Muhammed Muktar

Project Report gRPC Application

The project includes three different components:

main/project2.go:

The file includes the dispatcher and the consolidator:

The dispatcher acts as a server listening to a client that will request jobs. The dispatcher will create jobs from a given datafile read in N bytes or less. The dispatcher will first create and put its jobs into a job queue. The dispatcher will also listen for grpc to request (RequestJob), which will attempt to fetch a job from the result queue and return the job to the client, else it will request an empty job.

The consolidator acts as a server listen to clients to push the results of a job. The consolidator will listen for a grpc to push jobs results (PushResults), which will attempt to put the results received from the client into the result queue. The consolidator also takes care of the termination of the workers and the dispatcher, carefully waiting for all workers to terminate. It achieves this by keeping track of connections from the workers using a grpc (EstablishConnection), which is used by the workers to notify the consolidator of its termination. Furthermore, the consolidator uses the number of expected jobs given by this dispatcher to track when it is finished. If it notices it received all its expected jobs, it will request a worker to terminate by returning a termination request message to the worker. It will then receive a confirmation that worker has successfully terminated, in which case once all workers terminate will close the result queue and signal to the dispatcher to close.

worker/worker.go:

The file includes the worker client:

The worker acts as a client that connects to three different servers: the dispatcher, the consolidator, and the file server. The worker will let the consolidator track its connection with the establish connection grpc. At random intervals the worker will request a job from the dispatcher server using the request job grpc. If a job is received from the dispatcher, the worker will request push the job to the file sever which will stream back the data that is relevant to the job. It will let the fileserver to read the job with the given C bytes or less at a time. The worker will then compute if the data given is a prime and will continue to count primes until the job is finished being streamed by the file server. Then the worker will push the results to the consolidator using the push result grpc. The worker receives a termination request message after pushing to the consolidator the worker will push an empty result with a termination confirmation message. The worker will also let the consolidator track its disconnection with the establish connection grpc, then finally terminate.

fileserver/fileserver.go:

The file server program acts as a sever that will stream back file data to the client. The server will listen for a get file chunk grpc, which retrieves a file segment request. The server will stream the entire file from the path given from the start index to the length requested and read the data in C bytes also given by the client.

The project also contains two different proto file that structure the file grpc calls with its messages and the job related grpc calls with its messages:

fileproto/file.proto –

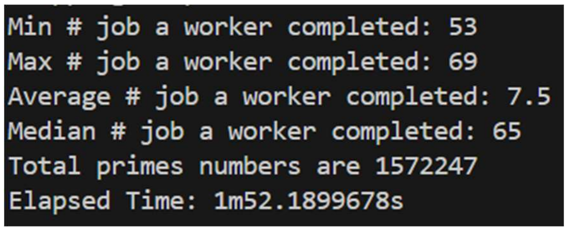
- FileSegmentRequest message, which stores information required to read into a file along with the byte to start at, the byte length to read, and the interval in which to read.
- FileData message, which is the message used to stream bytes.
- GetFileChunk rpc, which is the function call that will take a file segment request message and use that information to return a stream of file date messages.

jobproto/job –

- TerminateRequest message, which stores a Boolean that will indicate a request to terminate.
- TerminateConfirmation message, which stores a Boolean that will indicate a confirmation of termination, along with the number of completed job, which is used to store statistical data.
- Job message, which stores everything needed to represent a job (data file, start index, length of job, c value to read job)
- JobResult message, which stores a job message along with the number of primes in the respective job.
- PushInfor message, which stores the job result along with a terminate confirmation message.
- Connected message, which stores a Boolean to represent a connection to a server.
- Request job rpc, which is a function that takes no message and returns a Job
- PushResult rpc, which is a function that takes a PushInfo message and returns a TerminationRequest message to indicate if a worker should be terminated.
- EstablishConnection rpc, which is a function that takes a Connected message, indicating a client connection or disconnection and returns nothing.

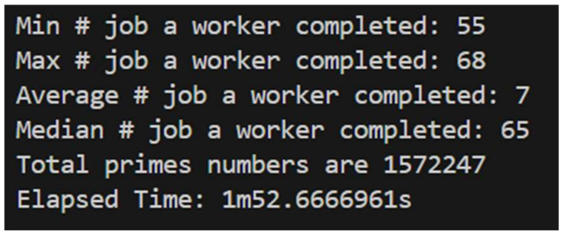
1. Using a pre-defined value for N = 1MB, C = 8KB, M=16, and a random file of 1GB

Run results # 1:



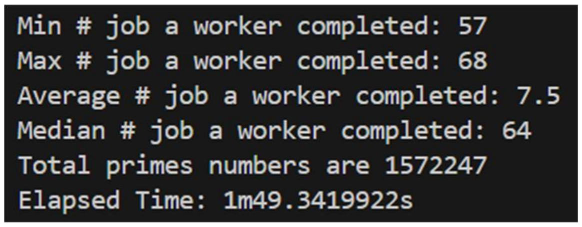
```
Min # job a worker completed: 53
Max # job a worker completed: 69
Average # job a worker completed: 7.5
Median # job a worker completed: 65
Total primes numbers are 1572247
Elapsed Time: 1m52.1899678s
```

Run results # 2:



```
Min # job a worker completed: 55
Max # job a worker completed: 68
Average # job a worker completed: 7
Median # job a worker completed: 65
Total primes numbers are 1572247
Elapsed Time: 1m52.6666961s
```

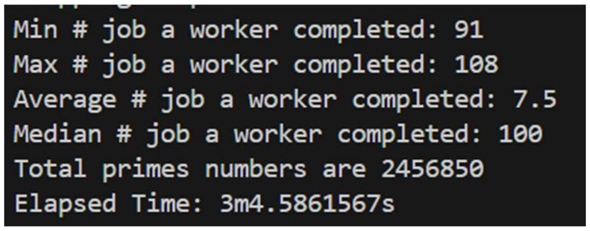
Run results # 3:



```
Min # job a worker completed: 57
Max # job a worker completed: 68
Average # job a worker completed: 7.5
Median # job a worker completed: 64
Total primes numbers are 1572247
Elapsed Time: 1m49.3419922s
```

From these results there is an average elapsed time of 1m 51.39s for a data file the size of 1GB.

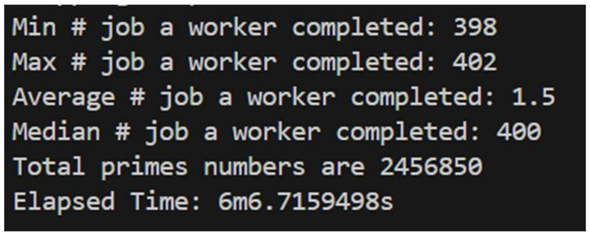
2. Using the 16 M workers, N = 1MB and C = 8KB, the largest file I was able to run is a data file the size of 1.56 GB. As seen in the screen shot below, I received an elapsed time of around 3 mins using this size.



```
Min # job a worker completed: 91
Max # job a worker completed: 108
Average # job a worker completed: 7.5
Median # job a worker completed: 100
Total primes numbers are 2456850
Elapsed Time: 3m4.5861567s
```

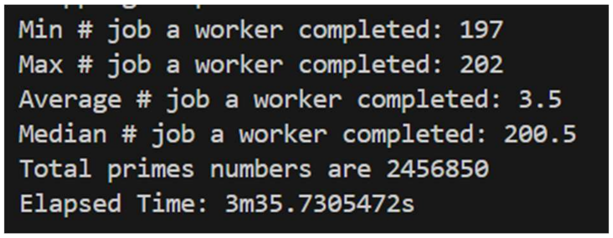
3. Using N = 1MB and C = 8KB, and a file the size of 1.56 GB

- M = 4:



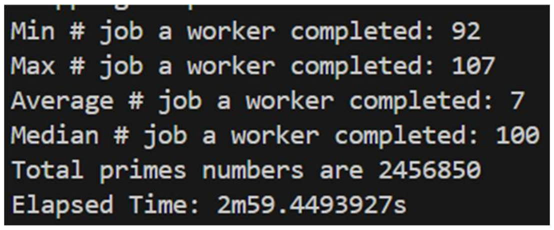
```
Min # job a worker completed: 398
Max # job a worker completed: 402
Average # job a worker completed: 1.5
Median # job a worker completed: 400
Total primes numbers are 2456850
Elapsed Time: 6m6.7159498s
```

- M = 8:



```
Min # job a worker completed: 197
Max # job a worker completed: 202
Average # job a worker completed: 3.5
Median # job a worker completed: 200.5
Total primes numbers are 2456850
Elapsed Time: 3m35.7305472s
```

- M = 16:



```
Min # job a worker completed: 92
Max # job a worker completed: 107
Average # job a worker completed: 7
Median # job a worker completed: 100
Total primes numbers are 2456850
Elapsed Time: 2m59.4493927s
```

In this test, I ran the program with 4, 8, 16 M workers. As the workers increased, the elapsed time for each session decreases every time the more workers there are. The more workers the more even the work is spread to finish the jobs. However, I noticed that as M increasing there is diminishing returns. The jump from 4 to 8 workers almost halves the elapsed time, but the jump from 8 to 16 workers does not show the same improvement. The increase of M improves elapsed time, but the rate of improvement decreases with each incrementation.