Using the BSP model on Clouds

Daniel CORDEIRO\textsuperscript{a}, Alfredo GOLDMAN\textsuperscript{a,1}, Alessandro KRAEMER\textsuperscript{b}, and Francisco PEREIRA JUNIOR\textsuperscript{b}
\textsuperscript{a}Department of Computer Science, University of São Paulo, Brazil
\textsuperscript{b}Federal Technological University of Paraná, Brazil

Abstract. Nowadays the concepts and infrastructures of Cloud Computing are becoming a standard for several applications. Scalability is not only a buzzword anymore, but is being used effectively. However, despite the economical advantages of virtualization and scalability, some factors as latency, bandwidth and processor sharing can be a problem for doing Parallel Computing on the Cloud.

We will provide an overview on how to tackle these problems using the BSP (Bulk Synchronous Parallel) model. We will introduce the main advantages of CGM (Coarse Grained Model), where the main goal is to minimize the number of communication rounds, which can have an important impact on BSP algorithms performance. We will also briefly present our experience on using BSP in an opportunistic grid computing environment. Then we will show several recent models for distributed computing initiatives based on BSP. Finally we will present some preliminary experiments presenting the performance of BSP algorithms on Clouds.

Keywords. BSP, CGM, Cloud Computing

Introduction

The IT community has embraced the Cloud Computing model [1] as the solution for their scalability and provisioning problems. Commercial Cloud Computing providers allow easy access to a large pool of computational resources. Cloud Computing is a type of Utility Computing model [2], where the application can choose, on demand, the type and the amount of resources needed at any given time and gets charged only by the resources effectively used (like a metered service).

The academic and scientific communities were also attracted by many of the benefits brought by Cloud Computing platforms (virtualization, elasticity of resources, elimination of hardware setup and maintenance costs, etc.). However, as Gupta and Milojicic [3] points out, problems as poor network performance, performance variation and the overhead caused by the virtualized operating system configures new challenges for the execution of High Performance Computing (HPC) applications on this kind of platform.

These drawbacks can have a huge impact on the performance of some applications, specially ones that analyze very large data sets (typically the case for e-Science applica-
If the application presents good data parallelism, popular programming models for Cloud Computing such as the MapReduce model [4] achieves very good performance. If not, some recent research [5–10] have proposed more sophisticated ways to achieve better performances on Cloud Computing platforms.

These papers are based on a interesting model, namely, Bulk Synchronous Parallel (BSP) [11], proposed by Valiant in 1990. The main motivation was to propose a simple model which could be used to represent several parallel computers, without taking into account technical details and at the same time being powerful enough to serve as a good abstraction to write efficient algorithms. The great innovation was to split computation from communication, providing a strong abstraction where algorithms could be conceived. At the time this model was proposed, it make all the sense, as there were plenty of different communication topologies and the abstraction allowed to focus on the algorithmic part and not on the details of each computer. Latter on, with the advent of fast switches, it was possible to abstract the topology and to have direct point to point communication among all nodes of a parallel computer.

More or less at the same time, Dehne et al. [12] proposed a theoretical model, Coarse Grained MultiComputer where there was no connection to actual machines, but also captured the basics of parallel computing, the memory limitation on each node and the communication limitations. As on BSP, the communications are modeled by h-relations [13], which imply in synchronous steps. However, the main focus was on the algorithm development, providing efficient algorithms where the number of h-relations are minimized at the same time that the computation is balanced.

Recently, there were also some interesting proposals showing the interest of the BSP model for very specific environments like GPUs [14] and for more close to the hardware details, like multi-core [15]. In this chapter we explore the possibilities of using BSP for Cloud / Multi Clouds environments, where as before, it is not straightforward to model the communication, as virtual machines can be in the same computer, in different computers or even in computers connected by the Internet.

In this chapter, we present the BSP/CGM model as an alternative for modeling parallel applications on Cloud Computing platforms. In Section 1 we describe the BSP model, and on the next section the CGM model. In Section 3 we show how this model have been successfully applied on opportunistic Grid Computing systems. We provide in Section 4 a brief description of some frameworks based on BSP. We present some experiments related to virtual machines allocation and scalability on Section 5 and the we conclude the chapter.

1. The BSP model

The Bulk Synchronous Parallel (BSP) model was first introduced by Leslie Valiant [11]. The main goal of this model was to provide a bridging model which could represent different architectures sufficiently well, without considering all the hardware details.

To capture the essential characteristics of a machine, the BSP model is defined as a combination of three attributes:

- a set of virtual processors, each associated to a local memory;
- a router, that delivers the messages in a point-to-point manner;
- a synchronization mechanism for all or a subset of processors.
The computational is organized as a sequence of supersteps. Each superstep consists of a computational phase, where local independent computations runs in parallel and a communication phase, where the communications of data involve all the processors in a personalized all-to-all communication. Between two supersteps there is a synchronization barrier, that guarantees the simultaneous start of all processors for the next step.

Having global synchronizations at the end of each superstep may be seem seen as cumbersome, but there are significant advantages of doing so, especially for large scale distributed systems. First of all, to find a good way to overlap communication with computation, a precise knowledge of the hardware is essential. This means that a fine tuning of the application must be redone every time there is a chance in the hardware. Secondly, global synchronization points can be used straightforwardly to provide checkpoints to store consistent partial states. Thirdly, there are several works exploring how the h-relation can be effectively done, providing efficient implementations on a large diversity of scenarios [13].

2. The CGM model

Dehne et al. [12] introduced a slightly different model for parallel programs that, in fact, can be considered as a special case of the BSP model. The Coarse Grained Multicomputers (CGM) model defines how a problem of size \( n \) will be executed by \( p \) processors each with a local memory limited to \( O(n/p) \). In this model we assume that the size of the problem is considerably larger than the number of available processors.

Like in BSP, a CGM algorithm consists of a sequence of rounds with distinct phases for local computation and global communication. Usually, each processor will apply the best sequential algorithm for the data available locally. The goal of CGM algorithm design is to minimize the number of supersteps and the amount of local computation [16].

However, the main effort on CGM was on providing a new abstract model, better suited to the reality rather than the PRAM. As the PRAM model was too powerful, the designed algorithms could not be implemented in practice with the same performance. CGM with the its global communications, which are expensive, and should be minimized is closer to the reality.

So, the main CGM’s contribution was on providing efficient and realistic algorithms which could be implemented directly on real machines, using BSP or not. It is important to point out that there is a lot of expertise on algorithms done on CGM, for instance with the notion of high probability [17], where the algorithms are guaranteed to have a good performance (reasonable number of steps) in most of the cases.

2.1. Some algorithms

We present on Table 2.1 a short list with several algorithms designed using the CGM model, where interesting complexity results are also presented. The list is not exhaustive. It is important to notice that some of the algorithms can be used on several practical problems.
Table 1. BSP/CGM Algorithms suggested for experiments on Cloud Computing.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Local computational cost</th>
<th>Communication cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Longest Repeated Suffix Ending at Each Point in a Word [18]</td>
<td>$O\left(\frac{n^2}{p}\right)$</td>
<td>$O(2p - 3)$</td>
</tr>
<tr>
<td>Two-Dimensional Parallel Pattern Matching [19]</td>
<td>$O\left(\frac{n^2}{p}\right)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Graph Planarity Testing [20]</td>
<td>$O(\log(p) \frac{n}{p})$</td>
<td>$O(\log(p))$</td>
</tr>
<tr>
<td>Maximum Weight Matching in Trees [21]</td>
<td>$O(\log(p) \frac{n}{p})$</td>
<td>$O(\log(p))$</td>
</tr>
<tr>
<td>Dynamic Programming for Solving the String Editing Problem [22]</td>
<td>$O(\log(m) \frac{n}{p})$</td>
<td>$O(\log(p))$</td>
</tr>
<tr>
<td>Parallel Similarity [16]</td>
<td>$O\left(\frac{n^2}{p}\right)$</td>
<td>$O(p)$</td>
</tr>
<tr>
<td>All-Substrings Longest Common Subsequence [23]</td>
<td>$O\left(\frac{n^2 \alpha}{p}\right)$</td>
<td>$O(\log(p))$</td>
</tr>
<tr>
<td>Maximal Independent Set Problem [24]</td>
<td>$O\left(\frac{m \log p}{p}\right)$</td>
<td>$O(\log(p))$</td>
</tr>
<tr>
<td>Knapsack [25]</td>
<td>$O\left(\frac{m}{p}\right)$</td>
<td>$O(p)$</td>
</tr>
<tr>
<td>Maximum Matching in Convex Bipartite Graphs [26]</td>
<td>$O(\left(\frac{V}{p}\right) \log(n) \log(p))$</td>
<td>$O(\log(p))$</td>
</tr>
<tr>
<td>Matrix Chain Product [27]</td>
<td>$O\left(\frac{n^3}{p}\right)$</td>
<td>$O(p)$</td>
</tr>
<tr>
<td>Transitive Closure [28]</td>
<td>$O\left(\frac{n^3}{p}\right)$</td>
<td>$O\left(\frac{3}{p}\right)$</td>
</tr>
<tr>
<td>Biological Sequence Comparison [29]</td>
<td>$O\left(\frac{n^3}{p}\right)$</td>
<td>$O(p)$</td>
</tr>
<tr>
<td>Approximation Hitting set Algorithm for Gene Expression Analysis [30]</td>
<td>$O\left(\frac{n^3}{p}\right)$</td>
<td>$O(k)$</td>
</tr>
<tr>
<td>Integer Sorting [31]</td>
<td>$O\left(\frac{n}{p}\right)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>List Ranking [32]</td>
<td>$O\left(\frac{n}{p}\right)$</td>
<td>$O(\log(p))$</td>
</tr>
<tr>
<td>Euler Tour [32]</td>
<td>$O\left(\frac{V}{p}\right)$</td>
<td>$O(\log(p))$</td>
</tr>
<tr>
<td>Connected Components and Spanning Tree [32]</td>
<td>$O\left(\frac{E}{p}\right)$</td>
<td>$O(\log(p))$</td>
</tr>
<tr>
<td>Lowest Common Ancestor [32]</td>
<td>$O\left(\frac{V}{p}\right)$</td>
<td>$O(\log(p))$</td>
</tr>
<tr>
<td>Open Ear Decomposition and Biconnected Components [32]</td>
<td>$O\left(\frac{V}{p}\right)$</td>
<td>$O(\log(p))$</td>
</tr>
<tr>
<td>Chordal Graph Recognition [32]</td>
<td>$O\left(\frac{V}{p}\right)$</td>
<td>$O(\log(n) \log(p))$</td>
</tr>
</tbody>
</table>

2.2. BSP and CGM in practice

Several implementations of the BSP model have been developed since the initial proposal by Valiant. They provide to the users full control over communication and synchronization in their applications. BSP implementations developed in the past include: Oxford’s BSpLib [33], JBSP [34], a Java version, PUB [35] and BSP-G [36]. A comparison of several libraries can be found in [37]. More recent examples include jMigBSP [38] and an object oriented BSP library for multi-core programming [39].

On the CGM side there are only few libraries like [40, 41].

3. BSP model on opportunistic grid systems

One example of how the BSP/CGM model can be applied for the execution of large distributed applications came from research of opportunistic Grid Computing platforms.
InteGrade [42] is a Grid Computing system aimed at commodity workstations such as household PCs, corporate employee workstations, and PCs in shared laboratories. It uses the idle computing power of these machines to perform useful computation.

The BSP/CGM model was successfully applied by the InteGrade middleware to give several advantages for application developers on this platform. More than providing a simple (yet powerful) programming model for the developers, through the use of a standard API compatible with the Oxford’s BSPlib [43], the use of the BSP/CGM model allowed the development of very efficient algorithms, with satisfactory performance even on opportunistic grids [44]. Also, the regular structure of BSP algorithms allowed also the implementation of checkpoint-based rollback recovery for BSP parallel applications [45].

4. BSP-like frameworks

With the new opportunities brought by Cloud Computing, large-scale applications that were restricted before to large research centers and super computers owners, can now be executed with modest investments on infrastructure and maintenance. Smaller operational costs, allied with the easy access to a huge pool of computational resources that can be rented on-demand, are allowing the creation of new kinds of highly scalable algorithms and applications. These applications (e-Science applications, Web 2.0 sites with large number of users, etc.) typically generates and process huge quantities of data. One of the main challenges on Cloud Computing now is how to deal with problems on how to store, manipulate and analyze this huge amount of data.

Performance, scalability and reliability are classical concerns related to high performance computing that must be reconsidered when using Cloud Computing platforms. The features presented on Section 1 motivated some researchers and developers to consider the BSP programming model as an alternative to deal with these concerns.

Several work using BSP as a programming model for Cloud Computing first appeared on the context of large graph processing applications. These applications naturally appears on problems from different areas of knowledge (Computer Science, Biology, Engineering, etc.) and uses graphs to represent relations between different entities, such as links on web pages, users relations on social networks, flow of goods, order of DNA fragments, etc.

In this section we will present and analyze frameworks for Cloud Computing that are currently being used to solve large graph processing problems. These problems represent a very broad class of applications which are relevant to the applications typically deployed on Cloud Computing platforms. We focus on frameworks capable of dealing with huge graphs, with number of edges and vertices on the order of $10^9$.

**Apache Hadoop** The Apache Hadoop framework [46] was the first framework considered to be used on graph processing applications due to the popularity of the MapReduce programming model on large-scale data processing applications. The MapReduce model was first introduced by Google [4] and was implemented and released as a open source framework by the Apache Foundation.

In order to facilitate the processing of large volumes of data, the framework offers a simple mechanism to build data processing applications following the MapReduce model. This mechanism is composed of a simple API, in conjunction with a reliable mechanism...
to distribute all the data between the available nodes (the Hadoop Distributed File System, or HDFS).

The MapReduce model and the Apache Hadoop implementation proved to be an efficient way to develop distributed systems and is now the de facto programming model for applications that processes large volumes of data. However, the MapReduce model is not well-suited [5, 10, 47, 48] to implement graph algorithms with large input graphs mainly because of the following reasons:

1. MapReduce programs are specified using input data represented as tuples of key-value pairs. This abstraction is not appropriate to model graph problems and impose significant effort on adapting the existing graph algorithms;
2. most of the graph algorithms are iterative algorithms. Mapping these algorithms to the MapReduce model involves creating programs that does multiple iterations [49]. In the original MapReduce model these iterations are completely independent from each other, so an extra I/O overhead is caused by data serialization and reading between the iterations. This overhead is not negligible when dealing with large graphs;
3. there is no support by the MapReduce model for direct communication between the processing nodes.

These limitations with the MapReduce model and the necessity of large scale graph processing systems motivated the creation of new models for this kind of applications. The first system to use the BSP as an alternative model for Cloud Computing computation was Pregel [5], introduced by Google. This work inspired the development of several frameworks for large scale graphs with different characteristics: Apache Hama [47], Apache Giraph [48], GPS [50], GoldenOrb [51], GraphLab [52], Spark project [53], Proebus [54], HipG [55], Piccolo [56], Haloop [10], Ciel [57], among others. All these projects share the following features:

- performance efficient: the framework allows iterative applications to run on graphs with large number of vertices and edges;
- fault tolerant: dealing with a large data set and executing on a large number of nodes requires that the systems must deal with the possibility of multiple nodes failing simultaneously;
- graph oriented programming model: the graph abstraction must be a first-class citizen of the programming model

In the next sections we will present and discuss the most important frameworks currently being for large-scale graph processing.

### 4.1. Google Pregel

The Google Pregel system was first introduced by Malewicz et al. [5] on 2010. They argue that by using the BSP programming model they were able to build a fault-tolerant, scalable platform. Their flexible API was proven to be powerful to express graph algorithms and to run efficiently on the several clusters. Their approach centers the computation on the vertices of the graph and their execution flow is expressed by BSP primitives.

The input data of a Pregel application can be given using customized formats, but usually is given by a text file describing the graph. Each vertex of the graph has a unique
id, an associated value and a list of output weighted edges. The computation is organized using a Master/Slave architecture and the input data is arbitrarily partitioned and stored on a distributed storage system like the GFS [58] or BigTable [59].

Given the input graph, the execution flow of a Pregel application follows the BSP model and is composed of several supersteps. The Pregel system maintains the vertices and edges stored on the node that will perform the computation – like on MapReduce, where the work is migrated to where the data is stored. Figure 1 illustrates the computation flow.

On each superstep the worker nodes invoke the method `Compute()` of each active vertex that is under its control. During the first superstep all vertices are active. The method `Compute()` is responsible for the execution of the business rule of the algorithm and is allowed, among other actions, to invoke other methods, compute new values for the vertex, add and/or remove vertices and edges, and send messages to other vertices. These messages are exchanged directly among the vertices, even if the vertices are being executed on different machines of the platform. The messages are sent asynchronously in order to allow the overlapping of computation and communication, but are delivered to the destination vertex only on the beginning of the next superstep. If a vertex declares that all its processing was done, it sends a message informing all the other nodes and deactivates itself. The master node stops the execution of the application after receiving this message from all the participants.

In order to guarantee the reliability of the system, Pregel performs regular checkpoints on user-defined intervals. The checkpoints are executed at the beginning of a superstep, each worker persists on the storage system the state of the subset of the graph it is responsible for, including the values associated to its vertices and edges and the messages that were received. In order to detect faulty nodes, the master regularly sends to the workers a ping message. If a worker stays too long without receiving a message from its master, it automatically finishes its computation and deactivates itself. Likewise, if the
master does not receive a response from one worker, it stops the interactions with this worker and restarts its work on another node using the last checkpoint available.

In order to improve the performance and usability of the system, Pregel includes mechanisms such as:

**Aggregators:** a mechanism for global communication, monitoring, and data. An aggregator combines information produced during a superstep using a reduce operator and the result is made available to all vertices on the next superstep.

**Combiners:** mechanism that reduce several messages addressed to a same vertex (or machine) on a single message, reducing the number of messages that must be transmitted.

An important feature required by graph algorithms is the ability of change the topology of the graph. For example, a clustering algorithm might replace each cluster with a single vertex, and a minimum spanning tree algorithm might keep only the tree edges. In order to avoid conflicting changes on the topology (for instance, multiple vertices could issue a command to create a vertex V with different initial values), Pregel uses two mechanisms to achieve determinism: partial ordering (removals are performed first, with edge removal before vertex removal and additions follow removals, with vertex addition before edge addition) and handlers (user defined conflict-resolution policies). Any change on the topology is only performed on the next superstep, before the invocation of the Compute() method.

Google Pregel is considered the main reference of application using the BSP model on Cloud Computing platforms. It is now a reference about large-scale graph processing systems. Its use is restricted because of its proprietary nature, but several open source initiatives were created following its specification.

### 4.2. Apache Hama

The Apache Hama (HAdoop MAttrix) [47] is a distributed computing framework that uses the BSP model and was inspired by the work on Google Pregel. The Apache Hama framework aims to provide a more general-purpose framework than Pregel, supporting massive scientific computations such as matrix, graph and network algorithms.

The framework is not only restricted to graph processing; it provides a full set of primitives that allows the creation of generic BSP applications. It architecture is composed of three main components:

- **BSPMaster:** it is the component responsible for controlling the execution of the supersteps, scheduling the jobs, assigning the jobs to GroomServers, keeping the status and information about availability and performance of the other servers.
- **GroomServer:** it runs on each node and is responsible for the execution of the jobs (called BSPTasks). Usually runs on top of a distributed file system.
- **ZooKeeper:** centralized service that provides and manages the distributed synchronization barriers.

The Apache Hama framework have an architecture that is very similar to the one used by Apache Hadoop. Both frameworks follows the Master/Slave architecture and their architectural components can be directly related to each other. On the master node, Hadoop runs the JobTracker (process that manages the jobs – MapTasks and ReduceTasks
– in execution), while Hama runs the BSPMaster. On the worker nodes, Hadoop runs the TaskTracker (responsible for the execution of the Map and Reduce jobs), while the Hama runs the GroomServer (responsible for the execution of the BSP jobs). Figure 2 compares both architectures.

The main difference that can be observed on Figure 2 is that the BSPTasks can communicate with each other, while this is forbidden between MapTasks and ReduceTasks. The MapReduce model imposes that all MapTasks must finish their execution before the execution of any ReduceTask, which means that the only form of communication between them is through the persistence of the data on the disk. With Apache Hama, we can have direct messages exchanged by the BSPTasks. Also, the data can be kept always on memory (if possible), avoiding performance loss caused by I/O operations.

The Apache Hama framework relies on some kind of distributed data management system. It is usually used with the Hadoop Distributed File System (HDFS), but it is flexible enough to be used with any other distributed file system.

The Apache Hama have the following strengths: explicit support to message passing; a small, simple and flexible API; better performance if compared to MPI application that too much communication; does not allows conflicts and deadlines during the communication (as a consequence of following the BSP model). It have also some limitations, like the fact that the API lacks more graph manipulation functions and the BSPMaster is a single point of failure (if the node dies, the application stops).

4.3. Apache Giraph

Apache Giraph [48] is an open-source framework for high-performance, large-scale graph processing. Inspired by Google Pregel, it is based on the frameworks Hadoop, ZooKeeper and Maven – all of them projects of The Apache Software Foundation. The development of the Apache Giraph was first started at Yahoo!, by Avery Ching and Christian Kunz, in December 2010.

In August 2011, Apache Giraph was no longer part of Yahoo!: it was selected as a project for the Apache Incubator. Since then, the project was a partnership of nine major committees, formed by relevant companies such as LinkedIn, Twitter, HortonWorks, Facebook, TrendMicro, Yahoo! itself, and by universities in different countries, such
as VU Amsterdam (Netherlands), TU Berlin (Germany) and Korea University (South Korea).

Like Pregel, Apache Giraph follows the BSP programming model, provides fault tolerance mechanisms, and keeps the data in memory. The computational model also follows the vertex-oriented model introduced by Pregel. In this implementation, given a certain graph, each vertex exchange messages with other vertices. These messages are delivered at the beginning of each BSP superstep. It also allows the creation of directed graphs (digraphs) and undirected graphs, weighted and unweighted graphs, and multigraphs (graphs in which is permitted to have multiple edges connecting a pair of vertices). The execution flow of a Giraph application is similar to Google Pregel (shown in Figure 1).

As previously mentioned, Apache Giraph uses the same infrastructure used by Hadoop and, therefore, follows the Master/Slave structure. In the following, we describe the main functions of each component:

- **Master** – is responsible for: monitoring the state of the worker nodes; perform partitioning of graphs before each superstep; determine the range of vertices that will be distributed to the worker nodes; coordinating the application by performing the synchronization of the supersteps (e.g., checking when they should be started and concluded);
- **Worker** – responsible for: performing the method `compute()` for each vertex attributed to it; sending messages to other workers (that will be delivered only in the beginning of the next superstep);
- **ZooKeeper** – is responsible for: maintaining the coordination state of the whole application, and also ensuring fault tolerance.

Comparing the two Apache projects that uses BSP programming model for large scale computing (Giraph and Hama), it is possible to observe that they both have resources for parallel processing of large scale graphs. While the Giraph features a vertex-oriented implementation and reuses the Hadoop infrastructure, Apache Hama offers a more general-purpose API (not just graphs) and does not depend on Hadoop.

5. Experiments on the Cloud

Several scientists attempt to understand the Cloud frameworks and their impact on the performance of parallel applications. Vecchiola et al. [60] compared different solutions of Cloud Computing frameworks, like Amazon EC2, Google AppEngine, Microsoft Azure and Makjrasoft Aneka. Gupta and Milojicic [3] have explored Cloud Computing in order to measure performance of HPC applications, showing that different Cloud Computing frameworks have impact on algorithms performance. Taifi et al. [61] shows that it is not trivial to run distributed applications using scalable strategies.

One usual need is to configure many virtual machines, one at a time, attempting to relate them in the same distributed context, which certainly would require lots of effort. Therefore, conducting experiments inside Cloud Computing is not trivial and requires knowledge of the mechanisms used for virtual machines management and placement.

A common use case in Cloud Computing consists of the instantiation of multiple virtual machines to run parallel algorithms. The Cloud Computing framework has the
role of managing virtual machines and the parallel algorithms has the role of processing
data and communicating with other nodes, which are other virtual machines inside the
Cloud Computing. Rego et al. [62] shows that the distribution of virtual machines on
hosts managed by the Cloud framework can impact upon results of execution of parallel
algorithms, especially on the data communication time. There is a clear trade-off between
the distribution of the load of the virtual machines among the multiple nodes of the
infrastructure and the communication overhead (that could be avoided by executing all
virtual machines that communicates within the same node). Figure 3 presents a Cloud
Computing architecture with instances of virtual machines.

The virtual machines distribution is carried out by the Cloud Computing frame-
work, which uses algorithms like Round-Robin or First-Fit. According to Kaur [63], the
Round-Robin distribution algorithm utilizes a single processing loop. According to Li et
al. [64], the First-Fit algorithm distributes the virtual machines on the first host from a
list of available physical machines inside the Cloud infrastructure. However, according
to Peixoto [65], none of those algorithms present the ability to guarantee processing per-
fomance of virtual machines regardless of the physical machine on which it is allocated.
Kaur [63] proposes enhancements in the performance using a load balancing algorithm
called ESCE (Current Equally Spread Execution). Li et al. [64] proposes DDR (Dynamic
Round-Robin) and Hybrid (DDR and First-Fit). Another form of distribution that can
be used is to specify computers inside the Cloud infrastructure where virtual machines
should be instantiated. But this is a case of private Cloud usage. In public Clouds, the
low-level management is hidden to the end user.

In the following section we present some experiments using our OpenNebula test bed
and virtual machines templates with MPI. The BSP/CGM algorithms are implemented on
top of these templates.


We have conducted some experiments to observe the scalability of algorithms written
using the BSP model on Cloud Computing platforms. Our test bed uses an installation of
the OpenNebula framework as the Cloud Computing platform, as well as the software
used to implement the virtual machines templates with MPI and the algorithms modeled
using BSP/CGM.
The OpenNebula is a IaaS (Infrastructure as a Service) framework. OpenNebula platforms are managed by a master node as shown in Figure 4. The figure also depicts the physical infrastructure utilized in our experiments.

Each of the 11 computer nodes used in the experiment has 1 Core Duo E6750 2.66 GHz, 2 GB RAM, 230 GB HD SATA II and NIC FastEthernet (100 Mbps). Each node runs a Ubuntu Server 10 Linux distribution and the entire infrastructure is managed by OpenNebula version 3.2.1.

Each virtual machine has a virtual hard disk of 2 GB and runs the Debian 6 Linux distribution. The applications were developed using the MPICH2 framework and Cgmlib 0.9.5. The applications are shared among the nodes using the NFS nfs-kernel-server distributed file system.

To instantiate virtual machines on different hosts within the Cloud we used the distribution mechanism with explicit assignment of virtual machines to physical nodes. This mechanism is provided by the OpenNebula user interface and allowed us to evaluate the performance of algorithms using specific scenarios. We represent the hosts with open and close parentheses "()", which indicates the number of virtual machines instantiated within each node. For instance, (3) (3) (3) (3) (3), which represent 6 hosts containing 3 virtual machines inside each one.

The BSP/CGM algorithms evaluated are ParallelSorting [31] and h-relation [66]. The h-relation is used for communication tests of the BSP/CGM algorithms. With the implementation of these algorithms it is possible to observe computational and communication times.

In the experiment, each ParallelSorting and h-relation algorithm was executed 30 times for each distributed scenario. Figures 5, 6 and 7 presents the average of execution times with a confidence interval of 95%.

Cáceres et al. [32], and Chan and Dehner [66] show that the BSP/CGM algorithms communication cost does not depend on the amount of input data. This can also be seen on our h-relation experiments. As an example of this, consider the problem of transmitting 400,000 bytes in a scenario involving nine virtual machines. Every node may transmit 44,445 bytes in each superstep. If another scenario was constructed with 18 nodes, each node may transmit 22,223 bytes.

The amount of supersteps remains constant and the size of messages decreases with the increase of the number of nodes. Bisseling [67] shows that network transmission latencies and message processing have also an increasing effect on the communication
Figure 5. Different distribution of 9 virtual machines.
Figure 6. Different distributions of 18 virtual machines.

time. This may explain the variances of the average communication times observed in the graphics. In general we could verify that the amount of time expend with h-relations remained constant.

The main contribution on the h-relations experiment was to show that the communication time is scalable. The total amount of communication remains the same, but the time does not increase when more nodes are added. This presents evidences that on ideal BSP algorithms (where the load balancing is also perfect), scalable results can also be obtained.
Our experiments also indicate that the computational times of the algorithms increased when a larger amount of virtual machines run in the same host. This occurs due to the concurrent use of the host CPU. For example, the computational time for distribution (3) (3) (3) is bigger than (3) (2) (2) (1) (1). However, as intuitively expected, the communication time is smaller when each Cloud host has more virtual machines. The best communication times were obtained with (3) (3) (3), (3) (3) (3) (3) (3) and (3) (3) (3) (3) (3). The virtual bridges inside each host avoid the use of the Cloud infrastructure network, which could generate additional latencies.

6. Conclusion

In this text we presented several possibilities related to the use of the BSP model for Cloud Computing. We presented the model and also the possibilities on using a theoretical model, CGM, to write BSP algorithms. We justified the use of BSP on Cloud Computing platforms showing several frameworks which already use its concepts for large scale distributed processing. Finally we conducted some experiments showing that BSP algorithms may be scalable in Clouds.

In our experiments with a private cloud, the results were influenced by different distributions of virtual machines, by the network latency differences and by the concurrent CPU usage inside the cloud hosts. An interesting future work would be to analyze with more details the influence of virtual machines allocation for programs written in BSP, possibly adding some QoS parameters.

References


[25] Edson Norberto Cáceres and Christiane Nishibe. 0-1 knapsack problem: Bsp/cgm algorithm and im-


