JCL: A High Performance Computing Java Middleware

André Luís Barroso Almeida[1,2], Saul Emanuel Delabrida Silva[1], Antonio C. Nazare Jr.[3] and Joubert de Castro Lima[1]

[1]*DECOM, Universidade Federal de Ouro Preto, Ouro Preto, Minas Gerais, Brazil*

[2]*CODAAUT, Instituto Federal de Minas Gerais, Ouro Preto, Minas Gerais, Brazil*

[3]*DCC, Universidade Federal de Minas Gerais, Belo Horizonte, Minas Gerais, Brazil*

Keywords: Big Data, Internet of Things, Middleware, Reflective Computing, High Performance Computing, Distributed Shared Memory, Remote Method Invocation.

Abstract: Java Cá&Lá or just JCL is a distributed shared memory reflective lightweight middleware for Java developers whose main goals are: *i*) provide a simple deployment strategy, where automatic code registration occurs, *ii*) support a collaborative multi-developer cluster environment where applications can interact without explicit dependencies, *iii*) execute existing sequential Java code over both multi-core machines and cluster of multi-core machines without refactorings, enabling the separation of business logic from distribution issues in the development process, *iv*) provide a multi-core/multi-computer portable code. This paper describes JCL's features and architecture; compares and contrasts JCL to other Java based middleware systems, and reports performance measurements of JCL applications.

1 INTRODUCTION

We live in a world where large amounts of data are stored and processed every day (Han et al., 2011). Despite the significant increase in the performance of today's computers, there are still big problems that are intractable by sequential computing approaches (Kaminsky, 2015). Big data (Brynjolfsson, 2012), Internet of Things (IoT) (Perera et al., 2014) and elastic cloud services (Zhang et al., 2010) are results of this new decentralized, dynamic and communication-intensive society. Many fundamental services and sectors such as electric power supply, scientific/technological research, security, entertainment, financial, telecommunications, weather forecast and many others, use solutions that require high processing power. For instance, "Walmart handles more than a million customer transactions each hour and it estimated that the transaction database contains more than 2.5 petabytes of data" (Troester, 2012). Thus, these solutions are executed over parallel and distributed computer architectures. In this context, a new demand for both computer architectures and applications to handle such big problems has emerged.

High Performance Computing (HPC) is based on the concurrence principle, so high speedups are achievable, but the development process becomes complex when concurrence is introduced. Therefore, middleware systems and frameworks are designed to help reducing the complexity of such development.

The use of middleware as a software layer on top of an operating system became usual in last years in order to organize a computer cluster or grid (Tanenbaum and Van Steen, 2007). The challenging issue is how to provide sufficient support and general high-level mechanisms using middleware for rapid development of distributed and parallel applications. Furthermore, the middleware systems found in the literature have no information of their use on different computational platforms. For instance, there is no evidence that a set of embedded devices are able to work with a cloud platform using the same middleware. To developers, the system integration is not transparent and depends of different skills. In Perera et al. (2014), the authors mention the existence of six machine classes in IoT and they advocate that middleware systems should run over all of them (Perera et al., 2014).

Middleware systems can be adopted for general purposes, such as Message Passing Interface (MPI) (Forum, 1994), Java Remote Method Invocation (RMI) (Pitt and McNiff, 2001), Hazelcast (Veentjer, 2013), JBoss (Watson et al., 2005) and many others, but they can also be designed for a specific purpose, like gaming, mobile computing and real-time computing, for instance (Murphy et al., 2006; Gokhale et al., 2008; Tariq et al., 2014). They support a programming model based on shared memory, message passing or event based (Ghosh, 2014). Among various programing languages for middleware systems, the interest in Java for HPC is rising (Taboada et al., 2013). This interest is based on many features, such built-in networking, multithreading support, platform independence, portability, type safety, security, extensive API and wide community of developers (Taboada et al., 2009).

Besides the computational performance, the issues presented below must be considered in the design and development of a modern middleware.

Refactorings: Usually, middleware systems introduce some dependencies to HPC applications, thus, end-users need to follow some standards specified by them, since methods and global variables must be distributed over a data network. Consequently, an ordinary Java or C++ object [1] must implement several middleware classes or statements to become distributed. There are many middleware examples with such dependencies, including standards and market leaders like Java RMI (Pitt and McNiff, 2001), JBoss (Watson et al., 2005) and Mapreduce based solutions (Hindman et al., 2011; Zaharia et al., 2010). As a consequence of these dependencies, two problems emerge: *i*) the end-user cannot separate business logic from distribution issues during the development process and; *ii*) existing and well tested sequential applications cannot be executed over HPC architectures without refactorings. A zero-dependency middleware is unrealistic, but a middleware with few adaptations is fundamental to achieve low coupling with the existing code.

Deployment: Deployment can be a time consuming task in large clusters, i.e. any live update of an application module or class often interrupts the execution of all services running in the cluster. Some middleware systems adopt third-party solutions to distribute and update modules in a cluster (Henning and Spruiell, 2006; Nester et al., 1999; Veentjer, 2013; Pitt and McNiff, 2001), but sometimes updating during application runtime and without stoppings is a requirement. This way, middleware systems capable of deploying a distributed application transparently, as well as updating its modules during runtime and programmatically, are very useful to reduce maintenance costs caused by several unnecessary interruptions.

Collaboration: Cloud computing introduces opportunities, since it allows collaborative development or development as a service in cloud stack. A middleware providing a multi-developer environment where applications can access methods and user typed objects from each other without explicit references is fundamental to introduce development as a service or just to transform a cluster into a collaborative development environment.

Portable Code: Portable multi-core/multi-computer code is an important aspect to consider during development process, since in many institutions, such as research ones, there can be huge multi-core machines and several clusters of ordinary PCs to solve a couple of problems. This way, code portability is very useful to test algorithms and data structures in different computer architectures without refactorings. A second justification for offering at least two releases in a middleware is that clusters are nowadays multi-core, so middleware systems must implement shared memory architectural designs in conjunction with distributed ones.

The main goal of this work is to introduce a new middleware that fill the issues previously shown, precisely:*i*) a simple deployment strategy and capacity to update internal modules during runtime; *ii*) a collaborative multi-developer environment; *iii*) a service to execute existing sequential Java code over both multi-core machines and cluster of multi-core machines without refactorings; *iv*) multi-core/multi-computer portable code. This middleware, called Java Cá&Lá[2] or just JCL, is a tool for develop HPC applications. In this paper, the three main components of our architecture are presented and evaluated. Is not the focus of this work to be a "how to" guide, although some source codes are shown in order to clarify the reader's understanding. The main contributions of JCL are:

i) A middleware that gathers several features presented separately in the last decades of middleware literature, enabling building distributed applications with few portable instructions to clusters made from different platforms;

ii) A comparative study of market leaders and well established middleware standards for Java community. This paper emphasizes the importance of several features and how JCL and its counterparts fulfill them;

iii) A scalable middleware over multi-core and multi-computer architectures;

iv) A feasible middleware alternative to fast prototype portable Java HPC applications.

This work is organized as follows. Section 2 discusses works that are similar to the proposed middle-

[1]"An object is a self-contained entity consisting of data and procedures to manipulate data" (Egan, 2005)

[2]Java Cá&Lá is available for download at http://www.joubertlima.com.br

ware, pointing out their benefits and limitations. Section 3 details the JCL middleware, presenting its architecture and features. Section 4 describes a user case application. Section 5 presents our experimental evaluation and discusses the results. Finally, in Section 6, we conclude our work and point out future improvements of JCL.

2 RELATED WORK

In this section, we describe the most promising middleware systems in various stages of development. We evaluated each work in terms of: *i*) requirement for low/medium/high refactorings, *ii*) automatic or manual deployment, *iii*) support for collaborative development, *iv*) implementation of both multi-core and multi-computer portable code. Other analyses were made to verify if the middleware is discontinued, and if it is fault tolerant in terms of storage and processing. Academic and commercial solutions are put together and their limitations/improvements are highlighted in Table 1. Middleware systems that present high similarities with JCL are described in detail in this section. The remaining related work is described just in Table 1.

Infinispan by JBoss/RedHat (Team, 2015) is a popular open source distributed in-memory key/value data store (Di Sanzo et al., 2014) which enables two ways to access the cluster: *i)* the first way enables an API avaliable in a Java library ; *ii)* the second way enables several protocols, such as HotRod, REST, Memcached and WebSockets, making *Infinispan* a language independent solution. Besides storage services, the middleware can execute tasks remotely and asynchronously, but end-users must implement Runnable or Callable interfaces. Furthermore, it is necessary to register these tasks at Java virtual machine (JVM) classpath of each cluster node, so *Infinispan* does not have the dynamic class loading feature.

Java Parallel Processing Framework *JPPF* is an open source grid computing framework based on pure Java language (Xiong et al., 2010) which simplifies the process of parallelizing applications that demand high processing, allowing end-users to focus on their core software development (Cohen, 2015). It implements the dynamic class loading feature in cluster nodes, but it does not support collaborative development, i.e. methods cannot be shared among different *JPPF* applications, producing many services over a cloud infrastructure, for instance. JPPF does not implement shared memory services just execute methods.

Hazelcast (Veentjer, 2013) is a promising middleware in the industry. It offers the concept of functions, locks and semaphores. Hazelcast provides a distributed lock implementation and makes it possible to create a critical section within a cluster of JVM; so only a single thread from one of the JVM's in the cluster is allowed to acquire that lock. (Veentjer, 2013). Besides an Application Programming Interface (API) for asynchronous remote method invocations, Hazelcast has a simple API to store objects in a computer grid. JCL separates business logic from distribution issues and, in Hazelcast, both requirements are put together, so flexibility and dynamism are reduced during execution time. Hazelcast cannot instantiate a global variable remotely like JCL, i.e., it always maintains double copies of each variable at each remote instantiation. Hazelcast has manual scheduling for global variables and executions, so the end-user can control the cluster machine to store or run an algorithm. *Hazelcast* does not implement automatic deployment, so it is necessary to manually add each end-user class to the JVM classpath before starting each *Hazelcast* node.

Oracle Coherence is an in-memory data grid commercial middleware that offers database caching, HTTP session management, grid agent invocation and distributed queries (Seovic et al., 2010). Coherence provides an API for all services, including cache services and others. It enables an agent deployment mechanism, so there is the dynamic class loading feature in cluster nodes, but such agents must implement the EntryProcessor interface, thus refactorings are necessary.

RAFDA (Walker et al., 2003) is a reflective middleware. It permits arbitrary objects in an application to be dynamically exposed for remote access, allowing applications to be written without concern for distribution (Walker et al., 2003). RAFDA objects are exposed as Web services without requiring reengineering to provide distributed access to ordinary Java classes. Applications access RAFDA functionalities by calling methods on infrastructure objects named RAFDA runtime (RRT). Each RRT provides two interfaces to application programmers: one for local RRT accesses and the other for remote RRT accesses. RRT has peer-to-peer communication, so it is possible to execute a task in a specific cluster node, but if the end-user needs to submit several tasks to more than one remote RRT, a scheduler must be implemented from the scratch. RAFDA has no portable multi-core and multi-computer versions.

In the beginning of 2000's, an interesting middleware, named *FlexRMI*, was proposed by Taveira et al. (2003) to enable asynchronous remote method invocation using the standard Java Remote Method

Table 1: JCL and its counterparts' features.

Feature / Tool	Fault Tolerant	Refactoring required	Simple Deploy	Collaborative	Portable Code	Support Available
JCL	No	No	Yes	Yes	Yes	Yes
Infinispan	Yes	Low	No	Yes	No	Yes
JPPF	Yes	No	Yes	No	No	Yes
Hazelcast	Yes	Low	No	Yes	No	Yes
Oracle Coherence	Yes	Medium	NF^1	Yes	No	Yes
RAFDA	No	No	Yes	Yes	No	Yes
PJ	No	NF^1	NF^1	No	Yes	Yes
FlexRMI	No	Medium	No	No	No	No
RMI	No	Medium	No	No	No	Yes
Gridgain	Yes	Low	No	Yes	No	Yes
ICE	Yes	High	No	No	No	Yes
MPJ Express	No	Medium	No	No	Yes	Yes
Jessica	NF^1	No	Yes	No	Yes	Yes

1 - NF: Not found

Invocation (RMI) API. FlexRMI is a hybrid model allowing both asynchronous or synchronous remote methods invocations. There are no restrictions in the ways a method is invoked in a program. The same method can be called asynchronously at one point and synchronously at another point in the same application. It is the programmer's responsibility the decision on how the method call is to be made. (Taveira et al., 2003) FlexRMI changes Java RMI stub and skeleton compilers to achieve high transparency. *FlexRMI* is the RMI asynchronous, so there is no multi-core version. Furthermore, it requires at least "java.rmi.Remote" and "java.rmi.server.UnicastRemoteObject" extensions to produce a RMI application. Since it does not implement the dynamic class loading feature, all classes and interfaces must be stored in nodes before a RMI (and also FlexRMI) application starts, making deployment a time-consuming effort.

Gelibert et al. (2011) proposed a new middleware using Distributed Shared Memory (DSM) principles to efficiently simplify the clustering of dynamic services. The proposed approach consists in transparently integrating DSM into Open Services Gateway Initiative (OSGi) (OSGi, 2010) service model using containers and annotations. The authors use *Terracotta framework* (Terracotta Inc., 2008) as a kernel of the entire solution. Gelibert et al. (2011) point out limitations of using static types in the code, since instrumentation is done at runtime, thus the compiler cannot perform static verification on the application code. This creates complicated debugging scenarios when problems, especially transient ones, occur.

Programming Graphical Processing Unit (GPU) clusters with a distributed shared memory abstraction offered by a middleware layer is a promising solution for some specific problems, i.e. Single Instruction Multiple Data (SIMD) solutions. In (Karantasis and Polychronopoulos, 2011), an extension of *Pleiad* middleware (Karantasis and Polychronopoulos, 2009) is implemented enabling Java developers to work with a local GPU abstraction over several machines with one-four GPU devices each.

3 JCL ARCHITECTURE

This section details the architecture of the proposed middleware. The reflective capability of several programming languages, including Java, is an elegant way for middleware systems to introduce low coupling between distribution and business logic, as well as simplify the deployment process and introduce cloud multi-developer environments. Thus, reflection is the basis for many JCL features.

JCL has two versions: multi-computer and multi-core. The multi-computer version, named "Pacu", stores objects and executes tasks over a cluster where all communications are done via TCP/UDP protocol. On the other hand, the multi-core version, named

"Lambari", turns the User component into a local host component without the overhead of TCP/UDP communications. All objects and tasks are respectively stored and executed locally on the end-user machine. The architecture of JCL is composed of three main components: *User*, *Server*, and *Host*. While *User* is designed to expose the middleware services in a unique API to be adopted by developer, *Server* is responsible to manage the JCL cluster. Finally, the *Host* component is where the objects are stored and the registered methods are invoked. The next sections describe the design choices of JCL to provide the previously cited features.

3.1 User Component

To achieve portability, a single access point to JCL cluster is mandatory and the User component is responsible for that. It represents a unified API for both versions (multi-computer and multi-core) and it is where asynchronous remote method invocations and object storage take place. The user selects which version to start with according to a property file configured by the end-user. In the multi-core version, User avoids network protocols, performing shared memory communications with Host. In the multi-computer version, UDP and TCP/IP protocols are adopted, thus marshalling/unmarshalling, location, naming and several other components are introduced. These components are fundamental to distributed systems and are explained in details in Coulouris et al. (2007).

During the application execution, User component follows a pipeline composed of six steps: 1) receive end-user application calls; 2) generate unique identifiers for these calls; 3) return the identifiers to the end-user application; 4) schedule them; 5) submit them to Hosts; 6) and finally, store results of submitted calls locally. Since JCL is by default asynchronous, end-user application calls receive a ticket for each submitted task. After a complete execution of the pipeline above, the result is ready to be obtained using the identifier provided by User. Step 5 can be optimized in the multi-computer version if successive submissions occur, i.e. successive calls are buffered and submitted in batch to a Host.

This component adopts different strategies to schedule processing and storage calls in the multi-computer version. It allocates Hosts to handle processes according to the number of cores in the entire cluster. For instance, in a cluster with ten quad-core machines, we have forty cores, so User submits chunks of processing calls to the first machine, where each chunk size must be multiple of four, since it is a quadcore processor. Internally, a Host allocates a pool of threads, also with size multiple of four, to consume such processing calls. After the first chunk, User sends the second, the third and so on. After ten submissions, User starts submitting to the first machine again. The circular list behavior continues as long as there are processing calls to be executed. Heterogeneous clusters are possible, since JCL automatically allocates a number of chunks proportional to the number of cores of each machine.

There is a scheduling strategy for storage calls, so the User component calculates a function F to determine in which host the global variable will be stored (Equation 1), where $hash(vn)$ is the global variable name hashcode, nh is the number of JCL hosts and F is the node position. Experiments with incremental global variable names like "p_ij" or "p_i", where i and j are incremented for each variable and p is any prefix, showed that F achieves an almost uniform distribution for object storage over a cluster in several scenarios with different variable name combinations, however there is no guarantee of a uniform distribution for all scenarios. For this reason, User introduces a delta (d) property that normally ranges from $0-10\%$ of nh. The delta property relaxes function F result enabling two or more Hosts as alternatives to store a global variable.

$$F = \frac{|hash(vn)|}{nh} \tag{1}$$

In general, d relaxes a fixed JCL Host selection without introducing overhead in F. A drawback introduced by d is that JCL must check $(2*d)+1$ machines to search for an object, i.e., if d is equal to 2, JCL must check five machines (two machines before and two after the machine identified by function F in the logical ring). JCL checks all five alternatives in parallel, so the drawback is very small, as our experiments demonstrate. JCL with delta equals 2, 1 and 0 has similar execution time in clusters with 5, 10 and 15 machines.

3.2 Server Component

This component was designed to manage the cluster and is responsible for receiving the information from each Host and distributing it to all registered User components, enabling them to directly communicate with each Host. The Server also implements the possibility for the end-user to assign the placement of objects stored in the cluster, thereby disrespecting the Host selection obtained from the function F presented in Equation 1.

The Server fulfills the function of centralizing component, receiving the features of the computer where each Host is installed. Before adding a new Host, the Server notifies its presence to all registered Hosts that, after receiving the new member registration notification, recalculate the function F and change the Host objects, if necessary. After all changes have taken place, the Server is notified, fulfilling the registration of the new Host.

When there is an end-user application running, at least one Host registration needs to be completed successfully, so that such application can receive the cluster map, enabling direct communication with the registered Hosts and eliminating the necessity to search for a Host in the Server at every new demand.

One of JCL's advantages is the possibility of storing Java objects in a specific Host. In this case, the end-user can specify Hosts that are different from those calculated by function F. To guarantee that all running applications in a specific cluster have access to all the instantiated objects, the locations assigned manually by the end-user are centralized in the Server, this way they do not depend on the function F. It is possible to note that the increase of manually assigned variables concentrates the workload on the Server, thus variables with high amount of accesses can cause bottlenecks in the Server component. The end-user can also choose the Host to execute their methods, therefore JCL scales or allows its developers to scale their demands.

3.3 Host Component

JCL Host has two basic functions: to store the objects sent by User or by another Host and invoke previously registered class methods. Its architecture allows the Host component to dynamically determine the number of threads (workers) running the end-user class methods. By default, the application is configured to use the total number of cores available on the Host, that is, if the computer has four cores, four workers are created. Nevertheless, the user can create as many workers as necessary by simply setting the property that assigns the number of threads to be created. The justification for such feature is the possibility to combine CPU-bound methods with I/O bound method executions, requiring the operating system to schedule them, what increases the number of context switches, however such extra workload usually pays off, since there is a possibility of prefetching CPU bound method executions while waiting for I/O results.

Before the Host component publishes its services to User components, it starts a JCL pipeline composed of three steps: 1) the Host notifies the Server its intention to join the cluster; 2) the Server propagates the existence of a new Host; 3) the Server allows the Host to join JCL cluster.

A property that differentiates JCL from most of its counterparts is the class registration process that simplifies deployment. This process, invoked by User and performed by Host, adds the necessary classes to JVM classpath at runtime, which enables the end-user to store objects and remotely execute methods without manually registering the class in each JVM of each cluster Host.

To perform object storage, two different alternatives are adopted. In the first one, the end-user creates the object and sends it to be stored in a Host, which may or may not be chosen by him. In the second one, the end-user defines which object should be created, passing its arguments for the constructor and, therefore, enabling the object instantiation directly at the Host. It is possible to note that the second storage option allows the creation of massive objects at Host without transferring them through the data network.

As described previously in this section, the middleware is based on Java reflection. This way, there is no need to adapt any classes so that they can be executed at Hosts. Once registered, the target classes, as well as their dependencies, are sent to Host where methods are mapped and available for execution. This feature enables JCL applications to separate business logic code from distribution code, as well as simplifies deployment and enables distributed objects storage.

4 USE CASE

This section aims to evaluate JCL in terms of fundamental computer science algorithms development, such as sorting. The JCL BIG sorting application was implemented, since it represents a solution with intensive communication, processing and I/O.

The distributed BIG sorting application is a sorting solution where data are partitioned and also sorted, i.e. there is no centralized sorting mechanism. Data are generated and stored in a binary file by each Host thread, performing parallel I/O on each Host component. Data are integers between -10^9 to $+10^9$. The final sorting contains one million different numbers and their frequencies distributed over a cluster, but the original input data were generated from two billion possibilities.

The sorting application is a simple and elegant sorting solution based on items frequencies. The frequency of each number of each input data partition is obtained locally by each Host thread and a chunk

```
24 JCL_facade jcl = JCL_FacadeImpl.getInstance();
25
26 int numJCLclusterCores = jcl.getClusterCores();
27 //registering
28 jcl.register(Random_Number.class, "Random_Number");
29
30 //builds the input data, partitioned over JCL cluster
31 Object[][] args = new Object[numJCLclusterCores][];
32 for(int i=0;i<numJCLclusterCores;i++) {
33     Object[] oneArg = {sementes, "output"+i};
34     args[i]= oneArg;
35 }
36 List<String> tickets = jcl.executeAllCores("Random_Number", "Create1GB", ↩
       args);
37 jcl.getAllResultBlocking(tickets);
38 for(String aTicket:tickets) jcl.removeResult(aTicket);
39 tickets.clear();
40 tickets=null;
41 System.err.println("Time to create input (sec): " + ↩
       (System.nanoTime()-time)/1000000000);
```

Figure 1: Main class - how to generate pseudo-random numbers in the JCL cluster.

strategy builds a local data partition for the entire JCL cluster, i.e. each thread knows how many JCL threads are alive, so all number frequencies (nf) divided by number of cluster threads (nct) create a constant C, i.e. $C = nf/nct$. Each different number in an input data partition is retrieved and its frequency is aggregated in a global frequency GF. When GF reaches C value, a new chunk is created, so C is fundamental to produce chunks with similar number frequencies without storing the same number multiple times. When JCL avoids equal number values it also reduces communication costs, since numbers of one Host thread must be sent to other threads in the JCL cluster to perform a fair distributed sorting solution.

The sorting is composed of three phases, besides the data generation and a validation phase to guarantee that all numbers from all input data partitions are retrieved and checked against JCL sorting distributed structure. The sorting has approximately 350 lines of code, three classes and only the main class must be a JCL class, i.e. inherit JCL behavior. The pseudo-aleatory number generation phase illustrates how JCL executes existing sequential Java classes on each Host thread with few instructions (Figure 1). Lines 24, 26 and 28 of the main class illustrate how to instantiate JCL, obtain JCL cluster number of cores and register a class named "Random_Number" in JCL, respectively. Lines 31-35 represent all arguments of all "Create1GB" method calls, so in our example we have "numJCLClusterCores" method arguments and each of them is a string labeled "output_suffix", where the suffix varies from 0 to "numJCLClusterCores" variable value.

Line 36 represents a list of tickets, adopted to store all JCL identifiers for all method calls, since JCL is by default asynchronous. The JCL method "executeAllCores" executes the same method "Create1GB" in all Host threads with unique arguments on each method call. Line 37 is a synchronization barrier, where big sorting main class waits until some tasks, identified by "tickets" variable, have finished. From lines 38-40 objects are destroyed locally and remotely (line 38), and finally in line 41 there is the time elapsed to generate pseudo-random numbers over a cluster of multi-core machines and in parallel. The "Random_Number" class is a sequential Java class and method "Create1GB" adopts Java Random math class to generate 1GB numbers on each input data partition binary file.

Phases one, two and three are similar to Figure 1, i.e. they are inside the main class and they behave basically splitting method calls over the cluster threads and then waiting all computations to end. Precisely, at phase one JCL reads the input and produces the set of chunks, as well as each chunk frequency or the frequencies of its numbers. C is calculated locally in phase one, i.e. for a single input data partition, so in C equation nf represents how many numbers an input data partition contains and nct represents the number of JCL Host threads. Phase one finishes its execution after storing all number frequencies locally in a JCL Host to avoid a second file scan. It is possible to note that phase one does not split the numbers across the local chunks, since the algorithm must ensure a global chunk decision for that.

After phase one, the main class constructs a global

```
169 long load=0; int b; String result = "";
170 for(Integer ac:sorted){
171     load+=map.get(ac);
172     if(load>(totalF/(numOfJCLThreads))){
173         b=ac;
174         result+=b+":";
175         load=0;
176     }
177 }
```

Figure 2: Main class - how to mount the global chunk schema to partition the cluster workload.

```
97 for(int r=0;r<numJCLThreads;r++) {
98 JCLMap<Integer, Map<Integer, Long>> h = new JCLHashMap<Integer, ↩
       Map<Integer,Long>>(String.valueOf(r));
99 h.put(id, final[r]);
```

Figure 3: Sorting class - how to deliver chunks to other Host threads.

sorting schema with fair workload. Figure 2 illustrates how the main class produces chunks with similar number frequencies. Each result of phase one contains a schema to partition the cluster workload, so a global schema decision must consider all numbers inside all chunks of phase one.

The main class calculates the total frequency of the entire cluster, since each thread in phase one also returns the chunk frequency. Variable "totalF" represents such a value. Lines 169 to 177 represent how JCL sorting produces similar chunks with a constant C as a threshold. The global schema is submitted to JCL Host threads and phase two starts.

Phase two starts JCL Host threads and each thread can obtain the map of numbers and their frequencies, generated and stored at phase one. The algorithm just scans all numbers and inserts them into specific chunks according to the global schema received previously. Phase two ends after inserting all numbers and their frequencies into JCL cluster to enable any JCL Host thread to access them transparently. Figure 3 illustrates JCL global variable concept, where Java objects lifecycles are transparently managed by JCL over a cluster. The sorting class obtains a global JCL map labelled "h" (Figure 3). Each JCL map ranges from 0 to number of JCL threads in the cluster (line 97), so each thread manages a map with its numbers and frequencies, where each map entry is a chunk of other JCL Host thread, i.e. each JCL Host thread has several chunks created from the remaining threads. Line 99 of Figure 3 represents a single entry in a global map "h", where "id" represents the current JCL Host thread identification and "final" variable represents the numbers/frequencies of such a chunk. Phase three of sorting application just merges all chunks into a unique chunk per JCL Host thread. This way, JCL guarantees that all numbers are sorted, but not centralized in a Server or Host component, for instance.

Our sorting experiments were conducted with JCL multi-computer version. The first set of experiments evaluated JCL in a desktop cluster composed of 15 machines, where 5 machines were equipped with Intel I7-3770 3.4GHz processors (4 physical cores and 8 cores with hyper-threading technology) and 16GB of of RAM DDR 1333Mhz, and the other 10 machines were equipped with Intel I3-2120 3.3GHz processors (2 physical cores and 4 cores with hyper-threading technology) and 8GB of RAM DDR 1333Mhz. The Operating System was a Ubuntu 14.04.1 LTS 64 bits kernel 3.13.0-39-generic and all experiments could fit in RAM memory. Each experiment was repeated five times and both higher and lower runtimes were removed. An average time was calculated from the three remaining runtime values. JCL distributed BIG Sorting Application version sorted 1 TB in 2015 seconds and the OpenMPI version took 2121 seconds, being JCL 106 seconds faster. Both distributed BIG sorting applications (JCL and MPI) implement the explained idea and are available at JCL website.

The second experiments evaluated JCL in an embedded cluster composed of two raspberry pi devices, each one with an Arm ARM1176JZF-S processor, 512MB of RAM and 8GB of external memory, and one raspberry pi 2 with a quadcore processor operating at 900MHz, 1GB of RAM and 8GB of external memory. The Operating System was Raspbian Wheezy and all experiments could fit in RAM memory. Each experiment was repeated five times and both higher and lower runtimes were removed. An average time was calculated from the three remaining runtime values.

The JCL distributed BIG sorting was modified to enable devices with low disk capacity to also sort a big amount of data. Basically, the new sorting version does not store the pseudo-random numbers in external memory. It gathers the number generation phase with the phase where number frequencies are calculated. Differently from other IoT middleware systems (Perera et al., 2014), where small devices such as raspberry pi are adopted only for sensing, JCL introduces the possibility to implement general purpose applications and not only sensing ones. Furthermore, JCL sorting can run on large or small clusters, as well as massive muti-core machines with a unique portable code. The small raspberry pi cluster sorted 60GB of data in $2,7$ hours.

5 EXPERIMENTAL EVALUATION

Experiments were conducted with JCL multi-computer and multi-core versions. Initially, the JCL middleware was evaluated in a desktop cluster composed of 15 machines, the same cluster used to test JCL distributed BIG sorting application. The middleware was evaluated in terms of throughput, i.e., the number of processed JCL operations per second. The goal of these experiments is to stress JCL measuring how many executions it supports per second and also how uniform function F, presented in Equation 1, can be when both incremented global variable names and random names are adopted.

In the first set of experiments, we tested JCL asynchronous remote method invocation (Figure 4). For each test we fixed the number of remote method invocations to 100 thousand executions. JCL Protocol Buffer Algorithm (PBA) algorithm was adopted, so JCL differentiates the sizes of both machines of the cluster to configure the workload. The experiments were composed of two different methods: the first one is a void method with a book as argument, where a book is a user type class composed of authors, editors, edition, pages and year attributes (Figure 4 A); and the second method is composed of an array of string and two integer values as arguments which are adopted by algorithms for calculating Levenshtein distance, Fibonacci series and prime numbers (Figure 4 B). We measured the throughput of each cluster configuration (5, 10 and 15 machines).

The results demonstrated that JCL's throughput rises when cluster size increases as the task becomes more CPU bound. There is a throughput decrease when the cluster increases from 5 to 10 machines and

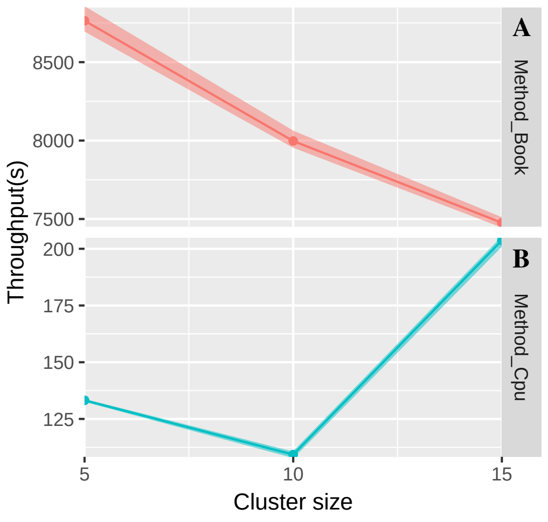

Figure 4: Method invocation.

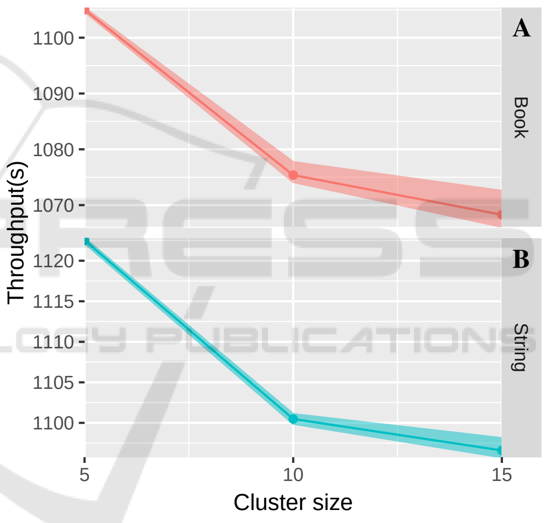

Figure 5: Global variable experiments.

the justification is that the new five Hosts are smaller than the first five ones, so the throughput does not increase at the same rate (Figure 4 B). The second test represents non CPU bound scenarios, so it is clear that network overhead is greater than task processing (Figure 4 A).

In the second set of experiments (Figure 5), we fixed the number of instantiated global variables to 40 thousand instantiations. We tested the book class instantiation explained previously (Figure 5 A) and also a smaller object like a string with 10 characters (Figure 5 B). We tested the five best machines first and then added the ten worst machines, thus this strategy may influence the throughput results negatively. As the cluster enlarges, the number of connections and other issues also become time-consuming, thus a re-

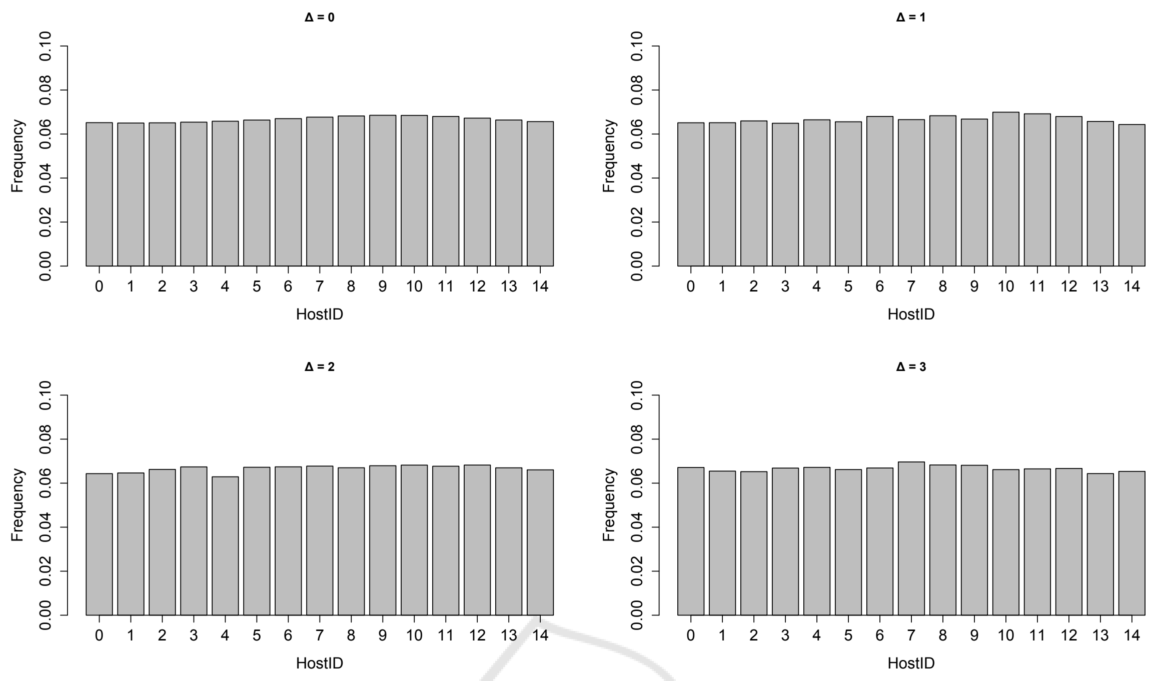

Figure 6: Variable names with autoincrement.

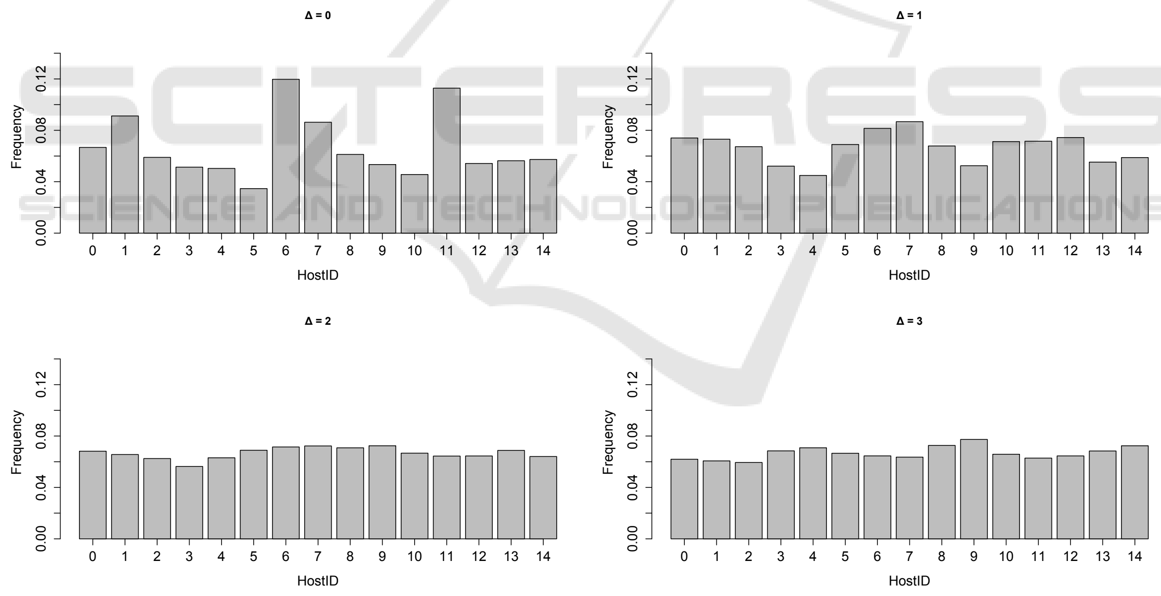

Figure 7: Bag of words.

duction in throughput should be expected. Another important observation is the synchronous behavior of JCL shared memory services, which is another bottleneck when the cluster becomes bigger. The results demonstrate that as the global variable becomes more complex, the data network overhead increases. Precisely, the throughput of String variable reduces 2,3% when JCL cluster increases from 5 to 15 Hosts, but when book variable is adopted, the throughput reduces 3,3% in the same cluster configurations. The positive aspect of such a scenario is that JCL over bigger clusters stores more data.

In the third set of experiments, we evaluated the uniformity of function F presented in Equation 1 plus a delta d and instantiated 40 thousand variable names. JCL cluster size was set to 15 machines, and then we tested different prefix variable names, and also autoincrement suffixes, i.e., variable names like "p_i" and

"p_ij", where p is a prefix and i and j are autoincrement values. We also tested $F+d$ distribution for an arbitrary bag of words and chose the Holy Bible's words to verify how JCL data partition performs. The results are illustrated in Figure 6 and 7, where Δ is delta size. Usually, JCL achieves an almost uniform distribution using delta between zero and two.

The result of the bag of arbitrary words becomes more uniform as delta increases, so even when the end-user decides to adopt arbitrary variable names in the code, JCL can achieve an almost fair data partition. We also tested the JCL overhead when a variable content is retrieved using delta zero, one and two, and there are almost no overhead varying deltas, but data partition uniformity is reduced as delta tends to zero, what can be seen in Figure 7. The justification is that network communication times for data checkings are irrelevant when compared with remote instantiation runtimes.

Finally, we also evaluated JCL multi-core version against a Java thread implementation provided by Oracle. An Intel I7-3770 3.4GHz processor with 8 cores, including hyper-threading technology, and 16GB of RAM was used in the experiment. We implemented a sequential version for a CPU bound task composed of existing Java sequential algorithms for calculating Levenshtein distance, Fibonacci series and prime numbers. We calculated JCL and Java threads speedups, and the results demonstrated similar speedups, i.e. in a machine with four physical cores and four virtual cores, JCL achieved speedup of 5.61 and Oracle Java threads the speedup of 5.77.

6 CONCLUSION

In this paper, we present a novel reflective middleware that is able to invoke remote methods asynchronously and also manage Java objects lifecycle over a cluster of JVMs. JCL is designed for multi-core, multi-computer and hybrid computer architectures. End-users write portable JCL applications, where global variables are also multi-developer, so different applications can transparently share resources without explicit references over a computer cluster. JCL can execute existing Java code or JCL code, this way JCL can build complex applications. Reflection capabilities enable JCL to separate distribution from business logic, enabling both existing sequential code executions over many high performance computer architectures with zero changes and multiple distribution strategies for a single sequential algorithm according to a hardware specification. Deployment in JCL is not time consuming, i.e. a JCL cluster without end-user code is sufficient to run any Java application in JCL. No other middleware solution puts all these features together in a unique solution.

Experiments demonstrate that JCL is a promising solution, although many improvements must be done. JCL must implement security methods. End-users should be able to lock/unlock global variables and execute tasks from private groups over a single JCL cluster, enabling different collaboration levels or profiles. JCL must be fault tolerant in storage and processing. Future systems should be able to recover on their own. Self-stabilization, self-healing, self-reconfiguration and recovery-oriented computing implement several algorithms/protocols that can be incorporated into JCL. JCL must implement the concept of multi-server, therefore a JCL server can manage, for instance, a cluster of JCL hosts with invalid IPs and communicate with other JCL servers, providing a multi-cluster JCL solution. GPU execution abstractions, where location and copies are transparent to end-users, are fundamental to JCL. A heuristic based scheduler, where cloud requirements are considered, is also an important improvement to JCL. An API for sensing is fundamental to JCL for IoT. Cross-platform Host component, including platforms without JVM, with JVMs that are not compatible with JSR 901 (Java Language Specification) or platforms without operating system, are mandatory to IoT. Built-in modules for monitoring and administration should be added to JCL.

ACKNOWLEDGEMENTS

We thank José Estevão Eugênio de Resende and Gustavo Silva Paiva for helping with experiments, Universidade Federal de Ouro Preto (UFOP) and Instituto Federal de Minas Gerais (IFMG) for the infrastructure, and Fundação de Amparo à Pesquisa do Estado de Minas Gerais (FAPEMIG) for financial support.

REFERENCES

Brynjolfsson, E. M. (2012). Big data: The management revolution. *Harvard Business Review*, 90(10):6066.

Cohen, L. (2015). *Java Parallel Processing Framework*. Available from: ⟨http://www.jppf.org/⟩.[15 Dezember 2015].

Coulouris, G., Dollimore, J., and Kindberg, T. (2007). *Sistemas Distribuídos - 4ed: Conceitos e Projeto*. Bookman Companhia.

Di Sanzo, P., Quaglia, F., Ciciani, B., Pellegrini, A., Didona, D., Romano, P., Palmieri, R., and Peluso, S.

(2014). A flexible framework for accurate simulation of cloud in-memory data stores. *arXiv preprint arXiv:1411.7910*.

Egan, S. (2005). *Open Source Messaging Application Development: Building and Extending Gaim*. Apress.

Forum, M. P. (1994). Mpi: A message-passing interface standard. Technical report, Knoxville, TN, USA.

Gelibert, A., Rudametkin, W., Donsez, D., and Jean, S. (2011). Clustering osgi applications using distributed shared memory. In *Proceedings of International Conference on New Technologies of Distributed Systems (NOTERE 2011)*, pages 1–8.

Ghosh, S. (2014). *Distributed systems: an algorithmic approach*. CRC press.

Gokhale, A., Balasubramanian, K., Krishna, A. S., Balasubramanian, J., Edwards, G., Deng, G., Turkay, E., Parsons, J., and Schmidt, D. C. (2008). Model driven middleware: A new paradigm for developing distributed real-time and embedded systems. *Science of Computer programming*, 73(1):39–58.

Han, J., Kamber, M., and Pei, J. (2011). *Data Mining: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 3rd edition.

Henning, M. and Spruiell, M. (2006). Distributed programming with ice reading.

Hindman, B., Konwinski, A., Zaharia, M., Ghodsi, A., Joseph, A. D., Katz, R., Shenker, S., and Stoica, I. (2011). Mesos: A platform for fine-grained resource sharing in the data center. In *Proceedings of the USENIX Conference on Networked Systems Design and Implementation (NSDI 2011)*, pages 295–308.

Kaminsky, A. (2015). *Big CPU, Big Data: Solving the World's Toughest Computational Problems with Parallel Computing*. Unpublished manuscript. Retrieved from http://www.cs.rit.edu/˜ark/bcbd.

Karantasis, K. and Polychronopoulos, E. (2011). Programming gpu clusters with shared memory abstraction in software. In *Proceedings of Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP 2011)*, pages 223–230.

Karantasis, K. I. and Polychronopoulos, E. D. (2009). Pleiad: A cross-environment middleware providing efficient multithreading on clusters. In *Proceedings of ACM Conference on Computing Frontiers (CF 2009)*, pages 109–116.

Murphy, A. L., Picco, G. P., and Roman, G.-C. (2006). Lime: A coordination model and middleware supporting mobility of hosts and agents. *ACM Trans. Softw. Eng. Methodol.*, 15(3):279–328.

Nester, C., Philippsen, M., and Haumacher, B. (1999). A more efficient rmi for java. In *Proceedings of the ACM 1999 conference on Java Grande*, pages 152–159. ACM.

OSGi (2010). Osgi specification release 4.2.

Perera, C., Liu, C. H., Jayawardena, S., and Chen, M. (2014). A survey on internet of things from industrial market perspective. *Access, IEEE*, 2:1660–1679.

Pitt, E. and McNiff, K. (2001). *Java.Rmi: The Remote Method Invocation Guide*. Addison-Wesley Longman Publishing Co., Inc.

Seovic, A., Falco, M., and Peralta, P. (2010). *Oracle Coherence 3.5*. Packt Publishing Ltd.

Taboada, G. L., Ramos, S., Expósito, R. R., Touriño, J., and Doallo, R. (2013). Java in the high performance computing arena: Research, practice and experience. *Science of Computer Programming*, 78(5):425–444.

Taboada, G. L., Touriño, J., and Doallo, R. (2009). Java for high performance computing: assessment of current research and practice. In *Proceedings of the 7th International Conference on Principles and Practice of Programming in Java*, pages 30–39. ACM.

Tanenbaum, A. S. and Van Steen, M. (2007). *Distributed systems*. Prentice-Hall.

Tariq, M. A., Koldehofe, B., Bhowmik, S., and Rothermel, K. (2014). Pleroma: a sdn-based high performance publish/subscribe middleware. In *Proceedings of the 15th International Middleware Conference*, pages 217–228. ACM.

Taveira, W. F., de Oliveira Valente, M. T., da Silva Bigonha, M. A., and da Silva Bigonha, R. (2003). Asynchronous remote method invocation in java. *Journal of Universal Computer Science*, 9(8):761–775.

Team, I. (2015). *Infinispan 8.1 Documentation*. Available from: ⟨http://infinispan.org/docs/8.1.x/index.html⟩.[15 Dezember 2015].

Terracotta Inc. (2008). *The Definitive Guide to Terracotta: Cluster the JVM for Spring, Hibernate and POJO Scalability*. Springer Science & Business.

Troester, M. (2012). Big data meets big data analytics. *Cary, NC: SAS Institute Inc*.

Veentjer, P. (2013). *Mastering Hazelcast*. Hazelcast.

Walker, S. M., Dearle, A., Norcross, S. J., Kirby, G. N. C., and McCarthy, A. (2003). Rafda: A policy-aware middleware supporting the flexible separation of application logic from distribution. Technical report, University of St Andrews. Technical Report CS/06/2.

Watson, R. T., Wynn, D., and Boudreau, M.-C. (2005). Jboss: The evolution of professional open source software. *MIS Quarterly Executive*, 4(3):329–341.

Xiong, J., Wang, J., and Xu, J. (2010). Research of distributed parallel information retrieval based on jppf. In *2010 International Conference of Information Science and Management Engineering*, pages 109–111. IEEE.

Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S., and Stoica, I. (2010). Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, pages 10–10.

Zhang, Q., Cheng, L., and Boutaba, R. (2010). Cloud computing: state-of-the-art and research challenges. *Journal of internet services and applications*, 1(1):7–18.