Concurrency vs. Parallelism

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

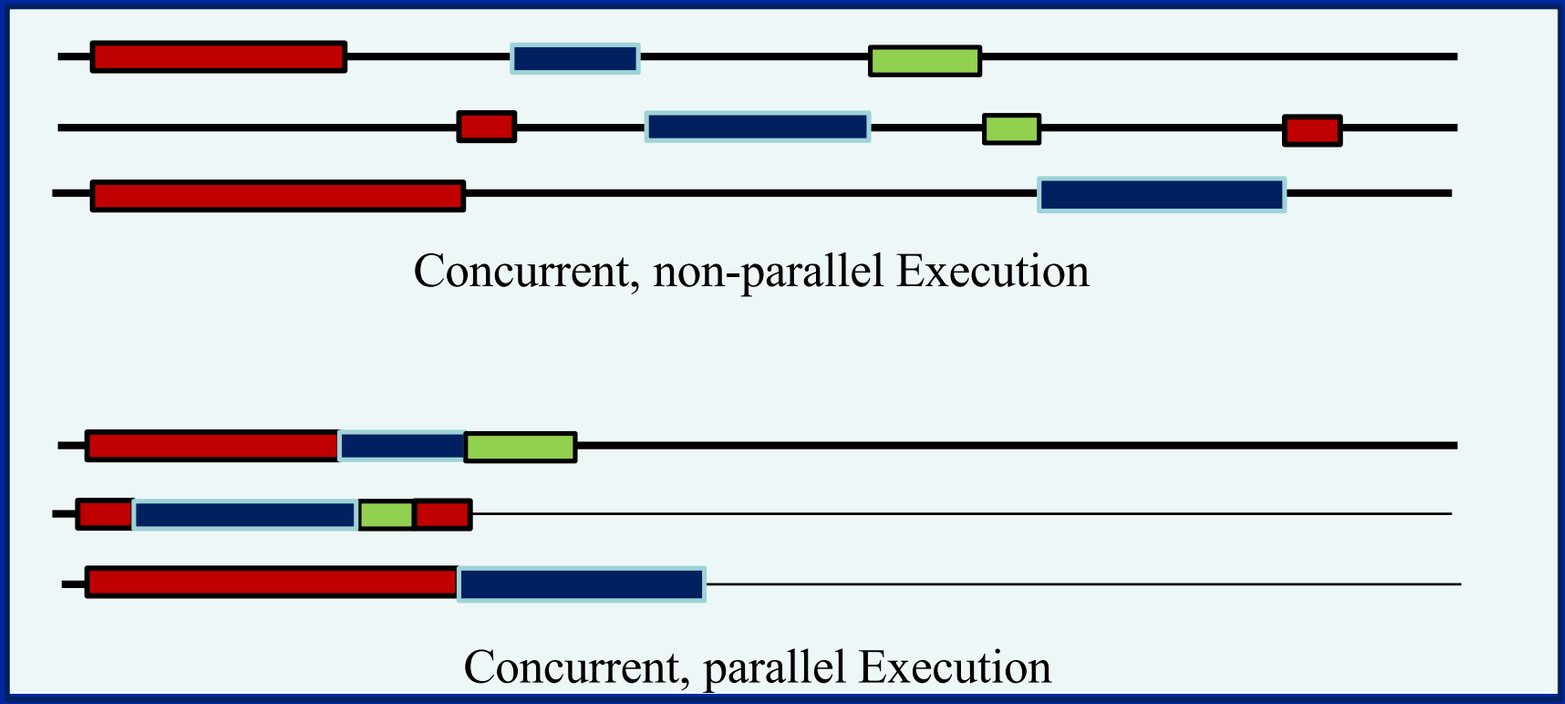


Figure from "An Introduction to Concurrency in Programming Languages" by J. Sottile, Timothy G. Mattson, and Craig E Rasmussen, 2010

Concurrency vs. Parallelism

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

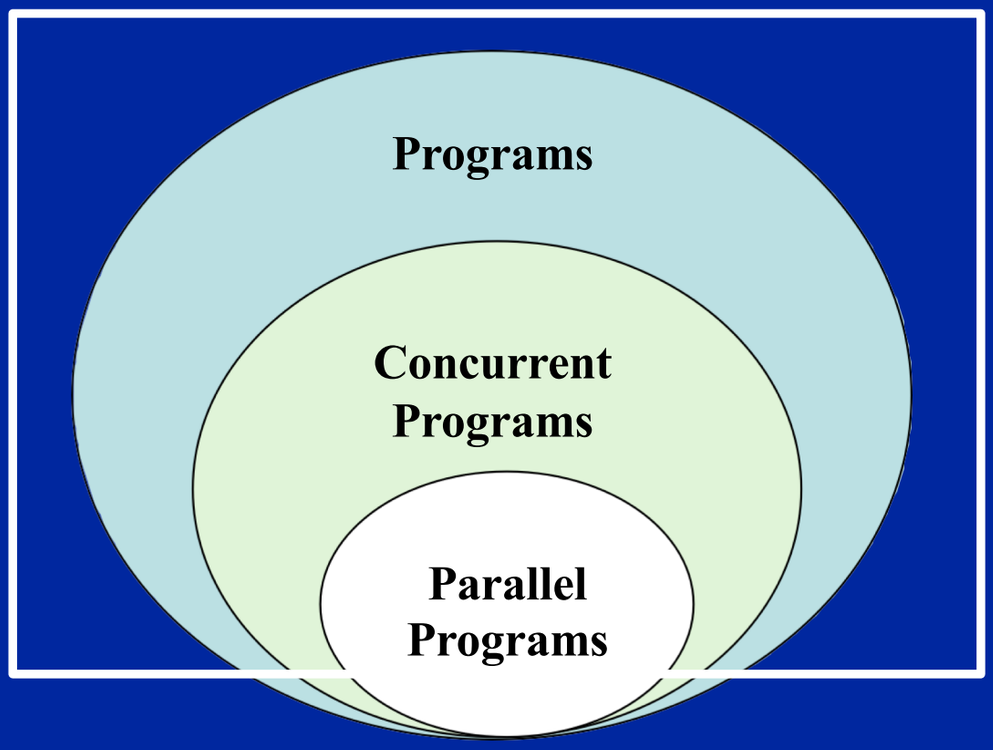


Figure from "An Introduction to Concurrency in Programming Languages" by J. Sottile, Timothy G. Mattson, and Craig E Rasmussen, 2010

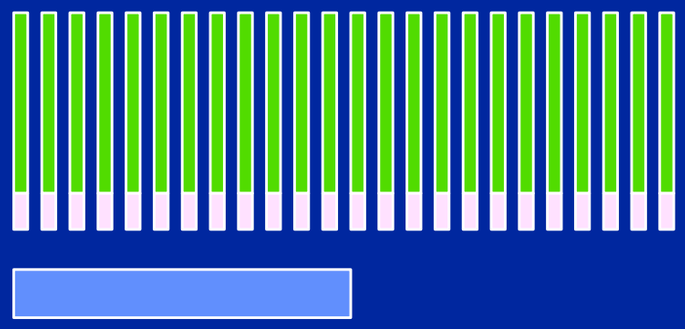
Concurrent vs. Parallel applications

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

The Parallel programming process:

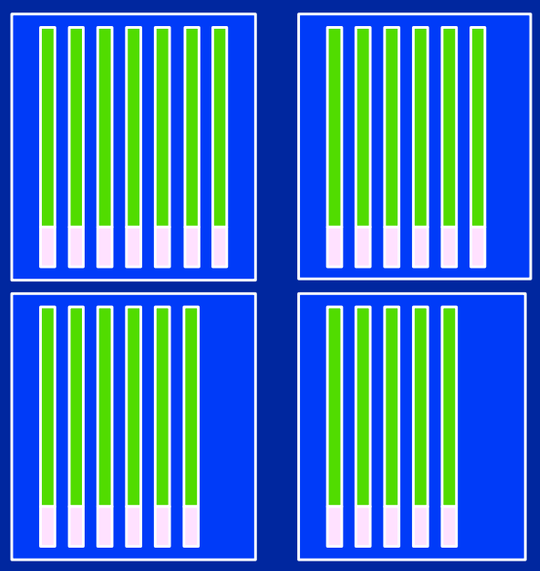
Original Problem

Find Concurrency



Tasks, shared and local data

Algorithm strategy



Units of execution + new shared data for extracted dependencies

Implementation strategy

```
Program SPMD_Emb_Par ()
Program SPMD_Emb_Par ()
Program SPMD_Emb_Par ()
Program SPMD_Emb_Par ()
{
TYPE *tmp, *func();
global_array Data(TYPE);
global_array Res(TYPE);
int Num = get_num_procs();
int id = get_proc_id();
if (id==0) setup_problem(N, Data);
for (int I= ID; I<N;I=I+Num){
tmp = func(I, Data);
Res.accumulate( tmp);
}
}
```

Corresponding source code

OpenMP* Overview:

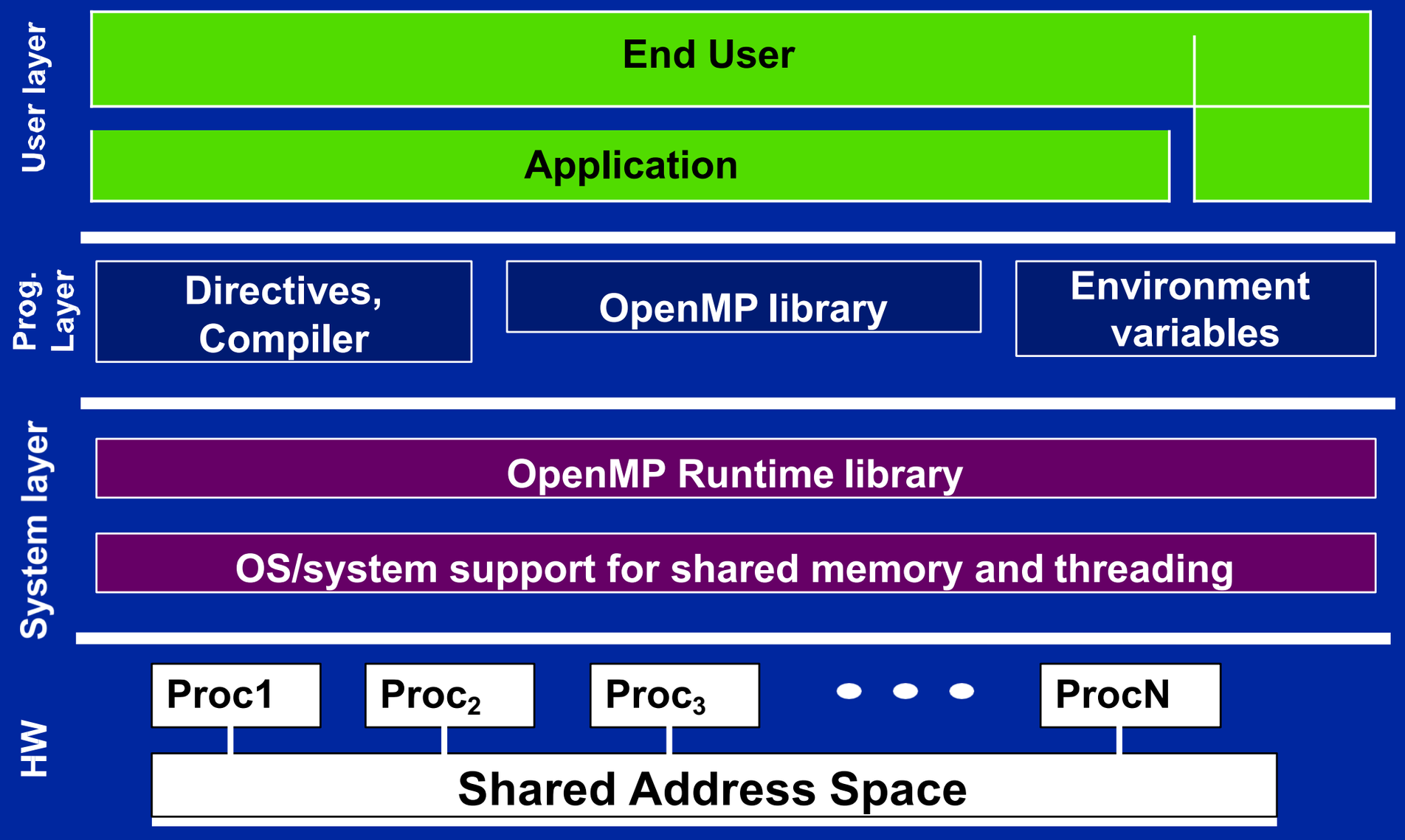
```
C$OMP FLUSH
#pragma omp critical
C$OMP THREADPRIVATE(/ABC/)
CALL OMP_SET_NUM_THREADS(10)
C$OMP PARALLEL COPYIN(/blk/)
C$OMP DO lastprivate(XX)
Nthrds = OMP_GET_NUM_PROCS()
omp_set_lock(lck)
```

OpenMP: An API for Writing Multithreaded Applications

- A set of compiler directives and library routines for parallel application programmers
- Greatly simplifies writing multi-threaded (MT) programs in Fortran, C and C++
- Standardizes last 20 years of SMP practice

* The name "OpenMP" is the property of the OpenMP Architecture Review Board.

OpenMP Basic Defs: Solution Stack



OpenMP core syntax

- Most of the constructs in OpenMP are compiler directives.

  #pragma omp *construct [clause [clause]…]*

  - Example

    #pragma omp parallel num_threads(4)

- Function prototypes and types in the file:

  #include <omp.h>

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

Compiler notes: Other

- Linux and OS X with gcc:

  for the Bash shell

  > gcc -fopenmp foo.c
  > export OMP_NUM_THREADS=4
  > ./a.out

- Linux and OS X with PGI:

  > pgcc -mp foo.c
  > export OMP_NUM_THREADS=4
  > ./a.out

Exercise 1, Part A: Hello world

Verify that your environment works

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

```
int main()
{

    int ID = 0;

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

}
```

Exercise 1, Part B: Hello world

Verify that your OpenMP environment works

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

Linux and OS X	**gcc -fopenmp**
PGI Linux	**pgcc -mp**
Intel windows	**icl /Qopenmp**
Intel Linux and OS X	**icpc –openmp**

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

Exercise 1: Solution

A multi-threaded "Hello world" program

- **Write a multithreaded program where each thread prints "hello world".**

```
#include "omp.h"
int  main()
{

#pragma omp parallel
 {

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

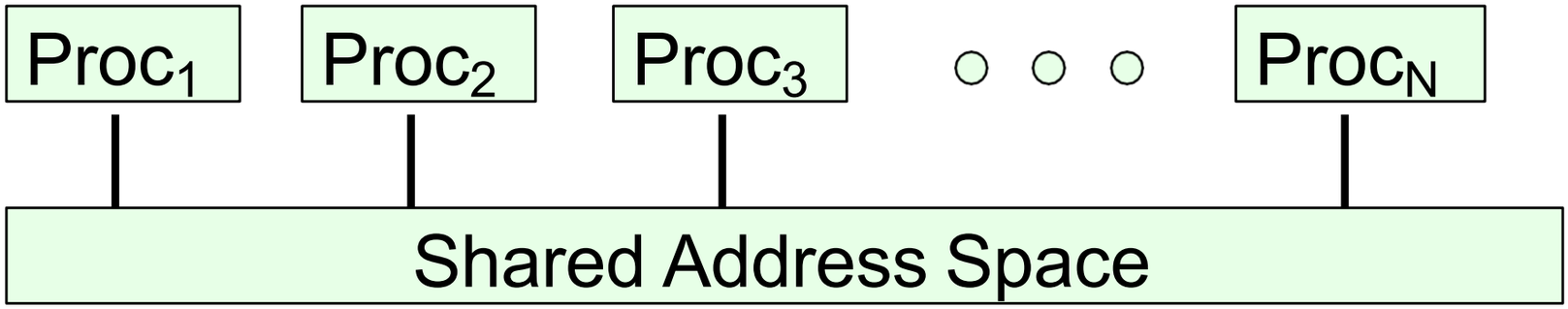
OpenMP include file

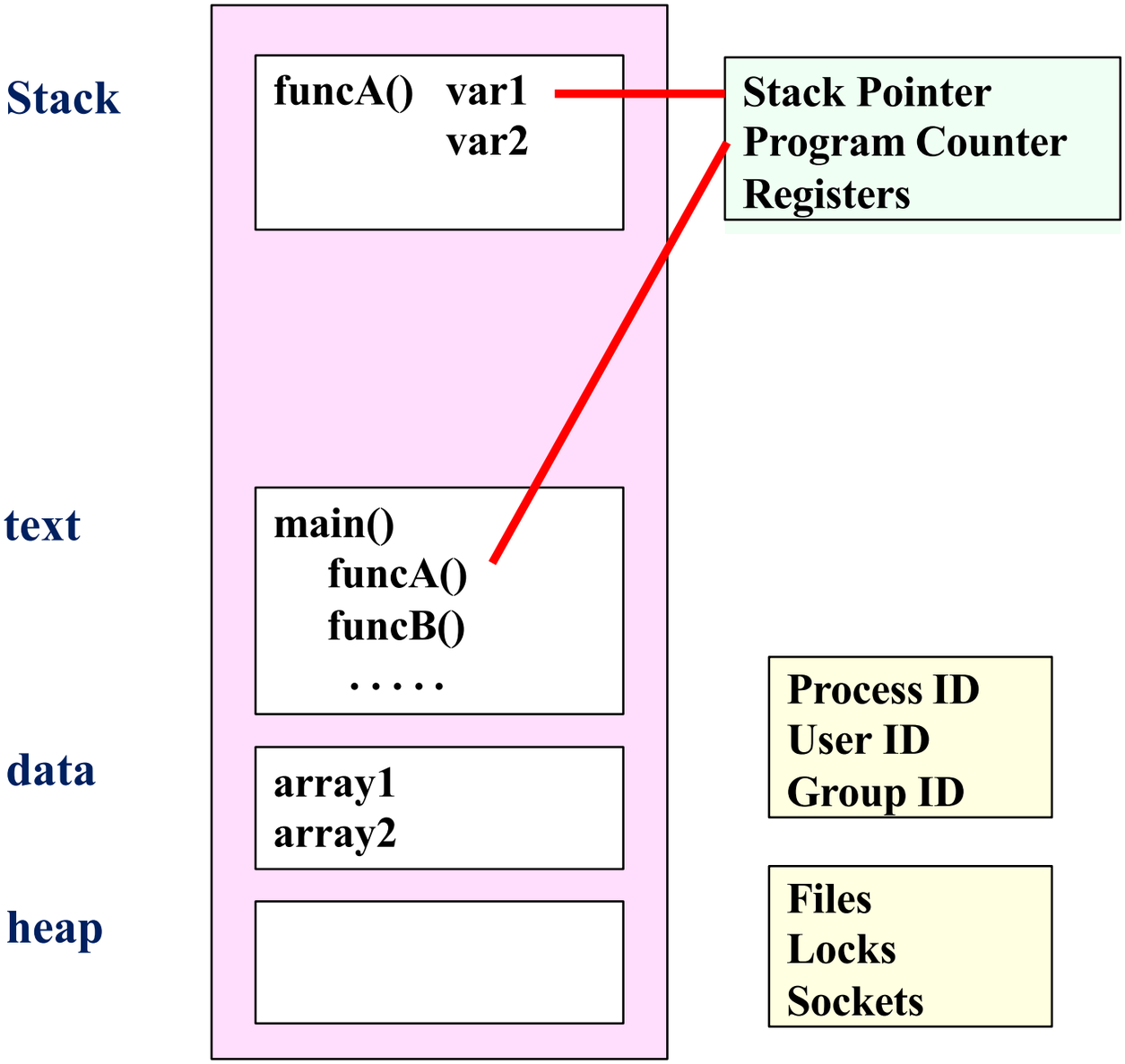
Parallel region with default number of threads

Runtime library function to return a thread ID.

End of the Parallel region

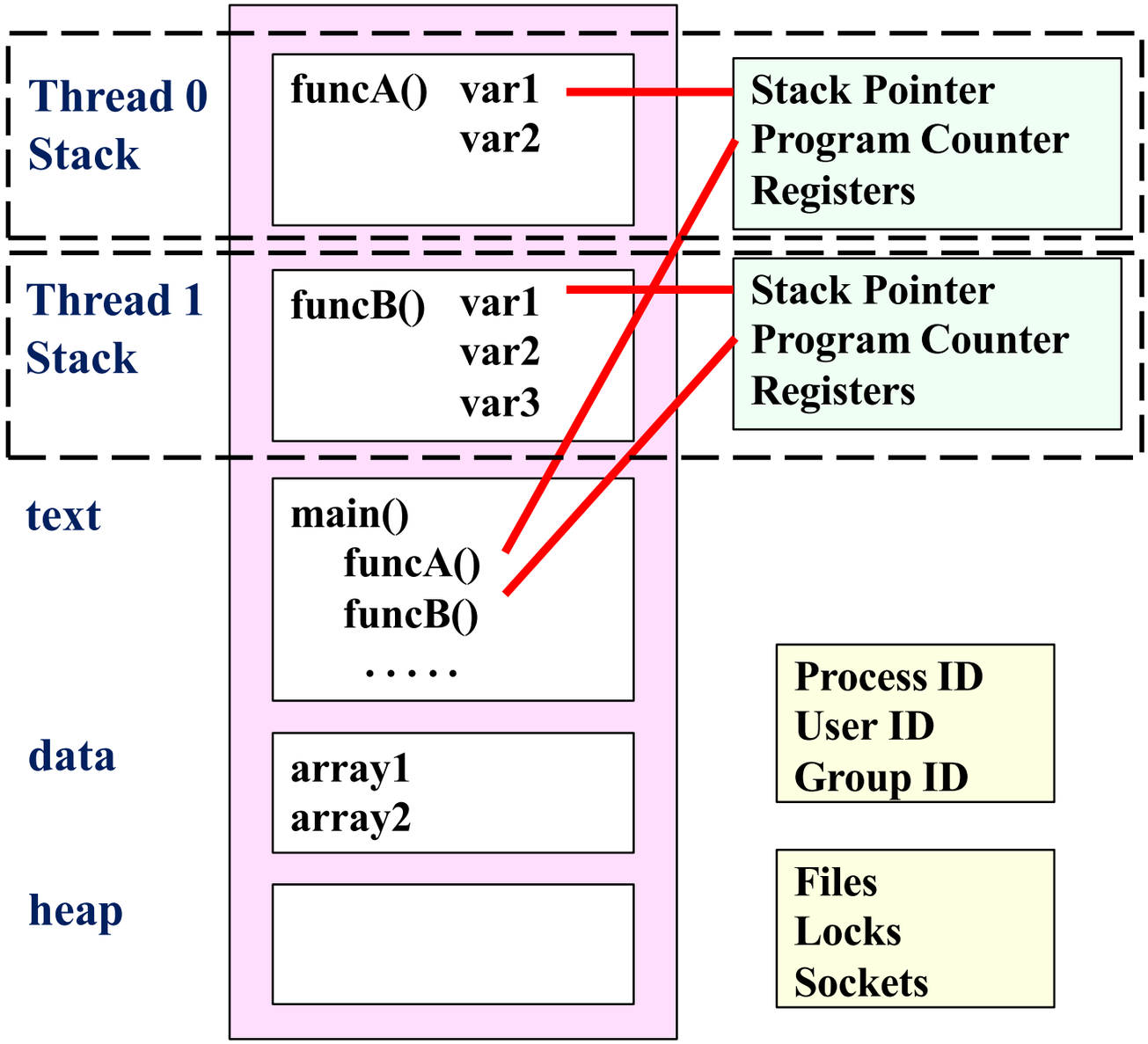
- **Shared memory computer** : any computer composed of multiple processing elements that share an address space. Two Classes:
  - **Symmetric multiprocessor** (**SMP**): a shared address space with "equal-time" access for each processor, and the OS treats every processor the same way.
  - **Non Uniform address space multiprocessor** (**NUMA**): different memory regions have different access costs … think of memory segmented into "Near" and "Far" memory.





Process

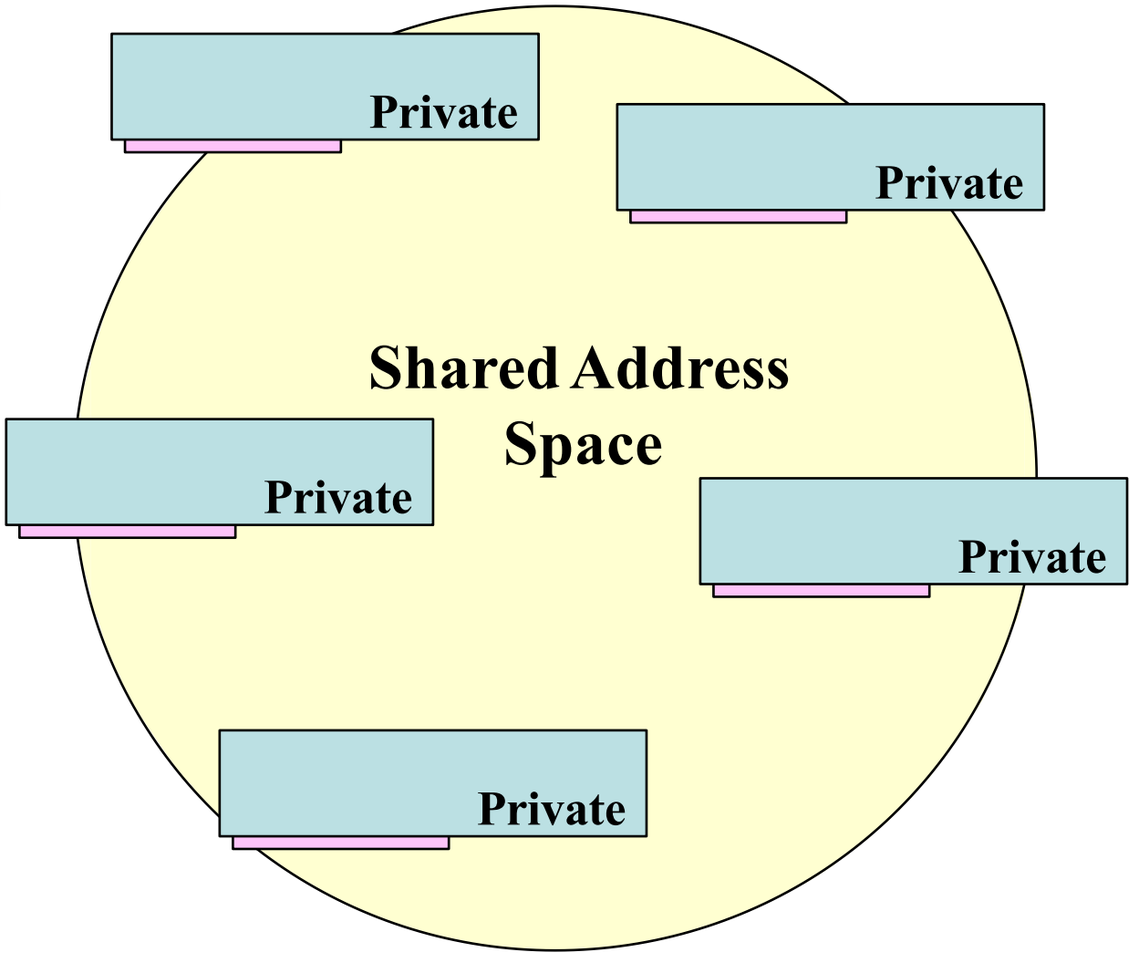
- **An instance of a program execution.**
- **The execution context of a running program … i.e. the resources associated with a program's execution.**



Threads:

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

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



Exercise 1: Solution

A multi-threaded "Hello world" program

- **Write a multithreaded program where each thread prints "hello world".**

```
#include "omp.h"
int  main()
{

#pragma omp parallel
 {

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

OpenMP include file

Parallel region with default number of threads

End of the Parallel region

Runtime library function to return a thread ID.

Sample Output:

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

world(0)

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

world(2)

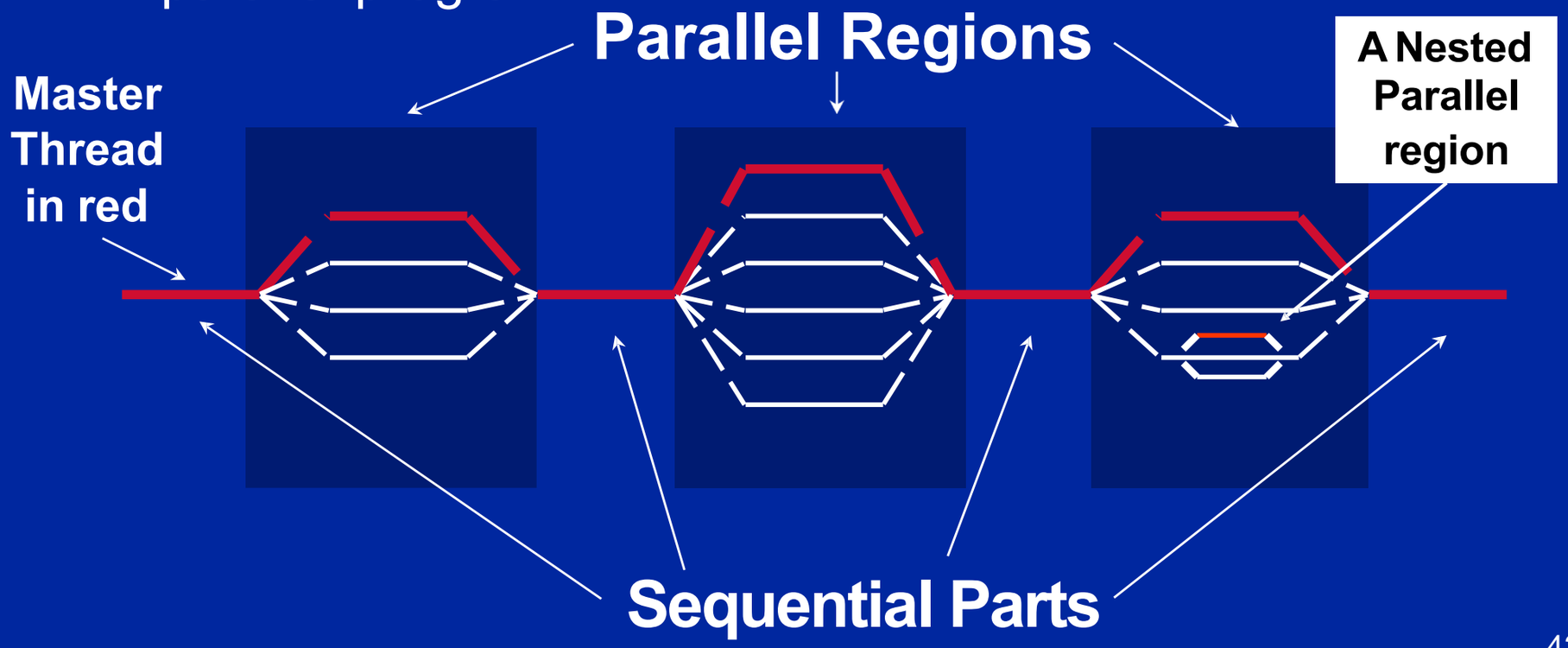
OpenMP Overview:
How do threads interact?

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

OpenMP Programming Model:

Fork-Join Parallelism:

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



Thread Creation: Parallel Regions

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

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

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

Runtime function to request a certain number of threads

Runtime function returning a thread ID

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

Thread Creation: Parallel Regions

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

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

```
double A[1000];

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

clause to request a certain number of threads

Runtime function returning a thread ID

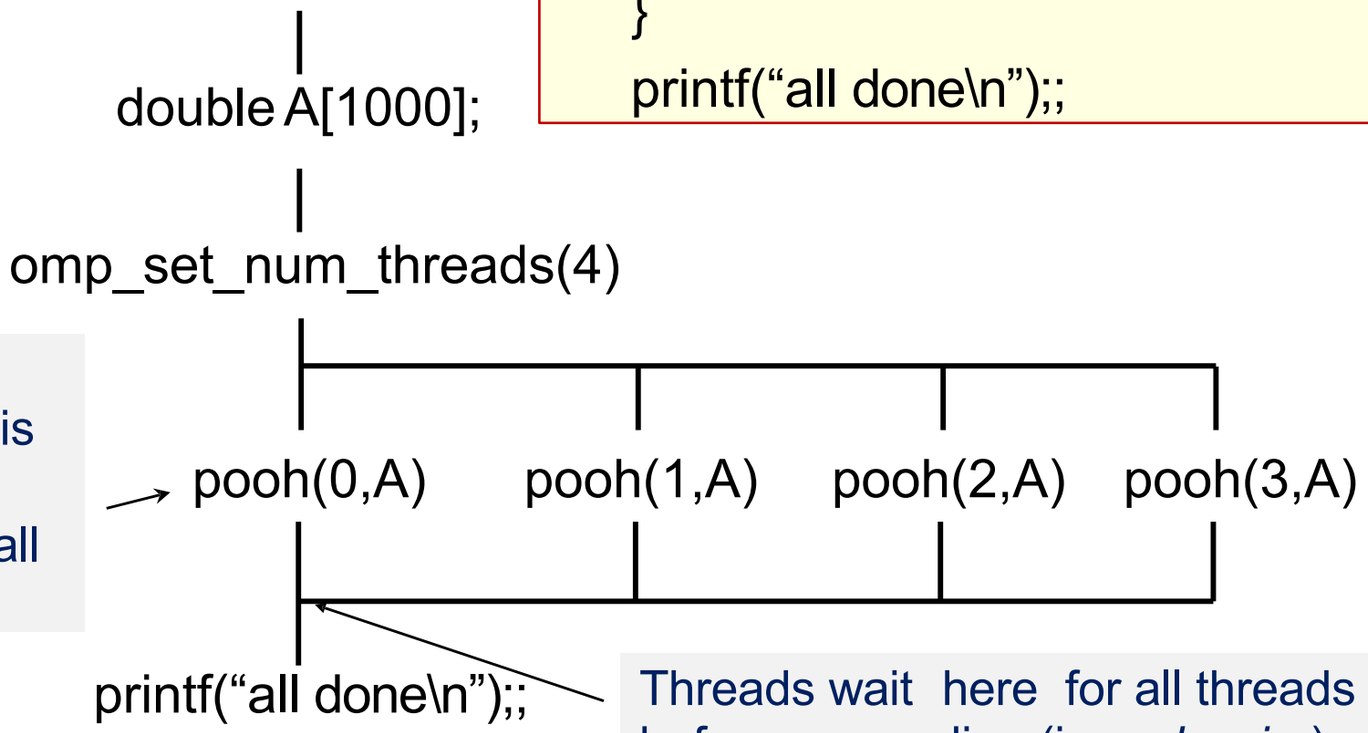
- **Each thread calls `pooh(ID,A)` for `ID` = 0 to 3**

* The name "OpenMP" is the property of the OpenMP Architecture Review Board

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*)

* The name "OpenMP" is the property of the OpenMP Architecture Review Board

OpenMP: what the compiler does

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

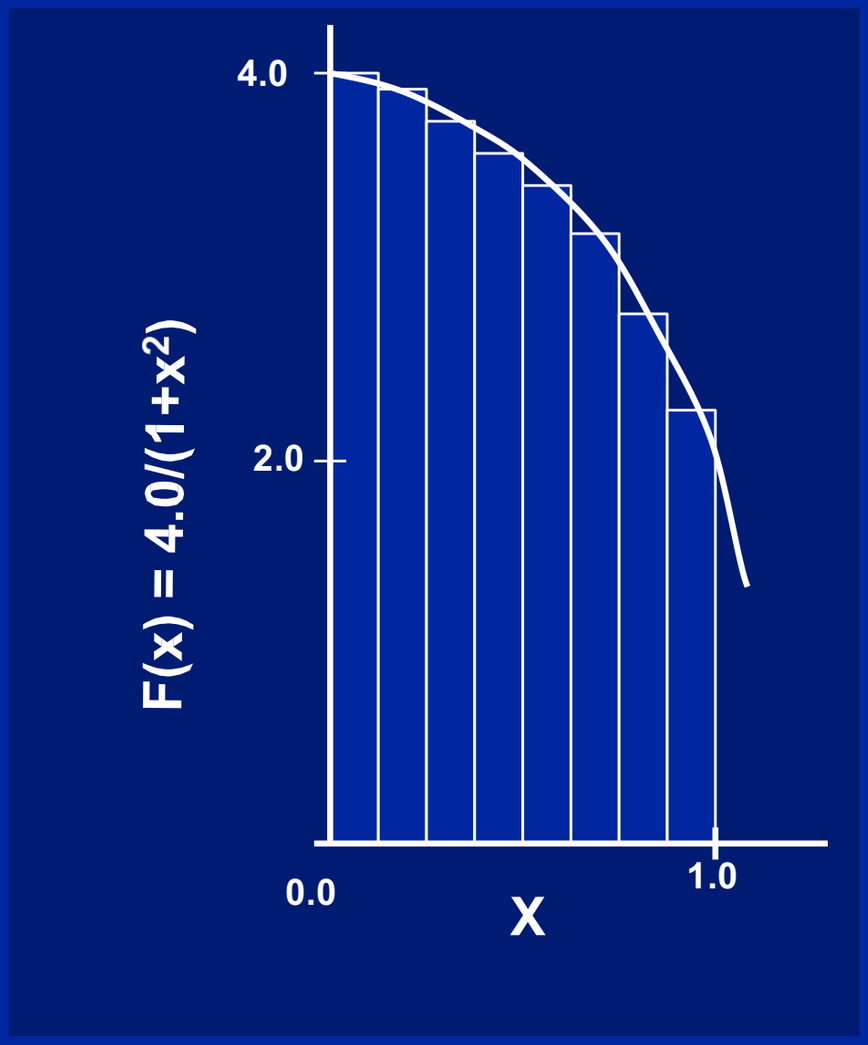
- The OpenMP compiler generates code logically analogous to that on the right of this slide, given an OpenMP pragma such as that on the top-left
- All known OpenMP implementations use a thread pool so full cost of threads creation and destruction is not incurred for reach parallel region.
- Only three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

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

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

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

Exercises 2 to 4:
Numerical Integration



Mathematically, we know that:

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

We can approximate the integral as a sum of rectangles:

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

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

Exercises 2 to 4: Serial PI Program

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

        step = 1.0/(double) num_steps;

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

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

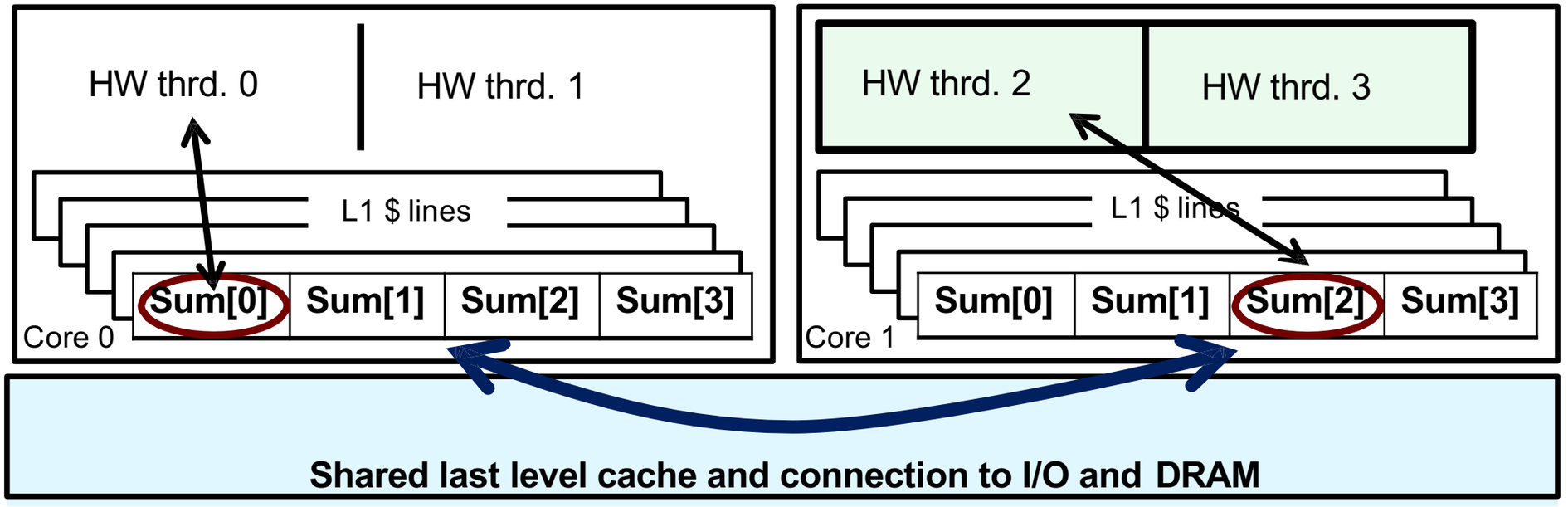
```
#include <omp.h>
static long num_steps = 100000;        double step;
#define NUM_THREADS 2
void main ()
{           int i, nthreads;  double pi, sum[NUM_THREADS];
            step = 1.0/(double) num_steps;
            omp_set_num_threads(NUM_THREADS);
   #pragma omp parallel
   {
            int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

*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.

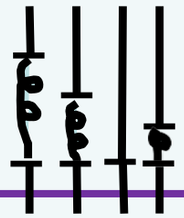
OpenMP Overview:
How do threads interact?

Recall our high level overview of OpenMP?

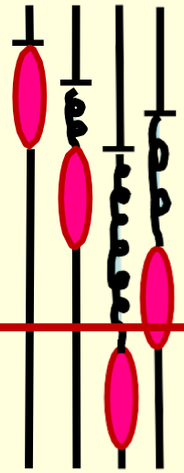
- OpenMP is a multi-threading, shared address model.
  - Threads communicate by sharing variables.
- Unintended sharing of data causes race conditions:
  - race condition: when the program's outcome changes as the threads are scheduled differently.
- To control race conditions:
  - Use synchronization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

tion:

- 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

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

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

Discussed later

Synchronization: Barrier

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

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

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

Synchronization: critical

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

```
float  res;

#pragma omp parallel

{     float B;   int i, id, nthrds;

      id = omp_get_thread_num();

      nthrds = omp_get_num_threads();

      for(i=id;i<niters;i+=nthrds){

            B =  big_job(i);

#pragma omp critical
            res += consume (B);

      }
}
```

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

Synchronization: Atomic (basic form)

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

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

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

- x 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

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

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 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.**

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 = 1000
void main ()
{   int i;          double x, pi, su
    step = 1.0/(double) num_s
    #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	1st SPMD	1st SPMD padded	SPMD critical	PI Loop
1	1.86	1.86	1.87	1.91
2	1.03	1.01	1.00	1.02
3	1.08	0.69	0.68	0.80
4	0.97	0.53	0.53	0.68

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

Loops (cont.)

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

    `omp_set_schedule()`

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

Outline

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

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

no implicit barrier due to nowait

implicit barrier at the end of a parallel region

Master Construct

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

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

Single worksharing Construct

- **The single construct denotes a block of code that is executed by only one thread (not necessarily the master thread).**
- **A barrier is implied at the end of the single block (can remove the barrier with a *nowait* clause).**

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

Sections worksharing Construct

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

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

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

Synchronization: Lock routines

- **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.

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.

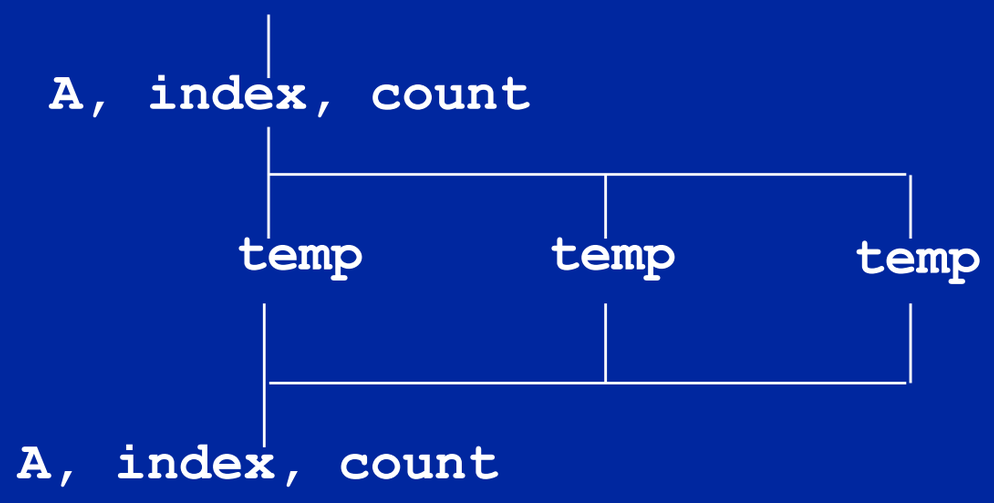
Data sharing: Examples

```
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

Serial PI Program

Now that you understand how to modify the data environment, let's take one last look at our pi program.

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

          step = 1.0/(double) num_steps;

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

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

Example: Pi program … minimal changes

```
#include <omp.h>
static long num_steps = 100000;        double step;

void main ()
{          int i;    double x, pi, sum = 0.0;
           step = 1.0/(double) num_steps;
#pragma omp parallel for private(x) reduction(+:sum)
           for (i=0;i< num_steps; i++){
                      x = (i+0.5)*step;
                      sum = sum + 4.0/(1.0+x*x);
           }
           pi = step * sum;
}
```

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

i private by default

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

Major OpenMP constructs we've covered so far

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

Where variable_list is a comma separated list of variables

Print the value of the macro

_OPENMP

And its value will be

yyyymm

For the year and month of the spec the implementation used

Consider simple list traversal

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

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

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

Exercise 6: linked lists the hard way

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

list traversal

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

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

Linked lists without tasks

- See the file Linked_omp25.c

```
while (p != NULL) {
    p = p->next;
    count++;
}
p = head;
for(i=0; i<count; i++) {
     parr[i] = p;
     p = p->next;
  }
#pragma omp parallel
{
    #pragma omp for schedule(static,1)
    for(i=0; i<count; i++)
       processwork(parr[i]);
}
```

Count number of items in the linked list

Copy pointer to each node into an array

Process nodes in parallel with a for loop

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

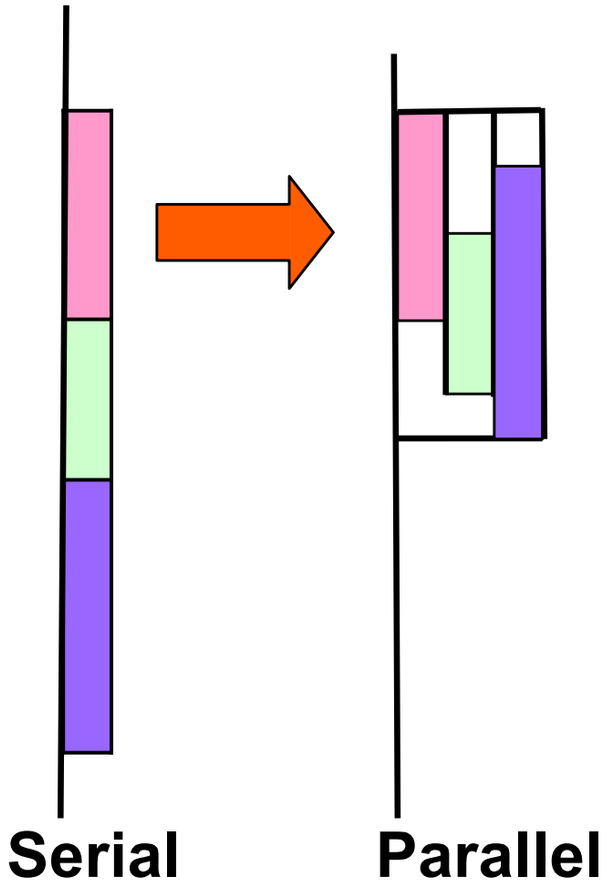
Results on an Intel dual core 1.83 GHz CPU, Intel IA-32 compiler 10.1 build 2

Conclusion

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

OpenMP Tasks

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



Definitions

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

When are tasks guaranteed to complete

- Tasks are guaranteed to be complete at thread barriers:

  #pragma omp barrier

- or task barriers

  #pragma omp taskwait

```
#pragma omp parallel
{
   #pragma omp task
   foo();
   #pragma omp barrier
   #pragma omp single
   {
      #pragma omp task
      bar();
   }
}
```

Multiple foo tasks created here – one for each thread

All foo tasks guaranteed to be completed here

One bar task created here

bar task guaranteed to be completed here

Data Scoping with tasks: Fibonacci example.

This is an instance of the divide and conquer design pattern

```
int fib ( int n )
{

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

n is private in both tasks

x is a private variable
y is a private variable

What's wrong here?

A task's private variables are undefined outside the task

Data Scoping with tasks: Fibonacci example.

```
int fib ( int n )
{

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

n is private in both tasks

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

Data Scoping with tasks: List Traversal example

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

What's wrong here?

Possible data race !
Shared variable e
updated by multiple tasks

Data Scoping with tasks: List Traversal example

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

Good solution – e is firstprivate

Execution of tasks

Have potential to parallelize irregular patterns and recursive function calls

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