Université Grenoble Alpes, Grenoble INP, UFR IM^{2}AG

Master 1 Informatique and Master 1 MOSIG

UE Parallel Algorithms and Programming

Lab # 3

2021

1 About this lab

- This lab is an introduction to MPI. The proposed exercises cover the main concepts introduced during the lecture, and introduce additional ones.
- This lab is not graded
- There is no report to submit at the end of the lab
- All the concepts introduced during this lab will be re-used during the graded lab.

2 About MPI programs

Writing MPI programs. Every MPI program:

- Includes the MPI header file `mpi.h`
- Calls a routine to initialize the MPI environment (`MPI_Init`)
- Calls a routine to terminate the MPI environment (`MPI_Finalize`)

Compilation and execution of a MPI program

- Compilation: `mpicc -o myprogram.run myprogram.c`
- Execution with 4 MPI processes using Open MPI (available on the machines in the lab room): `mpirun -n 4 ./myprogram.run`

Execution platforms The options to execute MPI programs are detailed in the following document: instructions_mpi.pdf

In a few worlds, the possible options are:

- Using the computers from the lab rooms
  - Allows running over multiple machines but required being physically present on site.

- Using the servers of the university
- Using your own machine
- Using Google Cloud

3 Discovering MPI

3.1 Hello World!

You are provided with an `hello_world` program in which each process displays its ID (rank) and the total number of processes.

Observe the code of the provided program, compile and run it.

3.2 Action based on the rank

Modify the `hello_world` program so that processes with an even rank number print a different message from the other processes.

3.3 Distributed execution (optional)

Warning: This exercise can only be run if you have access to multiple machines where SSH connections are allowed between the machines. This exercise can only be done if you are on site. It could also be done using Google Cloud. But, in this case, we suggest you to come back to this exercise when you have done everything else.

Code modifications: Modify the `hello_world` program so that each process gets and displays the hostname of the machine it is running on. Try running this program on multiple hosts. Make the number of processes vary to observe how Open MPI places processes by default.

Running on multiple machines: Assuming that you have access to multiple machines, here are the main steps to run a MPI program over these machines:

- Allow ssh connections between hosts without password
  1. Generate ssh keys without password

     ```
     ssh-keygen -t rsa
     ```

  2. Add the new public key to the list of authorized keys

     ```
     cd ${HOME}/.ssh; cat id_rsa.pub >>authorized_keys;
     ```

  3. Prevent ssh from checking host key at first connection by adding the following lines to `${HOME}/.ssh/config`:

     ```
     Host *
        StrictHostKeyChecking no
     ```

- Create a hostfile containing the hostname (or IP address) of the machines where the application should be executed
  - For more information on the syntax of the `hostfile`, please read `https://www.open-mpi.org/faq/?category=running#mpirun-hostfile`.

- Run your application with a command like:

```
mpirun -np 4 --hostfile my_hosts ./myprogram
```

3.4 Simple communication

Write a program in which process with rank 0 sends an array of 10 64-bit floating point numbers to the process with rank 1. Process 0 will fill in the array and process 1 will print it.

3.5 Ping/pong

Modify the previous program so process 1 sends the modified array back to process 0. Modify the program again to measure round-trip time using `MPI_Wtime()`.

3.6 Communication ring

Write a program in which a token is passed among processes in a ring. At the beginning, process 0 owns the token. Then it is passed from process to process (with modification) until reaching process 0 again.

3.7 Non-blocking communication

Write a program in which two processes want to send a message to each other at the same time (send and receive). Try a version using blocking communication primitives and then correct it using non-blocking communication primitives. Make tests with small and large messages to see whether the behavior changes.

4 Collective operations

4.1 Broadcast

Write a program in which process 0 sends an array of 10 integers to all other processes (without using point-to-point communication).

4.2 Total sum

Write a program which computes the sum of all processes ID and sends the result to all of them.

Write 3 versions of this program:

1. A version with a reduction followed by a broadcast
2. A version using a global reduction (Allreduce).
3. A version where you don't use collective operations but use (Waitall) to implement the reduction (not recommended, just to practice).

4.3 Barrier

Write a program in which processes synchronize using a barrier. To better understand the semantic of a barrier, use the `sleep` function to simulate different execution time before calling the barrier, and observe the total execution time of the application.

4.4 Computing PI

To compute an approximate value of PI, a simple method is to evaluate the integral of $\frac{4}{1+x\times x}$ between 0 and 1. To approximate the value of the integral, a sum of n intervals can be computed: the approximation to the integral in each interval is $\frac{1}{n} \times \frac{4}{1+x\times x}$.

This approach is called *rectangular approximation*. In each rectangle, we suggest you to use the value of x at the left of the interval.

Implement a parallel version of the computation of PI. Rank 0 asks the user for the number of intervals and broadcast this number to all of the other processes. Find a strategy to automatically distribute the work between the MPI processes. Finally, aggregate the local computation of all MPI processes and display the computed value of PI.

5 Cartesian virtual topologies

A virtual topology describes a mapping/ordering of MPI processes into a geometric *shape*. The are two main types of topologies supported by MPI: Cartesian (grid) and Graph. These topologies are built upon MPI communicators and groups.

Being able to create a cartesian topology is especially useful as several parallel algorihtms in message passing systems are designed based on such a topology. Examples of algorithms based on a cartesian topology will be given to you in the next lecture.

(1) Write a program in which you create a virtual grid of processes (use `MPI_Dims_create` to define the dimensions of the grid and `MPI_Cart_create` to create the grid). Print the new and the previous rank of each process and their coordinates in the grid (use `MPI_Cart_coords`). Try different numbers of processes. Figure 1 presents an example of cartesian topology[1].

0 (0,0)	1 (0,1)	2 (0,2)	3 (0,3)
4 (1,0)	5 (1,1)	6 (1,2)	7 (1,3)
8 (2,0)	9 (2,1)	10 (2,2)	11 (2,3)
12 (3,0)	13 (3,1)	14 (3,2)	15 (3,3)

Figure 1: A cartesian topology

(2) Modify your program to create a virtual $q \times q$ grid of processes. Be sure to create a square grid even if the number of MPI ranks is not a square number (the extra processes are simply doing nothing).

(3) Create a communicator per row and per column with the function `MPI_Cart_sub` to group all processes in the same row or in the same column in one communicator.

(4) Modify your program to pass a token from rank to rank in each row and each column. Initially rank 0 in each communicator owns the token.

[1] source: `https://computing.llnl.gov/tutorials/mpi/#Virtual_Topologies`

(5) Modify your program so that each process broadcasts an integer to other processes on the same row.

(6) Make global sums on columns.

(7) Make shifts on rows and columns and print the neighbors of each process in the grid (use `MPI_Cart_shift`). What is happening if a process doesn't have its four neighbors in the grid?

6 Deployment in the Cloud

Try setting up an environment and executing your MPI applications in the Cloud using the tutorial provided here: `https://roparst.gricad-pages.univ-grenoble-alpes.fr/cloud-tutorials/mpi/`