High Performance Computing

Homework #6
Part B

Due: Tuesday April 26 2016 by 11:59 PM

Email-based help Cutoff: 5:00 PM on Mon, April 25 2016

Total Maximum Points: 15

Submission Instructions

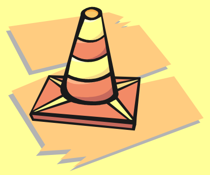
This homework assignment must be turned-in electronically via Niihka. Ensure your C++ source code is named `MUid_Homework6_PartB.cpp`, where *MUid* is your Miami University unique ID. Upload just your source file(s) onto Niihka before the due date.

Objective

The objectives of this homework are:

- Parallelize a given problem using MPI
- Gain further familiarity with MPI's blocking communication functions

Grading Rubric:



This is an advanced course and consequently the expectations in this course are higher. Accordingly, the program submitted for this homework must pass necessary tests in order to qualify for earning a full score.

NOTE: Program that do not compile, have methods longer than 25 lines, or just some skeleton code will be assigned zero score.

Scoring for this assignment will be determined as follows assuming your program compiles (and is not skeleton code):

- **10 points**: Allocated for overall implementation of an effective solution that operates correctly with 'n' MPI processes.
- **5 Points**: For completing the performance report for this homework.
- **-1 Points**: for each warning generated by `g++` when compiling your C++ program with the –Wall option.
- **-1 Point**: For each style violation reported by the CSE department's C++ style checker program `cpplint.py`.

- **NOTE:** Points will be deducted for violating stylistic qualities of the program such as: program follows formatting requirements (spacing, indentation, suitable variable names with appropriate upper/lowercase letters, etc). The program includes suitable comments at appropriate points in each method to elucidate flow of thought/logic in each method. Program strives to appropriately reuse as much code as possible.

Starter Code

You are supplied with a sequential version of a program that you are expected to parallelize. Copy the necessary files from the shared folder on Red Hawk using the following command (don't miss the period at the end):

```
$ cp -r /shared/raodm/csex43/homeworks/homework6 .
```

In this homework, you are supplied with the following file(s):

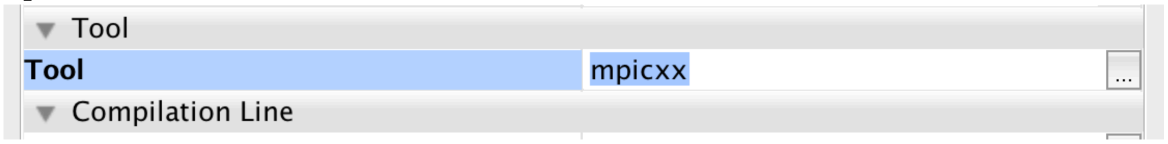
1. `exercise14.cpp`: This file is the same starter code supplied for the lab exercise. This program uses exactly 2 processes to compute the number of factors for a given number.

2. A set of data files `numbers_100.txt`, `numbers_1000.txt`, and `numbers_10000.txt` files to be used for testing and performance verification.

Compiling:

Ensure you appropriately configure your NetBeans project to use `mpicxx` for compiling and linking your program as shown below.

1. In your project properties, select the `C++ Compiler` entry and set the `Tool` to `mpicxx` as shown in the screenshot below:



▼ Tool
Tool: mpicxx
▼ Compilation Line

2. Next select the Linker entry and set the Tool to `mpicxx` as shown in the screenshot above.

Homework: Parallelize the application

Description:

The objective of this homework is to parallelize the prime number verification program discussed in class and introduced via Exercise #10. The serial version of the program (`exercise14.cpp`) is supplied along with this homework for your reference. The objective of this homework is to parallelize the program to use 2 or more processes (recollect that in the exercise only 2 processes were used, one for manager and another one for worker) while adhering to the following requirements (**pay attention to all of these requirements**):

1. Your final solution source code for this homework must be named with the convention `MUid_Homework8_PartB.cpp`.

2. Given *n* parallel processes to use, your program must use one of them to serve as a manager and the remaining *n*-1 processes as workers. The operations to be performed by a worker and a manager are described further below for your immediate reference.

3. The worker processes must use the existing `getFactorCount()` method (from the supplied `exercise14.cpp`) and associated logic to determine if a given number is a prime number.

4. All interactions between worker and manager must use only MPI's blocking communication functions (`MPI_Send` and `MPI_Recv`). Non-blocking communication calls must not be used in this homework exercise.

5. The manager and worker may exchange only one number (`int` or `long long int`) per send or receive call. This feature is already setup in the program and you are expected to retain this setup in your final solution as well.

6. Only the manager process is permitted to perform any I/O (such as: file reading and displaying results on the console) operations. The manager process reads data from the file specified as command-line argument. This feature is already setup in the starter code. You can reuse the `main` method as is.

7. All testing must be performed with the supplied `numbers_100.txt`, `numbers_75.txt`, and `numbers_10000.txt` files.

Required behaviors of Manager and Worker:

The following operations are the minimal set of operations that a Manager and a Worker process are expected to perform. The processes may perform additional tasks in order to streamline the program to extract additional performance.

- The manager process must perform the following tasks:
  - Repeatedly reads numbers (`long long int`) from a text file (name of the file would change depending on the test).
  - For each number read, the manager process:
    - i. Sends the number to one of the worker process using `MPI_Send`
    - ii. Obtains the number of factors from the worker using `MPI_Recv` (possibly as and when responses are available).
    - iii. Displays number of factors on the screen. The order of output of results does not need to be maintained.
    - iv. Finally it sends number -1 to all the workers indicating work is done.

- A worker process must perform the following tasks:
  - Repeatedly reads only a single number from the manager and stops when manager sends -1 as the number. For each number:
    - i. It computes the number of factors for the number using `getFactorCount()`
    - ii. Sends the number of factors back to the manager

Tips & Suggestions:

- Try to keep the workers as busy as possible.
- In order to keep workers busy one possible approach is to:
  - First distribute one number to all the workers.
  - Whenever a worker responds with a result assign another number to the worker. You can use `MPI_ANY_SOURCE` to receive results from any of the worker processes and find the actual worker's rank from status provided by `MPI_Recv`.
- As a general rule of thumb, always compile your code with `–Wall` (report all warnings) flag in the `g++` command line. It will minimize runtime errors in your code and save you a lot of aggravation on the long run.

Sample outputs

See supplied sample output files `numbers_75_output.txt`, `numbers_100_output.txt`, and `numbers_10000_output.txt` files for details.

Testing

You can test results from your program using the following command (where `hw6.o3712612` is output from your PBS job with `numbers_100.txt` as input):

```
$ sort hw6.o3712612 | diff numbers_100_output.txt -
```

Turn-in

Submit your C++ source file (`MUid_Homework8_PartB.cpp`) that meets the requirements of this homework via Niihka. No credit will be given for submitting code that does not compile or is just skeleton code. Verify that your program meets all the requirements as stated in the grading rubric. Ensure your C++ source files are named with the stipulated naming convention. Upload all the necessary C++ source files to onto Niihka. Do not submit zip/7zip/tar/gzip files. Upload each file independently.