Homework2: Distributed Memory/MPI

Due: 5 P.M. on Monday, May 18, 2020

- Running MPI jobs on the HPC cluster:
  `https://hpc.oit.uci.edu/running-jobs#_mpi_using_two_or_more_nodes`

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 music 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.

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.

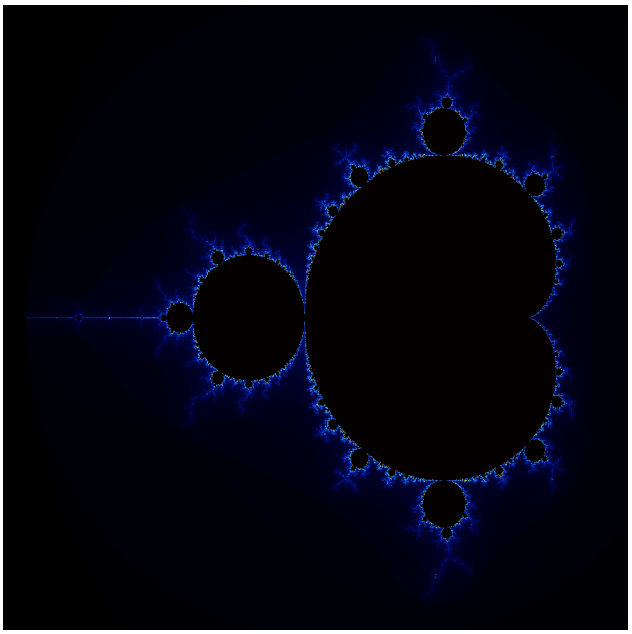
The Mandelbrot set can be explained with the equation

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

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.



Getting the scaffolding code

Use a ssh client and your UCInetID login/password to log into the cluster. Since we want to display the 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/UCI-HPC/EECS224-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/UCI-HPC/EECS224-hw2.git
```

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

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 $n = 0, 1, 2, ...$ 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 $pN, pN + 1, pN + 2, ...pN + (N - 1)$ 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 the maximum processes on the class queues (i.e., one MPI process per core) and more on the public queues if you wish and an image size of your choice. Submit the plots and a discussion of your results.

HINT: Try different image sizes and observe the scaling behavior of the 2 parallelization approaches.

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 you've written up answers to all of the above questions, turn in your write-up and zip/-tarball of your code by uploading it to Canvas.

Good luck, and have fun!