Advanced Parallel Computing - Exercise 4
Submission date: 22/1/2018

General Instructions

This is a programming exercise, to be submitted in pairs. Program using the C language with the MPI library. The compilation process is the same as with standard C code, but rather than using *gcc*, we use the *mpicc* compiler[1]. Communication and synchronization will be done using MPI directives. Do not use *sleep* or similar local mechanisms. As discussed in the lecture, the computational elements that execute in parallel during an MPI run are "processes". However, as is standard convention, we will sometimes refer to them as "ranks". Whenever we refer to the value of a rank, we shall mean the enumeration in the MPI_COMM_WORLD communicator.

Submission Guidelines

Submit your outputs and source code files in a single .tar archive. Please submit a single .tar archive containing the following files:

1. README (including names, usernames and IDs of both students)
2. hello_mpi.c
3. transpose.c
4. grid.c
5. 3d-mult.c

You may define additional source files. If you do so, please submit a Makefile that compiles each program.

Inputs and Outputs

If your code requires input, each MPI rank can randomly generate its own. Make sure that you set a distinct seed value for the random generator function on. Save the randomly generated inputs to file (either a single file for the entire input, or one file per MPI rank) so correctness can be verified. Save your outputs into files with the same format. Do not include the time needed for generating data or saving it to file when performing benchmarks.

Testing the Code

The maximum number of ranks allowed in an execution of an MPI program is *independent* of the physical core-count of the machine. For example, if we have a computer with a single 2-core processor, and we spawn an MPI program with 4 ranks, then we can have 2 MPI ranks assigned to each physical core. This is called *oversubscription*, and naturally has a strong impact on the performance of your code. However, code that runs correctly on a million cores should also run correctly on a single core (with obvious slowdown). You can therefore use any environment that has MPI installed to develop and test your code. In particular, it is recommended to use the Aquarium computers for your development and initial testing. Except where otherwise stated, there is no need to benchmark your execution.

[1] Strictly speaking, *mpicc* is a wrapper built on top of a compiler.

Running Code on a Cluster

Lucy is a cluster composed of 3 nodes, each equipped with 24 physical cores. You will be able to access Lucy using ssh. Each user will be able to send up to 3 jobs in parallel to the job queue. To do this, you will need to use submission scripts: these will be provided for you. Your executions on Lucy will be limited to 10 minutes each (your code should only need a couple of minutes at most). After this, the task will be killed. A detailed explanation will be posted on Moodle once the environment is set up. If you work on an environment other than the Aquarium, you need to make sure you know how to connect remotely to the Aquarium computers. Accessing Lucy is very similar.
You are expected to use Lucy only for the final testing and benchmarking and not for the primary development process.

Question 1 - Hello World

Filename: hello_mpi.c
Input: None
Output: Each MPI rank will print its own rank, and the total number of ranks in the system.
After all ranks are finished printing, the rank 0 processor prints "Hello World!".

Question 2 - Matrix Transpose

Let P be the overall processor count, and suppose the processors are set on a Cartesian grid of size $\sqrt{P} \times \sqrt{P}$, so a process p has coordinates (x, y).
Filename: transpose.c
Input: A matrix A of size $5\sqrt{P} \times 5\sqrt{P}$ containing doubles. Each process $p = (x, y)$ locally stores a block A_{xy} of size 5×5 from A.
Output: A matrix $B = A^{\intercal}$, where each rank $p = (x, y)$ locally stores a block B_{xy} of size 5×5 from B.
If $\sqrt{P}$ is not an integer, the process with rank 0 will print an error message and the program exits.
Using collective communication operations for this question is allowed, but not required.
Test your code on $P = 3^2, 4^2, \ldots, 8^2$. No need to benchmark your code.

Question 3 - Small-Kernel Convolution

In this exercise we compute a convolution of distributed input with the small kernel $\begin{pmatrix} 0 & 1 & 0 \\ 3 & 1 & 4 \\ 0 & 2 & 0 \end{pmatrix}$. That is, we set the processors on a virtual 2-dimensional torus topology of dimension $\sqrt{P} \times \sqrt{P}$. Each process computes a weighted sum of its direct neighbors.
Filename: grid.c
Input: Each process has a local vector v_p containing 5 integers.
Output: Each process p stores and prints the sum of the following vector $v_p + v_U + 2 \cdot v_D + 3 \cdot v_L + 4 \cdot v_R$, where v_U, v_D, v_L, v_R are the vectors of the processes above, below, to the left and to the right of p, respectively and $\bmod \sqrt{P}$.
If $\sqrt{P}$ is not an integer, the process with rank 0 will print an error message and the program will exit.
Note: Avoid *serialization*! If p needs data from p_1 and from p_4, there is no need to wait for the communication with P_1 to successfully complete before the communication with P_4 can be initiated.
Test your code on $P = 3^2, 4^2, \ldots, 8^2$. No need to benchmark your code.

Question 4

In this exercise we implement a classical $3D$ matrix multiplication algorithm. For this question you are asked to benchmark your execution: you can use the function MPI_Wtime() to query a clock.
Filename: 3d-mult.c
Input: Each process p has local matrices of size $k \times k$, denotedA_p and B_p, both containing doubles.
Output: $C = AB$.
If $\sqrt[3]{P}$ is not an integer, the program prints an error and exits.
Hints:

1. It is easier to implement this algorithm using collective communications rather than using point-to-point operations.
2. Define multiple communicators, one for each axis of the cube.

3. The local computation performed by each process is not required to be highly-efficient.

Run your code with $P = 2^3, 3^3, 4^3$.
When working on the Aquarium workstations (or your preferred equivalent), set the dimensions of each matrix so the size of a local block is 5×5.
When working on Lucy, set the dimensions of each complete matrix to be 1440×1440.