Journal of Computer Sciences

Investigations

Load Balancing For Cloud-Based Dynamic Data Processing

Talal Talib Jameel

Department of Dentistry, Al Yarmouk University College, Baghdad, Iraq

Article history
Received: 22-12-2016
Revised: 04-03-2017
Accepted: 20-05-2017

Email: talal.alhabeeb@gmail.com

Abstract: The Map/Reduce paradigm has dominated cloud computing since its beginnings. However, there are some scenarios in which Map/Reduce is not the best model. Once such situation is a system that collects data dynamically, with intermittent arrival times. In this study, we study a modified form of Map/Reduce that uses a load balancer to distribute work, rather than simply assigning a Map node in an ad-hoc fashion. We show that this approach performs significantly better than standard Map/Reduce. In particular, it reduces the amount of time data is waiting in a queue to be processed.

Keywords: Cloud Computing, Load Balancing, Map/Reduce, Dynamic Data, Resource Allocation, Data Center

Introduction

Current Challenges

Given a parallelized algorithm A, a set of resources R (typically a set of virtual machines and I/O channels in the cloud) and a set of data D, it is usually straightforward to distributed this data to the resources R. Assuming that the resources are uniform and that the number of resources is $N = |R|$, then the standard approach would be to assign D/N data values to each resource. This is what is referred to as a "static model," because all the data D is available in advance, before the algorithm is run.

This simple approach will not work for the "dynamic model," where new data can arrive while the algorithm is running. Simulated systems will typically follow the static model, while production systems with live data being produced by sensors will typically follow the dynamic model (Darema, 2004). Furthermore, in the dynamic data model the data may be arriving according to a "smooth" distribution or a "punctuated" distribution. In a smooth distribution, the number of data items that arrive in a time interval T is always the same. In a punctuated distribution, the data can arrive at any time (Yin *et al*., 2013). As an example, consider CPU fan speed. In a smooth distribution model, the CPU fan speed would be adjusted according to the CPU temperature sensor. The temperature sensor would be read every T seconds, so that in this case there would always be 1 new data point per measurement. In a punctuated distributions model, the CPU fan speed would be adjusted only if the CPU temperature exceeded a certain threshold value V. In this model there could be long intervals during which there is no data (because the temperature is less than V during that interval), followed by one or more data points in which the fan speed has to be changed because the CPU temperature has exceeded the threshold (Darema, 2005). Since the future history of the CPU temperature is effectively unpredictable, it is also not possible to determine the number of data points that will arrive during the interval T.

For a system with dynamic data, the simple D/N allocation algorithm is unlikely to work. Data must be distributed uniformly to each resource (Chandra *et al*., 2003). If time T has passed and the amount of data that has arrived is $D_{actual} < D/N$, then it may be better to wait before sending any of this data to a resource. If $D_{actual} > D/N$ then it may be better to partition the data, sending some of it to one or more resources, while keeping the rest of the data in a queue.

The fundamental question for a system with dynamic data is "What is the best strategy for distributing this data to the resources?" For the purposes f this document we define the "best strategy" to be the one that finishes processing a fixed amount of data D in the least amount of time. This paper proposes a data distribution strategy based on a specific form of load balancing and demonstrates that this load balancing approach performs significantly better than the static D/N approach.

This paper addressed the current challenges associated with load balancing in cloud-based dynamic data settings. It further introduced the load balancing to provide some insights about the current problems in resource allocations. Then, the simulation environment is

introduced followed by the method and results. The obtained result is discussed at the end of the paper from different perspectives.

Load Balancing

The problem of resource allocation is a common problem that arises in many different scenarios (Shirazi *et al*., 1995). In our particular situation we are faced with a resource allocation problem that is very similar to the problem of web server architecture in the face of high traffic. Heavily used web sites (for example cnn.com) may have millions of page load requests per minute (Bharadwaj, 1996). The typical solution that is used features replication (putting the page data on multiple servers), caching (keeping a copy of a formatted web page so long as the content has not changed) and load balancing (Ranganathan *et al*., 2002). For our purposes replication and caching are not needed, but load balancing is needed. In the server case, a load balancer is an algorithm that routes tasks to the least busy machine that it can locate. Unfortunately the concept of "least busy" is not well defined. It depends on a number of factors, including CPU load, memory availability and I/O bandwidth. When we specify our particular form of load balancer algorithm we will provide a precise definition of how a machine (typically a virtual machine) can compute its own "busy factor" so that the load balancer can decide how to distribute data to available machine resources.

There is another critical way in which our distributed algorithm differs from web server load balancing. This is the issue of granularity. For a web browser loading a web page, the user is unlikely to notice the difference between 0.5 sec to load and 0.6 sec to load, but will certainly notice the difference between 0.5 and 5 sec. Thus, for page loads, the granularity is very small, because it is based on the human perception of time. The distributed algorithms we use often need to perform complex cryptographic operations. These operations put high demands on the CPU, since even a single operation may take seconds (Cardellini *et al*., 1999). Thus, these algorithms have a very large granularity. They also put high demand on memory, because the algorithms need to allocate very large blocks of memory in order to run. The I/O bandwidth requirements are different, however. Typical algorithms will require very high I/O bandwidth during initialization, in order to distribute the key material. After the initialization phase is complete, however, I/O requirements are typically very low. During this steady state phase of the algorithm the I/O packets will consist of new data elements or partial results, which will require far less bandwidth than key exchange. When we describe our load balanced algorithm we will show how initialization cost is handled.

Simulation Environment

In order to validate our load balancing algorithm to the greatest extent we wish to perform experiments on many different configurations. In an actual cloud deployment it may be difficult to adjust some of the parameters that we wish to vary. For this reason we chose to implement and test our algorithm on a simulated cloud environment, namely cloudsim 4.0. Cloudsim is a Java-based cloud simulation environment that allows the developer control of all parameters within a configuration, from the most local parameters (individual characteristics of a single virtual machine) to the most global parameters (network configuration within a data center). This environment is ideal for research on cloud algorithms without incurring the development limitations one would encounter in a specific cloud deployment.

Using cloudsim also allowed us to build a generic plugin architecture to represent any algorithm. To do this we defined a Java interface named Ialgorithm. This interface defines methods for initialization, sending data, receiving data and also performing a computation based on data already received. Any Java class that implements this interface can be deployed to a computational resources. This allows us the flexibility to test the responsiveness of the load balancing algorithm for situations in which the computational resources have different algorithms deployed to them. This approach makes it easy to simulate a map/reduce framework, in which the control node is running one algorithm, the map nodes are running a second algorithm and the reduce node(s) are running a third algorithm. Since most of the algorithm work done so far has used a map/reduce paradigm, the idea of subclassing from the Ialgorithm interface has proven very useful in quickly setting up a simulation. Algorithms are packaged as Java jar files, so that it is easy to load them at compile time or runtime.

Method

The system architecture for our simulation work is shown in Fig. 1.

The simulation has three operational phases: Initialization, run and termination. When the simulation is started it reads a fixed configuration file. From this file it learns the names of the jar files that will be used by itself, the load balancer, the map nodes and the reduce node(s). The control node sends a message to each node, commanding the node to load and run the appropriate jar file. The control node then performs its own initialization (such as the generation of crypto key material) and then commands the load balancer to gather initialization statistics. The load balancer then uses an algorithm named Init (describe below) to assign each computational node a score.

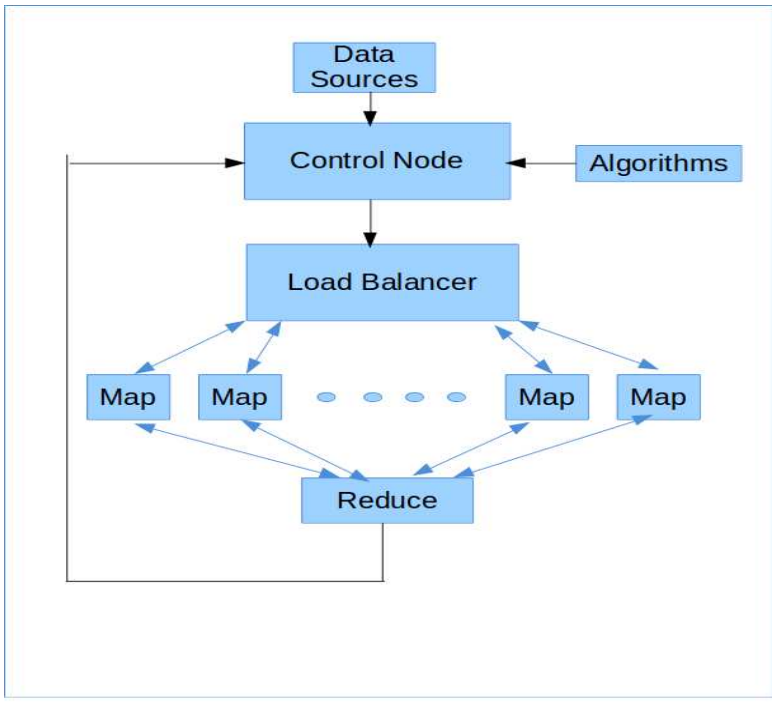

Fig. 1. System architecture

It sorts these scores from lowest to highest and delivers the sorted list to the control node. The control node then uses this list to command each computational node to perform its own initialization. Once the control node has received acknowledgments from each computational node that initialization is done, the control node reads the data source information from the configuration file and connects to the corresponding data source. Note that the control node may implement an immediate connection protocol, in which all data sources are connected during initialization, or it my implement a deferred connection protocol, in which some data sources can be added during runtime. Once the control node has finished connecting to data sources, it transitions to run mode.

Run mode is straightforward. The control node queries each data source in a round robin fashion to see if data is available. If any data is available it reads that data. It then commands the load balancer to gather runtime data from each compute node. The load balancer then uses its algorithm Run, described below, to sort the runtime information into an array of triples of the form <NodeId, Load, HowMany>. Here NodeId is the identifier for each Map or Reduce node; Load is the load factor on that node; and, HowMany is the maximum number of data points that can be accepted by that node. The load balancer also collects the round trip time of the command/response sequence to each node and includes that information with the array. The array is then sent to the control node. The control node then typically implements a greedy allocation algorithm for the data points in its queue. The maximum number possible is sent to the least busy node; if there is any data left then the maximum amount of data is sent to the second least busy node and so forth. It may be the case that the greedy algorithm is unable to distribute all the newly arrived data because all the nodes are too heavily loaded. In this case, the control node queues up the remaining data points for the next iteration. Once the data has been received by the Map and Reduce nodes, each node processes it according to the selected algorithm. This may involve Map nodes sending partial data to Reduce nodes and also for Reduce node(s) to send partial data back to the control nodes. Then transmissions are handled using asynchronous I/O, since the amount of data in these packets is typically very small.

The termination phase can be triggered by two events. If all data sources report that they have sent all available data, the control node closes the connections to the data sources and sends a termination command to all nodes. The termination phase can also be entered if there is an iteration counter present in the configuration file. Such a counter indicates the maximum number of

iterations of the simulation may run. Once this counter is reached, the control node again close all data connections and sends a termination command to all nodes. Note that the load balancer is not used for termination processing. Once all the Map nodes have completed processing, they will send their final data, along with a termination command, to the Reduce node(s). Once the Reduce node(s) have finished all their processing, they will send the final data to the control node and append a termination flag to indicate that all processing is done. At this point it is the responsibility of the control node to handle the results according to the configuration file. This may involve generation of graphs, construction of tables, logging, or any other supported termination action's from each node: CPU load average L, percentage of available memory M, number of connections C and the time it takes the node to read a block (4096 bytes) from global storage R. The score for that node is then computed as $C*(k_1*R + k_2*M + k_3*L)$, where k_1, k_2 and k_3 are configurable constants. This approach is sometimes called connection-based load balancing, because the more data connections a node has, the higher its score will be. The algorithm Run requests the following information from each node: L, M, C; the amount of time it takes to read a small block (512) bytes from a network socket; and the number of available data slots in the node D. It is assumed that each computational node implements a slot allocation policy, where newly arrived data is placed into an array of slots. As data is processed by the computational algorithm, slots are emptied. Thus we can write D = TD-TI, where TD is the total number of slots and TI is the number f slots currently in use. It is certainly not necessary to use a slot based allocation policy, but simple experiments have shown that this is the best memory allocation strategy. When each node starts up it requests a single allocation large enough to hold all slots. If, instead, each node were to allocated memory when needed and free memory when not needed then the memory utilization percentage M would fluctuate wildly and it would be very unlikely to achieve optimum allocation. The run algorithm uses the score of $C*(k_1*S + k_2*M + k_3*L)$ and provides D as the value of HowMany.

Results

In evaluating the results of the simulation, we compare computation time and other factors versus the naive D/N allocation strategy discussed in the first part of this document. In particular comparative data will be presented for scenarios with 10-100 Vms allocated to Map nodes, 1 VM allocated for both the control algorithm and the load balancer and 1 VM allocated for the Reduce node. We will compare total time of execution (Fig. 2) and also the average time spent in a blocked state waiting for data (Fig. 3). These figures clearly show that the load balancing approach provides a significant improvement in resource allocation for dynamic data. In our simulations data delivery (size and timing) is modeled as random, although alternative scenarios using different data delivery strategies yield almost identical results.

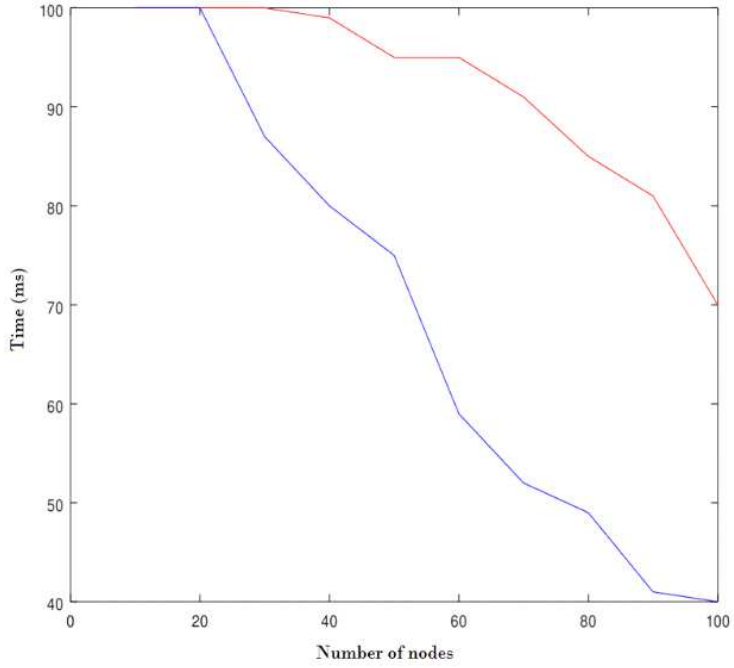

Fig. 2. Normalized execution time for D/N (red) and LB (blue) algorithms

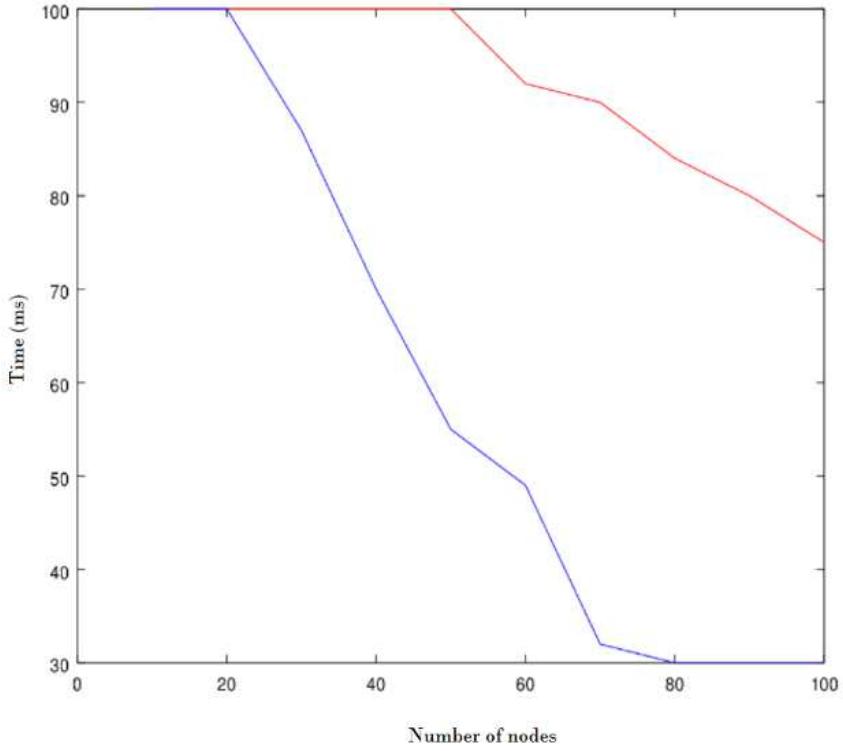

Fig. 3. Average blocking time

The obtained result enrich previous findings on the role of graph partitioning of (Khayyat *et al*., 2013) in which using single method is considered to be insufficient for minimizing end-to-end computation. This is usually apply when the data is very large or the runtime behavior of the algorithm is unknown, an adaptive approach is needed. The result also provide more support to the work of previous work in (Hao *et al*., 2009) who argued about the effectiveness of a basic network infrastructure to migrate virtual machines across multiple networks without losing service continuity. Authors in Hao *et al*. (2009) assumed that providing a mechanisms using a network-virtualization architecture that relies on a set of distributed forwarding elements with centralized control would help enhancing dynamic cloud-based services. As such, the present work extend the previous efforts on the potential of Map/Reduce in cloud related applications.

Conclusion

In this study we have presented a novel method for distribution and processing of dynamic data, particularly in the case of computationally difficult algorithms that take a long time to process. We have used a load balancer algorithm to mediate the distribution of data. By every performance measure this approach performs better and a naive strategy of distributing the data evenly over all compute nodes. We have also developed a simulation framework that is easily extensible to simulations of other cloud-based algorithms. Future works can further study the feasibility of the proposed method in various cloud related systems. This include extending the simulation environment to include additional settings.

Ethics

This article is original and contains unpublished material. The corresponding author confirms that all of the other authors have read and approved the manuscript and no ethical issues involved.

References

Bharadwaj, V., 1996. Scheduling Divisible Loads in Parallel and Distributed Systems. 1st Edn., John Wiley and Sons, Los Alamitos, ISBN-10: 0818675217, pp: 292.

Cardellini, V., M. Colajanni and S.Y. Philip, 1999. Dynamic load balancing on web-server systems. IEEE Internet Comput., 3: 28-39. DOI: 10.1109/4236.769420

Chandra, A., W. Gong and P. Shenoy, 2003. Dynamic resource allocation for shared data centers using online measurements. Proceedings of the International Workshop on Quality of Service, Jun. 02-04, Springer-Verlag, Berkeley, pp: 381-398.

Darema, F., 2004. Dynamic data driven applications systems: A new paradigm for application simulations and measurements. Proceedings of the International Conference on Computational Science, (CCS' 04), Springer, Berlin, Heidelberg, pp: 662-669. DOI: 10.1007/978-3-540-24688-6_86

Darema, F., 2005. Grid computing and beyond: The context of dynamic data driven applications systems. Proc. IEEE, 93: 692-697. DOI: 10.1109/JPROC.2004.842783

Hao, F., T. Lakshman, S. Mukherjee and H. Song, 2009. Enhancing dynamic cloud-based services using network virtualization. Proceedings of the 1st ACM Workshop on Virtualized Infrastructure Systems and Architectures, Aug. 17-17, ACM., Barcelona, Spain, pp: 37-44. DOI: 10.1145/1592648.1592655

Khayyat, Z., K. Awara, A. Alonazi, H. Jamjoom and D. Williams et al., 2013. Mizan: A system for dynamic load balancing in large-scale graph processing. Proceedings of the 8th ACM European Conference on Computer Systems, Apr. 15-17, ACM, Prague, Czech Republic, pp: 169-182. DOI: 10.1145/2465351.2465369

Ranganathan, K., A. Iamnitchi and I. Foster, 2002. Improving data availability through dynamic model-driven replication in large peer-to-peer communities. Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid, May 21-24, IEEE Xplore Press, pp: 376-376. DOI: 10.1109/CCGRID.2002.1017164

Shirazi, B.A., K.M. Kavi and A.R. Hurson, 1995. Scheduling and Load Balancing in Parallel and Distributed Systems. 1st Edn., Wiley, Los Alamitos, ISBN-10: 0818665874, pp: 520.

Yin, S., S.X. Ding, A.H. Abandan Sari and H. Hao, 2013. Data-driven monitoring for stochastic systems and its application on batch process. Int. J. Syst. Sci., 44: 1366-1376. DOI: 10.1080/00207721.2012.659708