Parallel Programming

Shared-Memory Programming with OpenMP

(Part I)

Professor Yi-Ping You (游逸平)
Department of Computer Science
http://www.cs.nctu.edu.tw/~ypyou/

Outline

- Introduction to OpenMP
- Compiler Directives
  - Creating Threads
  - Synchronization
  - Worksharing Constructs
  - More on Synchronization
  - Variable Scopes (`shared`/`private`)
- Run-time Libraries and Environment Variables

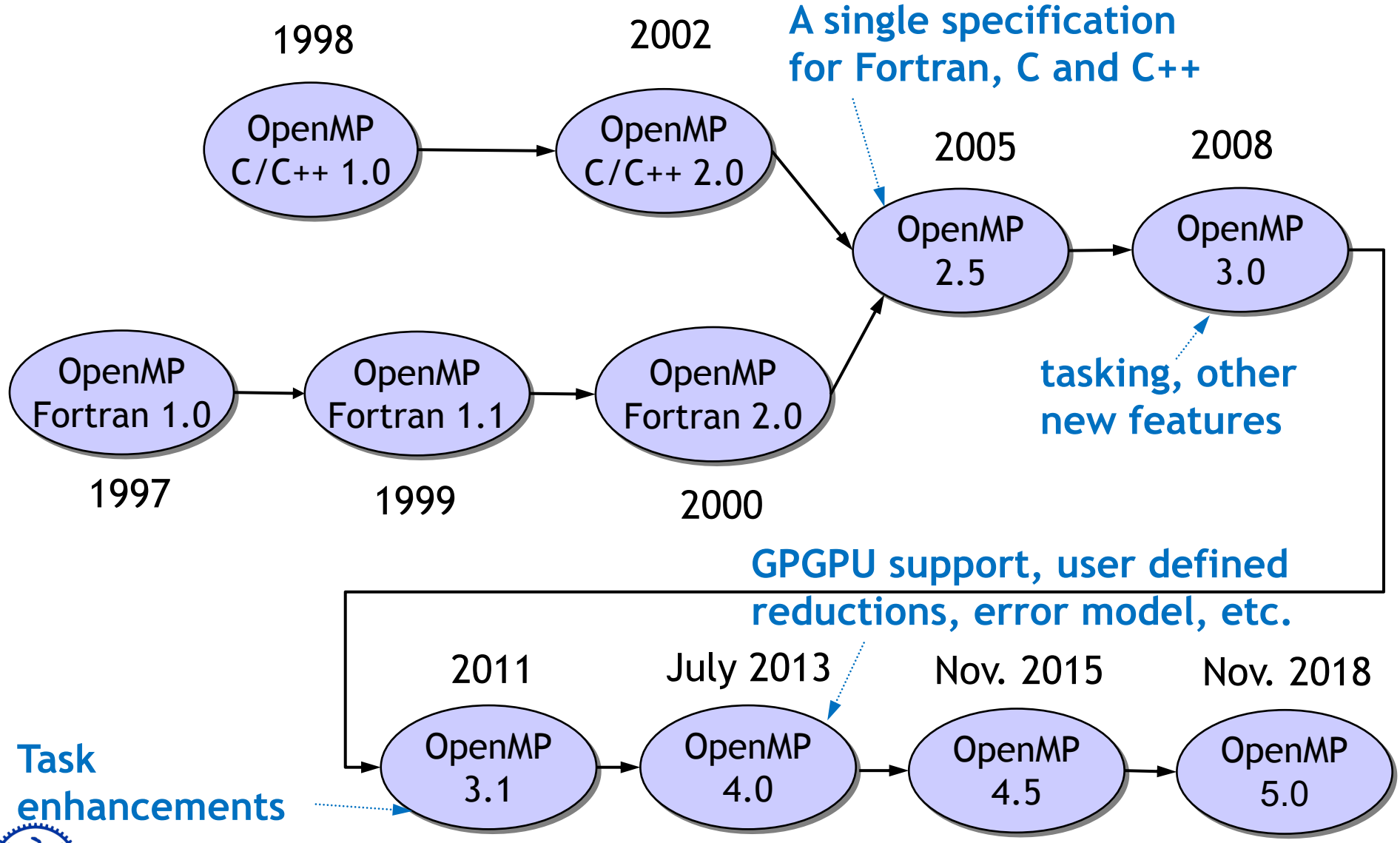
Motivation

- Thread libraries are hard to use
  - Pthreads threads have many library calls for initialization, synchronization, thread creation, condition variables, etc.
  - Programmer must code with multiple threads in mind
- Synchronization between threads introduces a new dimension of program correctness
- Wouldn't it be nice to write serial programs and somehow parallelize them "automatically"?
  - OpenMP can parallelize many serial programs with relatively few annotations that specify parallelism and independence
  - It is not automatic: you can still make errors in your annotations

Introduction to OpenMP

- What is OpenMP?
  - Open specification for Multi-Processing
  - "Standard" API for defining multi-threaded shared-memory programs
    - Standardizes loop-level & task parallelism
    - Standardizes ~ 20 years of compiler-directed threading experience
- High-level API
  - Preprocessor (compiler) directives (~ 80%)
  - Library Calls (~ 19%)
  - Environment Variables (~ 1%)

OpenMP Release History



A Programmer's View of OpenMP

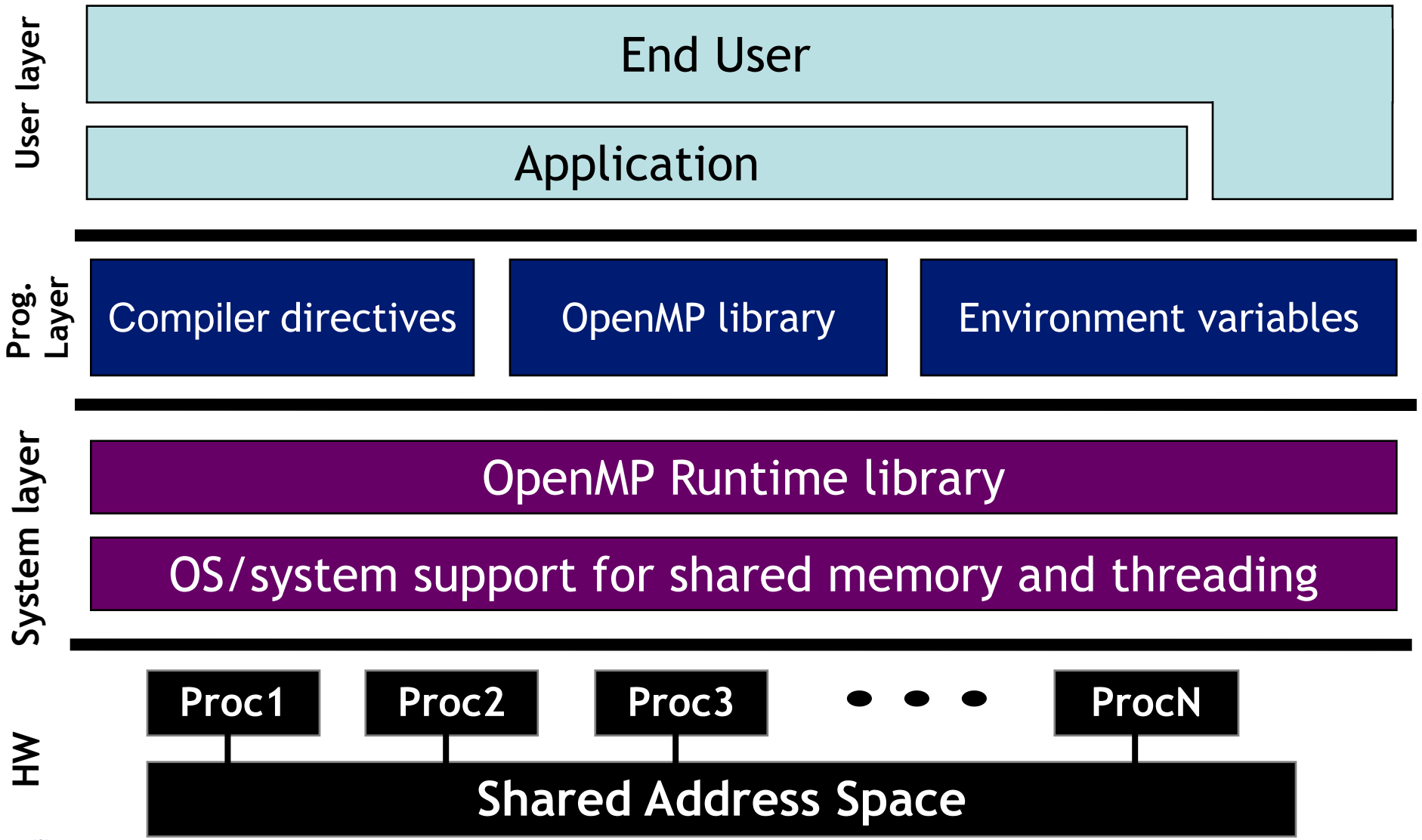
- OpenMP is a portable, threaded, shared-memory programming specification with "light" syntax
  - Exact behavior depends on OpenMP implementation!
  - Requires compiler support (C/C++ or Fortran)
- OpenMP will:
  - Allow programmers to incrementally parallelize existing serial programs
  - Allow a programmer to separate a program into serial regions and parallel regions, rather than T concurrently-executing threads
  - Hide stack management
  - Provide synchronization constructs
- OpenMP will NOT:
  - Parallelize automatically
  - Guarantee speedup
  - Provide freedom from data races

What is OpenMP?

- Three components:

1. Set of compiler directives (80%) for
   - creating teams of threads
   - sharing the work among threads
   - synchronizing the threads
2. Library routines (19%) for setting and querying thread attributes
3. Environment variables (1%) for controlling run-time behavior of the parallel program

OpenMP Basic Defs: Solution Stack



OpenMP Core Syntax

- Most of the constructs in OpenMP are compiler directives
  - **`#pragma omp construct [clause [clause]…]`**
  - E.g., `#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 (single-entry, single-exit)
    - It's OK to have an exit() within the structured block

OpenMP Example: Hello World

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

```
int thread_count = 3;
void main() {



    printf(“Hello from thread %d of %d”,
      ID, thread_count);

}
```

OpenMP Example: Hello World

- Tell the compiler to pack code into a function, fork the threads, and join when done …

```
#include "omp.h"
int thread_count = 3;
void main() {
#pragma omp parallel
   {
      int ID = omp_get_thread_num();
      printf("Hello from thread %d of %d",
         ID, thread_count);
   }
}
```

OpenMP header file

Parallel region with default number of threads

Runtime library function to return a thread ID

End of the parallel region

Sample Output:
Hello from thread 0 of 3
Hello from thread 1 of 3
Hello from thread 2 of 3

Compilation

- To compile with gcc, we need to include the `-fopenmp` option
  - `$ gcc -fopenmp -o omp_hello omp_hello.c`
- Options vary among different compilers

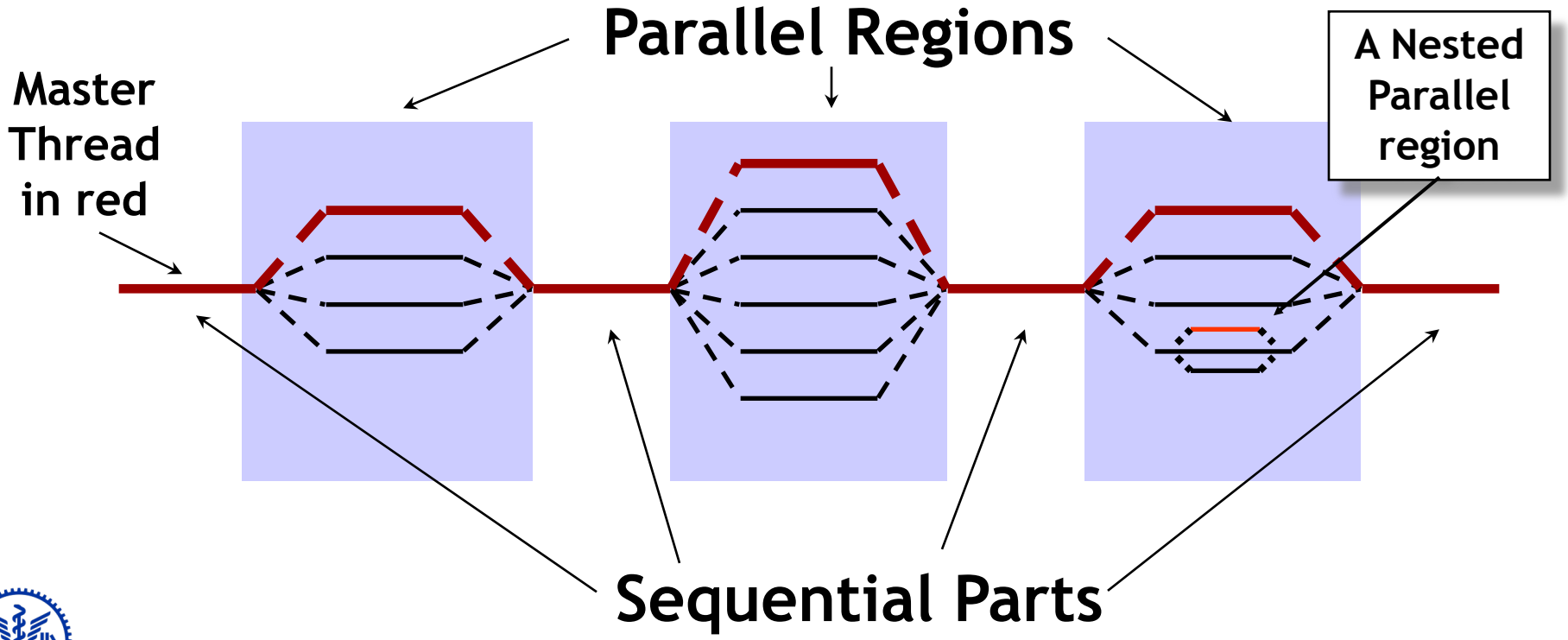
Platform	Compile option
Linux and OS X	gcc -fopenmp
PGI Linux	pgcc -mp
Intel windows	icl /Qopenmp
Intel Linux and OS X	icpc -openmp

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 Execution Model

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



Outline

- Introduction to OpenMP
- Compiler Directives
  - Creating Threads
  - Synchronization
  - Worksharing Constructs
  - More on Synchronization
  - Variable Scopes (`shared/private`)
- Run-time Libraries and Environment Variables

Thread Creation: Parallel Regions (1/2)

- You create threads in OpenMP with the parallel construct
- For example, to create a 4-thread parallel region:

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

Runtime function to request a certain number of threads

Runtime library function to return a thread ID

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

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

Thread Creation: Parallel Regions (2/2)

- You create threads in OpenMP with the parallel construct
- For example, to create a 4-thread parallel region:

```
double A[1000];

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

Clause to request a certain number of threads

Runtime library function to return a thread ID

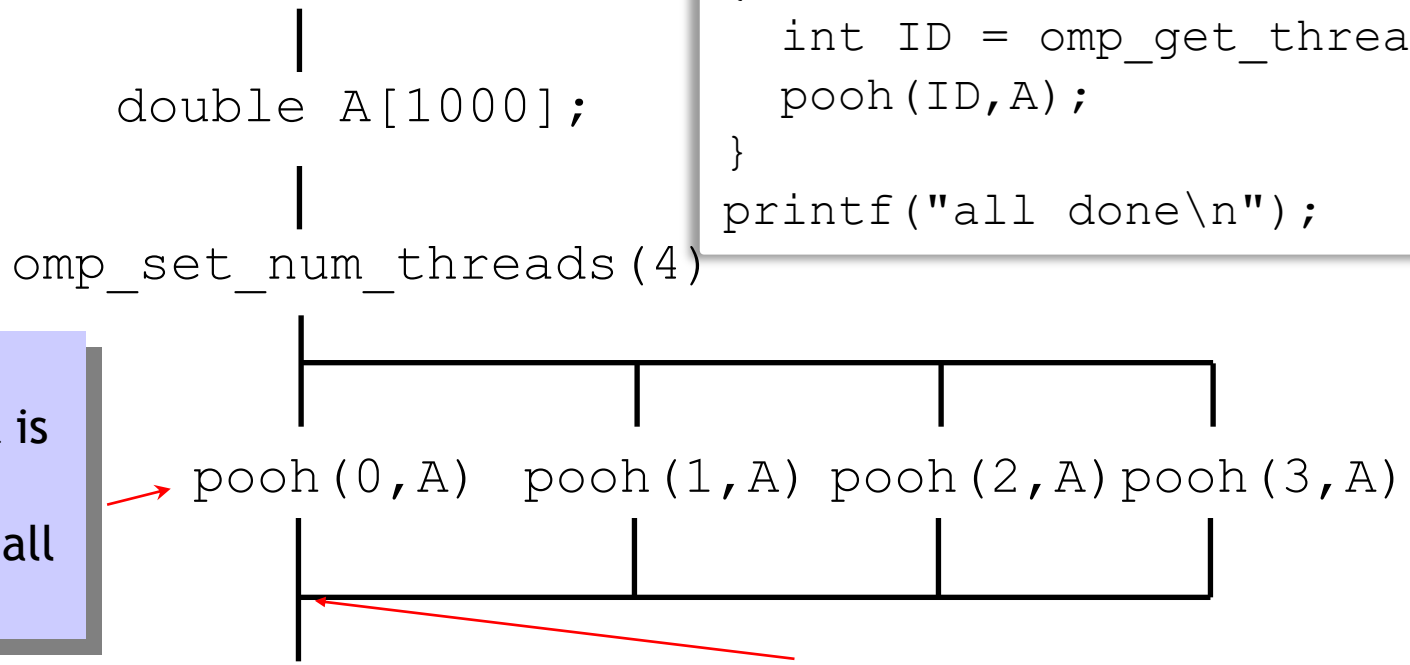
Each thread executes a copy of the code within the structured block

- 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);
#pragma omp parallel
{
   int ID = omp_get_thread_num();
   pooh(ID,A);
}
printf("all done\n");
```

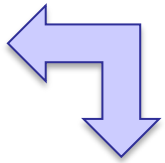


A single copy of `A` is shared between all threads

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

OpenMP: What the Compiler Does?

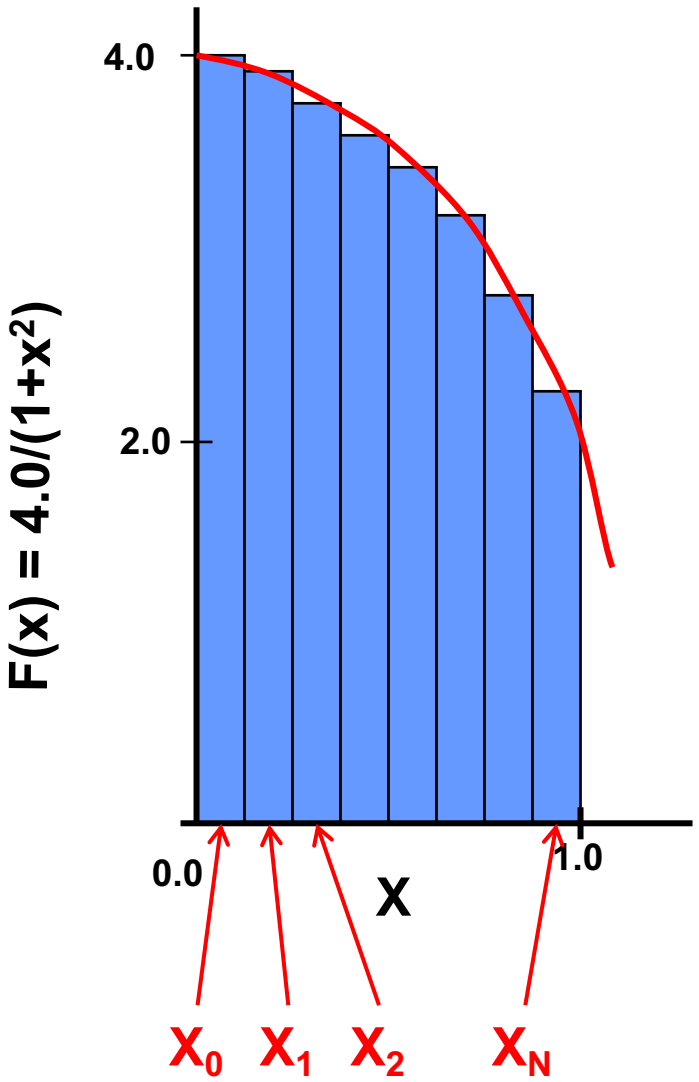
```
#pragma omp parallel num_threads(4)
{
  foobar();
}
```



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

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

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

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

Estimation of Pi: 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

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

Pi Program: Transform into a Pthreads 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++){
    sum = sum + 4.0/(1.0+x*x);
    x = (i+0.5)*step;
  }
  pi = step * sum;
}
```

Variable to accumulate thread results must be shared

Package this into a function

Assign loop iterations to threads

Remove loop-carried dependence

Assure safe update to `pi`

Numerical Integration: Pthreads (1/2)

```
#define NUMSTEPS 10000000
#define NUMTHREADS 4
double step = 0.0, Pi = 0.0;
pthread_mutex_t gLock;
int main() {
  pthread_t thrds[NUMTHREADS];
  int tRank[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;
}
```

Global variables (in data section)

Initialize the mutex variable

Create (fork) the threads

Post a join for each thread

Numerical Integration: Pthreads (2/2)

```
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;
}
```

Local variables (on each thread's own stack). `partialSum` to avoid competition

Use the global mutex to ensure exclusive access to `Pi`

A Simple Parallel Pi Program using OpenMP

```
#include <omp.h>
static long num_steps = 100000; double step;
#define NUM_THREADS 2
void main () {
  int i, nthreads;
  double x, 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;
}
```

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

It is probably the most commonly used pattern in the history of parallel programming

Results*

- Original Serial pi program with 100,000,000 steps ran in 1.83 seconds

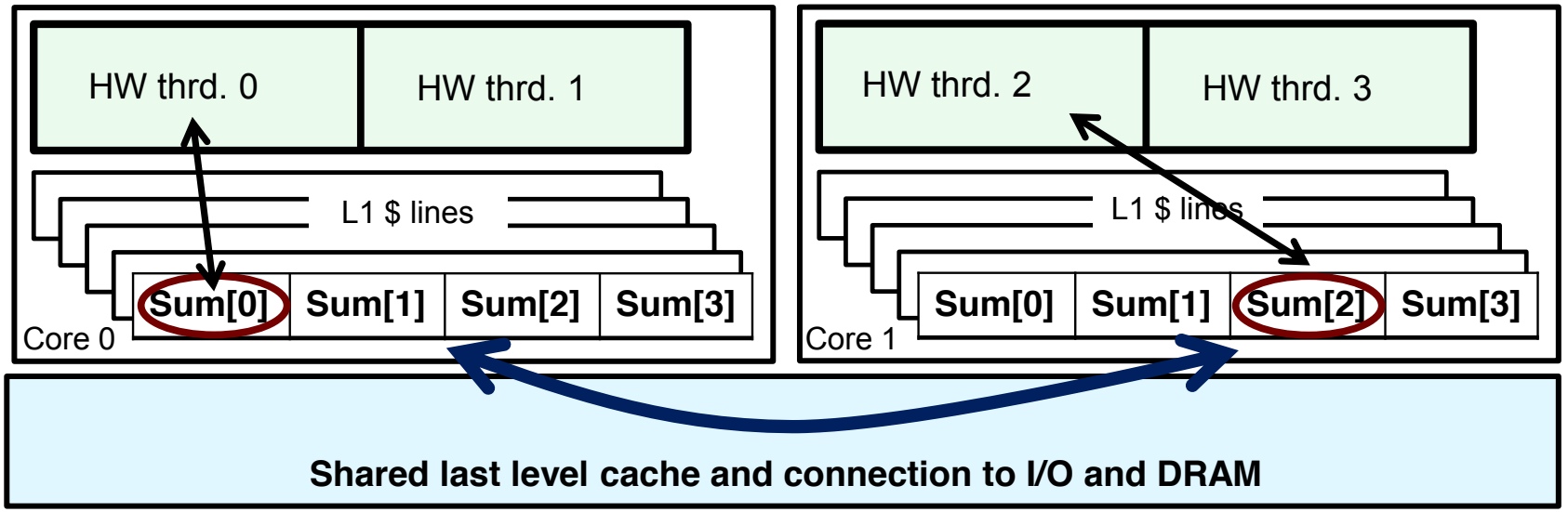
```
#include <omp.h>
static long num_steps = 100000; double step;
#define NUM_THREADS 2
void main () {
  int i, nthreads;
  double x, pi, sum[NUM_THREADS];
  step = 1.0/(double) num_steps;
  omp_set_num_threads(NUM_THREADS);
  #pragma omp parallel
  {
    int i, id, nthrds;
    double x;
    id = omp_get_thread_num();
    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	Eff.
1	1.86	0.98
2	1.03	0.89
3	1.08	0.56
4	0.97	0.47

*Intel compiler (icpc) with no optimization on Apple OS X 10.7.3 with a dual core (four HW threads) 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



- 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

Eliminating False Sharing by Padding sum

```
#include <omp.h>
static long num_steps = 100000; double step;
#define NUM_THREADS 2
#define PAD 8 // assume 64-byte L1 cache line size
void main () {
  int i, nthreads;
  double x, 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)

- Original Serial pi program with 100,000,000 steps ran in 1.83 seconds

```
#include <omp.h>
static long num_steps = 100000; double step;
#define NUM_THREADS 2
#define PAD 8 // assume 64-byte L1 cache line size
void main () {
  int i, nthreads;
  double x, 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	1st SPMD		1st SPMD (padded)	
1	1.86	0.98	1.86	0.98
2	1.03	0.89	1.01	0.9
3	1.08	0.56	0.69	0.88
4	0.97	0.47	0.53	0.86

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

Outline

- Introduction to OpenMP
- Compiler Directives
  - Creating Threads
  - Synchronization
  - Worksharing Constructs
  - More on Synchronization
  - Variable Scopes (`shared/private`)
- Run-time Libraries and Environment Variables

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)
- The statement inside the atomic must be one of the following forms:
  - `x binop= expr`
  - `x++`
  - `++x`
  - `x--`
  - `--x`

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

- `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

`critical` Construct vs `atomic` Construct

- `critical`
  - An OpenMP critical section is completely general
    - Incurring significant overhead every time a thread enters and exits the critical section
  - In OpenMP, all unnamed critical sections are considered identical
    - If one thread is in one [unnamed] critical section as above, no thread can enter any [unnamed] critical section
    - Naming a critical section: `#pragma omp critical(name)`
- `atomic`
  - An atomic operation has much lower overhead. It relies on the hardware providing (say) an atomic increment operation
  - The upsides are that the overhead is much lower
  - The downsides are that you aren't guaranteed any particular set of atomic operations on any particular platform, and you could loose portability
    - However, the compiler should tell you if the particular atomic isn't supported

Some Caveats (1/2)

- You shouldn't mix the different types of mutual exclusion for a single critical section
  - E.g.,

```
#  pragma omp atomic
   x += f(y);
```

```
#  pragma omp critical
   x = g(x);
```

  - The critical directive won't exclude the action executed by the atomic block
- There is no guarantee of fairness in mutual exclusion constructs
  - A thread can be blocked forever

```
   while(1) {
      . . .
#     pragma omp critical
      x = g(my_rank);
      . . .
   }
```

Some Caveats (2/2)

- It can be dangerous to "nest" mutual exclusion constructs

```
#  pragma omp critical
   y = f(x);
   . . .
   double f(double x) {
#     pragma omp critical
      z = g(x);  /* z is shared */
      . . .
   }
```

Guaranteed to deadlock

```
#  pragma omp critical(one)
   y = f(x);
   . . .
   double f(double x) {
#     pragma omp critical(two)
      z = g(x);  /* z is global */
      . . .
   }
```

- Naming critical sections might not help

Time	Thread u	Thread v
0	Enter crit. sect. `one`	Enter crit. sect. `two`
1	Attempt to enter `two`	Attempt to enter `one`
2	Block	Block

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 x, 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;
  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;
} // end of parallel region
} // end of main
```

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 100,000,000 steps ran in 1.83 seconds

```
#include <omp.h>
static long num_steps = 100000; double step;
#define NUM_THREADS 2
void main() {
  double x, pi; step = 1.0/(double) num_step
  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+nthreads){
    x = (i+0.5)*step;
    sum += 4.0/(1.0+x*x);
  }
  #pragma omp critical
  pi += sum * step;
} // end of parallel region
} // end of main
```

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 threads) Intel® Core™ i5 processor at 1.7 Ghz and 4 Gbyte DDR3 memory at 1.333 Ghz

Outline

- Introduction to OpenMP
- Compiler Directives
  - Creating Threads
  - Synchronization
  - Worksharing Constructs
    - Loop Constructs
    - Master/Single Constructs
    - Sections/Section Constructs
  - More on Synchronization
  - Variable Scopes (`shared/private`)
- Run-time Libraries and Environment Variables

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
    - Master/single construct
    - Sections/section constructs
    - Task construct (available in OpenMP 3.0)

Discussed in the next lecture

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, 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];
  }
}
```

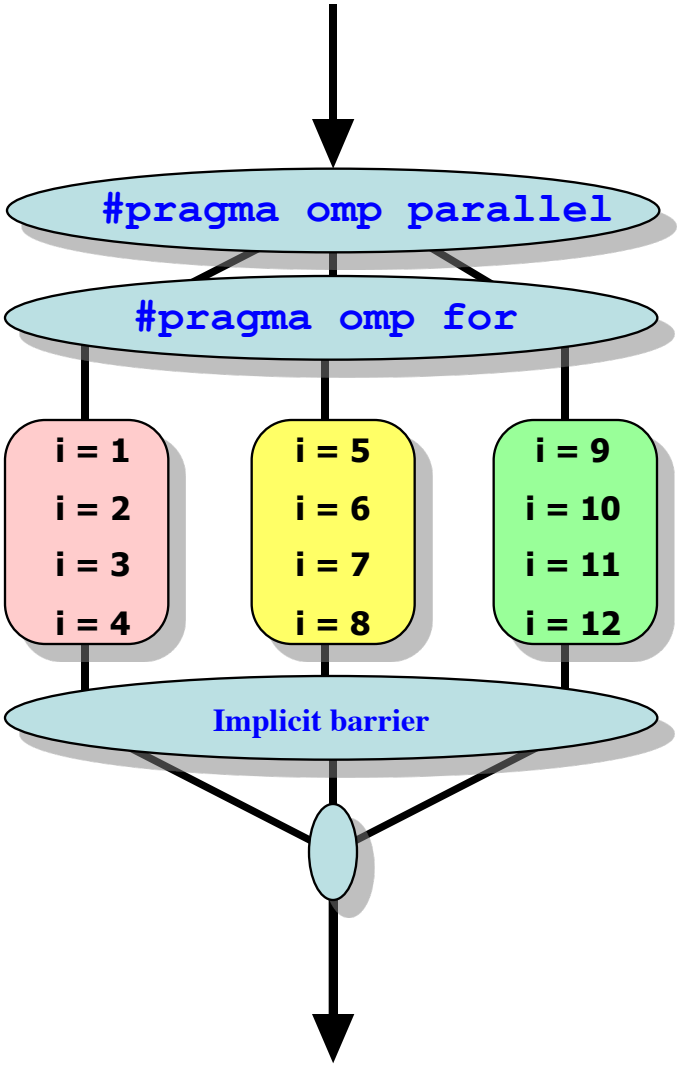
OpenMP parallel region and a worksharing for construct

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

omp for Construct

```
// assume N=12
#pragma omp parallel
#pragma omp for
    for(i = 1; i < N+1; i++)
        c[i] = a[i] + b[i];
```

- Threads are assigned an independent set of iterations
- Threads must wait at the end of work-sharing construct



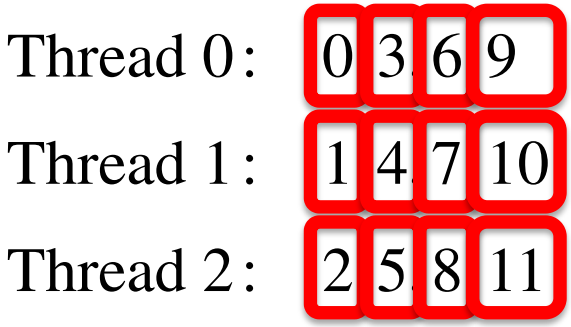
Loop Worksharing Constructs: `schedule` Clause (1/2)

- 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 … for OpenMP 3.0)
  - `schedule(auto)`
    - Schedule is left up to the runtime to choose (does not have to be any of the above)

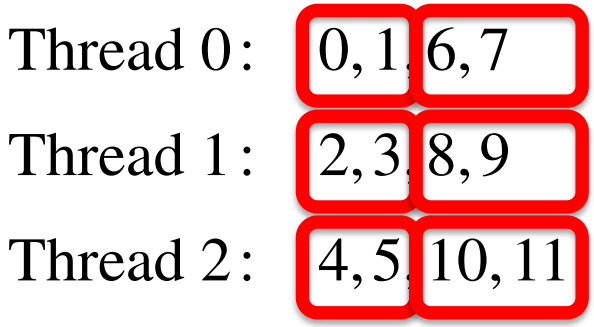
```
$ export OMP_SCHEDULE="static,1"
```

The `static` Schedule Type: Examples

- Default schedule:
  - `schedule(static,` total_iterations/thread_count`)`
- `schedule(static, 1)`



- `schedule(static, 2)`



The `guided` Schedule Type: Examples

Table 5.3 Assignment of Trapezoidal Rule Iterations 1–9999 using a `guided` Schedule with Two Threads

Thread	Chunk	Size of Chunk	Remaining Iterations
0	1–5000	5000	4999
1	5001–7500	2500	2499
1	7501–8750	1250	1249
1	8751–9375	625	624
0	9376–9687	312	312
1	9688–9843	156	156
0	9844–9921	78	78
1	9922–9960	39	39
1	9961–9980	20	19
1	9981–9990	10	9
1	9991–9995	5	4
0	9996–9997	2	2
1	9998–9998	1	1
0	9999–9999	1	0

Loop Worksharing Constructs: `schedule` Clause (1/2)

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

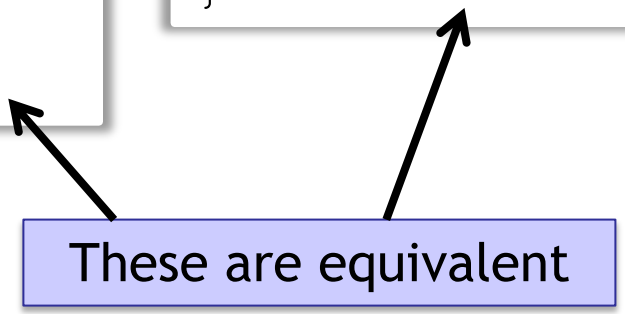
Most work at runtime : complex scheduling logic used at run-time

Combined Parallel/Worksharing Construct

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

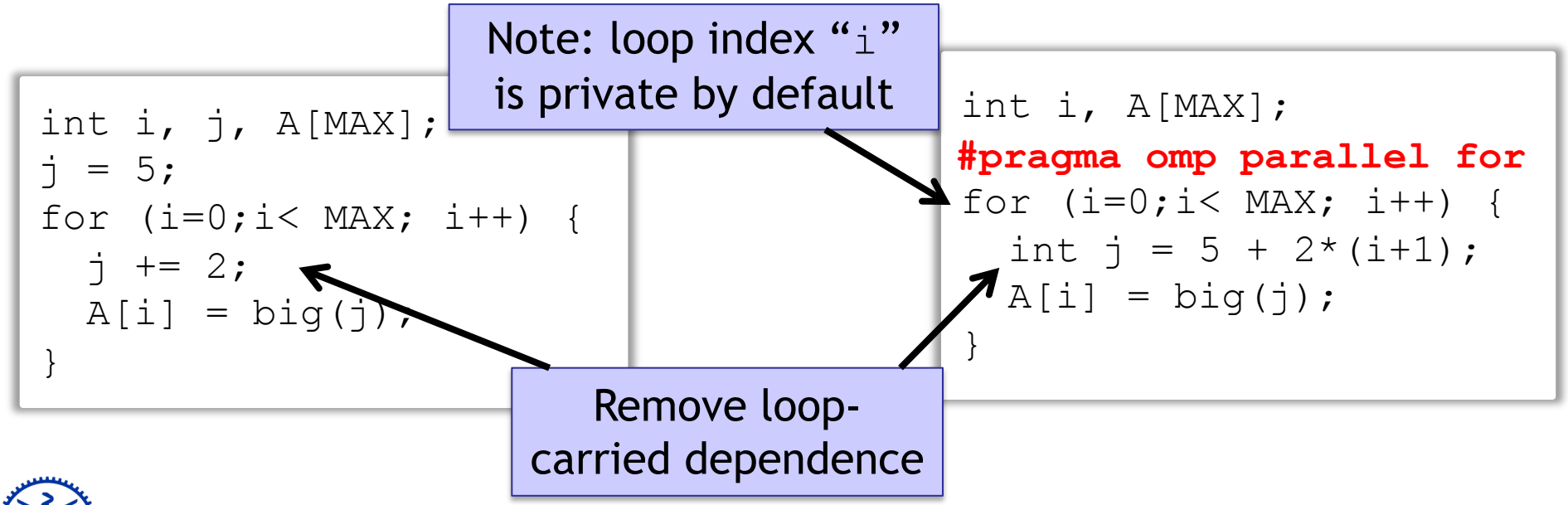
```
double res[MAX];
int i;
#pragma omp parallel
{
  #pragma omp for
  for (i=0;i< MAX; i++) {
    res[i] = huge();
  }
}
```

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



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



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 may complicate balancing the load

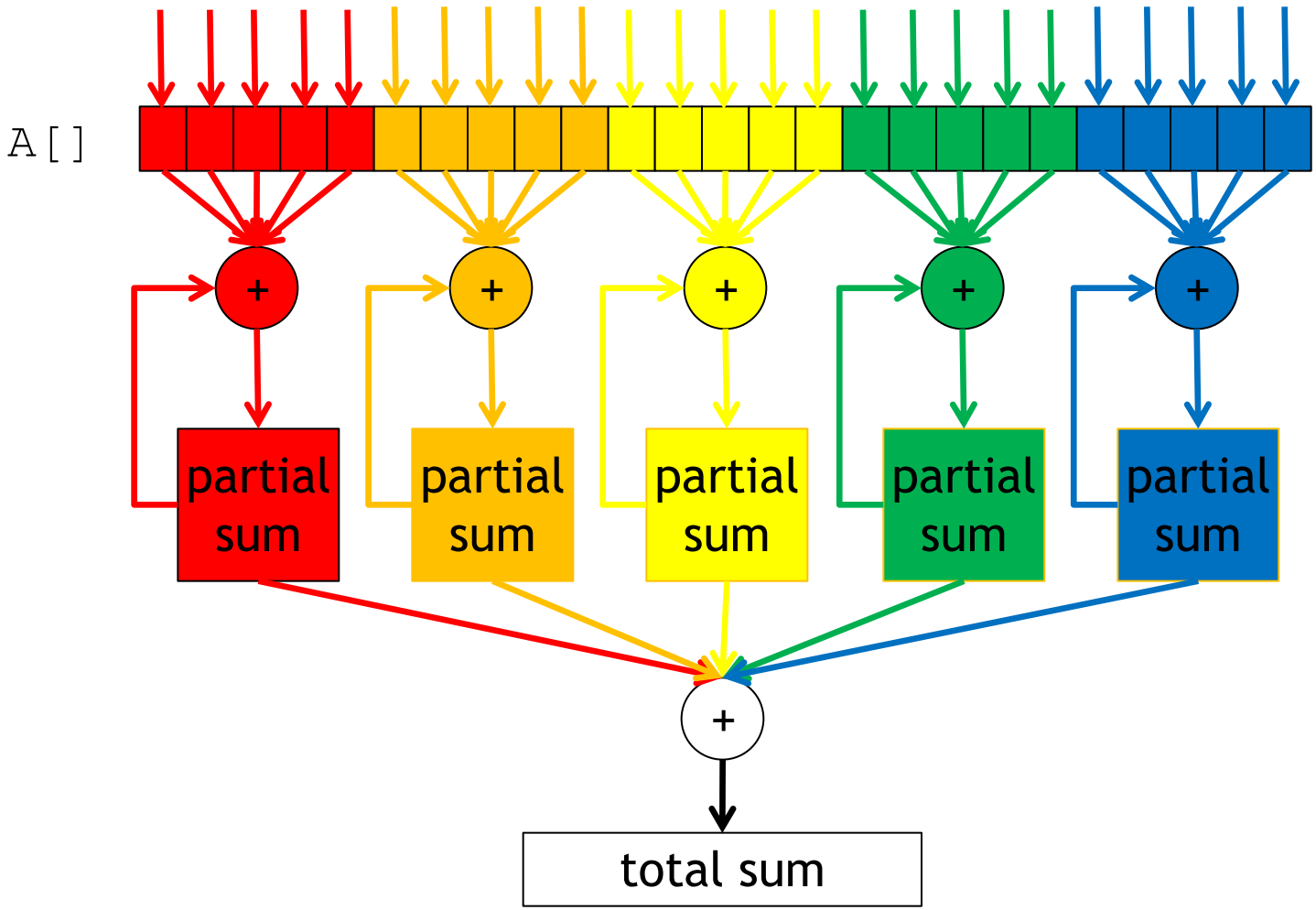
Reduction

- 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 (called a "reduction")
- Support for reduction operations is included in most parallel programming environments

Reduction (Cont'd)



`reduction` Clause

- 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" (identity value, 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

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.
MIN*	Largest pos. number
MAX*	Most neg. number

Example: Pi with a loop and a reduction

```
#include <omp.h>
static long num_steps = 100000;
double step;
void main() {
  int i; double 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 Program (omp for reduction)

- Original Serial pi program with 100,000,000 steps ran in 1.83 seconds

```
#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;
}
```

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 threads) Intel® Core™ i5 processor at 1.7 Ghz and 4 Gbyte DDR3 memory at 1.333 Ghz

`parallel for` Caveats

- OpenMP compilers don't check for dependences among iterations in a `parallel for` loop
  - We need to worry about loop-carried dependences
- OpenMP will only parallelize `for` loops
  - It won't parallelize `while` or `do-while` loops
- OpenMP will only parallelize `for` loops for which the number of iterations can be determined
  - E.g., `for ( ; ; ) { ... }` cannot be parallelized
  - OpenMP will only parallelize `for` loops that are in *canonical form*

`for` Loops in Canonical Form

- The variable `index` must have integer or pointer type (e.g., it can't be a float)
- The expressions start, end, and incr must have a compatible type
  - E.g., if index is a pointer, then incr must have integer type
- The expressions start, end, and incr must not change during execution of the loop
- During execution of the loop, the variable index can only be modified by the "increment expression" in the for statement

```
                                                  index++
                                                  ++index
                                 index < end      index--
                                 index <= end     --index
for ( index = start  ;           index >= end  ;  index += incr
                                 index > end      index -= incr
                                                  index = index + incr
                                                  index = incr + index
                                                  index = index - incr )
```

Outline

- Introduction to OpenMP
- Compiler Directives
  - Creating Threads
  - Synchronization
  - Worksharing Constructs
    - Loop Constructs
    - Master/Single Constructs
    - Sections/Section Constructs
  - More on Synchronization
  - Variable Scopes (`shared/private`)
- Run-time Libraries and Environment Variables

`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)

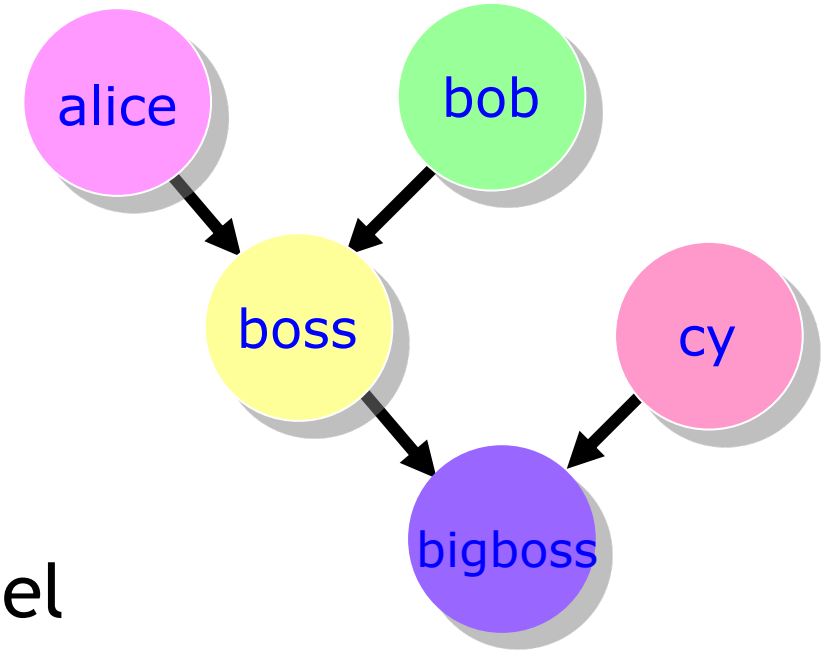
```
#pragma omp parallel
{
  do_many_things();
#pragma omp single
{
  exchange_boundaries();
}
  do_many_other_things();
}
```

Outline

- Introduction to OpenMP
- Compiler Directives
  - Creating Threads
  - Synchronization
  - Worksharing Constructs
    - Loop Constructs
    - Master/Single Constructs
    - Sections/Section Constructs
  - More on Synchronization
  - Variable Scopes (`shared/private`)
- Run-time Libraries and Environment Variables

Function-Level Parallelism

```
a = alice();
b = bob();
s = boss(a, b);
c = cy();
printf ("%6.2f\n", bigboss(s,c));
```



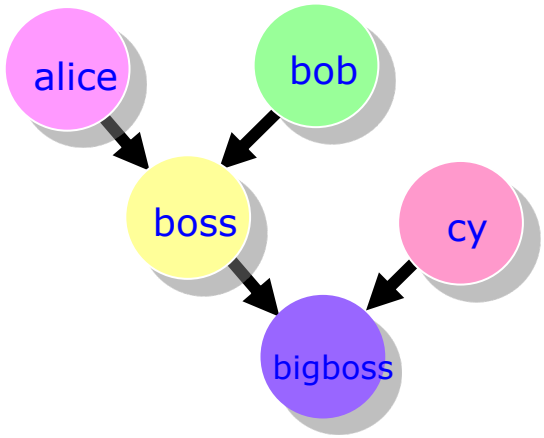
`alice`, `bob`, and `cy`
can be computed in parallel

omp sections/section

- `#pragma omp sections`
  - Must be inside a parallel region
  - Precedes a code block containing *N* sub-blocks of code that may be executed concurrently by *N* threads
  - Encompasses each `omp section`
- `#pragma omp section`
  - Precedes each sub-block of code within the encompassing block described above
  - Enclosed program segments are distributed for parallel execution among available threads

Function-Level Parallelism Using `omp sections`

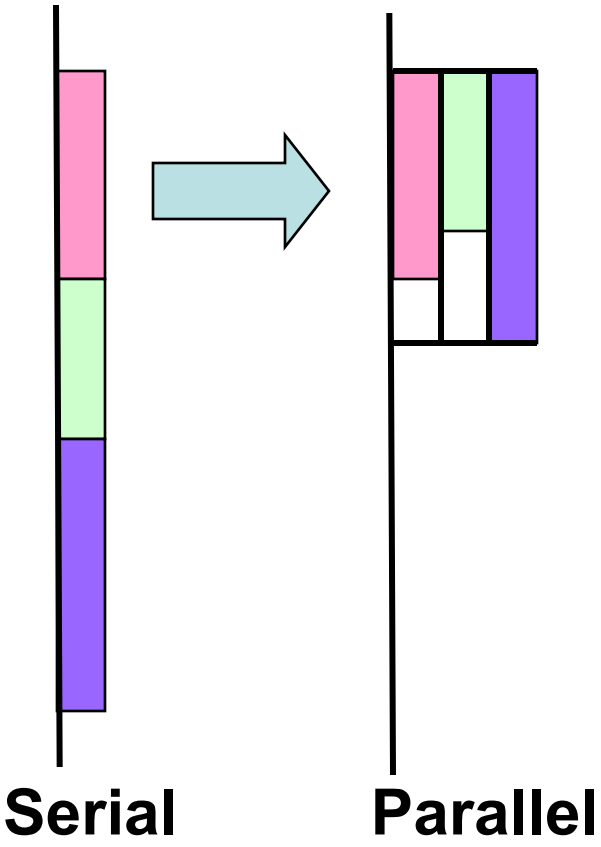
```
#pragma omp parallel sections
{
#pragma omp section
    double a = alice();
#pragma omp section
    double b = bob();
#pragma omp section
    double c = cy();
}

    double s = boss(a, b);
    printf ("%6.2f\n", bigboss(s,c));
```



Advantages of Parallel Sections

- Independent sections of code can execute concurrently – reduce execution time

```
#pragma omp parallel sections
{
#pragma omp section
        phase1();
#pragma omp section
        phase2();
#pragma omp section
        phase3();
}
```



`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.

Outline

- Introduction to OpenMP
- Compiler Directives
  - Creating Threads
  - Synchronization
  - Worksharing Constructs
  - More on Synchronization
  - Variable Scopes (`shared/private`)
- Run-time Libraries and Environment Variables

Synchronization: barrier

- Barrier: Each thread waits until all threads arrive

```
#pragma omp parallel shared (A, B, C) private(id) {
  id = omp_get_thread_num();
  A[id] = big_calc1(id);
#pragma omp barrier
#pragma omp for
  for(i=0;i<N;i++){ C[i] = big_calc3(i,A); }
#pragma omp for nowait
  for(i=0;i<N;i++){ B[i]=big_calc2(C, i); }
  A[id] = big_calc4(id);
}
```

implicit barrier at the end of a for worksharing construct

implicit barrier at the end of a parallel region

no implicit barrier due to nowait

Synchronization: ordered

- The ordered region executes in the sequential order

```
#pragma omp parallel private (tmp)
#pragma omp for ordered reduction(+:res)
for (I=0; I<N; I++) {
  tmp = NEAT_STUFF(I);
#pragma omp ordered
  res += consum(tmp);
}
```

Outline

- Introduction to OpenMP
- Compiler Directives
  - Creating Threads
  - Synchronization
  - Worksharing Constructs
  - More on Synchronization
  - Variable Scopes (`shared/private`)
- Run-time Libraries and Environment Variables

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` variables
  - Both: dynamically allocated memory (`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

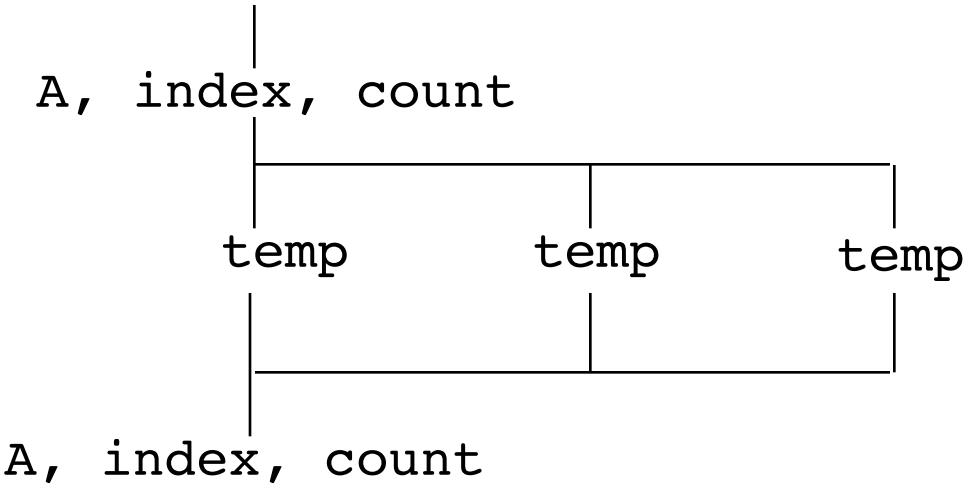
Data Sharing: Examples

```
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`)

*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 is uninitialized
  - In OpenMP 2.5 the value of the private variable is undefined 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`: unspecified in 2.5, 0 in 3.0 (implementations may reference the original variable or a copy. A dangerous programming practice!)

Data Sharing: `firstprivate` Clause

- `firstprivate` is a special case of `private`
  - Variables initialized from shared variable
  - C++ objects are copy-constructed

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

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

`tmp`: 0 in 3.0, unspecified in 2.5

Data Sharing: `lastprivate` Clause

- `lastprivate` is a special case of `private`
  - Variables update shared variable using value from last iteration
  - C++ objects are updated as if by assignment

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

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

`tmp` is defined as its value at the "last sequential" iteration (i.e., for `j=999`)

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)`

Example: Pi Program with 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` is private by default

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

Outline

- Introduction to OpenMP
- Compiler Directives
  - Creating Threads
  - Synchronization
  - Worksharing Constructs
  - More on Synchronization
  - Variable Scopes (`shared/private`)
- Run-time Libraries and Environment Variables

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.

Example: Simple Locks

- Protect resources with locks

```
omp_lock_t lck;
omp_init_lock(&lck);
#pragma omp parallel private (tmp, id)
{
      id = omp_get_thread_num();
      tmp = do_lots_of_work(id);
      omp_set_lock(&lck);
         printf("%d %d", id, tmp);
      omp_unset_lock(&lck);
}
omp_destroy_lock(&lck);
```

Wait here for your turn

Release the lock so the next thread gets a turn

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: Examples

- 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_get_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]"*
- ... Plus several less commonly used environment variables

Summary

- 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 newer and more advanced features of OpenMP

References

- Tim Mattson, Michael Wrinn, and Mark Bull, “A “Hands-on” Introduction to OpenMP”
- Tim Mattson, “Shared Memory Programming with OpenMP - Basics”, 2012 Short Course on Parallel Programming, Berkeley
- Tim Mattson, “More about OpenMP- New Features” , 2012 Short Course on Parallel Programming, Berkeley
- Intel Software College, “Programming with OpenMP”
- Gabriel Mateescu, “Writing and tuning OpenMP programs on distributed shared memory platforms”
- UCB CS267 Course, “Applications of Parallel Computers”