

UNIVERSIDADE FEDERAL DE OURO PRETO

ANDRÉ LUÍS BARROSO ALMEIDA

**A high performance Java middleware for general
purpose computing and capacity planning**

Ouro Preto

2016

UNIVERSIDADE FEDERAL DE OURO PRETO

ANDRÉ LUÍS BARROSO ALMEIDA

**A high performance Java middleware for general
purpose computing and capacity planning**

Dissertação de Mestrado submetida ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Ouro Preto como requisito parcial para a obtenção do título de Mestre.

Orientador:
Joubert de Castro Lima

Ouro Preto

2016

A447h

Almeida, André Luís Barroso.

A high performance Java middleware for general purpose computing and capacity planning [manuscrito] / André Luís Barroso Almeida. - 2016.

72f.: il.: color; grafs; tabs.

Orientador: Prof. Dr. Joubert de Castro Lima.

Dissertação (Mestrado) - Universidade Federal de Ouro Preto. Instituto de Ciências Exatas e Biológicas. Departamento de Computação. em Ciência da Computação.

Área de Concentração: Ciência da Computação.

1. Computação de alto desempenho. 2. Gerenciamento de memória (Computação) . 3. Interfaces (Computador) - Remote Method Invocation. 4. Programas de computador - Middleware. 5. Java (Linguagem de programação de computador). I. Lima, Joubert de Castro. II. Universidade Federal de Ouro Preto. III. Título.

CDU: 004.45



Ata da Defesa Pública de Dissertação de Mestrado

Aos 14 dias do mês de outubro de 2016, às 13:30 horas na Sala de Seminários do DECOM no Instituto de Ciências Exatas e Biológicas (ICEB), reuniram-se os membros da banca examinadora composta pelos professores: **Prof. Dr. Joubert de Castro Lima (presidente e orientador), Prof. Dr. Ricardo Augusto Rabelo Oliveira e Prof. Dr. Fábio Moreira Costa**, aprovada pelo Colegiado do Programa de Pós-Graduação em Ciência da Computação, a fim de argüirem o mestrando **André Luís Barroso Almeida**, com o título “**A High Performance Java Middleware for General Purpose Computing and Capacity Planning**”. Aberta a sessão pelo presidente, coube ao candidato, na forma regimental, expor o tema de sua dissertação, dentro do tempo regulamentar, sendo em seguida questionado pelos membros da banca examinadora, tendo dado as explicações que foram necessárias.

Recomendações da Banca:

() Aprovada sem recomendações

() Reprovada

(X) Aprovada com recomendações: considerar as recomendações da banca

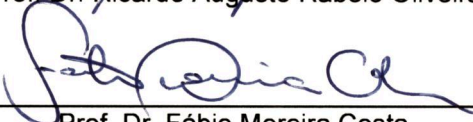
Banca Examinadora:




Prof. Dr. Joubert de Castro Lima



Prof. Dr. Ricardo Augusto Rabelo Oliveira



Prof. Dr. Fábio Moreira Costa



Prof. Dr. Anderson Almeida Ferreira
Coordenador do Programa de Pós-Graduação em Ciência da Computação
DECOM/ICEB/UFOP

Ouro Preto, 14 de outubro de 2016.

Resumo

Middlewares ou Frameworks são fundamentais no desenvolvimento de aplicações distribuídas devido a complexidade das mesmas. Muitas soluções foram propostas nas últimas três décadas de melhorias e a linguagem Java faz parte destes esforços. A comunidade Java é enorme e a linguagem oferece suporte para computação de alto desempenho (HPC), assim como para plataformas de pequeno porte, tais como as adotadas para IoT. Os middlewares Java para HPC implementam funcionalidades, tais como escalonamento de processos, tolerância a falhas, portabilidade de código, instalação simplificada em grandes clusters, desenvolvimento colaborativo na pilha de serviços em nuvem, execução de código existente sem refatoração, suporte a estruturas de dados distribuídas e nativas, execução de tarefas de forma assíncrona no cluster, suporte a criação de variáveis globais distribuídas, conceito de super-pares e muitas outras melhorias. Infelizmente, tais funcionalidades nunca foram reunidas em uma API única de uma solução de middleware simples e rápida. Neste trabalho, é apresentado o Java Cá&Lá ou simplesmente JCL, um middleware para desenvolvedores Java que adota computação reflexiva e possui modelo de programação baseado em endereçamento compartilhado e distribuído. O JCL reúne diversas funcionalidades apresentadas separadamente nas últimas décadas, permitindo construir aplicações paralelas ou distribuídas a partir de poucas instruções portáveis e sendo capaz de ser executado sobre diferentes plataformas, incluindo as IoT. Este trabalho apresenta as funcionalidades e a arquitetura do JCL, compara e contrasta JCL e seus concorrentes, e apresenta resultados experimentais de aplicações JCL.

Keywords: Java, General Purpose Computing, Middleware, High Performance Computing, Distributed Shared Memory, Remote Method Invocation.

Abstract

Middleware systems or frameworks are fundamental in the development of distributed applications due to their complexity. Several solutions were proposed in the last three decades of improvements and Java can be considered as part of these efforts. The Java community is huge and the language supports several features designed for high performance computing (HPC), but also for small platforms like the Internet of Things (IoT) ones. The Java middlewares proposed for HPC implement several features, such as scheduling, fault tolerance, code portability, simple deployment over large clusters, collaborative development in the cloud stack, execution of existing code without refactorings, native support for distributed data structures, asynchronous task execution, support for distributed global variables, super-peer concept and many others, but without integrating them. Unfortunately, these features were not put together in a simple and fast middleware solution. In this paper, we present Java Cálá or just JCL, a distributed shared memory lightweight middleware for Java developers that separates business logic from distribution issues during the development process and gathers several features presented separately in the last decades of middleware literature, allowing building distributed or parallel applications with few portable instructions and capable to run over different platforms, including small ones. This paper describes JCL's features, compares and contrasts JCL to other Java middleware systems, and reports performance measurements of JCL applications in several distinct scenarios.

Keywords: Java, General Purpose Computing, Middleware, High Performance Computing, Distributed Shared Memory, Remote Method Invocation.

Contents

List of Figures	vi
List of Tables	viii
1 Introduction	1
2 Features for HPC Middleware systems	4
3 Related work	8
4 JCL Architecture	14
4.1 Deployment	15
4.2 Refactoring	16
4.3 Scheduler	19
4.4 Distributed Hash Map	22
4.5 Collaboration	23
4.6 Portability	24
4.7 Super-peer	25
4.8 Task cost	27
5 Sorting Use Case	30
6 Experimental Evaluation	35
6.1 Throughput Experiments	35
6.2 Multi-core speedup experiments	41

6.3	Super-peer component overhead	41
6.4	CBC solver experiments	44
6.5	Experiments with a solution for the problem of minimizing the number of tool switches	48
7	Conclusion	54
	Bibliography	56

List of Figures

4.1	JCL architectures	15
4.2	JCL multi-computer deployment view	16
4.3	JCL multi-computer deployment view	17
4.4	JCL multi-computer deployment view	17
4.5	Business logic - Hello World.	18
4.6	Distribution logic - Hello World.	18
4.7	User component	19
4.8	Host Component	20
4.9	JCL HashMap	23
4.10	Developer one.	24
4.11	Developer two.	24
4.12	Super-peer topology	26
4.13	Super-peer component	27
5.1	Main class - how to generate pseudo-random numbers in JCL cluster.	31
5.2	Main class - how to mount the global chunk schema to partition the cluster workload.	32
5.3	Sorting class - how to deliver chunks to other Host threads.	33
6.1	Task execution experiments	36
6.2	Global variable experiments	37
6.3	Global variable experiments	39
6.4	JCLMap experiments	39
6.5	Variable names with autoincrement	40

6.6	Bag of words	40
6.7	Super-peer overhead in managing Hosts	42
6.8	Super-peer overhead topology 2	44
6.9	Task execution experiments	46
6.10	Tasks difference in terms of opened branches	47
6.11	Task runtimes distribution	47
6.12	Non-deterministic task execution experiments	48
6.13	Iterated Local Search applied to the MTSP 240 instances experiment. . . .	51
6.14	Iterated Local Search applied to the MTSP run rounds (1 - 64 Tasks) . . .	52
6.15	Iterated Local Search applied to the MTSP run rounds (96 - 160 Tasks) . .	52

List of Tables

1.1	Fundamental features for HPC Middleware systems	2
3.1	JCL and its counterparts' features - part 1	9
3.2	JCL and its counterparts' features - part 2	10
6.1	Number of Hosts	42
6.2	Super-peer overhead in different multi-cluster topologies	43

Chapter 1

Introduction

We live in a world where large amounts of data are stored and processed every day Han et al. (2011). According to the last International Data Corporation (IDC) report, in 2013 the amount of data stored reached 4.5 trillion gigabytes and this number will grow by a factor of 10, exceeding 40 trillion gigabytes in 2020 Turner et al. (2014). Despite the significant increase in the performance of today's computers, there are still 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 technologies that provide this new decentralized, dynamic and communication-intensive society.

Many fundamental services and sectors such as electric power supply, scientific/technological research, security, entertainment, finance, telecommunications, weather forecasting and many others, use solutions that require high processing power. Thus, these solutions, named high performance computing (HPC) applications, are executed over parallel and distributed computer architectures. 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 reduce the complexity of such development. The challenging issue is how to provide sufficient support and general high-level mechanisms using middleware for rapid development of distributed and parallel applications.

Among various programming languages for middleware systems, the interest in Java for High Performance Computing (HPC) is huge Taboada et al. (2013). This interest is based on many features, such as built-in networking, multithreading support, platform independence, reflection, portability, type safety, security, extensive Application Programming Interface (API) and a wide community of developers Taboada et al. (2009).

Besides programming models, language implementations and developers community size, several architectural, conceptual and implementation features, presented in Table 1.1, are fundamental in the design and development of a modern middleware solution.

Table 1.1: Fundamental features for HPC Middleware systems

General purpose computing (Shared memory, message passing, etc.)	Distributed Data Structures
Refactorings	Scheduling
Deployment	Super-peer
Collaboration	Task cost
Portability	Fault tolerance

As observed, distributed applications introduce many challenges to the development process, so middleware systems or frameworks become paramount. Features of the Table 1.1 were never put together in a middleware system, so the main goal of this paper is to present a middleware called JavaCá&Lá or just JCL¹ that fills most of this features, precisely: *i*) a simple deployment strategy and capacity to update internal modules during runtime; *ii*) a service to execute existing sequential Java code over multiple HPC architectures; *iii*) a unique *API* for capacity planning, storage and processing; *iv*) the support of super-peers for a multi-cluster topology; *v*) a distributed map that is a sub-type of Java Map interface; *vi*) an *API* to operate different platforms, including cloud and small ones like Raspberry Pi, Galileo, Cubieboard and many others Linux boards Upton and Halfacree (2014); Intel (2016); Schinagl (2014); *vii*) a service to get total time, network time, queue time, service time and the result retrieval time for each task submitted to the cluster; *viii*) a scheduler to manage the workload of a JCL cluster in terms of distributed/parallel processing and a hash function to achieve a fair distributed storage; *ix*) collaborative development without explicit dependencies; and *x*) multi-core/multi-computer portable code.

As a result of this paper, we have:

1. A middleware that gathers several features presented separately in the last decades of middleware literature, enabling building distributed applications with few portable instructions and capable to run over different platforms, including small ones;

2. A comparative study of market leaders and well established middleware standards for the Java community. This paper emphasizes the importance of several features and

¹Java Cá&Lá is available for download at <http://www.joubertlima.com.br/jcl>

how JCL and its counterparts fulfill them;

3. A scalable middleware over multi-core and multi-computer architectures;

4. A feasible middleware alternative to fast prototype portable Java HPC applications that separate business logic from distribution issues during the development process.

The paper is organized as follows. Chapter 2 discusses all important features required for the design and development of a modern middleware solution. Chapter 3 presents works that are similar to the proposed middleware, pointing out their benefits and limitations. Chapter 4 details how JCL middleware implements most of the features presented in Section 2. Chapter 6 presents our experimental evaluation and discusses the results. Chapter 5 describes a user case application. In chapter 7, we conclude our work and point out future improvements of JCL.

Chapter 2

Features for HPC Middleware systems

The design and development processes of a middleware system requires attempting several features, including implementation, architectural and conceptual ones. The features detailed in this chapter define a representative set of fundamental requirements for HPC middlewares and frameworks, so we evaluate JCL and the most similar literature according to them.

Refactorings: Usually, distributed shared memory middleware systems introduce some dependencies to HPC applications. Consequently, an ordinary Java or C++ object¹ must implement several middleware classes or interfaces 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 Map-reduce based solutions Hindman et al. (2011); Zaharia et al. (2010).

As a consequence of these dependencies, two problems emerge: *i*) the developer 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. Single Instruction Multiple Data (SIMD) applications are examples where sequential algorithms are replicated and executed over different data partitions. A zero-dependency middleware is necessary to solve this problems.

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

¹“An object is a self-contained entity consisting of data and procedures to manipulate data” Egan (2005)

(2013); Pitt and McNiff (2001), but sometimes updating during application runtime and without stopping 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 variables 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. Sometimes the collaborative environment requires access credentials to avoid prohibitive computations.

Portable Code: Portable multi-core/multi-computer code is an important aspect to consider during the development process, since in many institutions, such as research labs, there can be huge multi-core machines and several beowulf computer clusters Becker et al. (2002) to solve a handful 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 multi-core architectural designs in conjunction with multi-computer ones.

Distributed Data Structures: User typed object storage is implemented by many middleware systems, but few implement distributed data structures as part of a unified API Veentjer (2013); Watson et al. (2005); Team (2016). Usually, developers implement distributed storage using a specific framework or middleware, like HBase George (2011), Cassandra Cassandra (2013), Apache Pig Gates and Dai (2016), ScyllaDB Team (2015) and MongoDB Chodorow (2013). Often, third-party distributed storage solutions are focused on transaction aspects, i.e. database ACID (Atomicity, Consistency, Isolation, Durability) demands, so they are designed for applications with specific needs. Our focus is on local and sequential global variables adopted on every code. JCL and a few others extend Java collections standard APIs, such as Map, Set and List, which are part of Java since its beginning. Thus, little refactorings may occur when sequential global variables must be replaced with distributed ones.

Scheduling: HPC applications, in most of the cases, can be modeled as a SIMD solution, so the workload depends on data partition Boneti et al. (2008). Other problems can be modeled as a pipeline solution, where each pipe step can execute a different set of

instructions or a method, so pipeline steps normally have different workloads (multiple instruction and single data - MISD). Unfortunately, the load balancing problem emerges from both solutions, as Boneti et al. (2008) highlighted.

To reduce load balancing problems, we adopt scheduling algorithms Murata et al. (2006). Such algorithms goal is to reduce the workload difference from overloaded machines or cores by moving part of the load to underutilized ones Balasangameshwara and Raju (2012). Middleware systems like the Java Parallel Processing Framework (JPPF) Cohen (2015) and Gridgain GridGain Systems (2011) implement different scheduling techniques, but others, such as Java RMI Pitt and McNiff (2001) and MPI Forum (1994) , delegate scheduling issues to developers.

Super-peers: A super-peer is a node in a peer-to-peer network that both operates as a server to a set of clients and as an equal in a network of super-peers Yang and Garcia-Molina (2003). They take advantage of the heterogeneity of capabilities (e.g., bandwidth, processing power) across peers, but they also enable sub-networks with invalid IPs to be interconnected in a grid.

Furthermore, the super-peer concept turns possible the creation of multiple clusters, being each cluster organized according to developer needs (Ex. sensors clustered in a specific room of a house). In summary, the benefits of a super-peer extrapolates network infrastructure advantages. The hierarchical topology of super-peers is useful, as surveys pointed out Lua et al. (2005); SalemAlzboon et al., but all Java middleware systems found in literature just do not consider the advantages of it.

Fault tolerance: Fault tolerance in distributed computing is a very important feature that prevents data lost and corruption, but also processing malfunctioning. Middleware systems like HazelcastVeentjer (2013), Gridgain Team (2016) and Oracle CoherenceSeovic et al. (2010) implement fault tolerance for storage processes, performing double copies of distributed global variables. In terms of processing fault tolerance, JPPF resubmits the task when timeouts occur.

None of Java HPC middleware systems consider Byzantine fault tolerance ?, where a process is not only not responding, but producing incorrect results for many reasons. This hard problem is solved by a communication intensive protocol ?, requiring $2k+1$ correct processes for k corrupted ones.

General purpose computing: Middleware systems normally support a programming model based on shared memory, message passing or event Ghosh (2014). The shared

memory programming model considers global and local variables, but also the execution of tasks or procedures or methods as key abstractions. Event based and message passing programming models support other key abstractions, such as messages and events. The solutions capable to offer those abstractions are named general purpose solutions.

Middleware systems can be adopted for general purpose computing, 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).

Task cost: Middlewares systems like JPPFCohen (2015) and HazelcastVeentjer (2013) enable the user to monitor the health of every cluster member in terms of RAM, disk and CPU usage. Dashboards are implemented to visualize the cluster health, but for capacity planning it is fundamental to collect each task storage and processing costs, i.e. its different times and storage in each cluster member. The global variables sizes can also be monitored for better storage allocations.

These costs are fundamental to build scheduling algorithms or supervisory systems, adopted to delineate capacity planning of decentralized systems. A high standard deviation of the cluster queue time can indicate that the cluster cores are not enough and new members need be connected. The inverse can guarantee energy saving. Unfortunately, no related work implements such a detailed task cost model.

Chapter 3

Related work

In this chapter, 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*) implementation of simple deploy, *iii*) support for collaborative development, *iv*) implementation of both multi-core and multi-computer portable code, *v*) the cost of a task in terms of processing and storage, *vi*) super-peer concept, *vii*) implementation of distributed data structures, *viii*) scheduling support.

Other analyses were made to verify if the middleware has available support, and if it is fault tolerant in terms of storage and processing. Both academic and commercial solutions are considered and their limitations/improvements are highlighted in Tables 3.1 and 3.2. Middleware systems that present high similarities with JCL are described in detail in this chapter. The remaining related work is only briefly considered in Tables 3.1 and 3.2.

Infinispan by JBoss/RedHat Marchioni and Surtani (2012) is a popular open source distributed in-memory key-value data store solution Di Sanzo et al. (2014) which enables two ways to access the cluster: *i*) the first way uses an API available in a Java library; *ii*) the second way uses several protocols, such as HotRod, REST, Memcached and WebSockets Hickson (2011), making *Infinispan* a language independent solution. Besides storage services, the middleware can execute tasks remotely and asynchronously, but developers must implement Runnable or Callable interfaces. Furthermore, it is necessary to register these tasks in the Java virtual machine (JVM) classpath of each cluster node, as *Infinispan* does not have the dynamic loading class feature, which can delay the deployment process.

Java Parallel Processing Framework *JPPF* is an open source grid computing frame-

Table 3.1: JCL and its counterparts' features - part 1

Tool \ Feature	Fault Tolerant	Refactoring required	Simple Deploy	Collaborative	Portable Code
JCL	No	No	Yes	Yes	Yes
Infinispan (Marchioni and Surtani, 2012)	Yes	Low	No	Yes	Yes
JPPF (Cohen, 2015)	Yes	No	Yes	No	No
Hazelcast (Veentjer, 2013)	Yes	Low	No	Yes	No
Oracle Coherence (Seovic et al., 2010)	Yes	Medium	Yes	Yes	<i>NF</i> ¹
RAFDA (Walker et al., 2003)	No	No	Yes	Yes	No
PJ (Kaminsky, 2007)	No	Yes	Yes	No	Yes
FlexRMI (Taveira et al., 2003)	No	Medium	No	No	No
RMI (Pitt and McNiff, 2001)	No	Medium	No	No	No
Gridgain (GridGain Systems, 2011)	Yes	Low	No	Yes	No
ICE (Henning et al., 2013)	Yes	High	No	No	No
MPJ Express (Shafi et al., 2009)	No	Medium	No	No	Yes
Jessica (Zhu et al., 2002)	<i>NF</i> ¹	No	Yes	No	Yes
ProActive (Baduel et al., 2005)	Yes	<i>NF</i> ¹	<i>NF</i> ¹	<i>NF</i> ¹	No
F-MPJ (Taboada et al., 2012)	No	<i>NF</i> ¹	No	No	No
P2P-MPI (Genaud and Rattanapoka, 2007)	Yes	High	No	No	No
KaRMI (Philippsen et al., 1999)	No	<i>NF</i> ¹	No	No	<i>NF</i> ¹
RMIX (Kurzyniec et al., 2003)	No	<i>NF</i> ¹	No	No	No
open-mpi (Gabriel et al., 2004)	Yes	High	No	No	No
MPJava (Pugh and Spacco, 2003)	No	<i>NF</i> ¹	No	No	No

1 - NF: Not found

Table 3.2: JCL and its counterparts' features - part 2

Tool \ Feature	Task cost	Super-Peer	Distributed Data Structures	Scheduler	Support Available
JCL	Yes	Yes	Yes	Yes	Yes
Infinispan (Marchioni and Surtani, 2012)	<i>NF</i> ¹	<i>NF</i> ¹	Yes	Yes	<i>NF</i> ¹
JPPF (Cohen, 2015)	No	Yes	No	Yes	Yes
Hazelcast (Veentjer, 2013)	<i>NF</i> ¹	<i>NF</i> ¹	Yes	Yes	Yes
Oracle Coherence (Seovic et al., 2010)	<i>NF</i> ¹	<i>NF</i> ¹	Yes	Yes	<i>NF</i> ¹
RAFDA (Walker et al., 2003)	No	Yes	No	No	No
PJ (Kaminsky, 2007)	No	No	No	Yes	Yes
FlexRMI (Taveira et al., 2003)	No	No	No	No	No
RMI (Pitt and McNiff, 2001)	No	No	No	Yes	No
Gridgain (GridGain Systems, 2011)	No	Yes	Yes	Yes	Yes
ICE (Henning et al., 2013)	No	<i>NF</i> ¹	No	Yes	<i>NF</i> ¹
MPJ Express (Shafi et al., 2009)	No	No	No	Yes	<i>NF</i> ¹
Jessica (Zhu et al., 2002)	No	No	No	Yes	<i>NF</i> ¹
ProActive (Badel et al., 2005)	<i>NF</i> ¹	Yes	No	Yes	Yes
F-MPJ (Taboada et al., 2012)	No	No	No	No	No
P2P-MPI (Genaud and Rattanapoka, 2007)	No	No	No	Yes	Yes
KaRMI (Philippsen et al., 1999)	No	No	No	No	<i>NF</i> ¹
RMIX (Kurzyniec et al., 2003)	No	No	No	No	No
open-mpi (Gabriel et al., 2004)	No	No	No	Yes	No
MPJava (Pugh and Spacco, 2003)	No	No	No	No	No

1 - *NF*: Not found

work based on the Java language Xiong et al. (2010), which simplifies the process of parallelizing applications that demand high processing power, allowing developers to focus on their core software development Cohen (2015). It implements the dynamic loading class feature in cluster nodes, but it does not support collaborative development, i.e., methods and variables cannot be shared among different *JPPF* applications. *JPPF* has four predefined scheduler algorithms and they can be customized. Other features, such as fault tolerance and the possibility of interconnecting distinct networks, make *JPPF* one of the most complete solutions found in the literature.

Hazelcast Veentjer (2013) is a well-established 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 JVMs in the cluster is allowed to acquire that lock.” Veentjer (2013). Besides an API for asynchronous remote method invocations, Hazelcast has a simple API to store objects in a computer grid. Hazelcast does not separate business logic from distribution issues, so flexibility and dynamism are reduced. Hazelcast cannot instantiate a global variable remotely, i.e., it always maintains double copies of each variable at each instantiation, one local in the machine where the main code is executed and a second remote copy in a cluster node.

Hazelcast has the advantage of a manual scheduling alternative for global variables and executions, so the developer can opt to select the cluster machine to store or run an algorithm. It does not implement automatic deployment, so it is necessary to manually deploy each developer class, i.e., the JVM classpath must be manually changed before starting each middleware node, so deployment can become a time consuming activity. List, set and queue data structures are fault tolerant, but not distributed, since only map is a distributed and fault tolerant data structure in Hazelcast.

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 loading class feature in cluster nodes, but such agents must implement the **EntryProcessor** interface, thus refactorings are necessary. Boards with Linux support, like Raspberry Pi, Galileo, Cubieboard and others can be adopted for general purpose computing, but Oracle HPC products normally are not designed for small platforms.

ProActive Baduel et al. (2005) “ is Remote Method Invocation (RMI) based middle-

ware for parallel, multithreaded and distributed computing focused on Grid applications” Taboada et al. (2009). It offers an API for clusters or grids Amedro et al. (2009). In general, the use of RMI as its default transport layer adds significant overhead to Proactive operation. Proactive is fault tolerant and implements mobility and security transparencies.

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 to provide distributed access to ordinary Java classes. Applications call *RAFDA* functionalities using infrastructure objects named *RAFDA* runtime (RRT). Each RRT provides two interfaces to application programmers: one for local RRT access and the other for remote RRT access. As we can see, *RAFDA* introduces dependencies and, consequently, refactorings. RRT has peer-to-peer communication, so it is possible to execute a task in a specific cluster node, but if the developer 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.

The solution Parallel Java (PJ) Kaminsky (2007) has implemented several high-level programming abstractions, such as *ParallelRegion* (code to be executed in parallel), *ParallelTeam* (group of threads that execute a *ParallelRegion*) and *ParallelForLoop* (work parallelization among threads), allowing an easy thread-based shared memory programming model. Moreover, PJ is designed for hybrid shared/distributed memory systems such as multi-core clusters. It implements a message passing programming model for distributed computing, so the transparency of multi-core shared memory access is eliminated due to communication particularities. The middleware implements the concept of tuple space Murphy et al. (2006), but not in a distributed way. It has an API for Graphical Processing Unit (GPU) devices, offering Cuda Nvidia Corporation (2008) transparent services.

In the beginning of the 2000s, a middleware, named *FlexRMI*, was proposed by Taveira et al. (2003) to enable asynchronous remote method invocation using the standard Java RMI API. “FlexRMI is a hybrid model that allows both asynchronous or synchronous remote method 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 to decide 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. As with standard RMI, *FlexRMI* does not have a multi-core version to achieve portability, for instance. 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 loading class feature, all classes and interfaces must be stored in nodes before a RMI (and also FlexRMI) application starts, making deployment a time-consuming effort. No scheduler strategy is implemented.

Jessica Zhu et al. (2002) improves JVM to support distributed shared space for Java ordinary objects and threads, so Jessica enables thread migration. Java developers that are familiar with Java thread programming can easily develop applications with Jessica. Legacy Java thread applications can use Jessica transparently. JCL and the remaining related works are built on top of a JVM standard for a single machine, so we mention Jessica due to its high level of transparency, *i.e.*, it requires no new instructions to develop an asynchronous distributed thread based application. Jessica supports distributed shared objects, but it does not implement distributed data structures in the same way JCL and many of its counterparts do.

In Taboada et al. (2013), the authors present a survey on Java for HPC. The survey is a catalog of Middleware systems and libraries classified as shared memory, socket, RMI, and message-passing solutions. The Middleware systems were tested in two shared memory environments and two InfiniBand multi-core clusters using NAS Parallel Benchmarks (NPB) Bailey et al. (1991). The results showed that the Java language reached performance similar to natively compiled languages for both sequential applications and parallel applications, demystifying the concept that Java does not work for HPC.

Programming GPU clusters with a distributed shared memory abstraction offered by a middleware layer is a promising solution for some specific problems, *i.e.*, SIMD ones. 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.

Chapter 4

JCL Architecture

This section details the architecture of the proposed middleware and how it implements most of the features present in Table 1.1. The implementation adopts the reflective capability of Java programming language, so it incorporates an elegant way for middleware systems to introduce low coupling between distribution and business logic, as well as to simplify the deployment process and to introduce cloud-based multi-developer environments. Thus, reflection is the basis for many JCL features.

JCL has two versions: multi-computer and multi-core. Figure 4.1 illustrates how JCL multi-core and multi-computer architectures are designed. The multi-computer version, named "Pacu", stores objects and executes tasks over a cluster or multiple clusters, where all communications are done over IP version 4 (TCP and UDP communications, precisely). This version adopts a hybrid distributed architecture, i.e., it adopts a client-server behavior to provide location and registration services, but it also adopts a peer-to-peer (P2P) architecture style to provide processing and storage.

On the other hand, the multi-core version, named "Lambari", also present in the multi-computer version, 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 developer machine. All JCL applications are portable for both versions.

The architecture of JCL "Pacu" is composed of four main components (User, Server, Super-peer, and Host) and "Lambari" by only two (User and Host). The User component is designed to expose the middleware services in a unique API, as well as to provide scheduling and automatic version selection based on developer configurations. The Server component is responsible for managing the JCL clusters. The Host component is where

the objects are stored and the registered methods are invoked. JCL key-value pairs of a map are also stored in the Host component. The second phase of the JCL scheduling solution is solved in the Host component. Finally, the Super-peer component is responsible for the routing process and to partition a cluster according to logical entities of a specific domain. As an example, we can consider a smart house cluster that can be partitioned into several other clusters, one for the rooms, one for the swimming pool and many more, so all the machines of the house can be reorganized. The room cluster can have invalid IPs, so the Super-peer can also work as a bridge.

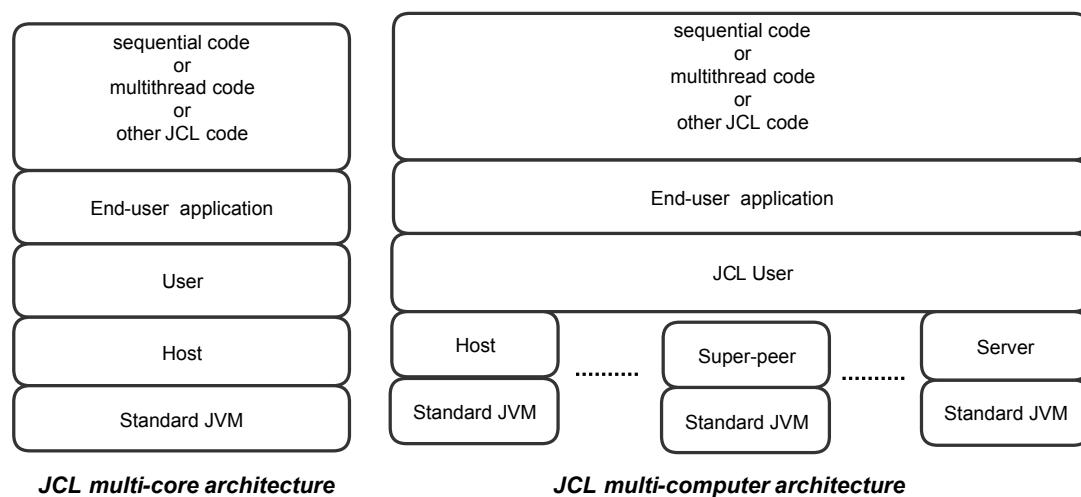


Figure 4.1: JCL architectures

In the following sub-sections, we present how JCL addresses most of the features illustrated in Table 1.1, precisely: deployment, refactoring, scheduling, distributed data structure, collaboration, portability, the super-peer concept and how to trace a task cost.

4.1 Deployment

The deployment process is a time consuming activity in most of the middleware systems. In some cases, its necessarily to reboot the system to make the deployment of a new application. To reduce the time consuming of this process, JCL adopts a simple deployment process base on Java reflective capacity.

The JCL simple deployment process is illustrated in Figures 4.2, 4.3 and 4.4. There is just one Server per JCL installation and it must be deployed first, since it registers and manages the remaining components (Figure 4.2-A). The Host components deployment

must occur after the Server in order to guarantee correct registrations, i.e., registrations in the same Server network. JCL supports one or many Hosts per cluster (Figure 4.2-B).

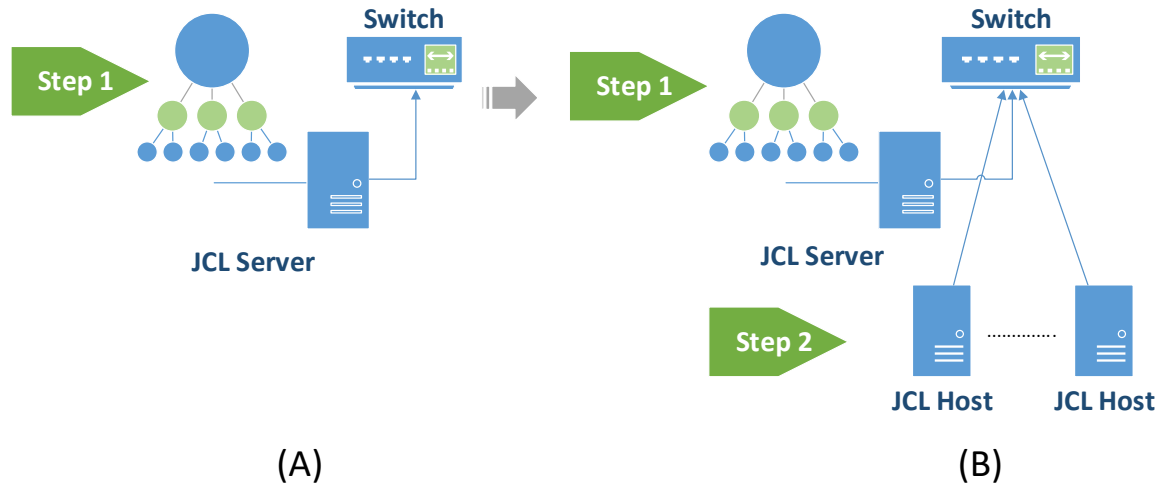


Figure 4.2: JCL multi-computer deployment view

Steps 3 and 4 of Figure 4.3 are optional, i.e., they are only adopted when we need to interconnect different data networks or when we want to create logical entities according to specific needs, such as a group of Hosts to attempt machine learning services or to collect sensing data from a garden of a smart house. The Super-peer component deployment occurs in a network gateway (Figure 4.3) or in the same network to create logical groups of Hosts.

In Figure 4.4, several User components are deployed on different machine types, precisely desktop and laptop ones. We assume each machine with a different developer application.

Following this procedure, the developer can deploy an application without reboot all JCL cluster. If is necessary to update a previously registered module, JCL only requires a new registration to perform all new deploys in the cluster, so even in live update scenarios the middleware does not stop its execution. To avoid register modules in the entire cluster all the time, there is a selective registration approach where only Hosts that will execute the developer application register it before the first execution.

4.2 Refactoring

One goal of JCL is to separate business logic from distribution issues. Normally, existing middleware solutions force their developers to implement several interfaces to guarantee distributed storage or asynchronous distributed executions. In JCL, there is no interface

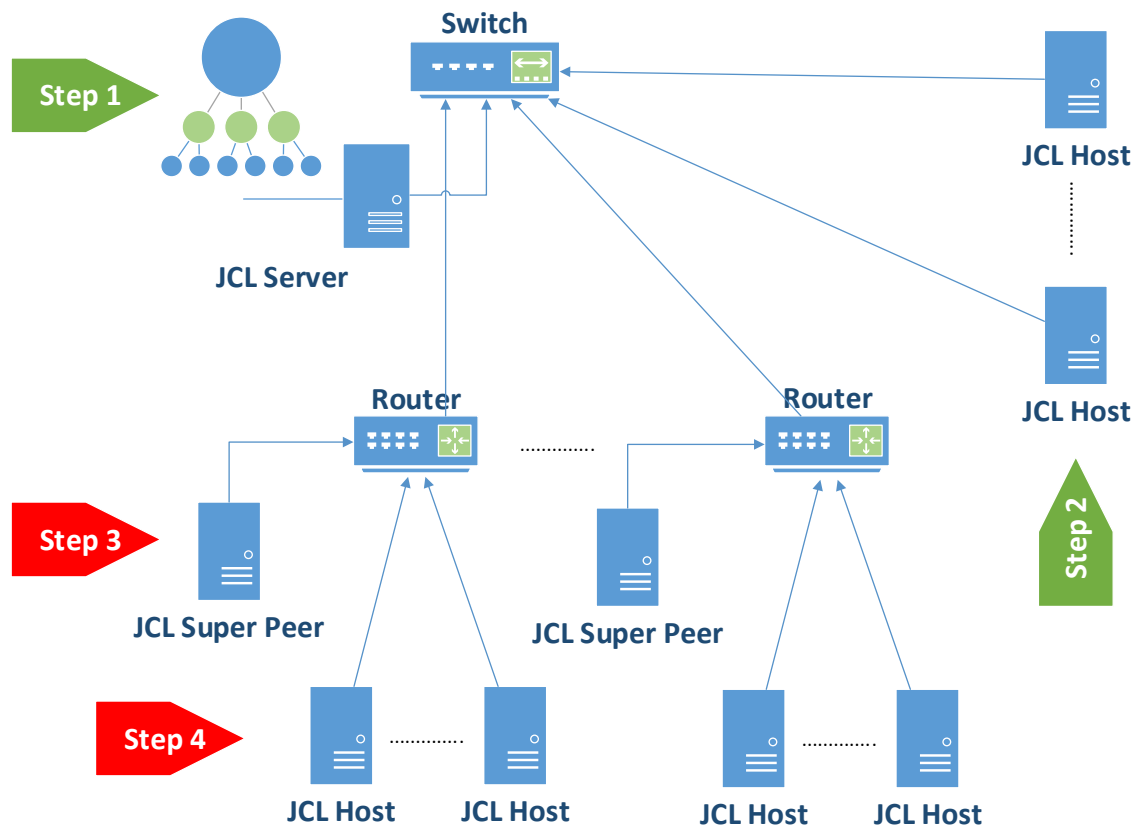


Figure 4.3: JCL multi-computer deployment view

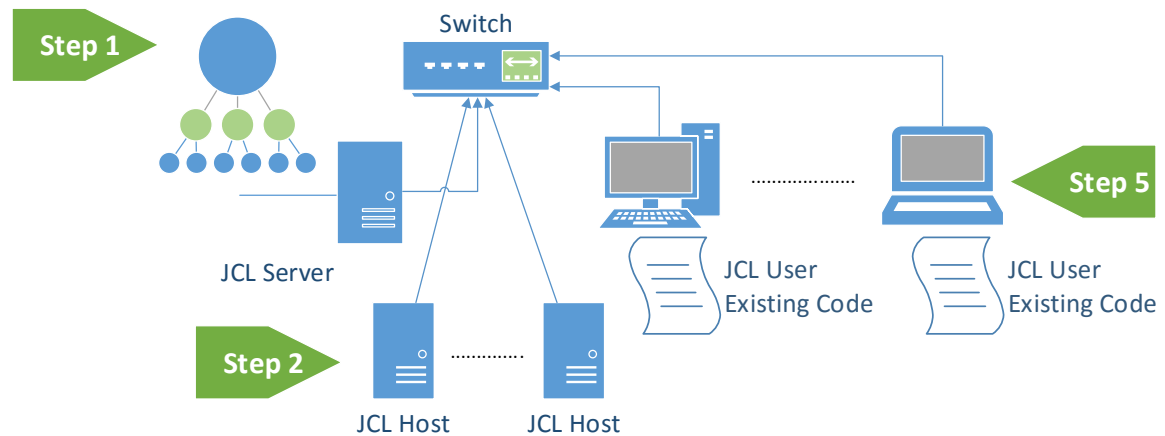


Figure 4.4: JCL multi-computer deployment view

to be implemented, since we adopt the reflective capability of Java language to avoid refactorings of existing and well tested methods, variables or algorithms.

To explain this feature, we use a well know application, the “Hello World”. Figure 4.5 illustrates a class with a sequential method named “print”. It represents the business logic of a developer. In our example, the method just print the sentence “Hello World!”, but the idea works for any other developer demand.

```
1 public class HelloWorld {
2
3     public void print() {
4         // Prints "Hello World!" in the console.
5         System.out.println("Hello World!");
6     }
7 }
```

Figure 4.5: Business logic - Hello World.

Figure 4.6 illustrates how JCL introduces distribution to an existing sequential code. At line four, the developer gets an instance of the JCL and at line five the class HelloWorld is registered, so it becomes visible to the entire JCL cluster. At line six, the developer requests a single execution of the method “print” of the registered class. JCL “execute” method requires the class nickname “Hello”, the method to be executed “print” and the arguments of such a method or null if no arguments are required.

```
1 public class JCLHelloWorld {
2
3     public static void main(String[] args) {
4         JCL_facade jcl = JCL_FacadeImpl.getInstance();
5         jcl.register(HelloWorld.class, "Hello");
6         String ticket = jcl.execute("Hello", "print", null);
7         // do any other task, including new JCL calls
8         jcl.getResultBlocking(ticket);
9     }
10 }
```

Figure 4.6: Distribution logic - Hello World.

In JCL, there are several ways to execute a task, including multiple executions of the same task on each Host or on each Host core, with or without the same task arguments. Furthermore, JCL enables the execution of complex applications with several dependencies. For that, it adopts Jar files as wrappers. In summary, the same principles explained for the “HelloWorld” example are maintained, so for better understanding we present the example as simple as possible. Finally, at line eight, the developer gets the result with a blocking call, i.e. the “JCLHelloWorld” class waits until the result is ready and returned to the machine where the “jcl.execute” was called.

The concept of ticket is adopted to implement the asynchronous mechanism in JCL. The same idea is adopted to store an existing and well tested object in JCL cluster, i.e. a registration is required and a JCL call is proceeded to instantiate an object remotely

or just to store a previously instantiated object in the cluster. Complex objects must be wrapped in Jar files, as explained before, and objects requiring arguments on their constructors are also possible.

4.3 Scheduler

JCL adopts different strategies to schedule task and global variable calls in the multi-computer version. The same assumptions to store ordinary global variable in JCL are used to store key-value pairs of distributed maps.

To schedule tasks, the User component allocates Hosts to handle them according to the number of cores available in the cluster. For instance, lets assume forty JCL calls in a cluster with ten quadcore machines. In this scenario, ORB Pacu (Figure 4.7) submits chunks of tasks to the first machine, where each chunk size must be multiple of four, since it is a quadcore processor. Internally, an ORB Lambari (Figure 4.7) allocates a pool of threads, also with size multiple of four, to consume such processing calls.

After the first chunk, ORB Pacu sends the second, the third and so on. After ten submissions, ORB Pacu starts submitting to the first machine again if needed. 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.

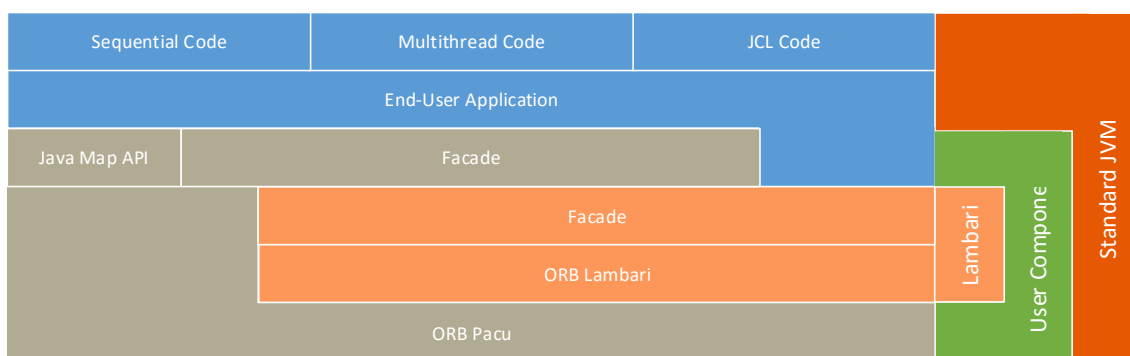


Figure 4.7: User component

To deal with applications where the number of processing calls is not proportional to the chunk size, JCL implements a watchdog. A watchdog is a thread that is started every 100 milliseconds and at each run it flushes the chunk, regardless the number of processing calls on it. The thread submits each task call to a specific Host, avoiding the idea to

submit chunks, since the watchdog timeout indicates that the application does not have many tasks to be executed by JCL.

There is a second scheduling phase, where the ORB of the Host components (Figure 4.8) collaborate to each other to turn JCL cluster workload more balanced. Each Host Worker thread, after executing its last task, tries to obtain and execute a new task from other threads in the JCL cluster. Just one task is obtained per time to avoid new redistributions in the JCL cluster. This collaborative behavior makes the cluster more balanced in terms of workload per Host thread, since it mitigates problems caused by the circular list scheduler, implemented in User.

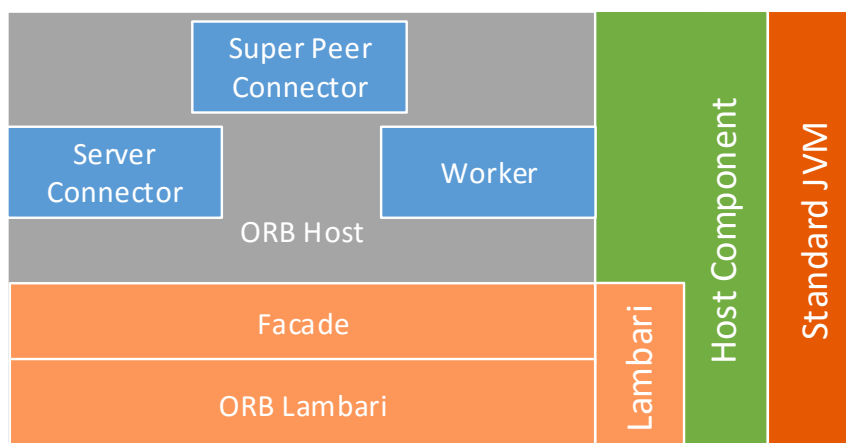


Figure 4.8: Host Component

A scenario where one Host receives the most CPU-bound tasks is possible to be achieved with a circular list scheduling technique, but the second phase redistributes these tasks with all other Host threads. A Host thread that handles a task from other Host must notify the User component to update its control data, since it contains each task metadata and the Host executing it. The Server component is not notified after the Host scheduler decisions to avoid architectural bottlenecks.

This technique for load balancing is classified in Patel et. al. (2016) as a Neighbor Based approach. It is a dynamic load balancing technique where the nodes transfer tasks among their neighbors, so after a number of iterations the whole system is balanced without introducing a global coordinator. The survey Patel et al. (2016) lists three algorithms that adopt such a strategy.

The first scheduling technique is PD_MinRC Balasangameshwara and Raju (2013),

a fusion of a neighbor based solution with MinRC, a fault tolerance scheme. The second is the AlgHybrid_LB Balasangameshwara and Raju (2012) technique, which is a hybrid load balancing algorithm that, initially, collects different control data from the cluster, i.e. the historical runtimes of tasks, the amount of free working memory in a node, the CPU utilization and many others. The third algorithm, named OP Balasangameshwara and Raju (2010), proposes that the interaction among nodes are fulfilled in pairs, which means that two nodes, the overload and the underload ones, try to transfer tasks to each other.

In JCL, we opt to reduce task retention in the first scheduling stage, so a circular list is faster than a sophisticated storage/processing collecting mechanism. Our assumption is that the second phase with a collaborative behavior mitigates efficiently any initial allocation drawback, so task retention becomes the hardest challenge for the first scheduling phase of a Neighbor Based approach.

To schedule global variable, the ORB at User component calculates a function F to determine in which Host the global variable will be stored. The equation 4.1 is responsible for a fair distribution, where $hash(vn)$ is the global variable name hashcode, nh is the number of JCL Hosts and F is the remainder of the division that correspond to the node position. JCL adopts the default Java hash code for string and primitive types, but user typed object requires a hashcode implementation.

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 global variable 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, the User component introduces a delta (d) property that normally ranges from 0 – 10% of nh . Delta property relaxes function F result, enabling two or more Hosts as alternatives to store a global variable.

$$F = Remainder\left(\frac{|hash(vn)|}{nh}\right) \quad (4.1)$$

A drawback introduced by d is that JCL must check $(2 * d) + 1$ machines to search for a global variable, 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).

The same F with d is adopted by the ORB of the User component to define where

each key-value pair of a distributed map is placed and also each map metadata. F is applied to each key and each map name, respectively, so maps become distributed with small overhead, since F calculus is fast.

The F function introduce a problem to the scheduler when a new Host was added to the cluster. In this scenario, the location of the global variable previously stored changes. To avoid storage replacements when a Host enters or exits the cluster, which can become a time consuming task, the Server and the User components maintain all previous cluster sizes after the first object instantiation in JCL (a global variable or a map instantiation, precisely), so F can be applied to each cluster configuration when data must be retrieved from the JCL cluster. All communications to obtain a certain global variable or a key-value pair are performed in parallel to reduce overhead.

4.4 Distributed Hash Map

JCL has a distributed implementation of the Java Map interface, allowing the developer to use a data structure that is familiar to the Java community. Its adoption requires minimal refactorings of existing Java code that considers the Java Map interface. In general, the developer just replaces a Map sub-type object (Ex. Tree Map, Hash Map, etc.) with a JCL Hash Map, so the storage, which before was done locally, is now done in a distributive manner over a cluster of multi-processor machines.

Internally, when the developer stores or requests a key-value pair of map objects, using for this the `put(key, value)` and `get(key)` methods, respectively, the hash of the object key is calculated and the location of the object is discovered using function F , described in section 4.3, which indicates the Host where the value is stored (Figure 4.9).

Each variable of the JCL Hash Map type has a single identifier provided by the developer at its creation. With such an identifier, any JCL application can have access to the map previously created in the cluster. Multiple JCL maps can have identical keys in the JCL cluster, however, different maps should have distinct identifiers to avoid overlaps in the cluster. In order to enable the efficient implementation of some methods of the Map interface, such as, `clear()`, `containsKey(Objectkey)` and `containsValue(Objectvalue)`, a list of all keys is stored in a Host.

To traverse the map items, the JCL Hash Map provides a new iterator implementation which, in order to optimize data transfer, initially identifies and gathers in bins all the keys of a map belonging to a single Host. Then, the first bin is sent to the User component,

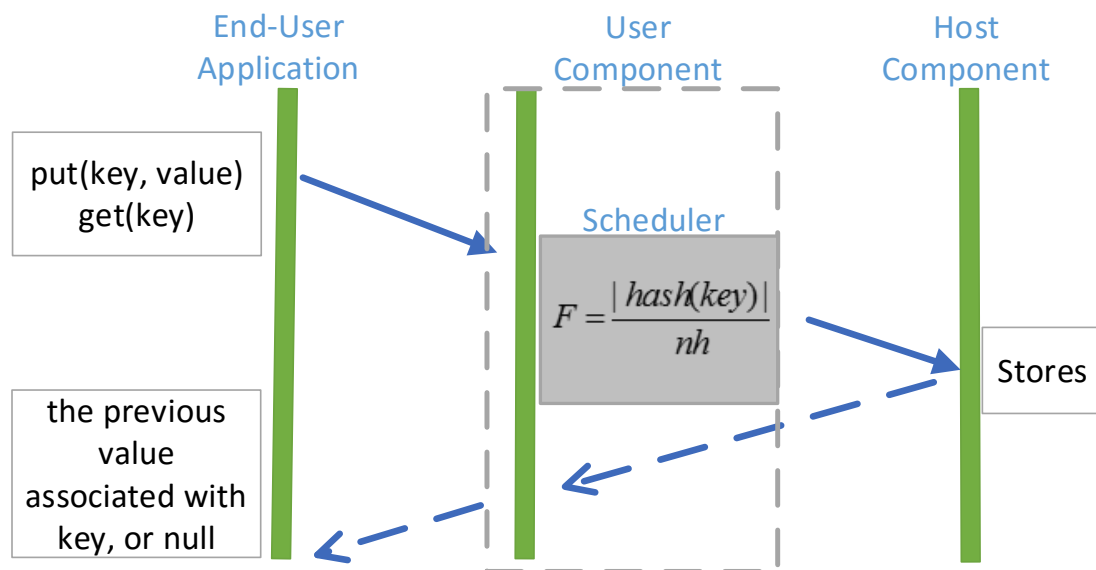


Figure 4.9: JCL HashMap

and after reading 40% of the key-value pairs already obtained by the User, the next bin is submitted until there are no more bins. The 40% value was obtained empirically after numerous tests with various types of objects for the key-value pairs. The anticipated load of bins is of fundamental importance to guarantee that massive maps can be traversed without stopping due to the communication between the User and the Host components.

Distributed mutual exclusion is also implemented at the level of individual keys, i.e., the developer calls the *getLock(key)* method, which guarantees safe and exclusive access to the value that represents the key. While the value is being manipulated by an application thread, another thread cannot write in this object. To unlock and release the access to other JCL cluster threads, it's necessary to call the *putUnlock(key, value)* method. The *put(key, value)* method of the Java Map Class is always thread-safe in the JCL implementation, yet the *get(key)* method just returns a value without blocking it.

4.5 Collaboration

JCL applications can share methods and global variables without explicit references, so a developer in a machine can access an object instantiated by another application using only its nickname. Using this features, developers around the world can share their algorithms and data structures with others and also share the computational power of multiple clusters.

The same collaborative development occurs with JCL maps. In summary, the developer just registers their Java classes to make it public to the remaining developers. In case of a hash map variable, the developer just calls JCL to create a map with a specific name, making it public and accessible to other applications over the cluster or the multi-cluster environment. For example, the developer one start a JCLMap named “Test” at line one of Figure 4.10 and put two key-value pairs, line two and three. The developer two can recover the Map named “Test” at line one and print the values of key “1” and “2” at line two and three of Figure 4.11.

```
1 Map<Integer, Integer> JCLMap = JCL_FacadeImpl.GetHashMap("Test");
2 JCLMap.put(0, 1);
3 JCLMap.put(1, 10);
```

Figure 4.10: Developer one.

```
1 Map<Integer, Integer> JCLMap = JCL_FacadeImpl.GetHashMap("Test");
2 System.out.println(JCLMap.get(0));
3 System.out.println(JCLMap.get(1));
```

Figure 4.11: Developer two.

If we consider a cloud environment with a JCL deployment (Server, multiple Hosts and Super-peers), multiple User components, deployed on different platforms of each developer machine, can code their JCL applications using a private/public cloud with the following services: register, storage, processing and task cost monitoring.

4.6 Portability

JCL was built with Java language, so it is portable to any Java Virtual Machine (JVM) that attempts Oracle specifications, thus JCL runs over massive multi-core machines, but also over small platforms, like Raspberry Pi, Galileo, Cubieborad and many other Linux boards with Java support. A big sorting distributed application, detailed in chapter 5, runs over a Raspberry pi cluster, demonstrating that JCL is small and light enough for new IoT demands.

JCL introduces its portability, so any JCL application can adopt “Pacu” or “Lambari” versions without changes. Different ways to instantiate a JCL Map, as well as different ways to execute a JCL task and different ways to instantiate/store an object are full compatible with both versions.

To achieve such a requirement, a single access point to both JCL versions (multi-computer and multi-core) is mandatory and the User component is responsible for that. It represents a unified API where asynchronous remote task invocations, their costs and global variables storage take place, as well as distributed Java maps.

The developer selects which JCL version to start via a property file or via API by calling static methods to get “Pacu” or “Lambari” versions. In the multi-core version, User component avoids network protocols, performing shared memory communications with Host. In the multi-computer version, IP version 4 protocols are adopted (TCP and UDP, respectively), 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. (2013).

4.7 Super-peer

In order to make the interface among networks and to add the capacity to partition the cluster into logical entities, JCL introduces the concept of Super-peer. This component was designed with two internal components. The first one behaves as a JCL Server component (referred to as Super-peer server) for a given network, and the second behaves as a JCL Host component (referred to as Super-peer host) for the network where the JCL Server or another Super-peer is installed.

The Super-peer server component receives requests from Hosts or from other Super-peers. As a server of a particular JCL cluster, it stores all the information that is under its domain. When the Super-peer host receives a storage request, it redirects to the Super-peer server component, which calculates function F , taking into consideration only the machines under its control. It is possible to locate an object in a multi-cluster environment with two calculations of F , the first being made in the User to know which Host to be selected and if such Host is a Super-peer, a second calculation of F is made to find where the object is really stored. The same idea is adopted by JCL to store key-value pairs of a map over multiple clusters.

When the Super-peer host receives a request to execute a task, it selects a Host from its domain to perform such a demand. Thus, the Super-peer adopts the same two-phase scheduling mechanism to find a Host. The second phase of the JCL task scheduling technique does not migrate tasks between networks administrated by different Super-peers, therefore the collaboration occurs only among Hosts of the same sub-network.

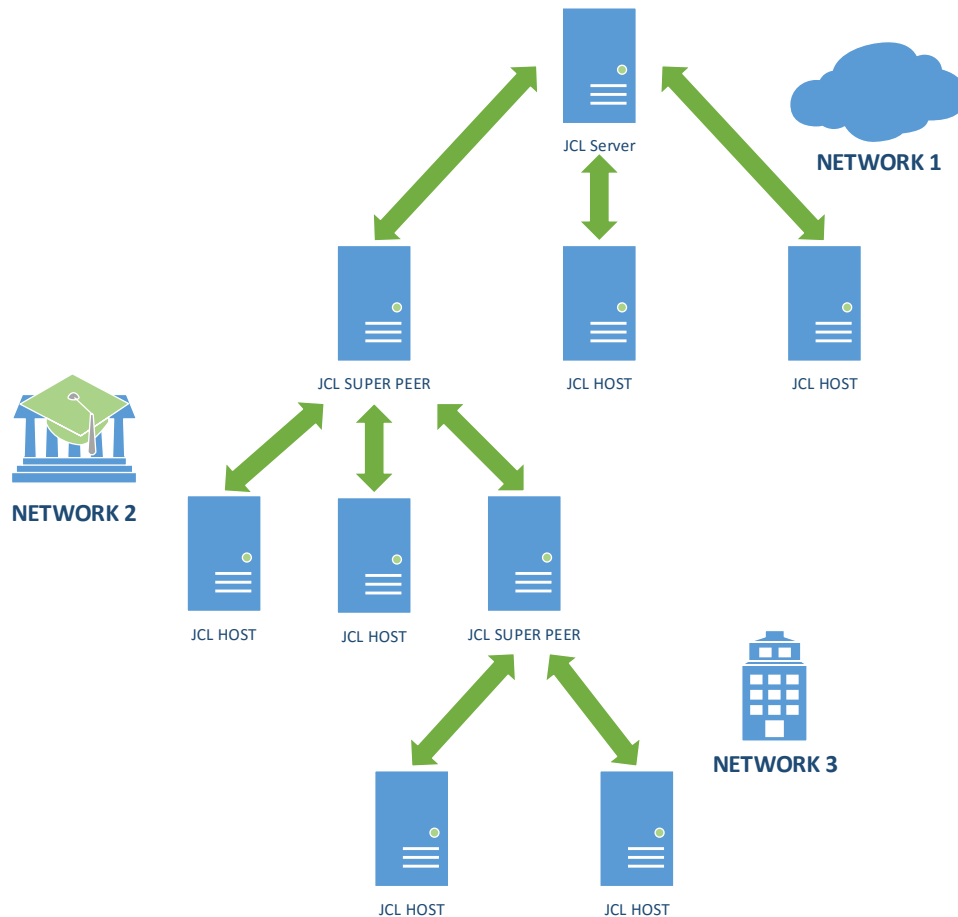


Figure 4.12: Super-peer topology

By adding Super-peers, the topology of the JCL cluster becomes hierarchical, i.e., it forms a tree data structure. The highest level, referred to as root node, represents a JCL Server component, having only one instance in the cluster. Below the root node we can add leaf nodes, i.e., JCL Host components, but also intermediary nodes, represented by JCL Super-peer components. The intermediary nodes branch out into new networks, so below these we can add leaf nodes, i.e., new JCL Host components (Figure 4.12).

One of the great challenges of the Super-peer is to provide the interconnection of different networks transparently, i.e., without any additional configuration. In order to overcome such a challenge, the Super-peer establishes a set, defined by the developer, of connections with the JCL Server or other JCL Super-peers. Such connections form tunnels of data between networks, and all messages that require be transmitted between different networks mandatorily need to adopt the tunnels.

The internal components of the Super-peer are managed by the ORB Super-peer (Figure 4.13), being the Super-peer host component responsible for maintaining the tunnels,

as well as to bypass messages to the worker threads. The Worker component calls the Super-peer server to distribute the storage or processing tasks among Hosts maintained by the Super-peer.

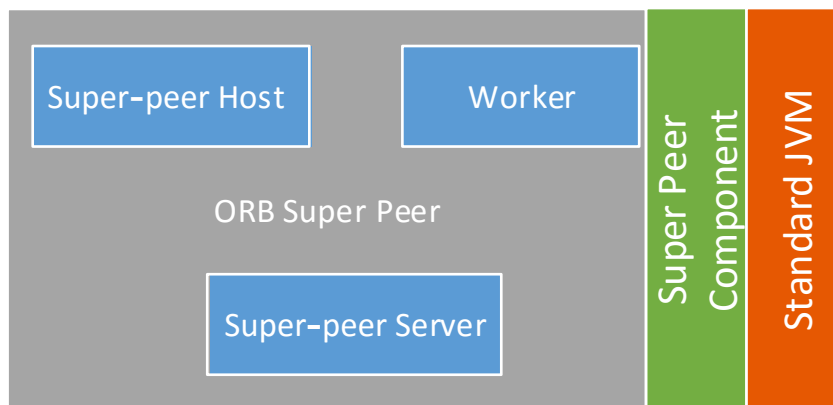


Figure 4.13: Super-peer component

4.8 Task cost

JCL has the possibility of collecting all the times involved in the execution of a specific task. To achieve this, the developer needs only the ticket obtained from a submitted task. After retrieving the result of the task, the developer may, via API, obtain such times with the *getTaskTimes(ticket)* method.

JCL returns a list containing six or eight time readings. The six time readings compose the timeline of a task that has not changed its Host due to the phase two of the scheduler, where the Hosts cooperate between each other. Each time reading is collected at the following time:

1st) It is collected immediately before the User component sends the task to the destination Host.

2nd) It is collected when the task reaches the Host.

3rd) It is collected when the execution of the task starts.

4th) It is collected when the execution of the task finishes.

5th) It is collected when the result leaves the Host.

6th) It is collected when the result arrives in the User component.

By the analysis of the timeline of six positions it is possible to calculate different times, such as network time, queue time, permanence time in the Host, among others. These times can be calculated as follows:

$$\textit{Total time} = \textit{timeline}(6) - \textit{timeline}(1) \quad (4.2)$$

$$\textit{Queue time} = \textit{timeline}(3) - \textit{timeline}(2) \quad (4.3)$$

$$\textit{Execution time} = \textit{timeline}(4) - \textit{timeline}(3) \quad (4.4)$$

$$\textit{Result retrieval time} = \textit{timeline}(5) - \textit{timeline}(4) \quad (4.5)$$

$$\textit{Time that a result remains in the Host} = \textit{timeline}(5) - \textit{timeline}(2) \quad (4.6)$$

$$\textit{Network time} = ((\textit{timeline}(6) - \textit{timeline}(1)) - (\textit{timeline}(5) - \textit{timeline}(2))) \quad (4.7)$$

The list that the JCL returns can contain eight times. This addition of two times to the timeline of the task is due to the collaborative scheduler, where a task can be transferred once to the Host that will process it. When there is task replacement, the times are collected as follows:

1st) It is collected immediately before the User component sends the task to the destination Host.

2nd) It is collected when the task reaches the first Host.

3rd) It is collected when the task leaves the first Host.

4th) It is collected when the task reaches the second Host.

5th) It is collected when the execution of the task starts.

6th) It is collected when the execution of the task finishes.

7th) It is collected when the result leaves the second Host.

8th) It is collected when when the result arrives in the User component.

The analysis of the timeline when there is a second Host is a little different, since the elapsed time in the first Host is added. In order to produce different times, the equations are:

$$\textit{Total time} = \textit{timeline}(8) - \textit{timeline}(1) \quad (4.8)$$

$$\textit{Queue time} = (\textit{timeline}(3) - \textit{timeline}(2)) + (\textit{timeline}(5) - \textit{timeline}(4)) \quad (4.9)$$

$$\textit{Execution time} = \textit{timeline}(6) - \textit{timeline}(5) \quad (4.10)$$

$$\textit{Result retrieval time} = \textit{timeline}(7) - \textit{timeline}(6) \quad (4.11)$$

$$\textit{Time that the task remains in the First Host} = \textit{timeline}(3) - \textit{timeline}(2) \quad (4.12)$$

$$\textit{Time that the task remains in the Second Host} = \textit{timeline}(7) - \textit{timeline}(4) \quad (4.13)$$

$$\textit{Network time} = ((\textit{timeline}(6) - \textit{timeline}(1)) - (\textit{timeline}(5) - \textit{timeline}(2))) \quad (4.14)$$

The overhead introduced by the task cost concept is about 320 microseconds, therefore it is worth using such a feature. These different and complementary times are fundamental to build reports of supervisory systems, adopted to delineate capacity planning of decentralized systems. A high standard deviation of the cluster queue time can indicate that the CPU clusters are not enough and new Hosts need be connected, for example. The inverse can guarantee energy saving.

These six or eight times can be adopted to evaluate which Host to submit a task, therefore new techniques for load balancing can be constructed. It is possible to build adjacent systems that monitor core systems, providing CPU and RAM on demand, as elasticity can be a requirement. It is important to reinforce that none of the middleware systems found in the literature implement this feature.

Chapter 5

Sorting Use Case

This chapter aims to evaluate JCL in terms of the implementation of fundamental computer science algorithms, such as sorting. The JCL BIG sorting application represents a solution with intensive communication, processing and I/O.

It is a sorting solution where data are partitioned to be sorted in every core of the JCL cluster, 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 and $+10^9$. 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 application is a simple and elegant sorting solution based on item frequencies. The frequency of each number of each input data partition is obtained locally by each Host thread and a chunk 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 the number of cluster threads (nct) create a constant C , so $C = nf/nct$. 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 stored in one Host thread must be sent to other threads in the 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 5.1). Lines 24, 26 and 28 of the main class illustrate how to instantiate JCL, obtain the 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 values.

```

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 5.1: Main class - how to generate pseudo-random numbers in JCL cluster.

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 5.1, i.e., they are inside the main class and they behave basically by splitting method calls over the cluster threads and then waiting for all computations to finish. 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 sorting schema with fair workload. Figure 5.2 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.

```
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 5.2: Main class - how to mount the global chunk schema to partition the cluster workload.

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 in phase one. The algorithm merely 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 the JCL

cluster to enable any JCL Host thread to access them transparently.

Figure 5.3 illustrates the JCL global variable concept, where Java object lifecycles are transparently managed by JCL over a cluster. The sorting class obtains a global JCL map labelled “h” (Figure 5.3). Each JCL map ranges from 0 to the 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 another JCL Host thread, i.e., each JCL Host thread has several chunks created from the remaining threads. Line 99 of Figure 5.3 represents a single entry in a global map “h”, where “id” represents the current JCL Host thread identification and the “final” variable represents the numbers/frequencies of such a chunk. Phase three of the sorting application 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.

```

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

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 DDR RAM 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 DDR RAM 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 same algorithm 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 large amounts of data. Basically, the new sorting version does not store the pseudo-random numbers in external memory. It combines 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 or over a cloud environment with a unique portable code. The small Raspberry Pi cluster sorted 60GB of data in 2,7 hours.

Chapter 6

Experimental Evaluation

The objective of this chapter is to evaluate the middleware in several distinct scenarios. In the first scenario the middleware is stressed and its throughput is measured when different JCL API methods are called. We measured in different execution rounds the methods: i) execute four type of task, void, sort, CPU bound and user typed argument, ii) instantiate a global variable, iii) instantiate a map, iv) iterate over a distributed map, v) put items in a map and vi) get a value from a map. JCL multi-computer version was adopted in the first experimental scenario.

In the second scenario, JCL multi-core speedup is compared with a Java thread implementation provided by Oracle. In the third scenario, JCL super-peer component overhead is discussed. In the fourth scenario we tested JCL executing an optimization solver, where the goal is to find promising input data for specific optimal results. In the fifth scenario, experiments with a non-deterministic solver for a real-world combinatorial problem were made to evaluate how JCL schedules non-deterministic tasks.

6.1 Throughput Experiments

This set of experiments were conducted with the Java Cá&Lá (JCL) multi-computer version. Initially, the JCL middleware was evaluated in a desktop cluster composed of 15 machines, where all machines were equipped with Intel(R) Core(TM) i5-2500 3.3GHz processors (4 physical cores) and 4GB of DDR3 1333Mhz RAM. The Operating System was Ubuntu 16.04 LTS 64 bits kernel 4.4 and all experiments fit in RAM memory. Each experiment was repeated five times. An average time was calculated. The middleware was evaluated in terms of throughput, i.e., the number of JCL operations processed 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 4.1, can be when both incremented global variable names and random names are adopted.

In the first round of experiments, we tested the JCL asynchronous remote method invocation (Figure 6.1). For each test we fixed the number of remote method invocations to ten thousand executions. The experiments were composed of four different methods: the first one is a method without argument (Figure 6.1 A); the second one is a method with an integer as an argument and the task is the generation and sorting of one million integers (Figure 6.1 B); the third method takes an array of strings and two integer values as arguments which are adopted to execute Levenshtein distance algorithm, Fibonacci series and prime numbers algorithms (Figure 6.1 C); and the fourth one is a void method with a book as argument, where a book is a user typed object composed of authors, editors, edition, pages and year attributes. In the book constructor, a list of objects is created to form the references (Figure 6.1 D). We measured the throughput per second of each cluster configuration, composed of 5, 10 and 15 machines.

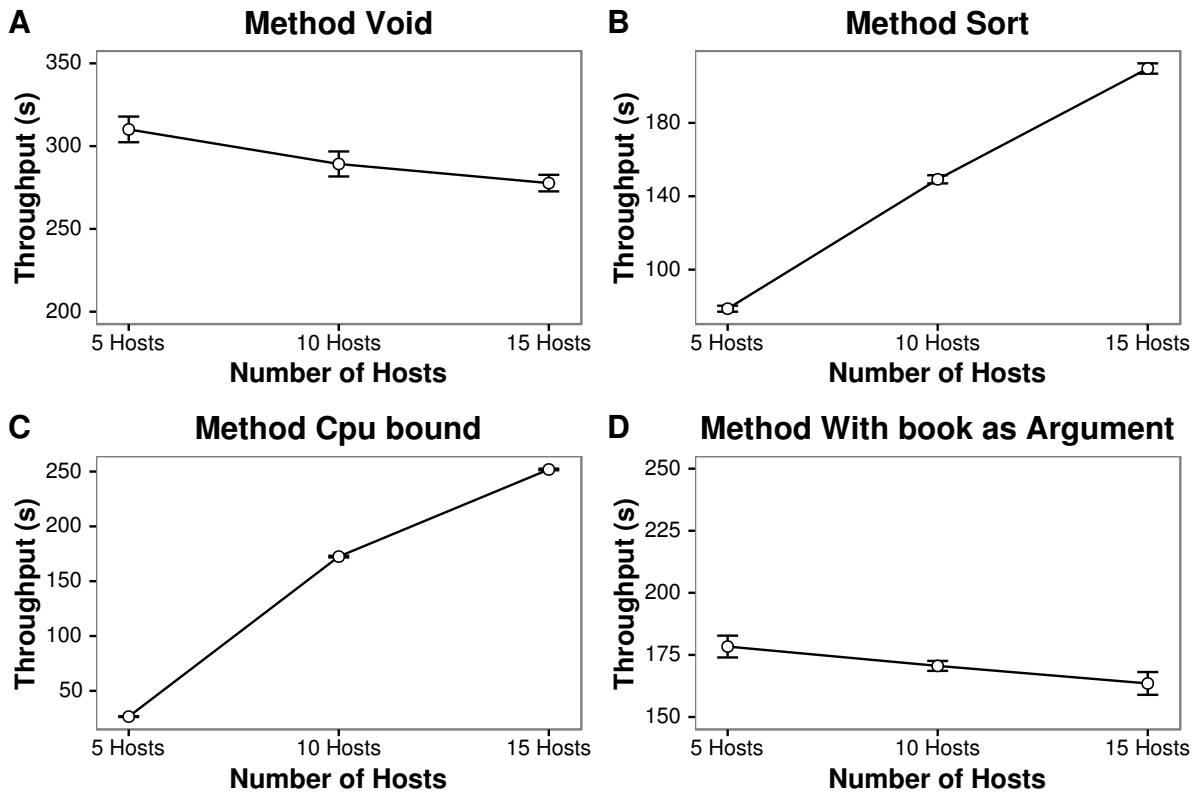


Figure 6.1: Task execution experiments

The results demonstrate that, for CPU bound tasks, JCL's throughput rises when cluster size increases (Figure 6.1 B and C). The first and fourth tests represent non CPU-bound executions, so it is clear that network overhead is greater than task processing

(Figure 6.1 A and D), indicating that these type of tasks must be submitted to the cluster in more coarse groups to increase task processing.

In the second round of experiments (Figure 6.2), we fixed the number of instantiated global variables to ten thousand instantiations. We tested the above-mentioned book class in four distinct ways. In the first, the instance of an object is created in the User and sent to the Host synchronously. In the second, the instance is created directly in the Host synchronously (Figure 6.2 A). In the third, the book object is created in the User and sent to the Host asynchronously and, finally, in the fourth experiment, the book object is created in the Host asynchronously (Figure 6.2 B).

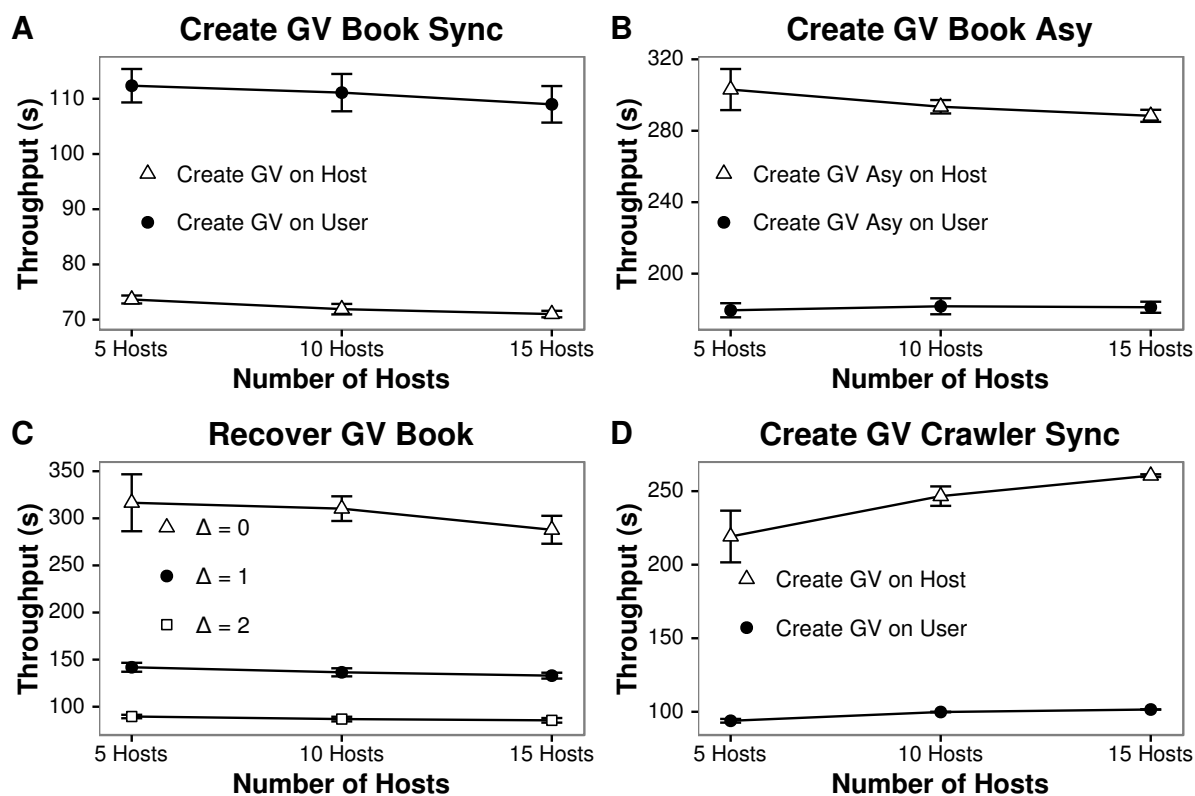


Figure 6.2: Global variable experiments

If we compare the synchronous forms to instantiate global variables, we can conclude that the alternative to instantiate in the User has greater throughput than the one instantiated in the Host. This behavior is due to the fact that when sending the variable to be instantiated in the Host, we must send both the arguments of the class constructor and the classes (including their dependencies) through the network, which ends up reducing the throughput. When we compare the asynchronous options, this behavior reverses itself, being the obtained throughput greater when we instantiate the variable in the Host. This behavior is warranted because the cost of creating the book object is not under the

responsibility of the User, but rather under the responsibility of the Host, which adopts a parallel solution.

In Figure 6.2 C, the *getValue(Objectkey)* method was used to recover a book variable previously instantiated in the User and sent to the cluster with different delta values that was explained in section 4.3. Although additional requests introduced by a delta different from zero are parallel, an overhead is introduced, so when the delta value duplicates, the throughput is reduced in 50%.

In Figure 6.2 D, the Crawler object, was substituted for the Book object simulating scan on 1000 web pages. The created object stores the visited pages, the pages to visit and the pages where a keyword was found. When we work with variables where the cost of sending the object via network is greater than the cost of sending the constructor parameters, the strategy of creating the variables in the Host has a greater throughput because the copy in the User and its transfer to the Host is avoided (Figure 6.2 D).

In the third round of experiments (Figure 6.3), the same number of instantiated global variables of the second round was used. We tested smaller objects like an integer value (Figure 6.3 A), a double value (figure 6.3 B), a string with 10 characters (figure 6.3 C) and an array of 20270 bytes (figure 6.3 D). As the cluster enlarges, the number of connections and other issues also become time-consuming, thus a reduction in throughput should be expected. Another important observation is the synchronous behavior of the JCL shared memory services, which is another bottleneck when the cluster becomes larger. JCL asynchronous calls are, on average, three times faster than their synchronous counterpart. The positive aspect to the cluster become bigger is the fact that the storage capacity increases, so there is a fundamental reason to grow up.

In the fourth round of experiments, we tested the JCL distributed map implementation. We inserted 10 thousand books in a map. *Put*, *PutAll* (Figure 6.4 A), *Get* and iterator methods were evaluated (Figure 6.4 B). The *Put* and *PutAll* methods have a huge throughput difference, as Figure 6.4 A illustrates. We assume that the optimization introduced by JCL is responsible for such a big difference, *i.e.*, buffering key-value pairs with identical $f + d$ results is fundamental in reducing network communications. Another huge throughput difference occurs when a value is retrieved individually versus when it is obtained via iterator (Figure 6.4 B). The iterator method optimizes key-value retrieval by requesting chunks of data stored in a Host. The *Get* method requests one key-value pair per time, so it is more efficient to adopt the iterator method instead of get method to traverse a distributed JCL map.

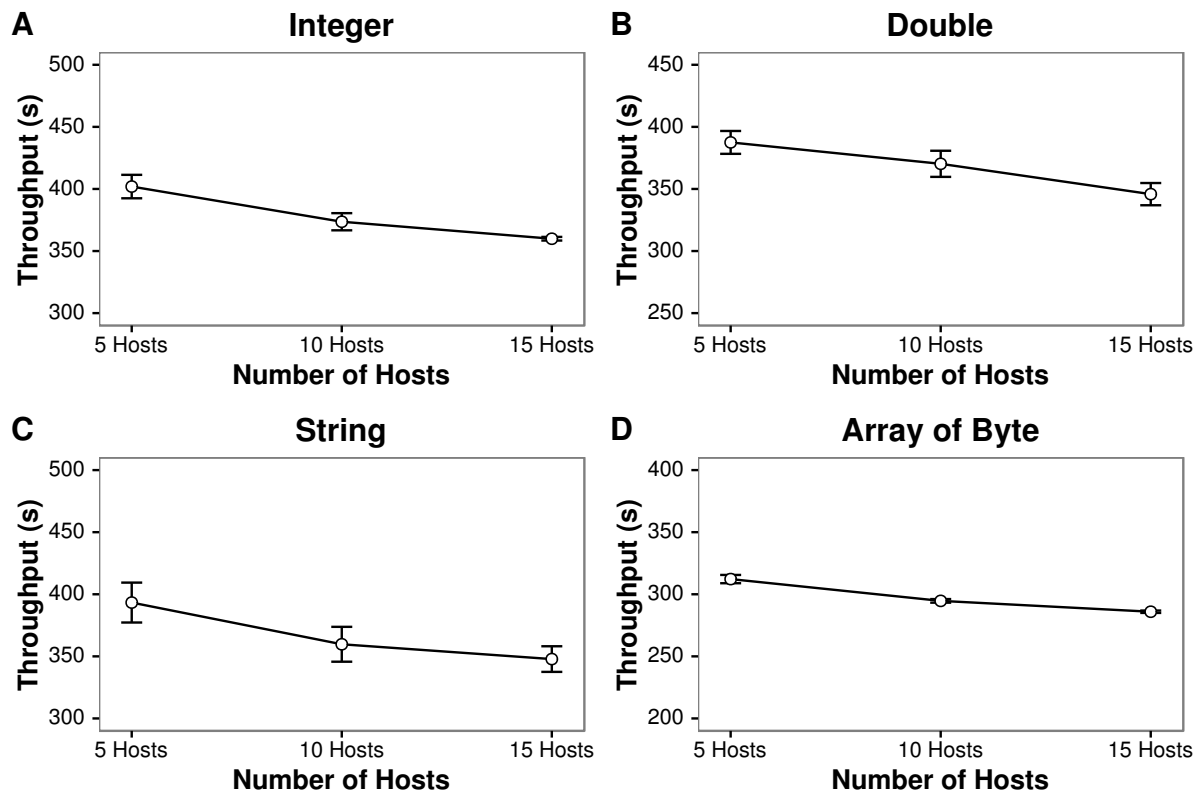


Figure 6.3: Global variable experiments

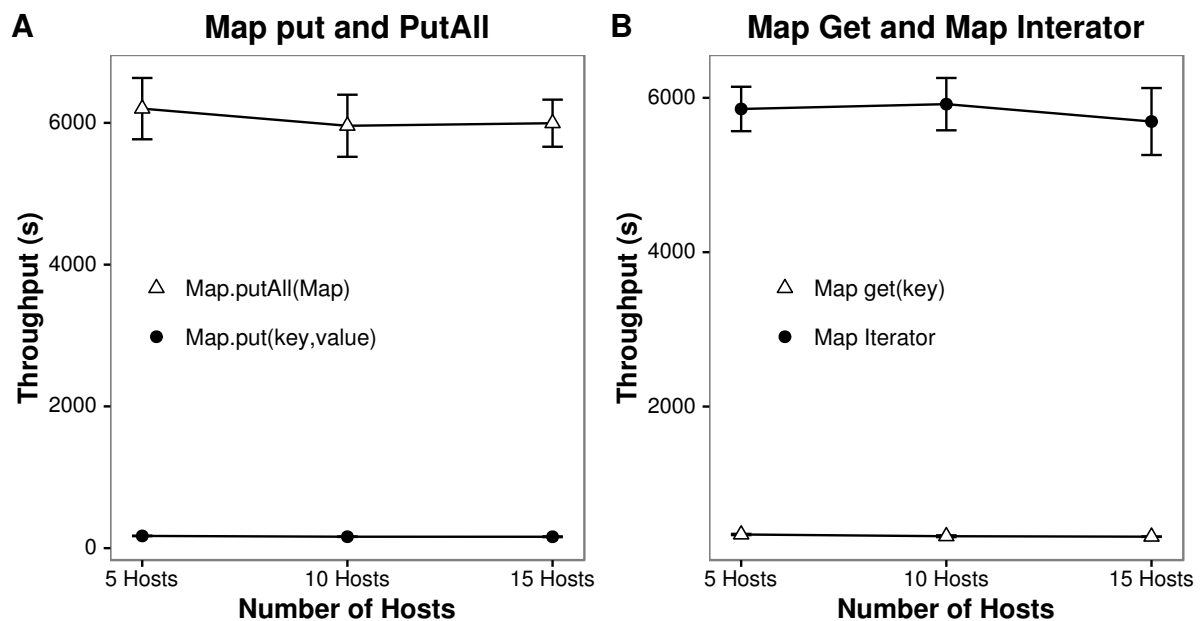


Figure 6.4: JCLMap experiments

In the fifth round of experiments, we evaluated the uniformity of function F presented in Equation 4.1 with different deltas value d . The experiment instantiated 40 thousand variable names. We tested different prefix variable names and also auto-increment suffixes, i.e., variable names like “ p_i ” and “ p_{ij} ”, where p is a prefix and i and j are auto-

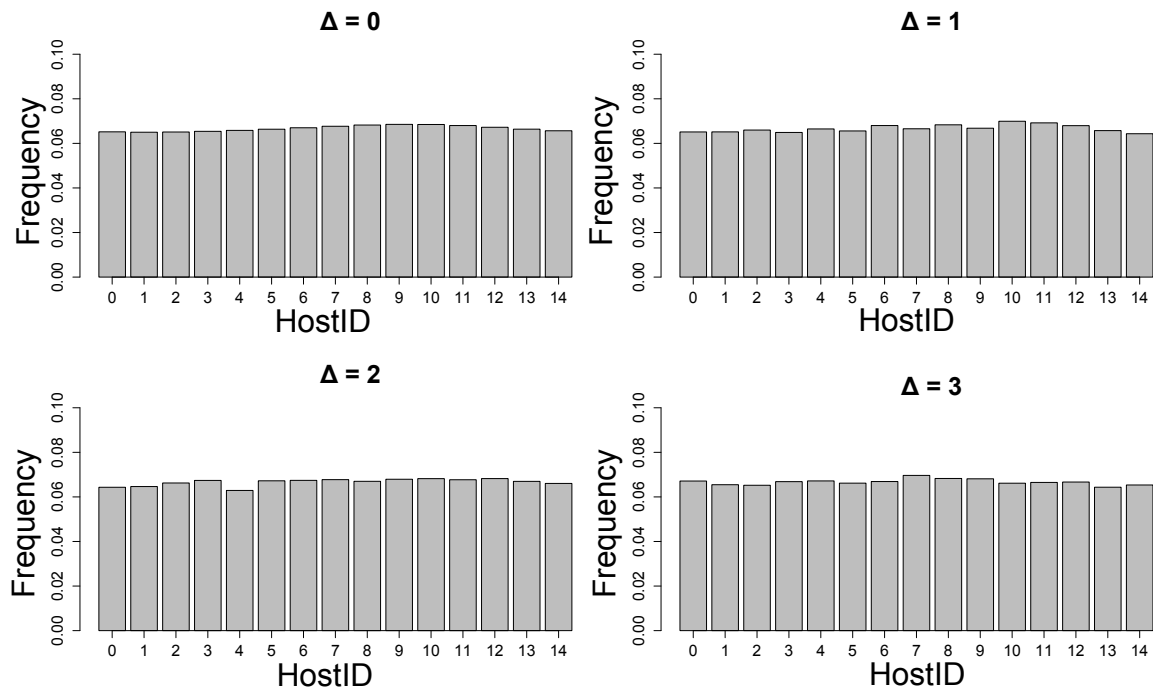


Figure 6.5: Variable names with autoincrement

increment values. We also tested function $F + d$ distribution for an arbitrary bag of words of the Christian Bible text to verify how the JCL data partition performs. The results are illustrated in Figure 6.5 and 6.6, where Δ is the delta size of the equation $F + d$ used to scheduler the global variables. Usually, JCL achieves an almost uniform distribution using delta between zero and two.

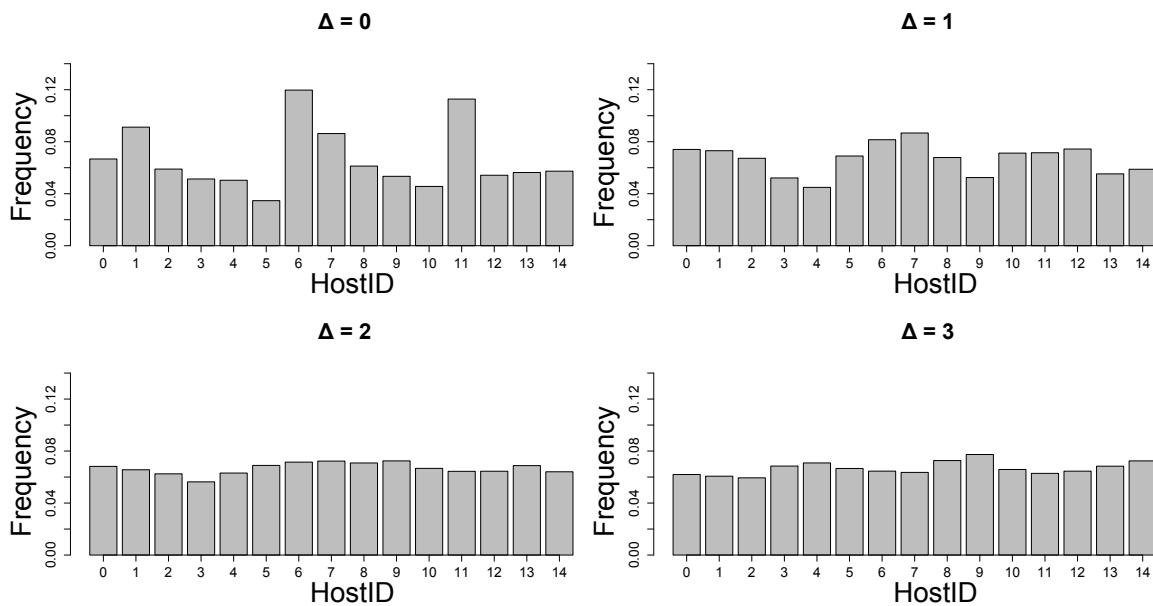


Figure 6.6: Bag of words

The result of the bag of arbitrary words becomes more uniform as delta increases, so even when the developer decides to adopt arbitrary variable key name in the code, JCL can achieve an almost fair data partition. We also tested the JCL overhead when the content of a variable is retrieved using delta equal to zero, one and two. Data partition uniformity is reduced as delta tends to zero, as can be seen in Figure 6.6. Several project management decisions will opt to reduce uniformity of the variables in each JCL Host and increases access throughput, but the opposite must be considered for storage reasons.

6.2 Multi-core speedup experiments

We evaluated the 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 thread speedups and the results demonstrated similar speedups. JCL achieved a speedup of 5.61 and Oracle Java threads a speedup of 5.77.

6.3 Super-peer component overhead

In this section, we evaluated the overhead of the Super-peer component with two different applications. We tested JCL with communication intensive sorting tasks and COIN-OR branch-and-cut (CBC) CPU bound tasks. Both applications were tested in the same environment with 20 machines, where all machines were equipped with Intel(R) Core(TM) i5-2500, 3.3GHz processors (4 physical cores) and 4GB of RAM DDR3 1333Mhz. The operating system was Ubuntu 14.04.1 LTS 64 bits kernel 3.13.0-39-generic. In the CBC experiments, 77 tasks were randomly selected among the 4221 non-deterministic tasks presented in section 6.4. In the sorting experiments, 50000 tasks were submitted. The sorting tasks are explained in Section 6.1.

We tested both applications in two scenarios. First, we configured the 20 machines in two different networks, one with a Server component and another with both Server and Super-peer components. The number of Hosts in the Super-peer network increased by 5 per round of tests, varying from 0 to 20. In contrast, the number of Hosts in the Server network decreased by 5 per round of tests, varying from 20 to 0. Table 6.1 illustrates

both networks in terms of number of Hosts.

Table 6.1: Number of Hosts

Round \ Network	1	2	3	4	5
Super-peer Network 192.168.0.0/24	0	5	10	15	20
Sever Network 10.10.10.0/24	20	15	10	5	0

Each one of the rounds of tests was repeated five times and the total average time of the two applications were calculated. The results illustrate that when the tasks are CPU bound, there is no significant overhead when the number of Hosts in each network alternates (Figure 6.7 B). When the tasks are communication intensive and not CPU bound, there is an overhead introduced by the Super-peer to manage different numbers of Hosts, as Figure 6.7 A illustrates. If we consider as a baseline the total time of the first round, where all the 20 Hosts are managed by the Server, the second round was 74% slower than the first, the third was 66% slower than the second, the fourth was 54% slower than the third and, finally, the fifth was 63% slower than the fourth.

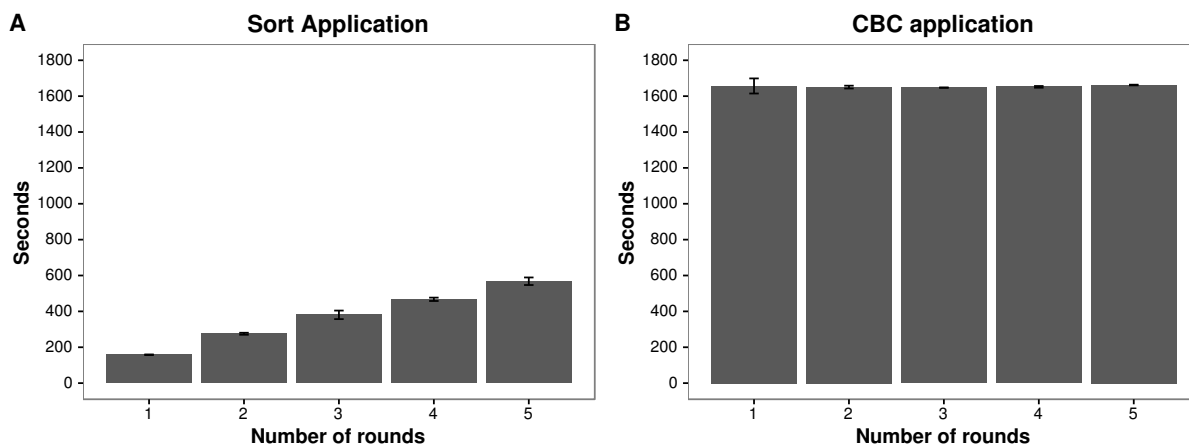


Figure 6.7: Super-peer overhead in managing Hosts

In general, there was an increase of 10% with the introduction of each Host. The largest overhead occurred in the second round of the sorting application, because in the second round there was the creation of a new Super-peer. In the other rounds the same Super-peer is adopted, having only an increase in the number of Hosts. In the case of the sorting application, the overhead is unacceptable, therefore one alternative is to group the 50000 tasks in, for example, 5000 tasks. Such an alternative has less communication and

consequently less overhead. CBC and sorting experimental results represent two extreme scenarios, reinforcing that JCL can introduce significant overhead if tasks are not CPU bound.

In the second scenario, we evaluated both applications as new Super-peers were being introduced, varying from 0 up to 4. Five rounds were executed, performing five network configurations with a different number of Super-peers (zero Super-peer, one, two, three and four Super-peers, respectively). In the fifth round, for instance, there are five networks and four Super-peers, where each Super-peer component manages five Hosts. The Server network had no Host in such a round, as Table 6.2 illustrates.

Table 6.2: Super-peer overhead in different multi-cluster topologies

Network \ Round	1	2	3	4	5
Sever Network 10.10.10.0/24	20	15	10	5	0
Super-peer Network 1 192.168.0.0/24	0	5	5	5	5
Super-peer Network 2 192.168.1.0/24	0	0	5	5	5
Super-peer Network 3 192.168.2.0/24	0	0	0	5	5
Super-peer Network 4 192.168.3.0/24	0	0	0	0	5

Each round was repeated five times, similar to the first scenario explained previously. The results illustrate, as in the first scenario, that the overhead increases as the communication exceeds tasks processing costs (Figure 6.8). In order to calculate the overhead of the Super-peer, we consider round 0 as a baseline, i.e., we measured the remaining rounds and compared them with round zero. The following values were obtained: 2nd round 74% slower; 3rd round 154% slower; 4th round 236% slower and 5th round 294% slower. The impact of adding new Super-peers is superior to that of adding new Hosts, as expected. In the case of the sorting application, the overhead is unacceptable, therefore more coarse grained sorting tasks should be modeled.

The Super-peers create tunnels with the Server, therefore each communication between User and Host components must be intercepted by Server and redirected by a Super-peer component. The Super-peer enables JCL to communicate with sub-networks with invalid IPs, a common scenario to several clusters in academic institutions, business

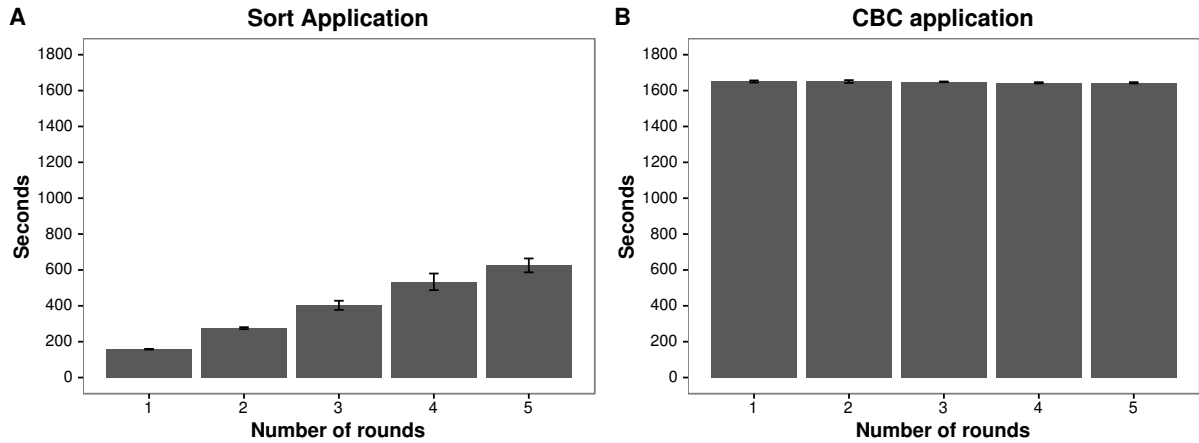


Figure 6.8: Super-peer overhead topology 2

companies and in our houses. However, the overhead can be enormous or unrealistic if we do not take care of the granularity of the tasks. New configurations for the Super-peer must be added to the JCL, because the Super-peer does not always have only invalid IPs. There are many cases where the Super-peer can be installed in a machine with two network cards, therefore the Super-peer is accessible to both Server and Users components, i.e., it has Internet access, as well as manages Hosts from a second data network, composed of machines with invalid IPs. Other future improvements are presented in chapter 7.

6.4 CBC solver experiments

In this section, the objective of the experiment is to evaluate JCL scheduling for deterministic and non-deterministic optimization problems. The solver, named COIN-OR branch-and-cut (CBC) Forrest and Lougee-Heimer (2005), is executed several times to calibrate input parameters for specific optimal results. The main goal is to find the best parameters that find the optimal solutions in shorter runtime or opening fewer branches. Note that, these CBC executions to calibrate parameters become a new combinatorial optimization problem.

The CBC solver is an open-source tool for many combinatorial optimization problems modeled as mixed integer linear programming solutions. CBC was written in C++, so these experiments also show that JCL can execute other language compiled modules. The CBC method consists of a combination of a cutting plane method with a branch-and-bound algorithm Mitchell (2002), thus adopted for solving a huge number of integer programming problems Mitchell (2002). JCL managed all tasks executions, i.e., thousands

of CBC executions to find the best parameters for a selected input.

The experiments were divided into two rounds. In both rounds, the developed application was the same: to evaluate 21 sets of parameters with each one of the 201 instances, generating at the end 4221 tasks (21×201). Some tasks can take more time than an acceptable solution for a specific computational resource. In order to solve such a limitation, a maximum execution limit was added to each task and this limit is calculated by the Benchmark ITC3-Linux-x86-64¹ in each Host. The benchmark collects some machine configurations to stipulate a limit. A time limit of one hour, normalized with the standard CPU used in the benchmark, was stipulated.

The test environment used was composed of 20 machines, where all machines were equipped with Intel(R) Core(TM) i5-2500, 3.3GHz processors (4 physical cores) and 4GB of RAM DDR3 1333Mhz. The Operating System was Ubuntu 14.04.1 LTS 64 bits, kernel 3.13.0-39-generic, and all experiments could fit in RAM memory. An average time was calculated from five executions, i.e., we submitted 4221 tasks five times and adopted a confidence interval of 95%. The middleware was evaluated in terms of runtime in seconds.

In the first round of experiments the 4221 tasks were submitted five times to the cluster with phase 2 of the JCL scheduler enabled and five times with phase 2 disabled. Different running times for each task were collected using the *getTaskTimes(ticket)* method explained in section 4.8. A deterministic execution of CBC was repeated five times. Figures 6.9 A-D illustrate total time, queue time, execution time and network time results. The total time (Figure 6.9 A) decreased by 50% when the collaborative scheduler behavior is turned on. Figure 6.9 B illustrates that the queue time was slightly greater in tests 1, 2, 3 and 5 and significantly greater in test 4 when the scheduler was disabled. This result occurred because with scheduler phase 2 disabled the tasks mandatorily get more queue time while waiting for CPU. The running execution times measured in the five rounds are similar, since the tasks are deterministic (Figure 6.9 C). The network time (Figure 6.9 D) demonstrates that the overhead introduced by the data exchange between Hosts was minimal.

Before illustrating the different times of the non-deterministic CBC executions, we present topological differences of each execution, i.e, how many non-deterministic tasks are different from deterministic ones in terms of branches opened to solve the same problems. In Figure 6.10, the X axis has seven time classes. We classify the tasks with total time between 0 and 600 seconds, between 601 and 1200 seconds, between 1201 and 1800

¹Benchmark_ITC3-Linux-x86-64 is available for download at <https://www.utwente.nl/ctit/hstt/itc2011/benchmark>

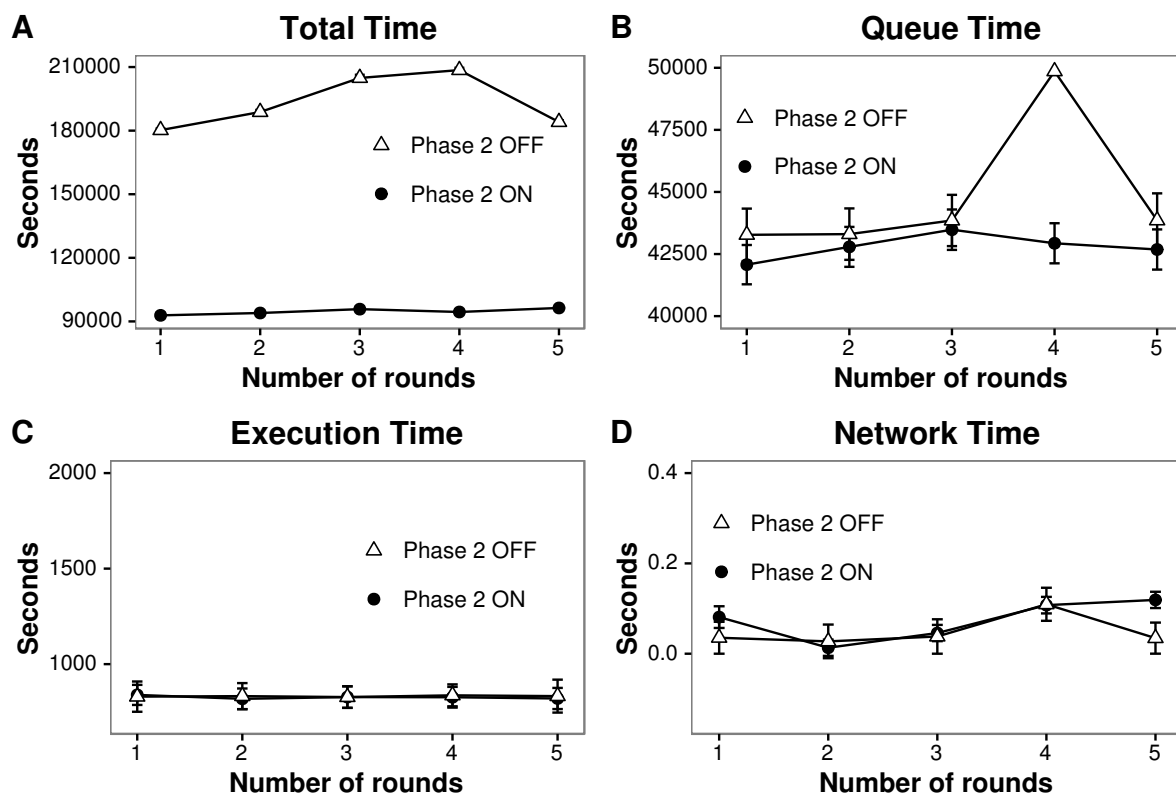


Figure 6.9: Task execution experiments

seconds, between 1801 and 2400 seconds, between 2401 and 3000 seconds, between 3001 and 3600 seconds and the tasks between 3601 and 4200 seconds. The Y axis illustrates the percentage of tasks in a specific time class that are different from their deterministic counterparts.

Close to 70% of the tasks adopted different tree branches in CBC when they were compared to the same tasks in the deterministic executions. There is about a 94% – 99% difference in the results classified between 601 and 1200 seconds. In the other 5 classes with times greater than 1200 seconds the difference varied between 81% – 98% of difference. Thus, the hypothesis presented in Fischetti and Monaci (2014) that is possible to create a non-deterministic behavior in a branch-and-cut algorithm is confirmed by Figure 6.10.

In terms of running time, the 4221 tasks have differences. Figure 6.11 illustrates in the X axis the percentage of difference in terms of running time, so there are 58% – 64% of the 4221 tasks with 0 to 20% difference in running time and 18% to 22% of the tasks with 20% to 40% difference in running time. The percentage of tasks with high running time differences, i.e., above 50%, varies from 4% to 8% of the 4221 tasks.

In Figure 6.12, the total time was reduced by around 2 to 3.5 times when phase two of the scheduler was enable. The behavior when phase two is off is more aleatory, being

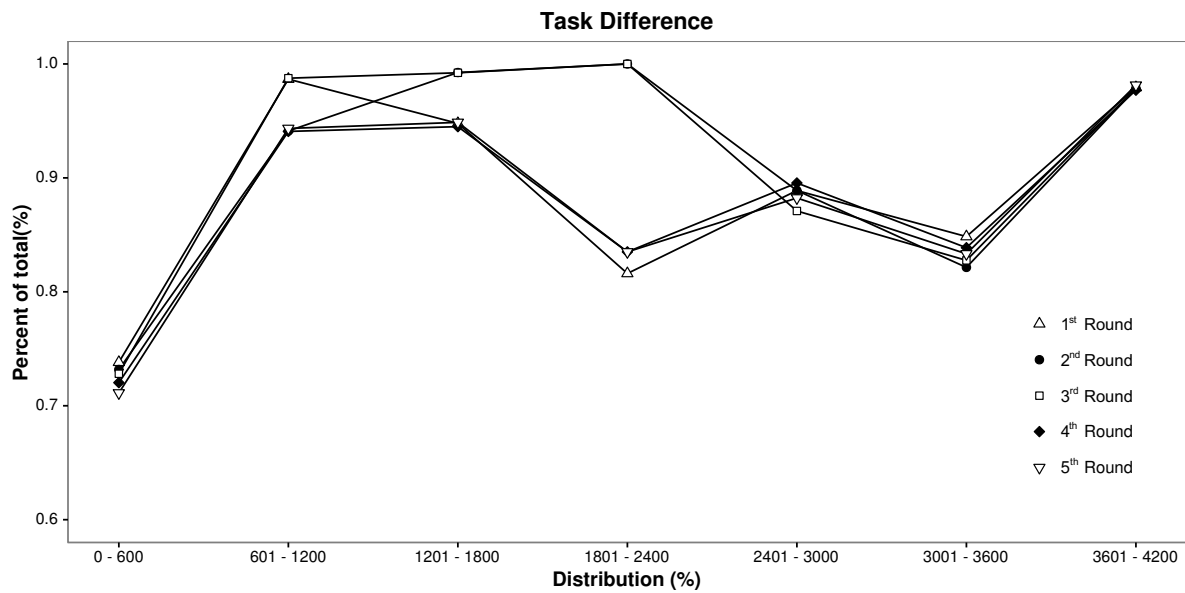


Figure 6.10: Tasks difference in terms of opened branches

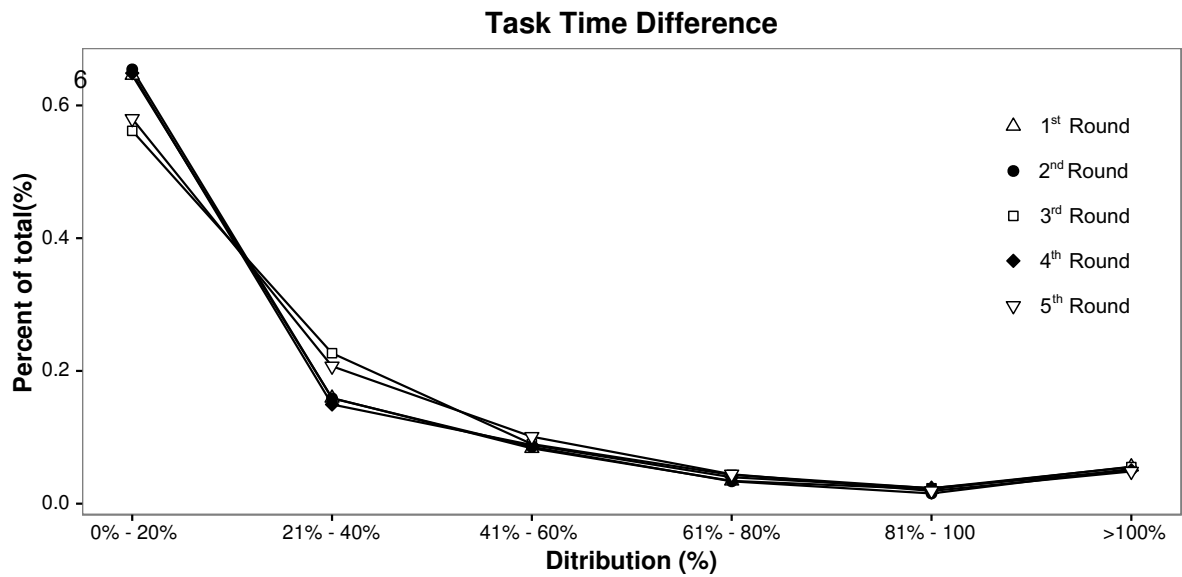


Figure 6.11: Task runtimes distribution

higher in the fifth round of Figure 6.12 A. The queue time is also aleatory, so when queue time increases phase two of the scheduler works more to balance the system. The number of task that move to other Host varies from 250 to 350 tasks, so we can argue that few scheduling interventions (4 – 8% of the tasks) accelerates the application by 3.5 times. Since there are few tasks with huge running time differences, the average execution time becomes similar (Figure 6.12 C). The network time is aleatory, depending on the number of scheduler interventions to replace tasks.

The sequential time to execute 4221 tasks was calculated by summing up the total

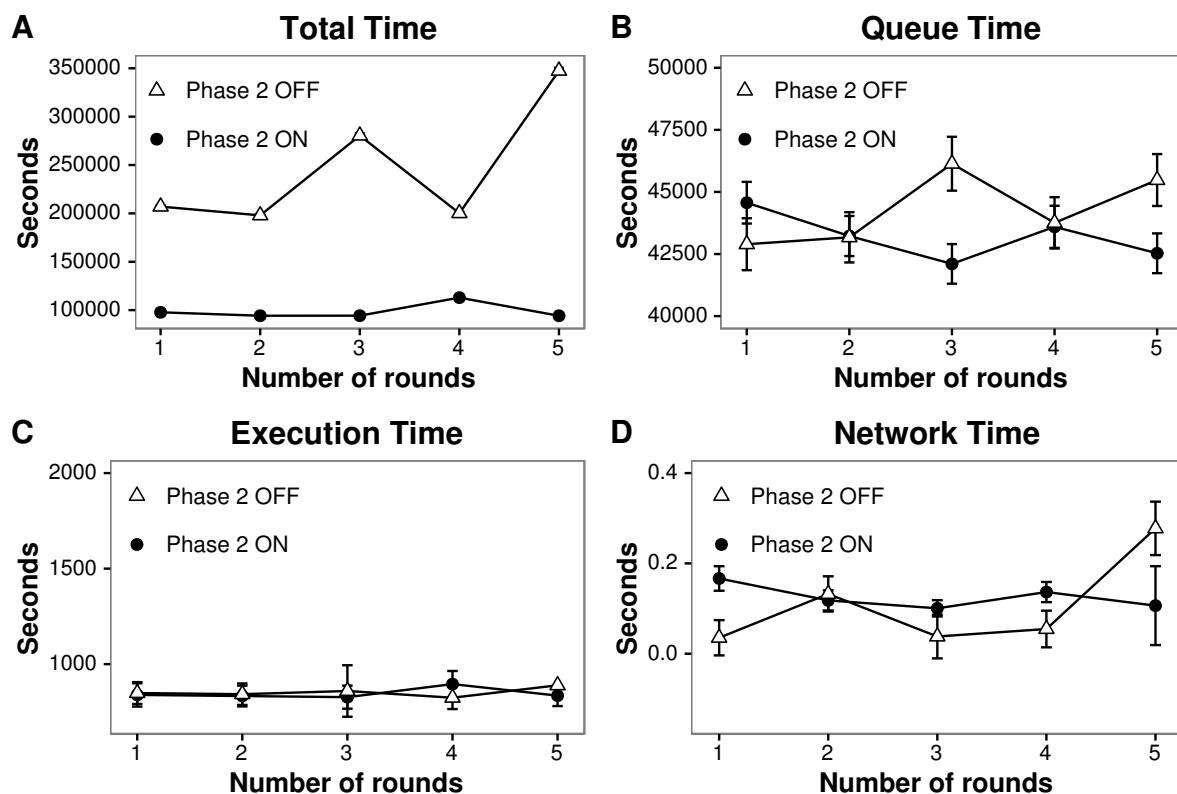


Figure 6.12: Non-deterministic task execution experiments

times. The result is 82 days. The average execution time of the same set of tasks with JCL was 25 hours with the phase 2 scheduler enabled, characterizing reduction of computational resources at about $78\times$ with adoption of JCL. Thus, compatible with the 80 cores of the cluster used in the experiments in order to accelerate finding best inputs for specific instances of a combinatorial problem.

6.5 Experiments with a solution for the problem of minimizing the number of tool switches

In the previous section, the workload difference was small in terms of percentage of the total tasks that were scheduled by JCL. Less than 10% of the CBC tasks were replaced in the cluster. To increase task replacements, we evaluated an optimization algorithm to solve a combinatorial real world problem with application in the industrial production context.

The Minimization of Tool Switches Problem (MTSP) is an \mathcal{NP} -Hard problem Crama et al. (1994), meaning there is no known efficient algorithm that solves it. According to Crama et al. (1994), the problem is stated as follows: “A batch of jobs has to be

successively processed on a single flexible machine. Each job requires a subset of tools, which have to be placed in the tool magazine of the machine before the job can be processed. The tool magazine has a limited capacity, and, in general, the number of tools needed to produce all the jobs exceeds this capacity. Hence, it is sometimes necessary to change tools between two jobs in a sequence. The problem is then to determine a job sequence and an associated sequence of loadings for the tool magazine, such that the total number of tool switches is minimized.”

The method chosen for this experiment was recently proposed by Paiva and Carvalho (2016). Currently, this is the state-of-the-art method for solving the MTSP. This method generates an initial solution using a new constructive heuristic based on graph search. After the initial solution is obtained, it is improved by an implementation of the traditional Iterated Local Search Lourenço et al. (2003) metaheuristic, “which consists of repeatedly applying local search methods and randomly modifying a solution until it reaches a stopping condition” Paiva and Carvalho (2016). The stopping condition adopted was 200 iterations of the metaheuristic method. Additionally, the local search methods implemented were the new *1-blocks grouping* Paiva and Carvalho (2016) and the classical *2-swap*, which exchanges the positions of two elements of a permutation. Given n elements in a solution, the *2-swap* algorithm may require all possible pairs of elements to exchange positions, thus, asymptotically it may perform up to $n!$ operations. Because it is not feasible to perform this number of operations, the Iterated Local Search performs a limited search of 30% of the search space (i.e., up to $0.3 \times n$ operations), where each pair of elements considered is randomly selected.

The tests were conducted for two purposes: i) to accelerate the Iterated Local Search runs for multiple MTSP instances, similarly to the experiments conducted with the CBC; and ii) to parallelize the *2-swap* algorithm runs in order to improve the accuracy of the MTSP solutions. However, in this work the focus is not on discussing improvements of accuracy, but rather the quality of the JCL scheduler.

In the experiments, 240 instances were used, 160 proposed by Crama et al. (1994) and 80 proposed by Catanzaro et al. (2015). The JCL, “Pacu” version, was used to distribute the sequential algorithm, written in C++, in the cluster of 20 machines and 80 cores, identical to the CBC environment. The Iterated Local Search has a parallel local search phase (the *2-swap* algorithm), i.e., it is possible to explore multiple different ranges of 30% of the search space simultaneously. For this, the multicore “Lambari” version of JCL is used, this being responsible for managing from 4 to 16 *2-swap* tasks concurrently. The

parallelism of the *2-swap* illustrates a scenario where a C++ application executes the JCL, i. e., the opposite of JCL scheduling the Iterated Local Search over the cluster. Thus, the JCL Iterated Local Search application confirms the simplicity of interoperating JCL with C++ in two complementary ways.

The test environment used to run the Iterated Local Search was composed of 20 machines, where all machines were equipped with Intel(R) Core(TM) i5-2500, 3.3GHz processors (4 physical cores) and 4GB of RAM DDR3 1333Mhz, summing up 80 cores in the cluster. The Operating System was an Ubuntu 14.04.1 LTS 64 bits kernel 3.13.0-39-generic and all experiments could fit in RAM memory. An average time was calculated from five executions and a confidence interval of 95% was adopted.

The results demonstrate that around 20% of 240 tasks submitted to JCL cluster were replaced, what implies an increase of more than 100% if compared to the CBC experiments, which replaced less than 10% of all tasks. Other results are illustrated in Figure 6.13. Each value in Figure 6.13 represents the average times of an execution round, i.e., the execution of all 240 instances and their parallel *2-swap* runs, summing up more than 240×4 tasks for each execution round. In terms of total time, when the two-phase scheduler is active the running time reduction was about 20% to 30% on average if compared with JCL without the two-phase scheduler. If compared to the CBC, the Iterated Local Search applied to the MTSP achieves fewer benefits from parallelization, since its tasks are less CPU bound, so the HPC benefits tend to be reduced.

In Figures 6.13 B and D the times are similar, thus sometimes more than 20% of task replacement does not affect both network and queue times. The explanation is because the Iterated Local Search tasks are small and their initial arguments are also small, so network transfer drawbacks are avoided. The non-determinism is illustrated by Figure 6.13 C, where sometimes running times with the scheduler active is worse than with the scheduler inactive in JCL. This behavior also produces a reduction in total time, but sometimes tasks are more CPU bound in some experiment rounds due to the non-determinism, so the scheduler impact increases when task times are worse, because the total time indicates 20% to 30% of improvement.

A second alternative to parallelize the Iterated Local Search applied to the MTSP was evaluated. One of the largest instance of the 240 available was selected and the *2-swap* method was parallelized via the JCL “Pacu” multi-computer version over the 80 cores. The *2-swap* method run concurrently by 1 to 160 tasks in each iteration. The scheduler is always active in these experiments. Each value in Figures 6.14 and 6.15 represents the

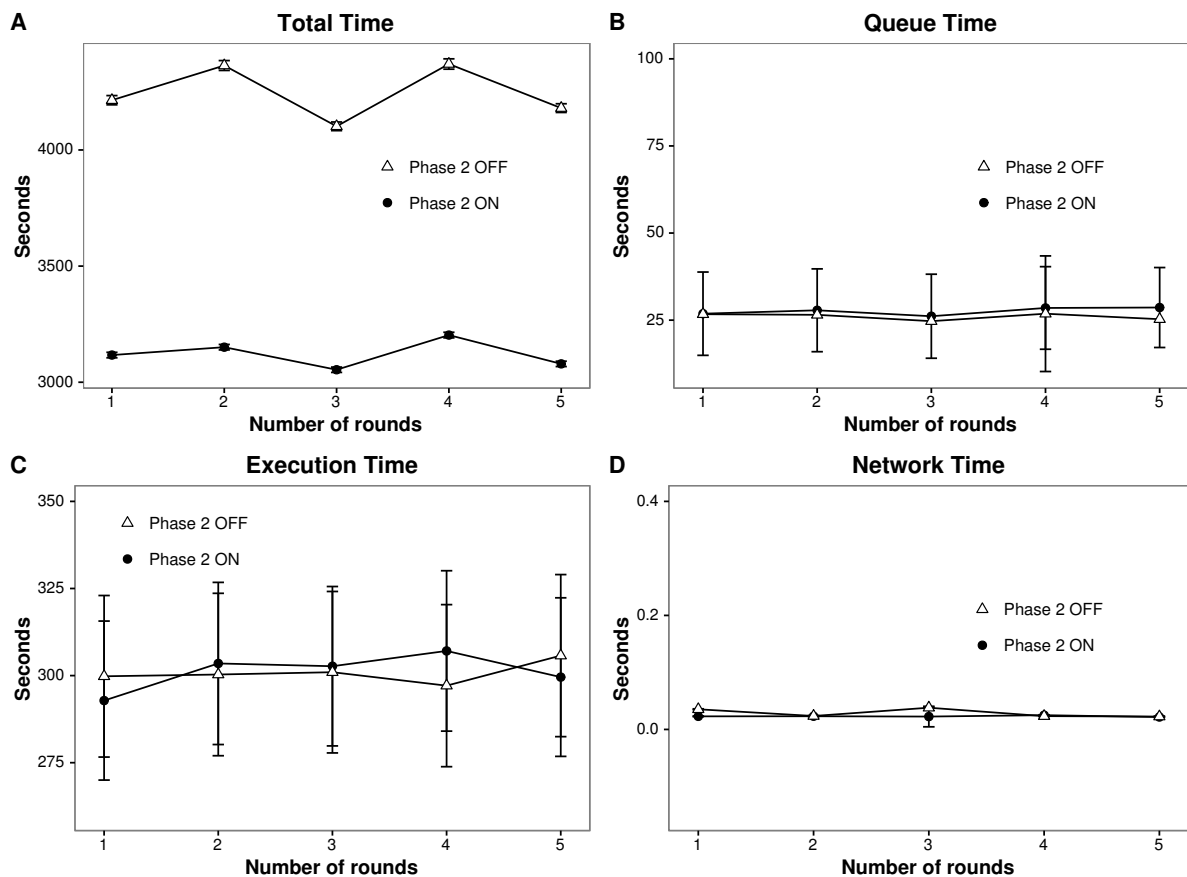


Figure 6.13: Iterated Local Search applied to the MTSP 240 instances experiment.

average times of 1 to 160 tasks in each execution round.

In Figure 6.14, JCL was tested with 1, 32 and 64 tasks to perform the *2-swap* method in parallel, and in Figure 6.15, JCL was tested managing 96, 128 and 160 parallel tasks, therefore, more tasks than the 80 cores available in the cluster.

As shown in Figure 6.14 A, the parallelization introduces nearly 80% of overhead, as the difference of 1 task from, for instance, 32 tasks illustrates. However, if we consider that the parallel runs perform 32 or 64 more *2-swap* method calls, the improvement is evident. Figures 6.14 C and 6.15 D reinforce the non-deterministic behavior of the Iterated Local Search applied to the MTSP. In Figure 6.15 A, the total times did not increase, what can be explained by scheduler opportune interventions. On average, 40% of overhead is introduced in total times if we compare rounds with 32 or 64 tasks to rounds with 96 or 128 tasks, and 80% if we compared with 160 tasks. Nonetheless, the JCL scheduler was effective, considering that *2-swap* runs increased by more than 300%. Figure 6.15 B, illustrates a queue time outlier when there are 160 tasks to be scheduled, then we can conclude that less than 128 tasks in parallel did not cause retention in the cluster,

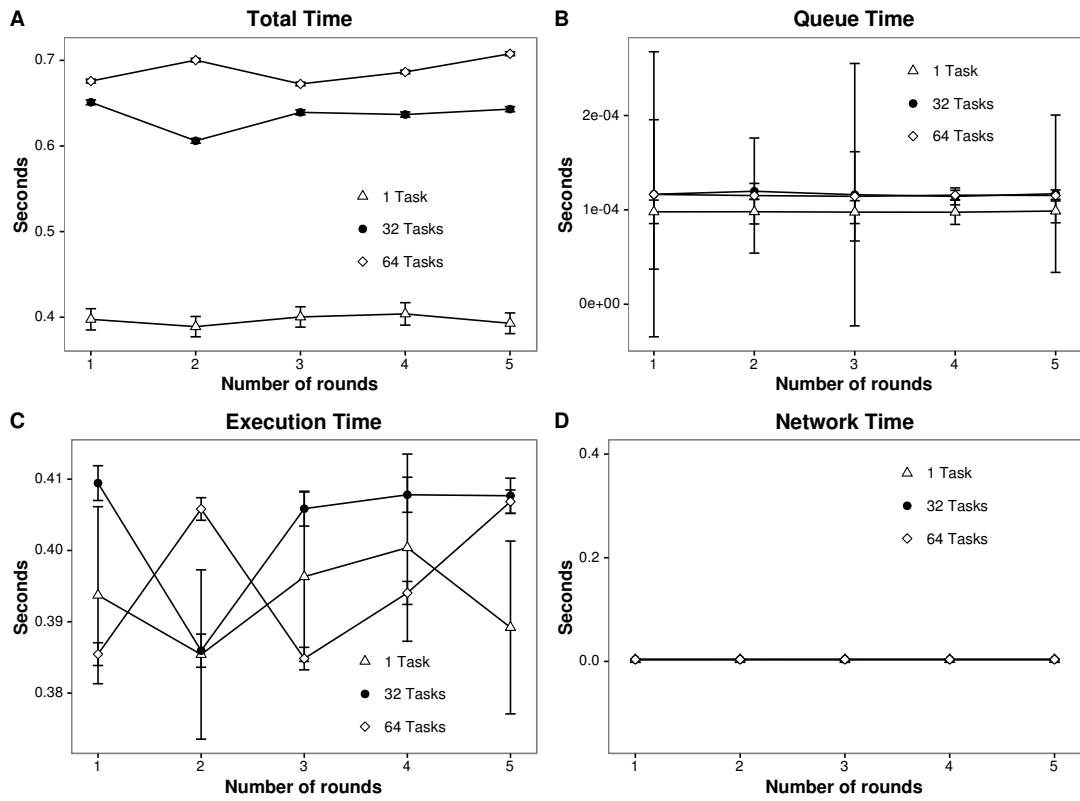


Figure 6.14: Iterated Local Search applied to the MTSP run rounds (1 - 64 Tasks)

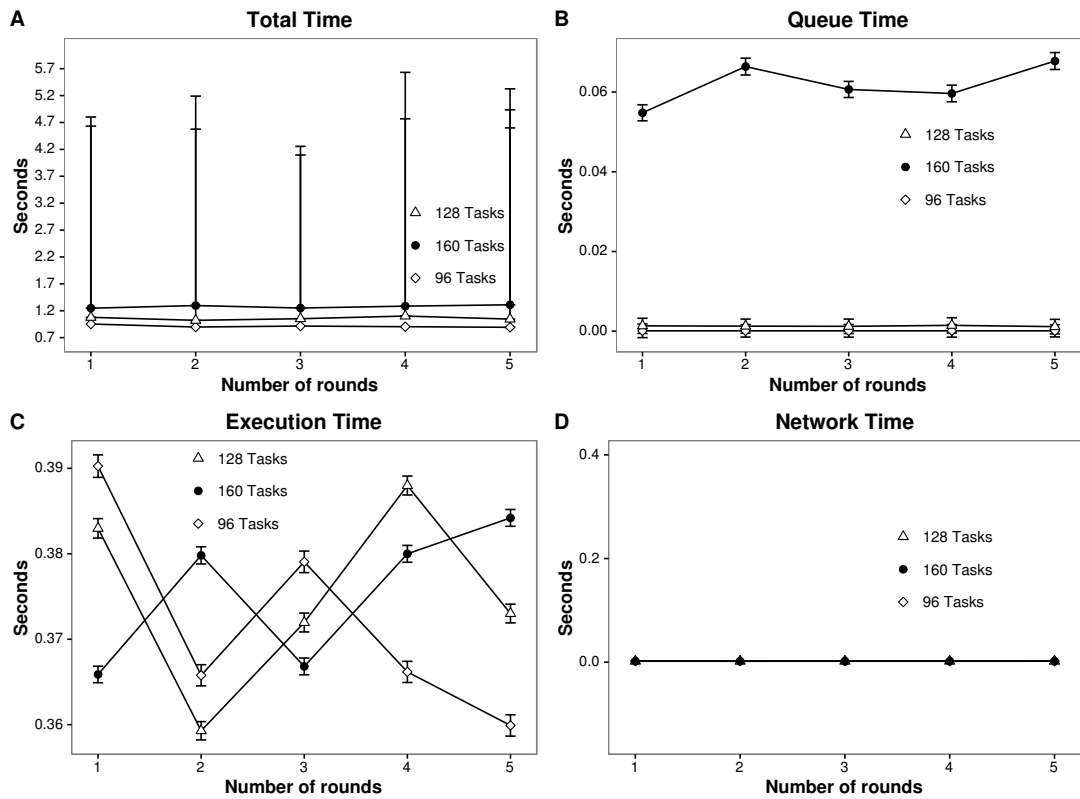


Figure 6.15: Iterated Local Search applied to the MTSP run rounds (96 - 160 Tasks)

indicating an important threshold for capacity planning. Around 20% of the 160 tasks were moved to a second Host.

Chapter 7

Conclusion

In this work, we present a novel middleware for capacity planning, storage and processing that can be operated in different platforms, including cloud platforms and boards with Linux support, such as Raspberry Pi, Galileo, Cubieboard and many others. The middleware solution is able to invoke tasks asynchronously, manage Java objects lifecycle over a cluster of JVMs and retrieve some useful times from a task execution. It is designed for multi-core, multi-computer and hybrid computer architectures. Developers 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. Reflection capabilities of Java enable JCL to separate distribution from business logic, enabling both existing sequential code executions over many high performance computer architectures with zero refactorings and multiple distribution strategies for a single sequential algorithm according to a hardware specification.

JCL also implements the concept of Super-peer and therefore a JCL Server can manage, for instance, a cluster of JCL Hosts with invalid IPs. JCL implements a native distributed map that is a sub-type of Java Map interface, so with low refactorings the developer can change from local to distributed storage. All storage and processing are efficiently managed by JCL, i.e., from a storage perspective a hash function is adopted to find a JCL Host to store key-value pairs or Java objects. From a processing perspective, a collaborative two-phase scheduler achieves fair load balancing for deterministic and non-deterministic tasks.

Experiments demonstrated that JCL is a promising solution for general purpose computing. JCL was tested for building an elegant distributed sorting implementation that outperforms a similar version in MPI. A second battery of tests was conducted with the open source Mixed-Integer programming named CBC. The results with CBC demon-

strated that the JCL scheduler introduced few tasks replacement, but it reduced the runtime by 50% to 150%. If we consider a sequential execution of CBC, JCL reduced the execution of 4221 tasks from 82 days to 25 hours. Throughput experiments demonstrated that JCL scales well when the cluster enlarges. Those experiments reaffirmed that processing must compensate data network delays. The last set of experiments evaluated a solution for MTSP problem. The results demonstrated that JCL scales well even when there are many non-deterministic tasks to be replaced.

There are many improvements to be done. JCL should be fault tolerant in storage and processing. A new strategy for JCL Super-peer components to interconnect different networks must be implemented. The current method is based on two network routers, one for the Server network and another for Super-peer networks. This concept causes significant network overhead when the application is communication intensive, but in contrast it transparently enables the connection of two networks, so even networks with invalid IPs can be integrated to JCL. A new alternative, where a Super-peer has two network interface card (NIC), is an important improvement to the current Super-peer solution. The Super-peer must avoid message decoding in the router component, i.e., the router can re-transmit all messages directly to the destinations without opening them.

GPU execution abstractions, where location and copies are transparent to developers, are very useful to JCL. New options for the first phase of JCL scheduler should be incorporated to better allocate tasks initially. IoT features, such as interoperability, context awareness, privacy and security must be part of JCL. The Cross-platform Host component, including platforms without JVM, with JVMs that are not compatible with JSR 901 (Java Language Specification) or platforms without Operating Systems, are mandatory to IoT. A supervisor application to manage the JCL cluster resources (maps, global variables, tasks, sensors, devices, etc.) must be implemented.

Bibliography

- Brian Amedro, Denis Caromel, Fabrice Huet, Vladimir Bodnartchouk, Christian Delbé, and Guillermo L Taboada. Proactive: using a java middleware for hpc design, implementation and benchmarks. *International journal of Computers and Communications*, 3(3):49–57, 2009.
- Leurent Baduel, Françoise Baude, and Denis Caromel. Object-oriented spmd. In *Cluster Computing and the Grid, 2005. CCGrid 2005. IEEE International Symposium on*, volume 2, pages 824–831. IEEE, 2005.
- David H Bailey, Eric Barszcz, John T Barton, David S Browning, Russell L Carter, Leonardo Dagum, Rod A Fatoohi, Paul O Frederickson, Thomas A Lasinski, Rob S Schreiber, et al. The nas parallel benchmarks. *International Journal of High Performance Computing Applications*, 5(3):63–73, 1991.
- Jasma Balasangameshwara and Nedunchezhian Raju. A decentralized recent neighbour load balancing algorithm for computational grid. *Int. J. of ACM Jordan*, 1(3):128–133, 2010.
- Jasma Balasangameshwara and Nedunchezhian Raju. A hybrid policy for fault tolerant load balancing in grid computing environments. *Journal of Network and computer Applications*, 35(1):412–422, 2012.
- Jasma Balasangameshwara and Nedunchezhian Raju. Performance-driven load balancing with a primary-backup approach for computational grids with low communication cost and replication cost. *IEEE Transactions on Computers*, 62(5):990–1003, 2013.
- Donald Becker, Phil Merkey, et al. The beowulf project. *Online at: <http://www.beowulf.org>*, 2002.
- Carlos Boneti, Roberto Gioiosa, Francisco J. Cazorla, and Mateo Valero. A dynamic scheduler for balancing hpc applications. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, SC '08, pages 41:1–41:12,

- Piscataway, NJ, USA, 2008. IEEE Press. ISBN 978-1-4244-2835-9. URL <http://dl.acm.org/citation.cfm?id=1413370.1413412>.
- E. McAfee Brynjolfsson. Big data: The management revolution. *Harvard Business Review*, 90(10):60–66, 2012. URL <http://hbr.org/2012/10/big-data-the-management-revolution/ar>.
- Apache Cassandra. The apache software foundation. *The Apache Cassandra project*, 2013.
- Daniele Catanzaro, Luis Gouveia, and Martine Labbé. Improved integer linear programming formulations for the job sequencing and tool switching problem. *European Journal of Operational Research*, 244(3):766–777, 2015.
- Kristina Chodorow. *MongoDB: the definitive guide*. " O'Reilly Media, Inc.", 2013.
- L. Cohen. *Java Parallel Processing Framework*, 2015. Available from: <http://www.jpff.org/>. [15 Dezember 2015].
- Yves Crama, Antoon W. J. Kolen, Alwin G. Oerlemans, and Frits C. R. Spijksma. Minimizing the number of tool switches on a flexible machine. *International Journal of Flexible Manufacturing Systems*, 6(1):33–54, 1994. ISSN 1572-9370. doi: 10.1007/BF01324874. URL <http://dx.doi.org/10.1007/BF01324874>.
- Pierangelo Di Sanzo, Francesco Quaglia, Bruno Ciciani, Alessandro Pellegrini, Diego Didona, Paolo Romano, Roberto Palmieri, and Sebastiano Peluso. A flexible framework for accurate simulation of cloud in-memory data stores. *arXiv preprint arXiv:1411.7910*, 2014.
- Sean Egan. *Open Source Messaging Application Development: Building and Extending Gaim*. Apress, 2005.
- Matteo Fischetti and Michele Monaci. Exploiting erraticism in search. *Operations Research*, 62(1):114–122, 2014.
- John Forrest and Robin Lougee-Heimer. Cbc user guide. *INFORMS Tutorials in Operations Research*, pages 257–277, 2005.
- Message P Forum. *Mpi: A message-passing interface standard*. Technical report, Knoxville, TN, USA, 1994.

- Edgar Gabriel, Graham E Fagg, George Bosilca, Thara Angskun, Jack J Dongarra, Jeffrey M Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, et al. Open mpi: Goals, concept, and design of a next generation mpi implementation. In *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*, pages 97–104. Springer, 2004.
- A. Gates and D. Dai. *Programming Pig*. O'Reilly Media, Incorporated, 2016. ISBN 9781491937099.
- Stéphane Genaud and Choopan Rattanapoka. P2p-mpi: A peer-to-peer framework for robust execution of message passing parallel programs on grids. *Journal of Grid Computing*, 5(1):27–42, 2007.
- Lars George. *HBase: the definitive guide*. " O'Reilly Media, Inc.", 2011.
- Sukumar Ghosh. *Distributed systems: an algorithmic approach*. CRC press, 2014.
- Aniruddha Gokhale, Krishnakumar Balasubramanian, Arvind S Krishna, Jaiganesh Balasubramanian, George Edwards, Gan Deng, Emre Turkay, Jeffrey Parsons, and Douglas C Schmidt. Model driven middleware: A new paradigm for developing distributed real-time and embedded systems. *Science of Computer programming*, 73(1):39–58, 2008.
- GridGain Systems. Gridgain 3.0 – high performance cloud computing whitepaper, 2011. Technical Report.
- Jiawei Han, Micheline Kamber, and Jian Pei. *Data Mining: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 3rd edition, 2011. ISBN 0123814790, 9780123814791.
- M Henning, M Spruiell, et al. Distributed programming with ice; zeroc. *Inc.: Jupiter, FL, USA*, 2013.
- Michi Henning and Mark Spruiell. Distributed programming with ice reading, 2006.
- Ian Hickson. The websocket api. *W3C Working Draft WD-websockets-20110929, September*, 2011.
- Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. 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, 2011.

- R Intel. galileo gen 2 development board, 2016.
- Alan Kaminsky. Parallel java: A unified api for shared memory and cluster parallel programming in 100% java. In *Proceedings of IEEE International Parallel and Distributed Processing Symposium (IPDPS 2007)*, pages 1–8, 2007.
- Alan Kaminsky. *Big CPU, Big Data: Solving the World’s Toughest Computational Problems with Parallel Computing*. Unpublished manuscript, 2015. Retrieved from <http://www.cs.rit.edu/~ark/bcbd>.
- K.I. Karantasis and E.D. Polychronopoulos. 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, 2011.
- Konstantinos I. Karantasis and Eleftherios D. Polychronopoulos. Pleiad: A cross-environment middleware providing efficient multithreading on clusters. In *Proceedings of ACM Conference on Computing Frontiers (CF 2009)*, pages 109–116, 2009.
- Dawid Kurzyniec, Tomasz Wrzosek, Vaidy Sunderam, and Aleksander Slominski. Rmix: A multiprotocol rmi framework for java. In *Parallel and Distributed Processing Symposium, 2003. Proceedings. International*, pages 6–pp. IEEE, 2003.
- Helena R Lourenço, Olivier C Martin, and Thomas Stützle. Iterated local search. In *Handbook of metaheuristics*, pages 320–353. Springer, 2003.
- Eng Keong Lua, Jon Crowcroft, Marcelo Pias, Ritu Sharma, and Sharon Lim. A survey and comparison of peer-to-peer overlay network schemes. *Communications Surveys & Tutorials, IEEE*, 7(2):72–93, 2005.
- Francesco Marchioni and Manik Surtani. *Infinispan data grid platform*. Packt Publishing Ltd, 2012.
- John E Mitchell. Branch-and-cut algorithms for combinatorial optimization problems. *Handbook of applied optimization*, pages 65–77, 2002.
- Yoshitomo Murata, Tsutomu Inaba, Hiroyuki Takizawa, and Hiroaki Kobayashi. A distributed and cooperative load balancing mechanism for large-scale p2p systems. In *Applications and the Internet Workshops, 2006. SAINT Workshops 2006. International Symposium on*, pages 4–pp. IEEE, 2006.

- Amy L. Murphy, Gian Pietro Picco, and Gruia-Catalin Roman. Lime: A coordination model and middleware supporting mobility of hosts and agents. *ACM Trans. Softw. Eng. Methodol.*, 15(3):279–328, July 2006. ISSN 1049-331X. doi: 10.1145/1151695.1151698. URL <http://doi.acm.org/10.1145/1151695.1151698>.
- Christian Nester, Michael Philippsen, and Bernhard Haumacher. A more efficient rmi for java. In *Proceedings of the ACM 1999 conference on Java Grande*, pages 152–159. ACM, 1999.
- Nvidia Corporation. Compute unified device architecture programming guide, 2008.
- Gustavo Silva Paiva and Marco Antonio Moreira Carvalho. Um método para planejamento da produção em sistemas de manufatura flexível. In *Simpósio Brasileiro de Pesquisa Operacional*, 2016.
- Deepak Kumar Patel, Devashree Tripathy, and CR Tripathy. Survey of load balancing techniques for grid. *Journal of Network and Computer Applications*, 65:103–119, 2016.
- Charith Perera, Chi Harold Liu, Srimal Jayawardena, and Min Chen. A survey on internet of things from industrial market perspective. *Access, IEEE*, 2:1660–1679, 2014.
- Michael Philippsen, Bernhard Haumacher, and Christian Nester. More efficient serialization and rmi for java. *To appear in: Concurrency: Practice & Experience*, 11, 1999.
- Esmond Pitt and Kathy McNiff. *Java.Rmi: The Remote Method Invocation Guide*. Addison-Wesley Longman Publishing Co., Inc., 2001. ISBN 0201700433.
- William Pugh and Jaime Spacco. Mpjava: High-performance message passing in java using java. nio. In *International Workshop on Languages and Compilers for Parallel Computing*, pages 323–339. Springer, 2003.
- Mowafaq SalemAlzboon, Suki Arif, M Mahmuddin, and Omar Dakkak. Peer to peer resource discovery mechanisms in grid computing: A critical review.
- Olliver M Schinagl. *Getting Started with Cubieboard*. Packt Publishing Ltd, 2014.
- Aleksandar Seovic, Mark Falco, and Patrick Peralta. *Oracle Coherence 3.5*. Packt Publishing Ltd, 2010.
- Aamir Shafi, Bryan Carpenter, and Mark Baker. Nested parallelism for multi-core {HPC} systems using java. *Journal of Parallel and Distributed Computing*, 69(6):532

- 545, 2009. ISSN 0743-7315. doi: <http://dx.doi.org/10.1016/j.jpdc.2009.02.006>. URL <http://www.sciencedirect.com/science/article/pii/S0743731509000252>.
- Guillermo L Taboada, Juan Touriño, and Ramón Doallo. 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, 2009.
- Guillermo L Taboada, Juan Touriño, and Ramón Doallo. F-mpj: scalable java message-passing communications on parallel systems. *The Journal of Supercomputing*, 60(1): 117–140, 2012.
- Guillermo L Taboada, Sabela Ramos, Roberto R Expósito, Juan Touriño, and Ramón Doallo. Java in the high performance computing arena: Research, practice and experience. *Science of Computer Programming*, 78(5):425–444, 2013.
- Muhammad Adnan Tariq, Boris Koldehofe, Sukanya Bhowmik, and Kurt Rothermel. Pleroma: a sdn-based high performance publish/subscribe middleware. In *Proceedings of the 15th International Middleware Conference*, pages 217–228. ACM, 2014.
- Wendell Figueiredo Taveira, Marco Tulio de Oliveira Valente, Mariza Andrade da Silva Bigonha, and Roberto da Silva Bigonha. Asynchronous remote method invocation in java. *Journal of Universal Computer Science*, 9(8):761–775, 2003.
- GridGain Team. Introducing the gridgain in-memory data fabric. Technical report, 2016.
- ScyllaDB Team. *ScyllaDB*, 2015. Available from: <http://www.scylladb.com/>. [15 Dezember 2015].
- Vernon Turner, John F Gantz, David Reinsel, and Stephen Minton. The digital universe of opportunities: Rich data and the increasing value of the internet of things. *International Data Corporation, White Paper, IDC_1672*, 2014.
- Eben Upton and Gareth Halfacree. *Raspberry Pi user guide*. John Wiley & Sons, 2014.
- Peter Veentjer. *Mastering Hazelcast*. Hazelcast, 2013.
- Scott M. Walker, Alan Dearle, Stuart J. Norcross, Graham N. C. Kirby, and Andrew McCarthy. Rafda: A policy-aware middleware supporting the flexible separation of application logic from distribution. Technical report, University of St Andrews, 2003. Technical Report CS/06/2.

- Richard T Watson, Donald Wynn, and Marie-Claude Boudreau. Jboss: The evolution of professional open source software. *MIS Quarterly Executive*, 4(3):329–341, 2005.
- Jing Xiong, Jianliang Wang, and Jianliang Xu. Research of distributed parallel information retrieval based on jppf. In *2010 International Conference of Information Science and Management Engineering*, pages 109–111. IEEE, 2010.
- Beverly Yang and Hector Garcia-Molina. Designing a super-peer network. In *Data Engineering, 2003. Proceedings. 19th International Conference on*, pages 49–60. IEEE, 2003.
- Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, pages 10–10, 2010.
- Qi Zhang, Lu Cheng, and Raouf Boutaba. Cloud computing: state-of-the-art and research challenges. *Journal of internet services and applications*, 1(1):7–18, 2010.
- Wenzhang Zhu, Cho-Li Wang, and F.C.M. Lau. Jessica2: a distributed java virtual machine with transparent thread migration support. In *Proceedings of IEEE International Conference on Cluster Computing (Cluster 2002)*, pages 381–388, 2002.