# Lab 10: Message Passing Interface (MPI)

Instructor: Vivek Sarkar, Co-Instructor: Mackale Joyner

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

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

## 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 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.

## Lab Projects

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

- `https://svn.rice.edu/r/comp322/turnin/S17/`*`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/S17/NETID/lab_10 lab_10
```

If you plan to submit manually rather than through the autograder, you should also check out the project on NOTS and complete the provided `myjob.slurm` file based on the contained TODOs.

*For this lab, you will only be able to test your code on NOTS. It likely will not run locally.* Local execution is not supported as this lab depends on compiled third-party binaries and a complex development environment that is only available on NOTS. However, if you choose, you will still be able to *compile locally* on your laptop (to fix compile-time errors) if you import the project dependencies from the provided pom.xml file into IntelliJ. See step 6 at:

https://wiki.rice.edu/confluence/display/PARPROG/Using+IntelliJ+to+Download+and+Run+lab_1

if you need to recall how to import dependencies from Maven into IntelliJ.

## 2 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.

# 3 Tip(s)

- 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.
- If you run through the autograder, you should ignore all errors related to the running of correctness tests (because there are none for this lab).
- 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 through the autograder, the performance of your code on 1, 2, 4, and 8 MPI processes will be printed in the pane titled "Performance Tests (16 cores)". It is safe to ignore any error warnings that begin with "Java HotSpot(TM) 64-Bit Server VM warning". As you scroll in that pane, 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.

# 4 Deliverables

Once you have completed the template MPI program by filling in the inter-process communication, submit myjob.slurm to NOTS or submit your code to the autograder for a final run. 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.