INTRODUCTION TO PARALLEL COMPUTING AND OPENMP

Plamen Krastev

Office: 38 Oxford, Room 204
Email: plamenkrastev@fas.harvard.edu

FAS Research Computing
Harvard University

Copy of slides is available at
https://software.rc.fas.harvard.edu/training
OBJECTIVES:

To introduce you to the major concepts and ideas in parallel computing

To give you the basic knowledge to write simple parallel OpenMP programs

To provide the information required for running your OpenMP applications efficiently on the Odyssey cluster
OUTLINE:

- Introduction to Parallel Computing
- Introduction to parallel programming with OpenMP
- Summary
I. Introduction to Parallel Computing
What is High Performance Computing?

Sequoia Supercomputer at LLNL
- 20 Petaflops IBM BG/Q system
- 98,304 nodes / 1,572,864 cores
- 1.6 petabytes of memory

Image Credit: LLNL
What is High Performance Computing?

Sequoia Supercomputer at LLNL
• 20 Petaflops IBM BG/Q system
• 98,304 nodes / 1,572,864 cores
• 1.6 petabytes of memory

Image Credit: LLNL

Using the world’s fastest and largest computers to solve large and complex problems.
Serial Computation:

Traditionally software has been written for serial computations:

- To be run on a single computer having a single Central Processing Unit (CPU)
- A problem is broken into a discrete set of instructions
- Instructions are executed one after another
- Only one instruction can be executed at any moment in time
Parallel Computing:

In the simplest sense, parallel computing is the *simultaneous use of multiple compute resources* to solve a computational problem:

- To be run using **multiple CPUs**
- A problem is broken into discrete parts that can be **solved concurrently**
- Each part is further broken down to a series of instructions
- Instructions from each part **execute simultaneously on different CPUs**
**Why Use HPC?**

**Major reasons:**

Save time and/or money: In theory, throwing more resources at a task will shorten its time to completion, with potential cost savings. Parallel clusters can be built from cheap, commodity components.
Why Use HPC?

Major reasons:

Save time and/or money: In theory, throwing more resources at a task will shorten its time to completion, with potential cost savings. Parallel clusters can be built from cheap, commodity components.

Solve larger problems: Many problems are so large and/or complex that it is impractical or impossible to solve them on a single computer, especially given limited computer memory.
Why Use HPC?

Major reasons:

Save time and/or money: In theory, throwing more resources at a task will shorten its time to completion, with potential cost savings. Parallel clusters can be built from cheap, commodity components.

Solve larger problems: Many problems are so large and/or complex that it is impractical or impossible to solve them on a single computer, especially given limited computer memory.

Provide concurrency: A single compute resource can only do one thing at a time. Multiple computing resources can be doing many things simultaneously.
Why Use HPC?

**Major reasons:**

- **Save time and/or money:** In theory, throwing more resources at a task will shorten its time to completion, with potential cost savings. Parallel clusters can be built from cheap, commodity components.

- **Solve larger problems:** Many problems are so large and/or complex that it is impractical or impossible to solve them on a single computer, especially given limited computer memory.

- **Provide concurrency:** A single compute resource can only do one thing at a time. Multiple computing resources can be doing many things simultaneously.

- **Use of non-local resources:** Using compute resources on a wide area network, or even the Internet when local compute resources are scarce.
Applications of Parallel Computing:

- Atmosphere, Earth, Environment, Space Weather
- Physics / Astrophysics – applied, nuclear, particle, condensed matter, high pressure, fusion, photonics
- Bioscience, Biotechnology, Genetics
- Chemistry, Molecular Sciences
- Geology, Seismology
- Mechanical and Aerospace Engineering
- Electrical Engineering, Circuit Design, Microelectronics
- Computer Science, Mathematics

Image credit: LLNL
Applications of Parallel Computing:

- Databases, data mining
- Oil exploration
- Web search engines, web based business services
- Medical imaging and diagnosis
- Drugs design
- Financial and economic modeling
- Advanced graphics and virtual reality, particularly in the entertainment industry
- Networked video and multi-media technologies
- And much, much more ...

Image credit: LLNL
HPC Usage Statistics:

Source: Top500.org
Future Trends:

Performance Development

<table>
<thead>
<tr>
<th>Name</th>
<th>FLOPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>yottaFLOPS</td>
<td>$10^{24}$</td>
</tr>
<tr>
<td>zettaFLOPS</td>
<td>$10^{21}$</td>
</tr>
<tr>
<td>exaFLOPS</td>
<td>$10^{18}$</td>
</tr>
<tr>
<td>petaFLOPS</td>
<td>$10^{15}$</td>
</tr>
<tr>
<td>teraFLOPS</td>
<td>$10^{12}$</td>
</tr>
<tr>
<td>gigaFLOPS</td>
<td>$10^{9}$</td>
</tr>
<tr>
<td>megaFLOPS</td>
<td>$10^{6}$</td>
</tr>
<tr>
<td>kiloFLOPS</td>
<td>$10^{3}$</td>
</tr>
</tbody>
</table>
Future Trends:

The race is already on for Exascale Computing!
HPC Terminology:

- **Supercomputing / High-Performance Computing (HPC)**
- **Flop(s)** – Floating point operation(s)
- **Node** – a stand-alone computer
- **CPU / Core** – a modern CPU usually has several cores (individual processing units)
- **Task** – a logically discrete section from the computational work
- **Communication** – data exchange between parallel tasks
- **Speedup** – time of serial execution / time of parallel execution
- **Massively Parallel** – refer to hardware of parallel systems with many processors (“many” = hundreds of thousands)
- **Embarrassingly Parallel** – solving many similar but independent tasks simultaneously. Requires very little communication
- **Scalability** - a proportionate increase in parallel speedup with the addition of more processors
Parallel Computer Memory Architectures:

Shared Memory:

- Multiple processors can operate independently, but share the same memory resources
- Changes in a memory location caused by one CPU are visible to all processors
Parallel Computer Memory Architectures:

**Shared Memory:**
- Multiple processors can operate independently, but share the same memory resources.
- Changes in a memory location caused by one CPU are visible to all processors.

**Distributed Memory:**
- Requires a communication network to connect inter-processor memory.
- Processors have their own local memory. Changes made by one CPU have no effect on others.
- Requires communication to exchange data among processors.
Hybrid Distributed-Shared Memory:

The largest and fastest computers in the world today employ both shared and distributed memory architectures.

- Processors on a compute node share same memory space
- Requires communication to exchange data between compute nodes
Parallel Programming Models: Parallel Programming Models exist as an abstraction above hardware and memory architectures

- Shared Memory (without threads)
- Shared Threads Models (Pthreads, OpenMP)
- Distributed Memory / Message Passing (MPI)
- Data Parallel
- Hybrid
- Single Program Multiple Data (SPMD)
- Multiple Program Multiple Data (MPMD)
Shared Threads Models:

POSIX Threads

- Library based; requires parallel coding
- C Language only
- Commonly referred to as Pthreads
- Most hardware vendors now offer Pthreads in addition to their proprietary threads implementations
- Very explicit parallelism; requires significant programmer attention to detail

OpenMP

- Compiler directive based; can use serial code
- Jointly defined and endorsed by a group of major computer hardware and software vendors
- Portable / multi-platform, including Unix and Windows platforms
- Available in C/C++ and Fortran implementations
- Can be very easy and simple to use - provides for "incremental parallelism"
Distributed Memory / Message Passing Models:

- A set of tasks that use their own local memory during computation. Multiple tasks can reside on the same physical machine and/or across an arbitrary number of machines.

- Tasks exchange data through communications by sending and receiving messages.

- Data transfer usually requires cooperative operations to be performed by each process. For example, a send operation must have a matching receive operation.

- Message Passing Interface (MPI) is the "de facto" industry standard for message passing, replacing virtually all other message passing implementations used for production work. MPI implementations exist for virtually all popular parallel computing platforms.
Hybrid Parallel Programming Models:

- Currently, a common example of a hybrid model is the combination of the message passing model (MPI) with the threads model (OpenMP)
  - Threads perform computationally intensive kernels using local, on-node data
  - Communications between processes on different nodes occurs over the network using MPI

This hybrid model lends itself well to the increasingly common hardware environment of clustered multi/many-core machines

- Another similar and increasingly popular example of a hybrid model is using MPI with GPU (Graphics Processing Unit) programming
  - GPUs perform computationally intensive kernels using local, on-node data
  - Communications between processes on different nodes occurs over the network using MPI
Can my code be parallelized?

- Does it have large loops that repeat the same operations?

- Does your code do multiple tasks that are not dependent on one another? If so is the dependency weak?

- Can any dependencies or information sharing be overlapped with computation? If not, is the amount of communications small?

- Do multiple tasks depend on the same data?

- Does the order of operations matter? If so how strict does it have to be?
Parallelizing your program:

You parallelize your program to run faster, and to solve larger problems.

How much faster will the program run?

Speedup:

\[ S(n) = \frac{T(1)}{T(n)} \]

Time to complete the job on one process

Time to complete the job on \( n \) process

Efficiency:

\[ E(n) = \frac{S(n)}{n} \]

Tells you how efficiently you parallelize your code
Oversimplified example:

\( p \rightarrow \) fraction of program that can be parallelized
\( 1 - p \rightarrow \) fraction of program that cannot be parallelized
\( n \rightarrow \) number of processors

Then the time of running the parallel program will be

\( 1 - p + \frac{p}{n} \) of the time for running the serial program

80% can be parallelized

20 % cannot be parallelized

\( n = 4 \)

\( 1 - 0.8 + \frac{0.8}{4} = 0.4 \) i.e., 40% of the time for running the serial code

You get 2.5 speed up although you run on 4 cores since only 80% of your code can be parallelized (assuming that all parts in the code can complete in equal time)
Oversimplified example, cont’d:

20% 80%

Serial

20% 20%

Parallel

20% 20%

Process 1

20% 20%

Process 2

20% 20%

Process 3

Not parallelized

parallelized

Process 4
More realistic example:

Serial

Parallel

Process 1

Process 2

Process 3

Process 4

Load unbalance
**Realistic example**: Speedup of matrix vector multiplication in large scale shell-model calculations

![Graph showing relative speedup vs. number of processors (proc) for 52Fe, 50 Lanczos iterations. The graph indicates a linear relationship with an ideal speedup (solid line) and a matvec speedup (dashed line). At 10,000 cores, the speedup is approximately 10 times.]
Basic guidance for efficient parallelization:

- Is it even worth parallelizing my code?
  - Does your code take an intractably long amount of time to complete?
  - Do you run a single large model or do statistics on multiple small runs?
  - Would the amount of time it take to parallelize your code be worth the gain in speed?

- Parallelizing established code vs. starting from scratch
  - Established code: Maybe easier / faster to parallelize, but my not give good performance or scaling
  - Start from scratch: Takes longer, but will give better performance, accuracy, and gives the opportunity to turn a “black box” into a code you understand
Basic guidance for efficient parallelization:

- Increase the fraction of your program that can be parallelized (identify the most time consuming parts of your program and parallelize them). This could require modifying your intrinsic algorithm and code’s organization.

- Balance parallel workload.

- Minimize time spent in communication.

- Use simple arrays instead of user defined derived types.

- Partition data. Distribute arrays and matrices – allocate specific memory for each MPI process.
INTRODUCTION TO PARALLEL COMPUTING AND OPENMP

Plamen Krastev

Office: 38 Oxford, Room 204
Email:plamenkrastev@fas.harvard.edu

FAS Research Computing
Harvard University
II. Introduction to Parallel Programming with OpenMP
What is OpenMP?

- OpenMP (Open Multi-Processing) 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.

- OpenMP supports C/C++ and Fortran on a wide variety of architectures.
Goals of OpenMP:

- Standardization
  - Provide a standard among a variety shared memory architectures / platforms
  - Jointly defined and endorsed by a group of major computer hardware and software vendors
Goals of OpenMP:

- **Standardization**
  - Provide a standard among a variety 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 just a few directives
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 just a few directives

- Ease of Use
  - Provide the capability to incrementally parallelize a serial program
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 just a few directives

- **Ease of Use**
  - Provide the capability to incrementally parallelize a serial program

- **Portability**
  - Specified for C/C++ and Fortran
  - Most major platforms have been implemented including Unix/Linux and Windows
OpenMP Programming Model:

Shared Memory Model: OpenMP is designed for multi-processor / core, shared memory machines
OpenMP Programming Model:

Shared Memory Model: OpenMP is designed for multi-processor / core, shared memory machines

Thread Based Parallelism: OpenMP programs accomplish parallelism exclusively through the use of threads
**OpenMP Programming Model:**

**Shared Memory Model:** OpenMP is designed for multi-processor / core, shared memory machines

**Thread Based Parallelism:** OpenMP programs accomplish parallelism exclusively through the use of threads

**Explicit Parallelism:** OpenMP provides explicit (not automatic) parallelism, offering the programmer full control over parallelization
OpenMP Programming Model:

Shared Memory Model: OpenMP is designed for multi-processor / core, shared memory machines

Thread Based Parallelism: OpenMP programs accomplish parallelism exclusively through the use of threads

Explicit Parallelism: OpenMP provides explicit (not automatic) parallelism, offering the programmer full control over parallelization

Compiler Directive Based: Most of parallelism is specified through the use of compiler directives embedded in the C/C++ or Fortran code
**OpenMP Programming Model:**

**Shared Memory Model:** OpenMP is designed for multi-processor / core, shared memory machines.

**Thread Based Parallelism:** OpenMP programs accomplish parallelism exclusively through the use of threads.

**Explicit Parallelism:** OpenMP provides explicit (not automatic) parallelism, offering the programmer full control over parallelization.

**Compiler Directive Based:** Most of parallelism is specified through the use of compiler directives embedded in the C/C++ or Fortran code.

**I/O:** OpenMP specifies nothing about parallel I/O. It is up to the programmer to ensure that the I/O is conducted correctly in the context of the multi-threaded program.
Fork-Joint Model:

- OpenMP uses a Fork-Joint model for parallel execution.

- All OpenMP programs begin as a single thread – the master thread. The master thread executes sequentially until the first parallel region 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 executed in parallel among the team threads.

- **JOINT:** When the team threads complete the statements in the parallel region, they synchronize and terminate leaving the master thread.
Components of OpenMP:

The OpenMP API is comprised of three components:

- Compiler Directives
- Runtime Library Routines
- Environment Variables
Components of OpenMP:

The OpenMP API is comprised of three components:

- Compiler Directives
- Runtime Library Routines
- Environment Variables

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 the source code and are completely ignored by compilers unless you tell them otherwise – usually by specifying appropriate compiler flags.
Compiler Directives:

- Compiler directives appear as comments in the source code and are completely ignored by compilers unless you tell them otherwise – usually by specifying appropriate compiler flags.

- Compiler directives are used for various purposes, e.g.,
  - Spawning a parallel region
  - Dividing blocks of code among threads
  - Distributing loop iterations among threads
  - Synchronization of work among threads
Compiler Directives:

- Compiler directives appear as comments in the source code and are completely ignored by compilers unless you tell them otherwise – usually by specifying appropriate compiler flags.

- Compiler directives are used for various purposes, e.g.,
  - Spawning a parallel region
  - Dividing blocks of code among threads
  - Distributing loop iterations among threads
  - Synchronization of work among threads

- Compiler directives have the following syntax:

  \[
  \text{sentinel \ directive-name [clause,\ldots]}
  \]

  Fortran: !$OMP PARALLEL DEFAULT(SHARED) PRIVATE(BETA, PHI)
  C/C++: #pragma omp parallel default(shared) private(beta, phi)
Run-time Library Routines:

- OpenMP includes a growing number of run-time library routines
- These routines are used for various purposes, e.g.,
  - Setting and querying the number of threads
  - Querying threads’ unique identifier (thread ID)
  - Querying the thread pool size

**FORTRAN:** INTEGER FUNCTION GET_NUM_THREADS()

**C/C++:**
```c
#include<omp.h>
int omp_get_num_threads(void)
```
Environment Variables:

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

- These environment variables can be used for, e.g.,
  - Setting the number of threads
  - Specifying how loop iterations are divided
  - Enabling / disabling dynamic threads

- Setting OpenMP environment variables depends upon the shell you use

  **csh/tcsh:** `setenv OMP_NUM_THREADS 8`

  **sh/bash:** `export OMP_NUM_THREADS=8`
Example OpenMP Code Structure:

Fortran general code structure:

```
PROGRAM TEST
INTEGER VAR1, VAR2, VAR3

Serial code

!$OMP PARALLEL PRIVATE(VAR1, VAR2) SHARED(VAR3)
Parallel section executed by all threads

!$OMP END PARALLEL
Resume serial code

END PROGRAM TEST
```
Example OpenMP Code Structure:

C/C++ general code structure:

```c
#include <omp.h>
main () {
int var1, var2, var3;
Serial code
    ...
    ...
    ...
#pragma omp parallel private(var1, var2) shared(var3)
    {
        Parallel section executed by all threads
            ...
            ...
            ...
    }
Resume serial code
    ...
    ...
    ...
}
```
## Compiling OpenMP Programs:

<table>
<thead>
<tr>
<th>Compiler/Platform</th>
<th>Compiler</th>
<th>Flag</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel</td>
<td>icc icpc ifort</td>
<td>-openmp</td>
</tr>
<tr>
<td>GNU</td>
<td>gcc g++ g77 gfortran</td>
<td>-fopenmp</td>
</tr>
</tbody>
</table>
Compiling OpenMP Programs:

<table>
<thead>
<tr>
<th>Compiler/Platform</th>
<th>Compiler</th>
<th>Flag</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel</td>
<td>icc</td>
<td>-openmp</td>
</tr>
<tr>
<td></td>
<td>icpc</td>
<td></td>
</tr>
<tr>
<td></td>
<td>icpc</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ifort</td>
<td></td>
</tr>
<tr>
<td>GNU</td>
<td>gcc</td>
<td>-fopenmp</td>
</tr>
<tr>
<td></td>
<td>g++</td>
<td></td>
</tr>
<tr>
<td></td>
<td>g77</td>
<td></td>
</tr>
<tr>
<td></td>
<td>gfortran</td>
<td></td>
</tr>
</tbody>
</table>

**Intel**: ifort –o omp_test.x omp_test.f90 –openmp

**GNU**: gfortran –o omp_test.x omp_test.f90 –fopenmp
OpenMP Directives:

Fortran Directives Format:

- All Fortran OpenMP directives must begin with a sentinel
- The accepted sentinels depend upon the type of Fortran source

!$OMP, C$OMP, *$OMP

- Comments cannot appear on the same line as a directive
- Several Fortran OpenMP directives come in pairs

```fortran
!$OMP directive
[structured block of code]
!$OMP end directive
```
OpenMP Directives:

C/C++ Directive Format:

- All C/C++ directives begin with `#pragma omp`
- 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
Parallel Region Construct:

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

FORTRAN

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

block

$OMP END PARALLEL
```

C/C++

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

structured_block
```
Example: Parallel Region

Fortran

```fortran
program hello
  implicit none
  integer(4):: nthreads
  integer(4):: tid
  integer(4):: omp_get_num_threads
  integer(4):: omp_get_thread_num
  !$omp parallel private(tid)
  tid = omp_get_thread_num() ! Get thread ID
  write(6,*) "Hello World from thread =", tid
  if ( tid == 0 ) then
    nthreads = omp_get_num_threads() ! Get total number of threads
    write(6,*) "Number of threads =", nthreads
  end if
  !$omp end parallel
  stop
end program hello
```
Example: Parallel Region

C++

```cpp
#include <iostream>
#include <omp.h>
using namespace std;
int main () {
    int nthreads;
    int tid;
    #pragma omp parallel private(tid)
    {
        tid = omp_get_thread_num(); // Get thread ID
        cout << "Hello World from thread = " << tid << endl;
        if (tid == 0){
            nthreads = omp_get_num_threads(); // Get thread number
            cout << "Number of threads = " << nthreads << endl;
        }
    }
}
```
**Work-Sharing Constructs:**

- A work-sharing construct divides the execution of the enclosed code region among the members of the team that encounter it.

- Work-sharing constructs do not launch new threads.

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

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

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

**SINGLE** - serializes a section of code

---

![Diagram](image.png)
DO / FOR Directives:

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

```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++

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

for_loop
```
Example: DO / FOR Directives

Fortran

```
program vec_add_do
  implicit none
  integer(4) :: chunk, i
  integer(4), parameter :: n = 1000
  integer(4), parameter :: chunksize = 100
  real(4) :: a(n), b(n), c(n)
  do i = 1, n
    a(i) = i * 1.0
    b(i) = a(i)
  end do
  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)
  end do
  !$omp end do nowait
  !$omp end parallel
  stop
end program vec_add_do
```
```cpp
#include <iostream>
#include <omp.h>
using namespace std;
#define CHUNKSIZE 100
#define N 1000
int main(){
    int i, chunk;
    float a[N], b[N], c[N];
    for ( i = 0; i < N; i++ )
        a[i] = b[i] = i * 1.0;
    chunk = CHUNKSIZE;
    #pragma omp parallel shared(a,b,c,chunk) private(i)
    {
        #pragma omp for schedule(dynamic,chunk) nowait
        for ( i = 0; i < N; i++ )
            c[i] = a[i] + b[i];
    }
    return 0;
}
```
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. Different sections may be executed by different threads.

FORTRAN

```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++

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

    #pragma omp section   newline
        structured_block

}  ```
Examples: Sections Directive

Fortran

PROGRAM VEC_ADD_SECTIONS
  INTEGER N, I
  PARAMETER (N=1000)
  REAL A(N), B(N), C(N), D(N)
  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
Examples: Sections Directive

C/C++

```c
#include <omp.h>
#define N 1000
main ()
{
 int i;
 float a[N], b[N], c[N], d[N];
 for (i=0; i < N; i++) {
    a[i] = i * 1.5;
    b[i] = i + 22.35;
 }
#pragma omp parallel shared(a,b,c,d) private(i)
{
#pragma omp sections nowait
{
#pragma omp section
 for (i=0; i < N; i++)
    c[i] = a[i] + b[i];
#pragma omp section
 for (i=0; i < N; i++)
    d[i] = a[i] * b[i];
} /* end of sections */
} /* end of parallel section */
```

Synchronization Constructs:

OpenMP provides a variety of Synchronization Constructs that control how the execution of each thread proceeds relative to other team threads.
Synchronization Constructs:

OpenMP provides a variety of Synchronization Constructs that control how the execution of each thread proceeds relative to other team threads.

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
Data Scope Attribute Clauses:

- Because OpenMP is based upon the shared memory programming model, most variables are shared by default.

- Data Scope Attribute Clauses are used in conjunction with several directives (PARALLEL, DO/for, and SECTIONS) to control the scoping of enclosed variables.

- Global variables include:
  - **Fortran**: COMMON blocks, SAVE variables, MODULE variables
  - **C**: File scope variables, static variables

- Private variables include:
  - Loop index variables
  - Stack variables in subroutines called from parallel regions
Data Scope Attribute Clauses:

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:

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

  **FORTRAN:** PRIVATE (list)
  **C/C++:** private (list)

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

  **FORTRAN:** SHARED (list)
  **C/C++:** shared (list)

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

  **FORTRAN:** DEFAULT (PRIVATE | FIRSTPRIVATE | SHARED | NONE)
  **C/C++:** default (shared | none)
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.

**FORTRAN:** REDUCTION (operator | intrinsic: list)

**C/C++:** reduction (operator: list)
PROGRAM DOT_PRODUCT
  INTEGER N, CHUNKSIZE, CHUNK, I
  PARAMETER (N=100)
  PARAMETER (CHUNKSIZE=10)
  REAL A(N), B(N), RESULT
  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
Example: REDUCTION Clause

C/C++

```c
#include <omp.h>
main () {
int i, n, chunk;
float a[100], b[100], result;
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);
}
```
Running OpenMP Codes on Odyssey:

1. Compile your code, e.g.,

   ifort –o omp_code.x omp_code.f90 –ompenmp

2. Prepare a batch-job submission script (omp_test.run)

   #!/bin/bash
   #SBATCH –J omp_job
   #SBATCH –o slurm.out
   #SBATCH –e slurm.err
   #SBATCH –p computefest
   #SBATCH –mem=1750
   #SBATCH –n 8
   #SBATCH --nodes=1
   #SBATCH --ntasks-per-node=8
   export OMP_NUM_THREADS=8
   ./omp_test.x

3. Submit the job to the queue

   sbatch omp_test.run
INTRODUCTION TO PARALLEL COMPUTING AND OPENMP – EXERCISES

Plamen Krastev

Office: 38 Oxford, Room 204
Email:plamenkrastev@fas.harvard.edu

FAS Research Computing
Harvard University
Exercise 1:

Overview:

- Login to Odyssey
- Copy the exercise files to your home directory
- 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
Exercise 1:

(1) Copy the example files

In your home directory create a subdirectory for the example codes and then cd to it

```bash
mkdir openMP
cd openMP
```

Then copy the files to your openMP subdirectory

```bash
cp /n/holyscratch/computefest/openmp/* ~/openMP
```

(2) Compile the hello.f90 (or hello.cpp) program

Load an Intel compiler software module

```bash
module load hpc/intel-compilers-13.0.079
```

**Fortran:**

```bash
ifort -o hello.x hello.f90 -openmp
```

**C++**

```bash
icpc -o hello.x hello.cpp -openmp
```
Exercise 1:

(3) Prepare a batch-job submission script and submit your job

Prepare a file with the below contents (hello.run)

```
#!/bin/bash
#SBATCH -J omp_hello
#SBATCH -o hello.out
#SBATCH -e hello.err
#SBATCH -p computefets
#SBATCH -n 2
#SBATCH --mem=1750
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=2
export OMP_NUM_THREADS=2
./hello.x
```

Submit the job

```
sbatch hello.run
```
Exercise 1:

(4) Modify the number of threads and rerun the program

```bash
#!/bin/bash
#SBATCH -J omp_hello
#SBATCH -o hello.out
#SBATCH -e hello.err
#SBATCH -p computefest
#SBATCH -n 4
#SBATCH --mem=1750
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=4
export OMP_NUM_THREADS=4
./hello.x
```

(5) Examine the output (hello.out)

Hello World from thread = 1
Hello World from thread = 0
Number of threads = 4
Hello World from thread = 3
Hello World from thread = 2
Exercise 2:

Compute the dot product of two vectors in parallel

Each thread computes its part of the dot product (pdot)

The total dot product (ddot) is calculated via a reduction operation

(1) Compile the code `dot_product.f90`

    ifort -o dot_product.x dot_product.f90 -openmp

(2) Use the script `dot_product.run` to submit the job

    sbatch dot_product.run

(3) Examine the output (`dot_product.out`)

    Number of threads =        4
    Partial scalar product of A and B for thread 0 is   11050.00
    Partial scalar product of A and B for thread 1 is   74800.00
    Partial scalar product of A and B for thread 2 is  201050.0
    Partial scalar product of A and B for thread 3 is  389800.0
    Scalar product of A and B:  676700.0
Exercise 2:

(4) Modify the number of threads and rerun the program

```bash
#!/bin/bash
#SBATCH -J omp_dot_product
#SBATCH -o dot_product.out
#SBATCH -e dot_product.err
#SBATCH -p computefest
#SBATCH -n 4
#SBATCH --mem=1750
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=4
export OMP_NUM_THREADS=4
./dot_product.x
```

(5) Resubmit the job
Exercise 3:

Calculate PI via a parallel Monte-Carlo method

(1) Compile the code omp_pi.f90

ifort –o omp_pi.x omp_pi.f90 –openmp

(2) Submit the job

sbatch omp_pi.run

(3) Examine the output (omp_pi.out)

Program is running on 4 threads.
PI = 3.14176576000000
Contact information:

Harvard Research Computing Website:

http://rc.fas.harvard.edu

Email:

rchelp@fas.harvard.edu

plamenkrastev@fas.harvard.edu
The next workshop on “Parallel Programming with MPI” will be on Friday, January 17

Location: Science Center C
Time: 1:30 – 5:00 PM