Homework 5

MPCS 52060 – Parallel Programming

Due: August 1st @ 11:59pm

Preliminaries

You can use the the unix command `time` to time the execution of your program (I'm sure Windows has an equivalent command). The `time` command will print out the real-time (i.e., wall-time), user-time, and system time. You will want to use the "real" time measurement.

```
~ lamont$ time ls > output

real 0m0.005s
user 0m0.002s
sys  0m0.003s
```


Remember, the formula for speedup:

$$Speedup = \frac{\text{wall-clock time of serial execution}}{\text{wall-clock time of parallel execution}} \quad (1)$$

You will write your own script/program to produce the time results. For example, I wrote a Python script that uses `timeit` to produce my times and made a graph based on those times.

Initial Setup

1. Create a directory called **hw5** inside your repository. It must be called **hw5** with all lowercase letters.
2. The **hw5** directory should have the following (bold items represent a directory):
   - report.pdf (or some other readable file)
   - testing script - a single file that allows us to run it in a terminal window
   - proj1 work - all you prior project 1 work that your testing script will need.

There is no specific structure required for this homework assignment. You can make the structure as you see fit to get your testing script to work correctly.

Part 1: Performance Measurements

You will run timing measurements on both the sequential and parallel versions of `twitter.go`. The problem sizes (i.e., the total number of operations being processed for one program execution), P, will remain the same for both versions. For each problem size specified, I recommend you write a program that randomly generates a file that has P operations. For example, if $P = 7$ then it could potentially generate this file:

```
{"command": "ADD", "id": 1, "body": "1", "timestamp": 5}
{"command": "ADD", "id": 2, "body": "2", "timestamp": 4}
{"command": "ADD", "id": 3, "body": "3", "timestamp": 2}
{"command": "ADD", "id": 4, "body": "3", "timestamp": 3}
```

```
{"command": "CONTAINS", "id": 5, "timestamp": 4}
{"command": "REMOVE", "id": 6, "timestamp": 2}
{"command": "DONE"}
```

We don't actually care about the content of these generated files. However, their ids should be unique and there are no duplicated operations. What we care about is the throughput that is how well does your twitter.go file service these operations. On the homework page, I have a file called "generator.go" that is a rudimentary implementation of this generation. It contains a usage statement that tells you how to use it. Feel free to adapt and change it as you see fit. **Note:** I would recommend that you do not have any `"FEED"` operations because it will make the output file very large and take additional time to print out large amounts of feed data. Most likely generating a high number of operations (about 1 million) will take some time on your personal laptops. I will post on how you can also run this on the cluster machines. **Think about the type of operations you are generating in the file and what effect it may have on the performance of your application...**

For the parallel version, the block size (B), will remain fixed given the problem size and the number of workers. Additionally, for each execution you will spawn a certain number of worker threads (N).

Parameter Amounts:

- $P = \{50000, 100000, 500000, 1000000\}$
- $N = \{1, 2, 4, 6, 8\}$
- $B = ceil(P/N)$

Timings

For the sequential version, you will produce 4 timing results based on P:

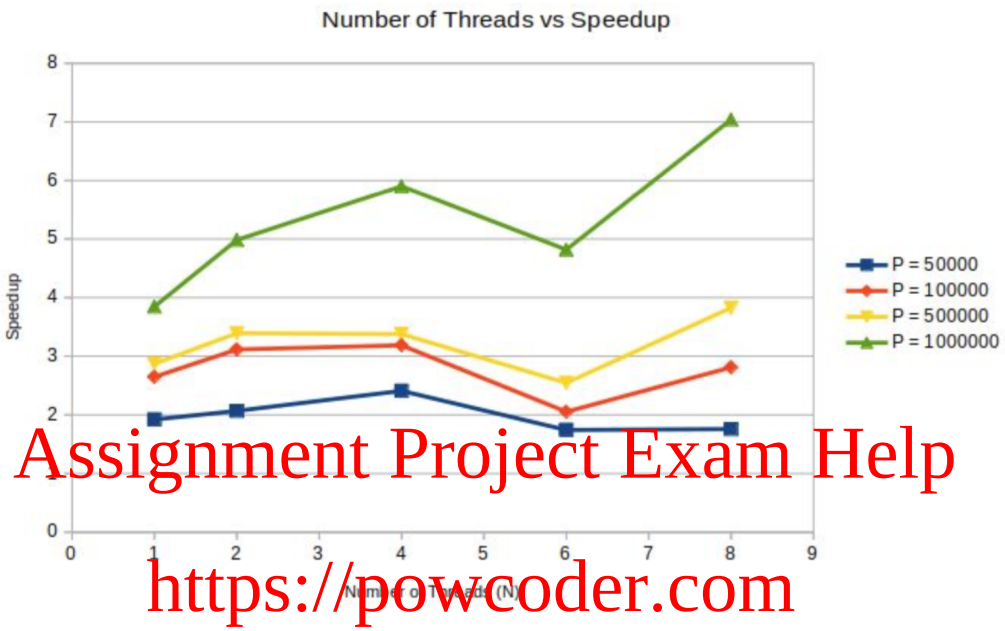
- Timing 1: $P = 50000$
- Timing 2: $P = 100000$
- Timing 3: $P = 500000$
- Timing 4: $P = 1000000$

For the parallel version, you will produce a timing result per number of worker threads with a fixed problem size and block size. Thus, you will have 20 timing results:

- Timing 5: $P = 50000, N = 1, B = ceil(P/N)$
- Timing 6: $P = 50000, N = 2, B = ceil(P/N)$
- Timing 7: $P = 50000, N = 4, B = ceil(P/N)$
- Timing 8: $P = 50000, N = 6, B = ceil(P/N)$
- Timing 9: $P = 50000, N = 8, B = ceil(P/N)$
- Timing 10: $P = 100000, N = 1, B = ceil(P/N)$
- Timing 11: $P = 100000, N = 2, B = ceil(P/N)$
- Timing 12: $P = 100000, N = 4, B = ceil(P/N)$
- Timing 13: $P = 100000, N = 6, B = ceil(P/N)$
- Timing 14: $P = 100000, N = 8, B = ceil(P/N)$

- ...

Using those timings, produce a speedup graph. The y-axis will list the speedup measurement and the x-axis will list the number of worker threads. Similar to the graph shown below. Make make sure to title the graph, and label each axis. Make sure to adjust your y-axis range so that we can accurately see the values. That is, if most of your values fall between a range of [0,1] then don't make your speedup range [0,14]. The below is graph from a prior student:



1

Note: Do not think your graph needs to look like this! Your results will vary based on your implementation. You may have speedups or you may not. It will be your job in your analysis section of your report to explain your results.

Testing Script

You will need to submit a testing script that will produce the required graph(s) from scratch. What I mean is that we should be able to open a terminal window and enter in a single command that runs, performs all the necessary timings, and produces your graph(s). For example, I use Python, which does all the timings and produces a Seaborn graph by calling my twitter program from within Python. Thus, the following command will produce a graph similar to the one above:

```
$: ipython make_graphs.py
```

Feel free to modify your twitter.go file such that you can do all the timings from within Golang. For example, you could augment your twitter.go file with code that is in charge of running the"actual" twitter.go code from project when necessary and record the timings so later you can produce graphs.

```
import (
   "time"
)
func main() {
// Note: You will most likely embed this code within some other code but the code.
// This for-loop runs the "actual" twitter.go code 20 times.
for i:=0; i <= 20; i++ {
```

```
    start := time.Now()
    /// Starting running the real twitter.go code from project 1.
    /// ...
    /// End of calling the twitter.go code
    t := time.Now()
    elapsed := t.Sub(start)
    // Now store elapsed time in a data structure so you can retrieve it later
    // when producing your graphs
}
// ...
}
```

Once you have all the timings, you can then use a graphing package to produce the graphs. You can perform a Google search and choose the Golang graphing package that works best for you. You could also save your timings to a file and then choose a programming language that you are more comfortable with, such as Python to be in charge of opening up the timings file and producing the graph.

You can be as creative as you wish to get the timmings for your program. There is no hard requirement that you must doing everything within Golang; therefore, feel free to use other resources to help with producing the timings and graphs. However, there are two main requirements:

1. You should provide us with test script
2. Your test script should run on a CS machine. You can remotely login to a CS machine by following the instructions in Note 14 on Piazza.

We should not have to install third-party libraries or packages to get your script running. If you are using another language with virtual environments then that would be the best solution to be able to install multiple packages. Remember you can use third party libraries in Golang because you actually download the source code so that gets compiled into the your executable; however, for other languages this is not the case. If the third-party libariries or packages are installed on a CS machine then it's fair game for you to use in your script.

Note: I would make this script as general as possible. You will need to do this same process for the next projects.

File Redirection

If you are going to be making a bash script then one option is to use file redirection for timining your program. For example, lets say you have a file called `100ops.txt` that contains 100 feed operations. You can run and time the parallel twitter program as such:

```
~ lamont$ go build twitter.go
~ lamont$ (time ./twitter 2 50) < 100ops.txt > out.txt
real        0m3.005s
user        0m0.002s
sys        0m0.003s
```

This saves all the response output to out.txt so it doesn't clutter your screen. If you want to redirect the timing out to another file then use `2> time_file`:

```
~ lamont$ go build twitter.go
~ lamont$ (time ./twitter 2 50) < 100ops.txt > out.txt 2>timing_100_1
~ lamont$ (time ./twitter 2 50) < 100ops.txt > out.txt 2>timing_100_2
~ lamont$ (time ./twitter 2 50) < 100ops.txt > out.txt 2>timing_100_3
```

Now the files `timing_100_1`, `timing_100_2`, and `timing_100_3` have the execution times after running the same command multiple times. **Make sure to include the parenthesis**.

Note: This is just one way of producing timed results. As I stated in the preliminaries section, I made a Python script to perform my timed tests, which also produced my speedup graph.

Calculating Timings

Unfortunately, running and timing a program once can give misleading timings because:

- A process may create a cache on its first execution; therefore, running faster subsequently on additional executions
- Other processes may cause the command to be starved of CPU or I/O time
- There might be random interrupt that causes the timing to be an outlier

Thus, for each timing indicated above you will actually run that timings **20 times** and take the average of those timings. Also, **make sure you close down all applications when performing your timing tests!** You want to make sure you are getting accurate results.
For this performance report, I will require that you run your final **test script** on the CS cluster. You must first get your test script working fully on your personal machine before running it on the cluster. I will post on Monday (July 27th, 2020) a script that will easily run your script on the CS cluster.

Part 2: Performance Analysis

Please submit a report (pdf document, text file, etc.) summarizing your results from the experiments and the conclusions you draw from them. Your report should include the graph as specified above and a self-contained report. That is, somebody should be able to read the report alone and understand what code you developed, what experiments you ran and how the data supports the conclusions you draw. The report **must** also include the following:

- A brief description of the problem (i.e., an explanation what you implemented in feed.go and twitter.go)

- Instructions on how to run your testing script.
- As stated previously, you need to explain the results of your graph. Based on your implementation why are you getting those results?
- Answers to the following questions:
  - What affect does the block size have on performance? How could changing the block size effect the speedup?
  - Does the type of operations being processed have any effect on the performance? For example, consider these percentages:
    - 99% of the operations to be executed are `{CONTAINS}` operations.
    - 99% of the operations to be executed are are a mixture of `{ADD}` and `{REMOVE}` operations.
  - Based on the topics we discussed in class, identify the areas in your implementation that could hypothetically see increases in performance if you were to use a different synchronization technique or improved queuing techniques. Explain why you would see those increases.