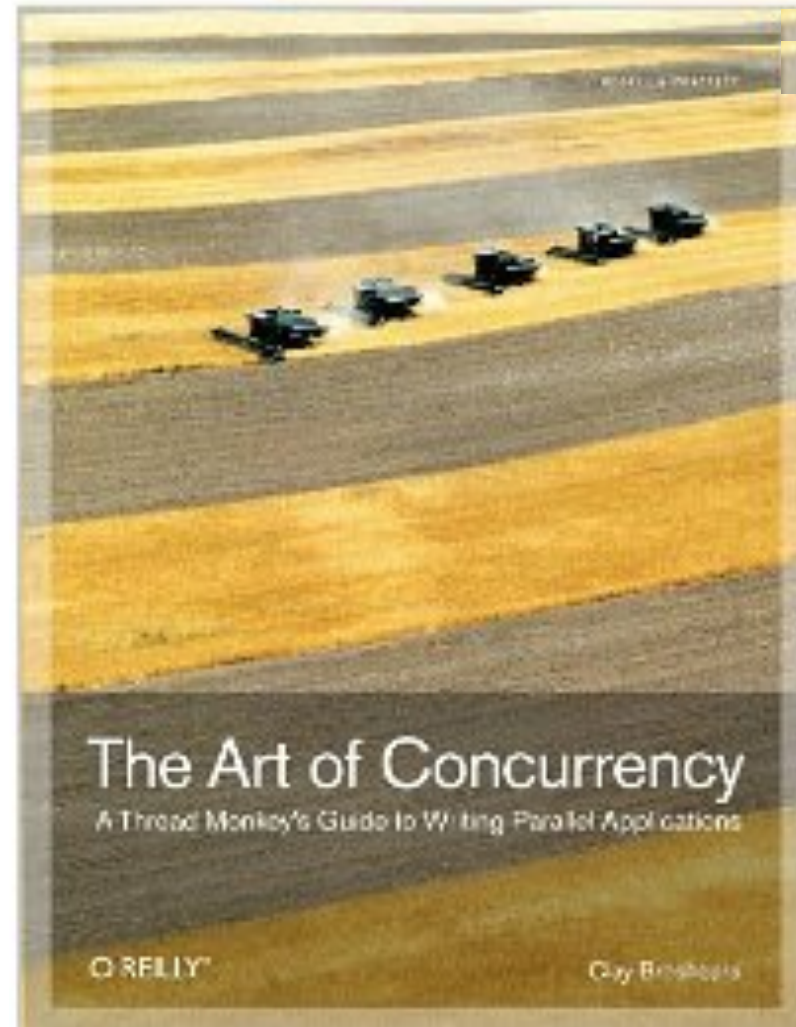
A book about how to “think parallel” with examples in OpenMP, MPI and Java



# Background references



A general reference that puts languages such as OpenMP in perspective (by Sottile, Mattson, and Rasmussen)



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

# The OpenMP reference card

A two page summary of all the OpenMP constructs ... don't write OpenMP code without it.

OpenMP 3.1 API C/C++

Page 1

# OpenMP

## C/C++

### OpenMP 3.1 API C/C++ Syntax Quick Reference Card

OpenMP (OpenMP Programming Interface) is a portable, scalable model that gives shared-memory parallel programmers a simple and flexible interface for developing parallel applications for platforms ranging from the desktop to the supercomputer.

[[Link](#)] refers to sections in the OpenMP API specification available at [www.openmp.org](http://www.openmp.org).

OpenMP supports multi-platform shared-memory parallel programming in C++ and Fortran on all architectures, including Unix platforms and Windows NT platforms.

A separate OpenMP reference card for Fortran is also available.

## Directives

An OpenMP executable directive applies to the succeeding structured block or an OpenMP Construct. A structured block is a single statement or a compound statement with a single entry at the top and a single exit at the bottom.

### Parallel [2.4]

The parallel construct forms a team of threads and starts parallel execution.

**Openmp omp parallel** [*clause*] { *clause*... }

*structured block*

*clause*:

```
{forall-expression
  num_threads(integer-expression)
  default(clause) | none
  private(list)
  copy_private(list)
  shared(list)
  update(list)
  reduction(operator list)
}
```

### Loop [2.5.1]

The loop construct specifies that the iterations of loops will be distributed among and executed by the accelerating team of threads.

**Openmp omp for** [*clause*] { *clause*... }

*for-range*

*clause*:

```
{private(list)
  firstprivate(list)
  lastprivate(list)
  reduction(operator list)
  schedule(type, [chunk_size])
  collapse(n)
  ordered
  nowait
}
```

Most common form of the for loop

```
for (i = 0; i < n; i++)
  /* relational-op to
  set i = i+1 */
```

*list*:

- static: iterations are divided into chunks of size *chunk\_size*. Chunks are assigned to threads in the team in round-robin fashion in order of thread number.
- dynamic: each thread executes a chunk of iterations; then requests another chunk until no chunks remain to be distributed.
- guided: each thread executes a chunk of iterations; then requests another chunk until no chunks remain to be assigned. The chunk sizes start large and shrink to the indicated *chunk\_size* as chunks are scheduled.
- auto: The decision regarding scheduling is delegated to the compiler and/or runtime system.
- runtime: The schedule and chunk size are taken from the run-time variable OX.

### Sections [2.5.3]

The sections construct contains a set of structured blocks that are to be distributed among and executed by the accelerating team of threads.

**Openmp omp sections** [*clause*] { *clause*... }

```
{
  (Openmp omp section)
  structured-block
  (Openmp omp section)
  structured-block
  ...
}
```

*clause*:

```
{forall-expression
  write
  default(clause) | none
  mergeable
  private(list)
  firstprivate(list)
  shared(list)
  update(list)
  nowait
}
```

### Single [2.5.2]

The single construct specifies that the associated structured block is executed by only one of the threads in the team (not necessarily the master thread), in the context of its implicit task.

**Openmp omp single** [*clause*] { *clause*... }

*structured-block*

*clause*:

```
{forall(list)
  firstprivate(list)
  copyprivate(list)
  nowait
}
```

### Parallel Loop [2.5.1]

The parallel loop construct is a shortcut for specifying a parallel construct containing one or more associated loops and no other directives.

**Openmp omp parallel for** [*clause*] { *clause*... }

*for-range*

*clause*:

Any accepted by the parallel or for directive, except the *nowait* clause, with identical meanings and restrictions.

### Simple Parallel Loop Example

The following example demonstrates how to parallelize a simple loop using the parallel loop construct.

```
void simple(int n, float *a, float *b)
{
  for (i = 0; i < n; i++)
    /* i is private by default */
    b[i] = a[i] * a[i] * i;
}
```

### Parallel Sections [2.5.3]

The parallel sections construct is a shortcut for specifying a parallel construct containing one sections construct and no other directives.

**Openmp omp parallel sections** [*clause*] { *clause*... }

```
{
  (Openmp omp section)
  structured-block
  (Openmp omp section)
  structured-block
  ...
}
```

*clause*:

Any of the clauses accepted by the parallel or sections directives, except the *nowait* clause, with identical meanings and restrictions.

### Task [2.5.2]

The task construct defines an explicit task; the data environment of the task is created according to the data-sharing attributes clauses on the task construct and any defaults that apply.

**Openmp omp task** [*clause*] { *clause*... }

*structured-block*

*clause*:

```
{forall-expression
  write
  default(clause) | none
  mergeable
  private(list)
  firstprivate(list)
  shared(list)
  update(list)
  nowait
}
```

### Taskwait [2.5.2]

The taskwait construct specifies that the current task can be suspended for a time of execution of a different task.

**Openmp omp taskwait**

### Master [2.5.1]

The master construct specifies a structured block that is executed by the master thread of the team. There is no implicit barrier after entry to, or exit from, the master construct.

**Openmp omp master**

*structured-block*

### Critical [2.5.3]

The critical construct restricts execution of the associated structured block to a single thread at a time.

OpenMP API 3.1 C/C++

Page 2

## Runtime Library Routines

### Execution Environment Routines [3.1.1]

Execution environment routines affect and monitor threads, processors, and the parallel environment.

#### `void omp_get_num_threads(`

`int *num_threads)`

Affects the number of threads used for subsequent parallel regions that do not specify a new threads clause.

#### `int omp_get_max_threads(void)`

Returns the number of threads in the current team.

#### `int omp_get_num_threads(void)`

Returns maximum number of threads that could be used to form a new team, using a parallel construct without a new\_threads clause.

#### `int omp_get_max_rank(void)`

Returns the ID of the enclosing thread where ID ranges from zero to one less than the team minus 1.

#### `int omp_get_num_procs(void)`

Returns the number of processors available to the program.

### Data Types for Runtime Library Routines

`omp_int_1` Represents a single int.

`omp_int_2` Represents a double int.

`omp_int_3` Represents a schedule.

#### `int omp_is_parallel(void)`

Returns true if the call to the routine is executed by an active parallel region; otherwise, it returns false.

#### `void omp_set_dynamic(`

`int dynamic, threads)`

Enables or disables dynamic adjustment of the number of threads available by setting the value of the `dynamic` flag.

#### `int omp_set_dynamic(void)`

Returns the value of the `dynamic` flag, determining whether dynamic adjustment of the number of threads is enabled or disabled.

#### `void omp_set_schedule(sched_type, int order)`

Enables or disables nested parallelism, by setting the `sched_type` flag.

#### `int omp_set_schedule(void)`

Returns the value of the `sched_type` flag, which determines if nested parallelism is enabled or disabled.

#### `void omp_set_schedule(sched_type, int kind, int mode)`

Enables the schedule that is applied when routine is used as a schedule kind, by setting the value of the `kind` flag.

`kind` is one of static, dynamic, guided, auto, or an implementation-defined schedule. See loop-construct [3.1.2] for descriptions.

#### `void omp_set_schedule(`

`omp_sched_1 *kind, int *mode)`

Returns the value of `kind` and `mode` flags, which is the schedule applied when routine is used.

See kind above.

#### `int omp_set_thread_limit(void)`

Returns the value of the `thread_limit` flag, which is the maximum number of OpenMP threads available to the program.

#### `void omp_set_wtime_active_level(`

`int time_level)`

Limits the number of nested active parallel regions, by setting `time_level` flag.

#### `int omp_set_wtime_active_level(void)`

Returns the value of `time_level` flag, which determines the maximum number of nested active parallel regions.

#### `int omp_set_wtime(void)`

Returns the number of nested parallel regions enclosing the task that contains the call.

#### `int omp_set_wtime_thread_max(`

`int time_max)`

Returns, for a given nested level of the current thread, the thread number of the ancestor or the current thread.

#### `int omp_set_wtime_thread_max(`

`int time_max)`

Returns, for a given nested level of the current thread, the size of the thread tree to which the ancestor or the current thread belongs.

#### `int omp_set_wtime_thread_max(`

Returns the number of nested, active parallel regions enclosing the task that contains the call.

#### `int omp_is_thread(void)`

Returns true if the routine is executed in a thread or included task region; otherwise, it returns false.

### Lock Routines [3.1.3]

Lock routines support synchronization with OpenMP locks.

#### `void omp_lock_init(omp_lock_t *lock)`

These routines initialize an OpenMP lock.

#### `void omp_unlock_init(omp_lock_t *lock)`

These routines initialize an OpenMP lock.

#### `void omp_lock_test_lock(omp_lock_t *lock)`

These routines ensure that the OpenMP lock is initialized.

#### `void omp_test_lock(omp_lock_t *lock)`

These routines provide a means of waiting on OpenMP lock.

#### `void omp_unlock_test_lock(omp_lock_t *lock)`

These routines provide a means of waiting on OpenMP lock.

#### `int omp_test_lock(omp_lock_t *lock)`

These routines attempt to set an OpenMP lock but do not support recursion of the task executing the routine.

#### `int omp_unlock_test_lock(omp_lock_t *lock)`

These routines attempt to set an OpenMP lock but do not support recursion of the task executing the routine.

#### `int omp_test_lock(omp_lock_t *lock)`

These routines attempt to set an OpenMP lock but do not support recursion of the task executing the routine.

#### `int omp_unlock_test_lock(omp_lock_t *lock)`

These routines attempt to set an OpenMP lock but do not support recursion of the task executing the routine.

#### `double omp_get_wtime(void)`

Returns elapsed wall clock time in

<http://openmp.org/mp-documents/OpenMP3.1-CCard.pdf>