IMPLEMENTATION OF MAP REDUCE WITH ACTOR MODEL

Soroosh Zare
soroosh.zare00@gmail.com

Farzin Nasiri
farzin-nasiri@hotmail.com

ABSTRACT

Consider $\{P, M, R\}$, where P is an abstract problem, M is a function which transform initial input P' in the same class of problems as P to an intermediate list, and R is a function that merges the result of intermediate transformations to achieve the solution for a bigger problem. Usually R is called **Reduce** from our point of view. Now, given these 2 functions, we can use them to solve P even if they are black boxes, and we have no understanding of how they work. We can go further and divide the computation between different processes and provide them the function M, and finally use R on the gathered result to reach a final answer. The benefit of this solution is that the work can be done in parallel. This algorithm was first proposed by Dean & Ghemawat (2004), and is called **MapReduce**. This algorithm is often used for large data where it's not even possible to keep the whole data in memory of a single computer, and also problems where we have associative property, meaning we can merge intermediate lists in any order and still get a valid answer. The goal of this project was to implement a MapReduce framework built on top of abstractions for Map and Reduce function, and do some experiments on different scenarios. We also wanted to have support for fault-tolerance, meaning the algorithm can still achieve the desired result even if some parts of the system crash from time to time.

1 INTRODUCTION

The concept of mapping and reducing data has been almost around since the dawn of computer programming. However the **MapReduce** framework is a relatively new idea and was first proposed by Dean & Ghemawat (2004). Since then many attempts have been made to implement this framework. The most notable of all is Hadoop, a software for distributed computing, and Riak a distributed key value store. However this frameworks has its drawbacks, so many have tried to improve and build upon MapReduce, Spark is one of the most used examples.

1.1 WHEN TO USE MAP REDUCE

As we see in the following sections, MapReduce is not suitable for all scenarios. MapReduce is used for special sets of problems, that can be described using map and reduce functions, and when computation and data size can not fit on one process/computer.

1.2 HOW MAPREDUCE WORKS

In a MapReduce system, computation is done by workers, the smallest unit of computation we define in our system. Workers can be lightweight run-time processes or standalone computers. A worker usually have three roles:

1. **Map**: Each worker node applies the map function to the local data, and writes the output to a temporary storage
2. **Shuffle**: Worker nodes redistribute data based on the output keys (produced by the map function), such that all data belonging to one key is located on the same worker node
3. **Reduce**: Worker nodes now process each group of intermediate output data, per key, in parallel.

Also a master node(coordinator or main process) takes care of coordinating the system and keeping it alive. It dispatches tasks to workers and in case of any worker failures it restarts the worker or replaces it with another one.

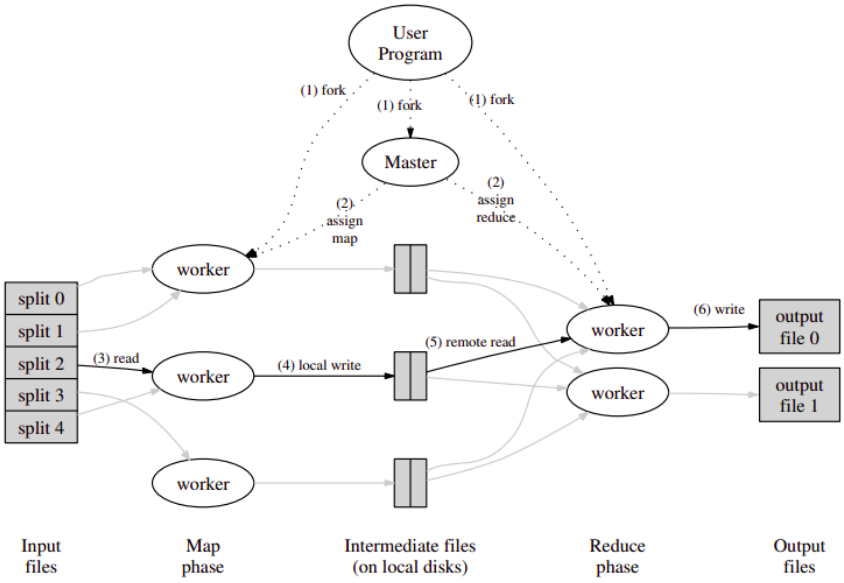

Figure 1: a complete outlook on a typical MapReduce system

2 Our implementation(Available on GitHub)

We used the Elixir programming language to implement MapReduce. We used actor model to implement the framework, meaning different components can only communicate by sending asynchronous messages, since each part of the system is isolated from every other part and has it's own memory. Elixir has out of the box support for this kind of programming, which was one of the main reason we chose it for our project. In elixir, each unit of computation is called a process (which is much more lightweight than actual OS processes), and processes can communicate with each other by sending messages. Each process has a mailbox, where the messages sent to it will eventually end up (in case they are not lost in the network). It can later on read these messages and act accordingly.

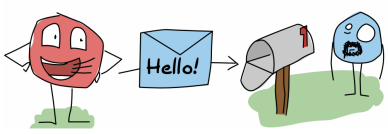

Figure 2: the red process sending a message to the blue process[Hebert (2013)]

In our implementation, each worker is a process. we also have a master process which schedules the map and reduce jobs. It simply keeps the list of currently free workers and pick a worker from this list to execute a given job (or put the job in a job queue is no worker is available at the moment). When a worker finished executing a job, it send a signal message to the master, and master put that process back in the list of available workers (or pick the first item in the job queue and passes it to the worker to be executed)

2.1 Fault Tolerance

We use the concept of supervision tree, which allow some processes to be monitored by some other processes, named supervisors. It is possible that a supervisor itself has a supervisor. When a process

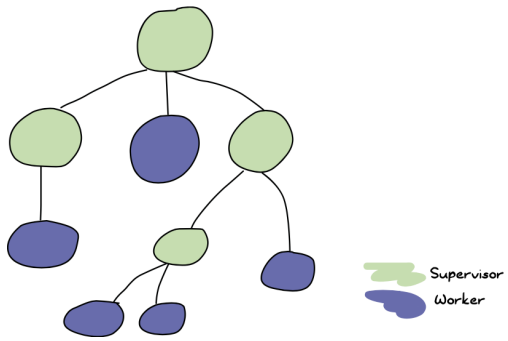

Figure 3: supervision tree[Hebert (2013)]

crashes, a crash signal will be sent to it's supervisor. The supervisor can then decide what action to take, in a way that the system will continue to work. In our case, the supervisor creates a new worker when it's informed that a worker is crashed, and reassign the jobs of the dead worker to the new worker.

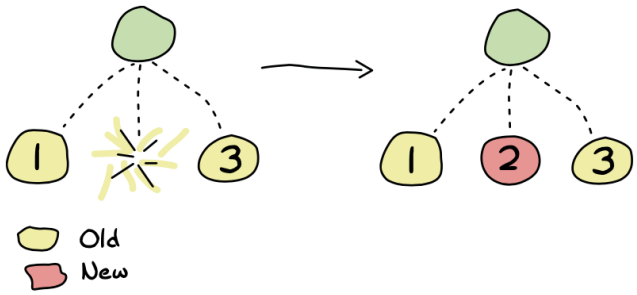

Figure 4: supervisor creating a new worker when a worker crashes[Hebert (2013)]

One other problem that we had to tackle is when a worker is alive, but appears dead because of networks delays (therefore there's no crash signal to be received by the supervisor). To tackle this problem we created a monitor process, where it will supervise all the workers, and also expects a heart beat signal from every worker in every time window of Δ seconds. if a process crashes or doesn't send a heart beat signal in time window of Δ seconds, the monitor will crash. The monitor itself is supervised by another process, and that supervisor can act accordingly and create a new worker instead of the (supposed) dead worker, and restart the monitor for the current list of workers.

```
defp monitor_heartbeats(worker_pids) do
    Enum.each(worker_pids, fn worker_pid ->
      try do
        :alive = GenServer.call(worker_pid, :heart_beat, 100)
      catch
        :exit, _ ->
          Process.exit(self(), {:heartbeat_loop_dead,
              worker_pid})
      end
    end)

    :timer.sleep(500)
    monitor_heartbeats(worker_pids)
 end
```

Listing 1: The monitor process

Another possible problem, is the situation where a worker A is supposedly dead ,but in fact alive. Suppose a process B is created to do the jobs previously assigned to A. We should have a mechanism for the master process to don't take into account multiple job responses (in this example, from processes A and B) where they actually map to the same job. To achieve this, we simply assigned a unique id to each job and made sure the response for each id is taken into account exactly once.

2.2 Data Locality

It is important to move the computations towards data, instead of moving data towards computation. The reason is simply because moving data is much more expensive, and moving computation is much more efficient.

2.3 Caching

The intermediate results from executing the map function are passed to a process called **Bucket**. Bucket can cache these results and write them on a file system once they become larger than a threshold. Once a worker needs these data for reducing purposes, it will send a message to the Bucket, asking for an specific partition. The Bucket will then give the worker all the data saved in the file system for that partition, and also the current cached buffer. The worker will merge the buffer and the file system data and will use it.

3 Final Goals & Evaluation

3.1 Benchmarking methodology and tools

Our main focus here is to measure the performance of our MapReduce implementation. By performance we mean the time it took to finish a job for some input of size n. Note that in MapReduce, contrary to many other distributed algorithms, stopping failures don't have any effect on the validity of the output. However failures can affect performance but for our purposes we did not include failures in our tests. Also to measure performance of the code we used Benchee, a benchmarking library for elixir. Finally we used the *word count* problem as our main problem so that verifying the validity of the output would be easily done with comparing the results with a valid sequential algorithm.

3.2 Datasets used for testing purposes

In the process of testing the implementation, we often needed to test in on large datasets. For this purpose, we used the Yelp dataset.

3.3 Experiments' Criteria

There are three main parameters that we can change:

1. number of workers
2. number of partitions(intermediate key-value pairs)
3. input size(number of words in the input)

3.4 Results

Unfortunately limited computation resources and time constraints prevented us to test all kinds of permutations. Although we got some mixed results too, but the following two results where good enough to reach some conclusions. Note that this results only represent this specific implementation and can not be extended to any other implementation or project.

3.4.1 Execution Time vs Number of workers

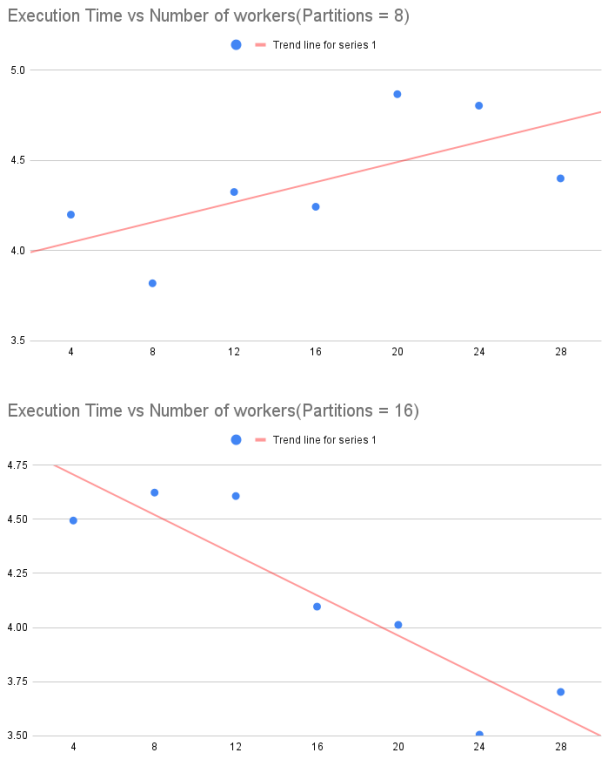

Figure 5: Execution Time vs Number of workers on a 28 core 32 GB ram system

Usually we expect a faster running time with increasing the number of workers. However, we have two different results here. In the first experiment increasing the number of workers does not result in better performance. But when we increased the number of partitions, we got the expected results (both experiments were run on a 28 core computers)

Although parallelism is an important factor and using it right can scale systems and improve their performance, but this comes with a trade off: as the number of workers increases, the overhead of context-switch between them also increases. There should be a good balance between the number of workers, number of input splits, number of partitions and our computational and storage resources.

In our first experiment we don't have a good balance, so as the number of workers increase, the overhead increases faster than the improvement caused by parallelism. Part of this is because some workers were starved and didn't have anything to do for some time.

In the second experiment, the number of partitions were enough and overhead was little so we got a boost on performance when increasing the number of workers.

3.4.2 Execution Time of MapReduce vs Sequential algorithm

Although distributed algorithms can improve the overall performance of our system in some scenarios, it is not always the case. Here we show that when the size of the input is small, the overhead of MapReduce is quit large but with increasing the input size, distributed computing makes much more sense.

In the first experiment we used a *84MB* file with 3331250 words. As you can see MapReduce performed much worse than a regular sequential algorithm.

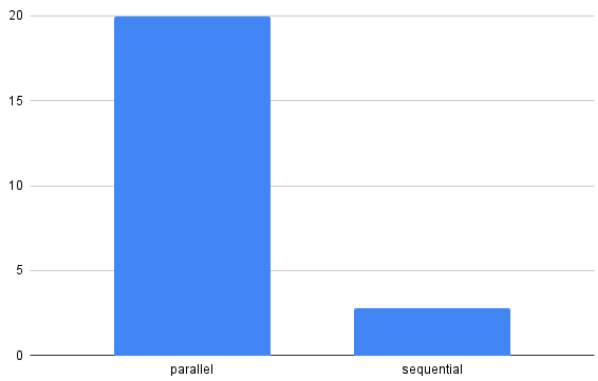

Figure 6: Sequential VS Parallel(MapReduce) execution time in seconds

Name	Iterations per Second	Average	Deviation	Median	Mode	Minimum	Maximum	Sample size
sequential	0.42	2.38 s	±14.99%	2.17 s	none	2.17 s	2.79 s	3
parallel	0.0502	19.93 s	±0.00%	19.93 s	none	19.93 s	19.93 s	1

Figure 7: Sequential VS Parallel(MapReduce) execution time details

In the second experiment we used a *6GB* file(15 times the first one). As you can see MapReduce performed much better than a regular sequential algorithm.

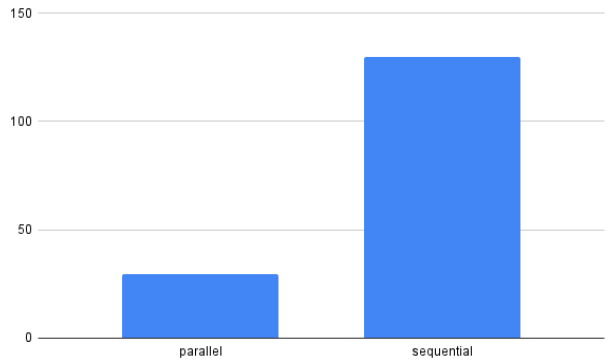

Figure 8: Sequential VS Parallel(MapReduce) execution time in minutes

Name	Iterations per Second	Average	Deviation	Median	Mode	Minimum	Maximum	Sample size
parallel	0.00057	0.49 h	±0.00%	0.49 h	none	0.49 h	0.49 h	1
sequential	0.00013	2.16 h	±0.00%	2.16 h	none	2.16 h	2.16 h	1

Figure 9: Sequential VS Parallel(MapReduce) execution time details

4 Conclusion

We conclude that although MapReduce is a powerful and simple framework, but implementing it can be quite challenging. If implemented correctly, this tool can solve many problems in a robust and fault tolerant manner. However, as we saw in our experiments, tuning the algorithm and the size of the problem are very important factors.

References

Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI'04: Sixth Symposium on Operating System Design and Implementation*, pp. 137–150, San Francisco, CA, 2004.

Fred Hebert. *Learn You Some Erlang for Great Good! A Beginner's Guide*. No Starch Press, USA, 2013. ISBN 1593274351.