COMP90024 Project 1 Report

Tingsheng (Tinson) Lai 781319, Mofan Li 741567

April 2019

1 Code Design

The full project source is available as a repository on GitLab: https://gitlab.com/laitingsheng/comp90024-project-1. It will be made public after the deadline of the assignment.

1.1 General Outline

The whole project is written in C++ (conforming to the latest standard ISO/IEC 14882:2017 with several usages of GNU extensions). It also includes external libraries which are available on the project Spartan[1], specifically, GNU OpenMP (version 4.5), OpenMPI (version 3.1.0) and Boost C++ Library (version 1.69). The code should be compiled with GCC with version of at least 8 which has full support of C++17. Boost.Test unit testing framework is also used to guarantee the correctness of the implementation.

1.2 Processing

The whole procedure is broken down to three smaller tasks: grid reading, data preprocessing and final processing/sorting.

1.2.1 Grid

The class `grid` defined in `grid.cpp` associated with header `grid.h` handles the process of reading the predefined grid and convert it into corresponding data structure. The grid will only be read once throughout the whole process. Each cell will be encoded numerically to avoid usage of `unordered_map` in the following processing step and accelerate the marshalling process (provided by Boost.Serialization) when Boost.MPI is used to support OpenMPI.
The constructed `grid` class can be used to determine if a given tweet is located in our interested area. In the final output stage, it will also be used as a decoder to convert the encoded number back to the actual grid cell names.

1.2.2 Processors

In the directory named `processor`, there are four types of processors designed and optimised for different configurations where as `processor` is the common abstract ancestor of `processor_sn_st` (Single Node Single Thread), `processor_sn_mt` (Single Node Multi Thread), `processor_mn_st` (Multi Node Single Thread) and `processor_mn_mt` (Multi Node Multi Thread).

1. The `processor`, as stated previously, defines the core processing functions for its descendants. The `process_line` function will first locate the substring `"coordinates"`. If it does not exists, the function will safely skip the following procedures. To handle the deprecated `"geo"` attribute as stated in Twitter's official development documentation [2], RegEx (or Regular Expression) is used to extract the attribute name and the coordinates from the string. Once the coordinate is extracted, current line will also be ignored if it is determined to be outside the region we are interested in. We also use RegEx to extract hash tags from the `"hashtags"` attribute and increase the corresponding counters.

2. The `process_block` function will iterate over a block of lines and apply the `process_line` function to each line within the block. The major idea is to divide the whole text files into several blocks. The number of blocks is determined by the number of nodes times the number of cores per node, i.e., the total number of cores available. To deal with the problem that the separation position between two blocks may cut one line of twitter object into halves, every core will read forward till the next newline character. The first core can also safely do so because the first line is some irrelevant information of the current file and it can be ignored. Each core should also tell the previous core their starting position since this will be the ending position of the previous core.

3. To reduce the file I/O time, we use `mmap` to map the entire file into the memory address space. In our implementation, we use the `mapped_file_source` class defined in Boost.IOStreams since it is fully optimised. This is comparatively faster than the operations provided by the traditional `std::ifstream` class, and it is much faster than the old style C file operations. Notes that each node will have its own file mapping within its own process so we don't have to transfer any data amongst different nodes. The only information which will be passed through MPI is the positions of each node, as described in the following section.

4. Concrete processors

   (a) SNST: only one node and one process will be handling the file and it is also responsible for the final processing/sorting.

   (b) SNMT: compare to SNST, multithreading, specifically OpenMP, is enabled to accelerate the procedure using the power of multi core computing. Since this is a SIMD/SIMT model on the same node, there will be no communication overhead since all threads share the same address space and they can use memory to communicate with each other. In our implementation, we use arrays to store necessary information of each thread. Final processing will use the parallel sorting algorithm from Boost, block indirect sort [3], on each grid cell, then sort the resulting object accordingly.

   (c) MNST: SPMD model, multiple processes/nodes will be spawned and each process will be assigned only one core. Boost.MPI, combined with OpenMPI, is used in this implementation. For process with rank $n > 0$, they will send their starting position to process rank $n - 1$ and the process rank $n - 1$ will need to receive it from process rank n. An MPI All Gather action will be performed on all cores to collect results from other cores to be prepared for the following processing step. Each process will be responsible for sorting processing the counting of hashtags of several cell grids as well, then the root process (with rank 0) will gather all information and output the result.

   (d) MNMT: multiple processes/nodes will be spawned and each process will bind to identical number of cores. This is a combination of SIMD/SIMT and SPMD model. In this model assume we have n processors and each of them has t threads, processor n thread 0 will send its own starting position to the previous processor, as described in MNST section, and processor $n - 1$ thread $t - 1$ will receive that. The remaining procedure will be basically a combination of SNMT and MNST.

5. All MPI-enabled processors requires the serialisation support from Boost.Serialization [4] to marshal the complex data structures of the C++ Standard Template Library.

1.3 Notes

1. There is a build script, named `build.sh`, under the project root directory. It will first loads all necessary libraries from the module system of Spartan and execute `build.py` to run the actual building process. This must be run beforehand. The generated executables and object files are stored in the `bin` folder.

2. The generator script, named `gen.sh`, is used to generate SLURM scripts for different configurations. It is just a bash wrapper of `gen.py` since the Python 3 module needs to be loaded before executing it. All scripts will be stored under SLURM folder in the project root directory. This is used to generate the data required in the next section.

3. We also put three SLURM scripts in the root directory for the three basic configurations stated in the specification. This can be run directly using the `sbatch` command.

2 Performance

As noticed in the `gen.py` script, the generated SLURM script will run the program 20 times (with dependency). The use of dependency is to avoid excessive amount of access from our own script, which can have a severe impact on other scripts. This is also why we didn't use Job Array. The following result is based on the average time of the 20 executions for each configuration.

2.1 Required Configurations

As required explicitly in the specification, the actual execution time of the three configurations [1] $(1, 1)$, $(1, 8)$ and $(2, 4)$ are shown in the left. The right two images are the speed up and efficiency.

[1] Configuration is written as a tuple of $(nodes, cores\_per\_node)$

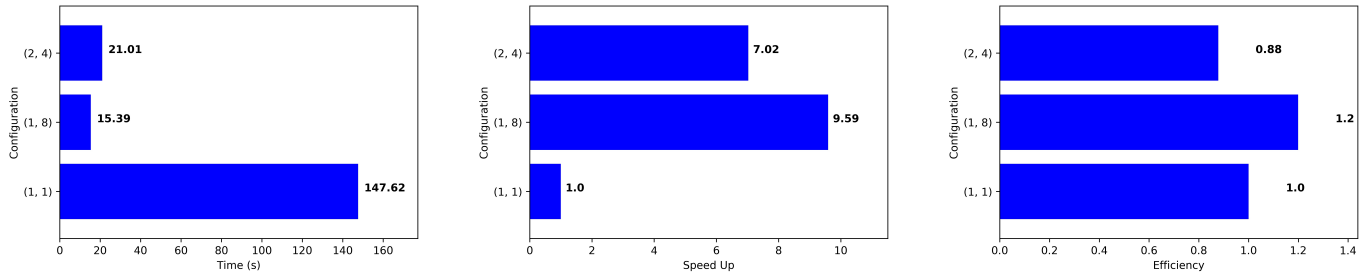

Figure 1: Required Configurations

2.1.1 Time

From the left graph, we can clearly see that using OpenMP is indeed advantageous for single node parallel computation. However, it can be spotted that the configuration $(2, 4)$ is not performing well in our implementation though it occupies the same number of total cores as $(1, 8)$, but the time of completion is still magnificent. This is due to MPI All Gather action used in our implementation, which requires several serialisation/deserialisation processes of complex and large data objects. Although we can use a MPI Gather instead and do the final proessing/sorting in only one node, we want to generalise the design so that we can parallelise such procedure to deal with grid which may be extended to the whole world instead of just focusing on locations in Melbourne.

2.1.2 Speed Up and Efficiency

Accordingly, the speed up is calculated by

$$S(p) = \frac{T(n)}{t(n)}$$

where $T(n)$ is the sequential execution time and $t(n)$ is the execution time of the parallelised model. Efficiency is, therefore, calculated by

$$E = \frac{S(p)}{p}$$

where $S(p)$ is the speed up and p is the number of processors.
It is mysterious and surprising to have an efficiency greater than 1. However, we think it is explainable because the effect of IO overhead is relatively minimised and the total IO overhead time will be distributed across these spawned threads.

2.2 More

We also ran our code on several different configurations as well. Here are the three graphs in the same order as those in previous section.
We can clearly see that the level of speed up is increasing when we add more processors and nodes. Obviously, the execution times are reducing alongside with the addition of total processors. By Amdahl's Law,

$$S(p) = \frac{t_s}{f \cdot t_s + (1 - f) \cdot \frac{t_s}{p}} = \frac{p}{1 + (p - 1) \cdot f}$$

where f stands for the time (complexity) of sequential run time within the program. Therefore,

$$E = \frac{S(p)}{p} = \frac{1}{1 + (p - 1) \cdot f}$$

As f is irreducible, the efficiency E will gradually decrease when the number of processors p increases. This is also reflected in the rightmost graph as well. Apparently, the efficiency is indeed decreasing. In our implementation, since we need a final merging after the data is processed, the amount of time taken to do the merge will increase even though it is also parallelised. For execution distributed across multiple nodes, the effect of serialisation/deserialisation is even more severe. Nevertheless, the speed up is remarkable and efficient but the MNMT processor is probably not optimised enough.

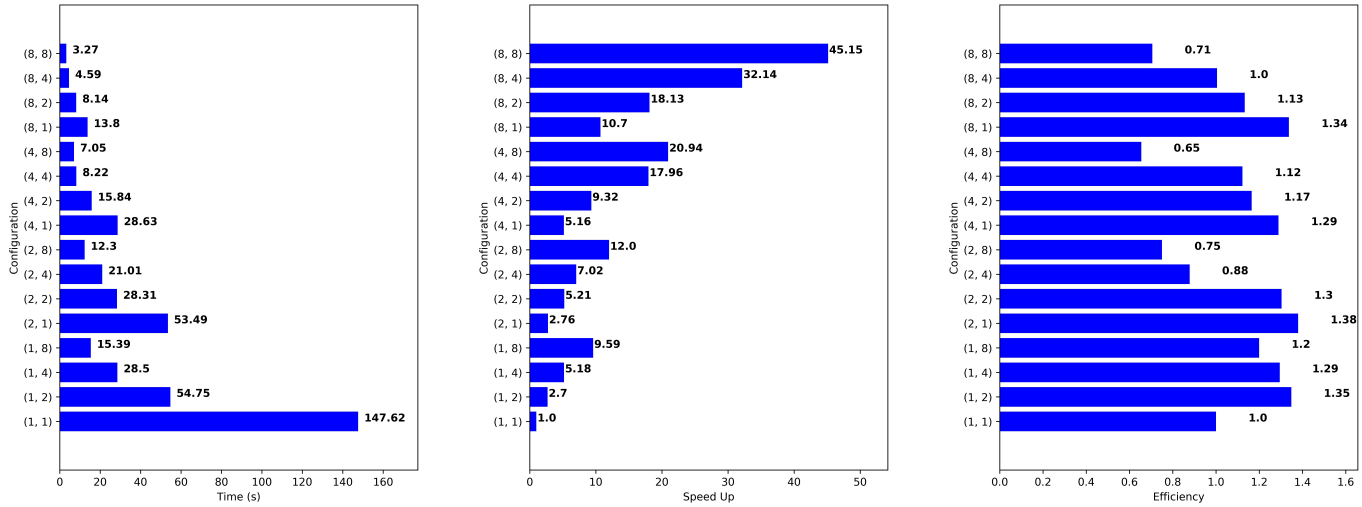

Figure 2: Extra Configurations

3 Improvement

1. In previous versions, we try to implement a work stealer for the SIMD/SIMT model, but eventually we found that the overhead of avoiding race condition is much larger than what we expected. Therefore, we finally abandon that method. However, we still think it is possible to do so and it may further improve the performance of our program.

2. In real world application, the Twitter data will be retrieved as a stream of realtime data. This will require a master-slave architecture to handle it, for example, the master node dispatches work to the slave nodes and gather the result. But we chose to aggressively optimise our program for this assignment.

References

[1] Lev Lafayette, Greg Sauter, Linh Vu, Bernard Meade, Spartan Performance and Flexibility: An HPC-Cloud Chimera, OpenStack Summit, Barcelona, October 27, 2016. doi.org/10.4225/49/58ead90dceaaa

[2] Twitter, Tweet object. https://developer.twitter.com/en/docs/tweets/data-dictionary/overview/tweet-object

[3] Boost C++ Libraries, Parallel Algorithms. https://www.boost.org/doc/libs/1_69_0/libs/sort/doc/html/sort/parallel.html

[4] Boost C++ Libraries, Serialization, https://www.boost.org/doc/libs/1_69_0/libs/serialization/doc/