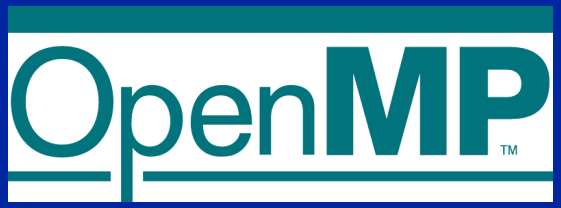


A "Hands-on" Introduction to OpenMP*

Tim Mattson
Intel Corporation
timothy.g.mattson@intel.com

Slightly adapted...

Michael Wrinn
Intel Corporation
michael.wrinn@intel.com

Mark Bull
EPCC, The University of Edinburgh
markb@epcc.ed.ac.uk

With help over the years from Larry Meadows (Intel), Rudi Eigenmann (Purdue), Barbara Chapman (Univ. of Houston), and Sanjiv Shah (Intel)

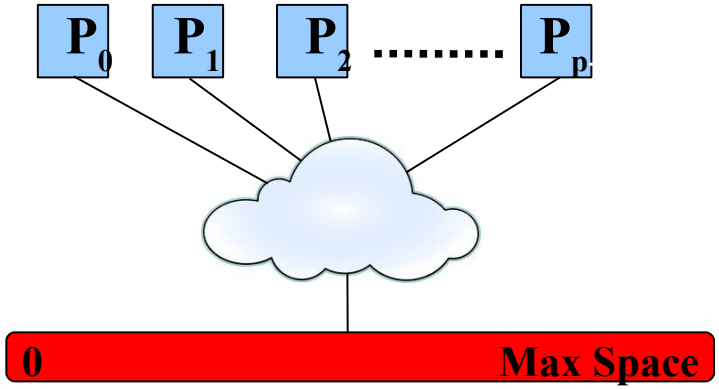
Outline

- Introduction to OpenMP
- Creating Threads
- Synchronization
- Parallel Loops
- Synchronize single masters and stuff
- Data environment
- OpenMP 3.0 and Tasks
- Memory model
- Threadprivate Data

Shared Memory Computers

One common memory area

Visible to all processors



Size of system limited by interconnect

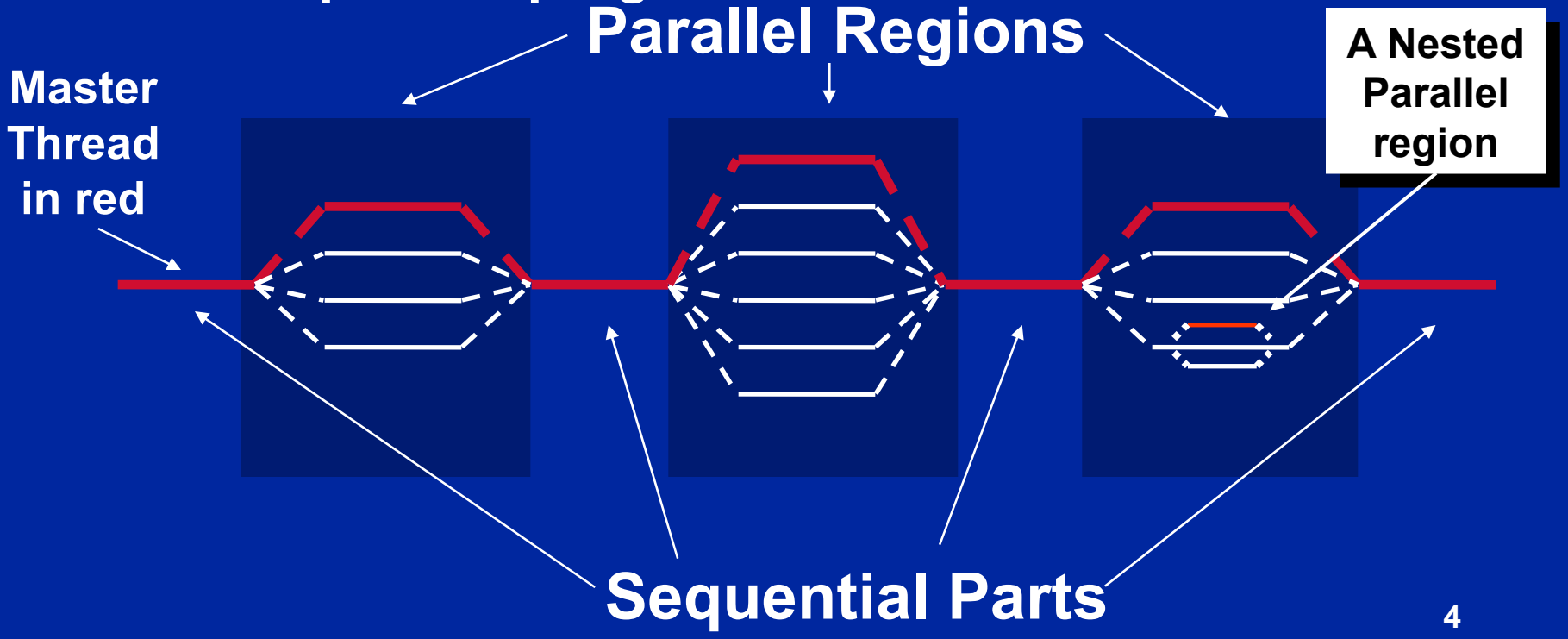
Medium size systems 2-128 processors

Can have both local and shared variables.

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.



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.

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”.

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

Switches for compiling and linking

gcc -fopenmp gcc

pgcc -mp pgi

icl /Qopenmp intel (windows)

icc –openmp intel (linux)

Variables inside the parallel region:

Declared *outside* of the parallel environment: Shared

- **Each thread accesses the same memory cell**
- **Can override by using *private()* construct**

Declared *inside* the parallel environment: Private

- **Each thread gets a local variable that only it can see**
- **Only exists inside the parallel environment**

Exercise 1: Solution
A multi-threaded "Hello world" program

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

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

OpenMP include file

Parallel region with default number of threads

End of the Parallel region

Runtime library function to return a thread ID.

Sample Output:

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

world(0)

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

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.

Outline

- Introduction to OpenMP
- Creating Threads
- Synchronization
- Parallel Loops
- Synchronize single masters and stuff
- Data environment
- OpenMP 3.0 and Tasks
- Memory model
- Threadprivate Data

Thread Creation: Parallel Regions

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

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

Runtime function to request a certain number of threads

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

Runtime function returning a thread ID

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

Thread Creation: Parallel Regions

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

```
double A[1000];

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

clause to request a certain number of threads

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

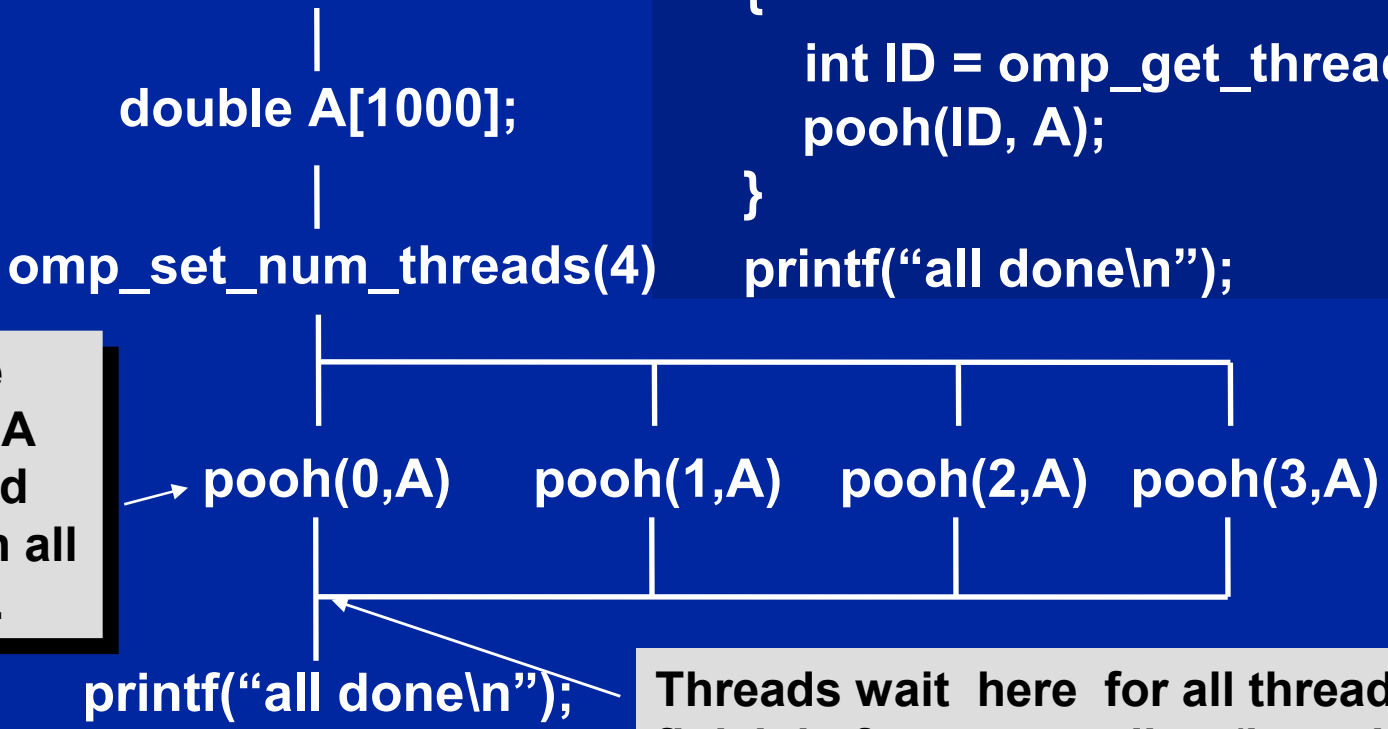
Runtime function returning a thread ID

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

Thread Creation: Parallel Regions example

- Each thread executes the same code redundantly.

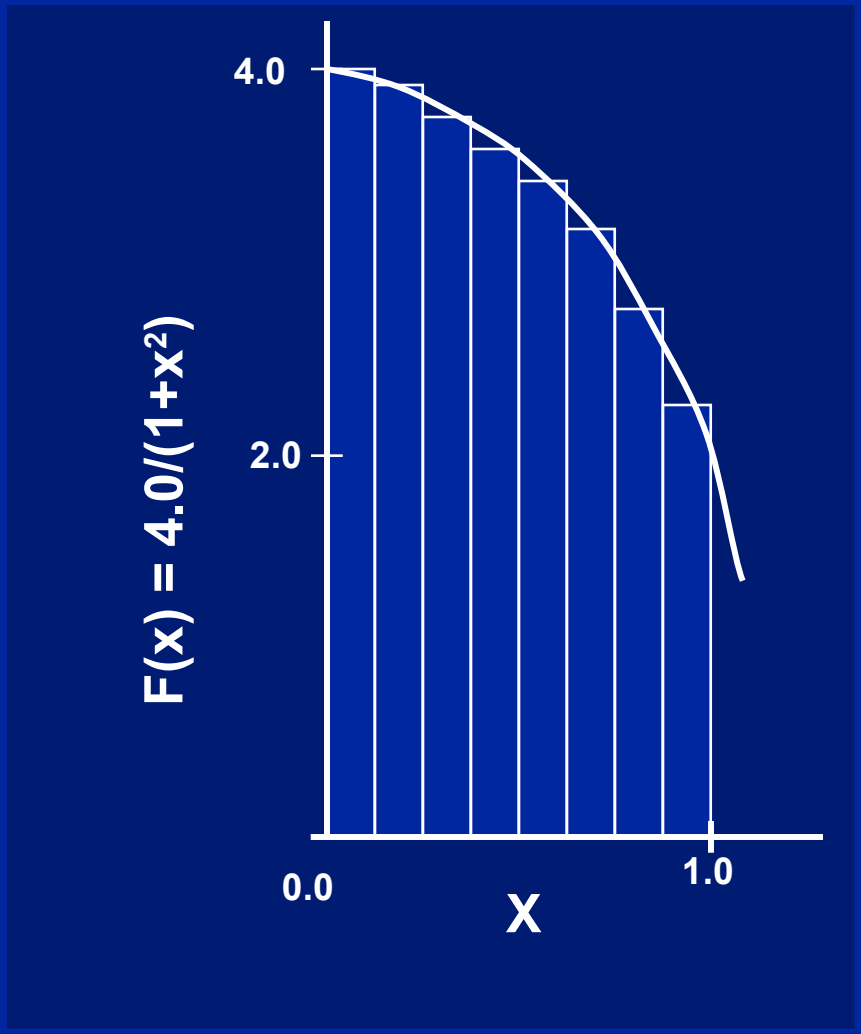
```
double A[1000];
omp_set_num_threads(4);
#pragma omp parallel
{
    int ID = omp_get_thread_num();
    pooh(ID, A);
}
printf("all done\n");
```



A single copy of A is shared between all threads.

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

Exercises 2 to 4:
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.

Exercises 2 to 4: 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;
}
```

Exercise 2

- Create a parallel version of the pi program using a parallel construct.
- Pay close attention to shared versus private variables.
- In addition to a parallel construct, you will need the runtime library routines
  - int omp_get_num_threads();
  - int omp_get_thread_num();
  - double omp_get_wtime();

Number of threads in the team

Thread ID or rank

Time in Seconds since a fixed point in the past

The SPMD pattern

- The most common approach for parallel algorithms is the SPMD or Single Program Multiple Data pattern.
- Each thread runs the same program (Single Program), but using the thread ID, they operate on different data (Multiple Data) or take slightly different paths through the code.
- In OpenMP this means:
  - A parallel region “near the top of the code”.
  - Pick up thread ID and num_threads.
  - Use them to split up loops and select different blocks of data to work on.

Exercise 2: A simple SPMD 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

Outline

- Introduction to OpenMP
- Creating Threads
- Synchronization
- Parallel Loops
- Synchronize single masters and stuff
- Data environment
- OpenMP 3.0 and Tasks
- Memory model
- Threadprivate Data

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.

```
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
          consume (B, res);
     }
}
```

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

Note: critical directive applies to next code block

```
#pragma omp critical
x = x + 1;        // Only this line is critical

#pragma omp critical
{                 // This whole block is critical
   x = x + 1;
   y = y*2;
}
```

Synchronization: Atomic

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

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

Note that all variables allocated in big_ugly() will be local!

Atomic only protects the read/update of X

Exercise 3

- In exercise 2, you probably 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: Each update invalidates the cache line … in essence "sloshing independent data" back and forth between threads.
- Modify your "pi program" from exercise 2 to avoid false sharing due to the sum array.

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.
  - Result … poor scalability
- Solution:
  - When updates to an item are frequent, work with local copies of data instead of an array indexed by the thread ID.
  - Pad arrays so elements you use are on distinct cache lines.

Exercise 3: SPMD Pi without 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

Outline

- Introduction to OpenMP
- Creating Threads
- Synchronization
- Parallel Loops
- Synchronize single masters and stuff
- Data environment
- OpenMP 3.0 and Tasks
- Memory model
- Threadprivate Data

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 …. Available in OpenMP 3.0

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 … for OpenMP 3.0).

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

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

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

Note: loop index "i" is private by default

Remove loop carried dependence

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

Reduction

- How do we handle this case?

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

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

Reduction

- OpenMP reduction clause:

  reduction (op : list)

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

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

OpenMP: Reduction operands/initial-values

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

Operator	Initial value
+	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

Exercise 4: 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.

Exercise 4: solution

```
#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;
}
```

Exercise 4: solution

```
#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 code and by adding 2 simple lines of text!

Outline

- Introduction to OpenMP
- Creating Threads
- Synchronization
- Parallel Loops
- Synchronize single masters and stuff
- Data environment
- OpenMP 3.0 and Tasks
- Memory model
- Threadprivate Data

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);
}
```

Barrier is good for debugging: If one thread does not reach the barrier no thread can progress

implicit barrier at the end of a for worksharing construct

implicit barrier at the end of a parallel region

no implicit barrier due to nowait

Master Construct

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

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

Single worksharing Construct

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

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

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: 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 ordered
                res += consum(tmp);
        }
```

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

- 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

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

… Plus several less commonly used environment variables.

STOP!

Exercises for next week

- Parallelize your matrix multiplication programs.
- Can you optimize the program by playing with how the loops are scheduled?
- More exercises will be posted on the web page.

End of exercises!

Outline

- Introduction to OpenMP
- Creating Threads
- Synchronization
- Parallel Loops
- Synchronize single masters and stuff
- Data environment
- OpenMP 3.0 and Tasks
- Memory model
- Threadprivate Data

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.

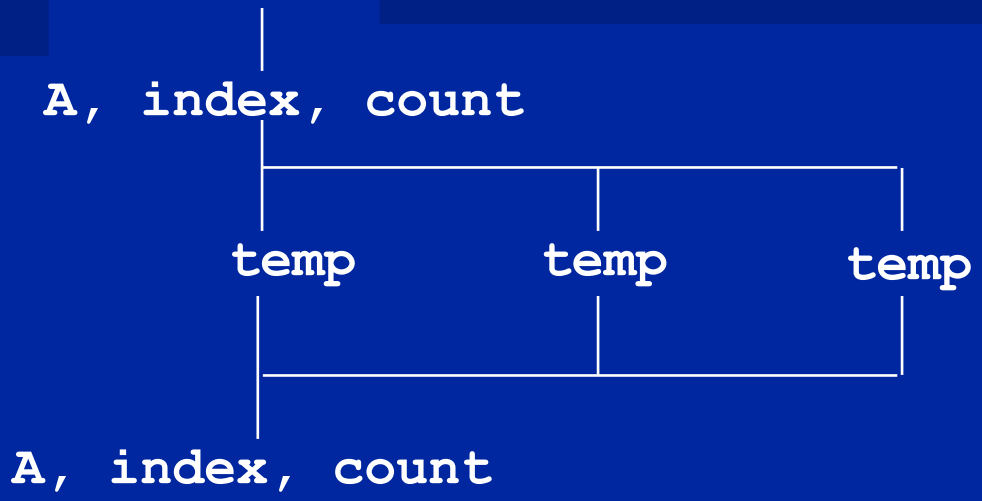
Data sharing: Examples

```
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 is uninitialized
  - In OpenMP 2.5 the value of the shared variable is undefined after the region

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

tmp was not initialized

tmp: 0 in 3.0, unspecified in 2.5

Data Sharing: Private Clause When is the original variable valid?

- The original variable's value is unspecified in OpenMP 2.5.
- In OpenMP 3.0, if it is referenced outside of the construct
  - Implementations may reference the original variable or a copy ….. A dangerous programming practice!

```
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

Data Sharing: Firstprivate Clause

- Firstprivate is a special case of private.
  - Initializes each private copy with the corresponding value from the master thread.

```
void useless() {
    int tmp = 0;
#pragma omp 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 passes the value of a private from the last iteration to a global variable.

```
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 i=999)

Data Sharing:
A data environment test

- Consider this example of PRIVATE and FIRSTPRIVATE

```
variables A,B, and 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

Outside this parallel region ...

- The values of "B" and "C" are unspecified in OpenMP 2.5, and in OpenMP 3.0 if referenced in the region but outside the construct.

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

Data Sharing: tasks (OpenMP 3.0)

- The default for tasks is usually firstprivate, because the task may not be executed until later (and variables may have gone out of scope).
- Variables that are shared in all constructs starting from the innermost enclosing parallel construct are shared, because the barrier guarantees task completion.

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

A is shared
B is firstprivate
C is private

Exercise 6: Molecular dynamics

- The code supplied is a simple molecular dynamics simulation of the melting of solid argon.
- Computation is dominated by the calculation of force pairs in subroutine `forces` (in forces.c)
- Parallelise this routine using a parallel for construct and atomics. Think carefully about which variables should be SHARED, PRIVATE or REDUCTION variables.
- Experiment with different schedules kinds.

Exercise 6 (cont.)

- Once you have a working version, move the parallel region out to encompass the iteration loop in main.c
  - code other than the forces loop must be executed by a single thread (or workshared).
  - how does the data sharing change?
- The atomics are a bottleneck on most systems.
  - This can be avoided by introducing a temporary array for the force accumulation, with an extra dimension indexed by thread number.
  - Which thread(s) should do the final accumulation into f?

Outline

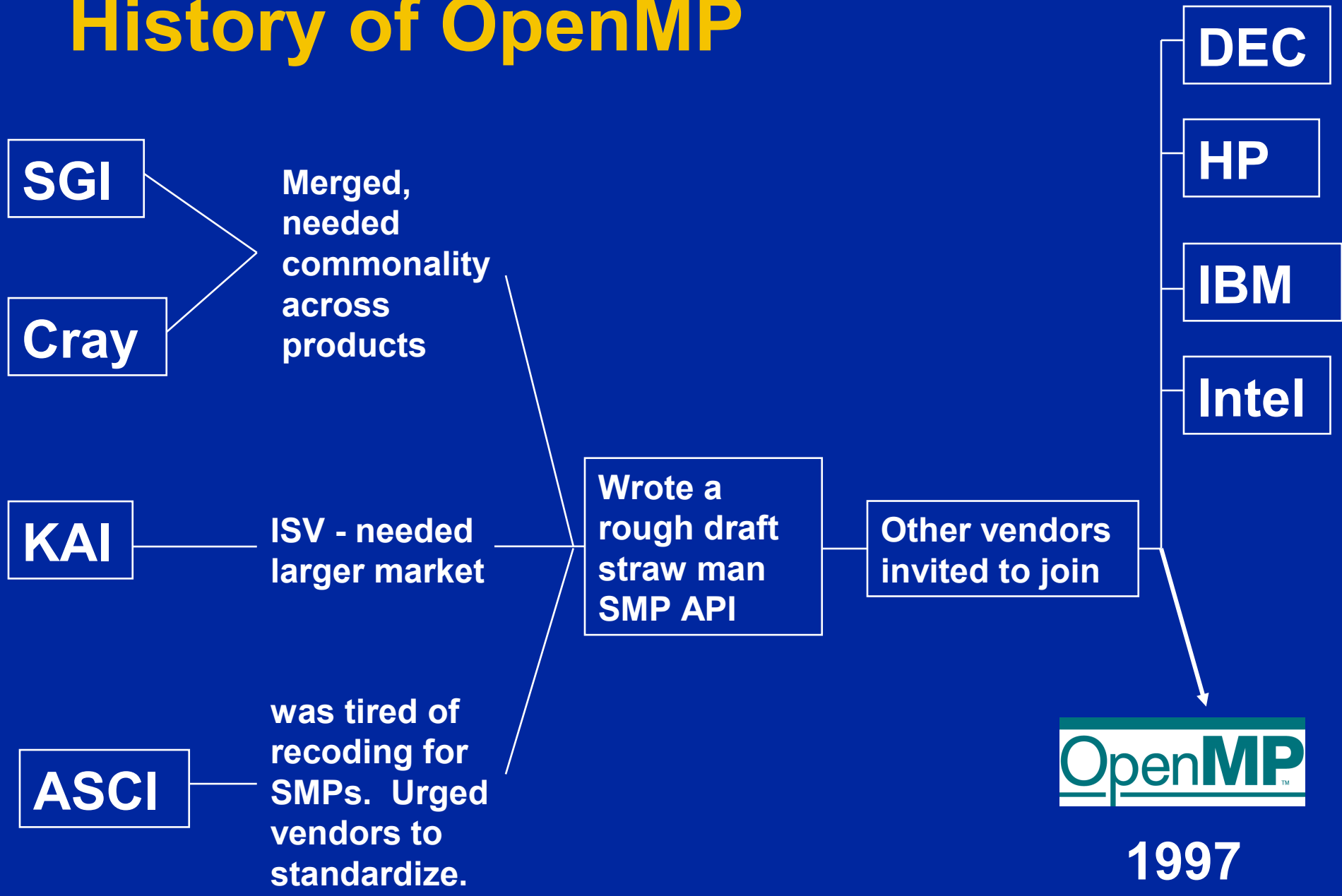
- Introduction to OpenMP
- Creating Threads
- Synchronization
- Parallel Loops
- Synchronize single masters and stuff
- Data environment
- OpenMP 3.0 and Tasks
- Memory model
- Threadprivate Data

OpenMP pre-history

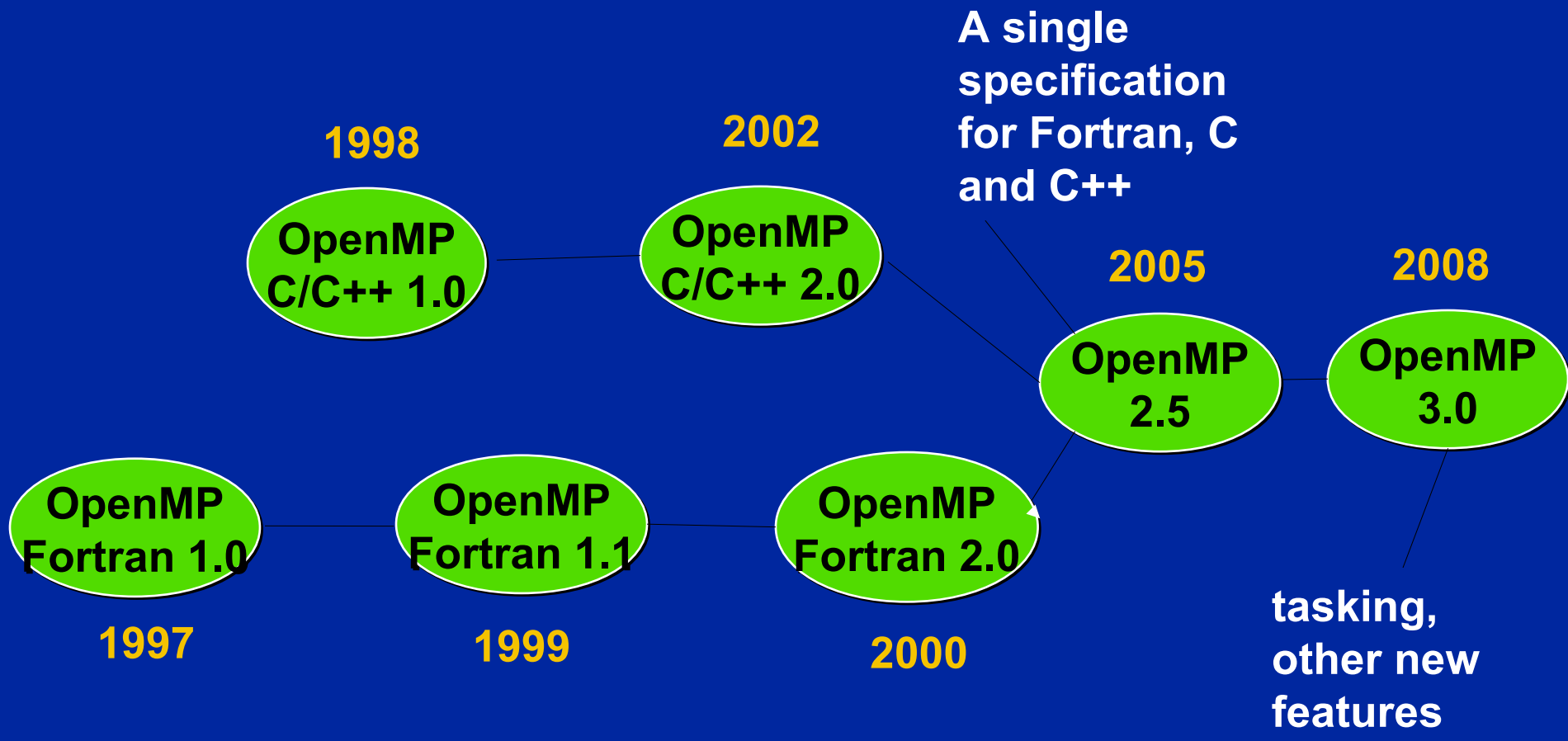
- OpenMP based upon SMP directive standardization efforts PCF and aborted ANSI X3H5 – late 80's
  - Nobody fully implemented either standard
  - Only a couple of partial implementations
- Vendors considered proprietary API's to be a competitive feature:
  - Every vendor had proprietary directives sets
  - Even KAP, a "portable" multi-platform parallelization tool used different directives on each platform

PCF – Parallel computing forum KAP – parallelization tool from KAI.

History of OpenMP



OpenMP Release History



Tasks

- Adding tasking is the biggest addition for 3.0
- Worked on by a separate subcommittee
  - led by Jay Hoeflinger at Intel
- Re-examined issue from ground up
  - quite different from Intel taskq's

General task characteristics

- A task has
  - Code to execute
  - A data environment (it *owns* its data)
  - An assigned thread that executes the code and uses the data
- Two activities: packaging and execution
  - Each encountering thread packages a new instance of a task (code and data)
  - Some thread in the team executes the task at some later time

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

Tasks and OpenMP

- Tasks have been fully integrated into OpenMP
- Key concept: OpenMP has always had tasks, we just never called them that.
  - Thread encountering `parallel` construct packages up a set of *implicit* tasks, one per thread.
  - Team of threads is created.
  - Each thread in team is assigned to one of the tasks (and *tied* to it).
  - Barrier holds original master thread until all implicit tasks are finished.
- We have simply added a way to create a task explicitly for the team to execute.
- Every part of an OpenMP program is part of one task or another!

`task` Construct

```
#pragma omp task [clause[[,]clause] ...]
         structured-block
```

where *clause* can be one of:

```
if (expression)
untied
shared (list)
private (list)
firstprivate (list)
default( shared | none  )
```

The `if` clause

- When the `if` clause argument is false
  - The task is executed immediately by the encountering thread.
  - The data environment is still local to the new task...
  - ...and it's still a different task with respect to synchronization.
- It's a user directed optimization
  - when the cost of deferring the task is too great compared to the cost of executing the task code
  - to control cache and memory affinity

When/where are tasks complete?

- At thread barriers, explicit or implicit
  - applies to all tasks generated in the current parallel region up to the barrier
  - matches user expectation
- At task barriers
  - i.e. Wait until all tasks defined in the current task have completed.

```
#pragma omp taskwait
```

  - Note: applies only to tasks generated in the current task, not to “descendants” .

Example – parallel pointer chasing using tasks

```
#pragma omp parallel
{
  #pragma omp single private(p)
   {
    p = listhead ;
    while (p) {
       #pragma omp task
                process (p);
       p=next (p) ;
     }
   }
}
```

`p` is firstprivate inside this task

Example – parallel pointer chasing on multiple lists using tasks

```
#pragma omp parallel
{
    #pragma omp for private(p)
    for ( int i =0; i <numlists ; i++) {
        p = listheads [ i ] ;
        while (p ) {
        #pragma omp task
            process (p);
        p=next (p ) ;
        }
    }
}
```

Example: postorder tree traversal

```
void postorder(node *p) {
    if (p->left)
      #pragma omp task
        postorder(p->left);
    if (p->right)
      #pragma omp task
        postorder(p->right);
#pragma omp taskwait // wait for descendants
  process(p->data);
}
```

Task scheduling point

- Parent task suspended until children tasks complete

Task switching

- Certain constructs have task scheduling points at defined locations within them
- When a thread encounters a task scheduling point, it is allowed to suspend the current task and execute another (called *task switching*)
- It can then return to the original task and resume

Task switching example

```
#pragma omp single
{
  for (i=0; i<ONEZILLION; i++)
    #pragma omp task
      process(item[i]);
}
```

- Too many tasks generated in an eye-blink
- Generating task will have to suspend for a while
- With task switching, the executing thread can:
  - execute an already generated task (draining the "*task pool*")
  - dive into the encountered task (could be very cache-friendly)

Thread switching

```
#pragma omp single
{
  #pragma omp task  untied
    for (i=0; i<ONEZILLION; i++)
      #pragma omp task
        process(item[i]);
}
```

- Eventually, too many tasks are generated
- Generating task is suspended and executing thread switches to a long and boring task
- Other threads get rid of all already generated tasks, and start starving…

- With thread switching, the generating task can be resumed by a different thread, and starvation is over
- Too strange to be the default: the programmer is responsible!

Dealing with taskprivate data

- The Taskprivate directive was removed from OpenMP 3.0
  - Too expensive to implement
- Restrictions on task scheduling allow threadprivate data to be used
  - User can avoid thread switching with tied tasks
  - Task scheduling points are well defined

Conclusions on tasks

- Enormous amount of work by many people
- Tightly integrated into 3.0 spec
- Flexible model for irregular parallelism
- Provides balanced solution despite often conflicting goals
- Appears that performance can be reasonable

Nested parallelism

- Better support for nested parallelism
- Per-thread internal control variables
  - Allows, for example, calling `omp_set_num_threads()` inside a parallel region.
  - Controls the team sizes for next level of parallelism
- Library routines to determine depth of nesting, IDs of parent/grandparent etc. threads, team sizes of parent/grandparent etc. teams

```
omp_get_active_level()
omp_get_ancestor(level)
omp_get_teamsize(level)
```

Parallel loops

- Guarantee that this works … i.e. that the same schedule is used in the two loops:

```
!$omp do schedule(static)
do i=1,n
   a(i) = ....
end do
!$omp end do nowait
!$omp do schedule(static)
do i=1,n
  .... = a(i)
end do
```

Loops (cont.)

- Allow collapsing of perfectly nested loops

```
!$omp parallel do collapse(2)
do i=1,n
     do j=1,n
          .....
     end do
end do
```

- Will form a single loop and then parallelize that

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

Portable control of threads

- Added environment variable to control the size of child threads' stack

`OMP_STACKSIZE`

- Added 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 program execution

- Added environment variable and runtime routines to get/set the maximum number of active levels of nested parallelism

```
OMP_MAX_ACTIVE_LEVELS
omp_set_max_active_levels()
omp_get_max_active_levels()
```

- Added environment variable to set maximum number of threads in use

```
OMP_THREAD_LIMIT
omp_get_thread_limit()
```

Odds and ends

- Allow unsigned ints in parallel for loops
- Disallow use of the original variable as master thread's private variable
- Make it clearer where/how private objects are constructed/destructed
- Relax some restrictions on allocatable arrays and Fortran pointers
- Plug some minor gaps in memory model
- Allow C++ static class members to be threadprivate
- Improve C/C++ grammar
- Minor fixes and clarifications to 2.5

Exercise 7: tasks in OpenMP

- Consider the program linked.c
  - Traverses a linked list computing a sequence of Fibonacci numbers at each node.
- Parallelize this program using tasks.
- Compare your solution's complexity to an approach without tasks.

Exercise 8: 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 defined in OpenMP 2.5 (loop worksharing constructs … i.e. don't use OpenMP 3.0 tasks).
- Once you have a correct program, optimize it.

Conclusion

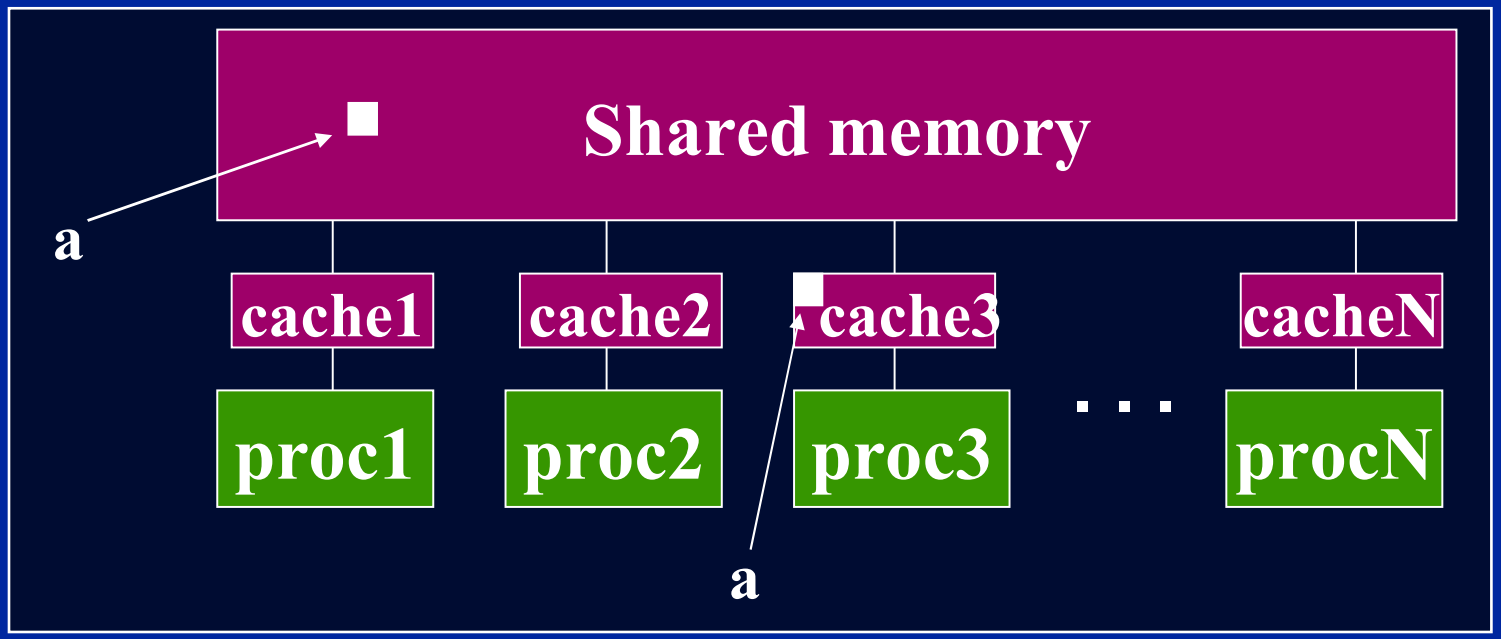
- OpenMP 3.0 is a major upgrade … expands the range of algorithms accessible from OpenMP.

Outline

- Introduction to OpenMP
- Creating Threads
- Synchronization
- Parallel Loops
- Synchronize single masters and stuff
- Data environment
- OpenMP 3.0 and Tasks
- Memory model
- Threadprivate Data

OpenMP memory model

- OpenMP supports a shared memory model.
- All threads share an address space, but it can get complicated:



- Multiple copies of data may be present in various levels of cache, or in registers.

OpenMP and Relaxed Consistency

- OpenMP supports a relaxed-consistency shared memory model.
  - Threads can maintain a temporary view of shared memory which is not consistent with that of other threads.
  - These temporary views are made consistent only at certain points in the program.
  - The operation which enforces consistency is called the flush operation

Flush operation

- Defines a sequence point at which a thread is guaranteed to see a consistent view of memory
  - All previous read/writes by this thread have completed and are visible to other threads
  - No subsequent read/writes by this thread have occurred
  - A flush operation is analogous to a fence in other shared memory API's

Flush and synchronization

- A flush operation is implied by OpenMP synchronizations, e.g.
  - at entry/exit of parallel regions
  - at implicit and explicit barriers
  - at entry/exit of critical regions
  - whenever a lock is set or unset

  ….

  (but not at entry to worksharing regions or entry/exit of master regions)

Example: producer-consumer pattern

Thread 0

```
a = foo();
flag = 1;
```

Thread 1

```
while (!flag);
b = a;
```

- This is incorrect code
- The compiler and/or hardware may re-order the reads/writes to `a` and `flag`, or `flag` may be held in a register.
- OpenMP has a `#pragma omp flush` directive which specifies an explicit flush operation
  - can be used to make the above example work
  - … but it's use is difficult and prone to subtle bugs
  - … To learn more about flush and its use, see the appendix

Outline

- Introduction to OpenMP
- Creating Threads
- Synchronization
- Parallel Loops
- Synchronize single masters and stuff
- Data environment
- OpenMP 3.0 and Tasks
- Memory model
- Threadprivate Data

Data sharing: Threadprivate

- Makes global data private to a thread
  - Fortran: COMMON blocks
  - C: File scope and static variables, static class members
- Different from making them PRIVATE
  - with PRIVATE global variables are masked.
  - THREADPRIVATE preserves global scope within each thread
- Threadprivate variables can be initialized using COPYIN or at time of definition (using language-defined initialization capabilities).

A threadprivate example (C)

Use threadprivate to create a counter for each thread.

```
int counter = 0;
#pragma omp threadprivate(counter)

int increment_counter()
{
    counter++;
    return (counter);
}
```

Data Copying: Copyin

You initialize threadprivate data using a copyin clause.

```
      parameter (N=1000)
      common/buf/A(N)
!$OMP THREADPRIVATE(/buf/)

C Initialize the A array
      call init_data(N,A)

!$OMP PARALLEL COPYIN(A)

 … Now each thread sees threadprivate array A initialied
 … to the global value set in the subroutine init_data()

!$OMP END PARALLEL

end
```

Data Copying: Copyprivate

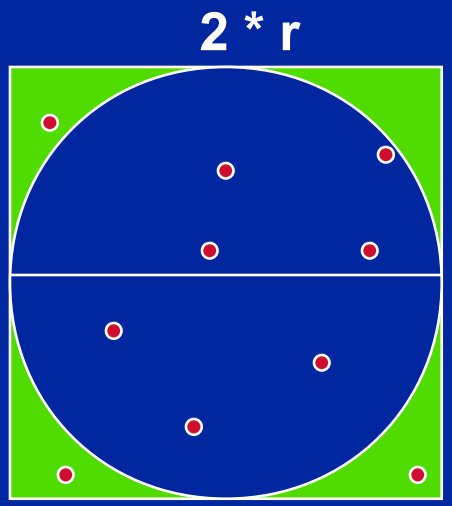
Used with a single region to broadcast values of privates from one member of a team to the rest of the team.

```
#include <omp.h>
void input_parameters (int, int); // fetch values of input parameters
void do_work(int, int);

void main()
{
  int Nsize, choice;

  #pragma omp parallel private (Nsize, choice)
  {
     #pragma omp single copyprivate (Nsize, choice)
           input_parameters (Nsize, choice);

     do_work(Nsize, choice);
  }
}
```

Exercise 9: Monte Carlo Calculations

Using Random numbers to solve tough problems

- Sample a problem domain to estimate areas, compute probabilities, find optimal values, etc.
- Example: Computing π with a digital dart board:



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

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

  $A_c = r^2 * \pi$

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

  $P = A_c/A_s = \pi /4$

- Compute π by randomly choosing points, count the fraction that falls in the circle, compute pi.

Exercise 9

- We provide three files for this exercise
  - pi_mc.c: the monte carlo method pi program
  - random.c: a simple random number generator
  - random.h: include file for random number generator
- Create a parallel version of this program without changing the interfaces to functions in random.c
  - This is an exercise in modular software … why should a user of your parallel random number generator have to know any details of the generator or make any changes to how the generator is called?
  - The random number generator must be threadsafe.
- Extra Credit:
  - Make your random number generator numerically correct (non-overlapping sequences of pseudo-random numbers).

Conclusion

- We have now covered the full sweep of the OpenMP specification.
  - We've left off some minor details, but we've covered all the major topics … remaining content you can pick up on your own.
- Download the spec to learn more … the spec is filled with examples to support your continuing education.
  - www.openmp.org
- Get involved:
  - get your organization to join the OpenMP ARB.
  - Work with us through Compunity.

Appendices

- Sources for Additional information
- Solutions to exercises
  - Exercise 1: hello world
  - Exercise 2: Simple SPMD Pi program
  - Exercise 3: SPMD Pi without false sharing
  - Exercise 4: Loop level Pi
  - Exercise 5: Matrix multiplication
  - Exercise 6: Molecular dynamics
  - Exercise 7: linked lists with tasks
  - Exercise 8: linked lists without tasks
  - Exercise 9: Monte Carlo Pi and random numbers
- Flush and the OpenMP Memory model.
  - the producer consumer pattern
- Fortran and OpenMP
- Compiler Notes

OpenMP Organizations

- OpenMP architecture review board URL, the "owner" of the OpenMP specification:

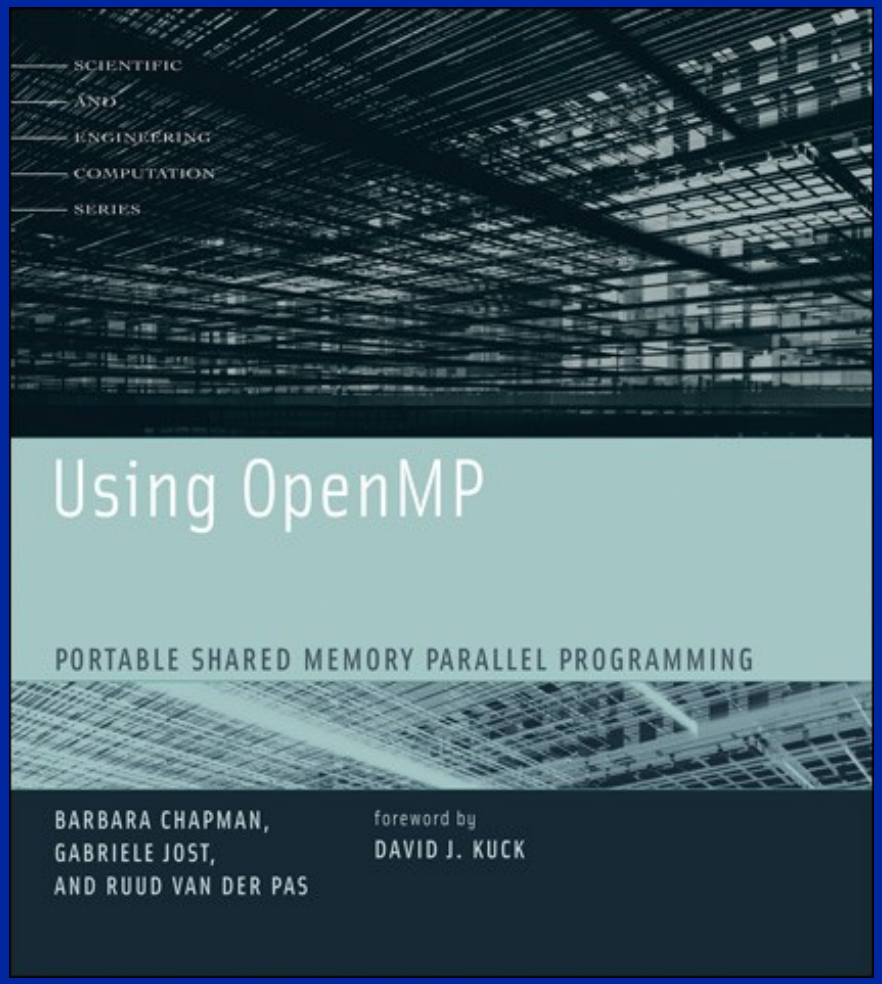
  www.openmp.org

- OpenMP User's Group (cOMPunity) URL:

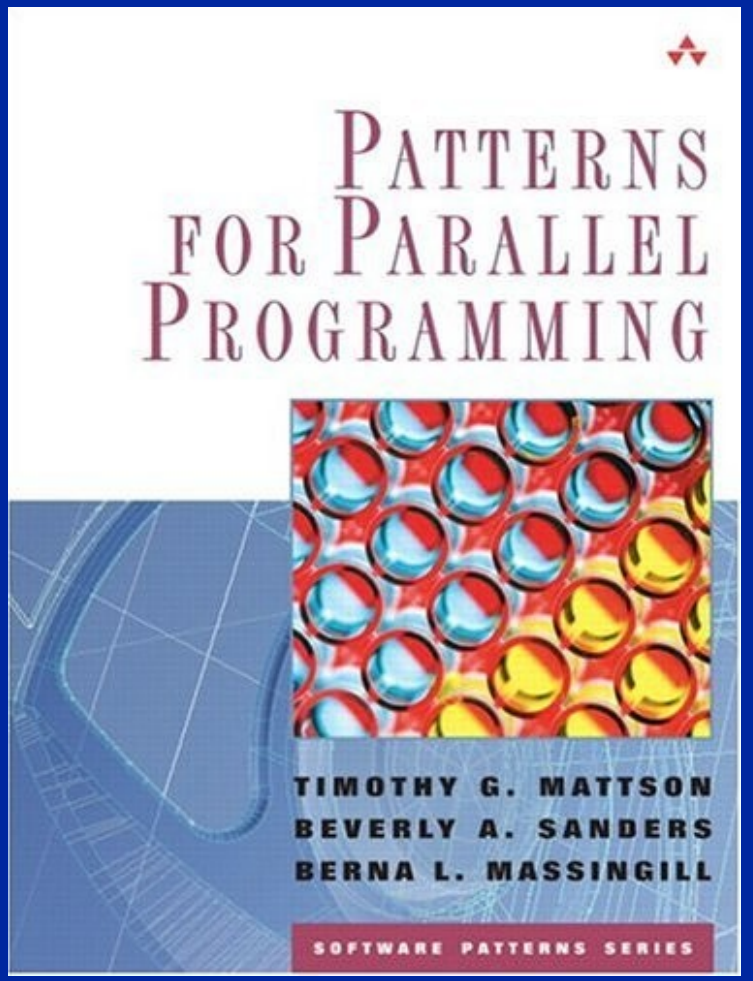
  www.compunity.org

Get involved, join compunity and help define the future of OpenMP

Books about OpenMP



- A new book about OpenMP 2.5 by a team of authors at the forefront of OpenMP's evolution.



- A book about how to "think parallel" with examples in OpenMP, MPI and java

OpenMP Papers

- Sosa CP, Scalmani C, Gomperts R, Frisch MJ. Ab initio quantum chemistry on a ccNUMA architecture using OpenMP. III. Parallel Computing, vol.26, no.7-8, July 2000, pp.843-56. Publisher: Elsevier, Netherlands.
- Couturier R, Chipot C. Parallel molecular dynamics using OPENMP on a shared memory machine. Computer Physics Communications, vol.124, no.1, Jan. 2000, pp.49-59. Publisher: Elsevier, Netherlands.
- Bentz J., Kendall R., "Parallelization of General Matrix Multiply Routines Using OpenMP", Shared Memory Parallel Programming with OpenMP, Lecture notes in Computer Science, Vol. 3349, P. 1, 2005
- Bova SW, Breshearsz CP, Cuicchi CE, Demirbilek Z, Gabb HA. Dual-level parallel analysis of harbor wave response using MPI and OpenMP. International Journal of High Performance Computing Applications, vol.14, no.1, Spring 2000, pp.49-64. Publisher: Sage Science Press, USA.
- Ayguade E, Martorell X, Labarta J, Gonzalez M, Navarro N. Exploiting multiple levels of parallelism in OpenMP: a case study. Proceedings of the 1999 International Conference on Parallel Processing. IEEE Comput. Soc. 1999, pp.172-80. Los Alamitos, CA, USA.
- Bova SW, Breshears CP, Cuicchi C, Demirbilek Z, Gabb H. Nesting OpenMP in an MPI application. Proceedings of the ISCA 12th International Conference. Parallel and Distributed Systems. ISCA. 1999, pp.566-71. Cary, NC, USA.

OpenMP Papers (continued)

- Jost G., Labarta J., Gimenez J., What Multilevel Parallel Programs do when you are not watching: a Performance analysis case study comparing MPI/OpenMP, MLP, and Nested OpenMP, Shared Memory Parallel Programming with OpenMP, Lecture notes in Computer Science, Vol. 3349, P. 29, 2005
- Gonzalez M, Serra A, Martorell X, Oliver J, Ayguade E, Labarta J, Navarro N. Applying interposition techniques for performance analysis of OPENMP parallel applications. Proceedings 14th International Parallel and Distributed Processing Symposium. IPDPS 2000. IEEE Comput. Soc. 2000, pp.235-40.
- Chapman B, Mehrotra P, Zima H. Enhancing OpenMP with features for locality control. Proceedings of Eighth ECMWF Workshop on the Use of Parallel Processors in Meteorology. Towards Teracomputing. World Scientific Publishing. 1999, pp.301-13. Singapore.
- Steve W. Bova, Clay P. Breshears, Henry Gabb, Rudolf Eigenmann, Greg Gaertner, Bob Kuhn, Bill Magro, Stefano Salvini. Parallel Programming with Message Passing and Directives; SIAM News, Volume 32, No 9, Nov. 1999.
- Cappello F, Richard O, Etiemble D. Performance of the NAS benchmarks on a cluster of SMP PCs using a parallelization of the MPI programs with OpenMP. Lecture Notes in Computer Science Vol.1662. Springer-Verlag. 1999, pp.339-50.
- Liu Z., Huang L., Chapman B., Weng T., Efficient Implementationi of OpenMP for Clusters with Implicit Data Distribution, Shared Memory Parallel Programming with OpenMP, Lecture notes in Computer Science, Vol. 3349, P. 121, 2005

OpenMP Papers (continued)

- B. Chapman, F. Bregier, A. Patil, A. Prabhakar, "Achieving performance under OpenMP on ccNUMA and software distributed shared memory systems," *Concurrency and Computation: Practice and Experience.* 14(8-9): 713-739, 2002.
- J. M. Bull and M. E. Kambites. JOMP: an OpenMP-like interface for Java. Proceedings of the ACM 2000 conference on Java Grande, 2000, Pages 44 - 53.
- L. Adhianto and B. Chapman, "Performance modeling of communication and computation in hybrid MPI and OpenMP applications, Simulation Modeling Practice and Theory, vol 15, p. 481-491, 2007.
- Shah S, Haab G, Petersen P, Throop J. Flexible control structures for parallelism in OpenMP; Concurrency: Practice and Experience, 2000; 12:1219-1239. Publisher John Wiley & Sons, Ltd.
- Mattson, T.G., How Good is OpenMP? Scientific Programming, Vol. 11, Number 2, p.81-93, 2003.
- Duran A., Silvera R., Corbalan J., Labarta J., "Runtime Adjustment of Parallel Nested Loops", Shared Memory Parallel Programming with OpenMP, Lecture notes in Computer Science, Vol. 3349, P. 137, 2005

Appendices

- Sources for Additional information
- Solutions to exercises
  - Exercise 1: hello world
  - Exercise 2: Simple SPMD Pi program
  - Exercise 3: SPMD Pi without false sharing
  - Exercise 4: Loop level Pi
  - Exercise 5: Matrix multiplication
  - Exercise 6: Molecular dynamics
  - Exercise 7: linked lists with tasks
  - Exercise 8: linked lists without tasks
  - Exercise 9: Monte Carlo Pi and random numbers
- Flush and the OpenMP Memory model.
  - the producer consumer pattern
- Fortran and OpenMP
- Mixing OpenMP and MPI
- Compiler Notes

Exercise 1: Solution
A multi-threaded "Hello world" program

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

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

#pragma omp parallel
 {

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

OpenMP include file

Parallel region with default number of threads

End of the Parallel region

Runtime library function to return a thread ID.

Sample Output:

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

world(0)

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

Appendices

- Sources for Additional information
- Solutions to exercises
  - Exercise 1: hello world
  - Exercise 2: Simple SPMD Pi program
  - Exercise 3: SPMD Pi without false sharing
  - Exercise 4: Loop level Pi
  - Exercise 5: Matrix multiplication
  - Exercise 6: Molecular dynamics
  - Exercise 7: linked lists with tasks
  - Exercise 8: linked lists without tasks
  - Exercise 9: Monte Carlo Pi and random numbers
- Flush and the OpenMP Memory model.
  - the producer consumer pattern
- Fortran and OpenMP
- Mixing OpenMP and MPI
- Compiler Notes

Appendices

- Sources for Additional information
- Solutions to exercises
  - Exercise 1: hello world
  - Exercise 2: Simple SPMD Pi program
  - Exercise 3: SPMD Pi without false sharing
  - Exercise 4: Loop level Pi
  - Exercise 5: Matrix multiplication
  - Exercise 6: Molecular dynamics
  - Exercise 7: linked lists with tasks
  - Exercise 8: linked lists without tasks
  - Exercise 9: Monte Carlo Pi and random numbers
- Flush and the OpenMP Memory model.
  - the producer consumer pattern
- Fortran and OpenMP
- Mixing OpenMP and MPI
- Compiler Notes

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.
  - Result … poor scalability
- Solution:
  - When updates to an item are frequent, work with local copies of data instead of an array indexed by the thread ID.
  - Pad arrays so elements you use are on distinct cache lines.

Exercise 3: SPMD Pi without 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

Appendices

- Sources for Additional information
- Solutions to exercises
  - Exercise 1: hello world
  - Exercise 2: Simple SPMD Pi program
  - Exercise 3: SPMD Pi without false sharing
  - Exercise 4: Loop level Pi
  - Exercise 5: Matrix multiplication
  - Exercise 6: Molecular dynamics
  - Exercise 7: linked lists with tasks
  - Exercise 8: linked lists without tasks
  - Exercise 9: Monte Carlo Pi and random numbers
- Flush and the OpenMP Memory model.
  - the producer consumer pattern
- Fortran and OpenMP
- Mixing OpenMP and MPI
- Compiler Notes

Exercise 4: solution

```
#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;
}
```

Exercise 4: solution

```
#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 code and by adding 2 simple lines of text!

Appendices

- Sources for Additional information
- Solutions to exercises
  - Exercise 1: hello world
  - Exercise 2: Simple SPMD Pi program
  - Exercise 3: SPMD Pi without false sharing
  - Exercise 4: Loop level Pi
  - Exercise 5: Matrix multiplication
  - Exercise 6: Molecular dynamics
  - Exercise 7: linked lists with tasks
  - Exercise 8: linked lists without tasks
  - Exercise 9: Monte Carlo Pi and random numbers
- Flush and the OpenMP Memory model.
  - the producer consumer pattern
- Fortran and OpenMP
- Mixing OpenMP and MPI
- Compiler Notes

Matrix multiplication

```
#pragma omp parallel for private(tmp, i, j, k)
   for (i=0; i<Ndim; i++){
         for (j=0; j<Mdim; j++){
                  tmp = 0.0;
                  for(k=0;k<Pdim;k++){
                           /* C(i,j) = sum(over k) A(i,k) * B(k,j) */
                           tmp += *(A+(i*Ndim+k)) *  *(B+(k*Pdim+j));
                  }
                  *(C+(i*Ndim+j)) = tmp;
         }
   }
```

- On a dual core laptop
  - 13.2 seconds 153 Mflops one thread
  - 7.5 seconds 270 Mflops two threads

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

Appendices

- Sources for Additional information
- Solutions to exercises
  - Exercise 1: hello world
  - Exercise 2: Simple SPMD Pi program
  - Exercise 3: SPMD Pi without false sharing
  - Exercise 4: Loop level Pi
  - Exercise 5: Matrix multiplication
  - Exercise 6: Molecular dynamics
  - Exercise 7: linked lists with tasks
  - Exercise 8: linked lists without tasks
  - Exercise 9: Monte Carlo Pi and random numbers
- Flush and the OpenMP Memory model.
  - the producer consumer pattern
- Fortran and OpenMP
- Mixing OpenMP and MPI
- Compiler Notes

Exercise 6 solution

Compiler will warn you if you have missed some variables

```
#pragma omp parallel for default (none) \
    shared(x,f,npart,rcoff,side) \
    reduction(+:epot,vir) \
    schedule (static,32)
    for (int i=0; i<npart*3; i+=3) {
     ……
```

Loop is not well load balanced: best schedule has to be found by experiment.

Exercise 6 solution (cont.)

```
.....
#pragma omp atomic
        f[j]    -= forcex;
#pragma omp atomic
        f[j+1]  -= forcey;
#pragma omp atomic
        f[j+2]  -= forcez;
      }
    }
#pragma omp atomic
    f[i]    += fxi;
#pragma omp atomic
    f[i+1]   += fyi;
#pragma omp atomic
    f[i+2]   += fzi;
  }
 }
```

All updates to f must be atomic

Exercise 6 with orphaning

```
#pragma omp single
{
   vir    = 0.0;
   epot   = 0.0;
}
#pragma omp for reduction(+:epot,vir) \
    schedule (static,32)
    for (int i=0; i<npart*3; i+=3) {
……
```

Implicit barrier needed to avoid race condition with update of reduction variables at end of the for construct

All variables which used to be shared here are now implicitly determined

Exercise 6 with array reduction

```
      ftemp[myid][j]   -= forcex;
      ftemp[myid][j+1] -= forcey;
      ftemp[myid][j+2] -= forcez;
    }
  }
  ftemp[myid][i]       += fxi;
  ftemp[myid][i+1]     += fyi;
  ftemp[myid][i+2]     += fzi;
}
```

Replace atomics with accumulation into array with extra dimension

Exercise 6 with array reduction

```
....
#pragma omp for
    for(int i=0;i<(npart*3);i++){
        for(int id=0;id<nthreads;id++){
            f[i] += ftemp[id][i];
            ftemp[id][i] = 0.0;
        }
    }
```

Reduction can be done in parallel

Zero ftemp for next time round

Appendices

- Sources for Additional information
- Solutions to exercises
  - Exercise 1: hello world
  - Exercise 2: Simple SPMD Pi program
  - Exercise 3: SPMD Pi without false sharing
  - Exercise 4: Loop level Pi
  - Exercise 5: Matrix multiplication
  - Exercise 6: Molecular dynamics
  - Exercise 7: linked lists with tasks
  - Exercise 8: linked lists without tasks
  - Exercise 9: Monte Carlo Pi and random numbers
- Flush and the OpenMP Memory model.
  - the producer consumer pattern
- Fortran and OpenMP
- Mixing OpenMP and MPI
- Compiler Notes

Linked lists with tasks (intel taskq)

- See the file Linked_intel_taskq.c

```
#pragma omp parallel
{
   #pragma intel omp taskq
   {
     while (p != NULL) {
           #pragma intel omp task    captureprivate(p)
            processwork(p);
          p = p->next;
      }
   }
}
```

	Array, Static, 1	Intel taskq
One Thread	45 seconds	48 seconds
Two Threads	28 seconds	30 seconds

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

Linked lists with tasks (OpenMP 3)

- See the file Linked_omp3_tasks.c

```
#pragma omp parallel
{
  #pragma omp single
  {
     p=head;
    while (p) {
      #pragma omp task firstprivate(p)
            processwork(p);
        p = p->next;
    }
  }
}
```

Creates a task with its own copy of "p" initialized to the value of "p" when the task is defined

Appendices

- Sources for Additional information
- Solutions to exercises
  - Exercise 1: hello world
  - Exercise 2: Simple SPMD Pi program
  - Exercise 3: SPMD Pi without false sharing
  - Exercise 4: Loop level Pi
  - Exercise 5: Matrix multiplication
  - Exercise 6: Molecular dynamics
  - Exercise 7: linked lists with tasks
  - Exercise 8: linked lists without tasks
  - Exercise 9: Monte Carlo Pi and random numbers
- Flush and the OpenMP Memory model.
  - the producer consumer pattern
- Fortran and OpenMP
- Mixing OpenMP and MPI
- Compiler Notes

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

Linked lists without tasks: C++ STL

- See the file Linked_cpp.cpp

```
std::vector<node *> nodelist;
for (p = head; p != NULL; p = p->next)
   nodelist.push_back(p);
```

Copy pointer to each node into an array

```
int j = (int)nodelist.size();
#pragma omp parallel for schedule(static,1)
   for (int i = 0; i < j; ++i)
      processwork(nodelist[i]);
```

Count number of items in the linked list

Process nodes in parallel with a for loop

	C++, default sched.	C++, (static,1)	C, (static,1)
One Thread	37 seconds	49 seconds	45 seconds
Two Threads	47 seconds	32 seconds	28 seconds

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

Appendices

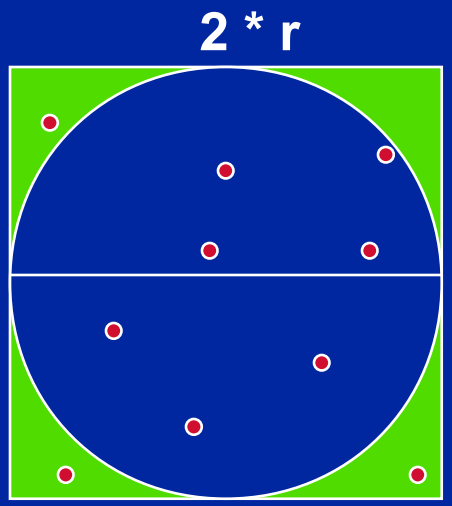
- Sources for Additional information
- Solutions to exercises
  - Exercise 1: hello world
  - Exercise 2: Simple SPMD Pi program
  - Exercise 3: SPMD Pi without false sharing
  - Exercise 4: Loop level Pi
  - Exercise 5: Matrix multiplication
  - Exercise 6: Molecular dynamics
  - Exercise 7: linked lists with tasks
  - Exercise 8: linked lists without tasks
  - Exercise 9: Monte Carlo Pi and random numbers
- Flush and the OpenMP Memory model.
  - the producer consumer pattern
- Fortran and OpenMP
- Mixing OpenMP and MPI
- Compiler Notes

Computers and random numbers

- We use “dice” to make random numbers:
  - Given previous values, you cannot predict the next value.
  - There are no patterns in the series … and it goes on forever.
- Computers are deterministic machines … set an initial state, run a sequence of predefined instructions, and you get a deterministic answer
  - By design, computers are not random and cannot produce random numbers.
- However, with some very clever programming, we can make “pseudo random” numbers that are as random as you need them to be … but only if you are very careful.
- Why do I care? Random numbers drive statistical methods used in countless applications:
  - Sample a large space of alternatives to find statistically good answers (Monte Carlo methods).

Monte Carlo Calculations:
Using Random numbers to solve tough problems

- Sample a problem domain to estimate areas, compute probabilities, find optimal values, etc.
- Example: Computing π with a digital dart board:



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

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

  $A_c = r^2 * \pi$

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

  $P = A_c/A_s = \pi /4$

- Compute π by randomly choosing points, count the fraction that falls in the circle, compute pi.

Parallel Programmers love Monte Carlo algorithms

Embarrassingly parallel: the parallelism is so easy its embarrassing.

Add two lines and you have a parallel program.

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

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

Linear Congruential Generator (LCG)

- LCG: Easy to write, cheap to compute, portable, OK quality

```
random_next = (MULTIPLIER  * random_last + ADDEND)% PMOD;
random_last = random_next;
```

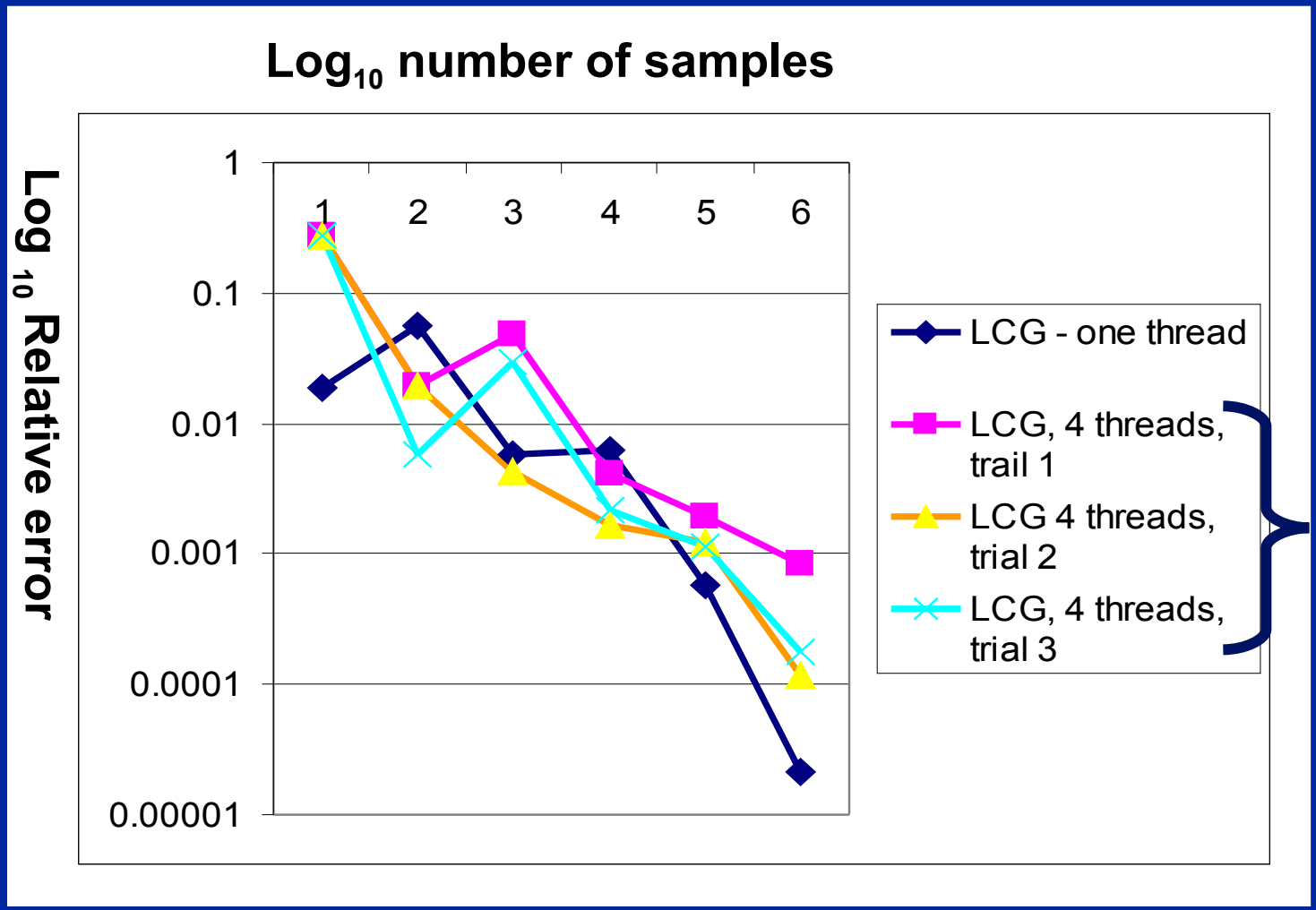
- If you pick the multiplier and addend correctly, LCG has a period of PMOD.
- Picking good LCG parameters is complicated, so look it up (Numerical Recipes is a good source). I used the following:
  - MULTIPLIER = 1366
  - ADDEND = 150889
  - PMOD = 714025

LCG code

```
static long MULTIPLIER  = 1366;
static long ADDEND      = 150889;
static long PMOD        = 714025;
long random_last = 0;
double random ()
{
    long random_next;

    random_next = (MULTIPLIER  * random_last + ADDEND)% PMOD;
    random_last = random_next;

   return  ((double)random_next/(double)PMOD);
}
```

Seed the pseudo random sequence by setting random_last

Running the PI_MC program with LCG generator



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

That is not acceptable!

Issue: my LCG generator is not threadsafe

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

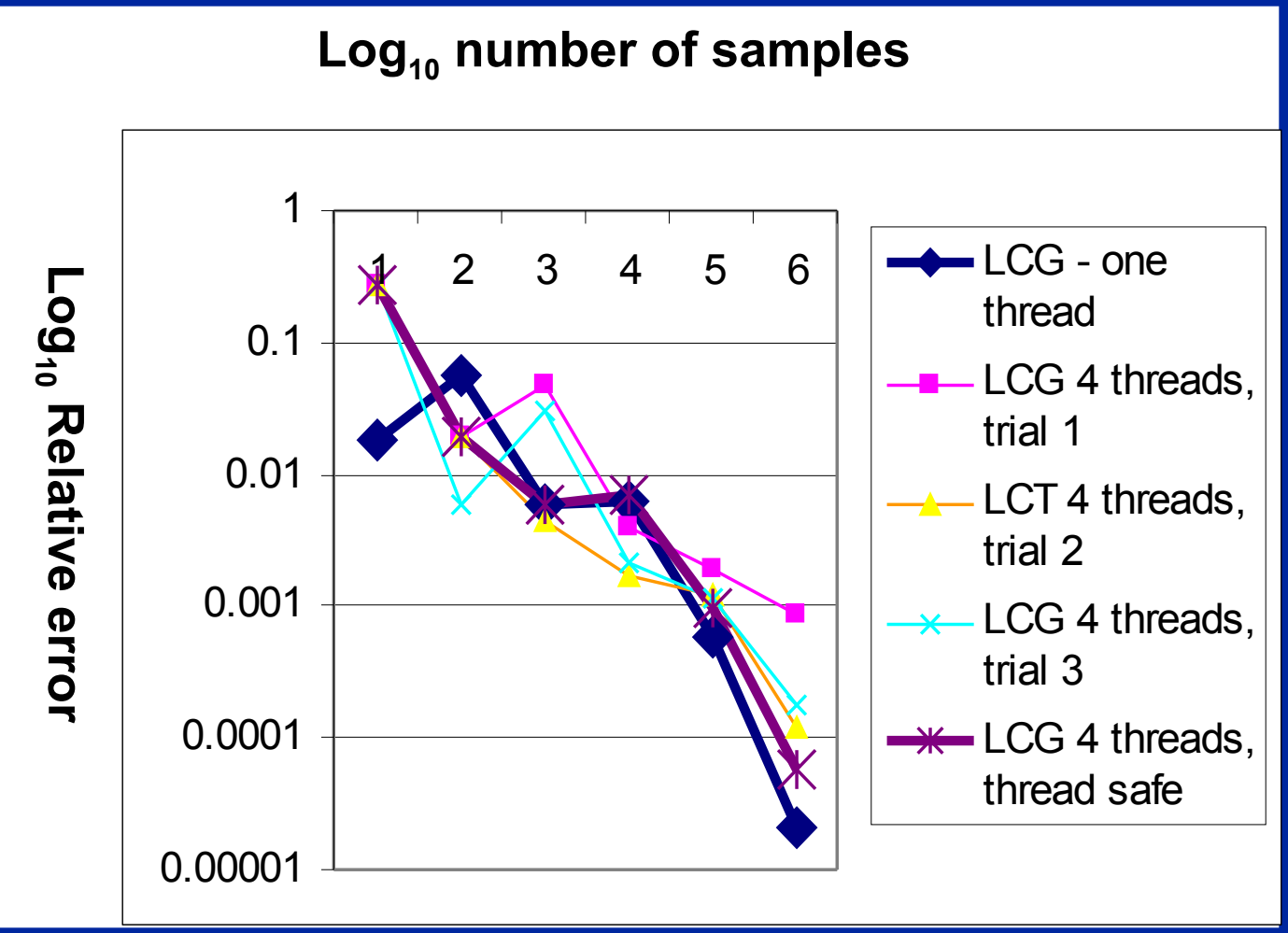
LCG code: threadsafe version

```
static long MULTIPLIER  = 1366;
static long ADDEND      = 150889;
static long PMOD        = 714025;
long random_last = 0;
#pragma omp threadprivate(random_last)
double random ()
{
   long random_next;

   random_next = (MULTIPLIER  * random_last + ADDEND)% PMOD;
   random_last = random_next;

  return  ((double)random_next/(double)PMOD);
}
```

random_last carries state between random number computations,

To make the generator threadsafe, make random_last threadprivate so each thread has its own copy.

Thread safe random number generators



Thread safe version gives the same answer each time you run the program.

But for large number of samples, its quality is lower than the one thread result!

Why?

Pseudo Random Sequences

- Random number Generators (RNGs) define a sequence of pseudo-random numbers of length equal to the period of the RNG

- In a typical problem, you grab a subsequence of the RNG range

- Grab arbitrary seeds and you may generate overlapping sequences
  - E.g. three sequences … last one wraps at the end of the RNG period.

- Overlapping sequences = over-sampling and bad statistics … lower quality or even wrong answers!

Parallel random number generators

- Multiple threads cooperate to generate and use random numbers.
- Solutions:
  - Replicate and Pray
  - Give each thread a separate, independent generator
  - Have one thread generate all the numbers.
  - Leapfrog … deal out sequence values "round robin" as if dealing a deck of cards.
  - Block method … pick your seed so each threads gets a distinct contiguous block.
- Other than "replicate and pray", these are difficult to implement. Be smart … buy a math library that does it right.

If done right, can generate the same sequence regardless of the number of threads …

Nice for debugging, but not really needed scientifically.

Intel's Math kernel Library supports all of these methods.

MKL Random number generators (RNG)

- MKL includes several families of RNGs in its vector statistics library.
- Specialized to efficiently generate vectors of random numbers

```
#define BLOCK 100
double  buff[BLOCK];
VSLStreamStatePtr stream;

vslNewStream(&ran_stream, VSL_BRNG_WH, (int)seed_val);

vdRngUniform (VSL_METHOD_DUNIFORM_STD, stream,
                    BLOCK, buff, low, hi)

vslDeleteStream( &stream );
```

Select type of RNG and set seed

Initialize a stream or pseudo random numbers

Fill buff with BLOCK pseudo rand. nums, uniformly distributed with values between lo and hi.

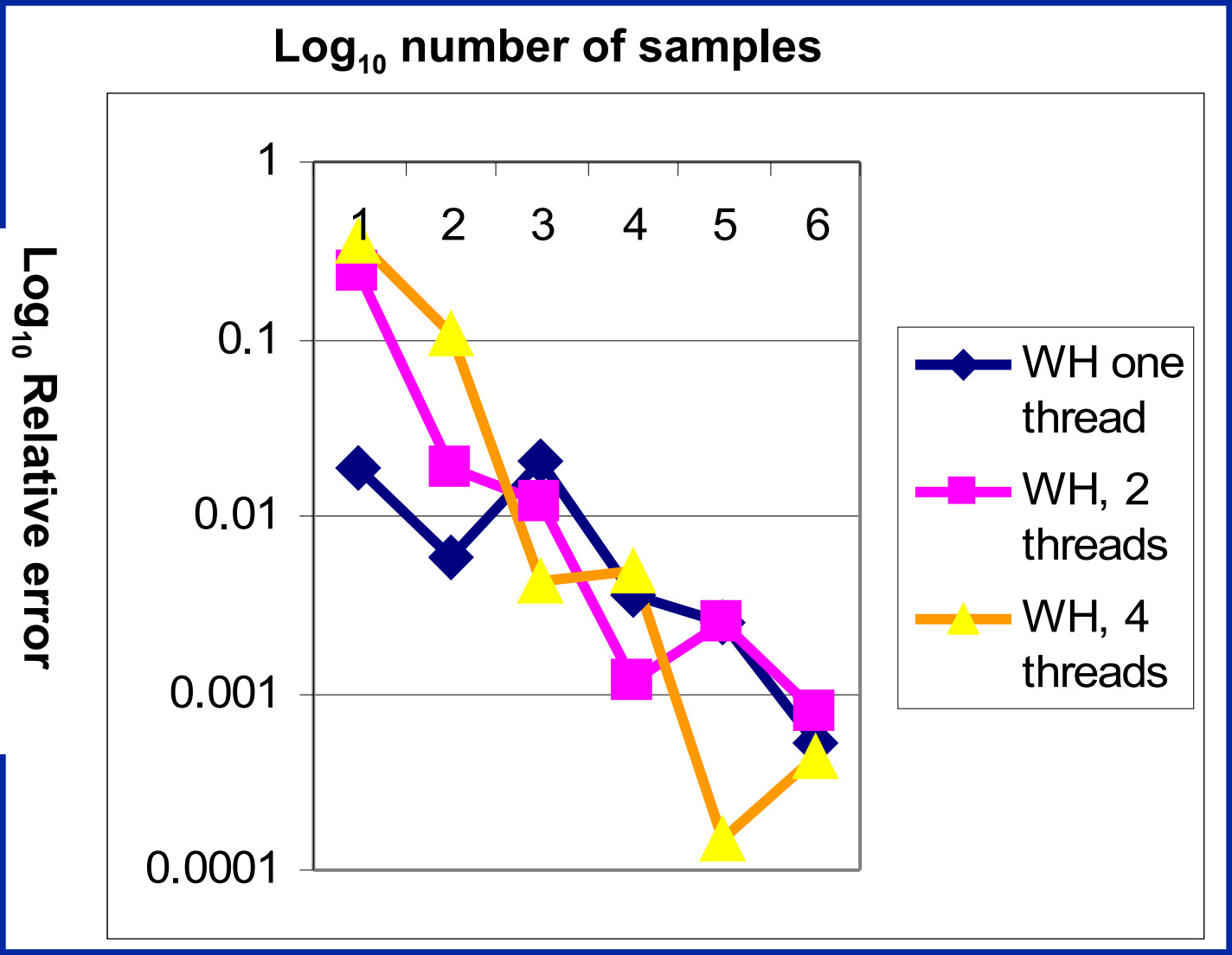
Delete the stream when you are done

Wichmann-Hill generators (WH)

- WH is a family of 273 parameter sets each defining a non-overlapping and independent RNG.
- Easy to use, just make each stream threadprivate and initiate RNG stream so each thread gets a unique WG RNG.

```
VSLStreamStatePtr stream;

#pragma omp threadprivate(stream)

                    …

vslNewStream(&ran_stream, VSL_BRNG_WH+Thrd_ID, (int)seed);
```

Independent Generator for each thread



Notice that once you get beyond the high error, small sample count range, adding threads doesn't decrease quality of random sampling.

Leap Frog method

- Interleave samples in the sequence of pseudo random numbers:
  - Thread i starts at the i^{th} number in the sequence
  - Stride through sequence, stride length = number of threads.
- Result … the same sequence of values regardless of the number of threads.

```
#pragma omp single
{   nthreads = omp_get_num_threads();
    iseed = PMOD/MULTIPLIER;     // just pick a seed
    pseed[0] = iseed;
    mult_n = MULTIPLIER;
    for (i = 1; i < nthreads; ++i)
   {
       iseed = (unsigned long long)((MULTIPLIER * iseed) % PMOD);
       pseed[i] = iseed;
       mult_n = (mult_n * MULTIPLIER) % PMOD;
   }

 }
 random_last = (unsigned long long) pseed[id];
```

One thread computes offsets and strided multiplier

LCG with Addend = 0 just to keep things simple

Each thread stores offset starting point into its threadprivate “last random” value

Same sequence with many threads.

- We can use the leapfrog method to generate the same answer for any number of threads

Steps	One thread	2 threads	4 threads
1000	3.156	3.156	3.156
10000	3.1168	3.1168	3.1168
100000	3.13964	3.13964	3.13964
1000000	3.140348	3.140348	3.140348
10000000	3.141658	3.141658	3.141658

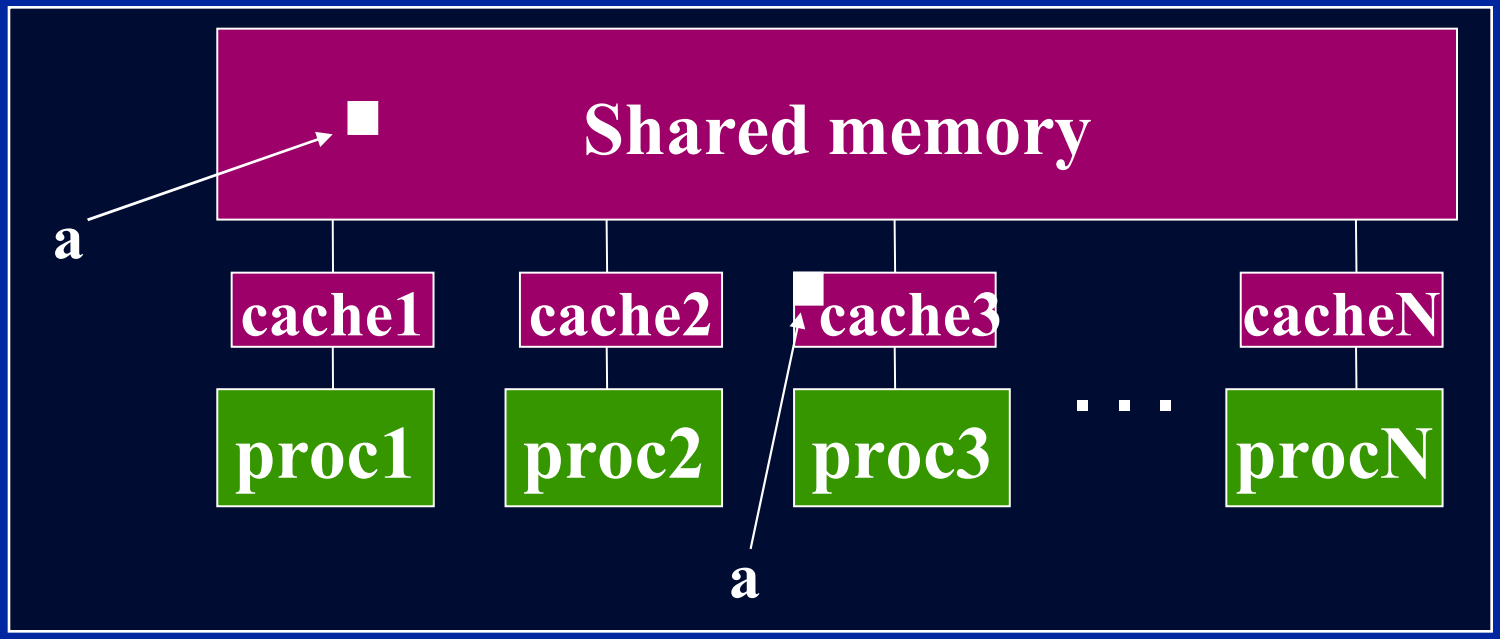
Used the MKL library with two generator streams per computation: one for the x values (WH) and one for the y values (WH+1). Also used the leapfrog method to deal out iterations among threads.

Appendices

- Sources for Additional information
- Solutions to exercises
  - Exercise 1: hello world
  - Exercise 2: Simple SPMD Pi program
  - Exercise 3: SPMD Pi without false sharing
  - Exercise 4: Loop level Pi
  - Exercise 5: Matrix multiplication
  - Exercise 6: Molecular dynamics
  - Exercise 7: linked lists with tasks
  - Exercise 8: linked lists without tasks
  - Exercise 9: Monte Carlo Pi and random numbers
- Flush and the OpenMP Memory model.
  - the producer consumer pattern
- Fortran and OpenMP
- Mixing OpenMP and MPI
- Compiler Notes

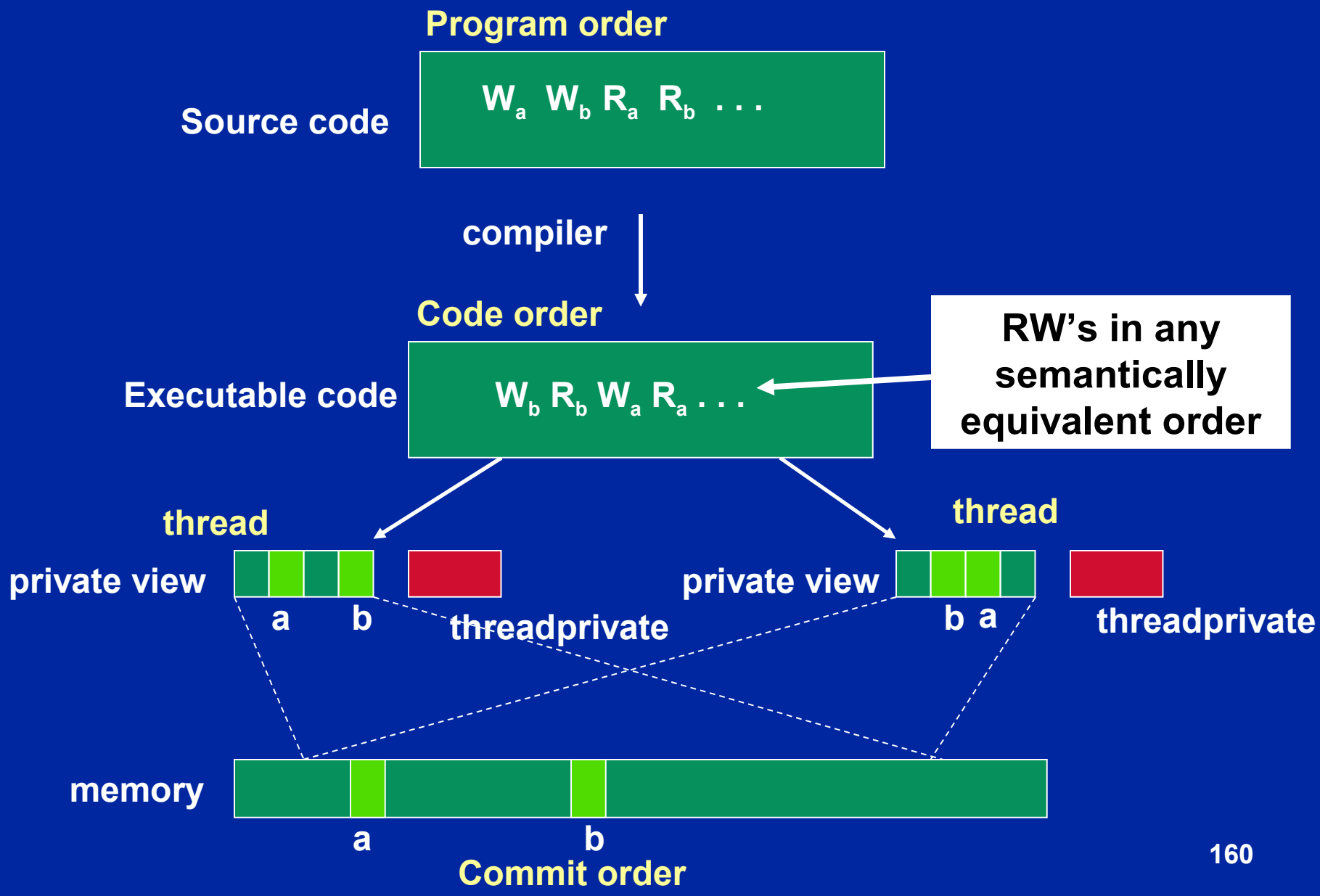
OpenMP memory model

- OpenMP supports a shared memory model.
- All threads share an address space, but it can get complicated:



- A memory model is defined in terms of:
  - Coherence: Behavior of the memory system when a single address is accessed by multiple threads.
  - Consistency: Orderings of reads, writes, or synchronizations (RWS) with various addresses and by multiple threads.

OpenMP Memory Model: Basic Terms



Consistency: Memory Access Re-ordering

- Re-ordering:
  - Compiler re-orders program order to the code order
  - Machine re-orders code order to the memory commit order
- At a given point in time, the "private view" seen by a thread may be different from the view in shared memory.
- Consistency Models define constraints on the orders of Reads (R), Writes (W) and Synchronizations (S)
  - … i.e. how do the values "seen" by a thread change as you change how ops follow (→) other ops.
  - Possibilities include:
    - R→R, W→W, R→W, R→S, S→S, W→S

Consistency

- Sequential Consistency:
  - In a multi-processor, ops (R, W, S) are sequentially consistent if:
    - They remain in program order for each processor.
    - They are seen to be in the same overall order by each of the other processors.
  - Program order = code order = commit order
- Relaxed consistency:
  - Remove some of the ordering constraints for memory ops (R, W, S).

OpenMP and Relaxed Consistency

- OpenMP defines consistency as a variant of weak consistency:
  - S ops must be in sequential order across threads.
  - Can not reorder S ops with R or W ops on the same thread
    - Weak consistency guarantees

      $S \rightarrow W,\quad S \rightarrow R,\ R \rightarrow S,\ W \rightarrow S,\ S \rightarrow S$
- The Synchronization operation relevant to this discussion is flush.

Flush

- Defines a sequence point at which a thread is guaranteed to see a consistent view of memory with respect to the “flush set”.
- The flush set is:
  - “all thread visible variables” for a flush construct without an argument list.
  - a list of variables when the “flush(list)” construct is used.
- The action of Flush is to guarantee that:
  - All R,W ops that overlap the flush set and occur prior to the flush complete before the flush executes
  - All R,W ops that overlap the flush set and occur after the flush don’t execute until after the flush.
  - Flushes with overlapping flush sets can not be reordered.

Memory ops: R = Read, W = write, S = synchronization

Synchronization: flush example

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

```
double A;

A = compute();

flush(A);   // flush to memory to make sure other
            //  threads can pick up the right value
```

Note: OpenMP's flush is analogous to a fence in other shared memory API's.

What is the Big Deal with Flush?

- Compilers routinely reorder instructions implementing a program
  - This helps better exploit the functional units, keep machine busy, hide memory latencies, etc.
- Compiler generally cannot move instructions:
  - past a barrier
  - past a flush on all variables
- But it can move them past a flush with a list of variables so long as those variables are not accessed
- Keeping track of consistency when flushes are used can be confusing … especially if "flush(list)" is used.

Note: the flush operation does not actually synchronize different threads. It just ensures that a thread's values are made consistent with main memory.

Appendices

- Sources for Additional information
- Solutions to exercises
  - Exercise 1: hello world
  - Exercise 2: Simple SPMD Pi program
  - Exercise 3: SPMD Pi without false sharing
  - Exercise 4: Loop level Pi
  - Exercise 5: Matrix multiplication
  - Exercise 6: Molecular dynamics
  - Exercise 7: linked lists with tasks
  - Exercise 8: linked lists without tasks
  - Exercise 9: Monte Carlo Pi and random numbers
- Flush and the OpenMP Memory model.
  - the producer consumer pattern
- Compiler Notes

Exercise 10: producer consumer

- Parallelize the “prod_cons.c” program.
- This is a well known pattern called the producer consumer pattern
  - One thread produces values that another thread consumes.
  - Often used with a stream of produced values to implement “pipeline parallelism”
- The key is to implement pairwise synchronization between threads.

Exercise 10: prod_cons.c

```
int main()
{
  double *A, sum, runtime;     int flag = 0;

  A = (double *)malloc(N*sizeof(double));

  runtime = omp_get_wtime();

  fill_rand(N, A);        // Producer: fill an array of data

  sum = Sum_array(N, A);  // Consumer: sum the array

  runtime = omp_get_wtime() - runtime;

  printf(" In %lf seconds, The sum is %lf \n",runtime,sum);
}
```

Pair wise synchronizaion in OpenMP

- OpenMP lacks synchronization constructs that work between pairs of threads.
- When this is needed you have to build it yourself.
- Pair wise synchronization
  - Use a shared flag variable
  - Reader spins waiting for the new flag value
  - Use flushes to force updates to and from memory

Exercise 10: producer consumer

```
int main()
{
   double *A, sum, runtime;     int numthreads, flag = 0;
   A = (double *)malloc(N*sizeof(double));

   #pragma omp parallel sections
   {
     #pragma omp section
      {
        fill_rand(N, A);
        #pragma omp flush
        flag = 1;
        #pragma omp flush (flag)
      }
      #pragma omp section
      {
        #pragma omp flush (flag)
        while (flag != 1){
            #pragma omp flush (flag)
        }
        #pragma omp flush
        sum = Sum_array(N, A);
      }
    }
}
```

Use flag to Signal when the "produced" value is ready

Flush forces refresh to memory. Guarantees that the other thread sees the new value of A

Flush needed on both "reader" and "writer" sides of the communication

Notice you must put the flush inside the while loop to make sure the updated flag variable is seen

Appendices

- Sources for Additional information
- Solutions to exercises
  - Exercise 1: hello world
  - Exercise 2: Simple SPMD Pi program
  - Exercise 3: SPMD Pi without false sharing
  - Exercise 4: Loop level Pi
  - Exercise 5: Matrix multiplication
  - Exercise 6: Molecular dynamics
  - Exercise 7: linked lists with tasks
  - Exercise 8: linked lists without tasks
  - Exercise 9: Monte Carlo Pi and random numbers
- Flush and the OpenMP Memory model.
  - the producer consumer pattern
- Fortran and OpenMP
- Mixing OpenMP and MPI
- Compiler Notes

Fortran and OpenMP

- We were careful to design the OpenMP constructs so they cleanly map onto C, C++ and Fortran.
- There are a few syntactic differences that once understood, will allow you to move back and forth between languages.
- In the specification, language specific notes are included when each construct is defined.

OpenMP:
Some syntax details for Fortran programmers

- Most of the constructs in OpenMP are compiler directives.
  - For Fortran, the directives take one of the forms:

    C$OMP *construct [clause [clause]…]*

    !$OMP *construct [clause [clause]…]*

    *$OMP *construct [clause [clause]…]*
- The OpenMP include file and lib module

  use omp_lib

  Include omp_lib.h

OpenMP:
Structured blocks (Fortran)

- Most OpenMP constructs apply to structured blocks.
  - Structured block: a block of code with one point of entry at the top and one point of exit at the bottom.
  - The only "branches" allowed are STOP statements in Fortran

```
C$OMP PARALLEL
10    wrk(id) = garbage(id)
      res(id) = wrk(id)**2
      if(conv(res(id)) goto 10
C$OMP END PARALLEL
      print *,id
```

A structured block

```
C$OMP  PARALLEL
10    wrk(id) = garbage(id)
30    res(id)=wrk(id)**2
      if(conv(res(id))goto 20
      go to 10
C$OMP END PARALLEL
      if(not_DONE) goto 30
20    print *, id
```

Not A structured block

OpenMP:
Structured Block Boundaries

- In Fortran: a block is a single statement or a group of statements between directive/end-directive pairs.

```
C$OMP PARALLEL
10    wrk(id) = garbage(id)
      res(id) = wrk(id)**2
      if(conv(res(id)) goto 10
C$OMP END PARALLEL
```

```
C$OMP PARALLEL DO
      do I=1,N
            res(I)=bigComp(I)
      end do
C$OMP END PARALLEL DO
```

- The "construct/end construct" pairs is done anywhere a structured block appears in Fortran. Some examples:
  - DO … END DO
  - PARALLEL … END PARREL
  - CRICITAL … END CRITICAL
  - SECTION … END SECTION
  - SECTIONS … END SECTIONS
  - SINGLE … END SINGLE
  - MASTER … END MASTER

Runtime library routines

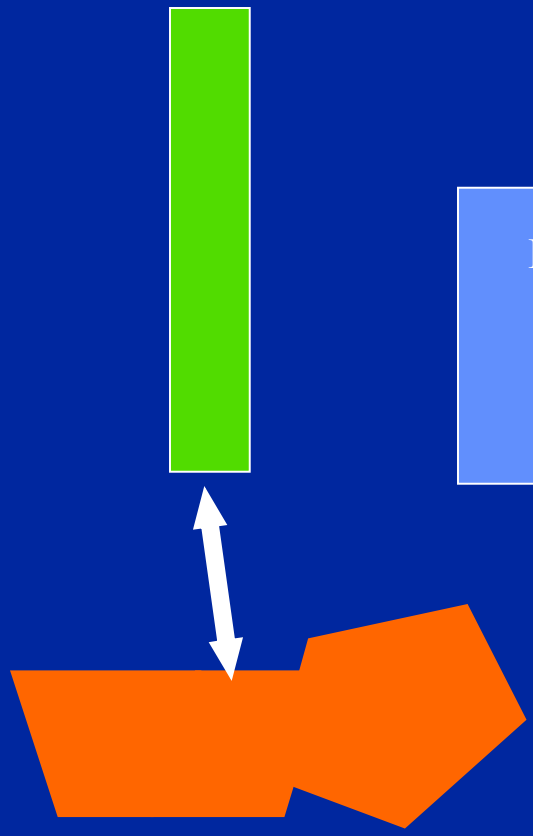
- The include file or module defines parameters
  - Integer parameter omp_locl_kind
  - Integer parameter omp_nest_lock_kind
  - Integer parameter omp_sched_kind
  - Integer parameter openmp_version
    - With value that matches C's _OPEMMP macro
- Fortran interfaces are similar to those used with C
  - Subroutine omp_set_num_threads (num_threads)
  - Integer function omp_get_num_threads()
  - Integer function omp_get_thread_num()\
  - Subroutine omp_init_lock(svar)
    - Integer(kind=omp_lock_kind) svar
  - Subroutine omp_destroy_lock(svar)
  - Subroutine omp_set_lock(svar)
  - Subroutine omp_unset_lock(svar)

Appendices

- Sources for Additional information
- Solutions to exercises
  - Exercise 1: hello world
  - Exercise 2: Simple SPMD Pi program
  - Exercise 3: SPMD Pi without false sharing
  - Exercise 4: Loop level Pi
  - Exercise 5: Matrix multiplication
  - Exercise 6: Molecular dynamics
  - Exercise 7: linked lists with tasks
  - Exercise 8: linked lists without tasks
  - Exercise 9: Monte Carlo Pi and random numbers
- Flush and the OpenMP Memory model.
  - the producer consumer pattern
- Fortran and OpenMP
- Mixing OpenMP and MPI
- Compiler Notes

How do people mix MPI and OpenMP?

A sequential program working on a data set

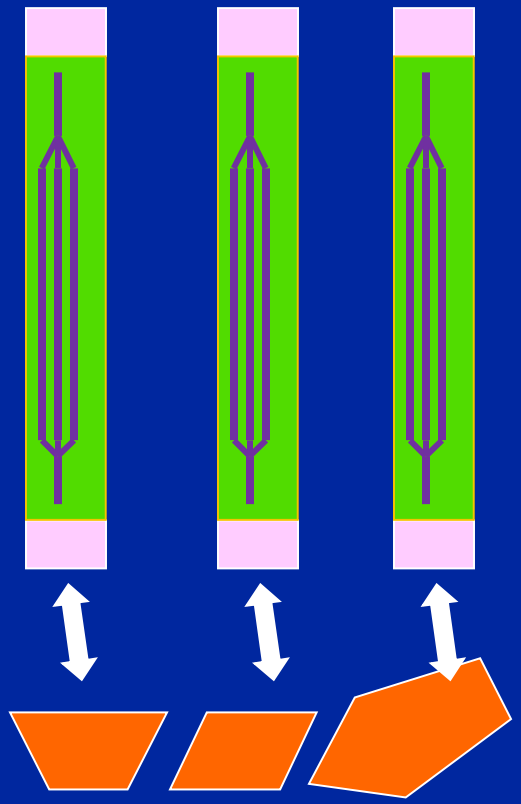


Replicate the program.

Add glue code

Break up the data

- Create the MPI program with its data decomposition.
- Use OpenMP inside each MPI process.



Pi program with MPI and OpenMP

```
#include <mpi.h>
#include "omp.h"
void main (int argc, char *argv[])
{
        int i, my_id, numprocs;  double x, pi, step, sum = 0.0 ;
        step = 1.0/(double) num_steps ;
        MPI_Init(&argc, &argv) ;
        MPI_Comm_Rank(MPI_COMM_WORLD, &my_id) ;
        MPI_Comm_Size(MPI_COMM_WORLD, &numprocs) ;
        my_steps = num_steps/numprocs ;
#pragma omp parallel for reduction(+:sum) private(x)
        for (i=my_id*my_steps; i<(m_id+1)*my_steps ; i++)
        {
                x = (i+0.5)*step;
                sum += 4.0/(1.0+x*x);
        }
        sum *= step ;
        MPI_Reduce(&sum, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,
                MPI_COMM_WORLD) ;
}
```

Get the MPI part done first, then add OpenMP pragma where it makes sense to do so

Key issues when mixing OpenMP and MPI

1. Messages are sent to a process not to a particular thread.
   - Not all MPIs are threadsafe. MPI 2.0 defines threading modes:
     - MPI_Thread_Single: no support for multiple threads
     - MPI_Thread_Funneled: Mult threads, only master calls MPI
     - MPI_Thread_Serialized: Mult threads each calling MPI, but they do it one at a time.
     - MPI_Thread_Multiple: Multiple threads without any restrictions
   - Request and test thread modes with the function:

     MPI_init_thread(desired_mode, delivered_mode, ierr)
1. Environment variables are not propagated by mpirun. You'll need to broadcast OpenMP parameters and set them with the library routines.

Dangerous Mixing of MPI and OpenMP

- The following will work only if MPI_Thread_Multiple is supported … a level of support I wouldn't depend on.

```
MPI_Comm_Rank(MPI_COMM_WORLD, &mpi_id) ;
#pragma omp parallel
{
    int tag, swap_neigh, stat, omp_id = omp_thread_num();
    long buffer [BUFF_SIZE], incoming [BUFF_SIZE];
    big_ugly_calc1(omp_id, mpi_id, buffer);
                                        // Finds MPI id and tag
so
    neighbor(omp_id, mpi_id, &swap_neigh, &tag);  // messages don't
conflict

    MPI_Send (buffer,   BUFF_SIZE, MPI_LONG, swap_neigh,
              tag, MPI_COMM_WORLD);
    MPI_Recv (incoming, buffer_count, MPI_LONG, swap_neigh,
              tag,  MPI_COMM_WORLD, &stat);

    big_ugly_calc2(omp_id, mpi_id, incoming, buffer);
#pragma critical
    consume(buffer, omp_id, mpi_id);
```

Messages and threads

- Keep message passing and threaded sections of your program separate:
  - Setup message passing outside OpenMP parallel regions (MPI_Thread_funneled)
  - Surround with appropriate directives (e.g. critical section or master) (MPI_Thread_Serialized)
  - For certain applications depending on how it is designed it may not matter which thread handles a message. (MPI_Thread_Multiple)
    - Beware of race conditions though if two threads are probing on the same message and then racing to receive it.

Safe Mixing of MPI and OpenMP

Put MPI in sequential regions

```
MPI_Init(&argc, &argv) ;      MPI_Comm_Rank(MPI_COMM_WORLD, &mpi_id) ;

// a whole bunch of initializations

#pragma omp parallel for
for (I=0;I<N;I++) {
    U[I] =  big_calc(I);
}

    MPI_Send (U,   BUFF_SIZE, MPI_DOUBLE, swap_neigh,
              tag, MPI_COMM_WORLD);
     MPI_Recv (incoming, buffer_count, MPI_DOUBLE, swap_neigh,
              tag,  MPI_COMM_WORLD, &stat);

#pragma omp parallel for
for (I=0;I<N;I++) {
    U[I] =  other_big_calc(I, incoming);
}

consume(U, mpi_id);
```

Technically Requires MPI_Thread_funneled, but I have never had a problem with this approach … even with pre-MPI-2.0 libraries.

Safe Mixing of MPI and OpenMP

Protect MPI calls inside a parallel region

```
MPI_Init(&argc, &argv) ;     MPI_Comm_Rank(MPI_COMM_WORLD, &mpi_id) ;

// a whole bunch of initializations

#pragma omp parallel
{
#pragma omp for
   for (I=0;I<N;I++)    U[I] =  big_calc(I);

#pragma master
{
    MPI_Send (U,   BUFF_SIZE, MPI_DOUBLE, neigh, tag,  MPI_COMM_WORLD);
     MPI_Recv (incoming, count, MPI_DOUBLE, neigh,  tag,  MPI_COMM_WORLD,
                                                                  &stat);
}
#pragma omp barrier
#pragma omp for
   for (I=0;I<N;I++)   U[I] =  other_big_calc(I, incoming);

#pragma omp master
   consume(U, mpi_id);
}
```

Technically Requires MPI_Thread_funneled, but I have never had a problem with this approach … even with pre-MPI-2.0 libraries.

Hybrid OpenMP/MPI works, but is it worth it?

- Literature* is mixed on the hybrid model: sometimes its better, sometimes MPI alone is best.
- There is potential for benefit to the hybrid model
  - MPI algorithms often require replicated data making them less memory efficient.
  - Fewer total MPI communicating agents means fewer messages and less overhead from message conflicts.
  - Algorithms with good cache efficiency should benefit from shared caches of multi-threaded programs.
  - The model maps perfectly with clusters of SMP nodes.
- But really, it's a case by case basis and to large extent depends on the particular application.

*L. Adhianto and Chapman, 2007

Appendices

- Sources for Additional information
- Solutions to exercises
  - Exercise 1: hello world
  - Exercise 2: Simple SPMD Pi program
  - Exercise 3: SPMD Pi without false sharing
  - Exercise 4: Loop level Pi
  - Exercise 5: Matrix multiplication
  - Exercise 6: Molecular dynamics
  - Exercise 7: linked lists with tasks
  - Exercise 8: linked lists without tasks
  - Exercise 9: Monte Carlo Pi and random numbers
- Flush and the OpenMP Memory model.
  - the producer consumer pattern
- Fortran and OpenMP
- Mixing OpenMP and MPI
- Compiler Notes

Compiler notes: Intel on Windows

- Intel compiler:
  - Launch SW dev environment … on my laptop I use:
    - start/intel software development tools/intel C++ compiler 11.0/C+ build environment for 32 bit apps
  - cd to the directory that holds your source code
  - Build software for program foo.c
    - icl /Qopenmp foo.c
  - Set number of threads environment variable
    - set OMP_NUM_THREADS=4
  - Run your program
    - foo.exe

To get rid of the pwd on the prompt, type

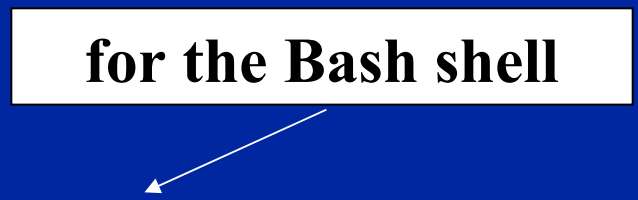
prompt = %

Compiler notes: Visual Studio

- Start “new project”
- Select win 32 console project
  - Set name and path
  - On the next panel, Click “next” instead of finish so you can select an empty project on the following panel.
  - Drag and drop your source file into the source folder on the visual studio solution explorer
  - Activate OpenMP
    - Go to project properties/configuration properties/C.C++/language … and activate OpenMP
- Set number of threads inside the program
- Build the project
- Run “without debug” from the debug menu.

Compiler notes: Other

- Linux and OS X with gcc:
  - 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



OpenMP constructs

- #pragma omp parallel
- #pragma omp for
- #pragma omp critical
- #pragma omp atomic
- #pragma omp barrier
- Data environment clauses
  - private (variable_list)
  - firstprivate (variable_list)
  - lastprivate (variable_list)
  - reduction(+:variable_list)
- Tasks (remember … private data is made firstprivate by default)
  - pragma omp task
  - pragma omp taskwait
- #pragma threadprivate(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

Put this on a line right after you define the variables in question