Published on *High Performance Computing* (**https://hpc.llnl.gov (https://hpc.llnl.gov)**)

OpenMP Tutorial

Table of Contents

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, 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 ideal for those who are new to parallel programming with OpenMP. A basic understanding of parallel programming in 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

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
- An abbreviation for: Open Multi-Processing

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
- Required to check for data dependencies, data conflicts, race conditions, deadlocks, or code sequences that cause a program to be classified as non-conforming
- Designed to handle parallel I/O.
- 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 • 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	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)	Absoft Corporation Edinburgh Portable Compilers GENIAS Software GmbH Myrias Computer Technologies, Inc. The Portland Group, Inc. (PGI)

Release History

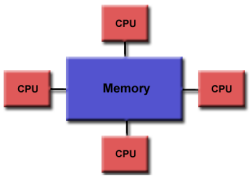
- OpenMP continues to evolve - new constructs and features are added with each release.
- Initially, the API specifications were released separately for C and Fortran. Since 2005, they have been released together.
- The table below chronicles the OpenMP API release history.

Date	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 2015	OpenMP 4.5
Nov 2018	OpenMP 5.0

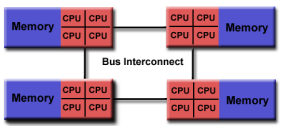
OpenMP Programming Model

Shared Memory Model:

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



Uniform Memory Access

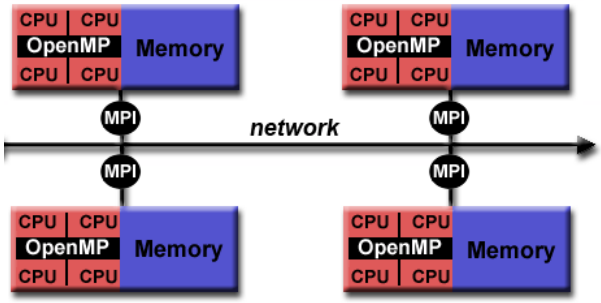


Non-Uniform Memory Access

- Because OpenMP is designed for shared memory parallel programming, it largely limited to single node parallelism. Typically, the number of processing elements (cores) on a node determine how much parallelism can be implemented.

Motivation for Using OpenMP in HPC:

- By itself, OpenMP parallelism is limited to a single node.
- For High Performance Computing (HPC) applications, OpenMP is combined with MPI for the distributed memory parallelism. This is often referred to as Hybrid Parallel Programming.
  - OpenMP is used for computationally intensive work on each node
  - MPI is used to accomplish communications and data sharing between nodes
- This allows parallelism to be implemented to the full scale of a cluster.



Hybrid OpenMP-MPI Parallelism

Thread Based Parallelism:

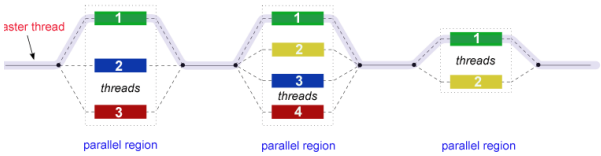
- OpenMP programs accomplish parallelism exclusively through the use of threads.
- A thread of execution is the smallest unit of processing that can be scheduled by an operating system. The idea of a subroutine that can be scheduled to run autonomously might help explain what a thread is.
- Threads exist 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.

Data Scoping:

- Because OpenMP is a shared memory programming model, most data within a parallel region is shared by default.
- All threads in a parallel region can access this shared data simultaneously.
- OpenMP provides a way for the programmer to explicitly specify how data is "scoped" if the default shared scoping is not desired.
- This topic is covered in more detail in the **Data Scope Attribute Clauses (/print/857#Clauses)** section.

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. This is particularly important if multiple threads attempt to write/read from the same file.
- If every thread conducts I/O to a different file, the issues are not as significant.
- It is entirely up to the programmer to ensure that I/O is conducted correctly within the context of a multi-threaded program.

Memory Model: FLUSH Often?

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

(/print/857#AppendixA) OpenMP API Overview

Three Components:

- The OpenMP 3.1 API is comprised of three distinct components:
  - Compiler Directives (19)
  - Runtime Library Routines (32)
  - Environment Variables (9)

  Later APIs include the same three components, but increase the number of directives, runtime library routines and environment variables.

- 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 (/print/857#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 (/print/857#RunTimeLibrary)** 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 iterations 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 (/print/857#EnvironmentVariables)** section later.

Example OpenMP Code Structure:

Fortran - General Code Structure

```
      PROGRAM HELLO
      INTEGER VAR1, VAR2, VAR3

      Serial code
            .
            .
            .

      Beginning of parallel region. Fork a team of threads.
      Specify variable scoping

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

         Parallel region 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
```

```
1
8
1
9
2
0
2
1
2
2
2
3
2
4
2
5
2
6
2
7
2
8
2
9
3
0
```

Compiling OpenMP Programs

LC OpenMP Implementations:

- As of June 2019, the documentation sources for LC's default compilers claim the following OpenMP support:

Platform	Compiler	Version Flag	Default Version	Supports
Linux	Intel C/C++, Fortran	--version	18.0.1	OpenMP 4.5 (most)
	GNU C/C++, Fortran	--version	4.9.3	OpenMP 4.0

	PGI C/C++, Fortran	-V --version	19.4	OpenMP 4.5 (limitations - no GPU offloading)
	Clang C/C++	--version	6.0.0	OpenMP 4.5 (limitations - no GPU offloading or device support
SIERRA CORAL EA	IBM XL C/C++	-qversion	16.1.1	OpenMP 4.5
	IBM XL Fortran	-qversion	16.1.1	OpenMP 4.5
	GNU C/C++	--version	4.9.3	OpenMP 4.0
	GNU Fortran	--version	4.9.3	OpenMP 4.0
	PGI C/C++, Fortran	-V --version	18.10	OpenMP 4.5 (limitations - no GPU offloading)
	Clang C/C++ (IBM)	--version	9.0	OpenMP 4.5

- To view all LC compiler versions, use the command: module avail
- Best place to view OpenMP support by a range of compilers: **https://www.openmp.org/resources/openmp-compilers-tools/ (https://www.openmp.org/resources/openmp-compilers-tools/)**.

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.
- For MPI compiler commands, see: **https://computing.llnl.gov/tutorials/mpi/#BuildScripts (https://computing.llnl.gov/tutorials/mpi/#BuildScripts)**

Compiler / Platform	Compiler Commands	OpenMP Flag
Intel Linux	icc icpc ifort	-qopenmp
GNU Linux IBM Blue Gene Sierra, CORAL EA	gcc g++ g77 gfortran	-fopenmp

PGI Linux Sierra, CORAL EA	pgcc pgCC pgf77 pgf90	-mp
Clang Linux Sierra, CORAL EA	clang clang++	-fopenmp
IBM XL Blue Gene	bgxlc_r, bgcc_r bgxlC_r, bgxlc++_r bgxlc89_r bgxlc99_r bgxlf_r bgxlf90_r bgxlf95_r bgxlf2003_r	-qsmp=omp
IBM XL Sierra, CORAL EA	xlc_r xlC_r, xlc++_r xlf_r xlf90_r xlf95_r xlf2003_r xlf2008_r	-qsmp=omp

- Compiler Documentation:
  - IBM Sierra, CORAL EA: Select the relevant version of Little Endian documents at **http://www-01.ibm.com/support/docview.wss?uid=swg27036675 (http://www-01.ibm.com/support/docview.wss?uid=swg27036675)** (C/C++) and **http://www-01.ibm.com/support/docview.wss?uid=swg27036672 (http://www-01.ibm.com/support/docview.wss?uid=swg27036672)** (Fortran).
  - Intel and PGI: compiler docs are included in the /opt/compilername directory. Otherwise, see Intel or PGI web pages.
  - GNU: **gnu.org (http://gnu.org)**
  - Clang: **http://clang.llvm.org/docs/ (http://clang.llvm.org/docs/)**
  - IBM BlueGene Fortran: **http://www-01.ibm.com/support/docview.wss?uid=swg27027154 (http://www-01.ibm.com/support/docview.wss?uid=swg27027154)** and C/C++: **http://www-01.ibm.com/support/docview.wss?uid=swg27027155 (http://www-01.ibm.com/support/docview.wss?uid=swg27027155)**

OpenMP Directives

Fortran Directives 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:	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.

```
!$OMP
C$OMP
*$OMP
```

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++ Directives 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.

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.
- 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:

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

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


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

STATIC EXTENT
The DO directive occurs within an enclosing parallel region

ORPHANED DIRECTIVES
The CRITICAL and SECTIONS directives occur outside an enclosing parallel region

DYNAMIC EXTENT
The CRITICAL and SECTIONS directives occur within the dynamic extent of the DO and PARALLEL directives.

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 (/print/857#BindingNesting)** 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 | NON
E)
               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 region. 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:
    - Evaluation of the IF clause
    - Setting of the NUM_THREADS clause
    - Use of the omp_set_num_threads() library function
    - Setting of the OMP_NUM_THREADS environment variable

- Implementation default - usually the number of CPUs on a node, though it could be dynamic (see next bullet).

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

Dynamic Threads:

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

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

  - The omp_set_dynamic() library routine

  - Setting of the OMP_DYNAMIC environment variable to TRUE

Nested Parallel Regions:

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

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 (/print/857#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

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

Fortran - Parallel Region Example

```
1
2
3
4
5
6
7
```

```
        PROGRAM HELLO

        INTEGER NTHREADS, TID, OMP_GET_NUM_THRE
ADS,
     +   OMP_GET_THREAD_NUM

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

   !     Obtain and print thread id
       TID = OMP_GET_THREAD_NUM()
       PRINT *, 'Hello World from thread = ', T
ID
```

```
8                !     Only master thread does this
                 IF (TID .EQ. 0) THEN
9                  NTHREADS = OMP_GET_NUM_THREADS()
1                  PRINT *, 'Number of threads = ', NTHRE
0              ADS
1                 END IF
1
1                !     All threads join master thread and dis
2              band
1               !$OMP END PARALLEL
3
1                END
4
1
5
1
6
1
7
1
8
1
9
2
0
2
1
2
2
```

C / C++ - Parallel Region Example

```
1
2
3
4
5
6
7
8
9
1
0
1
1
1
2
1
3
1
4
1
5
1
6
1
7
1
8
1
9
2
```

```
#include <omp.h>

main(int argc, char *argv[]) {

int nthreads, tid;

  /* Fork a team of threads with each thread h
aving 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", ti
d);

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

  }    /* All threads join master thread and t
erminate */

}
```

```
0
2
1
2
2
2
3
2
4
```

OpenMP Exercise 1

Getting Started

Overview:

- Login to the workshop cluster using your workshop username and OTP token
- Copy the exercise files to your home directory
- Familiarize yourself with LC's OpenMP environment
- Write a simple "Hello World" OpenMP program
- Successfully compile your program
- Successfully run your program
- Modify the number of threads used to run your program

GO TO THE EXERCISE HERE (https://hpc.llnl.gov/exercise.html)

Approx. 20 minutes

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.

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

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

   do_loop

!$OMP END DO  [ NOWAIT ]
```

C/C++

```
#pragma omp for [clause ...]  newline
               schedule (type [,chunk])
               ordered
               private (list)
               firstprivate (list)
               lastprivate (list)
               shared (list)
               reduction (operator: list)
               collapse (n)
               nowait

   for_loop
```

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 **http://openmp.org/forum/viewtopic.php?f=3&t=83 (http://openmp.org/forum/viewtopic.php?f=3&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.

Note: compilers differ in how GUIDED is implemented as shown in the "Guided A" and "Guided B" examples below.

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 order of the iterations in the collapsed iteration space is determined as though they were executed sequentially. May improve performance.

- Other clauses are described in detail later, in the **Data Scope Attribute Clauses (/print/857#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 invariant 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

```
1
 2
 3
 4
 5
 6
 7
 8
 9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
```

```
      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

```
1
 2
 3
 4
 5
 6
 7
 8
 9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
```

```
#include <omp.h>
#define N 1000
#define CHUNKSIZE 100

main(int argc, char *argv[]) {

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) pri
vate(i)
  {

  #pragma omp for schedule(dynamic,chunk) no
wait
  for (i=0; i < N; i++)
    c[i] = a[i] + b[i];

  }     /* end of parallel region */

}
```

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 (/print/857#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

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
```

```
#include <omp.h>
#define N 1000

main(int argc, char *argv[]) {

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) privat
e(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 region */

}
```

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
```

Clauses:

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

- Clauses are described in detail later, in the **Data Scope Attribute Clauses (/print/857#Clauses)** section.

Restrictions:

- It is illegal to branch into or out of a SINGLE 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 argc, char *argv[]) {

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 Exercise 2

Work-Sharing Constructs

Overview:

- Login to the LC workshop cluster, if you are not already logged in
- Work-Sharing DO/for construct examples: review, compile and run
- Work-Sharing SECTIONS construct example: review, compile and run

GO TO THE EXERCISE HERE (https://hpc.llnl.gov/exercise.html%23Exercise2)

Approx. 20 minutes

OpenMP Directives

Synchronization Constructs

- Consider a simple example where two threads are both trying to update variable x at the same time:

THREAD 1:

```
update(x)
{
    x = x + 1
}


x = 0
update(x)
print(x)
```

THREAD 2:

```
update(x)
{
    x = x + 1
}


x = 0
update(x)
print(x)
```

- One possible execution sequence:
  - Thread 1 initializes x to 0 and calls update(x)
  - Thread 1 adds 1 to x.
    x now equals 1
  - Thread 2 initializes x to 0 and calls update(x)
    x now equals 0
  - Thread 1 prints x, which is equal to 0 instead of 1
  - Thread 2 adds 1 to x.
    x now equals 1.
  - Thread 2 prints x as 1.

- To avoid a situation like this, the updating 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 <omp.h>

main(int argc, char *argv[]) {

int x;
x = 0;

#pragma omp parallel shared(x)
  {

  #pragma omp critical
  x = x + 1;

  }    /* end of parallel region */

}
```

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:

- OpenMP 3.1 feature
- 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 is a stand-alone directive, 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
ATOMIC Directive

Purpose:

- The atomic construct ensures that a specific storage location is accessed atomically, rather than exposing it to the possibility of multiple, simultaneous reading and writing threads that may result in indeterminate values. In essence, this directive provides a mini-CRITICAL section.

Format:

Fortran

```
!$OMP ATOMIC [ read | write | update | capture ]

   statement_expression
```

C/C++

```
#pragma omp atomic  [ read | write | update | capture ] newli
ne

   statement_expression
```

Restrictions:

- The directive applies only to a single, immediately following statement

- An atomic statement must follow a specific syntax. See the most recent OpenMP specs for this.

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.

- 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
  END PARALLEL
  CRITICAL and END CRITICAL
  END DO
  END SECTIONS
  END SINGLE
  ORDERED and END ORDERED

- barrier
  parallel - upon entry and exit
  critical - upon entry and exit
  ordered - upon entry and exit
  for - upon exit
  sections - upon exit
  single - upon exit

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:

Fortran

```
!$OMP DO ORDERED [clauses...]
   (loop region)

!$OMP ORDERED

   (block)

!$OMP END ORDERED

   (end of loop region)
!$OMP END DO
```

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 specifies that variables are replicated, with each thread having its own copy.
- Can be used to make global file scope variables (C/C++/Fortran) or common blocks (Fortran) local and persistent to a thread through the execution of multiple parallel regions.

Format:

Fortran

```
!$OMP THREADPRIVATE (list)
```

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.

- 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 regions of a code.

- Examples:

Fortran - THREADPRIVATE Directive Example

```
      PROGRAM THREADPRIV

      INTEGER A, B, I, TID, OMP_GET_THREAD_NUM
      REAL*4 X
      COMMON /C1/ A

!$OMP THREADPRIVATE(/C1/, X)

  !     Explicitly turn off dynamic threads
      CALL OMP_SET_DYNAMIC(.FALSE.)

      PRINT *, '1st Parallel Region:'
!$OMP PARALLEL PRIVATE(B, TID)
      TID = OMP_GET_THREAD_NUM()
      A = TID
      B = TID
      X = 1.1 * TID + 1.0
      PRINT *, 'Thread',TID,':   A,B,X=',A,B,X
!$OMP END PARALLEL

      PRINT *, '******************************
******'
      PRINT *, 'Master thread doing serial wor
k here'
      PRINT *, '******************************
******'

      PRINT *, '2nd Parallel Region: '
!$OMP PARALLEL PRIVATE(TID)
      TID = OMP_GET_THREAD_NUM()
      PRINT *, 'Thread',TID,':   A,B,X=',A,B,X
!$OMP END PARALLEL

```

```
      END
```

Output: 1st Parallel Region: Thread 0 : A,B,X= 0 0 1.000000000 Thread 1 : A,B,X= 1 1 2.099999905 Thread 3 : A,B,X= 3 3 4.300000191 Thread 2 : A,B,X= 2 2 3.200000048 ********** Master thread doing serial work here ********** 2nd Parallel Region: Thread 0 : A,B,X= 0 0 1.000000000 Thread 2 : A,B,X= 2 0 3.200000048 Thread 3 : A,B,X= 3 0 4.300000191 Thread 1 : A,B,X= 1 0 2.099999905

C/C++ - threadprivate Directive Example

```
 #include <omp.h>
1
 int  a, b, i, tid;
2
 float x;
3
 #pragma omp threadprivate(a, x)
4
 main(int argc, char *argv[]) {
5
     /* Explicitly turn off dynamic threads */
   omp_set_dynamic(0);
6
   printf("1st Parallel Region:\n");
7
 #pragma omp parallel private(b,tid)
   {
8
   tid = omp_get_thread_num();
   a = tid;
9
   b = tid;
1
   x = 1.1 * tid +1.0;
0
   printf("Thread %d:   a,b,x= %d %d %f\n",tid,
1
a,b,x);
1
   }    /* end of parallel region */
1
2
   printf("************************************
1
\n");
3
   printf("Master thread doing serial work here
1
\n");
4
   printf("************************************
1
\n");
5
1
   printf("2nd Parallel Region:\n");
6
 #pragma omp parallel private(tid)
1
   {
7
   tid = omp_get_thread_num();
1
   printf("Thread %d:   a,b,x= %d %d %f\n",tid,
8
a,b,x);
1
   }  /* end of parallel region */
9
2
```

```
0
2
1
2
2
2
3
2
4
2
5
2
6
2
7
2
8
2
9
3
0
3
1
3
2
3
3
3
4
```

```
}
```

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.

- Fortran: common block names must appear between slashes: /cb/

- See the most recent OpenMP specs for additional restrictions not listed here.

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 regions of the program (and back)

  - They define which variables will be visible to all threads in the parallel regions 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 (/print/857#ClausesDirectives)** is provided for convenience.

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
  - Should be assumed to be uninitialized for each thread

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)

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

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.

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

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.

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)
```

REDUCTION Clause

Purpose:

- The REDUCTION clause performs a reduction operation on the variables that appear in its list.
- A private copy for each list variable is created and initialized 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

```
REDUCTION (operator: list)
```

C/C++

```
reduction (operator: list)
```

Valid Operators and Initialization Values

Operation	Fortran	C/C++	Initialization
Addition	+	+	0
Multiplication	*	*	1

Subtraction	-	-	0
Logical AND	.and.	&&	.true. / 1
Logical OR	.or.	||	.false. / 0
AND bitwise	iand	&	all bits on / ~0
OR bitwise	ior	|	0
Exclusive OR bitwise	ieor	^	0
Equivalent	.eqv.		.true.
Not Equivalent	.neqv.		.false.
Maximum	max	max	Most negative #
Minimum	min	min	Largest positive #

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 PARALLEL DO

      PRINT *, 'Final Result= ', RESULT
      END
```

C / C++ - reduction Clause Example

```
#include <omp.h>

main(int argc, char *argv[])  {

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:

- The type of a list item must be valid for the reduction operator.
- List items/variables can not be declared shared or private.
- Reduction operations may not be associative for real numbers.
- See the OpenMP standard API for additional restrictions.

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						
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.

- These routines are used for a variety of purposes as shown in the table below:

Routine	Purpose
OMP_SET_NUM_THREADS (/print/857#OMP_SET_NUM_THREADS)	Sets the number of threads that will be used in the next parallel region
OMP_GET_NUM_THREADS (/print/857#OMP_GET_NUM_THREADS)	Returns the number of threads that are currently in the team executing the parallel region from which it is called
OMP_GET_MAX_THREADS (/print/857#OMP_GET_MAX_THREADS)	Returns the maximum value that can be returned by a call to the OMP_GET_NUM_THREADS function
OMP_GET_THREAD_NUM (/print/857#OMP_GET_THREAD_NUM)	Returns the thread number of the thread, within the team, making this call.
OMP_GET_THREAD_LIMIT (/print/857#OMP_GET_THREAD_LIMIT)	Returns the maximum number of OpenMP threads available to a program
OMP_GET_NUM_PROCS (/print/857#OMP_GET_NUM_PROCS)	Returns the number of processors that are available to the program
OMP_IN_PARALLEL (/print/857#OMP_IN_PARALLEL)	Used to determine if the section of code which is executing is parallel or not
OMP_SET_DYNAMIC (/print/857#OMP_SET_DYNAMIC)	Enables or disables dynamic adjustment (by the run time system) of the number of threads available for execution of parallel regions
OMP_GET_DYNAMIC (/print/857#OMP_GET_DYNAMIC)	Used to determine if dynamic thread adjustment is enabled or not
OMP_SET_NESTED (/print/857#OMP_SET_NESTED)	Used to enable or disable nested parallelism
OMP_GET_NESTED (/print/857#OMP_GET_NESTED)	Used to determine if nested parallelism is enabled or not
OMP_SET_SCHEDULE (/print/857#OMP_SET_SCHEDULE)	Sets the loop scheduling policy when "runtime" is used as the schedule kind in the OpenMP directive
OMP_GET_SCHEDULE (/print/857#OMP_GET_SCHEDULE)	Returns the loop scheduling policy when "runtime" is used as the schedule kind in the OpenMP directive

OMP_SET_MAX_ACTIVE_LEVELS (/print/857#OMP_SET_MAX_ACTIVE_LEVELS)	Sets the maximum number of nested parallel regions
OMP_GET_MAX_ACTIVE_LEVELS (/print/857#OMP_GET_MAX_ACTIVE_LEVELS)	Returns the maximum number of nested parallel regions
OMP_GET_LEVEL (/print/857#OMP_GET_LEVEL)	Returns the current level of nested parallel regions
OMP_GET_ANCESTOR_THREAD_NUM (/print/857#OMP_GET_ANCESTOR_THREAD_NUM)	Returns, for a given nested level of the current thread, the thread number of ancestor thread
OMP_GET_TEAM_SIZE (/print/857#OMP_GET_TEAM_SIZE)	Returns, for a given nested level of the current thread, the size of the thread team
OMP_GET_ACTIVE_LEVEL (/print/857#OMP_GET_ACTIVE_LEVEL)	Returns the number of nested, active parallel regions enclosing the task that contains the call
OMP_IN_FINAL (/print/857#OMP_IN_FINAL)	Returns true if the routine is executed in the final task region; otherwise it returns false
OMP_INIT_LOCK (/print/857#OMP_INIT_LOCK)	Initializes a lock associated with the lock variable
OMP_DESTROY_LOCK (/print/857#OMP_DESTROY_LOCK)	Disassociates the given lock variable from any locks
OMP_SET_LOCK (/print/857#OMP_SET_LOCK)	Acquires ownership of a lock
OMP_UNSET_LOCK (/print/857#OMP_UNSET_LOCK)	Releases a lock
OMP_TEST_LOCK (/print/857#OMP_TEST_LOCK)	Attempts to set a lock, but does not block if the lock is unavailable
OMP_INIT_NEST_LOCK (/print/857#OMP_INIT_LOCK)	Initializes a nested lock associated with the lock variable
OMP_DESTROY_NEST_LOCK (/print/857#OMP_DESTROY_LOCK)	Disassociates the given nested lock variable from any locks
OMP_SET_NEST_LOCK (/print/857#OMP_SET_LOCK)	Acquires ownership of a nested lock
OMP_UNSET_NEST_LOCK (/print/857#OMP_UNSET_LOCK)	Releases a nested lock
OMP_TEST_NEST_LOCK (/print/857#OMP_TEST_LOCK)	Attempts to set a nested lock, but does not block if the lock is unavailable
OMP_GET_WTIME (/print/857#OMP_GET_WTIME)	Provides a portable wall clock timing routine

OMP_GET_WTICK (/print/857#OMP_GET_WTICK)	Returns a double-precision floating point value equal to the number of seconds between successive clock ticks

- 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

OMP_PROC_BIND

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

OMP_NESTED

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

OMP_STACKSIZE

Controls the size of the stack for created (non-Master) threads. Examples:
setenv OMP_STACKSIZE 2000500B
setenv OMP_STACKSIZE "3000 k "
setenv OMP_STACKSIZE 10M
setenv OMP_STACKSIZE " 10 M "
setenv OMP_STACKSIZE "20 m "
setenv OMP_STACKSIZE " 1G"
setenv OMP_STACKSIZE 20000

OMP_WAIT_POLICY

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

OMP_MAX_ACTIVE_LEVELS

Controls the maximum number of nested active parallel regions. The value of this environment variable must be a non-negative integer. The behavior of the program is implementation defined if the requested value of OMP_MAX_ACTIVE_LEVELS is greater than the maximum number of nested active parallel levels an implementation can support, or if the value is not a non-negative integer. Example:
setenv OMP_MAX_ACTIVE_LEVELS 2

OMP_THREAD_LIMIT

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

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 at **https://lc.llnl.gov/confluence/display/TLCC2/mpibind (https://lc.llnl.gov/confluence/display/TLCC2/mpibind)**.

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.
  - Master thread Stack Trace Pane showing original routine
  - Process/thread status bars differentiating threads
  - Master thread Stack Frame Pane showing shared variables
  - Worker thread Stack Trace Pane showing outlined routine.
  - Worker thread Stack Frame Pane
  - Root Window showing all threads
  - Threads Pane showing all threads plus selected thread

- See the **TotalView Debugger tutorial (https://hpc.llnl.gov/../totalview/index.html)** for details.

- The Linux ps command provides several flags for viewing thread information. Some examples are shown below. See the **man page (https://hpc.llnl.gov/../pthreads/man/ps.txt)** 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/development-environment-software (https://hpc.llnl.gov/software/development-environment-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

OpenMP Exercise 3

Assorted

Overview:

- Login to the workshop cluster, if you are not already logged in
- Orphaned directive example: review, compile, run
- Get OpenMP implementation environment information
- Hybrid OpenMP + MPI programs
- Check out the "bug" programs

GO TO THE EXERCISE HERE (https://hpc.llnl.gov/exercise.html%23Exercise3)

This completes the tutorial.

(https://hpc.llnl.gov/../evaluation/index.html)	Please complete the online evaluation form - unless you are doing the exercise, in which case please complete it at the end of the exercise.

Where would you like to go now?

- **Exercise (https://hpc.llnl.gov/exercise.html)**
- **Agenda (https://hpc.llnl.gov/../agenda/index.html)**
- **Back to the top (/print/857#top)**

References and More Information

- Original Author: Blaise Barney; Contact: **hpc-tutorials@llnl.gov (mailto:hpc-tutorials@llnl.gov)**, Livermore Computing.

- The OpenMP web site, which includes the C/C++ and Fortran Application Program Interface documents. **www.openmp.org (http://www.openmp.org)**

Appendix A: Run-Time Library Routines

OMP_SET_NUM_THREADS

Purpose:

- Sets the number of threads that will be used in the next parallel region. Must be a positive integer.

Format:

Fortran

```
SUBROUTINE OMP_SET_NUM_THREADS(scalar_integer_expression)
```

C/C++

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

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

```
INTEGER FUNCTION OMP_GET_NUM_THREADS()
```

C/C++

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

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

```
INTEGER FUNCTION OMP_GET_MAX_THREADS()
```

C/C++

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

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++

```
#include <omp.h>
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_THREAD_LIMIT

Purpose:

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

Format:

Fortran

```
INTEGER FUNCTION OMP_GET_THREAD_LIMIT
```

C/C++

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

Notes:

- Also see the OMP_THREAD_LIMIT environment variable.

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++

```
#include <omp.h>
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

```
LOGICAL FUNCTION OMP_IN_PARALLEL()
```

C/C++

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

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

```
SUBROUTINE OMP_SET_DYNAMIC(scalar_logical_expression)
```

C/C++

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

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

```
LOGICAL FUNCTION OMP_GET_DYNAMIC()
```

C/C++

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

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

```
SUBROUTINE OMP_SET_NESTED(scalar_logical_expression)
```

C/C++

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

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++

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

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_SET_SCHEDULE

Purpose:

- This routine sets the schedule type that is applied when the loop directive specifies a runtime schedule.

Format:

Fortran

```
SUBROUTINE OMP_SET_SCHEDULE(KIND, MODIFIER)
INTEGER (KIND=OMP_SCHED_KIND) KIND
INTEGER MODIFIER
```

C/C++

```
#include <omp.h>
void omp_set_schedule(omp_sched_t kind, int modifier)
```

OMP_GET_SCHEDULE

Purpose:

- This routine returns the schedule that is applied when the loop directive specifies a runtime schedule.

Format:

Fortran

```
SUBROUTINE OMP_GET_SCHEDULE(KIND, MODIFIER)
INTEGER (KIND=OMP_SCHED_KIND) KIND
INTEGER MODIFIER
```

C/C++

```
#include <omp.h>
void omp_get_schedule(omp_sched_t * kind, int * modifier )
```

OMP_SET_MAX_ACTIVE_LEVELS

Purpose:

- This routine limits the number of nested active parallel regions.

Format:

Fortran

```
SUBROUTINE OMP_SET_MAX_ACTIVE_LEVELS (MAX_LEVELS)
INTEGER MAX_LEVELS
```

C/C++

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

Notes & Restrictions:

- If the number of parallel levels requested exceeds the number of levels of parallelism supported by the implementation, the value will be set to the number of parallel levels supported by the implementation.

- This routine has the described effect only when called from the sequential part of the program. When called from within an explicit parallel region, the effect of this routine is implementation defined.

OMP_GET_MAX_ACTIVE_LEVELS

Purpose:

- This routine returns the maximum number of nested active parallel regions.

Format:

Fortran

```
INTEGER FUNCTION OMP_GET_MAX_ACTIVE_LEVELS()
```

C/C++

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

OMP_GET_LEVEL

Purpose:

- This routine returns the number of nested parallel regions enclosing the task that contains the call.

Format:

Fortran

```
INTEGER FUNCTION OMP_GET_LEVEL()
```

C/C++

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

Notes & Restrictions:

- The omp_get_level routine returns the number of nested parallel regions (whether active or inactive) enclosing the task that contains the call, not including the implicit parallel region. The routine always returns a non-negative integer, and returns 0 if it is called from the sequential part of the program.

OMP_GET_ANCESTOR_THREAD_NUM

Purpose:

- This routine returns, for a given nested level of the current thread, the thread number of the ancestor or the current thread.

Format:

Fortran

```
INTEGER FUNCTION OMP_GET_ANCESTOR_THREAD_NUM(LEVEL)
INTEGER LEVEL
```

C/C++

```
#include <omp.h>
int omp_get_ancestor_thread_num(int level)
```

Notes & Restrictions:

- If the requested nest level is outside the range of 0 and the nest level of the current thread, as returned by the omp_get_level routine, the routine returns -1.

OMP_GET_TEAM_SIZE

Purpose:

- This routine returns, for a given nested level of the current thread, the size of the thread team to which the ancestor or the current thread belongs.

Format:

Fortran

```
INTEGER FUNCTION OMP_GET_TEAM_SIZE(LEVEL)
INTEGER LEVEL
```

C/C++

```
#include <omp.h>
int omp_get_team_size(int level);
```

Notes & Restrictions:

- If the requested nested level is outside the range of 0 and the nested level of the current thread, as returned by the omp_get_level routine, the routine returns -1. Inactive parallel regions are regarded like active parallel regions executed with one thread.

OMP_GET_ACTIVE_LEVEL

Purpose:

- The omp_get_active_level routine returns the number of nested, active parallel regions enclosing the task that contains the call.

Format:

Fortran

```
INTEGER FUNCTION OMP_GET_ACTIVE_LEVEL()
```

C/C++

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

Notes & Restrictions:

- The routine always returns a nonnegative integer, and returns 0 if it is called from the sequential part of the program.

OMP_IN_FINAL

Purpose:

- This routine returns true if the routine is executed in a final task region; otherwise, it returns false.

Format:

Fortran

```
LOGICAL FUNCTION OMP_IN_FINAL()
```

C/C++

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

OMP_INIT_LOCK
OMP_INIT_NEST_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++

```
#include <omp.h>
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
- For Fortran, *var* must be an integer large enough to hold an address, such as INTEGER*8 on 64-bit systems.

OMP_DESTROY_LOCK
OMP_DESTROY_NEST_LOCK

Purpose:

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

Format:

Fortran

```
SUBROUTINE OMP_DESTROY_LOCK(var)
SUBROUTINE OMP_DESTROY_NEST_LOCK(var)
```

C/C++

```
#include <omp.h>
void omp_destroy_lock(omp_lock_t *lock)
void omp_destroy_nest_lock(omp_nest_lock_t *lock)
```

Notes & Restrictions:

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

- For Fortran, *var* must be an integer large enough to hold an address, such as INTEGER*8 on 64-bit systems.

OMP_SET_LOCK
OMP_SET_NEST_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

```
SUBROUTINE OMP_SET_LOCK(var)
SUBROUTINE OMP_SET_NEST_LOCK(var)
```

C/C++

```
#include <omp.h>
void omp_set_lock(omp_lock_t *lock)
void omp_set_nest__lock(omp_nest_lock_t *lock)
```

Notes & Restrictions:

- It is illegal to call this routine with a lock variable that is not initialized.
- For Fortran, *var* must be an integer large enough to hold an address, such as INTEGER*8 on 64-bit systems.

OMP_UNSET_LOCK
OMP_UNSET_NEST_LOCK

Purpose:

- This subroutine releases the lock from the executing subroutine.

Format:

Fortran

```
SUBROUTINE OMP_UNSET_LOCK(var)
SUBROUTINE OMP_UNSET_NEST_LOCK(var)
```

C/C++

```
#include <omp.h>
void omp_unset_lock(omp_lock_t *lock)
void omp_unset_nest__lock(omp_nest_lock_t *lock)
```

Notes & Restrictions:

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

- For Fortran, *var* must be an integer large enough to hold an address, such as INTEGER*8 on 64-bit systems.

OMP_TEST_LOCK
OMP_TEST_NEST_LOCK

Purpose:

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

Format:

Fortran

```
SUBROUTINE OMP_TEST_LOCK(var)
SUBROUTINE OMP_TEST_NEST_LOCK(var)
```

C/C++

```
#include <omp.h>
int omp_test_lock(omp_lock_t *lock)
int omp_test_nest__lock(omp_nest_lock_t *lock)
```

Notes & Restrictions:

- For Fortran, .TRUE. is returned if the lock was set successfully, otherwise .FALSE. is returned.
- For Fortran, *var* must be an integer large enough to hold an address, such as INTEGER*8 on 64-bit systems.
- 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.

OMP_GET_WTIME

Purpose:

- Provides a portable wall clock timing routine
- Returns a double-precision floating point value equal to the number of elapsed seconds since some point in the past. Usually used in "pairs" with the value of the first call subtracted from the value of the second call to obtain the elapsed time for a block of code.
- Designed to be "per thread" times, and therefore may not be globally consistent across all threads in a team - depends upon what a thread is doing compared to other threads.

Format:

Fortran

```
DOUBLE PRECISION FUNCTION OMP_GET_WTIME()
```

C/C++

```
#include <omp.h>
double omp_get_wtime(void)
```

OMP_GET_WTICK

Purpose:

- Provides a portable wall clock timing routine
- Returns a double-precision floating point value equal to the number of seconds between successive clock ticks.

Format:

Fortran

```
DOUBLE PRECISION FUNCTION OMP_GET_WTICK()
```

C/C++

```
#include <omp.h>
double omp_get_wtick(void)
```

Lawrence Livermore National Laboratory is operated by Lawrence Livermore National Security, LLC, for the U.S. Department of Energy, National Nuclear Security Administration under Contract DE-AC52-07NA27344.