Chapter 3:

OPEN MPI

Dr. Khem Poudel

Abstract

OpenMP is an Application Program Interface (API), jointly defined by a group of major computer hardware and software vendors. OpenMP provides a portable, scalable model for developers of shared memory parallel applications. The API supports C/C++ and Fortran on a wide variety of architectures. This tutorial covers most of the major features of OpenMP 3.1, (with some features of 4.5) including its various constructs and directives for specifying parallel regions, work sharing, synchronization and data environment. Runtime library functions and environment variables are also covered. This tutorial includes both C and Fortran example codes and a lab exercise.

Level/Prerequisites: This tutorial is one of the tutorials in the 3-day "Using LLNL's Supercomputers" workshop. It is geared to those who are new to parallel programming with OpenMP. Basic understanding of parallel programming in C/C++ or Fortran is required. For those who are unfamiliar with Parallel Programming in general, the material covered in EC3500: Introduction to Parallel Computing would be helpful.

Introduction

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
- An abbreviation for:
  - Short version: **Open Multi-Processing**
  - Long version: **Open** specifications for **Multi-Processing** via collaborative work between interested parties from the hardware and software industry, government and academia.

OpenMP is not:

- Necessarily implemented identically by all vendors
- Guaranteed to make the most efficient use of shared memory
- Required to check for data dependencies, data conflicts, race conditions, or deadlocks
- Required to check for code sequences that cause a program to be classified as non-conforming
- Designed to guarantee that input or output to the same file is synchronous when executed in parallel. The programmer is responsible for synchronizing input and output.

Goals of OpenMP:

- **Standardization**:
  - Provide a standard among a variety of shared memory architectures/platforms
  - Jointly defined and endorsed by a group of major computer hardware and software vendors
- **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.
  - This goal is becoming less meaningful with each new release, apparently.
- **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**:
  - The API is specified for C/C++ and Fortran
  - Public forum for API and membership
  - Most major platforms have been implemented including Unix/Linux platforms and Windows

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.

However, not long after this, newer shared memory machine architectures started to become prevalent, and interest resumed.

The OpenMP standard specification started in the spring of 1997, taking over where ANSI X3H5 had left off.

Led by the OpenMP Architecture Review Board (ARB). Original ARB members and contributors are shown below. (Disclaimer: all partner names derived from the OpenMP web site)

APR Members	Endorsing Application Developers	Endorsing Software Vendors
Compaq / Digital	ADINA R&D, Inc.	Absoft Corporation
Hewlett-Packard Company	ANSYS, Inc.	Edinburgh Portable Compilers
Intel Corporation	Dash Associates	GENIAS Software GmBH
International Business Machines (IBM)	Fluent, Inc.	Myrias Computer Technologies, Inc.
Kuck & Associates, Inc. (KAI)	ILOG CPLEX Division	The Portland Group, Inc. (PGI)
Silicon Graphics, Inc.	Livermore Software Technology Corporation (LSTC)	
Sun Microsystems, Inc.	MECALOG SARL	
U.S. Department of Energy ASCI program	Oxford Molecular Group PLC	
	The Numerical Algorithms Group Ltd.(NAG)	

For more news and membership information about the OpenMP ARB, visit: openmp.org/wp/about-openmp.

Release History

OpenMP continues to evolve, with new constructs and features being added over time.

Initially, the API specifications were released separately for C and Fortran. Since 2005, they have been released together.

The table below chronicles the OpenMP API release history:

Month/Year	Version
Oct 1997	Fortran 1.0
Oct 1998	C/C++ 1.0
Nov 1999	Fortran 1.1
Nov 2000	Fortran 2.0
Mar 2002	C/C++ 2.0
May 2005	OpenMP 2.5
May 2008	OpenMP 3.0
Jul 2011	OpenMP 3.1
Jul 2013	OpenMP 4.0
Nov 2018	OpenMP 5.0
Nov 2020	OpenMP 5.1
Nov 2021	OpenMP 5.2

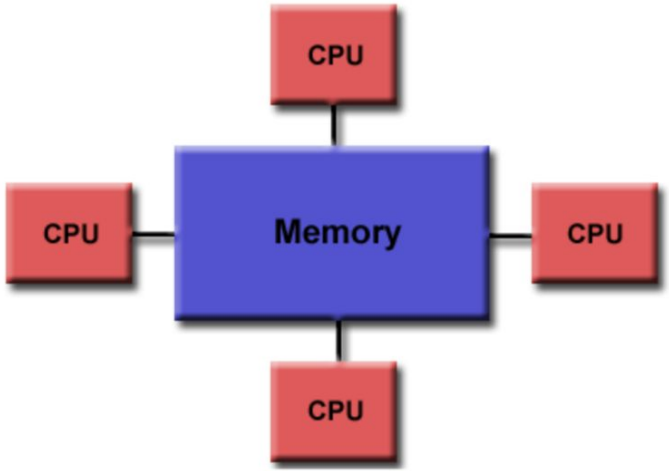
Note: The remainder of this tutorial, unless otherwise indicated, refers to OpenMP version 3.1. Syntax and features new with OpenMP 4.x and OpenMP 5.x are not currently covered.

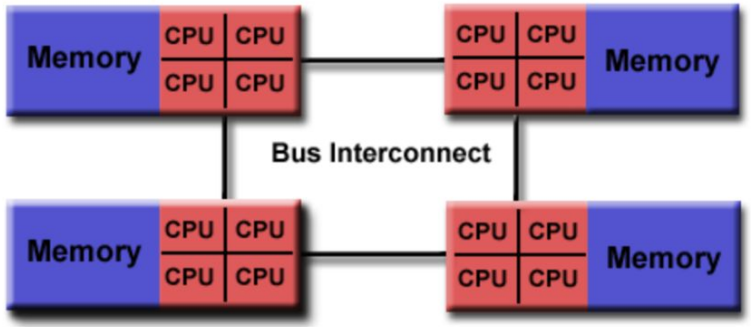
OpenMP Programming Model

Openmp has - Memory Model, Execution Model:

Shared Memory Model:

OpenMP is designed for multi-processor/core, shared memory machines. The underlying architecture can be shared memory UMA or NUMA.





Openmp Execution Model:

Thread Based Parallelism:

OpenMP programs accomplish parallelism exclusively through the use of threads.

A thread of execution is the smallest unit of processing that can be scheduled by an operating system. The idea of a subroutine that can be scheduled to run autonomously might help explain what a thread is.

Threads exist within the resources of a single process. Without the process, they cease to exist.

Typically, the number of threads match the number of machine processors/cores. However, the actual use of threads is up to the application.

Explicit Parallelism:

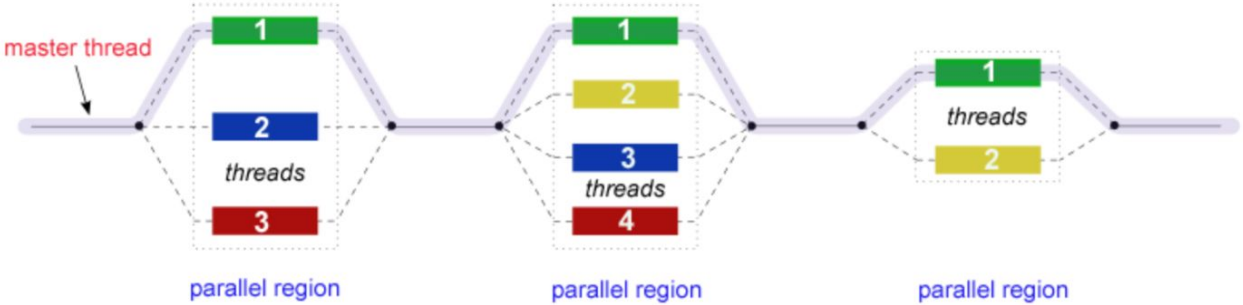
OpenMP is an explicit (not automatic) programming model, offering the programmer full control over parallelization.

Parallelization can be as simple as taking a serial program and inserting compiler directives....

Or as complex as inserting subroutines to set multiple levels of parallelism, locks and even nested locks.

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.

The number of parallel regions and the threads that comprise them are arbitrary.

Compiler Directive Based:

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

Nested Parallelism:

The API provides for the placement of parallel regions inside other parallel regions.

Implementations may or may not support this feature.

Dynamic Threads:

The API provides for the runtime environment to dynamically alter the number of threads used to execute parallel regions. Intended to promote more efficient use of resources, if possible.

Implementations may or may not support this feature.

I/O:

OpenMP specifies nothing about parallel I/O.

It is entirely up to the programmer to ensure that I/O is conducted correctly within the context of a multi-threaded program.

Execution Model and Memory Model Interactions:

Single-Program-Multiple-Data (SPMD) is underlying programming paradigm - all threads have potential to execute the same program code, however, each thread may access modify different data and traverse different execution paths.

OpenMP provides a “relaxed-consistency” and “temporary” view of thread memory - threads have equal access to shared memory where variables can be retrieved/stored. Each thread also has its own temporary copies of variables that may be modified independent from variables in memory.

When it is critical that all threads have a consistent view of a shared variable, the programmer (or compiler) is responsible for insuring that the variable is updated by all threads as needed, via an explict action - ie., FLUSH, or implicitly (via compiler recognition of program flow leaving a parallel regions).

OpenMP Programming:

- Method to start up paralell threads
- Method to discover how many threads are running
- Need way to uniquely identify threads
- Method to join threads for serial execution
- Method to synchronize threads
- Ensure consistent view of data items when necesasry
- Required to check for data dependencies, data conflicts, race conditions, or deadlocks

OpenMP API Overview

Three Components:

The OpenMP API is comprised of three distinct components. As of version 4.0:

- Compiler Directives (44)
- Runtime Library Routines (35)
- Environment Variables (13)

The application developer decides how to employ these components. In the simplest case, only a few of them are needed.

Implementations differ in their support of all API components. For example, an implementation may state that it supports nested parallelism, but the API makes it clear that may be limited to a single thread - the master thread. Not exactly what the developer might expect?

Compiler Directives:

Compiler directives appear as comments in your source code and are ignored by compilers unless you tell them otherwise - usually by specifying the appropriate compiler flag, as discussed in the Compiling section later.

OpenMP compiler directives are used for various purposes:

- Spawning a parallel region
- Dividing blocks of code among threads
- Distributing loop iterations between threads
- Serializing sections of code
- Synchronization of work among threads

Compiler directives have the following syntax:

```
sentinel      directive-name      [clause, ...]
```

For example:

Fortran	**!$OMP PARALLEL DEFAULT(SHARED) PRIVATE(BETA,PI)**
C/C++	**#pragma omp parallel default(shared) private(beta,pi)**

Compiler directives are covered in detail later.

Run-time Library Routines:

The OpenMP API includes an ever-growing number of run-time library routines.

These routines are used for a variety of purposes:

- Setting and querying the number of threads
- Querying a thread's unique identifier (thread ID), a thread's ancestor's identifier, the thread team size
- Setting and querying the dynamic threads feature
- Querying if in a parallel region, and at what level
- Setting and querying nested parallelism
- Setting, initializing and terminating locks and nested locks
- Querying wall clock time and resolution

For C/C++, all of the run-time library routines are actual subroutines. For Fortran, some are actually functions, and some are subroutines. For example:

Fortran	**INTEGER FUNCTION OMP_GET_NUM_THREADS()**
C/C++	**#include <omp.h>** **int omp_get_num_threads(void)**

Note that for C/C++, you usually need to include the `<omp.h>` header file.

Fortran routines are not case sensitive, but C/C++ routines are.

The run-time library routines are briefly discussed as an overview in the Run-Time Library Routines section, and in more detail in Appendix A.

Environment Variables:

OpenMP provides several environment variables for controlling the execution of parallel code at run-time.

These environment variables can be used to control such things as:

- Setting the number of threads
- Specifying how loop interations are divided
- Binding threads to processors
- Enabling/disabling nested parallelism; setting the maximum levels of nested parallelism
- Enabling/disabling dynamic threads
- Setting thread stack size
- Setting thread wait policy

Setting OpenMP environment variables is done the same way you set any other environment variables, and depends upon which shell you use. For example:

csh/tcsh	**setenv OMP_NUM_THREADS 8**
sh/bash	**export OMP_NUM_THREADS=8**

OpenMP environment variables are discussed in the Environment Variables section later.

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
                  .
      Other OpenMP directives
                  .
      Run-time Library calls
                  .
      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
                  .
Other OpenMP directives
                  .
Run-time Library calls
                  .
All threads join master thread and disband

 }

Resume serial code
      .
      .
      .

 }
```

Compiling OpenMP Programs

LC OpenMP Implementations:

As of June 2022, the documentation for LC's default compilers claims the following OpenMP support:

Compiler	Version	Supports
Intel C/C++, Fortran	intel/19.0.4	OpenMP 5.0
GNU C/C++, Fortran	4.9	OpenMP 4.5
PGI C/C++, Fortran	pgi/22.1	OpenMP 5.0
IBM Coral Systems C/C++	xl/2021.09.22	OpenMP 4.5
IBM Coral Systems Fortran	xl/2021.09.22	OpenMP 4.5
IBM Coral Systems GNU C/C++, Fortran	xl/2021.09.22	OpenMP 4.5

OpenMP 4.0 Support:

According to vendor documentation, OpenMP 4.0 is supported beginning with the following compiler versions:

- GNU: 4.9 for C/C++ and 4.9.1 for Fortran
- Intel: 14.0 has “some” support; 15.0 supports “most features”
- PGI: not currently available
- IBM Coral Systems: 4.0, 4.5 - depending upon compiler

Use the command `use -l compilers` to view compiler packages by version. You can also view compiler version information at: https://computing.llnl.gov/code/compilers.html

Compiling:

All of LC's compilers require you to use the appropriate compiler flag to "turn on" OpenMP compilations. The table below shows what to use for each compiler.

Compiler / Platform	Compiler	Flag
Intel Linux Opteron/Xeon	icc icpc ifort	-qopenmp
PGI Linux Opteron/Xeon	pgcc pgCC pgf77 pgf90	-mp
GNU Linux Opteron/Xeon IBM Blue Gene	gcc g++ g77 gfortran	-fopenmp
IBM Coral Systems	xlc_r, cc_r xlC_r, xlc++_r xlc89_r xlc99_r xlf_r xlf90_r xlf95_r xlf2003_r *Be sure to use a thread-safe compiler - its name ends with _r	-qsmp=omp

Compiler Documentation:

- IBM Coral Systems: https://hpc.llnl.gov/documentation/tutorials/using-lc-s-sierra-systems
- Intel: www.intel.com/software/products/compilers/
- PGI: www.pgroup.com
- GNU: www.gnu.org
- All: See the relevant man pages and any files that might relate in `/usr/local/docs`

OpenMP Directives: Fortran Directive Format

Format: (case insensitive)

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

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.
- Only one directive-name may be specified per 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 shown below. The "end" directive is optional but advised for readability.

```
!$OMP   <directive >

    [ structured block of code ]

!$OMP end   <directive >
```

OpenMP Directives: C/C++ Directive Format

Format:

#pragma omp	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. Precedes the structured block which is enclosed by this directive.

Example:

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

General Rules:

- Case sensitive.
- Directives follow conventions of the C/C++ standards for compiler directives.
- Only one directive-name may be specified per directive.
- Each directive applies to at most one succeeding statement, which must be a structured block.
- Long directive lines can be “continued” on succeeding lines by escaping the newline character with a backslash (“”) at the end of a directive line.

```
#pragma omp <directive>

    [ structured block of code ]
```

OpenMP Directives: Directive Scoping

Do we do this now...or do it later? Oh well, let's get it over with early...

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.

An orphaned directive can 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:

<table>
<tr>
<td>

```
PROGRAM TEST
  ...
!$OMP PARALLEL
  ...
!$OMP DO
  DO I=...
    ...
    CALL SUB1
    ...
  ENDDO
  ...
  CALL SUB2
  ...
!$OMP END PARALLEL
```

</td>
<td>

```
  SUBROUTINE SUB1
  ...
!$OMP CRITICAL
  ...
!$OMP END CRITICAL
  END

  SUBROUTINE SUB2
  ...
!$OMP SECTIONS
  ...
!$OMP END SECTIONS
  ...
  END
```

</td>
</tr>
<tr>
<td>STATIC EXTENT
The DO directive occurs within an enclosing parallel region</td>
<td>ORPHANED DIRECTIVES
The CRITICAL and SECTIONS directives occur outside an enclosing parallel region</td>
</tr>
<tr>
<td colspan="2">DYNAMIC EXTENT
The CRITICAL and SECTIONS directives occur within the dynamic extent of the DO and PARALLEL directives.</td>
</tr>
</table>

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.

OpenMP Directives: 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

```
!$OMP PARALLEL [clause ...]
               IF (scalar_logical_expression)
               PRIVATE (list)
               SHARED (list)
               DEFAULT (PRIVATE | FIRSTPRIVATE | SHARED | NONE)
               FIRSTPRIVATE (list)
               REDUCTION (operator: list)
               COPYIN (list)
               NUM_THREADS (scalar-integer-expression)

   block

!$OMP END PARALLEL
```

C/C++

```
#pragma omp parallel [clause ...]  newline
                     if (scalar_expression)
                     private (list)
                     shared (list)
                     default (shared | none)
                     firstprivate (list)
                     reduction (operator: list)
                     copyin (list)
                     num_threads (integer-expression)


   structured_block
```

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.

If any thread terminates within a parallel region, all threads in the team will terminate, and the work done up until that point is undefined.

How many threads?

The number of threads in a parallel region is determined by the following factors, in order of precedence:

1. Evaluation of the `IF` clause
2. Setting of the `NUM_THREADS` clause
3. Use of the `omp_set_num_threads()` library function
4. Setting of the `OMP_NUM_THREADS` environment variable
5. Implementation default - usually the number of CPUs on a node, though it could be dynamic.

Threads are numbered from `0` (master thread) to `N-1`.

Dynamic Threads

Use the `omp_get_dynamic()` library function to determine if dynamic threads are enabled.

If supported, the two methods available for enabling dynamic threads are:

1. The `omp_set_dynamic()` library routine
2. Setting of the `OMP_DYNAMIC` environment variable to TRUE

If not supported, a parallel region nested within another parallel region results in the creation of a new team, consisting of one thread, by default.

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.
- It is illegal to branch (goto) into or out of a parallel region.
- Only a single `IF` clause is permitted.
- Only a single `NUM_THREADS` clause is permitted.
- A program must not depend upon the ordering of the clauses.

Example: Parallel Region

Fortran

```
       PROGRAM HELLO

       INTEGER NTHREADS, TID, OMP_GET_NUM_THREADS, OMP_GET_THREAD_NUM

      # Fork a team of threads with each thread having a private TID variable
!$OMP PARALLEL PRIVATE(TID)

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

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

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

       END
```

C/C++

```
#include <omp.h>

main ()  {

int nthreads, tid;

/* Fork a team of threads with each thread having a private tid variable */
#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 */

}
```

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

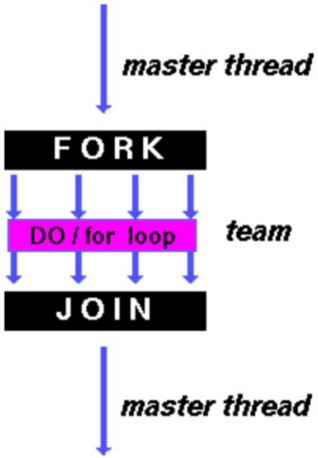
There is no implied barrier upon entry to a work-sharing construct, however there is an implied barrier at the end of a work sharing construct.

Types of Work-Sharing Constructs:

NOTE: The Fortran workshare construct is not shown here, but is discussed later.

DO / for

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

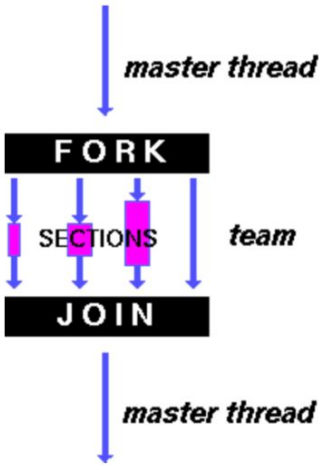


SECTIONS

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

SECTIONS

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

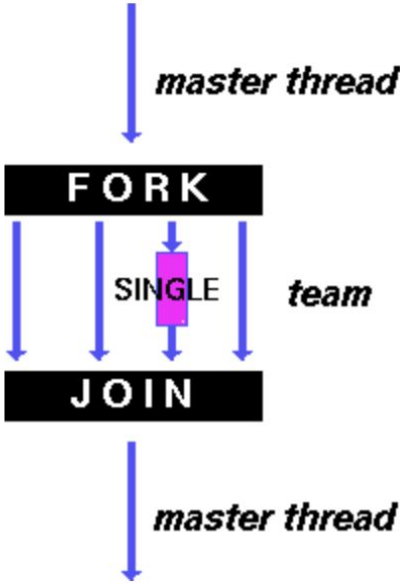


SINGLE

SINGLE serializes a section of code

SINGLE

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:

```
!$OMP DO [clause ...]
         SCHEDULE (type [,chunk])
         ORDERED
         PRIVATE (list)
         FIRSTPRIVATE (list)
         LASTPRIVATE (list)
         SHARED (list)
         REDUCTION (operator | intrinsic : list)
         COLLAPSE (n)

   do_loop

!$OMP END DO  [ NOWAIT ]
```

Clauses:

SCHEDULE:

Describes how iterations of the loop are divided among the threads in the team. The default schedule is implementation dependent. For a discussion on how one type of scheduling may be more optimal than others, see https://forum.openmp.org/viewtopic.php?t=83.

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

Iterations are dynamically assigned to threads in blocks as threads request them until no blocks remain to be assigned. Similar to DYNAMIC except that the block size decreases each time a parcel of work is given to a thread. The size of the initial block is proportional to:

`number_of_iterations / number_of_threads`

Subsequent blocks are proportional to

`number_of_iterations_remaining / number_of_threads`

The chunk parameter defines the minimum block size. 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.

AUTO

The scheduling decision is delegated to the compiler and/or runtime system.

NO WAIT / nowait:

If specified, then threads do not synchronize at the end of the parallel loop.

ORDERED:

Specifies that the iterations of the loop must be executed as they would be in a serial program.

COLLAPSE:

Specifies how many loops in a nested loop 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.

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 (goto) out of a loop associated with a DO/for directive.

The chunk size must be specified as a loop invarient integer expression, as there is no synchronization during its evaluation by different threads.

ORDERED, COLLAPSE and SCHEDULE clauses may appear once each.

See the OpenMP specification document for additional restrictions.

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 may be executed by different threads. It is possible for a thread to execute more than one section if it is quick enough and the implementation permits such.

Format:

Fortran:

```
!$OMP SECTIONS [clause ...]
               PRIVATE (list)
               FIRSTPRIVATE (list)
               LASTPRIVATE (list)
               REDUCTION (operator | intrinsic : list)

!$OMP  SECTION

   block

!$OMP  SECTION

    block

!$OMP END SECTIONS  [ NOWAIT ]
```

C/C++:

```
#pragma omp sections [clause ...]  newline
                     private (list)
                     firstprivate (list)
                     lastprivate (list)
                     reduction (operator: list)
                     nowait
  {

  #pragma omp section   newline

     structured_block

  #pragma omp section   newline

     structured_block

  }
```

Clauses:

There is an implied barrier at the end of a SECTIONS directive, unless the NOWAIT/nowait 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 (goto) into or out of section blocks.

SECTION directives must occur within the lexical extent of an enclosing SECTIONS directive (no orphan SECTIONs).

Example: SECTIONS Directive

Simple program demonstrating that different blocks of work will be done by different threads.

Fortran - SECTIONS Directive Example

```
      PROGRAM VEC_ADD_SECTIONS

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

!     Some initializations
      DO I = 1, N
        A(I) = I * 1.5
        B(I) = I + 22.35
      ENDDO

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

!$OMP SECTIONS

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

!$OMP SECTION
      DO I = 1, N
         D(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], 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 */

}
```

OpenMP Directives: Work-Sharing Constructs: WORKSHARE Directive

Purpose:

- **Fortran only**
- The WORKSHARE directive divides the execution of the enclosed structured block into separate units of work, each of which is executed only once.
- The structured block must consist of only the following:
  - array assignments
  - scalar assignments
  - FORALL statements
  - FORALL constructs
  - WHERE statements
  - WHERE constructs
  - atomic constructs
  - critical constructs
  - parallel constructs

See the OpenMP API documentation for additional information, particularly for what comprises a “unit of work”.

Format:

Fortran:

```
!$OMP WORKSHARE

   structured block

!$OMP END WORKSHARE [ NOWAIT ]
```

Restrictions:

The construct must not contain any user defined function calls unless the function is ELEMENTAL.

Example: WORKSHARE Directive

Simple array and scalar assigments shared by the team of threads. A unit of work would include: * Any scalar assignment * For array assignment statements, the assignment of each element is a unit of work

Fortran:

```
      PROGRAM WORKSHARE

      INTEGER N, I, J
      PARAMETER (N=100)
      REAL AA(N,N), BB(N,N), CC(N,N), DD(N,N), FIRST, LAST

!     Some initializations
      DO I = 1, N
        DO J = 1, N
          AA(J,I) = I * 1.0
          BB(J,I) = J + 1.0
        ENDDO
      ENDDO
 !$OMP PARALLEL SHARED(AA,BB,CC,DD,FIRST,LAST)

!$OMP WORKSHARE
      CC = AA * BB
      DD = AA + BB
      FIRST = CC(1,1) + DD(1,1)
      LAST = CC(N,N) + DD(N,N)
!$OMP END WORKSHARE NOWAIT

!$OMP END PARALLEL

      END
```

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:

```
!$OMP SINGLE [clause ...]
             PRIVATE (list)
             FIRSTPRIVATE (list)

   block

!$OMP END SINGLE [ NOWAIT ]
```

C/C++:

```
#pragma omp single [clause ...]  newline
                   private (list)
                   firstprivate (list)
                   nowait

     structured_block
```

OpenMP Directives: Combined Parallel Work-Sharing Constructs

OpenMP provides three directives that are merely conveniences:

- PARALLEL DO / parallel for
- PARALLEL SECTIONS
- PARALLEL WORKSHARE (fortran only)

For the most part, these directives behave identically to an individual PARALLEL directive being immediately followed by a separate work-sharing directive.

Most of the rules, clauses and restrictions that apply to both directives are in effect. See the OpenMP API for details.

An example using the PARALLEL DO / parallel for combined directive is shown below.

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: TASK Construct

Purpose:

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

The data environment of the task is determined by the data sharing attribute clauses.

Task execution is subject to task scheduling - see the OpenMP 3.1 specification document for details.

Also see the OpenMP 3.1 documentation for the associated taskyield and taskwait directives.

Format:

Fortran:

```
!$OMP TASK [clause ...]
             IF (scalar logical expression)
             FINAL (scalar logical expression)
             UNTIED
             DEFAULT (PRIVATE | FIRSTPRIVATE | SHARED | NONE)
             MERGEABLE
             PRIVATE (list)
             FIRSTPRIVATE (list)
             SHARED (list)

   block

!$OMP END TASK
```

C/C++:

```
#pragma omp task [clause ...]  newline
                   if (scalar expression)
                   final (scalar expression)
                   untied
                   default (shared | none)
                   mergeable
                   private (list)
                   firstprivate (list)
                   shared (list)

     structured_block
```

Clauses and Restrictions:

Please consult the OpenMP 3.1 specifications document for details.

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):

	THREAD 1:	THREAD 2:

High level code	increment(x)
	{
	x = x + 1;
	}
		increment(x)
		{
		x = x + 1;
		}

Assembly	10 LOAD A, (x address) 20 ADD A, 1 30 STORE A, (x address)	10 LOAD A, (x address) 20 ADD A, 1 30 STORE A, (x address)

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 incrementing of x must be synchronized between the two threads to ensure 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

```
!$OMP MASTER

   block

!$OMP END MASTER
```

C/C++

```
#pragma omp master  newline

   structured_block
```

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

```
!$OMP CRITICAL [ name ]

   block

!$OMP END CRITICAL [ name ]
```

C/C++

```
#pragma omp critical [ name ]  newline

   structured_block
```

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.

Fortran only: The names of critical constructs are global entities of the program. If a name conflicts with any other entity, the behavior of the program is unspecified.

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

main()
{

int x;
x = 0;

#pragma omp parallel shared(x)
  {

  #pragma omp critical
  x = x + 1;

  }  /* end of parallel section */

}
</pre>
```

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  newline
```

Restrictions:

All threads in a team (or none) must execute the BARRIER region.

The sequence of work-sharing regions and barrier regions encountered must be the same for every thread in a team.

OpenMP Directives: Synchronization Constructs: TASKWAIT Directive

Purpose:

- New with OpenMP 3.1
- The TASKWAIT construct specifies a wait on the completion of child tasks generated since the beginning of the current task.

Format:

Fortran

```
!$OMP TASKWAIT
```

C/C++

```
#pragma omp taskwait  newline
```

Restrictions:

Because the taskwait construct does not have a C language statement as part of its syntax, there are some restrictions on its placement within a program. The taskwait directive may be placed only at a point where a base language statement is allowed. The taskwait directive may not be used in place of the statement following an if, while, do, switch, or label. See the OpenMP 3.1 specifications document for details.

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.

There is a fair amount of discussion on this directive within OpenMP circles that you may wish to consult for more information. Some of it is hard to understand. Per the API:

> If the intersection of the flush-sets of two flushes performed by two different threads is non-empty, then the two flushes must be completed as if in some sequential order, seen by all threads.

Say what?

To quote from the openmp.org FAQ:

>

> **Q17:** Is the !$omp flush directive necessary on a cache coherent system?

> *A17: Yes the flush directive is necessary. Look in the OpenMP specifications for examples of it's uses. The directive is necessary to instruct the compiler that the variable must be written to/read from the memory system, i.e. that the variable can not be kept in a local CPU register over the flush "statement" in your code.*

> Cache coherency makes certain that if one CPU executes a read or write instruction from/to memory, then all other CPUs in the system will get the same value from that memory address when they access it. All caches will show a coherent value. However, in the OpenMP standard there must be a way to instruct the compiler to actually insert the read/write machine instruction and not postpone it. Keeping a variable in a register in a loop is very common when producing efficient machine language code for a loop. </i>

Also see the most recent OpenMP specs for details.

Format:

Fortran

```
!$OMP FLUSH  (list)
```

C/C++

```
#pragma omp flush (list)  newline
```

Notes:

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
END PARALLEL	parallel - upon entry and exit
CRITICAL and END CRITICAL	critical - upon entry and exit
END DO	ordered - upon entry and 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.
- Threads will need to wait before executing their chunk of iterations if previous iterations haven't completed yet.
- Used within a DO / for loop with an ORDERED clause.
- The ORDERED directive provides a way to "fine tune" where ordering is to be applied within a loop. Otherwise, it is not required.

Format:

C/C++

```
#pragma omp for ordered [clauses...]
   (loop region)

#pragma omp ordered  newline

   structured_block

   (endo of loop region)
```

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

```
!$OMP THREADPRIVATE (/cb/, ...)  cb is the name of a common block
```

C/C++

```
#pragma omp threadprivate  (list)
```

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:

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 */

}
```

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
```

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

- Also called Data-sharing 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.
- Data Scope Attribute Clauses are effective only within their lexical/static extent.
- **Important**: Please consult the latest OpenMP specs for important details and discussion on this topic.
- A Clauses / Directives Summary Table is provided for convenience.

OpenMP Directives: Data Scope Attribute Clauses: PRIVATE Clause

Purpose:

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

Format:

Fortran

```
PRIVATE (list)
```

C/C++

```
private (list)
```

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 should be assumed to be 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

OpenMP Directives: Data Scope Attribute Clauses: SHARED Clause

Purpose:

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

Format:

Fortran

```
SHARED (list)
```

C/C++

```
shared (list)
```

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).

OpenMP Directives: Data Scope Attribute Clauses: DEFAULT Clause

Purpose:

The DEFAULT clause allows the user to specify a default scope for all variables in the lexical extent of any parallel region.

Format:

Fortran

```
DEFAULT (PRIVATE | FIRSTPRIVATE | 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 or firstprivate as a possible default. However, actual implementations may provide this option.
- Using NONE as a default requires that the programmer explicitly scope all variables.

Restrictions:

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

OpenMP Directives: Data Scope Attribute Clauses: FIRSTPRIVATE Clause

Purpose:

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

Format:

Fortran

```
FIRSTPRIVATE (list)
```

C/C++

```
firstprivate (list)
```

Notes:

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

OpenMP Directives: Data Scope Attribute Clauses: 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 (list)
```

C/C++

```
lastprivate (list)
```

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

OpenMP Directives: Data Scope Attribute Clauses: 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 (list)
```

C/C++

```
copyin  (list)
```

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.

OpenMP Directives: Data Scope Attribute Clauses: COPYPRIVATE Clause

Purpose:

- The COPYPRIVATE clause can be used to broadcast values acquired by a single thread directly to all instances of the private variables in the other threads.
- Associated with the SINGLE directive
- See the most recent OpenMP specs document for additional discussion and examples.

Format:

Fortran

```
COPYPRIVATE (list)
```

C/C++

```
copyprivate  (list)
```

OpenMP Directives: Data Scope Attribute Clauses: 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:

C / C++ - reduction clause example

```
#include &ltomp.h&gt

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 operator expr ***x = expr operator x*** *(except subtraction)* ***x = intrinsic(x, expr)*** ***x = intrinsic(expr, x)***	***x = x op expr*** ***x = expr op x*** *(except subtraction)* ***x binop = 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 +, *, -, /, &, ^, |*

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	♦				♦	♦
COPYPRIVATE				♦		
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 worksharing region may not be closely nested inside a worksharing, explicit task, critical, ordered, atomic, or master region.
- A barrier region may not be closely nested inside a worksharing, explicit task, critical, ordered, atomic, or master region.
- A master region may not be closely nested inside a worksharing, atomic, or explicit task region.
- An ordered region may not be closely nested inside a critical, atomic, or explicit task region.
- An ordered region must be closely nested inside a loop region (or parallel loop region) with an ordered clause.
- A critical region may not be nested (closely or otherwise) inside a critical region with the same name. Note that this restriction is not sufficient to prevent deadlock.
- parallel, flush, critical, atomic, taskyield, and explicit task regions may not be closely nested inside an atomic region.

Run-Time Library Routines

Overview:

- The OpenMP API includes an ever-growing number of run-time library routines.

  See here

For C/C++, all of the run-time library routines are actual subroutines. For Fortran, some are actually functions, and some are subroutines. For example:

Fortran

```
INTEGER FUNCTION OMP_GET_NUM_THREADS()
```

C/C++

```
#include <omp.h>
int omp_get_num_threads(void)
```

- Note that for C/C++, you usually need to include the `<omp.h>` header file.
- Fortran routines are not case sensitive, but C/C++ routines are.
- 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:
  - Implementations may or may not support all OpenMP API features. For example, if nested parallelism is supported, it may be only nominal, in that a nested parallel region may only have one thread.
  - Consult your implementation's documentation for details - or experiment and find out for yourself if you can't find it in the documentation.
- The run-time library routines are discussed in more detail in Appendix A.

Environment Variables

OpenMP provides the following 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
```

Implementation notes:

- Your implementation may or may not support this feature.

OMP_PROC_BIND

Enables or disables threads binding to processors. Valid values are TRUE or FALSE. For example:

```
setenv OMP_PROC_BIND TRUE
```

Implementation notes:

- Your implementation may or may not support this feature.

OMP_NESTED

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

```
setenv OMP_NESTED TRUE
```

Implementation notes:

- Your implementation may or may not support this feature. If nested parallelism is supported, it is often only nominal, in that a nested parallel region may only have one thread.

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
```

Implementation notes:

- Your implementation may or may not support this feature.

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
```

Implementation notes:

- Your implementation may or may not support this feature.

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
```

Implementation notes:

- Your implementation may or may not support this feature.

OMP_THREAD_LIMIT

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
```

Implementation notes:

- Your implementation may or may not support this feature.

Thread Stack Size and Thread Binding

Thread Stack Size:

- 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. Using past versions of LC compilers as an example:

Compiler	Approx. Stack Limit	Approx. Array Size (doubles)
Linux icc, ifort	4 MB	700 x 700
Linux pgcc, pgf90	8 MB	1000 x 1000
Linux gcc, gfortran	2 MB	500 x 500

- Threads that exceed their stack allocation may or may not seg 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.
- If your OpenMP environment supports the OpenMP 3.0 OMP_STACKSIZE environment variable (covered in previous section), you can use it to set the thread stack size prior to program execution. For example:

```
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
```

- Otherwise, at LC, you should be able to use the method below for Linux clusters. The example shows setting the thread stack size to 12 MB, and as a precaution, setting the shell stack size to unlimited.

csh/tcsh	setenv KMP_STACKSIZE 12000000 limit stacksize unlimited
ksh/sh/bash	export KMP_STACKSIZE=12000000 ulimit -s unlimited

Thread Binding:

- In some cases, a program will perform better if its threads are bound to processors/cores.
- "Binding" a thread to a processor means that a thread will be scheduled by the operating system to always run on a the same processor.
- Otherwise, threads can be scheduled to execute on any processor and "bounce" back and forth between processors with each time slice.
- Also called "thread affinity" or "processor affinity"
- Binding threads to processors can result in better cache utilization, thereby reducing costly memory accesses. This is the primary motivation for binding threads to processors.
- Depending upon your platform, operating system, compiler and OpenMP implementation, binding threads to processors can be done several different ways.
- The OpenMP version 3.1 API provides an environment variable to turn processor binding "on" or "off". For example:

```
setenv OMP_PROC_BIND  TRUE
setenv OMP_PROC_BIND  FALSE
```

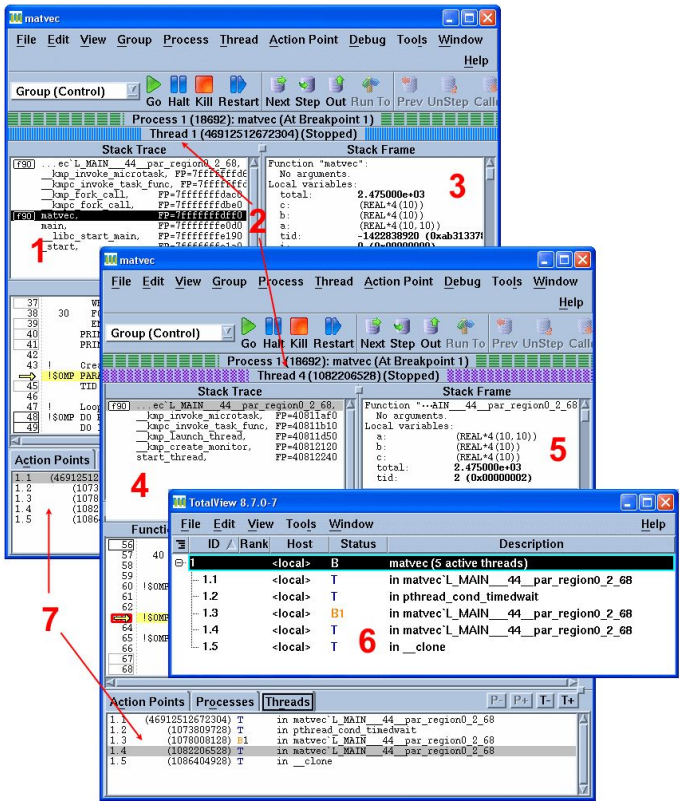
- At a higher level, processes can also be bound to processors.
- Detailed information about process and thread binding to processors on LC Linux clusters can be found HERE.

Monitoring, Debugging and Performance Analysis Tools for OpenMP

Monitoring and Debugging Threads:

- Debuggers vary in their ability to handle threads. The TotalView debugger is LC's recommended debugger for parallel programs. It is well suited for both monitoring and debugging threaded programs.
- An example screenshot from a TotalView session using an OpenMP code is shown below.

1. Master thread Stack Trace Pane showing original routine
2. Process/thread status bars differentiating threads
3. Master thread Stack Frame Pane showing shared variables
4. Worker thread Stack Trace Pane showing outlined routine.
5. Worker thread Stack Frame Pane
6. Root Window showing all threads
7. Threads Pane showing all threads plus selected thread



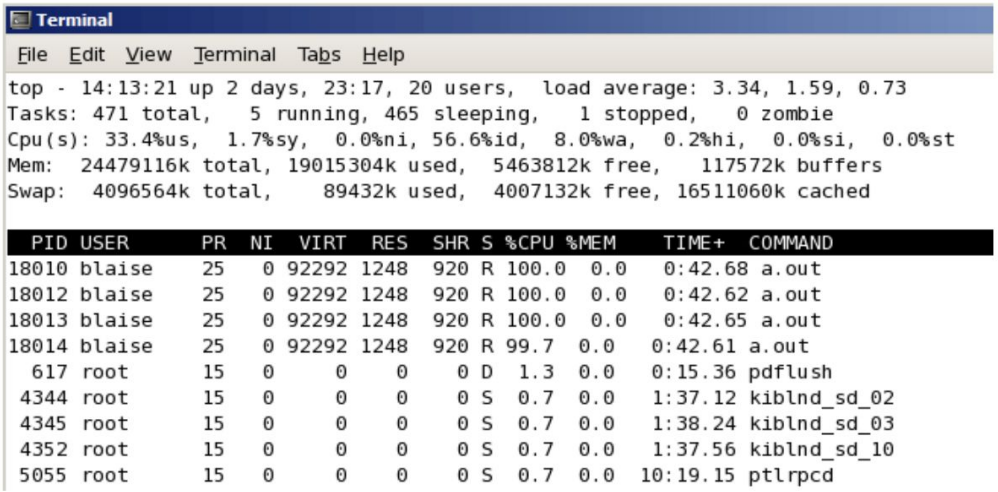
- The Linux `ps` command provides several flags for viewing thread information. Some examples are shown below. See the man page for details.

```
% ps -Lf
UID        PID  PPID   LWP  C NLWP STIME TTY          TIME CMD
blaise   22529 28240 22529  0    5 11:31 pts/53   00:00:00 a.out
blaise   22529 28240 22530 99    5 11:31 pts/53   00:01:24 a.out
blaise   22529 28240 22531 99    5 11:31 pts/53   00:01:24 a.out
blaise   22529 28240 22532 99    5 11:31 pts/53   00:01:24 a.out
blaise   22529 28240 22533 99    5 11:31 pts/53   00:01:24 a.out

% ps -T
  PID  SPID TTY          TIME CMD
22529 22529 pts/53   00:00:00 a.out
22529 22530 pts/53   00:01:49 a.out
22529 22531 pts/53   00:01:49 a.out
22529 22532 pts/53   00:01:49 a.out
22529 22533 pts/53   00:01:49 a.out

% ps -Lm
  PID   LWP TTY          TIME CMD
22529     - pts/53   00:18:56 a.out
    - 22529 -        00:00:00 -
    - 22530 -        00:04:44 -
    - 22531 -        00:04:44 -
    - 22532 -        00:04:44 -
    - 22533 -        00:04:44 -
```

LC's Linux clusters also provide the `top` command to monitor processes on a node. If used with the `-H` flag, the threads contained within a process will be visible. An example of the `top -H` command is shown below. The parent process is PID 18010 which spawned three threads, shown as PIDs 18012, 18013 and 18014.



Performance Analysis Tools:

- There are a variety of performance analysis tools that can be used with OpenMP programs. Searching the web will turn up a wealth of information.
- At LC, the list of supported computing tools can be found at: https://hpc.llnl.gov/software.
- These tools vary significantly in their complexity, functionality and learning curve. Covering them in detail is beyond the scope of this tutorial.
- Some tools worth investigating, specifically for OpenMP codes, include:
    - Open|SpeedShop
    - TAU
    - PAPI
    - Intel VTune Amplifier
    - ThreadSpotter