Parallel Job Workflows using OpenMP and MPI

Plamen Krastev, PhD
Harvard - FAS Research Computing
Objectives

- To advise you on the best practices for running parallel workflows on the FASRC cluster
- To provide the basic knowledge required for (implementing and) running your parallel OpenMP and MPI applications efficiently on the FASRC cluster
Overview

- Best Practices
- Brief Introduction to Parallel Computing
- Embarrassingly Parallel Jobs / Workflows
- OpenMP Jobs / Workflows
- MPI Jobs / Workflows
- Hybrid (MPI+OpenMP) Jobs / Workflows
Best Practices (1)

- Do small scale testing prior to large scale runs
- Ensure your jobs will run at least 10 minutes
- Make sure your jobs are well constrained
- Make sure your data is on a filesystem that can handle the I/O load
- Be aware of potential bottlenecks in your workflow
- Be cognizant of your fairshare [https://docs.rc.fas.harvard.edu/kb/fairshare/](https://docs.rc.fas.harvard.edu/kb/fairshare/)
Best Practices (2)

- Ensure your code is operating as expected
- Understand the scaling of your code
- Have your primary code in a **git** repo
- Keep backups of critical data
- Have checkpoints
- Optimize your code and workflow
Overview

- Best Practices

- Brief Introduction to Parallel Computing

- Embarrassingly Parallel Jobs / Workflows

- OpenMP Jobs / Workflows

- MPI Jobs / Workflows

- Hybrid (MPI+OpenMP) Jobs / Workflows
What is High Performance Computing (HPC)?

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
Overview

- Best Practices
- Brief Introduction to Parallel Computing
  - Embarrassingly Parallel Jobs / Workflows
  - OpenMP Jobs / Workflows
  - MPI Jobs / Workflows
  - Hybrid (MPI+OpenMP) Jobs / Workflows
Embarrassingly Parallel
Embarrassingly Parallel

- Running many serial jobs in parallel
  - Parameter Sweeps
    - [Link](https://www.rc.fas.harvard.edu/wp-content/uploads/2015/04/Parameter-Sweep.pdf)
  - Data Transfers
  - Data Analysis Pipelines

- When possible, use `serial_requeue partition`

- Potential Problems/Bottlenecks
  - Filesystem I/O
  - Re-queue
  - SLURM Thrashing
    - Short runs
    - Lots of scheduler queries
Submitting Large Number of Serial Jobs

- **Job Launcher Scripts**
  - Use scripting language (e.g., Bash, Python, Perl, R) to construct and submit jobs

- **SLURM Job Arrays**
  - Works best for individual tasks that take 10+ minutes

- **Single job: for loop in job-script**
  - Works best for lots really short tasks (seconds)

**Genuine Warning:** Resist the urge to use R / bash to create 1000s of files and submit each as a separate job

Reference:
[https://docs.rc.fas.harvard.edu/kb/submitting-large-numbers-of-jobs/](https://docs.rc.fas.harvard.edu/kb/submitting-large-numbers-of-jobs/)
Job Launcher Scripts

- Use scripting language (e.g., Bash, Python, R, Perl) to construct and submit jobs

- Advantages
  - Full Flexibility and Control

- Disadvantage
  - Can get rather complex depending on workflow

- Examples:
  - https://github.com/fasrc/slurm_migration_scripts
SLURM Job Arrays

- Use **SLURM job arrays** to process data

- Advantages
  - Easy to use
  - Quick
  - Easy on the scheduler

- Disadvantages
  - Problems must fit into the Job Array style

- Examples:
SLURM Job Arrays

- `#SBATCH --array=indexes`

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1-10</td>
<td>1,2,3,4,5,6,7,8,9,10</td>
</tr>
<tr>
<td>2-20:2</td>
<td>2,4,6,8,10,12,14,16,18,20</td>
</tr>
<tr>
<td>1,3,5,7,11,21</td>
<td>1,3,5,7,11,21</td>
</tr>
<tr>
<td>2-20%2</td>
<td>2,4 then 6,8 then 10,12 ...</td>
</tr>
</tbody>
</table>

- SLURM job script variables
  - `%A = JobId` and `%a = IndexID
    - **Ex:** `$SBATCH -o stdout-%A_%a.o`
  - `$SLURM_ARRAY_TASK_ID`
    - **Ex:** `R CMD input.R input.$SLURM_ARRAY_TASK_ID.out`
#!/bin/bash
#SBATCH -J array_test
#SBATCH -p shared
#SBATCH --array 1
#SBATCH -t 00:10:00
#SBATCH --mem=4G
#SBATCH -o %A-%a.o
#SBATCH -e %A-%a.e
#SBATCH --array=100,200,300

This is per array task resource needs

# Load software environment
module load R/4.3.1-fasrc01

input=serial_sum.R

# Execute code
srun -n 1 -c 1 R CMD BATCH $input $input.$SLURM_ARRAY_TASK_ID.out

https://github.com/fasrc/User_Codes/tree/master/Parallel_Computing/EP/Example1
Using SLURM Array Index in R

tid <- as.integer(Sys.getenv('SLURM_ARRAY_TASK_ID'))
res <- serial_sum(x=tid)
print(res)
Single Job: *for loop* in in job-script

```bash
#!/bin/bash
#SBATCH -J test_job
#SBATCH -p shared
#SBATCH -c 1
#SBATCH -t 00:10:00
#SBATCH --mem=4G
#SBATCH -o test_job.out
#SBATCH -e test_job.err

# Load software environment
module load R/4.1.0-fasrc01

input=serial_sum.R

# Execute code
for i in 100 200 300; do
    export inp=$i
    srun -n 1 -c 1 R CMD BATCH $input $input.$inp.out
done
```

Overview

- Best Practices
- Brief Introduction to Parallel Computing
- Embarrassingly Parallel Jobs / Workflows
  - OpenMP Jobs / Workflows
  - MPI Jobs / Workflows
  - Hybrid (MPI+OpenMP) Jobs / Workflows
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
Threading Languages Interfaces

Pthreads
OpenMP
OpenCL/CUDA
OpenACC
Python
R
Perl
MATLAB (PCT)
Others
Compiling OpenMP Programs

<table>
<thead>
<tr>
<th>Compiler/Platform</th>
<th>Compiler</th>
<th>Flag</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel</td>
<td>icx (C)</td>
<td>-qopenmp</td>
</tr>
<tr>
<td></td>
<td>icpx (C++)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ifx (Fortran)</td>
<td></td>
</tr>
<tr>
<td>GNU</td>
<td>gcc</td>
<td>-fopenmp</td>
</tr>
<tr>
<td></td>
<td>g++</td>
<td></td>
</tr>
<tr>
<td></td>
<td>g77</td>
<td></td>
</tr>
<tr>
<td></td>
<td>gfortran</td>
<td></td>
</tr>
</tbody>
</table>

Intel:

module load intel/23.2.0-fasrc01
icx -o omp_test.x omp_test.c -qopenmp

GNU:

module load gcc/13.2.0-fasrc01
gcc -o omp_test.x omp_test.c -fopenmp

https://github.com/fasrc/User_Codes/tree/master/Parallel_Computing/OpenMP
Running OpenMP Programs (1)

**Interactive / test jobs:**

1. Start an interactive bash shell
   ```bash
   > salloc -p test -c 4 --mem=4G -t 0-06:00
   ```

2. Load required modules, e.g.,
   ```bash
   > module load gcc/10.2.0-fasrc01
   ```

3. Compile (or use a Makefile)
   ```bash
   > gcc -o omp_hello.x omp_hello.c -fopenmp
   ```

4. Set number of OpenMP threads
   ```bash
   > export OMP_NUM_THREADS=4
   ```

5. Run the executable
   ```bash
   > ./omp_hello.x
   ```
   
   ```bash
   [pkrastev@holy7c19314 Example1]$ ./omp_hello.x
   Hello World from thread = 1
   Hello World from thread = 3
   Hello World from thread = 2
   Hello World from thread = 0
   Number of threads = 4
   ```
Running OpenMP Programs (2)

Batch Jobs:

(1) Prepare a batch-job submission script

```bash
#!/bin/bash
#SBATCH -J omp_job            # Job name
#SBATCH -o slurm.out          # STD output
#SBATCH -e slurm.err          # STD error
#SBATCH -p shared             # Queue / Partition
#SBATCH -t 0-00:30            # Time (D-HH:MM)
#SBATCH --mem=4000            # Reserved memory (default in MB)
#SBATCH -c 8                  # Number of threads
#SBATCH -N 1                  # Number of nodes
export OMP_NUM_THREADS=$SLURM_CPUS_PER_TASK
module load gcc/13.2.0-fasrc01 # Load required modules
srun -c $SLURM_CPUS_PER_TASK ./omp_test.x
```

(2) Submit the job to the queue

```bash
> sbatch omp_test.run
```
Example: Scaling - Compute PI in Parallel

Monte-Carlo approximation of PI

Images credit: LLNL

https://hpc.llnl.gov/documentation/tutorials/introduction-parallel-computing-tutorial##ExamplesPI
Example: Scaling - Compute PI in Parallel

(1) Setup - get a copy of the code and compile it, e.g.,
> mkdir ~/OpenMP
> cd OpenMP
> git clone https://github.com/fasrc/User_Codes.git

(2) Review the source code and compile the program
> cd User_Codes/Parallel_Computing/OpenMP/Example3
> module load intel/23.2.0-fasrc01
> make

(3) Run the program
> sbatch sbatch.run

(4) Explore the output (the “omp_dot.dat” file), e.g.,
> cat omp_pi.dat
Number of threads:  8
Exact value of PI:  3.14159
Estimate of PI:  3.14158
Time:  0.32 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 (*NOTE: Currently set up to run directly with 1, 2, 4, 8 threads and generate speedup figure*)

https://github.com/fasrc/User_Codes/tree/master/Parallel_Computing/Example3
Example: 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: 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.

<table>
<thead>
<tr>
<th>Nthreads</th>
<th>Walltime</th>
<th>Speedup</th>
<th>Efficiency (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2.56</td>
<td>1.00</td>
<td>100.00</td>
</tr>
<tr>
<td>2</td>
<td>1.28</td>
<td>2.00</td>
<td>100.00</td>
</tr>
<tr>
<td>4</td>
<td>0.64</td>
<td>4.00</td>
<td>100.00</td>
</tr>
<tr>
<td>8</td>
<td>0.32</td>
<td>8.00</td>
<td>100.00</td>
</tr>
</tbody>
</table>

https://github.com/fasrc/User_Codes/tree/master/Parallel_Computing/Example3
Example: Scaling - Compute PI in Parallel
Overview

- Best Practices
- Brief Introduction to Parallel Computing
- Embarrassingly Parallel Jobs / Workflows
- OpenMP Jobs / Workflows
- MPI Jobs / Workflows
- Hybrid (MPI+OpenMP) Jobs / Workflows
What is MPI?

- **MPI** = **M**assage **P**assing **I**nterface

- MPI is a *specification* for the developers and users of message passing libraries. By itself, it is *NOT* a library.

- MPI primarily addresses the message-passing parallel programming model: data is moved from the address space of one process to that of another process through cooperative operations on each process.

- Most recent version is MPI-3

- Actual MPI library implementations differ in which version and features of the MPI standard they support.
MPI Programming Model

- Originally MPI was designed for distributed memory architectures

- As architectures evolved, MPI implementations adapted their libraries to handle shared, distributed, and hybrid architectures

- Today, MPI runs on virtually any hardware platform
  - Shared Memory
  - Distributed Memory
  - Hybrid

- Programming model remains clearly distributed memory model, regardless of the underlying physical architecture of the machine

- Explicit parallelism – programmer is responsible for correct implementation of MPI
Reasons for using MPI

- **Standardization** - MPI is the only message passing specification which can be considered a standard. It is supported on virtually all HPC platforms.

- **Portability** - There is little or no need to modify your source code when you port your application to a different platform that supports (and is compliant with) the MPI standard.

- **Performance Opportunities** - Vendor implementations should be able to exploit native hardware features to optimize performance. Any implementation is free to develop optimized algorithms.

- **Functionality** - There are over 430 routines defined in MPI-3, which includes the majority of those in MPI-2 and MPI-1.

- **Availability** - A variety of implementations are available, both vendor and public domain.
MPI Language Interfaces

- C/C++
- Fortran
- Java
- Python (mpi4py, pyMPI, pypar, MYMPI)
- R (Rmpi)
- Perl (Parallel::MPI)
- MATLAB (Matlab Parallel Server / DCS)
- Others
## Compiling MPI Programs

<table>
<thead>
<tr>
<th>MPI Implementation</th>
<th>Compiler</th>
<th>Flag</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpenMPI</td>
<td>mpicc, mpicxx, mpif90, mpif77, mpifort</td>
<td>None</td>
</tr>
<tr>
<td>Intel MPI</td>
<td>mpiicx, mpiicpx, mpiifx</td>
<td>None</td>
</tr>
</tbody>
</table>

**Intel + OpenMPI / Mpich:**

module load intel/23.2.0-fasrc01
module load openmpi/4.1.5-fasrc03
mpicxx -o mpitest.x mpitest.cpp

**GNU + OpenMPI / Mpich:**

module load gcc/10.2.0-fasrc01
module load openmpi/4.1.1-fasrc01
mpicxx -o mpi_test.x mpi_test.cpp

**Intel + Intel-MPI:**

module load intel/23.2.0-fasrc01
module load intelmpi/2021.10.0-fasrc01
mpiicpx -o mpi_test.x mpi_test.cpp
Running MPI Programs (1)

Interactive test jobs:

(1) Start an interactive bash shell
> salloc -p test -n 4 --mem=4G -t 0-06:00

(2) Load required modules, e.g.,
> module load gcc/13.2.0-fasrc01 openmpi/4.1.5-fasrc03

(3) Compile your code (or use a Makefile)
> mpicxx -o mpitest.x mpitest.cpp

(4) Run the code
> mpirun -np 4 ./mpitest.x
Rank 0 out of 4
Rank 1 out of 4
Rank 2 out of 4
Rank 3 out of 4
End of program.
Running MPI Programs (2)

Batch jobs:

(1) Compile your code, e.g.,
> module load gcc/13.2.0-fasrc01 openmpi/4.1.5-fasrc03
> mpicxx -o mptest.x mptest.cpp

(2) Prepare a batch-job submission script

#!/bin/bash
#SBATCH -J mpi_job # Job name
#SBATCH -o slurm.out # STD output
#SBATCH -e slurm.err # STD error
#SBATCH -p shared # Queue / partition
#SBATCH -t 0-00:30 # Time (D-HH:MM)
#SBATCH --mem-per-cpu=4000 # Memory per MPI task
#SBATCH -n 8 # Number of MPI tasks
module load gcc/13.2.0-fasrc01 openmpi/4.1.5-fasrc03 # Load required modules
srun -n $SLURM_NTASKS --mpi=pmix ./hello_mpi.x

(3) Submit the job to the queue
> sbatch mpi_test.run
Running MPI Programs (3)

Intel & Intel-MPI

#!/bin/bash

#SBATCH -J mpitest       # job name
#SBATCH -o mpitest.out   # standard output file
#SBATCH -e mpitest.err   # standard error file
#SBATCH -p test          # partition
#SBATCH -n 8             # ntasks
#SBATCH -t 00:30:00      # time in HH:MM:SS
#SBATCH --mem-per-cpu=4000 # memory in megabytes

# --- Load the required software modules., e.g., ---
module load intel/23.2.0-fasrc01 intelmpi/2021.10.0-fasrc01

# --- Run the executable ---
# --- With Intel-MPI, you need to ensure it uses pmi2 instead of pmix ---
srun -n $SLURM_NTASKS --mpi=pmi2 ./mpitest.x
Running MPI Programs (4)

- Sometimes programs can be picky about having MPI available on all the nodes it runs on, so it is good to have MPI module loads in your `.bashrc` file.

- Some codes are topology sensitive thus the following `slurm` options can be helpful:
  - `--contiguous` # Contiguous set of nodes
  - `--ntasks-per-node` # Number of tasks per node
  - `--hint` # Bind tasks according to hints
  - `--distribution, -m` # Specify distribution method for tasks

- For hybrid mode jobs you would set both `--c` and `--n`.

https://slurm.schedmd.com/sbatch.html
https://slurm.schedmd.com/mc_support.html
https://www.rc.fas.harvard.edu/resources/documentation/software-development-on-odyssey/hybrid-mpiopenmp-codes-on-odyssey
MPI Examples

1. MPI Hello World program
2. Parallel FOR loops in MPI – dot product
3. Scaling – speedup and efficiency
4. Parallel Matrix-Matrix multiplication
5. Parallel Lanczos algorithm

https://github.com/fasrc/User_Codes/tree/master/Courses/CS205/MPI_2021
Overview

- Best Practices

- Brief Introduction to Parallel Computing

- Embarrassingly Parallel Jobs / Workflows

- OpenMP Jobs / Workflows

- MPI Jobs / Workflows

- Hybrid (MPI+OpenMP) Jobs / Workflows
Hybrid (MPI+OpenMP) Parallel Programming

- **OpenMP** is used for computationally intensive work on each node
- **MPI** is used for communication and data sharing between nodes
- This allows parallelism to be implemented to the full scale of a cluster

https://docs.rc.fas.harvard.edu/kb/hybrid-mpiopenmp-codes-on-odyssey/
Running Hybrid Applications

Example 1: 2 MPI tasks with 4 OpenMP threads per MPI task, using 8 cores in total

```bash
#!/bin/bash
#SBATCH -J hybrid_test
#SBATCH -o hybrid_test.out
#SBATCH -e hybrid_test.err
#SBATCH -p shared
#SBATCH -n 2
#SBATCH -c 4
#SBATCH --ntasks-per-node=1
#SBATCH --mem-per-cpu=4G
export OMP_NUM_THREADS=4
srun -n 2 -c 4 --mpi=pmix ./hybrid_test.x
```

Example 2: 4 Nodes with 1 MPI task per node and 32 OpenMP threads per MPI task, using 128 cores in total (across 4 nodes)

```bash
#!/bin/bash
#SBATCH -J hybrid_test
#SBATCH -o hybrid_test.out
#SBATCH -e hybrid_test.err
#SBATCH -p shared
#SBATCH -n 4
#SBATCH -c 32
#SBATCH --ntasks-per-node=1
#SBATCH --mem-per-cpu=128G
export OMP_NUM_THREADS=32
srun -n 4 -c 32 --mpi=pmix ./hybrid_test.x
```
Is it even worth parallelizing my code?

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

Parallelizing established code vs. starting from scratch

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

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

- Balance parallel workload.

- Minimize time spent in communication.

- Use simple arrays instead of user defined derived types.

- Partition data. Distribute arrays and matrices – allocate specific memory for each MPI process.

- For I/O intensive applications implement parallel I/O in conjunction with a high-performance parallel filesystem, e.g., Lustre.
Thank you! Questions? Comments?

Plamen Krastev, PhD
Harvard - FAS Research Computing