Lab 14: Message Passing Interface (MPI)

Instructor: Vivek Sarkar

Course wiki : https://wiki.rice.edu/confluence/display/PARPROG/COMP322

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

Importants tips and links

edX site : https://edge.edx.org/courses/RiceX/COMP322/1T2014R

Piazza site : https://piazza.com/rice/spring2015/comp322/home

Acknowledgements

This lab uses the software package mpiJava developed at Indiana University (http://www.hpjava.org/mpiJava.html).

Goals for this lab

- 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 the mpiJava package 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.

Lab Projects

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

- `https://svn.rice.edu/r/comp322/turnin/S15/`*`NETID`*`/lab_14`

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

```
$ svn checkout https://svn.rice.edu/r/comp322/turnin/S15/NETID/lab_14 lab_14
```

Once you have checked out the `lab_14` turnin folder, please complete the `myjob.slurm` file it contains by filling in the full path to the checked out folder on STIC at line 18.

For this lab, you will only be able to compile and test your code on STIC. Local compilation and execution is not supported as this lab depends on compiled third-party binaries and a complex development environment. This also more realistically represents working in a HPC environment.

This lab will use four different Maven projects, each of which executes an identical, parallel problem but with different amounts of parallelism:

1. `1process` executes a parallel matrix-matrix multiply on a single process.
2. `2process` executes a parallel matrix-matrix multiply on two processes.
3. `4process` executes a parallel matrix-matrix multiply on four processes.
4. `8process` executes a parallel matrix-matrix multiply on eight processes.

The provided `myjob.slurm` file executes each of these profiles.

2 Matrix Multiply using MPI-Java

Your assignment today is to fill in incomplete MPI calls in a matrix multiply example that uses mpiJava 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 33 and 34 for an overview of the mpiJava Send() and Recv() calls, and at http://www.hpjava.org/mpiJava/doc/api for the API details (click on the "Comm" link).

Though MPI is designed for execution on distributed-memory machines, today we will simply be creating multiple MPI Processes within a single STIC node. Thus, today you will be experimenting with multi-process parallelism (rather than the multi-threaded parallelism you have been using in most of the other labs).

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

1. The master process (`MPI.COMM_WORLD.Rank() == 0`) parses the command-line arguments to determine the size of the matrices to be multiplied and broadcasts the size to all other MPI processes.
2. Each MPI process allocates its own input matrices (`a`, `b`) and output matrix (`c`).
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.

3 Tip(s)

- Start with a small data size (e.g. `matrix.size` = 10) and flip the variable `MatrixMult.printOption` to true to visualize the matrices you are generating. This can be helpful in debugging issues in the implemented communication pattern.
- Set the flag `MatrixMult.validateResult` to true to verify that your parallel matrix multiply implementation is correct. This flag compares the results of the parallel computation against a sequential matrix-matrix multiply.

4 Deliverables

Once you have completed the template MPI program by filling in the inter-process communication, run myjob.slurm on a STIC node with `matrix.size` in the pom.xml file set to 1000. How does performance change as the number of processes is increased from 1 to 2 to 4 to 8? Is speedup linear? Explain why or why not.