CPSC 424/524 Fall 2023

Assignment 3: Parallel Matrix Multiplication

Due Date: Tuesday, November 14, 2023 (11:59 p.m.)

In this assignment, you will run programs using an implementation of MPI called OpenMPI. To use OpenMPI, you compile and/or link programs using special command wrappers (e.g., **`mpicc`** for the C language; **`mpifort`** for either Fortran77 or Fortran9x). Then, you use the **`mpirun`** command (or **`mpiexec`**, which is really the same thing) to execute the MPI program you've built.

Part 1: Understanding Some MPI Basics

Set up the OpenMPI environment

You must load the proper module file to use OpenMPI with the Intel compiler:

```
module load iomkl
```

This will load both OpenMPI (version 4.0.5) and the Intel compiler suite (version 19.1.3).

Now use the "**`module list`**" command to list out all the loaded module files, and you should see something like:

```
[ahs3@grace2 a3]$ module list

Currently Loaded Modules:
  1) StdEnv                              (S)
  2) GCCcore/10.2.0
  3) zlib/1.2.11-GCCcore-10.2.0
  4) binutils/2.35-GCCcore-10.2.0
  5) iccifort/2020.4.304
  6) numactl/2.0.13-GCCcore-10.2.0
  7) XZ/5.2.5-GCCcore-10.2.0
  8) libxml2/2.9.10-GCCcore-10.2.0
  9) libpciaccess/0.16-GCCcore-10.2.0
 10) hwloc/2.2.0-GCCcore-10.2.0
 11) UCX/1.9.0-GCCcore-10.2.0
 12) libfabric/1.11.0-GCCcore-10.2.0
 13) OpenMPI/4.0.5-iccifort-2020.4.304
 14) iompi/2020b
 15) imkl/2020.4.304-iompi-2020b
 16) iomkl/2020b

  Where:
   S:  Module is Sticky, requires --force to unload or purge
```

To check that the right environment is loaded, run the following commands:

```
which mpicc
which mpirun
mpicc --version
```

The output ought to be something like:

```
[ahs3@c01n02 ~]$ which mpicc
/vast/palmer/apps/avx2/software/OpenMPI/4.0.5-iccifort-2020.4.304/bin/mpicc
[ahs3@c01n02 ~]$ which mpirun
/vast/palmer/apps/avx2/software/OpenMPI/4.0.5-iccifort-2020.4.304/bin/mpirun
[ahs3@c01n02 ~]$ mpicc --version
icc (ICC) 19.1.3.304 20200925
Copyright (C) 1985-2020 Intel Corporation.  All rights reserved.
```

Task 1: Build and Run a Simple "Hello World" Program (5 Points)

`/gpfs/gibbs/project/cpsc424/shared/assignments/assignment3/part1` contains several files for you to use for this task:

```
Makefile   runtask1.sh   rwork.o   task1.c
```

If you wish, you may use the same timing routines as in prior assignments, though MPI also has its own timing function that may be more convenient (see below). The original routines are in:

```
/gpfs/gibbs/project/cpsc424/shared/utils/timing
```

Before building and running the program, let's go over it to see what it does. Looking at the `task1.c` file, you will see that it contains a number of calls to MPI functions (those starting with "`MPI_`") mixed in among ordinary C statements not involving MPI.

Two of the MPI calls are required ***exactly once in every MPI process of every MPI program***:

> `MPI_Init`: Initializes the MPI system and sets up the `MPI_COMM_WORLD` communicator that contains all the MPI processes in the program. No other MPI commands may be executed before `MPI_Init` is called.
>
> `MPI_Finalize`: Terminates the usage of the MPI system (*but not your program*). No other MPI commands may be executed after `MPI_Finalize` is called.

You should keep in mind that, in most cases, each of the cpus allocated to your job by the Slurm scheduler will run the same program. So, one thing done by `MPI_Init` is to assign a unique logical numerical process number or "rank" (starting with 0) to the program instance (MPI process) running on each cpu. The ranks can be used to distinguish one MPI process from another. By convention, the process with rank 0 is often considered to be a "manager" process, while the others are considered to be worker processes that operate under the control of the manager. On some systems, only the rank-0 process has access to the `stdin` input stream.

Once `MPI_Init` has been called, it makes sense for each process to ask "How many processes are there in `MPI_COMM_WORLD`?" and "What is my rank in `MPI_COMM_WORLD`?" These questions are answered by the calls to `MPI_Comm_size` and `MPI_Comm_rank`, respectively, that occur right after the call to `MPI_Init`.

Once the preliminaries are out of the way, program execution splits into two parts: the part done by the manager process (rank = 0), and the part done by the worker processes (rank > 0). Each MPI process determines which part to execute based on its assigned rank. In theory, each worker could execute completely different code from every other worker and the manager, but this is rarely done in practice.

The manager uses the `MPI_Send` function to send two messages to each of the workers. The first contains the string "`Hello, from process 0.`", while the second contains the single integer variable `sparm`. The workers receive the messages and then call the `rwork()` function to simulate some computational work. (Actually, `rwork()` simply sleeps for a certain amount of time that varies from worker to worker.) Once a worker has completed its "work," it prints a message to `stdout` using the `printf()` function. In addition, the manager also times the program by making calls to the `timing()` function (just as in earlier assignments). (As an alternative, you may want to consider using an MPI-provided timing function, `MPI_Wtime()` described below. I've included some commented-out calls to that function, in case you wish to try it out. Either way is fine for the assignment.)

Briefly, the MPI functions just discussed do the following:

> **MPI_Comm_size**: Returns the number of processes in **MPI_COMM_WORLD** (in the **size** argument).
>
> **MPI_Comm_rank**: Returns the calling process's rank in **MPI_COMM_WORLD** (in the **rank** argument).
>
> **MPI_Wtime**: Returns a double precision value equal to the elapsed time in seconds since some arbitrary time point in the past. As with the **timing()** function, the returned value is not useful in itself, but the difference between two such values measures the time between the corresponding calls to **MPI_Wtime()**.
>
> **MPI_Send**: Sends a data buffer (first argument) of a specified number of items (second argument) of a specified data type (third argument) from the calling process to another specified process (fourth argument). The destination process is specified by providing its rank within a specified communicator (sixth argument: **MPI_COMM_WORLD** in this case). The fifth argument is a message tag that we will ignore for now. (Note: The actual buffer length in bytes must be at least the product of the number of items times the length of the specified data type, possibly with some padding. When sending a C character string, remember that the invisible string termination character counts as an extra character.)
>
> **MPI_Recv**: Waits for a message and eventually receives data into a buffer (first argument) of up to some maximum number of items (second argument) of a specified data type (third argument) from a specified source process (fourth argument). As above, the source process is specified by giving its rank within a specified communicator (sixth argument), or by using the special MPI flag **MPI_ANY_SOURCE** to indicate that a message will be accepted from any process in the communicator. Once again, the fifth argument is a message tag, and we will ignore it for now except to note that the tag value must match the tag in the incoming message. (It's treated as a constant in **task1.c**.) The final argument **status** is an output argument that provides information about the message that is received (including the source process, the message tag, and the message length). In C, **status** is a structure of type **MPI_Status**; the element **status.MPI_SOURCE** is the rank of the sending process, and **status.MPI_TAG** is the tag associated with the message. Other entries in status may be used to determine the number of items actually received using the function **MPI_Get_count**. **Note that it is the programmer's responsibility to ensure that the actual receive buffer provided is large enough for the message received.**

In addition to the calls to MPI library functions, you may notice the use of several MPI data types and defined constants (e.g., **MPI_Status**, **MPI_COMM_WORLD**, **MPI_CHAR**, and **MPI_INT**). These are defined in the **mpi.h** include file and provide a level of abstraction to help with consistency across machines, operating systems, and programming languages. Corresponding include files exist for other languages. ***You must include a suitable MPI include file (or use an appropriate mod file for Fortran) in order for your MPI program to work correctly.***

Compilation

I've provided you with a Makefile, so, if you're running on a compute node (e.g., an OOD Remote Desktop), you could build the program by running: **make task1**. (Don't forget to load the proper module file first!).

Notes: The version of **mpicc** you're using is just a wrapper around **icc**. It invokes the **icc** compiler to compile the program, link in the libraries, and create an executable named **task1**. All the flags that can be used with **icc** can also be used with **mpicc**. The reason that you'll usually want to use **mpicc** instead of **icc** for MPI programs is that **mpicc** causes **icc** to become "MPI aware" so that MPI-related include files and libraries can be used without any special action on your part. Recall that the module file sets up the proper version of OpenMPI for the compiler, so if you were using, say, the Gnu compiler, then **mpicc** would be a wrapper around **gcc**, rather than **icc**. (You'd need to use a different module file for OpenMPI with the Gnu compilers.)

Build & Execute Task 1

To build and run the Task 1 program, you need to submit a job to the Slurm scheduler, and I've provided a Slurm script (**runtask1.sh**) for this purpose. You would submit the script file I've provided by running:

```
sbatch runtask1.sh
```

Here are a few items to note about the sbatch script file:

1. The script loads the proper module file, builds the program, and runs it three times. It allocates 4 cores (2 on each of 2 nodes) by requesting 4 tasks, specifying 1 cpu per task, and specifying 2 tasks per node.
2. In addition to the timing functions used in the program itself, the script uses the "**time**" operating system command to report additional timing data. That actual execution statement is:

   ```
   time mpirun –n 4 task1
   ```

   This is a two-part command: the (optional) command word "**time**" tells the system to provide overall job timing data at the end of the job's output file. The remainder of the command line invokes OpenMPI's **mpirun** command to run your program using 4 MPI processes (one per core) that will run on the node resources provided by Slurm.
3. Because of the two sbatch directives specifying --job-name and --output, the contents of **stdout** and **stderr** will be interleaved in a single file named something like "**MPI_Hello-17258884.out**" where the first part is the job name from the script file, and the number is the Slurm job number.

Once your job has finished, list out the contents of the file, for example by using the **less** command. You should see something like:

```
Currently Loaded Modules:
  1) StdEnv                          (S)    9) libpciaccess/0.16-GCCcore-10.2.0
  2) GCCcore/10.2.0                        10) hwloc/2.2.0-GCCcore-10.2.0
  3) zlib/1.2.11-GCCcore-10.2.0            11) UCX/1.9.0-GCCcore-10.2.0
  4) binutils/2.35-GCCcore-10.2.0          12) libfabric/1.11.0-GCCcore-10.2.0
  5) iccifort/2020.4.304                   13) OpenMPI/4.0.5-iccifort-2020.4.304
  6) numactl/2.0.13-GCCcore-10.2.0         14) iompi/2020b
  7) XZ/5.2.5-GCCcore-10.2.0               15) imkl/2020.4.304-iompi-2020b
  8) libxml2/2.9.10-GCCcore-10.2.0         16) iomkl/2020b

  Where:
   S:  Module is Sticky, requires --force to unload or purge




Working Directory:
/gpfs/gibbs/project/cpsc424/shared/assignments/assignment3/part1


Making task1
rm -f task1 task1.o
mpicc -g -O3 -xHost -fno-alias -std=c99 -
I/gpfs/gibbs/project/cpsc424/shared/utils/timing -c task1.c
mpicc -o task1 -g -O3 -xHost -fno-alias -std=c99
-I/gpfs/gibbs/project/cpsc424/shared/utils/timing task1.o
/gpfs/gibbs/project/cpsc424/shared/utils/timing/timing.o rwork.o


Node List:
r918u05n[01-02]
ntasks-per-node =  2
```

```
Run 1
Message printed by manager: Total elapsed time is 0.006429 seconds.
From process 3: I worked for 5 seconds after receiving the following message:
         Hello, from process 0.
From process 2: I worked for 10 seconds after receiving the following message:
         Hello, from process 0.
From process 1: I worked for 15 seconds after receiving the following message:
         Hello, from process 0.

real    0m17.443s
user    0m0.163s
sys     0m0.289s


Run 2
Message printed by manager: Total elapsed time is 0.016885 seconds.
From process 2: I worked for 5 seconds after receiving the following message:
         Hello, from process 0.
From process 1: I worked for 10 seconds after receiving the following message:
         Hello, from process 0.
From process 3: I worked for 15 seconds after receiving the following message:
         Hello, from process 0.

real    0m15.605s
user    0m0.183s
sys     0m0.335s


Run 3
Message printed by manager: Total elapsed time is 0.007963 seconds.
From process 1: I worked for 5 seconds after receiving the following message:
         Hello, from process 0.
From process 3: I worked for 10 seconds after receiving the following message:
         Hello, from process 0.
From process 2: I worked for 15 seconds after receiving the following message:
         Hello, from process 0.

real    0m15.567s
user    0m0.214s
sys     0m0.362s
```

The order of the worker printouts will probably differ in your file since **`rwork()`** randomizes which workers do 5, 10, and 15 seconds of work. The four MPI processes are independent, and the print statements are processed (and appear) in first-come-first-served order. (That's because they're all written to the same stream by independent processes, and this program doesn't contain any logic to enforce a particular order.) As you know, this sort of non-deterministic situation is one type of race condition, and it is usually desirable to avoid race conditions to ensure correct operation and consistency.

The initial part of the output lists the loaded module files, the name of the current working directory, output from the make commands, and information about what nodes were used for the job. The remainder of the output contains the output from three separate invocations of the program. Each invocation produces several different time printouts. The first one (printed by the manager) is the elapsed time between the two **`timing()`** calls. The next three are printed by the workers and indicate how long they worked. Finally, the last three are the result of the **`time`** command in your script and report different portions of the time used by the **`mpirun`** command. Most important for us is the "**`real`**" time, which is the elapsed wall-clock time for **`mpirun`**. The other times represent the cpu time used by your code (the "**`user`**" time) and the cpu time used by the operating system (the "**`sys`**" time).

Thought Questions for Task 1

Here are some questions about Task 1 for you to consider. (You don't need to address them in your report.)

1. Why is the elapsed time reported by the manager in `task1.c` so different from what is reported as "`real`" time by the time command? What is the manager actually measuring? Is it a useful measurement? If not, think about why not, and what might you do about it.
2. Could there be any competition for resources (e.g., hardware, software, or data) between your MPI program, the MPI runtime system, and the operating system for this program? If so, which resources?

Task 2: Modification to Avoid Worker Printouts (5 points)

It's not really a good idea for every worker process to use `printf()`. (Think about why, but don't bother to discuss in your report.) So, modify `task1.c` to create a new program `task2.c` in which only the manager uses `printf()`, and the printouts are always ordered by increasing worker number. After a worker receives the manager's message, it should call `rwork()` as before, but then return a message to the manager containing its rank and the time worked as reported by `rwork()`. For concreteness, please use the following message format:

```
Hello manager, from process <r> after working <worktime> seconds.
```

where "`<r>`" is replaced by the worker's rank, and "`<worktime>`" is replaced with the time worked.

For its part, the manager should receive the messages in a loop that receives messages from the workers in order of increasing rank. After receiving each message, the manager should sleep for 3 seconds (using the function call "`sleep(3)`") to simulate the time it might have spent in postprocessing the data it received. After sleeping, the manager should use `printf()` to print out a line of the form:

```
Message from process <r>: <message>
```

replacing "`<r>`" with the rank of the worker that sent the message, and "`<message>`" with the actual message received. The `printf()` command will be somewhat similar to the workers' `printf()` in `task1.c`.

Notes:

1. You will need to modify the Makefile and create a script file `runtask2.sh` to build and run the new program. (As before, please run the program 3 times in your new script.)
2. The workers can fill the message buffer with the text to return to the manager using `sprintf()`:

```
sprintf(message, "Hello manager, from process %d after working %d seconds.",
        rank, worktime);
```

Thought Questions for Task 2

Here are some questions about Task 2 for you to consider. (You don't need to address them in your report.)

1. How does the elapsed time reported by the manager in this task compare to the time reported as "`real`" time by the time command? Is the comparison qualitatively different from Task 1?
2. You'll probably see some differences in the total elapsed time among the three runs you made for Task 2. Why does this happen? Can you find a way to fix it. A good solution for Task 2 should show little variation in total elapsed time, regardless of the way that work is assigned to the workers.

Task 3: Correcting Performance Issues (5 points)

Modify `task2.c` to create a new program `task3.c` that cures the performance problem identified in Question 2 for Task 2. The only constraint on your program is that the printouts must be in order by increasing worker number. Run the new program three times to validate that it works as expected. This task will require adding to your Makefile and creating a new Slurm script `runtask3.sh` that builds the new program and runs it three times.

Part 2: Parallel Matrix Multiplication

The Problem. Given two $N \times N$ random double precision matrices A and B, develop an MPI program to compute the $N \times N$ matrix C that is the product of A and B.

Mathematically, you are computing the $N \times N$ matrix C whose entries are defined by:

$$c_{ij} = \sum_{k=1}^{N} a_{ik} b_{kj}, 1 \leq i, j \leq N \qquad (1)$$

Task 4: Serial Matrix Multiplication Program (5 points)

Two source files (`serial.c` and `matmul.c`) have been provided for a serial program that computes C using equation (1). These files are in the directory:

`/gpfs/gibbs/project/cpsc424/shared/assignments/assignment3/part2`

In addition, to check your results for accuracy, you will need the data files in the following directory:

`/gpfs/gibbs/project/cpsc424/shared/assignments/assignment3/data`

You may use any of the files in those directories, but the `.dat` files are very large, so please do not make copies of them. When you need them, just read them from where they are stored already, as illustrated in `serial.c`.

Run the serial program on randomly-generated input matrices using $N = 1000$, 2000, 4000 and 8000. The `serial.c` program contains appropriate code for this, using a seed setting so that the random matrices are reproducible. Please use that code for this purpose throughout this assignment to facilitate checking the numerical results. Along with the source files, I have provided binary result files containing the correct C matrices, and `serial.c` computes the Frobenius norm of the error in the computed C, which should be very close to zero. All your codes should illustrate performance and correctness by printing a table similar to the one below printed by `serial.c`. For timing, you may use the `timing()` functions from earlier assignments; see the `Makefile`. Alternatively, you may find it easier to use the standard MPI timing routine `MPI_WTime()`. Below is a sample output table from `serial.c`. (Your timings and error norm may not match mine exactly.) See the `Makefile` and the file `build-run-serial.sh` for a list of relevant `#SBATCH` settings for Slurm, required module files, and how to build the serial program.

```
Matrix multiplication times:
    N       TIME (secs)     F-norm of Error
  -----    -------------   -----------------
   1000        0.3203       0.000000000032
   2000        3.0214       0.000000000134
   4000       38.8284       0.000000000504
   8000      327.0215       0.000000001930
```

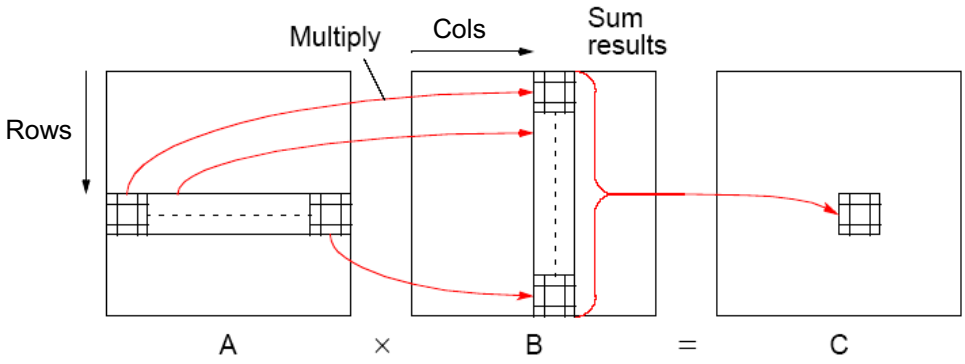
NOTES:

(a) Each job run for this assignment should need no more than ~30 minutes. Please limit your walltime requests so that everyone has fair access to the class reservation. Please debug using $N = 1000$ before trying larger cases. My own experience suggests that you may see variation in timings, so make several runs and report average timings.

(b) As you develop parallel codes, ensure that your programs produce the correct answers by having the manager compute and print error norms as in `serial.c`. (However, **DO NOT** include this work in your reported timings.)

Tasks 5-8: Parallel Programs

Parallelization Approach: Here's the approach you'll use for this assignment: Divide the input matrices into blocks of rows (for A) or columns (for B). Then a computational task would be to compute the matrix product of a block row of A and a block column of B to form a small rectangular block of C. Pictorially, this looks like:



To implement this using p MPI processes (including the manager, which should do real work in addition to managing the entire computation in this case), you will use the following "ring-pass" approach:

1. The manager process (rank 0) creates space for the matrices and initializes A and B as in `serial.c`.
2. The manager then partitions A and C into p block rows each, and it partitions B into p block columns. The block rows are of dimension $(N/p)\times N$, while the block columns are of dimension $N\times(N/p)$.
3. Next, the manager permanently assigns to each MPI process (including the manager) one block row each of A and C, and it assigns each MPI process one block column of B as its initial assignment. The manager must use MPI collective operations to distribute the assigned block rows or columns of A and B to each MPI process. (Each MPI process will need to allocate memory space for its blocks of A, B, and C.)
4. The computation then proceeds iteratively as follows:
    a. Each MPI process does all the computation it can with the data it has in hand.
    b. MPI processes then pass their block columns of B to the next higher-ranked MPI process. (MPI process p-1 passes to MPI process 0.)
    c. Repeat until done with the calculation.
5. The manager then uses MPI collective operations to assemble the full C matrix by collecting all the block rows of C from the other MPI processes.
6. Finally, the manager computes the error norm and prints a table similar to the one printed by `serial.c`.

Notes: You will use various MPI functions in this assignment, but you may not use `MPI_Sendrecv()`. Except for Task 8, you may assume that *N* is a multiple of *p*.

Task 5: Blocking MPI Parallel Program (30 points)

Implement an MPI program that computes the product of two square matrices using equation (1) and the ring-pass parallelization approach described above. **For Task 5, you must use MPI's blocking communication operations (including blocking collectives).** Your program should use exactly p block rows for A and C, and p block columns for B, where p is the total number of MPI processes (including the manager). **For Task 5, each block row or block column should contain *N/p* consecutive rows or columns.** Run your program on randomly-generated matrices using $N = 1000$, 2000, 4000 and 8000, and $p = 2$, 4, and 8. I suggest batch runs that each use a fixed value of p for all four matrix sizes (analogous to the serial runs in Task 4). In your report, please discuss the performance and scalability of your program.

Task 6: Non-Blocking MPI Parallel Program (30 points)

One way to improve performance might be to overlap computation and communication, effectively hiding communication and synchronization time behind the computations. To lay the groundwork for investigating this, start by simply modifying your Task-5 program to replace the blocking MPI communication operations (including collective operations) with non-blocking equivalents wherever possible. (This probably won't improve performance much, but it allows you to separate the data transfers in communication operations from the synchronizations associated with checking on completion.) **For Task 6, where possible, you must use non-blocking collective operations instead of the blocking collectives used in Task 5.** For information about the non-blocking collective operations, see the man pages for the collectives you use or consult `open-mpi.org`. Once you have a working program that uses non-blocking operations, repeat the runs and discussions/analyses from Task 5. (In terms of program changes, simply replacing blocking operations with non-blocking ones is all that is required for Task 6, so the performance discussion may well be quite similar to Task 5.)

Task 7: Performance Improvements (20 points)

To improve the performance significantly, you probably need to modify your Task-6 code to overlap the computation and communication. One approach to this would be to move around some of the MPI non-blocking calls to allow for overlap. Another idea might be to increase the number of block columns (by making them smaller) so that you could compute on some while communicating on others.

For Task 7 you are required to describe, implement and assess at least one approach to increasing performance by increasing the overlap between computation and communication. You do not need to do extensive performance testing, but you should describe what you did (and why it seemed promising) and demonstrate that your implementation produces correct answers (even if it doesn't actually improve the performance). In addition, you should collect timing data for $N = 8000$ and $p = 8$ that allows you to assess the change in performance due to the modifications you made. Note that some potential improvements might require introducing *a limited number* of additional communication buffers, and that would be perfectly fine.

Task 8: Generalization (Extra Credit: 10 points)

Modify your Task-5 program so that it works for arbitrary N and p. **For this task only, N is not presumed to be a multiple of p.** Demonstrate that your code works by running it for $N = 7633$ and $p = 7$ on 4 nodes (the manager alone on one node, with two MPI processes each on the other 3 nodes). Note that I have provided a correct answer for $N = 7633$, in the same directory as the other correct-answer files.

Procedures for Programming Assignments

For this class, we will use the Yale Canvas website to submit solutions to programming assignments.

Remember: While you may discuss the assignment with the instructor, a ULA or your classmates, the work you turn in must be yours alone!

What should you include in your solution package?

1. **All source code files, Makefiles, and scripts that you developed or modified.** All source code files should contain proper attributions, as needed. In addition, make sure that your source code contains suitable documentation (e.g., comments) so that someone looking at your code can understand what it's trying to do. In part, your grade will depend on the documentation, code organization, and comments for your code.
2. **A report in PDF format** containing:
   a. Your name, the assignment number, and course name/number.
   b. Information on building and running the code:
      i. A brief description of the software/development environment used. For example, you should list the module files you've loaded.
      ii. Steps/commands used to compile, link, and run the submitted code. Best is to use a Slurm script that you submit using `sbatch`. The script could use a Makefile to build the code and would then run it multiple times so that you can report average timings. (If you ran your code interactively, then your report will need to list the commands required to run it.)
      iii. Outputs from executing your program(s).
   c. Any other information required for the assignment (e.g., answers to questions that may be asked in the assignment).

How should you submit your solution?

1. On the cluster, create a directory named "`netid_ps3_cpsc424`". (For me, that would be "`ahs3_ps3_cpsc424`".) Put into it all the files you need to submit, including your report.
2. Create a compressed tar file of your directory by running the following in its parent directory:

   ```
   tar -cvzf netid_ps3_cpsc424.tar.gz netid_ps3_cpsc424
   ```

3. To submit your solution, click on the "Assignments" button on the Canvas website and select this assignment from the list. Then click on the "Submit Assignment" button and upload your solution file `netid_ps3_cpsc424.tar.gz`. (We will set Canvas to accept only files with a "`gz`" or "`tgz`" extension.) You may add additional comments to your submission, but your report, including the plot, should be included in the attachment. You can use OOD, tools like scp or rsync, or various GUI tools (e.g., CyberDuck) to move files back and forth to Grace.

Due Date and Late Policy

Due Date: **Tuesday, November 14, 2023 by 11:59 p.m.**

Late Policy: On time submission: Full credit

Up to 24 hours late: 85% credit

Up to 72 hours late: 50% credit

Up to 1 week late: 25% credit

More than 1 week late: No credit

General Statement on Collaboration

Unless instructed otherwise, all submitted assignments must be your own individual work. Neither copied work nor work done in groups will be permitted or accepted without prior permission.

However….

You may discuss questions about assignments and course material with anyone. You may always consult with the instructor or ULAs on any course-related matter. You may seek general advice and debugging assistance from fellow students, the instructor, ULAs, Canvas ED discussions, and Internet sites, including https://ask.cyberinfrastructure.org/c/yale/40 or similar sites.

However, except when instructed otherwise, the work you submit (e.g., source code, reports, etc.) must be entirely your own. If you do benefit substantially from outside assistance, then you must acknowledge the source and nature of the assistance. (In rare instances, your grade for the work may be adjusted appropriately.) It is acceptable and encouraged to use outside resources to *inform* your approach to a problem, but *plagiarism is unacceptable* in any form and under any circumstances. If discovered, plagiarism will lead to adverse consequences for all involved.

DO NOT UNDER ANY CIRCUMSTANCES COPY ANOTHER PERSON'S CODE—to do so is a clear violation of ethical/academic standards that, when discovered, will be referred to the appropriate university authorities for disciplinary action. Modifying code to conceal copying only compounds the offense.