HARVARD
UNIVERSITY

USING OPENMP on FAS-OnDemand

Francesco Pontiggia
support requests: fasondemand@rc.fas.harvard.edu

March 2021
CS205

Objectives

- Introduction to FAS-OnDemand and basic concepts of cluster computing.
- Give you basic knowledge to write simple parallel OpenMP programs.
- Provide the information required for running your OpenMP applications on FAS-OnDemand.

What is FAS-OnDemand

FAS-OnDemand is a High Performance Computing Cluster dedicated to support academic coursework.

Access to the cluster is provided via an interactive computing portal based on the OpenOnDemand project (OSC).

Students can access the portal directly from Canvas, and from there schedule interactive computational tasks like Jupyter Notebooks or RStudio sessions, or run interactive jobs from Linux command line interface.

Recommended browsers are Firefox and Chrome.

FAS-OnDemand is hosted by Harvard's FAS Research Computing group (FAS-RC) and supported by a collaboration of FAS-RC and FAS-Academic Technology.

Support requests can be sent to **fasondemand@rc.fas.harvard.edu**

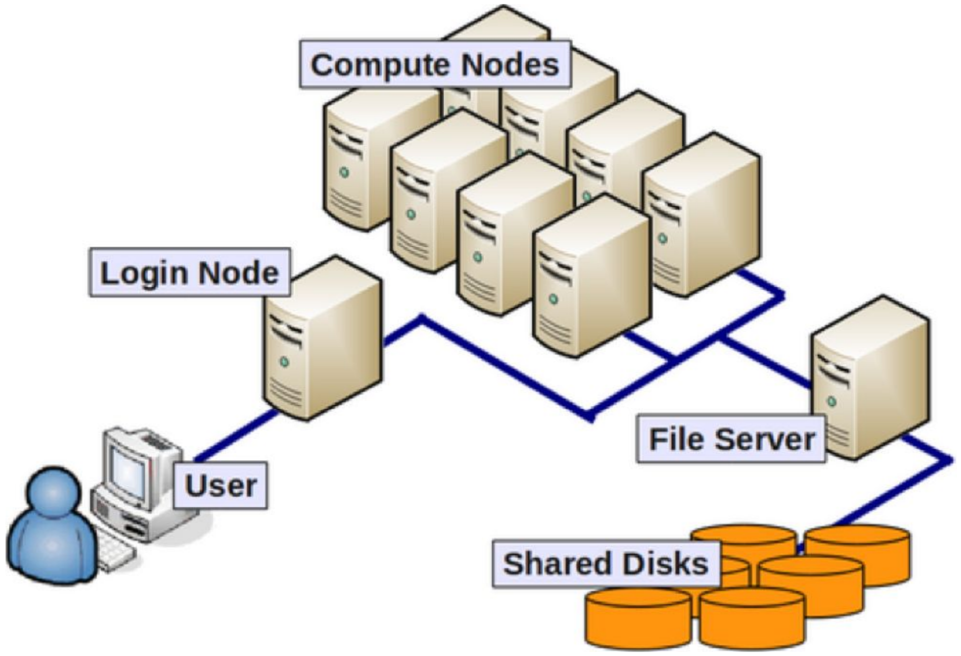
Cluster Basics

FAS-OnDemand Portal (aka Login Nodes):

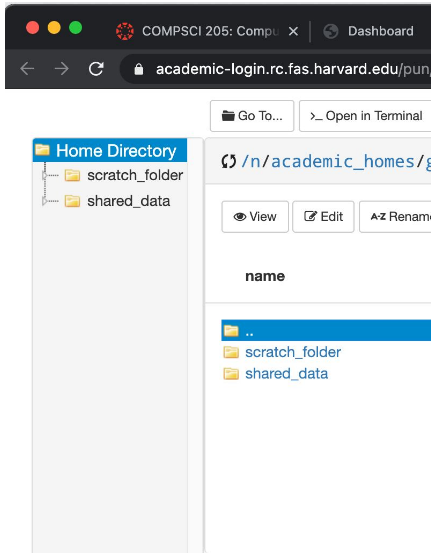
- prepare input and stage your calculations
- interact with the scheduler
- no computationally expensive processes allowed

Compute nodes:

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



Storage:

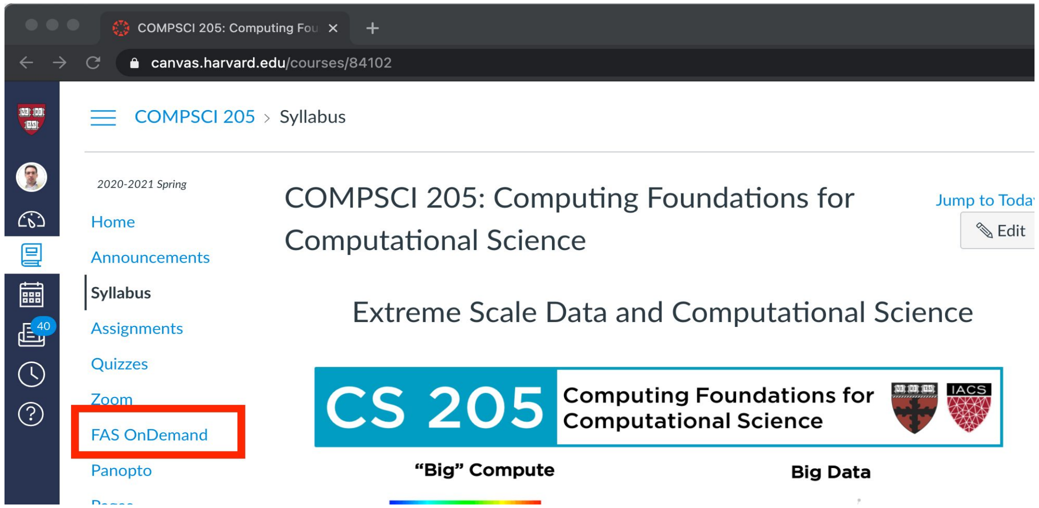


- **Home Directory**: 20G per user (not suitable for IO intense work)

- **scratch_folder**: 10T per class (recommended for IO intense work)

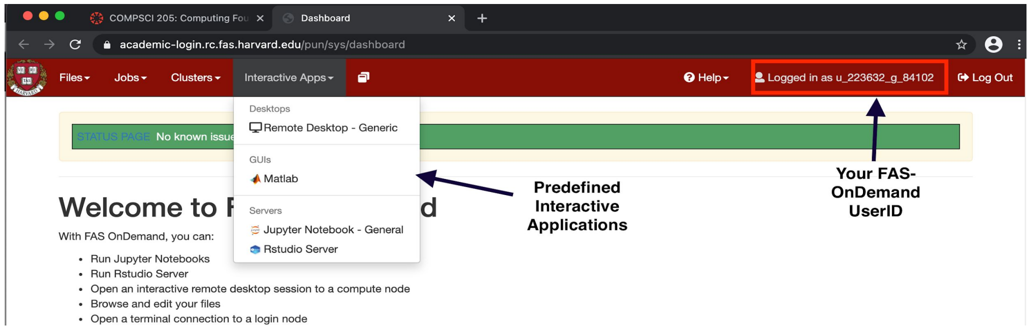
- **shared_data:** rw for instructors. ro for students.

Access to FAS-OnDemand



COMPSCI 205: Computing Foundations for Computational Science

Extreme Scale Data and Computational Science



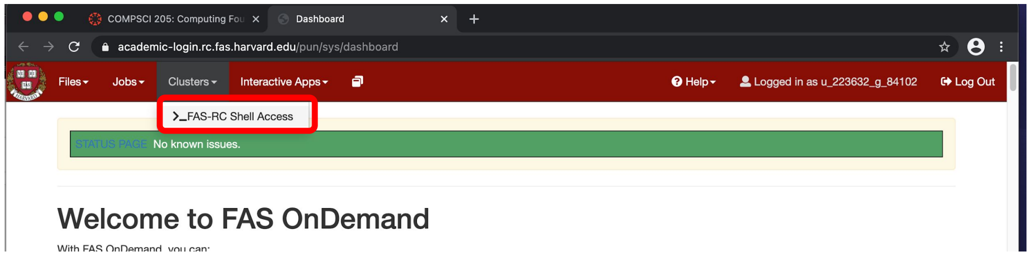
COMPSCI 205: Computing Fou
Dashboard
academic-login.rc.fas.harvard.edu/pun/sys/dashboard
Files
Jobs
Clusters
Interactive Apps
Help
Logged in as u_223632_g_84102
Log Out
Desktops
Remote Desktop - Generic
GUIs
Matlab
Servers
Jupyter Notebook - General
Rstudio Server
Predefined Interactive Applications
Your FAS-OnDemand UserID
STATUS PAGE

No known issue

Welcome to F d

With FAS OnDemand, you can:

- Run Jupyter Notebooks
- Run Rstudio Server
- Open an interactive remote desktop session to a compute node
- Browse and edit your files
- Open a terminal connection to a login node



COMPSCI 205: Computing Fou
Dashboard
academic-login.rc.fas.harvard.edu/pun/sys/dashboard
Files
Jobs
Clusters
Interactive Apps
Help
Logged in as u_223632_g_84102
Log Out
>_FAS-RC Shell Access
STATUS PAGE

No known issues.

Welcome to FAS OnDemand

With FAS OnDemand, you can:

1- Login to https://canvas.harvard.edu and navigate to CS205 course page

2- click on the "FAS-OnDemand" link on the left menu

3- the first time you access your account will be created on the fly.
(it might take a minute, … please be patient)

4- you will land on a dashboard like the one shown on the left.

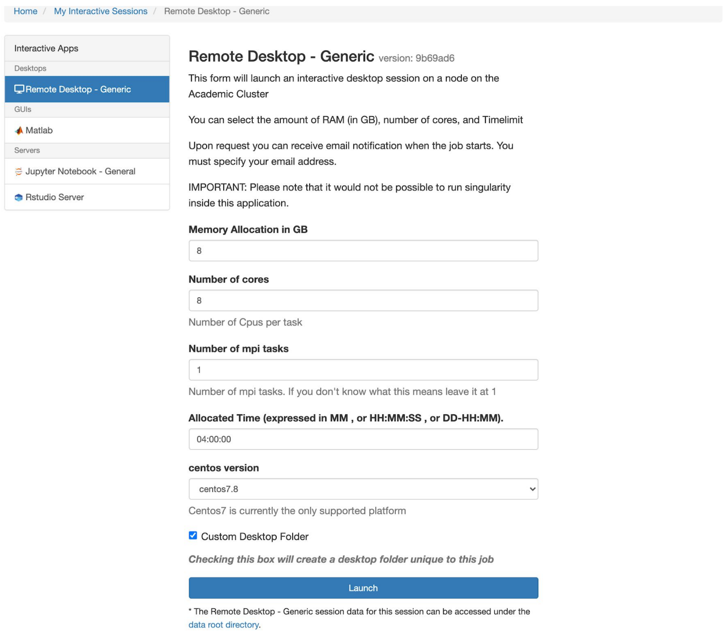
On top right you will see your UserID. Please always include that when you send a support request to fasondemand@rc.fas.harvard.edu

From "interactive apps" menu you can request sessions for some popular applications.

Or you can click on Clusters -> Shell Access to open a shell terminal and work directly from Linux CLI.

Managing Jobs

You can submit a job using one of the pre-defined interactive jobs forms from the menu “Interactive Apps”,



Home / My Interactive Sessions / Remote Desktop - Generic

Interactive Apps
Desktops
Remote Desktop - Generic
GUIs
Matlab
Servers
Jupyter Notebook - General
Rstudio Server

Remote Desktop - Generic version: 9b69ad6

This form will launch an interactive desktop session on a node on the Academic Cluster

You can select the amount of RAM (in GB), number of cores, and Timelimit

Upon request you can receive email notification when the job starts. You must specify your email address.

IMPORTANT: Please note that it would not be possible to run singularity inside this application.

Memory Allocation in GB
8

Number of cores
8
Number of Cpus per task

Number of mpi tasks
1
Number of mpi tasks. If you don't know what this means leave it at 1

Allocated Time (expressed in MM , or HH:MM:SS , or DD-HH:MM).
04:00:00

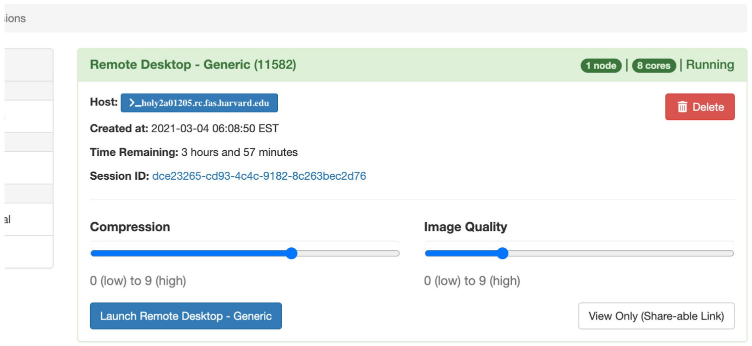
centos version
centos7.8
Centos7 is currently the only supported platform

☑ Custom Desktop Folder
Checking this box will create a desktop folder unique to this job

Launch

* The Remote Desktop - Generic session data for this session can be accessed under the data root directory.

And access your job from the stanza in the tab “My Interactive Sessions”,



Remote Desktop - Generic (11582) 1 node | 8 cores | Running

Host: >_holy2a01205.rc.fas.harvard.edu   Delete

Created at: 2021-03-04 06:08:50 EST

Time Remaining: 3 hours and 57 minutes

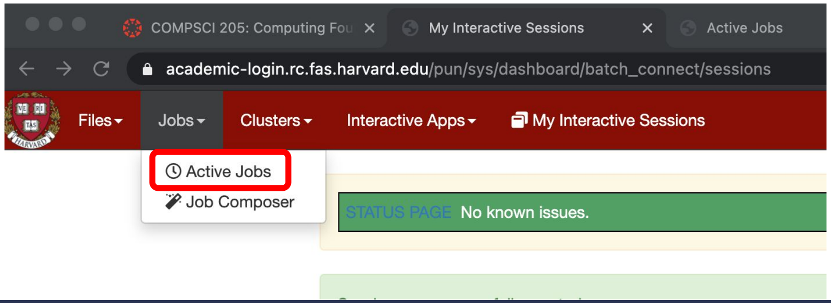
Session ID: dce23265-cd93-4c4c-9182-8c263bec2d76

Compression 0 (low) to 9 (high)

Image Quality 0 (low) to 9 (high)

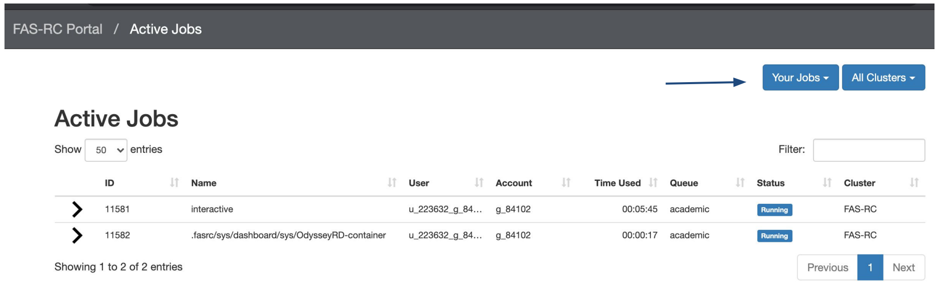
Launch Remote Desktop - Generic   View Only (Share-able Link)

If you want to check the status of all your jobs (interactive and not) you can use the app “



COMPSCI 205: Computing Fou
My Interactive Sessions
Active Jobs
academic-login.rc.fas.harvard.edu/pun/sys/dashboard/batch_connect/sessions
Files
Jobs
Clusters
Interactive Apps
My Interactive Sessions
Active Jobs
Job Composer
STATUS PAGE No known issues.

From there you can inspect your current jobs and delete them if needed



FAS-RC Portal / Active Jobs

Your Jobs   All Clusters

Active Jobs

Show 50 entries   Filter:

	ID	Name	User	Account	Time Used	Queue	Status	Cluster
>	11581	interactive	u_223632_g_84…	g_84102	00:05:45	academic	Running	FAS-RC
>	11582	.fasrc/sys/dashboard/sys/OdysseyRD-container	u_223632_g_84…	g_84102	00:00:17	academic	Running	FAS-RC

Showing 1 to 2 of 2 entries   Previous 1 Next

Managing Job from CLI

- 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
    ```
  - salloc: submit an interactive test job
    ```
    > salloc [ options for resource request ]
    ```
  - 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
    ```

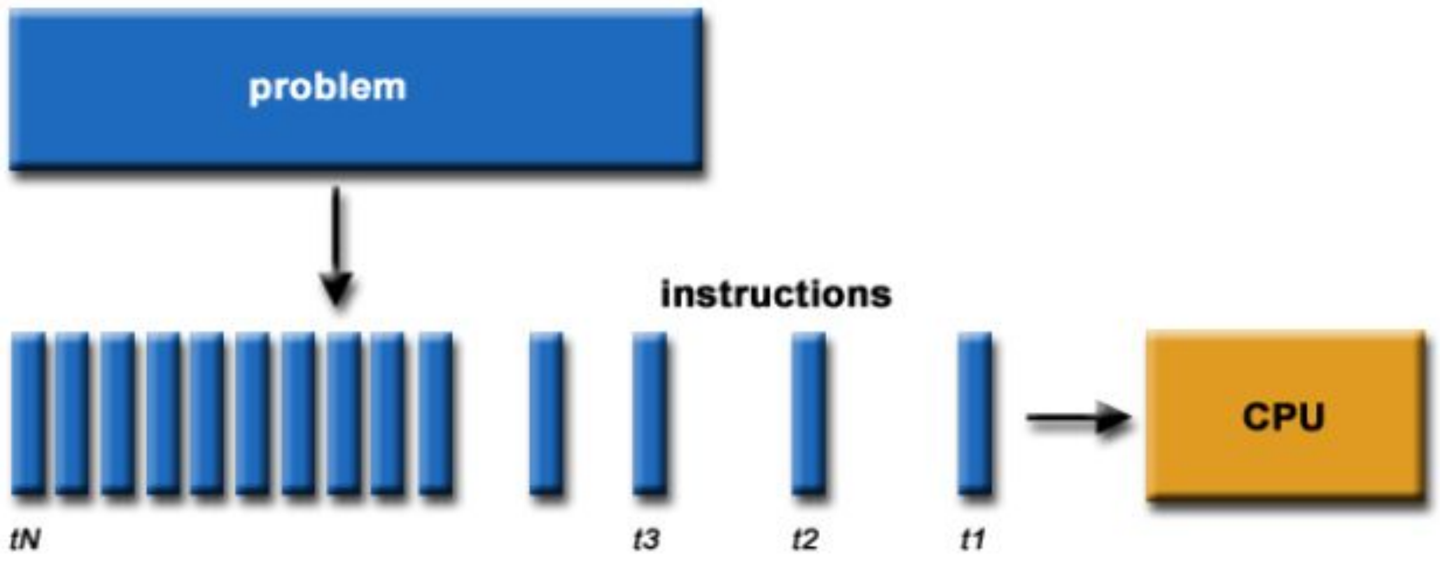
https://docs.rc.fas.harvard.edu/kb/convenient-slurm-commands/
https://docs.rc.fas.harvard.edu/kb/running-jobs/

Important Note: on FAS-OnDemand the partition is called “academic” and a “gpu” partition will be available shortly

Serial Computation

Traditionally software has been written for serial computations:

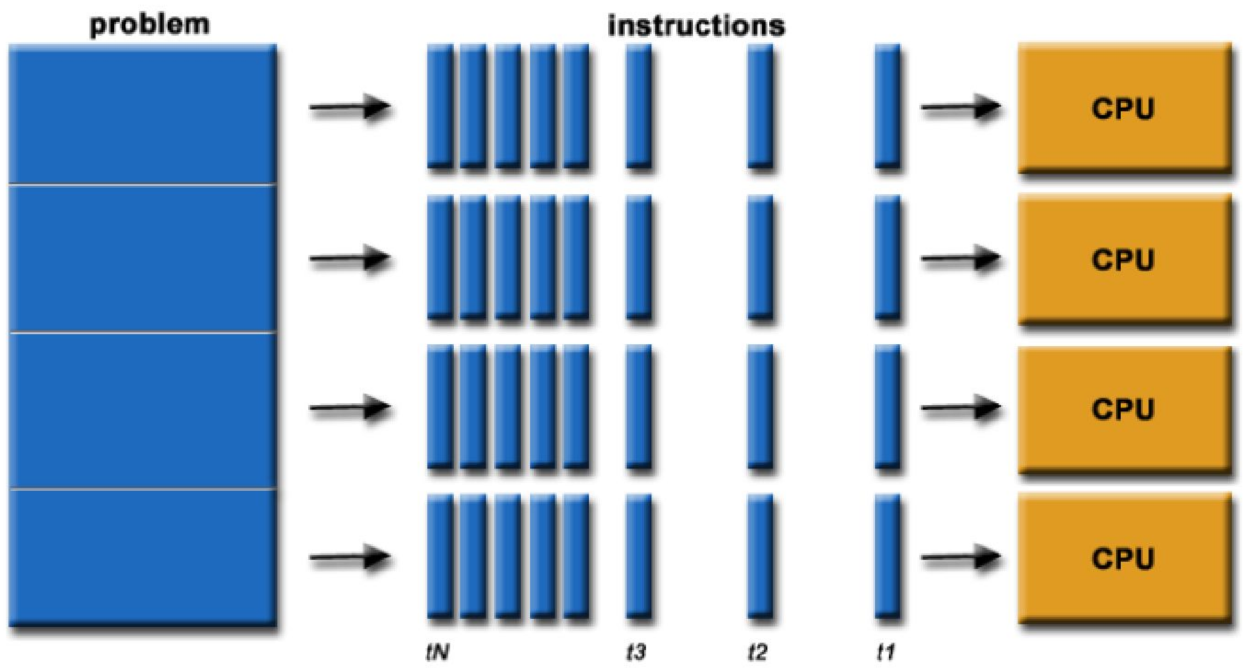
- To be run on a single computer having a single Central Processing Unit (CPU)
- A problem is broken into a discrete set of instructions
- Instructions are executed one after another
- Only one instruction can be executed at any moment in time



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



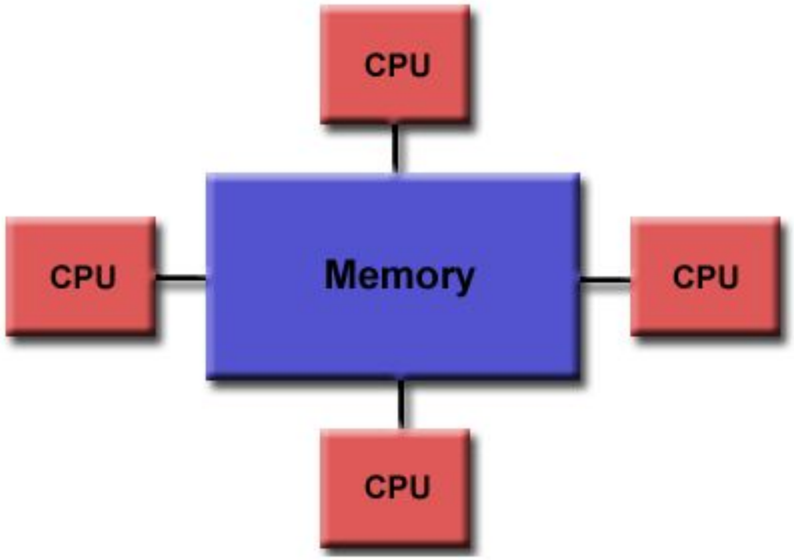
OpenMP Basics

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



Note on Additional Scientific Software

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

```
$> module load gcc/9.3.0-fasrc01  # Loads GCC compiler

$> module avail                   # Lists available modules

$> module list                    # Lists loaded modules

$> module purge                   # Unloads ALL modules

$> module-query gcc               # Finds modules

$> module-query gcc/9.3.0-fasrc01 # Gives more information
```

Compiling OpenMP Programs

Compiler/Platform	Compiler	Flag

Intel	`icc`
	`icpc`
	`ifort`
		`-openmp`

GNU	`gcc`
	`g++`
	`g77`
	`gfortran`
		`-fopenmp`

GNU: Recommended on FAS-OnDemand

```
module load gcc/9.3.0-fasrc01
gcc -o omp_test.x omp_test.c -fopenmp
```

Intel:

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

Running OpenMP Programs

Interactive test jobs:

(1) Open a remote desktop session
Or alternatively from the Shell App start an interactive session

```
> salloc -c 4 --mem=4G -t 120
```

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

```
> module load gcc/9.3.0-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=$SLURM_CPUS_PER_TASK
> ./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 academic         # Queue name
#SBATCH -t 0-00:30          # Time (D-HH:MM)
#SBATCH --mem=4G            # Memory in GB
#SBATCH -c 8                # Number of cores
module load gcc/9.3.0-fasrc01
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
   2b. adjust scheduling # in breakout rooms
3. Parallel sections in OpenMP
4. Reduction – parallel dot product
5. Orphaned directives
6. Scaling – speedup and efficiency # in breakout rooms
7. Matrix-Matrix multiplication
8. Helmholtz Equation # bonus exercises
9. Poisson Equation # bonus exercises
10. Molecular Dynamics (MD) # bonus exercises

Exercises - Setup

1. **Start a remote desktop session OR an interactive session with salloc**

```
@academic-login02 >$ salloc -t 120 --mem=4G -c 4
@holy2a01001 >$ #now you are on a compute node
```

2. Navigate to your scratch folder

```
>$ cd  scratch_folder
```

3. Get copy of the OpenMP exercises **from the shared_data folder**.

```
>$ cp ~/shared_data/OpenMP.tar .
>$ tar xf OpenMP.tar
>$ cd OpenMP
```

Or from Github at
https://github.com/fasrc/User_Codes/tree/master/Courses/CS205/OpenMP

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

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

```
>$ module load gcc/9.3.0-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 ~/scratch_folder/OpenMP/User_Codes/Courses/CS205/OpenMP/Example1
> make clean ; make
```

(3) Run the program

```
> export OMP_NUM_THREADS=$SLURM_CPUS_PER_TASK #equals to cores requested
> ./omp_hello.x
Hello World from thread = 2
Hello World from thread = 3
Hello World from thread = 1
Hello World from thread = 0
Number of threads = 4
```

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

```
> export OMP_NUM_THREADS=2
```

HARVARD UNIVERSITY FAS RC

Exercise 1b: use a batch job

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

(2) Compile the program

```
> cd ~/scratch_folder/OpenMP/User_Codes/Courses/CS205/OpenMP/Example1
> make clean ; 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.,

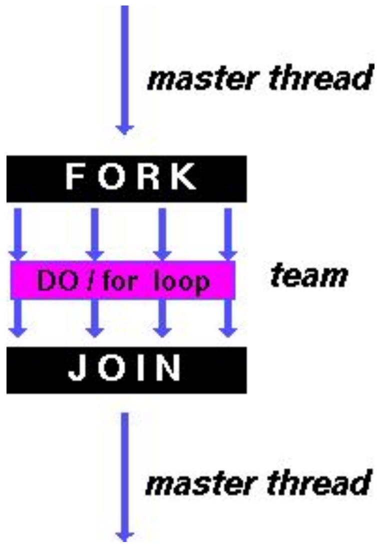
```
> cat omp_hello.dat
Hello World from thread = 0
Hello World from thread = 1
Hello World from thread = 2
Number of threads = 4
Hello World from thread = 3
```

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/Example2
> make
```

(4) Run the program

```
> export OMP_NUM_THREADS=$SLURM_CPUS_PER_TASK
> ./omp_loop.x
> ./omp_loop.x | sort
```

(5) Explore the output. Piping through the sort utility makes it easier to view how loop iterations were actually scheduled across the team of threads.

Exercise 2b: 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
```

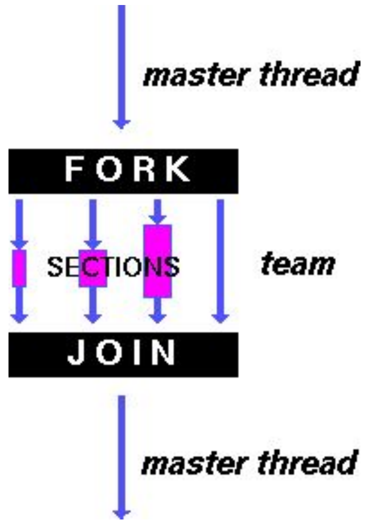
Notice the difference in output compared to dynamic scheduling. Specifically, notice that thread 0 gets the first chunk, thread 1 the second chunk, and so on.

(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/Example3
> make
```

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

```
> export OMP_NUM_THREADS=$SLURM_CPUS_PER_TASK
> omp_sections.x | sort
```

(5) Explore the output. 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/Example4
> make
```

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

```
> export OMP_NUM_THREADS=$SLURM_CPUS_PER_TASK
> omp_sections.x | sort
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
```

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

```
> export OMP_NUM_THREADS=2
```

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/Example5
> make
```

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

```
> export OMP_NUM_THREADS=$SLURM_CPUS_PER_TASK
> omp_dot2.x | sort
```

(4) Explore the output

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

Example 6: Scaling – Compute PI in Parallel

How much faster will the program run?

Speedup:

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

$T(1)$ → Time to complete the job on **one** thread

$T(n)$ → 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

Monte-Carlo Approximation of PI:

1. Inscribe a circle in a square

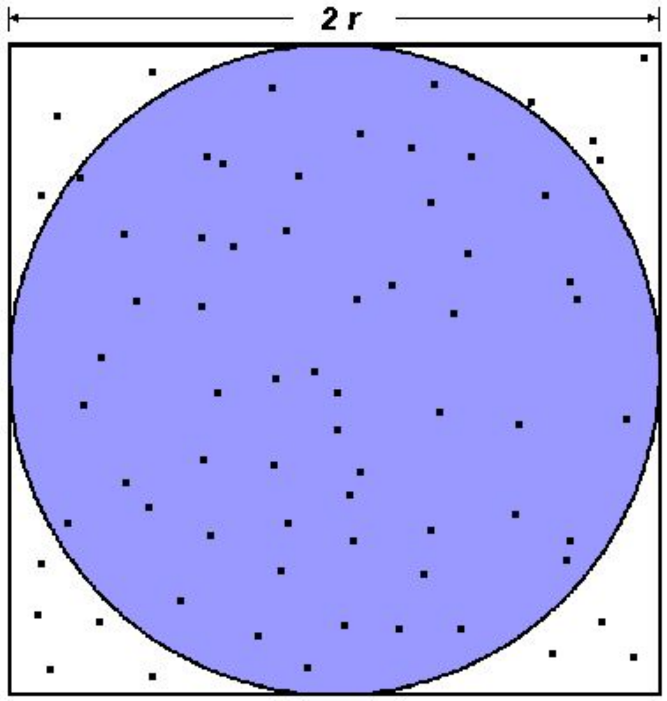
2. Randomly generate points in the square

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/Example6
> make
```

(3) Explore the sbatch.run file and submit it

```
> 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) Runing the program with different thread number – 1, 2, 4, 8 – and recording the run times for each case allows us to compute the speedup and efficiency

Example 6: Scaling – Compute PI in Parallel

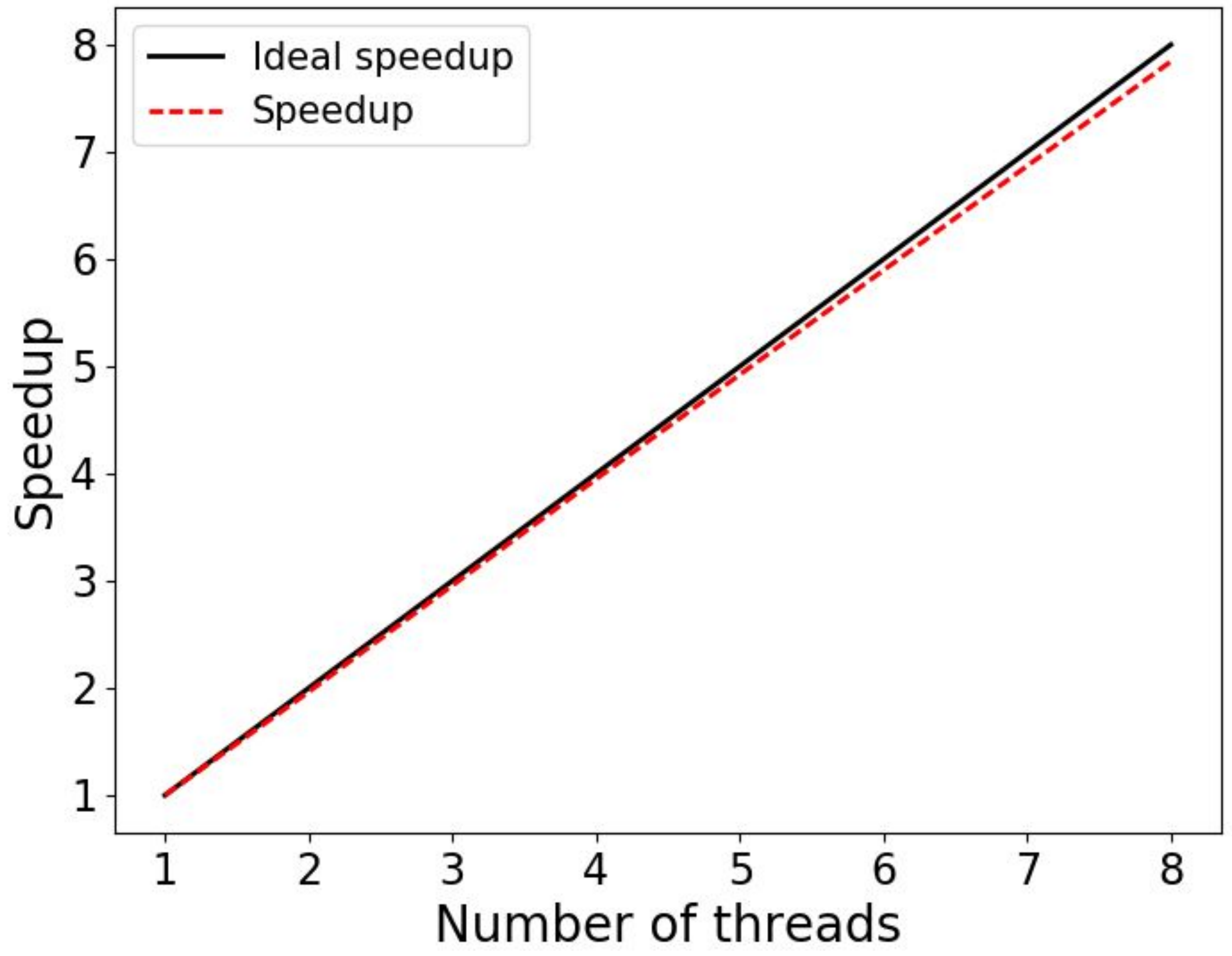
You may use the speedup.py Python code to generate to calculate the speedup and efficiency. It generates the below table plus a speedup figure

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

HARVARD UNIVERSITY

FAS RC

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/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/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

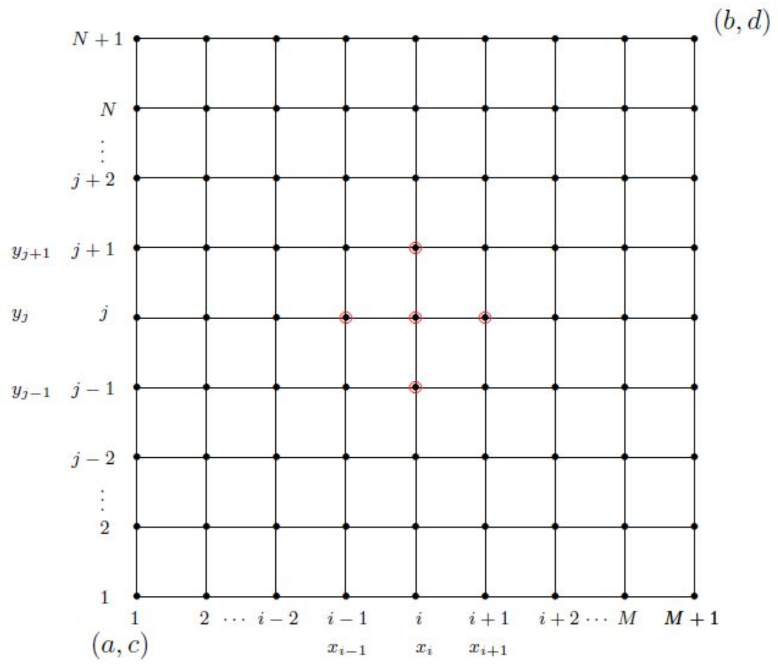
$$\nabla^2 u = f$$

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

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/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*

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

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

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

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/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