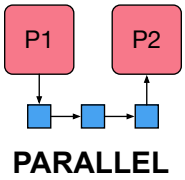

CISC 372: Parallel Computing

University of Delaware, Spring 2022

HW 3

This assignment is due Monday, March 7, at noon. Topics: beginning MPI, embarrassingly parallel algorithms, performance measurement, point-to-point communication, using a supercomputer

1. Problem 1: `rand5`

Create a subdirectory `rand5` of `hw03`. Write an MPI program `rand5.c` in which each process generates 5 random `int`s. (Use C's `rand()` and `srand()` functions.) Each process other than process 0 sends its 5 ints to process 0. Process 0 prints all the ints (including its own), then prints the componentwise sums. The output should look something like this:

```
Received from proc 0:     4     100     87     45     4
Received from proc 1:     0     455     34   2222    45
Received from proc 2:     5       3    444    223    45
========================================================
Sums:                     9     558    565   2490    94
```

Include a `Makefile` with rules for `rand5.exec`, `test` and `clean`. Test it on `cisc372.cis.udel.edu` using `srun`.

2. Problem 2: Deadlock?

Create a subdirectory `deadlock` of `hw03`. For each of the following code snippets, determine which of the following is true:

(A) The code cannot deadlock.
(B) The code may or may not deadlock.
(C) The code must deadlock.

Assume that MPI, `nprocs`, and `rank` have all been initialized properly, and that the program is run with at least 2 processes (`nprocs` ≥ 2).

(1)

```
int x = 0, y;
MPI_Send(&x, 1, MPI_INT, (rank+1)%nprocs, 0, MPI_COMM_WORLD);
MPI_Recv(&y, 1, MPI_INT, (rank+nprocs-1)%nprocs, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

(2)

```
int x = 0, y;
```

```
if (rank == 0) {
  MPI_Send(&x, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
  MPI_Recv(&y, 1, MPI_INT, 1, 1, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
} else if (rank == 1) {
  MPI_Send(&x, 1, MPI_INT, 0, 1, MPI_COMM_WORLD);
  MPI_Recv(&y, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}
```

(3)

```
int x = 0, y;
if (rank == 0) {
  MPI_Recv(&y, 1, MPI_INT, 1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
  MPI_Send(&x, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
} else if (rank == 1) {
  MPI_Recv(&y, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
  MPI_Send(&x, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
}
```

(4)

```
int x = 0, y;
if (rank < nprocs-1)
  MPI_Send(&x, 1, MPI_INT, rank+1, 0, MPI_COMM_WORLD);
if (rank > 0)
  MPI_Recv(&y, 1, MPI_INT, rank-1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

Put your answers in a plain text file named `deadlock.txt`. The file should have the following format:

```
(1) X
Brief explanation.

(2) X
Brief explanation.

(3) X
Brief explanation.

(4) X
Brief explanation.
```

Where each `X` is one of A, B, or C. The brief explanation is just a couple of sentences in your own words explaining your analysis.

3. Problem 3: You are perfect

A *perfect number* is a positive integer n with the property that the sum of the factors of n (including 1, but excluding n itself) equals n. For example, 6 is perfect because

$$6 = 1 + 2 + 3.$$

On the other hand, 8 is not perfect because

$$8 \neq 1 + 2 + 4.$$

The program `372-2022S/code/src/seq/perfect/perfect.c` consumes an integer n on the command line, and finds all perfect numbers less that or equal to n, reporting the total number of perfect numbers at the end. (If you have any questions about this program, ask using one of the usual channels.) Make sure you can run this sequential program on `cisc372.cis.udel.edu`.

Write a parallel version of the program using MPI. The output should be essentially the same as the sequential version; the only exception is that the perfect numbers might be found in a different order. Your program should report the total number of perfect numbers found and the time, exactly as the sequential version does. Call your program `perfect_mpi.c` and put it in directory `hw/hw03/perfect` in your private repository. Include a `Makefile` with the usual rules.

Measure the performance of your MPI program on `cisc372.cis.udel.edu`, by running it on $n = 10^7$ with the following numbers of processes: 1, 5, 10, 15, 20, 25, 30, 35. Record the time (in seconds) for each value of nprocs in a file named `cisc372.dat`. Each line of the file should consist of the value of nprocs followed by a space and then the time. (See `372-2022S/exp/sat_strong/beowulf/sat_mpi.dat` for an example).

Create a gnuplot file `perfect.gnu` for plotting the data. Running gnuplot on this file should create a PDF file named `perfect.pdf`. The x-axis should show the number of processes and the y-axis the time (in seconds).

Can you discern any pattern between the number of processes and the time? Write a brief (1–3 setences) analysis in a plain text file `perfect.txt`.

4. Using the Bridges-2 Supercomputer

4.1. **Configuring your Bridges workspace.** You should have already created an XSEDE account and told us your XSEDE user name (HW 1). If you haven't done that, do it now, and contact us immediately. As soon as we know your user name, we can add you to our Bridges account.

Now you must create your PSC (Pittsburgh Supercomputing Center) account, if you have not already done so. Direct your web browser to `https://apr.psc.edu`. Your PSC user name is the same as your XSEDE user name, but the passwords are different. The web form says you are resetting your PSC password, but you are actually creating it for the first time. Choose a password, record it somewhere, and follow the instructions on the form.

Now you should be able to ssh to Bridges-2. From a Terminal, type

```
ssh USER@bridges2.psc.xsede.org
```

where `USER` is your XSEDE/PSC user name. Enter your PSC password when prompted.

Once you are logged on, type

```
/ocean/projects/see200002p/shared/bin/configbash
```

This updates your `.bash_profile` and `.bashrc` files in your home directory to work for this class.

Log out and log back on to bridges2. Type `372queue`. If things are configured correctly, this command will show you the current job queue for the class (which may be empty). If you see a message like "Command not found", something is wrong, contact us immediately.

From your home directory (which is where you will be if you just logged in; but you can always get back to it with `cd ~`), check out your two repositories:

```
svn checkout svn://vsl.cis.udel.edu/cisc372/372-2022S
svn checkout --username USER svn://vsl.cis.udel.edu/cisc372/372-USER
```

where `USER` is your user name. Change into `372-2022S/code/src` and type `make`. All code should compile without warnings or errors. If that doesn't work, get help.

Once you have completed the steps in this section, you should never have to do them again.

4.2. **Running programs on bridges2.** Reference: `https://www.psc.edu/resources/bridges-2/user-guide-2/`, section *Batch jobs*.

Bridges also uses SLURM, but the protocol for executing programs is slightly different than that of `cisc372.cis.udel.edu`. You need to create a simple *batch script* specifying what to run. An example can be found here:

```
cd ~/372-2022S/code/src/mpi/pi
cat bridges2/pi_rect_4.sh
```

You should see:

```
#!/bin/bash
#SBATCH -p RM-shared
#SBATCH -t 00:05:00
#SBATCH -N 1
#SBATCH --ntasks-per-node 4
set -x
mpirun -np $SLURM_NTASKS ./pi_rect_mpi.exec
```

Here is what these lines mean:

- `#!/bin/bash` — use `bash` to execute this script. This will always be the same.
- `#SBATCH -p RM-shared` — use Bridges' "regular memory shared" partition — not the "regular memory" (`RM`) partition. See below.
- `#SBATCH -t 00:05:00` — put a time limit of 5 minutes on this job. The format is hh:mm:ss (hours, minutes, seconds). It's a really good idea to put a reasonable time limit on every job because you are charged for every second your program runs, and if you accidentally get into an infinite loop, ...

- `#SBATCH -N 1` — use one node. Not necessarily the full node.
- `#SBATCH --ntasks-per-node 4` — create 4 MPI processes, each on its own core. Hence we are using 4 cores on one node.
- `set -x` — echo all commands executed to stdout.
- `mpirun -np $SLURM_NTASKS ./pi_rect_mpi.exec` — execute the specified MPI program. The number of processes is determined by the arguments above: the product of the number of nodes and the number of tasks per node—in this case, 4.

Some explanation about the partitions. Each Bridges2 node has 128 cores. The shared nodes allow multiple jobs to run simultaneously on the node, using different cores. For example, my job above will use 4 cores, and may run at the same time as someone else's job using 2 cores, and another using 16 cores, and so on, as long as the total number of cores is at most 128. The advantage of using a shared node is that you are charged only for the cores you use: if you use 4 cores for 1 hour, you are charged exactly 4 core-hours.

The other option is the "regular memory" partition, `#SBATCH -p RM`. For these nodes, you are charged for the whole node (all 128 cores), even if you use only 1 core. Usually, you only use the `RM` partition if the number of processes (cores) is a multiple of 128. For example, if you want 256 cores, you would use `-p RM`, `-N 2`, and `--ntasks-per-node 128`.

My suggestion is to start by using small core counts on `RM-shared`, and only switch to `RM` with 1 full node (128 cores) once you have worked out all the kinks. Then move on to `RM` with 2 nodes, and scale up from there.

Submit a job to the queue using `sbatch`:

```
sbatch bridges2/pi_rect_4.sh
```

SLURM returns to you a job ID, and you can continue using the shell. The output from your job will go to a file with a name of the form `slurm-`*`jobid`*`.out`. You can continue doing other work, or even log out, without affecting your queued jobs.

To check on the status of the queued jobs from CISC372 students only, type `372queue`. You will see a table with one row per job. The columns provide various information about the job. See `https://slurm.schedmd.com/squeue.html` for a detailed description of all the information displayed here.

The column labeled `ST` is the *state* of the job. Common values are:

- `PD` — Pending. Job is awaiting resource allocation.
- `CF` — Configuring. Job is allocated and starting up.
- `R` — Running.
- `CG` — Completing. Job is wrapping up.
- `CA` — Canceled.
- `F` — Failed.

Wait for the `pi_rect_4` to complete and examine the output file to make sure it executed correctly. Once you have done this, you are ready to proceed to the next section.

5. Problem 4: Run perfect on Bridges

Working on Bridges, move into the directory containing your parallel perfect number program `perfect_mpi.c`. You have already tested this program on `cisc372.cis.udel.edu`, so now it is ready for the supercomputer.

To start, type `make` and make sure the program compiles with no errors or warnings.

Now make a subdirectory of `perfect` named `bridges2`. Within `bridges2`, create batch files to run the MPI program on $n = 10^7$ with the following process counts: 1, 2, 4, 8, 16, 32, 64, 128. For all but 128, use the RM-shared partition. (Exception: if you find long wait times, you may use the RM partition.) For 128, use RM. In all cases, the number of nodes is 1 (`-N 1`), and the time limit should be at most 5 minutes. Name these batch files `perfect_10_1.sh`, `perfect_10_2.sh`, `perfect_10_4.sh`, etc., and commit them to version control.

Now submit the batch files. If you are confident they are good, you can submit all at once. Log out and/or do something else for a while, come back, and check up on them. If something went wrong, you can keep re-submitting them (and ask for help).

Once all jobs have completed successfully, move the successful SLURM output files into the `bridges2` directory, renaming them `perfect_10_1.out`, `perfect_10_2.out`, `perfect_10_4.out`, etc., and commit them to version control.

Now try to run the program with $n = 5 * 10^7$ (50 million) using 128 cores. Did you find any new perfect numbers? Name your batch file `perfect_50_128.sh` and the output `perfect_50_128.out`, put them in the `bridges2` directory, and commit to version control.

Back in directory `perfect`: create `bridges.dat`, and `perfect_bridges.gnu` and use these to generate `perfect_bridges.pdf`, a plot of the times for the Bridges-2 executions. Add a new paragraph to `perfect.txt` analyzing your Bridges-2 data. Finally, add a paragraph:

```
The greatest perfect number found for n=5*10^7: ......
```

replacing the dots with the greatest perfect number you found.

To summarize, you should have committed all of the following in `hw03/perfect`:

```
perfect_mpi.c
Makefile
perfect.txt
cisc372.dat
perfect.gnu
bridges.dat
perfect_bridges.gnu
bridges/
  perfect_10_1.sh
  perfect_10_1.out
  perfect_10_2.sh
  perfect_10_2.out
  perfect_10_4.sh
```

```
perfect_10_4.out
perfect_10_8.sh
perfect_10_8.out
perfect_10_16.sh
perfect_10_16.out
perfect_10_32.sh
perfect_10_32.out
perfect_10_64.sh
perfect_10_64.out
perfect_10_128.sh
perfect_10_128.out
perfect_50_128.sh
perfect_50_128.out
```