



A “Hands-on” Introduction to OpenMP*

Tim Mattson

Intel Corp.

timothy.g.mattson@intel.com

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

Major OpenMP constructs we've covered so far

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

Where `variable_list` is a comma separated list of variables

Print the value of the macro

`_OPENMP`

And its value will be

`yyyymm`

For the year and month of the spec the implementation used

Outline

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



Consider simple list traversal

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

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

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

Linked lists without tasks

- See the file `Linked_omp25.c`

```
while (p != NULL) {  
    p = p->next;  
    count++;  
}
```

Count number of items in the linked list

```
p = head;  
for(i=0; i<count; i++) {  
    parr[i] = p;  
    p = p->next;  
}
```

Copy pointer to each node into an array

```
#pragma omp parallel  
{  
    #pragma omp for schedule(static,1)  
    for(i=0; i<count; i++)  
        processwork(parr[i]);  
}
```


Process nodes in parallel with a for loop

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


Conclusion

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

Outline

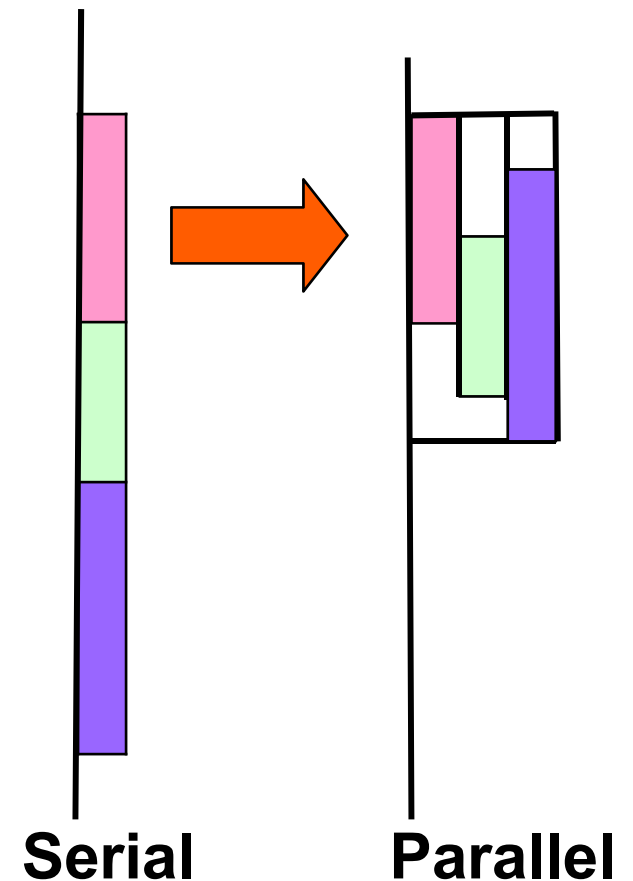
- **Unit 1: Getting started with OpenMP**
 - ♦ Mod1: Introduction to parallel programming
 - ♦ Mod 2: The boring bits: Using an OpenMP compiler (hello world)
 - ♦ Disc 1: Hello world and how threads work
- **Unit 2: The core features of OpenMP**
 - ♦ Mod 3: Creating Threads (the Pi program)
 - ♦ Disc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

Outline

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

OpenMP Tasks

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



Definitions

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

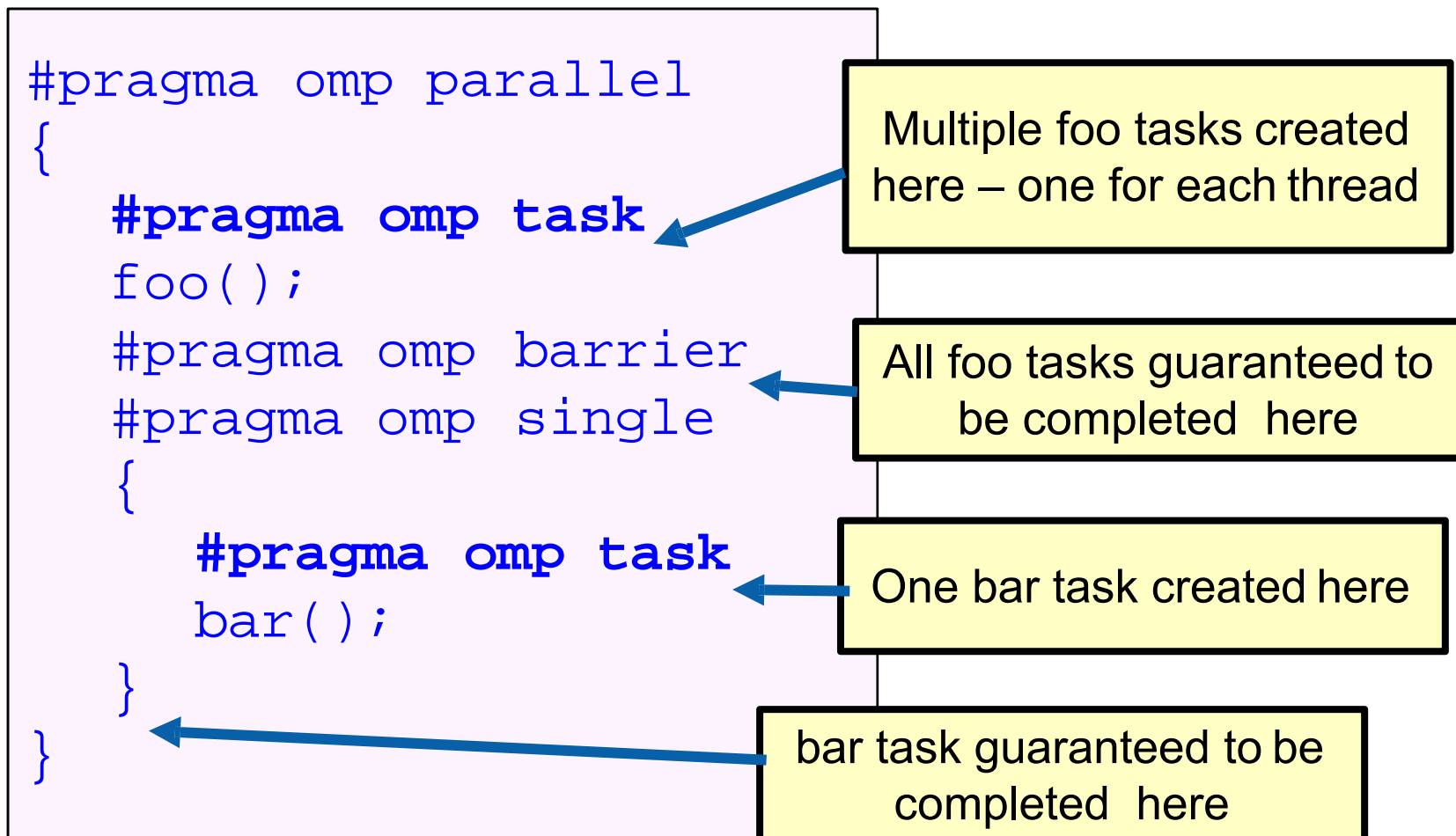
When are tasks guaranteed to complete

- Tasks are guaranteed to be complete at thread barriers:

`#pragma omp barrier`

- or task barriers

`#pragma omp taskwait`



Data Scoping with tasks: Fibonacci example.

This is an instance of the
divide and conquer design
pattern

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

n is private in both tasks

x is a private variable
y is a private variable

What's wrong here?

**A task's private variables are
undefined outside the task**

Data Scoping with tasks: Fibonacci example.

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

n is private in both tasks

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

Data Scoping with tasks: List Traversal example


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

What's wrong here?

**Possible data race !
Shared variable e
updated by multiple tasks**


Data Scoping with tasks: List Traversal example

```
List m1; //my_list
Element *e;
#pragma omp parallel
#pragma omp single
{
    for(e=m1->first;e;e=e->next)
    #pragma omp task firstprivate(e)
        process(e);
}
```



Good solution – e is
firstprivate

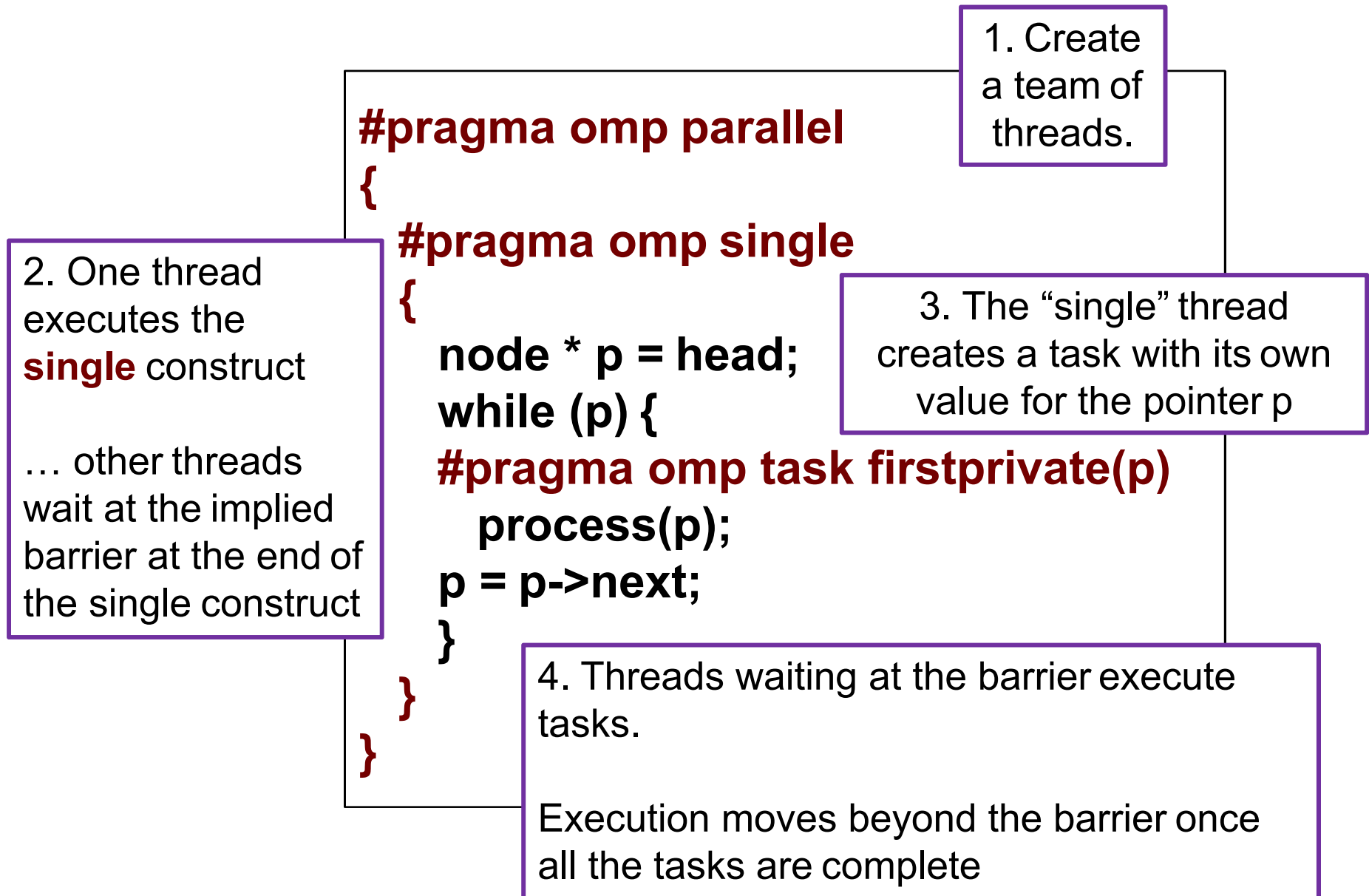
Outline

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

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.

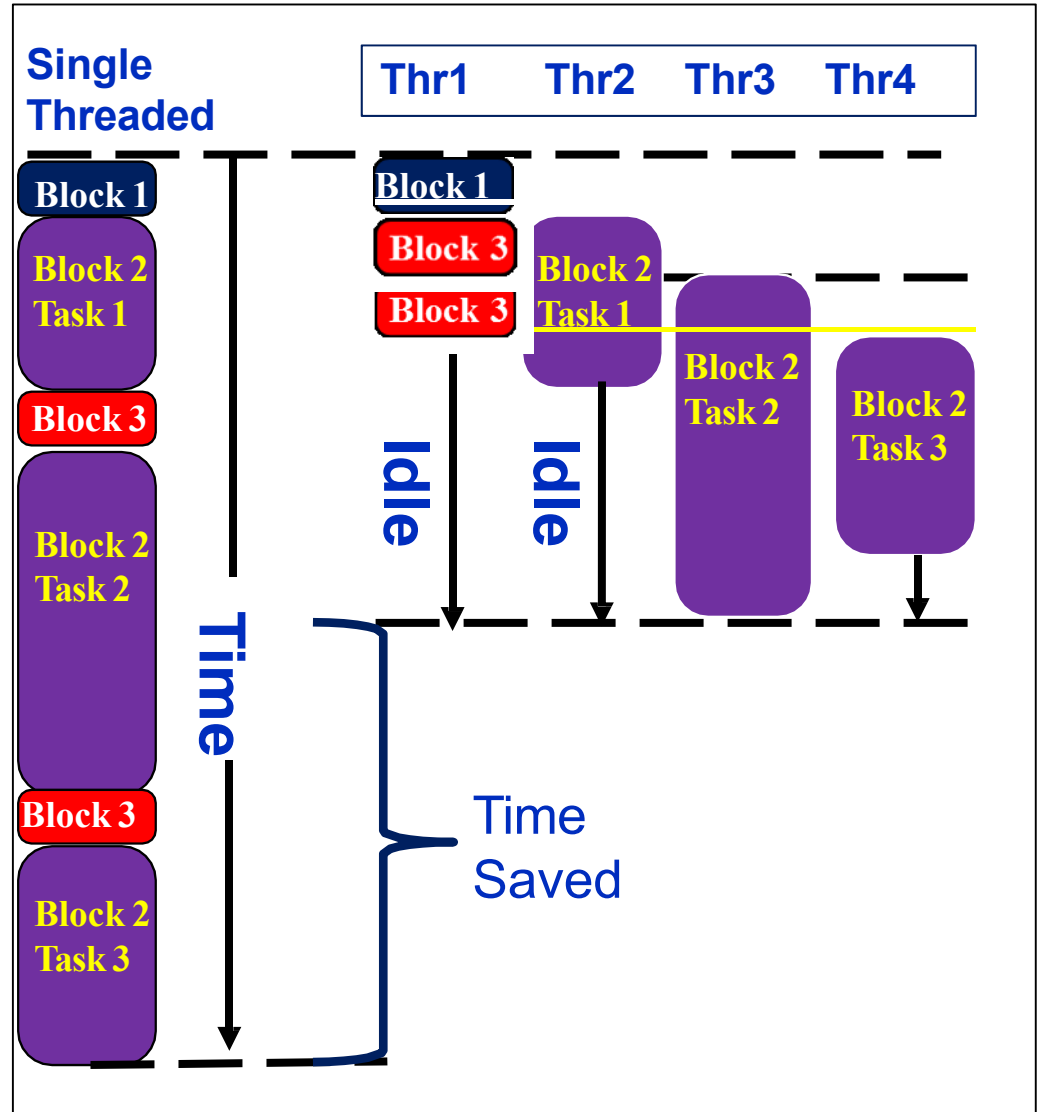
Task Construct – Explicit Tasks




Execution of tasks

Have potential to parallelize irregular patterns and recursive function calls

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

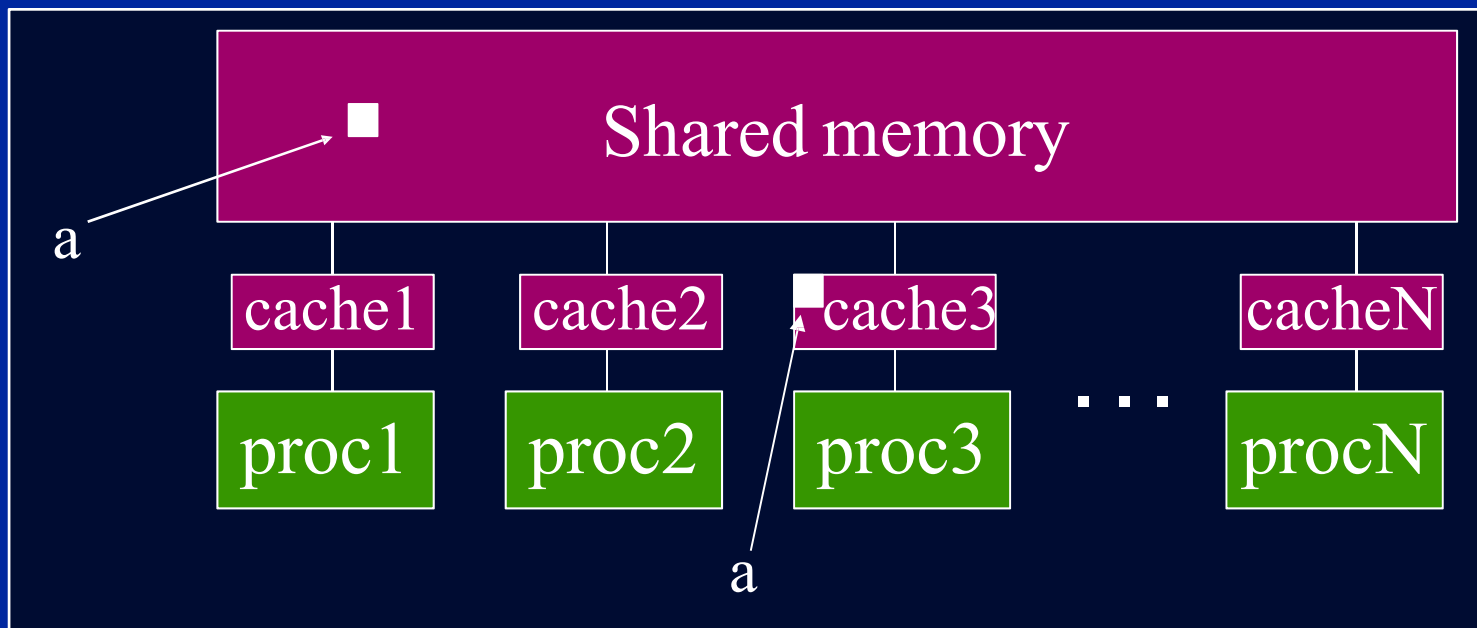


Outline

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

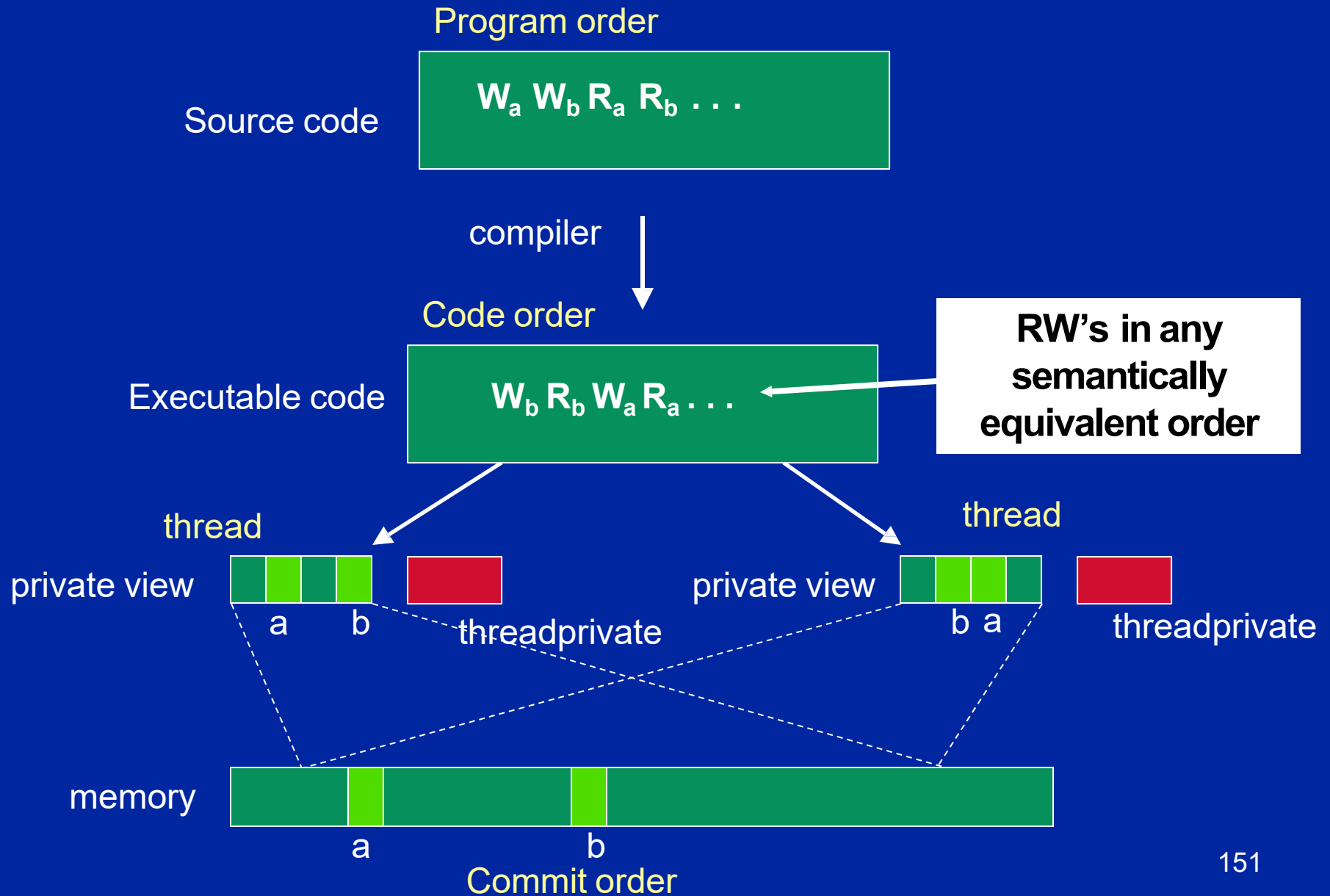
OpenMP memory model

- 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 (\rightarrow) other ops.
 - ◆ Possibilities include:
 - $R \rightarrow R$, $W \rightarrow W$, $R \rightarrow W$, $R \rightarrow S$, $S \rightarrow S$, $W \rightarrow S$

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:
 - ◆ Can not reorder S ops with R or W ops on the same thread
 - **Weak consistency guarantees**
 $S \rightarrow W, S \rightarrow R, R \rightarrow S, W \rightarrow S, S \rightarrow S$
- 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.

Flush and synchronization

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

What 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.

Outline

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

Example: prod_cons.c

- Parallelize a producer consumer program
 - One thread produces values that another thread consumes.

```
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 secs, The sum is %lf \n",runtime,sum);
}
```

- Often used with a stream of produced values to implement “pipeline parallelism”
- The key is to implement pairwise synchronization between threads.

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

Example: 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 == 0){
                #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

The problem is this program technically has
a race ... on the store and later load of flag.

The OpenMP 3.1 atomics (1 of 2)

- Atomic was expanded to cover the full range of common scenarios where you need to protect a memory operation so it occurs atomically:

pragma omp atomic [read | write | update | capture]

- Atomic can protect loads

pragma omp atomic read

v = x;

- Atomic can protect stores

pragma omp atomic write

x = expr;

- Atomic can protect updates to a storage location (this is the default behavior ... i.e. when you don't provide a clause)

pragma omp atomic update

x++; or ++x; or x--; or --x; or

x binop= expr; or x = x binop expr;

**This is the
original OpenMP
atomic**

The OpenMP 3.1 atomics (2 of 2)

- Atomic can protect the assignment of a value (its capture) AND an associated update operation:

pragma omp atomic capture
statement or structured block

- Where the statement is one of the following forms:

v = x++; v = ++x; v = x--; v = --x; v = x binop expr;

- Where the structured block is one of the following forms:

{v = x; x binop = expr;}

{x binop = expr; v = x;}

{v=x; x=x binop expr;}

{X = x binop expr; v = x;}

{v = x; x++;}

{v=x; ++x;}

{++x; v=x;}

{x++; v = x;}

{v = x; x--;}

{v= x; --x;}

{--x; v = x;}

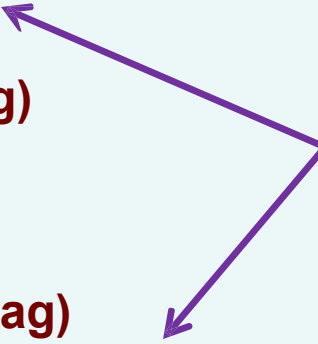
{x--; v = x;}

The capture semantics in atomic were added to map onto common hardware supported atomic ops and to support modern lock free algorithms.


Atoms and synchronization flags

```
int main()
{ double *A, sum, runtime;
  int numthreads, flag = 0, flg_tmp;
  A = (double *)malloc(N*sizeof(double));
  #pragma omp parallel sections
  {
    #pragma omp section
    { fill_rand(N, A);
      #pragma omp flush
      #pragma atomic write
      flag = 1;
      #pragma omp flush (flag)
    }
    #pragma omp section
    { while (1){
        #pragma omp flush(flag)
        #pragma omp atomic read
        flg_tmp= flag;
        if (flg_tmp==1) break;
      }
      #pragma omp flush
      sum = Sum_array(N, A);
    }
  }
}
```

This program is truly race free ... the reads and writes of flag are protected so the two threads can not conflict.



Outline

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

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 initialised
... 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 private variables 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;

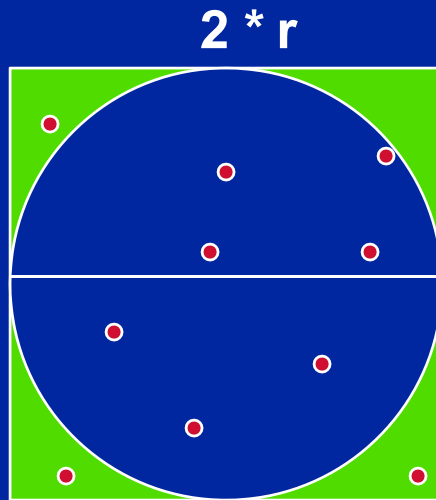
    #pragma omp parallel private (Nsize, choice)
    {
        #pragma omp single copyprivate (Nsize, choice)
            input_parameters (Nsize, choice);

        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	$\pi = 2.8$
N=100	$\pi = 3.16$
N= 1000	$\pi = 3.148$

- 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 π .

Outline

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



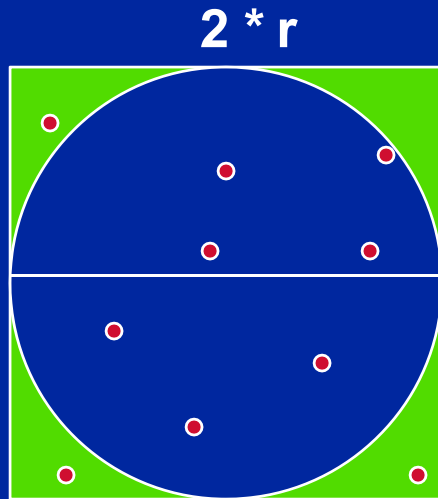
Computers and random numbers

- 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	$\pi = 2.8$
N=100	$\pi = 3.16$
N= 1000	$\pi = 3.148$

- 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

```
#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 square 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);
}
```

Embarrassingly parallel: the parallelism is so easy its embarrassing.

Add two lines and you have a parallel program.

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

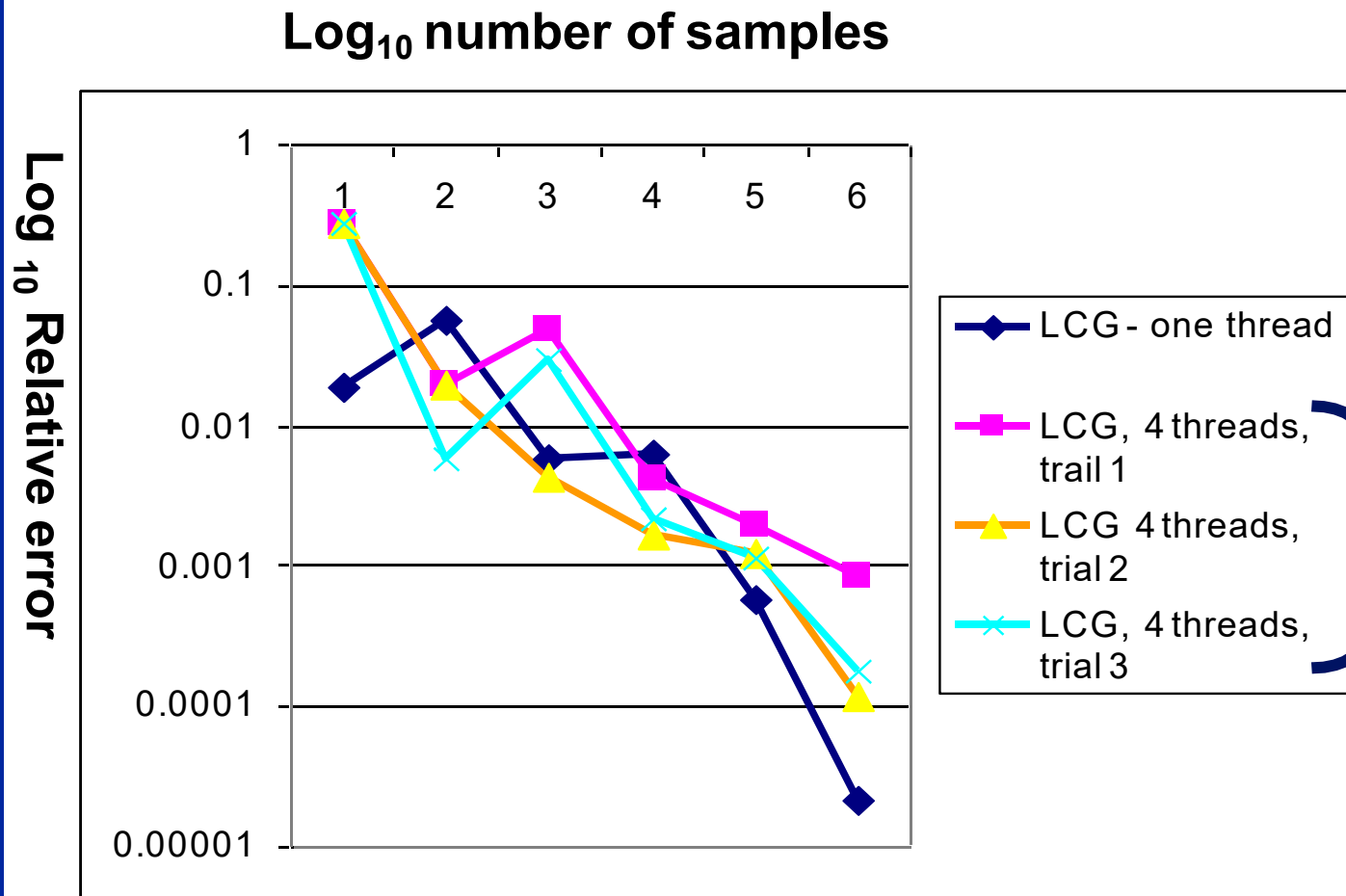
```
static long MULTIPLIER = 1366;
static long ADDEND     = 150889;
static long PMOD       = 714025;
long random_last = 0;
double random ()
{
    long random_next;

    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

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;

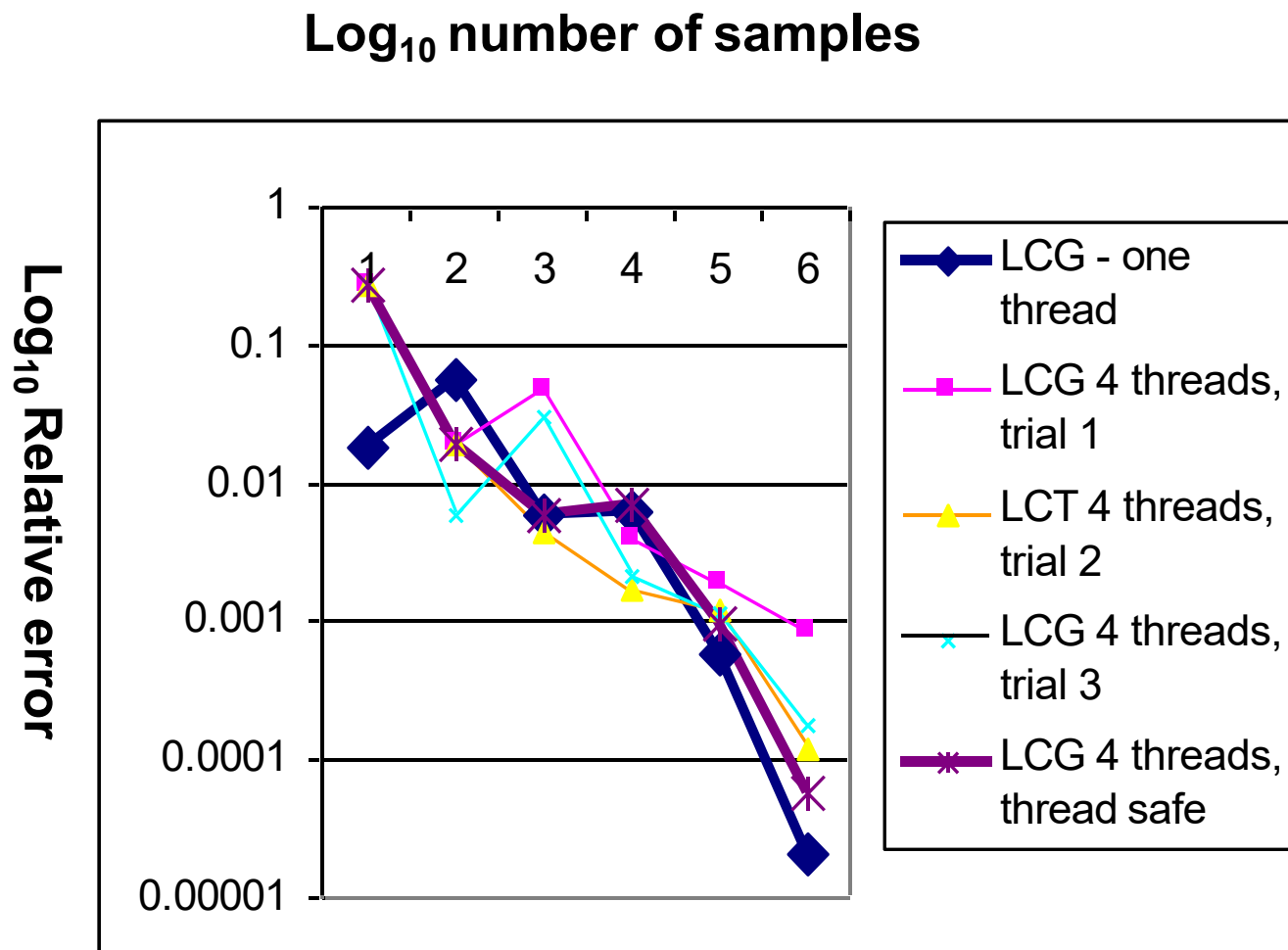
    random_next = (MULTIPLIER * random_last + ADDEND)% PMOD;
    random_last = random_next;

    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.

Outline

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

Summary

- We have now covered the most commonly used features of OpenMP.
- To close, let's consider some of the key parallel design patterns we've discussed..

SPMD: Single Program Multiple Data

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

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

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



OpenMP Pi program: SPMD pattern

```
#include <omp.h>
void main (int argc, char *argv[])
{
    int i, pi=0.0, step, sum = 0.0;
    step = 1.0/(double) num_steps ;
    #pragma omp parallel firstprivate(sum) private(x, i)
    {
        int id = omp_get_thread_num();
        int numprocs = omp_get_num_threads();
        int step1 = id *num_steps/numprocs ;
        int stepN = (id+1)*num_steps/numprocs;
        if (stepN != num_steps) stepN = num_steps;
        for (i=step1; i<stepN; i++)
        {
            x = (i+0.5)*step;
            sum += 4.0/(1.0+x*x);
        }
        #pragma omp critical
        pi += sum *step ;
    }
}
```

Loop parallelism

- Collections of tasks are defined as iterations of one or more loops.
- Loop iterations are divided between a collection of processing elements to compute tasks in parallel.

```
#pragma omp parallel for shared(Results) schedule(dynamic)
for(i=0;i<N;i++){
    Do_work(i, Results);
}
```

This design pattern is heavily used with data parallel design patterns.

OpenMP programmers commonly use this pattern.



OpenMP PI Program:

Loop level parallelism pattern

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

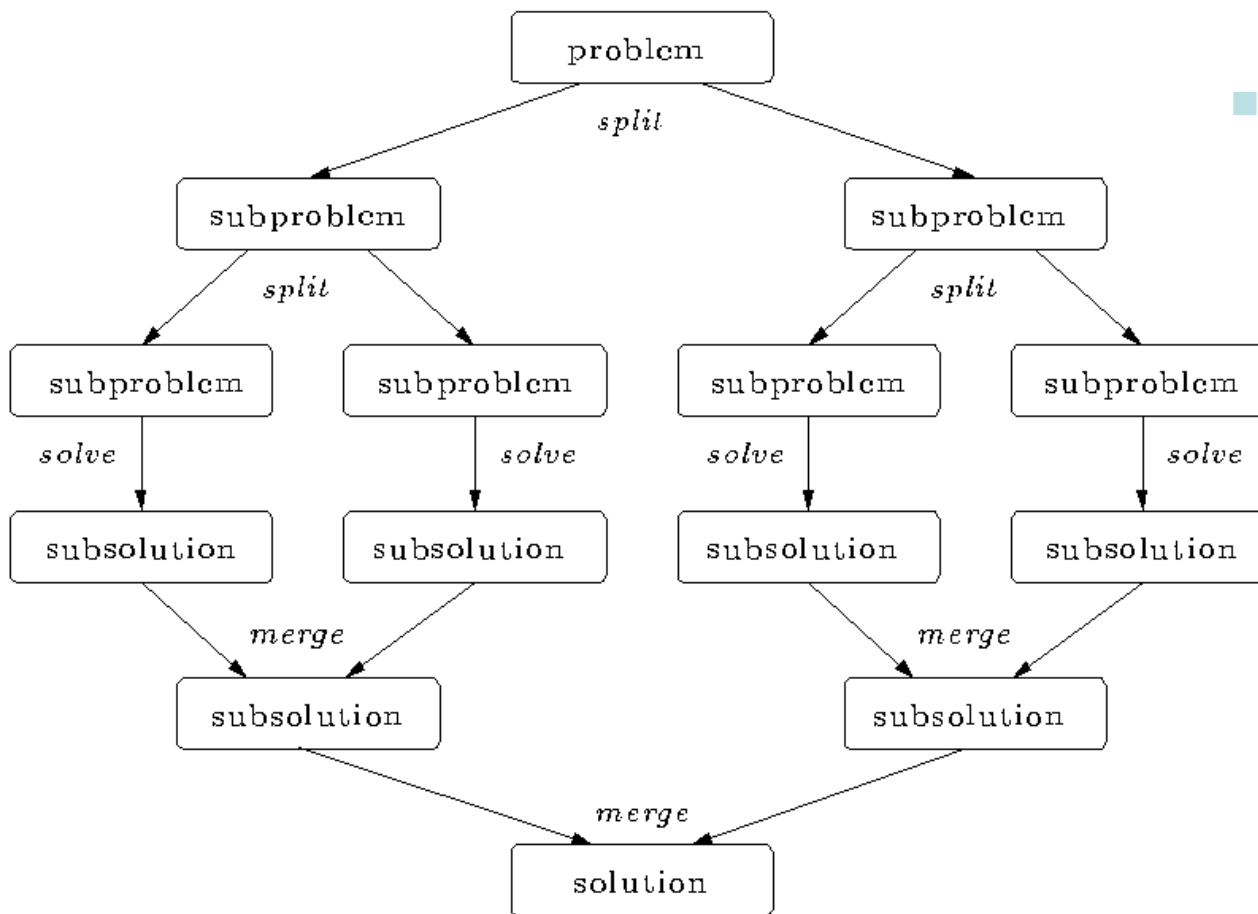
    pi = sum * step;
}
```

Divide and Conquer Pattern

- Use when:
 - A problem includes a method to divide into subproblems and a way to recombine solutions of subproblems into a global solution.
- Solution
 - Define a split operation
 - Continue to split the problem until subproblems are small enough to solve directly.
 - Recombine solutions to subproblems to solve original global problem.
- Note:
 - Computing may occur at each phase (split, leaves, recombine).

Divide and conquer

- Split the problem into smaller sub-problems. Continue until the sub-problems can be solve directly.



■ 3 Options:

- Do work as you split into sub-problems.
- Do work only at the leaves.
- Do work as you recombine.

Program: OpenMP tasks (divide and conquer pattern)

```
#include <omp.h>
static long num_steps = 1000000000;
#define MIN_BLK 10000000
double pi_comp(int Nstart,int Nfinish,double step)
{  int i,iblk;
  double x, sum = 0.0,sum1, sum2;
  if (Nfinish-Nstart < MIN_BLK){
    for (i=Nstart;i< Nfinish; i++){
      x = (i+0.5)*step;
      sum = sum + 4.0/(1.0+x*x);
    }
  }
  else{
    iblk = Nfinish-Nstart;
    #pragma omp task shared(sum1)
    sum1 = pi_comp(Nstart,      Nfinish-iblk/2,step);
    #pragma omp task shared(sum2)
    sum2 = pi_comp(Nfinish-iblk/2, Nfinish,      step);
    #pragma omp taskwait
    sum = sum1 + sum2;
  }return sum;
}
```

```
int main ()
{
  int i;
  double step, pi, sum;
  step = 1.0/(double) num_steps;
  #pragma omp parallel
  {
    #pragma omp single
    sum = pi_comp(0,num_steps,step);
  }
  pi = step * sum;
}
```

Results*: pi with tasks

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

Program: OpenMP tasks (divide and conquer pattern)				
<pre> #include <omp.h> static long num_steps = 100000000; #define MIN_BLK 10000000 double pi_comp(int Nstart,int Nfinish,double step) { int i,iblk; double x, sum = 0.0,sum1, sum2; if (Nfinish-Nstart < MIN_BLK){ for (i=Nstart;i< Nfinish; i++){ x = (i+0.5)*step; sum = sum + 4.0/(1.0+x*x); } } else{ iblk = Nfinish-Nstart; #pragma omp task shared(sum1) sum1 = pi_comp(Nstart, Nfinish-iblk); #pragma omp task shared(sum2) sum2 = pi_comp(Nfinish-iblk/2, Nfinish); #pragma omp taskwait sum = sum1 + sum2; } return sum; } int main () { int i; double step, pi, sum; step = 1.0/(double) num_steps; #pragma omp parallel </pre>				
threads	1 st SPMD	SPMD critical	PI Loop	Pi tasks
1	1.86	1.87	1.91	1.87
2	1.03	1.00	1.02	1.00
3	1.08	0.68	0.80	0.76
4	0.97	0.53	0.68	0.52

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

Learning more about OpenMP:

OpenMP Organizations

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

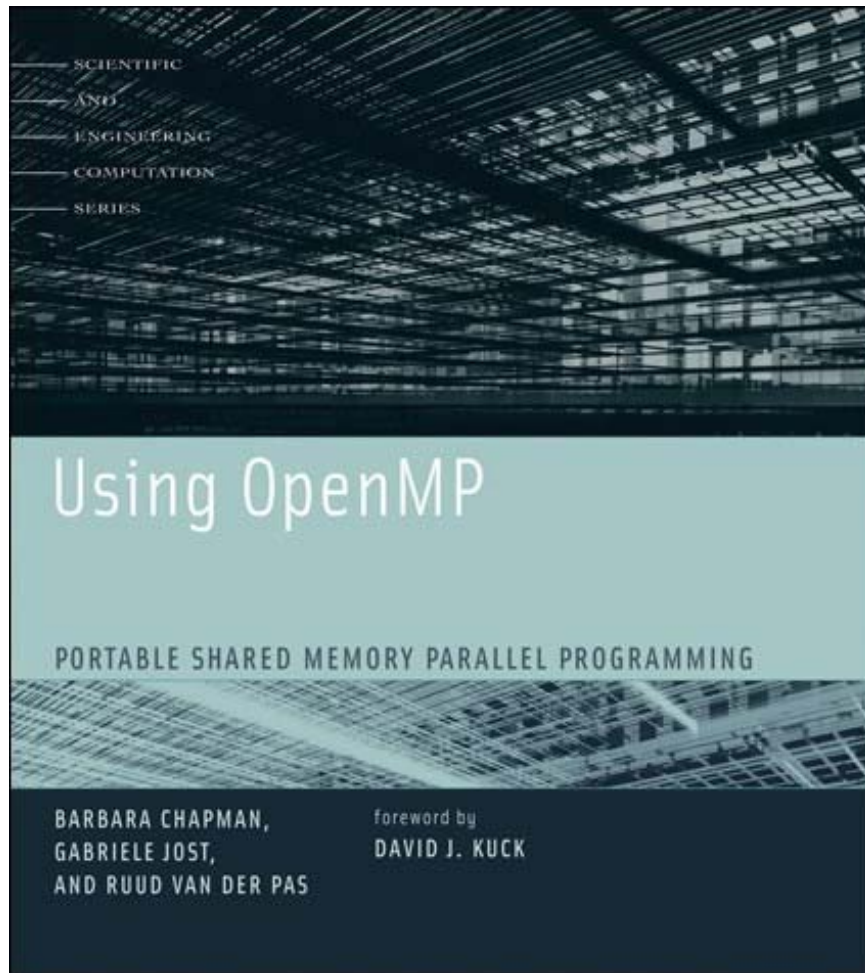
www.openmp.org

- OpenMP User’s Group (cOMPunity) URL:

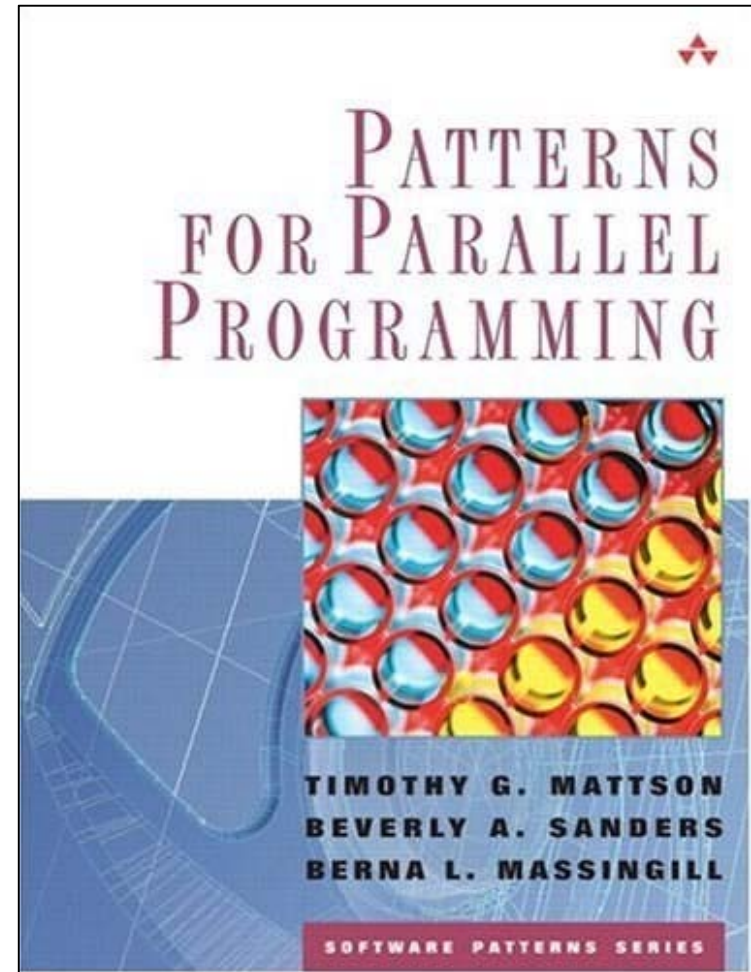
www.compunity.org

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

Books about OpenMP

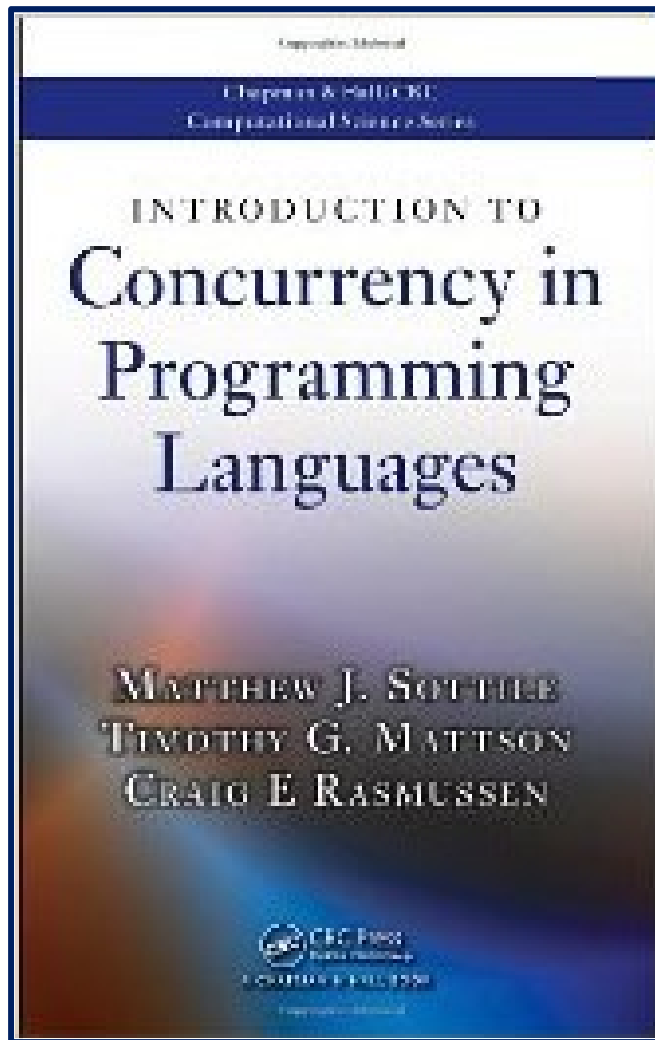


**An excellent book about using OpenMP
... though out of date (OpenMP 2.5)**

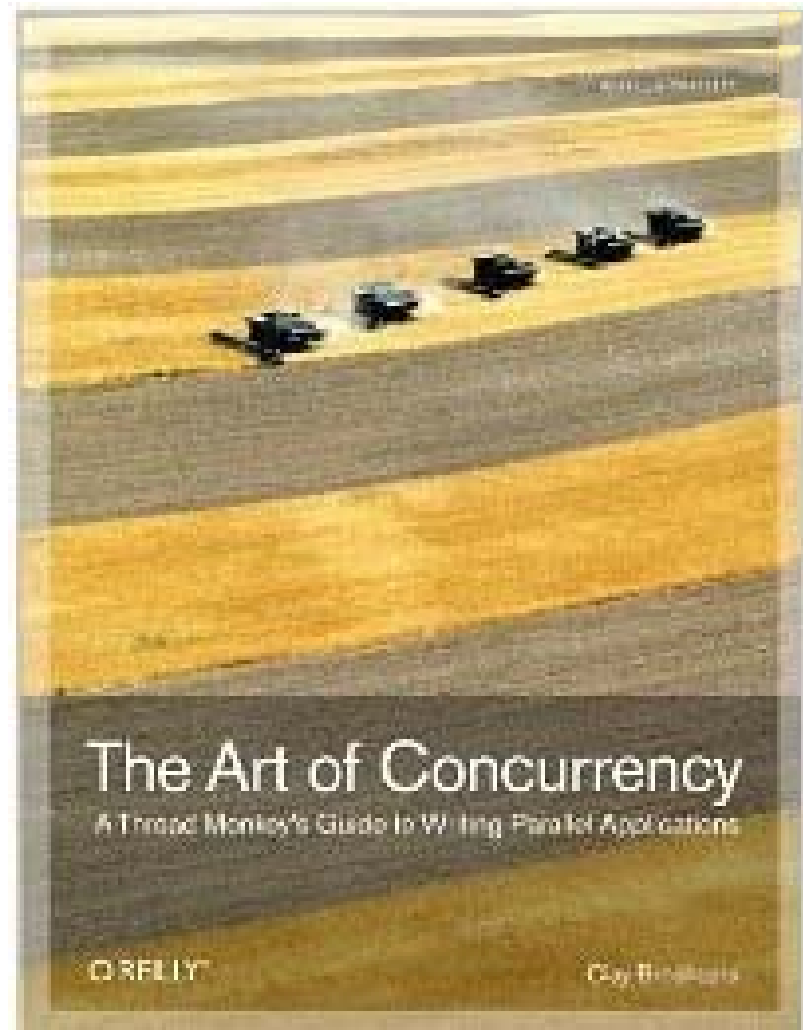


**A book about how to “think
parallel” with examples in
OpenMP, MPI and Java**

Background references



A general reference that puts languages such as OpenMP in perspective (by Sottile, Mattson, and Rasmussen)



An excellent introduction and overview of multithreaded programming (by Clay Breshears)

The OpenMP reference card

A two page summary of all the OpenMP constructs ... don't write OpenMP code without it.

OpenMP 3.1 API C/C++

Page 1

OpenMP 3.1 API C/C++ Syntax Quick Reference Card

OpenMP Application Program Interface (API) is a portable, scalable model that gives shared-memory parallel programming a simple and flexible interface for developing parallel applications for platforms ranging from the desktop to the supercomputer.

OpenMP supports multi-platform shared-memory parallel programming in C/C++ and Fortran on all architectures, including Linux platforms and Windows® platforms.

A separate OpenMP reference card for Fortran is also available.

[[Link](#)] refers to sections in the OpenMP API Specification available at www.openmp.org.

Directives

An OpenMP executable directive applies to the surrounding structured block or an OpenMP Construct. It also forms blocks in a single statement or a compound statement with a single entry at the top and a single exit at the bottom.

Parallel [2.3.4]

The parallel construct forms a team of threads and starts parallel execution.

```
Openmp omp parallel [clause] { ... }
{
  clause
  {
    #pragma expression
    new_thread(scalar-expression)
    default(clause) { none }
    private(scalar)
    private(first,last)
    shared(scalar)
    update(scalar)
    reduction(operator) (scalar)
  }
}
```

Loop [2.3.1]

The loop construct specifies that the iterations of loops will be distributed among and executed by the executing set of threads.

```
Openmp omp for [clause] { ... }
{
  clause
  {
    private(scalar)
    firstprivate(scalar)
    reduction(operator) (scalar)
    schedule(kind, chunk, size)
    collapse(n)
    ordered
    wait
  }
}
```

Thread execution form of the for loop

```
for (var = 0; var < n; var++)
  {
    // ...
  }
```

kind:

- static: the threads are divided into chunks of size chunk_size. Threads are assigned to threads in the team in a round-robin fashion in order of thread number.
- dynamic: each thread executes a chunk of iterations. This requires another chunk until iterations remain to be distributed.
- guided: each thread executes a chunk of iterations. This requires another chunk until iterations remain to be assigned. The chunk size starts large and shrinks to the smallest chunk_size as chunks are scheduled.
- auto: the decision regarding scheduling is delegated to the compiler and/or runtime system.
- autohint: The scheduler and chunk size are taken from the user-specified hint size.

Sections [2.3.2]

The sections construct contains a set of structured blocks that are to be executed among and executed by the executing team of threads.

```
Openmp omp sections [clause] { ... }
{
  {
    Openmp omp section
    structured-block
  }
  {
    Openmp omp section
    structured-block
  }
  ...
}
```

```
private(scalar)
```

```
firstprivate(scalar)
```

```
reduction(operator) (scalar)
```

```
wait
```

Single [2.3.3]

The single construct specifies that the associated structured block is executed by only one of the threads in the team (not necessarily the master thread), in the context of the implicit task.

```
Openmp omp single [clause] { ... }
{
  clause
  {
    private(scalar)
    firstprivate(scalar)
    update(scalar)
    wait
  }
}
```

Parallel Loop [2.3.1]

The parallel loop construct is a shortcut for specifying a parallel construct containing one or more associated loops and no other statements.

```
Openmp omp parallel for [clause] { ... }
{
  clause
  {
    Any clause by the parallel or the directive, except
    the wait clause, will inherit meanings and
    restrictions.
  }
}
```

Simple Parallel Loop Example

The following example demonstrates how to parallelize a simple loop using the parallel loop construct.

```
// ...
void example(n, float *a, float *b)
{
  int i;
  Openmp omp parallel for
  {
    for (i = 0; i < n; i++)
      b[i] = a[i] * a[i] * 2.0;
  }
}
```

Parallel Sections [2.3.2]

The parallel sections construct is a shortcut for specifying a parallel construct containing one or more sections and no other statements.

```
Openmp omp parallel sections [clause] { ... }
{
  {
    Openmp omp section
    structured-block
  }
  {
    Openmp omp section
    structured-block
  }
  ...
}
```

```
clause
```

Any of the clauses accepted by the parallel or sections directives, except the wait clause, will inherit meanings and restrictions.

Task [2.3.1]

The task construct defines an explicit task. The data environment of the task is created according to the task-creating attribute clauses on the task construct and any defaults that apply.

```
Openmp omp task [clause] { ... }
{
  clause
  {
    #pragma expression
    new_thread(scalar-expression)
    wait
    default(clause) { none }
    private(scalar)
    private(first,last)
    shared(scalar)
    update(scalar)
    wait
  }
}
```

Taskwait [2.3.2]

The taskwait construct specifies that the current task can be suspended in case of execution of a different task.

```
Openmp omp taskwait
```

Master [2.3.1]

The master construct specifies a structured block that is executed by the master thread of the team. There is no implicit barrier after execution by the master, so the master can exit.

```
Openmp omp master
structured-block
```

Critical [2.3.3]

The critical construct restricts execution of the associated structured block to a single thread at a time.

```
Openmp omp critical [clause]
structured-block
```

Barrier [2.3.3]

The barrier construct specifies a completion barrier at the point at which the current task appears.

```
Openmp omp barrier
```

Taskwait [2.3.2]

The taskwait construct specifies a wait at the completion of other tasks in the current task.

```
Openmp omp taskwait
```

Atomic [2.3.3]

The atomic construct ensures that a specific storage location is updated atomically, either by ensuring it is the possibility of multiple, simultaneous writes to the location.

```
Openmp omp atomic (read | write | update | capture)
expression; do;
```

```
Openmp omp atomic capture
structured-block
```

where an expression that may be one of the following for read, write, update, or capture:

if clause is:	expression is:
none	var
capture	var = expr
update	var = var + expr
read	var = var
write	var = expr
update	var = var + expr
capture	var = var

and also block that may be one of the following forms:

```
var = expr;
var = var + expr;
var = var * expr;
var = var / expr;
var = var & expr;
var = var | expr;
var = var ^ expr;
var = var << expr;
var = var >> expr;
```

OpenMP 3.1 C/C++-H

Page 2

Runtime Library Routines

Execution Environment Routines [3.1.2]

These routines are environment routines that affect and monitor threads, processors, and the parallel environment.

`omp_get_num_threads()`

Get `num_threads`.
Affects the number of threads used for subsequent parallel regions that do not specify a new `num_threads` clause.

`omp_get_num_threads(0)`

Returns the number of threads in the current team.

`omp_get_max_threads()`

Returns maximum number of threads that could be used to execute a new team, using a parallel construct without a new `num_threads` clause.

`omp_get_max_threads(0)`

Returns the O of the executing thread where O ranges from zero to the size of the team minus 1.

`omp_get_num_procs()`

Returns the number of processors available to the program.

Open Types For Runtime Library Routines

`omp_lock_t` represents a simple lock.

`omp_mutex_t` represents a mutex lock.

`omp_sched_t` represents a schedule.

`omp_in_parallel()`

Returns true if the call to the routine is executed by an active parallel region; otherwise, it returns false.

`omp_get_dyn_size()`

Enables or disables dynamic adjustment of the number of threads available by setting the value of the `dyn_size` flag.

`omp_get_dyn_size(0)`

Returns the value of the `dyn_size` flag, indicating whether dynamic adjustment of the number of threads is enabled or disabled.

`omp_get_nested()`

Enables or disables nested parallelism by setting the `nested` flag.

`omp_get_nested(0)`

Returns the value of the `nested` flag, which indicates if nested parallelism is enabled or disabled.

`omp_get_schedule_kind(0, 1, 2, 3, 4, 5)`

Affects the schedule that is applied when `schedule` is used in a schedule kind, by setting the value of the `omp_schedule_kind` flag.

Kind is one of static, dynamic, guided, auto, or an implementation-defined schedule. See `omp_schedule` [3.1.2.3] for more details.

`omp_get_schedule(omp_sched_t *result)`

Returns the value of `omp_schedule_kind`, which is the schedule applied when `schedule` is used.

See `omp_sched` for more details.

`omp_get_wtime()`

Returns the value of the wall-clock time in seconds.

`omp_get_wtick()`

Returns the value of the wall-clock time in seconds.

`omp_get_wtick(0)`

Returns the value of the wall-clock time in seconds.

`omp_get_wtick(0, 1, 2, 3, 4, 5)`

Returns the value of the wall-clock time in seconds.

`omp_get_wtick(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)`

Returns the value of the wall-clock time in seconds.

`omp_get_wtick(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10)`

Returns the value of the wall-clock time in seconds.

`omp_get_wtick(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11)`

Returns the value of the wall-clock time in seconds.

`omp_get_wtick(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12)`

Returns the value of the wall-clock time in seconds.

`omp_get_wtick(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13)`

Returns the value of the wall-clock time in seconds.

`omp_get_wtick(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14)`

Returns the value of the wall-clock time in seconds.

`omp_get_wtick(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15)`

Returns the value of the wall-clock time in seconds.

`omp_get_wtick(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16)`

Returns the value of the wall-clock time in seconds.

`omp_get_wtick(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17)`

Returns the value of the wall-clock time in seconds.

`omp_get_wtick(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18)`

Returns the value of the wall-clock time in seconds.

`omp_get_wtick(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19)`

Returns the value of the wall-clock time in seconds.

`omp_get_wtick(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20)`

Returns the value of the wall-clock time in seconds.

`omp_get_wtick(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15`

<http://openmp.org/mp-documents/OpenMP3.1-CCard.pdf>