Improving Performance of Information Quality Applications: A Scale-Out Approach

Gustavo Frainer\textsuperscript{1}, Luiz Carlos Ribeiro Junior\textsuperscript{1}, Mauro Storch\textsuperscript{1}, Rodrigo F. de Mello\textsuperscript{2}

\textsuperscript{1}Acxiom Brasil
\textsuperscript{2}Departmento de Ciências da Computação
Instituto de Ciências Matemáticas e de Computação
Universidade de São Paulo
{gustavo,luiz,mauro.storch}@acxiom.com.br, mello@icmc.usp.br

Abstract. Throughout the years, information quality has become an essential tool for marketing decisions and to support other business scenarios and operations. That happens due to the fast pace of increase in both the amount of data to be qualified and the complexity of qualification process. This has motivated the adoption of High Performance Computing (HPC) using scale-out environments. In that sense, this paper presents a suite of Information Quality application running over a scale-out environment using GoStorm, a middleware which was designed to transparently support the execution of Java applications over scale-out environments through application threads and data distribution. The experiments were conducted to evaluate the performance of those applications to qualify a large dataset over two scale-out environments (Cloud and Cluster).

1. Introduction

Information Quality (IQ) is the area concerned with assessing the quality of system information. IQ levels ensure the confidence we have on a particular information to meet a context requirement. IQ has been currently playing an important role in business development, where it provides relevant information to maintain and grow companies’ operations [Eckerson 2002]. In that sense, information accuracy is a very important metric to IQ applications. However there are other elements to be considered such as the information deadline.

Currently, companies require strategic and even operational information on demand and under given time constraints. That motivated the use of high performance computing (HPC) techniques to improve application performance through both application and data distribution.

In that sense, companies such as IBM [The BlueGene/L Team 2002] and Cray [Smith et al. 1990] designed multiprocessor machines with large-capacity integrated memory modules (shared memory). Those computers have well defined scalability limits and execute a single operating system image, which simplifies the management and portability of applications. This approach, named scale-up [Michael et al. 2007], was mainly adopted by specific application domains due to high investments and scalability issues. At the same time, microprocessors, memory modules, network interface cards and other
devices started being produced in a greater scale, which reduced their costs and made them accessible.

The accessibility to commodity hardware motivated the design of clusters of computers (or simply clusters), which were rapidly adopted to obtain high performance computing at lower costs. Clusters consider a different approach to provide scalability. In that architecture, computer nodes are interconnected through a network and communicate with each other by message passing protocols. This strategy, named scale-out [Michael et al. 2007], provides a priori unlimited scalability and requires every node to run a different operating system image, which tends to cause consistency issues and makes the management and administration more complex.

During the 1990’s, researchers started connecting clusters and composing larger environments. Other researchers also attempted to interconnect unusual resources to that architecture, such as laboratory instruments, visualization devices and expensive equipments which could be remotely shared. The grid computing concept is derived from those environments. Grids connect heterogeneous and geographically dispersed resources which cooperate to solve a computational problem [Foster et al. 2001]. After the term grid, other sectors introduced the term Cloud Computing [Buyya et al. 2008]. This latter is focused on providing software modules or components via Web services. Thus, when software needs to execute an operation, it makes a request to a service which is responsible for processing it and returning results.

In summary, the scale-up approach is easier to administer due to there only being one central operating system image, which also avoids consistency problems. Multithreaded applications can also take advantage of this approach without any source code modification. Fault tolerance is also addressed by scale-up machines. On the other hand, those computers are very expensive. In that sense, investment reasons have been motivating the adoption of scale-out approaches. Besides being much cheaper, this latter considers many operating system images, which make managing tasks such as the system update and administration more complex. The multiple copies also bring up consistency issues (data copies may be in different versions). In addition, the scale-out approach also requires applications to be rewritten. Despite all those problems, the scale-out approach has no prior scalability limit.

Besides all those points, scale-up approaches require a big investment at the beginning and, as long as the application is in the production environment, the cost is paid [Gillett 2008]. On the other hand, scale-out allows the addition of new computers to the environment as long as more resources are needed. In that way, investments are made only when necessary and they accompany the return (cash in) provided by the application execution. The availability of computing resources, investment and scalability issues have motivated improvements on communication systems, application adaptability, management of dispersed resources, security, process and data scheduling.

All those issues make clear that scale-up is simpler and easier to use, however it is too costly. The scale-out approach is more complex to manage and administer and it requires application adaptation (an extensive source code modification), however it is cheaper, which motivates its adoption in several application domains. This trade off motivated us to design and implement a high performance computing platform called GoStorm.
which provides transparent support for end-user applications and automatically manages
distributed hardware resources and applications on scale-out environments.

GoStorm is composed of a set of tools which provide autonomic computing fea-
tures for executing and monitoring applications. Besides that, GoStorm innovates in ap-
application execution due to it being capable of distributing any Java application without
source code modification and user interference. GoStorm also has the advantage (as de-
scribed in [Ferreira et al. 2003]) of providing a virtual environment to applications, in
which users do not need to worry about where the application is running. In that sense, all
distributed resources are transparent and the user sees the system as a virtual supercom-
puter.

The main objective of GoStorm is to take advantage of the lower investments re-
quired when designing scale-out environments and, at the same time, bring the easiness
of use, administration and management of scale-up architectures to scale-out environ-
ments. Therefore, GoStorm aims at reducing the cost-performance ratio of applications.
Besides all high performance theoretical and practical technological advances, our main
objective is to address Information Quality applications on top of this platform. Conse-
quently, we conducted several experiments to confirm its performance, robustness and
availability when executing IQ processes. Results confirm that GoStorm improves the
cost-performance ratio of IQ applications. Tests were conducted over two kinds of scale-
out environments: Cloud and Cluster.

This paper is organized as follows: first an overview of the company’s IQ applica-
tions that makes use of GoStorm is presented. Then the GoStorm middleware is described.
After that, the process of submitting applications to be run by GoStorm is explained. A
section containing several experiments follows. Finally, conclusions and possible future
works are presented.

2. Information Qualification Tool: Our Scenario

Our interest in distributing Information Quality applications over scale-out architectures
arose from the needs of some tools developed by our company. We will now discuss
two of those tools: Address Validator (AV) and Person/Company Validator (PCV), which
were the two first applications to execute on GoStorm.

AV and PCV are both Java applications which process large datasets in order to
enhance the quality of contained information. They do that by evaluating data records,
generating candidate records and choosing the most appropriate one (original including)
to replace the original. For example, there are several records with a wrong company
name. A large list of candidate records is generated for each original record and PCV
decides which of them contain the correct information. Afterwards, PCV updates the
whole dataset, guaranteeing accuracy.

AV works in the same manner, although it tends to be more computationally in-
tensive due to the techniques used to calculate the similarity between the candidates and
the original record. Both enrich the dataset, looking for complementary information on
addresses, companies and people. All that supports marketing actions.

Both AV and PCV have no dependency when processing data records, so datasets
can be divided to balance the workload of concurrent executions on top of scale-out envi-
environments. This parallelism model is especially simple to apply in this context. However, some team efforts have been considered to design other parallelism levels for information quality applications.

AV and PCV can also be connected to an information qualification flow by using the Information Quality tool developed by our company. The Information Quality tool is responsible for designing such flows which run over an ETL engine. Flows are composed of the input, output, join, handler, and qualifier nodes. Every flow is designed as a pipeline starting with an input node, which reads records from a dataset (i.e., files, databases), and ending with an output node, that writes information into the dataset. In between these two nodes, we can create sequences with join, handler and qualifier nodes. AV and PCV are associated to a qualifier node. For example, consider a flow composed of an input node connected to an AV node that is linked to an output node. This flow represents a simple Address Validator qualification process.

AV and PCV nodes were specifically developed to support Brazilian data qualification, on the other hand, both join and handler nodes are generic components. The join is used to receive data from two or more nodes simultaneously. The handler is used to change record data according to user-defined rules, for example word capitalization. Unlike join and handler nodes, qualifier nodes are capable of splitting data into parallel threads. As threads finish, the qualified information is submitted to the next flow node.

In addition to providing accuracy in data qualification process, these tools should be as fast as possible since they have to support the qualification of large data volumes. This has been a constant struggle, since there is often a tradeoff between accuracy and speed of processing. Despite our best efforts, processing very large dataset still takes a lot of processing power. In next section, the scaling handling of those processes are presented.

3. Scaling Out with GoStorm

Both importance and amount of information have been increasing in the current global business scenario. Combining data from different business data sources enriches the information, but it also increases the amount of data that must be dealt with. Moreover, the complexity to analyze such information and qualify it has also become a major challenge.

In order to deal with this complexity and with the constantly increasing amount of data and still obtain qualified data in a timely manner the processing power has to increase as well. In scaling computational power there are two approaches to be considered: (i) invest in scale-up environments; (ii) invest in scale-out environments.

In this manner, scale-up pays for itself only after time, while scale-out accompanies the business needs and, therefore, its pay off is instantaneous. Consequently, we consider the latter approach as the best as it allows us to keep qualification accuracy and it also supports gradual investments over time.

However, the scale-out approach has one important drawback: generally the application needs be modified before it can be used over a scale-out environment. This was the main motivation to start the GoStorm project: offering tools that allow Java applications to be transparently distributed over scale-out environments.

In summary, GoStorm takes advantage of scale-out concepts to provide four main
contributes: (i) reduce the time and cost to port applications to run over scale-out environments; (ii) reduce the need for great investments at once, thus, companies can increase their processing capacity as the business evolves; (iii) improve the computing performance of information quality processes; (iv) allow the qualification of very large datasets.

The GoStorm project provides the following main functionalities: communication, security, high availability, application adaptability, process and data scheduling and high-level configuration policies. All those functionalities are implemented in software modules which interact to structure and manage scale-out computing environments. Those modules, organized in a layered manner, compose the GoStorm architecture presented in Figure 1.

- Communication – GoStorm provides a peer-to-peer protocol to manage inter-node communication. This protocol builds up an overlay network to connect nodes. Every node is represented by a node identifier which is used to send and receive messages. The protocol was designed to support synchronous and asynchronous messages, as well as any transport and network protocols. Currently, it is implemented using TCP (Transmission Control Protocol) over IP (Internet Protocol);
- Cryptography – Communication operations can occur with or without encryption. This module adds an extra overhead to encrypt and decrypt messages, however it makes communication reliable and confidential;
- Checkpoint – GoStorm runs single and multithreaded applications. Those threads are distributed over the scale-out environment and execute on different computers. In order to avoid the waste processing time and make migration possible, the checkpoint module saves thread states into files. In case of thread failure, it can be resumed from the point it was saved. Those states are also copied to other computers and, in case of a node failure, it is resumed in another node. Besides that, the state can also be used to migrate a thread to other nodes and, afterwards, resume it;
- File Manager – This module divides files into data chunks and distributes them over a set of computers called chunk servers. Chunks are replicated on different chunk servers which increases the system availability. In that manner, if a chunk server goes down, another one can fulfill application requests. Besides chunk servers, there are still the meta-data servers that work as indices to chunk servers. A meta-data server references chunks in chunk servers. Meta-data are replicated among servers and when eventually a meta-data server goes down, another one can meet the applications needs. A catalog service lists all the meta-data servers. When an application needs to access files, it requests a meta-data server to the catalog. From then on, GoStorm manages the interaction in between the application and the meta-data server as well as chunk servers. Data is kept consistent among chunk servers and meta-data servers;
- Memory Manager – As previously mentioned, multithreaded applications are executed on top of GoStorm. Those threads are translated into GoStorm Threads and distributed over the environment. Threads traditionally communicate with one another by using the local shared memory. GoStorm simulates this shared memory in the Memory Manager module. The module provides a distributed shared memory layer on top of the communication module. In that way, threads need no modification to execute on GoStorm. They access the memory in the same way
they do locally;

- Benchmark – Every computer available in the scale-out environment has particular capacities. This benchmark module is responsible for extracting the CPU, memory, hard disk and network capacities of nodes. It obtains the maximum number of instructions that are executed by the processor, the memory and hard disk availability and performance, and the network interface card bandwidth. All that information is obtained when GoStorm is installed or whenever the administrator launches the module. After obtaining the information, it is kept in a local file for future use;

- Monitors – Two monitors are available in GoStorm. The first is the system monitor which obtains the processor, memory, hard disk and network usage for every node. This information, in conjunction with the benchmark, makes it possible to evaluate the percentage of free and occupied resources. The second is the application monitor which intercepts thread system calls, generating time series on resource occupation. Therefore, those series contain the history of every thread. By composing histories, we know the full application behavior;

- Predictor – The historical behavior of thread occupation (CPU, memory, hard disk and network) is saved into databases. When a new application is launched, its historical behavior is evaluated and, then, GoStorm gets a big picture about how resources will be used by threads. This resource occupation is sent to the Submitter in order to support process allocation. Besides this historical predictor, GoStorm was also designed to support an online time-series predictor which is under development. This predictor analyses time series observations and makes predictions on when a specific resource will be required and which process will need it. Using that information, the scheduler can predefine allocation and migration of workloads;

- Scheduler – All node capacity information (obtained using the benchmark module) and usage (system and application monitors) are submitted to the process scheduler which also considers historical predictions on thread occupation to take scheduling decisions. Based on all that information, the scheduler runs a policy to optimize application needs according to resource availability;

- Dispatcher – The dispatcher module launches applications on GoStorm. It basically receives the user command and parameters and starts the allocation operation. It communicates with the scheduler and requests the execution;

- Translator – Thread system calls are intercepted and saved by the application monitor, the checkpoint saves thread states and makes resuming possible, the distributed shared memory works transparently with applications. All this is possible due to the translator which is responsible for transparently adapting applications and preparing them to run on GoStorm;

- Policy Configuration – This module is responsible for modeling high-level management policies. It defines how long a computer will be available to GoStorm applications, when they will be and which applications can execute on those nodes. It also defines security policies such as available communication ports, supported protocols and users;

- Access Control – This defines a database of users and their respective rights. When a user is authenticated, a token is created and sent to every application launched by the user. The token informs the user rights to the available resources;
Figure 1. GoStorm architecture

The previously addressed modules are used to compose GoStorm entities. Those entities run on nodes to compose the scale-out environment. The entities are:

- **Worker** – This entity is responsible for executing user applications. It receives threads, executes them and generates results;
- **Broker** – Brokers communicate with each other to compose the overlay communication infrastructure. They are all registered into a catalog and, therefore, one Broker can find others. After finding each other, they usually connect to the nearby Brokers to make the peer-to-peer network available. Then, Workers connect to Brokers to provide resources;
- **Submitter** – Many submitters can be started in the scale-out environment. Every submitter connects to one or more Brokers. It also receives the request to launch user applications and, then, spreads a search over the overlay network, requesting historical information on that application. This request is sent to Brokers which spread it to Workers. Workers evaluate their local historical database looking for that application. Finding information on that application, the Worker summarizes it using a nearest neighbor approach and sends it back to the submitter [Senger et al. 2007]. Either they find information or not, Workers send their respective capacity and current workloads to the scheduler. After receiving all that information, the scheduler composes them all and executes an optimizer which is responsible for taking decisions about which resource every thread will be placed on. Communicating threads will preferably be allocated on nearby nodes. Independent threads can be allocated on any available and capable resource. By communicating threads, we mean the ones that access the distributed shared memory or the
distributed file system, since they can read and write data on both:

- **Distributed File System (DFS)** – The distributed file system entity is responsible for managing the meta-data and chunk servers of files. It is installed in multiple computers. It keeps data available, replicates and allows the parallel access of chunks. For example, consider one chunk c. The DFS entity replicates it to several chunk servers and, in case of writing operations, the DFS maintains all copies consistent. Having multiple copies of the same chunk c on different chunk servers, an application can request it in parallel and read parts of it from different servers. That makes reading operations faster. The same can be used to write data on chunk servers and keep it consistent;

- **Distributed Shared Memory (DSM)** – The distributed shared memory entity creates a layer on top of the peer-to-peer overlay network and provides memory to applications. This layer is transparent to applications. This means that applications do not need modifications to access the shared memory, consequently, they behave in the same way they do on local shared memory computers.

Technical details about the implementation of GoStorm can be found in [Mantini et al. 2009]. The next section details the steps to execute an application on GoStorm.

### 4. Application Submission

The submission of applications to scale-out environments is a very important issue, since available platforms (for clusters, grids and clouds) require a broad range of adaptations. However, there are several factors that preclude this adaptation, for example: the software can be developed by third parties and the user may not have full source-code access; source-code modifications consume financial resources and time; the modified source code hardly ever performs in the traditional environment, thus requiring two different versions. Moreover, different versions of the code make the maintenance more difficult as any update requires changes in both versions.

Aiming at addressing this problem, GoStorm delivers an innovative and transparent mechanism to automatically execute any Java application without source-code adaptations. Therefore, the user does not need to have the source code available nor make any kind of adjustment in the original application, i.e. the same Java code is executed on the distributed environment.

For this purpose, GoStorm provides a submission tool, which is responsible for making just-in-time changes at the byte-code level. As a first step, the submission tool receives the application classes, parameters and required dependencies (through the Java Classpath) in the same way the Java Virtual Machine does (via command line). On a second step, GoStorm sends the original application classes to the DFS, which makes them available to the whole environment (using the GoStorm Distributed File System). At last, the information of the main application thread is encapsulated in a GoStorm Job Object, which is submitted to the Scheduler.

Once the application is submitted to GoStorm, proper scheduling policies are applied and the Job is allocated on a Worker entity, where it will execute. Before executing, the Job needs to be prepared for execution on the distributed environment. The byte-code is swept and all file access calls are replaced by proxy calls, which redirect them to the DFS. The calls responsible for launching threads are transparently replaced by
GoStorm thread distribution calls. At last, an object-shared discovery algorithm identifies which objects should be managed by the DSM module. Byte-code instructions that access these objects are replaced by instructions that access the DSM. All such interceptions are performed on-the-fly, by modifying the compiled byte-code of the classes, using the third-party library ASM [Eric Bruneton et al. 2002] (this is conducted during the class loading). After such modifications, the application will access the DFS as storage (when reading or writing files) and have all threads distributed without human intervention. Threads launched by the application (Figure 2) are encapsulated in new Jobs, subsequently sent to the Scheduler (responsible to take scheduling decisions). All steps are repeated for every Job. When a thread finishes, its results (i.e. the final thread object) are reintegrated to the job that launched it.

This approach guarantees a transparent execution to end users, suppressing the need of modifying application source codes. Therefore, every application thread is automatically and gracefully distributed, simplifying the scalability of virtually any Java application.

5. Experiments

In order to evaluate GoStorm, we considered two types of scale-out environments: one cloud and one cluster computing environment. The cloud environment was composed of seven dual-core at 2.2Ghz, 2GB RAM and 100GB HDD connected through a 100Mb/s switch, thus, this environment provides 14 cores. The cluster computing environment was formed by three dual-core at 2.3GHz, 2GB RAM and 100GB HDD, forming a 6-core environment. In both environments, we deployed the following GoStorm components: one Catalog service, one Broker, and as many Workers as the number of cores available. Each component was launched on an isolated JVM (Java Virtual Machine) and Workers was configured to use up to 2GB main memory in the cloud scenario and 1.5GB in the cluster environment. Each Worker was able to run simultaneously a job by each core. Figure 3 illustrates how GoStorm components were deployed on top of both environments.

In the cloud environment, we executed a simple AV flow with an input node, used to read data from a file, an Address Validator node, which qualifies information, and an output node to write information back to a file. This flow was executed over a 1.5M dataset with incomplete People, Company and Address information. It is important to
mention that only AV threads were distributed.

The speedup results for this first experiment are presented in Figure 4. The flow was executed using from 1 to 14 cores. The execution was performed 20 times and the average was computed. The dotted lines show the ideal speedup while solid ones represent the speedup obtained. Although the speedup does not fit in the ideal linear curve (the ideal curve represent that performance increases in the same rate as computing elements are introduced in the environment), we observe a significant performance improvement as nodes are added (including more cores) to the environment. This experiment confirms that GoStorm provides scalability for the qualification of large datasets under distributed environments. In the cluster computing scenario, three different flows were considered in order to compare their speedup behavior: (i) a simple AV flow, (ii) a simple PCV flow and (iii) a flow with an AV and a PCV (in this sequence). Every flow started with an input node (reading from file) and finished with an output node (writing qualified records back to a file). Results are presented in Figure 5. Each flow was performed 20 times and the average was used to draw the speedup line. The line identified as AV is related to the simple AV flow. The line identified as PCV is related to the simple PCV flow. At last, the line identified as AV and PCV is related to the flow that contains an AV node followed by a PCV node (the ideal line represent that performance increases in the same rate as computing elements are introduced in the environment).

This set of experiments was created to demonstrate the speedup behavior in both isolated qualification nodes and in concatenated qualification nodes. The results show that the simple AV flow has a lower speedup when compared to the flow composed of an AV node and a PCV node. In the same way, the flow composed of an AV node and a PCV node obtained lower speedup than the simple PCV flow. This confirms that the lower speedup of the AV is limiting the total performance of the concatenated flow. This was expected since, as shown in Table 1, the AV consumes more processing time than PCV. Furthermore, in the concatenated flow the AV adds to each record a number of fields related to the qualification, which forces the PCV to deal with a larger volume of data than in the simple PCV flow increasing communication overhead.

Despite that, the concatenated flow still managed to achieve a speedup that was
between the speedups obtained by the components on simple flows. This indicates that
GoStorm obtains a significative speedup even when dealing with more complex flows, as
long as the individual components can be successfully distributed.

<table>
<thead>
<tr>
<th>Flow/Cores</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>AV</td>
<td>101690</td>
<td>50845</td>
<td>38991</td>
<td>32502</td>
<td>28401</td>
<td>25662</td>
</tr>
<tr>
<td>PCV</td>
<td>42604</td>
<td>21302</td>
<td>14545</td>
<td>11369</td>
<td>9320</td>
<td>8001</td>
</tr>
<tr>
<td>AV/PCV</td>
<td>126348</td>
<td>63174</td>
<td>48205</td>
<td>40106</td>
<td>34325</td>
<td>30836</td>
</tr>
</tbody>
</table>

Table 1. Time series in seconds.

Figure 4. AV speedup in cloud environment

6. Conclusion and Future Works

This paper presents a transparent and automatic middleware, named GoStorm, to dis-
tribute IQ applications on top of scale-out environments. GoStorm currently addresses
a suite of Information Quality applications. Experiments were conducted to evaluate the
performance provided by GoStorm to IQ flows on a cluster and a cloud computing envi-
ronment. Results confirm a considerable speedup provided by the proposed middleware.

As future work, we will conduct further experiments considering larger environ-
ments. We also intend to support transparent access to distributed database systems [Bern-
stein et al. 1981] like NoSql. This will be important to continue dealing with large datasets
for two reasons: (i) it will allow large datasets to be spread over several machines, elimi-
ning the need for one big and reliable machine capable of storing the whole dataset;
(ii) it leads to faster access to the data through the distribution of the queries, which will
be important to achieve good speedups. We are also considering conduct experiments
with other Information Quality tools like data merge solutions [De Mello et al. 2010] and
swarm optimization strategies [De Abreu et al. 2010]. Those applications test the shared
memory module to find out the best candidate records, improving application accuracy.
7. Acknowledgments

This paper is based upon work supported by FINEP\(^\text{1}\) (Financiadora de Estudo e Projetos), Brazil. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of FINEP.

References


\(^\text{1}\)http://www.finep.gov.br


