

OpenMP

Table of Contents

1. [Introduction](#)
 1. [What is OpenMP?](#)
 2. [History](#)
 3. [Goals of OpenMP](#)
2. [OpenMP Programming Model](#)
3. [OpenMP Directives](#)
 1. [Fortran Directive Format](#)
 2. [C/C++ Directive Format](#)
 3. [Directive Scoping](#)
 4. [PARALLEL Construct](#)
 5. [Work-Sharing Constructs](#)
 1. [DO / for Directive](#)
 2. [SECTIONS Directive](#)
 3. [SINGLE Directive](#)
 6. [Combined Parallel Work-Sharing Constructs](#)
 1. [PARALLEL DO / parallel for Directive](#)
 2. [PARALLEL SECTIONS Directive](#)
 7. [Synchronization Constructs](#)
 1. [MASTER Directive](#)
 2. [CRITICAL Directive](#)
 3. [BARRIER Directive](#)
 4. [ATOMIC Directive](#)
 5. [FLUSH Directive](#)
 6. [ORDERED Directive](#)
 8. [THREADPRIVATE Directive](#)
 9. [Data Scope Attribute Clauses](#)
 1. [PRIVATE Clause](#)
 2. [SHARED Clause](#)
 3. [DEFAULT Clause](#)
 4. [FIRSTPRIVATE Clause](#)
 5. [LASTPRIVATE Clause](#)
 6. [COPYIN Clause](#)
 7. [REDUCTION Clause](#)
 10. [Clauses / Directives Summary](#)
 11. [Directive Binding and Nesting Rules](#)
4. [Run-Time Library Routines](#)

1. [OMP_SET_NUM_THREADS](#)
 2. [OMP_GET_NUM_THREADS](#)
 3. [OMP_GET_MAX_THREADS](#)
 4. [OMP_GET_THREAD_NUM](#)
 5. [OMP_GET_NUM_PROCS](#)
 6. [OMP_IN_PARALLEL](#)
 7. [OMP_SET_DYNAMIC](#)
 8. [OMP_GET_DYNAMIC](#)
 9. [OMP_SET_NESTED](#)
 10. [OMP_GET_NESTED](#)
 11. [OMP_INIT_LOCK](#)
 12. [OMP_DESTROY_LOCK](#)
 13. [OMP_SET_LOCK](#)
 14. [OMP_UNSET_LOCK](#)
 15. [OMP_TEST_LOCK](#)
5. [Environment Variables](#)
 6. [LLNL Specific Information and Recommendations](#)
 7. [References and More Information](#)
 8. [Exercise](#)

Introduction

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

- What does OpenMP stand for?
Open specifications for Multi Processing via collaborative work with 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)
-

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.

▶ 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](#))

OpenMP Architecture Review Board (open to new members)

- Compaq / Digital
- Hewlett-Packard Company
- Intel Corporation
- International Business Machines (IBM)
- Kuck & Associates, Inc. (KAI)

- Silicon Graphics, Inc.
- Sun Microsystems, Inc.
- U.S. Department of Energy ASCI program

Endorsing software vendors:

- Absoft Corporation
- Edinburgh Portable Compilers
- GENIAS Software GmbH
- Myrias Computer Technologies, Inc.
- The Portland Group, Inc. (PGI)

Endorsing application developers:

- ADINA R&D, Inc.
- ANSYS, Inc.
- Dash Associates
- Fluent, Inc.
- ILOG CPLEX Division
- Livermore Software Technology Corporation (LSTC)
- MECALOG SARL
- Oxford Molecular Group PLC
- The Numerical Algorithms Group Ltd.(NAG)

► Release History

- October 1997: Fortran version 1.0
- Late 1998: C/C++ version 1.0
- June 2000: Fortran version 2.0
- April 2002: C/C++ version 2.0



- Visit the OpenMP website at <http://www.openmp.org/> for more information, including API specifications, FAQ, presentations, discussions, media releases, calendar and membership application.

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

▶ **Thread Based Parallelism:**

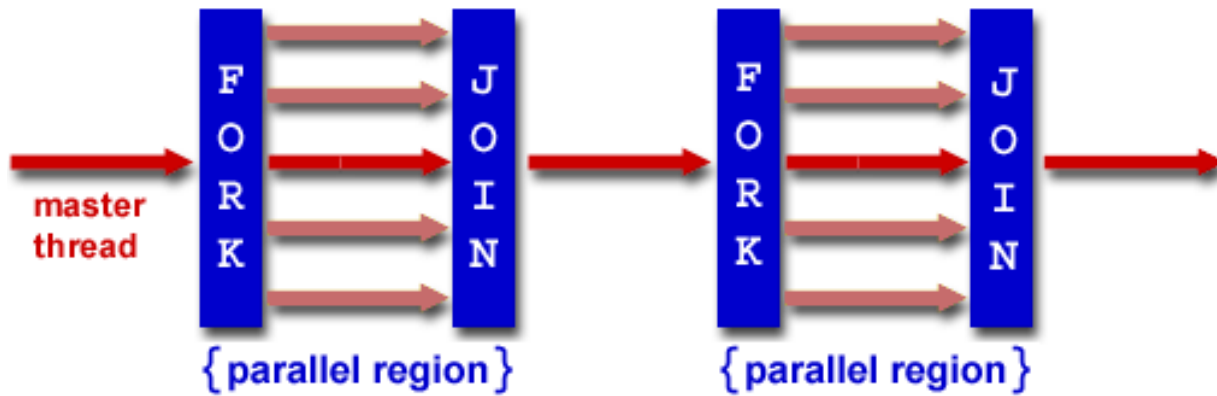
- A shared memory process can consist of multiple threads. OpenMP is based upon the existence of multiple threads in the shared memory programming paradigm.

▶ **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

► Compiler Directive Based:

- Virtually all of 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 be used to execute different parallel regions.
- Implementations may or may not support this feature.

Example OpenMP Code Structure



Fortran - General Code Structure

```
PROGRAM HELLO
```

```
INTEGER VAR1, VAR2, VAR3
```

```
Serial code
```

```
·  
·  
·
```

```
Beginning of parallel section. Fork a team of threads.  
Specify variable scoping
```

```
!$OMP PARALLEL PRIVATE(VAR1, VAR2) SHARED(VAR3)
```

```
Parallel section executed by all threads
```

```
·  
·  
·
```

```
All threads join master thread and disband
```

```
!$OMP END PARALLEL
```

```
Resume serial code
```

```
·  
·  
·
```

```
END
```



C / C++ - General Code 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
    .
    .
    .
}

```

OpenMP Directives

Fortran Directives Format

► Format:

sentinel	directive-name	[clause ...]
<p>All Fortran OpenMP directives must begin with a sentinel. The accepted sentinels depend upon the type of Fortran source. Possible sentinels are:</p> <p>!\$OMP C\$OMP *\$OMP</p>	<p>A valid OpenMP directive. Must appear after the sentinel and before any clauses.</p>	<p>Optional. Clauses can be in any order, and repeated as necessary unless otherwise restricted.</p>

► Example:

```
!$OMP PARALLEL DEFAULT(SHARED) PRIVATE(BETA,PI)
```

► Fixed Form Source:

- **!\$OMP C\$OMP *\$OMP** are accepted sentinels and must start in column 1

- All Fortran fixed form rules for line length, white space, continuation and comment columns apply for the entire directive line
- Initial directive lines must have a space/zero in column 6.
- Continuation lines must have a non-space/zero in column 6.

► Free Form Source:

- **!\$OMP** is the only accepted sentinel. Can appear in any column, but must be preceded by white space only.
- All Fortran free form rules for line length, white space, continuation and comment columns apply for the entire directive line
- Initial directive lines must have a space after the sentinel.
- Continuation lines must have an ampersand as the last non-blank character in a line. The following line must begin with a sentinel and then the continuation directives.

► General Rules:

- Comments can not appear on the same line as a directive
- Fortran compilers which are OpenMP enabled generally include a command line option which instructs the compiler to activate and interpret all OpenMP directives.
- Several Fortran OpenMP directives come in pairs and have the form:

```
!$OMP  directive  
  
    [ structured block of code ]  
  
!$OMP end  directive
```

OpenMP Directives

C / C++ Directives Format

► Format:

<code>#pragma omp</code>	directive-name	[clause, ...]	newline
Required for all OpenMP C/C++ directives.	A valid OpenMP directive. Must appear after the pragma and before any clauses.	Optional. Clauses can be in any order, and repeated as necessary unless otherwise restricted.	Required. Proceeds the structured block which is enclosed by this directive.

▶ Example:

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

▶ General Rules:

- Directives follow conventions of the C/C++ standards for compiler directives
- Case sensitive
- Only one directive-name may be specified per directive (true with Fortran also)
- 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

Directive Scoping

▶ Static (Lexical) Extent:

- The code textually enclosed between the beginning and the end of a structured block following a directive.
- The static extent of a directives does not span multiple routines or code files

▶ Orphaned Directive:

- An OpenMP directive that appears independently from another enclosing directive is said to be an orphaned

directive. It exists outside of another directive's static (lexical) extent.

- Will span routines and possibly code files

► Dynamic Extent:

- The dynamic extent of a directive includes both its static (lexical) extent and the extents of its orphaned directives.

► Example:

<pre> PROGRAM TEST ... !\$OMP PARALLEL ... !\$OMP DO DO I=... ... CALL SUB1 ... ENDDO ... CALL SUB2 ... !\$OMP END PARALLEL </pre>	<pre> SUBROUTINE SUB1 ... !\$OMP CRITICAL ... !\$OMP END CRITICAL END SUBROUTINE SUB2 ... !\$OMP SECTIONS ... !\$OMP END SECTIONS ... END </pre>
<p>STATIC EXTENT</p> <p>The DO directive occurs within an enclosing parallel region</p>	<p>ORPHANED DIRECTIVES</p> <p>The CRITICAL and SECTIONS directives occur outside an enclosing parallel region</p>
<p>DYNAMIC EXTENT</p>	

► Why Is This Important?

- OpenMP specifies a number of scoping rules on how directives may associate (bind) and nest within each other
- Illegal and/or incorrect programs may result if the OpenMP binding and nesting rules are ignored
- See [Directive Binding and Nesting Rules](#) for specific details

PARALLEL Region Construct

Purpose:

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

Format:

Fortran	<pre> !\$OMP PARALLEL [<i>clause ...</i>] IF (<i>scalar_logical_expression</i>) PRIVATE (<i>list</i>) SHARED (<i>list</i>) DEFAULT (PRIVATE SHARED NONE) FIRSTPRIVATE (<i>list</i>) REDUCTION (<i>operator: list</i>) COPYIN (<i>list</i>) <i>block</i> !\$OMP END PARALLEL </pre>
C/C++	<pre> #pragma omp parallel [<i>clause ...</i>] <i>newline</i> if (<i>scalar_expression</i>) private (<i>list</i>) shared (<i>list</i>) default (shared none) firstprivate (<i>list</i>) reduction (<i>operator: list</i>) copyin (<i>list</i>) <i>structured_block</i> </pre>

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.
- 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.

How Many Threads?

- The number of threads in a parallel region is determined by the following factors, in order of precedence:
 1. Use of the `omp_set_num_threads()` library function
 2. Setting of the `OMP_NUM_THREADS` environment variable
 3. Implementation default
- Threads are numbered from 0 (master thread) to N-1

► Dynamic Threads:

- By default, a program with multiple parallel regions will use the same number of threads to execute each region. This behavior can be changed to allow the run-time system to dynamically adjust the number of threads that are created for a given parallel section. The two methods available for enabling dynamic threads are:
 1. Use of the `omp_set_dynamic()` library function
 2. Setting of the `OMP_DYNAMIC` environment variable

► Nested Parallel Regions:

- A parallel region nested within another parallel region results in the creation of a new team, consisting of one thread, by default.
- Implementations may allow more than one thread in nested parallel regions

► 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.
- The remaining clauses are described in detail later, in the [Data Scope Attribute Clauses](#) section.

► Restrictions:

- A parallel region must be a structured block that does not span multiple routines or code files
- Unsynchronized Fortran I/O to the same unit by multiple threads has unspecified behavior
- It is illegal to branch into or out of a parallel region
- Only a single IF clause is permitted

Example: Parallel Region

- 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



Fortran - Parallel Region Example

```

PROGRAM HELLO

  INTEGER NTHREADS, TID, OMP_GET_NUM_THREADS,
+   OMP_GET_THREAD_NUM

C   Fork a team of threads giving them their own copies of variables
!$OMP PARALLEL PRIVATE(NTHREADS, TID)

C   Obtain and print thread id
TID = OMP_GET_THREAD_NUM()
PRINT *, 'Hello World from thread = ', TID

C   Only master thread does this
IF (TID .EQ. 0) THEN
  NTHREADS = OMP_GET_NUM_THREADS()
  PRINT *, 'Number of threads = ', NTHREADS
END IF

C   All threads join master thread and disband
!$OMP END PARALLEL

  END

```



C / C++ - Parallel Region Example

```

#include <omp.h>

main () {

  int nthreads, tid;

  /* Fork a team of threads giving them their own copies of variables */
  #pragma omp parallel private(nthreads, 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 */
}

```

OpenMP Directives

Work-Sharing Constructs

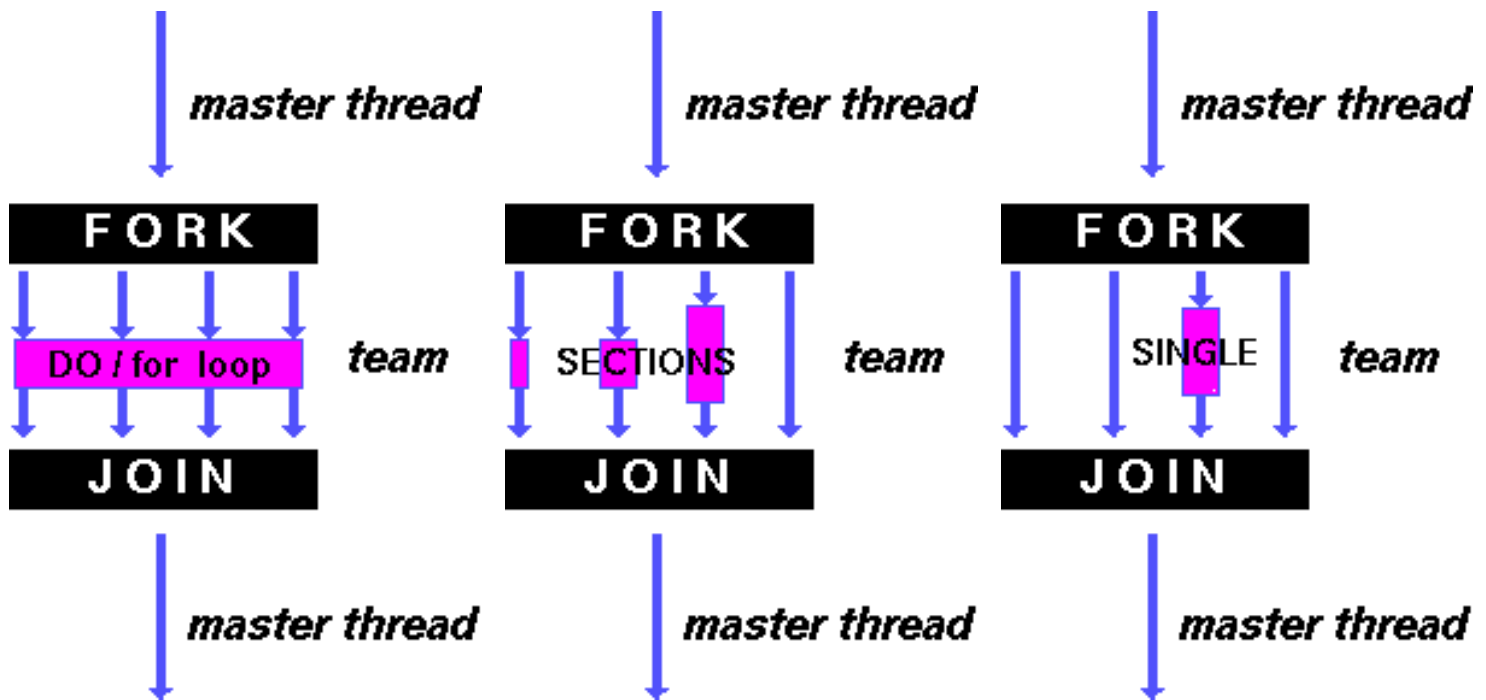
- A work-sharing construct divides the execution of the enclosed code region among the members of the team that encounter it.
- Work-sharing constructs do not launch new threads
- There is no implied barrier upon entry to a work-sharing construct, however there is an implied barrier at the end of a work sharing construct.

► Types of Work-Sharing Constructs:

DO / for - shares iterations of a loop across the team. Represents a type of "data parallelism".

SECTIONS - breaks work into separate, discrete sections. Each section is executed by a thread. Can be used to implement a type of "functional parallelism".

SINGLE - serializes a section of code



► Restrictions:

- A work-sharing construct must be enclosed dynamically within a parallel region in order for the directive to execute in parallel.
- Work-sharing constructs must be encountered by all members of a team or none at all
- Successive work-sharing constructs must be encountered in the same order by all members of a team

OpenMP Directives

Work-Sharing Constructs DO / for Directive

► Purpose:

- The DO / for directive specifies that the iterations of the loop immediately following it must be executed in parallel by the team. This assumes a parallel region has already been initiated, otherwise it executes in serial on a single processor.

► Format:

Fortran	<pre> !\$OMP DO [clause ...] SCHEDULE (type [,chunk]) ORDERED PRIVATE (list) FIRSTPRIVATE (list) LASTPRIVATE (list) SHARED (list) REDUCTION (operator / intrinsic : list) do_loop !\$OMP END DO [NOWAIT] </pre>
C/C++	<pre> #pragma omp for [clause ...] newline schedule (type [,chunk]) ordered private (list) firstprivate (list) lastprivate (list) shared (list) reduction (operator: list) nowait for_loop </pre>

► Clauses:

- **SCHEDULE** clause: Describes how iterations of the loop are divided among the threads in the team. For both C/C++ and Fortran:

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.

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.

GUIDED:

The chunk size is exponentially reduced with each dispatched piece of the iteration space. The chunk size specifies the minimum number of iterations to dispatch each time.. The default chunk size is 1.

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.

The default schedule is implementation dependent. Implementation may also vary slightly in the way the various schedules are implemented.

- **ORDERED** clause: Must be present when ORDERED directives are enclosed within the DO/for directive. See [Ordered Directive](#).
- **NO WAIT** (Fortran) / **nowait** (C/C++) clause: if specified, then threads do not synchronize at the end of the parallel loop. Threads proceed directly to the next statements after the loop. For Fortran, the END DO directive is optional with NO WAIT being the default.
- Other clauses are described in detail later, in the [Data Scope Attribute Clauses](#) section.

► Restrictions:

- The DO loop can not be a DO WHILE loop, or a loop without loop control. Also, the loop iteration variable must be an integer and the loop control parameters must be the same for all threads.
- Program correctness must not depend upon which thread executes a particular iteration.
- It is illegal to branch out of a loop associated with a DO/for directive.
- The chunk size must be specified as a loop invariant integer expression, as there is no synchronization during its evaluation by different threads.
- The C/C++ **for** directive requires that the for-loop must have canonical shape. See the [OpenMP API specification](#) for details.
- ORDERED and SCHEDULE clauses may appear once each.

Example: DO / for Directive

- Simple vector-add program
 - Arrays A, B, C, and variable N will be shared by all threads.
 - Variable I will be private to each thread; each thread will have its own unique copy.
 - The iterations of the loop will be distributed dynamically in CHUNK sized pieces.
 - Threads will not synchronize upon completing their individual pieces of work (NOWAIT).



Fortran - DO Directive Example

```

PROGRAM VEC_ADD_DO

INTEGER N, CHUNKSIZE, CHUNK, I
PARAMETER (N=1000)
PARAMETER (CHUNKSIZE=100)
REAL A(N), B(N), C(N)

!   Some initializations
DO I = 1, N
    A(I) = I * 1.0
    B(I) = A(I)
ENDDO
CHUNK = CHUNKSIZE

!$OMP PARALLEL SHARED(A,B,C,CHUNK) PRIVATE(I)

!$OMP DO SCHEDULE(DYNAMIC,CHUNK)
DO I = 1, N
    C(I) = A(I) + B(I)
ENDDO
!$OMP END DO NOWAIT

!$OMP END PARALLEL

END

```



C / C++ - 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 */
}

```

OpenMP Directives

Work-Sharing Constructs SECTIONS Directive

► Purpose:

- The SECTIONS directive is a non-iterative work-sharing construct. It specifies that the enclosed section(s) of code are to be divided among the threads in the team.
- Independent SECTION directives are nested within a SECTIONS directive. Each SECTION is executed once by a thread in the team. Different sections will be executed by different threads.

► Format:

Fortran	<pre> !\$OMP SECTIONS [<i>clause ...</i>] PRIVATE (<i>list</i>) FIRSTPRIVATE (<i>list</i>) LASTPRIVATE (<i>list</i>) REDUCTION (<i>operator</i> <i>intrinsic</i> : <i>list</i>) !\$OMP SECTION <i>block</i> !\$OMP SECTION <i>block</i> !\$OMP END SECTIONS [NOWAIT] </pre>
----------------	--

C/C++	<pre> #pragma omp sections [<i>clause ...</i>] <i>newline</i> private (<i>list</i>) firstprivate (<i>list</i>) lastprivate (<i>list</i>) reduction (<i>operator: list</i>) nowait { #pragma omp section <i>newline</i> <i>structured_block</i> #pragma omp section <i>newline</i> <i>structured_block</i> } </pre>
-------	--

▶ Clauses:

- There is an implied barrier at the end of a SECTIONS directive, unless the `nowait` (C/C++) or `NOWAIT` (Fortran) clause is used.
- Clauses are described in detail later, in the [Data Scope Attribute Clauses](#) section.

▶ Questions:



What happens if the number of threads and the number of SECTIONS are different? More threads than SECTIONS? Less threads than SECTIONS?



Which thread executes which SECTION?

▶ Restrictions:

- It is illegal to branch into or out of section blocks.
- SECTION directives must occur within the lexical extent of an enclosing SECTIONS directive

Example: SECTIONS Directive

- Simple vector-add program - similar to example used previously for the `DO / for` directive.
 - The first $n/2$ iterations of the DO loop will be distributed to the first thread, and the rest will be distributed to the second thread

- When each thread finishes its block of iterations, it proceeds with whatever code comes next (NOWAIT)



Fortran - SECTIONS Directive Example

```

PROGRAM VEC_ADD_SECTIONS

INTEGER N, I
PARAMETER (N=1000)
REAL A(N), B(N), C(N)

!   Some initializations
DO I = 1, N
    A(I) = I * 1.0
    B(I) = A(I)
ENDDO

!$OMP PARALLEL SHARED(A,B,C), PRIVATE(I)

!$OMP SECTIONS

!$OMP SECTION
DO I = 1, N/2
    C(I) = A(I) + B(I)
ENDDO

!$OMP SECTION
DO I = 1+N/2, N
    C(I) = A(I) + B(I)
ENDDO

!$OMP END SECTIONS NOWAIT

!$OMP END PARALLEL

END

```



C / C++ - sections Directive Example

```

#include <omp.h>
#define N      1000

main ()
{

int i;
float a[N], b[N], c[N];

/* Some initializations */
for (i=0; i < N; i++)

```

```

a[i] = b[i] = i * 1.0;

#pragma omp parallel shared(a,b,c) private(i)
{
    #pragma omp sections nowait
    {
        #pragma omp section
        for (i=0; i < N/2; i++)
            c[i] = a[i] + b[i];

        #pragma omp section
        for (i=N/2; i < N; i++)
            c[i] = a[i] + b[i];

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

```

OpenMP Directives

Work-Sharing Constructs SINGLE Directive

► Purpose:

- The SINGLE directive specifies that the enclosed code is to be executed by only one thread in the team.
- May be useful when dealing with sections of code that are not thread safe (such as I/O)

► Format:

Fortran	<pre> !\$OMP SINGLE [<i>clause ...</i>] PRIVATE (<i>list</i>) FIRSTPRIVATE (<i>list</i>) <i>block</i> !\$OMP END SINGLE [NOWAIT] </pre>
----------------	--

C/C++	<pre>#pragma omp single [clause ...] newline private (list) firstprivate (list) nowait structured_block</pre>
-------	--

► Clauses:

- Threads in the team that do not execute the SINGLE directive, wait at the end of the enclosed code block, unless a `nowait` (C/C++) or `NOWAIT` (Fortran) clause is specified.
- Clauses are described in detail later, in the [Data Scope Attribute Clauses](#) section.

► Restrictions:

- It is illegal to branch into or out of a SINGLE block.

OpenMP Directives

Combined Parallel Work-Sharing Constructs PARALLEL DO / parallel for Directive

- Iterations of the DO/for loop will be distributed in equal sized blocks to each thread in the team (SCHEDULE STATIC)



Fortran - PARALLEL DO Directive Example

```
PROGRAM VECTOR_ADD

INTEGER N, I, CHUNKSIZE, CHUNK
PARAMETER (N=1000)
PARAMETER (CHUNKSIZE=100)
REAL A(N), B(N), C(N)

! Some initializations
DO I = 1, N
  A(I) = I * 1.0
  B(I) = A(I)
ENDDO
CHUNK = CHUNKSIZE
```



```

!$OMP PARALLEL DO
!$OMP& SHARED(A,B,C,CHUNK) PRIVATE(I)
!$OMP& SCHEDULE(STATIC,CHUNK)

    DO I = 1, N
        C(I) = A(I) + B(I)
    ENDDO

!$OMP END PARALLEL DO

END

```

C / C++ - parallel for Directive 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];
}

```

OpenMP Directives

Combined Parallel Work-Sharing Constructs PARALLEL SECTIONS Directive

Purpose:

- The PARALLEL SECTIONS directive specifies a parallel region containing a single SECTIONS directive. The single SECTIONS directive must follow immediately as the very next statement.

► Format:

Fortran	<pre>!\$OMP PARALLEL SECTIONS [<i>clause ...</i>] DEFAULT (PRIVATE SHARED NONE) SHARED (<i>list</i>) PRIVATE (<i>list</i>) FIRSTPRIVATE (<i>list</i>) LASTPRIVATE (<i>list</i>) REDUCTION (<i>operator</i> <i>intrinsic</i> : <i>list</i>) COPYIN (<i>list</i>) ORDERED <i>structured_block</i> !\$OMP END PARALLEL SECTIONS</pre>
C/C++	<pre>#pragma omp parallel sections [<i>clause ...</i>] <i>newline</i> default (shared none) shared (<i>list</i>) private (<i>list</i>) firstprivate (<i>list</i>) lastprivate (<i>list</i>) reduction (<i>operator</i>: <i>list</i>) copyin (<i>list</i>) ordered <i>structured_block</i></pre>

► Clauses:

- The accepted clauses can be any of the clauses accepted by the PARALLEL and SECTIONS directives. Clauses not previously discussed, are described in detail later, in the [Data Scope Attribute Clauses](#) section.

OpenMP Directives

Synchronization Constructs

- Consider a simple example where two threads on two different processors are both trying to increment a variable *x* at the same time (assume *x* is initially 0):

<p>THREAD 1:</p> <pre>increment(x) { x = x + 1; }</pre> <p>THREAD 1:</p> <pre>10 LOAD A, (x address) 20 ADD A, 1 30 STORE A, (x address)</pre>	<p>THREAD 2:</p> <pre>increment(x) { x = x + 1; }</pre> <p>THREAD 2:</p> <pre>10 LOAD A, (x address) 20 ADD A, 1 30 STORE A, (x address)</pre>
---	---

- One 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.

- To avoid a situation like this, the incrementation of x must be synchronized between the two threads to insure that the correct result is produced.
- OpenMP provides a variety of Synchronization Constructs that control how the execution of each thread proceeds relative to other team threads.

OpenMP Directives

Synchronization Constructs

MASTER Directive

► Purpose:

- 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

► Format:

Fortran	<pre>!\$OMP MASTER <i>block</i> !\$OMP END MASTER</pre>
C/C++	<pre>#pragma omp master <i>newline</i> <i>structured_block</i></pre>

► Restrictions:

- It is illegal to branch into or out of MASTER block.

OpenMP Directives

Synchronization Constructs

CRITICAL Directive

► Purpose:

- The CRITICAL directive specifies a region of code that must be executed by only one thread at a time.

► Format:

Fortran	<pre>!\$OMP CRITICAL [<i>name</i>] <i>block</i> !\$OMP END CRITICAL</pre>
C/C++	<pre>#pragma omp critical [<i>name</i>] <i>newline</i> <i>structured_block</i></pre>

► Notes:

- 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.
- The optional name enables multiple different CRITICAL regions to exist:
 - Names act as global identifiers. Different CRITICAL regions with the same name are treated as the same region.
 - All CRITICAL sections which are unnamed, are treated as the same section.

► Restrictions:

- It is illegal to branch into or out of a CRITICAL block.

Example: CRITICAL Construct

- All threads in the team will attempt to execute in parallel, however, because of the CRITICAL construct surrounding the increment of x, only one thread will be able to read/increment/write x at any time



Fortran - CRITICAL Directive Example

```
PROGRAM CRITICAL

INTEGER X
X = 0

!$OMP PARALLEL SHARED(X)

!$OMP CRITICAL
  X = X + 1
!$OMP END CRITICAL

!$OMP END PARALLEL

END
```



C / C++ - critical Directive Example

```

#include <omp.h>

main()
{
  int x;
  x = 0;

  #pragma omp parallel shared(x)
  {
    #pragma omp critical
    x = x + 1;

  } /* end of parallel section */
}

```

OpenMP Directives

Synchronization Constructs

BARRIER Directive

► Purpose:

- The BARRIER directive synchronizes all threads in the team.
- When a BARRIER directive is reached, a thread will wait at that point until all other threads have reached that barrier. All threads then resume executing in parallel the code that follows the barrier.

► Format:

Fortran	!\$OMP BARRIER
C/C++	#pragma omp barrier <i>newline</i>

► Restrictions:

- For C/C++, the smallest statement that contains a barrier must be a structured block. For example:

WRONG	RIGHT
<pre>if (x == 0) #pragma omp barrier</pre>	<pre>if (x == 0) { #pragma omp barrier }</pre>

OpenMP Directives

Synchronization Constructs

ATOMIC Directive

► Purpose:

- The ATOMIC directive specifies that a specific memory location must be updated atomically, rather than letting multiple threads attempt to write to it. In essence, this directive provides a mini-CRITICAL section.

► Format:

Fortran	<pre>!\$OMP ATOMIC <i>statement_expression</i></pre>
C/C++	<pre>#pragma omp atomic <i>newline</i> <i>statement_expression</i></pre>

► Restrictions:

- The directive applies only to a single, immediately following statement
- An atomic statement must have one of the following forms:

Fortran	C / C++

$x = x \text{ operator } expr$ $x = expr \text{ operator } x$ $x = \text{intrinsic}(x, expr)$ $x = \text{intrinsic}(expr, x)$	$x \text{ binop} = expr$ $x++$ $++x$ $x--$ $--x$
<p>x is a scalar variable</p> <p>$expr$ is a scalar expression that does not reference x</p> <p>intrinsic is one of MAX, MIN, IAND, IOR, or IEOB</p> <p>operator is one of +, *, -, /, .AND., .OR., .EQV., or .NEQV.</p>	<p>x is a scalar variable</p> <p>$expr$ is a scalar expression that does not reference x</p> <p>binop is not overloaded, and is one of +, *, -, /, &, ^, , >>, or <<</p>

- Note: Only the load and store of x are atomic; the evaluation of the expression is not atomic.

OpenMP Directives

Synchronization Constructs

FLUSH Directive

Purpose:

- The FLUSH directive identifies a synchronization point at which the implementation must provide a consistent view of memory. Thread-visible variables are written back to memory at this point.

Format:

Fortran	<code>!\$OMP FLUSH (list)</code>
C/C++	<code>#pragma omp flush (list) newline</code>

Notes:

- Thread-visible variables include:
 - Globally visible variables (common blocks and modules)

- Local variables that do not have the SAVE attribute but have had their address used by another subprogram
 - Local variables that do not have the SAVE attribute that are declared shared in a parallel region within the subprogram
 - Dummy arguments
 - All pointer dereferences
- The optional list contains a list of named variables that will be flushed in order to avoid flushing all variables. For pointers in the list, note that the pointer itself is flushed, not the object it points to.
 - Implementations must ensure any prior modifications to thread-visible variables are visible to all threads after this point; ie. compilers must restore values from registers to memory, hardware might need to flush write buffers, etc
 - The FLUSH directive is implied for the directives shown in the table below. The directive is not implied if a NOWAIT clause is present.

Fortran	C / C++
BARRIER	barrier
CRITICAL and END CRITICAL	critical- upon entry and exit
END DO	ordered- upon entry and exit
END PARALLEL	parallel- upon exit
END SECTIONS	for- upon exit
END SINGLE	sections- upon exit
ORDERED and END ORDERED	single- upon exit

OpenMP Directives

Synchronization Constructs

ORDERED Directive

► Purpose:

- The ORDERED directive specifies that iterations of the enclosed loop will be executed in the same order as if they were executed on a serial processor.

► Format:

Fortran	<pre>!\$OMP ORDERED (block) !\$OMP END ORDERED</pre>
C/C++	<pre>#pragma omp ordered newline structured_block</pre>

► Restrictions:

- An ORDERED directive can only appear in the dynamic extent of the following directives:
 - DO or PARALLEL DO (Fortran)
 - for or parallel for (C/C++)
- Only one thread is allowed in an ordered section at any time
- It is illegal to branch into or out of an ORDERED block.
- An iteration of a loop must not execute the same ORDERED directive more than once, and it must not execute more than one ORDERED directive.
- A loop which contains an ORDERED directive, must be a loop with an ORDERED clause.

OpenMP Directives

THREADPRIVATE Directive

► Purpose:


- The THREADPRIVATE directive is used to make global file scope variables (C/C++) or common blocks (Fortran) local and persistent to a thread through the execution of multiple parallel regions.

► Format:

Fortran	<code>!\$OMP THREADPRIVATE (/cb/, ...)</code> <i>cb is the name of a common block</i>
C/C++	<code>#pragma omp threadprivate (list)</code>

Notes:

- The directive must appear after the declaration of listed variables/common blocks. Each thread then gets its own copy of the variable/common block, so data written by one thread is not visible to other threads. For example:

 **Fortran - THREADPRIVATE Directive Example**

```

PROGRAM THREADPRIV

INTEGER ALPHA(10), BETA(10), I
COMMON /A/ ALPHA

!$OMP THREADPRIVATE(/A/)

C      First parallel region
!$OMP PARALLEL PRIVATE(BETA, I)
DO I=1,10
ALPHA(I) = I
BETA(I) = I
END DO
!$OMP END PARALLEL

C      Second parallel region
!$OMP PARALLEL
PRINT *, 'ALPHA(3)=',ALPHA(3), ' BETA(3)=',BETA(3)
!$OMP END PARALLEL

END

```



C/C++ - threadprivate Directive Example

```

int alpha[10], beta[10], i;
#pragma omp threadprivate(alpha)

main () {

/* First parallel region */
#pragma omp parallel private(i,beta)
  for (i=0; i < 10; i++)
    alpha[i] = beta[i] = i;

/* Second parallel region */
#pragma omp parallel
  printf("alpha[3]= %d and beta[3]= %d\n",alpha[3],beta[3]);

}

```

- On first entry to a parallel region, data in THREADPRIVATE variables and common blocks should be assumed undefined, unless a COPYIN clause is specified in the PARALLEL directive
- THREADPRIVATE variables differ from PRIVATE variables (discussed later) because they are able to persist between different parallel sections of a code.

► Restrictions:

- Data in THREADPRIVATE objects is guaranteed to persist only if the dynamic threads mechanism is "turned off" and the number of threads in different parallel regions remains constant. The default setting of dynamic threads is undefined.
- The THREADPRIVATE directive must appear after every declaration of a thread private variable/common block.
- Fortran: only named common blocks can be made THREADPRIVATE.

OpenMP Directives

Data Scope Attribute Clauses

- An important consideration for OpenMP programming is the understanding and use of data scoping
- Because OpenMP is based upon the shared memory programming model, most variables are shared by default

- Global variables include:
 - Fortran: COMMON blocks, SAVE variables, MODULE variables
 - C: File scope variables, static
 - Private variables include:
 - Loop index variables
 - Stack variables in subroutines called from parallel regions
 - Fortran: Automatic variables within a statement block
 - The OpenMP Data Scope Attribute Clauses are used to explicitly define how variables should be scoped. They include:
 - PRIVATE
 - FIRSTPRIVATE
 - LASTPRIVATE
 - SHARED
 - DEFAULT
 - REDUCTION
 - COPYIN
 - Data Scope Attribute Clauses are used in conjunction with several directives (PARALLEL, DO/for, and SECTIONS) to control the scoping of enclosed variables.
 - These constructs provide the ability to control the data environment during execution of parallel constructs.
 - They define how and which data variables in the serial section of the program are transferred to the parallel sections of the program (and back)
 - They define which variables will be visible to all threads in the parallel sections and which variables will be privately allocated to all threads.
 - **Note:** Data Scope Attribute Clauses are effective only within their lexical/static extent.
 - See the [Clauses / Directives Summary Table](#) for the associations between directives and clauses.
-

PRIVATE Clause

► Purpose:

- The PRIVATE clause declares variables in its list to be private to each thread.

► Format:

Fortran	PRIVATE (<i>list</i>)
C/C++	private (<i>list</i>)

▶ Notes:

- PRIVATE variables behave as follows:
 - A new object of the same type is declared once for each thread in the team
 - All references to the original object are replaced with references to the new object
 - Variables declared PRIVATE are uninitialized for each thread
- Comparison between PRIVATE and THREADPRIVATE:

	PRIVATE	THREADPRIVATE
Data Item	C/C++: variable Fortran: variable or common block	C/C++: variable Fortran: common block
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 FIRSTPRIVATE	Use COPYIN

▶ Questions:



For the C/C++ and Fortran [THREADPRIVATE example codes](#), what output would you expect for alpha[3] and beta[3]? Why?

SHARED Clause

► Purpose:

- The SHARED clause declares variables in its list to be shared among all threads in the team.

► Format:

Fortran	SHARED (<i>list</i>)
C/C++	shared (<i>list</i>)

► Notes:

- A shared variable exists in only one memory location and all threads can read or write to that address
- It is the programmer's responsibility to ensure that multiple threads properly access SHARED variables (such as via CRITICAL sections)

DEFAULT Clause

► Purpose:

- The DEFAULT clause allows the user to specify a default PRIVATE, SHARED, or NONE scope for all variables in the lexical extent of any parallel region.

► Format:

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

► Notes:

- Specific variables can be exempted from the default using the PRIVATE, SHARED, FIRSTPRIVATE,

LASTPRIVATE, and REDUCTION clauses

- The C/C++ OpenMP specification does not include "private" as a possible default. However, actual implementations may provide this option.

► Restrictions:

- Only one DEFAULT clause can be specified on a PARALLEL directive

FIRSTPRIVATE Clause

► Purpose:

- The FIRSTPRIVATE clause combines the behavior of the PRIVATE clause with automatic initialization of the variables in its list.

► Format:

Fortran	<code>FIRSTPRIVATE (list)</code>
C/C++	<code>firstprivate (list)</code>

► Notes:

- Listed variables are initialized according to the value of their original objects prior to entry into the parallel or work-sharing construct.

LASTPRIVATE Clause

► Purpose:

- The LASTPRIVATE clause combines the behavior of the PRIVATE clause with a copy from the last loop iteration or section to the original variable object.

► Format:

Fortran	LASTPRIVATE (<i>list</i>)
C/C++	lastprivate (<i>list</i>)

► Notes:

- The value copied back into the original variable object is obtained from the last (sequentially) iteration or section of the enclosing construct.

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

COPYIN Clause

► Purpose:

- The COPYIN clause provides a means for assigning the same value to THREADPRIVATE variables for all threads in the team.

► Format:

Fortran	COPYIN (<i>list</i>)
C/C++	copyin (<i>list</i>)

► Notes:

- List contains the names of variables to copy. In Fortran, the list can contain both the names of common blocks and named variables.
- The master thread variable is used as the copy source. The team threads are initialized with its value upon entry into the parallel construct.

REDUCTION Clause

► Purpose:

- The REDUCTION clause performs a reduction on the variables that appear in its list.
- A private copy for each list variable is created for each thread. At the end of the 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.

► Format:

Fortran	<code>REDUCTION (operator/intrinsic: list)</code>
C/C++	<code>reduction (operator: list)</code>

► Example: REDUCTION - 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 construct, all threads will add their values of "result" to update the master thread's global copy.



Fortran - REDUCTION Clause Example

```

PROGRAM DOT_PRODUCT

INTEGER N, CHUNKSIZE, CHUNK, I
PARAMETER (N=100)
PARAMETER (CHUNKSIZE=10)
REAL A(N), B(N), RESULT

!   Some initializations
DO I = 1, N
  A(I) = I * 1.0
  B(I) = I * 2.0
ENDDO
RESULT= 0.0
CHUNK = CHUNKSIZE

!$OMP PARALLEL DO

```

```

!$OMP& DEFAULT(SHARED) PRIVATE(I)
!$OMP& SCHEDULE(STATIC,CHUNK)
!$OMP& REDUCTION(+:RESULT)

DO I = 1, N
  RESULT = RESULT + (A(I) * B(I))
ENDDO

!$OMP END DO NOWAIT

PRINT *, 'Final Result= ', RESULT
END

```

C / C++ - reduction Clause Example

```

#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);

}

```

Restrictions:

- Variables in the list must be named scalar variables. They can not be array or structure type variables. 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 following forms:

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

OpenMP Directives

Clauses / Directives Summary

- The table below summarizes which clauses are accepted by which OpenMP directives.

Clause	Directive					
	PARALLEL	DO/for	SECTIONS	SINGLE	PARALLEL DO/for	PARALLEL SECTIONS
IF	●				●	●
PRIVATE	●	●	●	●	●	●
SHARED	●	●			●	●
DEFAULT	●				●	●
FIRSTPRIVATE	●	●	●	●	●	●
LASTPRIVATE		●	●		●	●

REDUCTION	●	●	●		●	●
COPYIN	●				●	●
SCHEDULE		●			●	
ORDERED		●			●	
NOWAIT		●	●	●		

- The following OpenMP directives do not accept clauses:
 - MASTER
 - CRITICAL
 - BARRIER
 - ATOMIC
 - FLUSH
 - ORDERED
 - THREADPRIVATE
- Implementations may (and do) differ from the standard in which clauses are supported by each directive.

OpenMP Directives

Directive Binding and Nesting Rules



This section is provided mainly as a quick reference on rules which govern OpenMP directives and binding. Users should consult their implementation documentation and the OpenMP standard for other rules and restrictions.

- Unless indicated otherwise, rules apply to both Fortran and C/C++ OpenMP implementations.
- Note: the Fortran API also defines a number of Data Environment rules. Those have not been reproduced here.

▶ Directive Binding:

- The DO/for, SECTIONS, SINGLE, MASTER and BARRIER directives bind to the dynamically enclosing PARALLEL, if one exists. If no parallel region is currently being executed, the directives have no effect.
- The ORDERED directive binds to the dynamically enclosing DO/for.
- The ATOMIC directive enforces exclusive access with respect to ATOMIC directives in all threads, not just the current team.

- The CRITICAL directive enforces exclusive access with respect to CRITICAL directives in all threads, not just the current team.
- A directive can never bind to any directive outside the closest enclosing PARALLEL.

▶ Directive Nesting:

- A PARALLEL directive dynamically inside another PARALLEL directive logically establishes a new team, which is composed of only the current thread unless nested parallelism is enabled.
- DO/for, SECTIONS, and SINGLE directives that bind to the same PARALLEL are not allowed to be nested inside of each other.
- DO/for, SECTIONS, and SINGLE directives are not permitted in the dynamic extent of CRITICAL, ORDERED and MASTER regions.
- CRITICAL directives with the same name are not permitted to be nested inside of each other.
- BARRIER directives are not permitted in the dynamic extent of DO/for, ORDERED, SECTIONS, SINGLE, MASTER and CRITICAL regions.
- MASTER directives are not permitted in the dynamic extent of DO/for, SECTIONS and SINGLE directives.
- ORDERED directives are not permitted in the dynamic extent of CRITICAL regions.
- Any directive that is permitted when executed dynamically inside a PARALLEL region is also legal when executed outside a parallel region. When executed dynamically outside a user-specified parallel region, the directive is executed with respect to a team composed of only the master thread.

Run-Time Library Routines

- The OpenMP standard defines an API for library calls that perform a variety of functions:
 - Query the number of threads/processors, set number of threads to use
 - General purpose locking routines (semaphores)
 - Set execution environment functions: nested parallelism, dynamic adjustment of threads.
- For C/C++, it may be necessary to specify the include file "omp.h".
- For the Lock routines/functions:

- The lock variable must be accessed only through the locking routines
 - For Fortran, the lock variable should be of type integer and of a kind 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.
- Implementation notes:



Current OpenMP implementations for the SP (IBM and KAI) do not implement nested parallelism routines. KAI does implement dynamic threads library routines.

OMP_SET_NUM_THREADS

► Purpose:

- Sets the number of threads that will be used in the next parallel region.

► Format:

Fortran	<code>SUBROUTINE OMP_SET_NUM_THREADS(<i>scalar_integer_expression</i>)</code>
C/C++	<code>void omp_set_num_threads(int num_threads)</code>

► Notes & Restrictions:

- The dynamic threads mechanism modifies the effect of this routine.
 - Enabled: specifies the maximum number of threads that can be used for any parallel region by the dynamic threads mechanism.
 - Disabled: specifies exact number of threads to use until next call to this routine.
- This routine can only be called from the serial portions of the code
- This call has precedence over the `OMP_NUM_THREADS` environment variable

OMP_GET_NUM_THREADS

► Purpose:

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

► Format:

Fortran	<code>INTEGER FUNCTION OMP_GET_NUM_THREADS()</code>
C/C++	<code>int omp_get_num_threads(void)</code>

► Notes & Restrictions:

- 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.

OMP_GET_MAX_THREADS

► Purpose:

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

Fortran	<code>INTEGER FUNCTION OMP_GET_MAX_THREADS()</code>
C/C++	<code>int omp_get_max_threads(void)</code>

► Notes & Restrictions:

- 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.

OMP_GET_THREAD_NUM

► Purpose:

- 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

► Format:

Fortran	INTEGER FUNCTION OMP_GET_THREAD_NUM()
C/C++	int omp_get_thread_num(void)

► Notes & Restrictions:

- If called from a nested parallel region, or a serial region, this function will return 0.

► Examples:

- Example 1 is the correct way to determine the number of threads in a parallel region.
- Example 2 is incorrect - the TID variable must be PRIVATE
- Example 3 is incorrect - the OMP_GET_THREAD_NUM call is outside the parallel region



Fortran - determining the number of threads in a parallel region

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

```

OMP_GET_NUM_PROCS

► Purpose:

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

► Format:

Fortran	INTEGER FUNCTION OMP_GET_NUM_PROCS()
C/C++	int omp_get_num_procs(void)

OMP_IN_PARALLEL

► Purpose:

- May be called to determine if the section of code which is executing is parallel or not.

► Format:

Fortran	<code>LOGICAL FUNCTION OMP_IN_PARALLEL()</code>
C/C++	<code>int omp_in_parallel(void)</code>

► Notes & Restrictions:

- For Fortran, this function returns `.TRUE.` if it is called from the dynamic extent of a region executing in parallel, and `.FALSE.` otherwise. For C/C++, it will return a non-zero integer if parallel, and zero otherwise.

OMP_SET_DYNAMIC

► Purpose:

- Enables or disables dynamic adjustment (by the run time system) of the number of threads available for execution of parallel regions.

► Format:

Fortran	<code>SUBROUTINE OMP_SET_DYNAMIC(<i>scalar_logical_expression</i>)</code>
C/C++	<code>void omp_set_dynamic(int <i>dynamic_threads</i>)</code>

► Notes & Restrictions:

- For Fortran, if called with `.TRUE.` then the number of threads available for subsequent parallel regions can be adjusted automatically by the run-time environment. If called with `.FALSE.`, dynamic adjustment is disabled.
- For C/C++, if `dynamic_threads` evaluates to non-zero, then the mechanism is enabled, otherwise it is disabled.
- The `OMP_SET_DYNAMIC` subroutine has precedence over the `OMP_DYNAMIC` environment variable.
- The default setting is implementation dependent.
- Must be called from a serial section of the program.

OMP_GET_DYNAMIC

► Purpose:

- Used to determine if dynamic thread adjustment is enabled or not.

► Format:

Fortran	<code>LOGICAL FUNCTION OMP_GET_DYNAMIC()</code>
C/C++	<code>int omp_get_dynamic(void)</code>

► Notes & Restrictions:

- For Fortran, this function returns `.TRUE.` if dynamic thread adjustment is enabled, and `.FALSE.` otherwise.
- For C/C++, non-zero will be returned if dynamic thread adjustment is enabled, and zero otherwise.

OMP_SET_NESTED

► Purpose:

- Used to enable or disable nested parallelism.

► Format:

Fortran	<code>SUBROUTINE OMP_SET_NESTED(scalar_logical_expression)</code>
C/C++	<code>void omp_set_nested(int nested)</code>

► Notes & Restrictions:

- For Fortran, calling this function with `.FALSE.` will disable nested parallelism, and calling with `.TRUE.` will enable it.
- For C/C++, if nested evaluates to non-zero, nested parallelism is enabled; otherwise it is disabled.

- The default is for nested parallelism to be disabled.
 - This call has precedence over the OMP_NESTED environment variable
-

OMP_GET_NESTED

► Purpose:

- Used to determine if nested parallelism is enabled or not.

► Format:

Fortran	LOGICAL FUNCTION OMP_GET_NESTED
C/C++	void omp_get_nested

► Notes & Restrictions:

- For Fortran, this function returns `.TRUE.` if nested parallelism is enabled, and `.FALSE.` otherwise.
 - For C/C++, non-zero will be returned if nested parallelism is enabled, and zero otherwise.
-

OMP_INIT_LOCK

► Purpose:

- This subroutine initializes a lock associated with the lock variable.

► Format:

Fortran	SUBROUTINE OMP_INIT_LOCK(var) SUBROUTINE OMP_INIT_NEST_LOCK(var)
C/C++	void omp_init_lock(omp_lock_t *lock) void omp_init_nest_lock(omp_nest_lock_t *lock)

▶ Notes & Restrictions:

- The initial state is unlocked
-

OMP_DESTROY_LOCK

▶ Purpose:

- This subroutine disassociates the given lock variable from any locks.

▶ Format:

Fortran	<pre>SUBROUTINE OMP_DESTROY_LOCK(var) SUBROUTINE OMP_DESTROY_NEST_LOCK(var)</pre>
C/C++	<pre>void omp_destroy_lock(omp_lock_t *lock) void omp_destroy_nest__lock(omp_nest_lock_t *lock)</pre>

▶ Notes & Restrictions:

- It is illegal to call this routine with a lock variable that is not initialized.
-

OMP_SET_LOCK

▶ Purpose:

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

▶ Format:

Fortran	<pre>SUBROUTINE OMP_SET_LOCK(var) SUBROUTINE OMP_SET_NEST_LOCK(var)</pre>
C/C++	<pre>void omp_set_lock(omp_lock_t *lock) void omp_set_nest__lock(omp_nest_lock_t *lock)</pre>

▶ Notes & Restrictions:

- It is illegal to call this routine with a lock variable that is not initialized.
-

OMP_UNSET_LOCK

▶ Purpose:

- This subroutine releases the lock from the executing subroutine.

▶ Format:

Fortran	<pre>SUBROUTINE OMP_UNSET_LOCK(var) SUBROUTINE OMP_UNSET_NEST_LOCK(var)</pre>
C/C++	<pre>void omp_unset_lock(omp_lock_t *lock) void omp_unset_nest__lock(omp_nest_lock_t *lock)</pre>

▶ Notes & Restrictions:

- It is illegal to call this routine with a lock variable that is not initialized.
-

OMP_TEST_LOCK

▶ Purpose:

- This subroutine attempts to set a lock, but does not block if the lock is unavailable.

▶ Format:

Fortran	<pre>SUBROUTINE OMP_TEST_LOCK(var) SUBROUTINE OMP_TEST_NEST_LOCK(var)</pre>
C/C++	<pre>void omp_test_lock(omp_lock_t *lock) void omp_test_nest__lock(omp_nest_lock_t *lock)</pre>

▶ Notes & Restrictions:

- For Fortran, `.TRUE.` is returned if the lock was set successfully, otherwise `.FALSE.` is returned.
- For C/C++, non-zero is returned if the lock was set successfully, otherwise zero is returned.
- It is illegal to call this routine with a lock variable that is not initialized.

Environment Variables

- OpenMP provides four environment variables for controlling the execution of parallel code.
- All environment variable names are uppercase. The values assigned to them are not case sensitive.

OMP_SCHEDULE

Applies only to `DO`, `PARALLEL DO` (Fortran) and `for`, `parallel for` (C/C++) directives which have their schedule clause set to `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

Sets the maximum number of threads to use during execution. For example:

```
setenv OMP_NUM_THREADS 8
```

OMP_DYNAMIC

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

```
setenv OMP_DYNAMIC TRUE
```

OMP_NESTED

Enables or disables nested parallelism. Valid values are `TRUE` or `FALSE`. For example:

```
setenv OMP_NESTED TRUE
```

- Implementation notes:



The current IBM OpenMP implementations (IBM and KAI) for the SP do not implement nested parallelism. The KAI implementation does implement dynamic threads.

LLNL Specific Information and Recommendations

▶ LC OpenMP Implementations:

- OpenMP is fully supported in the native compilers of all IBM, Intel and Compaq systems. Additionally, the KAI Guide products, which fully support OpenMP, are available on LC production machines.
- LC maintains different versions of compilers. For the most recent information, please see: www.llnl.gov/asci/platforms/bluepac/CompsAvails.html

▶ Compiling:

- For IBM systems, use the flag `-qsmp=omp`
- For Intel systems, use the flag `-openmp`
- For Compaq systems, use the flag `-omp`
- For KAI on any system, there is no special OpenMP flag required.

▶ Documentation:

- IBM compiler documentation:
Vendor: www-4.ibm.com/software/ad/fortran and www-4.ibm.com/software/ad/caix
Locally: see the `/usr/local/doc/xlf*` and `/usr/local/doc/xlc*` files
- Intel compiler documentation:
Vendor: www.intel.com/software/products/compilers/
Locally: see the `/usr/local/doc/ia32_ref` and `/usr/local/doc/intel_compilers` files
- Compaq compiler documentation:
Vendor:
C (Developer's Toolkit): <http://h30097.www3.hp.com/dtk>
Fortran: <http://h18009.www1.hp.com/fortran/docs>
C++: <http://h30097.www3.hp.com/cplus>
Locally: see the relevant files in `/usr/local/docs`
- KAI C/C++ and Fortran compilers documentation:
Vendor: <http://www.kai.com/>

Locally: see the /usr/local/doc/Guide* files

References and More Information

- The OpenMP web site.
<http://www.openmp.org/>
- "OpenMP C and C++ Application Program Interface, Version 1.0". OpenMP Architecture Review Board. October 1998.
- "OpenMP Fortran Application Program Interface, Version 1.0". OpenMP Architecture Review Board. October 1997.
- "OpenMP". Workshop presentation. John Engle, Lawrence Livermore National Laboratory. October, 1998.
- "OpenMP". Alliance 98 Tutorial. Faisal Saied, NCSA.
- "Introduction to OpenMP Using the KAP/PRO Toolset". Kuck & Associates, Inc.
- "Guide Reference Manual (C/C++ Edition, Version 3.6)". Kuck & Associates, Inc.
- "Guide Reference Manual (Fortran Edition, Version 3.6)". Kuck & Associates, Inc.

