Project 3 - System Write-up

Project and System Description

This project implements a parallelized COVID data processor. The program takes as input a zipcode, a month, and a year specified by the user, parses through ≥ 500 csv files with potential duplications and invalidate entries to gather and generate the total number of COVID cases, tests, and deaths for the specified zipcode over the specified month and year.

The Dataset:

The datasets used come from City of Chicago Data Portal's COVID-19 Cases, Tests, and Deaths by ZIP Code dataset. The original unmodified data from the source with unique entries are stored in `covid_sample.csv` .

For the program, we will use modified 500 data files named `covid_NUM.csv` , where `0<=NUM<=500` , each consisting of about ~37,000 random lines sampled with replacement from the source file. The 500 files are generated in such a way that it is guaranteed that together they contain all entries from the source file. The modified datasets are available here: https://uchicago.app.box.com/s/6quo5pf75riwv6va6356g3yrgolmw4az

In some test cases, we will use more than 500 files. In those cases, we will simply recycle one of the 500 files for each file number greater than 500.

Parsing the Data:

Each file to be parsed is a csv fileand our program only need the columns below stored in `proj3/utils/utils.go`:

```
const zipcodeCol = 0
const weekStart = 2
const casesWeek = 4
const testsWeek = 8
const deathsWeek = 14
```

The `weekStart` column gives the start date of the week of a particular weekly record in the format `"MONTH/DAY/YEAR"` . If the `MONTH` and `YEAR` matches the month and year specified by the user, the program will treat the row as a valid entry. The case in which a week may span across end of previous month and beginning of next month is ignored for simplicity.

Furthermore, if any of the columns specified above is missing, or if the zipcode does not match the user's specified zipcode, the record is also ignored.

Implementations and Usage:

The program has a sequential implementation and 3 parallel implementations, which can be toggeled by specifying them in the usage statement below.

```
const usage =
    "Usage: go run proj3/covid mode size threads zipcode month year\n" +
    " mode = either 'static' or 'stealing' or 'bsp'\n" +
    " size = 500 or 1000 or 3000, the number of files to be processed\n" +
    " threads = the number of threads (i.e., goroutines to spawn).
                If bsp, must be > 2 \n" +
    " to run sequential mode, specify thread = 0 when the mode is either static or stealing\n" +
    " zipcode = a possible Chicago zipcode\n" +
    " month = the month to display for that zipcode, must be between 1-12 \n" +
    " year  = the year to display for that zipcode, must be 2020 or 2021 \n"
```

All four implementations are stored in the `modes` directory.

In the sequential implementation, the program simply parses through the files one at a time while keeping track of the running total.

In `static` mode, the program spawns `threads` number of goroutines and divides up the total amount of work specified by `size` (for sake of simpler benchmarking, limited to either 500 files, 3000 files, or 10000 files) following static distribution (that is, each thread will get `size / threads` number of files to parse, with the last thread taking in any remainders). Each thread will first process all the files assigned to them, accounting for any duplications, and once finished, will attempt to update the global record which is protected by a `TTAS` lock. The `TTAS` lock is used to improve performance by allowing threads to spin on a local cached copy of the flag and notified only when cache is invalidated, instead of all spinning on the same memory location and creating massive contention.

In `stealing` mode, the program implements work stealing on top of initial static distribution. The implementation makes use of auxiliary data structures and structs stored in the `stealing` package placed under the corresponding directory. Work stealing here is implemented using a double ended queue `DEQueue` of tasks, each task is a runnable where the task of parsing a particular file and adding its records to the global record is encapsulated. Therefore, compared to `static` implementation, in `stealing` mode synchronization using a `TTAS` lock occurs after each file is parsed. The main goroutine will spawn `threads` number of workers and `DEQueue` where each worker is assigned its own local queue. The main goroutine will initially assign equal number of tasks to each thread by enqueing roughly equal number of runnable tasks to each of the local queues. Then it will let the threads dequeue and execute runnables from their local queues from the bottom of the queue. If for some reason a thread finishes ahead of others, it will then attempt to steal a task from unfinished threads by random selecting a victim and trying to dequeue the stolen task from the top of the victim's queue. If all threads run out of tasks, then they will stop and exit.

I will dedicate the next section to explain the `bsp` mode, which is implemented for **Part 2: Advanced Feature(s) section.**

The details of each parallel implementation will be discussed in more detail in the speed up analysis section.

Advanced Feature (BSP)

`bsp` mode consists of series of interlocked super-steps and global synchronization step. In the super-steps, each thread is assigned a file, each file of roughly the equal size to process. The threads will process the file and keep a local count of the records from the file. If a thread finishes before others, it will wait for all its peers to finish. When all threads have finished processing their files, all of them enter the global synchronization step where they reflect their local records to the global records. Once that is done, another sets of files are assigned to each of the workers and the steps are repeated. The process ends when the global context determines that there are no more files to be processed in the next round. The coordination between the super-step and global synchronization step is done by using condition variables in the `sync` package. The main goroutine will run a function for global synchronization that's activated each time all workers finish their work and lay idle, while goroutines are spawned off to do work during the super step.

The codes for `bsp` mode is stored in `bsp.go` stored under the `modes` directory.

Scripts for Running the Program:

The program can be ran following the usage statements provided in the section above. In the `proj3/covid` directory, run the following commands to produce the results below:

```
$: go run proj3/covid static 500 0 60603 5 2020
2,48,0
$: go run proj3/covid bsp 3000 4 60640 2 2021
182,9961,5
$; go run proj3/covid static 1000 3 89149 2 2020
```

```
0,0,0
$; go run proj3/covid stealing 500 3 89149 2 2020
0,0,0
```

Since data is read in using `../data` path in the `utils.go` file, you must be in the `proj3/covid` folder to run the program. Otherwise, the program will fail to get the data.

Speed Up Graphs and Analysis:

Experiments Setup:

- **Experiment Parameters:**
  - To benchmark and test the performance of the parallel system, I ran the program for the specification `60603 5 2020` (get the COVID statistics of zipcode 60603 in May, 2020) varying the implementation mode (sequential and 3 different parallel modes), file numbers to process, and thread numbers
  - In particular, I varied the granularity of the system by running the program to parse through `500`, `1000`, and `3000` files respectively.
  - I varied the number of threads working simultaneously to parse files to be `N = {2, 4, 6, 8, 12}` respectively
  - I ran the 3 parallel implementations: `static` (static distribution), `stealing` (work stealing), and `bsp` (bulk synchronous parallel), and measured the speedup against `sequential` implementation using the formula:

$$Speedup = \frac{(wall - clock\ time\ of\ serial\ execution)}{(wall - clock\ time\ of\ parallel\ execution)}$$

- **Testing Environment**: The tests were ran on `debug` partition of the Peanut Cluster (SLURM) at University of Chicago. The specification of this partition can be checked by running `lscpu` on the cluster, and the specification is:

Architecture:	x86_64
CPU(s):	16
Thread(s) per core:	2
Core(s) per socket:	4
Socket(s):	2
NUMA node(s):	2

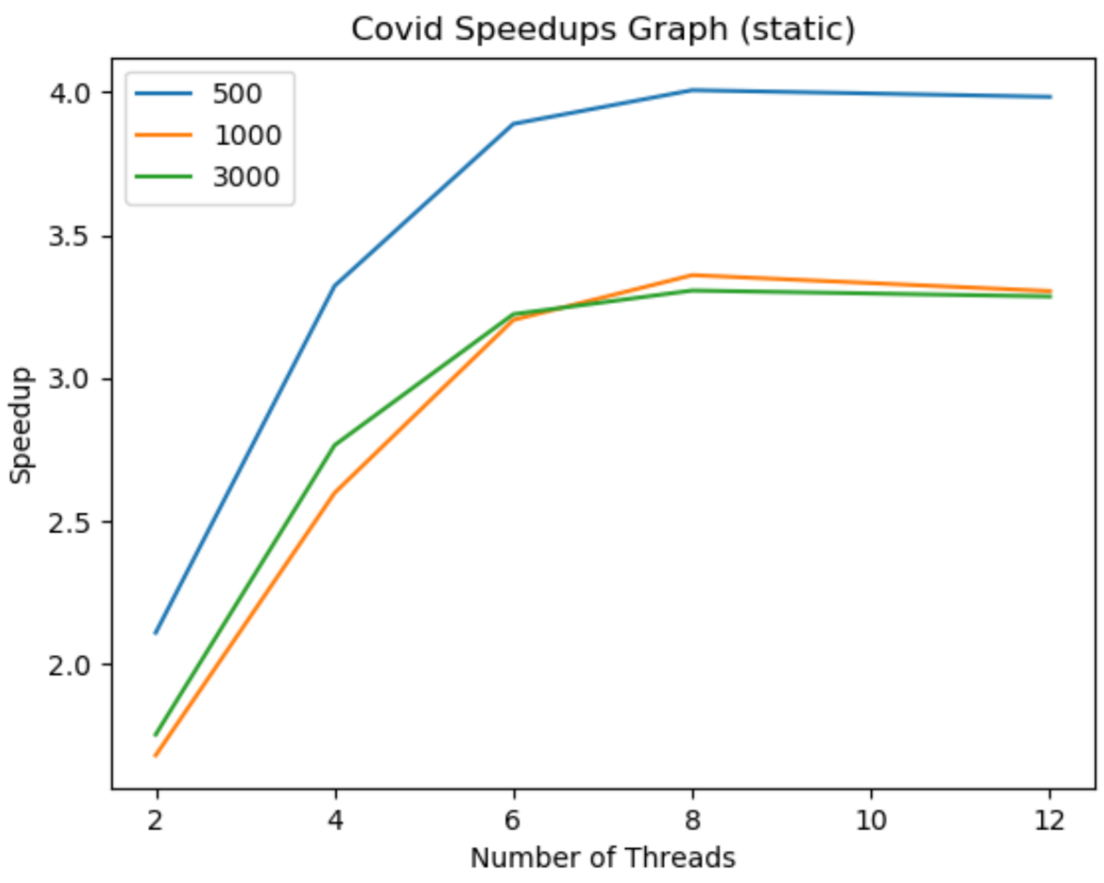
CPU family:	21
Model:	2
Model name:	AMD Opteron(tm) Processor 4386
CPU MHz:	3399.083

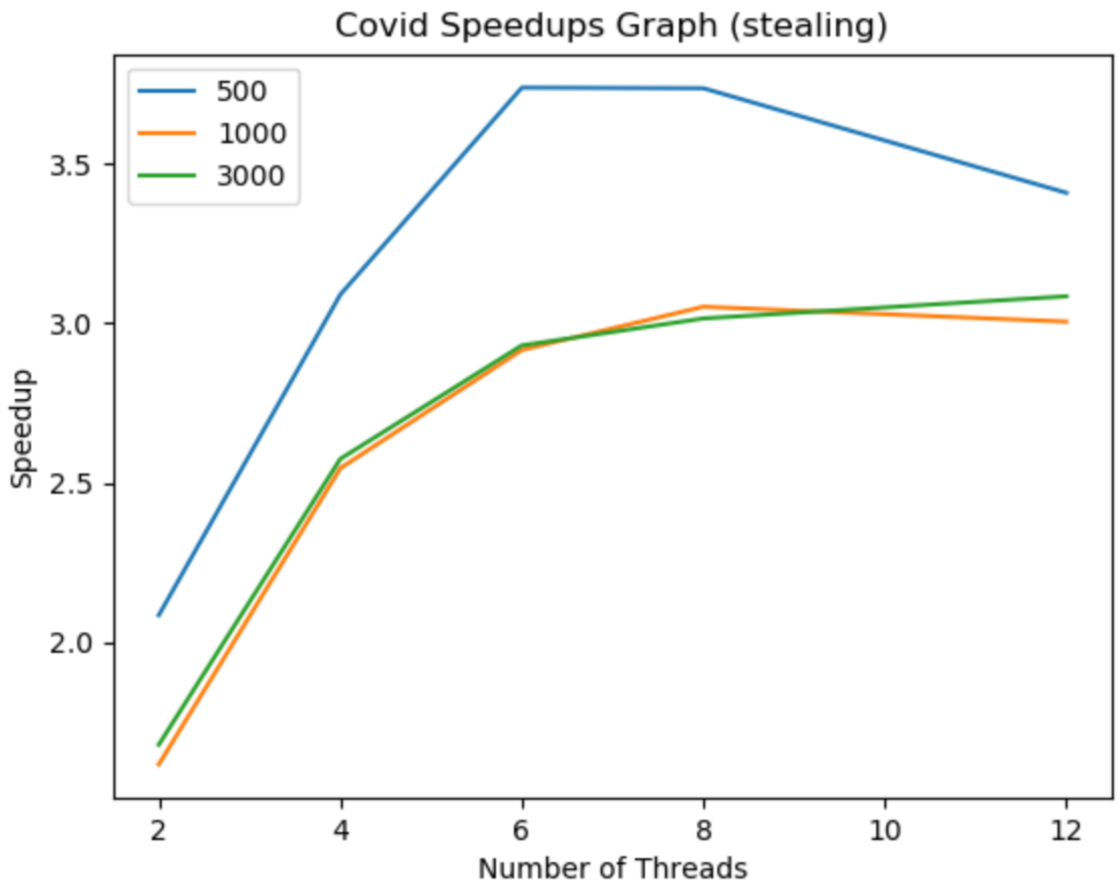
- Testing script:

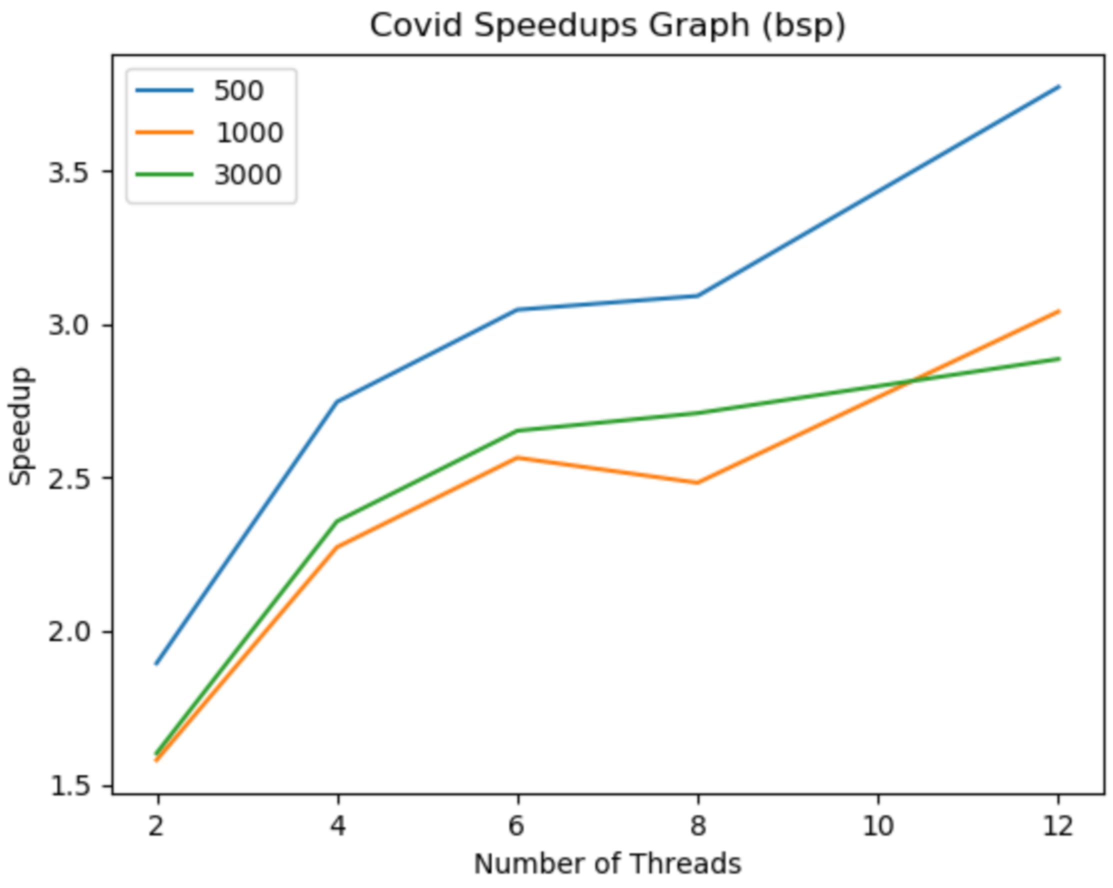
  I created a python script `graph.py` to run the experiment on Peanut cluster and produce the speedup graphs automatically. The script for benchmarking the program can be ran in the `benchmark` directory as below:

```
$: sbatch benchmark-proj3.sh
```

Speed Up Graphs:




Covid Speedups Graph (stealing)
500
1000
3000
Speedup
3.5
3.0
2.5
2.0
2
4
6
8
10
12
Number of Threads



- We can see that for all 3 parallel implementations, there is a sizable speedup that tapers off / plateaus around 3.5 ~ 4 as number of threads increase towards the number of physical cores of the machine (the Peanut cluster has lot more cores, but using the `debug` partition limits our access)
- Hotspot, bottlenecks, and challenges faced when implementing the system:
    - The hotspot of the program is parsing through each of the file. This is mostly parallelized by making different threads work on parsing a different file. In theory, we can completely parallelize the hotpsot if we have enough cores, since there is no data dependency. The order in which we parse through the files do not matter as long as we do parse through them all
    - There are two bottlenecks in this program. The first bottleneck is that the program is I/O heavy, as each thread is reading in csv file one by one. The second bottleneck is the necessary synchronization step to reflect records in local files to the global

record to sum up the COVID cases as well as preventing double count of duplicated records.

    - The challenges for implementing this sytem is to find an efficient parallel implementation will try to minimize the contention around updating the global record. Implementing this system using different parallel design patterns helped me understand how to identify the spots where synchronization is most needed and minimize unnecessary communication overheads

- The static distribution implementation turned out to be the most efficient parallel implementation with the higest speedup at 4.0x, and little degradation (pleateaus) as threads are increased towards limit. In this implementation, the bottleneck around synchronizing against global record is minimized because we make each threads finish all of their assigned works first before contending for the right ot update the global record. Each thread keeps a local tallied record with no duplicates and updates the global record when the local record is ready
- The work stealing implementation performs a little worse. The benefit of work stealing is that if for some reason some thread performs significantly worse than others due to some scheduling idiosyncracy, the threads finishing early can help out the slower threads. However, in our implementation, we try to distribute equal amount of work to each threads and they all perform roughly equally, so situation in which lots of stealing is needed does not occur often. Therefore, we don't see much performance benefit of work stealing over static distribution. Furthermore, for the particular way in which we implemented work stealing, contention on updating global record occurs each time a runnable is executed (a file is parsed), so we also see a degradation of performance as thread number increases due to too many threads contending for the critical section of updating global record.
    - Moreover, we also have not optimzied work stealing implementation. RIght now, if a thread finishes, it chooses a victim thread randomly to steal from. Since work has been distributed equally, it's like that by the time one thread finishes, most other threads also finished, so there is a high chance that the randomly chosen victim is empty, and the thief has to pick another victim. A different algorithm / mechanism to determine the victim such that stealing occur more effectively and frequently could improve the performance
    - Another way to solve this issue is to implement work balancing instead of work stealing. We can keep track fo the number of remaining tasks in each threads, and let threads with few tasks to balance workload with threads with lots of remaining

    tasks. This way work is more effecitvely exchanged between threads and minimizes the chance of threads being idle or overloaded

    - Finally, we can also redefine our runnables by limiting each task to just parsing the file, and keep a local record for each thread. When all threads are finished executing through all the tasks, we can have a sequential section in which the program go through each local records one by one and reflect to the global record. This could reduce the contention around global record

- For bsp implementation, speedup is the worst and also the slowest. Whereas the other implementations reach 3.5x speedups around 6 threads, bsp is only at 3.0x speedup at that point. However, among all implementations, bsp is scaling up the best as we still see an upward trend as threads increase and could potentially improve more. The downside of bsp implementation is that we are forcing sequential bottleneck at each iteration of the global synchronization step. In the global synchronizing step, we have to go through all of the local records of the workers to update the global record one by one. The upside is that there is no contention at all here. However, the synchronizing involved with condition variables and waking up threads may be large enough to offset those benefits. Since global synchronization step takes care of contention for the right to update global record, we see that bsp performance does not degrade when thread number increases, and the speedup increases almost linearly
- Finally, for all implementations, speedup is much better when the number of files to be parsed is small (500). The performance of 1000 and 3000 are roughly the same within a particular implementation for all implementations. The shape of the speedup however are roughly the same for all sizes. One potential reason to this is that I am recycling used files (when task number is > 500, I circle back to parsing file no.1 again and so on) to simulate having to parse more files. The filesystem may have kept some cache of the files to remove the I/O bottleneck that exist in the sequential program, which may have reduced the benefit of parallelization.