

An Introduction to OpenMP

Clayton Chandler, Louisiana Tech University

cfc004@latech.edu

Himanshu Chhetri, Louisiana Tech University

hch018@latech.edu

Information from “OpenMP Tutorial” from LLNL

And

A “Hands-on” Introduction to OpenMP from Super Computing
Conference 2009*

What Is OpenMP?

- **OpenMP Is:**
- **An Application Program Interface (API) that may be used to explicitly direct multi-threaded, shared memory parallelism**
- **Comprised of three primary API components:**
 - Compiler Directives
 - Runtime Library Routines
 - Environment Variables
- **Portable:**
 - The API is specified for C/C++ and Fortran
 - Multiple platforms have been implemented including most Unix platforms and Windows NT
- **Standardized:**
 - Jointly defined and endorsed by a group of major computer hardware and software vendors
 - Expected to become an ANSI standard later (pending)

What is OpenMP?

- **What does OpenMP stand for?**

- Open specifications for Multi Processing via collaborative work between interested parties from the hardware and software industry, government and academia.

- **OpenMP Is Not:**

- Meant for distributed memory parallel systems (by itself)
- Necessarily implemented identically by all vendors
- Guaranteed to make the most efficient use of shared memory (currently there are no data locality constructs)

OpenMP History

- **Ancient History**
- **In the early 90's, vendors of shared-memory machines supplied similar, directive-based, Fortran programming extensions:**
 - The user would augment a serial Fortran program with directives specifying which loops were to be parallelized
 - The compiler would be responsible for automatically parallelizing such loops across the SMP processors
- **Implementations were all functionally similar, but were diverging (as usual)**
- **First attempt at a standard was the draft for ANSI X3H5 in 1994. It was never adopted, largely due to waning interest as distributed memory machines became popular.**

OpenMP History

- **More Recent History**
- **The OpenMP standard specification started in the spring of 1997, taking over where ANSI X3H5 had left off, as newer shared memory machine architectures started to become prevalent.**
- **Partners in the OpenMP standard specification included:**
(Disclaimer: all partner names derived from the [OpenMP web site](#))
 - Compaq / Digital
 - Hewlett-Packard Company
 - Intel Corporation
 - Sun Microsystems, Inc.
 - U.S. Department of Energy ASC program

OpenMP History

■ Documentation Release History

- Oct 1997: Fortran version 1.0
- Oct 1998: C/C++ version 1.0
- Nov 2000: Fortran version 2.0
- Mar 2002: C/C++ version 2.0
- May 2005: C/C++ and Fortran version 2.5
- ??? : version 3.0

Goals of OpenMP

- **Standardization:**

- Provide a standard among a variety of shared memory architectures/platforms

- **Lean and Mean:**

- Establish a simple and limited set of directives for programming shared memory machines. Significant parallelism can be implemented by using just 3 or 4 directives.

- **Ease of Use:**

- Provide capability to incrementally parallelize a serial program, unlike message-passing libraries which typically require an all or nothing approach
- Provide the capability to implement both coarse-grain and fine-grain parallelism

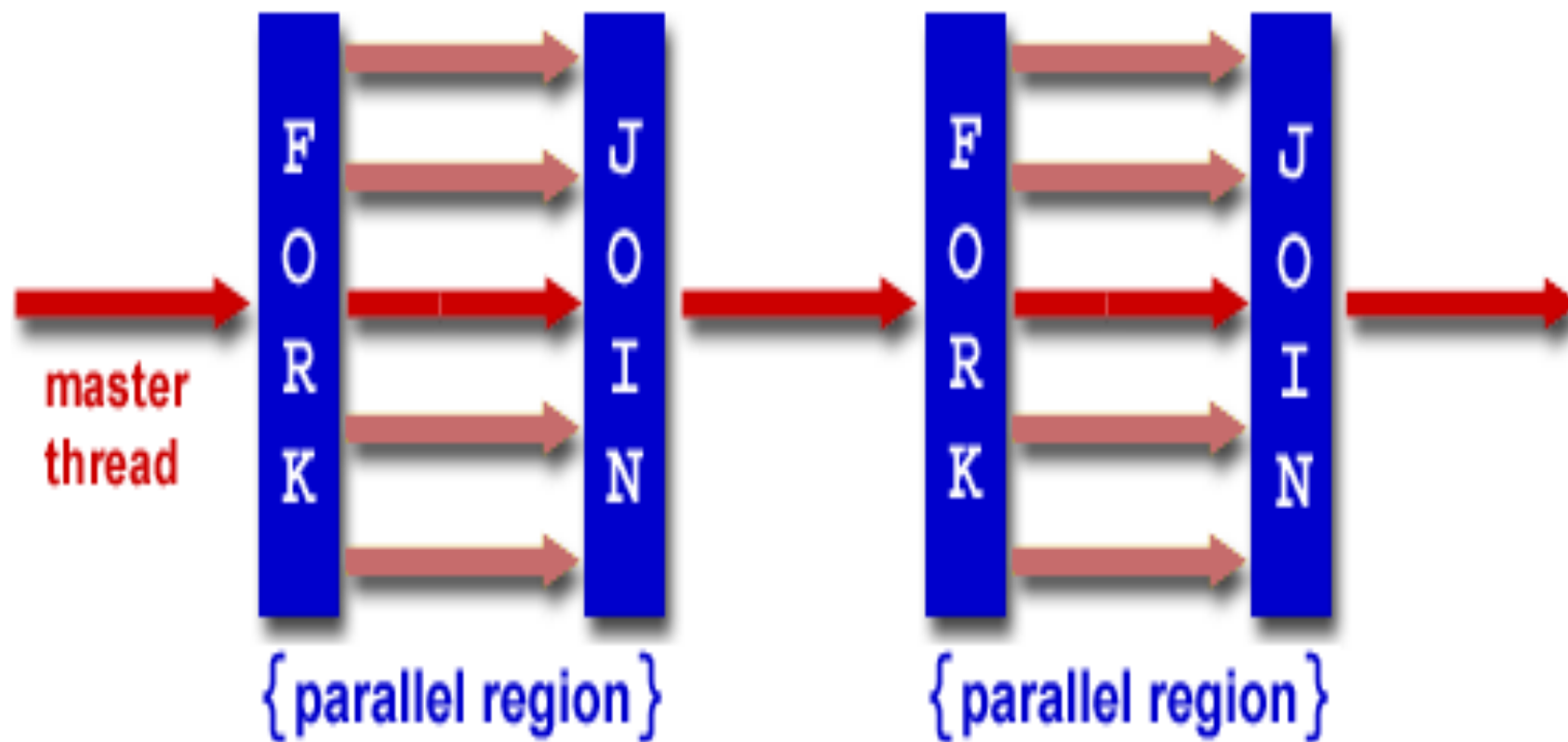
- **Portability:**

- Supports Fortran (77, 90, and 95), C, and C++
- Public forum for API and membership

OpenMP Programming Model

- **Shared Memory, Thread Based Parallelism:**
 - OpenMP is based upon the existence of multiple threads in the shared memory programming paradigm. A shared memory process consists of multiple threads.
- **Explicit Parallelism:**
 - OpenMP is an explicit (not automatic) programming model, offering the programmer full control over parallelization.
- **Fork - Join Model:**
 - OpenMP uses the **fork-join** model of parallel execution:
 - All OpenMP programs begin as a single process: the master thread. The master thread executes sequentially until the first parallel region construct is encountered.
 - **FORK:** the master thread then creates a *team* of parallel threads
 - The statements in the program that are enclosed by the parallel region construct are then executed in parallel among the various team threads
 - **JOIN:** When the team threads complete the statements in the parallel region construct, they synchronize and terminate, leaving only the master thread

Fork – Join Model



OpenMP Programming Model

- **Compiler Directive Based:**

- Most OpenMP parallelism is specified through the use of compiler directives which are imbedded in C/C++ or Fortran source code.

- **Nested Parallelism Support:**

- The API provides for the placement of parallel constructs inside of other parallel constructs.
- Implementations may or may not support this feature.

- **Dynamic Threads:**

- The API provides for dynamically altering the number of threads which may used to execute different parallel regions.
- Implementations may or may not support this feature.

OpenMP Programming Model

■ I/O:

- OpenMP specifies **nothing** about parallel I/O. This is particularly important if multiple threads attempt to write/read from the same file.
- If every thread conducts I/O to a different file, the issues are not as significant.
- It is **entirely up to the programmer** to insure that I/O is conducted correctly within the context of a multi-threaded program.

■ FLUSH Often?:

- OpenMP provides a "relaxed-consistency" and "temporary" view of thread memory (in their words). In other words, threads can "cache" their data and are not required to maintain exact consistency with real memory all of the time.
- When it is critical that all threads view a shared variable identically, the programmer is responsible for insuring that the variable is FLUSHed by all threads as needed.
- More on this later...

OpenMP Directives

- **Uses commenting of application source**
 - `#pragma omp [directive] [clause(s)] [newline]`
- **pragma Omp**
 - Always. No, I have no clue what it means.
- **Directive**
 - A valid OpenMP directive. Must appear after the pragma and before any clauses.
- **Clause(s)**
 - Optional. Clauses can be in any order, and repeated as necessary unless otherwise restricted.
- **Newline**
 - Required. Precedes the structured block which is enclosed by this directive.
- **Example**
 - `#pragma omp parallel default(shared) private(beta,pi)`

OpenMP Programming Model

■ General Rules:

- Case sensitive (because this is C. FORTRAN – not sensitive)
- Directives follow conventions of the C/C++ standards for compiler directives
- Only one directive-name may be specified per directive
- Each directive applies to at most one succeeding statement, which must be a structured block.
- Long directive lines can be "continued" on succeeding lines by escaping the newline character with a backslash ("\") at the end of a directive line.

OpenMP Directives (PARALLEL Directive)

■ Purpose:

- A parallel region is a block of code that will be executed by multiple threads. This is the fundamental OpenMP parallel construct.

■ Format:

- `#pragma omp parallel [clause ...] newline`
- Some clauses:
 - `if (scalar_expression)`
 - `private (list)`
 - `shared (list)`
 - `default (shared | none)`
 - `firstprivate (list)`
 - `reduction (operator: list)`
 - `copyin (list)`
 - `num_threads (integer-expression)`
- Then your structured block of code

OpenMP Directives (PARALLEL Directive)

■ Notes:

- When a thread reaches a PARALLEL directive, it creates a **team** of threads and becomes the master of the team. The master is a member of that team and has thread number 0 within that team. *(Remember when I mentioned thread pools on Monday? This is that concept in action)*
- Starting from the beginning of this parallel region, the code is duplicated and all threads will execute that code.
- There is an **implied** barrier at the end of a parallel section. Only the master thread continues execution past this point. (Note that FORTRAN has an explicit barrier)
- If any thread terminates within a parallel region, all threads in the team will terminate, and the work done up until that point is undefined.

OpenMP Directives (PARALLEL Directive)

■ How Many Threads?

- The number of threads in a parallel region is determined by the following factors, in order of precedence:
 - Evaluation of the IF clause
 - Setting of the NUM_THREADS clause
 - Use of the omp_set_num_threads() library function
 - Setting of the OMP_NUM_THREADS environment variable
 - Implementation default - usually the number of CPUs on a node, though it could be dynamic (see next bullet).
- Threads are numbered from 0 (master thread) to N-1

OpenMP Directives (PARALLEL Directive)

■ Dynamic Threads:

- Use the `omp_get_dynamic()` library function to determine if dynamic threads are enabled.
- If supported, the two methods available for enabling dynamic threads are:
 - The `omp_set_dynamic()` library routine
 - Setting of the `OMP_DYNAMIC` environment variable to `TRUE`

■ Nested Parallel Regions:

- Use the `omp_get_nested()` library function to determine if nested parallel regions are enabled.
- The two methods available for enabling nested parallel regions (if supported) are:
 - The `omp_set_nested()` library routine
 - Setting of the `OMP_NESTED` environment variable to `TRUE`
- If not supported, a parallel region nested within another parallel region results in the creation of a new team, consisting of one thread, by default.

OpenMP Directives (PARALLEL Directive)

■ Clauses:

- IF clause: If present, it must evaluate to .TRUE. (Fortran) or non-zero (C/C++) in order for a team of threads to be created. Otherwise, the region is executed serially by the master thread. (recall: this is evaluated first)
- The remaining clauses are described in the tutorial, in the [Data Scope Attribute Clauses](#) section. We may cover some if time permits.

■ Restrictions:

- A parallel region **must be a structured block** that does not span multiple routines or code files
- It is illegal to branch into or out of a parallel region
- Only a single IF clause is permitted
- Only a single NUM_THREADS clause is permitted

Structure

```
#include <omp.h>

main () { int var1, var2, var3;

Serial code

. . .

Beginning of parallel section.

Fork a team of threads. Specify variable scoping

#pragma omp parallel private(var1, var2) shared(var3)
{

Parallel section executed by all threads

. . .

All threads join master thread and disband }

Resume serial code . . .

}
```

Compilation

icc -openmp filename.c -o output

Hello World

- Simple "Hello World" program
 - Every thread executes all code enclosed in the parallel section
 - OpenMP library routines are used to obtain thread identifiers and total number of threads

http://hpci.latech.edu/openmp/hello_openmp.c.html

http://hpci.latech.edu/openmp/hello_openmp.c

stop



UNCLASSIFIED

Operated by Los Alamos National Security, LLC for the U.S. Department of Energy's NNSA



How do threads interact?

OpenMP is a multi-threading, shared address model

- Threads communicate by sharing variables.

Unintended sharing of data causes race conditions:

- race condition: when the program's outcome changes as the threads are scheduled differently.

To control race conditions:

- Use synchronization to protect data conflicts.

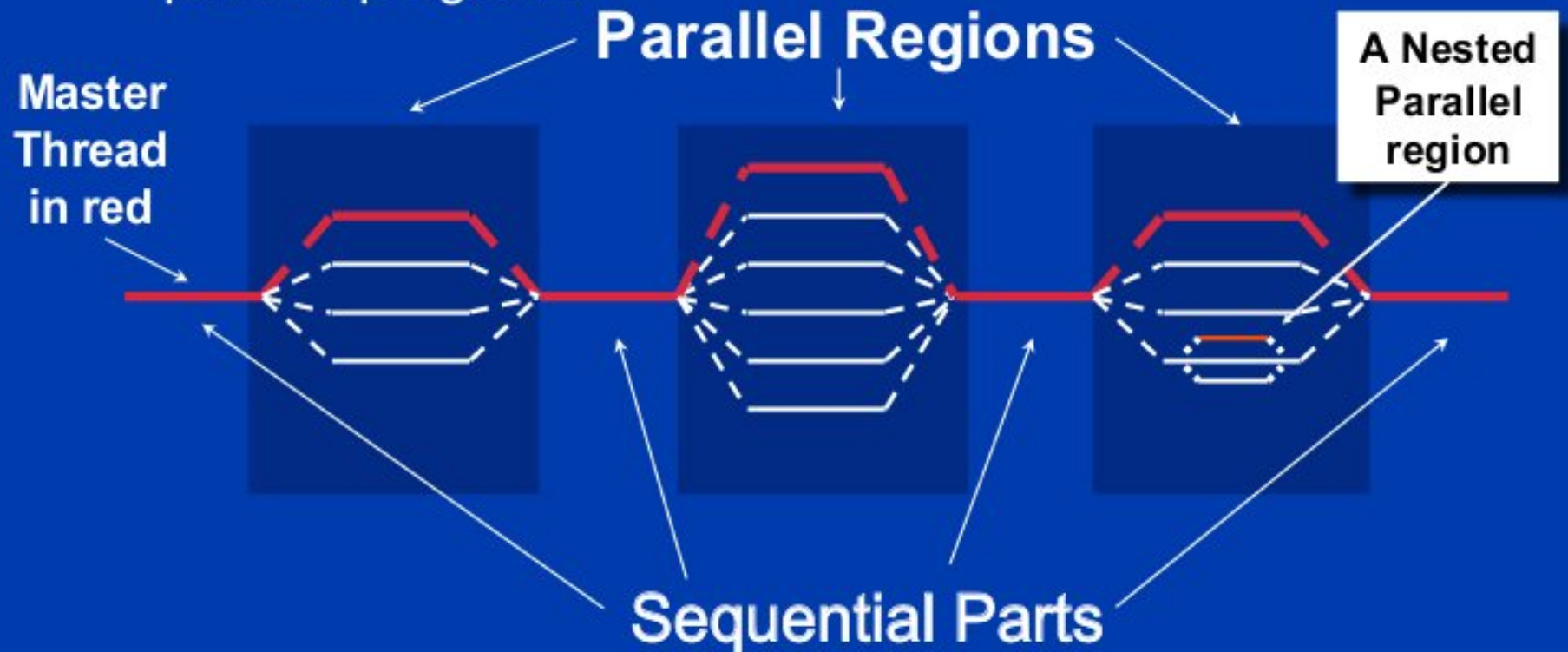
Synchronization is expensive so:

- Change how data is accessed to minimize the need for synchronization.

OpenMP Programming Model:

Fork-Join Parallelism:

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



Thread Creation: Parallel Regions

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

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

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

Runtime function to request a certain number of threads

Runtime function returning a thread ID

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

Thread Creation: Parallel Regions

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

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

```
double A[1000];
```

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

clause to request a certain number of threads

Runtime function returning a thread ID

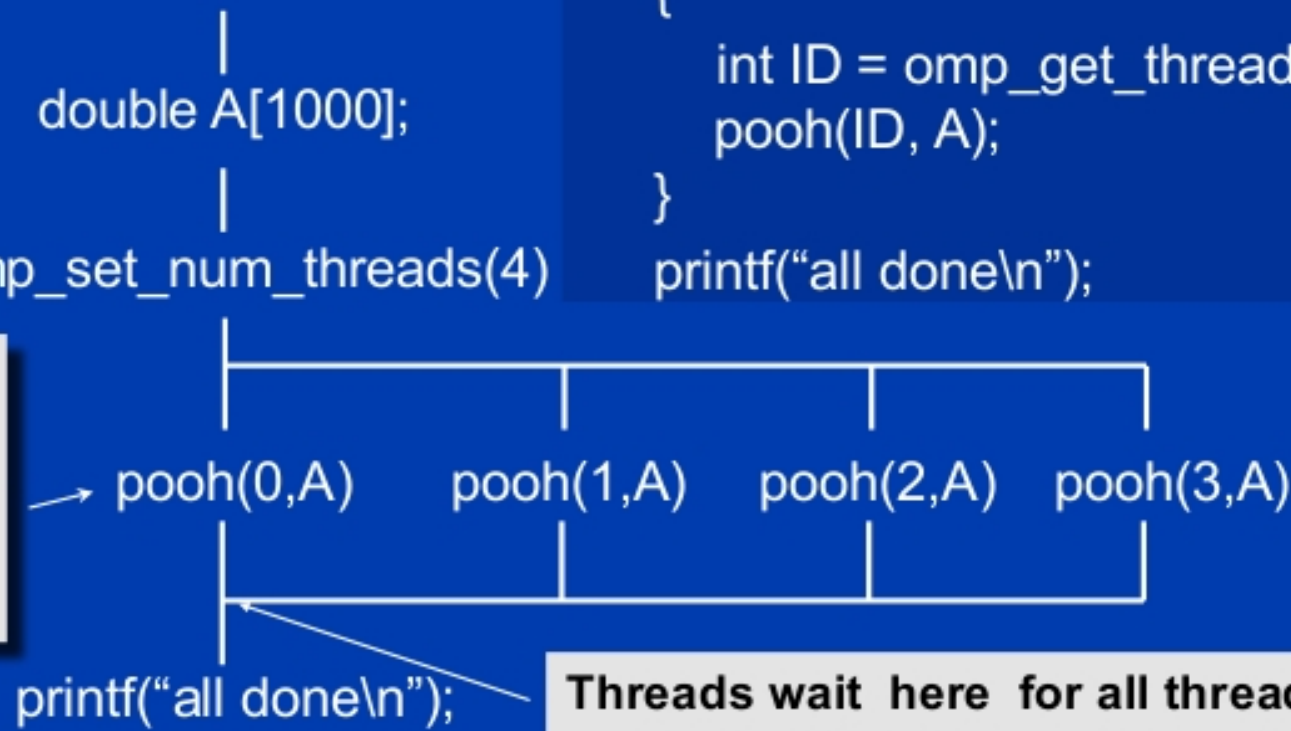
- Each thread calls **pooh(ID,A)** for **ID = 0 to 3**

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*)

Exercise 1, Part A: Hello world

Verify that your environment works

- Write a program that prints “hello world”.

```
void 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

-fopenmp gcc

-mp pgi

/Qopenmp intel

Exercise 1: Solution

A multi-threaded “Hello world” program

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

```
#include "omp.h" ← OpenMP include file
```

```
void main()  
{
```

Parallel region with default number of threads

```
#pragma omp parallel  
{
```

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

```
    }  
}
```

End of the Parallel region

Runtime library function to return a thread ID.

Sample Output:

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

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



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:


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();  
}
```

These are equivalent



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];}
```

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

Synchronization: Barrier

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

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

implicit barrier at the end of a
for worksharing construct

implicit barrier at the end
of a parallel region

no implicit barrier
due to nowait

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();
}
```


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.

Single worksharing Construct

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

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

Synchronization: ordered

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

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

Synchronization: Lock routines

- **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.

Runtime Library routines

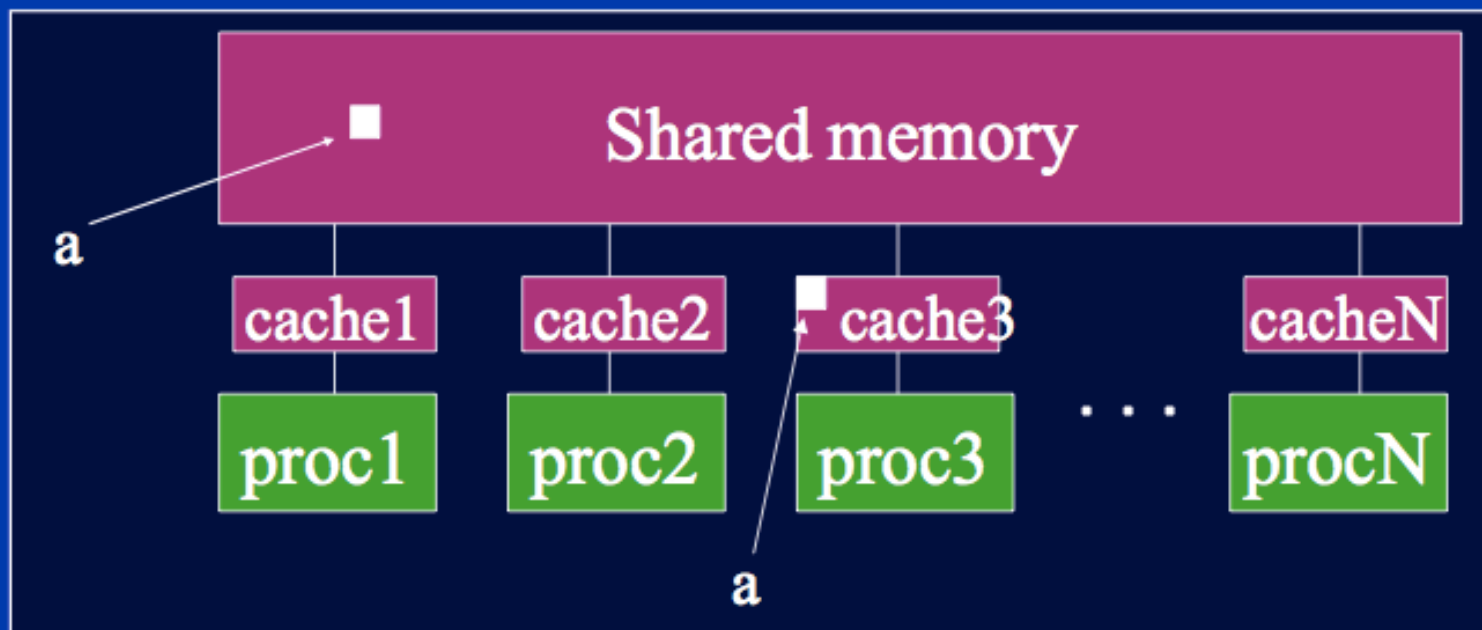
- **Runtime environment routines:**

- **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.

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.

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.

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: 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.



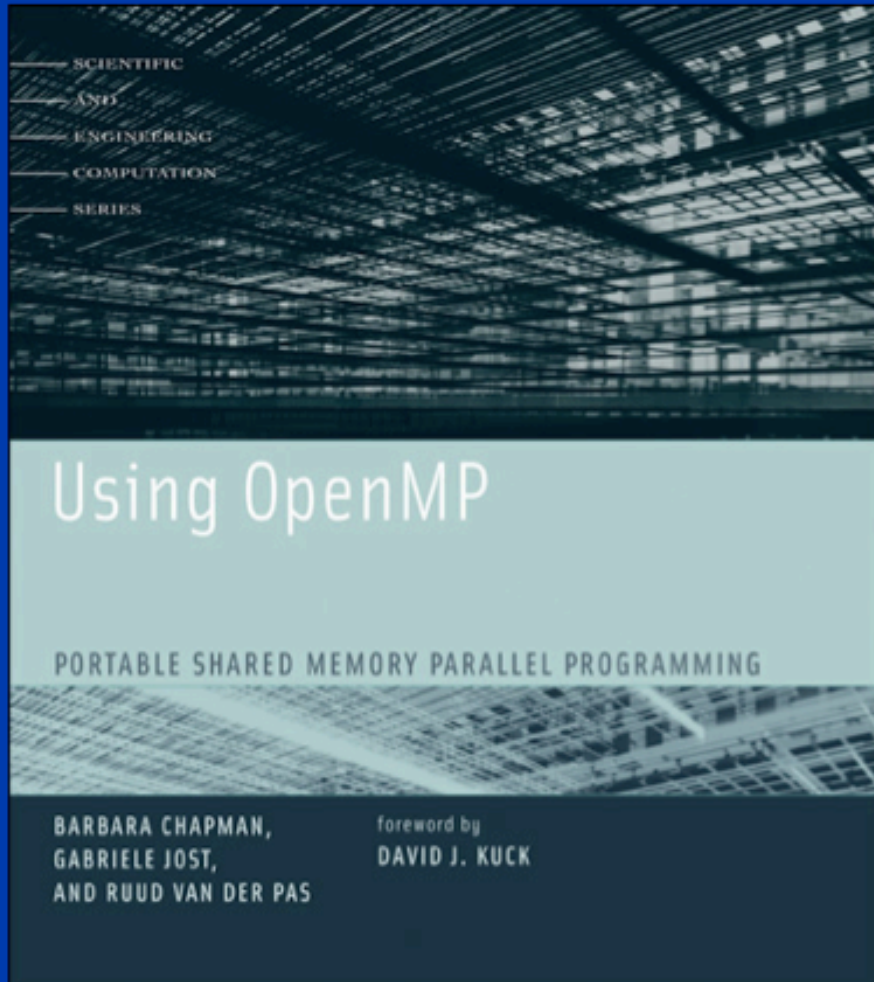
Only one thread should copy the number of threads to the global value to make sure multiple threads writing to the same address don't conflict.



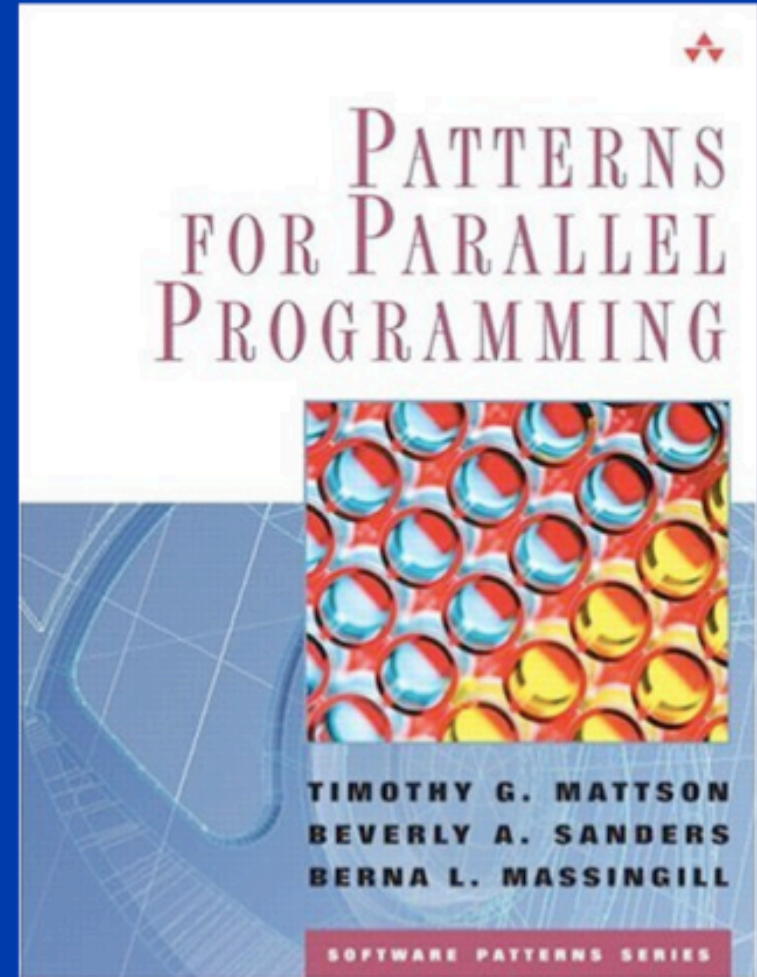
This is a common trick in SPMD programs to create a cyclic distribution of loop iterations



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