

Mini2

Basecamp Findings:

The purpose of base camp is to explore gRPC/OpenMPI and what the overall infrastructure of our actual mini may be. With that, we have some findings below.

I. gRPC protocol and how it functions

At the core of it all, gRPC is no more than remote code execution on another process. gRPC leverages HTTP/2 as the basis of its stack. A lot of the implementations, however, are abstracted away and we mainly worked with the APIs provided by gRPC's protobuf compiler. Protobuf, or protocol buffers, are definitions of parameters that are passed to and from gRPC calls. They are similar to parameters such as JSON, but with an emphasis on speed. gRPC starts with these protobuf definitions in a .proto file. Then within the .proto file, the function signature of gRPC calls are defined using prior protobuf definitions. The last step would be to invoke the proto buffer compiler to compile the definitions into a supported language. In our case, specified by the spec of Mini2, the definitions are compiled into python and C++. This compiled code effectively serves as the starting point.

To keep things simple, our gRPC is to count the number of characters in a given string. In other words, `int Count (string query)`.

First, the synchronous server was implemented along with a synchronous client. This is achieved by overriding the server definitions in C++. Then in python, we would create a client stub that can communicate with the server service.

Second, we move on to an asynchronous server. Following the example code from `grpc/examples/cpp`, there was an implementation that uses the completion queue. So once we iterate over that logic into our `Count`.

Third, according to gRPC's [best practices](#), the documentation emphasizes on using the callback API which is also implemented. The difference is that the user no longer has to manage the completion queue that manages the incoming and outgoing gRPC calls. Rather, defining a callback function is sufficient and the gRPC program handles other logic internally. We observed no real difference in communication between our server and client.

To test if there is a performance difference between the two. I performed 1000 asynchronous gRPC calls in python. The implementation is done via `asyncio` library.

```

(venv) ryantan@ryantanX1:~/Desktop/275Mini2/basecamp/client$ nvim speed_test.py
(venv) ryantan@ryantanX1:~/Desktop/275Mini2/basecamp/client$ python3 speed_test.py
testing completion queue
0.0.0.0:8765 Time for callback 14.877822 seconds
(venv) ryantan@ryantanX1:~/Desktop/275Mini2/basecamp/client$ python3 speed_test.py
testing completion queue
0.0.0.0:8765 Time for callback 17.220819 seconds
(venv) ryantan@ryantanX1:~/Desktop/275Mini2/basecamp/client$ python3 speed_test.py
testing completion queue
0.0.0.0:8765 Time for callback 12.951557 seconds
(venv) ryantan@ryantanX1:~/Desktop/275Mini2/basecamp/client$ python3 speed_test.py
testing completion queue
0.0.0.0:8765 Time for callback 12.796983 seconds
(venv) ryantan@ryantanX1:~/Desktop/275Mini2/basecamp/client$ python3 speed_test.py
testing completion queue
0.0.0.0:8765 Time for callback 12.832612 seconds
(venv) ryantan@ryantanX1:~/Desktop/275Mini2/basecamp/client$

```

Tests of 1000 count gRPC calls on completion server implementation

```

count_client.py count_pb2_grpc.py count_pb2.py count_pb2.py generate_pb2.py
(venv) ryantan@ryantanX1:~/Desktop/275Mini2/basecamp/client$ python3 speed_test.py
testing call back
0.0.0.0:8888 Time for callback 13.613386 seconds
(venv) ryantan@ryantanX1:~/Desktop/275Mini2/basecamp/client$ python3 speed_test.py
testing call back
0.0.0.0:8888 Time for callback 13.241100 seconds
(venv) ryantan@ryantanX1:~/Desktop/275Mini2/basecamp/client$ python3 speed_test.py
testing call back
0.0.0.0:8888 Time for callback 14.084975 seconds
(venv) ryantan@ryantanX1:~/Desktop/275Mini2/basecamp/client$ python3 speed_test.py
testing call back
0.0.0.0:8888 Time for callback 15.088457 seconds
(venv) ryantan@ryantanX1:~/Desktop/275Mini2/basecamp/client$ python3 speed_test.py
testing call back
0.0.0.0:8888 Time for callback 14.970191 seconds

```

Tests of 1000 count gRPC calls on callback server implementation

The finding is that there is no significant difference in performance between callback API and completion queue implementation.

II. OpenMPI

The next part of the base camp is now computing with multiple computer nodes.

1. IP assignment of machines will be required. Currently, on two machines, I assigned the main node with the server to be on 10.0.0.1, and 10.0.0.2 on the other server. This is done via the network configurations provided by the operating system UI.
2. OpenMPI will be required to be installed on both sides. Ubuntu has a package manager for it, and the running version is 4.1.6.

3. Main node must be able to SSH into other nodes. This is done via the OpenSSH package. 10.0.0.1 generates a RSA keypair. The keypair is then copied into 10.0.0.2 allowing for the computer to access. However, there was an issue we ran into with the display server protocol, wayland. So on 10.0.0.2, xhost + in the terminal command is needed to allow for execution.

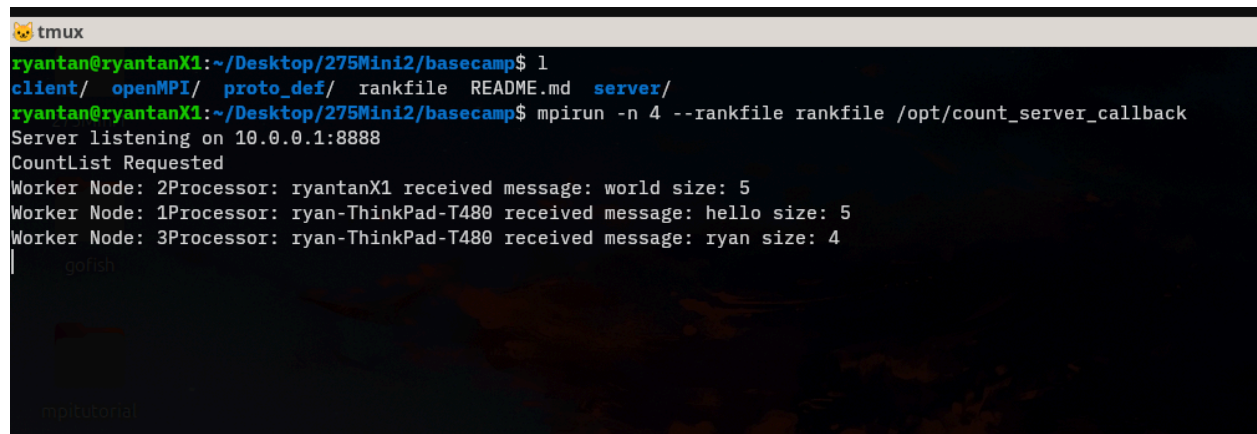
Here is an attempt of running the server on two different nodes with the current entry in the rank file.

rank 0=ryantan@10.0.0.1 slot=0

rank 1=ryan@10.0.0.2 slot=0

rank 2=ryantan@10.0.0.1 slot=1

rank 3=ryan@10.0.0.2 slot=1



```
tmux
ryantan@ryantanX1:~/Desktop/275Mini12/basecamp$ l
client/ openMPI/ proto_def/ rankfile README.md server/
ryantan@ryantanX1:~/Desktop/275Mini12/basecamp$ mpirun -n 4 --rankfile rankfile /opt/count_server_callback
Server listening on 10.0.0.1:8888
CountList Requested
Worker Node: 2Processor: ryantanX1 received message: world size: 5
Worker Node: 1Processor: ryan-ThinkPad-T480 received message: hello size: 5
Worker Node: 3Processor: ryan-ThinkPad-T480 received message: ryan size: 4
|
go!sh
mpirun
```

In this instance, ryantanX1 corresponds to a X1 carbon laptop on instance ryantan@10.0.0.1. And ryan-Thinkpad-T480 corresponds to a T480 thinkpad on instance ryan@10.0.0.2. The gRPC server is running on ryantanX1, port 8888.

A request was made to count the size of 3 words, ["hello", "world", "ryan"]. The distribution is a round robin algorithm. Rank 0 is the gRPC server / master node and 1,2, 3 are the worker nodes.

Implementation of CSV Query:

In this portion, we will be covering our implementation of a gRPC distributed server. The scope of the project will still be the same as Mini1. With the graded feedback on our Mini1, we now have data queries to access the data to extract meaningful context. We have queries to count the total number of dataset that matches with a specific borough, a range of number of passengers killed, and a range of number of passengers injured. These queries translate very naturally into functions which are defined in our protocol buffer file.

The first challenge of our implementation is sending the content of the csv through gRPC. A gRPC call should be short and concise. In our case, it isn't feasible to send the entire file over one gRPC call. The CSV has to be chunked into smaller sizes on the client. Then the final data is to be aggregated over a number of gRPC calls of chunked data. Once chunked, we make an asynchronous call to the server with the content.

Our server is of a master / workers architecture. We assign one specific node to be the master and the remaining nodes as workers. The master will assign out work to workers and consolidate once all workers are done. This is done via OpenMPI send / recv.

Our second challenge is high worker utilization, meaning that ideally, all workers should have something to do when a gRPC request comes in. We followed a scatter and gather approach. When a request comes in, the master node first identifies the length of the data. It then further chunks the data by the total number of worker nodes. Then each worker node is assigned out their chunk to perform. This way, we ensure that all worker nodes are utilized with an even workload.

A visual representation of the flow is provided below.

Clients <--gRPC--> **Server** (rank0) <--openMPI--> [rank1, rank2, rank3,]

Improvements:

Our baseline for improvement will be based on our query on the Motor Vehicle Collision CSV. We will be querying for all data types that match the query attribute borough that matches "QUEEN" and returning the count of entries that match.

Specs:

Thinkpad X1 Carbon (A) - 13th gen Intel i7, 12 cores, Ubuntu

Thinkpad 480 (B) - 8th gen Intel i5, 8 cores, Ubuntu

On the first iterations for our baseline, we are running 2 nodes, 1 master node and 1 worker nodes. 1 node on machine A, and 1 node on machine B. Our average run time rounds out to around 1 min 30 seconds per trial.

We continued to experiment with the different usage of nodes in the following format.

1 workernode@thinkpad - 00:01:30

2 workernode@thinkpad - 00:01:13

3 workernode@thinkpad - 00:01:09
4 workernode@thinkpad - 00:01:07

So far we've only increased the number of worker nodes on one machine, let's see what happens when we perform the test on multiple machines.

4 workernode@thinkpad
1 workernode@x1 - 00:00:56:76

4 workernode@thinkpad
2 workernode@x1 - 00:00:51:76

4 workernode@thinkpad
3 workernode@x1 - 00:00:47:33

4 workernode@thinkpad
4 workernode@x1 - 00:00:42:33

4 workernode@thinkpad
5 workernode@x1 - 00:00:41:33

4 workernode@thinkpad
6 workernode@x1 - 00:00:39:73

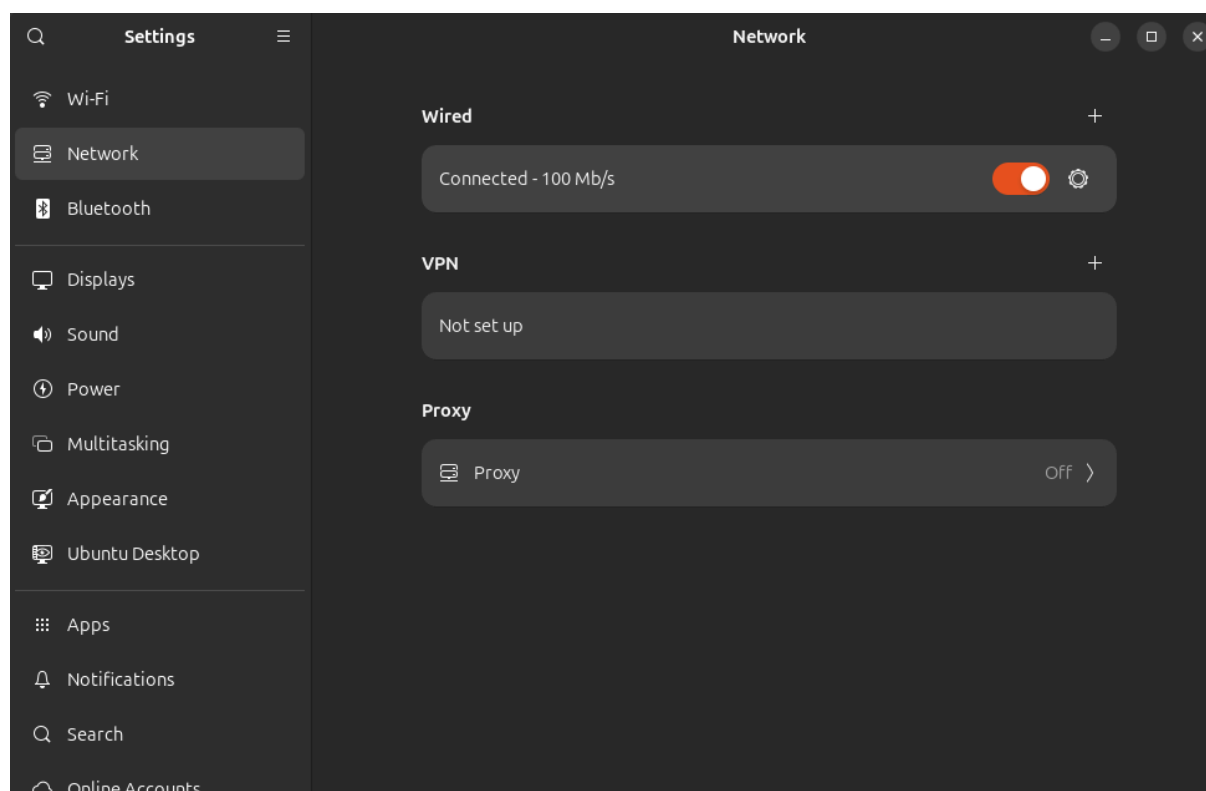
4 workernode@thinkpad
7 workernode@x1 - 00:00:37:73

4 workernode@thinkpad
8 workernode@x1 - 00:00:36:73

In our exploration of increasing the number of worker nodes, we begin to notice the effect of diminishing returns. The increase in computing power does not necessarily correlate to a linear increase in computing speed. This is in accordance with Amdahl's Law, which states that the system will be bottlenecked by other factors that aren't parallelizable. Some of the factors are latency, sequential programs, and system constraints.

We tinkered with the amount of data we can process, but

We considered an external factor, which was our hardware. X1 carbon does not have an ethernet port inherently. An adapter is required to connect X1 carbon to a LAN via ethernet. Upon investigation, the adapter rate of transfer was 100mb/s which is drastically lower performance than the Thinkpad 480 ethernet port transfer rate of 1000mb/s.



This desync in transfer is a major bottleneck in our system. Upon swapping to an adapter with a 1000mb/s transfer rate, we noticed the test time took only 28 seconds to perform. This was performed on 4 workernodes@thinkpad and 3 workernode@x1, which resulted in a 30% increase in speed from our original 48.

```
4 workernode@thinkpad
4 workernode@x1@1000mb/s      - 00:00:28:31
```

Interestingly, the increase of workersnodes@x1 with the same 1000mbs adapter saw little to no improvement from its original time of 37 seconds. We are not exactly sure why, but one potential hunch is that the skewed number of worker nodes on x1 could result in oversubscription on x1. This coupled with the fact that x1 is also the client laptop (running non-related processes) resulting in slowdowns.

```
4 workernode@thinkpad
8 workernode@x1@1000mb/s     -00:00:37:73
```

Thoughts and Takeaways (Post presentation)

This portion is written after seeing the class presentation on April 7th. It seems our direction with the project is drastically different from most of our peer's approaches. Our peer's approaches had inclusion of shared memory to store states on nodes. We couldn't help but compare our implementations with others.

The main thing we realized is the inclusion of state in the on-node shared memory architecture. In the event that multiple queries on a particular dataset are made to the server, the state saved from the first query has the potential to drastically increase any subsequent query performance if implemented. Most groups used a key-value store in memory which effectively serves as a database like Redis. The downside is synchronization is difficult to achieve using mutexes due to the distributed nature of the problem (one group discussed their program deadlocking).

In our implementation, data is transferred around in the message buffers provided by openMPI. There is no state that is stored, only the final computed results are returned. And while teams are using mutexes to manage synchronization, we are using MPI_sends and MPI_Recvs to handle synchronization. The upside of our method of synchronization is that it ensures only one node has access to the data at one time.

Ryan's note: Our implementation's contrast may be actually influenced by my understanding of shared memory. I read a golang [blog](#) recently about shared memory and their motto was. *"Do not communicate by sharing memory; instead, share memory by communicating."* Golang has goroutines (parallel computing) and channels (send/receive) for synchronization that effectively are akin to MPI_Send and MPI_Recv.