Assignment 4 - Introduction to Parallel Programming

Bernat Xandri Zaragoza

1. **Exercise 1**: Sieve of Eratosthenes (2 + 1 + 3 = 6 points in total)
   a) The previous assignments asked you to implement a parallel version of the Sieve of Eratosthenes algorithm for finding prime numbers using Posix threads and OpenMP. This exercise asks you to use MPI instead of shared-memory parallelism constructs for the parallelization of your solution.
   You need to provide and/or measure the following:
      a. An MPI program using MPI Send() and MPI Recv() measuring its performance on one server.
      b. The same program as above measuring its performance on the three servers your hosts file mentioned above.
      c. An MPI program using MPI Bcast() and MPI Reduce() measuring its performance on three servers.

   Make sure you use an N which is large enough for the parallelization to make sense. Besides your code, you need to provide a short section in your report that explains how you modified your solution and reports the speedup curves you got.
      - Was the MPI version easier or more difficult to write?
      - Are the speedups you get the same better or worse (and why)?

   For this implementation, I took the parallel implementation that I did in assignment 3 using OpenMP as a starting point, and due to the nature of MPI and how threads work in this implementation, some changes were needed in the structure of the algorithm. This can be seen for example in the adaptations done since in MPI we have a master 'thread'. For this, all the common part of the calculations, such as the unification of the partial results obtained by each process, and the plotting of the execution time, need to be done by only the master process.
   I also had to change the bool vector for an int vector, since the send and receive functions from MPI didn't work correctly with this data format.

   To compile we use

   bexa2945@gullviva:~$ ***mpic++ erato.cpp -o eratos -std=c++11***

   and to run with only one server like in question a), we execute with

   bexa2945@gullviva:~$ ***mpirun -np 1 ./eratos 150***

   and to run with multiple servers we use:

   bexa2945@gullviva:~$ ***mpiexec --hostfile hosts ./eratos 150***

   In this case we're calculating only 150, since it's just to test the code.

   When measuring performance, I found that tussilago.it.uu.se was down, and therefore I could only test the performance using gullviva.it.uu.se and vitsippa.it.uu.se. I tried to connect to tussilago during the whole 13th, but it didn't work. Therefore, I did the measurements with only 2 servers, since that is what was available at a time.

When testing the first implementation, calculating the prime numbers up to 1.000.000.000, we get the following times table:

Num of servers	Execution time (ns)
1	84.209.550.311
2	298.672.492.627

We can observe that this implementation is actually slower when using 2 servers instead of 1. This can be due to my implementation being inefficient (I think).

For the implementation on c), we had to do some changes. Since in the execution times we could see that the multi-process version was slower than when running with only 1 server, I did 2 implementations of this second part of the exercise.
First, I implemented the same code as in the first part, but using broadcast and reduce instead of send and receive.
For the second implementation of this second part, I tried to improve the algorithm to make a better use of the methods broadcast and reduce, taking the recommendations from assignment 2 to parallelize the algorithm (which I misunderstood in the assignment 2, but thanks to the correction and feedback, I understand them now).
The Execution times that I get with this implementation, using both the logic described in Assignment 2 and the broadcast and reduce methods, is as follows (when calculating 1.000.000.000). Again, using only 2 servers since tussilago was not working (at least for me):

Num of servers	Execution time (ns)
1	95.148.787.957
2	201.167.392.154

We can observe that we still get a worst time in the execution with 2 servers, and it's also worse than what we observed with the first implementation using send and receive. I do not understand the reason for this behaviour, so if some explanation can be provided in the feedback, I would be very thankful :)

As for the difficulty of the code, I would consider the OpenMP implementation to be easier to write and read.