Adaptive MapReduce: A More Efficient Distributed Computing Framework

Colin Wu

Abstract

Resources are inefficiently used if they are statically allocated for a program, which is a common strategy for current distributed computing framework. The static allocation also makes frameworks vulnerable to unexpected resource spike caused. The tiny change on one node in distributed applications could impact the framework in a large extent since distributed computing scales it up through its multi-node working style. It is beneficial to have an adaptive framework that has a stable resource utilization ratio through its runtime.

In this paper, we would like to present a new framework, Adaptive MapReduce, that works specifically on memory usage. The framework adapts its memory usage to the memory utilization goal in two ways: a) increase the number of processes when memory usage is low; b) decrease the number of processes when memory usage is high. This strategy raises the overall memory utilization rate and protects the framework from being killed by overusing the memory. During the first stage of this study, we developed a MapReduce framework that can work on a single machine; we measured the memory consumption of mappers and reducers for a specific task, word count; from the data obtained, we investigated the memory usage pattern of mappers and reducers. At last, we described the future plan of implementing and evaluating the Adaptive MapReduce framework.

1 Introduction

Distributed computing is becoming increasingly popular nowadays as programs need to deal with large-scale input more often, usually in thousands of Gigabytes or even more. It takes much longer time for a single computer to process large-scale data, which is impractical. Distributed computing is good for handling large-scale datasets because of its capability to spread the execution of a program to multiple machines. By starting several processes in parallel on different machines, distributed computing is able to accelerate the execution of a program drastically. Google's MapReduce [1] is one of the most widely-used applications of distributed computing. In general, MapReduce classifies one job into several phases which typically are input-dividing, mapping, combining, shuffling, and reducing. The task of processing the input is now shared with every node in the framework, hence nodes in the framework do not have to equip powerful hardware to accommodate the large-scale input. The speedup of implementing MapReduce framework is significant compared to single-machine scenarios.

However, current MapReduce implementation is reported to have several drawbacks. On one hand, it is usual for MapReduce to have unbalanced resource utilization [2]. For example, inactive memory could take up maximally 50% of the allocated memory in MapReduce framework during its execution. Additionally, previous work has shown that memory-intensive applications often overestimate their memory usage or allocate memory based on their peak memory usage [3]. This fact aggravates the resource waste in distributed computing. On the other hand, unexpected memory spike is a threat to MapReduce framework. The mapper and reducer functions could experience sudden memory increases which largely drag down the program performance [4]. Progress could be lost because the process that overuses memory on worker node is killed by the operating system. Consequently, users have to take extra actions in response to this unexpected circumstances, which is time- and resource-consuming.

Prevalent MapReduce frameworks, such as Apache Hadoop, do not allow users or the framework itself to change memory limit dynamically, making the framework very fragile especially when the memory usage for certain tasks is unpredictable. If unexpected memory spike happens on one node, the process on that node is going to be killed or the performance will be greatly influenced. In either way it hurts the overall execution of the framework. It is beneficial to have a MapReduce framework that has the flexibility to continue its execution even though the working status for some nodes are not in the best shape. Developers do not have to worry whether they have allocated enough memory for their frameworks because the framework can survive even as the memory runs out.

In contrast to the current implementation of MapReduce, we would like to present a new distributed MapReduce framework that can dynamically adjust its memory usage on every node in runtime. Specifically, for each node in the framework, they will keep a high percent of overall memory usage in real time, at the meantime avoiding any progress being lost accidentally. Based on the description above, the new framework should be able to perform these two tasks:

- Decrease the memory usage as the current memory usage hits a certain threshold;
- Increase the memory usage as the current memory usage percent is lower than the expected value.

The article will describe the means in which our framework increases or decreases the memory usage in section 3.

We conducted experiments for better learning the memory usage pattern. We gathered experimental data regarding the peak memory usage and the real-time memory usage of mappers and reducers for a specific MapReduce task, word count. By interpreting the collected data, we obtained preliminary knowledge concerning the memory usage pattern for mappers and reducers. We came to the conclusion that the pattern is predictable for word count task due to its simplicity. Moreover, we include a future plan illustrating the implementation and evaluation of the Adaptive MapReduce framework.

This paper is structured as follows. Section 2 will discuss the design details of the Adaptive MapReduce framework. Section 3 will explain the implementation of our MapReduce framework in current stage. Section 4 will present the tentative results.

2 System Design/Architecture

2.1 System Overview

The Adaptive MapReduce framework is going to have the following properties:

- As the framework receives the input, it will divide it into very small-scale pieces. Every input piece will have a unique identification so that they can be identified by the scheduler;
- In the framework, there are several worker nodes working simultaneously. Each worker node has multiple running mappers or reducers, and the number of processes is controlled by the scheduler;
- The master node controls how many worker nodes are in the framework but also keeps track of each input piece to make sure every input piece is executed.

In the following sections, we will discuss some details in the implementation of the new framework.

2.2 Input Division

The framework divides initial input into small pieces so that every worker node can finish executing the small input in a relatively short time. If a worker node has less memory available to continue executing the currently running programs (mappers or reducers), the worker node will kill the program that starts the latest and send the corresponding input piece to another available worker node. The framework chooses to kill the process instead of recover the process on other nodes because snapshotting a process takes too much extra work, which will slow down the overall performance unavoidably, while transmitting input files inside the network is much easier to achieve based on the existing technology. Hence, it is better off for the current framework not to recover processes.

The size of each small input piece depends on the hardware and user-defined algorithms. Because of the uncertainty of mapper and reducer algorithm, the intuitive idea is that the framework will run experiments on different sizes of input in a sand-box environment where the memory usage and execution time will be recorded. The framework then will decide which input size to use for the specific task. Similar strategies are mentioned by others' work [5]. The goal is that one worker node should be capable of processing at least one input piece at a time.

2.3 Scheduler

The scheduler on each worker node is an essential component of the new framework. Schedulers decide how many processes are running on the node and records the status of every input piece that it has processed, which is shared with every other worker node and master node. More importantly, a scheduler keeps track of the real-time memory usage of the worker node. A scheduler follows the rule specified below to manage the number of processes on one node. This is inspired by an algorithm in TCP congestion control: additive increase/multiplicative decrease.

If the memory usage hits the threshold, often set below the memory limit of the operating system, it will kill the process that starts the latest so as to release its memory consumption; if the memory usage is less than the threshold, meaning the vacant memory is big enough, it will start a new mapper or reducer process.

In a word, the scheduler will increase the total memory usage when enough vacant memory exists, it will decrease the total memory usage when the node runs out of memory.

2.4 Discussions

The proposed framework still needs improvement. For instance, the amount of "wasted processes" is uncontrollable. The "wasted processes" is defined as the processes that are killed because of the memory shortage. In the new framework, the "wasted processes" are recovered on other worker node and are not guaranteed to be successfully executed on other nodes either. One input piece may have to pass from one node to another several times before it is executed successfully, which may cause significant resource waste. The determination of input size needs to be polished as well. This strategy should work collaboratively to decrease the amount of "wasted processes". The framework should find a point where the resource is utilized in maximum (consider execution time and memory usage together).

2.5 Previous Works

Before this framework was proposed, we tried to investigate if it is practical to snapshot the process and recover it on other nodes, but it ended up with too much extra work and got abandoned. Process recovering requires the framework to take snapshot of the process that is going to be killed and saves the snapshot temporarily on the disk, which will slow down the execution speed greatly. No feasible solutions or strategies have been found in this field so far.

We also thought about determining the number of processes based on its peak memory. In this way, the framework does pre-experiments to obtain the peak memory usage, with which the framework can decide how many processes are able to run at the same time without exceeding the memory limit. Yet, this framework is not that ideal because mappers and reducers tend to consume less memory at the beginning and at the end of its execution, and we cannot assure that the framework has the knowledge of the memory limit beforehand. These narrow down the application of the framework. If the number of processes is fixed in runtime, then the total memory will not be used efficiently in certain time periods.

3 Implementation

We developed two frameworks so far. The MapReduce framework used in the experiments is coded in Python and guided by the instruction in Katsov's article [6]. A memory monitor, also coded in Python, is created in order to collect memory usage data in runtime. The memory monitor is designed to run on a Linux OS with permission to access `cgroups` and `proc` folder.

3.1 MapReduce Framework

The current MapReduce framework is able to accomplish simple tasks, such as word count, without any difficulties. Functions in the framework communicate via text files. The framework is designed to run in a single-machine scenario, which contains the following procedures:

- Input division: `divide_input();`
- Mapping: `mapper();`
- Combining: `combiner();`
- Shuffling: `shuffle();`
- Reducing: `reducer();`
- Integrating reducing results: `adder().`

This framework can only work on one machine, it creates multiple child processes using `multiprocessing` class in Python to simulate distributed computing. On one single machine, one core can be viewed as a worker node in distributed computing network. The framework was tested against different tasks, such as char count (to find how many words are in the same length in a text file), and the framework was proved to be general enough to finish different kinds of tasks.

In order to use this framework, users should first have their mapper, reducer, and combiner functions organized in one Python file. Then, users need to put the source file of the MapReduce framework in the same folder where their own functions are located. The framework can be called by the users as the framework is imported in the header of their function files. The framework currently has six pass-in parameters: the first and second parameters are the function pointers to mapper and reducer functions; the third parameter is the directory to the input file; the forth and fifth parameters are the number of mappers and reducers users would like to have in the execution; and the final parameter is the function pointer to a combiner function, which is optional.

3.2 Memory Monitor Framework

The central idea of the memory monitor is to create two threads running at the same time: one to execute MapReduce framework, and another to access certain fields which contain memory usage information and then record the memory usage. The memory monitor can measure both the peak memory usage and the real-time memory usage of a process.

3.2.1 Measurement of Peak Memory Usage

The memory monitor framework takes advantage of the control groups (`cgroups`) in Linux OS. The control groups can record the total memory usage of a process, including the forked or child processes it creates, according to its official documentation [7], and it is also helpful in later research because developers can customize the environment where a program is running, such as the memory limit, by using control groups. The framework uses one variable in the control groups called `memory.memsw.max_usage_in_bytes`. To create a control group, one should call `cgcreate`; to call the customized control group when executing, one should have `cgexec` in the command. It is recommended to write these commands in a shell script, and a sample script is available in the Github repository (available at https://github.com/rssys/dynamic-lambda).

3.2.2 Measurement of Real-Time Memory Usage

The memory usage of a single process is recorded in `proc/pid` folder. The memory monitor will access the memory data every 0.02 second and write it to a text file with the corresponding time stamp. This strategy works fine with one single process but is difficult to be applied to a framework which has several processes. With the memory record and time stamp, one can draw the real-time memory usage graph to learn the memory usage pattern for a specific task.

3.3 Implementation Plan

Working on the single-machine scenario is not enough. The current framework needs to be distributed to other machines in order to simulate the real-world application of serverless computing. This motivation leads us to look for the existing implementations, like Apache Hadoop. Apache Hadoop is an open-source MapReduce platform released in 2006. It is regularly maintained by a group of developers, making it an appropriate tool for this study. Apache Hadoop v2.9.2 released in November 2018 is going to be used in future study because of its stability and compatibility. Apache Hadoop can work on single machine or be distributed on a distributed network.

In response to the existing problems of current study, I plan to address these aspects in the future:

- Measure memory usage in Hadoop. The health status of a worker node is accessible in Hadoop framework. Users can do a simple health check when the framework is running, and Hadoop will generate a report concerning the health status of each worker node. This is a feature that can be used in future implementation.

- Implement schedulers on each client in Hadoop. It requires modifications of the source code. As an added feature of the original Hadoop framework, the scheduler needs to be implemented in Hadoop's client algorithm, hence new content will be added to the folder: `hadoop-mapreduce-client`.

- Test modified framework against different workloads. Interpreting experimental data collected from various workloads generalizes the final conclusion. Since current data is from only one workload, the result may not be applicable to other situations.

4 Evaluation

The experiment was conducted on Ubuntu v20.04 (carried by VirtualBox v6.1). The virtual machine has a 4-core Intel i5 CPU and 4-GB RAM capacity. The framework was running word count in the experiment. The memory usage of the MapReduce framework was tested with distinct input size and number of mappers and reducers.

- Input size: 0.5 MB, 11.7 MB, 23.6 MB, 47.1 MB, and 94.2 MB;

- Number of mappers and reducers: 1, 2, 5, 10, 20, 50, 100, 200, and 250

In the experiment, the number of mappers and that of reducers always stay the same.

Figure 1 and 2 summarize the peak memory usage pattern. The x-axis stands for the input size for the specific function in MB, and the y-axix stands for the peak memory usage in MB. Figure 3 and 4 summarize the real-time memory usage pattern. The x-axis stands for the runtime in second, and the y-axis stands for real-time memory usage in MB.

4.1 Peak Memory Usage Patterns

The peak memory usage patterns for a mapper and reducer are summarized in the following graphs.

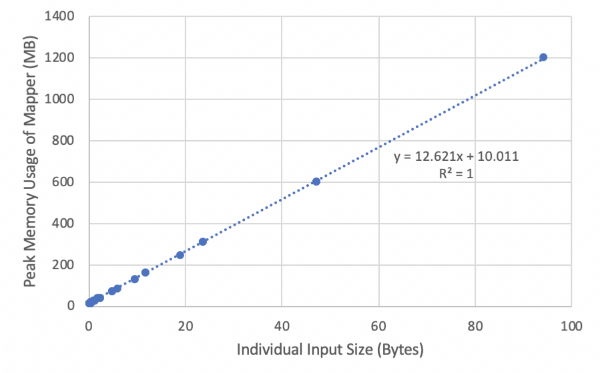

Figure 1: Peak memory usage pattern for mapper (word count)

We can observe a very strong linear relationship between peak memory usage of a mapper and its input size. For word count, the peak memory usage of a mapper is very predictable, at least for the input size less than 100 MB. When we know the input size for a mapper, we can compute its peak memory usage by substituting it to the linear regression function shown in Figure 1.

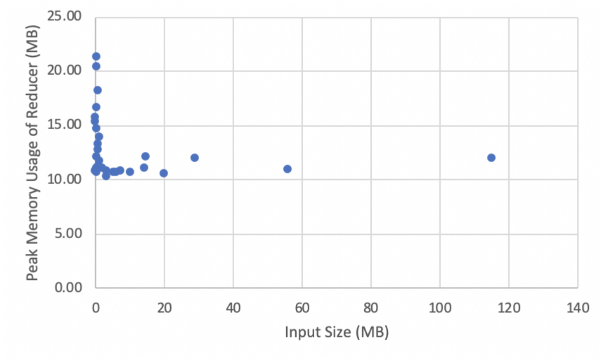

Figure 2: Peak memory usage pattern for reducer (word count)

We can observe that a flat line in Figure 2, which is a distinct behavior from that of mappers. The peak memory usage for a reducer remains as the input size increases. The inconsistent of values when input size is fairly small may be caused by the overhead of context switching. Small input size usually means the number of reducers is very high, and the context switch will occur frequently on each core as there are too many processes in the system.

4.2 Real-Time Memory Usage Patters

The memory monitor framework measures the real-time memory usage simultaneously. The experiment setting is almost the same as the previous experiment. The only difference is that the input size is fixed to 94.2 MB and the number of mappers and reducers is fixed to 1.

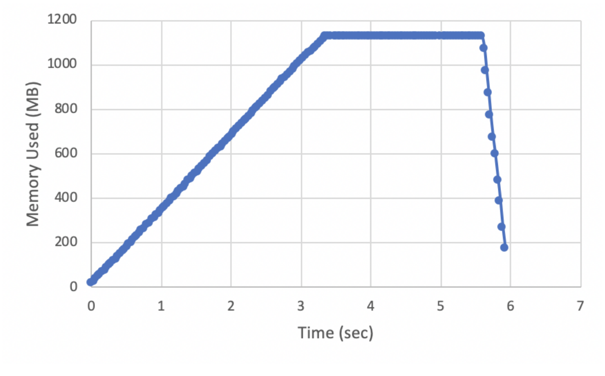

Figure 3: Real-time memory usage for mapper (word count)

The staging for real-time memory usage of mappers is very clear. It can be divided into three stages:

- Reading: mapper reads input word by word to generate key-value pair and store it in a local variable. Hence the memory usage climbs up gradually.
- Writing: mapper writes content (key-value pairs) stored in the local variable to an output file (text file). The memory usage will not increase at this time because the writing operation does not consume extra memory.
- Cleaning-up: After all information is written to the output file, the program will end soon. Python will free any allocated memory and the memory usage deceases rapidly.

This pattern is actually hard to achieve in real-world applications since real-world tasks are commonly more complex than word count.

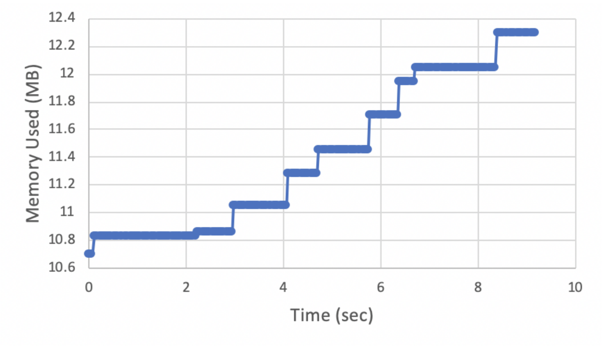

Figure 4: Real-time memory usage for reducer (word count)

The shape of the real-time memory usage for reducers looks similar to "step function" in mathematics: memory usage increases suddenly and stays constant for a while. Given that the range for the y-axis is rather small compared to that for mappers, only ranging from 10.6 MB to 12.4 MB, this graph is more like a magnifier of the reading stage in mappers, which reflects the way in which Python enlarges the size of a local variable. It doubles the size of local variable every time the variable is full, hence there are memory increases in a certain time period.

4.3 Discussions

In the peak memory usage graph for a reducer, we did not acquire enough number of data points when the input size is greater than 20 MB. As a result, we need more data points in that region to concrete the validity of our conclusion. Also, we will test various kinds of workloads in the future to learn if the memory usage of mappers or reducers are predictable. So far, for simple workloads, such as word count, we did observe a predictable memory usage pattern. However, the observation could be vary as the complexity of workloads increases. Lastly, we need to run the same experiment (peak memory test and real-time memory test) on real-world applications. We decide to use Apache Hadoop for our future study because of its various functions and outstanding stability. From the data obtained, we will generate peak memory usage versus input size graph and real-time memory usage versus runtime graph, which help us get a clearer view of the memory consumption of Hadoop. We can further modify our framework to better serve our research purposes, according to the graphs.

References

[1] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI'04: Sixth Symposium on Operating System Design and Implementation*, pages 137–150, San Francisco, CA, 2004. https://research.google/pubs/pub62/.

[2] A. Uta, A. Oprescu, and T. Kielmann. Towards resource disaggregation — memory scavenging for scientific workloads. In *2016 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 100–109, 2016. https://ieeexplore.ieee.org/document/7776483.

[3] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G. Shin. Efficient memory disaggregation with infiniswap. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 649–667, Boston, MA, March 2017. USENIX Association. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/gu.

[4] Ganesh Ananthanarayanan, Srikanth Kandula, Albert Greenberg, Ion Stoica, Yi Lu, Bikas Saha, and Edward Harris. Reining in the outliers in map-reduce clusters using mantri. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, page 265–278, USA, 2010. USENIX Association. https://dl.acm.org/doi/10.5555/1924943.1924962.

[5] Jordà Polo, Claris Castillo, David Carrera, Yolanda Becerra, Ian Whalley, Malgorzata Steinder, Jordi Torres, and Eduard Ayguadé. Resource-aware adaptive scheduling for mapreduce clusters. In Fabio Kon and Anne-Marie Kermarrec, editors, *Middleware 2011*, pages 187–207, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. https://link.springer.com/chapter/10.1007/978-3-642-25821-3_10.

[6] Ilya Katsov. Mapreduce patterns, algorithms, and use cases. 2012. https://highlyscalable.wordpress.com/2012/02/01/mapreduce-patterns/.

[7] Paul Menage. Cgroups. https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt.