AMS 250: An Introduction to High Performance Computing

OpenMP Primer

Shawfeng Dong

shaw@ucsc.edu

(831) 502-7743

Applied Mathematics & Statistics

University of California, Santa Cruz

$$\frac{\pi}{4} = tan^{-1}(1) = \int_0^1 \frac{dx}{1+x^2}$$

Serial code:

```
double x, pi, step, sum=0.0;
int i;
step = 1./(double)num_steps;
struct timeval tv;
gettimeofday(&tv, NULL);
// start time in milliseconds
start = (tv.tv_sec)*1000 + (tv.tv_usec)/1000;


for (i=0; i<num_steps; i++) {
        x = (i + .5)*step;
        sum = sum + 1.0/(1.+ x*x);
}
pi = 4.0*sum*step;
gettimeofday(&tv, NULL);
// stop time in milliseconds
stop = (tv.tv_sec)*1000 + (tv.tv_usec)/1000;
```

OpenMP code:

```
double x, pi, step, sum=0.0;
int i;
step = 1./(double)num_steps;
struct timeval tv;
gettimeofday(&tv, NULL);
// start time in milliseconds
start = (tv.tv_sec)*1000 + (tv.tv_usec)/1000;

#pragma omp parallel for private(x) reduction(+:sum)
for (i=0; i<num_steps; i++) {
        x = (i + .5)*step;
        sum = sum + 1.0/(1.+ x*x);
}
pi = 4.0*sum*step;
gettimeofday(&tv, NULL);
// stop time in milliseconds
stop = (tv.tv_sec)*1000 + (tv.tv_usec)/1000;
```

It is that simple with OpenMP!

Confession

A lot of blatant plagiarism of Blaise Barney's OpenMP tutorial at LLNL!

https://computing.llnl.gov/tutorials/openMP

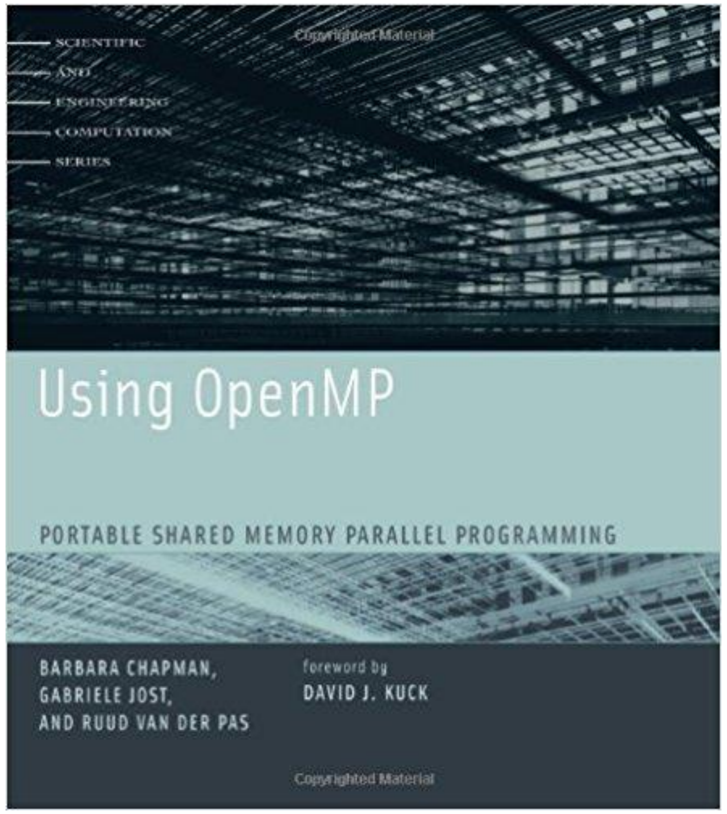
OpenMP Book and Examples

- Book:

  Using OpenMP: Portable Shared Memory Parallel Programming,
  by B. Chapman, G. Jost, & R. van der Pas, MIT press, 2007: http://ieeexplore.ieee.org/xpl/bkabstractplus.jsp?bkn=6267237

- Examples:

  https://github.com/shawfdong/ams250/tree/master/examples/openmp



Outline

What is OpenMP?

- http://openmp.org/

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

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, or deadlocks
- ☹ Required to check for code sequences that cause a program to be classified as non-conforming
- ☹ Meant to cover compiler-generated automatic parallelization and directives to the compiler to assist such parallelization
- ☹ Designed to guarantee that input or output to the same file is synchronous when executed in parallel. The programmer is responsible for synchronizing input and output.

Goals of OpenMP

- **Standardization:**
  - Provide a standard among a variety of shared memory architectures/platforms
  - Jointly defined and endorsed by a group of major computer hardware and software vendors
- **Lean and Mean:**
  - Establish a simple and limited set of directives for programming shared memory machines.
  - Significant parallelism can be implemented by using just 3 or 4 directives.

  ☹This goal is becoming less meaningful with each new release, apparently.
- **Ease of Use:**
  - Provide capability to *incrementally* parallelize a serial program, unlike message-passing libraries which typically require an all or nothing approach
  - Provide the capability to implement both coarse-grain and fine-grain parallelism
- **Portability:**
  - The API is specified for C/C++ and Fortran
  - Public forum for API and membership
  - Most major platforms have been implemented, including Unix/Linux platforms and Windows

OpenMP History

- In the early 90's, vendors supplied, similar but diverging, directive-based Fortran programming extensions for shared-memory machines
- In 1994, first attempt at a standard was the draft for ANSI X3H5 - never adopted as distributed memory machines became popular
- Not long after this, newer shared memory machines became prevalent
- In the spring of 1997, the OpenMP standard specification took over where ANSI X3H5 had left off
- Led by the OpenMP Architecture Review Board (ARB)

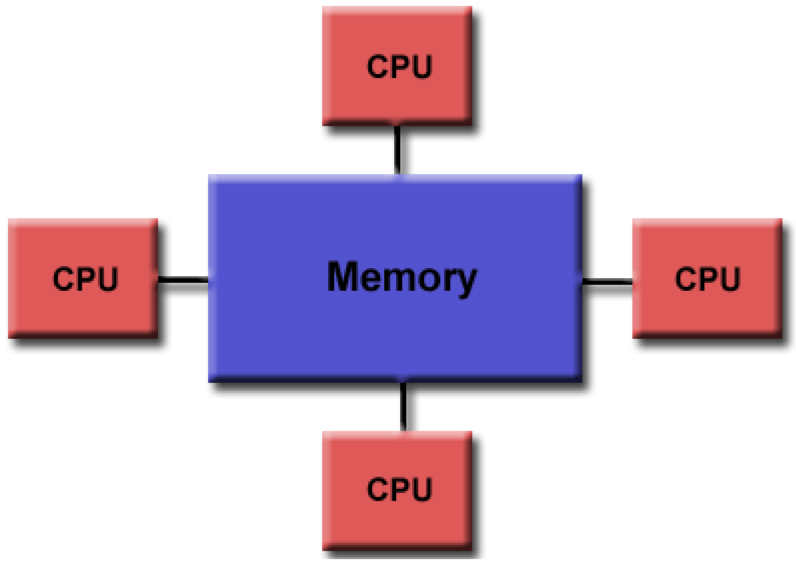
Month/Year	**Version**
Oct 1997	Fortran 1.0
Oct 1998	C/C++ 1.0
Nov 1999	Fortran 1.1
Nov 2000	Fortran 2.0
Mar 2002	C/C++ 2.0
May 2005	OpenMP 2.5
May 2008	OpenMP 3.0
Jul 2011	OpenMP 3.1
Jul 2013	OpenMP 4.0
Nov 2015	OpenMP 4.5

OpenMP 4.x

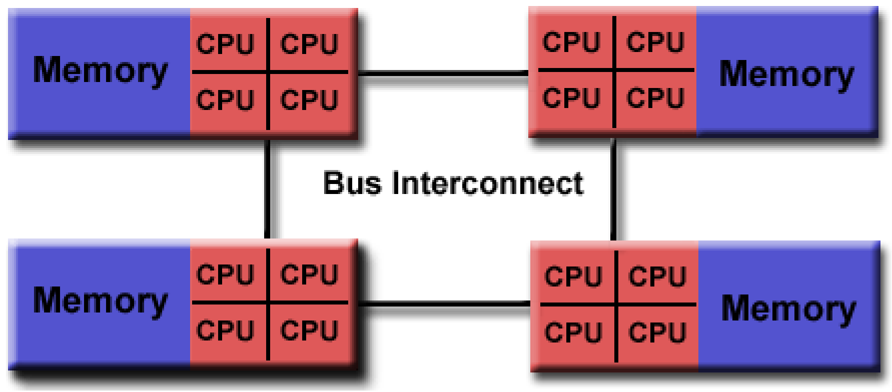
- OpenMP 4.x added support for:
  - Programming of accelerators and GPU devices
  - SIMD programming
  - Better optimization using thread affinity
  - Parallelization of loops with well-structured dependencies
- **Note**: this lecture mostly covers OpenMP version 3.1

Shared Memory Model

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



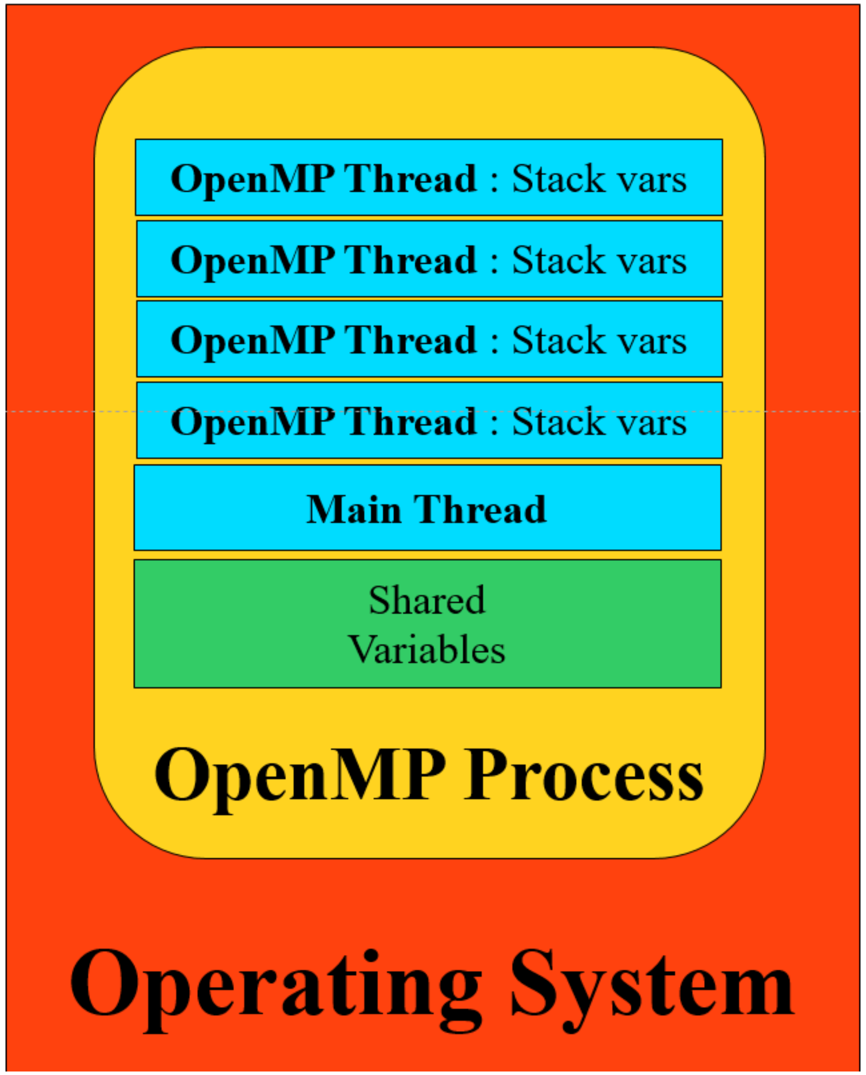
Uniform Memory Access



Non-Uniform Memory Access

Thread Based Parallelism

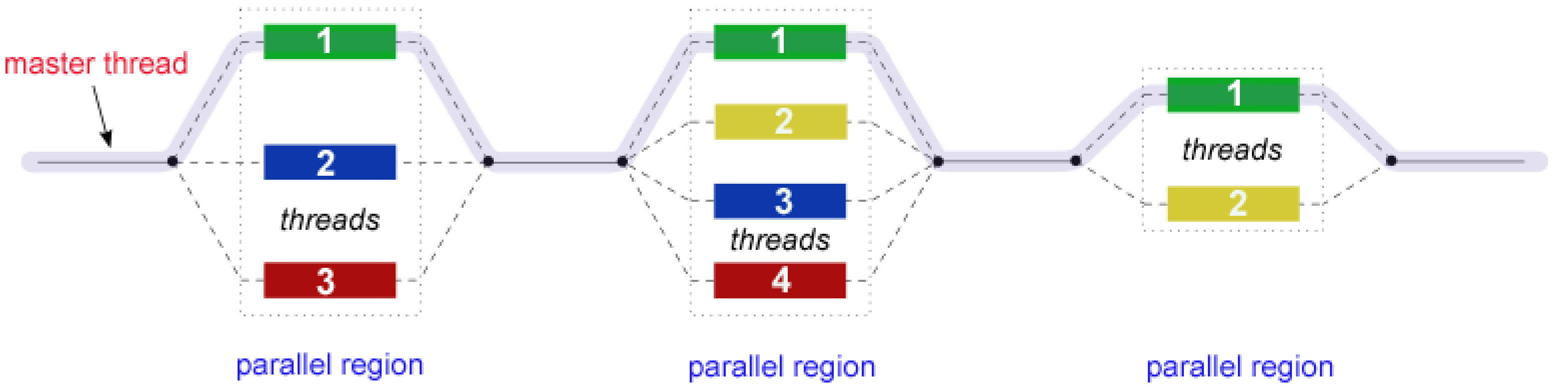
- OpenMP programs accomplish parallelism exclusively through the use of ***threads***
- A thread of execution is the smallest unit of processing that can be scheduled by an operating system
- Threads exist within the resources of a single process
- Typically, the number of threads match the number of machine processors/cores. However, the actual use of threads is up to the application


OpenMP Thread : Stack vars
OpenMP Thread : Stack vars
OpenMP Thread : Stack vars
OpenMP Thread : Stack vars
Main Thread
Shared Variables
OpenMP Process
Operating System

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.

OpenMP Programming Model

- Most OpenMP parallelism is specified through the use of compiler directives embedded in the source code
- The API allows nested parallelism (parallel regions inside other parallel regions)
- The API provides for the runtime environment to dynamically alter the number of threads used to execute parallel regions
- OpenMP specifies nothing about parallel I/O – important if multiple threads attempt to write/read from the same file
- OpenMP provides a "relaxed-consistency" and "temporary" view of thread memory – it is the programmer’s responsibility to insure a variable is FLUSHed by all threads as needed (FLUSH often ☹)

OpenMP API Overview

- The OpenMP API is comprised of three distinct components. As of version 4.0:
  - Compiler Directives (44)
  - Runtime Library Routines (35)
  - Environment Variables (13)
- The application developer decides how to employ these components. In the simplest case, only a few of them are needed.

Compiler Directives

- Compiler directives appear as comments in your source code and are ignored by compilers unless you tell them otherwise
- 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, ...]
```

e.g.:

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

Runtime Library Routines

- The OpenMP API includes an ever-growing number of runtime library routines, 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

e.g.:

Fortran	`INTEGER FUNCTION OMP_GET_NUM_THREADS()`

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

Environment Variables

- OpenMP provides several environment variables for controlling the execution of parallel code at runtime:
  - 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

e.g.:

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

Example OpenMP Code Structure

Fortran:

```
       PROGRAM HELLO

       INTEGER VAR1, VAR2, VAR3

        Serial code
              .
              .
              .

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

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

        Parallel section executed by all threads
              .
        Other OpenMP directives
              .
        Run-time Library calls
              .
        All threads join master thread and disband

!$OMP END PARALLEL

        Resume serial code
              .
              .
              .

        END
```

C/C++:

```
#include <omp.h>

main ()  {

int var1, var2, var3;

Serial code
      .
      .
      .

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

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

  Parallel section executed by all threads
           .
  Other OpenMP directives
           .
  Run-time Library calls
           .
  All threads join master thread and disband

  }

Resume serial code
      ...
}
```

OpenMP Compiler Support on NERSC

Compiler Suite	Module	Option
Intel Compilers (default)	PrgEnv-intel	`-qopenmp`
Cray Compilers	PrgEnv-cray	none needed; OpenMP is the default
GNU Compilers	PrgEnv-gnu	`-fopenmp`

Each compute node of Edison or Cori Phase I is a NUMA shared memory machine

http://www.nersc.gov/users/computational-systems/edison/programming/using-openmp/

OpenMP Compiler Support on Hyades

Compiler	Version	Supports
Intel C/C++, Fortran	14.0	OpenMP 3.1
Intel C/C++, Fortran	15.0	OpenMP 4.0
Intel C/C++, Fortran	16.0	OpenMP 4.0
PGI C/C++, Fortran	13.10	OpenMP 3.1
PGI C/C++, Fortran	14.10	OpenMP 3.1
PGI C/C++, Fortran	15.10	OpenMP 3.1
GNU C/C++, Fortran	4.4	OpenMP 3.0
GNU C/C++, Fortran	4.9	OpenMP 4.0

http://openmp.org/wp/openmp-compilers/

Compiling OpenMP Programs on Hyades

Platform	Compiler	Flag
Intel Compilers	icc icpc ifort	-qopenmp
PGI Compilers	pgcc pgCC / pgcpp pgfortran pgf77 pgf90 / pgf95	-mp
GNU Compiler Collection	gcc g++ gfortran	-fopenmp

Compiler Documentation

- Intel Compilers (default and recommended)
    - Intel C++ Compiler 16.0 User and Reference Guide: https://software.intel.com/en-us/INTEL-CPLUSPLUS-COMPILER-16.0-USER-AND-REFERENCE-GUIDE
    - Intel Fortran Compiler 16.0 User and Reference Guide: https://software.intel.com/en-us/intel-fortran-compiler-16.0-user-and-reference-guide
- PGI Compilers
    - PGI documentation: https://www.pgroup.com/resources/docs.htm
    - PGI Compiler User's Guide: https://www.pgroup.com/doc/pgiug.pdf
    - PGI Compiler Reference Manual: https://www.pgroup.com/doc/pgiref.pdf
- GCC (GNU Compiler Collection)
    - GCC documentation: https://gcc.gnu.org/onlinedocs/
    - GNU Fortran Compiler: https://gcc.gnu.org/onlinedocs/gfortran/

OpenMP “Hello, world!” in Fortran 77

```
      PROGRAM HELLO

      implicit none
      integer nthreads, tid, OMP_GET_THREAD_NUM,
     +        OMP_GET_NUM_THREADS

C$OMP PARALLEL PRIVATE(tid)
      tid = OMP_GET_THREAD_NUM()
      print *, "Hello, world! I am thread ", tid
C$OMP BARRIER
      if (tid .eq. 0) then
        nthreads = OMP_GET_NUM_THREADS()
        print *, "Number of threads = ", nthreads
      end if
C$OMP END PARALLEL

      end
```

- Fortran fixed form
- !$OMP C$OMP *$OMP are accepted sentinels and must start in column 1

Compiler directives

Runtime library routines

PARALLEL region

OpenMP “Hello, world!” in Fortran 90

```
program hello
  use omp_lib
  implicit none

  integer :: nthreads, tid

  !$OMP PARALLEL PRIVATE(tid)
  tid = OMP_GET_THREAD_NUM()
  print *, "Hello, world! I am thread ", tid

  !$OMP BARRIER

  if (tid .eq. 0) then
    nthreads = OMP_GET_NUM_THREADS()
    print *, "Number of threads = ", nthreads
  end if
  !$OMP END PARALLEL
end program hello
```

- Fortran free form
- !$OMP is the only accepted sentinel and can appear in any column

Compiler directives

Runtime library routines

PARALLEL region

OpenMP "Hello, world!" in C/C++

```
#include <stdio.h>
#include <omp.h>
int main()
{
  int nthreads, tid;
  #pragma omp parallel private(tid)
  {
    tid = omp_get_thread_num();
    printf("Hello, world! I am thread %d\n", tid);
    #pragma omp barrier
    if (tid == 0)
    {
      nthreads = omp_get_num_threads();
      printf("Number of threads = %d\n",nthreads);
    }
  }
  return 0;
}
```

- Case sensitive
- Each directive applies to at most one succeeding statement, which must be a structured block

Compiler directives

Runtime library routines

PARALLEL region

Compiling & Running OpenMP Programs on NERSC

Compile OpenMP programs using the *default* Intel compilers:

```
ftn -qopenmp [other options] omp_hello.f -o omp_hello.x
ftn -qopenmp [other options] omp_hello.f90 -o omp_hello.x
cc -qopenmp [other options] omp_hello.c -o omp_hello.x
CC -qopenmp [other options] omp_hello.cpp -o omp_hello.x
```

(Test) run OpenMP programs on the login node:

```
export OMP_NUM_THREADS=4 (sh/bash)
setenv OMP_NUM_THREADS 4 (csh/tcsh)
./omp_hello.x
```

For production runs, submit your OpenMP jobs to the batch scheduler.

Sample Batch Script for OpenMP Jobs on Edison

Batch script *omp.slurm:*

```
#!/bin/bash -l

#SBATCH -J omp
#SBATCH -p regular
#SBATCH -N 1
#SBATCH -t 4:00:00
#SBATCH -L SCRATCH
#SBATCH --mail-user=shaw@ucsc.edu
#SBATCH --mail-type=ALL

export OMP_NUM_THREADS=24
./omp_hello.x
```

Comments:

```
### your favorite shell

### job name
### job queue
### request 1 node (24 cores)
### and 4 hours walltime
### request a license for scratch file system
### ask SLURM to send emails
### when jobs aborts, starts and ends

### set the maximum no. of OpenMP threads to 24
### run your OpenMP executable
```

Or: `srun -n 1 -c 24 ./omp_hello.x`

To submit the job:

```
sbatch omp.slurm
```

http://www.nersc.gov/users/computational-systems/cori/running-jobs/example-batch-scripts/

Sample Batch Script for OpenMP Jobs on Cori Phase I

Batch script *omp.slurm:*

```
#!/bin/bash -l

#SBATCH -J omp
#SBATCH -p regular
#SBATCH -c haswell
#SBATCH -N 1
#SBATCH -t 4:00:00
#SBATCH -L SCRATCH
#SBATCH --mail-user=shaw@ucsc.edu
#SBATCH --mail-type=ALL

export OMP_NUM_THREADS=32
./omp_hello.x
```

Comments:

```
### your favorite shell

### job name
### job queue
### request a Haswell node
### request 1 node (32 cores)
### and 4 hours walltime
### request a license for scratch file system
### ask SLURM to send emails
### when jobs aborts, starts and ends

### set the maximum no. of OpenMP threads to 32
### run your OpenMP executable
```

Or: `srun -n 1 -c 32 ./omp_hello.x`

To submit the job:

```
sbatch omp.slurm
```

http://www.nersc.gov/users/computational-systems/cori/running-jobs/example-batch-scripts/

Compiling & Running OpenMP Programs on Hyades

Compile OpenMP programs using the *default* Intel compilers:

```
ifort -qopenmp [other options] omp_hello.f -o omp_hello.x
ifort -qopenmp [other options] omp_hello.f90 -o omp_hello.x
icc -qopenmp [other options] omp_hello.c -o omp_hello.x
icpc -qopenmp [other options] omp_hello.cpp -o omp_hello.x
```

(Test) run OpenMP programs on the master node:

```
export OMP_NUM_THREADS=4 (sh/bash)
setenv OMP_NUM_THREADS 4 (csh/tcsh)
./omp_hello.x
```

For production runs, submit your OpenMP jobs to the batch scheduler.

Sample Batch Script for OpenMP Jobs on Hyades

Batch script *omp.pbs:*

```
#!/bin/bash

#PBS -N omp
#PBS -q normal
#PBS -l nodes=1:ppn=16
#PBS -l walltime=4:00:00
#PBS -M shaw@ucsc.edu
#PBS -m abe

export OMP_NUM_THREADS=16
cd $PBS_O_WORkDIR
./omp_hello.x
```

Comments:

```
### your favorite shell

### job name
### job queue
### request 1 node (16 cores)
### and 4 hours walltime
### ask Torque to send emails
### when jobs aborts, starts and ends

### set the maximum no. of OpenMP threads to 16
### go to the directory where you submit the job
### run your OpenMP executable
```

To submit the job:

```
qsub omp.pbs
```

OpenMP Directives

- PARALLEL Region Construct
- Work-Sharing Constructs
  - DO / for Directive
  - SECTIONS Directive
  - WORKSHARE Directive
  - SINGLE DIRECTIVE
- Combined Parallel Work-Sharing Constructs
- TASK Construct
- Synchronization Constructs – *MASTER, CRITICAL, BARRIER, TASKWAIT, ATOMIC, FLUSH, ORDERED*
- THREADPRIVATE Directive

OpenMP Directives – PARALLEL Region Construct

Parallel region: block of code that will be executed by multiple threads

Fortran

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

C/C++

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

PARALLEL Region Construct (cont’d)

- When a thread reaches a PARALLEL directive, it creates a team of threads and becomes the master of the team. The master is a member of that team and has thread number 0 within that team.
- Starting from the beginning of this parallel region, the code is duplicated and all threads will execute that code.
- There is an implied barrier at the end of a parallel section. Only the master thread continues execution past this point.
- If any thread terminates within a parallel region, all threads in the team will terminate, and the work done up until that point is undefined.
- 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.

PARALLEL Region Construct (cont'd)

- The number of threads in a parallel region is determined by the following factors, in order of precedence:
    1. Evaluation of the **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
    2. Setting of the **NUM_THREADS** *clause*
    3. Use of the **omp_set_num_threads()** library function
    4. Setting of the **OMP_NUM_THREADS** environment variable
    5. Implementation default - usually the number of CPUs/cores on a node (but for PGI compilers, the default is 1 thread!)
- Threads are numbered from 0 (master thread) to N-1
- Data Scope Attribute Clauses, like *private* and *shared*, will be described in details shortly

OpenMP “Hello, world!” in C/C++

```
#include <stdio.h>
#include <omp.h>
int main()
{
  int nthreads, tid;
  #pragma omp parallel private(tid)
  {
    tid = omp_get_thread_num();
    printf("Hello, world! I am thread %d\n", tid);
    #pragma omp barrier
    if (tid == 0)
    {
      nthreads = omp_get_num_threads();
      printf("Number of threads = %d\n",nthreads);
    }
  }
  return 0;
}
```

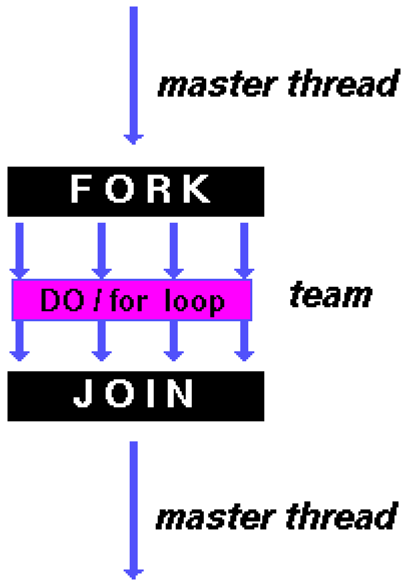
parallel region

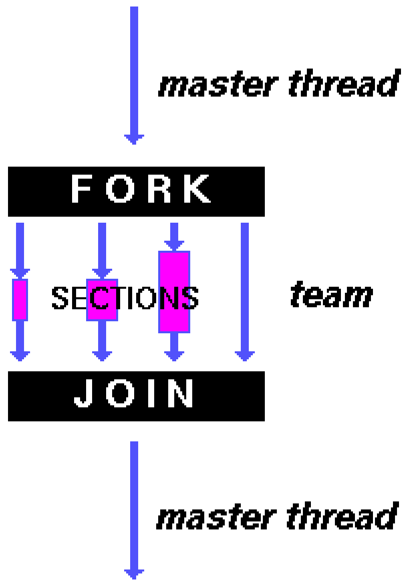
OpenMP Directives – Work-Sharing Constructs

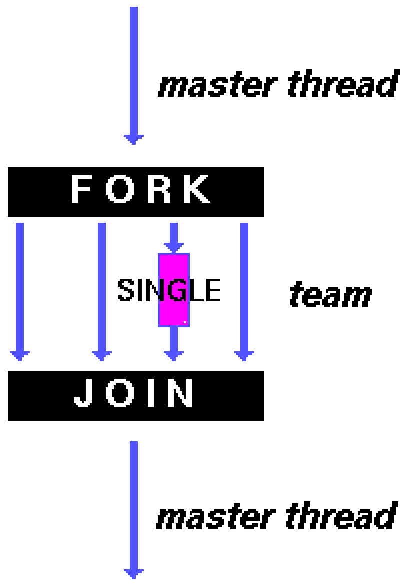
- A work-sharing construct *divides* the execution of the enclosed code region among the members of the team that encounter it.
- Work-sharing constructs do not launch new threads
- There is no implied barrier upon entry to a work-sharing construct, however there is an implied barrier at the end of a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

Types of Work-Sharing Constructs

DO / for - shares iterations of a loop across the team. Represents a type of "data parallelism".	**SECTIONS -** breaks work into separate, discrete sections. Each section is executed by a thread. Can be used to implement a type of "functional parallelism".	**SINGLE** - serializes a section of code







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

DO / for Directive

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.

Fortran

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

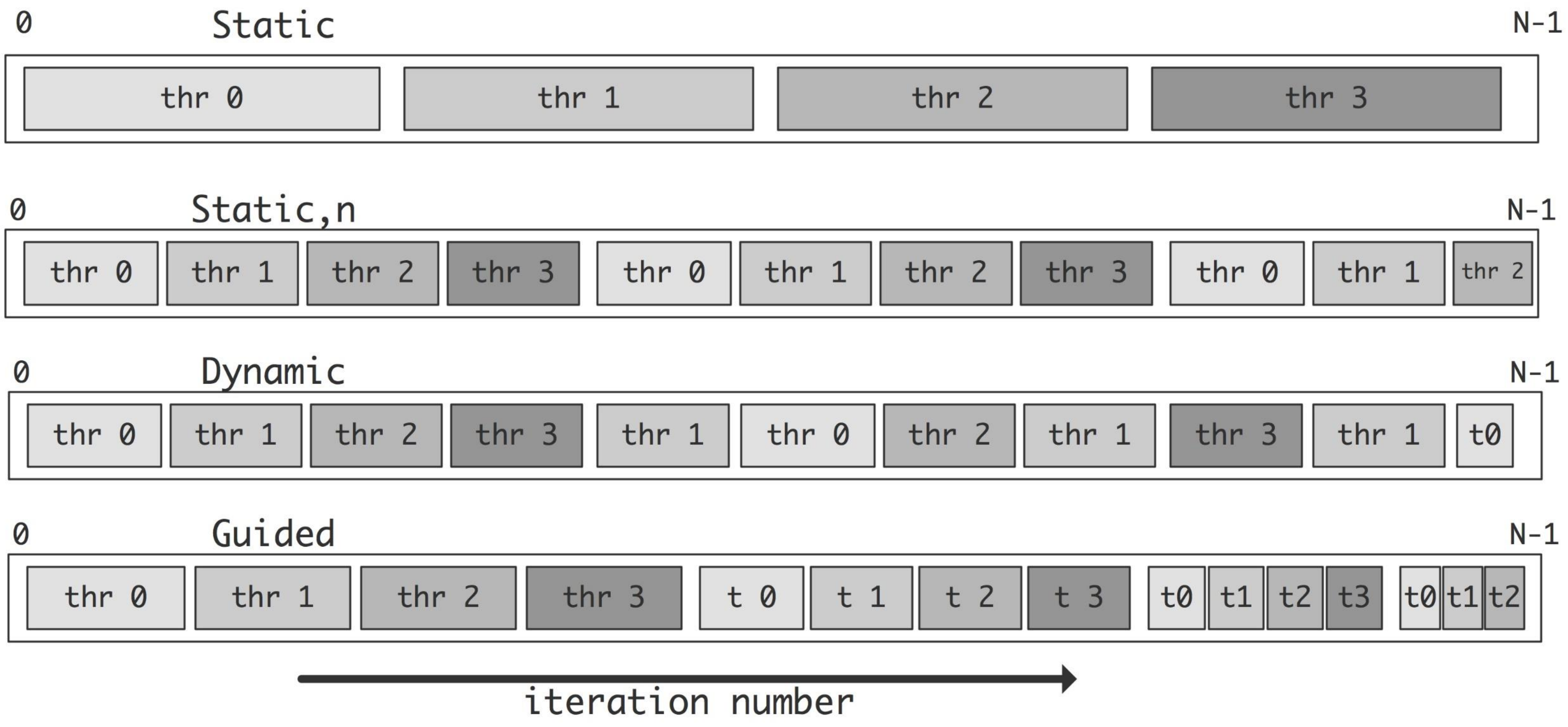
C/C++

```
#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
```

DO / for Directive Clauses

- **SCHEDULE**: Describes how iterations of the loop are divided among the threads in the team
  - **STATIC**: Loop iterations are divided into pieces of size *chunk* and then statically assigned to 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 *chunk* parameter defines the minimum block size. The default *chunk* size is 1.
  - **RUNTIME**: The scheduling decision is deferred until runtime by the environment variable **OMP_SCHEDULE**. It is *illegal* to specify a chunk size for this clause.
  - **AUTO**: The scheduling decision is delegated to the compiler and/or runtime system.

OpenMP Schedules



DO / for Directive Clauses (cont’d)

- **NOWAIT**: If specified, then threads do not synchronize at the end of the parallel loop.
- **ORDERED**: Specifies that the iterations of the loop must be executed as they would be in a serial program.
- **COLLAPSE**: Specifies how many loops in a nested loop should be collapsed into one large iteration space and divided according to the schedule clause. The sequential execution of the iterations in all associated loops determines the order of the iterations in the collapsed iteration space.

DO Directive Example in Fortran

```fortran
      PROGRAM VEC_ADD_DO
      INTEGER N, CHUNKSIZE, CHUNK, I
      PARAMETER (N=1000)
      PARAMETER (CHUNKSIZE=100)
      REAL A(N), B(N), C(N)
!     initializations omitted
      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
```

DO Directive

PARALLEL region

- 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).

for Directive Example in C/C++

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

int main ()
{
  int i, chunk;
  float a[N], b[N], c[N];
  /* initializations omitted */
  chunk = CHUNKSIZE;
  #pragma omp parallel shared(a,b,c,chunk) private(i)
  {
    #pragma omp for schedule(dynamic,chunk) nowait
    for (i=0; i < N; i++)
      c[i] = a[i] + b[i];
  }
  return 0;
}
```

for directive

parallel region

Which is better?

A

```
DO i=1,100
  DO j = 1,100
!$OMP DO
    DO k = 1,100
      A(i,j,k)=i*j*k
    END DO
!$OMP END DO
  END DO
END DO
```

B

```
!$OMP DO
DO i=1,100
  DO j = 1,100
    DO k = 1,100
      A(i,j,k)=i*j*k
    END DO
  END DO
END DO
!$OMP END DO
```

Answer: **B**

1. A lot more work per thread
2. Less creation/destruction of threads, thus less overhead

Can we do even better?

```
DO i=1,100
  DO j = 1,100
!$OMP DO
    DO k = 1,100
      A(i,j,k)=i*j*k
    END DO
!$OMP END DO
  END DO
END DO
```

```
!$OMP DO
DO i=1,100
  DO j = 1,100
    DO k = 1,100
      A(i,j,k)=i*j*k
    END DO
  END DO
END DO
!$OMP END DO
```

```
!$OMP DO
DO k=1,100
  DO j = 1,100
    DO i = 1,100
      A(i,j,k)=i*j*k
    END DO
  END DO
END DO
!$OMP END DO
```

Fortran arrays are stored in **column-major** format (C/C++ is row-major)
i.e., columns (first dimension) are contiguous in memory
Better cache performance
DO THIS ALWAYS FOR FORTRAN!

SECTIONS Directive

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.

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
}
```

SECTIONS Directive Example in Fortran

```
      PROGRAM VEC_ADD_SECTIONS
      INTEGER N, I
      PARAMETER (N=1000)
      REAL A(N), B(N), C(N), D(N)
!     initializations omitted
!$OMP PARALLEL SHARED(A,B,C,D), PRIVATE(I)
!$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
```

SECTION Directive

SECTION Directive

SECTIONS Directive

PARALLEL Region

sections Directive Example in C/C++

```c
#include <omp.h>
#define N     1000
int main () {
  int i;
  float a[N], b[N], c[N], d[N];
  /* initializations omitted */
  #pragma omp parallel shared(a,b,c,d) private(i)
  {
    #pragma omp sections nowait
    {
      #pragma omp section
      for (i=0; i < N; i++)
        c[i] = a[i] + b[i];
      #pragma omp section
      for (i=0; i < N; i++)
        d[i] = a[i] * b[i];
    }
  }
  return 0;
}
```

section directive

section directive

sections directive

parallel region

WORKSHARE Directive

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

Fortran	`!$OMP WORKSHARE`
	    `structured block`
	`!$OMP WORKSHARE [ NOWAIT ]`

WORKSHARE Directive Example in Fortran

```
      PROGRAM WORKSHARE
      INTEGER N, I, J
      PARAMETER (N=100)
      REAL AA(N,N), BB(N,N), CC(N,N), DD(N,N)
      REAL FIRST, LAST
!     initializations omitted
!$OMP PARALLEL SHARED(AA,BB,CC,DD,FIRST,LAST)

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

!$OMP END PARALLEL

      END
```

- Array assignments - the assignment of each element is a unit of work
- Scalar assignments
- Block of code is parallelized sequentially (!), unit by unit (**note**: incurs overhead)
- Variables which are referenced or modified within construct MUST be *shared* variables

SINGLE Directive

- 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).

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
```

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.

PARALLEL DO Directive Example in Fortran

```
      PROGRAM VEC_ADD_DO
      INTEGER N, CHUNKSIZE, CHUNK, I
      PARAMETER (N=1000)
      PARAMETER (CHUNKSIZE=100)
      REAL A(N), B(N), C(N)
!     initializations omitted
      CHUNK = CHUNKSIZE
!$OMP PARALLEL DO
!$OMP& SHARED(A,B,C,CHUNK) PRIVATE(I)
!$OMP& SCHEDULE(DYNAMIC,CHUNK)

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

!$OMP END PARALLEL DO

      END
```

vs. **DO** directive in a **PARALLEL** region

```
      PROGRAM VEC_ADD_DO
      INTEGER N, CHUNKSIZE, CHUNK, I
      PARAMETER (N=1000)
      PARAMETER (CHUNKSIZE=100)
      REAL A(N), B(N), C(N)
!     initializations omitted
      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
```

parallel for Directive Example in C/C++

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

int main ()
{
  int i, chunk;
  float a[N], b[N], c[N];
  /* initializations omitted */
  chunk = CHUNKSIZE;
  #pragma omp parallel for \
    shared(a,b,c,chunk) private(i) \
    schedule(dynamic,chunk)
    for (i=0; i < N; i++)
      c[i] = a[i] + b[i];

  return 0;
}
```

vs. **for** directive in a **parallel** region

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

int main ()
{
  int i, chunk;
  float a[N], b[N], c[N];
  /* initializations omitted */
  chunk = CHUNKSIZE;
  #pragma omp parallel \
    shared(a,b,c,chunk) private(i)
  {
    #pragma omp for schedule(dynamic,chunk) \
      nowait
    for (i=0; i < N; i++)
      c[i] = a[i] + b[i];
  }
  return 0;
}
```

Quick Notes on Directives Format - Fortran

- Fixed Form Fortran
  - **!$OMP C$OMP *$OMP** are accepted sentinels and must start in column 1.
  - Initial directive lines must have a space/zero in column 6.
  - Continuation lines must have a non-space/zero in column 6.

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

- Free Form Fortran
  - !$OMP is the only accepted sentinel. Can appear in any column, but must be preceded by white space only.
  - 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.

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

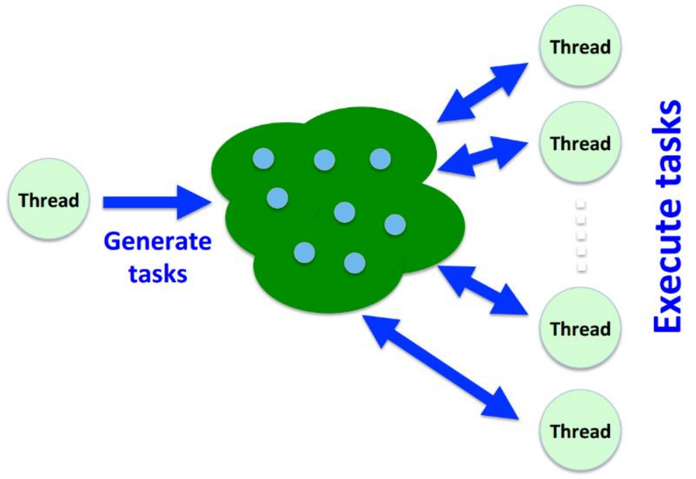
Quick Notes on Directives Format – C/C++

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

```
#pragma omp parallel for \
   shared(a,b,c,chunk) private(i) \
   schedule(dynamic,chunk)
```

TASK Construct

- Introduced in OpenMP 3.0
- When a thread encounters a **TASK** construct, a new task is generated
- The moment of execution of the task is up to the runtime system
- Execution can either be immediate or delayed
- The data environment of the task is determined by the data sharing attribute clauses.
- Completion of a task can be enforced through **task synchronization**



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`

TASK Construct Example in C/C++

```c
#include <stdio.h>
#include <omp.h>
int main () {
  #pragma omp parallel
  {
    #pragma omp single
    {
      printf("A ");
      #pragma omp task
      {printf("race ");}
      #pragma omp task
      {printf("car ");}
    }
  } // End of parallel region

  printf("\n");
  return 0;
}
```

```
$ cc -qopenmp omp_task.c -o omp_task.x
$ export OMP_NUM_THREADS=2
$ ./omp_task.x
A race car
$ ./omp_task.x
A race car
$ ./omp_task.x
A car race
```

http://openmp.org/sc13/sc13.tasking.ruud.pdf

TASK Example: Computing Fibonacci Numbers

```c
#include <stdio.h>
#include <omp.h>
int fib(int n) {
  int i, j;
  if (n<2)
    return n;
  else {
    #pragma omp task shared(i) firstprivate(n)
    i=fib(n-1);
    #pragma omp task shared(j) firstprivate(n)
    j=fib(n-2);
    #pragma omp taskwait
    return i+j;
  }
}

int main()
{
  int n = 10;
  #pragma omp parallel shared(n)
  {
    #pragma omp single
    printf ("fib(%d) = %d\n", n, fib(n));
  }
  return 0;
}
```

Explicitly wait on the completion of child tasks

Although only one thread executes the single directive, all team threads will participate in executing the tasks generated.

Synchronization Constructs

- Consider a simple example where two threads on two different processors are both trying to increment a variable x at the same time (assume x is initially 0)
- One possible execution sequence:
  1. Thread 1 loads the value of x into register A
  2. Thread 2 loads the value of x into register A
  3. Thread 1 adds 1 to register A
  4. Thread 2 adds 1 to register A
  5. Thread 1 stores register A at location x
  6. Thread 2 stores register A at location x

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

THREAD 1:

```
increment(x)
{
  x = x + 1;
}
```

THREAD 2:

```
increment(x)
{
  x = x + 1;
}
```

THREAD 1:

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

THREAD 2:

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

- To avoid a situation like this, the incrementing of x must be synchronized between the two threads to ensure that the correct result is produced.

OpenMP Synchronization Constructs

- Master Directive
- CRITICAL Directive
- BARRIER Directive
- TASKWAIT Directive
- ATOMIC Directive
- FLUSH Directive
- ORDERED Directive

MASTER Directive

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

Fortran	`!$OMP MASTER`
	
	`   block`
	
	`!$OMP END MASTER`

C/C++	`#pragma omp master newline`
	
	`   structured_block`

CRITICAL Directive

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

Fortran	`!$OMP CRITICAL [ name ]`
	`   block`
	`!$OMP END CRITICAL [ name ]`

C/C++	`#pragma omp critical [ name ] newline`
	`   structured_block`

CRITICAL Directive Example

Fortran:

```
      PROGRAM CRITICAL

      INTEGER X
      X = 0

!$OMP PARALLEL SHARED(X)

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

!$OMP END PARALLEL

      END
```

C/C++:

```
#include <omp.h>

int main()
{
  int x;
  x = 0;

  #pragma omp parallel shared(x)
  {
    #pragma omp critical
    x = x + 1;
  }  /* end of parallel section */
  return 0;
}
```

BARRIER Directive

- 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.

Fortran	`!$OMP BARRIER`
C/C++	`#pragma omp barrier` *`newline`*

TASKWAIT Directive

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

Fortran	`!$OMP TASKWAIT`
C/C++	`#pragma omp taskwait` *`newline`*

ATOMIC Directive

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

Fortran	`!$OMP ATOMIC`
	   `statement_expression`

C/C++	`#pragma omp atomic` *`newline`*
	   `statement_expression`

FLUSH Directive

- 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.

Fortran	`!$OMP FLUSH (list)`
C/C++	`#pragma omp flush (list)` *`newline`*

- The FLUSH directive is implied for the directives shown in the table below (FLUSH often). The directive is not implied if a NOWAIT clause is present.

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

ORDERED Directive

- 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
- Only one thread is allowed in an ordered section at any time
- Used within a DO / for loop with an ORDERED *clause*

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`
	`  (end of loop region)`

http://stackoverflow.com/questions/13224155/how-does-the-omp-ordered-clause-work

THREADPRIVATE Directive

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

Fortran	`!$OMP THREADPRIVATE (/cb/, ...)`
C/C++	`#pragma omp threadprivate` *`(list) newline`*

Note: *cb* is the name of a common block

THREADPRIVATE Directive Example in Fortran

```
      PROGRAM THREADPRIV
      INTEGER A, B, I, TID, OMP_GET_THREAD_NUM
      REAL*4 X
      COMMON /C1/ A
      SAVE X
!$OMP THREADPRIVATE(/C1/, X)
C     Explicitly turn off dynamic threads
      CALL OMP_SET_DYNAMIC(.FALSE.)
      PRINT *, '1st Parallel Region:'
!$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
!$OMP END PARALLEL
      PRINT *, '************************************'
      PRINT *, 'Master thread doing serial work here'
      PRINT *, '************************************'
      PRINT *, '2nd Parallel Region: '
!$OMP PARALLEL PRIVATE(TID)
      TID = OMP_GET_THREAD_NUM()
      PRINT *, 'Thread',TID,':   A,B,X=',A,B,X
!$OMP END PARALLEL
      END
```

X must have the SAVE attribute

THREADPRIVATE variables (X & A) persist between different parallel sections

B doesn’t persist between different parallel sections

B is undefined in the 2nd parallel region

Try it out

```
$ ftn -qopenmp threadpriv.f -o threadpriv.x
$ export OMP_NUM_THREADS=4
$ ./threadpriv.x
 1st Parallel Region:
 Thread           0 :   A,B,X=            0           0   1.000000
 Thread           3 :   A,B,X=            3           3   4.300000
 Thread           2 :   A,B,X=            2           2   3.200000
 Thread           1 :   A,B,X=            1           1   2.100000
 ************************************
 Master thread doing serial work here
 ************************************
 2nd Parallel Region:
 Thread           3 :   A,B,X=            3       40896   4.300000
 Thread           0 :   A,B,X=            0       40896   1.000000
 Thread           1 :   A,B,X=            1       40896   2.100000
 Thread           2 :   A,B,X=            2       40896   3.200000
```

THREADPRIVATE Directive Example in C/C++

```c
#include <omp.h>
int  a, b, i, tid; float x;
#pragma omp threadprivate(a, x)
int main () {
  /* Explicitly turn off dynamic threads */
  omp_set_dynamic(0);
  printf("1st Parallel Region:\n");
  #pragma omp parallel private(b,tid)
  {
    tid = omp_get_thread_num();
    a = tid;
    b = tid;
    x = 1.1 * tid +1.0;
    printf("Thread %d:   a,b,x= %d %d %f\n",tid,a,b,x);
  }  /* end of parallel section */
  printf("************************************\n");
  printf("Master thread doing serial work here\n");
  printf("************************************\n");
  printf("2nd Parallel Region:\n");
  #pragma omp parallel private(tid)
  {
    tid = omp_get_thread_num();
    printf("Thread %d:   a,b,x= %d %d %f\n",tid,a,b,x);
  }  /* end of parallel section */
  return 0;
}
```

THREADPRIVATE variables (a & x) persist between different parallel sections

b doesn't persist between different parallel sections

b is undefined in the 2nd parallel region

Try it out

```
$ cc -qopenmp threadpriv.c -o threadpriv.x
$ export OMP_NUM_THREADS=4
$ ./threadpriv.x
1st Parallel Region:
Thread 0:   a,b,x= 0 0 1.000000
Thread 1:   a,b,x= 1 1 2.100000
Thread 2:   a,b,x= 2 2 3.200000
Thread 3:   a,b,x= 3 3 4.300000
************************************
Master thread doing serial work here
************************************
2nd Parallel Region:
Thread 1:   a,b,x= 1 0 2.100000
Thread 3:   a,b,x= 3 0 4.300000
Thread 0:   a,b,x= 0 0 1.000000
Thread 2:   a,b,x= 2 0 3.200000
```

Directive Scoping

- Static (Lexical) Extent:
  - The code textually enclosed between the beginning and the end of a structured block following a directive.
  - The static extent of a directives does not span multiple routines or code files
- Orphaned Directive:
  - An OpenMP directive that appears independently from another enclosing directive is an orphaned directive. It exists outside of another directive's static (lexical) extent.
  - Will span routines and possibly code files
- Dynamic:
  - The dynamic extent of a directive includes both its static (lexical) extent and the extents of its orphaned directives.
- OpenMP specifies a number of scoping rules on how directives may associate (bind) and nest within each other

Directive Scoping Example

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

TASK Example: Computing Fibonacci Numbers

```
#include <stdio.h>
int fib(int n) {
  int i, j;
  if (n<2)
    return n;
  else {
    #pragma omp task shared(i) firstprivate(n)
    i=fib(n-1);
    #pragma omp task shared(j) firstprivate(n)
    j=fib(n-2);
    #pragma omp taskwait
    return i+j;
  }
}

int main()
{
  int n = 10;
  #pragma omp parallel shared(n)
  {
    #pragma omp single
    printf ("fib(%d) = %d\n", n, fib(n));
  }
  return 0;
}
```

Orphaned Directives

Dynamic Extent

Static Extent

Data Scope Attribute Clauses

- PRIVATE Clause
- SHARED Clause
- DEFAULT Clause
- FIRSTPRIVATE Clause
- LASTPRIVATE Clause
- COPYIN Clause
- COPYPRIVATE Clause
- REDUCTION Clause

PRIVATE Clause

- The PRIVATE clause declares variables in its list to be private to each thread.
- PRIVATE variables behave as follows:
  - A new object of the same type is declared once for each thread in the team
  - All references to the original object are replaced with references to the new object
  - Variables declared PRIVATE should be assumed to be uninitialized for each thread
- Format:

Fortran	`PRIVATE` *(list)*
C/C++	`private` *(list)*

PRIVATE vs. THREADPRIVATE

	PRIVATE	**THREADPRIVATE**
Data Item	C/C++: variable Fortran: variable or common block	C/C++: variable Fortran: common block
Where Declared	At start of region or work-sharing group	In declarations of each routine using block or global file scope
Persistent?	No	Yes
Extent	Lexical only - unless passed as an argument to subroutine	Dynamic
Initialized	Use **FIRSTPRIVATE**	Use **COPYIN**

SHARED Clause

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

Fortran	`SHARED` *(list)*
C/C++	`shared` *(list)*

DEFAULT Clause

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

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

FIRSTPRIVATE Clause

- The **FIRSTPRIVATE** clause combines the behavior of the **PRIVATE** clause with automatic initialization of the variables in its list.
- Listed variables are initialized according to the value of their original objects prior to entry into the parallel or work-sharing construct.
- Format:

Fortran	`FIRSTPRIVATE` *`(list)`*
C/C++	`firstprivate` *`(list)`*

LASTPRIVATE Clause

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

Fortran	`LASTPRIVATE` *(list)*
C/C++	`lastprivate` *(list)*

COPYIN Clause

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

Fortran	`COPYIN` *(list)*
C/C++	`copyin` *(list)*

COPYPRIVATE Clause

- 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
- Format:

Fortran	`COPYPRIVATE` *`(list)`*
C/C++	`copyprivate` *`(list)`*

REDUCTION Clause

- The **REDUCTION** clause performs a reduction on the variables that appear in its list.
- A private copy for each list variable is created for each thread. At the end of the reduction, the reduction variable is applied to all private copies of the shared variable, and the final result is written to the global shared variable.
- Format:

Fortran	`REDUCTION` *(operator|intrinsic: list)*
C/C++	`reduction` *(operator: list)*

REDUCTION Clause Example in Fortran

```
       PROGRAM DOT_PRODUCT
       INTEGER N, CHUNKSIZE, CHUNK, I
       PARAMETER (N=100)
       PARAMETER (CHUNKSIZE=10)
       REAL A(N), B(N), RESULT
!      initializations omitted
       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
```

- 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.

REDUCTION Clause Example in C/C++

```
#include <omp.h>
int main () {
  int   i, n, chunk;
  float a[100], b[100], result;
  /* initializations omitted*/
  n = 100;
  chunk = 10;
  result = 0.0;
  #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);
  return 0;
}
```

- 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.

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		●	●	●		

Clauses / Directives Summary (cont'd)

- 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.

Directive Binding Rules

- 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 Rules

- 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.

The term "closely nested region" means a region that is dynamically nested inside another region with no parallel region nested between them.

Runtime Library Routines

- The OpenMP API includes an ever-growing number of run-time library routines.
- For C/C++, all of the run-time library routines are actual subroutines. For Fortran, some are actually functions, and some are subroutines.
- 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.

Routine	Purpose
OMP_SET_NUM_THREADS	Sets the number of threads that will be used in the next parallel region
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	Returns the maximum value that can be returned by a call to the OMP_GET_NUM_THREADS function
OMP_GET_THREAD_NUM	Returns the thread number of the thread, within the team, making this call.
OMP_GET_THREAD_LIMIT	Returns the maximum number of OpenMP threads available to a program
OMP_GET_NUM_PROCS	Returns the number of processors that are available to the program
OMP_IN_PARALLEL	Used to determine if the section of code which is executing is parallel or not
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	Used to determine if dynamic thread adjustment is enabled or not
OMP_SET_NESTED	Used to enable or disable nested parallelism
OMP_GET_NESTED	Used to determine if nested parallelism is enabled or not
OMP_SET_SCHEDULE	Sets the loop scheduling policy when "runtime" is used as the schedule kind in the OpenMP directive
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	Sets the maximum number of nested parallel regions
OMP_GET_MAX_ACTIVE_LEVELS	Returns the maximum number of nested parallel regions
OMP_GET_LEVEL	Returns the current level of nested parallel regions
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	Returns, for a given nested level of the current thread, the size of the thread team
OMP_GET_ACTIVE_LEVEL	Returns the number of nested, active parallel regions enclosing the task that contains the call

OMP_IN_FINAL	Returns true if the routine is executed in the final task region; otherwise it returns false
OMP_INIT_LOCK	Initializes a lock associated with the lock variable
OMP_DESTROY_LOCK	Disassociates the given lock variable from any locks
OMP_SET_LOCK	Acquires ownership of a lock
OMP_UNSET_LOCK	Releases a lock
OMP_TEST_LOCK	Attempts to set a lock, but does not block if the lock is unavailable
OMP_INIT_NEST_LOCK	Initializes a nested lock associated with the lock variable
OMP_DESTROY_NEST_LOCK	Disassociates the given nested lock variable from any locks
OMP_SET_NEST_LOCK	Acquires ownership of a nested lock
OMP_UNSET_NEST_LOCK	Releases a nested lock
OMP_TEST_NEST_LOCK	Attempts to set a nested lock, but does not block if the lock is unavailable
OMP_GET_WTIME	Provides a portable wall clock timing routine
OMP_GET_WTICK	Returns a double-precision floating point value equal to the number of seconds between successive clock ticks

TASK Example: Computing Fibonacci Numbers

```
#include <stdio.h>
#include <omp.h>
int fib(int n) {
  int i, j;
  if (n<2)
    return n;
  else {
    #pragma omp task shared(i) firstprivate(n)
    i=fib(n-1);
    #pragma omp task shared(j) firstprivate(n)
    j=fib(n-2);
    #pragma omp taskwait
    return i+j;
  }
}

int main()
{
  int n = 10;
  #pragma omp parallel shared(n)
  {
    #pragma omp single
    printf ("fib(%d) = %d\n", n, fib(n));
  }
  return 0;
}
```

Inefficient!

Most intermediate values will be computed more than once!

More Efficient Fibonacci Computation

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

int *value, *done;
omp_lock_t *dolock;

int fib(int n) {
  int i, j;
  if (n<2)
    return n;
  else {
    omp_set_lock( &(dolock[n]) );
    if (!done[n]) {
      #pragma omp task shared(i) firstprivate(n)
      i=fib(n-1);
      #pragma omp task shared(j) firstprivate(n)
      j=fib(n-2);
      #pragma omp taskwait
      value[n] = i+j;
      done[n] = 1;
    }
    omp_unset_lock( &(dolock[n]) );
    return value[n];
  }
}
```

```c
int main() {
  int n = 10;
  value = (int *) malloc( n*sizeof(int) );
  done = (int *) malloc( n*sizeof(int) );
  dolock = (omp_lock_t *)
              malloc(n * sizeof(omp_lock-t));

  #pragma omp parallel shared(n)
  {
    #pragma omp single
    printf ("fib(%d) = %d\n", n, fib(n));
  }

  free(value);
  free(done);
  free(dolock);

  return 0;
}
```

Can you do better?

_OPEMMP Macro

OpenMP compilation adds the preprocessor definition "**_OPENMP**".

To write portable codes, use the macro to protect OpenMP runtime library routine calls; but it is unnecessary to protect OpenMP directives.

```
#if defined(_OPENMP)
#include <omp.h>
#endif
...
#if defined(_OPENMP)
    tid = omp_get_thread_num();
#else
    tid = 0;
#endif
```

Environment Variables

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 (bash):

```
export OMP_SCHEDULE="guided, 4"
export OMP_SCHEDULE="dynamic"
```

OMP_NUM_THREADS

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

```
export 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 (bash):

```
export OMP_DYNAMIC=TRUE
```

Environment Variables (cont'd)

OMP_PROC_BIND

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

```
export OMP_PROC_BIND=TRUE
```

OMP_NESTED

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

```
export OMP_NESTED=TRUE
```

OMP_STACKSIZE / KMP_STACKSIZE

Controls the size (bytes) of the stack for created (non-Master) threads. Examples (bash):

```
export OMP_STACKSIZE=10M
export KMP_STACKSIZE=10M
```

KMP_ : extensions

Environment Variables (cont'd)

OMP_WAIT_POLICY

Provides a hint to an OpenMP implementation about the desired behavior of waiting threads. 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. Examples (bash):

```
export OMP_WAIT_POLICY=ACTIVE
export 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. Example (bash):

```
export 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. Example (bash):

```
export OMP_THREAD_LIMIT=8
```

Thread Stack Size

- The OpenMP standard does not specify how much stack space a thread should have.
- Implementations differ in the default thread stack size. Default thread stack size can be easy to exhaust.

Compiler	Approx. Stack Limit	Approx. Array Size (doubles)
Intel	4 MB	700 x 700
PGI	8 MB	1000 x 1000
GCC	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.
- You can use the OMP_STACKSIZE (KMP_STACKSIZE) environment variable to set the OpenMP thread stack size (the Linux stack size is *unlimited* on Hyades):

```
export KMP_STACKSIZE=12M (sh/bash)
setenv KMP_STACKSIZE=12M (csh/tcsh)
```

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.
- The OpenMP version 3.1 API provides an environment variable to turn processor binding "on" or "off". For example (bash):
  ```
  export OMP_PROC_BIND=TRUE
  export OMP_PROC_BIND=FALSE
  ```
- At a higher level, processes can also be bound to processors.

Further Readings

1. OpenMP version 3.1 Complete Specifications: http://www.openmp.org/mp-documents/OpenMP3.1.pdf
2. OpenMP version 3.1 Summary Card C/C++: http://openmp.org/mp-documents/OpenMP3.1-CCard.pdf
3. OpenMP version 3.1 Summary Card Fortran: http://openmp.org/mp-documents/OpenMP3.1-FortranCard.pdf
4. Using OpenMP: Portable Shared Memory Parallel Programming, *by* B. Chapman, G. Jost, & R. van der Pas, MIT press, 2007: http://ieeexplore.ieee.org/xpl/bkabstractplus.jsp?bkn=6267237
5. Wikipedia: https://en.wikipedia.org/wiki/OpenMP