Lab 10: Message Passing Interface (MPI)

Instructor: Mackale Joyner, Co-Instructor: Zoran Budimlić

Course Wiki: http://comp322.rice.edu

Staff Email: comp322-staff@mailman.rice.edu

Goals for this lab

- Install VirtualBox and a Ubuntu virtual machine with MPI support on your laptop.
- Use MPI to distribute computation across multiple processes.
- Understand the parallelization of matrix-matrix multiply across multiple, separate address spaces.
- Complete an MPI implementation of matrix-matrix multiplication by filling in the correct communication calls.

1 Overview

In this lab you will use OpenMPI's Java APIs to gain experience with distributed computing using MPI. You will complete a dense matrix-matrix multiply implementation by filling in the missing MPI API calls in a partial MPI program.

2 Preliminaries

In order to avoid congestion on NOTS and to help you prepare for HW5 (which also uses MPI), we will be using a virtual machine installed locally on your laptop to write, test and run MPI programs.

First, you will need to download and install the latest version of VirtualBox on your laptop. You can download the version for your platform directly from VirtualBox:

- `https://www.virtualbox.org/`

Once you have downloaded and installed VirtualBox, you will need to download the virtual appliance from here:

- `https://rice.box.com/s/aqe89uz6f0f5opvloxzik5df36qmcngp`

This should download a file `COMP322-MPI-S19.ova` locally on your computer. Open the .ova file in VirtualBox and accept all the default settings.

The .ova file is a virtual machine that has Ubuntu installed, along with Java 8, IntelliJ and MPI, all the tools you will need for Lab 10 and HW5.

Once the virtual machine has started, you should have a desktop with IntelliJ link on the taskbar on the left. Click on that, and it should start the IntelliJ environment that should be very familiar to you by now. Obviously, you won't have any of your previous homewoks or labs here (since it's running in a brand new virtual machine), but that's not an issue. You should be able to check out and start working on your Lab 10 as described below.

Lab Projects

The template code and Maven project for this lab are located at:

- `https://svn.rice.edu/r/comp322/turnin/S19/`*`NETID`*`/lab_10`

Please use the subversion command-line client or IntelliJ to checkout the project into appropriate directories locally. For example, you can use the following command from a shell:

```
$ svn checkout https://svn.rice.edu/r/comp322/turnin/S19/NETID/lab_10 lab_10
```

Do all your development and testing within your local virtual machine. You will have to run the script

- `run.sh`

This script is in your project directory. It launches the MPI runtime within the virtual machine and runs the tests. For this lab, you will not be able to run the test cases directly from within IntelliJ, since IntelliJ cannot create multiple MPI processes needed for the MPI execution.

3 Matrix Multiply using MPI

Your assignment today is to fill in incomplete MPI calls in a matrix multiply example that uses MPI to distribute computation and data across multiple processes. You should complete all the necessary MPI calls in `MatrixMult.java` to make it work correctly. There are comments (TODOs numbered 1 to 14) in the code that will help you with modifying these MPI calls. You can look at the slides for Lectures 28 and 29 for an overview of the MPI send() and recv() calls, and at https://fossies.org/dox/openmpi-2.1.0/namespacempi.html for the API details.

The provided parallel matrix-matrix multiply example works as follows:

1. The master process (`MPI.COMM_WORLD.getRank() == 0`) gets the size of the matrices to be multiplied and the number of processes to use from the unit tests.
2. Each MPI process allocates its own input matrices (`a`, `b`) and output matrix (`c`). Note that this code uses a *flattened* representation for matrices, i.e., a square matrix of size $N \times N$ is stored as a one-dimensional array containing N^2 elements.
3. The master process initializes its local copies of each matrix and transmits their contents to all other MPI processes. At the same time the master process also assigns each process a set of matrix rows which that process is responsible for processing.
4. Each MPI process computes the contents of its assigned rows in the final output matrix `c`.
5. The master process collects the results of each worker process back to a single node and shuts down.

4 Tips

- There are only two provided unit tests. One runs a small experiment and prints the input and output matrices to help with debugging. The other processes larger matrices and will be used to verify the performance and correctness of your implementation.

- Note that all MPI send and recv APIs (e.g. https://fossies.org/dox/openmpi-2.1.0/classmpi_1_1Comm.html#a7e913a77ef4b8b1975035792cde6d717) accept **arrays** as their `buf` argument. Even when sending a single integer, you will need to box it as a singleton array. Passing scalars to the buf argument is by far the most common error made on this lab.

- When running tests, the performance of your code on 1, 2, 4, and 8 MPI processes will be printed. It is safe to ignore any error warnings that begin with "Java HotSpot(TM) 64-Bit Server VM warning". Look for text similar to "Processing a 1024 x 1024 matrix with 1 MPI processes" followed a few lines later by a print starting with "Time elapsed = ". There will be sections that look like this for each number of MPI processes, each of which include the time it took for your solution to run with that many MPI processes.

5 Deliverables

Once you have completed the template MPI program and tested it locally within the virtual machine, submit myjob.slurm to NOTS. You'll need to commit your changes you made in your virtual machine, then log into NOTS and checkout your project from SVN, then submit the Slurm job. The teaching staff will want to see some performance improvement from 1 to 2 to 4 to 8 processes, before checking off a successful completion of your lab.