Volume 3, Issue 9, September 2013 ISSN: 2277 128X

International Journal of Advanced Research in Computer Science and Software Engineering

Research Paper

Available online at: www.ijarcsse.com

Automated Cloud Based File Storage Nodes Balancer

Manjula L
M.Tech Scholar, Dept.of CSE,
Madanapalle Institute of Technology and Sciences,
Madanapalle, JNTUA, India

Sreedevi M
Assoc.professor, Dept.of CSE,
Madanapalle Institute of Technology and Sciences,
Madanapalle, JNTUA, India

Abstract— In Distributed file systems, nodes concurrently work for computing and storage functions; a file is partitioned into a number of chunks allocated in distinct nodes so that Map Reduce tasks can be performed in parallel over the nodes. But in this architecture, Storage nodes strongly depend on a central server for chunk reallocation which happens to be a manual and time consuming process. This dependence is clearly inadequate in a large-scale, failure-lying environment since the central load balancer is put under considerable workload as it simultaneously need to handle storage nodes and incoming client connections that is linearly scaled with the system size, and thus turn out the performance bottleneck and the single point of failure. One problem with Map Reduce is that it is basically batch-processing oriented. When you start the process, you cannot be easily update the input data and expect the output to be same. Therefore, MapReduce is poor at real-time processing. Still it will remain fine for latence-ignorant applications such as Extract-Transform-Load or number crunching operations. On the other hand due to the dynamic nature of the clouds and heterogeneous architecture between central server and storage nodes prompted us to explore other alternatives to support parallel processing besides MapReduce. Proposes to use a software load balancer equipped with a rebalancing algorithm in combination with Bulk Synchronous Parallel (BSP) bridging model for supporting parallel operations instead of MapReduce. Combined with BSP model to attain parallel processing and usage of automated software load balance we construct an efficient cloud file deliver model that has less file movement costs, and algorithmic overheads for chunking. A practical implementation validates the claim.

Keywords— MapReduce tasks, Bulk Synchronous Parallel (BSP), Load rebalancing algorithm, reduce the movement cost, Extract-Transform-Load

I. INTRODUCTION

Cloud storage system, consists of collection of storage nodes governed by a centralized server, providing long-term storage services over the Internet effectively and efficiently. Various numerical methods for scientific computations based on the patterns of scattering and gathering of data between processing nodes are listed as follows: Dense Linear-Algebra, Sparse Linear-Algebra, Spectral-Methods, N-Body Methods, StructuredGrids, Unstructured Grids, MapReduce[1], Combinational Logic, Graph Traversal, Dynamic Programming, Backtrack and Branch-and-Bound, Graphical Models, Finite State Machines.

MapReduce [1] provides regular programmers the capability to produce parallel distributed programs more easily, by making them to write only the simpler Map() and Reduce() functions, which focuses on the logic of the particular problem at hand, while the "MapReduce System" (also known as infrastructure & framework) spontaneously takes care of marshalling the distributed servers, running the different task parallelly, by managing all the communications and data transfers between the different parts of the system, providing for duplicates and failures, and the management of the whole process. Distributed file systems are key building blocks for cloud computing applications based on the MapReduce paradigm. In such type of file systems, nodes simultaneously serve computing and storage functions; a file is partitioned into a number of chunks allocated in distinct nodes so that MapReduce tasks can be performed parallely over the nodes. For example, consider a word count application that counts the number of distinct words and the frequency of each unique word in a large file. In such type of applications, a cloud partitions the file into a large number of disjointed and fixed-size pieces (or file chunks) and assigns them to different cloud storage nodes (i.e., chunkservers). Each storage node (or node for short) then calculates the frequency of each unique word by scanning and parsing its local file chunks. In such type of a distributed file system, load of a node is naturally proportional to the number of file chunks the node possesses. Because the files in a cloud be randomly created, deleted, appended, and nodes are upgraded, replaced and added in the file system, the file chunks are not distributed as uniformly as possible between the nodes. Load balance between the storage nodes is a crucial function in clouds. In load-balanced cloud, the resources be well utilized and conditioned, maximize the performance of MapReduce-based applications. In Distributed file systems, nodes simultaneously serve computing and storage functions; a file is partitioned into a number of chunks allocated in distinct nodes so that MapReduce tasks can be performed in parallel over the nodes. But in this architecture, Storage nodes strongly depend on a central server for chunk reallocation which happens to be a manual and time consuming process. This dependence is clearly inadequate in a large-scale, failure-lying environment since the central load balancer is put under considerable workload as it simultaneously need to handle storage nodes and incoming client

connections that is linearly scaled with the system size, and may therefore making the performance bottleneck and the single point of failure. Later Proposes to use a software load balancer equipped with a rebalancing algorithm in combination with centralized server and storage nodes. Specifically, in this study we implement offloading the load rebalancing task to storage nodes by having the storage nodes balance their loads essentially. This removes the dependency on the central nodes. The storage nodes are designed as a network based on distributed hash tables.

One problem with MapReduce is that it is basically batch-processing oriented. When you start the process, you cannot easily update the input data and expect the output to be same. Thus, MapReduce is poor at real-time processing .So far, it will remain fine for latence-ignorant applications such as Extract-Transform-Load or number crunching operations. But due to the dynamic nature of the clouds and heterogeneous architecture between central server and storage nodes prompted us to explore other alternatives to support parallel processing besides MapReduce. In this paper, we proposed to use Bulk Synchronous Parallel (BSP) bridging model for designing parallel algorithms. A bridging model "is intended neither as hardware nor a programming model but something in between them". A BSP having three components:

- Concurrent computation: Several computations take place on every contributing processor. Each process uses only the values stored in the local memory of the processor. The computations are not dependent, in the way that occurs asynchronously of all the others.
- Communication: The processes exchange data among themselves. This exchange takes the form of one-sided put and get calls, other than two-sided send and receive calls.
- Barrier synchronization: Whenever the process reaches the point (the barrier), it waits till all other processes have finished their communication actions.

Proposed system is associated with BSP model to gain parallel processing and usage of automated software load balance we build an efficient cloud file deliver model that has less file movement costs, and algorithmic overheads for chunking.

II. RELATED WORK

Bulk Synchronous Programming [2] and some MPI primitives [3] provides higher-level abstractions that make it simpler for the programmers to write parallel programs. A difference among these systems and MapReduce is that MapReduce misuses a restricted programming model to parallelize the user program automatically and to provide transparent fault-tolerance. Our sections optimization draws its inspiration from techniques as active disks, where computation is pressed into processing elements that are close to local disks, to reduce the quantity of data sent across I/O subsystems or the network. It moves along on the commodity processors to which a small number of disks are connected instead of moving along directly on disk controller processors, but the general approach is same.

Our backup task mechanism is same as the eager scheduling mechanism employed in the Charlotte System [4]. One of the limitations of simple eager scheduling is that if a given task causes repeated failures, the entire computation fails to execute. Some occurrences of this problem with mechanism of skipping faulty records. The MapReduce implementation depends on an in-house cluster management system that is reliable for distributing and moving along user tasks on a large collection of mutual machines. The cluster management system is similar to other systems such as Condor [5]. The sorting ability that is a part of the MapReduce library is alike in operation to NOW-Sort [6]. Source machines (map workers) partitions the data that is to be sorted and send it to one of R reduces workers. Each and every reduced worker sorts its data locally (in memory if possible). NOW-Sort does not have the user definable Map and reduces the functions that makes the library widely applicable. River [7] gives a programming model where the processes communicate with each other by sending the data over the distributed queues. Similar to MapReduce, the River system gives the average performance even in the presence of non-uniformities established by heterogeneous hardware or system perturbations. River accomplish this by careful scheduling of disk and network transfers to accomplish balanced completion times. MapReduce has a different approaches. By controlling the programming model, the MapReduce framework is capable to partition the problem into a large number of fine-grained tasks. These tasks are dynamically planned on accessible workers, for that faster workers process large tasks. The restricted programming model permit us to arrange redundant executions of tasks near to the end of job which greatly decreases the completion time in the presence of non-uniformities (such as slow or stuck workers).

BAD-FS [8] has a different programming model after MapReduce, and dissimilar MapReduce, is focused on the execution of jobs across a wide-area network. However, there are two fundamental similarities. (1) Both systems use duplicate execution to recover from data loss occurred by failures. (2) Both follows locality-aware scheduling to reduce the amount of data sent across stuffy network links. TACC [9] system built to simplify construction of highly-available networked services. Similar to MapReduce, it depends on re-execution as a mechanism for applying fault-tolerance.

III. BASIC BSP MODEL

The BSP model of parallel computation is well- defined as the combination of three attributes:
1. A number of components, performing processing and/or memory functions;
2. A router that distributes messages point to point between pairs of components; and
3. Capability for synchronizing all or a subset of the components at regular intervals of L time units where L is the periodicity parameter.

A BSP machine is a set of n processors with local memory, communicating with a router, whose computations are arrangements of super steps. In a super step, each and every processor (i) reads messages received in the previous super

step; (ii) carry out computation on locally available data; (iii) sends messages to other processors; and (iv) participate in global barrier synchronization.

A computation consists of arrangements of super steps. In each super step, each and every component is allocated a task containing of some combination of local computation steps, message transmissions and (implicitly) message coming from other components. After each and every period of L time units, a global check is done to examine whether the super step has been finished by all the components. If it has done, the machine proceeds to the next super step. Else, the next period of L units is assigned to the unfinished super step.

IV. BASIC PROPOSOL

The chunk servers in our proposed system are organized as a DHT network; i.e., each and every chunk server establishes a DHT protocol. A file in the system is divided into a number of fixed - size chunks, and each and every chunk has a unique chunk handle (or chunk identifier) named with a globally call as hash function. The hash function resumes a unique identifier for a given file's path name string and a chunk index. Each chunk server also has a unique ID. We represent the IDs of the chunk servers in V. Unless otherwise clearly indicated, we denote the successor of chunk server i as chunk server i + 1 and the successor of chunk server n as chunk server 1. In a typical DHT, a chunk server i hosts the file chunk, except for chunk server n, which manages the chunks whose handles are in (nn, 1n].To discover a file chunk, the DHT lookup operation is achieved. In large amount of the DHTs, the average number of nodes visited for a lookup is O (log n).

DHTs are used in our proposed system for the following reasons:

A. The chunk servers self-configure and self-heal in our proposed system, since their arrivals, departures, and failures, streamlining the system provisioning and management.

B. if a node leaves, then its locally hosted chunks is reliably migrated to its successor;

C. if a node joins, then it allocates the chunks whose IDs immediately precede the joining node from its successor to manage.

Our proposed system deeply depends on the node arrival and departure operations to migrate file chunks among nodes.

V. PROPOSED LOAD REBALANCING SYTEM

A large-scale distributed file system in a load-balanced state if each chunk server hosts no chunks. In our proposed algorithm, each chunk server node i first guess whether it is under loaded (light) or overloaded (heavy) without global knowledge. A node is lightly weighted, if the number of chunks it hosts is less numbered than the threshold.

(i) *Basic Algorithms*:

In the basic algorithm, each node implements the gossip-based aggregation protocol in to collect the load status of a sample of randomly choiced nodes. Exactly, each node contacts a number of randomly selected nodes in the system and builds a vector denoted by V. A vector consists of entries, and each entry consist of the ID, network address and load status of a randomly choiced node. By using this gossip-based protocol, each node i exchanges its locally maintained vector with its neighbours until its vector has s entries. It then calculates the average load of the s nodes denoted by Ai and regards it as an estimation of A. The nodes perform our load rebalancing algorithm regularly, and they balance their loads and minimize the movement cost in a best-effort fashion.

(ii) *Taking Advantage Of Node Heterogeneity*:

Nodes participating in the file system are possibly heterogeneous in terms of the numbers of file chunks that the nodes can accommodate. We assume that there is one block resource for optimization although a node's capacity in practice should be a function of computational power, network bandwidth and storage space. In the distributed file system for Map Reduce-based applications, the load of a node is normally proportional to the number of file chunks the node possesses. In Map Reduce-based applications, we use a software load balancer equipped with a rebalancing algorithm in combination with Bulk Synchronous Parallel (BSP) bridging model which supporting parallel operations which was mentioned above. Thus, the rationale of this design is to ensure that the number of file chunks managed by node i is proportional to its capacity.

(iii) *Managing Replicas*

In distributed file systems (e.g., Google GFS and Hadoop HDFS), a constant number of replicas for each and every file chunk take care for the distinct nodes to improve file availability with respect to node failures and departures. Our load balancing algorithm does not treat replicas differently. It improbable that two or more replicas are placed in an identical node because of the random nature of our load rebalancing algorithm. Give any file chunk, our proposal implements the directory-based scheme in to mark the locations of k replicas for the file chunk. Exactly, the file chunk is connected with k−1 pointers that keep track of k−1 randomly selected nodes storing the replicas.

VI. PERFORMANCE

• Low movement cost: As node i is the lightest node including all chunk servers, the number of chunks transferred because of i's departure is small with the goal of reducing the movement cost.

• Fast convergence rate: The least - loaded node i in the system seeks to release the load of the heaviest node j, leading to quick system convergence towards the load time in a sequence can be further improved to reach the global load - balanced system state.

The time complexity of the above technique can be decreased if each light node can know which heavy node it needs to request chunks beforehand, and then all light nodes can balance their loads in parallel. Our proposal struggle to balance the loads of nodes and reduce the demanded movement cost as much as possible, while taking the help of physical network locality and node heterogeneity. In the lack of representative real workloads (i.e., the distributions of file chunks in a large-scale storage system) in the public domain, we have examined the performance of our proposal and balanced it against competing algorithms through synthesized probabilistic distributions of file chunks. The production of workloads stress test the load balancing algorithms by creating a few storage nodes that are heavily loaded.

VII. CONCLUSION

One problem with MapReduce is that it is basically batch-processing oriented. When you start the process, you cannot easily update the input data and expect the output to be same. On the other hand due to the dynamic nature of the clouds and heterogeneous architecture among central server and storage nodes prompted us to explore other alternatives to support parallel processing besides MapReduce. In this paper, we proposed to use the Bulk Synchronous Parallel bridging model for designing parallel algorithms. A bridging model "is intended neither as a hardware nor a programming model but something in between them". Our Proposed system is shared with BSP model to obtain parallel processing and usage of automated software load balance we build an efficient cloud file deliver model that has less file movement costs, and algorithmic overheads for chunking.

REFERENCES

[1] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," in Proc. 6th Symp. Operating System Design and Implementation (OSDI'04), Dec. 2004, pp. 137–150.

[2] L. G. Valiant. A bridging model for parallel computation. Communications of the ACM, 33(8):103.111, 1997.

[3] William Gropp, Ewing Lusk, and Anthony Skjellum. Using MPI: Portable Parallel Programming with the Message-Passing Interface. MIT Press, Cambridge, MA, 1999.

[4] Arash Baratloo, Mehmet Karaul, Zvi Kedem, and Peter Wyckoff. Charlotte: Metacomputing on the web. In Proceedings of the 9th International Conference on Parallel and Distributed Computing Systems, 1996.

[5] Douglas Thain, Todd Tannenbaum, and Miron Livny. Distributed computing in practice: The Condor experience. Concurrency and Computation: Practice and Experience, 2004.

[6] Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, David E. Culler, Joseph M. Hellerstein, and David A. Patterson. High-performance sorting on networks of workstations. In Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data, Tucson, Arizona, May 1997.

[7] Remzi H. Arpaci-Dusseau, Eric Anderson, Noah Treuhaft, David E. Culler, Joseph M. Hellerstein, David Patterson, and Kathy Yelick. Cluster I/O with River: Making the fast case common. In Proceedings of the Sixth Workshop on Input/Output in Parallel and Distributed Systems (IOPADS '99), pages 10.22, Atlanta, Georgia, May 1999.

