COMP 428: Parallel Programming
Winter 2012 Assignment 1

Grégoire Morpain - #6398391
gregoire.morpain@gmail.com

General notes:

All programs have been written in C.
You can compile using the Makefile with the command `make`.
By default, programs are compiled in “release” mode but you can specify which type of compilation you want by explicitly calling either “make debug” or “make release”.

After compilation, binaries are in the `bin` folder.
The binaries for Sequential QuickSort, Parallel Sorting using Regular Sampling and Parallel QuickSort are respectively called, as required, `qsort`, `psrs` and `pqsort`.

There are two scripts that generate random numbers, the way they work is explained in the README(.md).

All programs read numbers from a file named “input.txt” located in the execution directory and write sorted numbers in a file named “output.txt” in the same directory, as required.

Q.1. Parallel Sorting using Regular Sampling (PSRS)

After I implemented the PSRS algorithm I was able to measure the following speed-up.

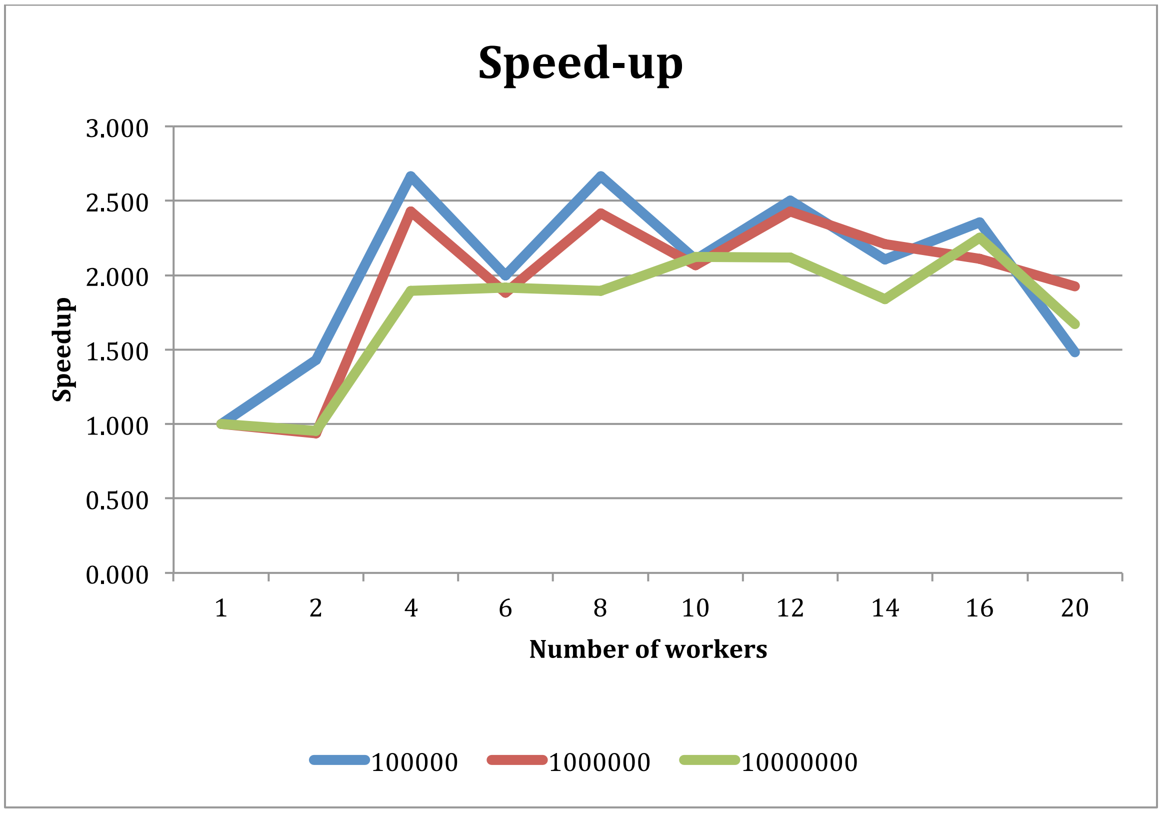
Notes:

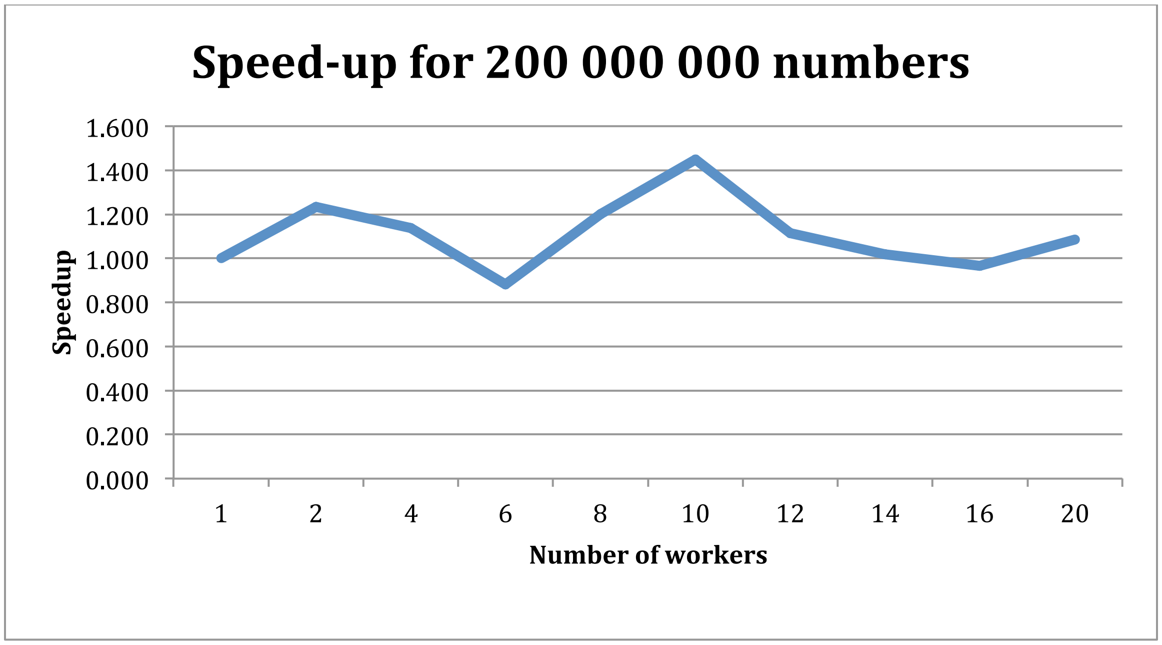
For some very large inputs, the time to read the data and write the output to another file can be very long and vary much. I measured the time inside the program so I can subtract it from total time.

All programs output the following text to stdout:

```
#> time bin/qsort
Loading values... done. (17.543s)
Sorting values... done. (26.585s)
Writing to file... done. (20.650s)
bin/qsort  60.70s user 4.13s system 71% cpu 1:31.07 total
#>
```

Time is taken from second output line.





NB: For such a large input (>1GB) I could only measure on my own laptop.

Q.2. Parallel QuickSort

Unfortunately my implementation of the parallel QuickSort on a d-dimensional hypercube is not functional so I could not see the actual performance by myself; but from my readings I can say a few things about their performances.

Initial partitioning can be very crucial in parallel Quicksort with MPI. With Parallel QuickSort, it's evident that the time efficiency gains from initial fixed partitioning are important. If not, a better way to perform initial partitioning through regular sampling is vital in order to gain higher performance.
The two parallel implementations tend to show less performance on higher number of processes since MPI communication overheads. This overhead has to compensate by limiting the amount of running processes according to the data set size.

Also partitioning through regular sampling can cause very irregular running time due to variable partitioning in each step. Also this may lead to a worst case of highly unbalanced data distributions that might lead to inefficient time performances.

Q.3. Consider the parallel formulation of quick sort for a d-dimensional hypercube that you implemented in Q.2. How does this algorithm compare with Bitonic sort using a d-dimensional hypercube? Does pivot selection play any role? Explain your answer either intuitively or analytically.

They can compare since they have (or almost) the same complexity. Pivot selection is crucial, indeed the way you select you pivots can give you balanced partitions or not, and this has lots of consequences on the performances, especially for the effective utilization of processors.

Q.4. In the parallel formulation of quicksort algorithm on shared-memory architectures (section 9.4.3) each iteration (step) is followed by a barrier synchronization. Is barrier synchronization necessary to ensure correctness of the algorithm? If not, then how does the performance change in the absence of barrier synchronization?

Barrier synchronization is unavoidable (for program correctness or program maintenance or shortage of time) but for some programs, a barrier abstraction may succinctly express the intent of the programmer but it could be a performance bottleneck. Relaxed barrier synchronization schemes could be more efficient than a full barrier and yet sufficient for correct execution of the program. The caveat is that custom synchronization code is generally hard to write, debug, understand and maintain.

Deciding which barrier (local or global) to use can be quite tricky. Programs such as quicksort and other divide-and-conquer algorithms that have a tree-like task graph are naturally expressed using local barriers.