USING OPENMP @ FASRC

Francesco Pontiggia
Sr. Research Consultant

March 2020
CS205

Objectives

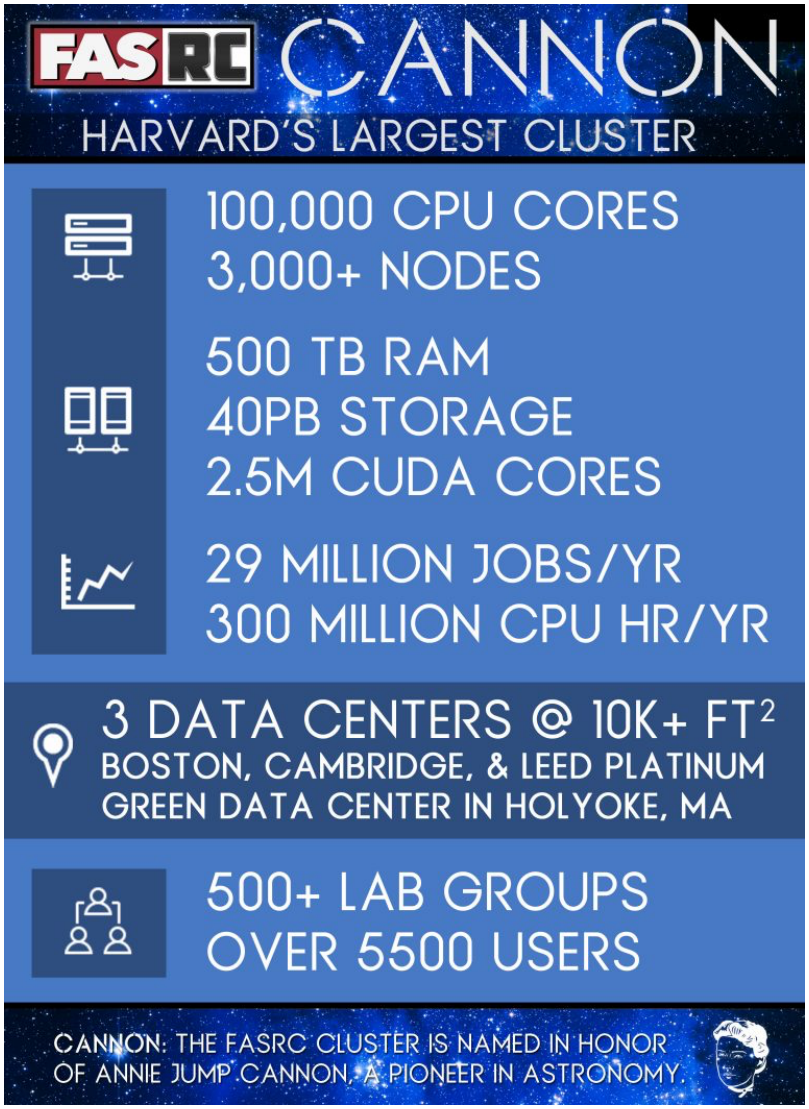
- To give you the basic knowledge and experience to write simple parallel OpenMP programs
- To provide the information required for running your OpenMP applications efficiently on the FASRC cluster

Introduction

Research Computing Resources

Compute:
100,000 compute cores
Cores/node: 8 to 64
Memory/node: 12GB to 512GB (4GB/core)
2,500,000 NVIDIA GPU cores
Software:
Operating System CentOS 7
Slurm job manager
1,000+ scientific tools and programs
https://portal.rc.fas.harvard.edu/apps/modules

Interconnect:
2 underlying networks connecting 3 data centers
TCP/IP network
Low-latency 200 GB/s HDR InfiniBand (IB) and 56 GB/s FDR IB network:
inter-node parallel computing
fast access to Lustre mounted storage



Cluster Basics

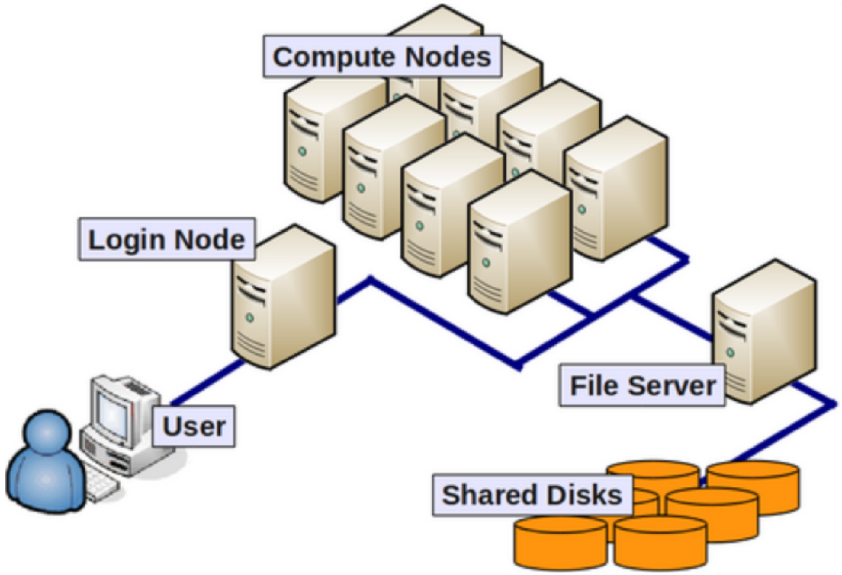
Login Nodes:

- prepare input and stage your calculations
- interact with the scheduler
- no computationally expensive processes allowed

https://docs.rc.fas.harvard.edu/kb/access-and-login/

Compute nodes:

- computational resource monitored and managed by the SLURM scheduler.
- resources are organized in partitions
- access only via scheduler
- servers are interconnected by Infiniband fabric (high throughput , low latency)



Storage:

	Home Folders	**Local Scratch**	**Global Scratch**
mount point	$HOME	/scratch	$SCRATCH/cs205/
size limit	100G	~ 270G/node	quota 50T/group
backup / retention	Hourly snapshots	only available during job	**90days retention** no backup
performance	Not suitable for intense I/O	Suited for small file I/O intensive jobs	Suited for large file I/O intensive jobs

Job Scheduler - SLURM

- SLURM = Simple Linux Utility for Resource Management
  User tasks (jobs) on the cluster are controlled by SLURM and isolated in cgroups so that users cannot interfere with other jobs or exceed their resource request (cores, memory, time)
- Basic SLURM commands:
  - sbatch: submit a batch job script
    ```
    > sbatch [ options for resource request ] myscript
    ```
  - srun: submit an interactive test job
    ```
    > srun --pty [ options for resource request ] /bin/bash
    ```
  - squeue: contact slurmctld for currently running jobs
    ```
    > squeue
    ```
  - sacct: contact slurmdb for accounting stats after job ends
    ```
    > sacct
    ```
  - scancel: cancel a job(s)
    ```
    > scancel some_job_ID
    ```

https://docs.rc.fas.harvard.edu/kb/convenient-slurm-commands/
https://docs.rc.fas.harvard.edu/kb/running-jobs/

SLURM Partitions

https://docs.rc.fas.harvard.edu/kb/running-jobs/#Slurm_partitions

Partitions:	shared	gpu	test	gpu_test	serial_requeue	gpu_requeue	bigmem	unrestricted	**pi_lab**
Time Limit	7 days	7 days	8 hrs	1 hrs	7 days	7 days	no limit	no limit	**varies**
# Nodes	530	15	16	1	1930	155	6	8	**varies**
# Cores / Node	48	32 + 4 V100	48	32 + 4 V100	varies	varies	64	64	**varies**
Memory / Node (GB)	196	375	196	375	varies	varies	512	256	**varies**

Looking at a Partition to learn more:
sinfo -p shared
scontrol show partition shared

Software on Odyssey

- CentOS 7
- Hundreds of software packages – compilers, numeric libraries, development packages, visualization tools, and much more
  - https://portal.rc.fas.harvard.edu/apps/modules
- Harvard modified environment module system LMOD
  - https://docs.rc.fas.harvard.edu/kb/software/
  - Dynamically change user environment
  - Software is loaded incrementally

```
module load gcc/8.2.0-fasrc01      # Loads GCC compiler module
module avail                       # Lists available modules
module list                        # Lists loaded modules
module purge                       # Unloads ALL modules
module-query gcc                   # Finds modules
module-query gcc/8.2.0-fasrc01     # Gives more information
```

XSEDE

- XSEDE = Extreme Science and Engineering Development Environment
- A single virtual system that scientists can use to interactively share computing resources, data and expertise
- https://www.xsede.org

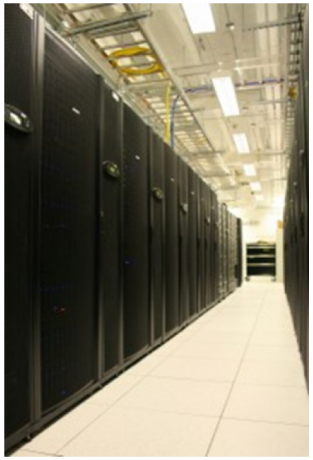
Harvard University XSEDE campus champions:

- Plamen Krastev - plamenkrastev@fas.harvard.edu
- Francesco Pontiggia - pontiggia@g.harvard.edu

What is High Performance Computing?

Summit: ORNL

Sierra: LLNL



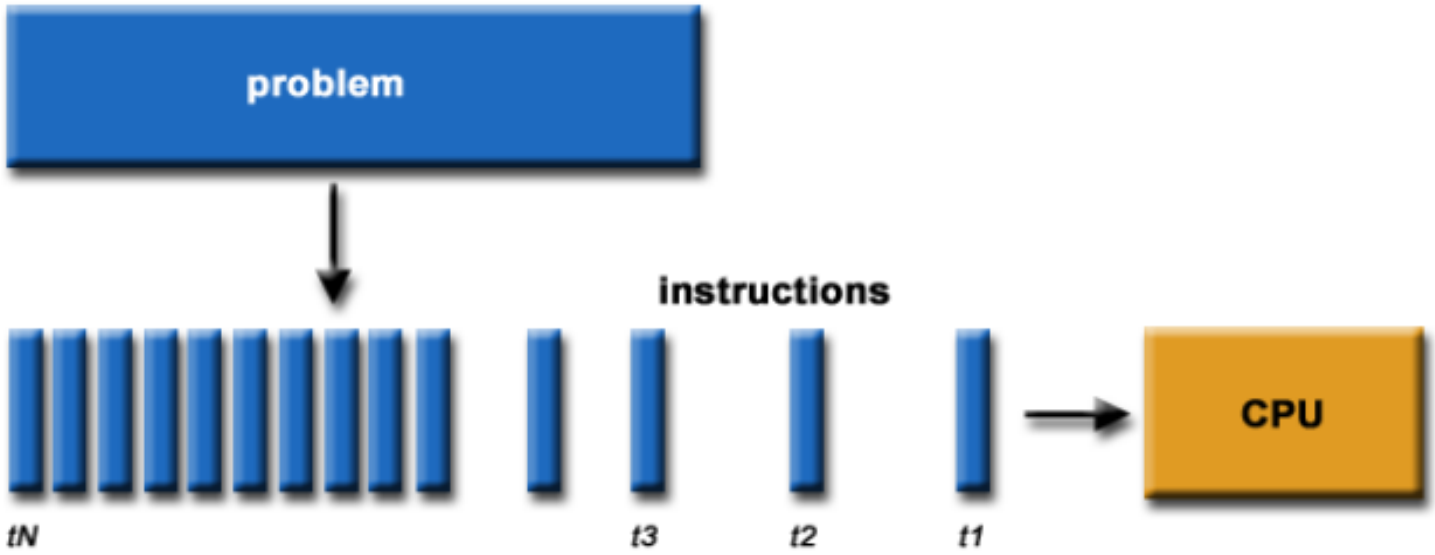
Odyssey: Harvard

Using the world's fastest and largest computers to solve large and complex problems.

Serial Computation

Traditionally software has been written for serial computations:

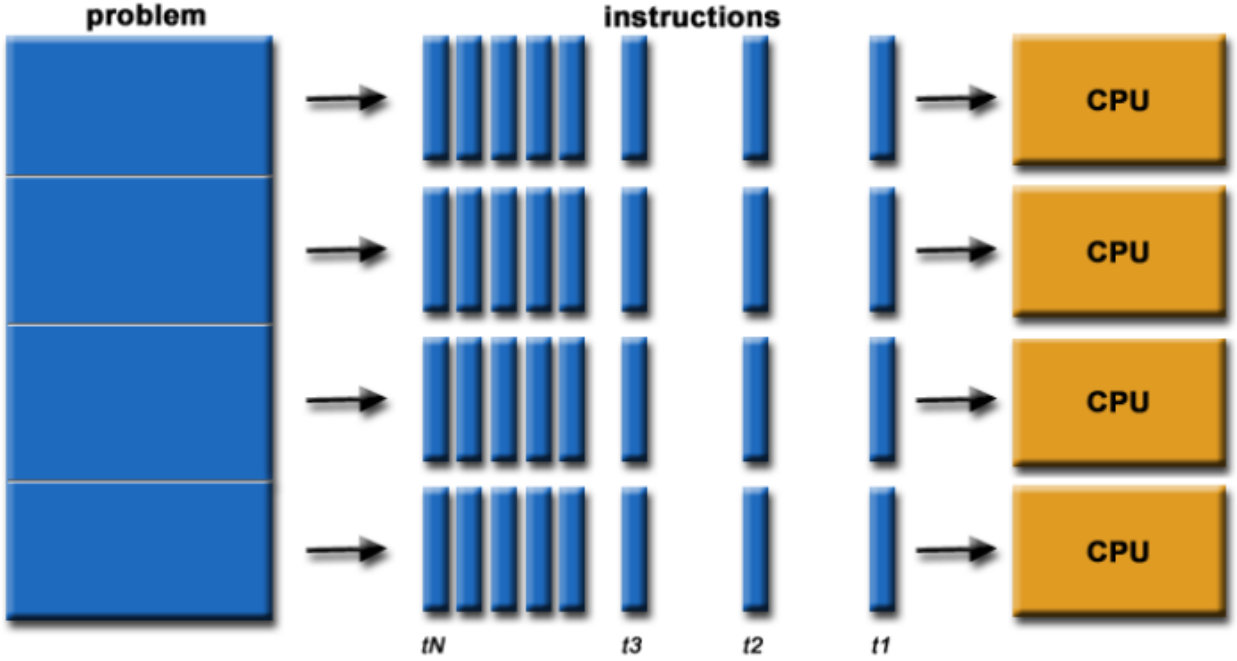
- To be run on a single computer having a single Central Processing Unit (CPU)
- A problem is broken into a discrete set of instructions
- Instructions are executed one after another
- Only one instruction can be executed at any moment in time



Parallel Computation

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.

Solve larger / more complex 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 HPC (not a complete list)

- 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

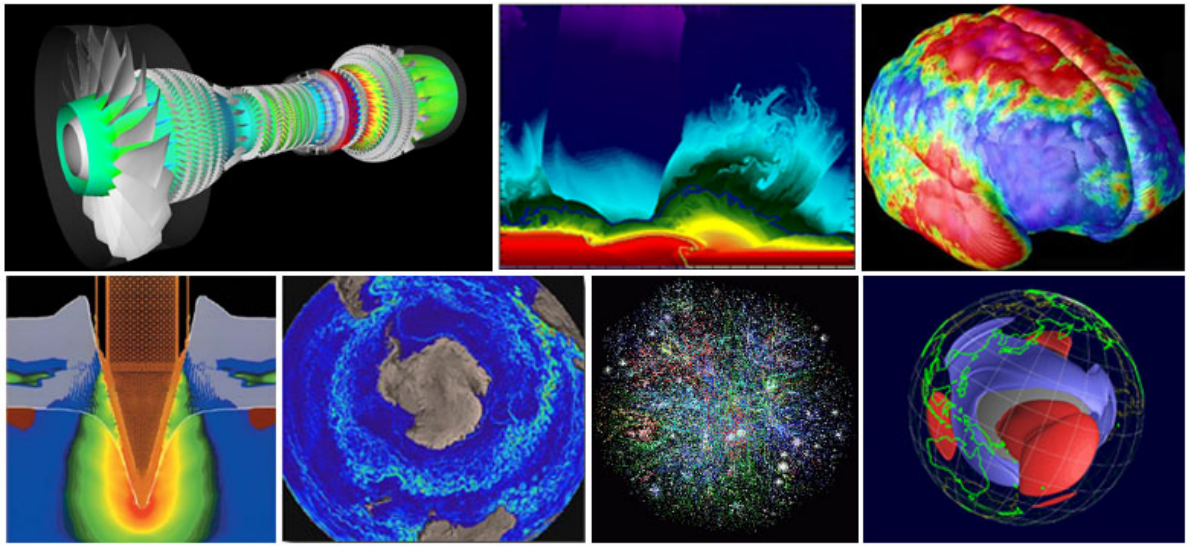


Image credit: LLNL

OpenMP Basics

What is OpenMP ?

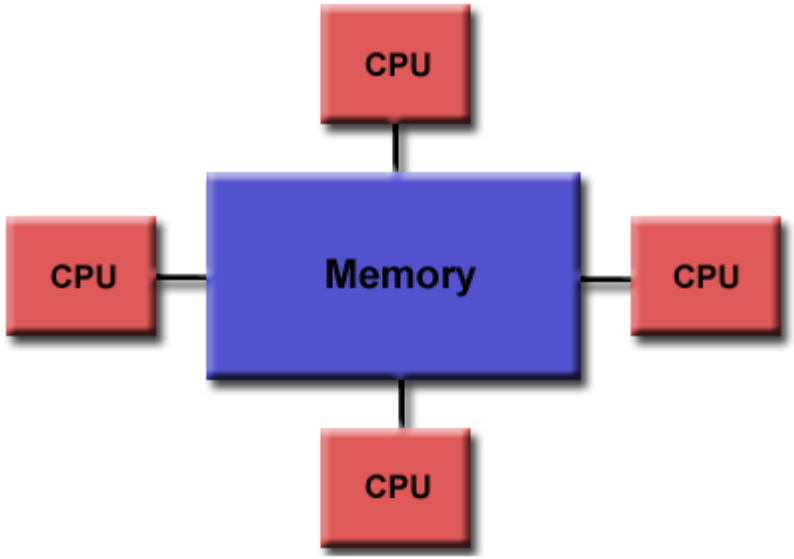
- OpenMP = Open Multi-Processing

- 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 Programming Model

- Shared Memory
- Single Node
- One thread per core
- Explicit Parallelism
- Not designed to handle parallel I/O



Compiling OpenMP Programs

Compiler/Platform	Compiler	Flag
Intel	icc icpc ifort	-openmp
GNU	gcc g++ g77 gfortran	-fopenmp

Intel:

```
module load  intel/19.0.5-fasrc01
icc -o omp_test.x omp_test.c -qopenmp
```

GNU:

```
module load gcc/8.2.0-fasrc01
gcc -o omp_test.x omp_test.c -fopenmp
```

Running OpenMP Programs

Interactive test jobs:

(1) Start an interactive bash shell

```
> srun -p test -c 4 --pty --mem=4G -t 0-06:00 /bin/bash
```

(2) Load required modules, e.g.,

```
> module load gcc/8.2.0-fasrc01
```

(3) Compile your code (or use a `Makefile`)

```
> gcc -o omp_hello.x omp_hello.c -fopenmp
```

(4) Run the code

```
> export OMP_NUM_THREADS=4
> ./omp_hello.x
```

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

Running OpenMP Programs

Batch Jobs:

(1) Compile your code (or use a `Makefile`)

```
> gcc -o omp_hello.x omp_hello.c -fopenmp
```

(2) Prepare a batch-job submission script, e.g.,

```
#!/bin/bash
#SBATCH -J omp_job          # Job name
#SBATCH -o slurm.out        # STD output
#SBATCH -e slurm.err        # STD error
#SBATCH -p shared           # Queue name
#SBATCH -t 0-00:30          # Time (D-HH:MM)
#SBATCH --mem=4G            # Memory in GB
#SBATCH -c 8
#SBATCH -N 1
export OMP_NUM_THREADS=$SLURM_CPUS_PER_TASK
srun -c $SLURM_CPUS_PER_TASK ./omp_hello.x
```

(3) Submit the job to the queue

```
> sbatch omp_test.run
```

OpenMP Exercises

Exercises - Overview

1. Parallel region construct
2. Parallel FOR loops in OpenMP
3. Parallel sections in OpenMP
4. Reduction – parallel dot product
5. Orphaned directives
6. Scaling – speedup and efficiency
7. Matrix-Matrix multiplication
8. Helmholtz Equation
9. Poisson Equation
10. Molecular Dynamics (MD)

Exercises - Setup

- Login to the cluster
- Make directory for this session, e.g.,

```
> mkdir ~/OpenMP
```

- Get copy of the OpenMP exercises. These are hosted at Github at https://github.com/fasrc/User_Codes/tree/master/Courses/CS205/OpenMP

```
> cd ~/OpenMP
> git clone https://github.com/fasrc/User_Codes.git
> cd User_Codes/Courses/CS205/OpenMP
```

- Load compiler software module (here the default compiler is GNU `gcc`)

```
> module load gcc/8.2.0-fasrc01
```

Exercise 1: Parallel Regions

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

#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

```
#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
      .
      .
      .
}
```

Exercise 1: Parallel Regions

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
int main (int argc, char *argv[]) {
  int nthreads;
  int tid;
  // Parallel region starts here.............................
#pragma omp parallel private(nthreads,tid)
  {
    // Get thread ID.........................................
    tid = omp_get_thread_num();
    printf("Hello World from thread = %d\n", tid);
    if ( tid == 0 ){
      // Get total number of threads.........................
      nthreads = omp_get_num_threads();
      printf("Number of threads = %d\n", nthreads);
    }
  }
  // End of parallel region.................................
}
```

Exercise 1: Parallel Regions

(1) Description – a simple parallel "Hello World" program printing out the number of OpenMP threads and thread IDs

(2) Compile the program

```
> cd ~/OpenMP/User_Codes/Courses/CS205/OpenMP/Example1
> make
```

(3) Run the program (the default is setup to 4 threads)

```
> sbatch sbatch.run
```

(4) Explore the output (the "omp_hello.dat" file), e.g.,

```
> cat omp_hello.dat
Hello World from thread = 0
Hello World from thread = 1
Hello World from thread = 2
Number of threads = 4
Hello World from thread = 3
```

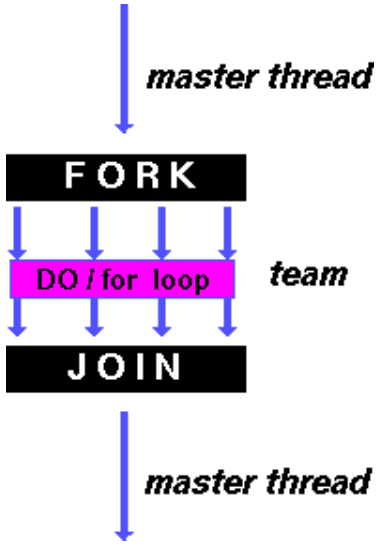
(5) Run the program with different thread number – e.g., 1, 2, 4, 8

Exercise 2: Parallel FOR Loops

The 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.

This is the easiest, fastest, and usually most efficient way to parallelize your code.

FOR - shares iterations of a loop across the team. Represents a type of "data parallelism"



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

for_loop

Exercise 2: Parallel FOR Loops

(1) Description – Vector addition. This example demonstrates use of the OpenMP loop work-sharing construct. Notice that it specifies dynamic scheduling of threads and assigns a specific number of iterations to be done by each thread.

(2) Review the source code

(3) Compile the program

```
> cd ~/OpenMP/User_Codes/Courses/CS205/OpenMP/Example2
> make
```

(4) Run the program (the default is setup to 4 threads)

```
> sbatch sbatch.run
```

(5) Explore the output (the “omp_loop.dat” file). Note that it is piped through the sort utility. This will make it easier to view how loop iterations were actually scheduled across the team of threads.

Exercise 2: Parallel FOR Loops

(6) Run the program a couple more times and review the output.

Typically, dynamic scheduling is not deterministic. Every time you run the program, different threads can run different chunks of work. It is even possible that a thread might not do any work because another thread is quicker and takes more work. In fact, it might be possible for one thread to do all of the work.

(7) Edit the "`omp_loop.c`" source file and change the **dynamic scheduling** to **static scheduling**.

(8) Recompile

```
> make clean
> make
```

Notice the difference in output compared to dynamic scheduling. Specifically, notice that thread 0 gets the first chunk, thread 1 the second chunk, and so on.

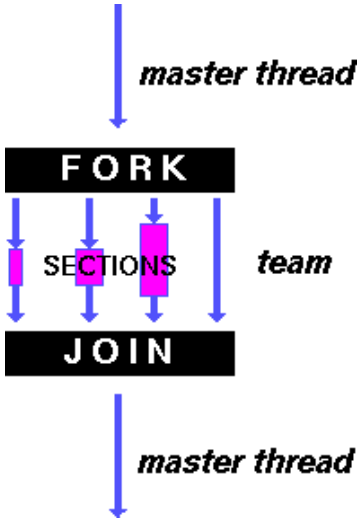
(9) Run the program a couple more times. Does the output change? With static scheduling, the allocation of work is deterministic and should not change between runs, and every thread gets work to do.

Exercise 3: Parallel Sections

- 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

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




```
#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

 }
```

Exercise 3: Parallel Sections

(1) Description – This example demonstrates use of the OpenMP SECTIONS work-sharing construct. Note how the PARALLEL region is divided into separate sections, each of which will be executed by one thread.

(2) Review the source code

(3) Compile the program

```
> cd ~/OpenMP/User_Codes/Courses/CS205/OpenMP/Example3
> make
```

(4) Run the program (the default is setup to 4 threads)

```
> sbatch sbatch.run
```

(5) Explore the output (the "`omp_sections.dat`" file). Note that it is piped through the sort utility.

(6) Run the program several times and observe any differences in output.

Because there are only two sections, you should notice that some threads do not do any work. It is even possible for one thread to do all of the work. Which thread does work is non-deterministic in this case.

Exercise 4: Reduction – Dot Product

- 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

C: reduction *(operator: list)*

Exercise 4: Reduction – Dot Product

(1) Description – Program performs dot product of 2 vectors in parallel

(2) Review the source code and compile the program

```
> cd ~/OpenMP/User_Codes/Courses/CS205/OpenMP/Example4
> make
```

(3) Run the program (the default is setup to 4 threads)

```
> sbatch sbatch.run
```

(4) Explore the output (the "`omp_dot.dat`" file), e.g.,

```
> cat omp_dot.dat
Global dot product = 656700.000000
Running on 4 threads.
Thread 0: partial dot product = 128300.000000
Thread 1: partial dot product = 150550.000000
Thread 2: partial dot product = 175300.000000
Thread 3: partial dot product = 202550.000000
```

(5) Run the program with different thread number – e.g., 1, 2, 4, 8

Exercise 5: Orphaned Directives

- OpenMP contains a feature called orphaning which dramatically increases the expressiveness of parallel directives.
- Orphaning is a situation when directives related to a parallel region are not required to occur lexically within a single program unit.
- Directives such as critical, barrier, sections, single, master, and do, can occur by themselves in a program unit, dynamically “binding” to the enclosing parallel region at run time.
- Orphaned directives enable parallelism to be inserted into existing code with a minimum of code restructuring.
- Orphaning can also improve performance by enabling a single parallel region to bind with multiple do directives located within called subroutines.

Exercise 5: Orphaned Directives – Dot Product Revised

(1) Description – Program performs dot product of 2 vectors in parallel. However it differs from previous Example4 because the parallel loop construct is orphaned - it is contained in a subroutine outside the lexical extent of the main program's parallel region.

(2) Review the source code and compile the program

```
> cd ~/OpenMP/User_Codes/Courses/CS205/OpenMP/Example5
> make
```

(3) Run the program (the default is setup to 4 threads)

```
> sbatch sbatch.run
```

(4) Explore the output (the "`omp_dot.dat`" file)

(5) Run the program with different thread number – e.g., 1, 2, 4, 8

Example 6: Scaling – Compute PI in Parallel

Monte-Carlo Approximation of PI:

1. Inscribe a circle in a square

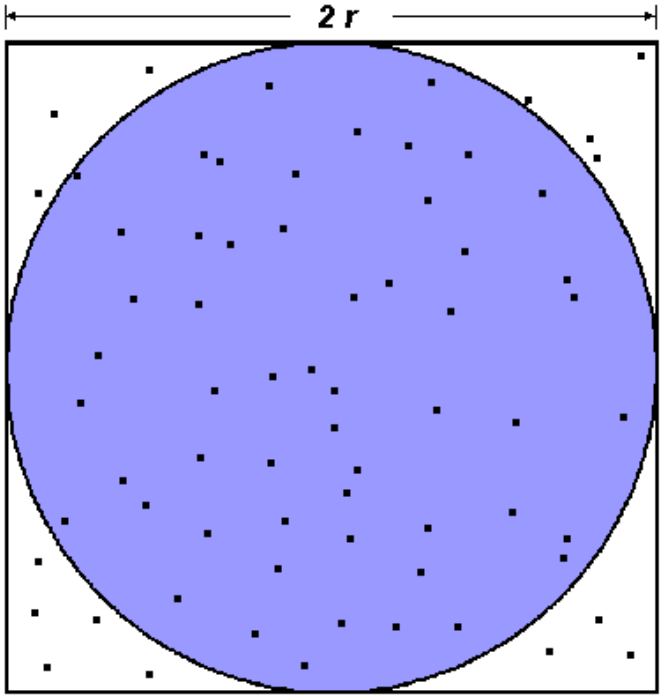
2. Randomly generate points in the square

3. Determine the number of points in the square that are also in the circle

4. Let r be the number of points in the circle divided by the number of points in the square

5. PI ~ 4 r

6. Note that the more points generated, the better the approximation



$$A_S = (2r)^2 = 4r^2$$

$$A_C = \pi r^2$$

$$\pi = 4 \times \frac{A_C}{A_S}$$

Example 6: Scaling – Compute PI in Parallel

(1) Description – Program performs parallel Monte-Carlo approximation of PI

(2) Review the source code and compile the program

```
> cd ~/OpenMP/User_Codes/Courses/CS205/OpenMP/Example6
> make
```

(3) Run the program

```
> sbatch sbatch.run
```

(4) Explore the output (the "`omp_dot.dat`" file), e.g.,

```
> cat omp_pi.dat
Exact value of PI: 3.14159
Estimate of PI:    3.14165
Time:    5.56 sec.
```

(5) Run the program with different thread number – 1, 2, 4, 8 – and record the run times for each case. This will be needed to compute the speedup and efficiency

Example 6: Scaling – Compute PI in Parallel

How much faster will the program run?

Speedup:

$$S(n) = \frac{T(1)}{T(n)}$$

Time to complete the job on **one** thread

Time to complete the job on **n** threads

Efficiency:

$$E(n) = \frac{S(n)}{n}$$

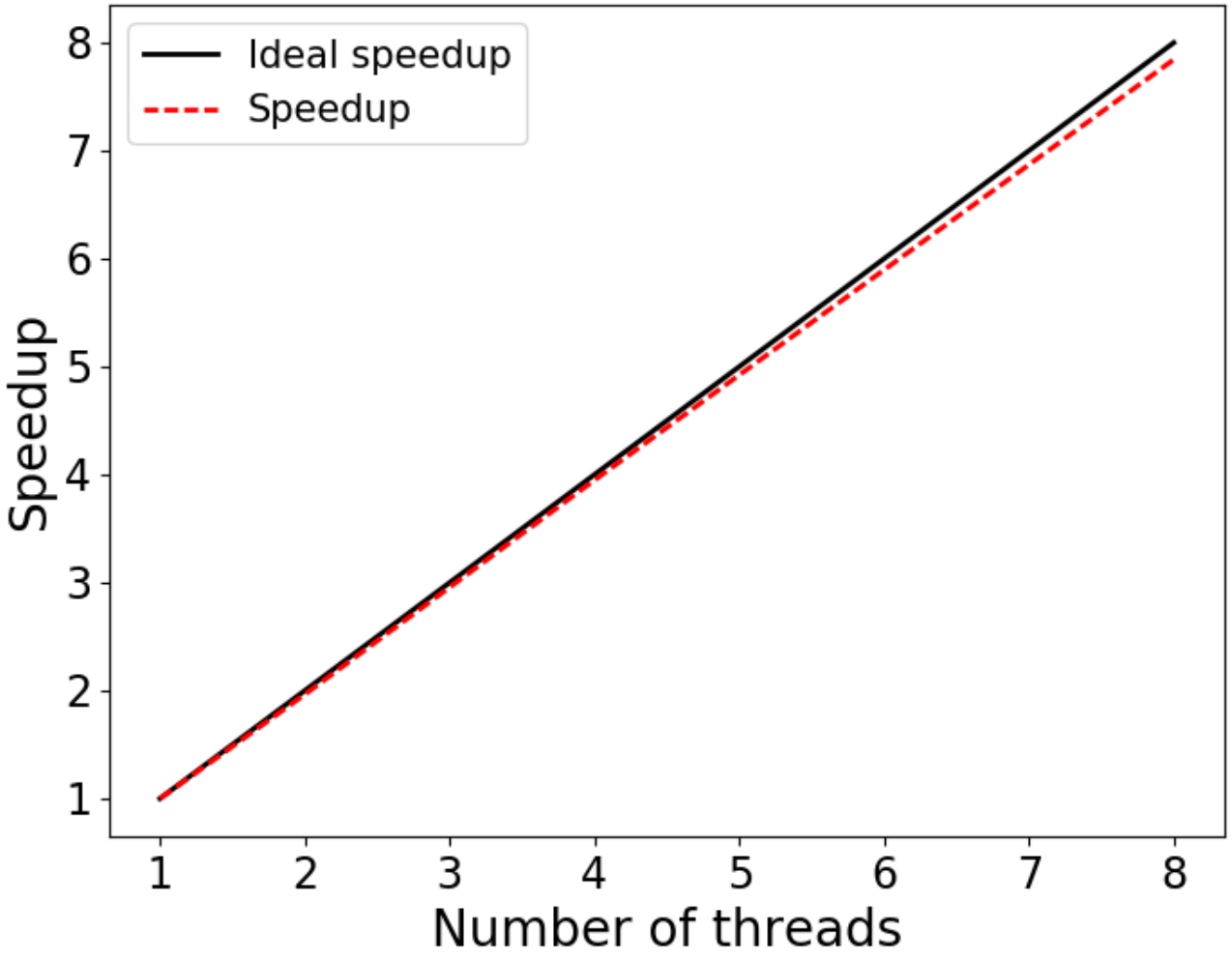
Tells you how efficiently you parallelize your code

Example 6: Scaling – Compute PI in Parallel

You may use the speedup.py Python code to generate to calculate the speedup and efficiency. It generates the below table plus a speedup figure

Nthreads	Walltime	Speedup	Efficiency (%)
1	22.51	1.00	100.00
2	11.46	1.96	98.21
4	5.70	3.95	98.73
8	2.87	7.84	98.04

Example 6: Scaling – Compute PI in Parallel



Exercise 7: Matrix-Matrix Multiplication

(1) Description – Program performs Matrix-Matrix multiplication in parallel.
Matrix dimension = 1000

(2) Review the source code and compile the program

```
> cd ~/OpenMP/User_Codes/Courses/CS205/OpenMP/Example7
> make
```

(3) Run the program (the default is setup to 4 threads)

```
> sbatch sbatch.run
```

(4) Explore the output (the "`omp_mm.dat`" file), e.g.,

```
> cat omp_mm.dat
```

(5) Run the program with different thread number – e.g., 1, 2, 4, 8 – and observe the FLOPS rate (FLOPS / elapsed time) with increasing thread count

Exercise 7: Matrix-Matrix Multiplication

Example output:

```
Matrix multiplication tests.

  Number of processors available = 4
  Number of threads              = 4

Matrix multiplication timing.
  A(LxN) = B(LxM) * C(MxN).
  L = 1000
  M = 1000
  N = 1000
  Floating point OPS roughly 2000000000
  Elapsed time dT = 0.152197
  Rate = MegaOPS/dT = 13140.852903

omp_mm:
  Normal end of execution.
```

Exercise 8: Helmholtz Equation

$$\nabla^2\psi + k^2\psi = 0$$

- Wave equation in frequency domain
  - Acoustics
  - Electromagnetics (Maxwell equations)
  - Diffusion/heat transfer/boundary layers
  - Telegraph, and related equations
  - k can be complex
- Quantum mechanics
  - Klein-Gordon equation
  - Schrödinger equation
- Relativistic gravity
- Molecular dynamics
- Appears in many other models

Exercise 8: Helmholtz Equation

(1) Description - solves a discretized 2D Helmholtz equation

The two dimensional region given is:

-1 <= X <= +1
-1 <= Y <= +1

The region is discretized by a set of M by N nodes:

P(I,J) = (X(I), Y(J))

where, for 0 <= I <= M-1, 0 <= J <= N - 1, (C/C++ convention)

X(I) = (2 * I - M + 1) / (M - 1)
Y(J) = (2 * J - N + 1) / (N - 1)

The Helmholtz equation for the scalar function U(X,Y) is

-Uxx(X,Y) - Uyy(X,Y) + ALPHA * U(X,Y) = F(X,Y)

where ALPHA is a positive constant. We suppose that Dirichlet boundary conditions are specified, that is, that the value of U(X,Y) is given for all points along the boundary.

Exercise 8: Helmholtz Equation

We suppose that the right hand side function F(X,Y) is specified in such a way that the exact solution is

U(X,Y) = (1 - X^2) * (1 - Y^2)

Using standard finite difference techniques, the second derivatives of U can be approximated by linear combinations of the values of U at neighboring points. Using this fact, the discretized differential equation becomes a set of linear equations of the form:

A * U = F

These linear equations are then solved using a form of the Jacobi iterative method with a relaxation factor.

Directives are used in this code to achieve parallelism.

All do loops are parallized with default 'static' scheduling.

Exercise 8: Helmholtz Equation

(2) Compile the code

```
> cd ~/OpenMP/User_Codes/Courses/CS205/OpenMP/Example8
> make
```

(3) Run the program with different thread counts, e.g., 1, 2, 4, and 8

```
> sbatch sbatch.run
```

(4) Explore the output, e.g.,

```
> cat omp_helmholtz.dat
```

Exercise 9: Poisson Equation

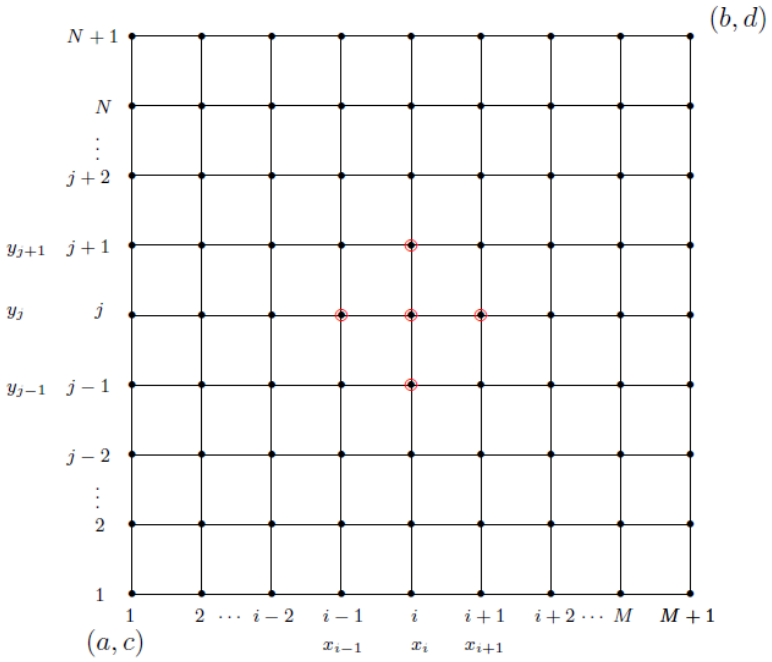
$$\nabla^2 u = f$$

$$\frac{u_{i-1,j} - 2u_{i,j} + u_{i+1,j}}{\Delta x^2} + \frac{u_{i,j-1} - 2u_{i,j} + u_{i,j+1}}{\Delta y^2} = f_{i,j}, \quad 2 \le i \le M, \quad 2 \le j \le N$$

2D discretization

$$u_{i,j}^{(n+1)} = \frac{\left(u_{i-1,j}^{(n)} + u_{i+1,j}^{(n)}\right)\Delta y^2 + \left(u_{i,j-1}^{(n)} + u_{i,j+1}^{(n)}\right)\Delta x^2 - \Delta x^2 \Delta y^2 f_{i,j}}{2\left(\Delta x^2 + \Delta y^2\right)}$$

Iterative update of Jacobi iterations



2D finite difference grid

Exercise 9: Poisson Equation

(1) Description – This example computes an approximate solution to the Poisson equation in a rectangular region, using OpenMP to carry out the Jacobi iteration in parallel.

The version of Poisson's equation being solved here is

- (d/dx d/dx + d/dy d/dy) U(x,y) = F(x,y)

over the rectangle 0 <= X <= 1, 0 <= Y <= 1, with exact solution

U(x,y) = sin (pi * x * y)

so that

F(x,y) = pi^2 * (x^2 + y^2) * sin (pi * x * y)

and with Dirichlet boundary conditions along the lines x = 0, x = 1, y = 0 and y = 1.

Approximate solution is computed by discretizing the geometry, assuming that DX = DY.

Along with the boundary conditions at the boundary nodes, we have a linear system for U. We can apply the Jacobi iteration to estimate the solution to the linear system.

Exercise 9: Poisson Equation

OpenMP is used in this example to carry out the Jacobi iteration in parallel.

For larger matrices the Jacobi iterations can converge very slowly.

In addition, different linear solvers can be used to improve performance.

(2) Compile the code

```
> cd ~/OpenMP/User_Codes/Courses/CS205/OpenMP/Example9
> make
```

(3) Run the program with different thread counts, e.g., 1, 2, 4, and 8

```
> sbatch sbatch.run
```

(4) Explore the output, e.g.,

```
> cat omp_poisson.dat
```

Exercise 10: Molecular Dynamics Simulations

Molecular Dynamics is a N-body simulation for studying the physical movements of atoms and molecules

The particles are allowed to interact for a fixed period of time

In most cases, particle trajectories are determined by solving numerically Newton's equations of motion for a system of interacting particles.

The forces between the particles and their potential energies are calculated using inter-particle potentials – empirical, semi-empirical, *ab initio*

$$m_i \ddot{\boldsymbol{r}}_i = \boldsymbol{f}_i \qquad \boldsymbol{f}_i = -\frac{\partial}{\partial \boldsymbol{r}_i}\mathcal{U}$$

$$\mathcal{U}_{\text{non-bonded}}(\boldsymbol{r}^N) = \sum_i u(\boldsymbol{r}_i) + \sum_i \sum_{j>i} v(\boldsymbol{r}_i, \boldsymbol{r}_j) + \dots .$$

$$v^{\text{LJ}}(r) = 4\varepsilon \left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6} \right]$$

Exercise 10: Molecular Dynamics Simulations

(1) Description – Program carries out a molecular dynamics simulation, using OpenMP for parallel execution. The specific example simulates the dynamics of 1000 particles performing 400 time steps.

(2) Compile the code

```
> cd ~/OpenMP/User_Codes/Courses/CS205/OpenMP/Example10
> make
```

(3) Run the program with different thread counts, e.g., 1, 2, 4, and 8

```
> sbatch sbatch.run
```

(4) Explore the output, e.g.,

```
> cat omp_md.dat
```

Good Practices and Summary

- Use “top” to check if your code is using the number of threads you set
  - The process should be using number of threads x 100% of load
  - Underloaded applications are caused by thread contention or thread starvation
- Run a scaling test
  - Take the same amount of work and divide it between 1, 2, 4, 8, etc., threads
  - Ideal scaling would be that the amount of time it takes to do work will half every time you double the number of threads
- After you complete your scaling test look at results and set thread count at the point where you still get appreciable performance gains due to parallelization