

Chapter 3 OpenMP (Parallel Programming)

Wuu Yang

National Chiao-Tung University, Taiwan, R.O.C.

first draft: March 11, 2011

current version: September 2, 2011

©September 2, 2011 by Wuu Yang. All rights reserved.

Table of Contents

- **§1 References**
- **§2 Introduction to OpenMP**
- **§3 OpenMP directives, clauses, library routines, and environment variables**
 - **§3.1 the parallel directive**
 - **§3.2 Worksharing constructs - do/for, section, single**
 - * **§3.2.1 The do/for directive**
 - * **§3.2.2 The sections directive**
 - * **§3.2.3 The workshare directive**
 - * **§3.2.4 The single directive**
 - **§3.3 the task directive**
 - **§3.4 Synchronization constructs**
 - * **§3.4.1 The master directive**

- * §3.4.2 The critical directive
- * §3.4.3 The barrier directive
- * §3.4.4 The taskwait directive
- * §3.4.5 The atomic directive
- * §3.4.6 The flush directive
- * §3.4.7 The ordered directive
- * §3.4.8 The threadprivate directive
- §3.5 Data scope (or data-sharing) attribute clauses
 - * §3.5.1 The private clause
 - * §3.5.2 The shared clause
 - * §3.5.3 The default clause
 - * §3.5.4 The firstprivate clause
 - * §3.5.5 The lastprivate clause
 - * §3.5.6 The copyin clause
 - * §3.5.7 The copyprivate clause
 - * §3.5.8 The reduction clause

- * **§3.5.9** Clauses and directives summary
- **§3.6** Directive binding and nesting rules
- **§3.7** Run-time library routines
 - * **§3.7.1** OPM_SET_NUM_THREADS
 - * **§3.7.2** OMP_GET_NUM_THREADS
 - * **§3.7.3** OMP_GET_MAX_THREADS
 - * **§3.7.4** OMP_GET_THREAD_NUM
 - * **§3.7.5** OMP_GET_THREAD_LIMIT
 - * **§3.7.6** OMP_GET_NUM_PROCS
 - * **§3.7.7** OMP_IN_PARALLEL
 - * **§3.7.8** OMP_SET_DYNAMIC
 - * **§3.7.9** OMP_GET_DYNAMIC
 - * **§3.7.10** OMP_SET_NESTED
 - * **§3.7.11** OMP_GET_NESTED
 - * **§3.7.12** New routines for OpenMP 3.0
 - * **§3.7.13** OMP_INIT_LOCK

- * **§3.7.14** OMP_DESTROY_LOCK
- * **§3.7.15** OMP_SET_LOCK
- * **§3.7.16** OMP_UNSET_LOCK
- * **§3.7.17** OMP_TEST_LOCK
- * **§3.7.18** OMP_GET_WTIME
- * **§3.7.19** OMP_GET_WTICK
- **§3.8** Environment variables
- **§3.9** Memory and performance issues

§1 References

- OpenMP, Blaise Barney, Livermore Computing, Lawrence Livermore National Laboratory. See the file “OpenMP20101130.pdf”.
- An Introduction Into OpenMP, Ruud van der Pas, Senior Staff Engineer, Scalable Systems Group, Sun Microsystems, IWOMP 2005, University of Oregon Eugene, Oregon, USA. June 1-4, 2005. See the file “iwomp2005_tutorial_openmp_rvdp.pdf”.
- COMP 322: Principles of Parallel Programming Lectures 15 and 16: OpenMP, contd. Fall 2009
<http://www.cs.rice.edu/~vsarkar/comp322>, Vivek Sarkar, Department of Computer Science Rice University
vsarkar@rice.edu
- Lin and Snyder, Principles of Parallel Programming, 2009, Prentice Hall.

§2 Introduction to OpenMP

OpenMP stands for Open Specification for MultiProcessing.

OpenMP programming model:

1. shared memory, thread-based parallelism
2. explicit parallelism (not automatic)
3. fork-join model

OpenMP design objective is to be portable, scalable, and standardized. Its API supports C/C++ and Fortran on both Unix and Windows NT platforms.

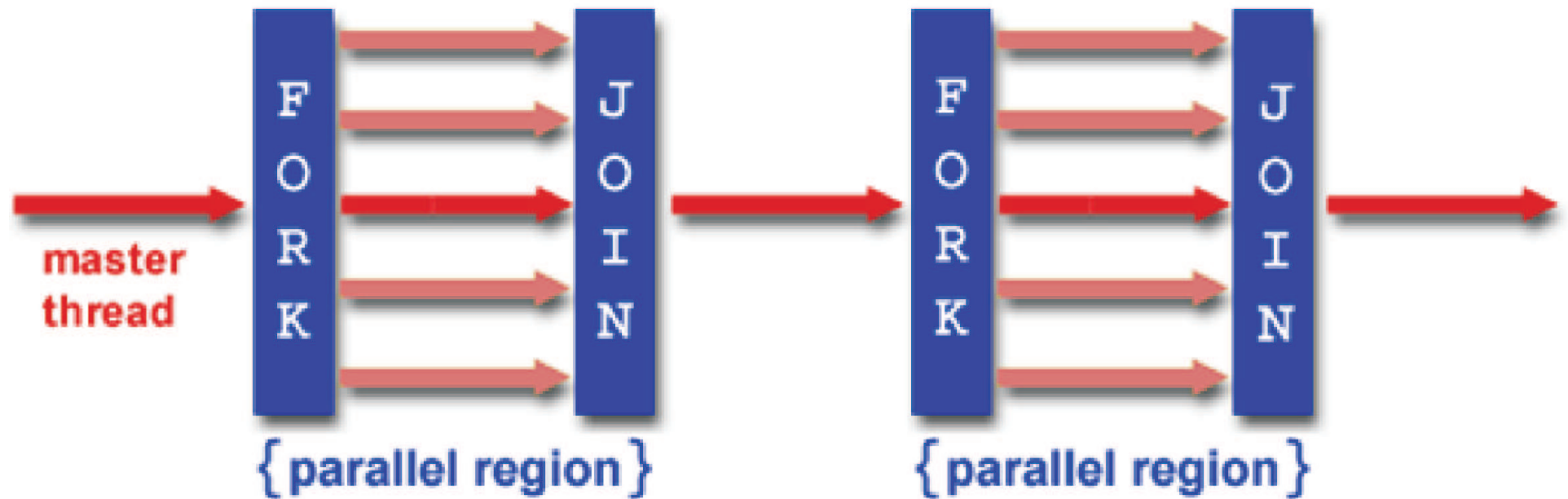
It consists three parts:

1. compiler directives: As a compiler is not powerful enough to discover all parallelism in a program, the programmer can add *hints* to guide the compiler. The directives are embedded in C/C++ and Fortran source code.
2. runtime library: Frequently used routines are collected in a standard library.
3. environment variables

Limitations of OpenMP:

1. OpenMP is not meant for distributed-memory parallel systems.
2. OpenMP is not necessarily implemented by all vendors.
3. OpenMP does not guarantee to make the most efficient use of shared memory.
4. OpenMP is not required to check for data dependencies, data conflicts, race conditions and deadlocks. These are the programmer's responsibilities.
5. In OpenMP, the programmer is responsible for synchronizing input and output among the parallel threads.
6. A programmer must explicitly (i.e., not automatic) express the fork/join/synchronization statements.

OpenMP adopts a *fork-and-join* model of parallel execution:



There is a single thread (*master*) in the beginning. New child threads are forked on parallel regions. Upon the completion of a parallel region, the child threads synchronize (*join*). Then only the master thread continues.

Example.

```
9   printf("%3s\t%10s\n", "#", "Output");
10 #pragma omp parallel
11   {
12       printf("%3d\t%10d\n", omp_get_thread_num(), rand());
13   }
```

#	Output
0	1164206856
2	2121261946
1	683113484
3	223575960

OpenMP supports nested parallelism. A parallel construct may be placed inside another parallel constructs.

Threads are created dynamically.

OpenMP does not enforce consistency of thread memory. A thread may make local copies of data and are not required to maintain exact consistency with real memory all of the time. In other words, it is the programmer's responsibility to maintain memory consistency if so required by the application.

Overall structure of C/OpenMP code:

```
#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 . . .
```

§3 OpenMP directives, clauses, library routines, and environment variables

The format of an OpenMP directive:

```
#pragma omp directive-name [ clause, ... ]
```

Note that the directives are case-sensitive. Each directive applies to one succeeding statement, which must be a structured block.

```
#pragma omp parallel default(shared) private(beta, pi)
```

§3.1 the parallel directive

The `parallel` directive defines a *parallel region*, which will be executed by multiple threads:

<code>\$pragma omp parallel</code>	<code>if (scalar expression)</code> <code>private (list)</code> <code>shared (list)</code> <code>default (shared — none)</code> <code>firstprivate (list)</code> <code>reduction (operator: list)</code> <code>copyin (list)</code> <code>num_threads (integer expression)</code>
------------------------------------	--

When a thread reaches a *parallel region*, it creates a team of threads and it becomes the master of the team (with thread number 0). All threads will execute the code in the parallel region

simultaneously. There is an implied *barrier* at the end of a parallel region. Only the master thread will continue past the barrier. If any thread terminates within a parallel region, all threads in the team will terminate and the work done is undefined.

The number of threads created is determined from (1) the `if` clause; (2) the `NUM_THREADS` clause; (3) the `omp_set_num_threads` library function; (4) the `OMP_NUM_THREADS` environment variable; (5) implementation default, which is usually the number of CPUs on a node. Threads are numbered 0, 1, 2, ...

The `omp_set_dynamic` library routine or the `OMP_DYNAMIC` environment variable enable dynamic threads. The `omp_get_dynamic` library function determines if dynamic threads are enabled.

The `omp_set_nested` library routine and the `OMP_NESTED` environment variable enables nested parallel region. The `omp_get_nested` library function determines if nested parallel regions are enabled.

If the `if` clause evaluates to false or 0, the parallel region is executed serially by the master thread only.

Example.

```
#include <omp.h>
main () { int nthreads, tid;
/* Fork a team of threads with each thread having a private tid.*/
#pragma omp parallel private(tid) {
    /* Obtain and print thread id */
    tid = omp_get_thread_num();
    printf("Hello World from thread = %d\n", tid);

    /* Only master thread does this */
    if (tid == 0) {
        nthreads = omp_get_num_threads();
        printf("Number of threads = %d\n", nthreads);
    } } /* All threads join master thread and terminate */
}
```

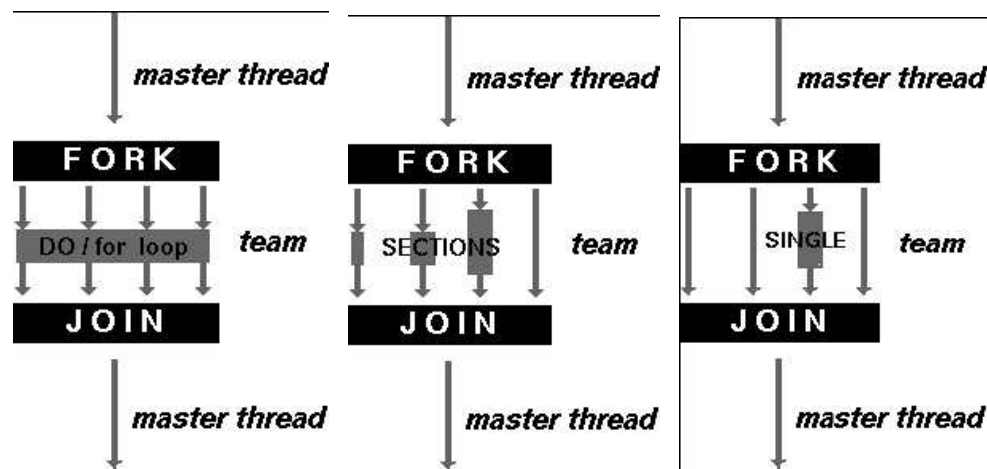
§3.2 Worksharing constructs - do/for, section, single

§3.2.1 The do/for directive

The `for` directive shares iterations of a loop across the team. It represents a type of “data parallelism”.

The `section` directive breaks work into separate, discrete sections. Each section is executed by a thread. It can be used to implement a type of “functional parallelism”.

The `single` directive serializes a section of code.



\$pragma omp for	schedule (type [, chunk]) ordered private (list) firstprivate (list) lastprivate (list) shared (list) reduction (operator: list) collapse(n) nowait
------------------	---

There are three kinds of schedules:

1. **static**: Loop iterations are divided into pieces of size *chunk* and then statically assigned to threads. If *chunk* is not specified, the iterations are evenly (if possible) divided contiguously among the threads.
2. **dynamic**: Loop iterations are divided into pieces of size *chunk*, and dynamically scheduled among the threads; when a thread finishes one chunk, it is dynamically assigned another. The default chunk size is 1.
3. **guided**: For a chunk size of 1, the size of each chunk is proportional to the number of unassigned iterations divided by the number of threads, decreasing to 1. For a chunk size with value k (greater than 1), the size of each chunk is determined in the same way with the restriction that the chunks do not contain fewer than k iterations (except for the last chunk to be

assigned, which may have fewer than k iterations). The default chunk size is 1.

4. **runtime**: The scheduling decision is deferred until runtime by the environment variable `OMP_SCHEDULE`. It is illegal to specify a chunk size for this clause.
5. **auto**: The scheduling decision is delegated to the compiler and/or runtime system.

The `nowait` clause specifies that the threads do not synchronize at the end of the parallel loop.

The `ordered` clause specifies that the iterations of the loop must be executed as they would be in a serial program. This is equivalent to serial execution.

The `collapse` clause specifies the number of loops in a nested loop that should be collapsed into one large iteration space and divided according to the `schedule` clause. The sequential execution of the iterations in all associated loops determines the order of the iterations in the collapsed iteration space.

The loop iteration variable must be an integer. The loop control parameters must be the same for all threads.

It is illegal to branch out of a loop associated with a `for` directive.

Example.

```
#include <omp.h>
#define CHUNKSIZE 100

#define N 1000 main () { int i, chunk; float a[N], b[N], c[N];
/* Some initializations */
for (i=0; i < N; i++)
    a[i] = b[i] = i * 1.0;
chunk = CHUNKSIZE;

#pragma omp parallel shared(a,b,c,chunk) private(i)
{

    #pragma omp for schedule(dynamic,chunk) nowait
    for (i=0; i < N; i++) c[i] = a[i] + b[i];
```

```
} /* end of parallel section */  
}
```

§3.2.2 The sections directive

The `section` directive specifies that the enclosed sections of code are to be divided among the threads in the team.

There is an implied barrier at the end of a `sections` directive, unless the `nowait` clause is used.

Two questions:

1. What happens if the number of threads and the number of `sections` are different? More threads than `sections`? Less threads than `sections`?
2. Which thread executes which `section`?

It is illegal to branch into or out of section blocks.

The `section` directives must occur within the lexical extent of an enclosing `sections` directive

<pre>\$pragma omp sections</pre>	<pre>private (list) firstprivate (list) lastprivate (list) reduction (operator: list) collapse(n) nowait</pre>
<pre>{ #pragma omp section structured_block #pragma omp section structured_block }</pre>	

Example.

```
#include <omp.h>
#define N 1000
main () { int i; float a[N], b[N], c[N], d[N];
/* Some initializations */
for (i=0; i < N; i++) { a[i] = i * 1.5;
b[i] = i + 22.35; }

#pragma omp parallel shared(a,b,c,d) private(i)
{
#pragma omp sections nowait
{
#pragma omp section
    for (i=0; i < N; i++) c[i] = a[i] + b[i];
#pragma omp section
    for (i=0; i < N; i++) d[i] = a[i] * b[i];
```

```
    } /* end of sections */  
} /* end of parallel section */ }
```

§3.2.3 The workshare directive

Only for Fortran.

§3.2.4 The `single` directive

The `single` directive specifies that the enclosed code is to be executed by only one thread in the team. This is useful for sections of code that are not thread safe, such as I/O.

Threads in the team that do not execute the `single` directive wait at the end of the enclosed code block, unless a `nowait` clause is specified.

<code>\$pragma omp single</code>	<code>private (list)</code> <code>firstprivate (list)</code> <code>nowait</code>
----------------------------------	--

We may combine parallel and worksharing constructs.

Example.

```
#include <omp.h>
#define N 1000
#define CHUNKSIZE 100
main () { int i, chunk; float a[N], b[N], c[N];
/* Some initializations */
for (i=0; i < N; i++) a[i] = b[i] = i * 1.0;
chunk = CHUNKSIZE;

#pragma omp parallel for \
        shared(a,b,c,chunk) private(i) \
        schedule(static,chunk)
    for (i=0; i < n; i++) c[i] = a[i] + b[i];
}
```

§3.3 the task directive

The `task` directive defines an explicit task, which may be executed by the encountering thread, or deferred for execution by any other thread in the team.

Task execution is subject to task scheduling.

\$pragma omp task	if (scalar expression) untied default(shared — none) private (list) firstprivate (list) shared (list)
-------------------	--

§3.4 Synchronization constructs

THREAD 1: increment(x) { x = x + 1; }	THREAD 2: increment(x) { x = x + 1; }
THREAD 1: 10 LOAD A, (x address) 20 ADD A, 1 30 STORE A, (x address)	THREAD 2: 10 LOAD A, (x address) 20 ADD A, 1 30 STORE A, (x address)

Here is a possible execution sequence:

1. Thread 1 loads the value of x into register A.
2. Thread 2 loads the value of x into register A.
3. Thread 1 adds 1 to register A.
4. Thread 2 adds 1 to register A.

5. Thread 1 stores register A at location x.
6. Thread 2 stores register A at location x.

The resultant value of x will be 1, not 2 as it should be.

When two (or more) threads access the same variable, they need be properly synchronized in order to insure a correct result is produced. OpenMP provides several synchronization constructs .

§3.4.1 The master directive

The `master` directive specifies a region that is to be executed only by the master thread of the team. All other threads on the team skip this section of code. There is no implied barrier associated with this directive.

<code>\$pragma omp master</code>	
----------------------------------	--

§3.4.2 The critical directive

The `critical` directive specifies a region of code that must be executed by only one thread at a time. If a thread is currently executing inside a `critical` region and another thread reaches that `critical` region and attempts to execute it, it will block until the first thread exits that `critical` region.

A critical region can be given a name. All critical regions with the same name are treated as the same region, that is, they are guided by the same lock.

All critical regions that have no name are also treated as the same critical region.

<code>\$pragma omp critical [name]</code>	
---	--

Example.

```
#include <omp.h>
main() {
    int x; x = 0;
    #pragma omp parallel shared(x)
    {
        #pragma omp critical
            x = x + 1;
    } /* end of parallel section */
}
```

§3.4.3 The barrier directive

The `barrier` directive sets up a point on which all threads in the team will wait until all threads reach that point. Then all threads resume execution as usual.

All threads in the team must execute the `barrier` region.

<code>\$pragma omp barrier</code>	
-----------------------------------	--

§3.4.4 The `taskwait` directive

The `taskwait` directive requires waiting for the completion of all child tasks that are generated since the beginning of the current task.

There is not a region of code for the `taskwait` directive. This directive can be placed where a C statement is allowed.

<code>\$pragma omp taskwait</code>	
------------------------------------	--

§3.4.5 The atomic directive

The `atomic` directive specifies that a specific memory location must be updated atomically, rather than letting multiple threads attempt to write to it. This is a mini-critical region.

<code>\$pragma omp atomic</code>	
----------------------------------	--

§3.4.6 The flush directive

The `flush` directive specifies a synchronization point at which all threads have a consistent view of memory. This means all copies of variables in the threads must be written back to memory.

The `flush` directive is needed even for a cache-coherent system.

The optional list names some variables that need to be flushed.

This is to avoid flushing all variables.

<code>\$pragma omp flush (list)</code>	
--	--

The `flush` directive is implied for the following directives. It is not implied if a *nowait* clause is present.

barrier

parallel - upon entry and exit

critical - upon entry and exit

ordered - upon entry and exit

for - upon exit

sections - upon exit

single - upon exit

§3.4.7 The ordered directive

The `ordered` directive specifies that the iterations of the enclosed loop must be executed in the same order as in a serial execution.

In an implementation, a thread will have to wait until all previous iterations are done.

An `ordered` directive can appear only within the dynamic extend of a `for` or `parallel for` directives.

Only one thread is allowed in an ordered section at any time.

```
$pragma omp for ordered [ clauses ... ]
```

```
... loop region ...
```

```
$pragma omp ordered
```

```
... structured block ...
```

```
... end of loop region ...
```

§3.4.8 The `threadprivate` directive

The `threadprivate` directive limits the scope of global file scope variables (C/C++) local and persistent to a single thread through the execution of multiple parallel regions.

The variables in the list must be declared before this directive.

<code>\$pragma omp threadprivate (list)</code>	
--	--

On the first entry into a parallel region, data in `threadprivate` variables should be assumed undefined, unless a `copyin` clause is specified in the `parallel` directive.

`Threadprivate` variables are different from `private` variables in that they persist between different parallel sections of code.

In Fortran only named `common` blocks can be made `threadprivate`.

Example.

C/C++ - threadprivate Directive Example

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

```
int a, b, i, tid; float x;
```

```
#pragma omp threadprivate(a, x)
```

```
main () { /* Explicitly turn off dynamic threads */
```

```
omp_set_dynamic(0);
```

```
printf("1st Parallel Region:\n");
```

```
#pragma omp parallel private(b,tid) {
```

```
    tid = omp_get_thread_num();
```

```
    a = tid;
```

```
    b = tid;
```

```
    x = 1.1 * tid + 1.0;
```

```
    printf("Thread %d: a,b,x= %d %d %f\n",tid,a,b,x);
```

```
} /* end of parallel section */
```

```

printf("*****\n");
printf("Master thread doing serial work here\n");
printf("*****\n");

printf("2nd Parallel Region:\n");
#pragma omp parallel private(tid) {
    tid = omp_get_thread_num();
    printf("Thread %d: a,b,x= %d %d %f\n",tid,a,b,x);
} /* end of parallel section */
}

```

Note that a and x persist across different parallel sections while b does not.

Output:

1st Parallel Region:

```
Thread 0: a,b,x= 0 0 1.000000
Thread 2: a,b,x= 2 2 3.200000
Thread 3: a,b,x= 3 3 4.300000
Thread 1: a,b,x= 1 1 2.100000
*****
Master thread doing serial work here
*****
2nd Parallel Region:
Thread 0: a,b,x= 0 0 1.000000
Thread 3: a,b,x= 3 0 4.300000
Thread 1: a,b,x= 1 0 2.100000
Thread 2: a,b,x= 2 0 3.200000
```

§3.5 Data scope (or data-sharing) attribute clauses

These clauses include `private`, `firstprivate`, `lastprivate`, `shared`, `default`, `reduction`, and `copyin`. They are used together with `parallel`, `for` and `sections` directive to define how enclosed variables should be scoped.

In a sequential programming language, static scoping is usually adopted. We can decide the scope of a variable by examining its location (of declaration). In OpenMP, however, multiple threads will have execute the same code. It could be that all threads share a single copy of a variable or each thread has a private copy of that variable.

Because OpenMP is based on a shared-memory programming model, most variables are shared by default. Global variables in C include file-scope variables and static variables. Private variables include loop index variables and stack variables in subroutines

called from parallel regions.

The data scope attribute clauses:

1. They define how and which variables in the serial sections are transferred to the parallel sections (and back).
2. They define which variables are visible to all threads in the parallel sections and which variables are privately allocated to all threads.

§3.5.1 The private clause

The `private` clause specifies a list of variables that are to be private to each thread.

<code>private (list)</code>	
-------------------------------	--

A new object of the same type is declared once for each thread in the team.

All references to the variable are replaced with references to the object in the thread.

The `private` variables are assumed to be uninitialized in the beginning.

A comparison of `private` and `threadprivate`:

	<code>private</code>	<code>threadprivate</code>
Data Item	C/C++: variable	C/C++: variable
Where Declared	At start of region or work-sharing group	In declarations of each routine using block or global file scope
Persistent?	NO	YES
Extent	Lexical only - unless passed as an argument to subroutine	Dynamic
Initialized	Use <code>FIRSTPRIVATE</code>	Use <code>COPYIN</code>

§3.5.2 The shared clause

The variables declared as `shared` are shared among all threads in the team.

<code>shared (list)</code>	
------------------------------	--

There is only one copy of a shared variable. All threads can read and write it.

The programmer has to take care of the synchronized access to shared variables.

§3.5.3 The default clause

The `default` clause allows a programmer to specify a default scope for all variables in the lexical extent of a parallel region.

Fortran	DEFAULT (PRIVATE FIRSTPRIVATE SHARED NONE)
C/C++	default (shared none)

All variables not declared with `private`, `shared`, `firstprivate`, `lastprivate`, and `reduction` will assume the option in the `default` clause.

If the default is `none`, the programmer must declare the scope of all variables.

Only one `default` clause can be specified on a `parallel` directive.

§3.5.4 The `firstprivate` clause

The `firstprivate` clause combines the `private` clause and automatic initialization of the variables in the list. The variables are initialized according to the value of their original objects prior to entry into the parallel or work-sharing construct.

C/C++	<code>firstprivate (list)</code>
-------	------------------------------------

§3.5.5 The `lastprivate` clause

The `lastprivate` clause combines the `private` clause with a copy from the last loop iteration or section of the enclosing construct to the original variable object.

For example, the team member which executes the final iteration for a `DO` section, or the team member which does the last `SECTION` of a `SECTIONS` context performs the copy with its own values.

C/C++	<code>lastprivate (list)</code>
-------	-----------------------------------

§3.5.6 The copyin clause

The `copyin` clause assigns the same value to `threadprivate` variables for all threads in the team.

The variable in the master thread is used as the copy source. The team threads are initialized with its value upon entry into the parallel construct.

C/C++	<code>copyin (list)</code>
-------	------------------------------

§3.5.7 The `copyprivate` clause

The `copyprivate` clause is used to broadcast values acquired by a thread directly to all instances of the private variables in the other thread.

This clause is associated with the `single` directive.

C/C++	<code>copyprivate (list)</code>
-------	-----------------------------------

§3.5.8 The reduction clause

The `reduction` clause performs a reduction on the variables appearing on its list. A private copy for each list variable is created for each thread. At the end of reduction, the reduction variable is applied to all private copies of the shared variable, and the final result is written to the global shared variable.

C/C++	<code>reduction (operator : list)</code>
-------	--

Example. The following example compute the vector dot product. Iterations of the parallel loop will be distributed in equal sized blocks to each thread in the team (`schedule static`). At the end of the parallel loop, all threads will add their values to “result” to update the master thread’s global copy.

```
#include <omp.h>
main () {
  int i, n, chunk;
```

```
float a[100], b[100], result;

/* Some initializations */
n = 100;
chunk = 10;
result = 0.0;
for (i=0; i < n; i++) {
    a[i] = i * 1.0;
    b[i] = i * 2.0; }

#pragma omp parallel for \
    default(shared) private(i) \
    schedule(static,chunk) \
    reduction(+:result)
for (i=0; i < n; i++)
    result = result + (a[i] * b[i]);
```

```
printf("Final result= %f\n",result);  
}
```

Variables in the list must be named scalar variables. They cannot be arrays or structures. They must also be declared **shared** in the enclosing context.

Reduction operations may not be associative for real numbers.

The **reduction** clause is intended to be used on a region or work-sharing construct in which the reduction variable is used only in statements which have one of the following forms:

Fortran	C/C++
<p>x = x operator expr</p> <p>x = expr operator x (except subtraction)</p> <p>x = intrinsic(x, expr)</p> <p>x = intrinsic(expr, x)</p>	<p>x = x op expr</p> <p>x = expr op x (except subtraction)</p> <p>x binop = expr</p> <p>x++ , ++x , --x , x--</p>
<p>x is a scalar variable in the list.</p> <p>expr is a scalar expression that does not reference x.</p> <p>intrinsic is one of MAX, MIN, IAND, IOR, IEOR.</p> <p>operator is one of +, *, -, .AND., .OR., .EQV., .NEQV..</p>	<p>x is a scalar variable in the list.</p> <p>expr is a scalar expression that does not reference x.</p> <p>op is not overloaded, and is one of +, *, -, /, &, ^, , &&, .</p> <p>binop is not overloaded, and is one of +, *, -, /, &, ^, —.</p>

§3.5.9 Clauses and directives summary

clause	parallel	for	sections	single	parallel for for	parallel sections
if	x				x	x
private	x	x	x	x	x	x
shared	x	x			x	x
default	x				x	x
firstprivate	x	x	x	x	x	x
lastprivate		x	x		x	x
reduction	x	x	x		x	x
copyin	x				x	x
copyprivate				x		
schedule		x			x	
ordered		x			x	
nowait		x	x	x		

The following directives do not accept clauses: MASTER, CRITICAL, BARRIER, ATOMIC, FLUSH, ORDERED, THREADPRIVATE.

§3.6 Directive binding and nesting rules

§3.7 Run-time library routines

OpenMP defines an API for library calls for a variety of functions.

1. query the number of threads, processors; set the number of threads to use.
2. locking (general-purpose, semaphores)
3. portable wall-clock timing routines (for performance measurement)
4. set execution environment functions: nested parallelism, dynamic adjustment of threads

We need to include “`omp.h`”.

For the locking operations, the lock variable must be accessed only through the locking routines. For Fortran, the lock variable must have integer type and is large enough to hold an address. For C/C++, the lock variable must have type `omp_lock_t` or type

`omp_nest_lock_t`, depending on the function being used.

An implementation may or may not support nested parallelism or dynamic threads. If nested parallelism is supported, it is usually only nominal, that is, a nested parallel region may only have one thread.

§3.7.1 OPM_SET_NUM_THREADS

Sets the number of threads that will be used in the next parallel region. The parameter must be a positive integer.

C/C++	<pre>#include <omp.h> void omp_set_num_threads(int num_threads)</pre>
-------	--

When the dynamic threads mechanism is enabled, this specifies the maximum number of threads that can be used for any parallel region.

When the dynamic threads mechanism is disabled, this specifies the exact number of threads to use until the next call to this routine.

This `OPM_SET_NUM_THREADS` routine can only be invoked from the serial portions of the code.

This `OPM_SET_NUM_THREADS` call has precedence over the `OMP_NUM_THREADS` environment variable.

§3.7.2 OMP_GET_NUM_THREADS

Returns the number of threads that are currently in the team executing the parallel region from which it is called.

C/C++	<pre>#include <omp.h> int omp_get_num_threads(void)</pre>
-------	---

If this call is made from a serial portion of the program, or a nested parallel region that is serialized, it will return 1.

The default number of threads is implementation dependent.

§3.7.3 OMP_GET_MAX_THREADS

Returns the maximum value that can be returned by a call to the `OMP_GET_NUM_THREADS` function.

C/C++	<pre>#include <omp.h> int omp_get_max_threads(void)</pre>
-------	---

Generally reflects the number of threads as set by the `OMP_NUM_THREADS` environment variable or the `OMP_SET_NUM_THREADS()` library routine.

May be called from both serial and parallel regions of code.

§3.7.4 OMP_GET_THREAD_NUM

Returns the thread number of the thread, within the team, making this call. This number will be between 0 and OMP_GET_NUM_THREADS-1. The master thread of the team is thread 0.

C/C++	<pre>#include <omp.h> int omp_get_thread_num(void)</pre>
-------	--

Example 1. Correct.

```
PROGRAM HELLO
  INTEGER TID, OMP_GET_THREAD_NUM

!$OMP PARALLEL PRIVATE(TID)

  TID = OMP_GET_THREAD_NUM()
  PRINT *, 'Hello World from thread = ', TID
  ...
```

```
!$OMP END PARALLEL  
    END
```

Example 2. Incorrect.

```
    PROGRAM HELLO  
    INTEGER TID, OMP_GET_THREAD_NUM  
  
!$OMP PARALLEL  
  
    TID = OMP_GET_THREAD_NUM()  
    PRINT *, 'Hello World from thread = ', TID  
    ...  
!$OMP END PARALLEL  
    END
```

Example 3. Incorrect.

```
PROGRAM HELLO
INTEGER TID, OMP_GET_THREAD_NUM

TID = OMP_GET_THREAD_NUM()
PRINT *, 'Hello World from thread = ', TID

!$OMP PARALLEL
    . . .
!$OMP END PARALLEL
END
```

§3.7.5 OMP_GET_THREAD_LIMIT

Returns the maximum number of OpenMP threads available to a program.

C/C++	<pre>#include <omp.h> int omp_get_thread_limit (void)</pre>
-------	---

This routine is

related to the OMP_THREAD_LIMIT environment variable.

§3.7.6 OMP_GET_NUM_PROCS

Returns the number of processors that are available to the program.

C/C++	<pre>#include <omp.h> int omp_get_num_procs(void)</pre>
-------	---

§3.7.7 OMP_IN_PARALLEL

This routine determines if the section of code under execution is parallel or not.

C/C++	<pre>#include <omp.h> int omp_in_parallel(void)</pre>
-------	---

§3.7.8 OMP_SET_DYNAMIC

If `dynamic_threads` evaluates to non-zero, then the dynamic adjustment mechanism is enabled, otherwise it is disabled.

This mechanism can dynamically adjust (by the run time system) the number of threads available for execution of parallel regions.

C/C++	<pre>#include <omp.h> void omp_set_dynamic(int dynamic_threads)</pre>
-------	---

The `OMP_SET_DYNAMIC` subroutine has precedence over the `OMP_DYNAMIC` environment variable.

The default setting is implementation dependent.

`OMP_SET_DYNAMIC` must be called from a serial section of the program.

§3.7.9 OMP_GET_DYNAMIC

OMP_GET_DYNAMIC determine if dynamic thread adjustment is enabled or not.

C/C++	<pre>#include <omp.h> int omp_get_dynamic(void)</pre>
-------	---

§3.7.10 OMP_SET_NESTED

OMP_SET_NESTED enables or disables nested parallelism.

C/C++	<pre>#include <omp.h> void omp_set_nested(int nested)</pre>
-------	---

Nested parallelism is disabled by default.

This call has precedence over the OMP_NESTED environment variable.

§3.7.11 OMP_GET_NESTED

OMP_GET_NESTED determines if nested parallelism is enabled or not.

C/C++	<pre>#include <omp.h> int omp_get_nested (void)</pre>
-------	---

§3.7.12 New routines for OpenMP 3.0

Please consult the most recent specs for details.

OMP_SET_SCHEDULE, OMP_GET_SCHEDULE,
OMP_SET_MAX_ACTIVE_LEVELS, OMP_GET_MAX_ACTIVE_LEVELS,
OMP_GET_LEVEL, OMP_GET_ANCESTOR_THREAD_NUM,
OMP_GET_TEAM_SIZE, OMP_GET_ACTIVE_LEVEL, OMP_INIT_NEST_LOCK,
OMP_DESTROY_NEST_LOCK, OMP_SET_NEST_LOCK,
OMP_UNSET_NEST_LOCK, OMP_TEST_NEST_LOCK,

§3.7.13 OMP_INIT_LOCK

OMP_INIT_LOCK initializes a lock associated with the lock variable. The initial state is unlocked.

C/C++	<pre>#include <omp.h> void omp_init_lock(omp_lock_t *lock)</pre>
-------	--

§3.7.14 OMP_DESTROY_LOCK

OMP_DESTROY_LOCK disassociates the given lock variable from any locks. It is illegal to call this routine with a lock variable that is not initialized.

C/C++	<pre>#include <omp.h> void omp_destroy_lock(omp_lock_t *lock) void omp_destroy_nest__lock(omp_nest_lock_t *lock)</pre>
-------	--

§3.7.15 OMP_SET_LOCK

OMP_SET_LOCK forces the executing thread to wait until the specified lock is available. A thread is granted ownership of a lock when it becomes available.

C/C++	<pre>#include <omp.h> void omp_set_lock(omp_lock_t *lock) void omp_set_nest_lock(omp_nest_lock_t *lock)</pre>
-------	---

§3.7.16 OMP_UNSET_LOCK

OMP_UNSET_LOCK releases the lock from the executing subroutine.

C/C++	<pre>#include <omp.h> void omp_unset_lock(omp_lock_t *lock) void omp_unset_nest__lock(omp_nest_lock_t *lock)</pre>
-------	--

§3.7.17 OMP_TEST_LOCK

OMP_TEST_LOCK attempts to set a lock, but does not block if the lock is unavailable. For C/C++, non-zero is returned if the lock was set successfully, otherwise zero is returned.

C/C++	<pre>#include <omp.h> int omp_test_lock(omp_lock_t *lock) int omp_test_nest__lock(omp_nest_lock_t *lock)</pre>
-------	--

§3.7.18 OMP_GET_WTIME

OMP_GET_WTIME provides a portable wall clock timing routine. It returns a double-precision floating-point value equal to the number of elapsed seconds since some point in the past. We usually use two calls and use the difference as the elapsed between the two calls. The result is interpreted as “per thread” time, and therefore may not be globally consistent across all threads in a team. It is not very meaningful to compare the readings from two threads.

C/C++	<pre>#include <omp.h> double omp_get_wtime(void)</pre>
-------	---

§3.7.19 OMP_GET_WTICK

OMP_GET_WTICK is a portable wall clock timing routine. It returns a double-precision floating-point value equal to the number of seconds between successive clock ticks.

C/C++	<pre>#include <omp.h> double omp_get_wtick(void)</pre>
-------	---

§3.8 Environment variables

Environment variables are the default setting for controlling the execution of parallel code.

All environment variable names are uppercase. The values assigned to them are not case sensitive.

OMP_SCHEDULE:

The **OMP_SCHEDULE** variable applies to only `for` and `parallel for` (C/C++) directives whose `schedule` clause is `RUNTIME`. The value of this variable determines how iterations of the loop are scheduled on processors. For example:

```
setenv OMP_SCHEDULE "guided, 4"  
setenv OMP_SCHEDULE "dynamic"
```

OMP_NUM_THREADS:

OMP_NUM_THREADS sets the maximum number of threads

to use during execution. For example:

```
setenv OMP_NUM_THREADS 8
```

OMP_DYNAMIC:

OMP_DYNAMIC enables or disables dynamic adjustment of the number of threads available for execution of parallel regions. Valid values are TRUE and FALSE. For example:

```
setenv OMP_DYNAMIC TRUE
```

OMP_NESTED:

OMP_NESTED enables or disables nested parallelism. Valid values are TRUE and FALSE. An implementation may or may not support nested parallelism and/or dynamic threads. If nested parallelism is supported, it is often only nominal, in that a nested parallel region may only have one thread. For example:

```
setenv OMP_NESTED TRUE
```

OMP_STACKSIZE:

This variable is new with OpenMP 3.0. **OMP_STACKSIZE** controls the size of the stack for created (non-master) threads.

Examples:

```
setenv OMP_STACKSIZE 2000500B
setenv OMP_STACKSIZE "3000 k "
setenv OMP_STACKSIZE 10M
setenv OMP_STACKSIZE " 10 M "
setenv OMP_STACKSIZE "20 m "
setenv OMP_STACKSIZE " 1G"
setenv OMP_STACKSIZE 20000
```

OMP_WAIT_POLICY:

This variable is new with OpenMP 3.0. **OMP_WAIT_POLICY** provides a hint to an OpenMP implementation about the desired behavior of waiting threads. A compliant OpenMP implementation may or may not abide by the setting of the environment variable. Valid values are **ACTIVE** and **PASSIVE**. **ACTIVE** specifies that waiting threads should mostly be active, i.e., consume processor cycles, while waiting. **PASSIVE** specifies that waiting threads should mostly be passive, i.e., not consume processor cycles, while waiting. The details of the **ACTIVE** and **PASSIVE** behaviors are implementation defined. Examples:

```
setenv OMP_WAIT_POLICY ACTIVE
setenv OMP_WAIT_POLICY active
setenv OMP_WAIT_POLICY PASSIVE
setenv OMP_WAIT_POLICY passive
```

OMP_MAX_ACTIVE_LEVELS:

This variable is new with OpenMP 3.0.

OMP_MAX_ACTIVE_LEVELS controls the maximum number of nested active parallel regions. The value of this environment variable must be a non-negative integer. The behavior of the program is implementation defined if the requested value of **OMP_MAX_ACTIVE_LEVELS** is greater than the maximum number of nested active parallel levels an implementation can support, or if the value is not a non-negative integer. Example:

```
setenv OMP_MAX_ACTIVE_LEVELS 2
```

OMP_THREAD_LIMIT:

This variable is new with OpenMP 3.0. Sets the number of OpenMP threads to use for the whole OpenMP program. The value of this environment variable must be a positive integer. The behavior of the program is implementation defined if the requested value of **OMP_THREAD_LIMIT** is greater than the number of threads

an implementation can support, or if the value is not a positive integer. Example:

```
setenv OMP_THREAD_LIMIT 8
```

§3.9 Memory and performance issues

The OpenMP standard does not specify how much stack space a thread should have. Consequently, implementations will differ in the default thread stack size.

Default thread stack size can be easy to exhaust. It can also be non-portable between compilers. For example, the table below shows some approximate thread stack size limits at LC using the default version compilers (Jan '09):

compiler	approx. stack limit	approx. array size (doubles)
IBM POWER5 AIX: xlc, xlf	8 MB	707 x 707
AMD Opteron Linux icc, ifort	8 MB	707 x 707
AMD Opteron Linux pgcc, pgf90	16 MB	1024 x 1024
AMD Opteron Linux gcc, gfortran	4 MB	512 x 512
AMD Opteron Linux pathcc pathf90	67 MB >1.4GB	2048 x 2048 >9500 x 9500

Threads that exceed their stack allocation may or may not segment-fault. An application may continue to run while data is being corrupted.

Statically linked codes may be subject to further stack restrictions.

A user's login shell may also restrict stack size.

Examples for increasing the thread stack size to 12 MB at LC:

platform	shell command csh	shell command ksh
IBM POWER5 AIX	limit stacksize 12288 setenv XLSMPOPTS "stack=12000000"	ulimit -s 12288 export XLSMPOPTS= "stack=12000000"
AMD Opteron Linux	limit stacksize 12288 setenv KMP_STACKSIZE 12000000	ulimit -s 12288 export KMP_STACKSIZE=12000000

Compiling: Use the following compiler flags to turn on OpenMP compilations:

compiler	flag
IBM	-qsmp=omp
Intel	-openmp
PathScale	-mp
PGI	-mp
GNU	-fopenmp

This is the end of the slides.