Project 3 Parallel System Design Report

Siqi Li (SID:12333870)

December 3, 2021

Abstract

This report is a comparison of performance difference between the serial and bsp version of COVID data processing system based on the data generated from the result of running covid.go. In this report, there are five sections: system description, specifications, instructions on how to run testing script, experiment results and limitations.

1 System Description

1.1 Problem

This project is a data wrangling system for COVID data. There are both a sequential version and a parallel version of implementation calculating the total number of cases, tests, and deaths for a zip code in a given month and year. The input of the system is small, medium and large .csv datasets from the Chicago Data Portal. The output of the program is the cumulative total number of cases, tests and deaths for each category.

1.2 Solution Description

The file structure is shown in Figure 1. There are two modes for the COVID data processing system: sequential and BSP. The `covid.go` file is the main function. The `sequential` version is implemented in `covid.go` also. The BSP implementation is in the file `bsp.go`. The details of the sequential and BSP implementation are discussed in the subsections below. The other files in Figure 1 are used for correctness testing and performance testing. Details of testing is discussed in the next section.

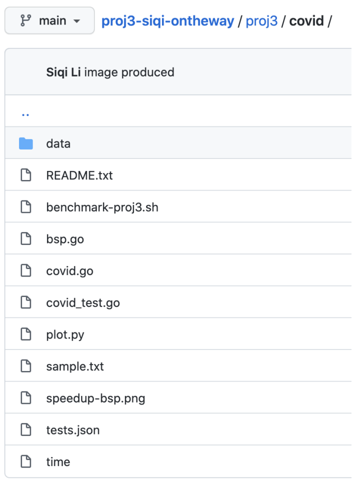

main
proj3-siqi-ontheway / proj3 / covid /
Siqi Li image produced
..
data
README.txt
benchmark-proj3.sh
bsp.go
covid.go
covid_test.go
plot.py
sample.txt
speedup-bsp.png
tests.json
time

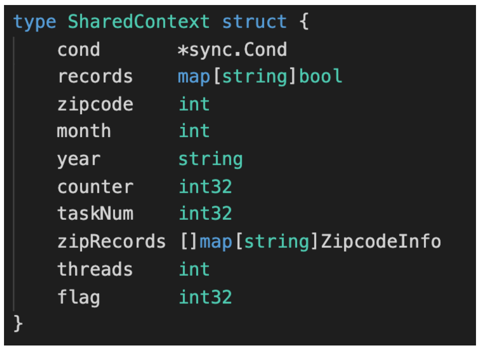
Figure 1: File Structure

1.2.1 Sequential Version

In sequential version, a single thread will read a .csv file line by line and then determine if the zipcode, month and year of the line satisfies the searching category. If yes, the tests, cases and deaths data will be stored in a map. This process will continue to be repeated for the total 500 .csv files one by one. After all files are processed, the cumulative data will be calculated by a for loop through all the records of the map.

1.2.2 BSP Version

In BSP version, a `SharedContext` (Figure 2) is created for the communication between different goroutines. Thread-Num of goroutines are spawned. Each single goroutine will work on a separate .csv file by reading the file and storing the data in the map in SharedContext. Once a goroutine is done with the assigned file, it reaches the sync point and waits for other goroutine to finish their work. It will wait until all goroutines reach the sync point and move on to the next bulk of tasks. Unlike in the image editor project, before entering the next phase, waiting goroutines will save the image, in covid project the waiting goroutines do not have a lot of work to do at the sync point. They are just waiting for the last goroutine to finish and `Broadcast` to notify all the waiting goroutines in the sync point so that all workers/goroutines will enter the next phase together.



```go
type SharedContext struct {
    cond       *sync.Cond
    records    map[string]bool
    zipcode    int
    month      int
    year       string
    counter    int32
    taskNum    int32
    zipRecords []map[string]ZipcodeInfo
    threads    int
    flag       int32
}
```

Figure 2: Shared Context Struct

As shown in Figure 2, `cond` field is used as a conditional variable to notify all goroutines to start the next phase together. The `zipRecords` field is used to store the COVID data for each goroutines. The `records` field is used in the final part when we need to combine searched data of all goroutines together.

1.3 Challenges

From data wrangling perspective, the process of gathering, and transforming data to get ready for doing data analytics is a tedious task because the raw data may have empty fields, duplicated records, be dispersed among many files, etc.

From parallelism perspective, the challenge of the project is to implement BSP model in order to increase the performance. Previously I implemented the BSP model in a image editor in project 2. The bottleneck of that project is more in the computation of CPU. However, the bottleneck of covid data processing is more in the I/O process to read data from file. So from this project, I want to learn how BSP mode will perform differently.

2 Specifications

2.1 Testing Machine

I ran my experiments on the linux.cs.uchicago.edu cluster. Specifications are:

- 16 Cores (2x 8core 3.1GHz Processors), 16 threads

- 64gb RAM

- 2x 500gb SATA 7200RPM in RAID1

2.2 Data Source

The data is downloaded from the Chicago Data portal and the covid*.csv files are placed inside the `data` folder under `covid` folder. There are three granularity of the parallel system, small, medium and large. The large data includes all 500 files. The medium data includes 250 files. The small data includes 100 files.

3 Run Testing Script

3.1 Preparation

Before running the program, please check if you have the following installed:

1) python3(python-cs-cluster)

2) Matplotlib.pyplot(install)

3) Pillow(install)

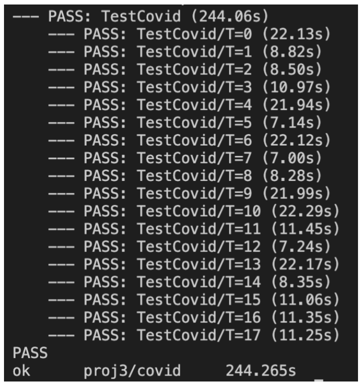
3.2 Run Test

Before running test, please download code into UChicago CS linux cluster machine:

```
git clone https://github.com/mpcs52060-aut21/hw5-siqi-ontheway
```

3.2.1 Correctness Test

You can run correctness test by running in the covid folder: `go test -v`. It will run the covid-test.go program and generate a testing report as shown in Figure 3 on the right.



```
--- PASS: TestCovid (244.06s)
    --- PASS: TestCovid/T=0 (22.13s)
    --- PASS: TestCovid/T=1 (8.82s)
    --- PASS: TestCovid/T=2 (8.50s)
    --- PASS: TestCovid/T=3 (10.97s)
    --- PASS: TestCovid/T=4 (21.94s)
    --- PASS: TestCovid/T=5 (7.14s)
    --- PASS: TestCovid/T=6 (22.12s)
    --- PASS: TestCovid/T=7 (7.00s)
    --- PASS: TestCovid/T=8 (8.28s)
    --- PASS: TestCovid/T=9 (21.99s)
    --- PASS: TestCovid/T=10 (22.29s)
    --- PASS: TestCovid/T=11 (11.45s)
    --- PASS: TestCovid/T=12 (7.24s)
    --- PASS: TestCovid/T=13 (22.17s)
    --- PASS: TestCovid/T=14 (8.35s)
    --- PASS: TestCovid/T=15 (11.06s)
    --- PASS: TestCovid/T=16 (11.35s)
    --- PASS: TestCovid/T=17 (11.25s)
PASS
ok      proj3/covid     244.265s
```

Figure 3: Correctness Testing Report

3.2.2 Performance Test

You can run performance test by running in the covid folder: `./benchmark-pro2.sh`

It will run the shell script which includes a python program to plot a speedup graph `speedup-bsp.png` based on the time to run the sequential and parallel version. The `sample.txt` is the results I got after running on the cluster and the data in this file is used for performance analysis in this report.

Please be noted that you can also sbatch to submit the script to slurm cluster. If you sbatch and cancel the sbatch task after running it, you need to clear the `time` file before re-submitting the task.

4 Experiment Result

4.1 Parallel Speedup

The parallel speedup is shown in Figure 4. As shown in the image above, as the goroutine number increases, the speedup will steadily increase to five times faster. The speedup is pretty much the same for small, medium and larget dataset.

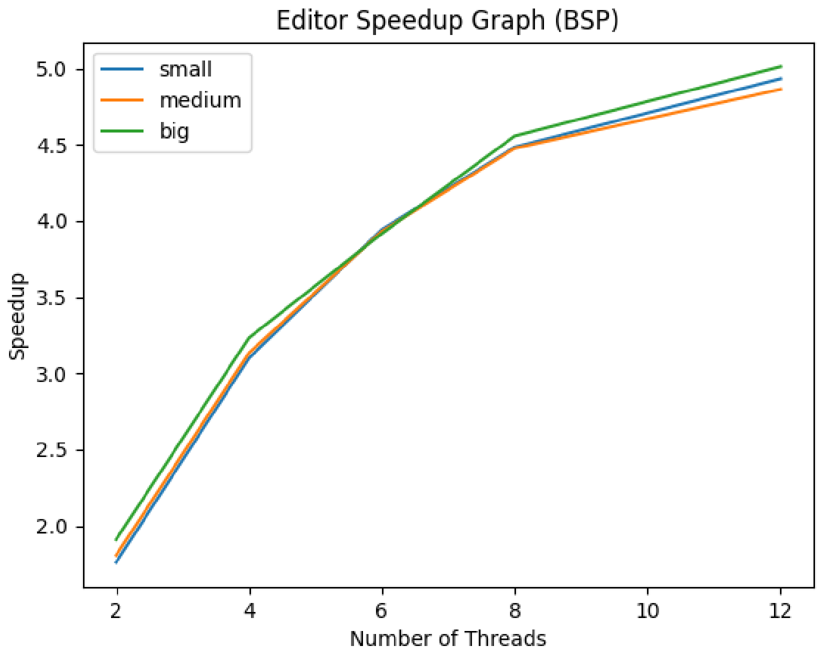


Figure 4: BSP Speedup

4.2 Hotspots and Bottlenecks

For the parallel version, workflow includes the following parts:
a) Read .csv data
b) Determine if the data satisfies the requirement and record the related data in the map.
c) Synchronize the local result with the global map.

I/O is usually the `bottleneck` of the program, as said in the previous lectures, because when the file is being read or written, other files will need to wait until their turn to be processed. This may cause a slow down in performance. The parallel design may reduce the bottleneck a bit because several goroutines may read different file together at the same time. Even though for every goroutine, the I/O part is still a bottleneck in each superstep.

As for the `hotspots`, since the recording of local map consumes the computation of CPUs most, compared to part c to synchronize local map to global map. So the recording of local map is the main hotspot of the program. The parallel design solves this by spawning several workers to record local maps for different files at the same time. This way, the hotspot is much reduced.

4.3 Limitation

The speedup is limited by the communication between goroutines. In BSP model when a single gorutine finishes reading its own file, it will wait at the sync point for other goroutines to finish reading their own file before proceeding to read the next file. So the waiting time is the main limitation of the speedup.

This conclusion can be supported by the tendency of speedup lines in Figure 4. As the number of spawned goroutines increase, the speedup lines increase fast at the start (2 goroutines)and increase slower at the end (12 goroutines). This is partially because as the goroutine number increase, the waiting time at the sync point will increase accordingly, which reduces the speedup performance.

References

[1] Class materials.