# CS301: HPC

## Module-2

# Terminology



- SISD and SIMD (So there is Instruction and there is Data )
- Task
- Pipelining
- Shared Memory
- Distributed memory
- Granularity
- Observed Speedup
- Communications
- Synchronization
- Parallel Overhead
- Massively Parallel .
- Embarrassingly Parallel
- Scalability

Different ways to classify parallel Computers (Flynn's Taxonomy)

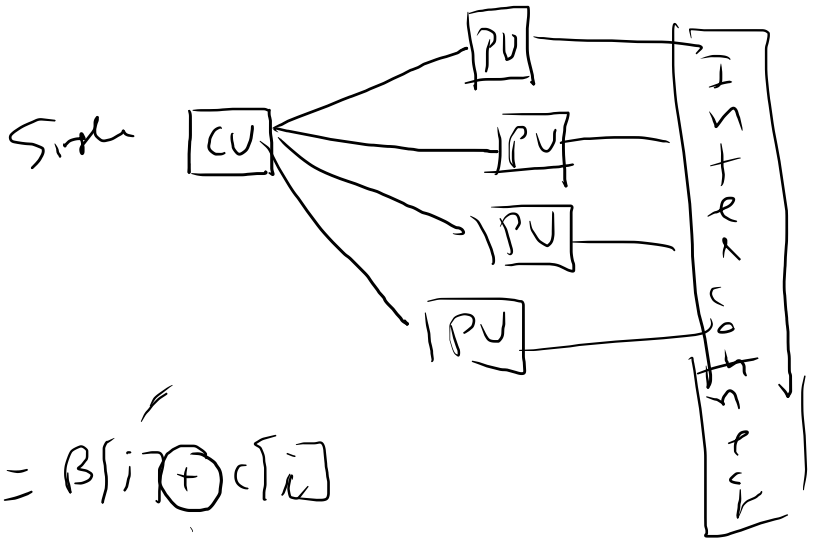
Instruction Stream
Data Stream
} two possible States — (Single) or Multiple

Stream → Sequence of flow

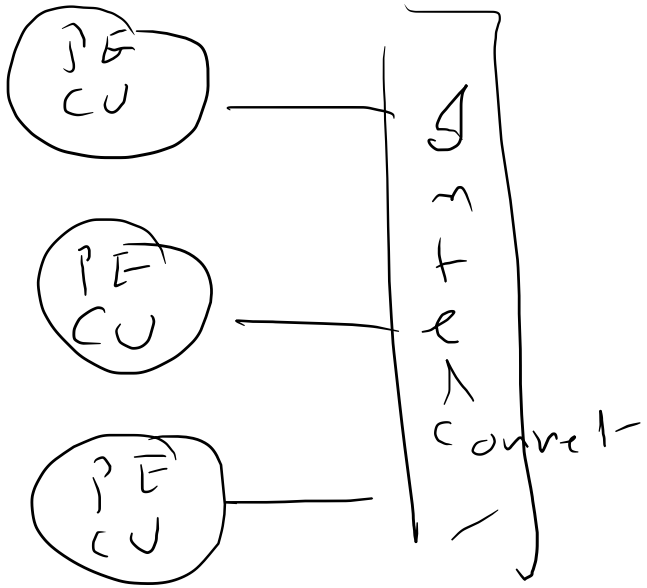
SIMD Architecture

| SISD | SIMD |
|---|---|
| MISD | MIMD |

PU (Processing unit) / Processing element




A[i] = B[i] + C[i]




SPMD

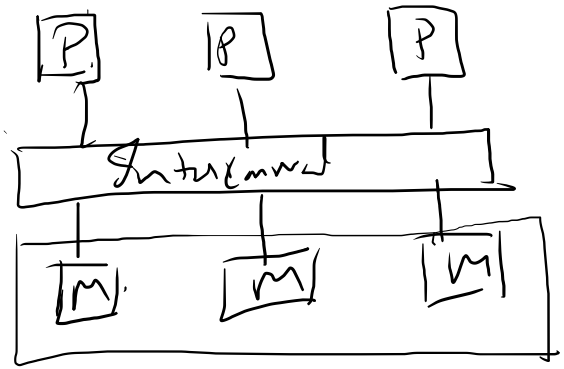
MIMD

# Shared Memory Model

Shared Address space → Accessible to all processors

Time taken by the processors to access Memory is identical

→ UMA multiprocessor





NUMA

# First example

**Compute n values and add them together**

**Serial code – simple !! →**
**Sum=0; for loop i=1,n; compute x; sum +=x.**

**Suppose you have "p" cores and "p<<n"**

**How you will do it in parallel?**

# First example

Compute n values and add them together

Suppose you have “p” cores and “p<<n”

“partial -sum” for each core

Each core → divide in independent blocks, private variables, Say “my_sum” of each core.

Then “global-sum” → sum of all “my_sum”

Master core gets all “my_sum” from other cores.

# First Example

Is it the best way always?

Method is generalization of the serial global sum ; divide the work among cores and then master core simply repeats the basic serial addition.

Efficient Parallelization is not parallelization of each step of the serial algorithm, but devising an entirely new algorithm.

Better algorithm is possible !!

- **Communication among the cores,**
- **load balancing,**
- **and synchronization.**

**Complexity of the algorithm.**

**Types of Parallelism. → Partitioning among cores.**

## Granularity

Ratio of computation to communication.

- ***Coarse:*** large amounts of computational work between communication events
- ***Fine:*** small amounts of computational work between communication events

## Parallel Overhead

The amount of time required to coordinate parallel tasks, as opposed to doing useful work. Parallel overhead includes:

- Task start-up time
- Synchronizations
- Data communications
- Software overhead imposed by parallel languages, libraries, operating system, etc.
- Task termination time

# Parallel Programming Complexity

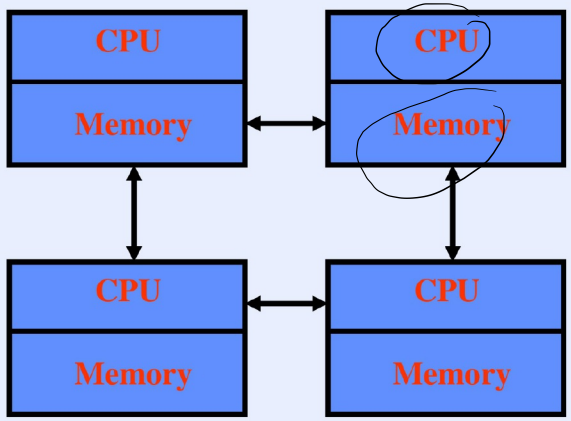
**Algorithm +**

- Scope of parallelism? (Amdahl's Law)
- Granularity
- Locality
- Load balance
- Coordination and Synchronization

**These makes parallel programming harder than sequential programming.**

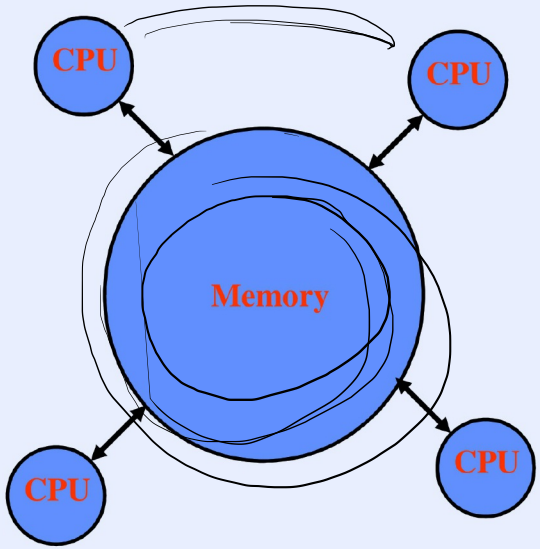
# Memory Types





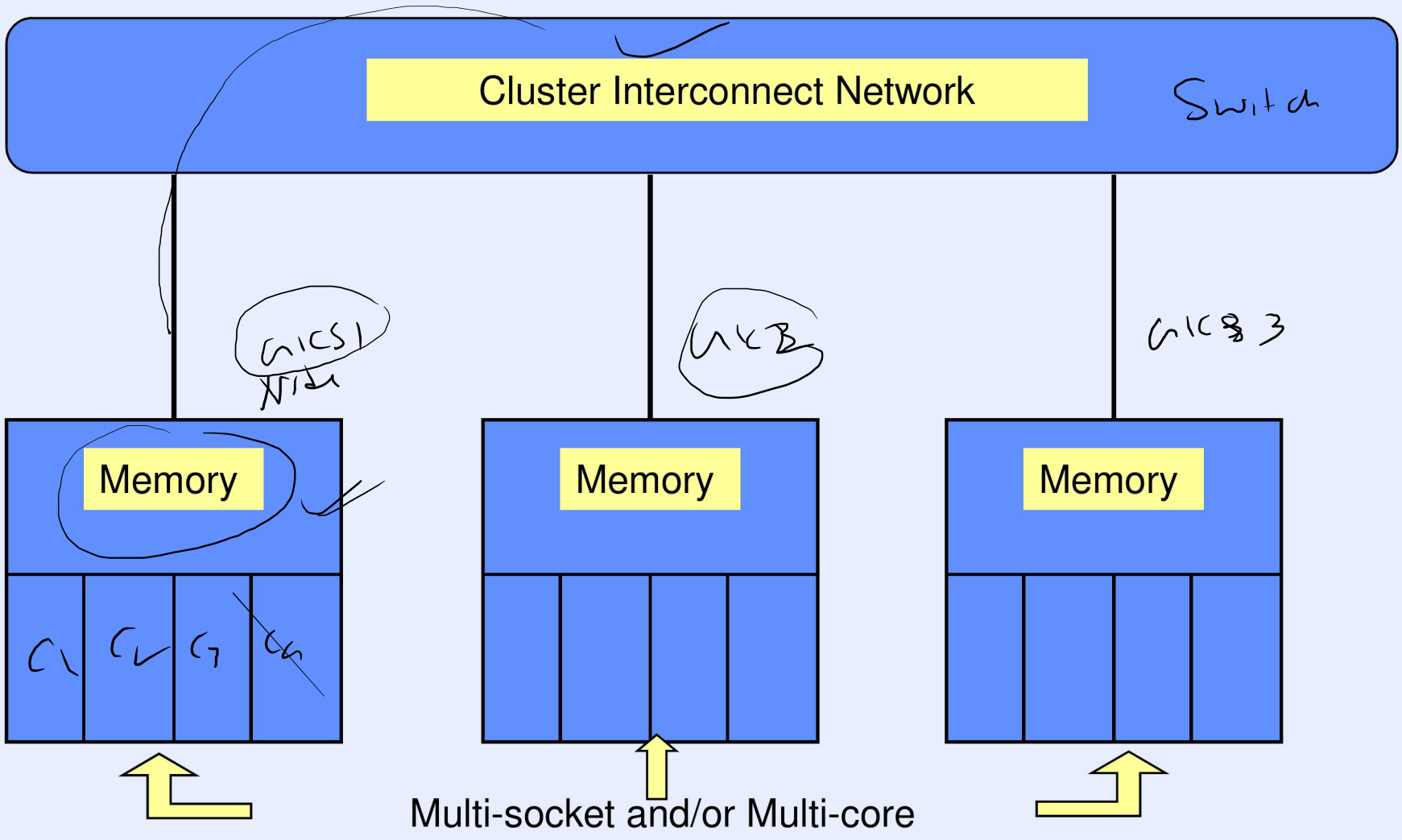

Distributed

Shared



# Clustered SMPs

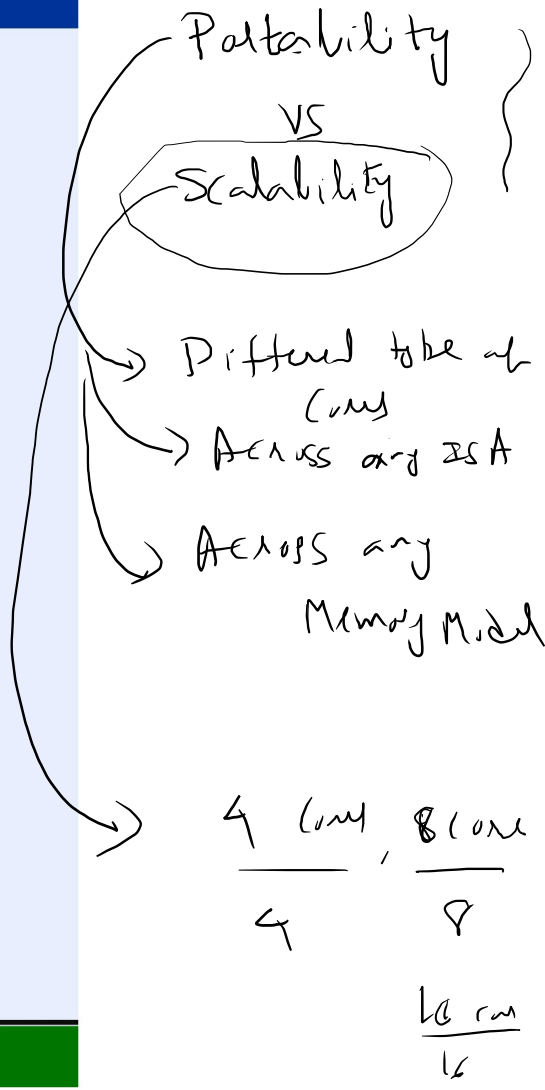




# Distributed vs. Shared Memory



- **Shared** - all processors share a global memory
  - *simpler to program*
  - *bus contention leads to poor scalability*
- **Distributed** - each processor physically has it's own (private) memory associated with it
  - *scales well*
  - *memory management is more difficult*



# Process vs Thread

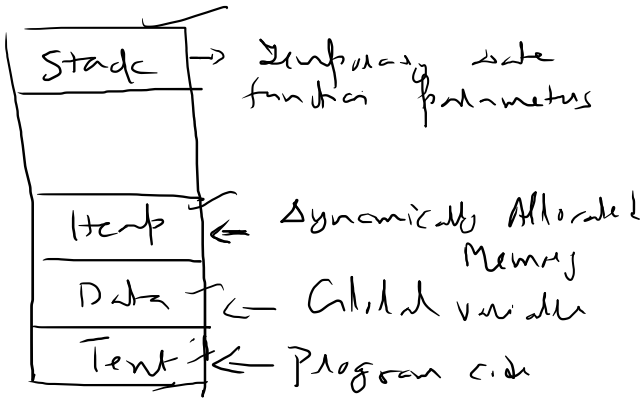
Process → Associated Address space

Process

↳ Instance of a program in execution

Process

* Associate Address space
* Process control block (PCB) → OS uses to track
  1) Process ID
  2) Process status → New, Ready, Running, Waiting, terminated
  3) CPU registers
  4) O/S Resources
     ↳ Memory management



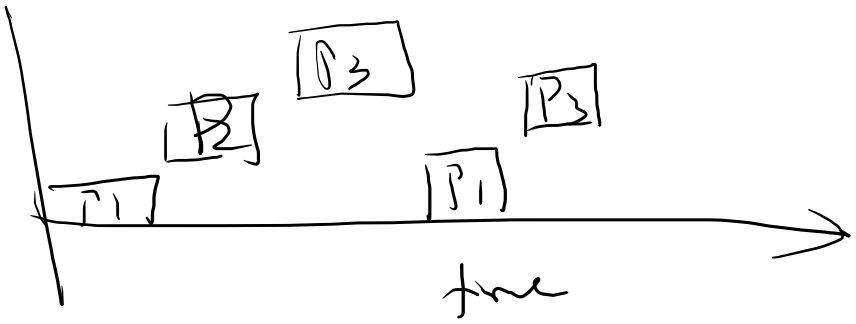

Process creation

* Load Code & data into memory
* Create & Initialize PCB
* Create first thread with call stack
* Make process known

* Stack vs heap
  ↓
  fast

* Stack vs heap
  ↳ faster
* Both are stored in RAM

A system is said to be *concurrent* if it can support two or more actions *in progress* at the same time.
A system is said to be *parallel* if it can support two or more actions executing simultaneously.




Switching

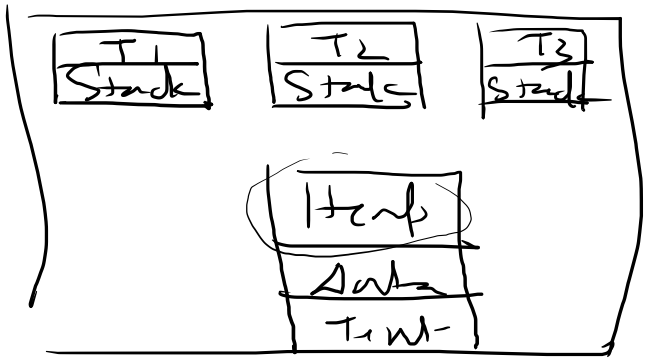
Concurrency is about dealing with lots of things at once. Parallelism is about doing lots of things at once.
Parallelism needs hardware with multiple processing units (essential).
In single-core CPU, you can get concurrency but NOT parallelism.
Parallelism is a specific kind of concurrency --> tasks are really executed simultaneously. ---> Performance

# Thread

- Basic unit of CPU utilization → lightweight process

Thread includes
- Thread ID
- Program Counter
- Register Set
- Stack

Shares Resources
- text Section
- Data Section
- Other OS Resources



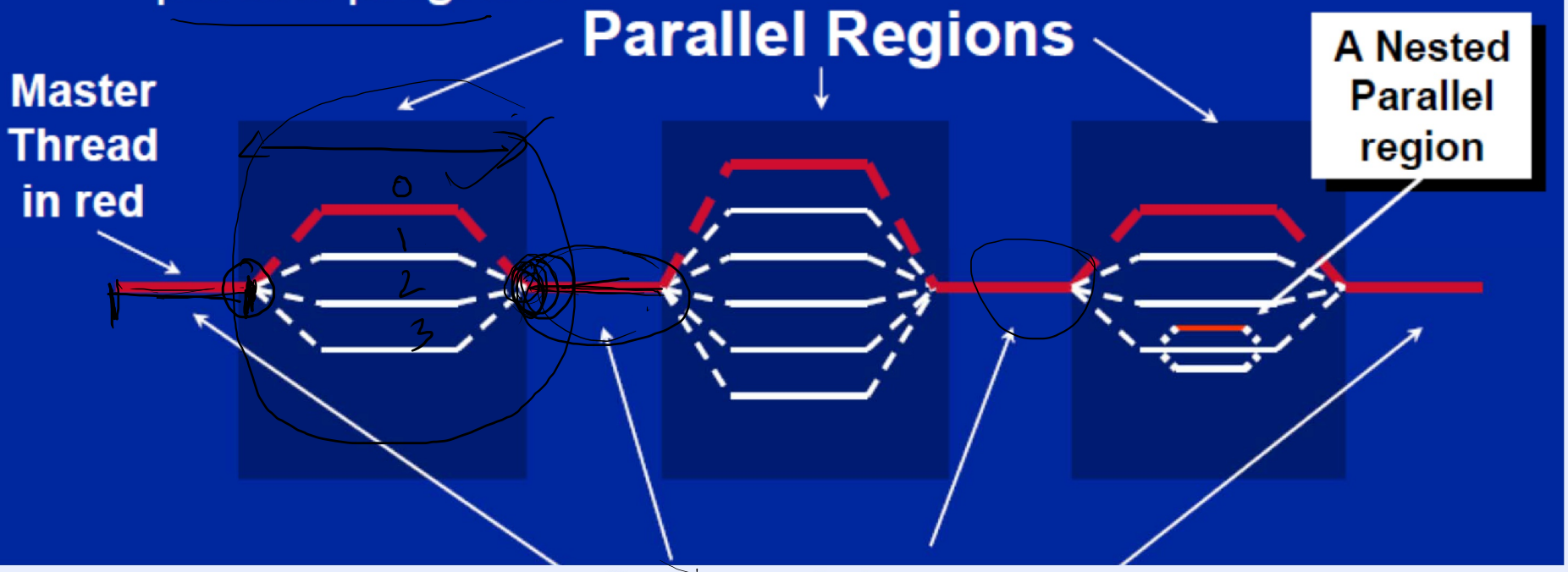
# Fork-Join Concept



- Fork-Join is a fundamental way of expressing concurrency within a computation
- ***Fork*** is called by a thread (*parent*) to create a new thread (*child*) of concurrency
  - *Parent continues after the Fork operation*
  - *Child begins operation separate from the parent*
  - *Fork creates concurrency*
- ***Join*** is called by both the parent and child
  - *Child calls Join after it finishes (implicitly on exit)*
  - *Parent waits until child joins (continues afterwards)*
  - *Join removes concurrency because child exits*

# 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.



# Fork-Join Concurrency Semantics



- Fork-Join is a concurrency control mechanism
  - *Fork increases concurrency*
  - *Join decreases concurrency*
- Fork-Join dependency rules
  - *A parent must join with its forked children*
  - *Forked children with the same parent can join with the parent in any order*

# Important terms



- UMA vs NUMA (memory)
- Directives/Pragmas
- Process vs threads (concurrency)
- Shared vs Private memory
- Structured block
- SPMD

Structured block ⟹ one point of entry & one point of exit

Call to function exit is allowed.

{ for i=1, N } ≡ { } ↓ branching is not allowed

# We need to understand

**programming** models,

**execution** models,

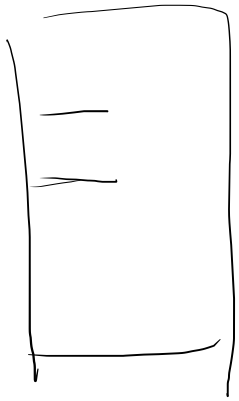
**communication** models and

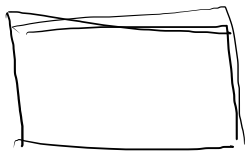
**memory** models!

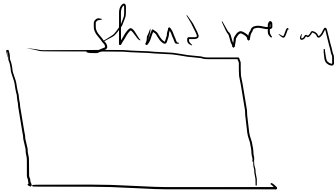
# OpenMP - User Interface Model

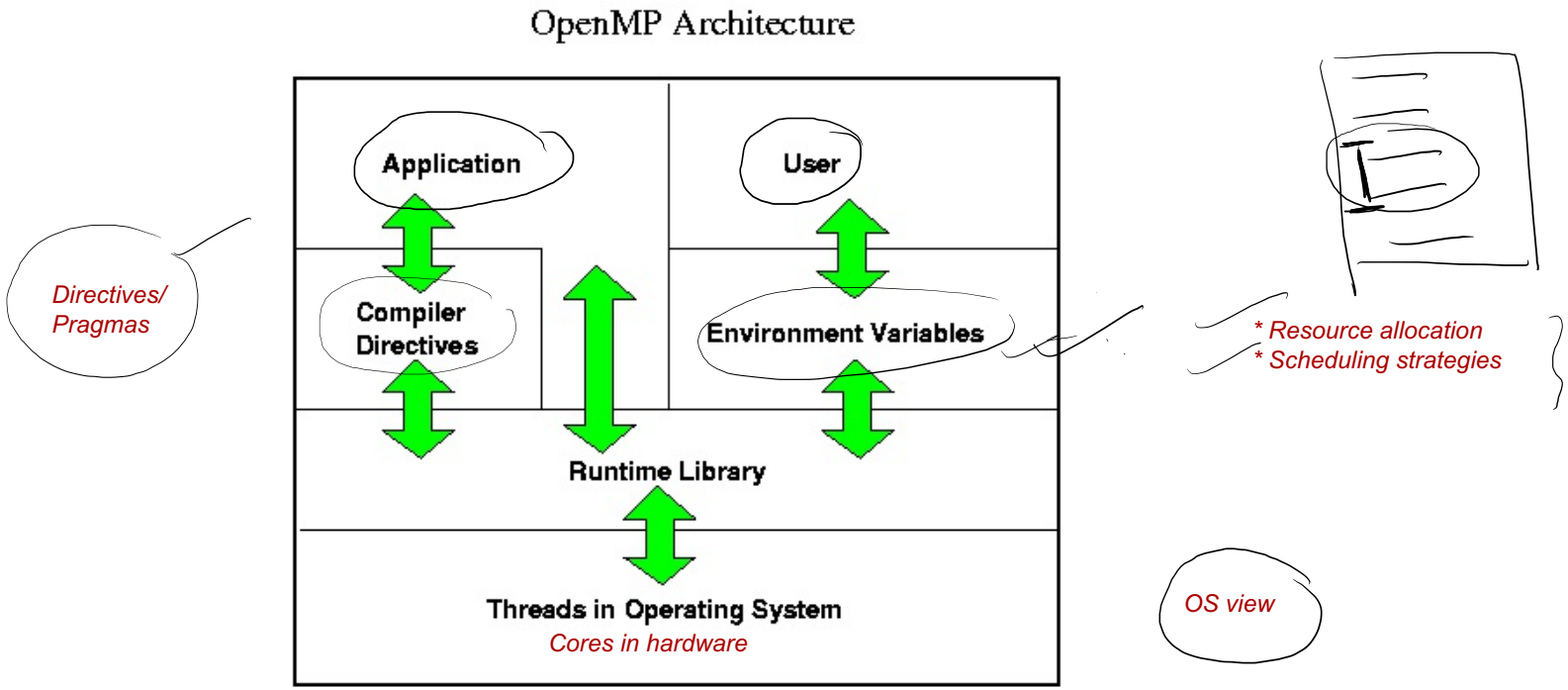


- Shared Memory with thread based parallelism
- Not a language
- Compiler directives, library calls and environment variables extend the base language
  - *f77, f90, f95, C, C++*
- Not automatic parallelization
  - *user explicitly specifies parallel execution*
  - *compiler does not ignore user directives even if wrong*










From the OS point of view, OpenMP functionality is based on the use of threads. Programmers job simply consists in inserting suitable parallelization directives into code.
These directives should not influence the sequential functionality of the code.
An OpenMP aware compiler is capable of transforming the code-blocks marked by OpenMP directives into threaded code; at run time the user can then decide (by setting suitable environment variables) what resources should be made available to the parallel parts of his executable, and how they are organized or scheduled.

# What is a thread?



- A thread is an independent instruction stream, thus allowing concurrent operation
- threads tend to share state and memory information and may have some private data
- Threads are usually lighter weight allowing faster context switching
- in OpenMP usually wants no more than one thread per core

# Execution Model



- OpenMP program starts single threaded
- To create additional threads, user starts a parallel region
  - *additional threads are launched to create a team*
  - *original (master) thread is part of the team*
  - *threads "go away" at the end of the parallel region: usually sleep*
- Repeat parallel regions as necessary
  - ***Fork-join model***

# Storage Model – Data Scoping



- Shared memory programming model: variables are shared by default
- Global variables are **SHARED** among threads
- Private Variables:
  - *exist only within the new scope, i.e. they are uninitialized and undefined outside the data scope*
  - *loop index variables*
  - *Stack variables in sub-programs called from parallel regions*

# Creating Parallel Regions



- Only one way to create threads in OpenMP API:

- Fortran:

```
!$OMP parallel
    < code to be executed in parallel >
!$OMP end parallel
```

- C

```
#pragma omp parallel
{
    code to be executed by each thread
}
```

# Compiling



Intel (icc, ifort)

- *-openmp*

GNU (gcc, gfortran, g++)

- *-fopenmp*

# Compiler Directives



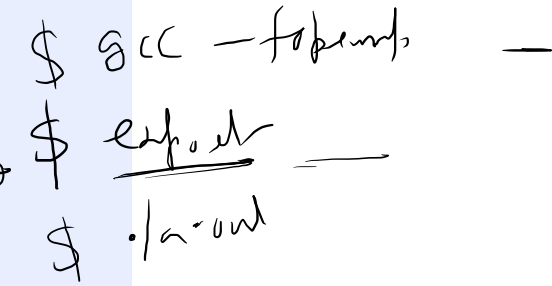
C Pragmas

- C pragmas are case sensitive
- Use curly braces, {}, to enclose parallel regions

# Specifying threads



- The simplest way to specify the number of threads used on a parallel region is to set the environment variable (in the shell where the program is executing)
  - *OMP_NUM_THREADS*
- For example
  - *setenv OMP_NUM_THREADS 4*
- In bash
  - *export OMP_NUM_THREADS=4*
- Also There are other ways to specify this



# OpenMP Features



**Most of the constructs in OpenMP are compiler directives.**

**Example :** ***#pragma omp parallel num_threads(4)***

**Function prototypes and types in the file:** **#include <omp.h>**

- Compiler Directives – 3 categories
  - *Control Constructs*
    - parallel regions, distribute work
  - *Data Scoping for Control Constructs*
    - control shared and private attributes
  - *Synchronization*
    - barriers, atomic, …
- Runtime Control
  - *Environment Variables*
    - OMP_NUM_THREADS
  - *Library Calls*
    - OMP_SET_NUM_THREADS(…)

# example

Write a program that prints “hello world”

```
void main()
{
int ID = 0;
printf(“ hello(%d) ”, ID);
printf(“ world(%d) \n”, ID);
}
```

hello 0
world

# To Verify that OpenMP environment works

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

```
#include "omp.h"
void main()
{
#pragma omp parallel
{
int ID = 0;
printf(" hello(%d) ", ID);
printf(" world(%d) \n", ID);
}}
```

**Switches for compiling and linking**

**-fopenmp gcc**

$ export OMP_ =4

0, 1, 2, 3

```
#include “omp.h”
void 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

End of the Parallel region

Runtime library function to return a thread ID.

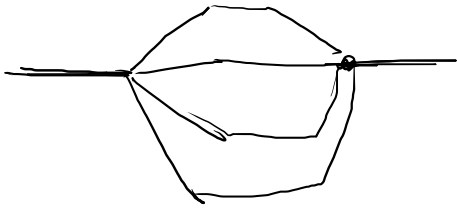
**Sample Output:**

hello(1) hello(0) world(1)

world(0)

hello (3) hello(2) world(3)

world(2)



# Thread Creation: Parallel Regions

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

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

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

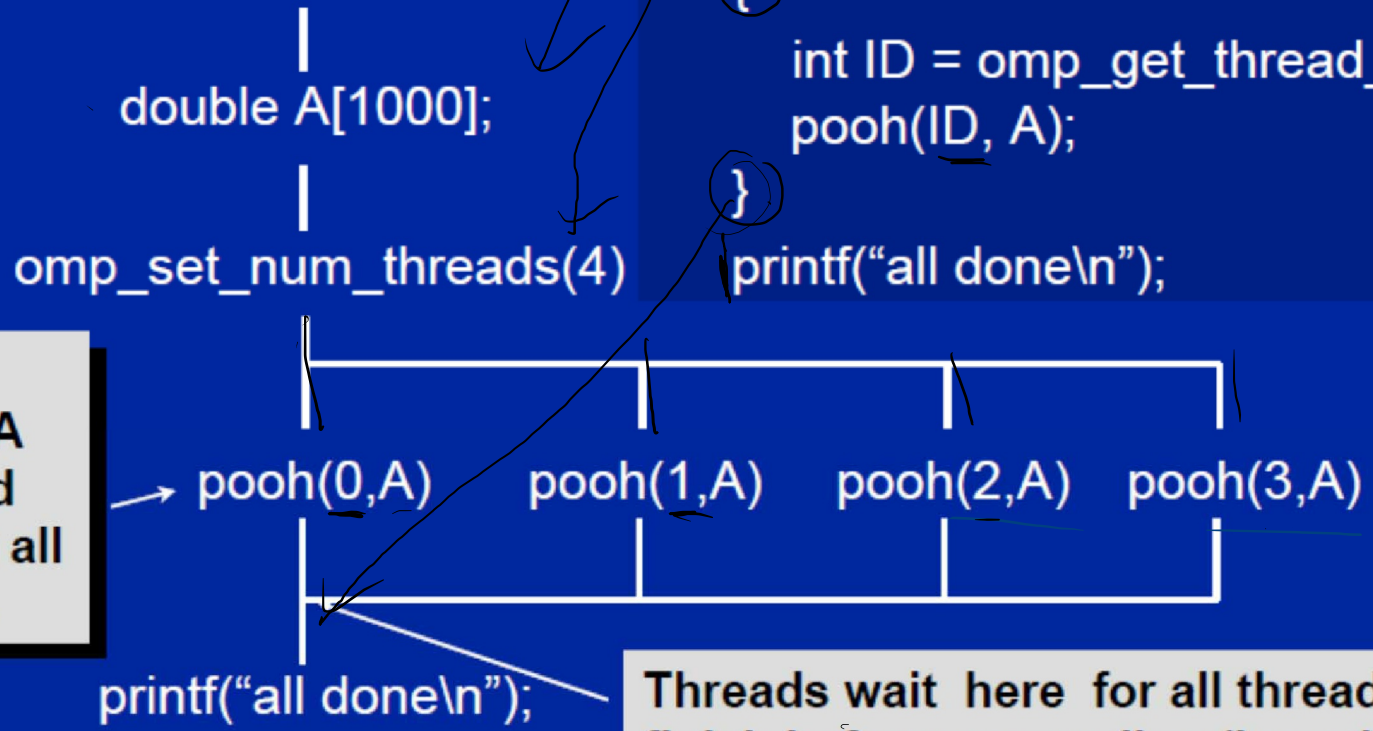
Runtime function to request a certain number of threads

Runtime function returning a thread ID

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

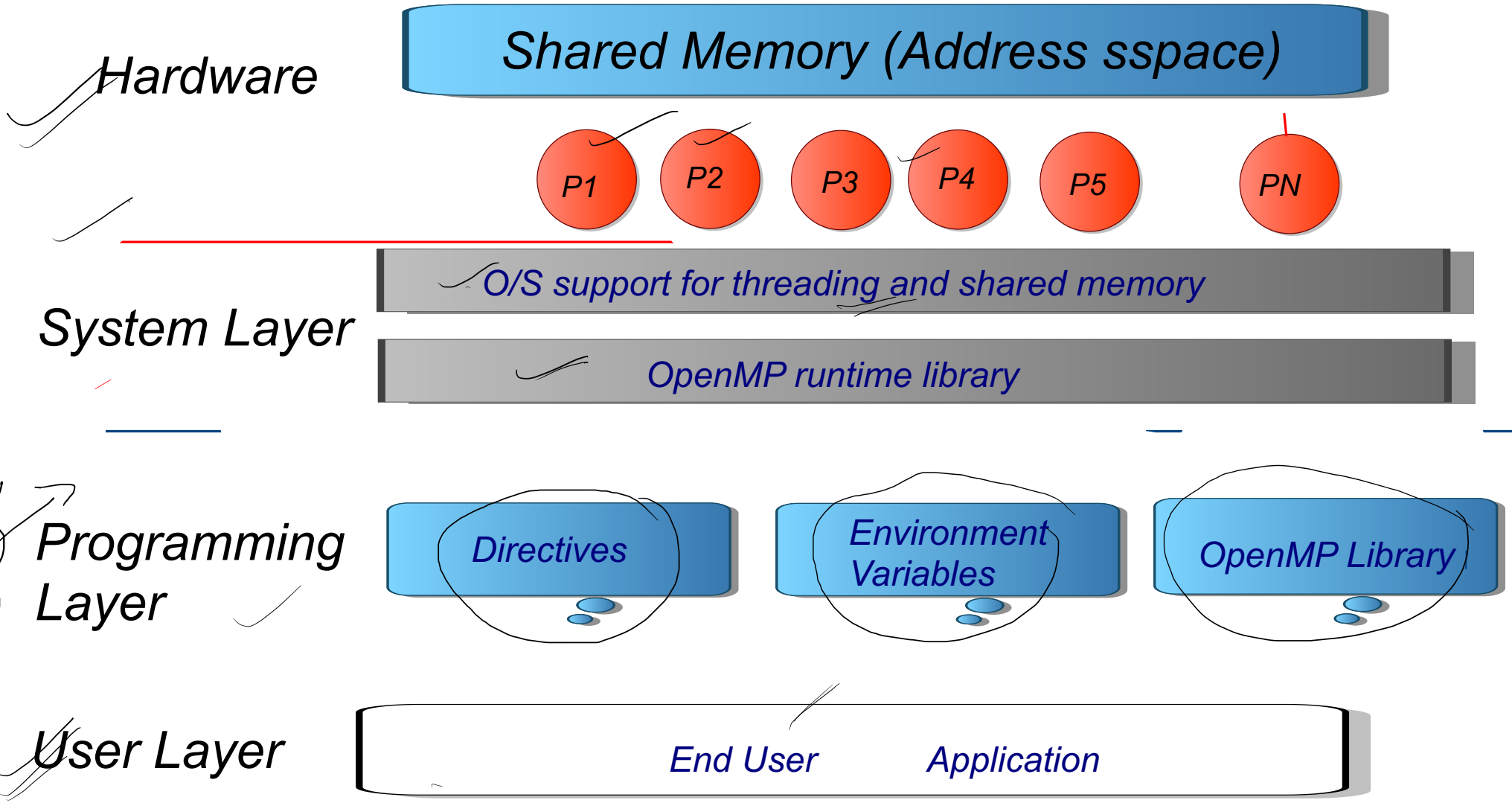
- **Each thread executes the same code redundantly.**

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




A single copy of A is shared between all threads.

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

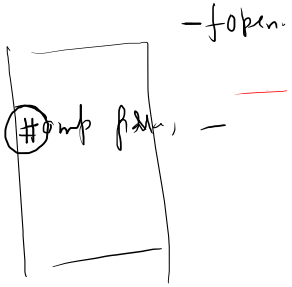

Hardware
Shared Memory (Address sspace)
P1
P2
P3
P4
P5
PN
System Layer
O/S support for threading and shared memory
OpenMP runtime library
Programming Layer
Directives
Environment Variables
OpenMP Library
User Layer
End User
Application

# Directives

- A program which processes the source code before it passes through the compiler is known as preprocessor.
- The commands of the preprocessor are known as preprocessor directives. It begins with a # symbol.

- Compiler Directives: Compiler directives appear as comments in the source code and are ignored by compilers unless told otherwise - usually by specifying the appropriate compiler flag.

- OpenMP compiler directives are used for various purposes:
  - Spawning a parallel region
  - Dividing blocks of code among threads
  - Distributing loop iterations between threads
  - Serializing sections of code
  - Synchronization of work among threads



- Compiler directives have the following syntax:

```
sentinel   directive-name   [clause, ...]
```

*OpenMP directives exploit shared memory parallelism by defining various types of parallel regions*

1. *The `#pragma omp` pragmas for parallel regions in which work is done by threads in parallel (`#pragma omp parallel`).*

2. *The `#pragma omp` pragmas to define work is distributed or shared across the threads in a parallel region (`#pragma omp sections`, `#pragma omp for` `#pragma omp single`, `#pragma omp task`).*

3. *The `#pragma omp` pragmas to control synchronization among threads (`#pragma omp atomic`, `#pragma omp master`, `#pragma omp barrier`, `#pragma omp critical`, `#pragma omp flush`, `#pragma omp ordered`) .*

4. *The `#pragma omp` pragmas that let you define the scope of data visibility across parallel regions within the same thread (`#pragma omp threadprivate`).*

*Adding certain clauses to the `#pragma omp` pragmas can fine tune the behavior of the parallel or work-sharing regions.*

## Run-time Library Routines:

- The OpenMP API includes an ever-growing number of run-time library routines.
- These routines are used for a variety of purposes:
  - Setting and querying the number of threads
  - Querying a thread's unique identifier (thread ID), a thread's ancestor's identifier, the thread team size
  - Setting and querying the dynamic threads feature
  - Querying if in a parallel region, and at what level
  - Setting and querying nested parallelism
  - Setting, initializing and terminating locks and nested locks
  - Querying wall clock time and resolution

# Environment Variables:

- OpenMP provides several environment variables for controlling the execution of parallel code at run-time.
- These environment variables can be used to control such things as:
  - Setting the number of threads
  - Specifying how loop iterations are divided
  - Binding threads to processors
  - Enabling/disabling nested parallelism; setting the maximum levels of nested parallelism
  - Enabling/disabling dynamic threads
  - Setting thread stack size
  - Setting thread wait policy

# First example

```
#include <stdio.h>
#include <omp.h>
int main()
{
  #pragma omp parallel
  {
    int me = omp_get_thread_num();
    int nthr = omp_get_num_threads();
    printf("Hello from thread %d of %d\n", me, nthr);
  }
}
```

Compiler directive

Library calls

Parallel region

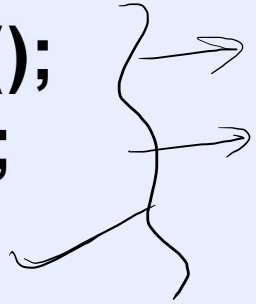
- Compile
  - gcc **-fopenmp** hello.c
- Execute with 4 threads:
  - export **OMP_NUM_THREADS=4**
  - ./a.out

Environment variable

# Assignment : Create a parallel version of the pi program

**Runtime library routines required**

```
int omp_get_num_threads();
int omp_get_thread_num();
double omp_get_wtime();
```



# Thread Creation: Parallel Regions

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

```
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

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

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:

```
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

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

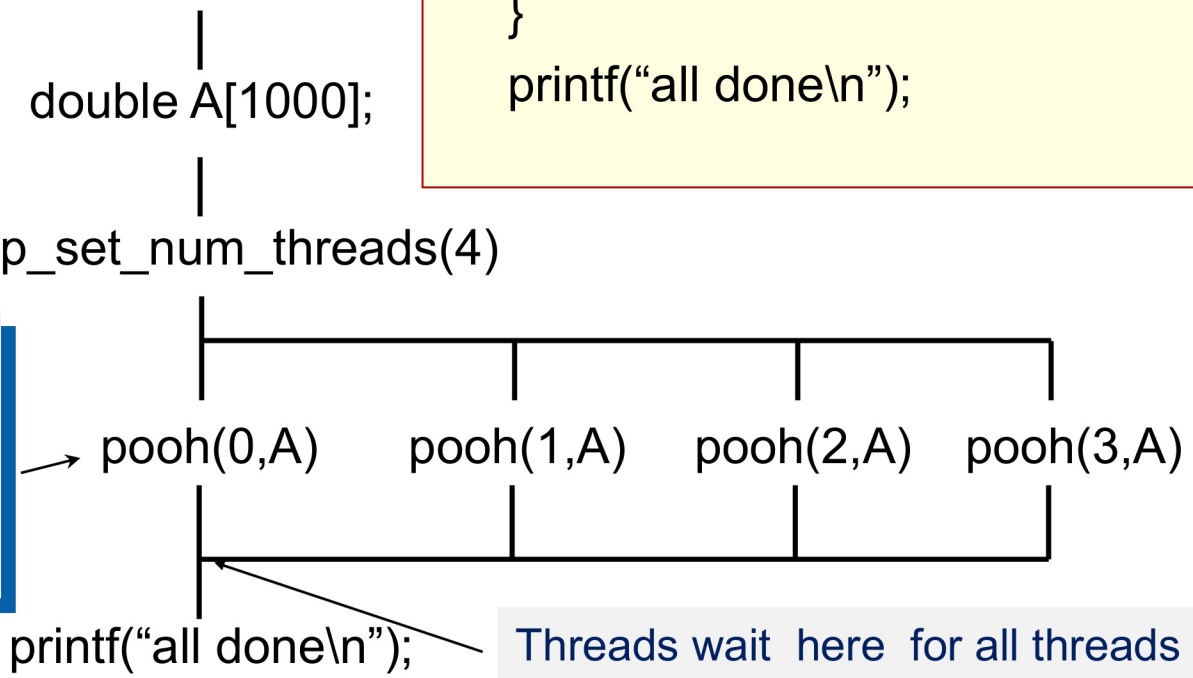
Runtime function returning a thread ID

- 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");
```




A single copy of A is shared between all threads.

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

# Assignment – calculation of PI

Numerical integration

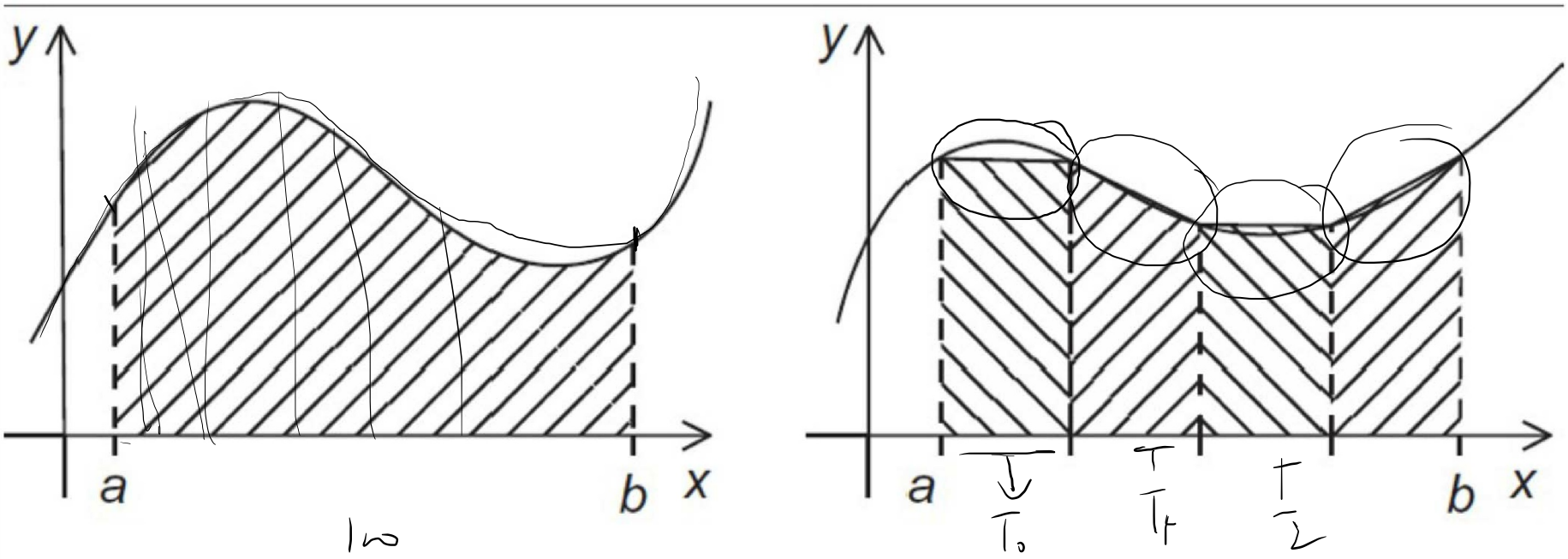
Divided into several rectangles
Each rectangle has a width and height.

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

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

# Second example – Integration :Trapezoidal rule

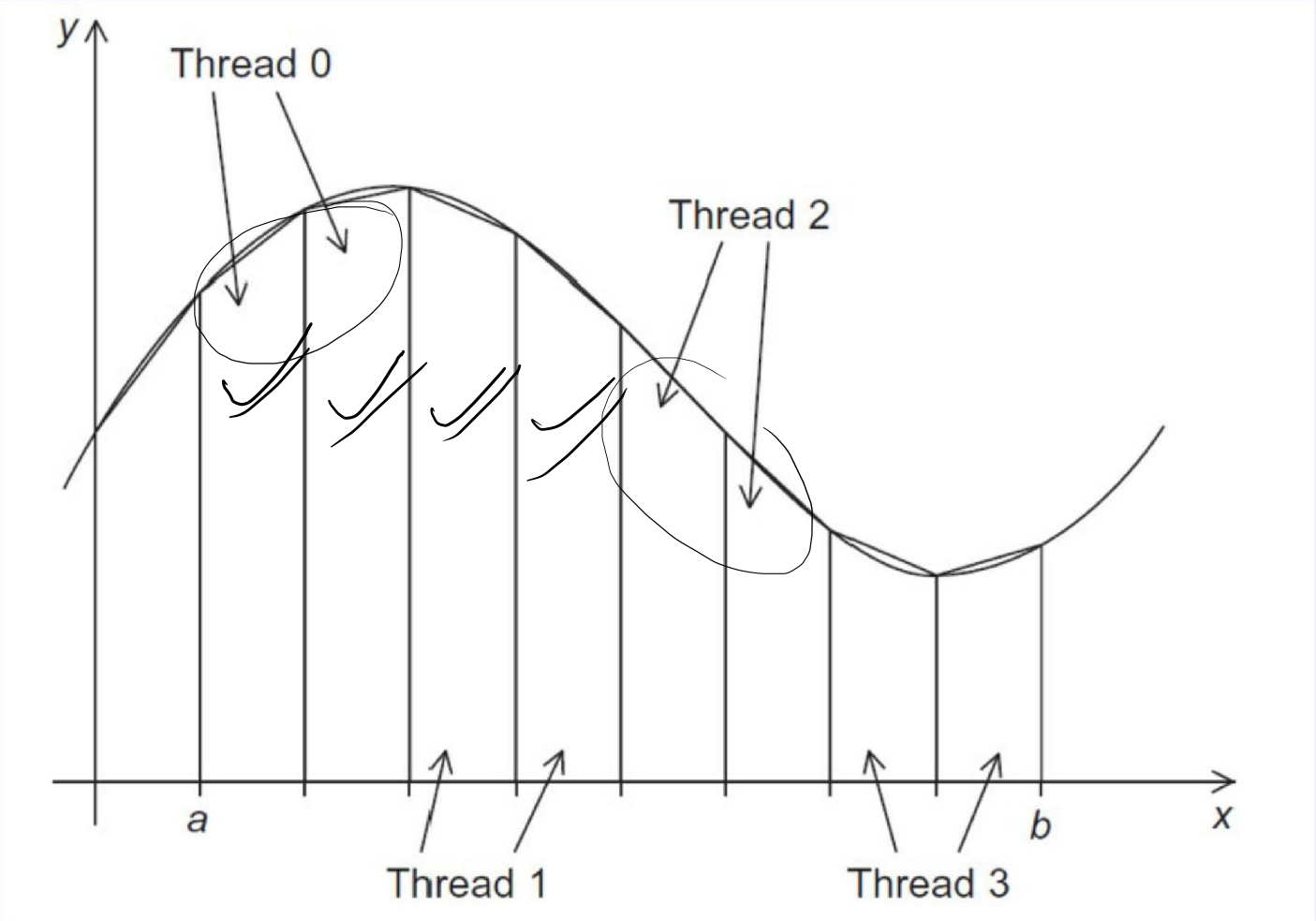





**Calculation of pi !!**

# Assigning the threads to trapezoid






Trapezoids >> core

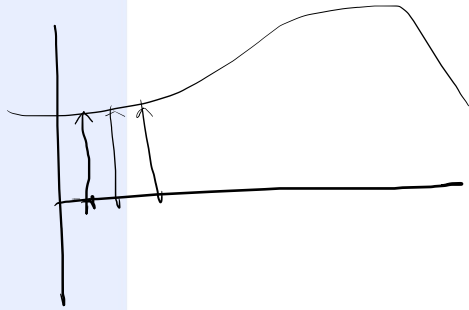
Communication → adding the areas of trapezoids from each thread

# Serial code

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



# Assignment

- 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();
  - int omp_get_thread_num();
  - double omp_get_wtime();

Number of threads in the team

Thread ID or rank

Time in Seconds since a fixed point in the past

# Serial PI Program

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

        step = 1.0/(double) num_steps;

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

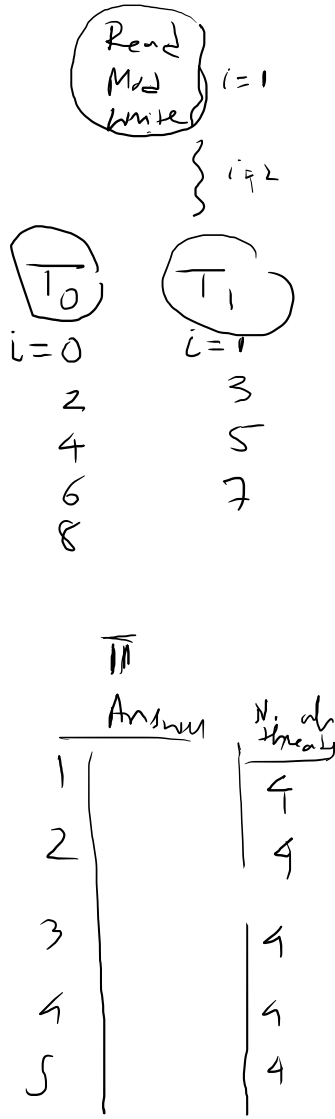
# Example: A simple Parallel pi program

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

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

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

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



# Algorithm strategy:

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

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

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

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

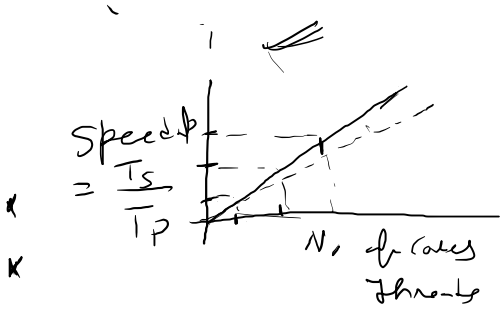
# Results*

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

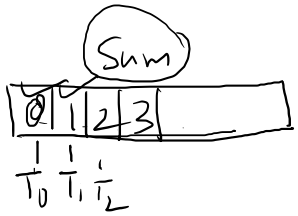
## Example: A simple Parallel pi program

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

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



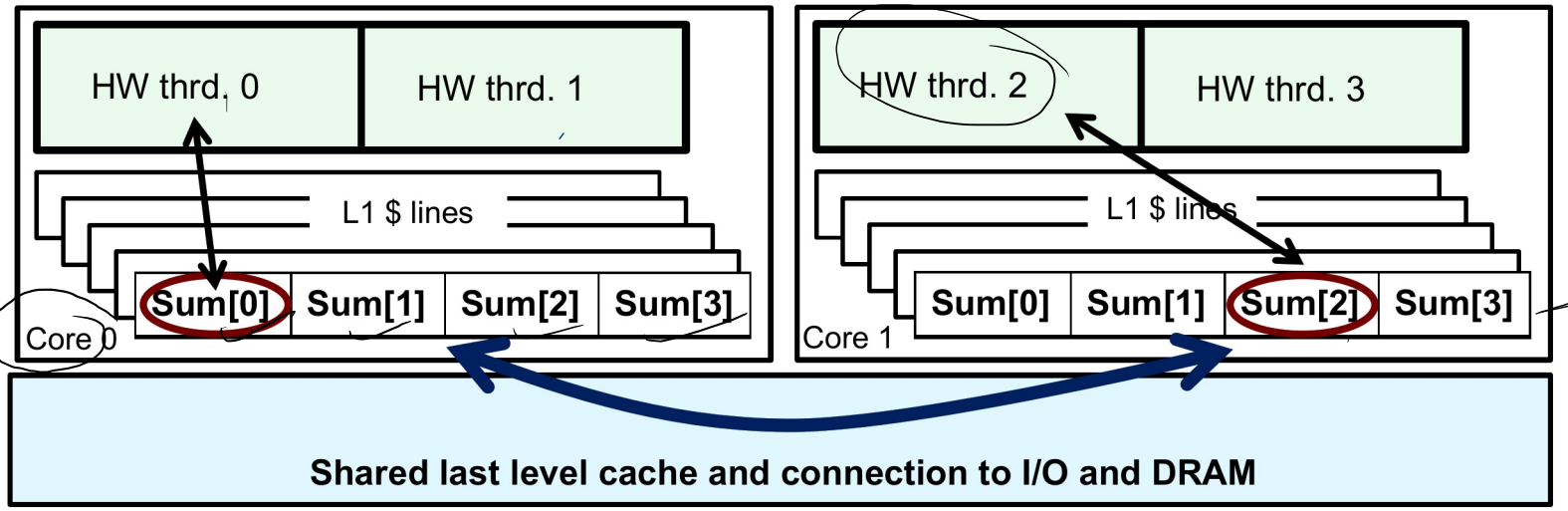

Not Scalable




*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"**.



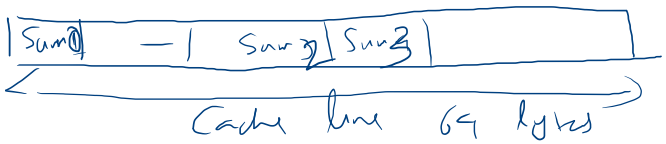

- If you promote scalars to an array to support creation of an SPMD program, the array elements are contiguous in memory and hence share cache lines … Results in poor scalability.
- Solution: Pad arrays so elements you use are on distinct cache lines.

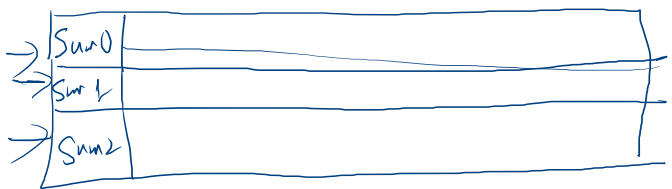
# Example: eliminate False sharing by padding the sum array

```
#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, nthreads;  double pi, sum[NUM_THREADS][PAD};
           step = 1.0/(double) num_steps;
           omp_set_num_threads(NUM_THREADS);
   #pragma omp parallel
   {
           int i, id,nthrds;
          double x;
          id = omp_get_thread_num();
          nthrds = omp_get_num_threads();
          if (id == 0)   nthreads = nthrds;
           for (i=id, sum[id]=0.0;i< num_steps; i=i+nthrds) {
                    x = (i+0.5)*step;
                    sum[id][0] += 4.0/(1.0+x*x);
           }
   }
           for(i=0, pi=0.0;i<nthreads;i++)pi += sum[i][0] * step;
}
```

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

*Intel compiler (icpc) with 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.

✗ Scalable

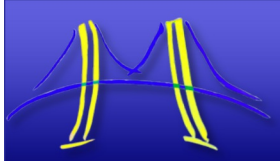
✗ Portable ✗

# OpenMP Overview:
## How do threads interact?

**Recall our high level overview of OpenMP?**

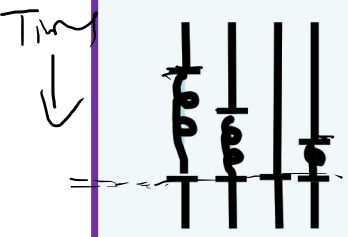
- OpenMP is a multi-threading, shared address model.
  - Threads communicate by sharing variables.
- Unintended sharing of data causes race conditions:
  - race condition: when the program's outcome changes as the threads are scheduled differently.
- To control race conditions:
  - Use synchronization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

sum[id]

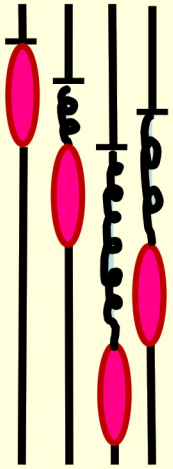


# Synchronization:

- 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

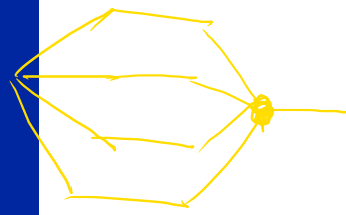
Discussed later

# 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: 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 binop= 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.

# Pi program with false sharing*

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

## Example: A simple Parallel pi program

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

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

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

*Intel compiler (icpc) with 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)   nthreads = 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)   nthreads = 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;
}
}
```

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

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

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

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

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

No array, so no false sharing.

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

# Components of OpenMP



| Directives |
| --- |
| Parallel regions |
| Work sharing |
| Synchronization |
| Data scope attributes :<br>• private<br>• firstprivate<br>• last private<br>• shared<br>• reduction |
| |

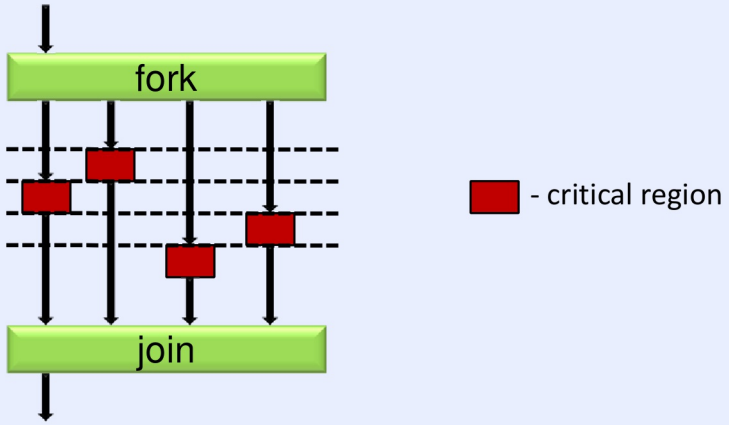
| Runtime library routines |
| --- |
| Number of threads |
| Thread ID |
| Dynamic thread adjustment |
| Nested Parallelism |
| Timers |
| API for locking |

| Environment variables |
| --- |
| Number of threads |
| Scheduling type |
| thread adjustment |
| Nested Parallelism |

# OpenMP *critical* directive : Explicit Synchronization

- Race conditions can be avoided by controlling access to shared variables by allowing threads to have exclusive access to the variables
- *Mutual exclusion* synchronization is provided by the *critical* directive of OpenMP
- Code block within the *critical region* defined by critical /end critical directives can be executed only by one thread at a time.
- Other threads in the group must wait until the current thread exits the critical region. Thus only one thread can manipulate values in the critical region.

```
int x
x=0;
#pragma omp parallel shared(x)
{
  #pragma omp critical
      x = 2*x + 1;
} /* omp end parallel */
```



# 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)   nthreads = nthrds;
                id = omp_get_thread_num();
              nthrds = omp_get_num_threads();
                for (i=id, sum=0.0;i< num_steps; i=i+nthreads){
                              x = (i+0.5)*step;
                              sum += 4.0/(1.0+x*x);
                }
                 sum = sum*step;
            #pragma atomic
                        pi += sum ;
}
```

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

No array, so no false sharing.

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

# 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**

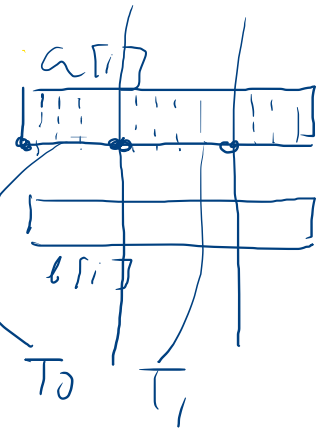
```
for(i=0;i<N;i++)  { a[i] = a[i] + b[i];}
```

**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

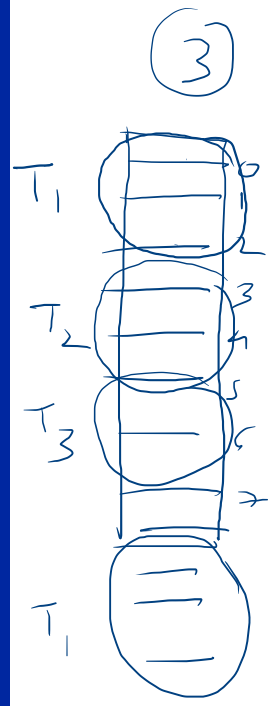
- **The schedule clause affects how loop iterations are mapped onto threads**
  - schedule(static [,chunk])
    - Deal-out blocks of iterations of size “chunk” to each thread.
  - schedule(dynamic[,chunk])
    - Each thread grabs “chunk” iterations off a queue until all iterations have been handled.
  - 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 |
|---|---|
| STATIC | Pre-determined and predictable by the programmer |
| DYNAMIC | Unpredictable, highly variable work per iteration |
| GUIDED | Special case of dynamic to reduce scheduling overhead |
| AUTO | When the runtime can "learn" from previous executions of the same loop |

Least work at runtime : scheduling done at compile-time

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



# Combined parallel/worksharing construct

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

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

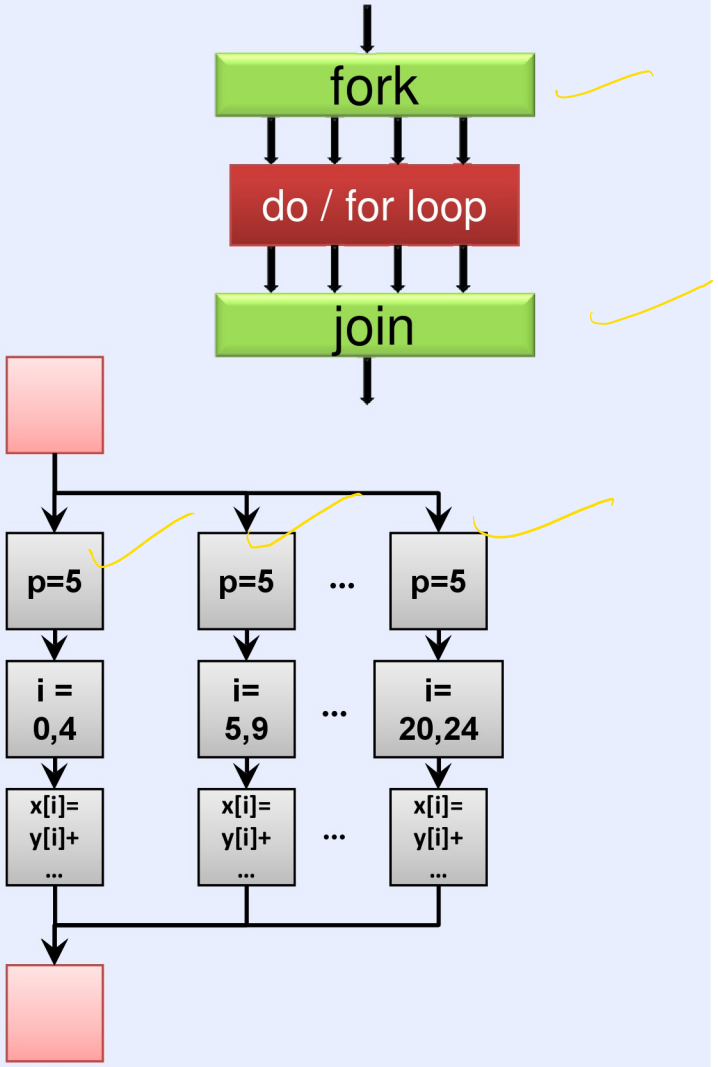
```
double res[MAX]; int i;
#pragma omp parallel for
  for (i=0;i< MAX; i++) {
    res[i] = huge();
  }
```

These are equivalent

# OpenMP *for* directive

- *for* directive helps share iterations of a loop between a group of threads
- If *nowait* is specified then the threads do not wait for synchronization at the end of a parallel loop
- The *schedule* clause describes how iterations of a loop are divided among the threads in the team (discussed in detail in the next few slides)

```
#pragma omp parallel
{
  p=5;
  #pragma omp for
     for (i=0; i<24; i++)
         x[i]=y[i]+p*(i+3)
      …
  …
} /* omp end parallel */
```



# Example: OpenMP work sharing Constructs



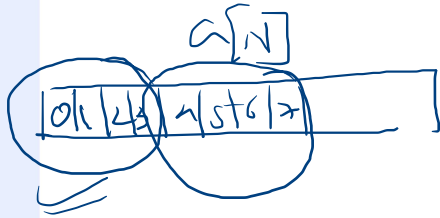
```
#include <omp.h>
#define N    16
main ()
{
int i, chunk;
float a[N], b[N], c[N];
for (i=0; i < N; i++)
  a[i] = b[i] = i * 1.0;
chunk = 4;
printf("a[i]  +  b[i]  =  c[i] \n");
#pragma omp parallel shared(a,b,c,chunk) private(i)
  {
  #pragma omp for schedule(dynamic,chunk) nowait
  for (i=0; i < N; i++)
    c[i] = a[i] + b[i];
  } /* end of parallel section */
for (i=0; i < N; i++)
   printf(" %f  +  %f  =  %f  \n",a[i],b[i],c[i]);
}
```

Initializing the vectors a[i], b[i]

Instructing the runtime environment that a,b,c,chunk are shared variables and I is a private variable

The nowait ensures that the child threads donot synchronize once their work is completed

Load balancing the threads using a DYNAMIC policy where array is divided into chunks of 4 and assigned to the threads



# 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

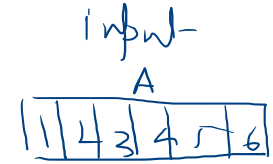
- Will form a single loop of length NxM and then parallelize that.
- Useful if N is O(no. of threads) so parallelizing the outer loop makes balancing the load difficult.

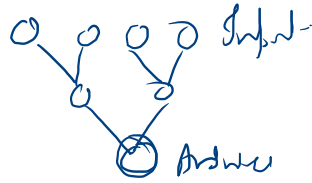
# Reduction

- 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.





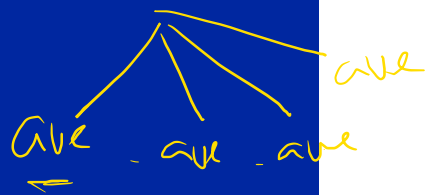
# Reduction

- 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;
```

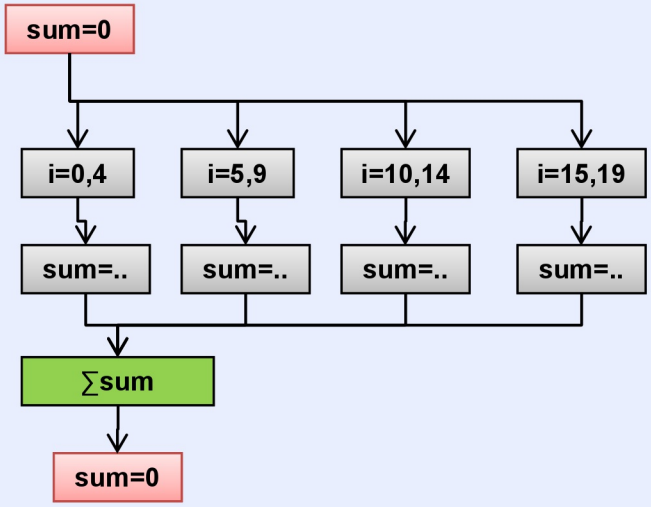


# OpenMP : Reduction



- performs reduction on *shared variables* in list based on the operator provided.
- for C/C++ operator can be any one of :
  - *+, *, -, ^, |, ||, & or &&*
  - *At the end of a reduction, the shared variable contains the result obtained upon combination of the list of variables processed using the operator specified.*

```
sum = 0.0
#pragma omp parallel for reduction(+:sum)
  for (i=0; i < 20; i++)
    sum = sum + (a[i] * b[i]);
```



# 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. |

# 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 indix is local to a thread by default.

# 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();
}
```

# Single directive



**The enclosed code is executed by exactly one thread, which one is unspecified**

`#pragma omp single` *[clause[[,]clause]…] [nowait]*

- **The other threads in the team skip the enclosed section of code and continue execution. There is an implied barrier at the exit of the `single` section!**

- **`nowait` clause at start of parallel region suppresses synchronization**

# Sections worksharing Construct

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

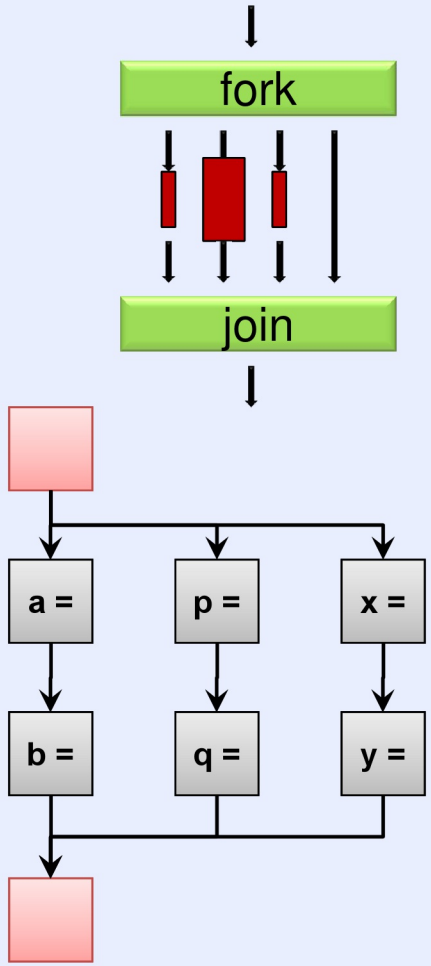
```
#pragma omp parallel
{
  #pragma omp sections
  {
  #pragma omp section
        X_calculation();
  #pragma omp section
        y_calculation();
  #pragma omp section
        z_calculation();
  }
}
```

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

# OpenMP *sections* directive

- *sections* directive is a non iterative work sharing construct.
- Independent *section* of code are nested within a *sections* directive
- It specifies enclosed *section* of codes between different threads
- Code enclosed within a *section* directive is executed by a thread within the pool of threads

```
#pragma omp parallel private(p)
{
#pragma omp sections
{{  a=…;
   b=…;}
 #pragma omp section
 {  p=…;
   q=…;}
 #pragma omp section
 {  x=…;
   y=…;}
 } /* omp end sections */
} /* omp end parallel */
```



# Synchronization: Lock routines

- **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()

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

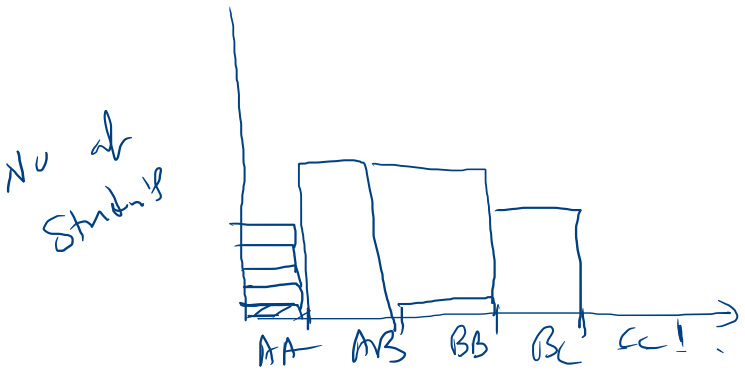
- **Nested Locks**
  - **A nested lock is available if it is unset or if it is set but owned by the thread executing the nested lock function**
    - omp_init_nest_lock(), omp_set_nest_lock(), omp_unset_nest_lock(), omp_test_nest_lock(), omp_destroy_nest_lock()

**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.**

# Histogram

Hist[No of bins]

Hist[5]




4 – 5 bins

N Students

for i=1, N

Marks

$T_1 \rightarrow i=1$

$T_2 = i=2$

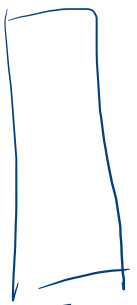
$T_3 = i=3$

AA →

AB →

BB →

BC



Sum = N

Histogram → Sum

# Synchronization: Simple Locks

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

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

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

One lock per element of hist

Enforce mutual exclusion on update to hist array

Free-up storage when done.

# 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**
    - **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**

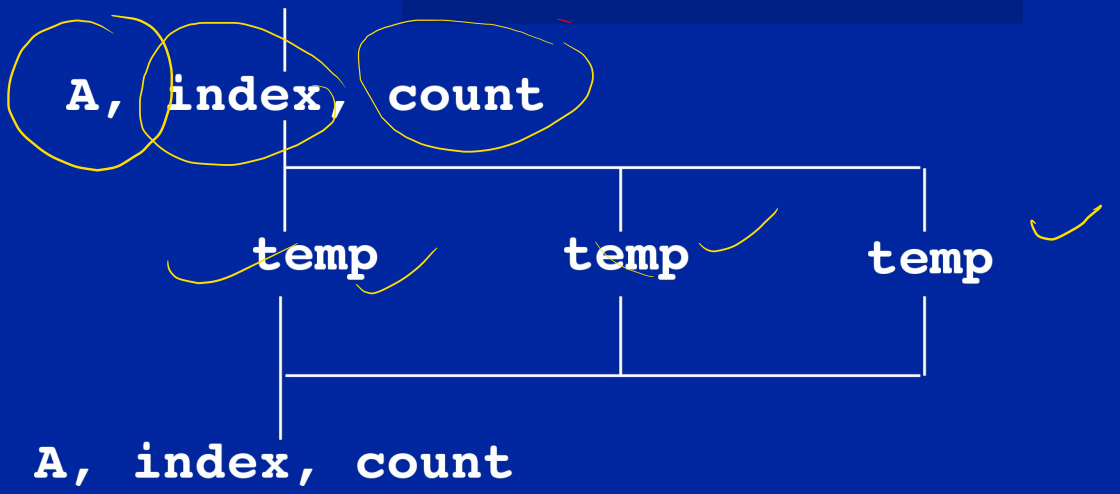
# Data environment:
## Default storage attributes

- Shared Memory programming model:
  - Most variables are shared by default
- Global variables are SHARED among threads
  - Fortran: COMMON blocks, SAVE variables, MODULE variables
  - C: File scope variables, static
  - Both: dynamically allocated memory (ALLOCATE, malloc, new)
- But not everything is shared...
  - Stack variables in subprograms(Fortran) or functions(C) called from parallel regions are PRIVATE
  - Automatic variables within a statement block are PRIVATE.

# Data sharing: Examples

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

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




**A, index and count are shared by all threads.**

**temp is local to each thread**

# Data sharing:
## Changing storage attributes

- One can selectively change storage attributes for constructs using the following clauses*
  - SHARED
  - PRIVATE
  - FIRSTPRIVATE

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

- 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 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

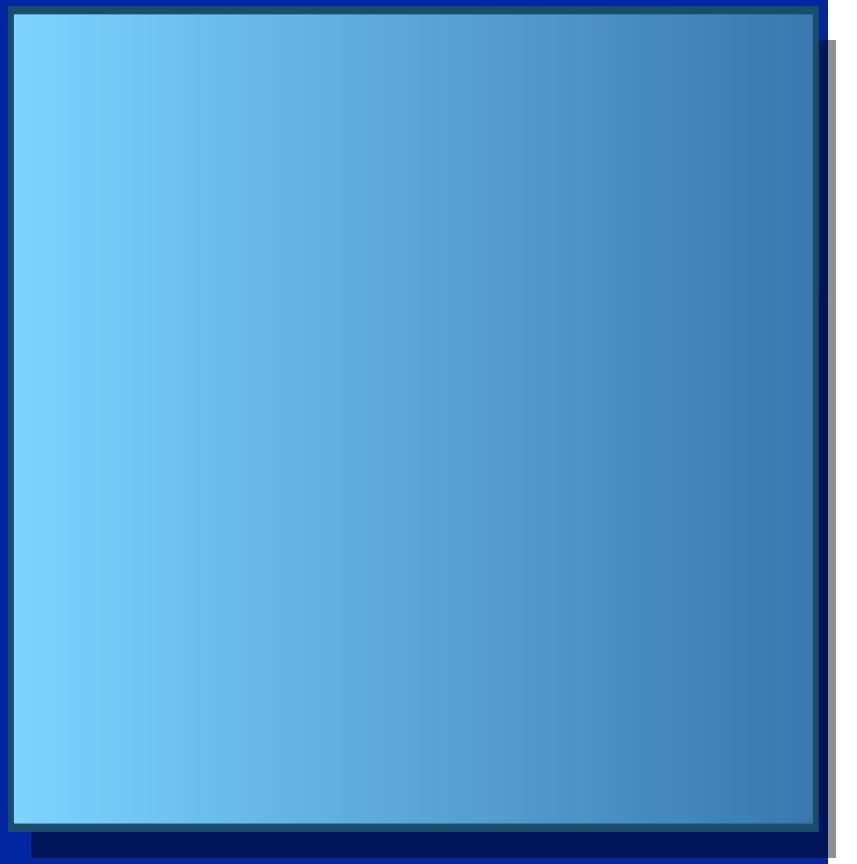
```
      itotal = 1000
C$OMP PARALLEL PRIVATE(np, each)
      np = omp_get_num_threads()
      each = itotal/np
      ……..
C$OMP END PARALLEL
```

```
      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**

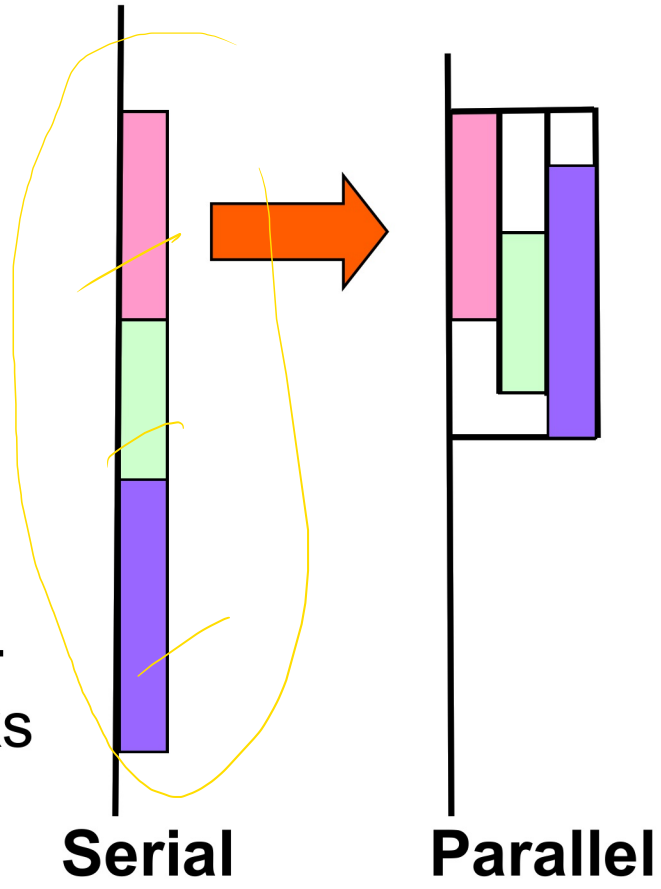
# Major OpenMP constructs we've covered so far

- 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)



# 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

# Strategies to Improve Performance



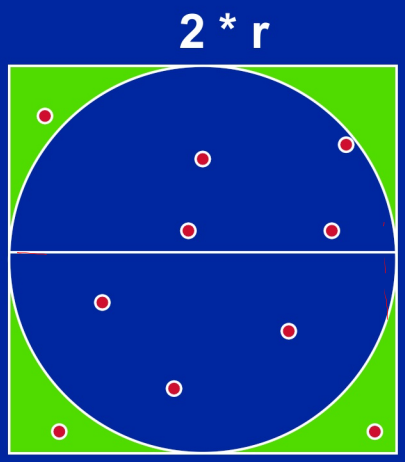
- **parallel startup costs (incurred at parallel directive)**
- **avoid parallelizing small loops**
- **watch for load imbalances (threads have different amounts of work)**
- **unnecessary synchronization (for example critical or ordered directives)**
- **non-cache friendly programs**
  - memory contention –
  - false sharing –
  - use private variables where possible
  - change size and arrangement of arrays to avoid cache misses

**Barrier, single, critical**

# 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 π with a digital dart board:




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

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

  $A_c = r^2 * \pi$

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

  $P = A_c/A_s = \pi /4$
- Compute π by randomly choosing points, count the fraction that falls in the circle, compute pi.

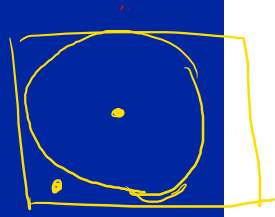
# Parallel Programmers love Monte Carlo algorithms

**Embarrassingly parallel: the parallelism is so easy its embarrassing.**

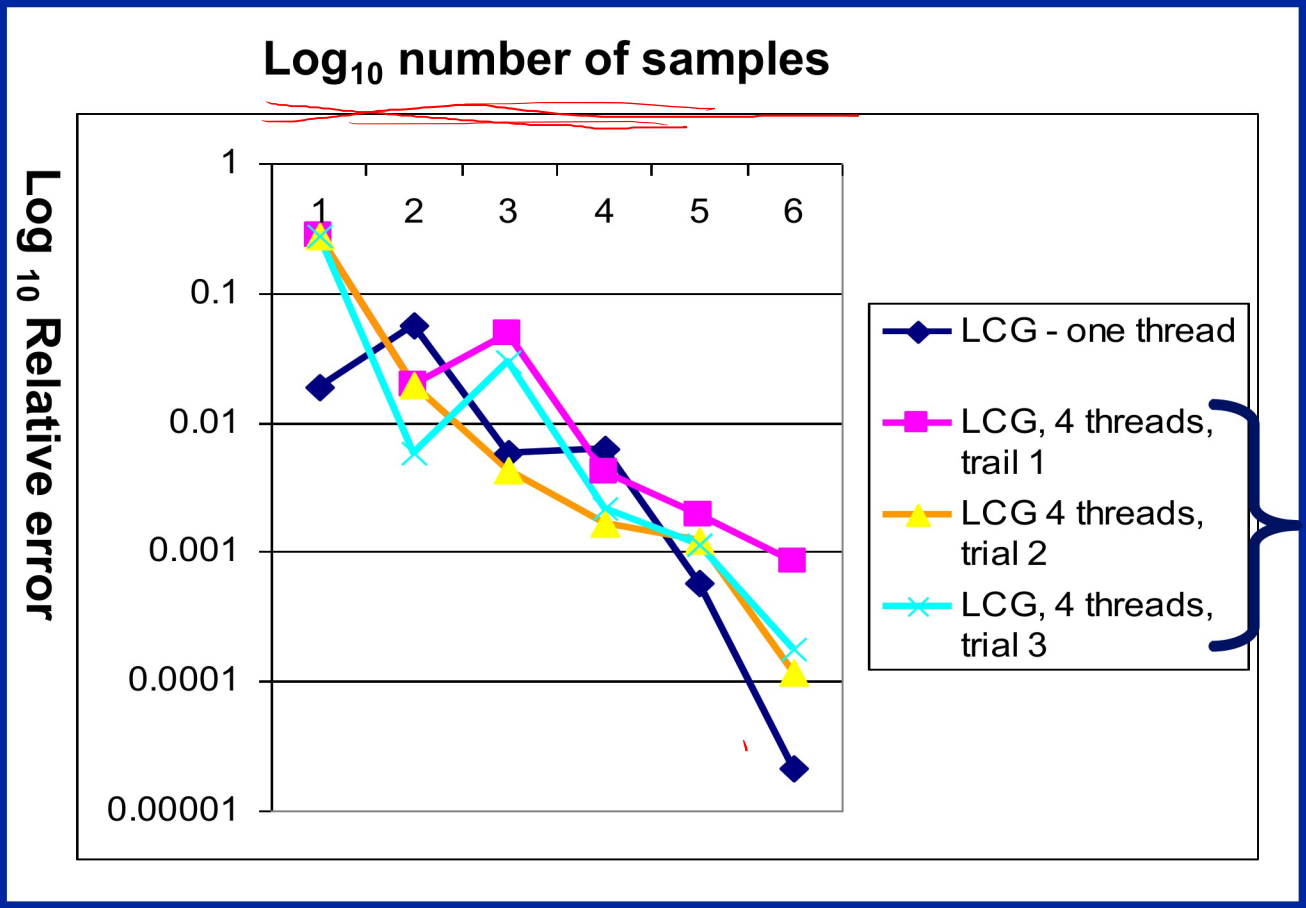
**Add two lines and you have a parallel program.**

```
#include “omp.h”
static long num_trials = 10000;
int main ()
{
   long i;     long Ncirc = 0;      double pi, x, y;
   double r = 1.0;   // radius of circle. Side of squrare is 2*r
   seed(0,-r, r);  // The circle and square are centered at the origin
   #pragma omp parallel for private (x, y) reduction (+:Ncirc)
   for(i=0;i<num_trials; i++)
   {
     x = random();        y = random();
     if ( x*x + y*y) <= r*r)   Ncirc++;
    }

    pi = 4.0 * ((double)Ncirc/(double)num_trials);
    printf("\n %d trials, pi is %f \n",num_trials, pi);
}
```



# Running the PI_MC program with LCG generator




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

**That is not acceptable!**

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

Program written using the Intel C/C++ compiler (10.0.659.2005) in Microsoft Visual studio 2005 (8.0.50727.42) and running on a dual-core laptop (Intel T2400 @ 1.83 Ghz with 2 GB RAM) running Microsoft Windows XP.

# 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);
}
```

# References for openMP

**https://computing.llnl.gov/tutorials/openMP/**

**Youtube:** Introduction to OpenMP - Tim Mattson (Intel)