



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

**Tim Mattson**

**Intel Corp.**

**[timothy.g.mattson@intel.com](mailto:timothy.g.mattson@intel.com)**

# 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).

# 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:



# 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.

# Outline



- **Unit 1: Getting started with OpenMP**
  - ◆ Mod1: Introduction to parallel programming
  - ◆ Mod 2: The boring bits: Using an OpenMP compiler (hello world)
  - ◆ Disc 1: Hello world and how threads work
- **Unit 2: The core features of OpenMP**
  - ◆ Mod 3: Creating Threads (the Pi program)
  - ◆ Disc 2: The simple Pi program and why it sucks
  - ◆ Mod 4: Synchronization (Pi program revisited)
  - ◆ Disc 3: Synchronization overhead and eliminating false sharing
  - ◆ Mod 5: Parallel Loops (making the Pi program simple)
  - ◆ Disc 4: Pi program wrap-up
- **Unit 3: Working with OpenMP**
  - ◆ Mod 6: Synchronize single masters and stuff
  - ◆ Mod 7: Data environment
  - ◆ Disc 5: Debugging OpenMP programs
  - ◆ Mod 8: Skills practice ... linked lists and OpenMP
  - ◆ Disc 6: Different ways to traverse linked lists
- **Unit 4: a few advanced OpenMP topics**
  - ◆ Mod 8: Tasks (linked lists the easy way)
  - ◆ Disc 7: Understanding Tasks
  - ◆ Mod 8: The scary stuff ... Memory model, atomics, and flush (pairwise synch).
  - ◆ Disc 8: The pitfalls of pairwise synchronization
  - ◆ Mod 9: Threadprivate Data and how to support libraries (Pi again)
  - ◆ Disc 9: Random number generators
- **Unit 5: Recapitulation**

# Outline

- **Unit 1: Getting started with OpenMP**



- ◆ Mod1: Introduction to parallel programming
- ◆ Mod 2: The boring bits: Using an OpenMP compiler (hello world)
- ◆ Disc 1: Hello world and how threads work

- **Unit 2: The core features of OpenMP**

- ◆ Mod 3: Creating Threads (the Pi program)
- ◆ Disc 2: The simple Pi program and why it sucks
- ◆ Mod 4: Synchronization (Pi program revisited)
- ◆ Disc 3: Synchronization overhead and eliminating false sharing
- ◆ Mod 5: Parallel Loops (making the Pi program simple)
- ◆ Disc 4: Pi program wrap-up

- **Unit 3: Working with OpenMP**

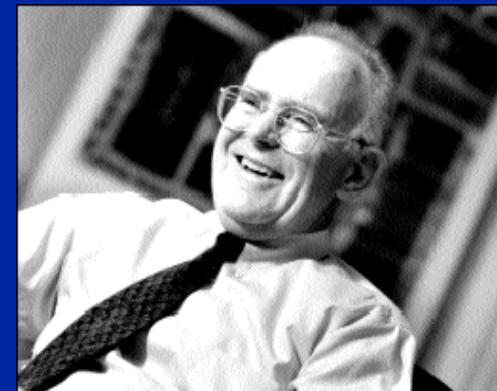
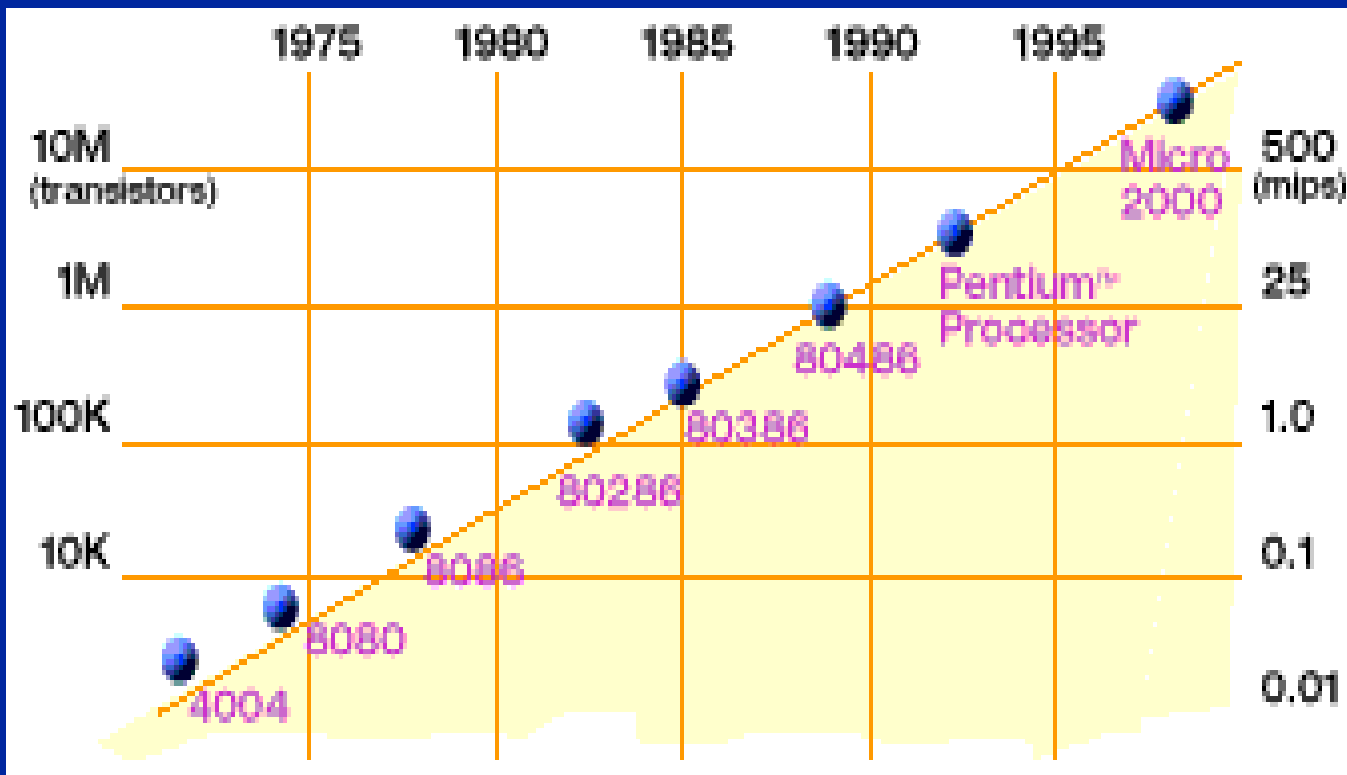
- ◆ Mod 6: Synchronize single masters and stuff
- ◆ Mod 7: Data environment
- ◆ Disc 5: Debugging OpenMP programs
- ◆ Mod 8: Skills practice ... linked lists and OpenMP
- ◆ Disc 6: Different ways to traverse linked lists

- **Unit 4: a few advanced OpenMP topics**

- ◆ Mod 8: Tasks (linked lists the easy way)
- ◆ Disc 7: Understanding Tasks
- ◆ Mod 8: The scary stuff ... Memory model, atomics, and flush (pairwise synch).
- ◆ Disc 8: The pitfalls of pairwise synchronization
- ◆ Mod 9: Threadprivate Data and how to support libraries (Pi again)
- ◆ Disc 9: Random number generators

- **Unit 5: Recapitulation**

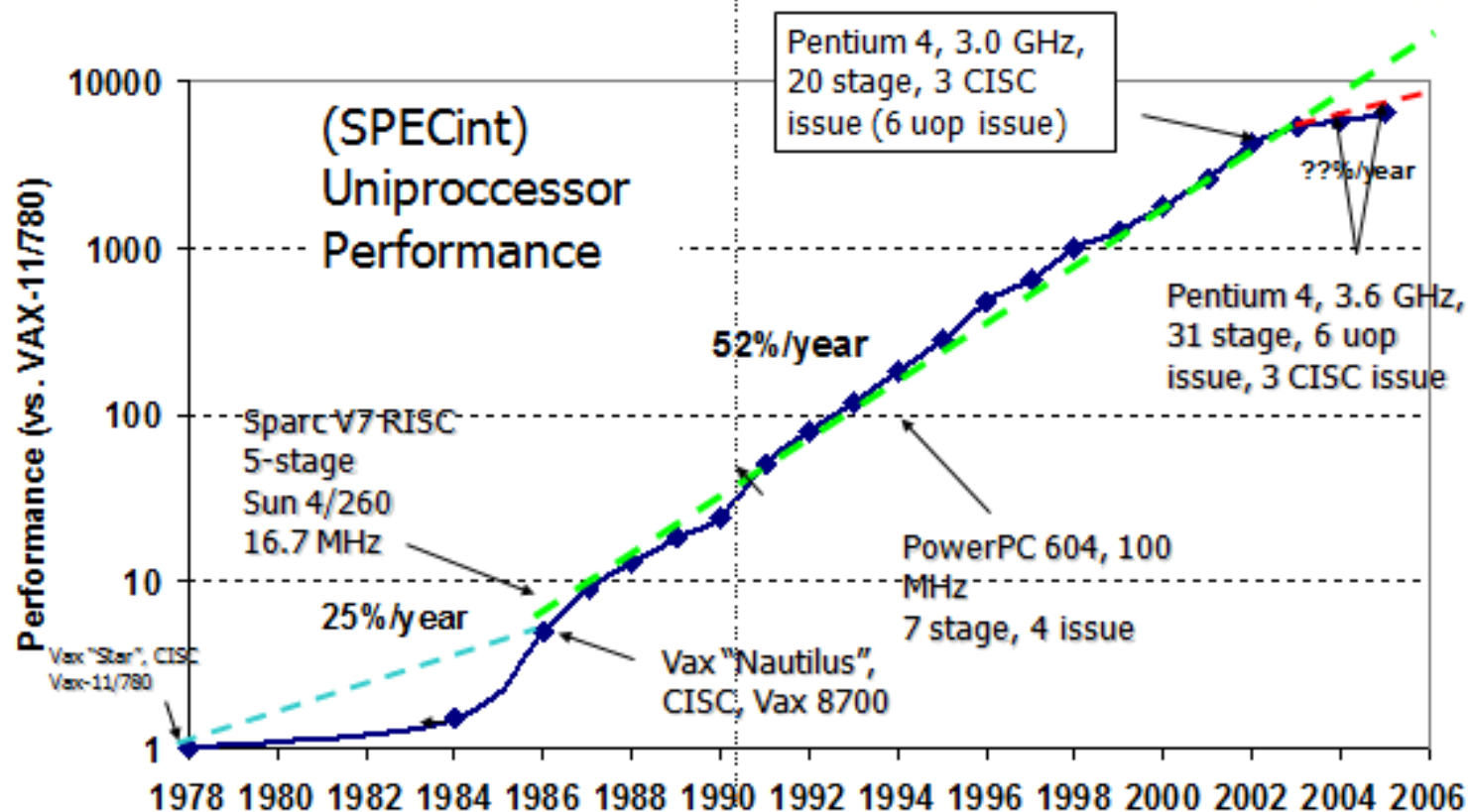
# Moore's Law



- In 1965, Intel co-founder Gordon Moore predicted (from just 3 data points!) that semiconductor density would double every 18 months.
  - ◆ *He was right!* Transistors are *still* shrinking as he projected.

# Consequences of Moore's law...

## The good old days ...



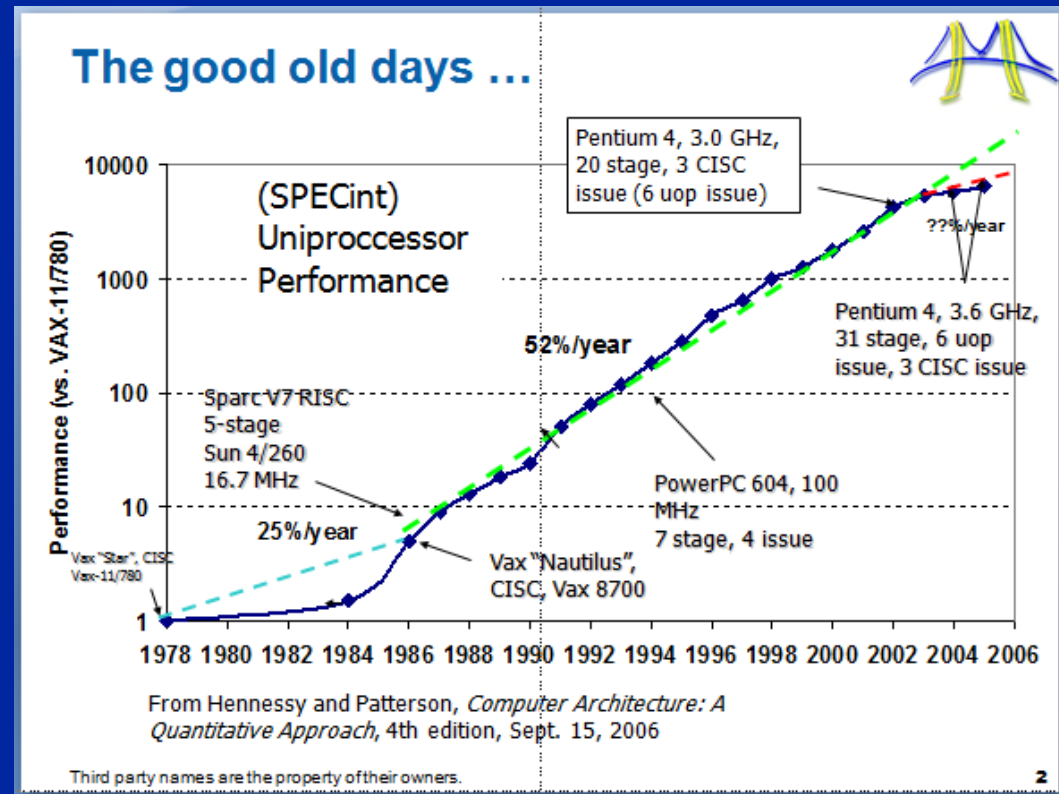
From Hennessy and Patterson, *Computer Architecture: A Quantitative Approach*, 4th edition, Sept. 15, 2006

Third party names are the property of their owners.



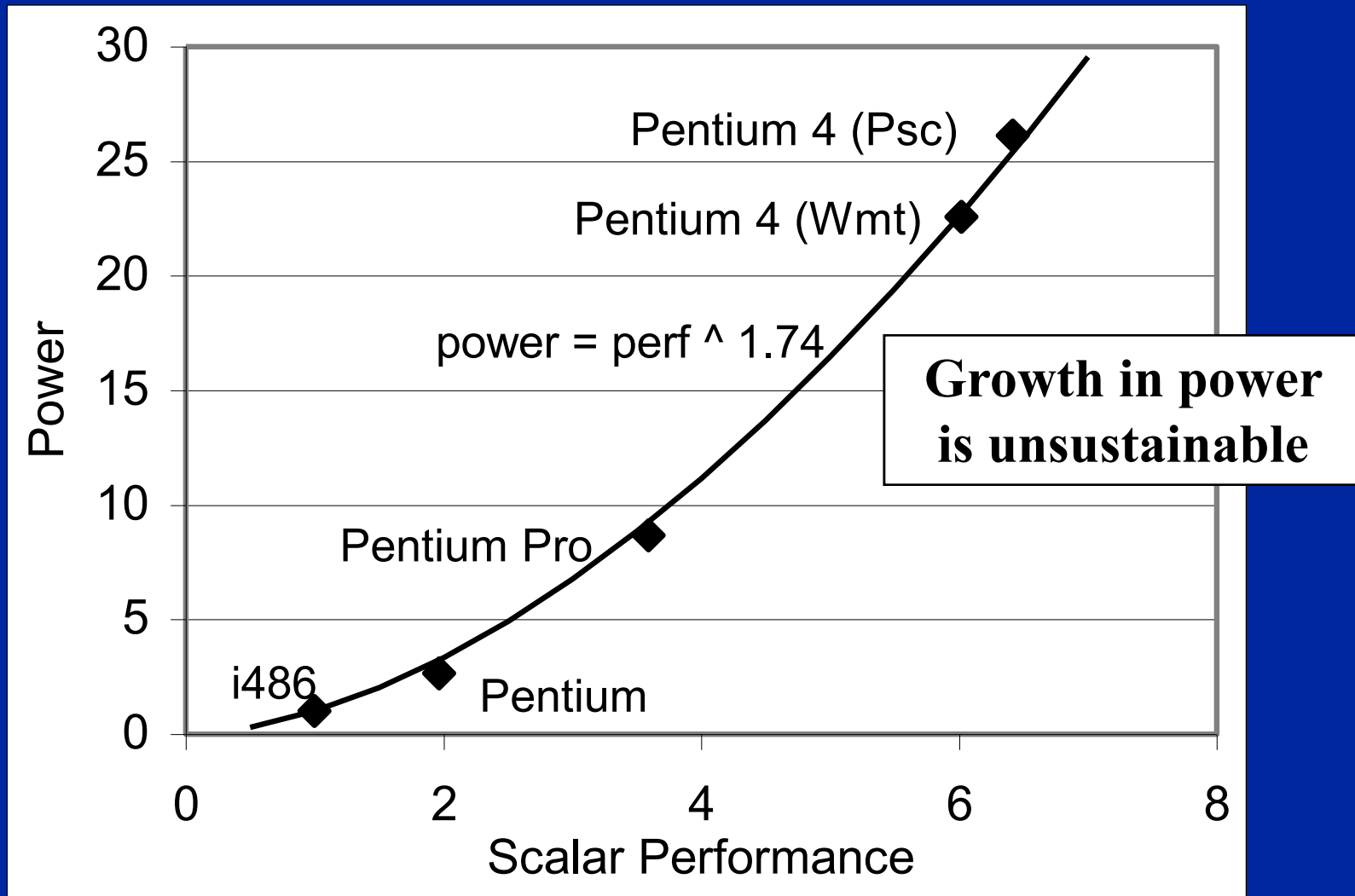
# The Hardware/Software contract

- Write your software as you choose and we HW-geniuses will take care of performance.



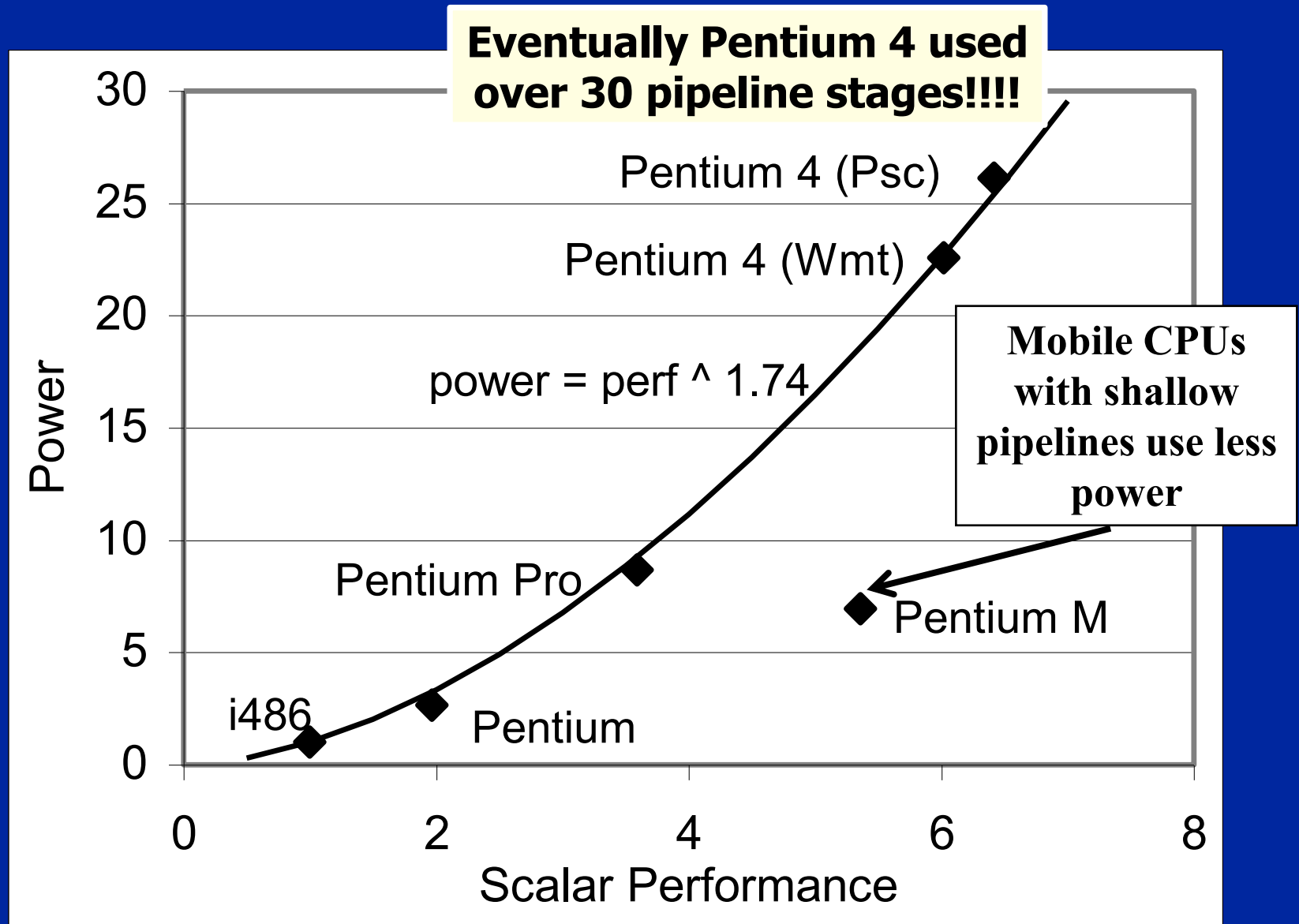
- The result: Generations of performance ignorant software engineers using performance-handicapped languages (such as Java) ... which was OK since performance was a HW job.

## ... Computer architecture and the power wall

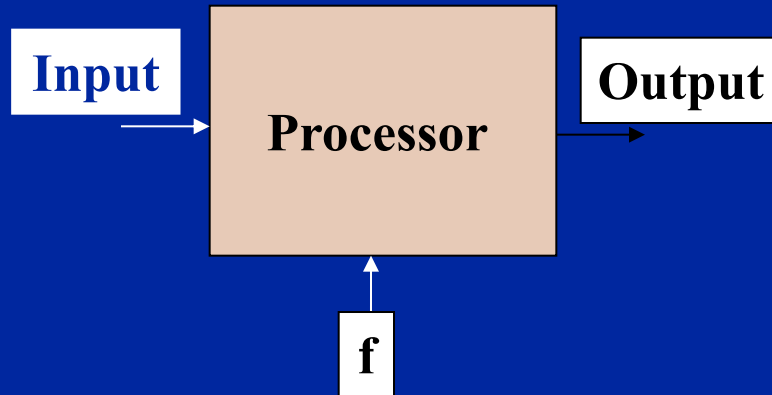
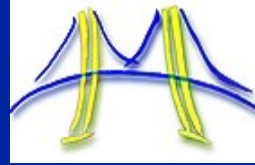


Source: E. Grochowski of Intel

## ... partial solution: simple low power cores



# For the rest of the solution consider power in a chip ...



Capacitance =  $C$   
Voltage =  $V$   
Frequency =  $f$   
Power =  $CV^2f$

$C$  = capacitance ... it measures the ability of a circuit to store energy:

$$C = q/V \rightarrow q = CV$$

Work is pushing something (charge or  $q$ ) across a "distance" ... in electrostatic terms pushing  $q$  from 0 to  $V$ :

$$V * q = W.$$

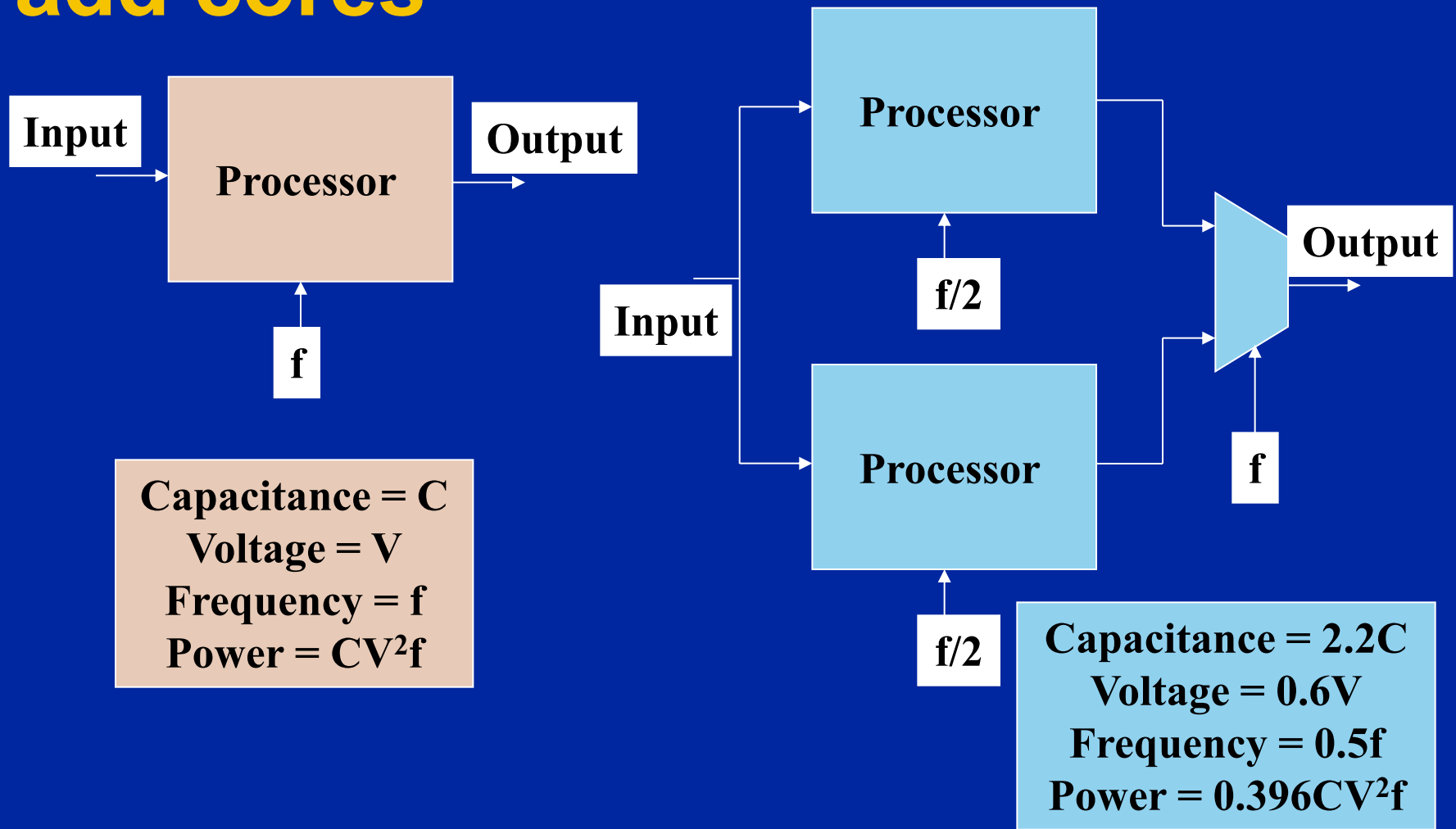
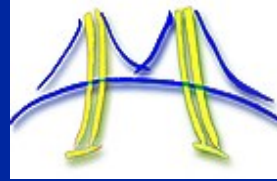
But for a circuit  $q = CV$  so

$$W = CV^2$$

power is work over time ... or how many times in a second we oscillate the circuit

$$\text{Power} = W * F \rightarrow \text{Power} = CV^2f$$

# ... The rest of the solution add cores

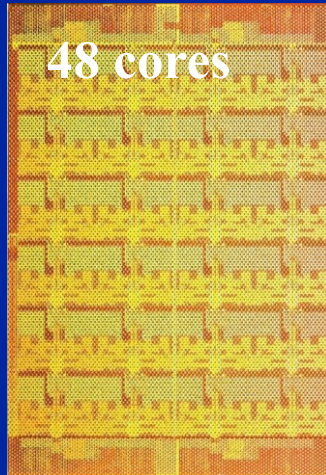


Chandrakasan, A.P.; Potkonjak, M.; Mehra, R.; Rabaey, J.; Brodersen, R.W.,  
"Optimizing power using transformations," *IEEE Transactions on Computer-Aided  
Design of Integrated Circuits and Systems*, vol.14, no.1, pp.12-31, Jan 1995

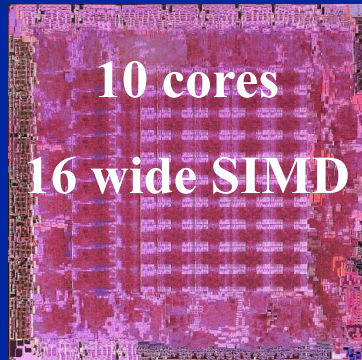
Source:  
Vishwani Agrawal

# Microprocessor trends

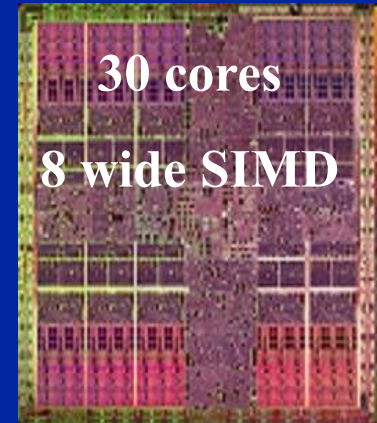
Individual processors are many core (and often heterogeneous) processors.



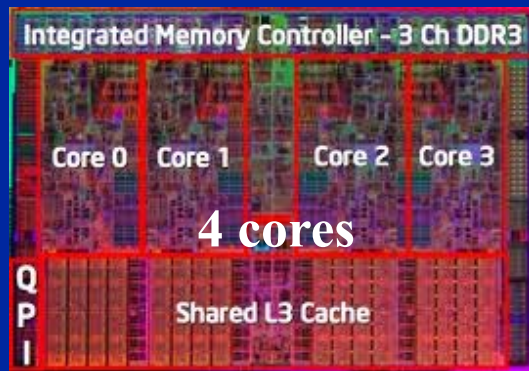
Intel SCC Processor



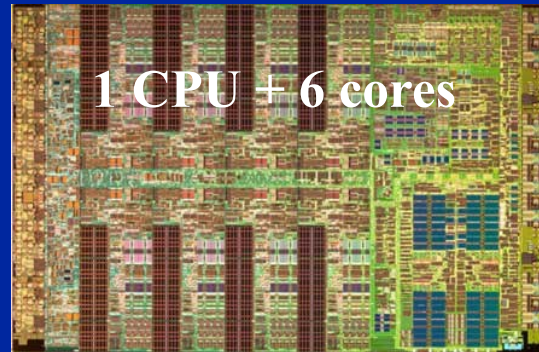
AMD ATI RV770



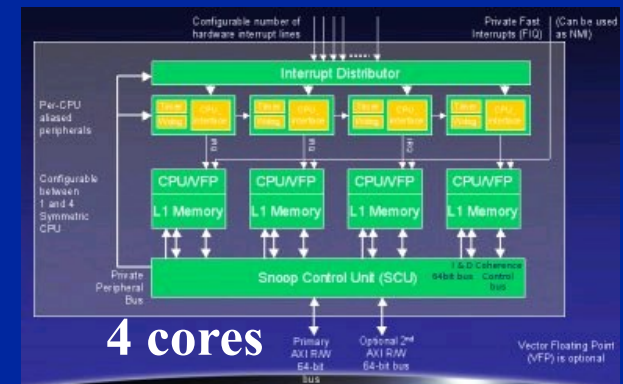
NVIDIA Tesla C1060



Intel® Xeon® processor



IBM Cell

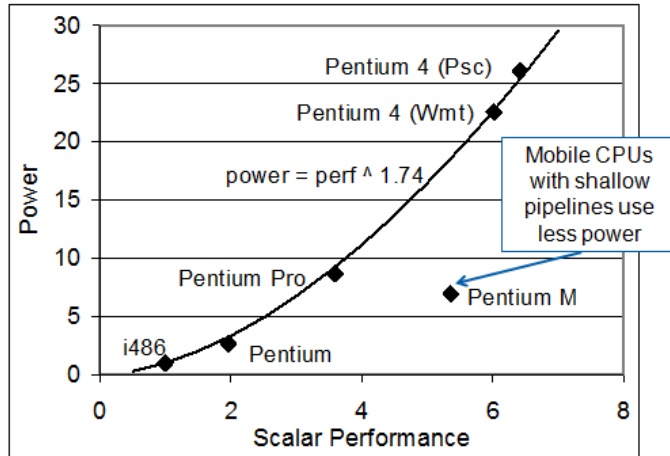


ARM MPCORE

Source: OpenCL tutorial, Gaster, Howes, Mattson, and Lokhmotov, HiPEAC 2011

# The result...

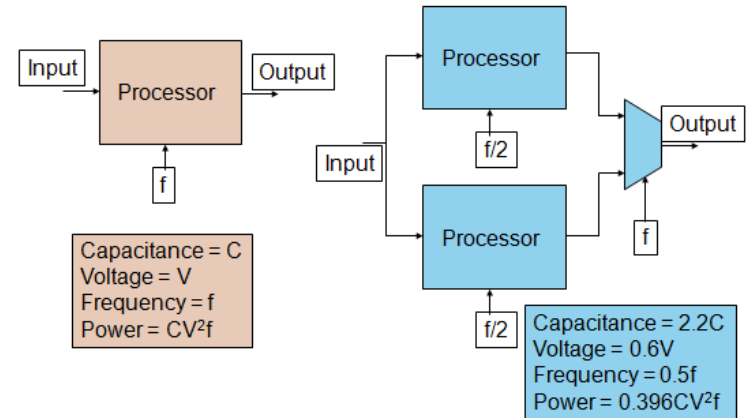
... partial solution: simple low power cores



Source: E. Grochowski of Intel

+

How multiple cores reduce power



Chandrakasan, A.P.; Potkonjak, M.; Mehra, R.; Rabaey, J.; Brodersen, R.W., "Optimizing power using transformations," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol.14, no.1, pp.12-31, Jan 1995

Source: Vishwani Agrawal

=

**A new contract ... HW people will do what's natural for them (lots of simple cores) and SW people will have to adapt (rewrite everything)**

**The problem is this was presented as an ultimatum ... nobody asked us if we were OK with this new contract ... which is kind of rude.**



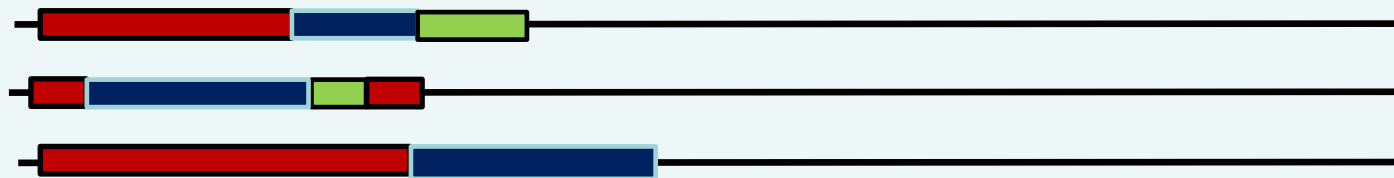
# Concurrency vs. Parallelism

- Two important definitions:

- ◆ Concurrency: A condition of a system in which multiple tasks are *logically* active at one time.
- ◆ Parallelism: A condition of a system in which multiple tasks are actually active at one time.



Concurrent, non-parallel Execution

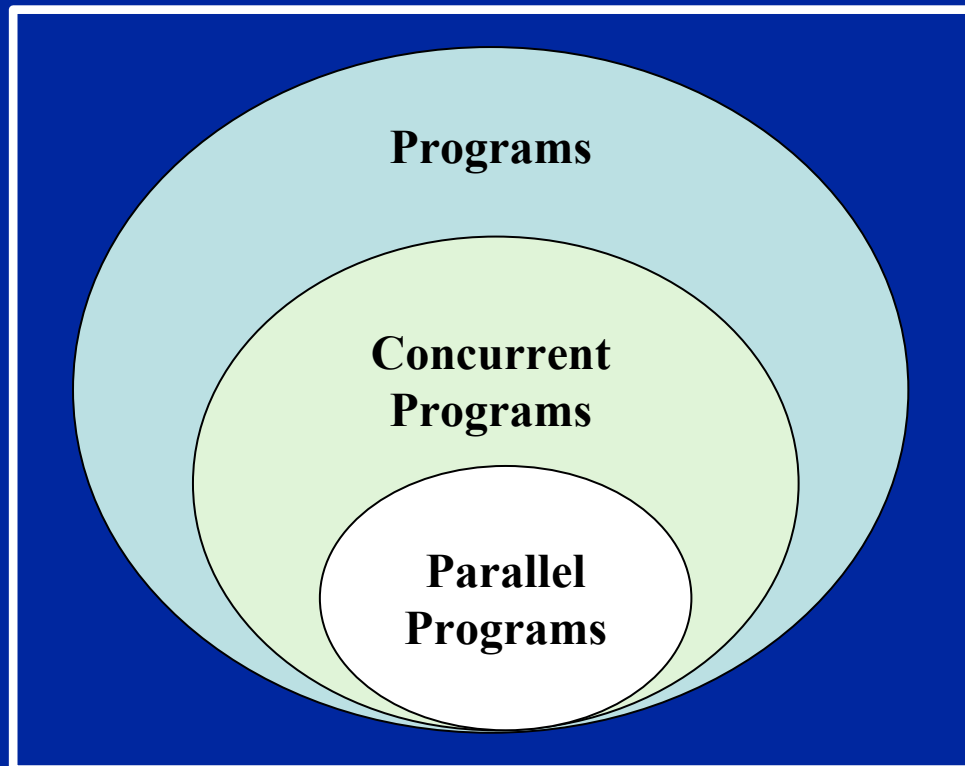


Concurrent, parallel Execution



# Concurrency vs. Parallelism

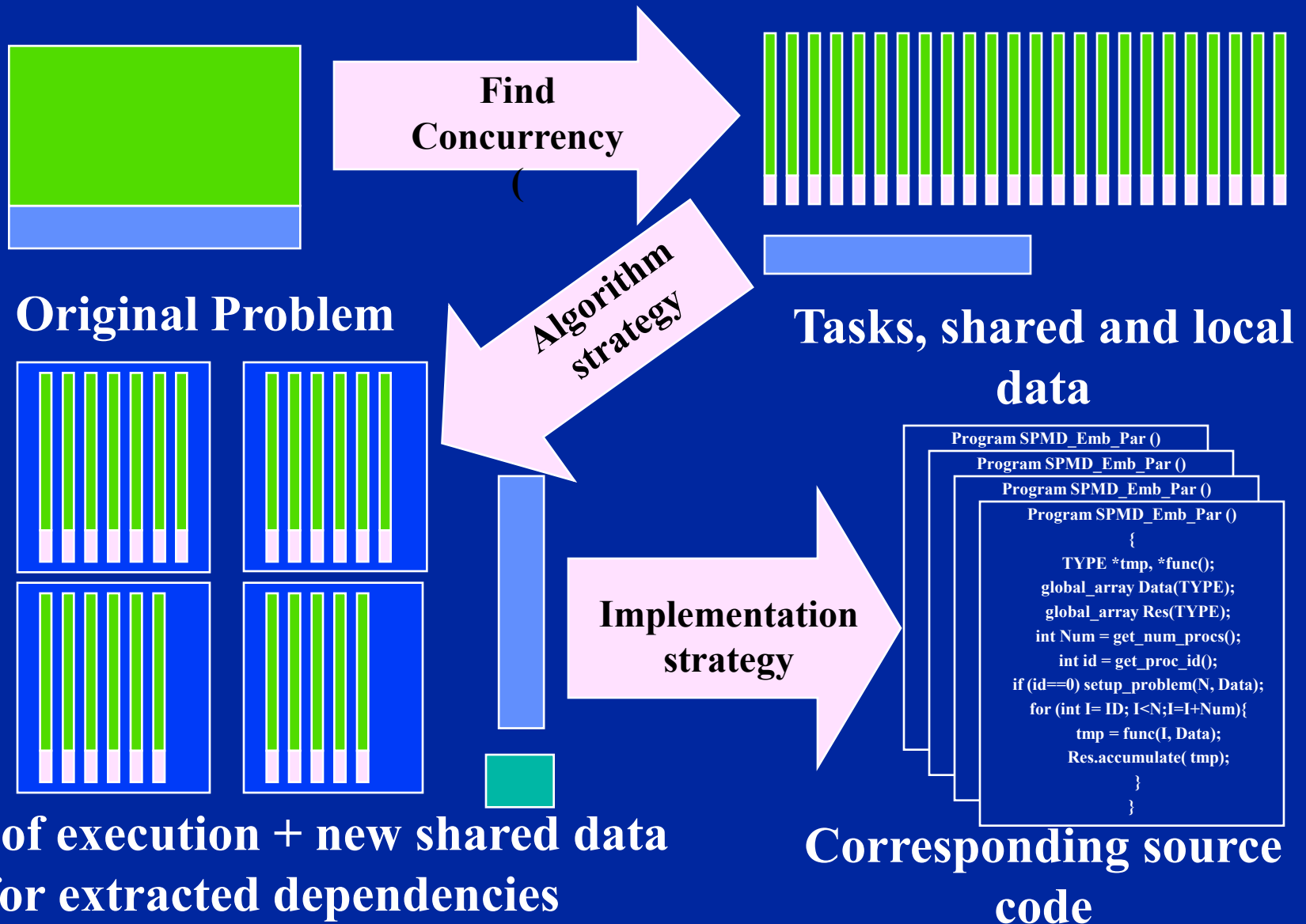
- Two important definitions:
  - ◆ Concurrency: A condition of a system in which multiple tasks are *logically* active at one time.
  - ◆ Parallelism: A condition of a system in which multiple tasks are actually active at one time.



# Concurrent vs. Parallel applications

- We distinguish between two classes of applications that exploit the concurrency in a problem:
  - Concurrent application: An application for which computations **logically** execute simultaneously due to the semantics of the application.
    - The problem is fundamentally concurrent.  
Eg- Web Server
  - Parallel application: An application for which the computations **actually** execute simultaneously in order to complete a problem in less time.
    - The problem doesn't inherently require concurrency ... you can state it sequentially.

# The Parallel programming process:



# 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 last 20 years of SMP practice

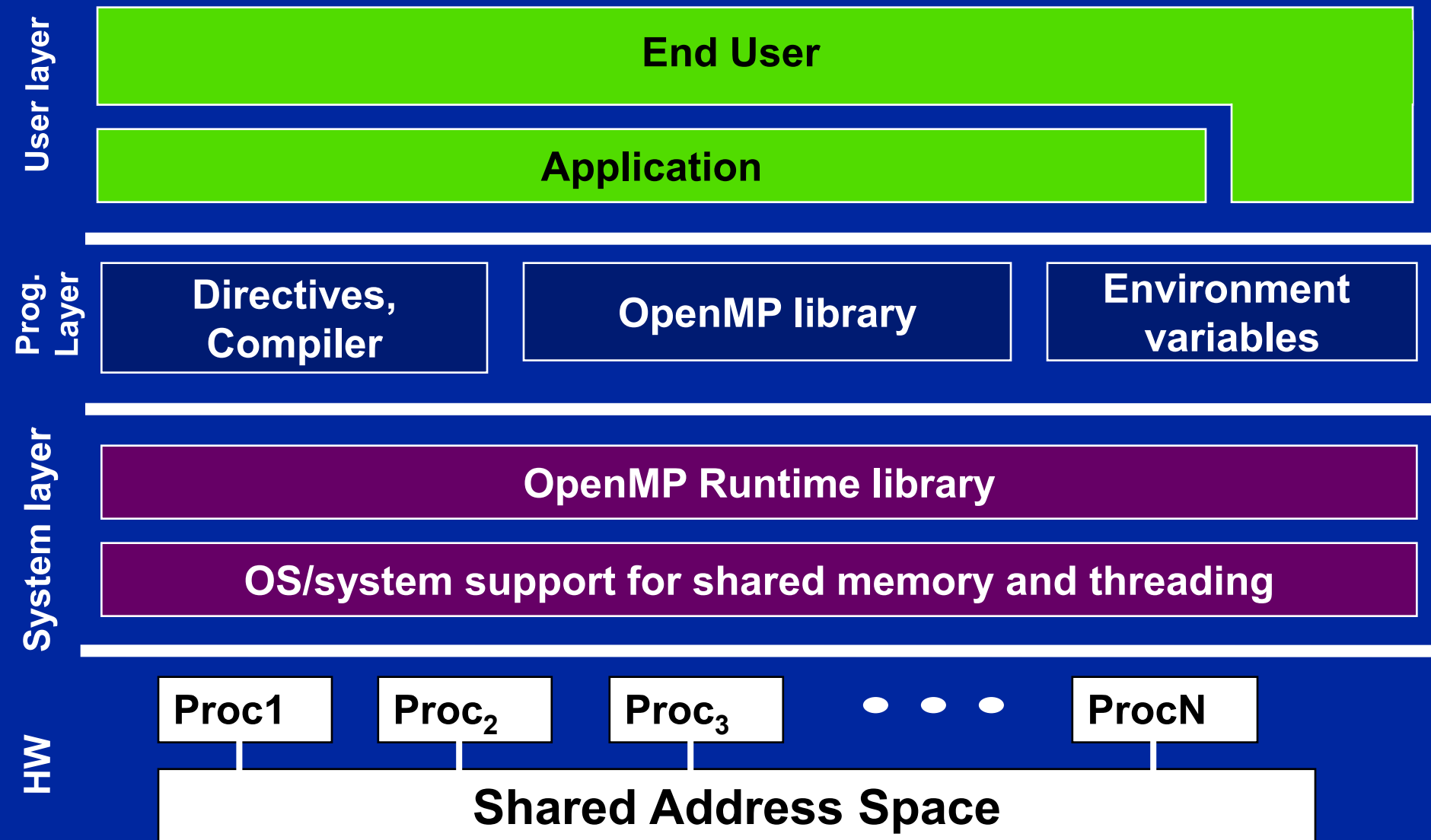
`C$OMP PARALLEL COPYIN (/blk/)`

`C$OMP DO lastprivate (XX)`

`Nthrds = OMP_GET_NUM_PROCS ()`

`omp_set_lock (lck)`

# 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”.

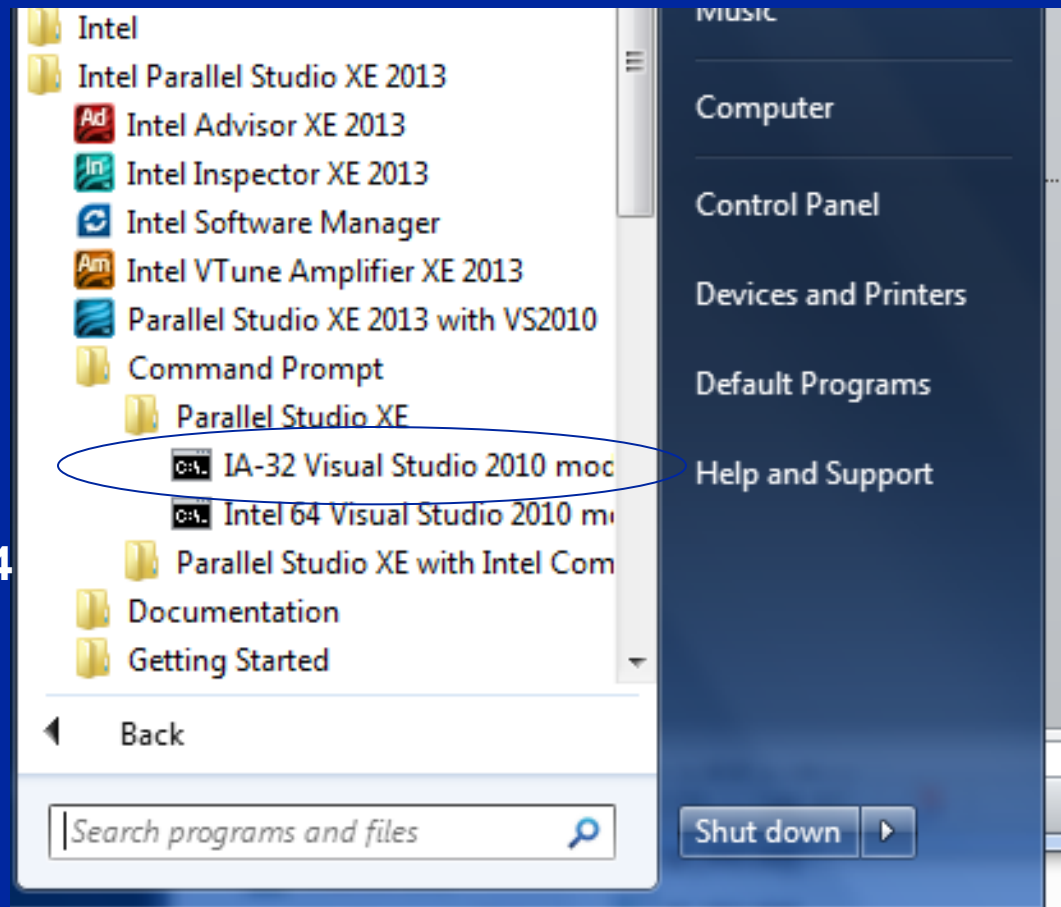
- ◆ Structured block: a block of one or more statements with one point of entry at the top and one point of exit at the bottom.
- ◆ It's OK to have an exit() within the structured block.

# Outline

- **Unit 1: Getting started with OpenMP**
  - ◆ Mod1: Introduction to parallel programming
  - ➡ ◆ Mod 2: The boring bits: Using an OpenMP compiler (hello world)
  - ◆ Disc 1: Hello world and how threads work
- **Unit 2: The core features of OpenMP**
  - ◆ Mod 3: Creating Threads (the Pi program)
  - ◆ Disc 2: The simple Pi program and why it sucks
  - ◆ Mod 4: Synchronization (Pi program revisited)
  - ◆ Disc 3: Synchronization overhead and eliminating false sharing
  - ◆ Mod 5: Parallel Loops (making the Pi program simple)
  - ◆ Disc 4: Pi program wrap-up
- **Unit 3: Working with OpenMP**
  - ◆ Mod 6: Synchronize single masters and stuff
  - ◆ Mod 7: Data environment
  - ◆ Disc 5: Debugging OpenMP programs
  - ◆ Mod 8: Skills practice ... linked lists and OpenMP
  - ◆ Disc 6: Different ways to traverse linked lists
- **Unit 4: a few advanced OpenMP topics**
  - ◆ Mod 8: Tasks (linked lists the easy way)
  - ◆ Disc 7: Understanding Tasks
  - ◆ Mod 8: The scary stuff ... Memory model, atomics, and flush (pairwise synch).
  - ◆ Disc 8: The pitfalls of pairwise synchronization
  - ◆ Mod 9: Threadprivate Data and how to support libraries (Pi again)
  - ◆ Disc 9: Random number generators
- **Unit 5: Recapitulation**

# 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

# 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

# Outline

- **Unit 1: Getting started with OpenMP**
  - ◆ Mod1: Introduction to parallel programming
  - ◆ Mod 2: The boring bits: Using an OpenMP compiler (hello world)
  - ➡ ◆ Disc 1: Hello world and how threads work
- **Unit 2: The core features of OpenMP**
  - ◆ Mod 3: Creating Threads (the Pi program)
  - ◆ Disc 2: The simple Pi program and why it sucks
  - ◆ Mod 4: Synchronization (Pi program revisited)
  - ◆ Disc 3: Synchronization overhead and eliminating false sharing
  - ◆ Mod 5: Parallel Loops (making the Pi program simple)
  - ◆ Disc 4: Pi program wrap-up
- **Unit 3: Working with OpenMP**
  - ◆ Mod 6: Synchronize single masters and stuff
  - ◆ Mod 7: Data environment
  - ◆ Disc 5: Debugging OpenMP programs
  - ◆ Mod 8: Skills practice ... linked lists and OpenMP
  - ◆ Disc 6: Different ways to traverse linked lists
- **Unit 4: a few advanced OpenMP topics**
  - ◆ Mod 8: Tasks (linked lists the easy way)
  - ◆ Disc 7: Understanding Tasks
  - ◆ Mod 8: The scary stuff ... Memory model, atomics, and flush (pairwise synch).
  - ◆ Disc 8: The pitfalls of pairwise synchronization
  - ◆ Mod 9: Threadprivate Data and how to support libraries (Pi again)
  - ◆ Disc 9: Random number generators
- **Unit 5: Recapitulation**

# Exercise 1: Solution

## A multi-threaded “Hello world” program

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

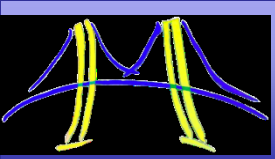
```
#include "omp.h"
int main()
{
    #pragma omp parallel
    {
        int ID = omp_get_thread_num();
        printf(" hello(%d) ", ID);
        printf(" world(%d) \n", ID);
    }
}
```

OpenMP include file

Parallel region with default number of threads

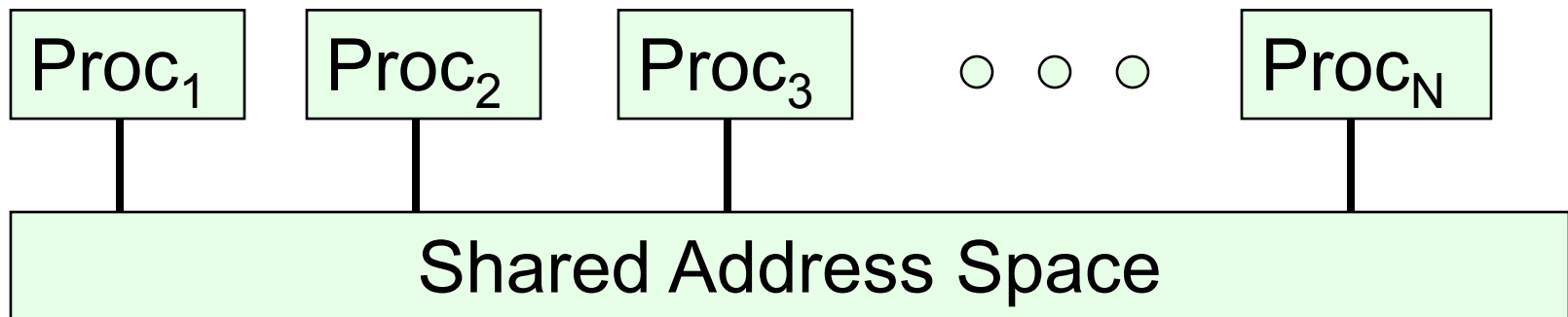
Runtime library function to return a thread ID.

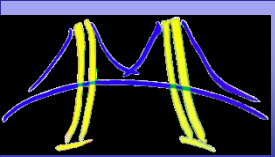
End of the Parallel region



# 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.



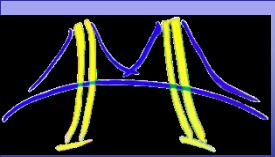


# Shared memory machines: SMP

- Cray-2 ... the last large scale SMP computer.
- Released in 1985 with 4 “heads”, 1.9 GFLOPS peak performance (fastest supercomputer in the world until 1990).
- The vector units in each “head” had equal-time access to the memory organized into banks to support high-bandwidth parallel memory access

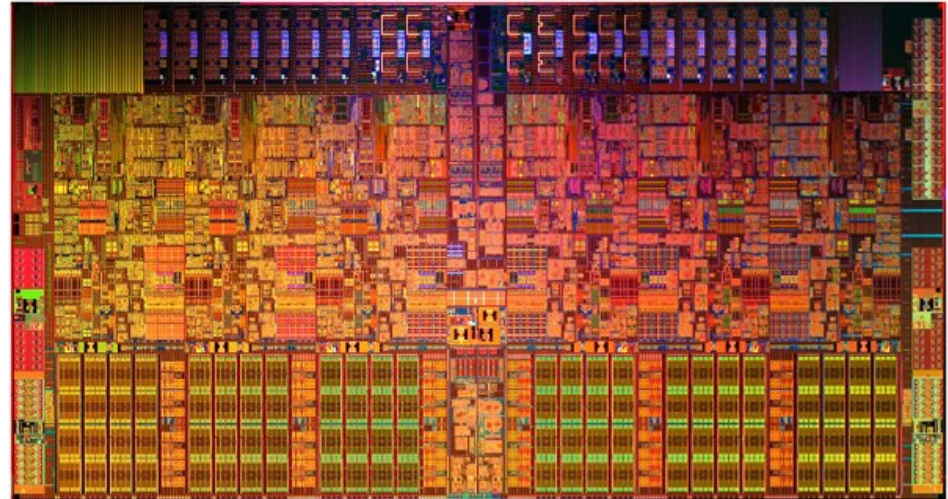
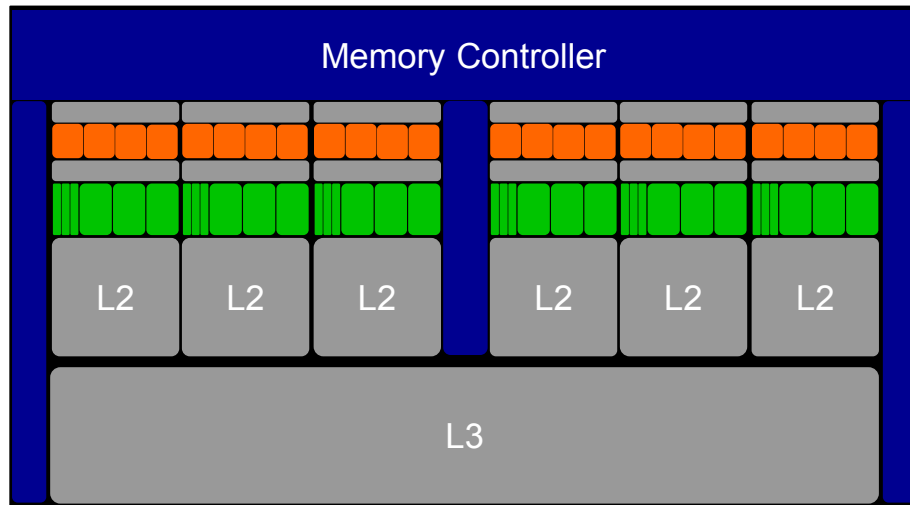






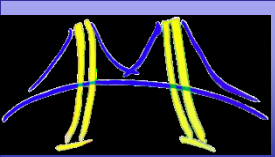
# Shared memory machines: SMP

Intel® Core™ i7-970 processor: Often called an SMP, but is it?



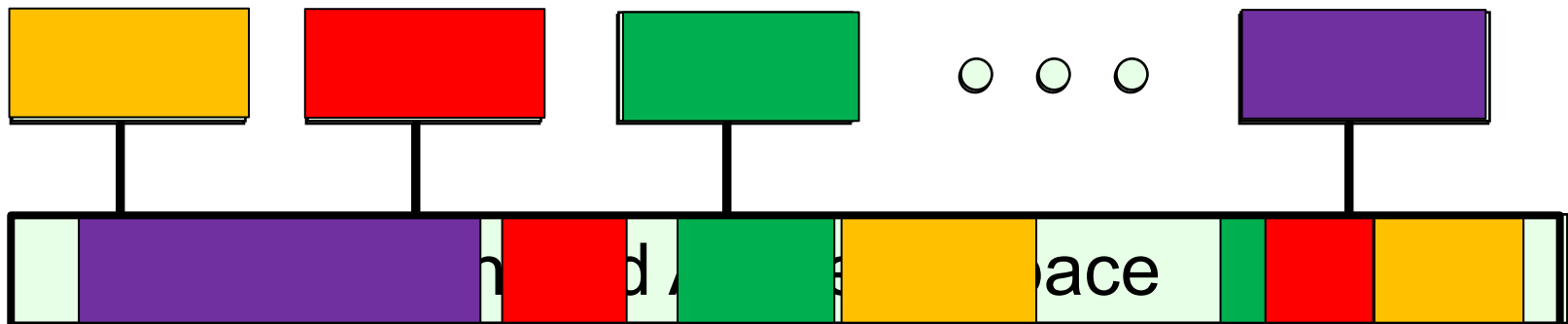
- 6 cores, 2-way multithreaded, 6-wide superscalar, quad-issue, 4-wide SIMD (on 3 of 6 pipelines)
- 4.5 KB (6 x 768 B) “Architectural” Registers, 192 KB (6 x 32 KB) L1 Cache, 1.5 MB (6 x 256 KB) L2 cache, 12 MB L3 Cache
- MESIF Cache Coherence, Processor Consistency Model
- 1.17 Billion Transistors on 32 nm process @ 2.6 GHz

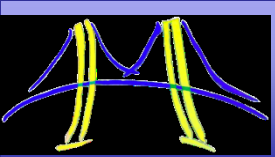
Cache hierarchy means different processors have different costs to access different address ranges .... It's NUMA



# Shared memory computers

- Shared memory computers are everywhere ... most laptops and servers have multicore multiprocessor CPUs
- The shared address space and (as we will see) programming models encourage us to think of them as SMP systems.
- Reality is more complex ... any multiprocessor CPU with a cache is a NUMA system. Start out by treating the system as an SMP and just accept that much of your optimization work will address cases where that case breaks down.





# Programming shared memory computers

**Stack**

funcA() var1  
var2

Stack Pointer  
Program Counter  
Registers

## Process

- An instance of a program execution.
- The execution context of a running program ... i.e. the resources associated with a program's execution.

**text**

main()  
funcA()  
funcB()  
.....

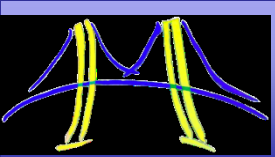
**data**

array1  
array2

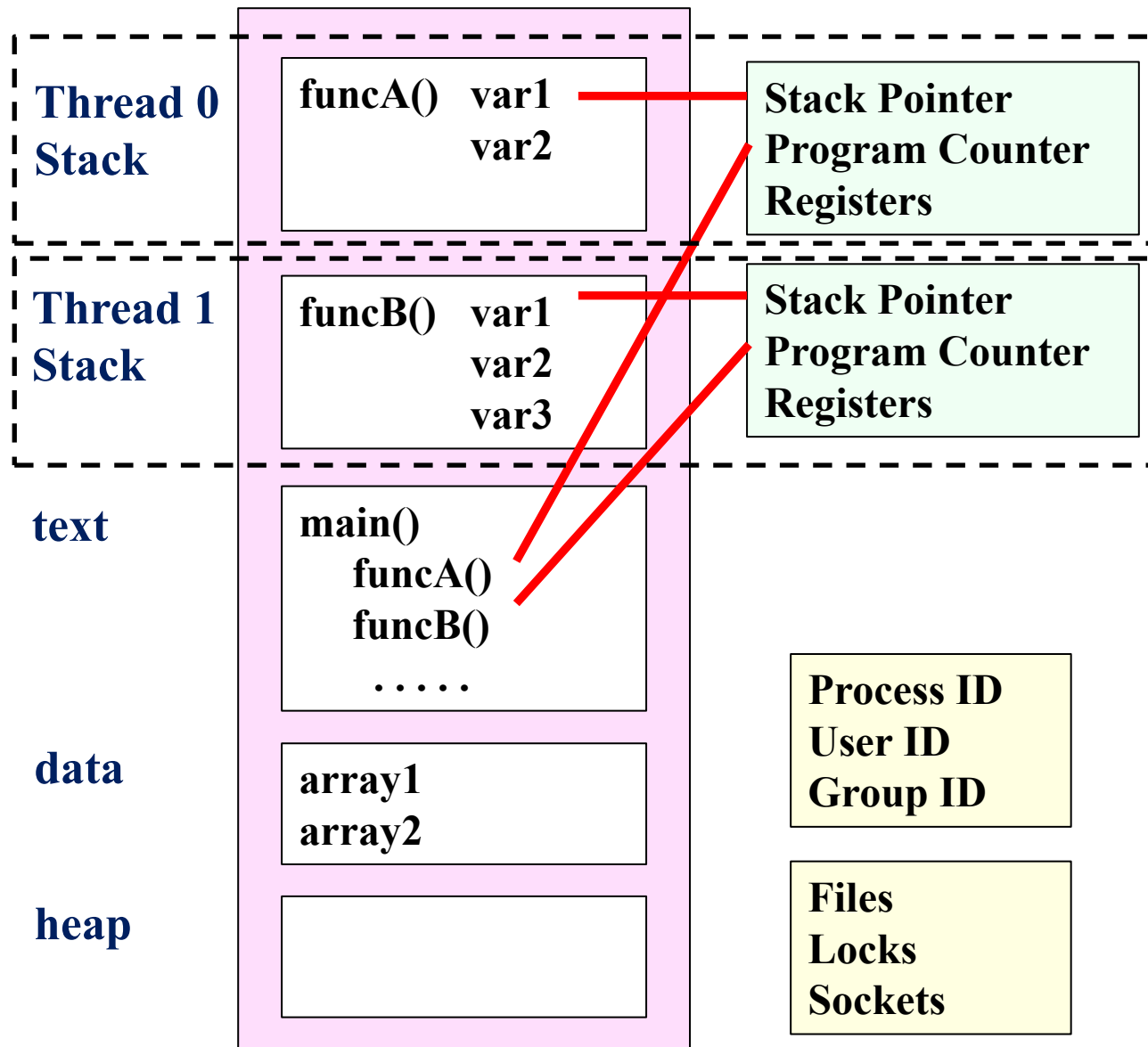
Process ID  
User ID  
Group ID

**heap**

Files  
Locks  
Sockets

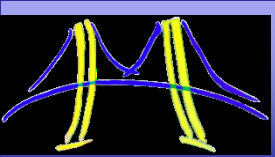


# Programming shared memory computers



## Threads:

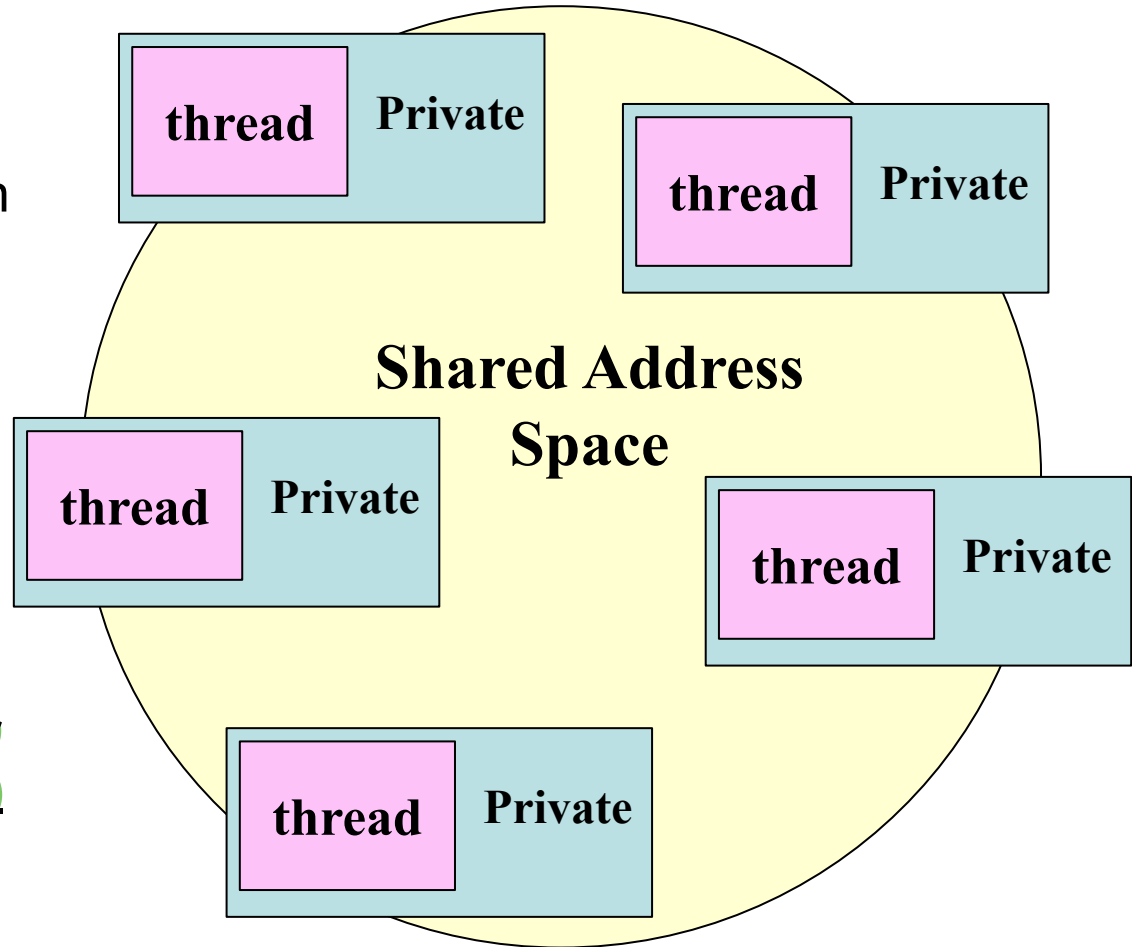
- Threads are "light weight processes"
- Threads share Process state among multiple threads ... this greatly reduces the cost of switching context.



# 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.**



# Exercise 1: Solution

## A multi-threaded “Hello world” program

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

```
#include "omp.h" ← OpenMP include file
```

```
int main()  
{
```

Parallel region with default  
number of threads

```
#pragma omp parallel  
{
```

```
    int ID = omp_get_thread_num();  
    printf(" hello(%d) ", ID);  
    printf(" world(%d) \n", ID);
```

```
    }  
}
```

End of the Parallel region

Runtime library function to  
return a thread ID.

### Sample Output:

```
hello(1) hello(0) world(1)  
world(0)  
hello (3) hello(2) world(3)  
world(2)
```

# 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.

# Outline

- **Unit 1: Getting started with OpenMP**
  - ◆ Mod1: Introduction to parallel programming
  - ◆ Mod 2: The boring bits: Using an OpenMP compiler (hello world)
  - ◆ Disc 1: Hello world and how threads work
- ● **Unit 2: The core features of OpenMP**
  - ◆ Mod 3: Creating Threads (the Pi program)
  - ◆ Disc 2: The simple Pi program and why it sucks
  - ◆ Mod 4: Synchronization (Pi program revisited)
  - ◆ Disc 3: Synchronization overhead and eliminating false sharing
  - ◆ Mod 5: Parallel Loops (making the Pi program simple)
  - ◆ Disc 4: Pi program wrap-up
- **Unit 3: Working with OpenMP**
  - ◆ Mod 6: Synchronize single masters and stuff
  - ◆ Mod 7: Data environment
  - ◆ Disc 5: Debugging OpenMP programs
  - ◆ Mod 8: Skills practice ... linked lists and OpenMP
  - ◆ Disc 6: Different ways to traverse linked lists
- **Unit 4: a few advanced OpenMP topics**
  - ◆ Mod 8: Tasks (linked lists the easy way)
  - ◆ Disc 7: Understanding Tasks
  - ◆ Mod 8: The scary stuff ... Memory model, atomics, and flush (pairwise synch).
  - ◆ Disc 8: The pitfalls of pairwise synchronization
  - ◆ Mod 9: Threadprivate Data and how to support libraries (Pi again)
  - ◆ Disc 9: Random number generators
- **Unit 5: Recapitulation**



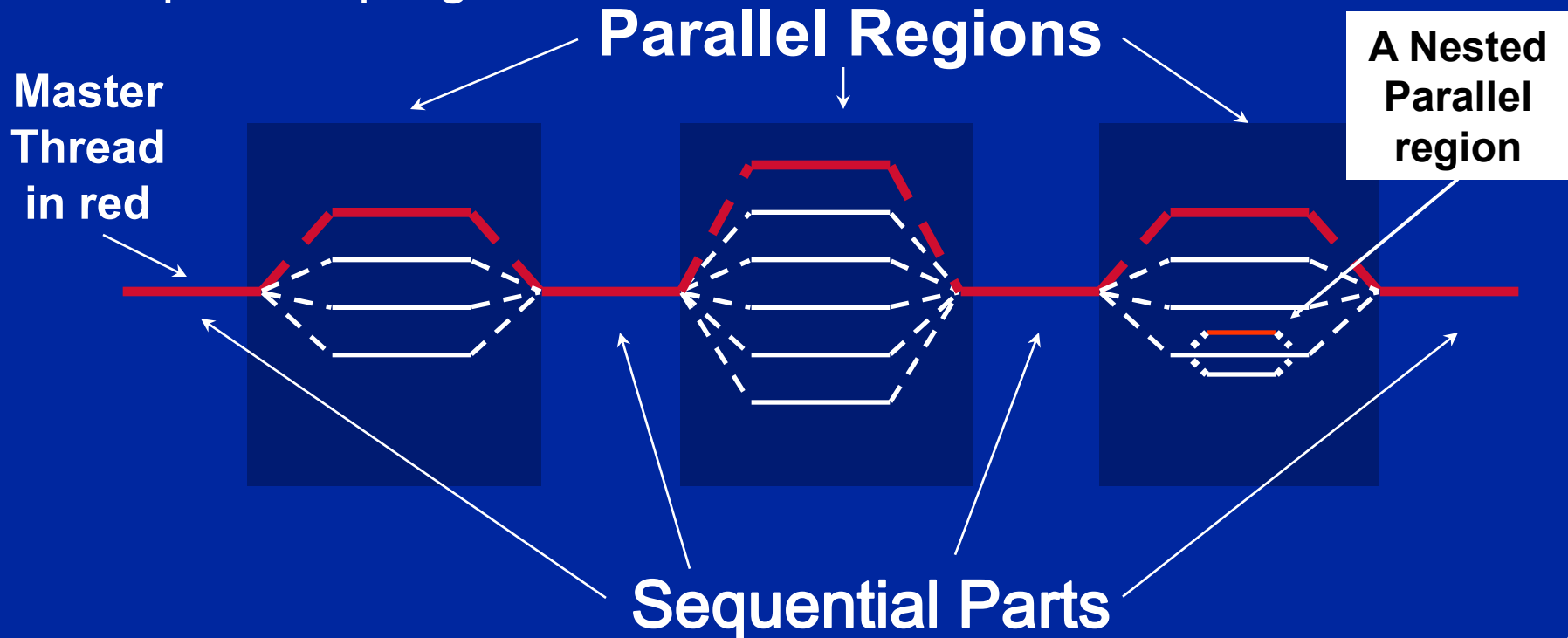
# Outline

- **Unit 1: Getting started with OpenMP**
  - ◆ Mod1: Introduction to parallel programming
  - ◆ Mod 2: The boring bits: Using an OpenMP compiler (hello world)
  - ◆ Disc 1: Hello world and how threads work
- **Unit 2: The core features of OpenMP**
  - ➡ ◆ Mod 3: Creating Threads (the Pi program)
  - ◆ Disc 2: The simple Pi program and why it sucks
  - ◆ Mod 4: Synchronization (Pi program revisited)
  - ◆ Disc 3: Synchronization overhead and eliminating false sharing
  - ◆ Mod 5: Parallel Loops (making the Pi program simple)
  - ◆ Disc 4: Pi program wrap-up
- **Unit 3: Working with OpenMP**
  - ◆ Mod 6: Synchronize single masters and stuff
  - ◆ Mod 7: Data environment
  - ◆ Disc 5: Debugging OpenMP programs
  - ◆ Mod 8: Skills practice ... linked lists and OpenMP
  - ◆ Disc 6: Different ways to traverse linked lists
- **Unit 4: a few advanced OpenMP topics**
  - ◆ Mod 8: Tasks (linked lists the easy way)
  - ◆ Disc 7: Understanding Tasks
  - ◆ Mod 8: The scary stuff ... Memory model, atomics, and flush (pairwise synch).
  - ◆ Disc 8: The pitfalls of pairwise synchronization
  - ◆ Mod 9: Threadprivate Data and how to support libraries (Pi again)
  - ◆ Disc 9: Random number generators

# 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

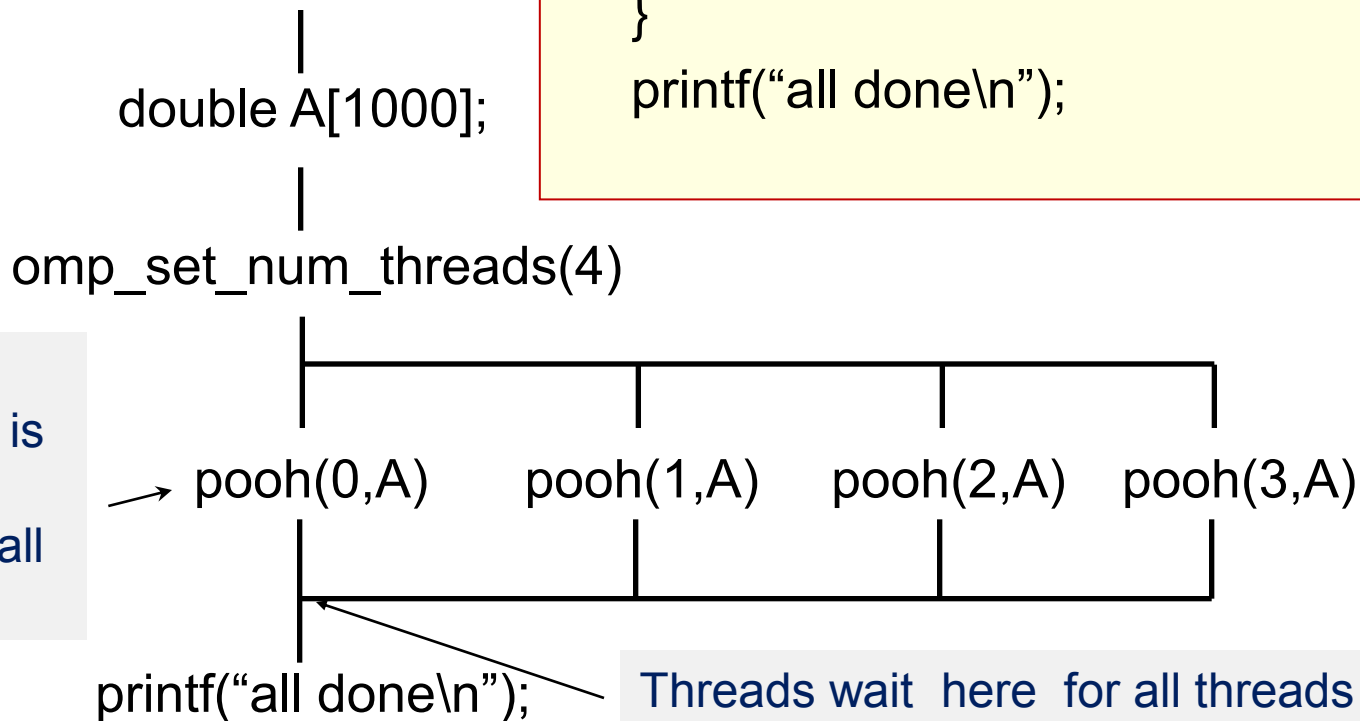
Runtime function returning a thread ID

- 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");
```



# 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

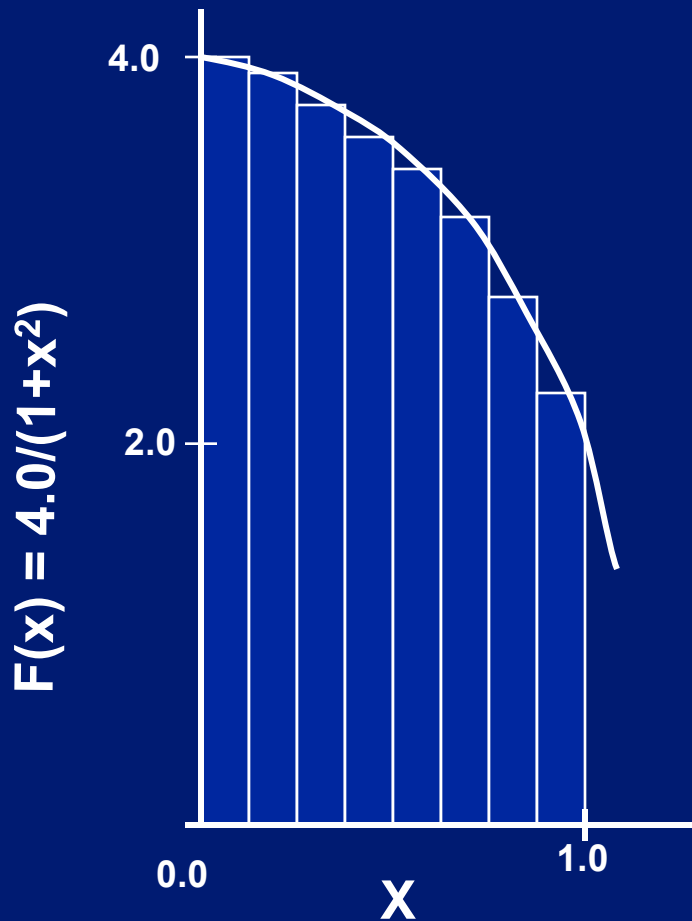
Mathematically, we know that:

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

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.
- 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

# Outline

- **Unit 1: Getting started with OpenMP**
  - ◆ Mod1: Introduction to parallel programming
  - ◆ Mod 2: The boring bits: Using an OpenMP compiler (hello world)
  - ◆ Disc 1: Hello world and how threads work
- **Unit 2: The core features of OpenMP**
  - ◆ Mod 3: Creating Threads (the Pi program)
  - ➡ ◆ Disc 2: The simple Pi program and why it sucks
  - ◆ Mod 4: Synchronization (Pi program revisited)
  - ◆ Disc 3: Synchronization overhead and eliminating false sharing
  - ◆ Mod 5: Parallel Loops (making the Pi program simple)
  - ◆ Disc 4: Pi program wrap-up
- **Unit 3: Working with OpenMP**
  - ◆ Mod 6: Synchronize single masters and stuff
  - ◆ Mod 7: Data environment
  - ◆ Disc 5: Debugging OpenMP programs
  - ◆ Mod 8: Skills practice ... linked lists and OpenMP
  - ◆ Disc 6: Different ways to traverse linked lists
- **Unit 4: a few advanced OpenMP topics**
  - ◆ Mod 8: Tasks (linked lists the easy way)
  - ◆ Disc 7: Understanding Tasks
  - ◆ Mod 8: The scary stuff ... Memory model, atomics, and flush (pairwise synch).
  - ◆ Disc 8: The pitfalls of pairwise synchronization
  - ◆ Mod 9: Threadprivate Data and how to support libraries (Pi again)
  - ◆ Disc 9: Random number generators
- **Unit 5: Recapitulation**

# 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

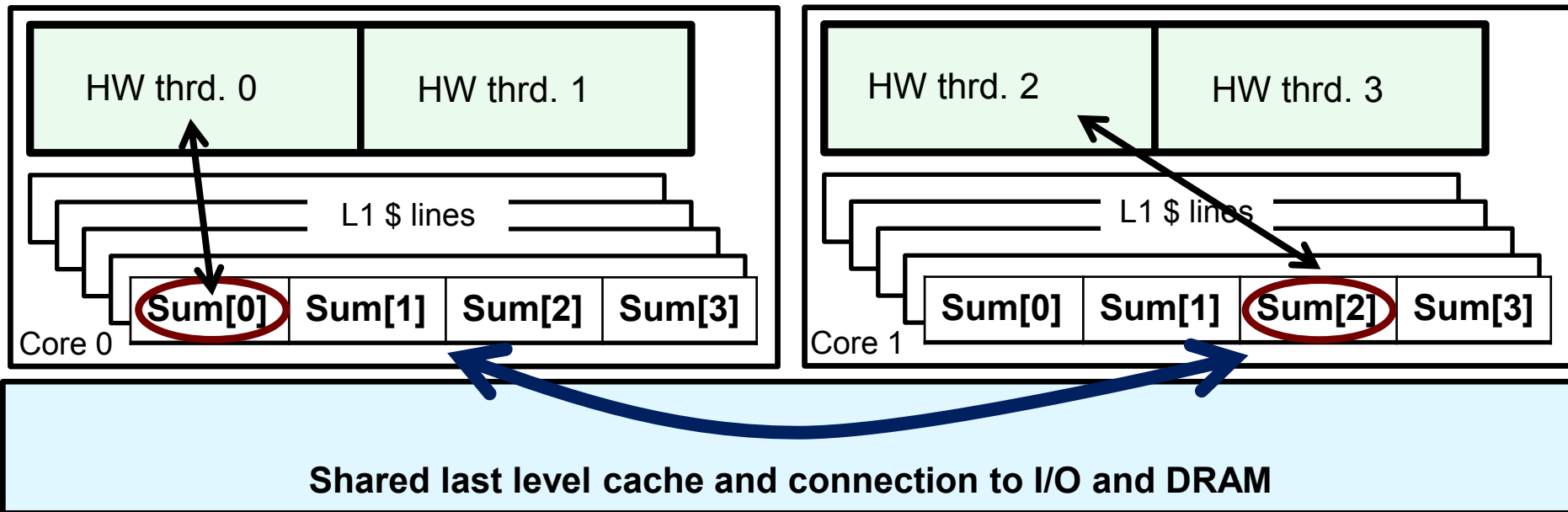
```
#include <omp.h>
static long num_steps = 100000;    double step;
#define NUM_THREADS 2
void main ()
{
    int i, nthreads; double pi, sum[NUM_THREADS];
    step = 1.0/(double) num_steps;
    omp_set_num_threads(NUM_THREADS);
    #pragma omp parallel
    {
        int i, id, nthrds;
        double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

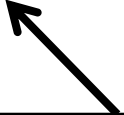


- If you promote scalars to an array to support creation of an SPMD program, the array elements are contiguous in memory and hence share cache lines ... Results in poor scalability.
- Solution: Pad arrays so elements you use are on distinct cache lines.

# 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.

# Outline

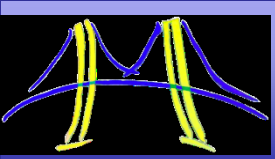
- **Unit 1: Getting started with OpenMP**
  - ◆ Mod1: Introduction to parallel programming
  - ◆ Mod 2: The boring bits: Using an OpenMP compiler (hello world)
  - ◆ Disc 1: Hello world and how threads work
- **Unit 2: The core features of OpenMP**
  - ◆ Mod 3: Creating Threads (the Pi program)
  - ◆ Disc 2: The simple Pi program and why it sucks
  -  ◆ Mod 4: Synchronization (Pi program revisited)
  - ◆ Disc 3: Synchronization overhead and eliminating false sharing
  - ◆ Mod 5: Parallel Loops (making the Pi program simple)
  - ◆ Disc 4: Pi program wrap-up
- **Unit 3: Working with OpenMP**
  - ◆ Mod 6: Synchronize single masters and stuff
  - ◆ Mod 7: Data environment
  - ◆ Disc 5: Debugging OpenMP programs
  - ◆ Mod 8: Skills practice ... linked lists and OpenMP
  - ◆ Disc 6: Different ways to traverse linked lists
- **Unit 4: a few advanced OpenMP topics**
  - ◆ Mod 8: Tasks (linked lists the easy way)
  - ◆ Disc 7: Understanding Tasks
  - ◆ Mod 8: The scary stuff ... Memory model, atomics, and flush (pairwise synch).
  - ◆ Disc 8: The pitfalls of pairwise synchronization
  - ◆ Mod 9: Threadprivate Data and how to support libraries (Pi again)
  - ◆ Disc 9: Random number generators
- **Unit 5: Recapitulation**

# OpenMP Overview:

## How do threads interact?

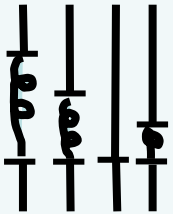
Recall our high level overview of OpenMP?

- 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.

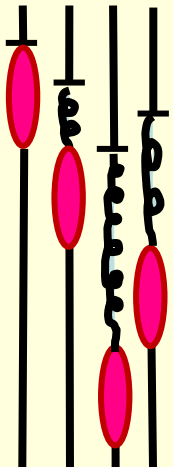


# Synchronization:

- 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

Discussed later

# 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.

```
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);  
    }  
}
```

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





# 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.

# Outline

- **Unit 1: Getting started with OpenMP**
  - ◆ Mod1: Introduction to parallel programming
  - ◆ Mod 2: The boring bits: Using an OpenMP compiler (hello world)
  - ◆ Disc 1: Hello world and how threads work
- **Unit 2: The core features of OpenMP**
  - ◆ Mod 3: Creating Threads (the Pi program)
  - ◆ Disc 2: The simple Pi program and why it sucks
  - ◆ Mod 4: Synchronization (Pi program revisited)
  - ➡ ◆ Disc 3: Synchronization overhead and eliminating false sharing
  - ◆ Mod 5: Parallel Loops (making the Pi program simple)
  - ◆ Disc 4: Pi program wrap-up
- **Unit 3: Working with OpenMP**
  - ◆ Mod 6: Synchronize single masters and stuff
  - ◆ Mod 7: Data environment
  - ◆ Disc 5: Debugging OpenMP programs
  - ◆ Mod 8: Skills practice ... linked lists and OpenMP
  - ◆ Disc 6: Different ways to traverse linked lists
- **Unit 4: a few advanced OpenMP topics**
  - ◆ Mod 8: Tasks (linked lists the easy way)
  - ◆ Disc 7: Understanding Tasks
  - ◆ Mod 8: The scary stuff ... Memory model, atomics, and flush (pairwise synch).
  - ◆ Disc 8: The pitfalls of pairwise synchronization
  - ◆ Mod 9: Threadprivate Data and how to support libraries (Pi again)
  - ◆ Disc 9: Random number generators
- **Unit 5: Recapitulation**

# 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) 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<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 <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+nthrds){
```

```
        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



# Outline

- **Unit 1: Getting started with OpenMP**
  - ◆ Mod1: Introduction to parallel programming
  - ◆ Mod 2: The boring bits: Using an OpenMP compiler (hello world)
  - ◆ Disc 1: Hello world and how threads work
- **Unit 2: The core features of OpenMP**
  - ◆ Mod 3: Creating Threads (the Pi program)
  - ◆ Disc 2: The simple Pi program and why it sucks
  - ◆ Mod 4: Synchronization (Pi program revisited)
  - ◆ Disc 3: Synchronization overhead and eliminating false sharing
  -  ◆ Mod 5: Parallel Loops (making the Pi program simple)
  - ◆ Disc 4: Pi program wrap-up
- **Unit 3: Working with OpenMP**
  - ◆ Mod 6: Synchronize single masters and stuff
  - ◆ Mod 7: Data environment
  - ◆ Disc 5: Debugging OpenMP programs
  - ◆ Mod 8: Skills practice ... linked lists and OpenMP
  - ◆ Disc 6: Different ways to traverse linked lists
- **Unit 4: a few advanced OpenMP topics**
  - ◆ Mod 8: Tasks (linked lists the easy way)
  - ◆ Disc 7: Understanding Tasks
  - ◆ Mod 8: The scary stuff ... Memory model, atomics, and flush (pairwise synch).
  - ◆ Disc 8: The pitfalls of pairwise synchronization
  - ◆ Mod 9: Threadprivate Data and how to support libraries (Pi again)
  - ◆ Disc 9: Random number generators
- **Unit 5: Recapitulation**

# 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

Discussed later

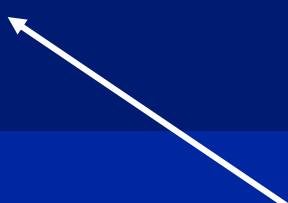
# The loop worksharing Constructs

- 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
<b>STATIC</b>	Pre-determined and predictable by the programmer
<b>DYNAMIC</b>	Unpredictable, highly variable work per iteration
<b>GUIDED</b>	Special case of dynamic to reduce scheduling overhead
<b>AUTO</b>	When the runtime can “learn” from previous executions of the same loop

Least work at runtime :  
scheduling done at compile-time



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



# Combined parallel/worksharing construct

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

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

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

These are equivalent



# Working with loops

- Basic approach

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

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

Note: loop index  
“i” is private by  
default

Remove loop  
carried  
dependence

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



# Nested loops

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

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

Number of  
loops to be  
parallelized,  
counting from  
the outside

- Will form a single loop of length  $N \times M$  and then parallelize that.
- Useful if  $N$  is  $O(\text{no. of threads})$  so parallelizing the outer loop makes balancing the load difficult.

# 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 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.

# Outline

- **Unit 1: Getting started with OpenMP**
  - ◆ Mod1: Introduction to parallel programming
  - ◆ Mod 2: The boring bits: Using an OpenMP compiler (hello world)
  - ◆ Disc 1: Hello world and how threads work
- **Unit 2: The core features of OpenMP**
  - ◆ Mod 3: Creating Threads (the Pi program)
  - ◆ Disc 2: The simple Pi program and why it sucks
  - ◆ Mod 4: Synchronization (Pi program revisited)
  - ◆ Disc 3: Synchronization overhead and eliminating false sharing
  - ◆ Mod 5: Parallel Loops (making the Pi program simple)
  - ➡ ◆ Disc 4: Pi program wrap-up
- **Unit 3: Working with OpenMP**
  - ◆ Mod 6: Synchronize single masters and stuff
  - ◆ Mod 7: Data environment
  - ◆ Disc 5: Debugging OpenMP programs
  - ◆ Mod 8: Skills practice ... linked lists and OpenMP
  - ◆ Disc 6: Different ways to traverse linked lists
- **Unit 4: a few advanced OpenMP topics**
  - ◆ Mod 8: Tasks (linked lists the easy way)
  - ◆ Disc 7: Understanding Tasks
  - ◆ Mod 8: The scary stuff ... Memory model, atomics, and flush (pairwise synch).
  - ◆ Disc 8: The pitfalls of pairwise synchronization
  - ◆ Mod 9: Threadprivate Data and how to support libraries (Pi again)
  - ◆ Disc 9: Random number generators
- **Unit 5: Recapitulation**

# 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.

# Parallel loops

- OpenMP 3.0 guarantees that this works ... i.e. that the same schedule is used in the two loops:


```
!$omp do schedule(static)
do i=1,n
    a(i) = ....
end do
!$omp end do nowait
!$omp do schedule(static)
do i=1,n
    .... = a(i)
end do
```

# Loops (cont.)

- Made **schedule(runtime)** more useful
  - can get/set it with library routines

```
omp_set_schedule()  
omp_get_schedule()
```
  - allow implementations to implement their own schedule kinds
- Added a new schedule kind **AUTO** which gives full freedom to the runtime to determine the scheduling of iterations to threads.
- Allowed C++ Random access iterators as loop control variables in parallel loops

# Outline

- **Unit 1: Getting started with OpenMP**
  - ◆ Mod1: Introduction to parallel programming
  - ◆ Mod 2: The boring bits: Using an OpenMP compiler (hello world)
  - ◆ Disc 1: Hello world and how threads work
- **Unit 2: The core features of OpenMP**
  - ◆ Mod 3: Creating Threads (the Pi program)
  - ◆ Disc 2: The simple Pi program and why it sucks
  - ◆ Mod 4: Synchronization (Pi program revisited)
  - ◆ Disc 3: Synchronization overhead and eliminating false sharing
  - ◆ Mod 5: Parallel Loops (making the Pi program simple)
  - ◆ Disc 4: Pi program wrap-up
-  ● **Unit 3: Working with OpenMP**
  - ◆ Mod 6: Synchronize single masters and stuff
  - ◆ Mod 7: Data environment
  - ◆ Disc 5: Debugging OpenMP programs
  - ◆ Mod 8: Skills practice ... linked lists and OpenMP
  - ◆ Disc 6: Different ways to traverse linked lists
- **Unit 4: a few advanced OpenMP topics**
  - ◆ Mod 8: Tasks (linked lists the easy way)
  - ◆ Disc 7: Understanding Tasks
  - ◆ Mod 8: The scary stuff ... Memory model, atomics, and flush (pairwise synch).
  - ◆ Disc 8: The pitfalls of pairwise synchronization
  - ◆ Mod 9: Threadprivate Data and how to support libraries (Pi again)
  - ◆ Disc 9: Random number generators
- **Unit 5: Recapitulation**

# Outline

- **Unit 1: Getting started with OpenMP**
  - ◆ Mod1: Introduction to parallel programming
  - ◆ Mod 2: The boring bits: Using an OpenMP compiler (hello world)
  - ◆ Disc 1: Hello world and how threads work
- **Unit 2: The core features of OpenMP**
  - ◆ Mod 3: Creating Threads (the Pi program)
  - ◆ Disc 2: The simple Pi program and why it sucks
  - ◆ Mod 4: Synchronization (Pi program revisited)
  - ◆ Disc 3: Synchronization overhead and eliminating false sharing
  - ◆ Mod 5: Parallel Loops (making the Pi program simple)
  - ◆ Disc 4: Pi program wrap-up
- **Unit 3: Working with OpenMP**
  - ➡ ◆ Mod 6: Synchronize single masters and stuff
  - ◆ Mod 7: Data environment
  - ◆ Disc 5: Debugging OpenMP programs
  - ◆ Mod 8: Skills practice ... linked lists and OpenMP
  - ◆ Disc 6: Different ways to traverse linked lists
- **Unit 4: a few advanced OpenMP topics**
  - ◆ Mod 8: Tasks (linked lists the easy way)
  - ◆ Disc 7: Understanding Tasks
  - ◆ Mod 8: The scary stuff ... Memory model, atomics, and flush (pairwise synch).
  - ◆ Disc 8: The pitfalls of pairwise synchronization
  - ◆ Mod 9: Threadprivate Data and how to support libraries (Pi again)
  - ◆ Disc 9: Random number generators
- **Unit 5: Recapitulation**

# Synchronization: Barrier

- **Barrier**: Each thread waits until all threads arrive.

```
#pragma omp parallel shared (A, B, C) private(id)
{
```

```
    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

# Master Construct

- The **master** construct denotes a structured block that is only executed by the master thread.
- The other threads just skip it (no synchronization is implied).

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

# 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();
}
```



# Sections worksharing Construct

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

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

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

# Synchronization: Lock routines

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

- **Simple Lock routines:**

- ◆ A simple lock is available if it is unset.

- `omp_init_lock()`, `omp_set_lock()`,  
`omp_unset_lock()`, `omp_test_lock()`,  
`omp_destroy_lock()`

- **Nested Locks**

- ◆ A nested lock is available if it is unset or if it is set but owned by the thread executing the nested lock function

- `omp_init_nest_lock()`, `omp_set_nest_lock()`,  
`omp_unset_nest_lock()`, `omp_test_nest_lock()`,  
`omp_destroy_nest_lock()`

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

# Synchronization: Simple Locks

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

```
#pragma omp parallel for
for(i=0;i<NBUCKETS; i++){
    omp_init_lock(&hist_locks[i]);    hist[i] = 0;
}
```

One lock per element of hist

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

Enforce mutual exclusion on update to hist array

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

Free-up storage when done.

# Runtime Library routines

- **Runtime environment routines:**
  - **Modify/Check the number of threads**
    - `omp_set_num_threads()`, `omp_get_num_threads()`,  
`omp_get_thread_num()`, `omp_get_max_threads()`
  - **Are we in an active parallel region?**
    - `omp_in_parallel()`
  - **Do you want the system to dynamically vary the number of threads from one parallel construct to another?**
    - `omp_set_dynamic`, `omp_get_dynamic()`;
  - **How many processors in the system?**
    - `omp_num_procs()`

...plus a few less commonly used routines.

# Runtime Library routines

- To use a known, fixed number of threads in a program, (1) tell the system that you don't want dynamic adjustment of the number of threads, (2) set the number of threads, then (3) save the number you got.

```
#include <omp.h>
void main()
{  int num_threads;
    omp_set_dynamic( 0 );
    omp_set_num_threads( omp_num_procs() );
#pragma omp parallel
    {  int id=omp_get_thread_num();
#pragma omp single
        num_threads = omp_get_num_threads();
        do_lots_of_stuff(id);
    }
}
```

Disable dynamic adjustment of the number of threads.

Request as many threads as you have processors.


Protect this op since Memory stores are not atomic

Even in this case, the system may give you fewer threads than requested. If the precise # of threads matters, test for it and respond accordingly.

# Environment Variables

- Set the default number of threads to use.
  - **OMP\_NUM\_THREADS** *int\_literal*
- OpenMP added an environment variable to control the size of child threads' stack
  - **OMP\_STACKSIZE**
- Also added an environment variable to hint to runtime how to treat idle threads
  - **OMP\_WAIT\_POLICY**
    - **ACTIVE** keep threads alive at barriers/locks spinning has cost
    - **PASSIVE** try to release processor at barriers/locks
- Process binding is enabled if this variable is true ... i.e. if true the runtime will not move threads around between processors.
  - **OMP\_PROC\_BIND** true | false

# Outline

- **Unit 1: Getting started with OpenMP**
  - ◆ Mod1: Introduction to parallel programming
  - ◆ Mod 2: The boring bits: Using an OpenMP compiler (hello world)
  - ◆ Disc 1: Hello world and how threads work
- **Unit 2: The core features of OpenMP**
  - ◆ Mod 3: Creating Threads (the Pi program)
  - ◆ Disc 2: The simple Pi program and why it sucks
  - ◆ Mod 4: Synchronization (Pi program revisited)
  - ◆ Disc 3: Synchronization overhead and eliminating false sharing
  - ◆ Mod 5: Parallel Loops (making the Pi program simple)
  - ◆ Disc 4: Pi program wrap-up
- **Unit 3: Working with OpenMP**
  - ◆ Mod 6: Synchronize single masters and stuff
  -  ◆ Mod 7: Data environment
  - ◆ Disc 5: Debugging OpenMP programs
  - ◆ Mod 8: Skills practice ... linked lists and OpenMP
  - ◆ Disc 6: Different ways to traverse linked lists
- **Unit 4: a few advanced OpenMP topics**
  - ◆ Mod 8: Tasks (linked lists the easy way)
  - ◆ Disc 7: Understanding Tasks
  - ◆ Mod 8: The scary stuff ... Memory model, atomics, and flush (pairwise synch).
  - ◆ Disc 8: The pitfalls of pairwise synchronization
  - ◆ Mod 9: Threadprivate Data and how to support libraries (Pi again)
  - ◆ Disc 9: Random number generators
- **Unit 5: Recapitulation**

# 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.



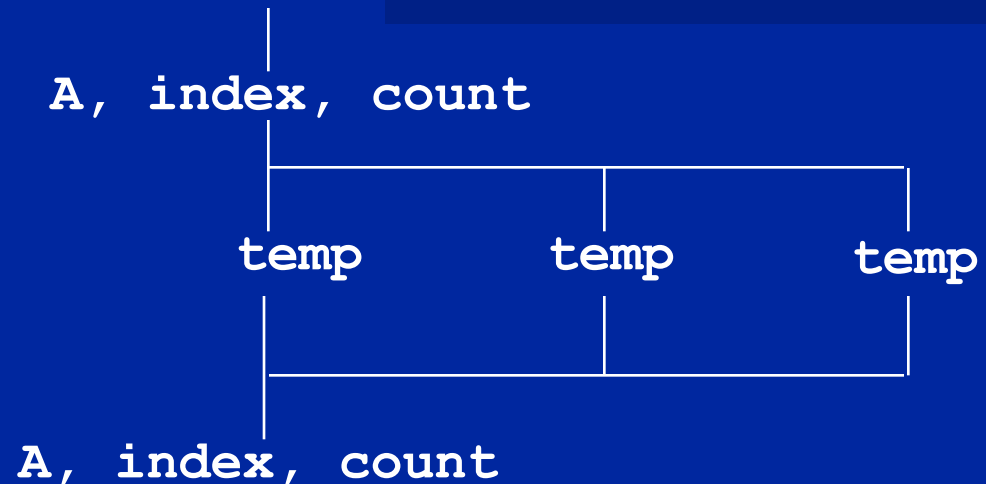
# Data sharing: Examples

```
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)**  
DEFAULT(PRIVATE) *is Fortran only*

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

\*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



# 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

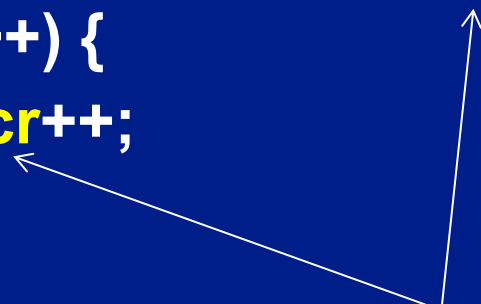
```
extern int tmp;  
void work() {  
    tmp = 5;  
}
```

unspecified which copy of tmp

# Firstprivate Clause

- Variables initialized from 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

# Lastprivate Clause

- Variables update shared variable using value from last iteration
- C++ objects are updated as if by assignment

```
void sq2(int n, double *lastterm)
{
    double x; int i;
    #pragma omp parallel for lastprivate(x)
    for (i = 0; i < n; i++){
        x = a[i]*a[i] + b[i]*b[i];
        b[i] = sqrt(x);
    }
    *lastterm = x;
}
```

“x” has the value it held for the “last sequential” iteration (i.e., for  $i=(n-1)$ )

# 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 local 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 local 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

- Note that the default storage attribute is **DEFAULT(SHARED)** (so no need to use it)
  - ◆ Exception: **#pragma omp task**
- To change default: **DEFAULT(PRIVATE)**
  - ◆ *each* variable in the construct is made private as if specified in a private clause
  - ◆ mostly saves typing
- **DEFAULT(NONE)**: *no* default for variables in static extent. Must list storage attribute for each variable in static extent. Good programming practice!

Only the Fortran API supports default(private).

C/C++ only has default(shared) or default(none).



# Data Sharing: Default Clause Example

```
    itotal = 1000
C$OMP PARALLEL PRIVATE(np, each)
    np = omp_get_num_threads()
    each = itotal/np
    .....
C$OMP END PARALLEL
```

```
    itotal = 1000
C$OMP PARALLEL DEFAULT(PRIVATE) SHARED(itotal)
    np = omp_get_num_threads()
    each = itotal/np
    .....
C$OMP END PARALLEL
```

**These two  
code  
fragments are  
equivalent**

# Exercise 5: 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).

# Exercise 5 (cont.)

- 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.

# Outline

- **Unit 1: Getting started with OpenMP**
  - ◆ Mod1: Introduction to parallel programming
  - ◆ Mod 2: The boring bits: Using an OpenMP compiler (hello world)
  - ◆ Disc 1: Hello world and how threads work
- **Unit 2: The core features of OpenMP**
  - ◆ Mod 3: Creating Threads (the Pi program)
  - ◆ Disc 2: The simple Pi program and why it sucks
  - ◆ Mod 4: Synchronization (Pi program revisited)
  - ◆ Disc 3: Synchronization overhead and eliminating false sharing
  - ◆ Mod 5: Parallel Loops (making the Pi program simple)
  - ◆ Disc 4: Pi program wrap-up
- **Unit 3: Working with OpenMP**
  - ◆ Mod 6: Synchronize single masters and stuff
  - ◆ Mod 7: Data environment
  - ➡ ◆ Disc 5: Debugging OpenMP programs
  - ◆ Mod 8: Skills practice ... linked lists and OpenMP
  - ◆ Disc 6: Different ways to traverse linked lists
- **Unit 4: a few advanced OpenMP topics**
  - ◆ Mod 8: Tasks (linked lists the easy way)
  - ◆ Disc 7: Understanding Tasks
  - ◆ Mod 8: The scary stuff ... Memory model, atomics, and flush (pairwise synch).
  - ◆ Disc 8: The pitfalls of pairwise synchronization
  - ◆ Mod 9: Threadprivate Data and how to support libraries (Pi again)
  - ◆ Disc 9: Random number generators
- **Unit 5: Recapitulation**

# The Mandelbrot Area program

```
#include <omp.h>
# define NPOINTS 1000
# define MXITR 1000
void testpoint(void);
struct d_complex{
    double r;    double i;
};
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,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();
        }
    }
    area=2.0*2.5*1.125*(double)(NPOINTS*NPOINTS-
numoutside)/(double)(NPOINTS*NPOINTS);
    error=area/(double)NPOINTS;
}
```

```
void testpoint(void){
    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) {
            numoutside++;
            break;
        }
    }
}
```

**When I run this program, I get a different incorrect answer each time I run it ... there is a race condition!!!!**

# Debugging parallel programs

- Find tools that work with your environment and learn to use them. A good parallel debugger can make a huge difference.
- But parallel debuggers are not portable and you will assuredly need to debug “by hand” at some point.
- There are tricks to help you. The most important is to use the `default(none)` pragma

```
#pragma omp parallel for default(none) private(c, 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();
    }
}
```

Using `default(none)` generates a compiler error that `j` is unspecified.

# 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) V
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 atomic
            numoutside++;
            break;
        }
    }
}
```

Other errors found using a debugger or by inspection:

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

# Serial PI Program

Now that you understand how to modify the data environment, let's take one last look at our 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;
}
```

What is the minimum change I can make to this code to parallelize it?



# 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!

# Outline

- **Unit 1: Getting started with OpenMP**
  - ◆ Mod1: Introduction to parallel programming
  - ◆ Mod 2: The boring bits: Using an OpenMP compiler (hello world)
  - ◆ Disc 1: Hello world and how threads work
- **Unit 2: The core features of OpenMP**
  - ◆ Mod 3: Creating Threads (the Pi program)
  - ◆ Disc 2: The simple Pi program and why it sucks
  - ◆ Mod 4: Synchronization (Pi program revisited)
  - ◆ Disc 3: Synchronization overhead and eliminating false sharing
  - ◆ Mod 5: Parallel Loops (making the Pi program simple)
  - ◆ Disc 4: Pi program wrap-up
- **Unit 3: Working with OpenMP**
  - ◆ Mod 6: Synchronize single masters and stuff
  - ◆ Mod 7: Data environment
  - ◆ Disc 5: Debugging OpenMP programs
  -  ◆ Mod 8: Skills practice ... linked lists and OpenMP
  - ◆ Disc 6: Different ways to traverse linked lists
- **Unit 4: a few advanced OpenMP topics**
  - ◆ Mod 8: Tasks (linked lists the easy way)
  - ◆ Disc 7: Understanding Tasks
  - ◆ Mod 8: The scary stuff ... Memory model, atomics, and flush (pairwise synch).
  - ◆ Disc 8: The pitfalls of pairwise synchronization
  - ◆ Mod 9: Threadprivate Data and how to support libraries (Pi again)
  - ◆ Disc 9: Random number generators
- **Unit 5: Recapitulation**

# Major OpenMP constructs we've covered so far

- To create a team of threads
  - ◆ `#pragma omp parallel`
- To share work between threads:
  - ◆ `#pragma omp for`
  - ◆ `#pragma omp single`
- To prevent conflicts (prevent races)
  - ◆ `#pragma omp critical`
  - ◆ `#pragma omp atomic`
  - ◆ `#pragma omp barrier`
  - ◆ `#pragma omp master`
- Data environment clauses
  - ◆ `private (variable_list)`
  - ◆ `firstprivate (variable_list)`
  - ◆ `lastprivate (variable_list)`
  - ◆ `reduction(+:variable_list)`

Where `variable_list` is a comma separated list of variables

Print the value of the macro

`_OPENMP`

And its value will be

`yyymm`

For the year and month of the spec the implementation used

# 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.

# Exercise 6: linked lists the hard way

- Consider the program `linked.c`
  - ◆ Traverses a linked list computing a sequence of Fibonacci numbers at each node.
- Parallelize this program using constructs described so far (i.e. even if you already know about them, don't use tasks).
- Once you have a correct program, optimize it.

# Outline

- **Unit 1: Getting started with OpenMP**
  - ◆ Mod1: Introduction to parallel programming
  - ◆ Mod 2: The boring bits: Using an OpenMP compiler (hello world)
  - ◆ Disc 1: Hello world and how threads work
- **Unit 2: The core features of OpenMP**
  - ◆ Mod 3: Creating Threads (the Pi program)
  - ◆ Disc 2: The simple Pi program and why it sucks
  - ◆ Mod 4: Synchronization (Pi program revisited)
  - ◆ Disc 3: Synchronization overhead and eliminating false sharing
  - ◆ Mod 5: Parallel Loops (making the Pi program simple)
  - ◆ Disc 4: Pi program wrap-up
- **Unit 3: Working with OpenMP**
  - ◆ Mod 6: Synchronize single masters and stuff
  - ◆ Mod 7: Data environment
  - ◆ Disc 5: Debugging OpenMP programs
  - ◆ Mod 8: Skills practice ... linked lists and OpenMP
  - ➡ ◆ Disc 6: Different ways to traverse linked lists
- **Unit 4: a few advanced OpenMP topics**
  - ◆ Mod 8: Tasks (linked lists the easy way)
  - ◆ Disc 7: Understanding Tasks
  - ◆ Mod 8: The scary stuff ... Memory model, atomics, and flush (pairwise synch).
  - ◆ Disc 8: The pitfalls of pairwise synchronization
  - ◆ Mod 9: Threadprivate Data and how to support libraries (Pi again)
  - ◆ Disc 9: Random number generators
- **Unit 5: Recapitulation**

# list traversal

- When we first created OpenMP, we focused on common use cases in HPC ... Fortran arrays processed over “regular” loops.
- Recursion and “pointer chasing” were so far removed from our Fortran focus that we didn’t even consider more general structures.
- Hence, even a simple list traversal is exceedingly difficult with the original versions of OpenMP.

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

# Linked lists without tasks

- See the file `Linked_omp25.c`

```
while (p != NULL) {
```

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

```
    count++;
```

```
}
```

```
p = head;
```

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

```
    parr[i] = p;
```

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

```
}
```

```
#pragma omp parallel
```

```
{
```

```
    #pragma omp for schedule(static,1)
```

```
    for(i=0; i<count; i++)
```

```
        processwork(parr[i]);
```

```
}
```

Count number of items in the linked list

Copy pointer to each node into an array

Process nodes in parallel with a for loop

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



# Linked lists without tasks: C++ STL

- See the file `Linked_cpp.cpp`

```
std::vector<node *> nodelist;
```

```
for (p = head; p != NULL; p = p->next)
```

```
    nodelist.push_back(p);
```

Copy pointer to each node into an array

```
int j = (int)nodelist.size();
```

Count number of items in the linked list

```
#pragma omp parallel for schedule(static,1)
```

```
    for (int i = 0; i < j; ++i)
```

```
        processwork(nodelist[i]);
```


Process nodes in parallel with a for loop

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

# Conclusion

- We were able to parallelize the linked list traversal ... but it was ugly and required multiple passes over the data.
- To move beyond its roots in the array based world of scientific computing, we needed to support more general data structures and loops beyond basic for loops.
- To do this, we added tasks in OpenMP 3.0

# Outline

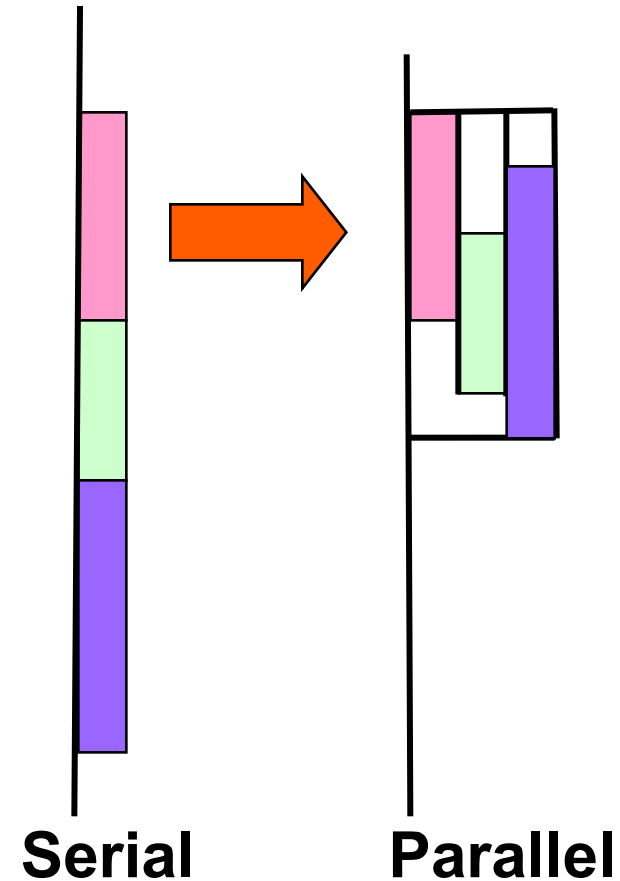
- **Unit 1: Getting started with OpenMP**
  - ◆ Mod1: Introduction to parallel programming
  - ◆ Mod 2: The boring bits: Using an OpenMP compiler (hello world)
  - ◆ Disc 1: Hello world and how threads work
- **Unit 2: The core features of OpenMP**
  - ◆ Mod 3: Creating Threads (the Pi program)
  - ◆ Disc 2: The simple Pi program and why it sucks
  - ◆ Mod 4: Synchronization (Pi program revisited)
  - ◆ Disc 3: Synchronization overhead and eliminating false sharing
  - ◆ Mod 5: Parallel Loops (making the Pi program simple)
  - ◆ Disc 4: Pi program wrap-up
- **Unit 3: Working with OpenMP**
  - ◆ Mod 6: Synchronize single masters and stuff
  - ◆ Mod 7: Data environment
  - ◆ Disc 5: Debugging OpenMP programs
  - ◆ Mod 8: Skills practice ... linked lists and OpenMP
  - ◆ Disc 6: Different ways to traverse linked lists
-  ● **Unit 4: a few advanced OpenMP topics**
  - ◆ Mod 8: Tasks (linked lists the easy way)
  - ◆ Disc 7: Understanding Tasks
  - ◆ Mod 8: The scary stuff ... Memory model, atomics, and flush (pairwise synch).
  - ◆ Disc 8: The pitfalls of pairwise synchronization
  - ◆ Mod 9: Threadprivate Data and how to support libraries (Pi again)
  - ◆ Disc 9: Random number generators
- **Unit 5: Recapitulation**

# Outline

- **Unit 1: Getting started with OpenMP**
  - ◆ Mod1: Introduction to parallel programming
  - ◆ Mod 2: The boring bits: Using an OpenMP compiler (hello world)
  - ◆ Disc 1: Hello world and how threads work
- **Unit 2: The core features of OpenMP**
  - ◆ Mod 3: Creating Threads (the Pi program)
  - ◆ Disc 2: The simple Pi program and why it sucks
  - ◆ Mod 4: Synchronization (Pi program revisited)
  - ◆ Disc 3: Synchronization overhead and eliminating false sharing
  - ◆ Mod 5: Parallel Loops (making the Pi program simple)
  - ◆ Disc 4: Pi program wrap-up
- **Unit 3: Working with OpenMP**
  - ◆ Mod 6: Synchronize single masters and stuff
  - ◆ Mod 7: Data environment
  - ◆ Disc 5: Debugging OpenMP programs
  - ◆ Mod 8: Skills practice ... linked lists and OpenMP
  - ◆ Disc 6: Different ways to traverse linked lists
- **Unit 4: a few advanced OpenMP topics**
  - ➡ ◆ Mod 8: Tasks (linked lists the easy way)
  - ◆ Disc 7: Understanding Tasks
  - ◆ Mod 8: The scary stuff ... Memory model, atomics, and flush (pairwise synch).
  - ◆ Disc 8: The pitfalls of pairwise synchronization
  - ◆ Mod 9: Threadprivate Data and how to support libraries (Pi again)
  - ◆ Disc 9: Random number generators
- **Unit 5: Recapitulation**

# 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



# Definitions

- ***Task construct*** – task directive plus structured block
- ***Task*** – the package of code and instructions for allocating data created when a thread encounters a task construct
- ***Task region*** – the dynamic sequence of instructions produced by the execution of a task by a thread

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



**Good solution** – e is  
firstprivate

# Exercise 7: tasks in OpenMP

- Consider the program `linked.c`
  - ◆ Traverses a linked list computing a sequence of Fibonacci numbers at each node.
- Parallelize this program using tasks.
- Compare your solution's complexity to an approach without tasks.

# Outline

- **Unit 1: Getting started with OpenMP**
  - ◆ Mod1: Introduction to parallel programming
  - ◆ Mod 2: The boring bits: Using an OpenMP compiler (hello world)
  - ◆ Disc 1: Hello world and how threads work
- **Unit 2: The core features of OpenMP**
  - ◆ Mod 3: Creating Threads (the Pi program)
  - ◆ Disc 2: The simple Pi program and why it sucks
  - ◆ Mod 4: Synchronization (Pi program revisited)
  - ◆ Disc 3: Synchronization overhead and eliminating false sharing
  - ◆ Mod 5: Parallel Loops (making the Pi program simple)
  - ◆ Disc 4: Pi program wrap-up
- **Unit 3: Working with OpenMP**
  - ◆ Mod 6: Synchronize single masters and stuff
  - ◆ Mod 7: Data environment
  - ◆ Disc 5: Debugging OpenMP programs
  - ◆ Mod 8: Skills practice ... linked lists and OpenMP
  - ◆ Disc 6: Different ways to traverse linked lists
- **Unit 4: a few advanced OpenMP topics**
  - ◆ Mod 8: Tasks (linked lists the easy way)
  - ➡ ◆ Disc 7: Understanding Tasks
  - ◆ Mod 8: The scary stuff ... Memory model, atomics, and flush (pairwise synch).
  - ◆ Disc 8: The pitfalls of pairwise synchronization
  - ◆ Mod 9: Threadprivate Data and how to support libraries (Pi again)
  - ◆ Disc 9: Random number generators
- **Unit 5: Recapitulation**

# Exercise 7: tasks in OpenMP

- Consider the program `linked.c`
  - ◆ Traverses a linked list computing a sequence of Fibonacci numbers at each node.
- Parallelize this program using tasks.
- Compare your solution's complexity to an approach without tasks.

# Task Construct – Explicit Tasks

```
#pragma omp parallel
```

```
{
```

```
#pragma omp single
```

```
{
```

```
node * p = head;
```

```
while (p) {
```

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

```
process(p);
```

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

```
}
```

```
}
```

```
}
```

1. Create a team of threads.

2. One thread executes the **single** construct

... other threads wait at the implied barrier at the end of the single construct

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

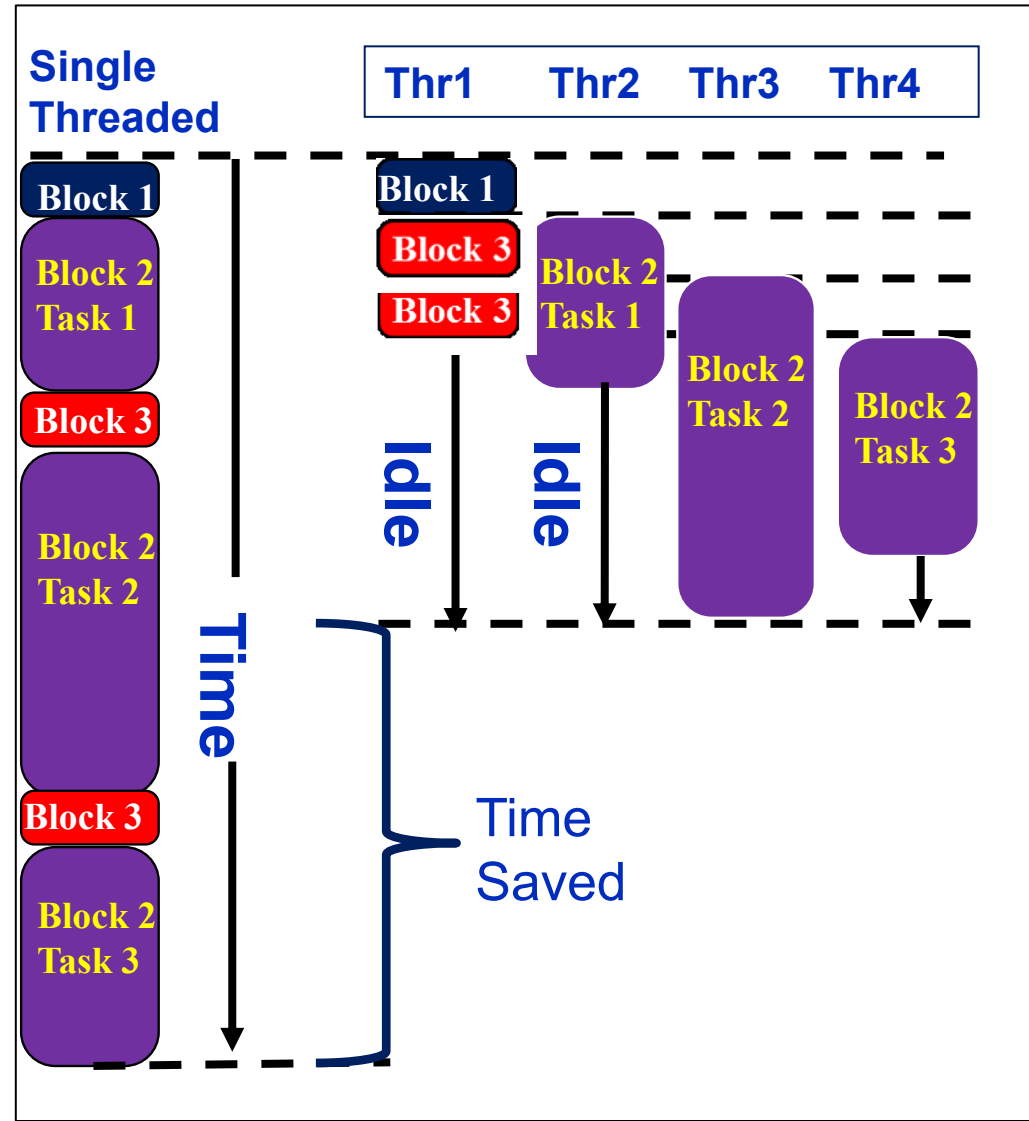
4. Threads waiting at the barrier execute tasks.

Execution moves beyond the barrier once all the tasks are complete

# Execution of tasks

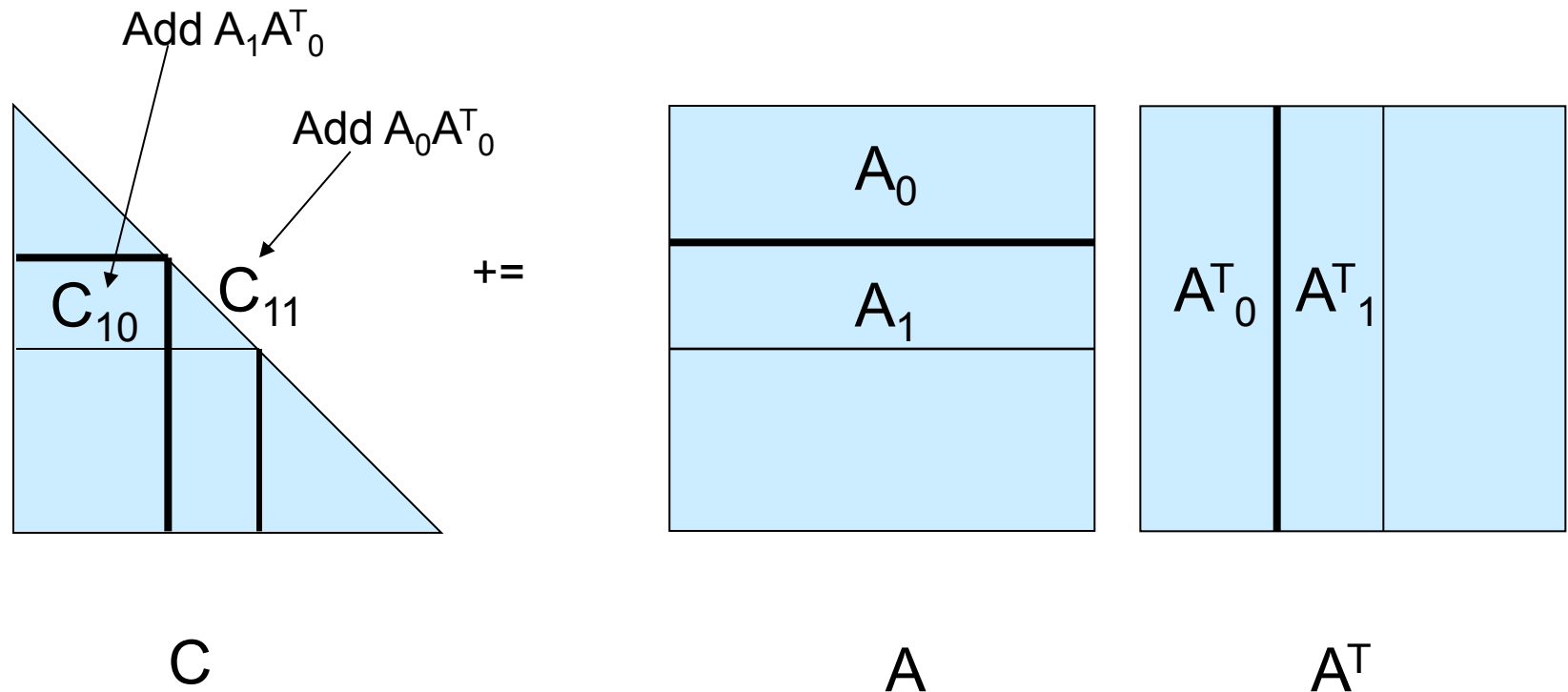
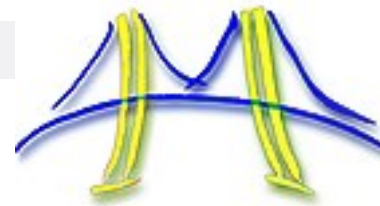
Have potential to parallelize irregular patterns and recursive function calls

```
#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
        }
    }
}
```

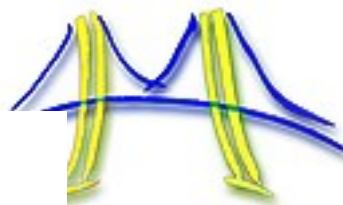




# A real example: Symmetric rank-k update



Note: the iteration sweeps through  $C$  and  $A$ , creating a new block of rows to be updated with new parts of  $A$ . These updates are completely independent.



```

while ( FLA_Obj_length( CTL ) < FLA_Obj_length( C ) ){
    b = min( FLA_Obj_length( CBR ), nb_alg );

    FLA_Repart_2x2_to_3x3( CTL, /**/ CTR,    &C00, /**/ &C01, &C02,
                          /***/ /***/
                          &C10, /**/ &C11, &C12,
                          CBL, /**/ CBR,    &C20, /**/ &C21, &C22,
                          b, b, FLA_BR );
    FLA_Repart_2x1_to_3x1( AT,              &A0,
                          /* ** */          /* ** */
                          &A1,
                          AB,              &A2,    b, FLA_BOTTOM );
    /*-----*/

    FLA_Gemm( FLA_NO_TRANSPOSE, FLA_TRANSPOSE, ONE, A0, A1, ONE, C10 );
    FLA_Syrk( FLA_LOWER_TRIANGULAR, FLA_NO_TRANSPOSE, ONE, A1, ONE, C11 );

    /*-----*/
    FLA_Cont_with_3x3_to_2x2( &CTL, /**/ &CTR,  C00, C01, /**/ C02,
                              C10, C11, /**/ C12,
                              /***/ /***/
                              &CBL, /**/ &CBR,  C20, C21, /**/ C22,
                              FLA_TL );
    FLA_Cont_with_3x1_to_2x1( &AT,              A0,
                              A1,
                              /* ** */          /* ** */
                              &AB,              A2,    FLA_TOP );
}

```

**#pragma omp parallel**

**{**

**#pragma omp single**

**{**

```
while ( FLA_Obj_length( CTL ) < FLA_Obj_length( C ) ){  
    b = min( FLA_Obj_length( CBR ), nb_alg );
```

```
    FLA_Repart_2x2_to_3x3( CTL, /**/ CTR,    &C00, /**/ &C01, &C02,  
                           /*****/          /*****/  
                           &C10, /**/ &C11, &C12,  
                           CBL, /**/ CBR,    &C20, /**/ &C21, &C22,  
                           b, b, FLA_BR );
```

```
    FLA_Repart_2x1_to_3x1( AT,                &A0,  
                          /* ** */          /* ** */  
                          &A1,  
                          AB,                &A2,    b, FLA_BOTTOM );
```

```
    /*-----*/
```

**#pragma omp task firstprivate(A0, A1, C10, C11)**

**{**

```
    FLA_Gemm( FLA_NO_TRANSPOSE, FLA_TRANSPOSE, ONE, A0, A1, ONE, C10 );
```

```
    FLA_Syrk( FLA_LOWER_TRIANGULAR, FLA_NO_TRANSPOSE, ONE, A1, ONE, C11 );
```

```
} /* end task */
```

```
    /*-----*/
```

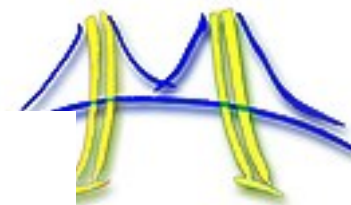
```
    FLA_Cont_with_3x3_to_2x2( &CTL, /**/ &CTR,  C00, C01, /**/ C02,  
                              C10, C11, /**/ C12,  
                              /*****/          /*****/  
                              &CBL, /**/ &CBR,  C20, C21, /**/ C22,  
                              FLA_TL );
```

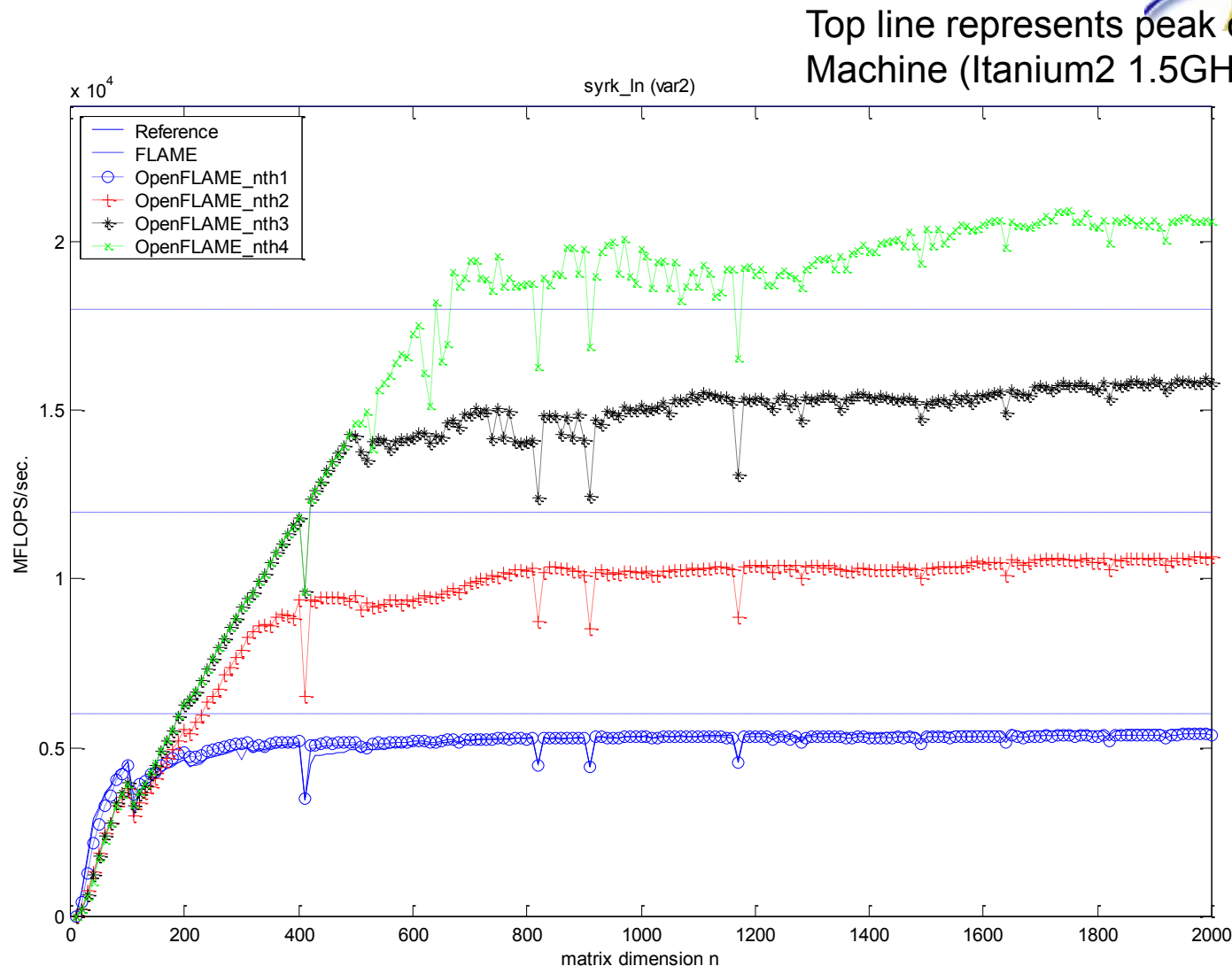
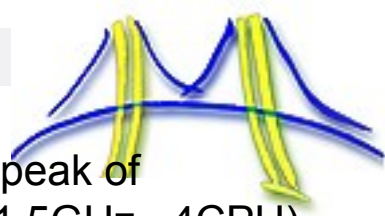
```
    FLA_Cont_with_3x1_to_2x1( &AT,                A0,  
                              A1,  
                              /* ** */          /* ** */  
                              &AB,                A2,    FLA_TOP );
```

```
}
```

**} // end of task-queue**


**} // end of parallel region**





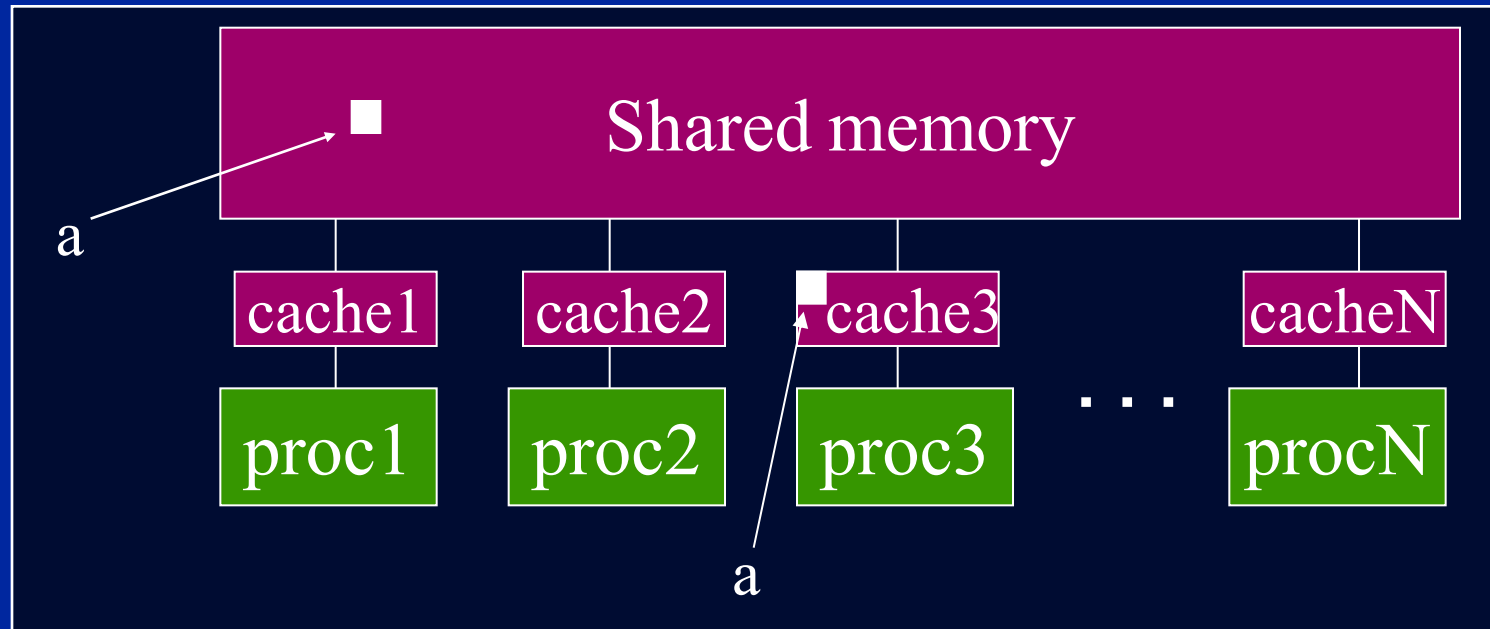
Note: the above graphs is for the most naïve way of marching through the matrices. By picking blocks dynamically, much faster ramp-up can be achieved.

# Outline

- **Unit 1: Getting started with OpenMP**
  - ◆ Mod1: Introduction to parallel programming
  - ◆ Mod 2: The boring bits: Using an OpenMP compiler (hello world)
  - ◆ Disc 1: Hello world and how threads work
- **Unit 2: The core features of OpenMP**
  - ◆ Mod 3: Creating Threads (the Pi program)
  - ◆ Disc 2: The simple Pi program and why it sucks
  - ◆ Mod 4: Synchronization (Pi program revisited)
  - ◆ Disc 3: Synchronization overhead and eliminating false sharing
  - ◆ Mod 5: Parallel Loops (making the Pi program simple)
  - ◆ Disc 4: Pi program wrap-up
- **Unit 3: Working with OpenMP**
  - ◆ Mod 6: Synchronize single masters and stuff
  - ◆ Mod 7: Data environment
  - ◆ Disc 5: Debugging OpenMP programs
  - ◆ Mod 8: Skills practice ... linked lists and OpenMP
  - ◆ Disc 6: Different ways to traverse linked lists
- **Unit 4: a few advanced OpenMP topics**
  - ◆ Mod 8: Tasks (linked lists the easy way)
  - ◆ Disc 7: Understanding Tasks
  -  ◆ Mod 8: The scary stuff ... Memory model, atomics, and flush (pairwise synch).
  - ◆ Disc 8: The pitfalls of pairwise synchronization
  - ◆ Mod 9: Threadprivate Data and how to support libraries (Pi again)
  - ◆ Disc 9: Random number generators
- **Unit 5: Recapitulation**

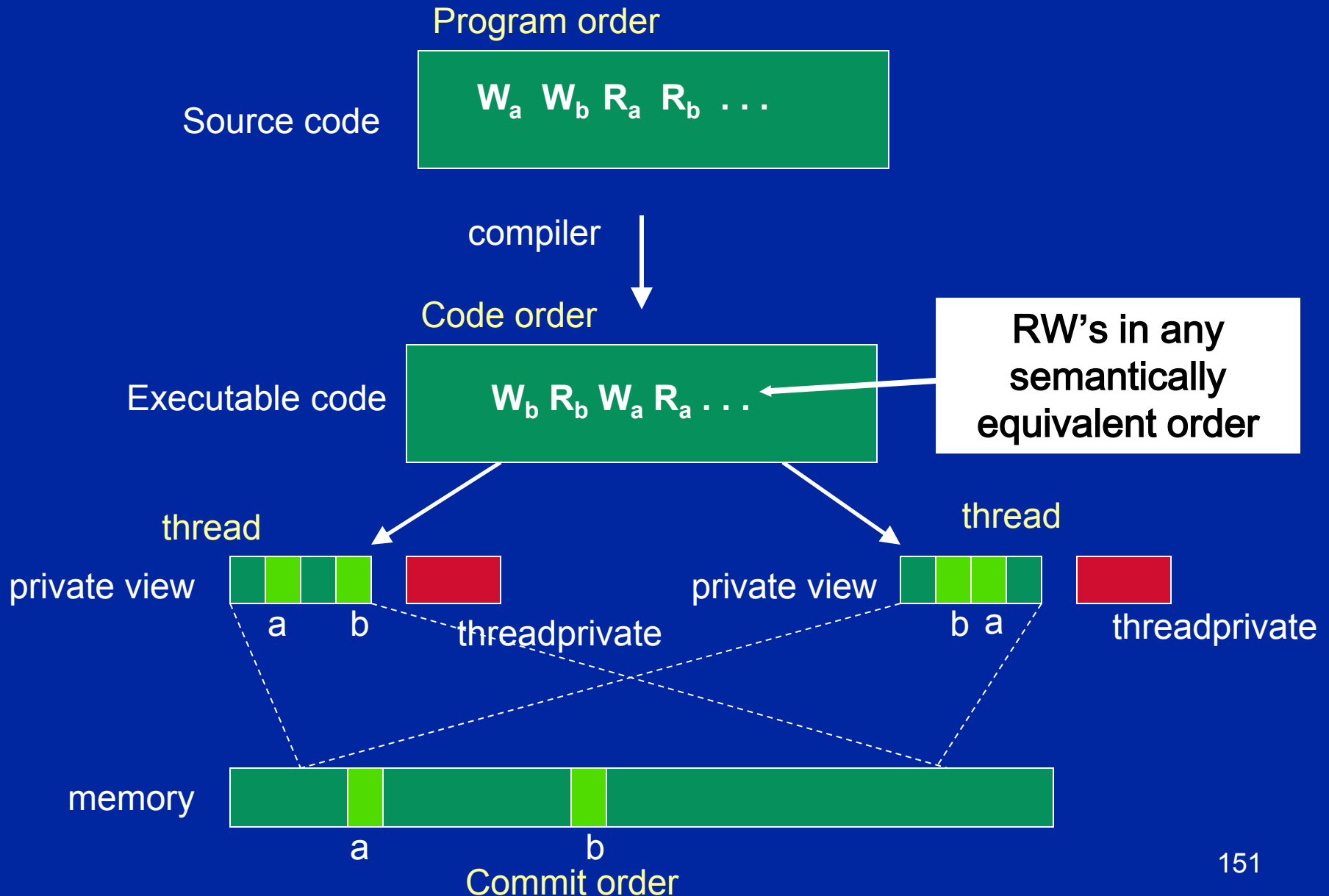
# OpenMP memory model

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



- A memory model is defined in terms of:
  - ◆ **Coherence**: Behavior of the memory system when a single address is accessed by multiple threads.
  - ◆ **Consistency**: Orderings of reads, writes, or synchronizations (RWS) with various addresses and by multiple threads.

# OpenMP Memory Model: Basic Terms



# Consistency: Memory Access Re-ordering

- **Re-ordering:**
  - ◆ Compiler re-orders program order to the code order
  - ◆ Machine re-orders code order to the memory commit order
- At a given point in time, the “private view” seen by a thread may be different from the view in shared memory.
- **Consistency Models** define constraints on the orders of Reads (R), Writes (W) and Synchronizations (S)
  - ◆ ... i.e. how do the values “seen” by a thread change as you change how ops follow ( $\rightarrow$ ) other ops.
  - ◆ Possibilities include:
    - $R \rightarrow R$ ,  $W \rightarrow W$ ,  $R \rightarrow W$ ,  $R \rightarrow S$ ,  $S \rightarrow S$ ,  $W \rightarrow S$



# Consistency

- Sequential Consistency:

- ◆ In a multi-processor, ops (R, W, S) are sequentially consistent if:
  - They remain in program order for each processor.
  - They are seen to be in the same overall order by each of the other processors.
- ◆ Program order = code order = commit order

- Relaxed consistency:

- ◆ Remove some of the ordering constraints for memory ops (R, W, S).

# OpenMP and Relaxed Consistency

- OpenMP defines consistency as a variant of weak consistency:
  - ◆ Can not reorder S ops with R or W ops on the same thread
    - **Weak consistency guarantees**  
 $S \rightarrow W, S \rightarrow R, R \rightarrow S, W \rightarrow S, S \rightarrow S$
- The Synchronization operation relevant to this discussion is flush.

# Flush

- Defines a sequence point at which a thread is guaranteed to see a consistent view of memory with respect to the “flush set”.
- The flush set is:
  - ◆ “all thread visible variables” for a flush construct without an argument list.
  - ◆ a list of variables when the “flush(list)” construct is used.
- The action of Flush is to guarantee that:
  - All R,W ops that overlap the flush set and occur prior to the flush complete before the flush executes
  - All R,W ops that overlap the flush set and occur after the flush don’t execute until after the flush.
  - Flushes with overlapping flush sets can not be reordered.

Memory ops: R = Read, W = write, S = synchronization

# Synchronization: flush example

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

```
double A;  
  
A = compute();  
  
flush(A); // flush to memory to make sure other  
          // threads can pick up the right value
```

**Note: OpenMP's flush is analogous to a fence in other shared memory API's.**

# Flush and synchronization

- A flush operation is implied by OpenMP synchronizations, e.g.
    - ◆ at entry/exit of parallel regions
    - ◆ at implicit and explicit barriers
    - ◆ at entry/exit of critical regions
    - ◆ whenever a lock is set or unset
- ....
- (but not at entry to worksharing regions or entry/exit of master regions)

# What is the Big Deal with Flush?

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

**Note: the flush operation does not actually synchronize different threads. It just ensures that a thread's values are made consistent with main memory.**

# Exercise 8: producer consumer

- Parallelize the “prod\_cons.c” program.
- This is a well known pattern called the producer consumer pattern
  - ◆ One thread produces values that another thread consumes.
  - ◆ Often used with a stream of produced values to implement “pipeline parallelism”
- The key is to implement pairwise synchronization between threads.

# Outline

- **Unit 1: Getting started with OpenMP**
  - ◆ Mod1: Introduction to parallel programming
  - ◆ Mod 2: The boring bits: Using an OpenMP compiler (hello world)
  - ◆ Disc 1: Hello world and how threads work
- **Unit 2: The core features of OpenMP**
  - ◆ Mod 3: Creating Threads (the Pi program)
  - ◆ Disc 2: The simple Pi program and why it sucks
  - ◆ Mod 4: Synchronization (Pi program revisited)
  - ◆ Disc 3: Synchronization overhead and eliminating false sharing
  - ◆ Mod 5: Parallel Loops (making the Pi program simple)
  - ◆ Disc 4: Pi program wrap-up
- **Unit 3: Working with OpenMP**
  - ◆ Mod 6: Synchronize single masters and stuff
  - ◆ Mod 7: Data environment
  - ◆ Disc 5: Debugging OpenMP programs
  - ◆ Mod 8: Skills practice ... linked lists and OpenMP
  - ◆ Disc 6: Different ways to traverse linked lists
- **Unit 4: a few advanced OpenMP topics**
  - ◆ Mod 8: Tasks (linked lists the easy way)
  - ◆ Disc 7: Understanding Tasks
  - ◆ Mod 8: The scary stuff ... Memory model, atomics, and flush (pairwise synch).
  - ➡ ◆ Disc 8: The pitfalls of pairwise synchronization
  - ◆ Mod 9: Threadprivate Data and how to support libraries (Pi again)
  - ◆ Disc 9: Random number generators
- **Unit 5: Recapitulation**



# Example: prod\_cons.c

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

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

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

    runtime = omp_get_wtime();

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

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

    runtime = omp_get_wtime() - runtime;

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

- Often used with a stream of produced values to implement “pipeline parallelism”
- The key is to implement pairwise synchronization between threads.

# Pair wise synchronizaion in OpenMP

- OpenMP lacks synchronization constructs that work between pairs of threads.
- When this is needed you have to build it yourself.
- Pair wise synchronization
  - ◆ Use a shared flag variable
  - ◆ Reader spins waiting for the new flag value
  - ◆ Use flushes to force updates to and from memory

# Example: producer consumer

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

Use flag to Signal when the  
“produced” value is ready

Flush forces refresh to memory.  
Guarantees that the other thread  
sees the new value of A

Flush needed on both “reader” and “writer”  
sides of the communication

Notice you must put the flush inside the  
while loop to make sure the updated flag  
variable is seen

The problem is this program technically has  
a race ... on the store and later load of flag.

# The OpenMP 3.1 atomics (1 of 2)

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

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

- Atomic can protect loads

**# pragma omp atomic read**

**v = x;**

- Atomic can protect stores

**# pragma omp atomic write**

**x = expr;**

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

**# pragma omp atomic update**

**x++; or ++x; or x--; or --x; or**

**x binop= expr; or x = x binop expr;**

**This is the  
original OpenMP  
atomic**

# The OpenMP 3.1 atomics (2 of 2)

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

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

- Where the statement is one of the following forms:

**v = x++;      v = ++x;      v = x--;      v = --x;      v = x binop expr;**

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

**{v = x; x binop = expr;}**

**{x binop = expr; v = x;}**

**{v=x; x=x binop expr;}**

**{X = x binop expr; v = x;}**

**{v = x; x++;}**

**{v=x; ++x;}**

**{++x; v=x;}**

**{x++; v = x;}**

**{v = x; x--;}**

**{v= x; --x;}**

**{--x; v = x;}**

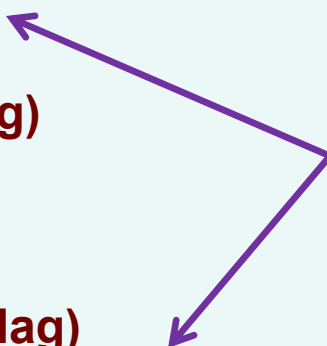
**{x--; v = x;}**

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


# Atomics and synchronization flags

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

This program is truly race free ... the reads and writes of flag are protected so the two threads can not conflict.



# Outline

- **Unit 1: Getting started with OpenMP**
  - ◆ Mod1: Introduction to parallel programming
  - ◆ Mod 2: The boring bits: Using an OpenMP compiler (hello world)
  - ◆ Disc 1: Hello world and how threads work
- **Unit 2: The core features of OpenMP**
  - ◆ Mod 3: Creating Threads (the Pi program)
  - ◆ Disc 2: The simple Pi program and why it sucks
  - ◆ Mod 4: Synchronization (Pi program revisited)
  - ◆ Disc 3: Synchronization overhead and eliminating false sharing
  - ◆ Mod 5: Parallel Loops (making the Pi program simple)
  - ◆ Disc 4: Pi program wrap-up
- **Unit 3: Working with OpenMP**
  - ◆ Mod 6: Synchronize single masters and stuff
  - ◆ Mod 7: Data environment
  - ◆ Disc 5: Debugging OpenMP programs
  - ◆ Mod 8: Skills practice ... linked lists and OpenMP
  - ◆ Disc 6: Different ways to traverse linked lists
- **Unit 4: a few advanced OpenMP topics**
  - ◆ Mod 8: Tasks (linked lists the easy way)
  - ◆ Disc 7: Understanding Tasks
  - ◆ Mod 8: The scary stuff ... Memory model, atomics, and flush (pairwise synch).
  - ◆ Disc 8: The pitfalls of pairwise synchronization
  -  ◆ Mod 9: Threadprivate Data and how to support libraries (Pi again)
  - ◆ Disc 9: Random number generators
- **Unit 5: Recapitulation**

# Data sharing: Threadprivate

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



# A threadprivate example (C)

Use threadprivate to create a counter for each thread.

```
int counter = 0;  
#pragma omp threadprivate(counter)  
  
int increment_counter()  
{  
    counter++;  
    return (counter);  
}
```

# Data Copying: Copyin

You initialize threadprivate data using a copyin clause.

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

```
C Initialize the A array  
call init_data(N,A)
```

```
!$OMP PARALLEL COPYIN(A)
```

... Now each thread sees threadprivate array A initialised  
... to the global value set in the subroutine init\_data()

```
!$OMP END PARALLEL
```

```
end
```

# Data Copying: Copyprivate

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

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

void main()
{
    int Nsize, choice;

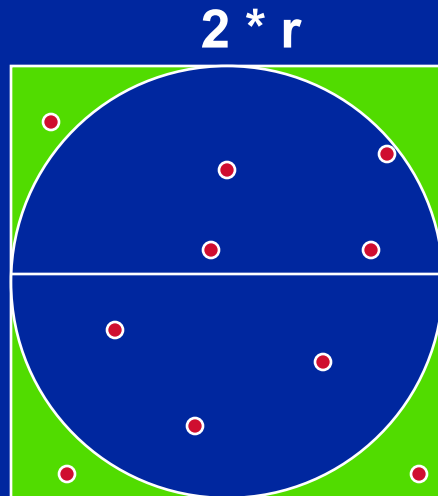
    #pragma omp parallel private (Nsize, choice)
    {
        #pragma omp single copyprivate (Nsize, choice)
            input_parameters (Nsize, choice);

        do_work(Nsize, choice);
    }
}
```

# Exercise 9: Monte Carlo Calculations

Using Random numbers to solve tough problems

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



N= 10	$\pi = 2.8$
N=100	$\pi = 3.16$
N= 1000	$\pi = 3.148$

- Throw darts at the circle/square.
- Chance of falling in circle is proportional to ratio of areas:  
$$A_c = r^2 * \pi$$
$$A_s = (2*r) * (2*r) = 4 * r^2$$
$$P = A_c/A_s = \pi / 4$$
- Compute  $\pi$  by randomly choosing points, count the fraction that falls in the circle, compute  $\pi$ .

# Exercise 9

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

# Outline

- **Unit 1: Getting started with OpenMP**
  - ◆ Mod1: Introduction to parallel programming
  - ◆ Mod 2: The boring bits: Using an OpenMP compiler (hello world)
  - ◆ Disc 1: Hello world and how threads work
- **Unit 2: The core features of OpenMP**
  - ◆ Mod 3: Creating Threads (the Pi program)
  - ◆ Disc 2: The simple Pi program and why it sucks
  - ◆ Mod 4: Synchronization (Pi program revisited)
  - ◆ Disc 3: Synchronization overhead and eliminating false sharing
  - ◆ Mod 5: Parallel Loops (making the Pi program simple)
  - ◆ Disc 4: Pi program wrap-up
- **Unit 3: Working with OpenMP**
  - ◆ Mod 6: Synchronize single masters and stuff
  - ◆ Mod 7: Data environment
  - ◆ Disc 5: Debugging OpenMP programs
  - ◆ Mod 8: Skills practice ... linked lists and OpenMP
  - ◆ Disc 6: Different ways to traverse linked lists
- **Unit 4: a few advanced OpenMP topics**
  - ◆ Mod 8: Tasks (linked lists the easy way)
  - ◆ Disc 7: Understanding Tasks
  - ◆ Mod 8: The scary stuff ... Memory model, atomics, and flush (pairwise synch).
  - ◆ Disc 8: The pitfalls of pairwise synchronization
  - ◆ Mod 9: Threadprivate Data and how to support libraries (Pi again)
  - ➡ ◆ Disc 9: Random number generators
- **Unit 5: Recapitulation**

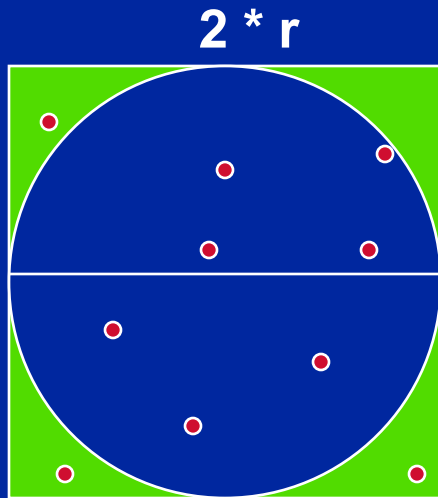
# Computers and random numbers

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

# Monte Carlo Calculations:

## Using Random numbers to solve tough problems

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



N= 10	$\pi = 2.8$
N=100	$\pi = 3.16$
N= 1000	$\pi = 3.148$

- Throw darts at the circle/square.
- Chance of falling in circle is proportional to ratio of areas:  
$$A_c = r^2 * \pi$$
$$A_s = (2*r) * (2*r) = 4 * r^2$$
$$P = A_c / A_s = \pi / 4$$
- Compute  $\pi$  by randomly choosing points, count the fraction that falls in the circle, compute pi.



# Parallel Programmers love Monte Carlo algorithms

```
#include "omp.h"
```

```
static long num_trials = 10000;
```

```
int main ()
```

```
{
```

```
    long i;    long Ncirc = 0;    double pi, x, y;
```

```
    double r = 1.0; // radius of circle. Side of square is 2*r
```

```
    seed(0,-r, r); // The circle and square are centered at the origin
```

```
    #pragma omp parallel for private (x, y) reduction (+:Ncirc)
```

```
    for(i=0;i<num_trials; i++)
```

```
    {
```

```
        x = random();    y = random();
```

```
        if ( x*x + y*y) <= r*r)  Ncirc++;
```

```
    }
```

```
    pi = 4.0 * ((double)Ncirc/((double)num_trials);
```

```
    printf("\n %d trials, pi is %f \n",num_trials, pi);
```

```
}
```

Embarrassingly parallel: the parallelism is so easy its embarrassing.

Add two lines and you have a parallel program.

# Linear Congruential Generator (LCG)

- LCG: Easy to write, cheap to compute, portable, OK quality

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

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

# LCG code

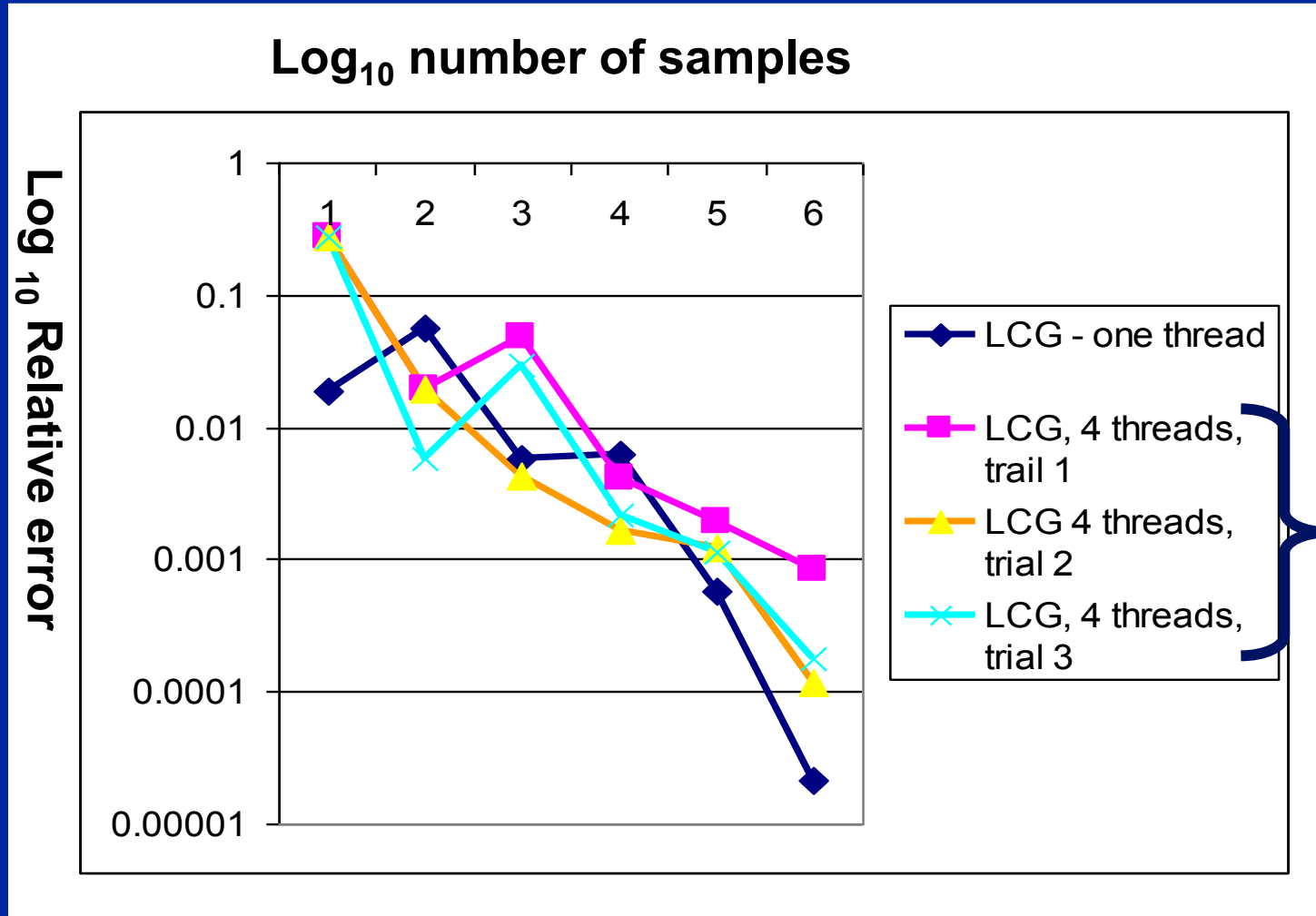
```
static long MULTIPLIER = 1366;
static long ADDEND     = 150889;
static long PMOD       = 714025;
long random_last = 0;
double random ()
{
    long random_next;

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

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

**Seed the pseudo random  
sequence by setting  
random\_last**

# Running the PI\_MC program with LCG generator



**Run the same program the same way and get different answers!**

**That is not acceptable!**

**Issue: my LCG generator is not threadsafe**

# LCG code: threadsafe version

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

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

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

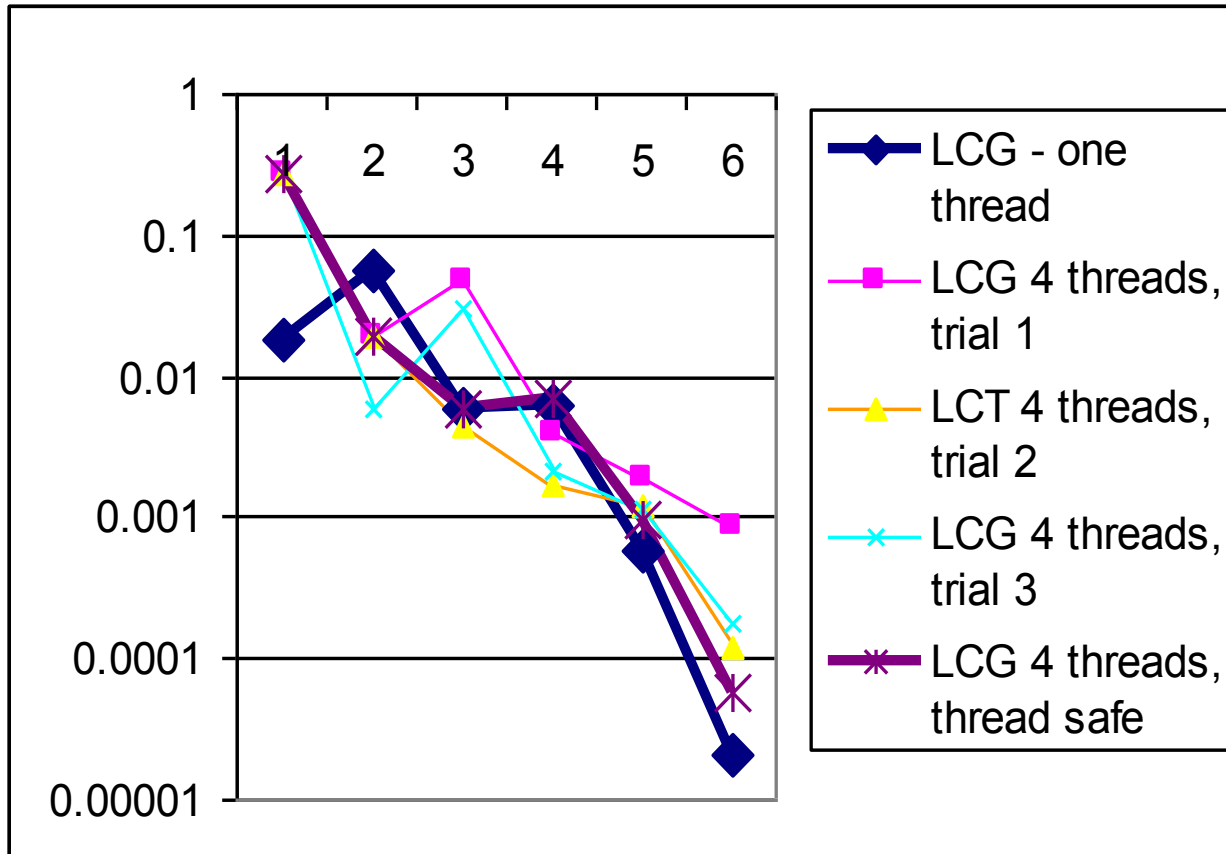
**random\_last** carries state between random number computations,

To make the generator threadsafe, make **random\_last** **threadprivate** so each thread has its own copy.

# Thread safe random number generators

Log<sub>10</sub> number of samples

Log<sub>10</sub> Relative error



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

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

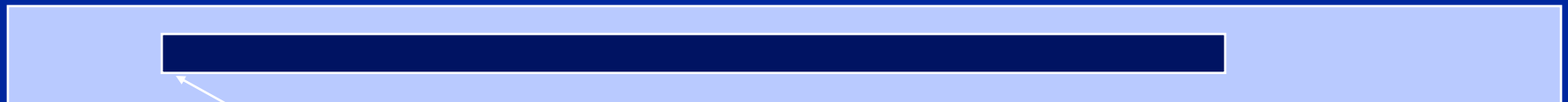
**Why?**

# Pseudo Random Sequences

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



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



Seed determines starting point

- Grab arbitrary seeds and you may generate overlapping sequences
  - ◆ E.g. three sequences ... last one wraps at the end of the RNG period.



- Overlapping sequences = over-sampling and bad statistics ... lower quality or even wrong answers!

# Parallel random number generators

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

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

Nice for debugging, but not really needed scientifically.



# Leap Frog method

- Interleave samples in the sequence of pseudo random numbers:
  - ◆ Thread  $i$  starts at the  $i^{\text{th}}$  number in the sequence
  - ◆ Stride through sequence, stride length = number of threads.
- Result ... the same sequence of values regardless of the number of threads.

```
#pragma omp single
{
    nthreads = omp_get_num_threads();
    iseed = PMOD/MULTIPLIER;    // just pick a seed
    pseed[0] = iseed;
    mult_n = MULTIPLIER;
    for (i = 1; i < nthreads; ++i)
    {
        iseed = (unsigned long long)((MULTIPLIER * iseed) % PMOD);
        pseed[i] = iseed;
        mult_n = (mult_n * MULTIPLIER) % PMOD;
    }
}

random_last = (unsigned long long) pseed[id];
```

One thread  
computes offsets  
and strided  
multiplier

LCG with Addend = 0 just  
to keep things simple

Each thread stores offset starting  
point into its threadprivate "last  
random" value

# Same sequence with many threads.

- We can use the leapfrog method to generate the same answer for any number of threads

Steps	One thread	2 threads	4 threads
1000	3.156	3.156	3.156
10000	3.1168	3.1168	3.1168
100000	3.13964	3.13964	3.13964
1000000	3.140348	3.140348	3.140348
10000000	3.141658	3.141658	3.141658

# Outline

- **Unit 1: Getting started with OpenMP**
  - ◆ Mod1: Introduction to parallel programming
  - ◆ Mod 2: The boring bits: Using an OpenMP compiler (hello world)
  - ◆ Disc 1: Hello world and how threads work
- **Unit 2: The core features of OpenMP**
  - ◆ Mod 3: Creating Threads (the Pi program)
  - ◆ Disc 2: The simple Pi program and why it sucks
  - ◆ Mod 4: Synchronization (Pi program revisited)
  - ◆ Disc 3: Synchronization overhead and eliminating false sharing
  - ◆ Mod 5: Parallel Loops (making the Pi program simple)
  - ◆ Disc 4: Pi program wrap-up
- **Unit 3: Working with OpenMP**
  - ◆ Mod 6: Synchronize single masters and stuff
  - ◆ Mod 7: Data environment
  - ◆ Disc 5: Debugging OpenMP programs
  - ◆ Mod 8: Skills practice ... linked lists and OpenMP
  - ◆ Disc 6: Different ways to traverse linked lists
- **Unit 4: a few advanced OpenMP topics**
  - ◆ Mod 8: Tasks (linked lists the easy way)
  - ◆ Disc 7: Understanding Tasks
  - ◆ Mod 8: The scary stuff ... Memory model, atomics, and flush (pairwise synch).
  - ◆ Disc 8: The pitfalls of pairwise synchronization
  - ◆ Mod 9: Threadprivate Data and how to support libraries (Pi again)
  - ◆ Disc 9: Random number generators



- **Unit 5: Recapitulation**

# 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.

# 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;
}
```

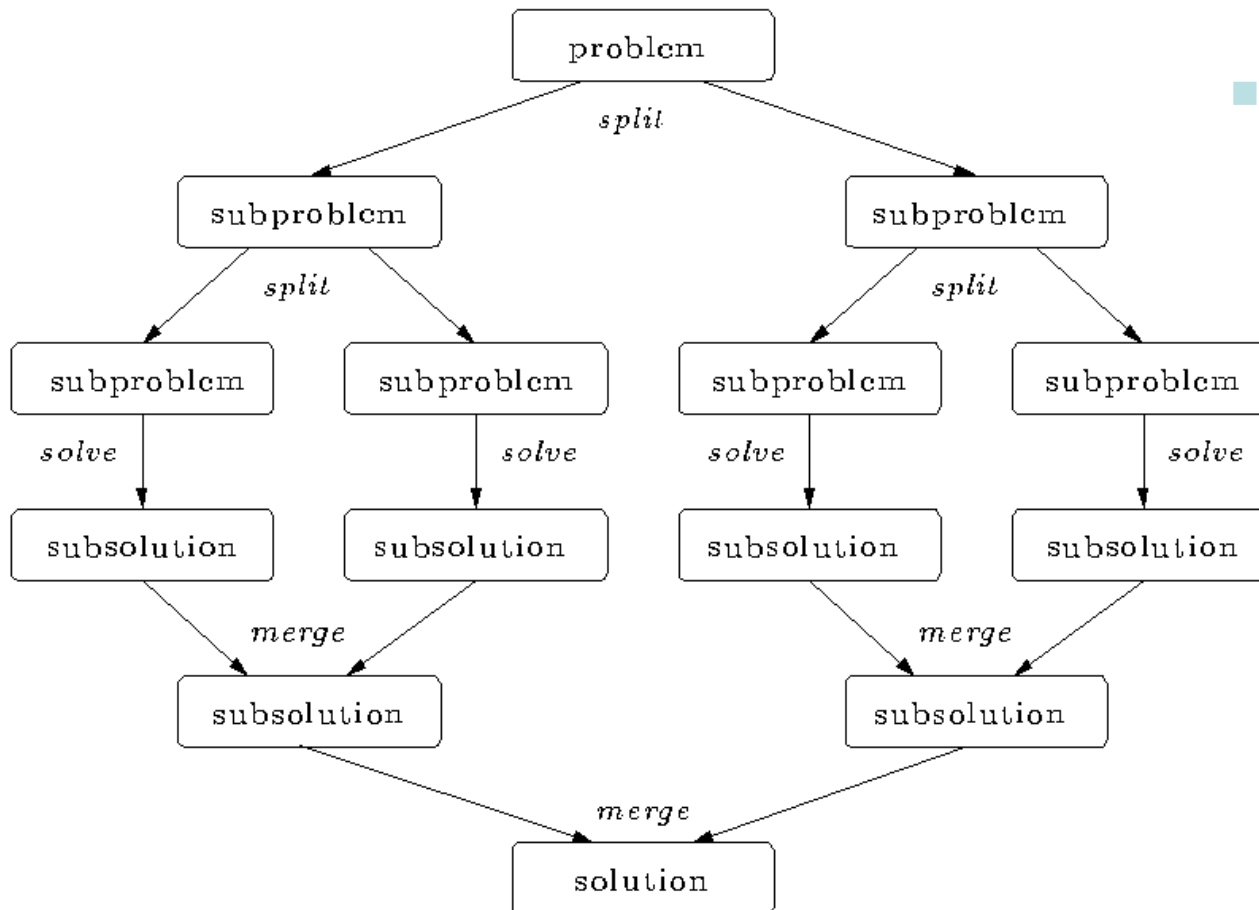


# Divide and Conquer Pattern

- 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;
}
```

# Results\*: pi with tasks

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

## 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);
    #pragma omp task shared(sum2)
    sum2 = pi_comp(Nfinish-iblk/2, Nfinish);
    #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
```

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.

# Learning more about OpenMP:

## OpenMP Organizations

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

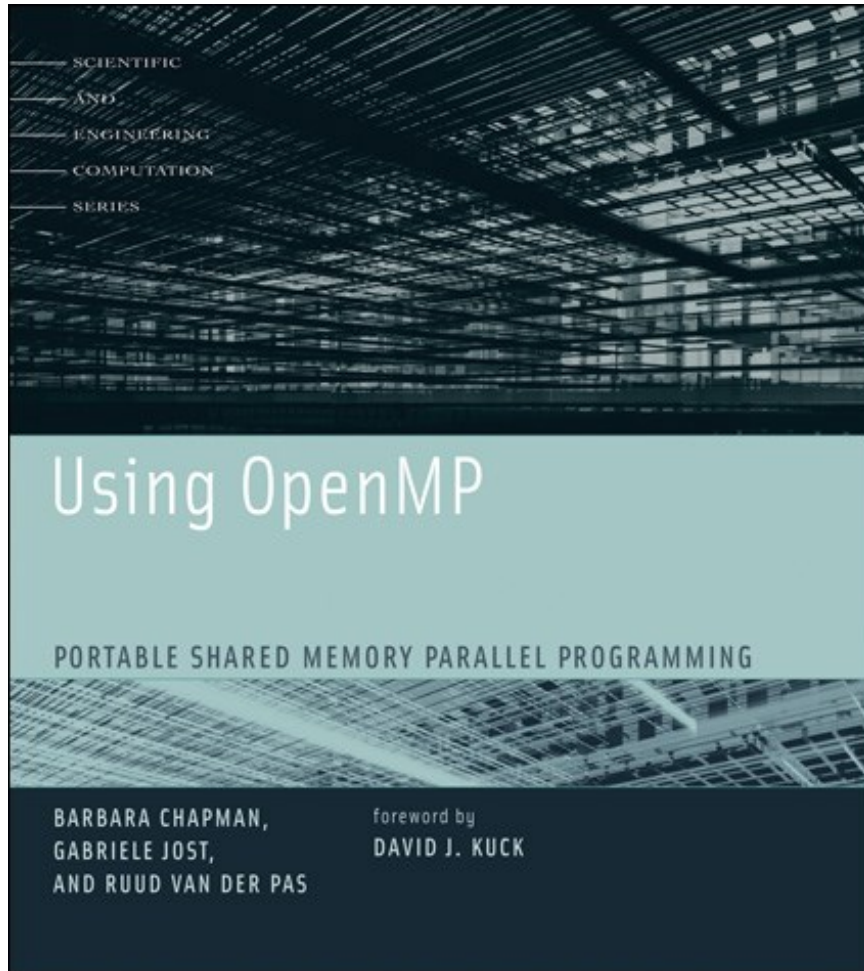
**[www.openmp.org](http://www.openmp.org)**

- OpenMP User’s Group (cOMPunity) URL:

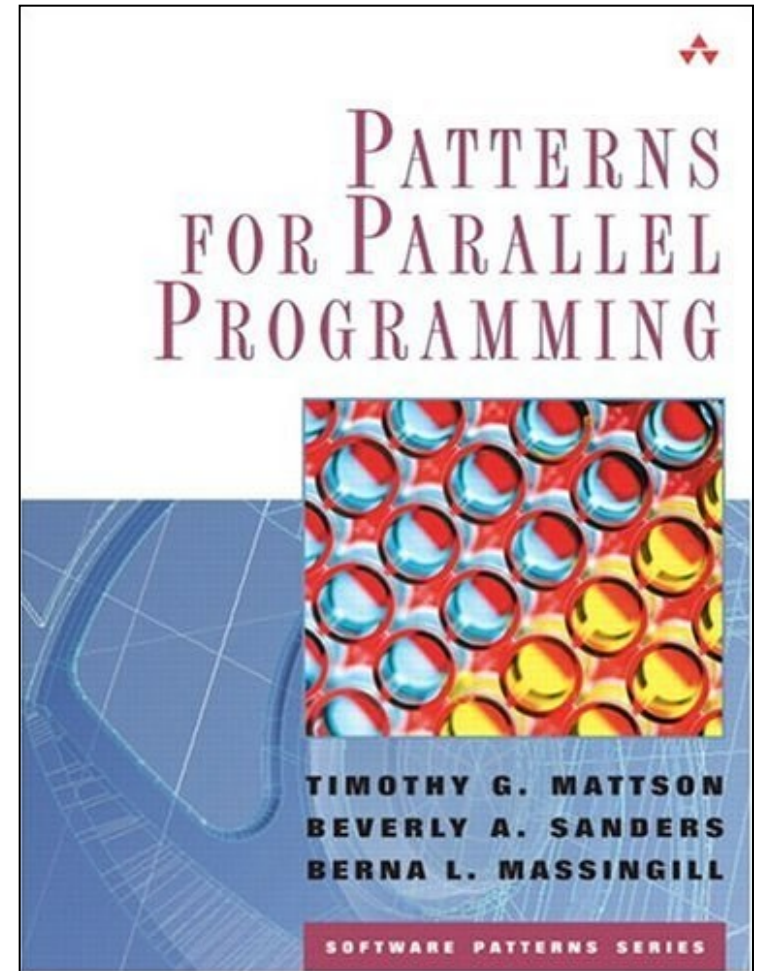
**[www.compunity.org](http://www.compunity.org)**

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

# Books about OpenMP

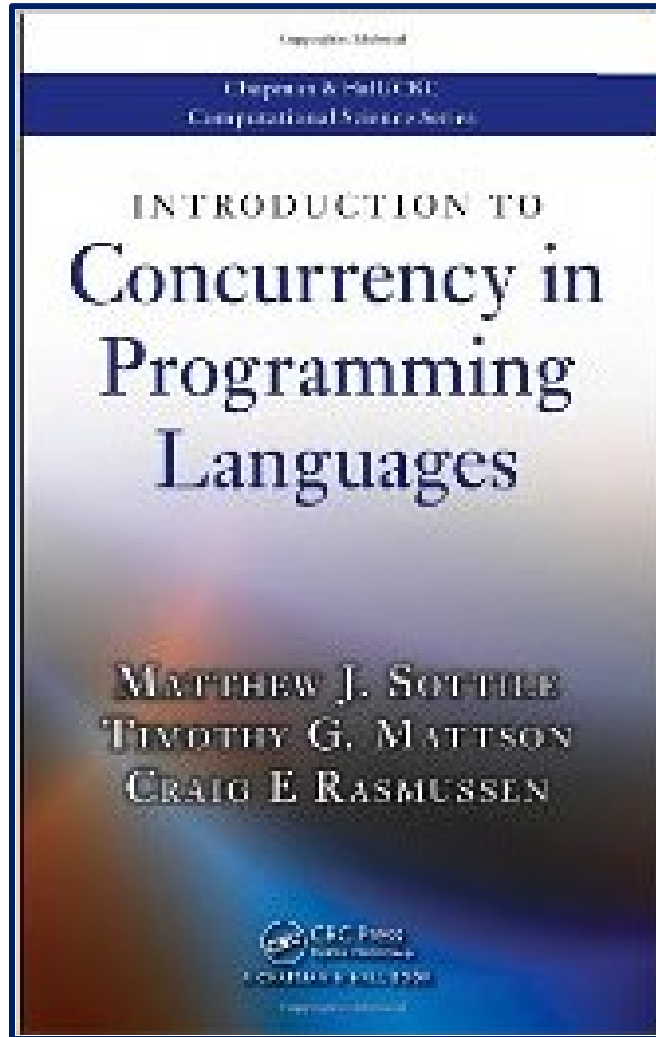


**An excellent book about using OpenMP  
... though out of date (OpenMP 2.5)**

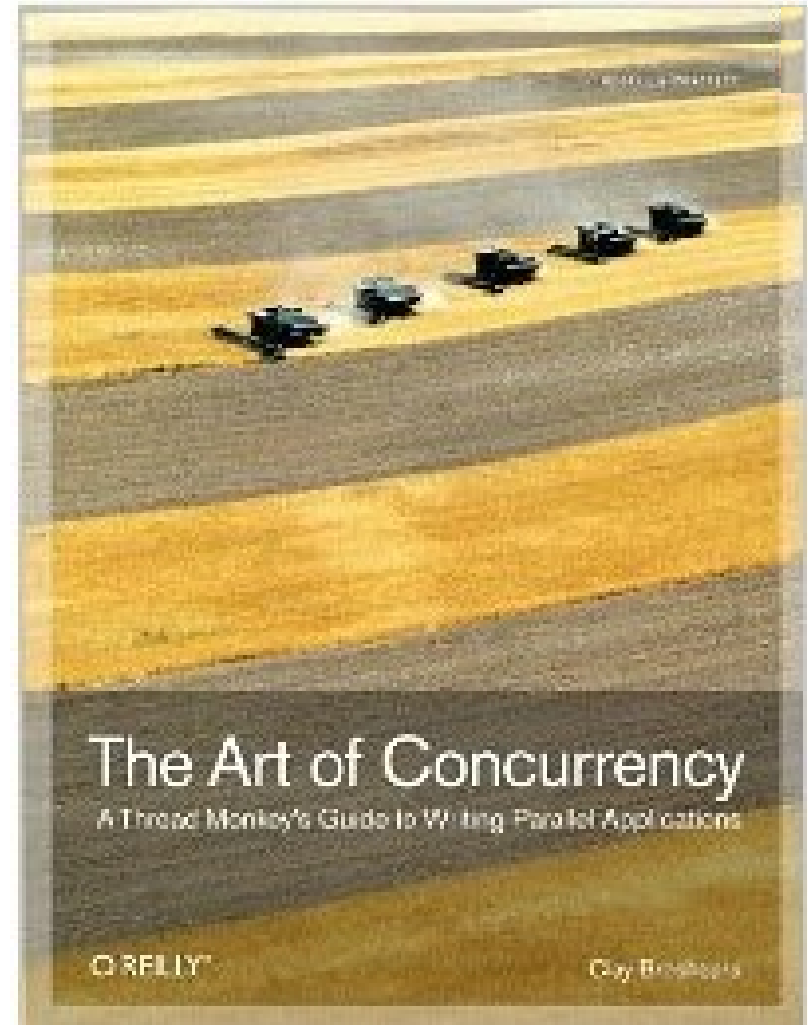


**A book about how to “think  
parallel” with examples in  
OpenMP, MPI and Java**

# 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)**



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

OpenMP API 3.1 C/C++

Page 2

## Runtime Library Routines

### Execution Environment Routines [3.1.2]

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

#### `void omp_get_num_threads( int *n_threads );`

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

#### `int omp_get_num_threads(void);`

Returns the number of threads in the current team.

#### `int omp_get_max_threads(void);`

Returns the maximum number of threads that could be created for a new team using a parallel construct without a new `n_threads` clause.

#### `int omp_get_num_procs(void);`

Returns the ID of the enclosing thread where IDs range from zero to the size of the team minus 1.

#### `int omp_get_max_procs(void);`

Returns the number of processors available to the program.

### Data Types For Runtime Library Routines

`omp_lock_t` Represents a simple lock.

`omp_mutex_t` Represents a mutex lock.

`omp_sched_t` Represents a schedule.

#### `int omp_get_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 flag );`

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

#### `int omp_get_dynamic(void);`

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

#### `int omp_get_max_parallel(void);`

Returns the maximum number of parallel regions, by setting the `max_parallel` flag.

#### `int omp_get_parallel(void);`

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

#### `void omp_set_schedule(omp_sched_t kind, int mode);`

Affects the schedule that is applied when routines are used as schedule hint, by setting the value of the `schedule` flag. `kind` is one of static, dynamic, guided, auto, or an implementation-defined schedule. See `omp_sched_t` [3.1.2] for descriptions.

#### `void omp_get_schedule( omp_sched_t *kind, int *mode );`

Returns the value of `omp_sched_t`, which is the schedule applied when routines are used as schedule hint.

See also:

#### `int omp_get_thread_id(void);`

Returns the value of the thread ID for the current thread, which is the maximum number of OpenMP threads available to the program.

#### `void omp_get_thread_id(void);`

Returns the value of the thread ID for the current thread, which is the maximum number of OpenMP threads available to the program.

#### `int omp_get_thread_id(void);`

Returns the value of the thread ID for the current thread, which is the maximum number of OpenMP threads available to the program.

#### `int omp_get_thread_id(void);`

Returns the value of the thread ID for the current thread, which is the maximum number of OpenMP threads available to the program.

#### `int omp_get_thread_id(void);`

Returns the value of the thread ID for the current thread, which is the maximum number of OpenMP threads available to the program.

#### `int omp_get_thread_id(void);`

Returns the value of the thread ID for the current thread, which is the maximum number of OpenMP threads available to the program.

#### `int omp_get_thread_id(void);`

Returns the value of the thread ID for the current thread, which is the maximum number of OpenMP threads available to the program.

#### `int omp_get_thread_id(void);`

Returns the value of the thread ID for the current thread, which is the maximum number of OpenMP threads available to the program.

#### `int omp_get_thread_id(void);`

Returns the value of the thread ID for the current thread, which is the maximum number of OpenMP threads available to the program.

## Clauses

The set of clauses that is valid on a particular directive is described with the directive. Most clauses accept a comma-separated list of list items. All list items appearing in a clause must be valid.

### Data Sharing Attribute Clauses [3.1.2]

Qualifying attribute clauses apply to directives whose scope is visible in the construct on which the clause appears.

#### `default(shared) [none]`

Defines the default data-sharing attributes of variables that are referenced in a parallel or task construct.

#### `private([t])`

Declares one or more list items to be shared by tasks generated by a parallel or task construct.

#### `private([t])`

Declares one or more list items to be private to a task.

#### `private([t])`

Declares one or more list items to be private to a task, and indicates that all of them with the value that the corresponding original item has when the construct is encountered.

#### `private([t])`

Declares one or more list items to be private to an implicit task, and causes the corresponding original item to be updated after the end of the region.

### Reduction Clauses [3.1.2]

Reduction clauses use the list items within the indicated associative operator. Associative operators take a prefix copy for each list item which is then combined with the original item.

#### `OpenMP for reduction( reduction_operator, list_items )`

Reduction clauses use the list items within the indicated associative operator. Associative operators take a prefix copy for each list item which is then combined with the original item.

Reduction clauses use the list items within the indicated associative operator. Associative operators take a prefix copy for each list item which is then combined with the original item.

Reduction clauses use the list items within the indicated associative operator. Associative operators take a prefix copy for each list item which is then combined with the original item.

Reduction clauses use the list items within the indicated associative operator. Associative operators take a prefix copy for each list item which is then combined with the original item.

Reduction clauses use the list items within the indicated associative operator. Associative operators take a prefix copy for each list item which is then combined with the original item.

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