

---

# Shared Memory Programming: (mostly) OpenMP

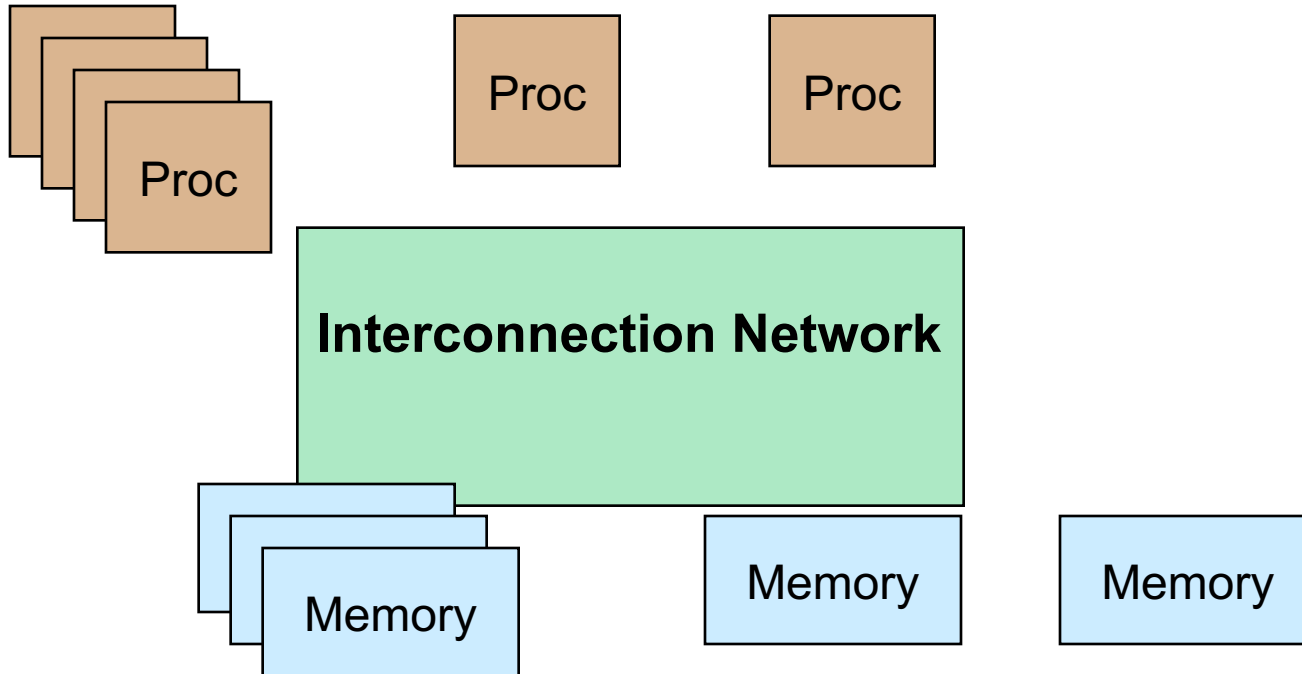
## Lecture 3

Aydin Buluc

(thanks to many slides from Tim Mattson)

<https://sites.google.com/lbl.gov/cs267-spr2020/>


# A generic parallel architecture



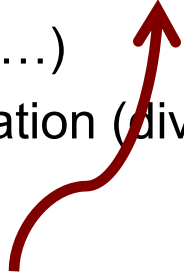
- **Where is the memory physically located?**
- **Is it connected directly to processors?**
- **What is the connectivity of the network?**

# A Brief History of Parallel Languages

- When vector machines were king
  - Parallel “languages” were loop annotations (IVDEP)
  - Performance was fragile, but good user support
- When SIMD machines were king
  - Data parallel languages popular and successful (CMF, \*Lisp, C\*, ...)
  - Irregular data (sparse mat-vec multiply OK), but irregular computation (divide and conquer, adaptive meshes, etc.) less clear
- When shared memory multiprocessors (SMPs) were king
  - Shared memory models, e.g., Posix Threads, OpenMP, are popular
- When clusters took over
  - Message Passing (MPI) became dominant
- With the addition of accelerators
  - OpenACC, CUDA were added
- In Cloud Computing
  - Hadoop, SPARK, ...



Mapping to  
MPPs/Clusters  
proved hard



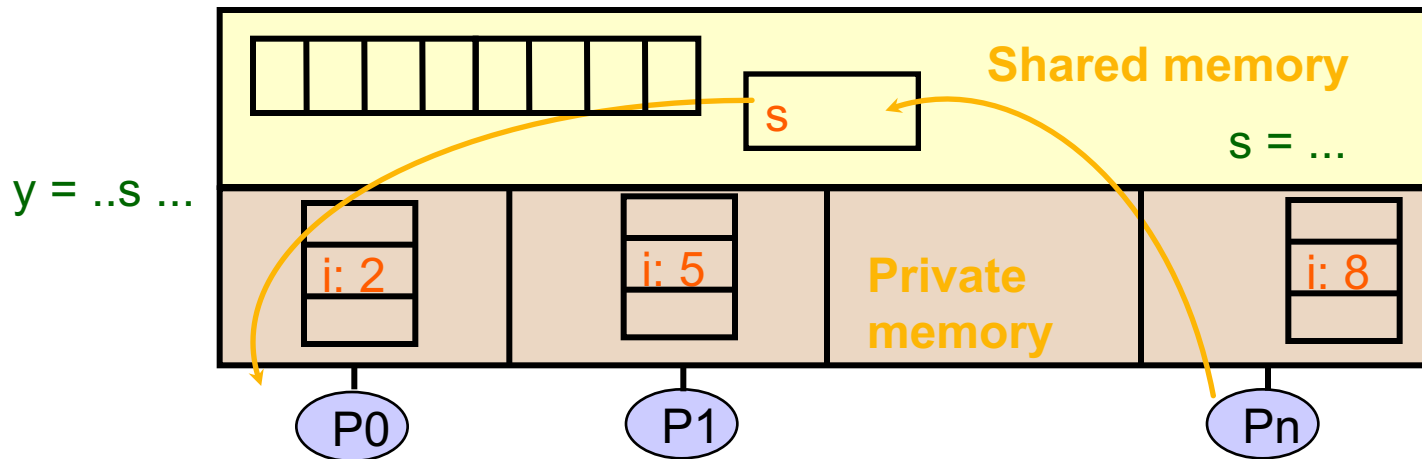
You'll see the  
most popular in  
each category

# Outline

- Shared memory parallelism with threads
- What and why OpenMP?
- Parallel programming with OpenMP
- Introduction to OpenMP
  1. Creating parallelism
  2. Parallel Loops
  3. Synchronizing
  4. Data sharing
- Beneath the hood
  - Shared memory hardware
- Summary

# Recall Programming Model 1: Shared Memory

- Program is a collection of threads of control.
  - Can be created dynamically, mid-execution, in some languages
- Each thread has a set of **private variables**, e.g., local stack variables
- Also a set of **shared variables**, e.g., static variables, shared common blocks, or global heap.
  - Threads communicate **implicitly** by writing and reading shared variables.
  - Threads coordinate by **synchronizing** on shared variables



---

# **Parallel Programming with Threads**

# Overview of POSIX Threads

- POSIX: *Portable Operating System Interface*
  - Interface to Operating System utilities
- PThreads: The POSIX threading interface
  - System calls to create and synchronize threads
  - Should be relatively uniform across UNIX-like OS platforms
- PThreads contain support for
  - Creating parallelism
  - Synchronizing
  - No explicit support for communication, because shared memory is implicit; a pointer to shared data is passed to a thread

# Forking Posix Threads

Signature:

```
int pthread_create(pthread_t *,
                  const pthread_attr_t *,
                  void * (*)(void *),
                  void *);
```

Example call:

```
errcode = pthread_create(&thread_id; &thread_attribute
                        &thread_fun; &fun_arg);
```

- **thread\_id** is the thread id or handle (used to halt, etc.)
- **thread\_attribute** various attributes
  - Standard default values obtained by passing a NULL pointer
  - Sample attributes: minimum stack size, priority
- **thread\_fun** the function to be run (takes and returns void\*)
- **fun\_arg** an argument can be passed to thread\_fun when it starts
- **errorcode** will be set nonzero if the create operation fails



# “Simple” Threading Example

```
void* SayHello(void *foo) {  
    printf( "Hello, world!\n" );  
    return NULL;  
}
```

Compile using gcc -lpthread

```
int main() {  
    pthread_t threads[16];  
    int tn;  
    for(tn=0; tn<16; tn++) {  
        pthread_create(&threads[tn], NULL, SayHello, NULL);  
    }  
    for(tn=0; tn<16 ; tn++) {  
        pthread_join(threads[tn], NULL);  
    }  
    return 0;  
}
```

# Loop Level Parallelism

- Many scientific application have parallelism in loops
  - With threads:

```
... my_stuff [n][n];  
for (int i = 0; i < n; i++)  
    for (int j = 0; j < n; j++)  
        ... pthread_create (update_cell[i][j], ...,  
                             my_stuff[i][j]);
```
- But overhead of thread creation is nontrivial
  - update\_cell should have a significant amount of work
  - 1/p-th of total work if possible

# Recall Data Race Example

```
static int s = 0;
```

**Thread 1**

```
for i = 0, n/2-1  
  s = s + f(A[i])
```

**Thread 2**

```
for i = n/2, n-1  
  s = s + f(A[i])
```

- Problem is a race condition on variable `s` in the program
- A **race condition** or **data race** occurs when:
  - two processors (or two threads) access the same variable, and at least one does a write.
  - The accesses are concurrent (not synchronized) so they could happen simultaneously

# Basic Types of Synchronization: Mutexes

Mutexes -- mutual exclusion aka locks

- threads are working mostly independently
- need to access common data structure

```
lock *l = alloc_and_init();    /* shared */
acquire(l);
    access data
release(l);
```

- Locks only affect processors using them:
  - If a thread accesses the data without doing the acquire/release, locks by others will not help
- Java, C++, and other languages have lexically scoped synchronization, i.e., synchronized methods/blocks
  - Can't forgot to say “release”
- Semaphores generalize locks to allow k threads simultaneous access; good for limited resources

# Mutexes in POSIX Threads

- To create a mutex:

```
#include <pthread.h>
```

```
pthread_mutex_t amutex = PTHREAD_MUTEX_INITIALIZER;
```

```
// or pthread_mutex_init(&amutex, NULL);
```

- To use it:

```
int pthread_mutex_lock(amutex);
```

```
int pthread_mutex_unlock(amutex);
```

- To deallocate a mutex

```
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

- Multiple mutexes may be held, but can lead to problems:

thread1

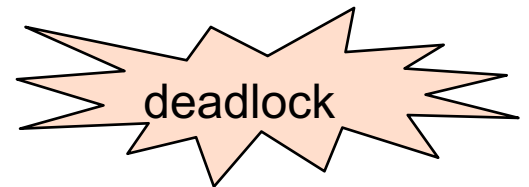
lock (a)

lock (b)

thread2

lock (b)

lock (a)



- Deadlock results if both threads acquire one of their locks, so that neither can acquire the second

# Summary of Programming with Threads

- POSIX Threads are based on OS features
  - Can be used from multiple languages (need appropriate header)
  - Familiar language for most of program
  - Ability to shared data is convenient
- Pitfalls
  - Overhead of thread creation is high (1-loop iteration probably too much)
  - Data race bugs are very nasty to find because they can be intermittent
  - Deadlocks are usually easier, but can also be intermittent
- Researchers look at transactional memory an alternative
- OpenMP is commonly used today as an alternative
  - Helps with some of these, but doesn't make them disappear

# What is OpenMP?

- OpenMP = Open specification for Multi-Processing
  - [openmp.org](https://openmp.org) – Talks, examples, forums, etc.
  - Spec controlled by the ARB
- Motivation: capture common usage and simplify programming
- OpenMP Architecture Review Board (ARB)
  - A nonprofit organization that controls the OpenMP Spec
  - Latest spec: OpenMP 5.0 (Nov. 2018)
- High-level API for programming in C/C++ and Fortran
  - **Preprocessor (compiler) directives ( ~ 80% )**  
`#pragma omp construct [clause [clause ...]]`
  - **Library Calls ( ~ 19% )**  
`#include <omp.h>`
  - **Environment Variables ( ~ 1% )**  
all caps, added to srun, etc.

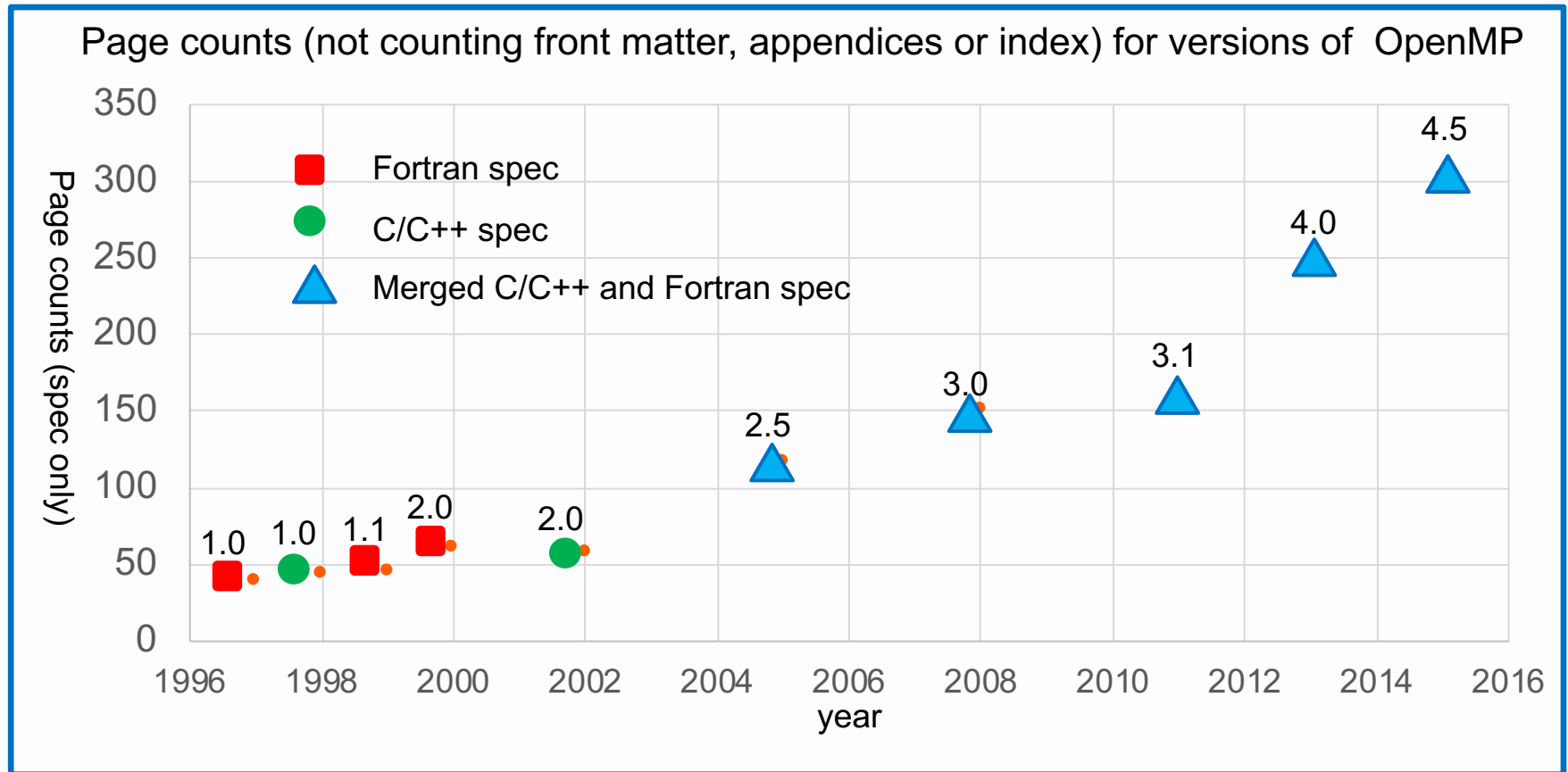
# A Programmer's View of OpenMP

- OpenMP is a portable, threaded, shared-memory programming *specification* with “light” syntax
  - Requires compiler support (C, C++ or Fortran)
- OpenMP will:
  - Allow a programmer to separate a program into *serial regions* and *parallel regions*, rather than P concurrently-executing threads.
  - Hide stack management
  - Provide synchronization constructs
- OpenMP will not:
  - Parallelize automatically
  - Guarantee speedup
  - Provide freedom from data races



# The growth of complexity in OpenMP

- OpenMP started out in 1997 as a simple interface for the application programmers more versed in their area of science than computer science.
- The complexity has grown considerably over the years!

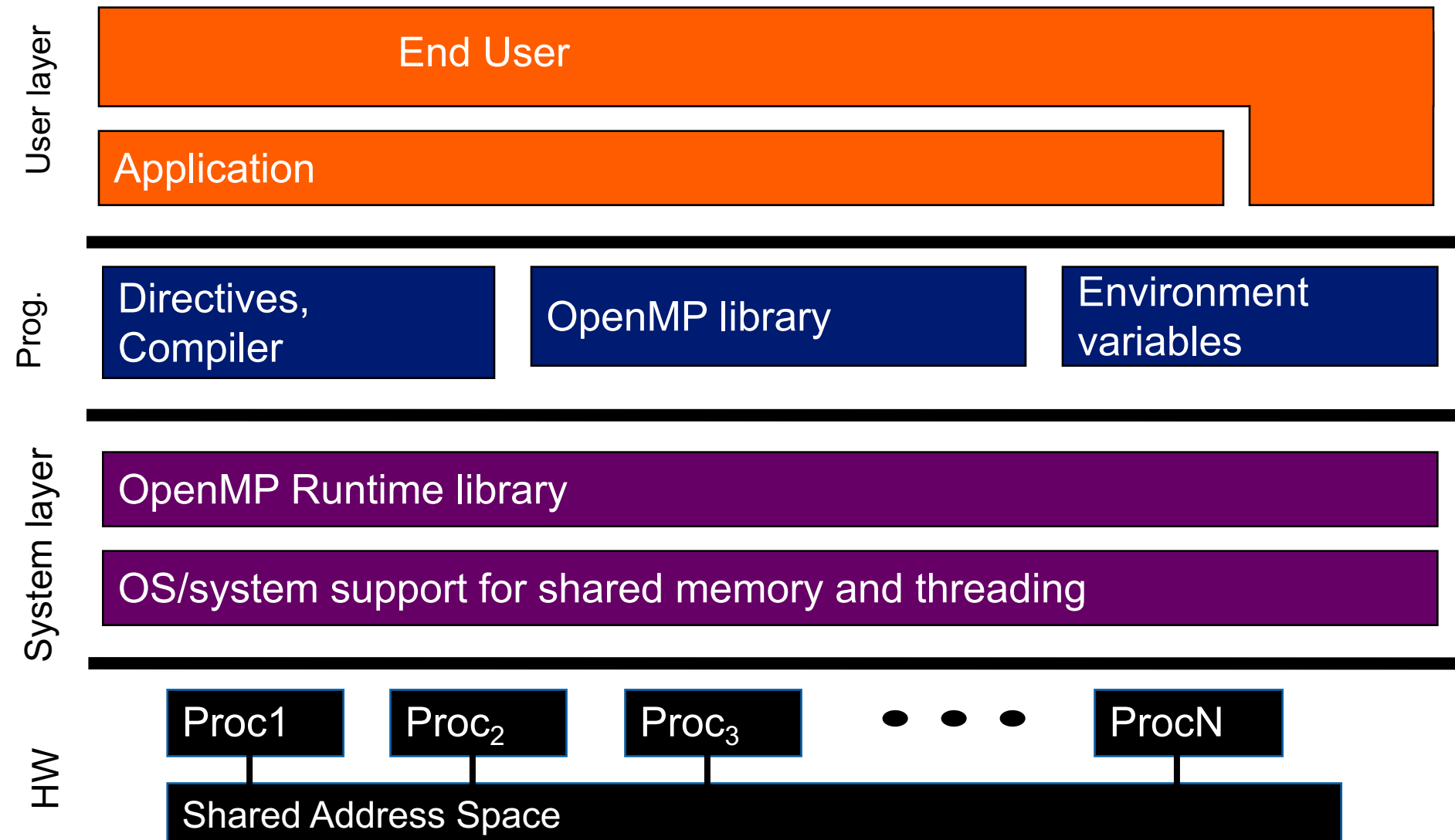


OpenMP 5.0 (November 2018) is actually **666 pages**.

# The OpenMP Common Core: Most OpenMP programs only use these 19 items

| OpenMP pragma, function, or clause   | Concepts   |
|--|--|
| <code>#pragma omp parallel</code>  | Parallel region, teams of threads, structured block, interleaved execution across threads  |
| <code>int omp_get_thread_num()</code><br><code>int omp_get_num_threads()</code>          | Create threads with a parallel region and split up the work using the number of threads and thread ID                              |
| <code>double omp_get_wtime()</code>  | Speedup and Amdahl's law.<br>False Sharing and other performance issues  |
| <code>setenv OMP_NUM_THREADS N</code>  | Internal control variables. Setting the default number of threads with an environment variable                                     |
| <code>#pragma omp barrier</code><br><code>#pragma omp critical</code>                    | Synchronization and race conditions. Revisit interleaved execution.  |
| <code>#pragma omp for</code><br><code>#pragma omp parallel for</code>                    | Worksharing, parallel loops, loop carried dependencies   |
| <code>reduction(op:list)</code>  | Reductions of values across a team of threads  |
| <code>schedule(dynamic [,chunk])</code><br><code>schedule (static [,chunk])</code>       | Loop schedules, loop overheads and load balance  |
| <code>private(list)</code> , <code>firstprivate(list)</code> , <code>shared(list)</code> | Data environment   |
| <code>nowait</code>  | Disabling implied barriers on workshare constructs, the high cost of barriers, and the flush concept (but not the flush directive) |
| <code>#pragma omp single</code>  | Workshare with a single thread   |
| <code>#pragma omp task</code><br><code>#pragma omp taskwait</code>                       | Tasks including the data environment for tasks.  |

# OpenMP basic definitions: Basic Solution stack



# OpenMP basic syntax

- Most of the constructs in OpenMP are compiler directives.

| C and C++   | Fortran  |
|---|--|
| Compiler directives   |  |
| <b><i>#pragma omp construct [clause [clause]...]</i></b>  | <b><i>!\$OMP construct [clause [clause] ...]</i></b>                                   |
| Example   |  |
| <b><i>#pragma omp parallel private(x)</i></b><br><b><i>{</i></b><br><br><br><br><br><b><i>}</i></b> | <b><i>!\$OMP PARALLEL</i></b><br><br><br><br><br><br><b><i>!\$OMP END PARALLEL</i></b> |
| Function prototypes and types:  |  |
| <b><i>#include &lt;omp.h&gt;</i></b>  | <b><i>use OMP_LIB</i></b>  |

- 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.

# Hello world in OpenMP

- Write a program that prints “hello world”.

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

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

}
```

# Hello world in OpenMP

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

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

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

    }
}
```

## Switches for compiling and linking

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

# Hello world in OpenMP

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

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

```
#include <stdio.h>
```

```
int main()
```

```
{
```

```
#pragma omp parallel
```

```
{
```

```
    printf(" hello ");
```

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

```
}
```

```
}
```

OpenMP include file

Parallel region with  
default number of threads

End of the Parallel region

Sample Output:

hello hello world

world

hello hello world

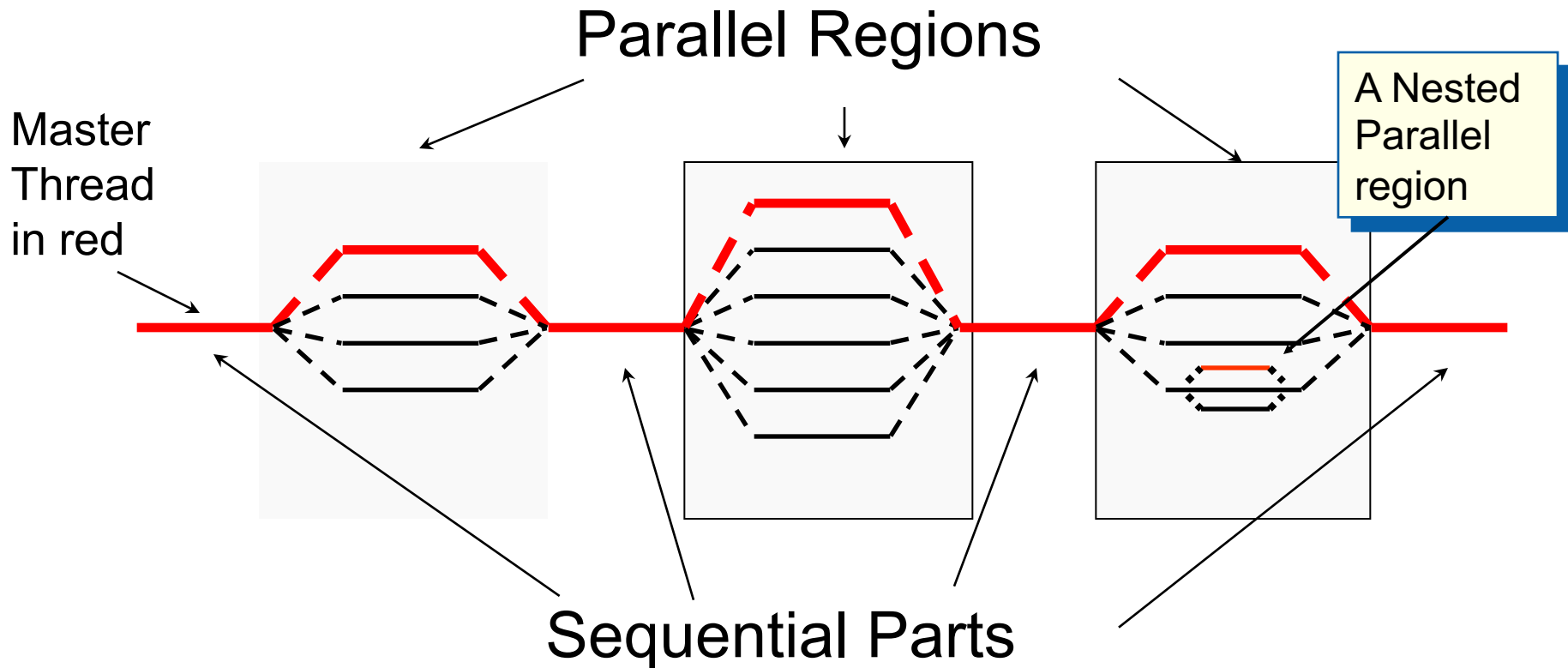
world

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

# OpenMP programming model:

## Fork-Join Parallelism:

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





# Thread creation: Parallel regions

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

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

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

Runtime function to request a certain number of threads

Runtime function returning a thread ID

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

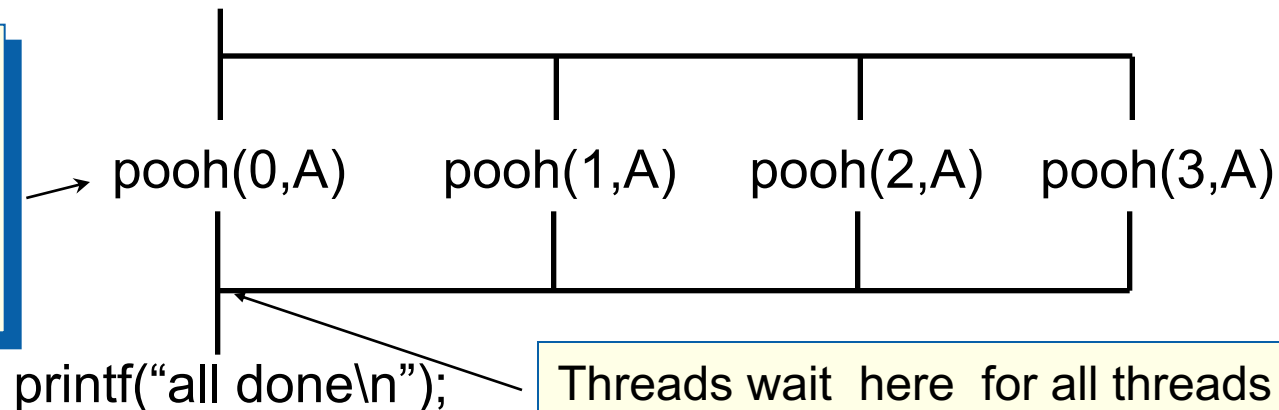
# Thread creation: Parallel regions example

- Each thread executes the same code redundantly.

```
double A[1000];  
omp_set_num_threads(4)
```

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

A single copy of A is shared between all threads.



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

# Thread creation: How many threads did you actually get?

- You create a team threads in OpenMP\* with the parallel construct.
- You can request a number of threads with `omp_set_num_threads()`
- But is the number of threads requested the number you actually get?
  - NO! An implementation can silently decide to give you a team with fewer threads.
  - Once a team of threads is established ... the system will not reduce the size of the team.

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

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

Runtime function to request a certain number of threads

Runtime function to return actual number of threads in the team

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

# An interesting problem to play with

## Numerical integration

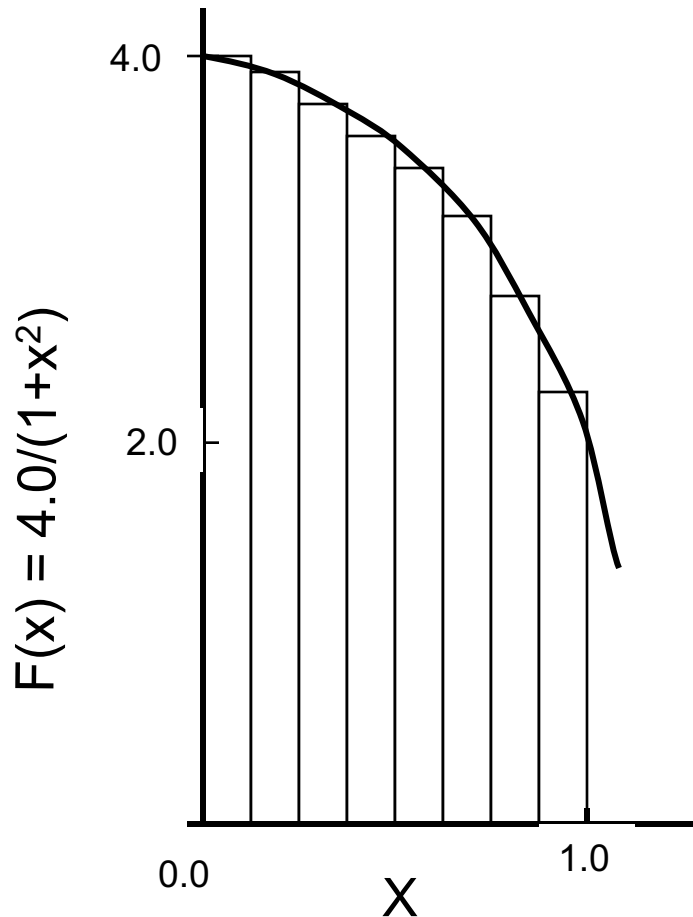
Mathematically, we know that:

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

We can approximate the integral as a sum of rectangles:

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

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



# Serial PI program

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

    step = 1.0/(double) num_steps;

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

# Serial PI program

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

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

```
double step;
```

```
int main ()
```

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

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

```
    tdata = omp_get_wtime();
```

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

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

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

```
    }
```

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

```
    tdata = omp_get_wtime() - tdata;
```

```
    printf(" pi = %f in %f secs\n",pi, tdata);
```

```
}
```

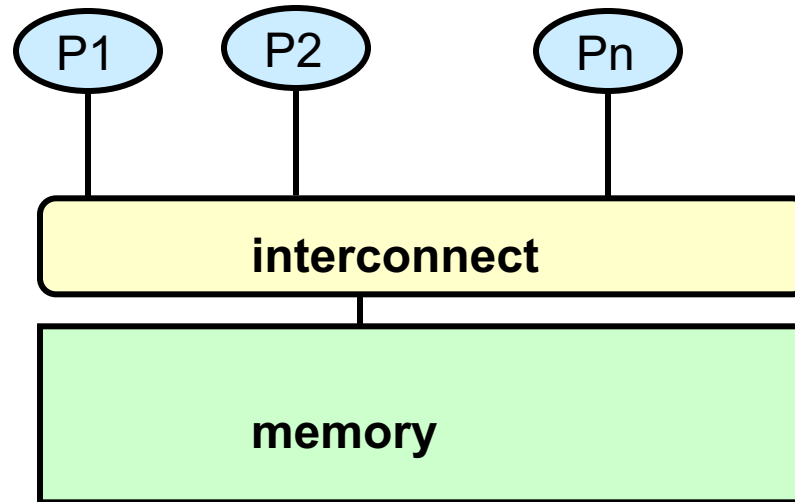
The library routine `get_omp_wtime()` is used to find the elapsed “wall time” for blocks of code

---

# **Shared Memory Hardware and Memory Consistency**

# Basic Shared Memory Architecture

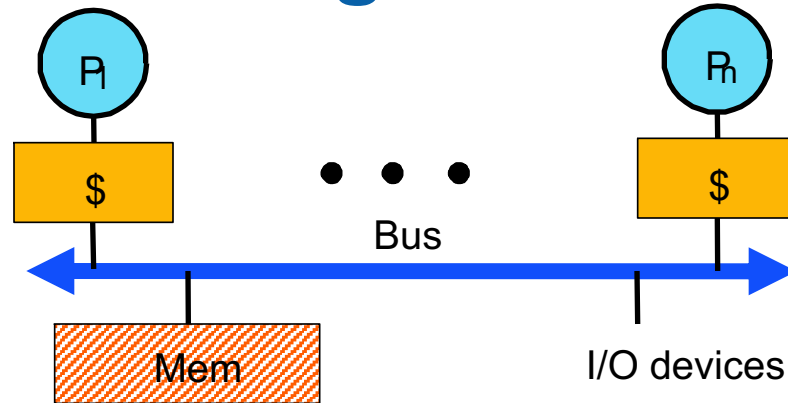
- Processors all connected to a large shared memory
  - Where are caches?



- Now take a closer look at structure, costs, limits, programming

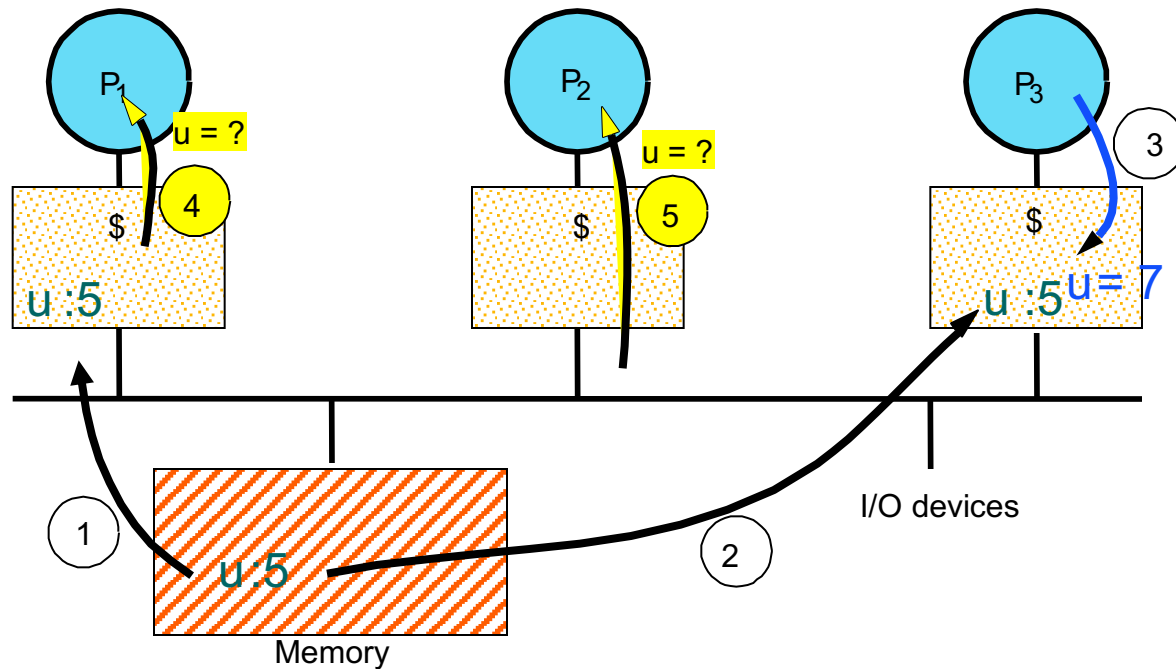


# What About Caching???



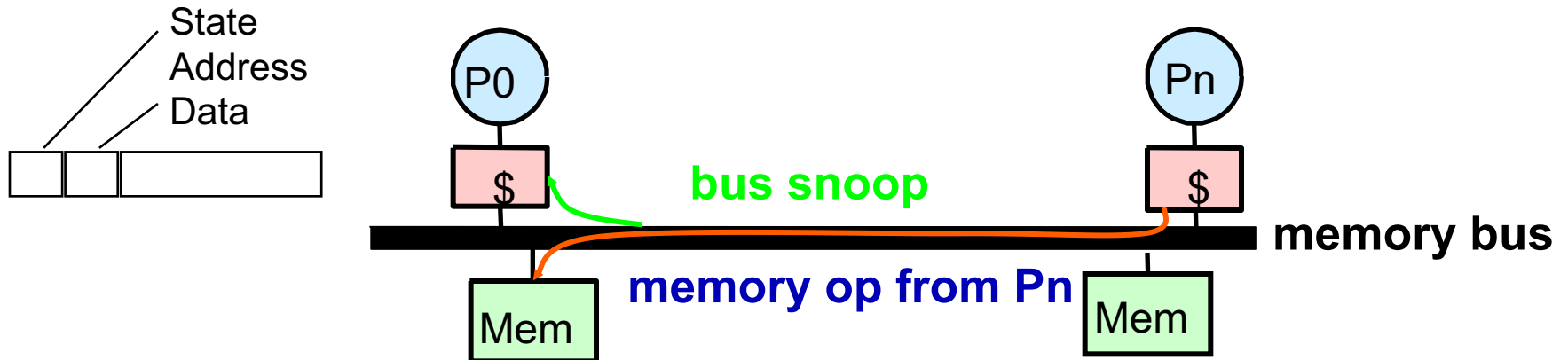
- Want high performance for shared memory: Use Caches!
  - Each processor has its own cache (or multiple caches)
  - Place data from memory into cache
  - Writeback cache: don't send all writes over bus to memory
- Caches reduce average latency
  - Automatic replication closer to processor
  - *More* important to multiprocessor than uniprocessor: latencies longer
- Normal uniprocessor mechanisms to access data
  - Loads and Stores form very low-overhead communication primitive
- Problem: Cache Coherence!

# Example Cache Coherence Problem



- Things to note:
  - Processors could see different values for  $u$  after event 3
  - With write back caches, value written back to memory depends on happenstance of which cache flushes or writes back value when
- How to fix with a bus: Coherence Protocol
  - Use bus to broadcast writes or invalidations
  - Simple protocols rely on presence of broadcast medium
- Bus not scalable beyond about 100 processors (max)
  - Capacity, bandwidth limitations

# Snoopy Cache-Coherence Protocols



- Memory bus is a broadcast medium
- Caches contain information on which addresses they store
- Cache Controller “snoops” all transactions on the bus
  - A transaction is a relevant transaction if it involves a cache block currently contained in this cache
  - Take action to ensure coherence
    - invalidate, update, or supply value
  - Many possible designs (see CS252 or CS258)

# Intuitive Memory Model

- Reading an address should **return the last value written** to that address
- Easy in uniprocessors
  - except for I/O
- Cache coherence problem in MPs is more pervasive and more performance critical
- More formally, this is called **sequential consistency**:

“A multiprocessor is *sequentially consistent* if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.” [Lamport, 1979]

# Parallel PI program (2013)

- Original Serial pi program with 100 million steps ran in 1.83 seconds\*.

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

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

```
#define NUM_THREADS 4
```

```
void main ()
```

```
{   int i, nthreads; double pi, sum[NUM_THREADS];
```

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

```
    omp_set_num_threads(NUM_THREADS);
```

```
    #pragma omp parallel
```

```
    {   int i, id, nthrds;
```

```
        double x;
```

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

```
        nthrds = omp_get_num_threads();
```

```
        if (id == 0) nthreads = nthrds;
```

```
        for (i=id, sum[id]=0.0; i< num_steps; i=i+nthrds) {
```

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

```
            sum[id] += 4.0/(1.0+x*x);
```

```
        }
```

```
    }
```

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

```
}
```

| threads | 1 <sup>st</sup><br>SPMD* |
|---------|--------------------------|
| 1       | 1.86                     |
| 2       | 1.03                     |
| 3       | 1.08                     |
| 4       | 0.97                     |

\* Intel compiler (icpc) with default optimization level (O2) on Apple OS X 10.7.3 with a dual core (four HW thread) Intel® Core™ i5 processor at 1.7 Ghz and 4 Gbyte DDR3 memory at 1.333 Ghz.

# Parallel PI program (2020)

- Original Serial pi program with 1 billion steps ran in 0.985 seconds\*.

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

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

```
#define NUM_THREADS 4
```

```
void main ()
```

```
{   int i, nthreads; double pi, sum[NUM_THREADS];
```

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

```
    omp_set_num_threads(NUM_THREADS);
```

```
    #pragma omp parallel
```

```
    {   int i, id, nthrds;
```

```
        double x;
```

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

```
        nthrds = omp_get_num_threads();
```

```
        if (id == 0) nthreads = nthrds;
```

```
        for (i=id, sum[id]=0.0; i< num_steps; i=i+nthrds) {
```

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

```
            sum[id] += 4.0/(1.0+x*x);
```

```
        }
```

```
    }
```

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

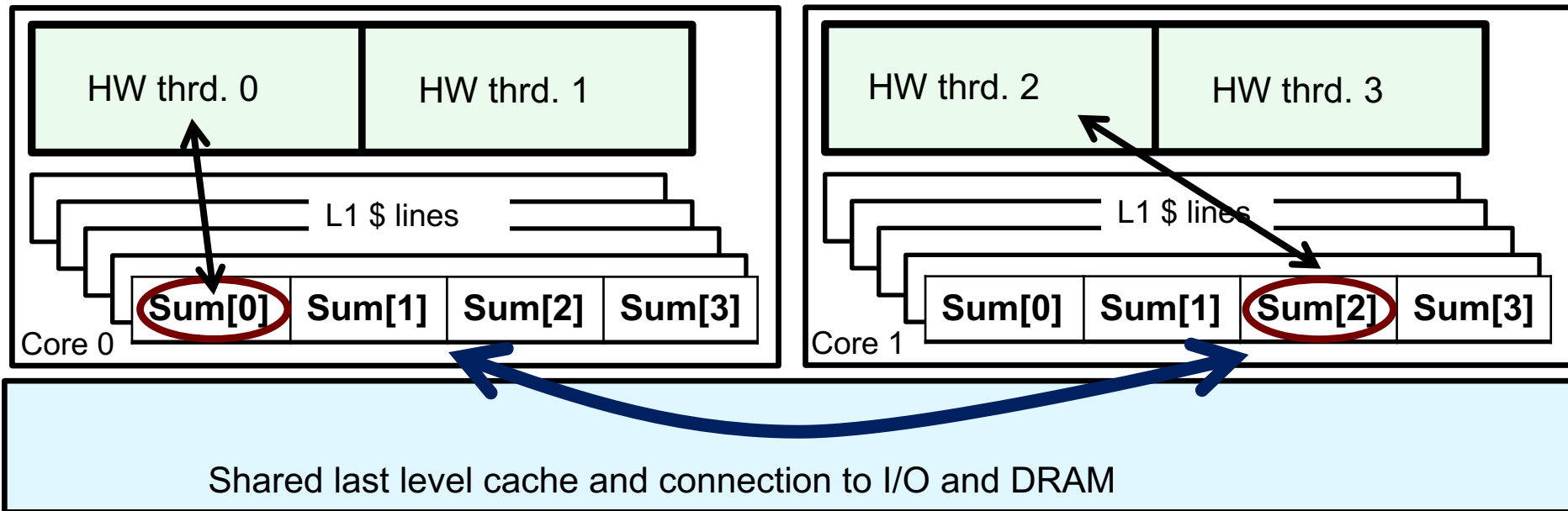
```
}
```

| threads | 1 <sup>st</sup><br>SPMD* |
|---------|--------------------------|
| 1       | 1.102                    |
| 2       | 0.512                    |
| 4       | 0.280                    |
| 8       | 0.276                    |

\*GCC with -O3 optimization on Apple macOS X 10.14.6 with a quad core (8 HW threads) 2.8 GHz Intel Core i7 and 16 Gbyte LPDDR3 memory at 2.133 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 4
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]=0.0; 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 (2013)

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

```
#include <omp.h>
static long num_steps = 100000;      double step;
#define PAD 8 // assume 64 byte L1 cache line size
#define NUM_THREADS 4
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) nthrds = nthrds;
        for (i=id, sum[id][0]=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 | 1 <sup>st</sup><br>SPMD | 1 <sup>st</sup><br>SPMD<br>padded |
|---------|-------------------------|-----------------------------------|
| 1       | 1.86                    | 1.86                              |
| 2       | 1.03                    | 1.01                              |
| 3       | 1.08                    | 0.69                              |
| 4       | 0.97                    | 0.53                              |

\*Intel compiler (icpc) with default optimization level (O2) on Apple OS X 10.7.3 with a dual core (four HW thread) Intel® Core™ i5 processor at 1.7 Ghz and 4 Gbyte DDR3 memory at 1.333 Ghz.

# Results\*: pi program padded accumulator (2020)

- Original Serial pi program with 1 billion steps ran in 0.985 seconds\*.

```
#include <omp.h>
static long num_steps = 100000;      double step;
#define PAD 8           // assume 64 byte L1 cache line size
#define NUM_THREADS 4
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]=0.0;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 | 1 <sup>st</sup><br>SPMD | 1 <sup>st</sup><br>SPMD<br>padded |
|---------|-------------------------|-----------------------------------|
| 1       | 1.102                   | 0.987                             |
| 2       | 0.512                   | 0.496                             |
| 4       | 0.280                   | 0.271                             |
| 8       | 0.276                   | 0.268                             |

\*GCC with -O3 optimization on Apple macOS X 10.14.6 with a quad core (8 HW threads) 2.8 GHz Intel Core i7 and 16 Gbyte LPDDR3 memory at 2.133 Ghz.

# Synchronization

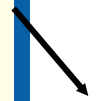
- High level synchronization included in the common core (the full OpenMP specification has MANY more):
  - critical
  - barrier

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

# Synchronization: critical

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

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



```
float res;  
  
#pragma omp parallel  
{   float B;  int i, id, nthrds;  
    id = omp_get_thread_num();  
    nthrds = omp_get_num_threads();  
    for(i=id;i<niters;i+=nthrds){  
        B = big_job(i);  
  
        #pragma omp critical  
        res += consume (B);  
    }  
}
```

# Synchronization: barrier

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

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


omp_set_num_threads(8)

#pragma omp parallel
{  int id, nthrds;

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

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

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



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

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

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

```
#define NUM_THREADS 4
```

```
void main ()
```

```
{    int nthreads; double pi=0.0;        step = 1.0/(double) num_steps;  
    omp_set_num_threads(NUM_THREADS);
```

```
#pragma omp parallel
```

```
{
```

```
    int i, id, nthrds;    double x, sum;
```

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

```
    nthrds = omp_get_num_threads();
```

```
    if (id == 0)    nthreads = nthrds;
```

```
    for (i=id, sum=0.0; i< num_steps; i=i+nthrds) {
```

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

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

```
    }
```

```
#pragma omp critical
```

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

```
}
```

```
}
```

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

No array, so no false sharing.

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

# Results\*: pi program critical section

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

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

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

| threads | 1 <sup>st</sup><br>SPMD | 1 <sup>st</sup><br>SPMD<br>padded | SPMD<br>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 default optimization level (O2) on Apple OS X 10.7.3 with a dual core (four HW thread) Intel® Core™ i5 processor at 1.7 Ghz and 4 Gbyte DDR3 memory at 1.333 Ghz.

# The loop worksharing constructs

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

```
#pragma omp parallel
```

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

Loop construct name:

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

The loop control index I is made “private” to each thread by default.

Threads wait here until all threads are finished with the parallel loop before any proceed past the end of the loop



# Loop worksharing constructs

## A motivating example

Sequential code

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

OpenMP parallel region

```
#pragma omp parallel
```

```
{
```

```
    int id, i, Nthrds, istart, iend;
```

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

```
    Nthrds = omp_get_num_threads();
```

```
    istart = id * N / Nthrds;
```

```
    iend = (id+1) * N / Nthrds;
```

```
    if (id == Nthrds-1) iend = N;
```

```
    for(i=istart;i<iend;i++) { a[i] = a[i] + b[i];}
```

```
}
```

OpenMP parallel region and a worksharing for construct

```
#pragma omp parallel
```

```
#pragma omp for
```

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

# Loop worksharing constructs:

## The schedule clause

- The schedule clause affects how loop iterations are mapped onto threads
  - `schedule(static [,chunk])`
    - Deal-out blocks of iterations of size “chunk” to each thread.
  - `schedule(dynamic[,chunk])`
    - Each thread grabs “chunk” iterations off a queue until all iterations have been handled.

| Schedule Clause | When To Use  |
|-----------------|--|
| <b>STATIC</b>   | <b>Pre-determined and predictable by the programmer</b>  |
| <b>DYNAMIC</b>  | <b>Unpredictable, highly variable work per iteration</b> |

Least work at runtime :  
scheduling done at compile-time

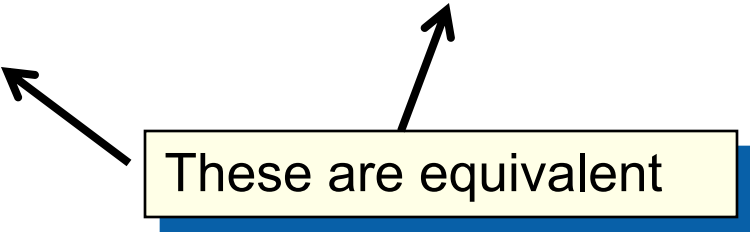
Most work at runtime :  
complex scheduling logic used at run-time

# Combined parallel/worksharing construct

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

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

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



These are equivalent

# Working with loops

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

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

Note: loop index  
"i" is private by  
default

Remove loop  
carried  
dependence

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

# Reduction

- How do we handle this case?

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

- We are combining values into a single accumulation variable (ave) ... there is a true dependence between loop iterations that can't be trivially removed
- This is a very common situation ... it is called a “reduction”.
- Support for reduction operations is included in most parallel programming environments.

# Reduction

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

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

# OpenMP: Reduction operands/initial-values

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

| Operator   | Initial value       |
|------------|---------------------|
| <b>+</b>   | <b>0</b>            |
| <b>*</b>   | <b>1</b>            |
| <b>-</b>   | <b>0</b>            |
| <b>min</b> | Largest pos. number |
| <b>max</b> | Most neg. number    |

| C/C++ only        |               |
|-------------------|---------------|
| Operator          | Initial value |
| <b>&amp;</b>      | <b>~0</b>     |
| <b> </b>          | <b>0</b>      |
| <b>^</b>          | <b>0</b>      |
| <b>&amp;&amp;</b> | <b>1</b>      |
| <b>  </b>         | <b>0</b>      |

| Fortran Only  |                    |
|---------------|--------------------|
| Operator      | Initial value      |
| <b>.AND.</b>  | <b>.true.</b>      |
| <b>.OR.</b>   | <b>.false.</b>     |
| <b>.NEQV.</b> | <b>.false.</b>     |
| <b>.IEOR.</b> | <b>0</b>           |
| <b>.IOR.</b>  | <b>0</b>           |
| <b>.IAND.</b> | <b>All bits on</b> |
| <b>.EQV.</b>  | <b>.true.</b>      |

# 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><br>SPMD | 1 <sup>st</sup><br>SPMD<br>padded | SPMD<br>critical | PI Loop<br>and<br>reduction |
|---------|-------------------------|-----------------------------------|------------------|-----------------------------|
| 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 default optimization level (O2) on Apple OS X 10.7.3 with a dual core (four HW thread) Intel® Core™ i5 processor at 1.7 Ghz and 4 Gbyte DDR3 memory at 1.333 Ghz.

# The nowait clause

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

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

```
#pragma omp parallel
```

```
{
```

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

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

```
#pragma omp barrier
```

```
#pragma omp for
```

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


```
#pragma omp for nowait
```

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

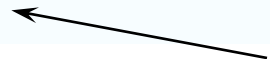
```
    A[id] = big_calc4(id);
```

```
}
```

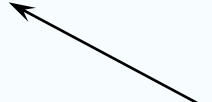
implicit barrier at the end of a for  
worksharing construct



implicit barrier at the end  
of a parallel region



no implicit barrier  
due to nowait



# Data environment:

## Default storage attributes

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

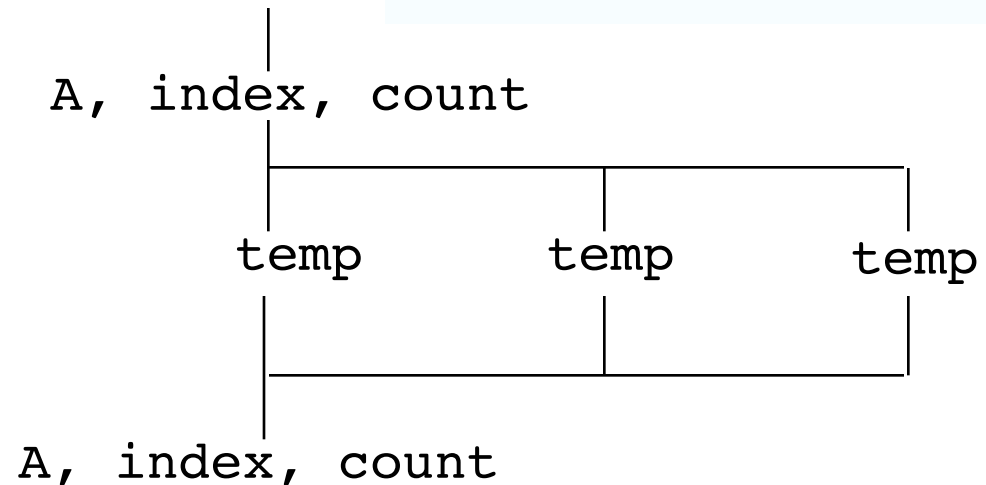
# Data sharing: Examples

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

A, index and count are shared by all threads.

temp is local to each thread

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



# Data sharing:

## Changing storage attributes

- One can selectively change storage attributes for constructs using the following clauses\* (note: list is a comma-separated list of variables)
  - shared(list)
  - private(list)
  - firstprivate(list)
- These can be used on parallel and for constructs ... other than shared which can only be used on a parallel construct
- Force the programmer to explicitly define storage attributes
  - default (none)

These clauses apply to the OpenMP construct NOT to the entire region.

default() can be used on parallel constructs

# Data sharing: Private clause

- `private(var)` creates a new local copy of `var` for each thread.
  - The value of the private copies is **uninitialized**
  - The value of the original variable is unchanged after the region

```
void wrong() {  
    int tmp = 0;  
    #pragma omp parallel for private(tmp)  
    for (int j = 0; j < 1000; ++j)  
        tmp += j;  
    printf("%d\n", tmp);  
}
```

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

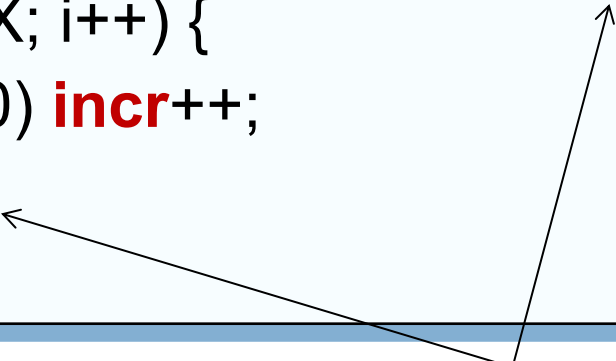
tmp is 0 here

tmp was not initialized

# Firstprivate clause

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

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



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

# Data sharing:

## A data environment test

- Consider this example of PRIVATE and FIRSTPRIVATE

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

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

Inside this parallel region ...

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

Following the parallel region ...

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



# Data sharing: Default clause

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

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

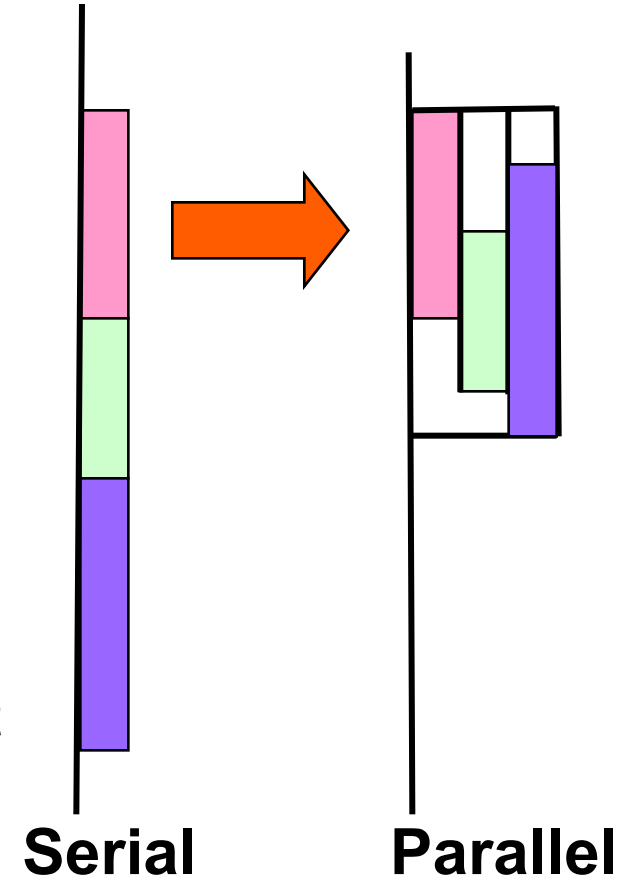
The static extent is the code in the compilation unit that contains the construct.

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

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

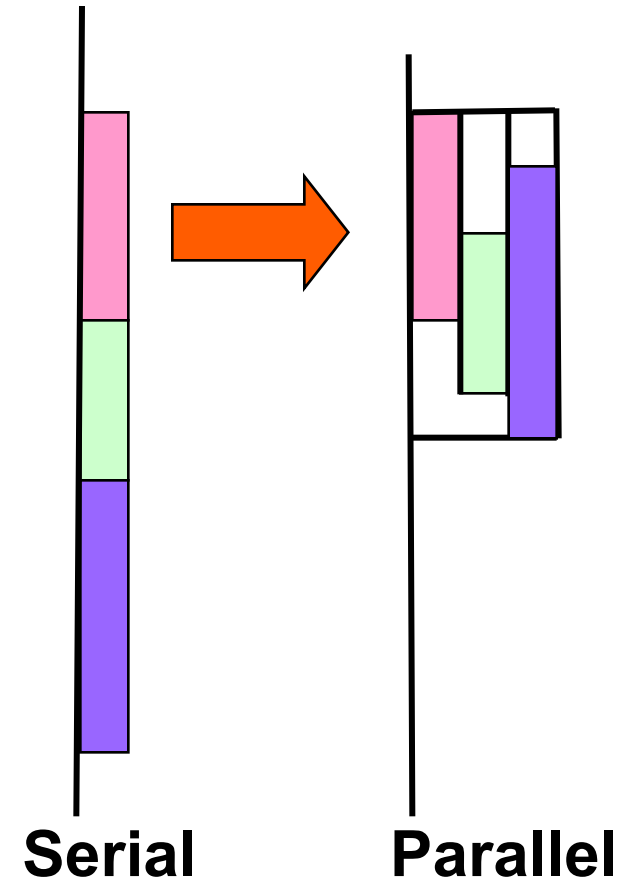
# What are tasks?

- Tasks are independent units of work
- Tasks are composed of:
  - code to execute
  - data to compute with
- Threads are assigned to perform the work of each task.
  - The thread that encounters the task construct may execute the task immediately.
  - The threads may defer execution until later



# What are tasks?

- The task construct includes a structured block of code
- Inside a parallel region, a thread encountering a task construct will package up the code block and its data for execution
- Tasks can be nested: i.e. a task may itself generate tasks.



A common Pattern is to have one thread create the tasks while the other threads wait at a barrier and execute the tasks

# 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();
}
```

# Task Directive

`#pragma omp task [clauses]`  
*structured-block*

```
#pragma omp parallel  
{
```

Create some threads

```
    #pragma omp single  
    {
```

One Thread  
packages tasks

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

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

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

Tasks executed by  
some thread in some  
order

```
    }
```

```
}
```

All tasks complete before this barrier is released


# When/where are tasks complete?

- At thread barriers (explicit or implicit)
  - applies to **all tasks generated in the current parallel** region up to the barrier
- At taskwait directive
  - i.e. wait until all tasks defined within the scope of the current task have completed.  
`#pragma omp taskwait`
  - Note: **applies only to tasks generated in the current task, not to “descendants”** .
  - To also wait for descendants, there is the `taskgroup *region*`

# Example

```
#pragma omp parallel
{
    #pragma omp single
    {
        #pragma omp task
        fred();
        #pragma omp task
        daisy();
        #pragma taskwait
        #pragma omp task
        billy();
    }
}
```

`fred()` and `daisy()`  
must complete before  
`billy()` starts

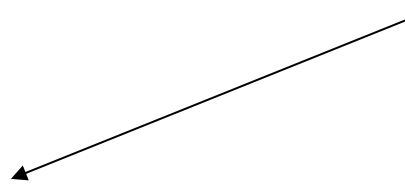


# Data scoping defaults

- The behavior you want for tasks is usually firstprivate, because the task may not be executed until later (and variables may have gone out of scope)
  - Variables that are private when the task construct is encountered are firstprivate by default [OpenMP grandmothering you here]
- Variables that are shared in all constructs starting from the innermost enclosing parallel construct are shared by default

```
#pragma omp parallel shared(A) private(B)
{
    ...
    #pragma omp task
    {
        int C;
        compute(A, B, C);
    }
}
```

A is shared  
B is firstprivate  
C is private





# Example: Fibonacci numbers

```
int fib (int n)
{
    int x,y;
    if (n < 2) return n;

    x = fib(n-1);
    y = fib(n-2);
    return (x+y);
}
```

```
int main()
{
    int NW = 5000;
    fib(NW);
}
```

- $F_n = F_{n-1} + F_{n-2}$
- Inefficient  $O(2^n)$  recursive implementation!

# Parallel Fibonacci

```
int fib (int n)
{  int x,y;
   if (n < 2) return n;

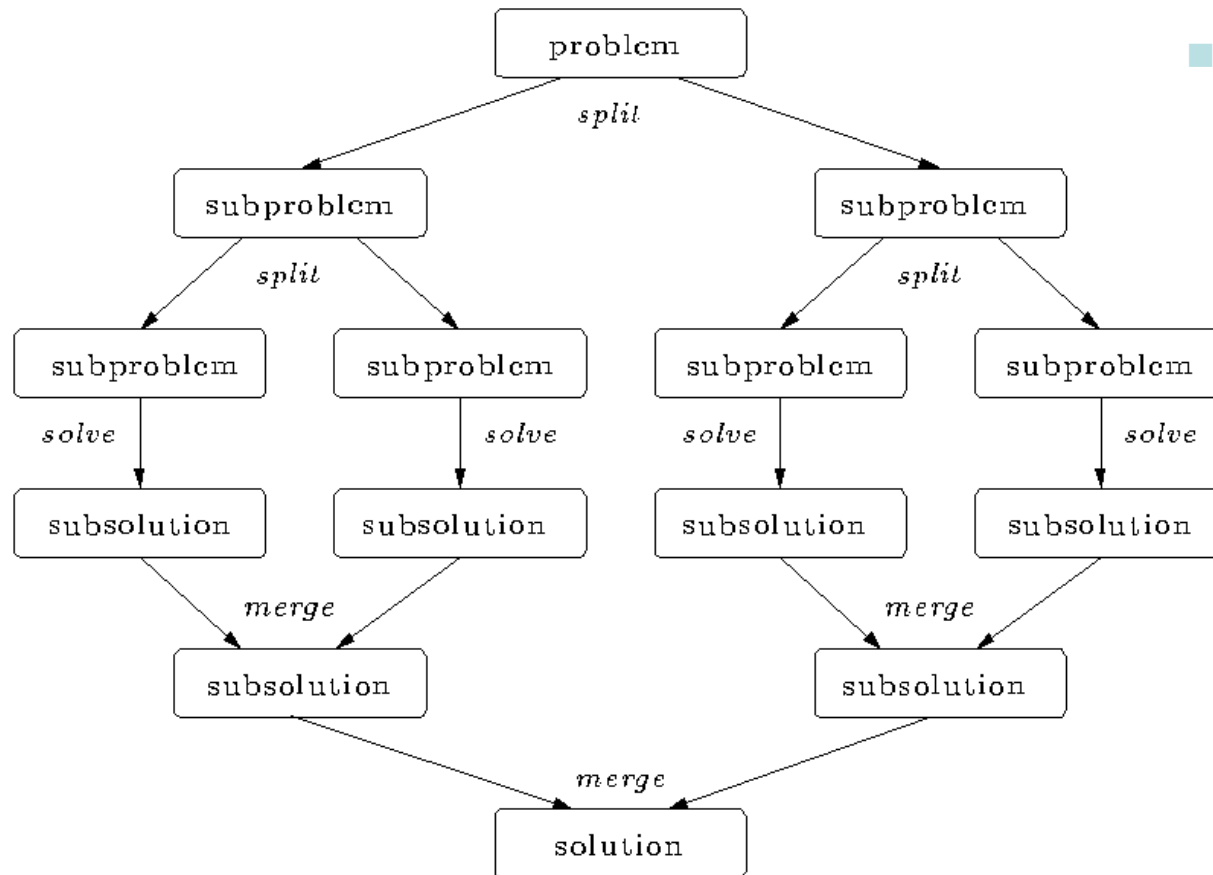
   #pragma omp task shared(x)
   x = fib(n-1);
   #pragma omp task shared(y)
   y = fib(n-2);
   #pragma omp taskwait
   return (x+y);
}
```

```
int main()
{  int NW = 5000;
   #pragma omp parallel
   {
       #pragma omp single
       fib(NW);
   }
}
```

- Binary tree of tasks
- Traversed using a recursive function
- A task cannot complete until all tasks below it in the tree are complete (enforced with taskwait)
- **x, y** are local, and so by default they are private to current task
  - must be shared on child tasks so they don't create their own firstprivate copies at this level!

# 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

# Synchronization

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

- High level synchronization:

- **critical**

- **barrier**

Covered earlier

- atomic

- ordered

- Low level synchronization

- flush

- locks (both simple and nested)

# Synchronization: atomic

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

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

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

# Synchronization: atomic

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

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

Atomic only protects the read/update of X

Additional forms of atomic were added in 3.1 (discussed later)

# Flush operation

- Defines a sequence point at which a thread enforces a consistent view of memory.
- For variables visible to other threads and associated with the flush operation (the **flush-set**)
  - The compiler can't move loads/stores of the flush-set around a flush:
    - All previous read/writes of the flush-set by this thread have completed
    - No subsequent read/writes of the flush-set by this thread have occurred
  - Variables in the flush set are moved from temporary storage to shared memory.
  - Reads of variables in the flush set following the flush are loaded from shared memory.

IMPORTANT POINT: The flush makes the calling threads temporary view match the view in shared memory. Flush by itself does not force synchronization.

# Memory consistency: flush example

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

```
double A;  
A = compute();  
#pragma omp flush(A)
```

```
// flush to memory to make sure other  
// threads can pick up the right value
```

Flush without a list: flush set is all thread visible variables

Flush with a list: flush set is the list of variables

```
T1: x=1; y=1;  
T2: int r1=y; int r2=x;
```

You think: *“If T2 sees value of y on read to be 1 then the following read of x should also return the value 1”*, but what if T1’s instructions were ordered?

Note: OpenMP’s flush is analogous to a fence in other shared memory APIs



# Flush and synchronization

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

# What to Take Away?

- Programming shared memory machines
  - May allocate data in large shared region without too many worries about where
  - Memory hierarchy is critical to performance
    - Even more so than on uniprocessors, due to coherence traffic
  - For performance tuning, watch sharing (both true and false)
- Semantics
  - Need to lock access to shared variable for read-modify-write
  - Sequential consistency is the natural semantics
    - Write race-free programs to get this
  - Architects worked hard to make this work
    - Caches are coherent with buses or directories
    - No caching of remote data on shared address space machines
  - But compiler and processor may still get in the way
    - Non-blocking writes, read prefetching, code motion...
    - Avoid races or use machine-specific fences carefully

# Shared Memory Programming

## Several Thread Libraries/systems

- PTHREADS is the POSIX Standard
  - Portable but; relatively heavyweight; low level
- OpenMP standard for application level programming
  - Support for scientific programming on shared memory, [openmp.org](http://openmp.org)
- TBB: Thread Building Blocks
  - Intel C++ template library for parallel (multicore) computing
- CILK: Language of the C “ilk”
  - Lightweight threads embedded into C
  - MIT → Cilk Arts → Intel
- Java threads
  - Built on top of POSIX threads; Object within Java language

# Resources

- Tim Mattson runs OpenMP tutorials at SC and Argonne Training Program. NERSC also runs training programs. One from last year is:  
<https://www.nersc.gov/users/training/events/openmp-common-core-february-2018/>
  - What you have seen today included a succinct summary of this common core (I spent about 1/3<sup>rd</sup> of the time).
- Other Shared Memory Models in C/C++: CILK, TBB, Raja, Kokkos (some are just template libraries)
- C++ extensions:  
<http://en.cppreference.com/w/cpp/experimental/parallelism>
- Shared memory languages: Java, Multilisp, Haskell,...