Intro to High-Performance Computing May 2, 2017
University of California, Irvine EECS 221

Homework/Lab 2: Distributed Memory/MPI

Due: 11:55 P.M. on Sunday, May 21, 2017

- Running MPI jobs on the HPC cluster: `http://hpc.oit.uci.edu/running-jobs#_mpi_using_two_or_more_nodes`
- OpenMPI documentation: `http://www.open-mpi.org/doc/v1.8`

Snailspeed Ltd. is actively trying to enter new markets. Your boss is starting a new project with the goal of developing Mandelbrot sets.

With the revolution in parallel computing, and since Snailspeed owns stock in the graphics company, your manager, Peter Sloanie, wants you to develop fractals that are parallelized and run super fast, so he can impress his competitors. Therefore, the code should run as fast as possible while making the most use of the hardware's capabilities.

As added incentive to write good code, when you are done, the project is going to be passed to your neighbor, Bob Halfbit. Bob is the sort of coworker who leaves the coffee pot empty, occasionally eats someone else's lunch, and listens to Michael Bolton just loud enough that you can't not be distracted by it, but not loud enough for you to say something. If he doesn't grok your code, you're going to be paired with him for the next year and a half to work on it. For the love of Pete (your manager), make your code readable and well-documented so you can continue to work on interesting projects.

The lab has two parts. In the first part, you will run a "ping-pong" micro-benchmark and use it to come up with a cost model for communication on the class cluster. You will not need to do any programming in this part. In part 2, you will implement a distributed algorithm for Mandelbrot computation. This part is an *offline* part that you will do at home.

You may work in **teams of two**. To simply grading of your assignments, submit only one copy of the assignment per team. Be sure to indicate your assignment partner by creating a `README` file as part of your submission.

Part 1 (in class): Ping-pong

The demo implements a ping-pong microbenchmark. The benchmark uses two nodes, which we will call Node 0 and Node 1. First, Node 0 sends a message to Node 1. Then, as soon as Node 1 receives the message, it immediately returns the message to Node 0. To get accurate timing data, the benchmark actually measures the time to complete 1000 ping-pong volleys, and then divides by 1000 to get the average time of a single volley.

Now lets download, compile, and run a demo. We will use the results of this demo to analyze the performance of the network on the HPC cluster.

Getting the scaffolding code

Use a ssh client and your UCInetID login/password to log into the cluster. Since we want to display the ping-pong benchmark output and final Mandelbrot fractal, we will enable X11 forwarding when logging into the cluster.

```
$ ssh -X <UCInetID>@hpc.oit.uci.edu
```

The baseline code for this assignment is available at this URL: `https://github.com/EECS221/hw2.git`

To get a local copy of the repository for your work, you need to use git to clone it. So, let's make a copy on the HPC cluster and modify it there. To do that, run the following command on the cluster.

```
$ git clone https://github.com/EECS221/hw2.git
```

There will be a new directory called **hw2**.

Compiling and running your code

To compile the benchmark, we will use `mpicc`, which is a MPI-aware wrapper script around the default GNU compiler, `gcc`. The following commands will turn `pingpong.c` into a binary executable named `pingpong`:

```
$  mpicc -o pingpong pingpong.c
```

As `mpicc` is just a wrapper around `gcc`, it accepts the usual compiler command-line flags for optimization, preprocessing, and linking.

Running the benchmark: Batch jobs

The HPC cluster is a shared computer. When you login to `hpc.oit.uci.edu`, you are using the login node. You should limit your use of the login node to light tasks, such as file editing, compiling, and small test runs of, say, just a few seconds. When you are ready to do a timing or *performance* run, then you submit a job request to a grid engine (GE) scheduler. There are two steps:

1. Create a batch job script, which tells GE what machine resources you want and how to run your program.

2. Submit this job script to GE, using a command called `qsub`.

A batch job script is a shell script file containing two parts: (i) the commands needed to run your program; and (ii) metadata describing your jobs resource needs, which appear in the script as comments at the top of the script. We have provided an example for the `pingpong` benchmark; this script is `pingpong.sh` in the `part1` subdirectory.

This example asks the scheduler for (a) two nodes, (b) includes the MPI command needed to launch the `pingpong` program, and (c) includes some additional post-processing commands (e.g., the call to `gnuplot`).

To submit this job request, use the qsub command:

```
$ qsub pingpong.sh
```

If successful, this command will register your job request and return a job identification number (job ID). Your job is first entered into a central queue, and depending on the current cluster load, might not run right away. As such, you can check on the status of your job requests by running:

```
$ qstat -u <UCInetID>
```

When your job eventually runs, any output to standard output or standard error (e.g., as produced print statements) will be automatically redirected into output files (with .o### and .e### files, labeled by the job ID ###).

Analyzing the results

If all went well above, you should see two new output files: results.dat and netplot.png.

- The results.dat file contains the raw benchmark results; inspect this now. The first column of this tab-delimited file is the message length in bytes; the second column is the average measured one-way time (in seconds) required to send the message.
- The netplot.png file is a plot of the above data, with message size on the x-axis and time on the y-axis. To view this file, run the following command. Note that the axes use a $\log_{10}$ scale.

  ```
  $ display netplot.png
  ```

Assuming you were able to complete the steps above, here is what you need to turn in. (Each member of the team should do their own submission but may submit identical work. Just be sure to say with whom you worked.) Submit your results.dat and netplot.png files. In addition, answer the following questions:

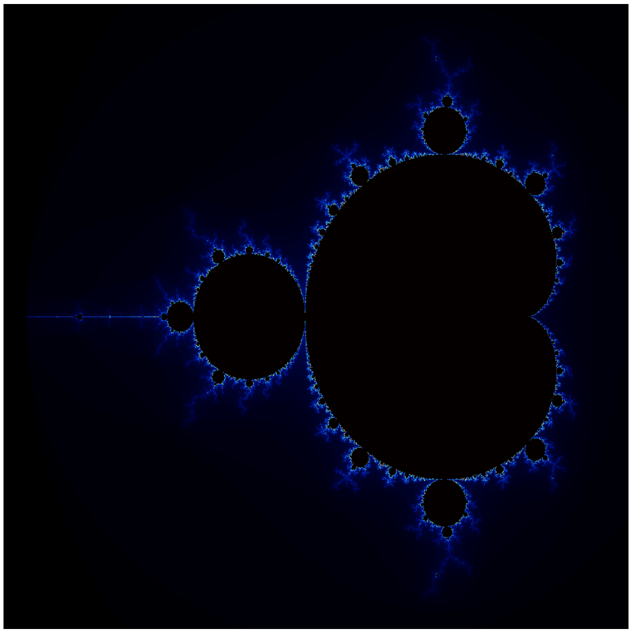
- Take a look at netplot.png. What is the minimum time to send a message of any size?
- When the message size is large, estimate the slope of the line. What does the slope represent in this case?
- Using your answers to the above questions, come up with a model (i.e., formula) that would allow someone to estimate $T(m)$, the time to send a message of size m bytes.
- Take a look at the function pingpong() in pingpong.c; for your convenience, the code for this function appears below. Go line-by-line and do your best to explain what the code is doing. Make a note of anything you don't understand as well.

```
void pingpong (int* msgbuf, const int len)
{
 const int MSG_PING = 1;
 const int MSG_PONG = 2;

 int rank;
 MPI_Comm_rank (MPI_COMM_WORLD, &rank);

 if (rank == 0) {
   MPI_Status stat;
   MPI_Send (msgbuf, len, MPI_INT, 1, MSG_PING, MPI_COMM_WORLD);
   MPI_Recv (msgbuf, len, MPI_INT, 1, MSG_PONG, MPI_COMM_WORLD, &stat);
 } else {
   MPI_Status stat;
   MPI_Recv (msgbuf, len, MPI_INT, 0, MSG_PING, MPI_COMM_WORLD, &stat);
   MPI_Send (msgbuf, len, MPI_INT, 0, MSG_PONG, MPI_COMM_WORLD);
 }
}
```

Part 2 (after class): Mandelbrot set



The Mandelbrot set is a famous example of a fractal in mathematics. It is named after mathematician Benoit Mandelbrot. It is important for chaos theory.

The Mandelbrot set can be explained with the equation

$$z_{n+1} = z_n^2 + c$$

where c and z are complex numbers and n is zero or a positive integer (natural number).

Since we are calculating an infinite series, we need to approximate the result by cutting off the calculation at some point. If a point is inside the mandelbrot set, we can keep iterating infinitely. To avoid this, we set a limit on the maximum iterations we will ever do. In the code given to you, the maximum number of iterations is set to 511.

The Mandelbrot set above is embarrassingly parallel since computation of a pixel of the image does not depend on any other pixel. Hence, we can distribute the work across MPI processes in many different ways. In this assignment, you will conceptually think about load balancing this computation and implement a general strategy that works for a wide rage of problems.

Compiling and running your code

We have provided a small program, broken up into modules (separate C++ files and headers), that performs Mandelbrot set sequentially and renders the image. We have also provided a `Makefile` for compiling your program. To use it, just run `make`.

Run `mandelbrot_serial` on an input image size of `1000 x 1000` as follows:

```
$ ./mandelbrot_serial 1000 1000
```

To visualize the fractal, enter the following command:

```
$ display mandelbrot.png
```

If your network connection is slow, this may take a couple of minutes. Be patient.

Running batch jobs on the cluster

We have provided a sample job script, `mandelbrot.sh`, for running the serial Mandelbrot program you just compiled on a `1000 x 1000` image on the compute node of the cluster.

Go ahead and try this by entering the following commands:

```
$ qsub mandelbrot.sh
$ qstat -u <UCInetID>
```

Note that since the scaffolding code given to you is serial and runs on one core. When you finish parallelizing your code, to run with multiple MPI processes, follow the instructions at `http://hpc.oit.uci.edu/running-jobs#_mpi_using_two_or_more_nodes` very carefully.

Load balancing strategies

Since your boss is so eager to see results, you hire two interns to help you with the parallelization. Intern Susie Cyclic implements the above computation with P MPI processes. Her strategy is to make process p compute all of the (valid) rows $p + nP$ for $n = 0, 1, 2, \ldots$ and use MPI gather operation to collect all the values at the root process to render the fractal.

Intern Joe Block implements the above computation with P MPI processes as well. His strategy is to make process p compute all of the (valid) rows $pN, pN + 1, pN + 2, \ldots pN + (N - 1)$ where $N = \lfloor height/P \rfloor$ and then use MPI gather to collect all of the values at the root process for rendering the fractal.

Which do you think is better? Why? Which intern do you offer a full-time job?
HINT: The Mandelbrot function can require anywhere from 0 to 511 iterations per row.

Parallelization

Implement Susie and Joe's approaches in the respective files called `mandelbrot_susie.cc` and `mandelbrot_joe.cc`. Analyze the speedup and efficiency of the two approaches against the provided serial version. Use atleast upto 256 processes and more if you wish and an image size of your choice. Submit the plots and a discussion of your results.

In general, you may not know beforehand how best to distribute the tasks. In the worst case, you could get a list of jobs that makes any specific distribution the worst possible. Another option when the number of jobs is much greater than the number of processes is to let each process request a job whenever it has finished the previous one it was given. This is the master/slave model. The master is responsible for giving each process a unit of work, receiving the result from any slave that completes its job, and sending slaves new units of work. A slave process is responsible for receiving a unit of work, completing the unit of work, sending the result back to the master, and repeating until there is no work left. Implement the Mandelbrot image computation using a master/slave MPI strategy in `mandelbrot_ms.cc` where a job is defined as computing a row of the image. Communicate as little as possible. Compare the master/slave strategy with Susie/Joe's implementation. Which do you think will scale to very large image sizes? Why?

Submission

When youve written up answers to all of the above questions, turn in your write-up and tarball of your code by uploading it to eee.uci.edu dropbox.

Good luck, and have fun!