Apache Hadoop: Memory and Resource Management

Colin Wu, Purdue University

Abstract

Big data is the new trend in the world of IT, and serverless/distributed computing has been applied to process and handle huge amount of data. The efficiency is the key criterion to distinguish an outstanding framework from others. An efficient framework can save users not only time but also cost. Apache Hadoop is a prevalent serverless computing framework that implements MapReduce to further speed up its processing speed on large-scale datasets. In this paper, the author first examined the overall memory utilization of Hadoop. The result shows that Hadoop well controls its memory usage based on the available memory of the OS. Then, the author also illustrates the cause of the memory fluctuation in real-time memory usage graph of Hadoop. By reading others' works, the author points out that the resource allocated for each Container is static and inflexible. At the end, the author analyzes the memory consumption data of each Container specifically and discovers that the memory is not used efficiently for large Container sizes.

1 Introduction

Apache Hadoop[1] is a prevalent serverless computing framework that implements MapReduce to further speed up its processing speed on large-scale datasets. Hadoop clusters can execute tasks, which used to take days and months to finish, in minutes or even seconds. However, researchers have found that the resource utilization of Hadoop is problematic and the rigid job scheduling method is dragging down the performance of the framework. For example, Yao et al.[2] noticed that the current scheduling algorithm of Hadoop does not yield the optimal resource arrangement; Alanazi et al.[3] put forward a novel strategy to configure the parameters of Hadoop to achieve a better performance. Since users usually deploy the framework on commercial cloud computing platforms, such as Google Cloud Platform (GCP) and Amazon Web Services (AWS), they have to pay money for the resource used by the framework. If the resource is not utilized efficiently, meaning some resources have been wasted in the execution, the cost of using Hadoop increases, and it makes the framework less attracting and promising.

In the first place, the author conducts experiments to visualize the overall memory utilization of Hadoop, and the data indicates the framework is using memory in an efficient manner: the framework uses all memory available one the system level. Then, the author realizes that Hadoop is using Containers to manage its job scheduling and resource allocation, from the previous works. Experiments that look inside the Container are designed and conducted. The author finds out that the resource allocation for each Container is static. More importantly, the resource is not utilized efficiently for each Container: even though resource is allocated for each Container, they may not use all of them to process one task in runtime.

The article contains the following segments: Section 2 provides background information about Apache Hadoop and MapReduce; Section 3 presents the experiment design, the experimental results, and the interpretations; Section 4 is the summary of related work; and Section 5 details the conclusion of this article and the future plan of this project.

2 Background

Apache Hadoop was first released in April 2014, and it has already involved to its 3rd generation. Apache Hadoop is a project coded in Java, and it "develops open-source software for reliable, scalable, distributed computing", per its official website [1]. The transition from Hadoop 1 to Hadoop 2 drastically improved the efficiency of Hadoop thanks to the introduction of Hadoop YARN (Yet Another Resource Negotiator). The resource arrangement became more manageable and reasonable after YARN was implemented in the framework. Hadoop 3 had another powerful tool added in the framework: containers. The addition of containers further refined the resource management of Hadoop.

There are four main segments in Hadoop 3:

- Hadoop Distributed File System (HDFS),
- Yet Another Resource Negotiator (YARN),
- MapReduce,
- Hadoop Common.

HDFS is where inputs, output, and intermediate data are stored. The files in HDFS are divided into "blocks", and the block size can be customized in the configuration file. Each block serves as an input for a mapper/reducer. HDFS will spawn several daemons while the framework

is executing tasks. They are NameNode, DataNode, and SecondaryNameNode, respectively. As HDFS is built on a master/slave architecture, NameNode is the master of DataNode. NameNode stores the metadata of all inputs, while every node in the cluster has a DataNode running locally. DataNode is where the data on each node stores, and it will send reports to NameNode regularly concerning the data storage on the node. If one node does not have enough space for executing another task, the NodeManager will send inputs to other nodes in the cluster. Secondary NameNode is a backup of NameNode in case NameNode is down for some reasons.

YARN negotiates resources across different nodes and applications. ResourceManager is the daemon where most of the resource negotiation takes place. It receives updates from NodeManager, which is a per-node daemon to monitor resource utilization on each node and marks those nodes without enough resource as "unhealthy". The tasks on unhealthy nodes will be put on hold unless the condition of that node mitigates. ApplicationMaster is another important daemon in YARN. It is spawned for a specific application, and there could be multiple ApplicationMasters running on one node. It monitors the resource utilization of all processes for a specific application and sends update to ResourceManager.

MapReduce module contains the basic algorithms for performing MapReduce. Basically key/value pairs are generated in the mapping stage, and the reducing stage will use the intermediate key/value pairs to compute the final result. Hadoop Common provides useful tools and Java libraries for the regular operations of Hadoop.

Besides, Hadoop supports three modes: standalone, pseudo-distributed, and fully-distributed. Each mode is designed for unique purposes. The standalone mode is the fastest mode because the framework is not distributed to a cluster of nodes, but the computing power is greatly limited consequently. The pseudo-distributed mode only supports Hadoop running on one node, simulating the situation where Hadoop is distributed on different machines. It is suitable for developers to debug and test the project because the source code can be assembled fairly fast under this mode. The fully-distributed mode is the most commonly-used mode. It will distribute the project to a cluster of nodes.

3 Evaluation

3.1 Overall Memory Utilization of Hadoop

The first question raised for the memory utilization of Hadoop is: How well does Hadoop perform when the input size is relatively big? Memory usage of Hadoop should be visualized so as to answer this question.

As mentioned in previous section, NodeManager daemon is the monitor of the resource utilization of a node. `NodeResourceMonitorImpl.java` is the file in which NodeManager is implemented. In that file, `pmem` (line 146) and `vmem` (line 148) are the variables where the real-time physical and virtual memory usage of the framework are stored. According to `SysInfoLinux.java`, the overall memory usage of Hadoop is acquired from `/proc/meminfo`, which is a common place to look up memory usage of a particular process per its pid. In order to display the values of `pmem` and `vmem`, one can simply use `println()` method, and the result will be recorded in the `.out` file of NodeManager (which can be found in `logs` folder).

For this experiment, Hadoop is installed on a 64-bit Ubuntu 20.04.1 virtual machine (8 GB memory, 4 cores). The version of Hadoop is 3.3.0, and it is running on the pseudo-distributed mode. By modifying `monitoringInterval`, the frequency of the monitoring method being called can be adjusted. In this experiment, the time interval is set to 1 sec. In order to draw a more general conclusion, Hadoop is tested against inputs with three different sizes: 150 MB, 1,500 MB, and 3,000 MB. The framework is working on solving a word count task. The result is presented in Figure 1.

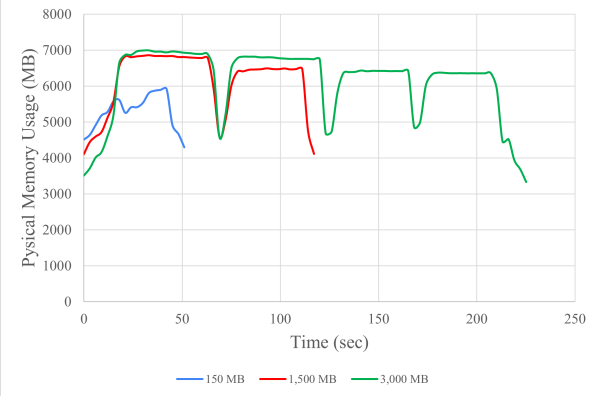

Figure 1: Real-Time physical memory usage for Hadoop

Clearly, none of the above inputs break the framework. The maximum usage of Hadoop (7 GB) is the total memory available for the virtual machine (8 GB) minus the memory allocated for the operating system (1 GB) for inputs with different sizes. The overall memory utilization ratio is nearly 100%, which is very outstanding. By closely examining the real-time physical memory usage for large input sizes (1,500 MB and 3,000 MB), it is easy to identify that the memory usage of Hadoop is not a flat line throughout the runtime. For example, for 3,000 MB input, the memory usage drops at 60, 125, and 175 sec, and recovers to normal level shortly.

3.2 Containers

From the interpretation of the real-time memory usage graph, the next question coming up is: What happened exactly when the memory usage fluctuates at some certain times?

This question leads to the mechanism Hadoop implements to manage its job scheduling and resource. It turns out Hadoop is using "containers" to optimize its resource management. In Hadoop, a container is an environment where a process (mapper/reducer) can run, and it can be initialized on any nodes in the cluster. While a container is being initialized, a certain amount of resource will be allocated to it based on the value of `mapreduce.map.memory.mb` and `mapreduce.reduce.memory.mb` (default value is used if they are not specified in `mapred-site.xml`). For a certain application, the ApplicationMaster will manage the number of containers according to the available resource on the node.

The log of NodeManager further proofs that a container is the basic unit of Hadoop resource management. The memory usage graph of 1,500 MB input will be taken as an example to illustrate how the lifetime of a container results in the fluctuation of memory usage.

For this experiment, the container which carries ApplicationMaster is allocated 2 GB memory and 1 vCore. The rest of containers all have 1 GB memory and 1 vCore, which is the default setting for resource allocation of containers. The first container (referred as "Container 1") spawned is the ApplicationMaster container which negotiates resource with NodeManager and does not execute any tasks. Container 1 is killed only when the job is done, which means Container 1 will be active throughout the runtime. Other containers are worker containers which executes either the mapping task or the reducing task. In this experiment specifically, the job is finished by 13 containers. Container 10 is the reducer, and Container 2 - 9 and 11 - 13 are mappers. If Container 1 takes up 2 GB memory and the total memory available is 8 GB for the OS, then the maximum number of containers that can be running at the same time is 7 (including Container 1).

This is the explanation of how containers result in the memory fluctuation. At 69 sec, the memory usage starts to drop. According to the log of NodeManager, this is when the process inside Container 5 finishes, and the resource allocated to Container 5 starts being released. Consequently, Container 8 (the first container in the second wave) is initializing. Since the initializing process takes longer than the releasing process, the total memory usage is dropping. One second after, Container 6 and 7 are done. At 72 sec, another 2 containers (Container 3 and 4) finish, and Container 9 - 12 start being initialized. At 75 sec, Container 2 finish and Container 13 starts being initialized. After that, all containers spawned in the first wave (Container 2 - 7) are all finished, and the second wave of containers are all initialized, hence the overall memory usage goes up again.

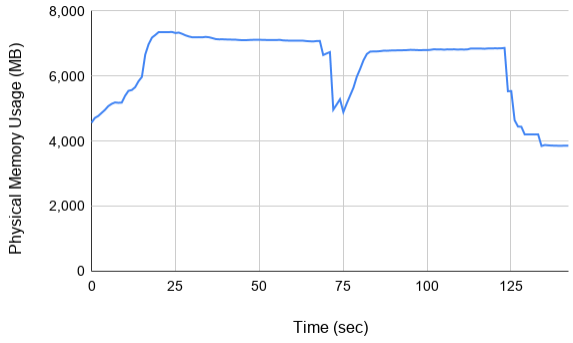

Figure 2: Real-Time memory usage of 1,500 MB input

3.3 Memory Usage of a Container

Unfortunately, the resource allocated for each container is static [4], meaning if the process running inside the container needs more resource than allocated, there is no means in Hadoop to bring more resource from outside. As a result, this container has to be killed, and then the job cannot be finished. Users have to shut down the job and increase the container size in the configuration file: `yarn-site.xml` manually, which is inconvenient in practice. In this section, the authors will prove that the resource allocated to a container is not used efficiently.

3.3.1 Execution Time of Different Container Sizes

First of all, the container size is the only factor inconsistent in the following experiment, and all other factors are controlled as the same. The experiment uses default block size - 128 MB. In this way, a 1,024 MB input will be split into 11 pieces while stored in HDFS. In other words, at least 13 containers (11 mappers + 1 reducer + 1 ApplicationMaster) in total are needed to successfully process the input. For different container sizes, the input size will be adjusted in order to make the number of input splits the same. Besides, the number of containers that are running simultaneously ought to be the same for different input sizes so that the affect of parallelism is eliminated. The way to achieve this goal is to limit the amount of memory available for the framework for different container sizes.

The result is summarized in Table 1. It can be observed that the difference between the execution time of 1024 MB and 256 MB is not statistically significant, and

Table 1: Execution time of different Container sizes

Container Size (MB)	No. of Input Splits	No. of Containers Running Simultaneously	Execution Time (sec)
1,024	11	3	212
512	11	3	218
256	11	3	218
128	11	N/A	N/A

the job cannot be finished when the container size is 128 MB, which makes sense, because the block size in HDFS is 128 MB. The container size should at least be greater than the block size. As a conclusion, larger container size does not significantly shorten the execution time for the same task. On the other hand, it proves that some memory allocated for large container remains vacant in runtime. If all resource is utilized for large input sizes, there should be a significant difference observed.

3.3.2 Statistical Analysis Memory Usage of a Container

It is worthy to get the memory usage data for a specific container to verify the interpretation generated from the last experiment. The experiment concerning the memory usage for a container is also conducted. In order to find the memory used by a container, one should go to `ContainerMonitorImpl.java` and print out the values of `containerId`, `currentPmemUsage`, and `currentVmemUsage` periodically.

For container size 1,024 MB, the mapper (Container 2) only uses around 50% of the allocated memory, which is about 500 MB, and the reducer (Container 10) only uses 25% of the allocated memory. The memory usage patterns for mapper and reducer match with the results from the previous report (for Summer research). Clearly, resource is not completely utilized for container size 1,024 MB, and this is part of the reasons why its execution time does not stand out from others.

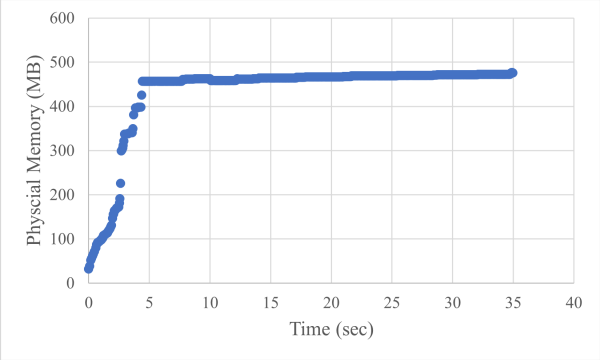

Figure 3: Memory usage of Container 2 (input size: 1,024 MB)

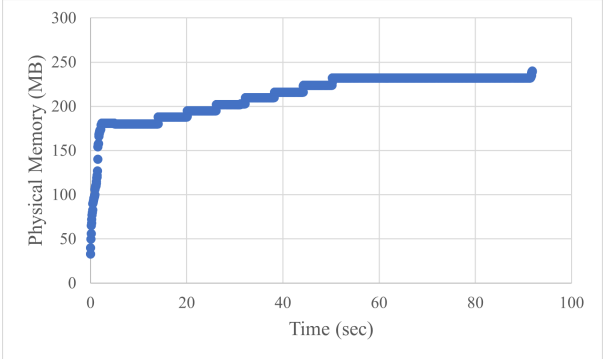

Figure 4: Memory usage of Container 10 (input size: 1,024 MB)

When the container size reduces 50% to 512 MB, the memory utilization ratio goes up. The memory utilization ratio for mappers goes up to 70%, and the reducer goes up to 50%. The actual memory consumed by mappers actually decreases, which explains the phenomenon that the execution time for 512 MB input increases a bit. It is safe to predict if the available memory for the framework is not limited, then the execution time for 512 MB container is going to be faster than 1,024 MB container.

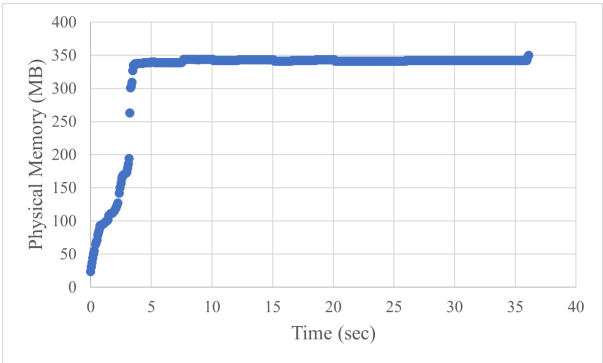

Figure 5: Memory usage of Container 2 (input size: 512 MB)

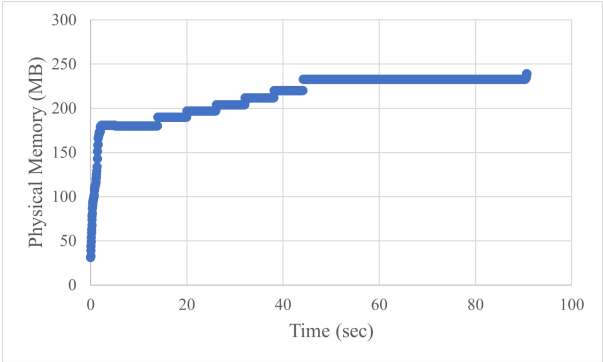

Figure 6: Memory usage of Container 10 (input size: 512 MB)

As the container size decreases to 256 MB, it is observed that the mapper actually uses all the memory allocated to it. The memory utilization ratio reaches 100% for mappers. And the memory utilization ratio also increases to 70%, which is also a huge improvement compared to the container size 1,024 MB.

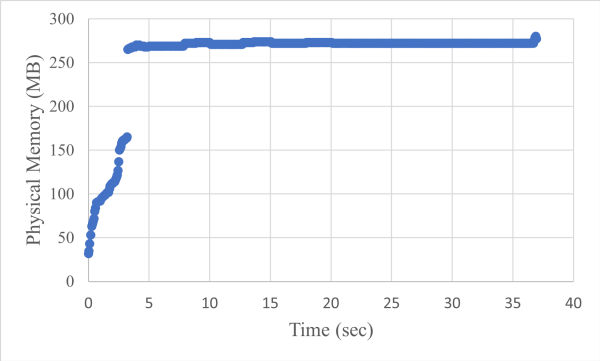

Figure 7: Memory usage of Container 2 (input size: 256 MB)

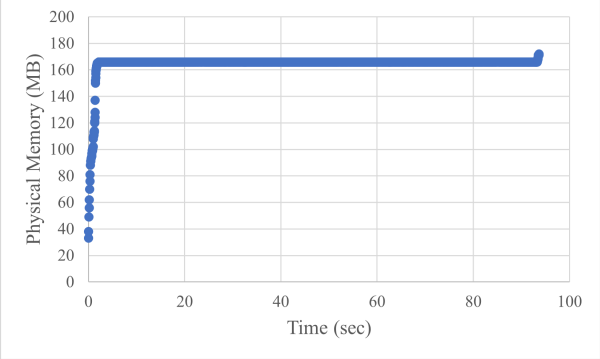

Figure 8: Memory usage of Container 10 (input size: 256 MB)

4 Related Work

The introduction of YARN improves the resource management of Hadoop to another level, however it is still possible for containers to waste memory and other resource when the task is executing.

Several previous studies have pointed out the rigid way in which YARN spawns containers. Guo et al.[4] state that YARN only allows different kinds of jobs to have distinct container sizes. On other words, the container size for one job has to be the same. If one container failed because of the lack of resource, the whole job could not be finished. Trying to resolve this problem, he authors proposed a new strategy. Instead of mitigating skew among different tasks, they try to balance the processing time of each task by adding virtual nodes to the cluster.

There are previous works focusing on optimizing the parameter configuration of Hadoop. As mentioned before, a number of Hadoop properties, such as container size and block size, are manually configured by the users, however new users are struggling finding out the optimal configuration for their jobs. Mathiya et al.[5] draw the conclusion that the default setting of Hadoop is not suitable for every job and cluster. They present the recommended parameter configuration in the paper, after looking up numerous documentations about Hadoop parameter tuning. Aung and Zaw [6] also have breakthrough on Hadoop parameter tuning. They put forward an algorithm that can tune the parameter for users. The introduction of this new tool let Hadoop become more user-friendly. Machine learning is also implemented to predict misconfigurations in Hadoop [7].

Some researchers are working on improving the job scheduling algorithm of Hadoop. Hussain et al.[8] introduced a new job scheduling algorithm by minimizing iterations. The algorithm is able to perfect the time cycle of fair scheduling and compute the time and space complexity more accurately.

5 Conclusion

The new generation of Hadoop uses containers to manage its resource allocation and job scheduling. Even though the implementation of containers provides a more fine-grained resource management, containers still utilize resource inefficiently. The amount of resource allocated for each container is determined in the configuration files and is static in runtime. This mechanism makes Hadoop vulnerable to unexpected memory spike. Ideally, the resource allocated for each container should be flexible and dynamic so that the framework can improve its efficiency by increasing the parallelism safely.

The next problem which should be addressed in the research is how JVM allocates memory and other resource for Hadoop containers. If the mechanism behind the resource management can be figured out, we can then modify the way in which resource is allocated to each container. If resource can be borrowed from those containers that have excessive resource, then the framework can have more parallelism and therefore shorten the completion time of a certain task.

References

[1] https://hadoop.apache.org.

[2] Y. Yao, H. Gao, J. Wang, B. Sheng, and N. Mi. New scheduling algorithms for improving performance and resource utilization in hadoop yarn clusters. *IEEE Transactions on*

Cloud Computing, pages 1–1, 2019. https://ieeexplore.ieee.org/abstract/document/8624318?casa_token=Sm0yhwrYHbcAAAAA:9Dbs-5B3fHU3F5gGEZhL3hfuUET04umUcreuuf-mz6zaw_tB7gd8vsO6LGA_PSc3V7j0R16lRg.

[3] R. Alanazi, F. Alhazmi, H. Chung, and Y. Nah. A multi-optimization technique for improvement of hadoop performance with a dynamic job execution method based on artificial neural network. *SN Computer Science*, (184), 2020. https://link.springer.com/article/10.1007%2Fs42979-020-00182-3#citeas.

[4] Y. Guo, J. Rao, C. Jiang, and X. Zhou. Moving hadoop into the cloud with flexible slot management and speculative execution. *IEEE Transactions on Parallel and Distributed Systems*, 28(3):798–812, 2017. https://ieeexplore.ieee.org/abstract/document/7506079?casa_token=ZIpqk6CxEdUAAAAA:-_SXfOsEnViKt2IbdQ9y8nAZE9EeS1GzB42bB9vWSHmMV_qoq1zTXs9DKza-cWnrF9FFudUE5Q.

[5] B. J. Mathiya and V. L. Desai. Apache hadoop yarn parameter configuration challenges and optimization. In *2015 International Conference on Soft-Computing and Networks Security (ICSNS)*, pages 1–6, 2015. https://ieeexplore.ieee.org/document/7292373.

[6] T. H. Aung and W. T. Zaw. Improved job scheduling for achieving fairness on apache hadoop yarn. In *2020 International Conference on Advanced Information Technologies (ICAIT)*, pages 188–193, 2020. https://ieeexplore.ieee.org/abstract/document/9261793?casa_token=xHmH6U4N01AAAAAA:WZt2pkBEPDnBVeMK01HEGE-_uN4xdw4FWE8TMPdP_AWd6-WWEhNezG7h3hTP9nd61Sj7N97MrQ.

[7] A. Robert, A. Gupta, V. Shenoy, D. Sitaram, and S. Kalambur. Predicting hadoop misconfigurations using machine learning. In *Software: Practice and Experience*, volume 50, pages 1168–1183, 2020. https://onlinelibrary.wiley.com/doi/full/10.1002/spe.2790?casa_token=UjP32oIjQ0IAAAAA%3AMK4Shz59teLyrIO405vo3ws44IJJwGRbBeoCkCpt6QakeeaYFbBxEw3vC0F38dW3LR2PkkeURP_hga6G.

[8] R. Hussain, M. Rahman, K. I. Masud, S. M. Roky, M. N. Akhtar, and T. A. Tarin. A novel approach of fair scheduling to enhance performance of hadoop distributed file system. In *2019 International Conference on Electrical, Computer and Communication Engineering (ECCE)*, pages 1–6, 2019. https://ieeexplore.ieee.org/abstract/document/8679252?casa_token=Kbu2oPm_IgUAAAAA:NfK83qFiC4bbavewmZKEgpUIXYdxjIUWo0jxLZQSlsT0O-IpACNDwofb