CPSC 424/524 Spring 2020
Assignment 3

Due Date: Monday, March 23, 2020, 9:00am

Total Value: 30 points

In this assignment, you will run programs using an implementation of MPI called OpenMPI. This assignment doesn't require much programming, but it will give you some experience running MPI programs. For more about MPI, you may wish to consult our textbook (chapters 9-11), pp. 1-42 of *Using MPI,* and Chapter 3 of the Pachecho book. (The latter two are in the resource list in Lecture #1.) For information about OpenMPI specifically, use man pages, or see http://www.open-mpi.org.

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 **`mpiexec`** command (or **`mpirun`**, which is really the same thing) to execute the MPI program you've built.

Set up the OpenMPI environment

To use OpenMPI, you need to load the proper OpenMPI module file:

```
module load OpenMPI/3.1.1-iccifort-2018.3.222-GCC-7.3.0-2.30
```

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/7.3.0
  3) binutils/2.30-GCCcore-7.3.0
  4) icc/2018.3.222-GCC-7.3.0-2.30
  5) ifort/2018.3.222-GCC-7.3.0-2.30
  6) iccifort/2018.3.222-GCC-7.3.0-2.30
  7) zlib/1.2.11-GCCcore-7.3.0
  8) numactl/2.0.11-GCCcore-7.3.0
  9) XZ/5.2.4-GCCcore-7.3.0
 10) libxml2/2.9.8-GCCcore-7.3.0
 11) libpciaccess/0.14-GCCcore-7.3.0
 12) hwloc/1.11.10-GCCcore-7.3.0
 13) OpenMPI/3.1.1-iccifort-2018.3.222-GCC-7.3.0-2.30

  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 mpiexec
mpicc --version
```

The output ought to be something like:

```
[ahs3@grace2 a3]$ which mpicc
/gpfs/loomis/apps/avx/software/OpenMPI/3.1.1-iccifort-2018.3.222-GCC-7.3.0-
2.30/bin/mpicc
[ahs3@grace2 a3]$ which mpiexec
/gpfs/loomis/apps/avx/software/OpenMPI/3.1.1-iccifort-2018.3.222-GCC-7.3.0-
2.30/bin/mpiexec
[ahs3@grace2 a3]$ mpicc --version
icc (ICC) 18.0.3 20180410
Copyright (C) 1985-2018 Intel Corporation.  All rights reserved.
```

Task 1: Execute a Simple "Hello World" Program (5 Points)

`/home/cpsc424_ahs3/assignments/assignment3` contains several files for you to use for this assignment:

```
Makefile
runtask.sh
task1.c
timing.h
timing.o
rwork.o.
```

(I've included the same timing routines that are in the class `utils` directory.)

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 usually considered to be a "master" process, while the others are considered to be worker processes that operate under the control of the master. On some systems, only the process with rank 0 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 master 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 master, but this is rarely done in practice.

The master 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 master 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.)

Briefly, the MPI functions just described 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 9x) 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, you could build the program by running: **make task1**. (Don't forget to load the OpenMPI module file first!). However, for this assignment, I've provided a Slurm script that builds and executes the program, so you won't have to build it separately.

Notes: Just a reminder: Never build programs on a cluster login node, since the proper libraries may not be installed there. Use an interactive session instead. The version of **mpicc** you're using invokes the **icc** compiler to compile the program, link in the libraries, and create an executable named **task1**. Since **mpicc** is just a wrapper around **icc**, 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 OpenMPI module file sets up the proper version of OpenMPI for your compiler, so if you were using, say, the Gnu compiler, then **mpicc** would be a wrapper around **gcc**, rather than **icc**.

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 (**runtask.sh**) for this purpose. You can use the script file I've provided by running:

```
sbatch runtask.sh
```

Here are a couple of items to note about the 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 mpiexec -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 **mpiexec** 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 combined in a single file named something like "**MPI_Hello-50590118.out**" where the first part is the job name specified in the script file, and the number is the job number.

Now list out the contents of the file, for example by using the **less** command. If your program ran correctly, you should see something like:

```
Currently Loaded Modules:
  1) StdEnv                                                (S)
  2) GCCcore/7.3.0
  3) binutils/2.30-GCCcore-7.3.0
  4) icc/2018.3.222-GCC-7.3.0-2.30
  5) ifort/2018.3.222-GCC-7.3.0-2.30
  6) iccifort/2018.3.222-GCC-7.3.0-2.30
  7) zlib/1.2.11-GCCcore-7.3.0
  8) numactl/2.0.11-GCCcore-7.3.0
  9) XZ/5.2.4-GCCcore-7.3.0
 10) libxml2/2.9.8-GCCcore-7.3.0
 11) libpciaccess/0.14-GCCcore-7.3.0
 12) hwloc/1.11.10-GCCcore-7.3.0
 13) OpenMPI/3.1.1-iccifort-2018.3.222-GCC-7.3.0-2.30

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

Working Directory:
/home/cpsc424_ahs3/assignments/assignment3

Making task1
rm -f task1 task1.o task2 task2.o task3 task3.o
mpicc -g -O3 -xHost -fno-alias -std=c99 -c task1.c
mpicc -o task1 -g -O3 -xHost -fno-alias -std=c99 task1.o timing.o rwork.o
```

```
Node List:
c04n[05,07]
ntasks-per-node =  2


Run 1
Message printed by master: Total elapsed time is 0.004646 seconds.
From process 1: 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 3: I worked for 15 seconds after receiving the following message:
         Hello, from process 0.

real    0m17.009s
user    0m0.750s
sys     0m0.645s


Run 2
Message printed by master: Total elapsed time is 0.005001 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    0m15.567s
user    0m0.665s
sys     0m0.722s


Run 3
Message printed by master: Total elapsed time is 0.001630 seconds.
From process 3: 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 2: I worked for 15 seconds after receiving the following message:
         Hello, from process 0.

real    0m15.580s
user    0m0.644s
sys     0m0.490s
```

The order of the worker printouts may be different 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, 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 almost always desirable to avoid race conditions in order to ensure correct operation and have consistency from run to run.

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 master) 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 `mpiexec` command. Most important for us is the "`real`" time, which is the elapsed wall-clock time for `mpiexec`. 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).

Some questions for Task 1

Here are some questions about the program in Task 1 that you should address in your report:

1. Why is the elapsed time reported by the master in `task1.c` so different from what is reported as "`real`" time by the time command? What is the master 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? (Hint: Think about the resources used by MPI and for I/O.)

Task 2: Modification to Avoid Worker Printouts (10 points)

It's not really a good idea for every worker process to use `printf()`. (Understand 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 master uses `printf()` and the printouts are always ordered by increasing worker number. After a worker receives the master's message, it should call `rwork()` as before, but then return a message to the master containing its rank and the time worked as reported by `rwork()`. For concreteness, please use the following message format:

```
Hello master, 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 master should receive the messages in a loop that receives messages from the workers in order of increasing rank. After receiving each message, the master 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 master 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 similar to the workers' `printf()` in `task1.c`.

Notes:

1. You will need to modify the Makefile and add lines to `runtask.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 master using `sprintf()`:

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

Questions for Task 2

Here are some questions about the program in Task 2 that you should address in your report:

1. How does the elapsed time reported by the master in this task compare to the time reported as "`real`" time by the time command? Is the comparison qualitatively different from Task 1? Explain.
2. You'll probably see some differences in the total elapsed time among the three runs you made for Task 2. Explain why this happens and describe (in words) how to fix it. A good solution should show little variation in total elapsed time, regardless of the way that work is assigned to the workers.

Task 3: Correcting Performance Issues (15 points)

Modify `task2.c` to create a new program `task3.c` that cures the performance problem identified in Question 2 for Task 2, while still keeping the printouts 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 extending the Slurm script so that it builds all the programs and runs all three tasks (three times each).

Procedures for Programming Assignments

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

Remember: While you may discuss the assignment with me, a ULA, or your classmates, the source code you turn in must be yours alone and should not represent collaborations or the ideas of others!

What should you include in your solution package?

1. **All source code files, one (1) Makefile, and the final runtask.sh script that you developed, used, or modified.** All source code files should contain proper attributions and suitable comments to explain your code.
2. For this assignment, your package should also include:
   a. A single `Makefile` that can build all of the programs you use in the assignment, using a different target for each program. (Note: The `Makefile` for Task 1 is a good starting point.)
   b. One Slurm script (`runtask.sh`) that builds the three programs and runs them three times each.
3. **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. Essentially, this is a description of your Slurm script.
      iii. Outputs from executing your program. Since the final Slurm script runs all three of the task programs, you only need to provide that final output.

c. Any other information required for the assignment, including answers to the questions you were asked for Task 1 and Task 2.

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.
2. Create a compressed tar file of your directory by running the following in its parent directory:

   ```
   tar -cvzf NetID_ps2_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`. You may add additional comments to your submission, but your report should be included in the attachment. You can use scp or rsync or various GUI tools to move files back and forth to Grace.

Due Date and Late Policy

Due Date: **Monday, March 23, 2020 by 9:00 a.m.**

Late Policy: On time submission: Full credit

Up to 24 hours late: 90% credit

Up to 72 hours late: 75% credit

Up to 1 week late: 50% credit

More than 1 week late: 35% 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, Piazza conversations, and Internet 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.