IARJSET

ISSN (Online) 2393-8021
ISSN (Print) 2394-1588

International Advanced Research Journal in Science, Engineering and Technology
Vol. 3, Issue 6, June 2016

Parallel Memory-Intensive MapReduce Applications

Ramya.P[1], Dr. V. Venkatesakumar[2]

PG Scholar, Department of CSE, Anna University Regional Campus Coimbatore, Tamilnadu, India[1]

Assistant Professor, Department of CSE, Anna University Regional Campus Coimbatore, Tamilnadu, India[2]

Abstract: The Map Reduce platform has been widely used for large-scale data processing and analysis recently. It works well if the hardware of a cluster is well configured. However, our survey has indicated that common hardware configurations in small- and medium-size enterprises may not be suitable for such tasks. This situation is more challenging for memory-constrained systems, in which the memory is a bottleneck resource compared with the CPU power and thus does not meet the needs of large-scale data processing. The traditional high performance computing (HPC) system is an example of the memory-constrained system according to our survey. The proposed new MapReduce system, which aims to improve Map Reduce performance using efficient memory management. The parallel multi-buffer technique to balance data production from CPU and data consumption of disk I/O's, which implements the non-blocking I/O. The parallel also caches the final merged files output by Map tasks in memory to avoid re-reading them from disks before transferring them to remote reduce tasks. All Map/Reduce tasks in a physical node run inside the execution engine, and therefore in a multi JVM, which is one of the key architectural differences between multi JVM and Hadoop. A multi-threaded execution engine, which is based on Hadoop but runs in a multi JVM on a node. In the execution engine, we have implemented the algorithm of hyper scheduling to job assignment, such as sequential disk accessing, multi-cache and shuffling from memory, and solved the problem of full garbage collection in the MJVM. We have conducted extensive experiments to compare parallel Mammoth with scheduling algorithm against the native Hadoop platform. The results show that the modified mammoth system can reduce the job execution time by more than 80 percent in typical cases, without requiring any modifications of the Hadoop programs. Given the growing importance of supporting large-scale data processing and analysis and the proven success of the MapReduce platform, the parallel Mammoth system can have a promising potential and impact.

Keywords: Map Reduce, HPC, CPU, Mammoth, Hadoop, HDFS, MJVM.

I. INTRODUCTION

Apache Hadoop has been generating a lot of headlines lately. For those who are not aware, Hadoop is an open source project that provides a distributed file system and MapReduce framework for massive amounts of data. The primary hardware used for Hadoop is clusters of commodity servers. File sizes can easily be in the petabyte range and use hundreds or thousands of computer servers. Hadoop also has many components that live on top of the core Hadoop file system (HDFS) and MapReduce mechanism. Interestingly, HPC and Hadoop clusters share some features, but how much crossover you will see between the two disciplines depends on the application. Hadoop strengths lie in the sheer size of data it can process and its high redundancy and toleration of node failures without halting user jobs.

MapReduce can be classified as a SIMD (single-instruction, multiple-data) problem. Indeed, the map step is highly scalable because the same instructions are carried out over all data. Parallelism arises by breaking the data into independent parts with no forward or backward dependencies (side effects) within a Map step; that is, the Map step may not change any data (even its own). The reducer step is similar, in that it applies the same reduction process to a different set of data (the results of the Map step).

In Hadoop, the tasks are scheduled according to the number of CPU cores, without considering other resources. This scheduling decision leads to long waiting time of CPUs, which influences the total execution time due to the performance gap between the CPU and the I/O system. In Hadoop, every task is loaded with a JVM. Every task has an independent memory allocator.

A Hadoop task contains several phases that involve memory allocation: task sort buffer, file reading and writing, and application-specific memory usage.

Most memory allocation is pre-set with parameters in the job configuration without considering the real tasks' demand. Besides, it does not have a memory scheduler for all the tasks in a Task Tracker. These designs will lead to the problem of buffer concurrency among Hadoop tasks. Another issue is that disk operations in Hadoop are not scheduled cooperatively.

Every task reads and writes data independently according to its demand without coordination, which potentially leads to heavy disk seeks. For instance, in the merge and shuffle phases, the overhead of uncoordinated disk seeks and the contention in accesses are so big that the I/O wait occupies up to 50 percent of the total time as observed,

which significantly degrades the overall system performance.

Memory is an important resource to bridge the gap between CPUs and I/O devices. However, we observed the following main problems for the memory usage in Hadoop. First, the memory management in Hadoop is rather coarse grained. The memory size available for a Map or Reduce task is set in a static configuration file, and is fixed at runtime. Therefore, even after the Map tasks are completed, the buffers used by those tasks cannot be used by the Reduce tasks. Second, the memory mainly used by the Map tasks is the intermediate buffer. When the buffer cannot accommodate the intermediate data, the entire intermediate data will be spilled to the local disk. For example, if the intermediate data for a Map task are 100 MB, but the buffer size is only 80 MB, then the intermediate data will be spilled to the disk as an 80 MB file and a 20 MB file. As the result, the final merge-sort phase in Hadoop will be largely affected by reading an 80 MB file from the disk. Finally, although different Map tasks may produce the intermediate data with different sizes (e.g., the "grep" application), Hadoop does not provide the mechanism for the concurrently running Map tasks to coordinate their memory usage with each other.

The I/O operations may also cause very inefficient usage of resources. Firstly, a merge-sort algorithm is widely used in Hadoop. In this algorithm, the operations of CPU computing (sort) and disk spilling are mashed together. There are a multitude of I/O waits during this procedure. Secondly, parallel I/O is performed in Hadoop whenever possible. Parallel I/O may cause vast disk seeks. Especially, the situation may become even worse when there is only one disk on a node. Finally, as mentioned above, the Reduce tasks will have to pull the output files of the Map tasks, which should be performed as early as possible in order to improve the read performance.

II. RELATED WORK

The existing system is that these jobs run multiple tasks in parallel and a job is sped up only when inputs of all such parallel tasks are cached. Indeed, a single task whose input is not cached can slow down the entire job. To meet this "all-or-nothing" property, we have built PACMan, a caching service that coordinates access to the distributed caches. This coordination is essential to improve job completion times and cluster efficiency. To this end, we have implemented two cache replacement policies on top of PACMan's coordinated infrastructure LIFE that minimizes average completion time by evicting large incomplete inputs, and LFU-F that maximizes cluster efficiency by evicting less frequently accessed inputs. In exiting HaLoop, a modified version of the Hadoop MapReduce framework that is designed to serve these applications. HaLoop not only extends MapReduce with programming support for iterative applications, it also dramatically improves their efficiency by making the task scheduler loop-aware and by adding various caching mechanisms. We evaluated HaLoop on real queries and real datasets. Compared with Hadoop, on average, HaLoop reduces query runtimes by 1.85, and shuffles only 4% of the data between mappers and reducers.

The memory management in Mammoth and Spark are rather different. Mammoth is based on Map- Reduce. We have carefully analysed the characteristics of memory usage in different phases of the MapReduce framework, and designed a novel rule-based heuristic to prioritize memory allocations and revocations among execution units (mapper, shuffler, reducer, etc.). In this way, we can maximize the holistic benefits of the Map/Reduce job when scheduling each memory unit. In Spark, the memory can be used for the resilient distributed data sets (RDD) cache and running the framework itself. As for the RDD cache, it depends on the users themselves when and how the data are cached, which increases the uncertainty of the memory usage. For the iterative and the interactive applications, caching the frequently used RDDs will significantly improve the applications performance. However, for many batch processing applications, the RDD cache cannot exhibit its advantages, and therefore those applications can only rely on the memory management in the Spark framework itself. Spark directly requests and revokes the memory from the JVM, and does not have a global memory manager in the application level. Spark uses the hash table to aggregate the data, which is different from the sort way used by Mammoth and Hadoop. When the memory is sufficient, hash will certainly be quicker than sort. However, when the memory is insufficient, it will have to spill the data to disks, and its performance will decrease significantly. Thanks to the holistic manner of memory usage, Mammoth can adapt much better to various memory situations, even when the memory is insufficient. On the contrary, the performance achieved by Spark is excellent when there is the sufficient memory, but not so when the memory is insufficient.

Spark and Mammoth are different. Spark writes the data to the disk on one side and reads them from the disk on the other, while Mammoth stores the Map tasks results in the Send Buffer, and sends them to the Reduce Tasks Receive Buffer directly (Mammoth will write the data in the Send Buffer to the disks only for the purpose of fault tolerance). Another existing system use Main Memory Map Reduce (M3R) is a new implementation of the Hadoop Map Reduce (HMR) API targeted at online analytics on high mean-time-to-failure clusters.

It does not support resilience, and supports only those workloads which can fit into cluster memory. In return, it can run HMR jobs unchanged -- including jobs produced by compilers for higher-level languages such as Pig, Jaql, and System ML and interactive front-ends like IBM Big Sheets while providing significantly better performance than the Hadoop engine on several workloads (e.g. 45x on some input sizes for sparse matrix vector multiply). M3R also supports extensions to the HMR API which can enable Map Reduce jobs to run faster on the M3R engine, while not affecting their performance under the Hadoop engine.

IARJSET
ISSN (Online) 2393-8021
ISSN (Print) 2394-1588
International Advanced Research Journal in Science, Engineering and Technology
Vol. 3, Issue 6, June 2016

III. PROPOSED SYSTEM

MapReduce approach aiming to maximize data locality and task parallelism of MapReduce applications on Hadoop. Reuse and adapt an efficient MapReduce implementation for shared memory multiprocessor to Hadoop, called MJVM (multi JVM). MapReduce on multicore and aggressively exploits task parallelism and data locality on multicore. The proposed scheduling algorithm and find the most suitable runtime environment for execution on datasets of varying sizes. The proposed design a novel parallel memory allocation and revocation among execution units (mapper, shuffler, reducer, etc.), to maximize the holistic benefits of the Map/Reduce job when scheduling each memory unit.

A. Load dataset from HDFS

1) To start the file read operation, client opens the required file by calling open() on Filesystem object which is an instance of DistributedFileSystem. Open method initiate HDFS client for the read request.
2) DistributedFileSystem interacts with Namenode to get the block locations of file to be read. Block locations are stored in metadata of name node. For each block, Namenode returns the sorted address of Datanode that holds the copy of that block. Here sorting is done based on the proximity of Data node with respect to Namenode, picking up the nearest Data node first.
3) Distributed File System returns an FS Data Input Stream, which is an input stream to support file seeks to the client. FS Data Input Stream uses a wrapper DFS Input Stream to manage I/O operations over Name node and Data node. Following steps are performed in read operation.

a) Client calls read() on DFS Input Stream. DFSInputStream holds the list of address of block locations on Datanode for the first few blocks of the file. It then locates the first block on closest Datanode and connects to it.

b) Block reader gets initialized on target Block/Datanode along with below information:

- Block ID.
- Data start offset to read from.
- Length of data to read.
- Client name.

c) Data is streamed from the Datanode back to the client in form of packets, this data is copied directly to input buffer provided by client's client is reading and performing checksum operation and updating the client buffer

d) Read () is called repeatedly on stream till the end of block is reached. When end of block is reached DFSInputStream will close the connection to Datanode and search next closest Datanode to read the block from it.

4) Blocks are read in order, once DFSInputStream done through reading of the first few blocks, it calls the Namenode to retrieve Datanode locations for the next batch of blocks.
5) When client has finished reading it will call Close() on FSDataInputStream to close the connection.
6) If Datanode is down during reading or DFSInputStream encounters an error during communication, DFSInputStream will switch to next available Datanode where replica can be found. DFSInputStream remembers the Datanode which encountered an error so that it does not retry them for later blocks.

B. Dynamic Slot allocation for MAP/Reduce

MapReduce suffers from a under-utilization of the respective slots as the number of map and reduce tasks varies over time, resulting in occasions where the number of slots allocated for map/reduce is smaller than the number of map/reduce tasks. Our dynamic slot allocation policy is based on the observation that at different period of time there may be idle map (or reduce) slots, as the job proceeds from map phase to reduce phase. We can use the unused map slots for those overloaded reduce tasks to improve the performance of the MapReduce workload, and vice versa. For example, at the beginning of MapReduce workload computation, there will be only computing map tasks and no computing reduce tasks, i.e., all the computation workload lies in the map-side. In that case, we can make use of idle reduce slots for running map tasks. That is, we break the implicit assumption for current MapReduce framework that the map tasks can only run on map slots and reduce tasks can only run on reduce slots. Instead, we modify it as follows: both map and reduce tasks can be run on either map or reduce slots.

However, there are two challenges that should be considered as follows:

1) Intra-phase dynamic slot allocation. Each pool is split into two sub-pools, i.e., map-phase pool and reduce-phase pool. At each phase, each pool will receive its share of slots. An overloaded pool, whose slot demand exceeds its share, can dynamically borrow unused slots from other pools of the same phase. For example, an overloaded map-phase Pool 1 can borrow map slots from map-phase Pool 2 or Pool 3 when Pool 2 or Pool 3 is under-utilized, and vice versa, based on max-min fair policy.
2) Inter-phase dynamic slot allocation. After the intraphase dynamic slot allocation for both the map-phase and reduce-phase, we can now perform dynamic slot allocation across typed phases. That is, when there are some unused reduce slots at the reduce phase, and the number of map slots at the map phase is insufficient for map tasks, it will borrow some idle reduce slots for map tasks, to maximize the cluster utilization, and vice versa.

3) Memory Optimization

The Multi JVM that allows the correct functioning of the Hadoop jobs. To summarize:

- Use the latest (stable) Linux distribution that allows for the correct functioning of the Hadoop jobs
- Use the latest (stable) Hadoop distribution for the Hadoop workload at hand
- Use the M JVM and 3d-party libraries that the underlying Hadoop workload depends on

IARJSET
ISSN (Online) 2393-8021
ISSN (Print) 2394-1588
International Advanced Research Journal in Science, Engineering and Technology
Vol. 3, Issue 6, June 2016

Reader and Spiller to overlap the I/O operations mainly use the parallel multi-buffer. However, it is likely that the CPU computation or the disk I/O operations become the performance bottleneck when both CPU computation and I/O operations are being performed. For example, if the CPU produces the data faster than the disk I/O, the disk I/O becomes the performance bottleneck and the data will accumulate in the buffer queue. Otherwise, the CPU computation is the bottleneck. This is the reason why every queue in Spiller or Reader has a threshold memory size of 10 MB by default. Once the size of the accumulated data exceeds the threshold of the queue, the CPU will be blocked until there is the spare space in the queue. Once there are the data added to the queue, it will be written to the disk soon. The situation for Reader is the same, and therefore its discussion is omitted.

The Scheduler executes as a parallel model (i.e., a multi instance) in the execution engine. A simple design is to design a parallel component that is responsible for all memory allocations and revocations. However, such a design will cause the Cache Scheduler to become a performance bottleneck at runtime. This is because the cache scheduler must interact with the map/reduce tasks at runtime to assign and recycle memory, and the interactions can be very frequent.

C. Data Optimization

Dynamic job priorities are used in HSA for reducing the latency of variable length concurrent jobs, while maintaining data locality. The dynamic scheduling algorithm considers the estimated runtime, job size, and waiting time of the job in the queue. The waiting time is increased as the job waits in the queue, whereas the remaining size of the job is reduced when its individual tasks are completed.

HYBRID SCHEDULING ALGORITHM (HCA)

Based on the experimental results and analysis, we propose a hybrid scheduler for scalable and heterogeneous Hadoop systems. Propose a Hybrid Scheduler algorithm based on dynamic priority in order to reduce the latency for variable length concurrent jobs, while maintaining data locality.

Input: Q: queue of incoming jobs; Inputload: number of clients for transactional or data size for MapReduce job; P CLUSTER: cluster of physical nodes; V CLUSTER: cluster of virtual nodes; JCTdesired[]: vector of jobs desired completion times.

1: for each job Ji in Q=J1,J2,...,Jn do
2: if Ji ∈ transactional workload then
3: Place Ji on V CLUSTER
4: else if Ji ∈ batch MapReduce workload then
5: Profile Ji using MJVM Algorithm 1 to obtain the vector of estimated job completion time (JCTestimated[]).
6: if JCTestimated[i] ≥ JCTdesired[i] then
7: Place Ji on P CLUSTER
8: else
9: Place Ji on V CLUSTER
10: end if
11: end if
12: return jobs assigned to P CLUSTER and V CLUSTER
13: end for

The dynamic priorities can accommodate multiple task lengths, job sizes, and job waiting times by applying a MJVM for job task processor assignment. The estimated runtime of Map and Reduce tasks are provided to the HCA dynamic priorities from the historical Hadoop log files. In addition to dynamic priority, we implement a reordering of task processor assignment to account for data availability to automatically maintain the benefits of data locality in this environment.

IV. EXPERIMENTAL RESULTS

The size of intermediate data has a big impact on performance of Hadoop. Three typical built-in benchmark applications in Hadoop are used in these experiments: Word Count without Combiner, Sort and Word-Count with combiner (WCC). These three benchmarks represent different relations between intermediate data and input data. Word Count without combiner, Sort, and Word-Count with combiner represent the cases where the size of intermediate data is larger than, equal to and smaller than the size of input data, respectively.

WordCount is a canonical MapReduce application, in which the Map function translates every word in the input data to a <word; 1> pair in the intermediate data and the Reduce function sums the word's occurrences and transmits a <word;N> pair. WordCount without combiner refers to this version of WordCount. Suppose that the average length of the words is x bytes. In the intermediate data, the value of the <key; value> pair is an integer and its length is 4 bytes in Java. Then the ratio between the size of the intermediate data and the size of the input data is 4 x.

The Word Count with Combiner refers to Word Count application with a combiner function. The combiner function aggregates the Map task's results by summing up the word's occurrences and transmitting a <word;N> pair for a map task. Based on this function, the intermediate data will be <word;N> pairs, which will be smaller than the input words. Sort is the useful measurement of MapReduce performance, in which the MapReduce framework will sort the data automatically, and the Map function just transmits all the input <key; value> items as the intermediate items.

TABLE I
COMPARE PERFORMANCE RELATIVE TO AVAILABLE MEMORY

Methods	Memory					
	128	64	48	32	16	8
Hadoop	3.2	2.0	2.4	4.8	6.4	8.5
Mammoth	2.8	1.6	1.2	3.1	4.7	7.3
Proposed	2.3	1.2	0.8	2.7	3.2	5.0

IARJSET
ISSN (Online) 2393-8021
ISSN (Print) 2394-1588
International Advanced Research Journal in Science, Engineering and Technology
Vol. 3, Issue 6, June 2016

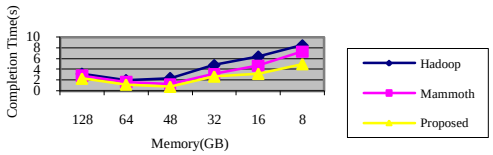

Fig. 1 Performance relative to available memory.

TABLE III
THE PERFORMANCE RELATIVE TO INTERMEDIATE DATA SIZE FOR WC

Methods	Data Size				
	2	4	6	8	10
Hadoop	87	103	148	187	212
Mammoth	65	87	112	154	181
Proposed	32	43	78	101	121

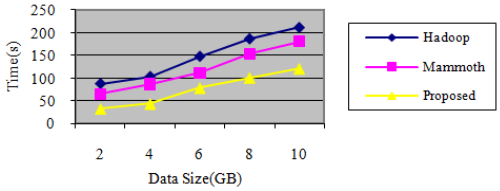

Fig. 2 Compare Job execution Time for WC

TABLE IIIII
PERFORMANCE COMPARISON AMONG SPARK, HADOOP, MODIFIED MAMMOTH AND PROPOSED

Methods	Data Size				
	2	4	6	8	10
Spark	694	743	799	832	956
Hadoop	542	643	689	743	843
Mammoth	432	324	365	432	541
Proposed	321	287	293	312	386

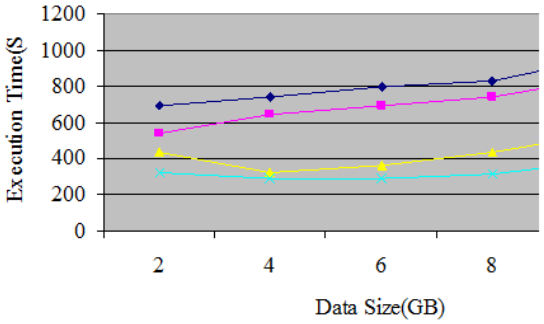

Fig. 3 Performance comparison among Spark, Hadoop, Mammoth and Proposed

The proposed system will compare the performance of Modified Mammoth and Hadoop in terms of i) job execution time, ii) CPU utilization, and iii) I/O utilizations. Random writer (native in Hadoop) produces the input data set of Sort. The size of the data set for Sort is 320 GB, 5 GB for each slave node. The input data set for Word Count is produced by random text writer (native in Hadoop too), and the size is also 320 GB with 5 GB for each slave node. Each job is run independently for three times and the collected results are then averaged.

V. CONCLUSIONS

Multi JVM based Hybrid scheduling algorithm, is presented for the Hadoop MapReduce environment. Show how the default parameters quickly resolve the data dependence between the Map and Reduce phases by improving the wait time of the last Map task on average. Additionally, we show simulation analysis of the optimal service level value and policies for overall performance as well as response time under a variety of conditions. In conclusion, MJVM is a fast and flexible scheduler that improves response time for multi-user Hadoop environments.

A new research angle along this research direction is that we still manage the memory in the application level, but try to manage the memory opportunistically instead of carefully crafting the memory usage as we did in proposed. In the further, we plan to design and implement the opportunistic approach to utilize the memory in proposed only focuses on the disk I/O currently. We plan to integrate the support of the network I/O into proposed in the future.

REFERENCES

[1]. G. Ananthanarayanan, A. Ghodsi, A. Wang, D. Borthakur, S. Kandula, S. Shenker, and I. Stoica, "PACMan: Coordinated memory caching for parallel jobs," in Proc. Symp. Netw. Syst. Des. Implementation, 2012, p. 20
[2]. Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst, "HaLoop: Eficient iterative data processing on large clusters," presented at the Int. Conf. Very Large Data Bases, Singapore, 2010.
[3]. J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," in Proc. USENIX Symp. Oper. Syst. Des. Implementations, 004, p. 10.
[4]. B. Hindman, A. Konwinski, MateiZaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica, "Mesos: A platform for fine-grained resource sharing in the data center," in Proc. Symp. Netw. Syst. Des. Implementation, 2011, p. 8.
[5]. T. Hoefler, J. Dongarra, and A. Lumsdaine, "Towards efficient mapreduce using MPI," in Proc. 16th Eur. PVM/MPI Users' Group Meeting in Recent Adv. Parallel Virtual Mach. Message Passing Interface, 2009, pp. 240–249.
[6]. U. Kang, C. E. Tsourakakis, and C. Faloutsos, "PEGASUS: A petascale graph mining system implementation and observations," in Proc. IEEE Int. Conf. Data Mining, 2009, pp. 229–238.
[7]. M. V. Neves, T. Ferreto, and C. D. Rose, "Scheduling mapreduce jobs in HPC clusters," in Proc. Eur. Conf. Parallel Distrib. Comput., 2012, pp. 179–190.
[8]. A. Rasmussen, G. Porter, M. Conley, H. V. Madhyastha, R. N.Mysore, A. Pucher, and A. Vahdat, "ThemisMR: An i/o-efficient mapreduce," in Proc. ACM Symp. Cloud Comput., 2012.
[9]. M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes, "Omega: Flexible, scalable schedulers for large compute clusters," in Proc. Eur. Conf. Comput. Syst., 2013, pp. 351–364.
[10]. A. Shinnar, D. Cunningham, and B. Herta, "M3R: Increased performance for in-memory Hadoop jobs," in Proc. Int. Conf. Very Large Data Bases, 2012, pp. 1736–1747.