Minnesota Supercomputing Institute
Parallel Computation Overview

Andrew Gustafson dgustaf@umn.edu
Brent Swartz swartzbr@umn.edu
David Porter dhp@umn.edu
Parallel Computation

Parallel Computation means dividing up calculations into independent parts, and computing the independent parts simultaneously.

Most CPU processors are not much faster than what you can buy in a Desktop PC. To perform faster computations you must parallelize.

There are different forms of parallel computation. What form to use depends both on the type of problem, and on the type of compute system available.
Shared Memory System

Source: http://en.wikipedia.org/wiki/Shared_memory
Strategies for a Shared Memory System

- Simple Parallelization: Collect Serial Calculations

- Thread parallelization
  - A single program uses multiple “threads” which can communicate using shared memory.
  - Coding for thread parallelization often means using OpenMP (which in turn is based on POSIX threads).

- Message Passing
  - Message passing frameworks such as MPI may be used, but are often not needed because thread parallel methods are sufficient.
Distributed Memory System: Cluster

Strategies for a Distributed Memory System

● Simple Parallelization: Collect Serial Calculations

● Message Passing
  ○ Message passing frameworks such as MPI may be used to pass messages between nodes.

● Message Passing + Threads within node
  ○ Possible to combine message passing between nodes, with thread communication within a node.
  ○ Often involves using both MPI and OpenMP.
Clusters at MSI

- **Mesabi**
  - About 17,700 total cores, on Intel Haswell processors.
  - 24 cores and 62 GB per node in the large primary queues.
  - Special queues with large memory (up to 1TB), and GPUs.
  - Allows node sharing: good for both small and large jobs.
  - [https://www.msi.umn.edu/content/mesabi](https://www.msi.umn.edu/content/mesabi)

- **Itasca**
  - About 9,000 total cores, on Intel Nehalem processors.
  - 8 cores and 22 GB per node in the large primary queue.
  - Special queues with larger memory and 16 cores per node.
  - [https://www.msi.umn.edu/content/itasca](https://www.msi.umn.edu/content/itasca)

- **Interactive (Lab) Server**
  - About 500 total cores, on older hardware.
  - For interactive, or small single node jobs.
  - 16 cores and 22 GB per node in the primary queue.
  - [https://www.msi.umn.edu/content/interactive-hpc](https://www.msi.umn.edu/content/interactive-hpc)
Heterogeneous Systems

Source: http://electronicdesign.com/digital-ics/gpu-architecture-improves-embedded-application-support
Coding for Heterogenous Systems

○ NVIDIA GPUS
  ▪ Coding can be done in a sub-language called CUDA, which is supported by the PGI Fortran/C compilers (module pg/15.7).
  ▪ Can also use OpenACC (http://www.openacc.org), which is a C/C++/Fortran standard similar to OpenMP (openmp.org), implemented on PGI and recent GNU compilers. OpenACC 2.0 is a supported feature of GCC 7.2.0 and later.
  ▪ MPI may need to be used if multiple CPU nodes are used.
Heterogeneous Systems at MSI

Mesabi (k40 queue)

- 40 nodes with 2 NVidia k40 GPUs per node
- k40 queue on Mesabi
The general view of programming difficulty is that programming becomes more complicated in this order:

- Simple Parallelization: Collect Serial Calculations
- OpenMP (thread parallel)
- MPI (message passing)
- MPI + OpenMP (hybrid message passing + threads)
- Accelerators (GPUs using CUDA or OpenACC, Phis)

(Note that GPUs can more easily be used via the nVidia supplied libraries, e.g. cuFFT, cuBLAS, cuSPARSE, etc. See: http://docs.nvidia.com/cuda/index.html#axzz3o0lsq3xx).

The more difficult strategies can also yield larger speed increases, but it is important to examine the calculation type.
Job Scheduling

Parallel jobs are scheduled using a queueing system so that the hardware is fairly shared.

http://sitiosatumedida.com/mysite/images/blueview.jpg
Job Scheduling

Jobs are scheduled using the Portable Batch System (PBS) queueing system

To schedule a job first make a PBS job script:

```bash
#!/bin/bash
#PBS -l walltime=8:00:00,nodes=3:ppn=8,pmem=1000mb
#PBS -m abe
#PBS -M sample_email@umn.edu

cd ~/program_directory
module load intel
module load openmpi/intel
mpirun -np 24 program_name < inputfile > outputfile
```
Job Submission

To submit a job script use the command:

```
qsub -q queuename scriptname
```

A list of queues available on different systems can be found here: https://www.msi.umn.edu/queues

Submit jobs to a queue which is appropriate for the resources needed.

Resources to consider when choosing a queue:

- Walltime
- Total cores and cores per node
- Memory
- Special hardware (GPUs, etc)
Interactive Jobs

Nodes may be requested for interactive use:

qsub -I -l walltime=1:00:00,nodes=2:ppn=8,mem=4gb

The terminal will hang until the job starts, and then it will return control. You can then use the nodes interactively for the job duration.
Simple Parallelization: Backgrounding
Most easily done with single node jobs.

#!/bin/bash -l
#PBS -l walltime=8:00:00,nodes=1:ppn=8,pmem=1000mb
#PBS -m abe
#PBS -M sample_email@umn.edu

cd ~/job_directory
module load example/1.0
./program1.exe < input1 > output1 &
./program2.exe < input2 > output2 &
./program3.exe < input3 > output3 &
./program4.exe < input4 > output4 &
./program5.exe < input5 > output5 &
./program6.exe < input6 > output6 &
./program7.exe < input7 > output7 &
./program8.exe < input8 > output8 &
wait
Simple Parallelization: Job Arrays
Works best on Mesabi.

Template Job Script, template.pbs:

```bash
#!/bin/bash -l
#PBS -l walltime=8:00:00,nodes=1:ppn=1,mem=2gb
#PBS -m abe
#PBS -M sample_email@umn.edu

cd ~/job_directory
module load example/1.0
./program.exe < input$PBS_ARRAYID > output$PBS_ARRAYID
```

Submit an array of 10 jobs:

```bash
qsub -t 1-10 template.pbs
```
Simple Parallelization: GNU Parallel

A way to spawn multiple threads to perform a shell task.

Example:
cat command_list.txt | parallel -j 24

This will take a list of command in command_list.txt, and then have GNU Parallel execute them simultaneously on one node using up to 24 concurrent threads.

Example:
find . -name '*.txt' | parallel -j 48 -sshloginfile $PBS_NODEFILE wc {}

This will find files ending in .txt, and then will use 48 threads to word count (wc) each of the files. Specifying sshloginfile makes it aware of all nodes being used.
Simple Parallelization: pdsh

A way to run multiple independent processes on multiple hosts.

Example:
```bash
pdsh -R ssh -w node0123,node0123,node0124 "./program.exe"
```
This would start two copies of program.exe on node0123, and one copy of program.exe on node0124, using ssh to connect.

Example:
```bash
pdsh -R ssh -w"$PBS_NODEFILE" "./program.exe"
```
This run one copy of program.exe on each of the cores assigned to the job.
OpenMP

OpenMP is for parallelization on shared memory systems (at MSI, usually one node, composed of 2 sockets). OpenMP is an abbreviation for: Open Multi-Processing

OpenMP is a specification for a set of compiler directives, library routines, and environment variables that can be used to specify high-level parallelism in Fortran and C/C++ programs.

OpenMP Compiler Directives
– Interpreted when OpenMP compiler option is turned on.
– Each directive applies to the succeeding structured block.
Intel Haswell Xeon Architecture

Source: http://www.theplatform.net/2015/05/05/intel-puts-more-compute-behind-xeon-e7-big-memory/

University of Minnesota
<table>
<thead>
<tr>
<th>Feature</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architecture</td>
<td>x86_64</td>
</tr>
<tr>
<td>CPU op-mode(s)</td>
<td>32-bit, 64-bit</td>
</tr>
<tr>
<td>Byte Order</td>
<td>Little Endian</td>
</tr>
<tr>
<td>CPU(s)</td>
<td>24</td>
</tr>
<tr>
<td>On-line CPU(s) list</td>
<td>0-23</td>
</tr>
<tr>
<td>Thread(s) per core</td>
<td>1</td>
</tr>
<tr>
<td>Core(s) per socket</td>
<td>12</td>
</tr>
<tr>
<td>Socket(s)</td>
<td>2</td>
</tr>
<tr>
<td>NUMA node(s)</td>
<td>2</td>
</tr>
<tr>
<td>Vendor ID</td>
<td>GenuineIntel</td>
</tr>
<tr>
<td>CPU family</td>
<td>6</td>
</tr>
<tr>
<td>Model</td>
<td>63</td>
</tr>
<tr>
<td>Stepping</td>
<td>2</td>
</tr>
<tr>
<td>CPU MHz</td>
<td>2497.320</td>
</tr>
<tr>
<td>BogoMIPS</td>
<td>4993.98</td>
</tr>
<tr>
<td>Virtualization</td>
<td>VT-x</td>
</tr>
<tr>
<td>L1d cache</td>
<td>32K</td>
</tr>
<tr>
<td>L1i cache</td>
<td>32K</td>
</tr>
<tr>
<td>L2 cache</td>
<td>256K</td>
</tr>
<tr>
<td>L3 cache</td>
<td>30720K</td>
</tr>
<tr>
<td>NUMA node0 CPU(s)</td>
<td>0-5,12-17</td>
</tr>
<tr>
<td>NUMA node1 CPU(s)</td>
<td>6-11,18-23</td>
</tr>
</tbody>
</table>

© 2013 Regents of the University of Minnesota. All rights reserved.
OpenMP: Arguments For & Against

Pros:
Programmability - Easier to program/debug than MPI. Allows incremental introduction of OpenMP, one loop at a time.
Maintainability - Code is easier to understand, so it may be more easily maintained. Allows for a single source version of code.
Minimal code modification - Serial code usually doesn’t require modification. Can still run the program as a serial code.
Performance - Most nodes on Itasca have 8 cores, and most nodes on Mesabi have 24 cores. An OpenMP application can use all cores on a node, giving a theoretical 24 fold performance improvement. Most modern laptops/servers have multi core CPUs.
Portability - OpenMP is a standard not an implementation. SIMD directive is the only portable way to force a loop to be vector.
OpenMP: Arguments For & Against

Cons:

Memory - Can only run on shared memory (usually one node). So can only use the memory on one node.

Compiler Support - Requires a compiler that supports OpenMP. All MSI compilers (Intel, GNU, PGI) support OpenMP.

False sharing - Possible data placement problem. Can be a problem if loops are not coded optimally.

First touch - Should try to initialize the memory in the same way you use it during computation. Discussed later.
OpenMP Terminology

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

Source: https://computing.llnl.gov/tutorials/openMP/
OpenMP Terminology

Thread Based Parallelism:

- OpenMP programs accomplish parallelism exclusively through the use of threads.
- A thread of execution is the smallest unit of processing that can be scheduled by an operating system. The idea of a subroutine that can be scheduled to run autonomously might help explain what a thread is.
- Threads exist within the resources of a single process. Without the process, they cease to exist.
- Typically, the number of threads match the number of machine processors/cores. However, the actual use of threads is up to the application.

Source: https://computing.llnl.gov/tutorials/openMP/
OpenMP Terminology

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.

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

For parallelization on shared memory systems.

Source: http://en.wikipedia.org/wiki/OpenMP
OpenMP Terminology

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.

Source: https://computing.llnl.gov/tutorials/openMP/
OpenMP Terminology

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

Nested Parallelism:
● The API provides for the placement of parallel regions inside other parallel regions.
● Implementations may or may not support this feature.

Dynamic Threads:
● The API provides for the runtime environment to dynamically alter the number of threads used to execute parallel regions. Intended to promote more efficient use of resources, if possible.
● Implementations may or may not support this feature.

Source: https://computing.llnl.gov/tutorials/openMP/
OpenMP Terminology

I/O:
- OpenMP specifies nothing about parallel I/O. This is particularly important if multiple threads attempt to write/read from the same file.
- If every thread conducts I/O to a different file, the issues are not as significant.
- It is entirely up to the programmer to ensure that I/O is conducted correctly within the context of a multi-threaded program.

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

Source: https://computing.llnl.gov/tutorials/openMP/
OpenMP API Overview

Three Components:

- The OpenMP API is comprised of three distinct components. As of version 4.0:
  - Compiler Directives (44)
  - Runtime Library Routines (35)
  - Environment Variables (13)

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

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

Source: https://computing.llnl.gov/tutorials/openMP/
OpenMP API Overview

Compiler Directives:

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

- OpenMP compiler directives are used for various purposes:
  - Spawning a parallel region
  - Dividing blocks of code among threads
  - Distributing loop iterations between threads
  - Serializing sections of code
  - Synchronization of work among threads

- Compiler directives have the following syntax:

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

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

Source: https://computing.llnl.gov/tutorials/openMP/
OpenMP API Overview

Run-time Library Routines:

- The OpenMP API includes an ever-growing number of run-time library routines.
- These routines are used for a variety of purposes:
  - Setting and querying the number of threads
  - Querying a thread's unique identifier (thread ID), a thread's ancestor's identifier, the thread team size
  - Setting and querying the dynamic threads feature
  - Querying if in a parallel region, and at what level
  - Setting and querying nested parallelism
  - Setting, initializing and terminating locks and nested locks
  - Querying wall clock time and resolution
- For C/C++, all of the run-time library routines are actual subroutines. For Fortran, some are actually functions, and some are subroutines. For example:
  
  Fortran: \texttt{INTEGER FUNCTION OMP\_GET\_NUM\_THREADS()}
  
  C/C++: \texttt{#include <omp.h> \linebreak int omp\_get\_num\_threads(void)}

Source: https://computing.llnl.gov/tutorials/openMP/
OpenMP API Overview

Run-time Library Routines:

- Note that for C/C++, you usually need to include the `<omp.h>` header file.
- Fortran routines are not case sensitive, but C/C++ routines are.
- The run-time library routines are briefly discussed as an overview in the Run-Time Library Routines section, and in more detail in Appendix A.

Source: https://computing.llnl.gov/tutorials/openMP/
OpenMP API Overview

Environment Variables:
- OpenMP provides several environment variables for controlling the execution of parallel code at run-time.
- These environment variables can be used to control such things as:
  - Setting the number of threads
  - Specifying how loop iterations are divided
  - Binding threads to processors
  - Enabling/disabling nested parallelism; setting the maximum levels of nested parallelism
  - Enabling/disabling dynamic threads
  - Setting thread stack size
  - Setting thread wait policy

Source: https://computing.llnl.gov/tutorials/openMP/
Environment Variables:

- Setting OpenMP environment variables is done the same way you set any other environment variables, and depends upon which shell you use. For example:
  
csh/tcsh: \texttt{setenv OMP\_NUM\_THREADS 8}  
sh/bash: \texttt{export OMP\_NUM\_THREADS=8}

OpenMP environment variables are discussed in the \texttt{Environment Variables} section.

Source: https://computing.llnl.gov/tutorials/openMP/
OpenMP Syntax

Fortran: case insensitive
– Add: use omp_lib or include “omp_lib.h”
– Fixed format
● Sentinel directive [clauses]
● Sentinel could be: !$OMP, *$OMP, c$OMP
– Free format
● !$OMP directive [clauses]

C/C++: case sensitive
– Add: #include “omp.h”
● #pragma omp directive [clauses] newline

Source: http://en.wikipedia.org/wiki/OpenMP
OpenMP Simple Example

Simple OpenMP example (code.c):

```c
#include <omp.h>
#include <stdio.h>
int main()
{
#pragma omp parallel
printf("Hello World from thread = %d, nthreads = %d\n",
omp_get_thread_num(), omp_get_num_threads());
}
```

Source: http://en.wikipedia.org/wiki/OpenMP
OpenMP Compilation

OpenMP compilation examples:

Intel (icc, ifort):  icc -openmp -o omp_hello code.c
GNU (gcc, g++, gfortran):  gcc -fopenmp -o omp_hello code.c
PGI(pgcc, pgCC, pgf77, pgf90):  pgcc -mp -o omp_hello code.c

Default behavior for number of threads (when OMP_NUM_THREADS not set):
- One thread for PGI.
- For Intel/GNU, as many threads as available cores.

Source: http://en.wikipedia.org/wiki/OpenMP
OpenMP execution example:
$ export OMP_NUM_THREADS=3
$ ./omp_helloc
Hello World from thread = 0, nthreads = 3
Hello World from thread = 2, nthreads = 3
Hello World from thread = 1, nthreads = 3

Source: http://en.wikipedia.org/wiki/OpenMP
PARALLEL Region Construct

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

```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)
```

Source: https://computing.llnl.gov/tutorials/openMP/
OpenMP C/C++ Directives

Notes:

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

Source: https://computing.llnl.gov/tutorials/openMP/
OpenMP C/C++ Directives

How Many Threads?
- The number of threads in a parallel region is determined by the following factors, in order of precedence:
  1. Evaluation of the IF clause
  2. Setting of the NUM_THREADS clause
  3. Use of the `omp_set_num_threads()` library function
  4. Setting of the OMP_NUM_THREADS environment variable
  5. Implementation default - usually the number of CPUs on a node, though it could be dynamic (see next bullet).

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

Dynamic Threads:
- Use the `omp_get_dynamic()` function to determine if dynamic threads are enabled.
- If supported, the two methods available for enabling dynamic threads are:
  1. The `omp_set_dynamic()` library routine
  2. Setting of the OMP_DYNAMIC environment variable to TRUE

Source: https://computing.llnl.gov/tutorials/openMP/
OpenMP C/C++ Directives

Clauses:
- **IF** clause: If present, it must evaluate to .TRUE. (Fortran) or non-zero (C/C++) in order for a team of threads to be created. Otherwise, the region is executed serially by the master thread.
- The remaining clauses are described in detail later, in the Data Scope Attribute Clauses section.

Restrictions:
- A parallel region must be a structured block that does not span multiple routines or code files
- It is illegal to branch (goto) into or out of a parallel region
- Only a single IF clause is permitted
- Only a single NUM_THREADS clause is permitted
- A program must not depend upon the ordering of the clauses

Source: https://computing.llnl.gov/tutorials/openMP/
OpenMP C/C++ 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.

Source: https://computing.llnl.gov/tutorials/openMP/
OpenMP C/C++ Directives

Work-Sharing Constructs

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

Source: [https://computing.llnl.gov/tutorials/openMP/](https://computing.llnl.gov/tutorials/openMP/)
OpenMP C/C++ Directives

Work-Sharing Constructs

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.

Source: https://computing.llnl.gov/tutorials/openMP/
OpenMP C/C++ Directives

Work-Sharing Constructs

DO / for Directive

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

Source: https://computing.llnl.gov/tutorials/openMP/
OpenMP C/C++ Directives

DO / for Directive

- Format:
- #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

Source: https://computing.llnl.gov/tutorials/openMP/
DO / for Directive

Clauses:

**SCHEDULE**: Describes how iterations of the loop are divided among the threads in the team. The default schedule is implementation dependent. For a discussion on how one type of scheduling may be more optimal than others, see http://openmp.org/forum/viewtopic.php?f=3&t=83.

- **STATIC**: Loop iterations are divided into pieces of size *chunk* and then statically assigned to threads. If chunk is not specified, the iterations are evenly (if possible) divided contiguously among the threads.
- **DYNAMIC**: Loop iterations are divided into pieces of size *chunk*, and dynamically scheduled among the threads; when a thread finishes one chunk, it is dynamically assigned another. The default chunk size is 1.

Source: https://computing.llnl.gov/tutorials/openMP/
OpenMP C/C++ Directives

DO / for Directive

Clauses:

SCHEDULE (cont.):

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

  Subsequent blocks are proportional to:
  - number_of_iterations_remaining / number_of_threads

  The chunk parameter defines the minimum block size. The default chunk size is 1.

- RUNTIME: The scheduling decision is deferred until runtime by the environment variable OMP_SCHEDULE. It is illegal to specify a chunk size for this clause.

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

Source: https://computing.llnl.gov/tutorials/openMP/
OpenMP C/C++ Directives

DO / for Directive

Clauses:

SCHEDULE (cont.):

- **NO WAIT / nowait**: If specified, then threads do not synchronize at the end of the parallel loop.
- **ORDERED**: Specifies that the iterations of the loop must be executed as they would be in a serial program.
- **COLLAPSE**: Specifies how many loops in a nested loop should be collapsed into one large iteration space and divided according to the schedule clause. The sequential execution of the iterations in all associated loops determines the order of the iterations in the collapsed iteration space.
- Other clauses are described in detail later, in the Data Scope Attribute Clauses section.

Source: https://computing.llnl.gov/tutorials/openMP/
C / C++ - for Directive Example

#include <omp.h>
#define CHUNKSIZE 100
#define N 1000
main ()
{
    int i, chunk;
    float a[N], b[N], c[N];
    /* Some initializations */
    for (i=0; i < N; i++)
        a[i] = b[i] = i * 1.0;
    chunk = CHUNKSIZE;
    #pragma omp parallel shared(a,b,c,chunk) private(i)
    {
        #pragma omp for schedule(dynamic,chunk) nowait
        for (i=0; i < N; i++)
            c[i] = a[i] + b[i];
    } /* end of parallel section */
} /* end of main */

Source: https://computing.llnl.gov/tutorials/openMP/
OpenMP C/C++ Directives

Combined Parallel Work-Sharing Constructs

- OpenMP provides three directives that are merely conveniences:
  - PARALLEL DO / parallel for
  - PARALLEL SECTIONS
  - PARALLEL WORKSHARE (fortran only)

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

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

Source: https://computing.llnl.gov/tutorials/openMP/
The number of cores per node varies with the type of Xeon on that node. We define:

\[
\text{MAXCORES} = \text{number of cores/node}
\]

e.g. Nehalem=8 (Itasca), Westmere=12, SandyBridge=16, IvyBridge=20, Haswell=24 (Mesabi).

Since MAXCORES will not vary within the PBS job (the Itasca and Mesabi queues are split by Xeon type), you can determine this at the start of the PBS job (in bash):

\[
\text{MAXCORES} = \text{`grep "core id" /proc/cpuinfo | wc -l`} \text{`export OMP_NUM_THREADS}=$\text{MAXCORES}
\]
OpenMP Parallel Loop

The following example demonstrates how to parallelize a simple loop using the parallel loop construct. The loop iteration variable is private by default, so it is not necessary to specify it explicitly in a private clause.

```c
void simple(int n, float *a, float *b)
{
    int i;
    #pragma omp parallel for
    for (i=1; i<n; i++) /* i is private by default */
        b[i] = (a[i] + a[i-1]) / 2.0;
}
```

Source: http://openmp.org/mp-documents/openmp-examples-4.0.2.pdf
OpenMP Parallel Loop

Threads share the work in loop parallelism. For example, using 8 threads with \( n=800 \) under the default “static” scheduling:
- thread 0 has \( i=1\text{-}100 \)
- thread 1 has \( i=101\text{-}200 \), etc.

```c
void simple(int n, float *a, float *b)
{
    int i;
    #pragma omp parallel for
    for (i=1; i<n; i++) /* i is private by default */
        b[i] = (a[i] + a[i-1]) / 4.0;
}
```

Source: http://openmp.org/mp-documents/openmp-examples-4.0.2.pdf
OpenMP Parallel Loop

_OPENMP is defined if the code is compiled with the openmp flag.

#ifdef _OPENMP
   #include <omp.h>
#endif
OpenMP 4.0 SIMD Directive

For best efficiency in all cases (serial, OpenMP, MPI, etc.), you should use SIMD (vectors) on the innermost loops. On Haswell, each core is able to retrieve 256 bits per clock in vector mode, but only 64 bits per clock without vectorization. So memory bandwidth (which limits most applications’ performance) is up to 4 times slower without vectorization.

Here is Intel’s video describing how to use the OpenMP 4.0 SIMD directive:

OpenMP Reduction Example

#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#define SUM_INIT 0
int main()
{
    int i, n = 250;
    int sum, a[n];
    int ref = SUM_INIT + (n-1)*n/2;
    for (i=0; i<n; i++)
        a[i] = i;

Source: http://openmp.org/examples/Using-OpenMP-Examples-Distr.zip
```c
#include <stdio.h>

int main(void)
{

    double a[1000];
    int n = 1000;
    int i, j;
    double sum = SUM_INIT;

    // Start of parallel region
    #pragma omp parallel for default(none) shared(n,a) reduction(+:sum)
    for (i=0; i<n; i++)
    {
        sum += a[i];
    }
    /*-- End of parallel reduction --*/

    printf("Value of sum after parallel region: %d\n",sum);
    printf("Check results: sum = %d (should be %d)\n",sum,ref);
    return(0);
}
```

Source: http://openmp.org/examples/Using-OpenMP-Examples-Distr.zip
OpenMP Optimization Tips

- Avoid thread migration for better data locality.
All scripts on MSI systems should set env var
OMP_PROC_BIND=true
although the current system gcc (4.4.6) does not use it, gcc
versions 4.7 and above do use it (Intel compiled apps, too).

Setting OMP_PROC_BIND=true will eliminate cache invalidations
due to processes switching to different cores. It will not necessarily
force OPTIMAL thread to core binding (to do this you need to set
KMP_AFFINITY for Intel compiled apps, GOMP_CPU_AFFINITY
for gcc compiled apps). OMP_PROC_BIND was added in
OpenMP 3.1.
OpenMP Optimization Tips

Memory Affinity: “First Touch” Memory

- Memory affinity: Allocate memory as close as possible to the core on which the task that requested the memory is running.
- Memory affinity is not decided by the memory allocation, but by the initialization. Memory will be local to the thread which initializes it. This is called “first touch” policy.
- Hard to do “perfect touch” for real applications. Instead, use a number of threads fewer than number of cores per NUMA domain.

Source: https://www.nersc.gov/assets/pubs_presos/hybridMPIOpenMP20150323.pdf
OpenMP Optimization Tips

Memory Affinity: “First Touch” Memory Example

Initialization

#pragma omp parallel for
for (j=0; j< VectorSize; j++) {
  a[j] = 1.0;
  b[j] = 2.0;
  c[j] = 0.0;
}

Compute

#pragma omp parallel for
for (j=0; j< VectorSize; j++) {
  a[j] = b[j] + d * c[j];
}

https://www.nersc.gov/assets/pubs_presos/hybridMPIOpenMP20150323.pdf
OpenMP Optimization Tips

- False sharing can cause significant performance degradation. See:
  

- In general it should be fastest to parallelize the outer loop only, provided there is sufficient parallelism to keep the threads busy and load balanced. Parallelizing the inner loop only adds an overhead for every parallel region encountered which (although dependent on the implementation and the number of threads) is typically of the order of tens of microseconds.
OpenMP References

OpenMP standard, summary cards, examples:  
http://www.openmp.org/

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

MSI hardware descriptions:  https://www.msi.umn.edu/content/hpc
MSI queue descriptions:  https://www.msi.umn.edu/queues

Source: http://openmp.org/mp-documents/openmp-examples-4.0.2.pdf
MPI

Message Passing Interface
MPI: Message Passing Interface

**MPI is a specification**
Library: subroutines, functions, constants & data types
Commands: starting apps across a cluster
Framework: control & communication

**MPI routines**
Called from source code (C,C++,Fortran)
Compiled & linked with MPI library
Work with framework for inter-process communication.

**MPI versions available at MSI**
- Intel MPI: module load impi
- OpenMPI: module load ompi
- Platform MPI: module load pmpi (Itasca only)
MPI: Motivation & Examples

Hierarchies
- Hardware
- Software

Parallel Applications & Message Passing

Source Code
- A short list of MPI routines is all you need to remember
- A simple example

MPI: Pros & Cons
## Mesabi: Hardware Hierarchy

<table>
<thead>
<tr>
<th>Level</th>
<th>Description</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>core</td>
<td></td>
<td>2.5 GHz clock; up to 16 merged (*) per clock</td>
</tr>
<tr>
<td>Processor</td>
<td></td>
<td>12 cores; 30 MB Cache memory</td>
</tr>
<tr>
<td>Node</td>
<td></td>
<td>2 processors &amp; 64+ GB shared memory</td>
</tr>
<tr>
<td>Level 1</td>
<td>Switch: Leaf</td>
<td>24 nodes (or fewer) 1x EDR to each node</td>
</tr>
<tr>
<td>Level 2</td>
<td>Switch: Island</td>
<td>8 Leafs 6x FDR to each leaf</td>
</tr>
<tr>
<td>Level 3</td>
<td>Switch: Cluster</td>
<td>4 Islands 12x FDR to each island</td>
</tr>
</tbody>
</table>
### MPI: Software Hierarchy

<table>
<thead>
<tr>
<th>Level</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread</td>
<td>Scheduled work in time slices</td>
</tr>
<tr>
<td>Process</td>
<td>1 MPI rank: variables, arrays, IO streams, one or more threads</td>
</tr>
</tbody>
</table>
| Application | 1 or more Ranks (processes)  
MPI communicators |
| Workflow    | 1 or more applications, scripts, ...                                       |
Each node can only directly access its own memory.
Nodes communicate through the network.
Each rank is an instance of your code
- Each rank can only see its own variables
- All the ranks working together are an MPI application.
A process is NOT a processor
A processor has physical cores & cache memory.
A process has time slices and process address space.

1 MPI Rank = 1 process
An MPI rank may have many cores or share a core.
An MPI rank usually is confined to a shared memory node.
An MPI rank ALWAYS has a process address space.
MPI ranks are visible in the process table.
Different MPI ranks have different address spaces.

#PBS -l ppn=... Really means: Cores Per Node
MPI: How You Use It

Start “ranks” (copies of your program) on a list of nodes

```bash
mpirun -np 8 -hostfile $PBS_NODEFILE program
```

Coordinate the operation of all these ranks

- **MPI_Init**: Initialize MPI within each rank
- **MPI_Comm_size**: Get the total number of ranks
- **MPI_Comm_rank**: Get the local rank
- **MPI_Finalize**: Shut down MPI framework

Enable these ranks to communicate

- **MPI_Send**(buffer, ...)  Rank i sends a message
- **MPI_Recv**(buffer, ...)  Rank j receives the message
MPI Starts Copies of Your App

Network

... Memory ... Rank i ... Rank j ...

mpirun starts ranks (instances of your app) on nodes
Your Application Generates Data

Rank i generates some *Data*
This *Data* is needed on Rank j
Message Passing

Rank i sends a buffer of data to rank j
Ranks MAY be on different nodes
Ranks WILL be different processes ⇒ Martial Bits
MPI Syntax

MPI_Init(&argc,&argv)
MPI_Comm_size(comm, &n ranks)
MPI_Comm_rank(comm, &my rank)
MPI_Finalize()

MPI_Send(data, length, type, destination,tag, comm)
MPI_Recv(data, length, type, origin, tag, comm, status)

MPI_Barrier(comm)
MPI_Bcast(data, length, type, origin, comm)
MPI_Reduce(data_in, data_out, length, type, operation, destination, comm)
MPI Routine Names & Calling Convention

**C / C++:**
- int ierror = MPI_Xxxx(....)
- Case sensitive
- All MPI calls are functions
- Program must include mpi.h
- Most parameters passed by reference

**FORTRAN:**
- Call MPI_XXXX(...,ierror)
- Case insensitive
- All MPI calls are subroutines
- ierror is always the last parameter
- Program must include mpif.h
Structure of an MPI Program

Program start

Uncoordinated parallel execution
  Declarations & Prototypes
  Usually identical setup
  \textit{MPI Include Statements}

\textit{Initialize MPI Environment}

Coordinated Parallel execution
  \textit{Divide work among ranks}
  \textit{Communicate between ranks}
  \textit{Wait on other ranks as needed}
  Do Work
  \textit{Collect results}

\textit{Terminate MPI Environment}

Uncoordinated parallel execution

Program End

\texttt{#include \textasciitilde mpi.h}\texttt{\
}\texttt{MPI\_Init}\texttt{\
}\texttt{MPI\_Comm\_size}\texttt{\
}\texttt{MPI\_Comm\_rank}\texttt{\
}\texttt{MPI\_Send}\texttt{\
}\texttt{MPI\_Recv}\texttt{\
}\texttt{MPI\_Barrier}\texttt{\
}\texttt{MPI\_Gather}\texttt{\
}\texttt{MPI\_Reduce}\texttt{\
}\texttt{MPI\_Finalize}
#include "mpi.h"
#include <cstdlib>
#include <iostream>

int main(int argc, char** argv){
    using namespace std;
    int iError, myrank;
    MPI_Status status;

    iError = MPI_Init(&argc, &argv);
    iError = MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    if(myrank == 0){
        int value_to_send = 5;
        iError = MPI_Send(&value_to_send, 1, MPI_INT, 1, 123, MPI_COMM_WORLD);
        cout << "Process " << myrank << " sent value " << value_to_send << endl;
    }
    else if(myrank == 1){
        int value_received;
        iError = MPI_Recv(&value_received, 1, MPI_INT, 0, 123, MPI_COMM_WORLD, &status);
        cout << "Process " << myrank << " received value " << value_received << endl;
    }
    iError = MPI_Finalize();
    return(0);
}
MPI: Arguments For & Against

**Pros:**

Performance - Strong Scaling & Cache Coherency
scalability - more cores AND more memory
flexibility - hardware topology & heterogeneity
portability - MPI is a standard not an implementation

**Cons:**

Need to restructure your code
   However: possibly not by much
May need to restructure IO & data formats
   However: may lead to much faster (parallel) IO
Can fall back to Rank 0 doing all IO
Current Architecture Trends

- Multi-socket nodes with rapidly increasing core counts.
- Memory per core decreasing.
- Memory bandwidth per core decreasing.
- Network bandwidth per core decreasing.

Need a hybrid programming model with 4 levels of parallelism

- Workflow: (GNU parallel) independent cases.
- Distributed Memory: (MPI) between nodes or sockets.
- Shared memory: (OpenMP) on the nodes/sockets.
- Vectorization: (SIMD) for lower level loops.
Optimization Strategies:

1. Serial Optimization: Compiler options, profile code, etc.
2. Increase vectorization (SIMD) for lower level loops.
3. Implement shared memory threading (using OpenMP or pthreads) on a node/socket.
4. Implement MPI between nodes or sockets.
5. Implement workflow: driven by scripts or database