

Using OpenMP for Intranode Parallelism

Tutorial Overview

Bronis R. de Supinski

Paul Petersen



Thanks to: Tim Mattson (Intel), Ruud van der Pas (Oracle),
Christian Terboven (RWTH Aachen University), Michael Klemm (Intel)

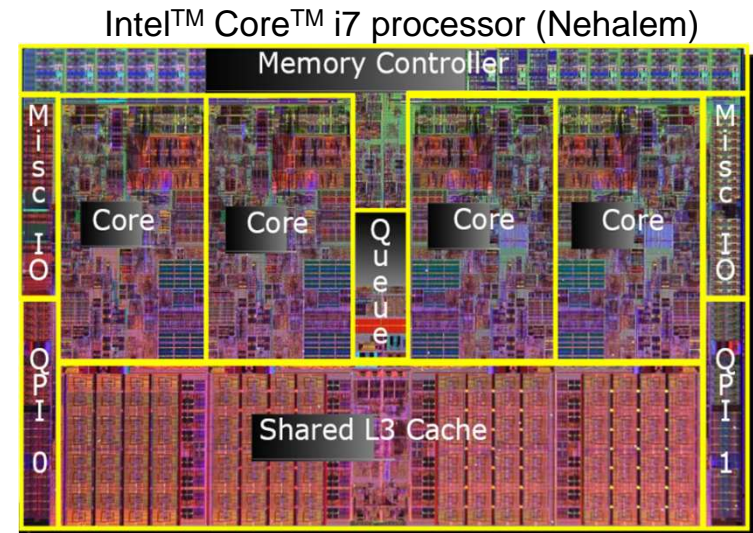
* The name "OpenMP" is the property of the OpenMP Architecture Review Board.

Outline

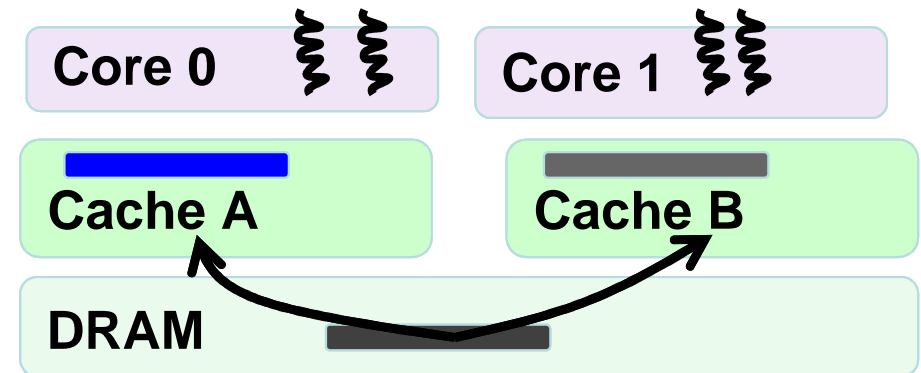
- ➔ • Threaded Programming
 - Introduction to OpenMP
 - Creating Threads
 - Synchronization
 - Parallel Loops
 - Synchronize single masters and stuff
 - Data environment

Assumed background...

- You know about parallel architectures ... multicore chips have made them very common.



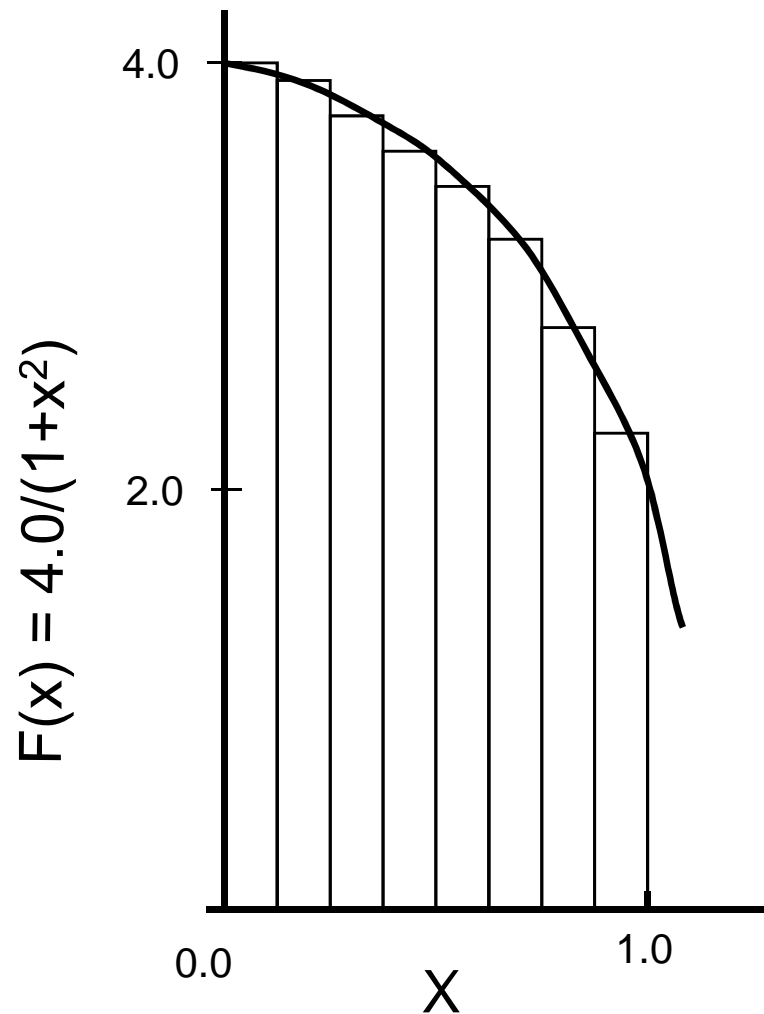
- You know about threads and cache coherent shared address spaces



- ... and you know about the POSIX Threads API (Pthreads) for writing multithreaded programs.

```
#include <pthread.h>
void * thrd_func (void *arg){ // thread entry point
    printf("[%d] Hello, world!\n", *(int*)arg);
}
int main (){
    pthread_t tid[10]; // thread handle
    int thrd_rank[10];
    for (int i = 0; i < 10; ++i){
        thrd_rank[i] = i;
        pthread_create (&tid[i], 0, thrd_func,
                        (void*) &thrd_rank[i]);
    }
}
```

A simple running example: 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 Δx and height $F(x_i)$ at the middle of interval i .

PI Program: Serial version

```
#define NUMSTEPS 100000
double step;
void main ()
{
    int i;    double x, pi, sum = 0.0;

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

PI Program: transform into a Pthreads program

Let's turn this into a parallel program using the Pthreads API.

```
#define NUMSTEPS 100000
double step;
void main ()
{   int i;   double x, pi, sum = 0.0;
```

Variable to accumulate thread results must be shared

Package this into a function

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

Assign loop iterations to threads

```
    x = 0.5 * step;
```

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

```
        x+=step;
```

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

```
    }
```

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

Assure safe update to sum ... correct for any thread schedule

```
}
```

Numerical Integration: PThreads (1 of 2)

Func(): the function run by the threads

```
#include <stdio.h>
#include <pthread.h>
#define NUMSTEPS 10000000
#define NUMTHREADS 4
double step = 0.0, Pi = 0.0;
void *Func(void *pArg)
{
    int myRank = *((int *)pArg);
    double partialSum = 0.0, x;
    for (int i = myRank; i < NUMSTEPS; i += NUMTHREADS)
    {
        x = (i + 0.5f) * step;
        partialSum += 4.0f / (1.0f + x*x);
    }
    pthread_mutex_lock(&gLock);
    Pi += partialSum * Step;
    pthread_mutex_unlock(&gLock);
    return 0;
}
```

Global variables ... on the heap

pthread_mutex_t gLock;

Cyclic loop distribution ... deal out loop iterations as you would a deck of cards

Put any code you want inbetween the Mutex_lock and unlock. This is called a **Critical section** ... only one thread at a time can execute this code

Numerical Integration: PThreads (2 of 2)

The main program ... managing threads

```
int main()
{
    pthread_t thrds[NUMTHREADS];
    int tNum[NUMTHREADS], i;
    pthread_mutex_init(&gLock, NULL);
    Step = 1.0 / NUMSTEPS;
    for ( i = 0; i < NUMTHREADS; ++i )
    {
        tRank[i] = i;
        pthread_create(&thrds[i], NULL, Func, (void)&tRank[i]);
    }
    for ( i = 0; i < NUMTHREADS; ++i )
    {
        pthread_join(thrds[i], NULL);
    }
    pthread_mutex_destroy(&gLock);
    printf("Computed value of Pi: %12.9f\n", Pi );
    return 0;
}
```

Initialize the mutex variable

Create (fork) the threads ...
passing each thread its rank

Post a join for each
thread ... hence waiting
for all of them to finish
before proceeding

The fork-join pattern

- This is an instance of the well known **Fork join pattern**:
 1. Start as a serial program.
 2. When work to do in parallel is encountered, pack it into a function.
 3. Fork a number of threads to execute the function.
 4. When the functions have completed, the threads join back together.
 5. Program continues as a serial program.

Numerical Integration: PThreads (2 of 2)

```
int main()
```

Numerical Integration: PThreads (1 of 2)

```
#include <stdio.h>
#include <pthread.h>
#define NUMSTEPS 10000000
#define NUMTHREADS 4
double gStep = 0.0, gPi = 0.0; pthread_mutex_t gLock;
void *Func(void *pArg)
{
    int myRank = *((int *)pArg);
    double partialSum = 0.0, x;
    for (int i = myRank; i < NUMSTEPS; i += NUMTHREADS)
    {
        x = (i + 0.5f) * gStep;
        partialSum += 4.0f / (1.0f + x*x);
    }
    pthread_mutex_lock(&gLock);
    gPi += partialSum * gStep;
    pthread_mutex_unlock(&gLock);

    return 0;
}
```

Global variables ... on the heap

Cyclic loop distribution ... deal out loop iterations as you would a deck of cards

Put any code you want inbetween the Mutex lock and unlock. This is called a **Critical section** ... only one thread at a time can execute this code

Source: Michael Wrinn of Intel

- If this pattern with such “mechanical” transformations is so common, can’t we come up with an easier, less intrusive way for this style of programming?
- Yes we can ... and its called OpenMP

Outline

- Threaded Programming
- ➔ • Introduction to OpenMP
- Creating Threads
- Synchronization
- Parallel Loops
- Synchronize single masters and stuff
- Data environment

OpenMP* Overview:

<http://www.openmp.org>

```
C$OMP FLUSH
```

```
#pragma omp critical
```

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

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

```
C$OM
```

OpenMP: An API for Writing Multithreaded Applications

```
C$OM
```

```
C$O
```

```
C
```

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

```
ED
```

```
#pragma omp parallel for private(A, B)
```

```
!$OMP BARRIER
```

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

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

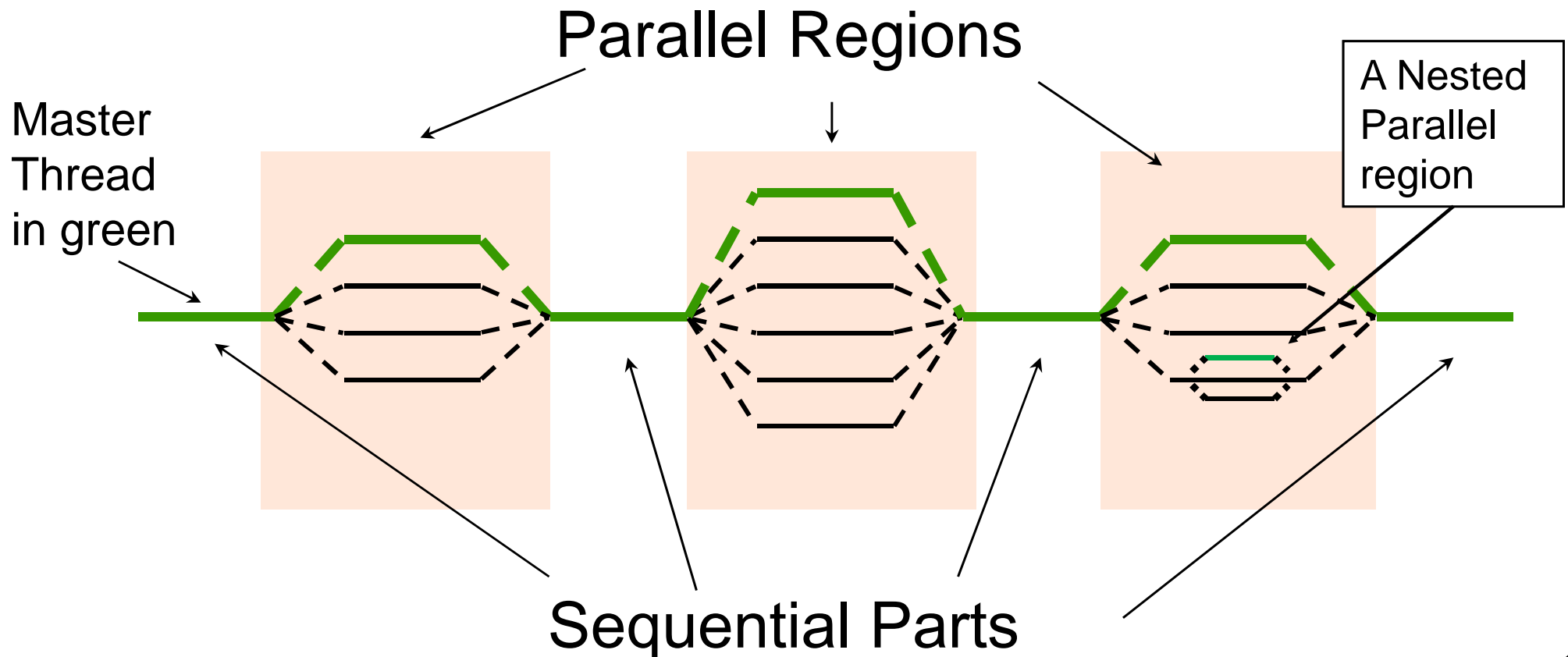
```
Nthrds = OMP_GET_NUM_PROCS()
```

```
omp_set_lock(lck)
```

Original OpenMP Execution Model:

Fork-Join pattern:

- ◆ **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.



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:

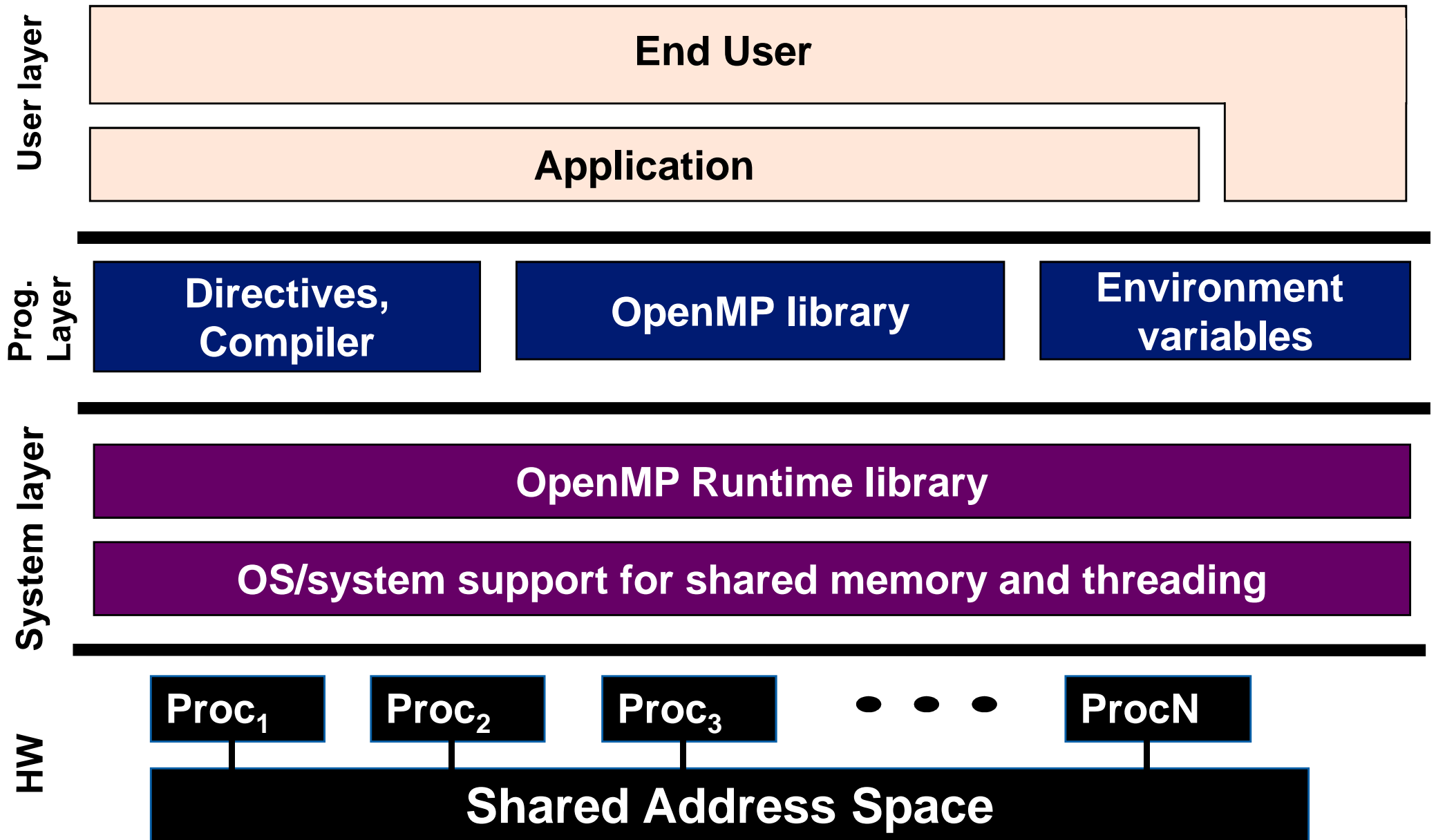
#include <omp.h>

- 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.

OpenMP Basic Defs: Solution Stack

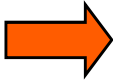


OpenMP Overview:

How do threads interact?

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

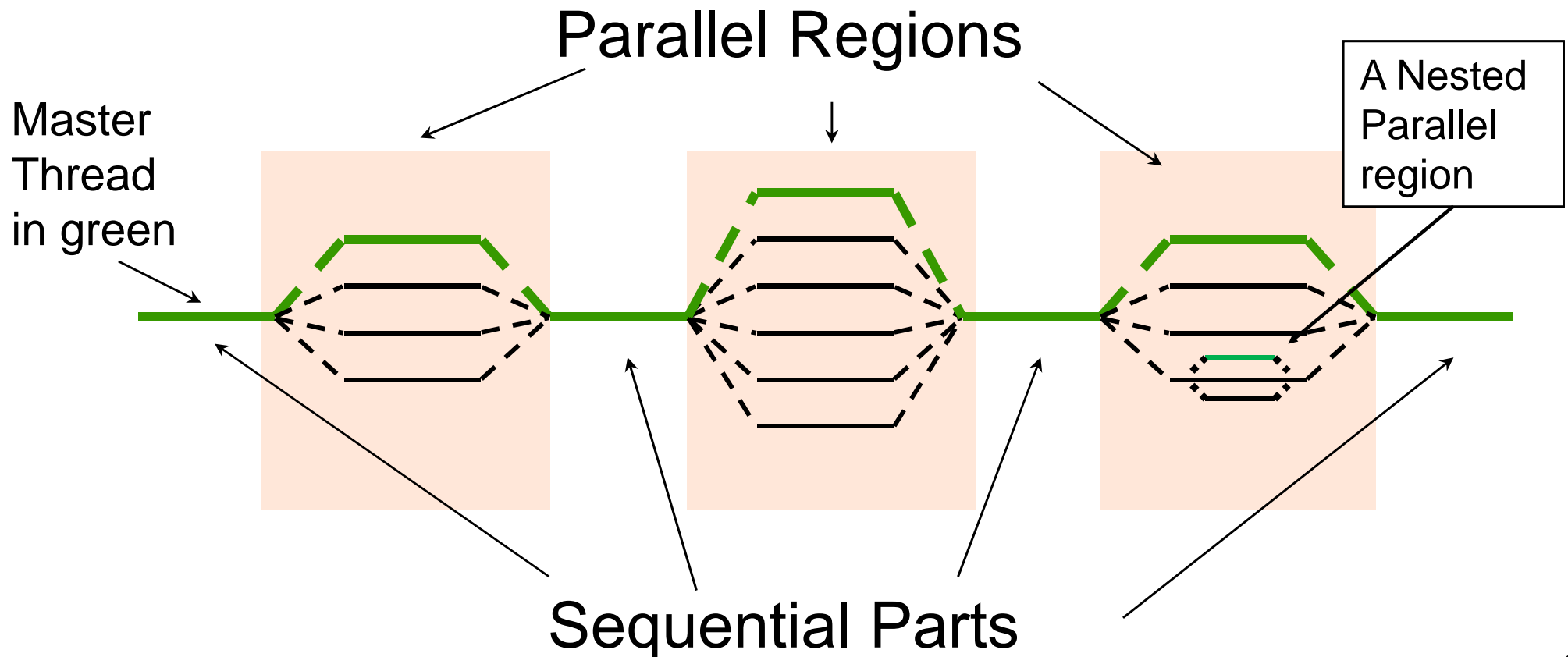
Outline

- Threaded Programming
- Introduction to OpenMP
-  • Creating Threads
- Synchronization
- Parallel Loops
- Synchronize single masters and stuff
- Data environment

Original OpenMP Execution Model:

Fork-Join pattern:

- ◆ **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);  
}  
  
printf("all done\n");
```

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

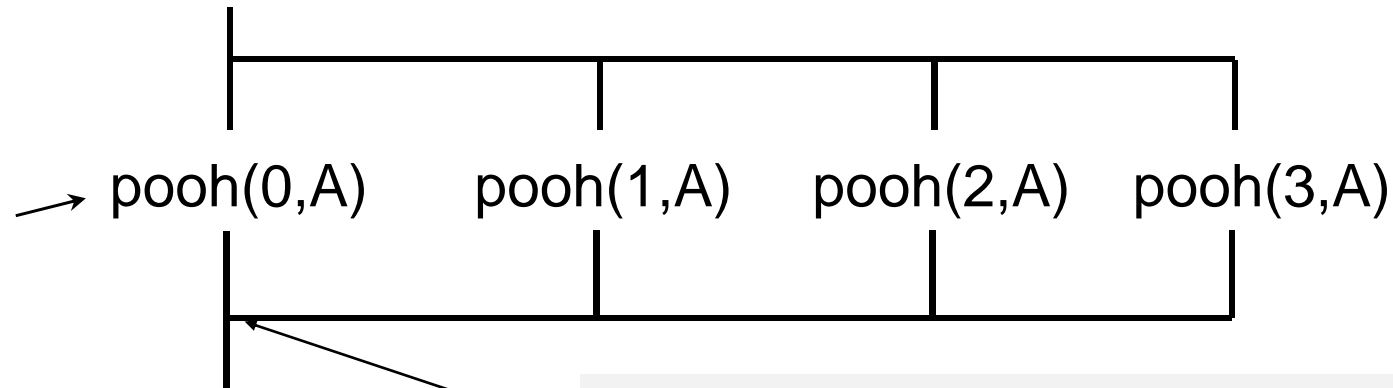
- Each thread executes the same code redundantly.

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

double A[1000];

omp_set_num_threads(4)

A single copy of A is shared between all threads.



printf("all done\n");

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

OpenMP: What the compiler does

Input

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

- All known OpenMP implementations use a thread pool so full cost of thread 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.

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

Output (approximate)

```
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]);
```

Exercise

- Create a parallel version of the pi program using a parallel construct.
- Pay close attention to shared versus private variables.
- In addition to a parallel construct, you will need the runtime library routines

– int omp_get_num_threads();

Number of threads in the team

– void omp_set_num_threads();

– int omp_get_thread_num();

Thread ID or rank

– double omp_get_wtime();

Time in seconds since a fixed point in the past

Example: Serial PI Program

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

    step = 1.0/(double) num_steps;

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

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

SPMD: Single Program Multiple Data

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

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

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

Results*

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

Example: A simple Parallel pi program

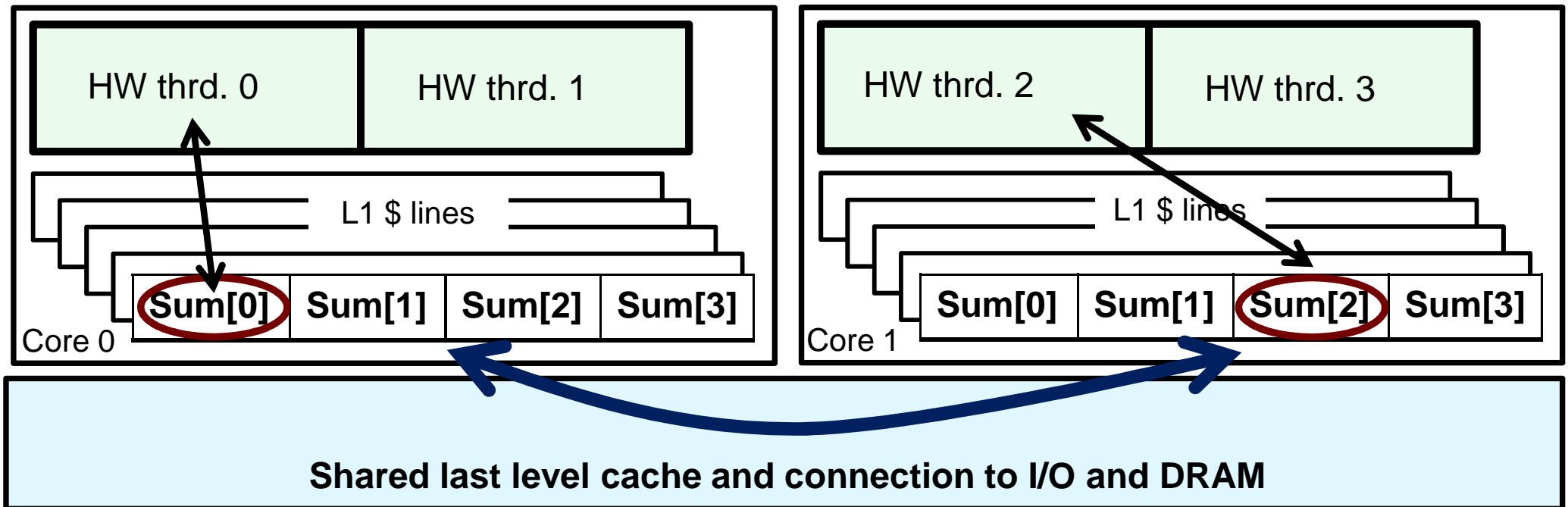
```
#include <omp.h>
static long num_steps = 100000;    double step;
#define NUM_THREADS 2
void main ()
{
    int i, nthreads; double pi, sum[NUM_THREADS];
    step = 1.0/(double) num_steps;
    omp_set_num_threads(NUM_THREADS);
    #pragma omp parallel
    {
        int i, id, nthrds;
        double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

threads	1 st 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

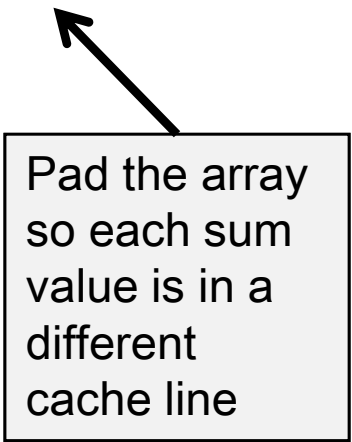
- If independent data elements happen to sit on the same cache line, each update will cause the cache lines to “slosh back and forth” between threads ... This is called **“false sharing”**.



- 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.

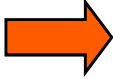
Example: eliminate False sharing by padding the sum array

```
#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) nthrds = nthrds;
        for (i=id, sum[id]=0.0; i< num_steps; i=i+nthrds) {
            x = (i+0.5)*step;
            sum[id][0] += 4.0/(1.0+x*x);
        }
    }
    for(i=0, pi=0.0; i<nthreads; i++) pi += sum[i][0] * step;
}
```

threads	1 st SPMD	1 st 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.

Outline

- Threaded Programming
- Introduction to OpenMP
- Creating Threads
-  • Synchronization
- Parallel Loops
- Synchronize single masters and stuff
- Data environment

Synchronization

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


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

Discussed later

Synchronization: critical

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

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



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

Synchronization: Atomic (basic form)

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

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

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

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

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

Synchronization: Barrier

Preview of worksharing construct... more later

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

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

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

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

```
#pragma omp barrier
```

```
#pragma omp for
```

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

```
#pragma omp for nowait
```

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

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

```
}
```

implicit barrier at the end of a for worksharing construct

implicit barrier at the end of a parallel region

no implicit barrier due to nowait

Exercise

- Earlier, we used an array to create space for each thread to store its partial sum.
- If array elements happen to share a cache line, this leads to false sharing.
 - Non-shared data in the same cache line so each update invalidates the cache line ... in essence “sloshing independent data” back and forth between threads.
- Modify your “pi program” from before to avoid false sharing due to the sum array using a critical section.

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();
        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_THREADS2
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();
        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 st SPMD	1 st 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.

Outline

- Threaded Programming
- Introduction to OpenMP
- Creating Threads
- Synchronization
- ➔ • Parallel Loops
- Synchronize single masters and stuff
- Data environment

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++){  
        a[i] = a[i] + b[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 for schedule(static)

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];}
```


Combined parallel/worksharing construct

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

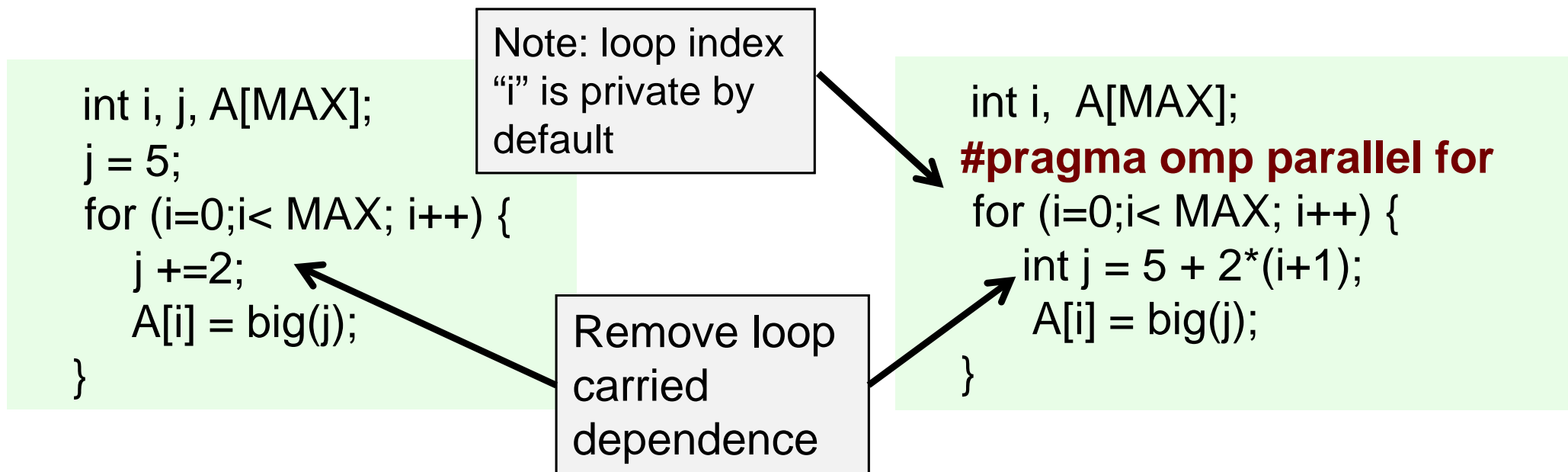
```
double res[MAX]; int i;  
#pragma omp parallel  
{  
    #pragma omp for  
    for (i=0;i< MAX; i++) {  
        res[i] = huge();  
    }  
}
```

```
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



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 number
Max	Smallest number

C/C++ only	
Operator	Initial value
&	~0
 	0
^	0
&&	1
 	0

Fortran Only	
Operator	Initial value
.AND.	.true.
.OR.	.false.
.NEQV.	.false.
.IEOR.	0
.IOR.	0
.IAND.	All bits on
.EQV.	.true.

Exercise: Pi with loops

- Go back to the serial pi program and parallelize it with a loop construct
- Your goal is to minimize the number of changes made to the serial program.

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;
}
```

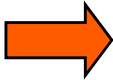
Results*: pi with a loop and a reduction

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

Example: Pi with a <pre> #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; } </pre>	threads	1 st SPMD	1 st 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.

Outline

- Threaded Programming
- Introduction to OpenMP
- Creating Threads
- Synchronization
- Parallel Loops
-  • Synchronize single masters and stuff
- Data environment

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();
}
```

Synchronization: ordered

- The **ordered** region executes in the sequential order.

```
#pragma omp parallel private (tmp)
#pragma omp for ordered reduction(+:res)

    for (l=0;l<N;l++){
        tmp = NEAT_STUFF(l);
#pragma ordered
        res += consum(tmp);
    }
```

Synchronization: Lock routines

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

- Simple Lock routines:

- A simple lock is available if it is unset.

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

- Nested Locks

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

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

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

Synchronization: Simple locks

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

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

#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
- Control how “omp for schedule(RUNTIME)” loop iterations are scheduled.
 - **OMP_SCHEDULE** “schedule[, chunk_size]”

Outline

- Threaded Programming
- Introduction to OpenMP
- Creating Threads
- Synchronization
- Parallel Loops
- Synchronize single masters and stuff
- ➔ • Data environment

Data environment: Default storage attributes

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

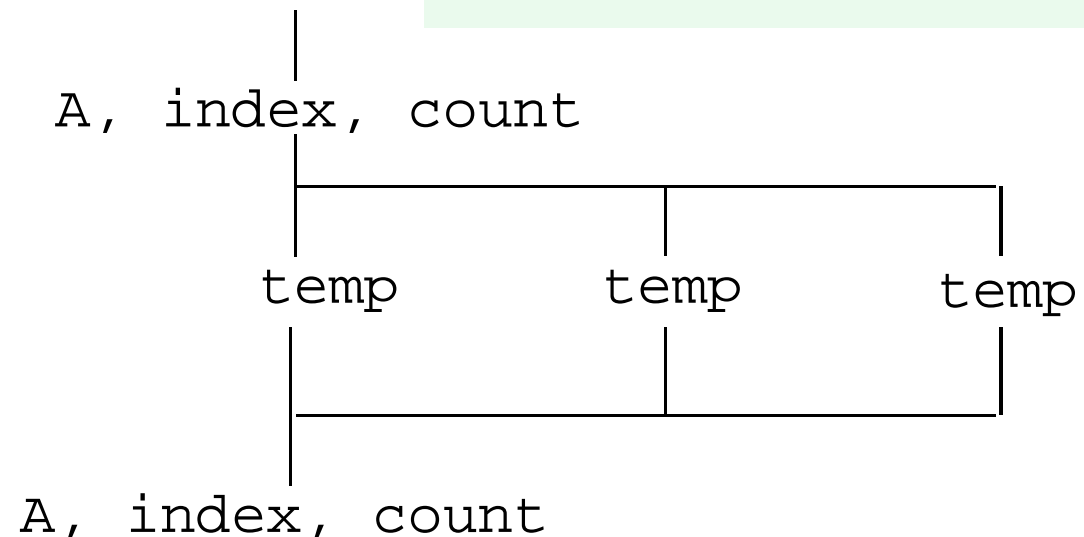
Data sharing: Examples

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

A, index and count are shared by all threads.

temp is local to each thread

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



Data sharing: Changing storage attributes

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

- SHARED
- PRIVATE
- FIRSTPRIVATE

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 is undefined
(may be 0) here

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

tmp was not
initialized

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 $i=(n-1)$)

Example: Pi program ... minimal changes

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

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

```
void main ()
```

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

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

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

i private by default

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

```
}
```

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

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

Conclusion

- OpenMP is one of the simplest APIs available for programming shared memory machines.
 - This simplicity means you can focus on mastering the key design patterns and applying them to your own problems
- We covered the following essential parallel programming design patterns:
 - Fork join
 - SPMD
 - Loop level parallelism
- Next steps?
 - Let's consider some of the more advanced features of OpenMP.
 - OpenMP is in active evolution to target the latest machine architectures.
 - Start writing parallel code ... you can only learn this stuff by writing lots of code.