Available online at www.sciencedirect.com

SciVerse ScienceDirect

Procedia Engineering 38 (2012) 2172 – 2178

Procedia Engineering

www.elsevier.com/locate/procedia

International Conference on Modeling, Optimization and Computing (ICMOC 2012)

Improved Task graph-based Parallel Data Processing for Dynamic Resource Allocation in Cloud

A. Ajitha[a], D. Ramesh[b]

[a]*M. E. Software Engineering, Anna University of Technology, Tiruchirappalli-620024, India*
[b]*Assistant Professor, CSE Department, Anna University of Technology, Tiruchirappalli-620024, India*

Abstract

In recent years large-set parallel data processing has gained quantum as one of the predominant applications of Infrastructure-as-a-Service (IaaS) clouds. Data processing frameworks like Google's MapReduce and its open source implementation Hadoop, Microsoft's Dryad and so on are currently in use for parallel data processing in cloud-based companies. But the problem with them is that they are designed for homogeneous environments like clusters and disregard the dynamic and heterogeneous nature of a cloud. As a result, allocation and de-allocation of compute nodes at runtime is ineffective thereby increasing processing time and cost. In this paper we present our approach towards parallel data processing exploiting dynamic resource allocation in IaaS clouds. Our architecture ensures parallel data processing using Directed Acyclic task graph. To reduce the latency and to improve throughput, load balancing is introduced in the architecture. Incoming jobs are divided into tasks that are assigned to different types of virtual machines that are dynamically instantiated and terminated during job execution.

Keywords: Parallel data processing; IaaS Cloud; Task graph; Heterogeneous environment; Load balancing

1. Introduction

Today most of the applications are Internet-based with millions of users and handling huge amounts of data. Representatives of such applications are search engines, space data analysis, seismic simulation, Natural Language Processing, social networks etc. As traditional data processing system proves to be unsuitable in this scenario, Cloud companies like Google, Microsoft, or Yahoo have developed their

1877-7058 © 2012 Published by Elsevier Ltd. Open access under CC BY-NC-ND license.
doi:10.1016/j.proeng.2012.06.261

customized data processing frameworks like MapReduce[1], Dryad[4] and Map-Reduce-Merge[6] respectively, for serving their million clients in a cost-effective way. MapReduce is apt for applications with large data with less number of tasks, where as Dryad suits big data and many tasks[7].
However, these frameworks are designed to suit cluster environments where the number and type of compute resources are pre-determined. Also, the compute nodes allotted to jobs cannot be changed once the job initiates. These assumptions make the existing frameworks less flexible for cloud environment wherein resources are allocated on demand. In an Iaas cloud, new virtual machines are allocated dynamically at any time, to the job, on the run. VMs which are no longer in use can automatically be de-allocated as job execution follows a pay-per-use model.

This paper is organised as follows. Section 2 describes the proposed system and its architecture. Section 3 explains the way in which a job is represented and how task graph is derived. We have concluded with Section 4.

2. Related Work

A framework called Nephele is the first existing parallel data processing framework for cloud [5]. Nephele is the first data processing framework to include the possibility of dynamically allocating/deallocating different compute resources from a cloud in its scheduling and during job execution. However, in the future, this framework aims at improving the ability to adapt to resource overload or underutilization during the job execution automatically. We have adapted this as our work in this paper.

Another such framework is Apache Hadoop, the open source version of Google's MapReduce, designed to run data analysis jobs on a large amount of data, which is expected to be stored across a large set of clusters. In cluster environment, the available compute resources are essentially considered to be a fixed set of homogeneous machines.

The Pegasus framework has been designed for mapping complex scientific workflows onto grid systems [10]. Similar to Nephele framework, Pegasus lets its users describe their jobs as a DAG with vertices representing the tasks to be processed and edges representing the dependencies between them.

Swift[11], is a system that combines a novel scripting language called SwiftScript with a powerful runtime system based on CoG Karajan and Falkon[12] to allow for the concise specification, and reliable and efficient execution, of large loosely coupled computations.

3. Proposed System

In this paper, we propose a data processing framework which fits well in a cloud for efficiently parallelizing the incoming set of tasks using large data. In this, a job initiates on one VM and on the go, based on the number and complexity of its subtasks, further VMs are allocated and de-allocated. Figure 1 presents the architecture of our framework.

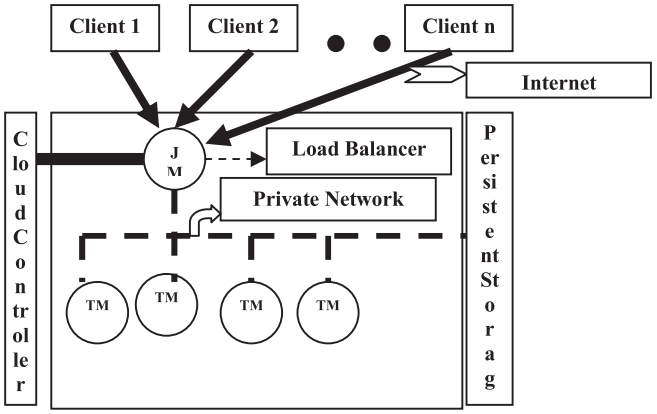

Fig. 1: Architecture of the proposed system

3.1. Job Manager (JM)

This is the backbone of the architecture. It receives job from several clients, divides them into its constituent tasks through user annotations and assigns suitable number and types of virtual machines to them. Also, it co-ordinates their execution. The instantiation and termination of virtual machines (their allocation and de-allocation based on the job execution phase) are performed with the help of the cloud controller, an interface between the cloud provider and the application developer. Job Manager is run on a single VM. Based on the complexity of the tasks, additional VMs are allocated.

3.2 Task Manager(TM)

Task Manager is run by a set of VMs(instances). When a new Task Manager is added, it registers itself with the Job Manager. It receives one or more tasks from Job Manager at a time, executes them and reports about their completion or error otherwise.

Both the Job Manager and the Task Manager must be able to access the persistent storage.

3.3 Load Balancer

In order to perform scheduling of tasks to VMs in a cost-effective way, we have used a load balancing algorithm named Join-Idle-Queue algorithm. It introduces an idle queue between the Job Manager and the Task Manager. Whenever a Task Manager becomes idle, it joins the queue.

When a job approaches the Job Manager, it fetches the first TM in the queue and allocates the job to it. Further details about the algorithm are provided in [2]. This algorithm avoids the Job Manager from enquiring every Task Manager for its availability. Hence, this algorithm reduces communication overhead and thereby improves throughput of the data processing system.

4. Job Description, Scheduling and Execution

4.1 Job Graph:

Several dependent tasks of a job can be represented using directed acyclic graph (DAG) [4]. Every vertex in the graph denotes a task and edges denote the communication path, ie, output of one task is input to the other.

When a job is submitted to the Job Manager, it constructs the Job Graph. Job Graph is the logical computation graph that is automatically mapped onto physical resources by the runtime. Each vertex has the sequential code of a task in a job. It is the responsibility of the scheduler to co-ordinate the simultaneous execution of the tasks. Application developers no more need to write parallel executing codes. Figure 2 shows a simple Job Graph.

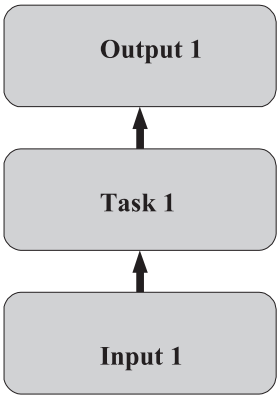

Fig. 2: Simple Job Graph

It is a Job Graph for a simple task with task 1 working on input 1 and producing output 1. The URL of the tasks can be mentioned in the graph when necessary.

4.2 Channel Types:

For each edge connecting two vertices of the Job Graph, the user can determine a channel type. Before executing a job, all edges of the original Job Graph have to be replaced by at least one channel of a specific type. The channel type dictates how records are transported from one subtask to another at runtime.

Currently used channel types are network, in-memory and file channels [4][5]. The type of channel chosen for the whole job determines the cost of execution of the job. In the runtime, channel type can be changed.

1. Network Channel:
   Network channel allows data to be transferred over TCP connection. This enables pipelined processing. Hence, any two subtasks connected by a network channel can run on two VMs. But the cost of using this channel is higher than the other channels.

2. In-memory Channel:
   Similar to the network channel, here also pipelined processing is enabled. However instead of using TCP connection, here the VMs' main memory is used for data storage and transfer. So any two subtasks communicating through this type should be on the same VM.

3. File Channel:
   A file channel allows two subtasks to exchange records via the local file system. The records of the producing task are first entirely written to an intermediate file and later read into the consuming subtask. Hence those subtasks should run on the same VM.

4.3 Execution Graph:

Before a job is executed, the Job Graph is converted into its equivalent Execution Graph. Unlike the Job Graph, the Execution Graph gives details about the number of subtasks of the job and the number and type of VMs required for the subtasks. Edges in the Job Graph are replaced by channel types. Unlike the abstract Job Graph, this provides complete information about scheduling, execution and parallelization involved in the job execution. Also it provides mapping of tasks to VMs.

Execution graph's structure resembles a graph with two different levels of details, an abstract and a concrete level. While the abstract graph describes the job execution on a task level (without parallelization) and the scheduling of VM allocation/de-allocation to the task, the concrete, more fine-grained graph defines the mapping of subtasks to the VMs and the communication channels between them. Figure 3 shows the Execution Graph for the Job Graph in Figure 2.

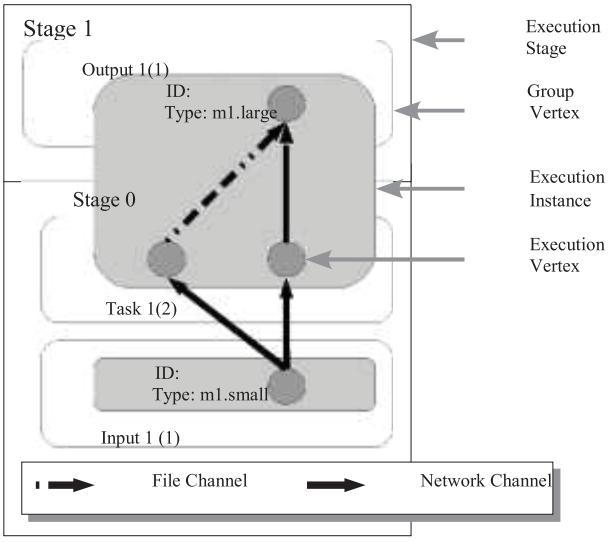

Fig. 3: Execution Graph for Job Graph in Fig. 2.

For every vertex of the original Job Graph there exists a so-called *Group Vertex* in the Execution Graph. Group Vertices also represent constituent tasks of a job but are not seen as executable units. Sometimes, output of two subtasks may be needed as an input to a consuming task. Such tasks are placed in a common *Execution Stage*. This is to ensure that VMs are available before a subtask begins. Before the processing of a new stage, all intermediate results of its preceding stages are stored in a persistent manner. Subtasks are represented by *Execution Vertices* in the Execution Graph. They are the actual executable job unit. To simplify management, each Execution Vertex is always controlled by its corresponding Group Vertex. The characteristics of the requested VMs can be adapted to the demands of the current processing phase. To reflect this relation in the Execution Graph, each subtask must be mapped to a so-called *Execution Instance*. An Execution Instance is defined by an ID and an instance type representing the hardware characteristics of the corresponding VM.

Before processing a new Execution Stage, the scheduler
collects all Execution Instances from that stage and tries to replace them with matching cloud instances. If all required instances could be allocated the subtasks are distributed among them and set up for execution.

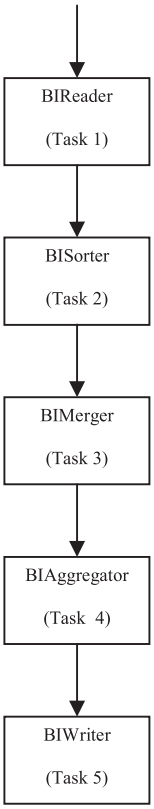

Fig.4: Job Graph for a sample job

Figure 4 depicts the Job Graph for a job. The job is executed in two steps. Given a set of random integer numbers, the first step is to determine the k smallest of those numbers. The second step is to calculate the average of these k smallest numbers.

These two steps of the sort/aggregation job are divided into five tasks. The first task namely, BigIntegerReader scans the input files and reads a set of integer numbers at random. The output of this task will be records with numbers. These records are given as input to the second task, BigIntegerSorter. This task performs quick sort on the incoming records and sends out sorted records. These sorted records are fed to the third task, BigIntegerMerger, which returns the k smallest numbers. The fourth task BigIntegerAggregator sums up these numbers and calculates their average and sends to BigIntegerWriter, the final task, which writes the value to main memory.

Conclusion

In this paper we have presented our architecture for efficient parallel data processing in cloud environments by exploiting the dynamic resource provisioning offered by today's IaaS clouds. Cloud has become a crucial platform for almost all of the web-based applications. So we believe that our work to ensure efficient processing of data in cloud adds an essential functionality to IaaS cloud.

To ensure automatic adaptation to under- or over-utilization of resources, we have introduced load balancing concept. In future, we have planned to evaluate its performance and compare it with the existing data processing frameworks.

References

[1] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In OSDI'04: Proceedings of the 6th conference on Symposium on Operating Systems Design & Implementation, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

[2] Yi Lu, Qiaomin Xie, Gabriel Kliot, Alan Geller, James R. Larus, Albert Greenberg. Join-Idle-Queue: A Novel Load Balancing Algorithm for Dynamically Scalable Web Services. Performance Evaluation Journal 68(11), Nov. 2011.

[3] R.Eswari, S.Nickolas. Expected Completion Time based Scheduling Algorithm for Heterogeneous Processors. In International Conference on Information Communication and Management, IPCSIT vol.16 (2011), IACSIT Press, Singapore.

[4] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly.Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks. In EuroSys '07: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007, pages 59–72, New York, NY, USA, 2007. ACM.

[5] D. Warneke and O. Kao. Nephele: Efficient Parallel Data Processing in the Cloud. In MTAGS '09: Proceedings of the 2nd Workshop on Many-Task Computing on Grids and Supercomputers, pages 1–10, New York, NY, USA, 2009. ACM.

[6] H. chih Yang, A. Dasdan, R.-L. Hsiao, and D. S. Parker. Map-Reduce-Merge: Simplified Relational Data Processing on Large Clusters. In SIGMOD '07: ACM SIGMOD International conference on Management of data, pages 1029–1040, New York, NY, USA, 2007. ACM.

[7] I. Raicu, I. Foster, and Y. Zhao. Many-Task Computing for Grids and Supercomputers. In Many-Task Computing on Grids and Supercomputers, 2008. MTAGS 2008. Workshop on, pages 1–11, Nov. 2008.

[8] The Apache Software Foundation. Welcome to Hadoop! http://hadoop.apache.org/, 2009.

[9] T. White. Hadoop: The Definitive Guide. O'Reilly Media, 2009.

[10] E. Deelman, G. Singh, M.-H. Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, G. B. Berriman, J. Good, A. Laity, J. C. Jacob, and D. S. Katz. Pegasus: A Framework for Mapping Complex Scientific Workflows onto Distributed Systems. Sci. Program., 13(3):219–237, 2005.

[11] Y. Zhao, M. Hategan, B. Clifford, I. Foster, G. von Laszewski, V. Nefedova, I. Raicu, T. Stef-Praun, and M. Wilde. Swift: Fast, Reliable, Loosely Coupled Parallel Computation. In Services, 2007 IEEE Congress on, pages 199–206, July 2007.

[12] I. Raicu, Y. Zhao, C. Dumitrescu, I. Foster, and M. Wilde. Falkon: a Fast and Light-weight tasK executiON framework. In SC '07: Proceedings of the 2007 ACM/IEEE conference on Supercomputing, pages 1–12, New York, NY, USA, 2007. ACM.