Dipartimento di Informatica e
Scienze dell'Informazione

Java frameworks for high-level distributed scientific programming

by
Marco Ferrante

Theses Series **DISI-TH-2010-07**

DISI, Università di Genova
Via Dodecaneso, 35 – 16146 Genova, Italy
http://www.disi.unige.it/

Universita degli Studi di Genova

Dipartimento di Informatica e Scienze dell'Informazione

Dottorato di Ricerca in Informatica

Ph.D. Thesis in Computer Science

Java frameworks for high-level distributed scientific programming

by

Marco Ferrante

February, 2010

Dottorato di Ricerca in Informatica
Dipartimento di Informatica e Scienze dell'Informazione
Universita degli Studi di Genova

DISI, Univiversità di Genova
via Dodecaneso 35
I-16146 Genova, Italy
http://www.disi.unige.it/

Ph.D. Thesis in Computer Science (S.S.D. INF/01)

Submitted by Marco Ferrante
DISI, Università di Genova
ferrante@disi.unige.it

Date of submission: February 2009

Title: Java frameworks for high-level distributed scientific programming

Advisor: GIOVANNI CHIOLA
DISI, Università di Genova
chiola@disi.unige.it

Ext. Reviewers:
MICHAEL R. BERTHOLD
Department of Computer and Information Science
Universität Konstanz
berthold@ieee.org

MAURO MIGLIARDI
Faculty of Statistics
Università di Padova
mauro.migliardi@dei.unipd.it

Abstract

Computing is progressively shifting to handle larger and larger collections of data. It is not uncommon that databases in the domains of health, biology, genomics, physics, astronomy, and engineering size in a range from gigabytes to petabytes. Moreover, the modern statistical approaches to knowledge discovery in raw data, usually named *machine learning* or *data mining*, are intrinsically computational intensive. As a result, more and more power and storage capabilities are needed.

The usual way to obtain high computing performances is to aggregate several off-the-shelf CPUs in *ad-hoc* networks of dedicated commodity PCs (*clusters*) or exploiting idle resources of non-dedicated workstations (*Desktop Grid*). Obviously, distributed computing requires more than connecting several CPUs and several hard drives by wires. The real challenge is coordinating tasks execution and data storage on such systems.

The research on grid architectures consistently focused on designing middlewares to allow programmers to manage large disparate resources, often forgetting that the acceptance of a software tool is often a problem of human and economic factors rather than a technical aspect. A programmer, like any other user, should have an interface that abstracts the complexity of the distributed system, allowing him to develop large scale applications in a not dissimilar way from the small local ones.

In recent years, the business software community is promoting a Java-based solution called *In-Memory Data Grids*, which has received only a little attention from the scientific software developers. In-Memory Data Grids provide a simple access interface, thanks to their abstraction of the familiar local data structures. In addition, In-Memory Data Grids are well suited to exploit the new MapReduce programming model, specifically designed to provide an easy and error-free environment for parallel distributed programming.

In this work are explore the opportunities of applying In-Memory Data Grids in data mining applications, providing an overview of parallel and distributed computing covering current Java approaches and their applications in data mining and machine learning, and In-Memory Data Grids as a novel approach to massive dataset handling. Then, several In-Memory Data Grids implementations are explored to understand their specific characteristics and an architecture based on Data Grids is compared to a classic grid implementation of a specific machine learning algorithm.

To the memory of Flavio Baroncelli, Franco Carlini and Vincenzo Tagliasco

Acknowledgements

To my supportive wife Katja, for her infinite patience.

A special thanks to Marina Ribaudo, without her support, this dissertation would have been much more difficult.

I wish to thank Giovanni Chiola, Giuseppe Ciaccio, Alessandro Verri, Annalisa Barla, and Paolo Romano, great people I meet in my doctorate.

A special mention for Patrizia Cepollina, Director of CSITA, for her kindness, and to Stefano Bencetti, Daniele Rossetto and Angelo Marando, for the help in my work.

I acknowledge the ShareGrid management team for the computing power provided through the ShareGrid distributed platform.

Thank to Cameron Purdy, Vice President at Oracle for the Oracle Coherence product, for his support with the Oracle license issues (see page 104).

Thank to Carmela Grassi for her helpfulness and to Nicola Rebagliati for his interesting hints.

I wish also thank everybody working on open source projects, especially Talip Ozturk (Hazelcast), the team of KNIME and the team of GridGain.

Table of Contents

1 Introduction

Computing is progressively shifting to handle larger and larger collections of data. The trend is the same in both business and scientific applications, but in science, a new approach to research is greatly contributing to the growth of the data size explored and generated. Classically, scientific data collections were thought as a reasoned catalogue of relevant facts organized *ex-ante* by a domain expert or they were collected to verify a specific theoretical hypothesis. Instead, nowadays has become common in many disciplines the idea of collecting the raw data and then using powerful statistical methods to identify patterns and rules *ex-post*.

It is not unusual that databases in the domains of health, biology, genomics, physics, astronomy, and engineering size in a range from gigabytes to petabytes. Moreover, modern statistical approaches to knowledge discovery in raw data, usually named *machine learning* or *data mining*, are intrinsically computational intensive. As a result, more and more power and storage capabilities are needed.

Starting from the ‘90s, the only feasible way to obtain high computing performances is to aggregate several off-the-shelf CPUs or GP-GPUs. These can be organised according three main architectures:

- custom hardware connecting thousands of processors[1] in big parallel machines;
- *ad-hoc* networks of dedicated commodity PCs (*clusters*);
- aggregate resources of non-dedicated workstations (*Desktop Grid*).

The scientific community has soon recognized the value of connecting and sharing these resources, and great efforts have been made in Europe, USA, China and Japan, to create the infrastructures known as Computational Grids. Nevertheless, many small laboratories and research groups still have no access to institutional grids, and lack of the budget and the technical staff needed to handle large clusters by themselves.

For such users, even the lower end of the multiprocessor spectrum can be beyond their possibilities. A medium-range dual 4-core CPU system costs less than 4-5 times an equivalent

1 Such as the IBM Roadrunner (6,912 dual-core AMD Opteron plus 12,960 Cell processors), the Cray Pleiades (51,200 Intel Xeon) or, at the lower end, the Asus ESC 1000 (960 NVIDIA Tesla core).

lower-end 4 processors system[2]. And increasing the number of processors, the curve of the price for processors/cores became further stepping. Thus, there is a clear trend of replacing high end systems with cheapest solutions which aggregate standard PCs using Ethernet networks.

Obviously, distributed computing requires more than connecting several CPUs and several hard drives by wires. The real challenge is coordinating tasks execution and data storage on such systems. While there are many frameworks that provide tools for distributed computing in grid and P2P architectures, as well as many systems that provide coordinated data storage across a network, there are only few systems that elegantly integrate both. In recent years, the business software community is promoting a new solution called *In-Memory Data Grids*[3], which has received only a little attention from the scientific software developers.

For a long time, scientific applications were identified with classic mathematical computations, such as matrix manipulation, Fast Fourier Transform, or partial differential equations solution, thus identifying scientific software in essence with *number-crunching*. Unfortunately, this vague definition does not help in characterizing scientific software and it is suited for engineering applications, image processing tools, financial forecasts, and even for many video games. In comparison, many statistical-based methods have instead a high data access ratio with relative elementary operations in which the computational cost of accessing the data could be comparable or above the cost of actually performing the process. Thus, performing data mining on a distributed system moving data from a central repository may not worth the cost and time in terms of bandwidth. Therefore, for data sets that are commonly processed several times, it makes sense to store them in a distributed way, at least for the experiment timeframe.

The research on grid architectures consistently focused on designing middlewares to allow programmers to manage large disparate resources, often forgetting that the acceptance of a software tool is often a problem of human and economic factors rather than technical aspects. A programmer, like any other user, should have an interface that abstracts the complexity of the distributed system, allowing him to develop large scale applications in a not dissimilar way from small local ones.

Scientific applications have a specific developing model. In software engineering, developers and final users are usually considered as two distinct communities: the users are

2 For example, in November 2009, a dual dual-core Xeon R410 Dell server costes € 1,099, while a machine with equivalent hardware but 4 processors costed € 10,179.

3 Do not confuse the well explored topic of *Data Grid* as intended in [CFKS+01], which refers to something similar to a global virtual filesystem, with the In-Memory Data Grid described ahead.

the “owners” of the problem, and the developers produce the software tools to solve the problem. This distinction does not cover the reality: in business and enterprises, advanced users have the habit of automatize or customize their office productivity tools, e.g. creating Microsoft Excel macro. Even home users who create complex web sites or book catalogues for themselves are not rare. These activities of creating, modifying or automatizing software artefacts are usually referred as *End-User Development* [cacm][Seg07].

In the landscape of End-User Development, scientific users hold a special place. Researchers are not usually frightened by the challenge of learning programming languages, and there is a long tradition of scientists developing their own software solutions to prove their theoretical models. Hence scientific software production has its own point:

- The distinction between developer and user is *fuzzy*: often there is no “external” customer and the primary user of the code is the developer himself, who want to add functionalities to advance in his research. Even when the application is shared with other users, the code often requires additions or modifications to be useful, and external users are supposed to code themselves their additions.
- Most developers are domain experts, not computer specialists: often the developers have no computer science or software engineering formal background, nor experience in professional software development. Project leaders usually find easier to teach to a domain scientist how to write the code than to explain to a computer specialist the deep scientific phenomena being captured by the code.

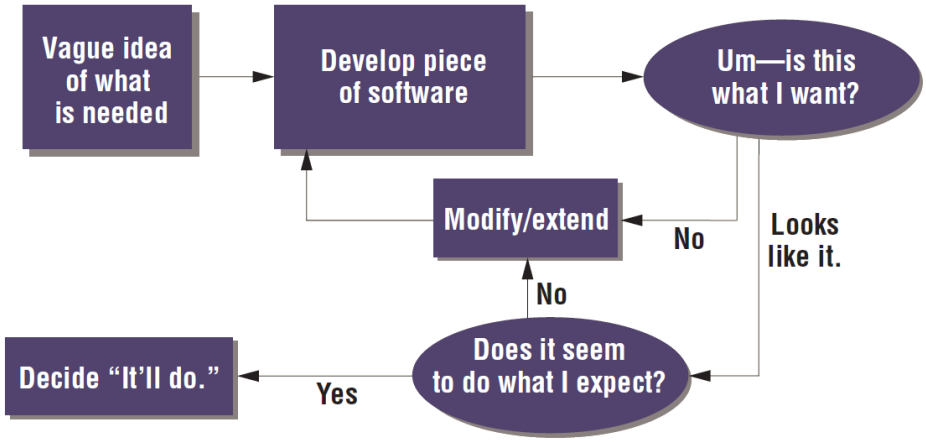


Figure 1: A model of scientific-software development (source [CLHK+06])

Unfortunately, empowering End-Users to develop they own applications is not a risk-free process. According to several investigations [CLHK+06][Seg08], scientific software development follows some typical paths:

- Science and portability are of primary concerns, while performance is not the driving force: even in projects based on parallel programming, where the code performance is clearly an important goal, the primary interest of developers is the scientific accuracy. Furthermore, these developers are aware that their code will be often used for decades, during which increasing performances will be achieved through new hardware. Therefore they spend more effort on portability than on speed, scalability and efficiency. For the same reason, the tuning for a specific system architecture is rarely performed.
- There is high turnover inside development teams in academic and research environments, where many project members, such as postdocs and graduate students, are involved for short periods. There is also a limited sharing of code and applications with other users in the same domain. Often, there is the assumption that only the developers can understand their own code. This result in poor support and documentation.
- There is little reuse of code and frameworks, even when developed internally. Third party and externally developed softwares and tools are viewed as a major risk factor and avoided. Often modern productivity tools, such as RAD, are avoided in favour of old-style ones.
- It is very difficult to verify whether the software is correct or not: in many partly-understood domains, the developers do not know the "right" answer. In the case of unexpected output, it is not clear where to find the source of the problem: if it is the underlying scientific model to be incorrect, or its translation in an algorithm, or if there is a genuine bug in the software.

A lesson on the attitude of scientists in developing software can be learned from the past fifteen years of parallel computing applications. In the cluster environment, years ago there was the choice between MPI (*Message Passing Interface*) and PVM (*Parallel Virtual Machine*). Despite the extra features that PVM offered, such as load balancing and fault tolerance, MPI's simplicity won out and, for many years, MPI has been the *de facto* standard in parallel programming and the primary parallel computing toolkit for most of the supercomputing centres. Nowadays, the relevance of MPI is decreasing due to several reasons: it is not well supported in many grid environments, Java is replacing C without offering MPI binding, and parallel programming community is now mainly focused on multicore and GU-GPU technologies. However, if a new paradigm has to be adopted, it must provide simplicity.

Java-based In-Memory Data Grids provide such simplicity, thanks to their abstraction of the familiar local data structures. In addition, In-Memory Data Grids are well suited to exploit the new MapReduce programming model. Although MapReduce in its essence could be see as an application of the classic *Divide&Conquer* technique, it was specifically designed to provide an easy and error-free environment for parallel distributed programming.

The work reported in this dissertation aims to explore the opportunities of applying In--Memory Data Grids in data mining applications and it is organized as follows:

- In Chapter 2, we provide an overview of parallel and distributed computing covering current Java approaches and their applications in data mining and machine learning, Desktop Grids architectures which exploit existing underused computational resources, and In-Memory Data Grids as a novel approach to massive dataset handling;
- In Chapter 3, several In-Memory Data Grids implementations are explored to understand their specific characteristics and evaluate their performances;
- In Chapter 4, an architecture based on Data Grids is compared to a classic grid implementation of a specific machine learning algorithm and the results obtained are applied to develop a plugin for an existent data mining application;
- In Chapter 5, the conclusions are presented plus some hypothesis for future work.

In the Appendix, a survey of the existing In-Memory Data Grid implementations is provided.

2 Background

In recent years, the idea that scientific researches could be performed digging in huge collection of raw data has become a common thinking in many fields, such as biology, social sciences, drug discovery, etc... This novel approach, known as *e-Science*, leverages in computationally intensive tasks carried out in distributed networks, composing external services and accessing to immense data sets.

At the same time, scientific applications have evolved from old-days command line tools for matrix manipulation, Fast Fourier Transform, sequence alignment, or partial differential equations solution. Nowadays scientists require modularity, nice GUIs, capability to access to web-services or databases, portability, and other features typically found in business applications. At the extreme end of the scientific software spectrum, the *e-Scientist* suggests that research can be carried out composing existing distributed resources, promoting the *point&click* Workflow Manager (*WfM* [wfmc]) as the ultimate scientific tool at the end-user level.

This has produced a plethora of WfMs applications, such as Triana [tri], Taverna [tav], Kepler [kep] and others. Many of them are mainly designed as tools for the orchestration of remote services, as the current vision in enterprise software, where the SOA (*Service Oriented Architecture*) approach is based on the composition of Web Services. In these WfMs, a node in the workflow corresponds to a service and the entities flowing in the graph are *control* data.

The basic idea under the WfMs is that the researcher will prepare and execute interactively complex workflows to visually discover unsuspecting data relations. In this envision, the computation is performed by backend grid services, and the WfM itself has not special performance requirements. Instead, it needs a wide support for different connection protocols and a modular architecture to host third-part plugins. These features are by far easier to support with Java than with other environments. And, in fact, almost all of the new generation WfMs are developed using this language.

As a result, and despite the criticisms regarding its poor performances, Java is now widespread in scientific applications. Actually, the performances of Java are not as bad as word-of-mouth says and scientific applications are not composed by number-crunching routines

only. Moreover, Java is an easy tool to start with for programming and this often wake up the do-it-yourself attitude in the end-users.

The "bad reputation" within the High Performance Computing community [PBGP+01] stuck to Java for a long time is nowadays unjustified. Just-in-Time (*JIT*) compilers, advanced Garbage Collectors and other improvements have lift the performances of modern JVM not so far from traditional FORTRAN and C languages, with results depending more on the quality of the algorithms than the language itself [Bul01][Goe05][ABCD+08][WN08]. Moreover, Java has a built-in support for enhanced multithreading since version 5, and therefore it allows for an easy exploitation of modern multi-core processors.

Probably, this progressive shift to Java in the context of scientific applications also benefits from the specific attitude of scientists, who are not worried by the challenge of learning programming languages and development tools [Seg07][CLHK+06]. As a result, researchers from diverse domains, such as biology, chemistry, astronomy, operational research, social studies, and even humanities, wanted to build their own software tools and found in Java an easy-to-learn environment to start with. Unfortunately, good-willing cannot replace specific skills and the resulting products are usually strong in domain-specific functionalities, but lack in documentation, interface functionalities, long term support and other side features.

These developers soon recognized that the computational power they need could originate from exploiting networks of commodity PCs and many scientific softwares have some built-in capability to run in a distributed fashion. But, these developers often preferred to build their home-made solutions instead of using a stable third-part toolkit for distributed computation. The reasons for this choice come sometimes from a kind of bias, such as the *not-invented-here* syndrome, but other times sound reasonable, such as having a partial view on state-of-the-art technology or fearing of sticking to a specific vendor. Unfortunately, distributed programming is difficult and fraught with danger: besides concurrency issues such as *race conditions*, *deadlocks*, *livelocks*, and other failures, there are also problems specific to distributed systems such as network unreliability, security and trustworthy, and even political issues related to span through multiple administrative domains [Deu94]. As a result, many of these custom programs are suboptimal (when not buggy), hard to maintain, and difficult to setup (often due to their poor documentation).

A wide body of research exists concerning the foundation of distributed Java-based parallel processing and data storage in grid environments, but only few hints are given to programmers who want to build an affordable and reliable real application. The solutions analysed here lay in the confluence of many topics:

1. Data mining and machine learning applications, intended as widely used general purpose scientific tools, as examples of Java-based and highly resource-demanding applications;
2. Java integration with grid computing, and specifically Desktop Grids, as an architecture which enables small organizations to exploit existing underused computational resources;
3. In-Memory Data Grids as a novel approach, born in the business software community, for massive dataset handling.

In order to provide an overview, each area will be explored in turn in the next sections.

2.1 Data mining and machine learning tools for scientific research

Data mining and machine learning softwares are especially interesting because, at the same time, they are a research topic and a research tool. It is not surprisingly that this is a prolific field for academic software production. Well known examples are the R programming language from the University of Auckland (NZ) [IG96], RapidMiner, initially developed as YALE (*Yet Another Learning Environment*) at the University of Dortmund (D) [MSKW+06], Weka born in the University of Waikato [WF05], KNIME from the University of Konstanz [BCDG+08], Orange developed at the Laboratory of Artificial Intelligence of the University of Ljubljana (SL) [DZLC04], and many others.

Many of these applications, as well as many commercial ones such as IBM SPSS Modeler (formerly SPSS Clementine®) [spss] or Accelrys Pipeline Pilot [app], are based on the WfMs paradigm. But in these cases, they are the visual incarnation of the *dataflow* programming model [Sut66], in which nodes exchange *data* rather than *controls*.

As an example of the progress in data mining applications, it is possible to follow the evolution of applications which implement *decision trees*, a basic machine learning algorithm. In the early '90s, when Ross Quinlan released an implementation of his C4.5 algorithm for decision trees [Qui93], the typical usage pattern of the program was the following:

- using a text editor, prepare two files: a `.names` file with the attribute names and a `.data` file containing the training instances;
- using standard Unix tools such as `grep`, `cut`, or `awk`, filter or transform the data file according the needs;
- launch from the shell the `c4.5` program to form a decision tree from a file of examples and save the tree in an intermediate file;

– then, using the `c4.5rules` program, re-read the unpruned decision tree to form production rules and save them in another file;

– finally, run the `consult` program to classify items using the rules previously saved [Ham00].

It is true, this operations pipeline might be automated by using BASH scripts or similar techniques, but combining several programs each one using different file formats and option switches, requires considerable computer skills and does not allow for an interactive approach to data mining.

In the late '90s, a new generation of tools, which have their prototype in Weka, introduced a coherent interface, available both as CLI and as GUI, to perform and combine all the operations of the pipeline. As an example, performing the same C4.5[4] based classification requires a single operation from the command line [WF05]. Using different arguments, the same command allows the user to invoke different algorithms and, moreover, the addition of filters and pre- or post-process operations is quite straightforward, since it requires only to set additional parameters:

```
$ java weka.classifiers.trees.J48 -C 0.25 -M 2 -t golf.arff -d
golf.model
...

$ java weka.classifiers.bayes.NaiveBayes -D -t golf.arff -d golf.model
...
```

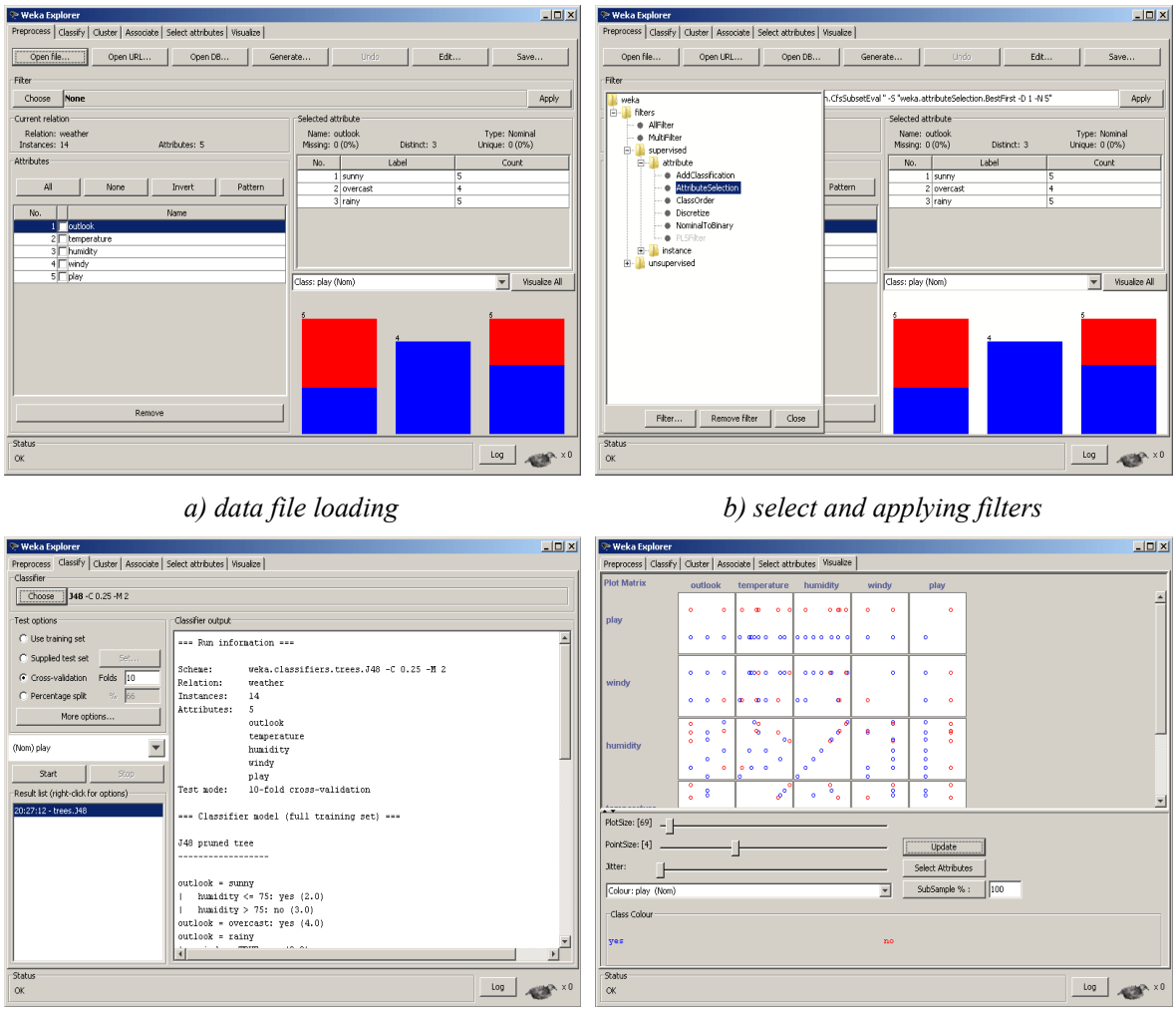
Listing 1: A command line session with Weka a J48 and a naïve Bayes classifiers applied to the same dataset

Using the graphical interface Weka Knowledge Explorer, the same operation of Listing 1 could be performed interactively and incrementally, as shown in Figure 2.

From the user perspective, the major improvements are that all the algorithms offer the same interface, the process pipeline can be saved and repeated at will, and using the GUI the process is interactive, allowing the user to tune the parameters and immediately get the feedback.

However, this is still a pipeline solution which produces linear execution stages, without branch, conditions, loops, parallelism, etc... With newer releases, Weka introduced additional interfaces, named Experimenter and KnowledgeFlow, to offer some of these functions.

4 C4.5 release 8 algorithm re-implemented in Java is named J48 in Weka, probably due to copyright reasons



a) data file loading *b) select and applying filters*

c) generate C4.5 classification tree *d) visualize the data results*

Figure 2: A Weka classification pipeline

At the beginning of the XXI Century, the workflow paradigm has become predominant. In the data mining tools panorama, it has been adopted initially by commercial products, and later on by academic open source softwares. One of the latter is KNIME, in which the workflows are visually composed using a graphical interface[5]. The C4.5 classification problem already discussed would appear in KNIME as in Figure 3.

5 Visually designed workflow can be later run in a batch mode

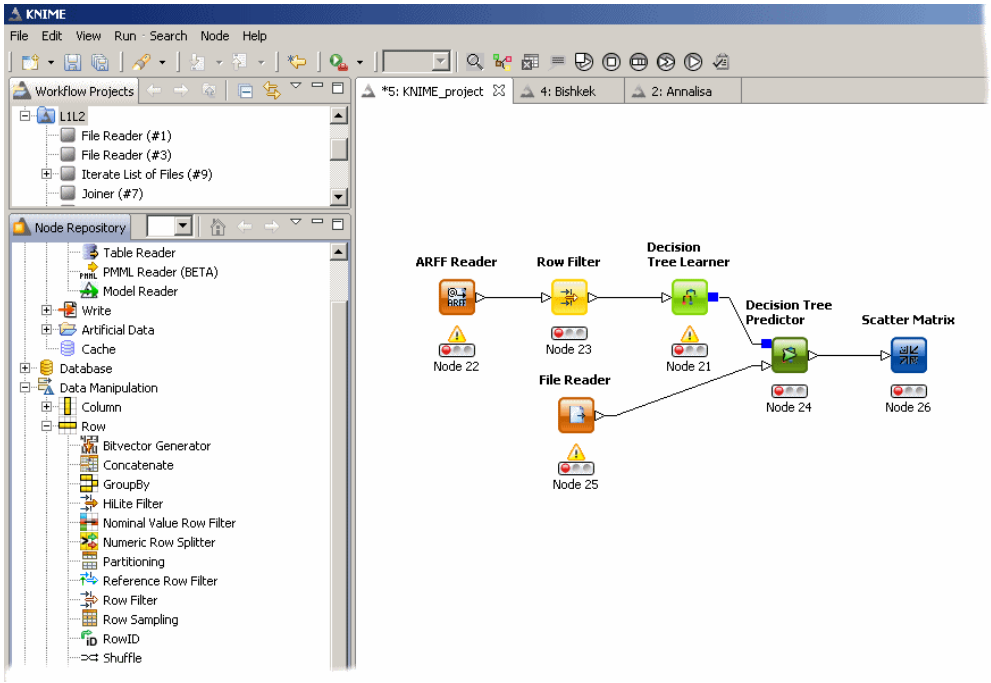


Figure 3: A KNIME workflow for C4.5 classification

This evolution has had several implications:

- the user interface and its extendibility capability have become more important than pure performances, resulting in the general acceptance of Java as the elective language for developing this class of applications;
- the users belong now to a wider audience[6] and often they even not have the basic computer skills to configure complex systems or to correctly evaluate the resources needed for performing some tasks;
- the users want, at the same time, to access to very large datasets, to apply complex algorithms and to get the answers immediately.

This new scenario has lead to a computational power "hungry" that could be satisfied only by distributed systems. In fact, many of the new tools use *ad-hoc* distributed computation facilities, such as the Weka Distributed Experiment, or they can access to specific clusters, such as the KNIME plug-in for Sun Grid Engine, or they use general grid infrastructures, such as Triana through the JavaGAT module.

6 As an extreme example, data mining has become a valuable tool for online poker players http://www.card-player.com/poker-news/8221-online-poker-the-data-mining-dilemma (last accessed 31 Dec 2009)

2.1.1 Reference applications

To understand the state-of-the-art and the current practices within developers, we deeply analysed a sample of existing data mining applications, providing that they are written in Java, have an academic origin, are licensed according to open source statements (thus allowing to explore the source code), have a good reputation within researchers, and a lively users community witnessed by updated discussions on forums and mailing lists. The applications selected according to these criteria are Weka and KNIME. In addition, we have analysed two young projects which explicitly stated to cope with memory problem, JDMP and Debellor.

Weka[7] (*Waikato Environment for Knowledge Analysis*) was developed in the late '90s by Witten and Frank at the University of Waikato (New Zealand) [WFTH+99] and it is one of the most established data mining suite of tools[8]. In 2006, the Pentaho Corporation acquired the license from the University for commercial application of Weka, but the software itself is still available as free software under the GNU General Public License. Besides using Weka as-is, many other machine learning applications integrate the Weka code in their algorithms library, making Weka de-facto standard in Java-based machine learning tools. In the suite, the built-in feature named Distributed Experiment, allows to spread a cross validation procedure among several machines, but it results cumbersome to use. Aside this, several projects have tried to extend Weka in order to exploit computational grids [CM02] [KZK04][SHSS+07][TTV05].

KNIME (*Konstanz Information Miner*) has been developed since 2005 by the group headed by Michael Berthold at the University of Konstanz (Germany) [BCDG+08]. KNIME is based on the Eclipse platform and exposes a modular and extensible design. It represents the new generation of data mining applications, in which the user visually composes data analysis workflows. Also KNIME offers commercial support, thanks to a spin-off company. KNIME was originally released under Aladdin Free Public License, but it has later switched to the GNU General Public License starting from version 2.1. KNIME can be integrated with the Cluster Execution Plugin to exploit a Sun Grid Engine computer cluster [knc]. Other distributed architectures have been evaluated in the past [SMB07], but no implementation has been publicly released.

JDMP (*Java Data Mining Package*) is a Java library for data analysis and machine learning rather than a fully-featured application, but it offers a simple graphical user interface. It has

7 Sometimes spelled WEKA

8 KDnuggets, a website specialised in datamining resources, reported Weka as the most used open source tools for many years. Recently, new competitor, including KNIME, took over or eroded the position of Weka. http://www.kdnuggets.com/polls/2009/data-mining-tools-used.htm (last accessed 30 Dec 2009)

been developed at the Technical University of Munich (Germany) starting from 2008. Some of the explicit goals of JDMP are the handling of large data sets that do not fit completely into main memory and the support for parallel processing in a computer cluster [Arn09]. These features are built on a backend library for matrix manipulation named UJMP (*Universal Java Matrix Package*) [ABN09]. JDMP is released under GNU Lesser Public License (LGPL).

Finally, Debellor is a framework for data mining and machine learning, developed since 2008 at the Warsaw University (Poland) [Woj08]. Even if it does not offer any GUI yet, the programming model is based on data flow, which seems suitable to be implemented using Data Grids. Debellor addresses the problem of handling massive datasets applying data streaming techniques. The application and the library are distributed under GNU General Public Licence.

2.1.2 Java issues with large data set

Memory requirements issues are a well known topic in high performance computing studies. Many of the known solutions suppose the direct memory manipulation, unfortunately not supported by Java. Thus, Java scientific applications have to face with both general issues and Java-specific ones.

In machine learning, especially when applied to bioinformatics or computer vision, there is a frequent need to process huge volumes of data, too large to fit in main memory. As an example, the `OutOfMemoryException` was a recurrent topic in Weka and KINIME mailing lists and forums[9]. Several causes contribute to make memory one of the major issues in Java-based data mining applications. Aside problems related to algorithms unable to handle data streaming, many others are tied to the memory management mechanism used by a Java system.

In the common commodity PCs architecture, a 32 bit hardware, 32 bit operative system[10], or 32 bit Java Virtual Machine cannot usually handle more than 4GB of physical and virtual memory. Although there are some workarounds, such as enabling the PAE (*Paging Address Extension*) extension on Windows or recompiling the Linux kernel with HIGHMEM op-

9 https://list.scms.waikato.ac.nz/mailman/htdig/wekalist/ http://forums.pentaho.org/forumdisplay.php?f=81 and http://www.knime.org/documentation/faq (last accessed 31 Dec 2009)

10 32 bit OSs are still the majority in common usage. It is hard to obtain statistics on this topic. For home users, Steam, a platform for online gaming, reported in December 2009 that only 27% of the 2.3 million of users has a 64 bit release of Microsoft Windows system (http://store.steampowered.com/hwsurvey). For web surfers, the Wikimedia Fundation reports, in Novemebr 2009, that more than 56% of about 4.5 billion of accesses was performed using Microsoft Windows XP, which was mainly distributed in the 32 bit version (http://stats.wikimedia.org/wikimedia/squids/SquidReportOperatingSystems.htm)

tions, these are demanding for a normal user. In addition, many 32 Java Virtual Machines require to allocate the heap in contiguous memory space which further reduces the available memory. Providing enough memory could be a problem even on 64 bit platforms: most of the consumer hardware does not support more than 16 Gbyte of physical RAM, and 64 bit JVMs use about 50% more memory than a 32 bit one to allocate the same object set.

Many of the In-Memory Data Grids products claim to mitigate this problem in two ways. The first, more traditional, is to use the disk as a memory extension and acting as a L1 cache. The second, is to exploit the aggregation of the memories of all the nodes in the cluster, presenting them as an unique heap to the local client. This second mechanism might work only when the Data Grid is configured for so-called *data partitioning*. In replicated memory scenario, each node allocates the same amount of data, and the memory available to the application is limited to the smallest heap in the cluster.

2.1.3 Memory issues in data mining

The evolution of the data mining tools has had an impact on the memory requirements. Within single-task applications, the memory footprint is strictly related to the algorithm implemented. Some of these algorithms allow a *stream* or *incremental* approach, in which the data are loaded item-by-item, used to computing a partial result and then discharged when consumed. This approach could potentially allow the analysis of huge datasets regardless the size of the available memory. Unfortunately, many of the known algorithms in machine learning requires to access to the whole dataset and in Java this means that all data must be *materialized* in memory.

Developers of data mining and machine learning tools have progressively applied more sophisticated strategies to cope with memory problems. Weka simply ignores this: data are represented using the class `weka.core.Instances` which wraps a non synchronized `Vector` replacement named `weka.core.FastVector`. Being a concrete class rather than an interface, it is not easy to replace `Instances` with a better implementation. Moreover, each node in the pipeline duplicates, in the meaning of Java reference handling, the data instead of providing a new view on them. As a result, long pipelines require a huge amount of memory. In applications similar to Weka, even if a single stage could have relative small memory requirements, the data of all processing stages are kept in memory, thus resulting in a very fast memory exhaustion.

Newer and more engineered applications recognized that memory in each stage (*node* in a workflow) could be managed independently, allowing sophisticated strategies. KNIME employs a heuristic caching strategy to move part of a data table to the hard disk when it becomes too large to fit the memory. This does not solve the problem at the node level, but

limits the overall memory requirements. As a collateral benefit, it offers a checkpointing functionality, since intermediate results are *passivate* on the disk and can be later restored within two user sessions.

Another strategy, at the moment applied only in few applications, is switching to a *stream architecture* which does not enforce data materialization. Debellor [Woj08] is a young data mining library based on this concept. Data are passed between interconnected algorithms one-by-one, as a stream of items that can be processed on the fly, without full materialization. Unfortunately, Debellor at this stage includes few stream-oriented native modules, while other algorithms are imported from the Weka library, thus resulting in buffering which re-raises the same problem of Weka.

2.1.4 Exploitable parallelism in data mining

Thanks to the recent diffusion of multicore CPUs, detecting program patterns which might benefit of parallelism is considered again an hot topic. Data mining applications, being composed of several stages of processing, offers many points for insertion of parallel solutions.

Specifically, in WfM-based data mining, there are some elective ways to exploiting parallelism in the workflow enacting:

- the concurrent execution of different paths in the workflow;
- the parallel processing of the rows of the data table;
- the application of parallel versions of machine learning algorithms;
- the concurrent execution of cycles or sub-workflows.

The parallel execution of different paths in a workflow is an obvious technique inherited from the overall data flow design and not tied to data mining applications. In Java based WfMs, due to the native support for multithreading, this technique is widespread. In both Weka Knowledge Flow and KNIME, for example, each node runs in its own thread.

Many tasks in data mining, especially those related to the data preprocessing such as row filtering or field transformation, handle the rows as independent data chunks and allow parallel processing. Using a framework such as the `java.util.concurrent` package, this parallelization strategy is easily exploitable even by casual programmers. Nevertheless, this technique is not widely adopted, often because the core libraries of many applications have been designed before Java 5 and they are not easily portable to this new idiom.

The development and the efficient implementation of parallel versions of machine learning algorithms is instead a matter for Computer Scientists. Many machine learning concepts imply strong dependencies and correlations in data which require a case-by-case analysis.

Often, complex or multistep operations in data mining might be considered as whole, and handled as a single operation reiterated with different parameter values (*parameter sweep*). We recall for example the cross-validation procedure, in which the data are repetitively partitioned into complementary subsets, called the *training set* and the *validation set*, the first used to train a machine learning algorithm according to some parameter values and the second to estimate the resulting model errors. This is a very expensive, but embarrassingly parallel, procedure and is not a surprise that almost all data mining applications support parallel or distributed cross validation.

A comprehensive bibliographical reference on the research on distributed data mining is kept updated by [BDLK].

2.2 Java-based grid computing

Computational grids are usually classified under two categories, *institutional grids* and *Desktop Grids*. Institutional grids aim to provide a transparent, secure, and coordinated access to various computing resources such as supercomputers, clusters, databases, scientific instruments, or storage facilities owned by different institutions by means of *virtual organizations*, which aggregate heterogeneous, large-scale, and multiple-institutional resources. On the other hand, there is the consensus of many authors [Cap07][CBKB+08][VC08] that the main characterization of Desktop Grids is the design goal of harvesting the idle CPU cycles of desktop Personal Computers (PCs) assigned to usual home/office activities in order to accelerate the performances of a third part application. The PCs can be connected over the Internet, in which case participation is usually on voluntary basis, or in a corporate/university network.

The earlier prototype of Desktop Grid was SETI@Home [ACKL+02], released in May 1999 with the original goal of analysing radio signals searching for clues of extra terrestrial intelligences. Besides this fascinating, but vane, intent which persuaded over 200,000 people in the first week and more than 3.83 million after three years to devolve their resources to the project, SETI@Home was a formidable proof of the viability and practicality of the Desktop Grid concept. The infrastructure underlying SETI@Home was generalized by David Anderson to create the *Berkeley Open Infrastructure for Internet Computing* (BOINC) [And04]. This and similar projects are based on a central service which dispatches the workloads, consisting of data to be analysed, to the applications running on remote PCs. The edge node

application is dedicated and vertically integrated, and no generic use of such computing resources is possible.

Several other architectures of Desktop Grid have been proposed, either from academy and industry, and for a complete taxonomy we refer to [CBKB+08].

2.2.1 Accessing the grid

The main components of a Desktop Grid infrastructure are the *resources* and the *scheduling managers*, which take care of keeping a registry of the resources, allocating them to the user according his request, dispatching the tasks to the resources, collecting the results, handling failures, and providing the required security level.

From the perspective of a programmer who wants to exploit a grid service for his application, the main concern is how to access to the resources of the Desktop Grid and how the tasks (and the data) have to be described and encoded.

In early design, the use of computational grids mimics the familiar local interaction, in the same way of a SSH remote command execution. As an example, using the old releases of Globus Toolkit to run the `date` command on the remote host `gridnode.somedomain`, requires to type the command in the shell, as shown in Listing 2.

```
$ globus-job-run gridnode.somedomain /bin/date
… runs /bin/date on the remote host
$ globus-job-run gridnode.somedomain -s myprog
… transfers the program file myprog on the remote host and runs it
```

Listing 2: Sample of Globus Toolkit command interface (surce Globus Toolkit 2.4 manual)

To be effective, the grid must offer some programmable entry points through an *Application Programming Interface* (API). According to Mateos et al. [MZC08], it is possible to identify the granularity of a grid access service "*as the granularity of the individual components*[11] *that constitute an executing gridified application from the point of view of the grid middleware*", where the components are the execution units (*jobs* or *tasks*) to which the grid provides scheduling and execution.

Grid access granularity takes continuous values ranging from the smallest to the largest possible component size. The spectrum can be divided into three main classes:

- **Coarse-grained**: the application element executed on the grid is a complete application behaving as a "black box" that receives a predetermined set of input parameters (e.g. a sequence of numerical arguments, input files, etc...), performs some computation and then returns back the result to the executor using the same mechanism of the

11 The authors use the term "component" to refer to any single piece of software included in a larger system

input (e.g. the standard output, a result file). The application as-is cannot take advantage of distribution, parallelization or scheduling to achieve higher efficiency. Access schemas provided by the Globus Toolkit [Fos05], the BES/JSDL standard [jsdl], or the Condor submission system [LLM88], are well known examples of coarse-grained API.

- **Medium-grained**: the programmer identifies in the application the modules suitable to run concurrently and remotely. Then, in the initialization phase, the middleware maps the modules on existing remote services or transfers the executables on remote nodes. The application accesses the modules using a RPC-style communication mechanism. Medium-grained access API are provided, as for example, by ProActive[12] [CKV98], or by the GigaSpaces XAP when default component deployment is used [Coh09]. This approach could provide some capabilities of dynamic component deployment and invocation.

- **Fine-grained**: the distributed components are generated and spawned at runtime on the invocation of a method or procedure. A key difference with the previous categories is the ability to cope with recursive distribution methods. The programmer is relieved of scheduling and synchronization issues, while he can focus on parallelism and asynchronism. The middleware is not a pure access interface anymore and it must provide sophisticated execution services to efficiently deal with a potentially large number of tasks. Examples of fine-grained APIs are offered by the middlewares JPPF [jppf], Satin [vNMK+05], or GridGain [ggp].

Being the above classification based on the access framework used by the developer to build a grid-enabled application, it is worth noting that it could be a matter of perspective. If the user is building a workflow, using a WfMs such as in Triana[13] or in Taverna, a coarse-grained grid is confined in a single node of the workflow graph and appears as a component. Many of the grid toolkits are bundled with a dedicated workflow manager, such as Karajan in the Java CoG Kit [LFGL01], or Pegasus in Condor.

Moreover, if the API is developed for a portable language with some built-in code serialization capability, such are Java or Python, even a coarse-grained API can be encapsulated in a finer interface. For example, we have developed a framework for the Python language [BFSV09] in which two classes, `Job` and `Grid`, take care of all the logic involved in creating and launching jobs on a grid infrastructure. The developer of the client application has

12 The authors of [MZC08] classified ProActive as a coarse-grained middleware, but ProActive's a*ctive objects* are clearly modules, not application. The same classification of ProActive as medium-grained could be found in [BBBC+06]

13 Triana can also establish Peer-to-Peer grids that connect Triana's instances running on different hosts

simply to implement a subclass of `Job`, mostly just wrapping the existing code to be distributed, and then inserting in the main function few calls to the methods of `Job`. The framework has been used to port an existing multithread medical image process application, developed for local use on multicore CPUs, on an infrastructure which offers a coarse-grained API [CBSA+03], changing only the invocation of threads with the dispatching of remote tasks.

From a Java programmer's perspective, the three classes just discussed could be exemplified as follow. Well known coarse-grained computational grid APIs are the Globus CoG Kit and JavaGAT. Both toolkits allow to run remote programs, redirecting the standard input and output, and staging input and output files. Listing 3, based on the Globus CoG Kit, shows how to create from Java the descriptor for the batch execution of the Unix command `ls -la`, reading the input from the file `testInput` and redirecting the output to the file `testOutput`.

```
JobSpecification spec = new JobSpecificationImpl();

spec.setExecutable("/bin/ls");
spec.addArguments("-l");
spec.addArguments("-a");

spec.setStdInput("testInput");

spec.setStdOutput("testOutput");

spec.setBatchJob(true);

spec.setAttribute("count","546");

task.setSpecification(spec);
```

Listing 3: Example of coarse-grained access to the grid (source Java CoG Kit examples wiki)

The `JobSpecification` will be passed to the runtime which will submit the task to the grid. Another Java fragment, using JavaGAT, is shown in Listing 4, where the execution of the application `hostname` is invoked, redirecting the output to the file `hostname.txt`. In this case, being JavaGAT an abstraction layer, the true grid infrastructure that will execute the task depends on an external configuration.

In both cases, exploiting a computational grid requires heavy file manipulations to handle input and output, very far from the usual Java programming style. Moreover, the exploitation of the grid is limited to run executables already available on the remote hosts or, in rare cases, to portable program staged-in where is available a suitable runtime and security measures do not prevent this.

These limits are overcome when adopting a component based API. Starting from the release 1.1 in 1997, the Java platform supplies a built-in framework for inter-process communication called RMI (*Remote Method Invocation*) [rmi]. RMI uses a RPC-style communication mechanism with object passing (using serialization [sun96]) inspired to Network Objects of Modula-3 [WRW96]. Using RMI, an object instantiated on one JVM can invoke methods on an object on another JVM, providing that the invoked object implements the interface `java.rmi.Remote`. In such a way, the access to a remote object is almost transparent and requires only few additional care compared to usual programming. In fact, most of the home-made distributed modules are RMI-based. Several medium-grained grid toolkits are based on some extension or reinterpretation of RMI [SHW98][vNMH+02][p2pmpi].

The key difference between a distributed component service in a broad sense, such one based on CORBA or Web Services, and a grid toolkit is that in the former the deployment of remote components is delegated to the administrator of the system, while in the latter the deployment is part of the client program itself. As an example, in the first part of Listing 5, based on ProActive, the `node` `VN` is obtained from the deployment description. Then, in the second part, an object of class `Worker` is deployed on the `node`, passing the `params` to the constructor, and the stub `charlie` is returned. The methods of `charlie` can be invoked later using standard Java calls, as in RMI. Note that the lifecycle of the remote object is independent from the launching program. In the case of medium-grained API, it is no evident how to efficiently distribute embarrassingly parallel tasks. The naïve solution seems to allocate the same object on different nodes and then apply some *ad-hoc* strategy to split the tasks between the nodes, but this requires a sensible effort by the programmer.

```
public class SubmitLocalJob {
     public static void main(String[] args) throws Exception {
          GATContext context = new GATContext();
          SoftwareDescription sd = new SoftwareDescription();
          sd.setLocation("file:///bin/hostname");
          File stdout = GAT.createFile(context, "hostname.txt");
          sd.setStdout(stdout);
          JobDescription jd = new JobDescription(sd);
          ResourceBroker broker = GAT.createResourceBroker(context);
          Job job = broker.submitJob(jd);
          while (job.getState() != Job.STOPPED
                    && job.getState() != Job.SUBMISSION_ERROR)
               Thread.sleep(1000);
     }
}
```

Listing 4: Example of coarse-grained access to the grid (source GridLab "GAT Tutorial")

```
/***** GCM Deployment *****/
File applicationDescriptor = new File(gcmaPath);

GCMApplication gcmad;
try {
     gcmad =
PAGCMDeployment.loadApplicationDescriptor(applicationDescriptor);
} catch (ProActiveException e) {
     e.printStackTrace();
     return;
}
gcmad.startDeployment();

// Take a node from the available nodes of VN
GCMVirtualNode vn = gcmad.getVirtualNode("VN");
vn.waitReady();
Node node = vn.getANode();
/*************************/

// Set the constructor parameters
Object[] params = new Object[] { new IntWrapper(26), "Charlie" };

Worker charlie;
try {
     charlie = PAActiveObject.newActive(Worker.class, params, node);
} catch (ActiveObjectCreationException aoExcep) {
     // the creation of ActiveObject failed
     System.err.println(aoExcep.getMessage());
} catch (NodeException nodeExcep) {
     System.err.println(nodeExcep.getMessage());
}
```

Listing 5: Example of medium-grained access to the grid (source ProActive 4.2.0 documentation)

It is advisable that an API design for parallel execution looks familiar to a Java developer. In this case, a more sounding pattern for a Java programmer would be offer by the standard `java.util.concurrent` package (`j.c.u` for short) and, specifically, by the interfaces `Future` and `ExecutorService`.

The model of `ExecutorService` has been followed by several Java-based grid toolkits [Fer09]. As in Listing 6, within the JPPF framework [jppf], tasks extend the `JPPFTask` abstract class, so they can be declared on-the-fly using anonymous classes, packed in a `JPPFJob` and submitted to remote execution. The task, once completed, returns the result or re-raise the exception occurred remotely. This role is quite similar to that played by `Callable`/`Future` in the `j.u.c` package, with the difference that `Callable` and `Future` are different interfaces rather than a single abstract classes.

```
public class JPPFTest implements Serializable {

   public void gridSqrt(final double d) throws Exception {
          // create a JPPF job
          JPPFJob job = new JPPFJob();

          // give this job a readable unique id that we can use to
          // monitor and manage it.
          job.setId("Square Root Job");

          // add a task to the job.
          job.addTask(new JPPFTask() {
          public void run() {
            setResult(Math.sqrt(d));
              }
          });

      JPPFClient jppfClient = new JPPFClient();
      List<JPPFTask> results = jppfClient.submit(job);

      System.out.println("SQRT: " + results.get(0).getResult());

      jppfClient.close();
   }
}
```

Listing 6: Example of fine-grained API: a task extending `JPPFTask` *is declared on-the-fly using anonymous class, packed in a* `JPPFJob` *and submitted to remote execution.*

In the design of JPPF and other similar frameworks, it is crucial to provide mechanisms that allow the remote agents to dynamically transfer the classes in the executable binary form if it si not available locally. Otherwise, the remote agent will be able to execute only classes accessible to the local *classloader*, preventing to realize a real on-demand grid facility. Currently, not all toolkits provide this functionality. When it is supported, the classes dynamically loaded must be discharged when the application terminates, otherwise newer releases of the same application could produce conflicts between different class versions. The lifecycle of the task is thus tightly bound with the application which creates it, rather than with the agent executing it on the remote node.

Notice that the same framework can offer different access granularity. The base execution unit in GigaSpaces XAP is the *Processing Unit*, a software component which bundles, a class conforming to the JavaBean convention regarding public methods, a shared registry identifier, the dependent libraries and a XML deploy descriptor. But XAP also exposes the *Async API*, a simpler support for asynchronous tasks and even an implementation of `j.u.c.ExecutorService` if neither asynchronism nor task routing is required[14].

14 http://www.gigaspaces.com/wiki/display/XAP7/Terminology+-+Basic+Components and http://www.-gigaspaces.com/wiki/display/XAP7/Task+Execution+over+the+Space (last accessed 13 Jan 2010)

Alternative to the model offered by the `j.c.u` executors, it is worth noting that the MapReduce paradigm, introduced later in Section 2.3.3, has had a special impact in the design of this generation of frameworks. Many recent implementations, such as Granules [PEF08], Satin, or GridGain, use a pure MapReduce model, based on a functional programming style, or a more general *Divide&Conquer* approach. This add the capability of splitting automatically the job into several tasks and then allocating them on different hosts using load balancing or fault tolerant strategies.

Another feature often implemented in frameworks designed with a fine-grained programming model is the auto-organization capability based on a P2P (*Peer-to-Peer*) architectures. From the programmer's viewpoint this results in a network configuration reduced to minimum or, often, no configuration need at all. This contrasts with the common experience when working with RMI. Another great advantage is the ability of a P2P grid of completing the task, even taking a longer time, in case of crashing or unreachability of all nodes except the user's client one.

2.2.2 Checkpoint and shared storage

In scientific computation it is frequent that a numerical calculation requires a long execution time, perhaps hours or even days, without any user interaction. If the computer crashes, the time and other resources spent as far are lost. Moreover, the user may wish to periodically interrupt the calculation, to check the intermediate results and adjust parameters if needed, or simply to use his computer for some other jobs. To cope with these problems, the calculation is periodically halted in some intermediate, but consistent, state (*checkpoint*) and the results obtained so far are saved in some persistent storage system. So that, if the calculation aborts, it might be restarted from the last available checkpoint. If the checkpoint system also allows to move a checkpoint from one host to another one, checkpointing can implement a dynamic load balancing facility.

In distributed computation, the presence of more failure points renders the previous scenario even worst, and checkpointing is a well known problem. In the current design of institutional grid frameworks, a checkpoint is a persistent snapshot of the running state of an application. It exists in some storage system, such as one or more files or database records. Checkpointing includes the ability to save and recover the application state on a single grid-connected resource, and it could also include the migration of checkpointed jobs to other resources, but it *does not include* the checkpoint and recovery of jobs running simultaneously across multiple computing resources [BHKM+04].

Checkpoint systems are classified as System-level and User-defined. System-level checkpoint systems provide automatic, transparent checkpointing of applications at the operating

system or middleware level. The application is handled as a black-box, and the checkpointing mechanism has no knowledge about any of its characteristics. Typically, this involves capturing the complete process image of the application. User-defined checkpointing relies on the programmer support for capturing the application state. A detailed comparison between the two approaches can be found in [SS98], while in this work we focus on User--defined checkpointing.

2.2.3 Desktop Grids as deployment tool

From a certain viewpoint, grid tookits can be intended as software deployment tools, which execute the *install*, *activate*, and *deactivate* steps of the user application lifecycle. The main differences with other remote deployment methods, such as SSH remote execution or Microsoft Active Directory Group Policy, are the capability of selecting resource based on rules, such as "deploy on host having RAM > 1024 Mbyte and running Linux OS", and the fault recovery mechanisms, which should ensure the user application terminates, even in case of node crash.

In order to exploit this opportunity, there are some environmental preconditions to meet: the grid run-time must allow the transfer and execution of arbitrary applications written in a high level or interpreted programming language such as Lisp, Perl, or Java, and, being this a serious security threat, the grid nodes must be hosted in a LAN with no critical applications and a certain control degree by a trusted technical staff. This is a common scenario in corporate LANs or university computer rooms.

Condor, OurGrid/ShareGrid [CBSA+03], or Zorilla [DvNH06] are middlewares that an user can manage in such way. To submit a job, the user has to prepare a description file, named *Job Description File*, *Submit Description File*, or similar, which specifies the environment desired to run the application, the file to stage-in and -out, and the command to execute.

```
Executable    = foo
Universe      = standard
Requirements  = Memory >= 32 && OpSys == "SOLARIS28" && Arch =="SUN4u"
Rank          = Memory >= 64

Error   = err.$(Process)
Input   = in.$(Process)
Output  = out.$(Process)
Log     = foo.log

Queue 150
```

Listing 7: A Condor's submit file which queues 150 runs of `foo` (source Condor 7.4 documentation)

As an example, the file in Listing 7 queues 150 runs of the program `foo` which has been compiled and linked to run on Sun Solaris 8 systems. Similarly, Listing 8 shows a Job Description File for OurGrid which loads the executable `mytask` on the remote nodes and queues two runs of it.

```
job :
     label : myjob7
     init :        put input input
               store mytask mytask
     remote : mytask < input > output-$TASK
     final : get output-$TASK output-$TASK

task :
     init :put alpha input
     store mytask mytask
task :
```

Listing 8: An OurGrid JDF file which submit two instances of program `mytask` (source OurGrid 4 documentation)

If the application deployed has P2P auto-organization capabilities, the peer nodes can establish their own overlay network to communicate each other regardless the grid infrastructure capability.

2.3 Distributed Data Storage

As discussed in Section 2.2.1, fine-grained Desktop Grid APIs mimic the usual Java multithread programming idiom and should offer an adequate abstraction level to most of the Java programmers. But these APIs do not offer any comparable help for sharing data between nodes.

One of the problem of "going distributed" is the challenge posed by the access to large collections of data. Traditional cluster and grid architectures usually rely on some shared file storage services, such as NFS in Condor clusters, or on-demand file transfer mechanisms, such as GridFTP in Globus framework[15]. Neither these strategies, based on files, have the level of abstraction familiar to Java programmers. Some custom distributed implementations, such as Weka Distributed Experimenter, use a RDBMS through JDBC as shared storage, but also this solution has not an adequate abstraction level. In fact, as demonstrated by the progressive diffusion of frameworks such as JPA (*Java Persistence API*), Hibernate, and other O/R tools, a Java programmer is habit to think in terms of objects and he wants to be able to make the object *persistent* in a transparent way.

15 A comprehensive evaluation of several file transfer systems oriented to grids is available in [AC04]

This problem has received from the scientific community less attention than the computing aspect, and only few usable proposals could be found [ABJ05], none of them designed for Java. Instead, as in many other cases of programming tools, it has caught the interest of the business software community, primarily within web applications, which haa developed several solutions under the definition of *In-Memory Data Grid* (IMDG) or *Distributed Cache*. The two definitions are partially overlapping and the products are progressively converging[16] to the same set of functionalities. For this reason, in the following we will use the term Data Grid for both.

A Data Grid allows to distribute data across the nodes of a network in their original form of objects with an API very similar to the local data structure usage. Basically, the storage layer of an IMDG offers the following features:

- **Data replication**: in this configuration, a copy of all the data is stored in all the nodes of the grid. This strategy consumes the most resources, however it is the most performing solution for scenarios in which reads far exceed writes, as data is available everywhere for immediate access. Data updates are notified or replicated to each node from the originating node using different strategies such as data invalidation, synchronous updates and others.
- **Data partitioning**: the whole data set is split into multiple subsets and every subset is assigned to a grid node. In the purest form, data is not replicated between nodes, and each node is the only responsible for its own subset of data; every access to a non local element requires a network operation and a crash of a node leads to a data loss. On the other side, the available storage space corresponds to the aggregate of the storages of all the nodes. However, partitioning usually includes the possibility of keeping a configurable number of backup copies in different nodes, to guarantee a certain degree of fault tolerance.
- **Distributed locks or transactions**: in order to ensure coherency within complex updates, Data Grids usually support at least a lock mechanism. Most of the Data Grids also support various distributed transactions, with different isolation levels, such as *Read Committed*, *Write Committed*, *Serializable*, etc... Some products are also compatible with J2EE or XA container managed transactions.
- **Persistent storage and overflow**: most of the products can replicate the data on the disk or on a database using a write-through or write-behind strategy, making it persistent. When the data is recalled through the associated key, if the entry does not ex-

16 At least two cache product, GigaSpaces XAP and IBM eXtreme Scale, available for years, have introduced the term Data Grid in the documentation only recently.

ist in-memory, the system retrieves it from the persistent store. In some cases, this allows to overflow data to the persistent store and flush them from the memory if the memory is close to fulfil.

- **Notifications**: the node could register callbacks, following the model of JavaBeans Events, to be notified of events regarding the grid (node leaving or node joining) or the storage (key added, data updates).

Notifications are enough to perform distributed computations, but Data Grids might also offer an execution layer, including one or more of the following features:

- **Distributed execution**: allows to execute a task, usually a serializable `Runnable` or `Callable` object, in the grid. The task could be executed, synchronously or asynchronously, on a specific node, on all or on a subset of the grid nodes.
- **Data affinity** or **co-location**: in a partitioned configuration, this is the possibility to explicitly suggest to store an entry on the same node hosting the key of another entry, typically in a parent-child relation. This feature is tightly related with the location aware distributed execution.
- **Location aware distributed execution**[17]: in systems supporting this feature, it is possible to associate a task with a key or a collection of keys identifying the data to be processed. The task will be transferred to the node which hosts the data and executed locally to the data itself. In case of multiple keys, eventually specified using a filter, tasks are replicated on all nodes hosting the data and executed in parallel.
- **Messaging**: this allows the nodes to communicate each others, usually supporting either point-to-point and broadcast messages.
- **Peer-to-peer architecture**: P2P products can discover other peer nodes and then join the grid automatically, usually using IP multicast.

The details on the features supported by each product are available in the survey in the Appendix.

It is worth to mention two other classes of product which partially overlap with the features of In-Memory Data Grids: embedded databases and *Single System Image* (SSI) systems.

In the Java ecosystem, embedded RDBMS, such as HSQLDB (also known as Hypersonic SQL[18]) or Apache Derby, are well established. An embedded database, in addiction to usual client/server mode, could operate in a *standalone*, or *in-process*, with *in-memory* tables ar-

17 This concept is not yet defined by an established term; vendors use “Task routing” (GigaSpaces), “Parallel data processing” combined with “Targeted execution” (Oracle Coherence), “Key based distributed executions” (Hazelcast)

chitecture. In the in-process mode, the database runs in a different thread of the same JVM of the application, using direct object exchanges to communicate each other. The main goal of embedded RDBMS is to support ACID properties and SQL query language. Compared to a Data Grid, embedded RDBMS lack of P2P features and distributed execution support, but they offer a better support and a familiar model for data persistence. Therefore, they are complementary and in some cases could be used together, using the RDBMS to handle the persistence and the IMDG as a write-behind cache. It is worth noticing that some IMDGs are starting to include SQL-like query capabilities.

SSI applications present to the programmer a virtual aggregation of several different machines; in case of Java, this means several JVMs [Lau03]. SSIs are rare, with the notably exception of Terracotta Distributed Shared Objects (*DSO*) [terr]. Terracotta instantiates an infrastructure very similar to an IMDG and can cooperate with it. Unfortunately, it uses a client/server architecture which results hard to dynamically deploy in a Desktop Grid-like infrastructure.

2.3.1 Data Grid APIs

Within Java binding to IMDG, there are three main approaches to hide the underlying complexity with a familiar façade, sometimes exposed together within the same product: exposing the familiar `java.util.Map` interface, implementing the extension of a JSR-107 (JCACHE) or using something like Linda's *tuplespace*.

In systems based on the `java.util.Map` model, the programmer usually obtains, using a factory, an object implementing the standard `Map` or `ConcurrentMap` interface which is backended in the Data Grid, as shown in Listing 9. All the instantiated maps are shared by all the nodes of the Data Grid. This is a nearly drop-in replacement for local data structure and allowed an easy “gridification” of existing applications. By contrast, usually the configuration of the toolkit cannot be modified by code and has to be performed by means of external files. A drawback in most of the current distributed `Map` implementations is the lack of support for generics. Except for Hazelcast, all other implementations require a type cast for each read operation.

The JSR-107 (JCACHE) is a proposal for an API and a semantics for temporary, in memory caching of Java objects, including object creation, shared access, spooling, invalidation, and consistency across JVM's [jsr107]. Even if a JSR-107 cache could resemble to a plain `Map` at first glance, there are some differences, such as: *put()* and *replace()* are different opera-

18 The original Hypersonic SQL project was stopped in 2001, and the HSQLDB Group take over the Hypersonic SQL code.

tions, a get for a non existing element raise an exception instead of returning null, and the cache can be divided in *regions*.

```
import com.hazelcast.core.Hazelcast;
import java.util.Map;
import java.util.Collection;

Map<String, Customer> mapCustomers = Hazelcast.getMap("customers");
mapCustomers.put("1", new Customer("Joe", "Smith"));
mapCustomers.put("2", new Customer("Ali", "Selam"));

Collection<Customer> colCustomers = mapCustomers.values();
for (Customer customer : colCustomers) {
        // process customer
}
```

Listing 9: A distributed data structure presenting a `Map` interface; each node in the cluster or grid accesses to the same entries if it obtains the map using the same name (source Hazelcast documentation)

Although designed for cache implementation, in which transaction support is not a requirement and invalid objects can be recreated from the original source, it has been progressively adopted by several implementations of Data Grids. The standardization process, started by Oracle, results inactive since 2001. After an initial enthusiasm, many projects which earlier adopted the JSR-107 interface threw away the most cumbersome aspects[19], such as regions, resulting in slightly incompatibilities between different libraries. Systems inspired to JSR-107 offer programmatically configuration capabilities, in addition to file based ones, and awareness of the distributed nature of the data as shown in Listing 10. JSR-107 caches provide also an event notification feature.

```
CacheManager singletonManager = CacheManager.create();
Cache memoryOnlyCache = new Cache("testCache", 5000, false, false, 5);
manager.addCache(memoryOnlyCache);
Cache test = singletonManager.getCache("testCache");

Element element = new Element("key1", "value1");
cache.put(element);
element.getCreationTime();
...
manager.shutdown();
```

Listing 10: A JSR-107 inspired cache. Configuration could be changed programmatically. The data are accessed through the `Cache` interface (source Ehcache documentation)

The less common interface is inspired to Linda's *tuplespace* [GCCC85], which presents an associative memory and an event system to concurrently operate on it. Java incarnation of

19 A detailed description of critical points in JCACHE can be found in JCS documentation. http://jakarta.apache.org/jcs/JCSandJCACHE.html (last accessed 5 Jan 2010)

the tuplespace is the JavaSpaces specification, part of JINI specifications [jini]. JavaSpaces is designed to support distributed transactions and persistence, but it is less straightforward to use and requires several additional steps compared to other solutions.

Most of the JavaSpace implementations currently available are designed for concurrent programming within single JVM and they do not support any distribution feature. Exception of distributed products are GigaSpaces XAP, the old IBM Tspace [tspace], and the open source project SemiSpace [ssp] which could integrate with Terracotta.

2.3.2 Execution layer and programming model

The In-Memory Data Grids are characterized by the design oriented to data sharing. The computational task is demanded to another component not necessarily included in the framework. From the developer perspective, current Data Grids implementations offer the choice between three main programming models. One is the already mentioned, long standing Linda language[20] model. An alternative, tightly bound to Java, is the `j.c.u.ExecutorService`. The third is the MapReduce model, a current hot topic. In the next paragraphs, we will explore the support offered by the selected frameworks to these programming models.

Highly integrated with the concept of tuplespace, Linda defines also a set of operations, in the form of programming language extensions, for facilitating parallel programming. In this architecture, concurrent processes exchange data by generating, reading, and consuming the entries (*tuples*) in the registry. The actions are coordinated by an event-driven mechanism and could use an implicit locking mechanism to ensure proper synchronization in case of multiple accesses. The basic operations of Linda are:

- *`rd(t)`* performs a non-destructive read from the tuplespace. If the required tuple `t` is not found, the invoking process will wait until an appropriate tuple is created by another process;
- *`in(t)`* is similar to *`rd(t)`*, except the read is destructive and the tuple is removed;
- *`out(t)`* writes the tuple `t` into the tuplespace;
- *`eval(exp)`* creates a new process which writes the tuple resulting from the evaluation of `exp` to tuplespace.

20 Usually Linda is referred as a "coordination language" meaning it is not a full programming language, rather then a set of coordination operations that can be added to any existing language. Linda primitives have been ported to many languages such as C, Prolog, Fortran, Smalltalk, and Java. However, for the sake of this discussion, Linda primitives are equivalent to library or toolkit functions.

A typical application is composed by a module which responds to an elementary event (such as “the tuple is now available in the space”), then elaborates it and returns the result to the tuplespace. For example, in a well known pattern for embarrassingly parallel applications, two modules are identified, the *master* and the *worker*:

- the master, usually deployed as a single instance, splits the job into discrete tasks and puts each task into the shared registry;
- the workers, running in many instances, are notified of the availability of the data;
- each worker retrieves the task and pushes the results back into the shared registry;
- the master or other workers are notified of the presence of the results in order to gather or to further elaborate them;
- workers are notified of work completion by meeting some conditions, or receiving a "poison pill" or by some other means, such as *sideband* communications.

The equivalent JavaSpaces API are the time-bounded methods `read()`, `take()` and `write()`.

Since most of the In-Memory Data Grids provide events notification concerning the storage, it could be a temptation to use them in a Linda-like fashion. Except for GigaSpaces XAP, which specifically supports multiple APIs including JavaSpaces ones, some important differences discourage to follow this path if a compatibility with different products is desired.

A first problem is the general design of IMDGs, in which an application can subscribe to receive events from any type of storage, regardless of whether it is partitioned, near, replicated, using read-through, write-through, write-behind, overflow to disk, and so on. Since the storage type should be transparent to the application, the most common architecture provides all the events, regardless of the storage topology, the number of nodes, and the node which operates the modifications will be delivered to the client application's listeners. This results in a considerable network traffic and high contention of the service thread dispatching events. Moreover, the APIs of many products do not have a primitive to determine if the event involves a entry owned by the local node. So all the agents listening for events will try to react to all changes in the cluster, not only to local ones.

Another critical point is more subtle: most of the implementations apply the same architecture, based on a service thread which invokes the listeners callbacks. This is the result of many years of experience in toolkit development which have to cope with multiple event source [Goe06][Ham04]. As a consequence, the operations in the callbacks cannot be blocking nor time-consuming. This requirement contrasts with both the Linda programming model, which assumes a transactional context, and the goal of performing heavy computations in

response to data changes. To prevent such problems, Coherence[21] and other implementations intercept at runtime calls to expensive methods in callbacks and raise an error if detect their presences.

Aside these architectural constraints, the programming model based on callbacks is hard to understand, error prone, and difficult to test for a non expert programmer. The problems in test-driven development applied to Swing-based GUI, which pose similar challenges, are well known and not completely solved yet.

A more sounding and straightforward pattern for a Java programmer is the model presented by the package `java.util.concurrent` and the interface `ExecutorService`, already implemented by some computational grid middlewares as explained in Section 2.2.1.

The key difference between a generic distributed `ExecutorService` and its implementation in a Data Grid is the ability to route the task to the node which holds the data to be processed. This is not easy to achieve, since the `ExecutorService` interface has no method for this. A possible workaround is implemented by Hazelcast [haz], in which the service (a singleton [GHJV94]) executes a submitted `Callable` object[22] on a node picked up at random.

```
Import com.hazelcast.core.DistributedTask;
// ...

public class Task implements Callable<T>, Serializable {
     public T call() {
     // . . .
}

// . . .

ExecutorService executorService = Hazelcast.getExecutorService();
IMap<Object, Row> map = Hazelcast.getMap("mymap");
for (Object key : map.keySet()) {
    DistributedTask<Row> ft = new DistributedTask<Row>(new
              Task<Row>() { ... }, key);
    executorService.execute(ft);
}
```

Listing 11: A submission of tasks to the distributed `ExecutorService` *in Hazelcast, each task will run on the same host which holds the corresponding* `key`

But, if the object extends the class `com.hazelcast.core.DistributedTask`, it might specify more options, such as the execution on a specific member of the cluster, or the paral-

21 http://coherence.oracle.com/display/COH35/Constraints+on+Re-entrant+Calls (last accessed 2 Feb 2010)

22 In this and other distributed implementations, the object must also implement the `java.io.Serializable` interface, causing the impossibility to use anonymous classes.

lel execution in multiple instances on all or on a subset of the members, or on the member hosting the data associated with a specific key. The code fragment in Listing 11 shows the submission of multiple tasks, where each task will be executed asynchronously on the host holding the master copy of `Row` data corresponding to the associated `key`.A similar result can be obtained using annotation. In GigaSpaces XAP, the task could define a method returning the key to be used as the routing value just marking the method with the `@SpaceRouting` annotation.

Other Data Grids implement similar but incompatible mechanisms, perhaps more flexible, to the executor service. For example, GemStone GemFire [gsg] has a data-aware execution service called `FunctionService` which can execute data-dependent tasks implementing the interface `Function`, extending `Serializable`, with the main method *`execute()`*. While `Function` has little difference from the classic `j.u.c.RunnableFuture`, the execution service accepts hints indicating the task is dependent on a key, a key region, a set of servers, or other characteristics, as shown in Listing 12.

```
Region clientRegion; // Region is the equivalente of Map or Cache
Set keySet = new HashSet();
keySet.add("key1");
keySet.add("key2");
Function myFunction = new Function() {
     // ...
}
Serializable args //...

ResultCollector rc = FunctionService.onRegion(clientRegion)
          .withArgs(args)
          .withFilter(keySet)
          .execute(myFunction.getId());

// Do something ...

// Retrieving the result
Serializable functionResult =  rc.getResult();
```

Listing 12: The GemFire execution service `FunctionService`*: the task* `myFunction` *is executed on the nodes owning the keys* `key1` *and* `key2`.

It is worth noting that, in case of Data Grids not including an own executor service, it might be possible to integrate them with a computational grid toolkit. For instance, GridGain shares many of the features of Data Grids we are interested, such as a P2P architecture, and it allows tasks allocation and checkpointing strategies to be plugged in. So it is easy integrate GridGain with a Data Grid in which the support for distributed execution lacks or does not fit the requirements, and handle the pair of applications as a single package.

2.3.3 The MapReduce support

In recent years, a new paradigm for huge date set management is becoming mainstream: the computation is delegated to the system storing the data, instead of moving the data to be processed from system to system. A prototype of this paradigm is the MapReduce technique developed at Google [DG04] for the processing of large files across large, but unreliable, clusters of computers.

Google uses a very large infrastructure that stores hundreds of Tbyte in thousands of computer files which are mostly read and infrequently updated. This scenario asks for a programming model which, at the same time, does not move data around and offers to the programmers an error-proof framework to take advantage of any exploitable parallelism.

The MapReduce is a such a simplified parallel programming model. It supplies a skeleton based on the `map()` function and the `reduce()` function. The `map()` function accepts a list of keys and associated values, and then produces an intermediate set of keys and values. The `reduce()` function combines these intermediate values into a final result. The programmer has to implement these two functions only, while the framework takes care of distributing them and collecting the results. As illustrated in Figure 4a), the master node decides how the file will be partitioned and allocated. Chunks of the file and metadata attributes are then sent from the user to the runtime system on the first chunk server, and then pipelined throughout the chain to distribute the replicas. The master then splits the tasks accordingly (Figure 4b), and sends the `map()` task to chunk servers, aiming to keep the computations as close to the stored data as possible. The master also assigns the `reduce()` task to one or more chunk servers. The chunk servers with intermediate output from `map()` ready to be processed by `reduce()` functions (Figure 4c), send their outputs to the appropriate chunk servers, which have been assigned the reduce tasks.

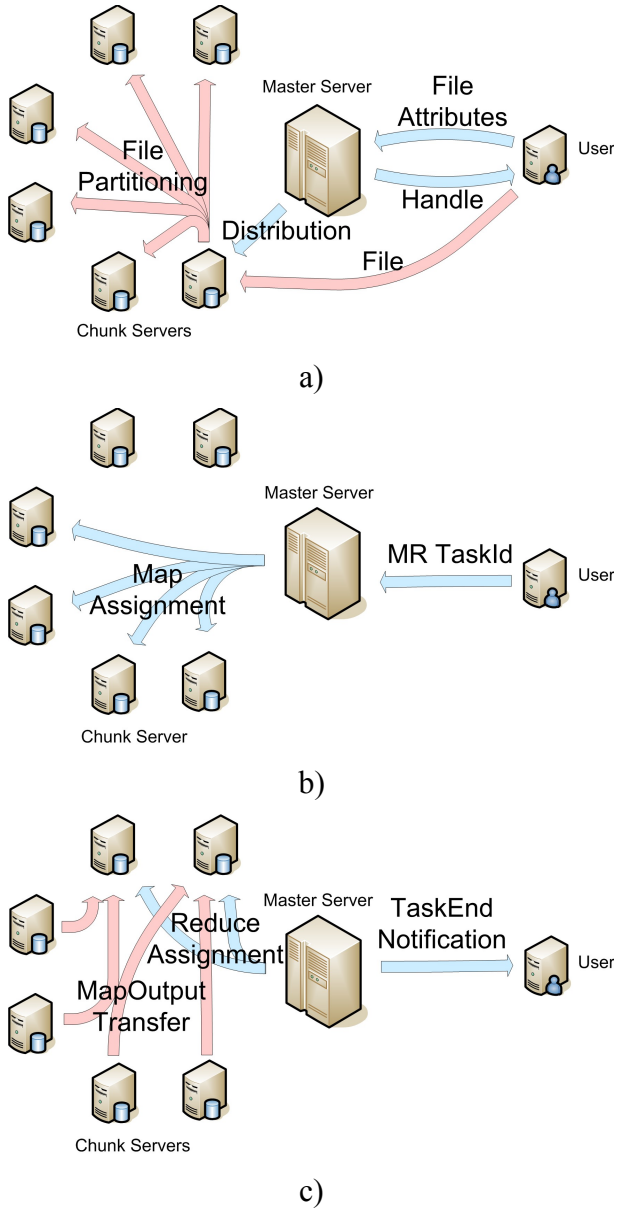


Figure 4: The MapReduce programming model (source [CSGA07])

In some cases, the intermediate keys produced by each map task can have significant repetition. It is the case, in Data Grids, in which the each node elaborate the data entry-by-entry and the `map()` outputs a list of result for each entry. If the `reduce()` function is commutative and associative, the user can specify an optional `combine()` function that does merging of partial results on the same node which performs the `map()` task, before sent them over the network. Typically, both the `combine()` and the `reduce()` functions are implemented by the same code. A UML representation of MapReduce activation sequence is shown in Figure 5.

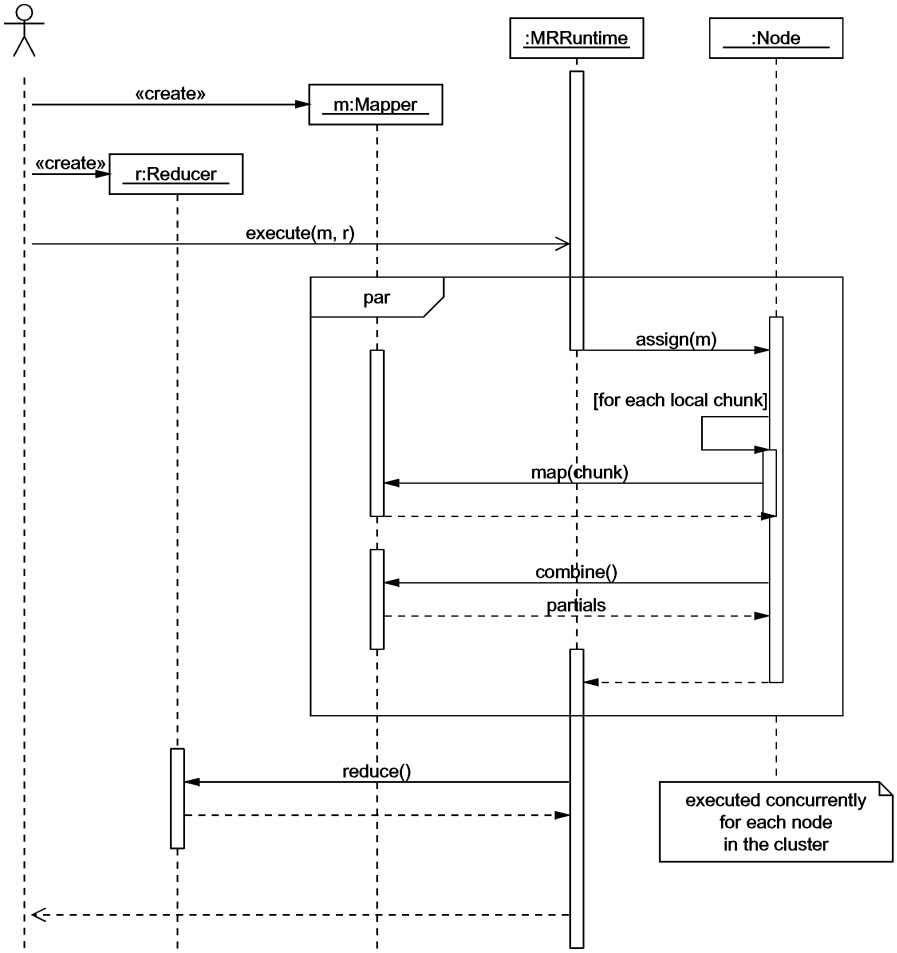


Figure 5: A sequence diagram for MapReduce, deduced from Hadoop documentation and code

The MapReduce approach has been tested in machine learning and data mining applications. Many different algorithms have been adapted to a MapReduce framework such as weighted linear regression, K-means, Naive Bayes, linear Support Vector Machines, independent component analysis, logistic regression, gaussian discriminant analysis, or Probabilistic Neural Network classifier [CKLY+06][CSGA07].

This model, characterized by moving the task to the store node instead of moving the data to the processing node, finds in full Java grid an ideal environment. In fact, it is quite natural for a Java programmer to think to objects with both the nature of data and tasks. And, since data can move around, even a task can be moved and executed on machines different from the one that instantiated it.

As already suggested, Data Grids have a strict affinity with the MapReduce programming model. Not surprisingly, many of them support that strategy in some flavour. Ideally, to achieve a linear scalability, a task running on a node should operate only on the partition owned by that node and all the operations should be executed concurrently across nodes and

partitions. Most versatile products can target the task execution on a single cluster node, in parallel on a subset of nodes, or in parallel on all members of the cluster. Advanced implementations also allow the programmer to specify object relationships concerning transactions boundary, such as a master/details relation, having the Data Grid co-located all related data to a single node, if possible. In such a way, the task running on the node likely access only to local data and concurrency locks traversing the network will be avoided.

Unfortunately, being both Data Grids and MapReduce young technologies, there is not a consolidate API model and MapReduce-like support in Java frameworks seems one of the most confusing and not-yet-canonized programming model. Here follows a sample of different APIs. As an instance, in Oracle Coherence the task executor is the cache itself (class `NamedCache`), which implements the `InvocableMap` interface (note the hesitation between the terms *map* and *cache*). To execute a task (*agent*, in Coherence idiom) on the grid node that owns the data, the cache itself exposes the method *`invoke()`*, with various overloaded form:

```
Object result = map.invoke(key, agent);
```

From the client perspective, the invocation is synchronous and the client must wait for the result. Coherence will determine the location where to execute the agent according to the configuration for the data topology, move the agent there, execute it (automatically handling concurrency control for the item while executing the agent), backup the modifications if any, and return a result. It is possible to target the task to a key collection, enumerated or selected by a query. If the task implements the interface `ParallelAwareAggregator`, described in Listing 13, it signals it is explicitly capable of being run in parallel in a distributed environment.

```
public interface InvocableMap.ParallelAwareAggregator {

    Object aggregate(Set setEntries);

    EntryAggregator getParallelAggregator();

    Object aggregateResults(Collection collResults);
}
```

Listing 13: Coherence `ParallelAwareAggregator` *subinterface*

A `ParallelAwareAggregator` operates as described in the Figure 6: the *`aggregate()`* method is invoked in each server passing the set of locally owned entries. Then, once the partial results from each server have been collected, the runtime obtains the combiner of partial results from *`getParallelAggregator()`* and uses this object (usually, implemented by the same task) to combine partial results with method *`aggregateResults()`*. Co-

herence API does not use generics and often returns `Object`, so it is very easy to be confused from these operations.

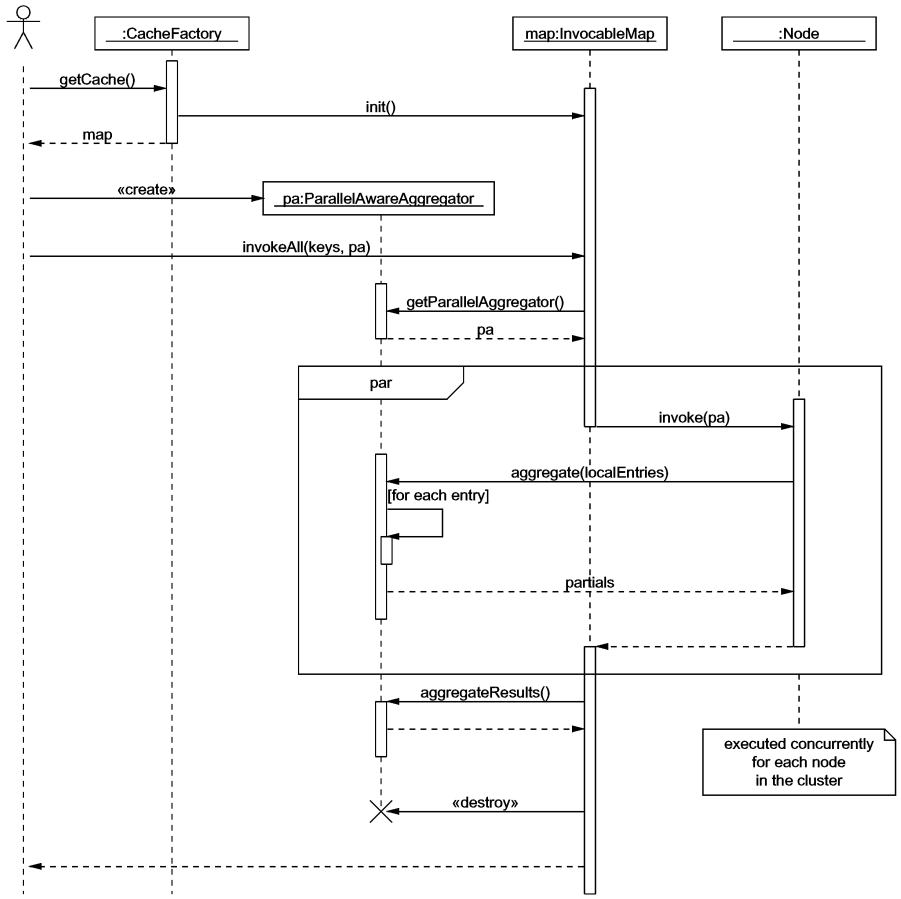


Figure 6: The sequence diagram for `ParallelAwareAggregator`*, deduced from Coherence documentation*

The same role in GigaSpaces XAP is played by the `DistributedTask` interface, shown in Listing 14. It expose two function, the mapper *`execute()`* function and the reducer *`reduce()`*. The mapper has not arguments and the data must be read directly from the space or be injected by the runtime. The task is submitted to the space and it is executed asynchronously using a `Future`, and the target keys must be enumerated:

```
AsyncFuture<Long> future = gigaSpace.execute(new MyTask(), 1, 4, 6, 7);
long result = future.get();
```

The sequence is illustrated in Figure 7.

```
public interface DistributedTask<T extends Serializable,R> {

     T execute() throws Exception;

     R reduce(List<AsyncResult<T>> results) throws Exception;

}
```

Listing 14: GigaSpaces XAP DistributedTask interface

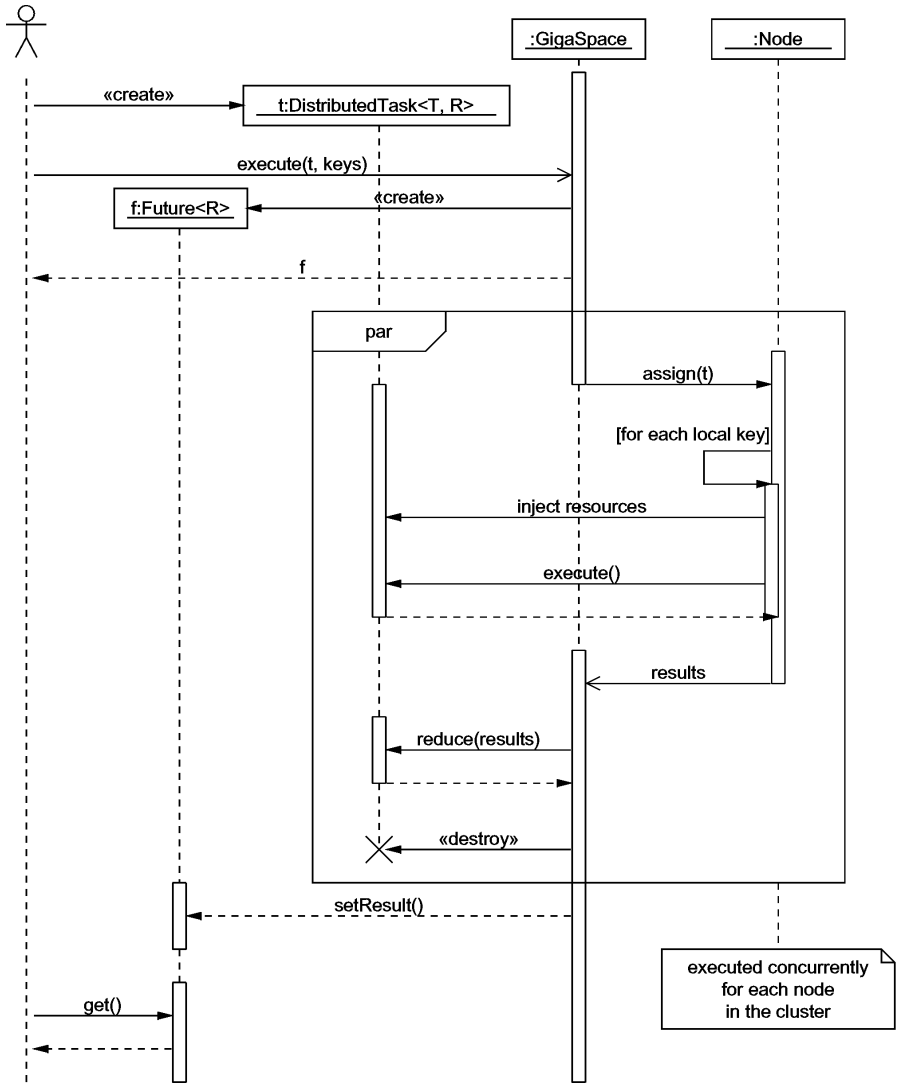


Figure 7: The sequence diagram for Gigaspaces XAP, deduced from the documentation

Other IMDGs have more different architectures, such as IBM WebSphere eXtreme Scale [ibm], with distinct synchronous mapper and reducer. It is quite evident that the MapReduce model is not well well-established yet, and any vendor has design a incompatible interface.

2.3.4 Facing the CAP theorem

From this brief introduction, it seems that the In-Memory Data Grid technology could solve all the problems distributed computing has faced in the last decades. This is obviously not true: the advantages of Data Grids have to cope with other weakness. It is well known that distributed computing cannot avoid trade-offs [WWWK94], and in 2000 Eric Brewer postulated the existence of uncircumventable limits in his CAP (*Consistency, Availability, and Partition*) conjecture [Bre00]. According to this conjecture, it is impossible for a distributed system to have simultaneously:

- Strong Consistency: all clients see the same view, even in presence of updates.
- High Availability: all clients can find some replica of the data, even in the presence of failures.
- Partition-tolerance: the system properties hold even when the system is partitioned.

As result, at any given time, at most two of these three desirable properties can be achieved.

The conjecture has been formally proven two years later by Gilbert and Lynch [GL02] and it is now known as the CAP theorem.

More precisely, consistency means that a service is fully operating or not at all. Gilbert and Lynch use the term “atomic” instead of consistent in their proof, for coherence with the A and C meaning in ACID. Availability means that the service will answer in bounded time. Partition-tolerance means the system can tolerate lost messages between nodes. Gilbert and Lynch pointed out that no set of failures less than total network failure is allowed to cause the system to respond incorrectly.

It is important to notice that the CAP properties are characteristics of a specific architecture or infrastructure deployment, not of a software toolkit. For example, since its origin, the LDAP protocol [HKY93] provides the elements to many different fault tolerance mechanisms. Using the same server implementation (e.g. Sun Directory Server or OpenLDAP) it is possible to deploy a service which alternatively provides:

- strong consistency, using only one server, and thus sacrificing the tolerance to the network partitions;
- eventually consistency, using a master/replica setup, and thus losing the availability of the write service during a partition: replicated data will be reconciled if and when the network will return functional;
- no consistency, in a multi-master configuration, allowing both writing and reading even in case of network failure, at the risk of not-reconcilable states [Fer00].

Most of the Data Grids have been designed to offer a configurable degree of Consistency, Availability and Partition-tolerance mix. In pure P2P systems, such as in Coherence, Hazelcast, Infinispan, or GridGain in case of computing grid, the peer subsystem runs in the same JVM of the application subsystem. With this architecture, the "master" node, corresponding to the interactive user application, provides autonomously all the feature offered by the whole grid. As a result, within the limit of available resources, the user application can continue to submit tasks and obtaining results even in case of network failures.

The question we can pose is the following: "Which are the most desirable characteristics of a Desktop Grid-based architecture devoted to interactive and/or situational data mining?" Obviously, being interactive, availability is unavoidable. The most frequent failure in a Desktop Grid is node outage, due to local user activity, while a data mining computation infrastructure probably relies on a RDBMS or reliable file system to store the data in their original form, before any manipulation. So, it seems reasonable that Partition-tolerance is more important than data consistency to survive to node outage, since the data can be reload and reprocessed from the original sources.

3 Comparison between different IMDG implementations

What are the desiderata features required to a Data Grid design for hot-deployment in LAN environments for data mining and machine learning applications?

First of all, the Data Grid must work. This might seems quite obvious and, indeed, all the available products “do something” but there are many subtle details that can cause a product to be useless or too unstable to be used. Other requirements are less trivial; the Data Grid design:

- should expose an easy and familiar API to the programmers;
- should work fast, at least as fast as a traditional database, especially in high concurrent situations;
- should present the resources of connected hosts, the memory in particular, as an aggregate capable to deal with large data collections;
- it should provide built-in checkpoints and persistence mechanisms.

All these features will be described in turn in the next sections.

3.1 In-Memory Data Grids platforms

Several software products, self-defining In-Memory Data Grid or having similar capabilities, are available and a survey can be found in the Appendix. Nevertheless, Data Grid is a novel and little explored architecture, and the lack of literature has required a preliminary assessment to understand if this technology is suitable for distributed machine learning scenarios. None of the following results should be considered as a benchmark, neither the goal was to determine the "best" product.

Some of the well known IMDGs and distributed cache platforms have been tested in a single node configuration to analyse basic characteristics. In the remaining part of this section, we will discuss the results obtained on the following implementations[23]:

23 Other applications have been subjected to a dry analysis, but not effectively tested. GemStone GemFire is not included in this work because the license of the evaluation version includes a non-disclosure clause. IBM WebSphere eXtreme Scale is available in a free evaluation edition but it has some limitation in func-

- Oracle Coherence grid edition release 3.5.2 dated 19 October 2009
- Terracotta Ehcache release 1.6.2 dated 23 August 2009, then some tests have been reproduced with the newer release 1.7.1 dated 30 November 2009
- Hazelcast release 1.7.1 dated 16 October 2009, then, due to an issue interfering with tests, repeated with a snapshot of release 1.8
- JBoss Infinispan release 4.0 RC3 dated 11 December 2009
- Apache Jakarta JCS release 1.3 dated 30 May 2007

A dummy data grid mock-up, based on a plain `java.util.Hashtable`, exposing the same façade interface, and storing regular POJO (*Plain Old Java Object*), was used as baseline and all the values are reported to it, using its performance on same operation as measure unit.

It is highly probable that the tested products could be fine-tuned to offer better performances in specific deployments, but the goal of this test was to produce a baseline for further investigations. It is also unlikely that the final user will explore the configuration options in details and the perspective of an application distributed in a Desktop Grid discourages platform-dependent configurations.

3.1.1 Load capacity test on a single node

The first test consists in a progressive load of data until the memory exhaustion causes the crash of the application. Data have been loaded in chunks of 512 entries of 128 `Integer` objects, the heap of JVM being limited to 128 Mbyte. The testing platform was an Athlon X2 5200, 6 Gbyte of RAM, with Microsoft Windows Vista 64 bit OS and SUN 64 bit 1.6.11 JVM. Results are shown in Figures from 8 to 13 and summarized in Table 1.

tionality which prevent long term usage. GigaSpaces eXtreme Application Platform (XAP) does not fit the usage scenario because is not based on pure P2P architecture and uses a more traditional controller/agents one. In order to run a grid node (*GigaSpaces Container* GSC), a discovery service (*Lookup Service* LUS) in the network and an agent (*GigaSpaces Manager* GSM) running on each host are required. This architecture makes difficult to create auto-deploying data grids. Moreover, XAP is released under a commercial license model which does not match with the spreading of components over a network spanning through different organizations. Cacheonix has been disregarded because the per-processor license model does not cope with the employ in a Desktop Grid fashion architecture which should collect as many PCs as possible.

	Average start-tup time	Stored entries	Average store time for entry	Average single entry read time
`java.util.Hashtable`	1	24,064	1	1
Oracle Coherence 3.5.2	382.76	55,296-55,808	6.9	3
Terracotta Ehcache 1.7.1	31.47	24,064-46,080	3.13	11.1
Hazelcast 1.8	223.8	35,328-36,864	5.21	4
JBoss Infinispan 4.0 RC3	22.66	24,064-46,080	2.23	-
Apache Jakarta JCS 1.3	0.9	22,528-23,040	1.51	-

Table 1: Data Grid results summary. All time measures use the performance of `Hashtable` *in the same as measure unit.*

Each test has been executed at least three times in different periods to compensate possible background activities in the host. In the graphs, the x-axis represents the data chunks allocated in sequence. The blue lines represent the total allocated heap, the red lines the time elapsed to allocate each block.

As clearly shown in Figure 8, the `java.util.Hashtable` had such a constant behaviour, so that the three lines almost coincide. For this reason, this implementation has been used as reference and all other time measures have been normalized to that.

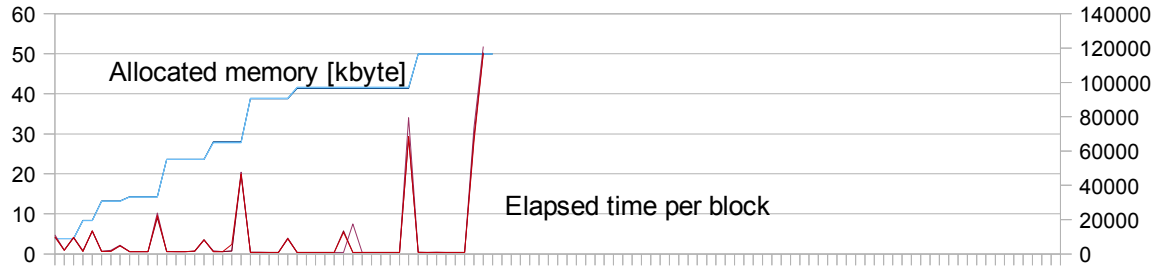


Figure 8: Single node allocation test, local `Hashtable`*. The x-axis represents the chunks allocated. The blue lines represent the total allocated memory, the red lines the time elapsed to allocate each block. This test has been executed three times but the lines almost coincide.*

Surprisingly, the storage capacity of many implementations (Coherence, Ehcache and Hazelcast) outperforms the plain local storage, due to a very efficient serialization of the objects in a compressed form. Obviously, since nothing comes for free, and write operations in Data Grids are from 2 to 7 times slower than in `Hashtable`.

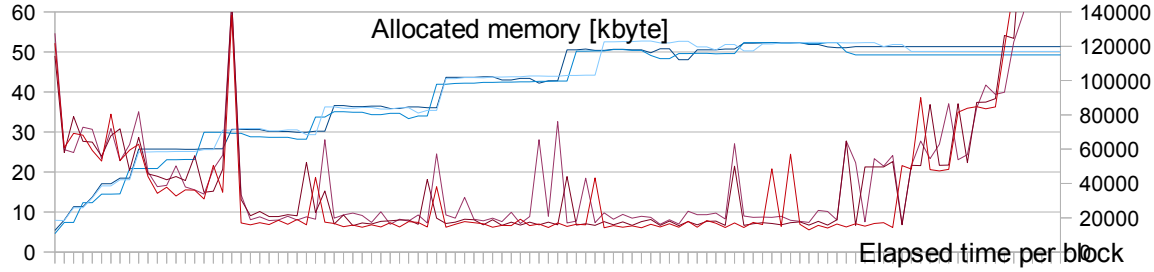


Figure 9: Single node allocation test, Oracle Coherence 3.5.2

Ehcache has a cache-oriented design, rather than being cluster-oriented, and applies by default a write-behind strategy. At write time, it stores the entry as-is, showing performances near to the plain `Hashtable`. Then, when it reaches a configurable memory limit, it enqueues the entries candidate to compression or overflow to the disk in a spool, according to the current policy (LRU or LFU). A background task performs these operation effectively. If the spool capacity is reached or the background task cannot operate at the same speed at which the entries are spooled, e.g. because it writes to the disk, exceeding entries are discharged from the memory and lost. This is an acceptable behaviour for a cache, in which a missed entry should be recoverable from the original source. Unfortunately, when setting Ehcache to use the *perpetual* eviction policy, in which entries are never discharged, the asynchronism between the operations at the user and background levels result in an erratic behaviour, in both storage capacity and elapsed time, as shown in Figure 10. It can be observed that the load lines interrupt at different steps, ranging from 24k to 46k chunks. This is the only implementation showing a such a behaviour, being the capacity variations in other IMDG less than 10%.

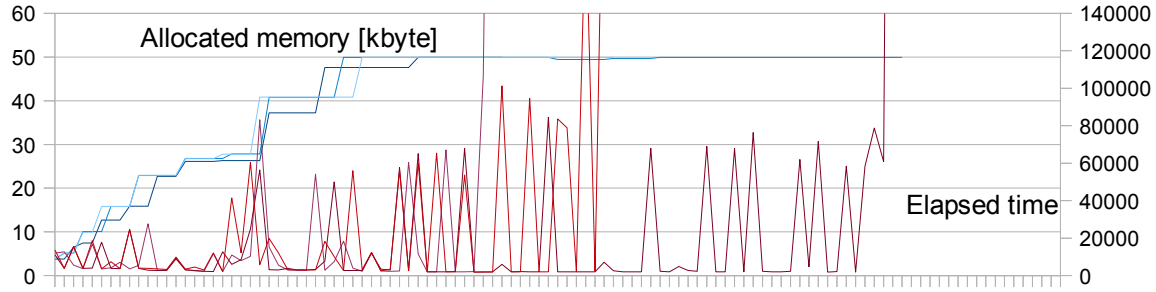


Figure 10: Single node allocation test, Ehcache 1.7.1

The Hazelcast library has demonstrated some problems to detect the `OutOfMemoryError` condition, sometimes because it occurs in the service thread and it does not propagate to the main thread; as a result, the application hangs instead of crashing[24]. A similar problem seems sporadically affecting Infinispan too, but it was tested in a not yet stable release.

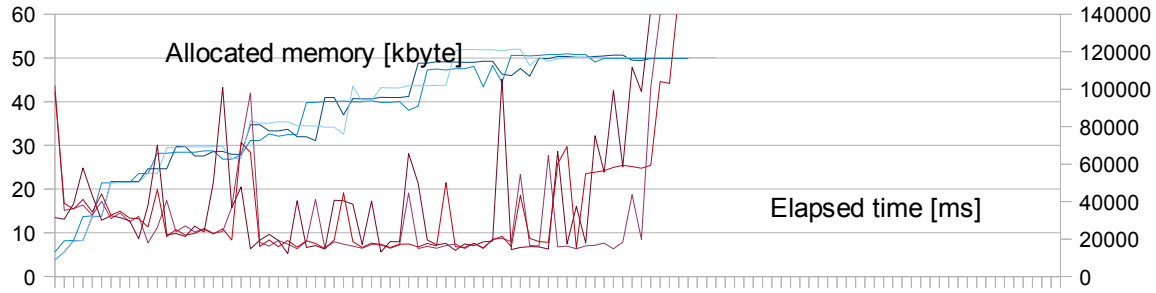


Figure 11: Single node allocation test, Hazelcast 1.8

24 The problem has been submitted to the maintainer of the project.

Infinispan was tested at the end of 2009 in a pre-release stage and it shown some troubles not easy to fix, without offering evident improvements. Nevertheless, the roadmap[25] of the project suggests to keep an eye on it, since JBoss plans to implement the state-of-the-art features set of Data Grids.

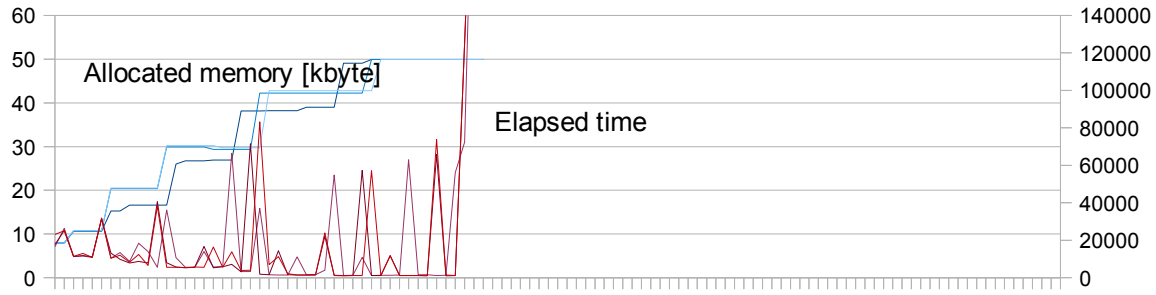


Figure 12: Single node allocation test, Infinispan 4 RC3

JCS has a design similar to Ehcache, but the documentation is not kept updated and it was very difficult to configure it correctly.

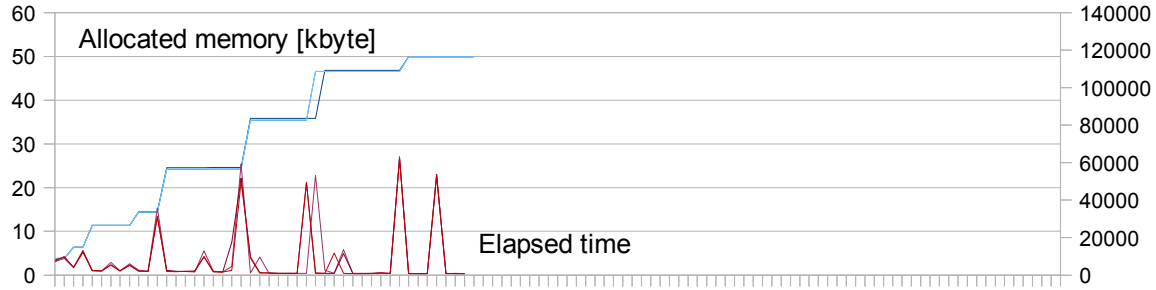


Figure 13: Single node allocation test, JCS 1.3

After these preliminary tests, Infinispan and JCS were no further investigated. Indeed, JCS is not actively maintained from 2007 and its performances, roughly equal to the plain `Hashtable`, did not warrant the use. On the other hand, the beta release of Infinispan was too unstable for a real use.

In order to check if contextual settings have influenced the results, some tests have been repeated under different conditions: using a Bea JRockit 32 bit JVM, the overall behaviour was the same but, as expected, each instance could accommodate more data (for example, Coherence allocated over 144 blocks instead of about 110). Moreover, using different object types, such as small `String`, seemed of no influence. Instead, very large objects could exhausted the memory much faster than as expected compared to the equivalent amount of small objects. This issue need a deeper investigation in case of systems designed to deal with large data such as image files.

25 http://community.jboss.org/wiki/infinispanroadmap (last accessed 15 Jan 2010)

3.1.2 Entry retrieval test

An entry retrieval and read performance test has been performed on the same data used in the previous experiment. Read of entries has been tested both in the same order they have been stored and in a random one. The results are shown from Figure 14 to 16. The x-axis represents the time elapsed to read all the entry in a chunck.

Oracle Coherence has shown a very homogeneous behaviour. The spikes, visible in Figure 14, are probably due to the Garbage Collector activity.

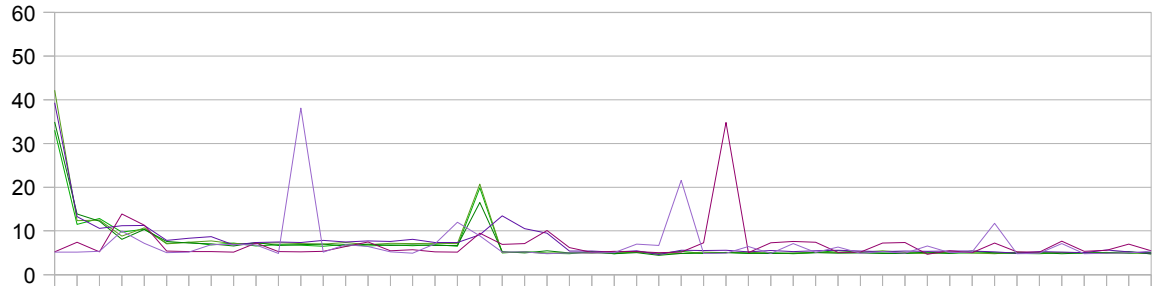


Figure 14: Coherence read test. Green lines show sequential read time per entry, purple lines show random reads. Time measures are scaled to the `Hashtable` *performance.*

Ehcache has shown less predictability and worst performances, probably for the same reason detected in the write test.

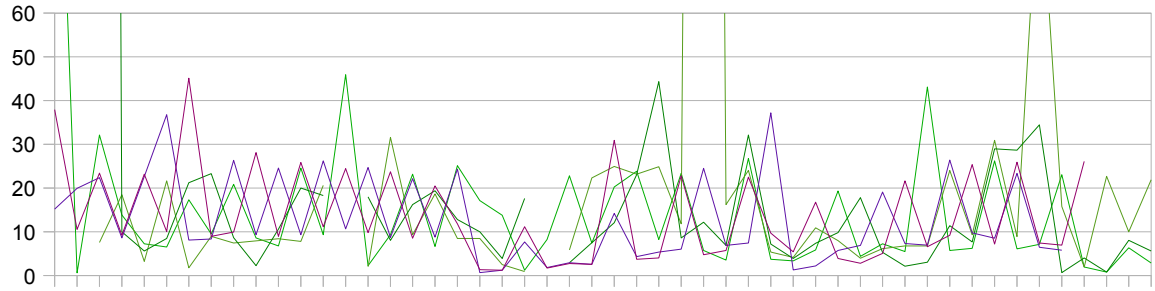


Figure 15: Ehcache read test

Finally, the behaviour of Hazelcast was a little chaotic, but affordable and time-bound.

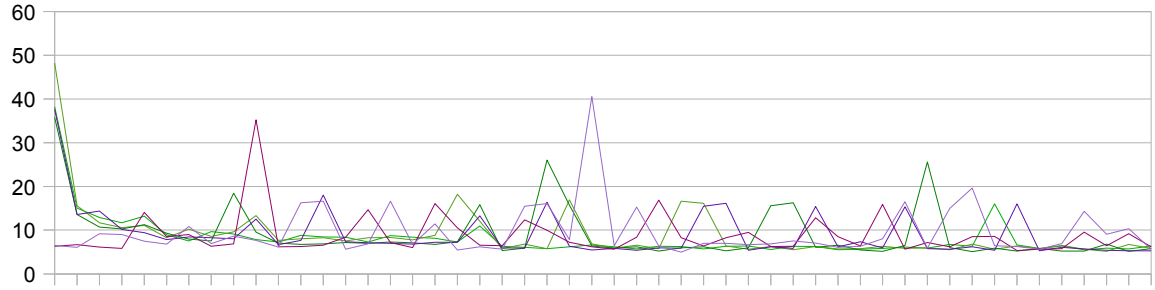


Figure 16: Hazelcast read test

In all the platforms, accessing the entries in a random order seems to have a small, but noticeable, impact on performances.

3.2 Memory allocation effectiveness

To investigate the behaviour of In-Memory Data Grid for memory allocation, other tests have been performed to verify the capability of effective memory usage.

The write tests of Section 3.1, have been repeated reserving 4 Gbyte for the heap. Results are shown in Figures 17 to 19. Test ran in the same conditions of above.

Oracle Coherence, in Figure 17, seems unable to exploit more than 1.5 Gbyte of memory[26]. However, thanks to a better compression, Coherence still stores more objects than the other implementations.

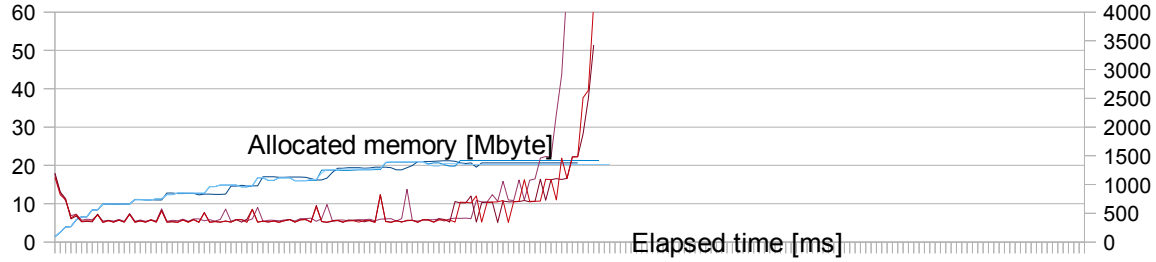


Figure 17: Coherence 4 Gbyte allocation test. In the x-axis the chunks allocated. The blue lines represent the total allocated memory, the red lines the time elapsed to allocate each block.Time scale is the same of the Figure 8, memory scale is in Mbyte instead of kbyte.

Ehcache confirms a less efficient memory usage, using around 3 Gbyte of RAM to allocate less than half of the objects allocated by Hazelcast. The only product capable to scale up to 4 Gbyte is Hazelcast, as shown in Figure 19.

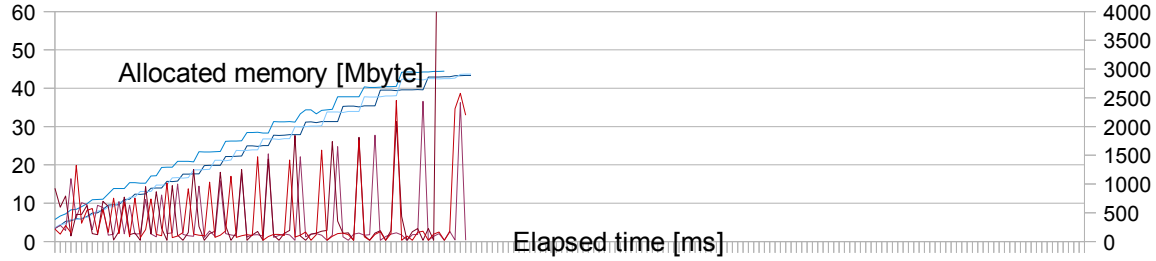


Figure 18: Ehcache 4 Gbyte allocation test.

In all the experiments, the interference of the Garbage Collector (GC) was quite evident when the memory allocation grown, with spikes in allocation time lines (less visible in Coherence due to the less amount of used memory). The GC in most of the Java Virtual Ma-

26 After a conversation with the Oracle Coherence technical support, the test has been repeated on a different platform obtaining the correct Coherence behaviour. The problem is under investigation.

chines has greatly improved over the years. Nevertheless, applications using large heaps might still manifest a "Stop-the-World" behaviour and it is not uncommon that a garbage collection phase requires tens of seconds to complete. These delays might be acceptable if the latency is not a big deal. Unfortunately, the Data Grids under examination have a P2P network layer which marks as suspect a peer not responding to the *heartbeat* notification. A long wait for the heartbeat produces a rearrangement of data partitions, resulting in further delays. In fact, from the cluster's perspective, the node that does not respond for long time (ranging from 5 seconds for Ehcache to 300 seconds for Hazelcast, using default values) might be dead and the cluster is allowed to exclude it from the configuration. When the node resumes after the long GC phase, it has to re-join the cluster and this requires a lot of new work, further worsening the response time. Instead, the “split brain syndrome”, where two or more instances attempt to control the cluster is unlikely to occur. IMDGs are specifically designed to cope with this problem and, in the assumed scenario, the client interface is defined as the “authoritative” node.

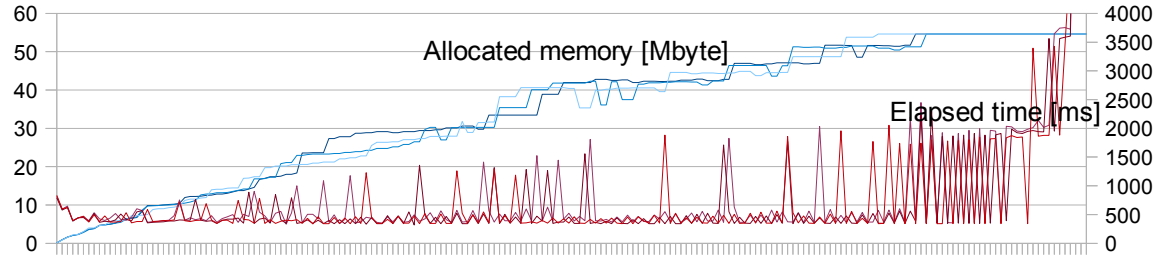


Figure 19: Hazelcast 4 Gbyte allocation test.

Based on similar observations, there are suggestions [Ime08] that applications based on P2P Data Grids could benefit from splitting the RAM between a set of JVMs running in the same host, instead of reserving all the memory for a single JVM, allocating a Virtual Machine per CPU core. In this way, it is supposed the effect of the GC might be reduced because the heap size of the individual JVMs is smaller and the GC process is spread across the cluster, reducing the impact on the individual machine.

3.3 Clustered operations

At this point we know that some Data Grid implementations work as declared, at least in isolation, but since they are designed for distributed and high concurrent applications, we need to analyse their behaviour in a clustered configuration.

The capability of memory aggregation using different nodes/heap combinations has been explored in this test. Since this architecture supposes a setup based on data partitions, and

Ehcache does not have this built-in capability[27], this platform has not been tested in this configuration. To avoid the overhead caused by the network layer, but offering enough cores to simulate a small cluster, the test has been performed using an Amazon EC2 High-CPU Extra Large Instance with 8 cores and 7 GB of memory, running Linux Fedora 64-bit and Sun 64 bit JDK 6 Java Virtual Machine. Results are shown in Figures 20 and 21. On the x-axis appears the cluster size, in the y-axis the storage capacity on the left graph and the elapsed time on the right graph, measured as in the previous load test.

At a first glance a write test does not involve the Garbage Collection, but in these architectures objects instantiated by the application are copied in a serialized form in the Grid and then discharged, thus becoming eligible for the GC. Hence, this test also verifies the correctness of the hypothesis about splitting memory among different JVMs to limit GC overhead.

Some products claim their partitioned memory allocation can exploit the aggregation of the memory of all nodes in a single storage space. In this case, assuming all the n nodes, including the user own PC, have h_m heap memory available, the expected total accessible storage is $h_{available} = n\ h_m / k$, where k is the number of configured backup copies (usually 1).

Coherence shows clear benefits in memory aggregation, as visible in Figure 20. In a cluster with more than two nodes, the storage capability grows as expected in a roughly linear way. Moreover, the aggregate memory is close to the whole memory: 8 nodes with 512 Mbyte of heap, 4 with 1024 Mbyte, or 2 with 2048 Mbyte, can allocate around the same amount of

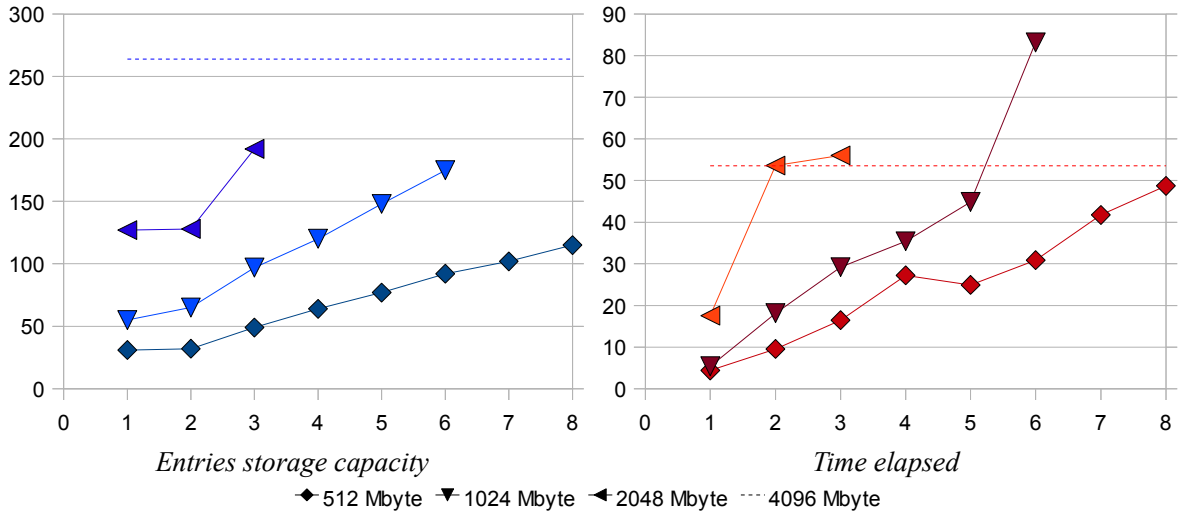


Figure 20: Oracle Coherence: storage capacity and time by nodes in the cluster. Each test has been managed to not exceed the host hardware, using up to 8 instances for 8 cores and no more than 6 Gbyte of RAM. The 4 Gbyte line is the capacity obtained allocating the memory in a one node configuration.

27 The client application could implement the strategy by itself, as explained in http://ehcache.org/EhcacheUserGuide.html#id.s32.3 (last accessed 10 Jan 2010). Unfortunately, this strategy lacks of the automatic relocation of block nor offers an obvious way to handle nodes dynamically joining and leaving the cluster.

objects. However, using several JVMs in a single machine resulted in less storage capability and high overhead in writing time, without any visible benefit.

From the point of view of memory effectiveness, Hazelcast simply ignores the clustering. Elapsed time grows, as shown in Figure 21, but not in an evident way such as in Coherence, and the storage capability does not change.

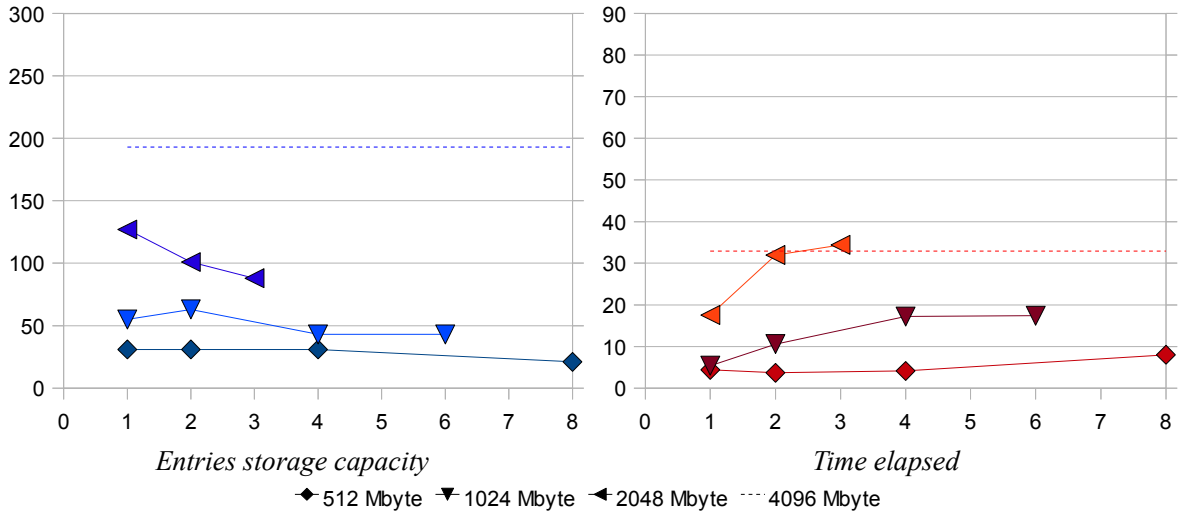


Figure 21: Hazelcast: storage capacity and time by nodes in the cluster

After these results, the proposal of using multiple JVM on the same host to minimize the impact of Garbage Collector seems not well funded. As an example, shown in Figure 22, the behaviour of one JVM allocating 4 Gbyte of heap or four instances with 1 Gbyte each are not clearly distinguished

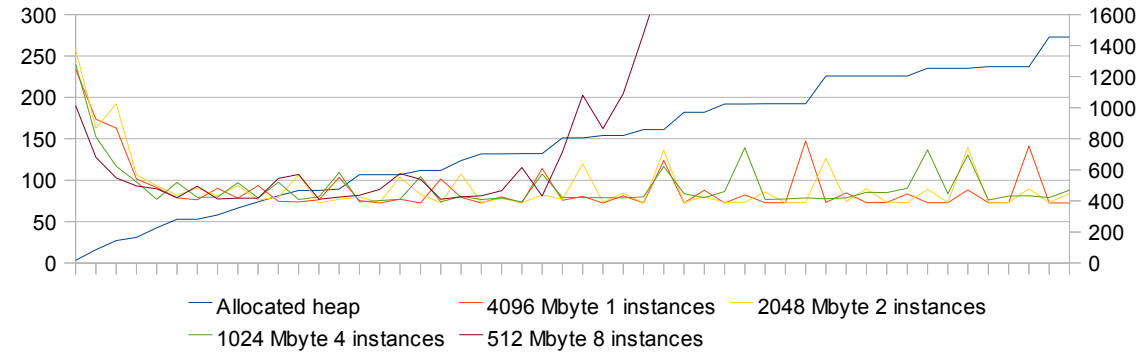


Figure 22: Oracle Coherence, multiple instance on the same host, replicated configuration. Configuration with 512 Mbyte of heap exhausted the memory early, the crash point of other configurations is outside the graph.

Another sensitive point is the operation throughput. We tested two scenarios, a mostly-read and a balanced-write, in a cluster configuration. In each case, we provided to not saturate the system, setting a limit for each node heap in a way the total heap being largely lower

than physical RAM, and running from two up to six nodes to ensure at most two free cores for handling background threads. Again the performances of the dummy local `Hashtable` have been used as measure unit.

In the mostly-read scenario, read operations are approximatively ten times more than the write operations. Differently from the results reported in Section 3.1.1, in this environment the performances of the `Hashtable` are less consistent, with a standard deviation of around 14%. This is caused by the virtualized environment, but it is not considered a critical point for the test, since the full run for the `Hashtable` elapsed few seconds, while in Data Grids spans over several minutes. The results are summarized in Table 2 and in Figure 25.

		2 nodes	3 nodes	4 nodes	5 nodes	6 nodes
Coherence	Mixed op/t	0,0286	0,0210	-	0,0148	0,0105
	Write op/t	0,0115	0,0099	-	0,0073	0,0053
	Read op/t	0,0339	0,0238	-	0,0166	0,0117
Hazelcast	Mixed op/t	0,0245	0,0170	0,0121	0,0116	0,0097
	Write op/t	0,0172	0,0094	0,0051	0,0043	0,0039
	Read op/t	0,0258	0,0187	0,0141	0,0145	0,0117

Table 2: Mean throughput for single node in clustered configuration..Values recorded for 4-node Coherence cluster present anomalies, but the test could not be reproduced on the same platform. 1 t unit is the equivalent performance of `Hashtable`

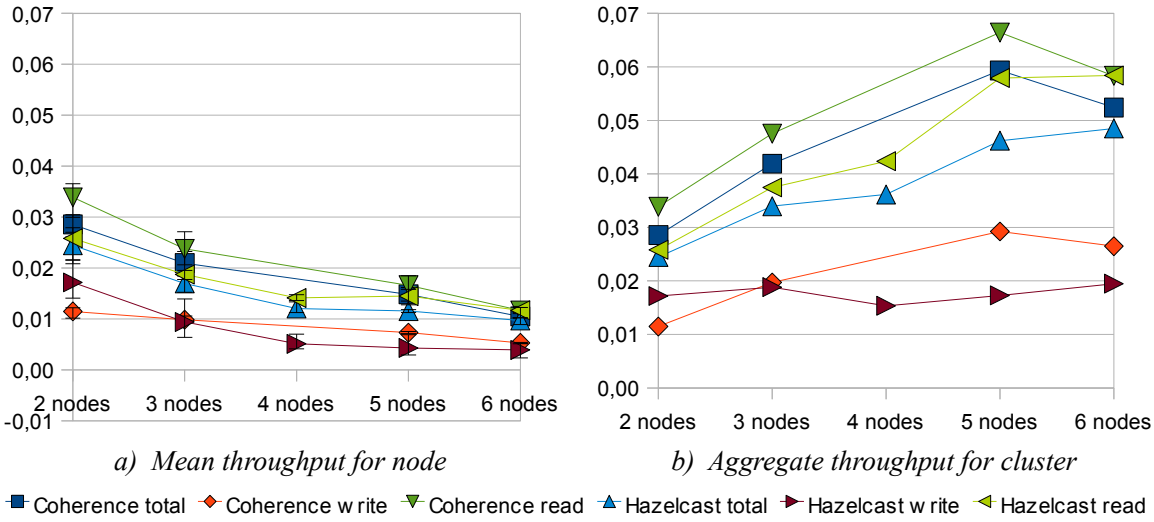


Figure 23: Throughput for node in cluster configuration. 1 is the throughput of Hashtable

The throughput per each node sensibly decreases with the cluster growth, but the aggregate throughput increases.

Both platforms benefit from a warm-up phase. Even when the cluster is started in advance, to let the P2P negotiation stabilize, it seems the population of an empty map requires a while to uniformly distribute the keys in the cluster and to initialize near caches, and performances improved after a little period, in the order of tens of seconds. This phenomenon should be taken into account for deployment on-the-fly, as an additional issue discouraging too short tasks. The warm-up phase is noticeable in Figure 24, referring to Coherence. Both green lines, representing the read throughput in entry per second of each node, stabilize at the middle of the graph. Hazelcast has a similar behaviour and thus it is not presented here.

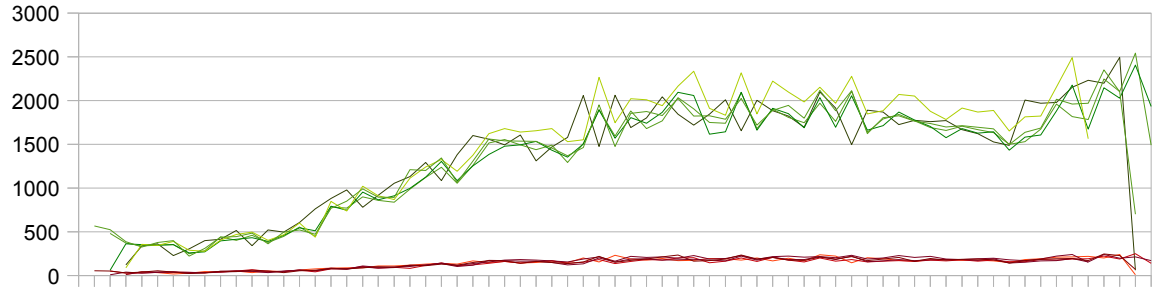


Figure 24: Oracle Coherence, 6-node cluster, interleaved read:write 1:10, throughput per second. Green lines represent the read throughput of each node, the red lines the write throughput. Entries are complex object with variable length array and collections as members..

The tests have been repeated with a balanced number of interleaved write and read operations. Also in this case, Coherence and Hazelcast had a similar behaviour which was not dissimilar from the previous test, only with a decrease of around 20% in the number of operations per second.

It is interesting to analyse the fault-tolerance of the cluster. Both products can easily handle the loss of one node at time. In Figure 25, each line shows one of five nodes in a Hazelcast cluster running the balanced read test, in which two nodes have been killed during execution. After a while, to allow the cluster reassigning the partitions, the work restarts and continues at the same speed.

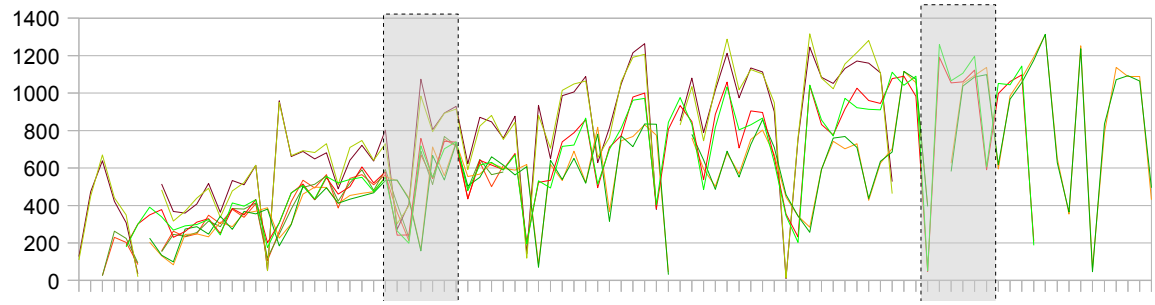


Figure 25: Cluster throughput per node , highlight rectangles show the loss of one node each

We also performed informal tests against a traditional RDBMS, in this case MySQL which has the reputation to be one of the fastest DB engine available. We found that the performances are strictly related to the O/R mapping applied. If the objects to be saved can be mapped on a single row of a table, MySQL can slightly outperform the best of the Data Grids we tested. But, as long as that the object includes a `Collection` field with variable length, which requires a normalization of the database and another table at least, the performances drop dramatically. We experienced ten times slower performances by just adding a variable length array of `double` to a class. Nevertheless, since MySQL and similar products cannot be deployed on-the-fly in a P2P network, this configuration has not been further investigated.

4 Experiments and results

The main goal of this work is to investigate if and how the emerging technology of In--Memory Data Grids, born in the community of business and web software, is easily exploitable by the developers of data massive applications in fields such as bioinformatics, data mining and machine learning.

In most cases, these applications are created by domain specialists, rather than by software engineers, who focus on their specific algorithms. The dissertation statement is that the Data Grids and the programming model they expose, wrapped by an abstraction layer, offer to a non-programmer specialists interfaces suitable to build distributed parallel applications with a minimal effort. In the trade-off between simplicity and performances, In-Memory Data Grids swap an easy-to-understand programming model and decent overall performances with a lack of fine grained control and sub-optimal resource usage. Nevertheless, they seems suitable to replace the current approach to distributed computation applied in many popular applications, which apply home-made solutions or have no distributed capability at all.

Known use cases for a Data Grid, explicitly advertised by the main vendors, are *caching*, in which frontend applications request data from a Data Grid rather than backend data sources, *analytics*, in which applications query the Data Grid, *transactions*, where the Data Grid acts as a transactional record system hosting both the data and the business logic, and *Complex Event Processing* (CEP) in which the system looks for sequences in a event stream trying to match a pattern and notifying "complex events" of interest. Classic applicative domains are market and reference data analysis, financial risk management, algorithmic trading, and fraud detection [ora08][gem05].

All these applications are designed for a scenario providing a stable server infrastructure and predeterminate tasks. Our goal is to exploit Data Grids as distributed datastore and checkpointing facility in volatile networks to satisfy the computational requirements of interactive data mining. To verify these hypothesis, several tests have been performed. To investigate specific data mining scenarios, we applied the proposal technology to a pre-existing application we have already used in the recent past, which performs the l_1l_2 regularization protocol on a traditional Desktop Grid architecture.

In this chapter, we provide a short background of the l_1l_2 distributed application and describe the testbed infrastructure. Then we provide an overview of the architecture of the Data Grid-aware version of l_1l_2 and show some performance results. Referring to the points discussed in Section 2.1.4, i.e. parallelization of data mining applications, the experiments apply both the decomposition of the data table in regions each assigned to different hosts, and the more classic technique of distributed parameter sweeps.

The experience coming from l_1l_2 is then applied to a general purpose data mining Workflow application in order to evaluate the advantage for programmer in using IMDGs coupled with MapReduce programming model.

4.1 Background

Distributed cross-validation and its use, such as the feature selection, is a well known topic in both the theoretical research and the proof-of-concept software. Production-grade support is instead less common and only few open source data mining applications have an implementation which is easy to use.

In order to evaluate the performance of In-Memory Data Grids in this class of tasks, we ported on this platform a framework we already used [BF09] to developed a distributed version of a program for the selection of relevant genes from DNA *microarray* data based on the l_1l_2 algorithm originally presented by [ZH05] and then studied and implemented by [DMDV+08][DMTV09].

Denoting with **X** the gene expressions matrix and with **Y** the vector of the classes labels, the l_1l_2 regularization aims to find β defined as:

$$\beta = argmin_{\beta}(\|\mathbf{Y} - \mathbf{X}\beta\|_2^2 + \lambda(\|\beta\|_1 + \epsilon\|\beta\|_2^2))$$

where the least square error is penalized with the l_1 and l_2 norm of the coefficient vector. The parameter ε in the functional is fixed a priori and governs the amount of correlation we wish to take into account. The features corresponding to nonzero values in the optimal coefficient vector are those selected as relevant. In practice, we use the selection protocol that combines the selection step described above with a Regularized Least Squares (*RLS*) classification phase.

$$min(\|\mathbf{Y} - \bar{\mathbf{X}}\beta\|_2^2 + \tau\|\beta\|_2^2)$$

where $\bar{\mathbf{X}}$ is the submatrix obtained by only using the columns of **X** corresponding to the variables selected in the first step. l_1l_2 translated in pseudocode is shown in Listing 15.

```
dataset = {Matrix X, Vector Y}

n = rowcount(X)  // n. Sample
d = columncount(X)  // features

Vector β
DoubleMatrix xt = x.transpose();

// Step size σ0
// if σ0 is not specified in input, it evaluates from dataset
σ0 = estimateSigma0(dataset)

tolerance = 0.01 // tolerance for stopping rule
kmax = 10000 // maximum number of iterations

// Element-wise right array division
Matrix XT = transpose(X) / (n * σ)

// initialize beta vector with RLS solution of
// ||Y – X * β ||2
Vector β0 = RLS(X,Y)  // initialize beta vector with RLS solution

// initialization
σ =  σ0 + μ
μs = μ / σ

// l1l2 algorithm
k = 0
repeat {
     Vector β = thresholding(β0 *(1 – μs) + XT * (Y – X * β0), ts);

} until (k < kmax && ∀ i = 1..n |β[i]-β0[i]| ≤ |β0[i]| * tolerance/(k+1) )
```

Listing 15: l_1l_2 pseudo code

One of the main challenge from the data analysis perspective, is to design an appropriate algorithms able to find the variables relevant to a given process, with good generalization properties on new data and avoiding the so-called *selection bias* [AM02]. In principle, the only way to assess predictive accuracy of a system is to classify a set of independent new cases. But in cases such as DNA microarray, obtaining new data is very expansive when not possible at all, so it is advisable to estimate the correct accuracy estimated from the data at hand using cross-validation methods.

In cross-validation methods, data are partitioned into two complementary subsets called *training set* and *testing set*, then using the training set to *learn* the parameters and the testing set to estimate the error on those parameters. Common cases of cross-validation split the sample into two partitions with a test:training ratio of 1/3:2/3, with three rotations of the

sets, or the 1:9, with ten rotations, also known as 10-folds cross validation. Usually such cross validations are used to estimate the errors on one, sometimes two, parameters.

The procedure applied to microarrays has quite different constrains. Microarrays presents about 54,000 features[28], are expensive and time-consuming to produce, and the genetic diseases under investigation are often, and fortunately, rare. Thus, usually such analysis can rely on few dozen of samples only. In such cases, the most widely adopted resampling procedure is the *leave-one-out* cross validation, where the number of partitions equals to the number of samples and the procedure is repeated for each sample. Moreover, the l_1l_2 algorithm requires to estimate two different parameters, the term τ which controls the l_1l_2 phase and term λ which controls the RLS, thus it requires two nested loop of cross-validation. As results, a single run of feature selection usually requires several thousands repetition of the learning/test cycle.

4.2 The classic approach

The original release of the l_1l_2 framework was implemented as a set of MATLAB [matl] scripts called `L1L2_TOOLBOX`. It runs on a single hi-end workstation, spending a time in the magnitude of weeks to perform a full analysis of a data set collected from 20-50 microarrays.

Cross-validation is an obviously parallel procedure, since it involves the execution of many independent tasks. We exploited the availability of ShareGrid infrastructure, described ahead, to develop a grid-enabled version on the `L1L2_TOOLBOX`. The refactoring mainly required to *unwind* the two nested loops, dynamically generate a Job Descriptor File, as described in Section 2.2.3, and then submit the scripts and the data to the grid.

In the first phase, we used the infrastructure in three complex experiments, reported in Table 3, in which the computation is only a small, but significant, part. After a tuning of the configuration, the distributed application gave us remarkable benefits with little effort. The speed up obtained was more promising that the dry numbers seem indicate. The first experiment was performed during Christmas 2008 holidays, when most of the computer rooms in Universities were closed and only few hosts were running the grid computation. A post-experiment diagnose suggested that when sufficient PCs are available, the benefits would be effective. The second experiment was used to evaluate an optimal task allocation strategy. The conclusion was that the size of each task should be not too small, to pay off the transfer time (many of the hosts were in Turin), but also not too big, to lower the probability of a student in the computer room starting to use the machine. Being the PCs in ShareGrid non ded-

28 Referring to Affymetrix HG-U133 Plus 2.0 GeneChip

icated resources, the local activity is preemptive and it kills suddenly the background computation, which must restart on another machine. We empirically evaluated the right size of task such as it might complete in around five minutes. The second improvement was to use a scheduling strategy which replicated the tasks on different grid machines, if idle, and run them simultaneously. This further reduced the probability of a task being killed in all its instances and restarted. The results of these optimizations are highlighted by the third experiment, which reports a speed up of 36 time. For our users, this means switching from a wait of three weeks to less than one day. After these trial experiments, the infrastructure has been used for other computations [SBDM+09].

Experiment name	**samples**	**features**	**questions**	**CPU time (h)**	**Wall-clock time (h)**
1 Brain tumor	68	54913	6	~ 3000	1200
2 Ependymomas	19	54913	9	1302	316
3 Breast cancer	198	up to 22238	2981	437	12

Table 3: Main gene selection experiments. Wall clock, from the launch of the experiment to the return of last result, time includes data transfer. CPU time is the sum of all single tasks run time recorded on the remote node and match with the estimated time required for a single workstation. One week is 168 hours.

Despite the satisfying performance improvement, several issues still remain to be solved. As explained in the in the introduction, this solution suffers from many of the flaws frequently found in scientific applications:

- the user needs to handle several different tools at the same time;
- the granularity of the tasks, once distributed, is fixed;
- troubleshooting is cumbersome and out-of-band;
- there is not an obvious way to distribute tasks requiring different library/base applications;
- data distribution is primitive;
- there is not checkpointing facility and the task restarts from the beginning when killed by the local user activity;
- input, output and interprocess data exchange is based on files.

Moreover, the solution adopted resulted results tightly bound to the algorithms we have implemented on it , not scalable, and not easily generalizable to other applications.

4.3 The distributed testbed infrastructure

In both the original l_1l_2 and the new Data Grid-enabled experiments, we exploited ShareGrid, an existing Desktop Grid infrastructure, to distribute the computation across several PCs. ShareGrid [ACGB+08] is a collaborative project which involves several Universities in Northern Italy. Each partner allows the others to use his own computational resources on a reciprocity basis. As of October 2009, the participants to ShareGrid were the Department of Computer Science of the University of Piemonte Orientale, the Department of Computer Science, the Department of Economic and Financial Sciences "G. Prato", and the Department of Drug Science and the Re.Te.-Centro di Interesse Generale d'Ateneo Reti e Telecomunicazioni of the University of Torino, TOP-IX, the Torino Internet traffic exchange point, the CSP - Innovazione nelle ICT, and the Department of Computer and Information Science of the University of Genova.

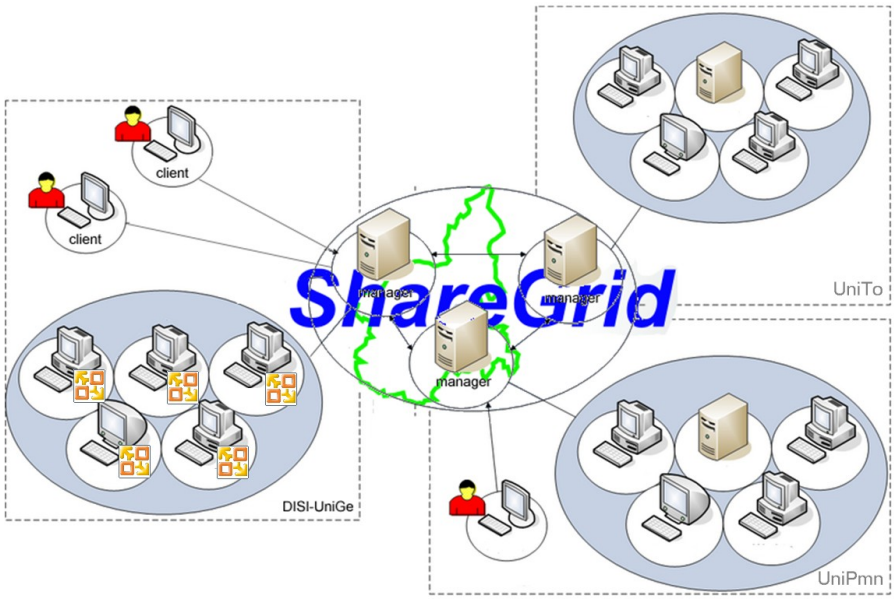


Figure 26: The ShareGrid/OurGrid architecture

The ShareGrid infrastructure is based on OurGrid middleware, developed at the Universidade Federal de Campina Grande (Brazil) and sponsored by HP. OurGrid is based on a two level peer-to-peer architecture, as described in Figure 26. At the first level, a department creates its own desktop grid infrastructure, installing on each PC an agent and using a central supervisor (*manager* or *peer*) to coordinate them. At the second level, the supervisors of different organizations connect in a mesh to establish a P2P network.

OurGrid is designed to support only the execution of the so-called *Bag-of-Tasks* applications, consisting in a set of independent tasks that do not communicate among them [CG89]. Despite their limitations, Bag-of-Tasks are used in a variety of domains, such as parameter sweeps, simulations, computational biology, and computer imaging.

We successfully used ShareGrid for different research works [BFSV09][BF09][FL09], but the lack of coordination between tasks, especially concerning splitting strategies and checkpointing, suggested us to develop the solution presented here. Unfortunately, not all ShareGrid's sites are suitable to run P2P Data Grids; the major obstacles found were incompatible JVM versions on the hosts and troubles with multicast traffic in some LAN configurations. Since one of the features of grid toolkits is to allow the selection of the resources, after a scouting run of a probe program, we identified the peers compatible with our experiment and then run the test on these nodes only.

4.3.1 Framework architecture

Over ShareGrid we apply this overall strategy to add checkpoiting, shared storage, and other features supplied by Data Grids to the application. The user, using the standard OurGrid tools, deploys the Data Grid nodes and additional libraries on several machine. As observerd in the previous analysis, Data Grids is not usually designed to work over WAN or in partitioned networks, hence the hosts must be selected from the same LAN. The hosts might also been selected to meet the application requirements, such as available memory, operative systems, etc.... When the nodes start, they discover each other and establish an overlay network, as illustrated in Figure 27.

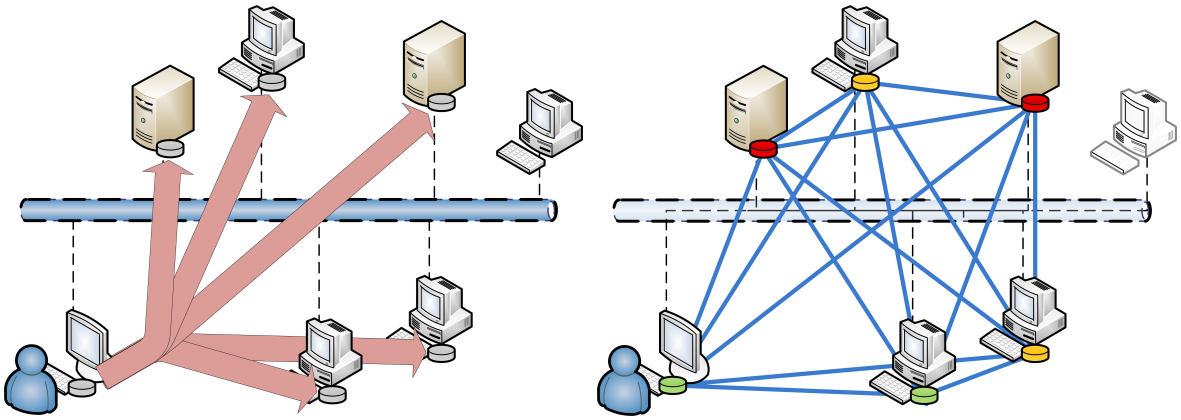


Figure 27: The user deploys Data Grid nodes using the grid tools and selecting suitable resources. At startup, the nodes discovered each other and setup an overlay network, distributing data partitions providing at least one backup for each partition

If a node leaves the Data Grid, for instance for the crash of the PC or because somebody on the local console starts to work on it, the other nodes rearranges partitions assignment to ensure the presence of backups, as in Figure 28. When the Data Grid cluster is established, the user can start his application, on its own PC if connected to the same LAN, or in another node of the remote grid.

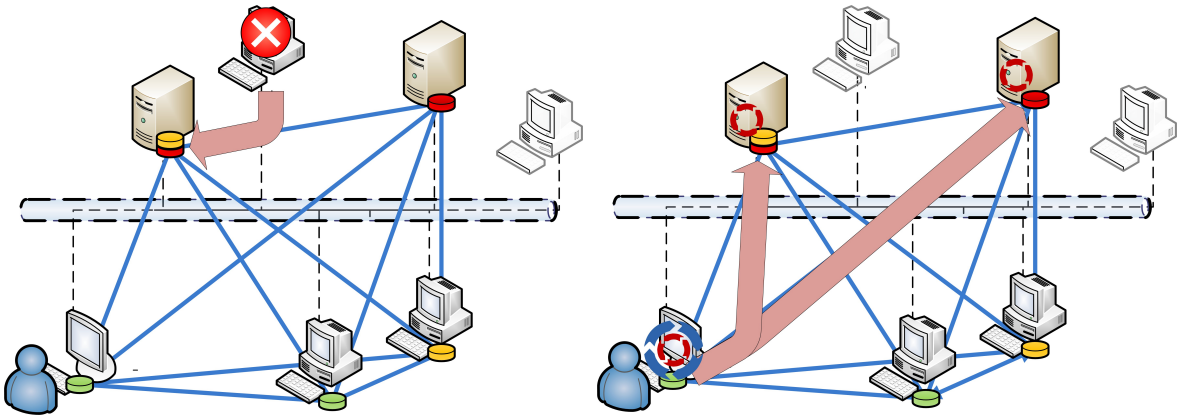


Figure 28: In case of Data Grid node leaves the cluster, the Data Grid rearranges partitions. The underlying Desktop Grid infrastructure could restart the crashing node. When the user application has a subtask associated to a specific data, it is routed to the nodes hosting that data in their partitions and executed locally

Any subtask of the application can be sent to the node hosting the partition and executed concurrently. The data in the Grid can survive to the loss of one or several nodes if the interval between each crash allows the partitions to be reallocated.

4.4 Data Grid-aware l_1l_2

The first step to this new architecture was the porting of the code to a version entirely rewritten in Java. A preliminary version was prepared mapping the original MATLAB code to Java, using JAMA [jama] as the base matrix manipulation library. The access to JAMA API is mediated by few abstraction classes, which allow the easy replacement of the library for linear algebra and matrix manipulation. It is worth noting that microarray matrices are dense, therefore different matrix implementations could hardly make a difference in memory footprint. Potential benefits can be expected from libraries which apply a strategy based on views rather than on copies for certain operations, but these are rare in l_1l_2.

Preliminary tests were also performed with UJMP [ABN09], which offers a more modern design and can seamlessly integrate other matrix libraries aside the built-in functions. In our

test UJMP performs more than two times slower than JAMA[29], and in the current release it requires additional libraries to implement some features such as Singular Value Decomposition, without offering notable benefits to our application. The support UJMP should be desirable for its capability of exposing `Map` objects, backended on disk possibly, as UJMP matrices, thus outlining a good integration with IMDG.

This first Java release of the l_1l_2 framework, hereafter referred as `L1L2Base`, was used as baseline for all remaining comparisons. As preliminary step, `L1L2Base` was compared with the original `L1L2_TOOLBOX`. Tests on a Windows XP platform shown that the Java implementation is slightly slower than the original `L1L2_TOOLBOX` executed in MATLAB, but outperforming twice the same script executed using the open source GNU Octave [Eat02].

This was a critical point: MATLAB is a commercial product whose license does not allow a distributed use, especially in a grid infrastructure spanning multiple institutions such as ShareGrid. To be complaint with the license, we adopted GNU Octave, which is highly compatible with MATLAB, but considerably slower. In the grid implementation, the huge number of running machines partially compensate this deficiency. However, the performance boosting obtained by using Java is largely sufficient to justify the switch to the new implementation.

4.4.1 A Data Grid framework

The `L1L2Base` implementation has serious memory requirements, as shown in Figure 29, that could dramatically increase in a parallel execution, due to the increasing number of matrices instantiated as intermediate results.

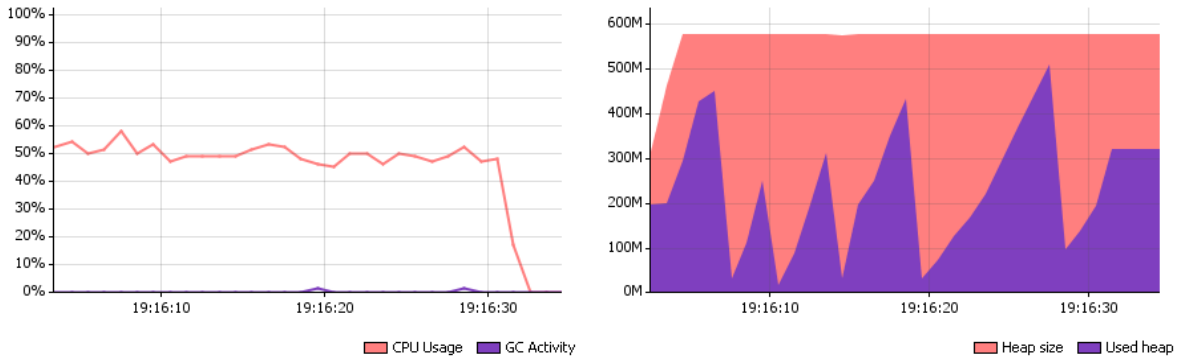


Figure 29: Java VisualVM console showing the memory allocation and the CPU utilization required to load 10 samples (image from a dual-core CPU, with the basic single thread l_1l_2 implementation).

29 This contrasts with the benchmarks published by UJMP (http://www.ujmp.org/java-matrix/benchmark/ last accessed 1 Nov 2009) which, on a limited set of operations (a single matrix multiplication and a single matrix transposition), seems indicate a completely different performance ranking among Octave, JAMA and UJMP. However, l_1l_2 application includes several matrix operations.

In-Memory Data Grids offer the opportunity of facing both the memory footprint and the computation performances at the same time.

The specific structure of data collected by microarrays, with few lines having many columns adopting the classical tabular representation of data mining, and the specific protocol of leave-one-out cross-validation, have suggested to load the data on the Data Grid associating each row to an entry. The size of each row resulted around 430 kbyte, probably too large to obtain optimal performances with some Data Grid implementations, but the tests with l_1l_2 did not manifest any anomaly. Being each line stored in the Data Grid, in a partitioned configuration possibly, the tasks are submitted to each node. Each task is defined by a tuple from the parameter space to explore and by a set of keys associated with lines forming the training set. For a generalised implementation, also the keys of the test set are included in the task definition, although in this specific case they could be derived as the complement of the training set. This also avoid to implement a naming strategy to keep separate entries in the map belonging to the dataset from others with different origin; even if some other process save a new entry in the Data Grid map, providing the key is not present, there is no possible overlapping.

If the underlying Data Grid implementation supports targeted or key-based routing, the task will execute on a node owning locally at least one of the involved entries. In this execution phase, because of the nature of leave-one-out cross-validation, in which each task needs all the dataset except one entry, and the design of partitioned Data Grid, in which redundancy is minimized, there is no chance to do a better job, and some network load is unavoidable. However, any other mechanism, such as shared file systems, RDBMS, etc... will need to transfer all the data to the node running the task. In Data Grid it is possible to use Near Cache feature, configurable at deployment time, to handle repeated access to the same keys in an efficient way.

At each stage, the task saves in the Grid its partial results, associated with a *meaningful* key, in the sense that the key alone permit to reproduce the result. As an example, each iteration of the inner loop of l_1l_2, is determined by a tuple of values from the parameter space and a set of row used for training. Thus, a class `InnerLoopParameters` is defined and an instance is used to keep the values at each iteration and as the key to store the result of the iteration. This design brings two benefits: first, it is easy to generate the complete list of `InnerLoopParameters` combinations without actually computing the partial results. Then each key can be sent to a different node for a parallel computation. The second benefit is that the task, before starting a computation, can check the map in the Data Grid to verify the presence of the entry; if so, the result has been already computed and the task can read it, proceeding directly to the next step. Saving the entry should be an idempotent operation to

ensure that multiple `put()` of the same key will not produced wrong effects but useless computation. Not always this is possible, since some machine learning methods, such as K-means clustering, rely on steps including random decisions. However, a careful design of the implementation can avoid these situations.

Saving the partial results associated with the values that generate them as key allows, in case of node crashing, the dead task can restart on another node and recover the results obtained so far. Thus, the checkpointing feature comes *for free* in a Data Grid.

The other interesting advantage over traditional grid approach comes from their underlying P2P architecture and MapReduce execution model. In a pure P2P implementation, each node is equivalent and each task running on a remote node can launch other distributed tasks as well. This characteristic might be accessible only in grids with a fine-grained API. In fact, at that level, there is an uniformity between the functions that the user invoke to submit a task and the functions that the task itself can invoke when running on a remote node.

In general, recursive task submission is error prone and should be handled as harmful by a casual developer. As in multithread programming, the most probable result is a deadlock or a race condition. The MapReduce programming model can help in this context. Although being criticized for its lacks of expressiveness or as a "step backwards" [DWS08], the MapReduce model was designed to "*allows programmers without any experience with parallel and distributed systems to easily utilize the resources of a large distributed system*" [DG04].

In Data Grids that in a broad sense support the MapReduce model, the risks factors introduced by careless programming are reduced. In fact, each framework supposes to receive a "map function" which accepts *one* parameter and produces a *list* of results, and a "reduce function" that accepts a *list* of results and returns *one* value. If such an API is combined with the Java generics facility, it is hardly to produce unwanted harmful code. It should be noted that, at the present, only few Java-based grids middlewares, such as GigaSpaces XAP and GridGain, support both MapReduce and generics.

4.4.2 Performances

To evaluate the performances of the system in a real scenario, we have repeated the same computation in two different configurations. In the first case, we used our departmental cluster composed of 20 dual Intel Xeon servers connected through a dedicated Gbit Ethernet network. Having not external interference, this facility provided a baseline of repeatable results. The tests have been repeated on two different ShareGrid sites during normal activity time in remote computer rooms. In both cases, each computation has been executed repeatedly. The number of repetitions ranging from at least three in the cluster, to some dozen

on ShareGrid, varying with the availability of the remote hosts. In the results, we show the average time for each test set. The computation consists in a feature selection run on a synthetic dataset which we normally use to validate and to test l_1l_2 implementations. Experiments have been repeated with 3k, 15k, 50k and 200k datasets, where the number indicates the thousand of cells in the matrix.

Results obtained from the cluster, summarized in Figure 30, show the Data Grid-enable l_1l_2 can scale when adding nodes, in the x-axis, even if less than expected. Execution times, in y-axis, are normalized to the one elapsed by the two nodes setup. For each dataset and node configuration, the standard deviation of the experiment was usually less then 10%, except for the strange behaviour of the 50k dataset which seems more susceptible to the cluster size, resulting sometimes faster than the ideal speed.

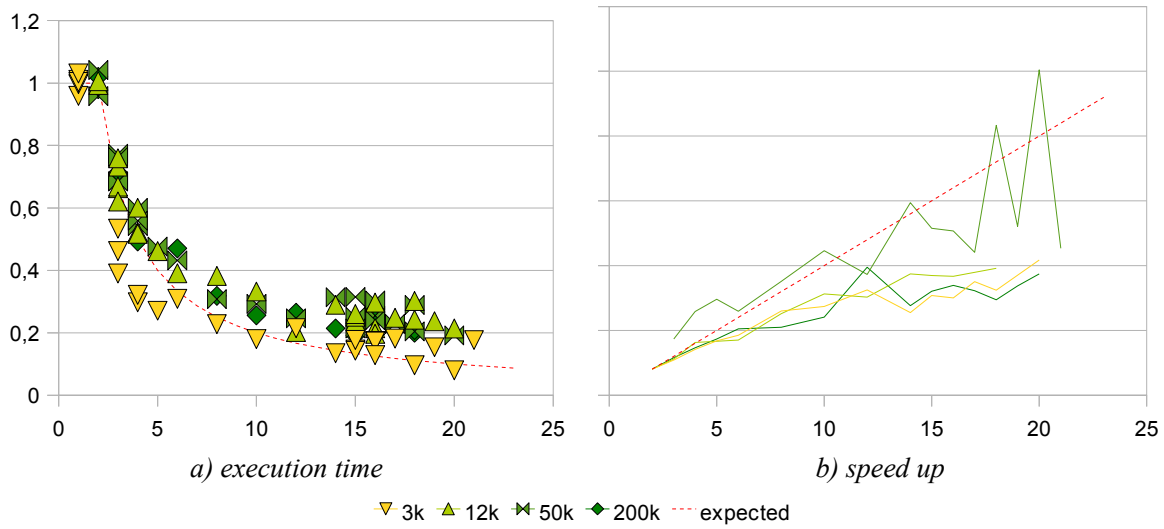


Figure 30: l_1l_2 cluster experiments. In the x-axis, the number of nodes in the cluster, in the y-axis, on the left, the execution time compared to the two-nodes result for the same dataset, on the right, the speedup.

The same tests have been repeated on ShareGrid. As expected, due to the dynamic of Desktop Grid infrastructures, the results were more variable e sometimes unpredictable. As exemplified in Figure 31, small datasets could have some benefits from the distribution in a Desktop Grid, as shown by the 12k dataset experiment executed on Turing lab, but larger problems suffer from the contention with local user activities, resulting in inconsistent performances.

An interesting point is that many executions on ShareGrid had to deal with node crashes. We experienced up to six nodes leaving and five nodes joining the cluster in a single run, with delays ranging from 15 seconds to 5 minutes, depending on data size and target laboratory. Rarely these events have caused the computation interruption.

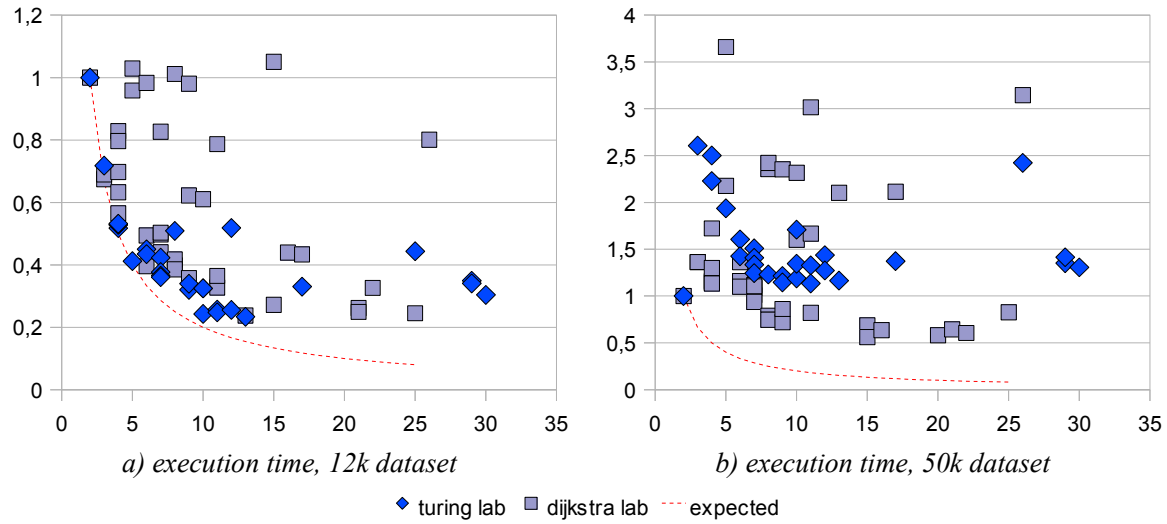


Figure 31: $l_1 l_2$ ShareGrid experiments. In the x-axis, the number of nodes in the cluster, in the y-axis, the execution time compared to the two-nodes result for the same dataset. The Turing lab is based on Microsoft Windows XP systems, the Dijkstra lab on Sun Solaris 10.

The data analysis has filtered out some outlier values. Less than a dozen of runs required more than ten times to complete, in few cases more than one hundred times, without no evident correlation with the status or the activity of the target LAN. Node logs might give some hints, but unfortunately the current OurGrid architecture returns the logs at the task fulfilment only, while frequently the PC is switched off or rebooted before the hosted Data Grid node shutdowns gracefully. Being many of the PCs owned by other organizations, it was not possible to read logs at runtime directly from the node console.

4.5 Data Grid integration with existing applications

The scenario presented in previous sections provides basic services and algorithms. In the aim of evaluate IMDG as distributed framework for data mining applications providing a programmer-friendly programming models, we have evaluated different applications, described in Section 2.1.1. We present here only the results obtained from KNIME. Previous prototypes based on Weka have shown several problems, due to the legacy of *code smell* [Flo99], the pre-Java 5 multithread techniques, and its old design. The main obstacle to interface with a Data Grid, which handles serializable object only, was the foundation class `weka.core.Instances`, as noted in Section 2.1.3, which cannot used nor extends as-is. The refactoring needed to convert Weka to an interface-based design is out of the scope of this work. Debellor re-uses Weka and thus present similar problems.

KNIME is based on a plugin architecture compliant to the OSGi platform [osgi] provided by the Eclipse platform [eclp]. This allows for a great extensibility e a clear separation of concerns in design. Being KNIME a visual Workflow Manager, the "building blocks" are identifiable with the workflow nodes, even if OSGi modules (*bundle*) can provide many different services entry point. As an example, the services of a computational grid might be exposed to the programmer as a platform-wide service through an API, or might be presented to the user as a workflow metanode. The first option represents a valid choice to implement the support for Data Grid. Unfortunately, it could require an heavy work in some core classes, producing an incompatible fork of the original project.

Thus, for the sake of simplicity, we have implemented the modules to access the Data Grid as nodes. Creating a node plugin is, in general, quite straightforward, just requiring to provide a concrete implementation of few methods in a skeleton generated by a wizard. In our case, there was complication. Many of the core interfaces and classes, such as `org.knime.core.data.DataTable` and its implementations, `o.k.c.d.DataRow`, and `o.k.c.node.BufferedDataTable`, do not implement the `Serializable` interface. These classes cannot be used as-is to move data from and to the Data Grid. Since elementary components, such as `o.k.c.d.DataCell` are serializable instead, we solved this problem implementing a new data type (*Port Object*) which encapsulates serializable components in a new class hierarchy which maps the core classes. In the prototype, the user needs to explicitly transfer the data to and from the Data Grid and special nodes are required to manipulate them. These nodes delimiting the scope of Data Grid are introduced as prototypical level without any logic inside. Node which operate on data are copied and adapted form the default KNIME release.

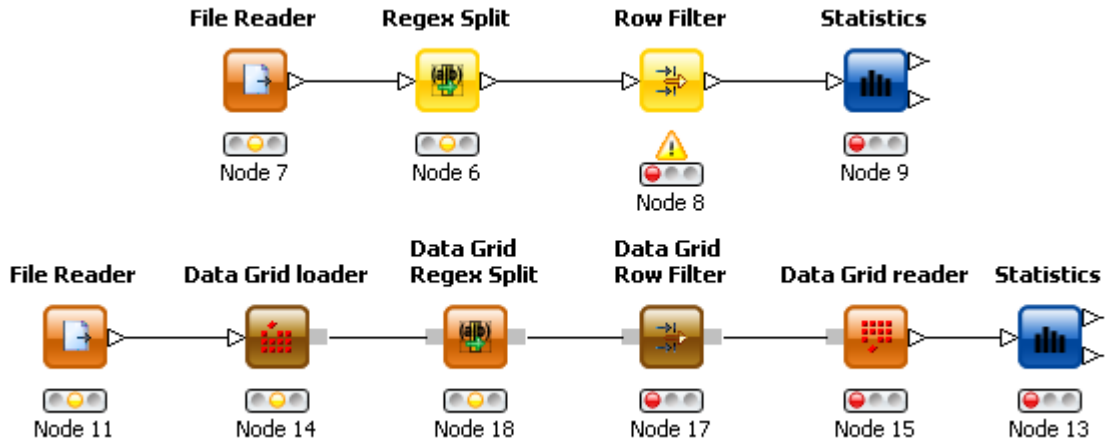


Figure 32: Data Grid-aware nodes in KNIME. The upper workflow reads a text file, split lines in records and filter the resulting data rows in order to compute basic statistics. The lower workflow operates the same data operations, but loading data in the Data Grid.

Thus, the prototype works as shown in Figure 32: data from a standard node are loaded into the Data Grid, then are elaborated by specialized version of nodes, sharing the same codebase of equivalent defaults, and finally the rows are collected from the Data Grid and returned to the local workflow.

This looks like the design of KNIME built-in database node group, with an important difference: it is only a tactical design choice to ensure the compatibility with the main project. As we will see, the pre-existing code can be converted to supports Data Grid natively.

To support different IMDG implementations, we developed a simple façade framework, named SDGF (*Simple Data Grid Façade*), which abstracts basic operations and presents them in with an uniform interface. Some details about SDGF are described in Section 4.6. In SDGF, the backend IMDG implementation can be choosen at deployment time, using a system property.

In this new design, plugin nodes require an unusual approach. The current programming idiom is expecting to iterate on data rows, applying some operation on them. For example, in the original code of node “Row Filter”, shown in Listing 16, the whole input table is iter-

```
protected BufferedDataTable[] execute(
          final BufferedDataTable[] inData,
          final ExecutionContext exec) throws Exception {

     DataTable in = inData[0];
     // ... initialisation code ...

     BufferedDataContainer container =
          exec.createDataContainer(in.getDataTableSpec());

     try {
          int count = 0;
          RowFilterIterator it =
                    new RowFilterIterator(in, m_rowFilter, exec);
          while (it.hasNext()) {
               DataRow row = it.next();
               count++;
               container.addRowToTable(row);
               exec.setMessage("Added row " + count + " (\""
                         + row.getKey() + "\")");
          }

     // ... catch clauses ...

     return new BufferedDataTable[]{container.getTable()};
}
```

Listing 16: Snippet from class
`org.knime.base.node.preproc.filter.row.RowFilterNodeModel`

ated and rows meeting certain criteria are copied in a new data table. In this case, “the data go to the task”, sometimes in literal meaning: if the table, for instance, is backended in a text file, to iterate through it implies read the file line-by-line from the disk. In Data Grid, to obtain any benefit, the flux must be inverted and the tasks moves where the data are. Being the storage unit of the Data Grid the associative map, this means each task must be targeted to a map.

The equivalent Data Grid-aware version of “Row Filter”, based on the SDGF framework and shown in Listing 17, creates a `Task` object, specialized for data copy, and sends it to the Data Grid specifying the target map. The underlying IMDG implementation, if capable, will spread the task on the nodes and each node will execute it locally against owned entries.

The `Task` interface of SDGF, and its subinterfaces such as `CopyTask`, has basically the same role of a `Callable` or `Runnable`. The main constrains in Data Grids is that each object must be serializable, thus `Task` extends `Serializable`. In the example, it is not possible to instantiate an anonymous class, since the model of a workflow node in KNIME is not serializable. In other cases, having an interface which extends both `Callable` and `Serializable` could help the programmer to quickly implement small tasks.

The Data Grid maps are handled by name, to ensure the maximum compatibility among IMDGs and the possibility to switch directly to the backend implementation API if advance features, not provided by SDGF, are needed.

Submitting a task on the Grid returns a `Future`, as in the familiar model of `j.u.c.Executor`, which allows asynchronous execution and, if the underlying middleware has this capability, also allows the cancellation of the task. This supports some usability features normally required in a GUI. Unfortunately, in this architecture is hard to provide a progress bar; since it is not clear how to compute the progress ratio and where execute the callback updating the values in such a parallel environment with an unknown number of nodes. Moreover, obtaining some information from the remote nodes, even if the middleware supports messaging, could result in an excessive network overhead.

```
public static class Filter implements CopyTask<SRowKey, SDataRow> {

     // ... plumbing code ...

     @Override
     public SDataRow execute(SRowKey key, SDataRow row)
                    throws Exception {
          if (m_filter.matches(row)) {
               return value;  // Copy to the new map if matchs
          }
          else {
               return null;
          }
     }
}

protected DatagridConnection[] execute(final PortObject[] inData,
          final ExecutionContext exec) throws Exception {

     DatagridConnection in = (DatagridConnection) inData[0];

     // ... initialisation code ...

     Map sink = Datagrid.newMap(m_mapname);  // Create sink map

     CopyEntryTask<SRowKey, SDataRow> filterTask = new Filter();
     filterTask.setSinkMap(m_mapname);

     Future future = Datagrid.submit(in.getMapName(), filterTask);

     future.get();  // Wait for completion

     // ... plumbing code ...
     return new DatagridConnection[] {dgTable};
}
```

Listing 17: A Data Grid-aware rewriting of RowFilter code

The MapReduce model can be applied as well. For example, it is easy to use a reduce-only procedure to compute basic statistics on a distributed table. The *`map()`* part is not required by SDGF which automatically creates a 1:1 map. In Listing 18 the usage of a *`reduce()`* function to compute a statistic table holding statistical moments, such as mean, variance, column sum, count missing values, minimum and maximum values, etc... is shown.

```
public static class StatReducer
          implements ReduceTask<SDataRow, StatisticsTable> {

     private final DataTableSpec dataSpec;

     // ... plumbing code ...

     public StatisticsTable reduce(List<SDataRow> results)
               throws Exception {
          // A modified version of default StatisticsTable
          // which allow incremental operations
          StatisticsTable stat = new StatisticTable(dataSpec);

          for (SDataRow row : results) {
               stat.addToAllMoment(row);
          }

          return stat;
     }
}

protected DatagridConnection[] execute(final PortObject[] inData,
          final ExecutionContext exec) throws Exception {

     // ... plumbing code ...

     StatReducer reduceTask = new StatReducer(m_mapname);

     Future future = Datagrid.mapReduce(dg.getMapName(),
               null, reduceTask);    // null map() is handle
automatically
                                     // to 1:1 map

     StatisticsTable stats = future.get();

     // ... plumbing code ...
}
```

Listing 18: A MapReduce implementation of basic statistics

The MapReduce-aware versions of many machine learning algorithms are available. As already noted, MapReduce is not a new programming technique and does not perspect radically different performances on parallel system. It is designed to cope with the difficulty of distributed computations in both execution on unreliable hardware and programming by experienced programmers. Since performances of parallel versions of machine learning algorithm are more dependant on the algorithm itself rather than by the middleware which executes them, we concentrate our tests on basic tasks, such as filtering and conversions, focusing on the benefit for the programmer.

4.5.1 A basic ETL test

In many cases, an IMDG might act as a store for transient data continuously flowing from sensors, diagnostic services, or OLTP systems. As an example, we could think to a Data Grid feed from system logs, emails, and SNMP trap events, which keeps copies of the original data for a short period. In this case, losing some data, due to overload or network problems or discharged old ones, is an acceptable event. The infrastructure could be used to discover anomalies or behavioural patterns for intrusion detection, failure prediction, or spam fighting.

The raw data must be preprocessed to adapt to the system. Such phase, known as ETL (*Extract, Transform, Load*) is a crucial process in data mining. Typical ETL is time consuming, because data are recorded on databases or files that, having an architecture designed to ensure integrity maintenance, slow down this type of operation which often does not require the same high level of integrity insurance. Even in well-designed infrastructure, the database is usually the bottleneck due to the fact that RDBMS can only scale up (buying more powerful, and expensive, hardware), and not scale out by adding other inexpensive servers.

We can expect In-Memory Data Grids might speed up ETL processes for continuous operation, such as in systems which continuously receive new data, in two way: firstly, the data are transformed on their arrival and not at load time, secondly, the processed data are keept as POJO (or other optimized form) rather than as database or textual data, without the overhead introduced by marshalling/unmarshalling operations.

For the same reasons, an In-Memory Data Grid could also be used has backend for passivation and checkpointing, hiding to the programmer most of the complexity of handling a configurable support for object persistence. Moreover, in situations in which the same pre-processed data have to be used in different clients or in long batch operations, table stored in the Data Grid could be shared with other applications and clients can detach from the Data Grid and return later to collect the results.

To test this scenario, we performed a cycle of ETL to analyse a log file generated by an Apache web server and designed a typical web analytic job, consisting in the extraction of cumulative statistics form an web server log file. The log lines are tokenized, then the fields are converted in the appropriate Java type. The resulting objects are then processed, such as extracting the network from the IP address, to compute aggregate statistics about bytes downloaded per network, preferred pages and their hit numbers and the set of user agents (*browsers*).

Lines came from a real web site Apache `httpd` server log, and have been loaded in the Data Grid, in batch starting form 1,000 lines to reach 512,000 ones. The tests have ran on our de-

partmental cluster, using a pool ranging from 1 to 8 nodes. Each machine have two Intel Xeon processors, and uses Slackware 64bit Linux and a Sun 64 bit JDK 6 Java Virtual Machine with a heap limit of 768 Mbyte.

After loading, the data was processed in this order:

1) each line was loaded in a table or map of the Data Grid using a progressive identifier as key; The typical log line and a sample of the result is shown in Listing 19.;
2) using a regular expression, each line was splitted in an `String` array of its components, invalid line were discharged, and resulting arrays were stored on the Data Grid;
3) each `String` array was copied in new array of objects of the correct class, converting the byte transmitted into an `Integer`, the timestamp into a `Date`, etc... Resulting arrays were also stored in the Data Grid; not convertible values were reported as `null` and handled as missing values;
4) from the whole table statistics were extracted: source network of the request and bytes downloaded per network, preferred pages and their hit numbers, and most common user agents.

```
// Typical log lines
95.108.128.242 - - [17/Sep/2009:06:41:12 +0100] "GET
/manuali/eudora5/smtpeudora.html HTTP/1.1" 200 5101 "-"
"Yandex/1.01.001 (compatible; Win16; I)"
65.55.207.119 - - [17/Sep/2009:06:41:35 +0100] "GET / HTTP/1.1" 200
10452 "-" "msnbot/2.0b (+http://search.msn.com/msnbot.htm)"
195.210.89.37 - - [17/Sep/2009:06:41:50 +0100] "HEAD
/genuanet/mrtg/scpo-month.png HTTP/1.1" 200 - "http://www.unige.it/"
"libwww-perl/5.805"

// Statistics
Byte/network:
     130.251.121=1356063104
     83.224.68=5590464
     ...
Url/hits:
     /genuanet/wm/=2368
     /genuanet/mrtg/ling.html=1152
     ...
Browsers:
     msnbot/2.0b (+http://search.msn.com/msnbot.htm)
     Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1)
     ...
```

Listing 19: Typical Apache log lines and analytic results

The tests have used the code of our KNIME custom nodes. As backend, both Coherence and Hazelcast have been used. Each operation has been performed using as much parallelisms as the underlay middleware could offer. Specifically, using Oracle Coherence, the operations on lines were implemented as instances of `InvocableMap.EntryProcessor` and the aggregate statistics were computed with instances of `ParallelAwareAggregator`.

Being the Data Grid configured to use one backup copy, the result of a two-nodes cluster have not significance, since this configuration handle exactly the same data of a single node, just adding a high communication overhead.

Coherence has been tested in various configurations. One of the most interesting feature is the Asynchronous Store Manager backend on a NIO buffer[30]. In this case, the operations involving data storage, as shown in Figure 33, require more time as the number of nodes grow, but the operation with a high degree of parallelism scale better.

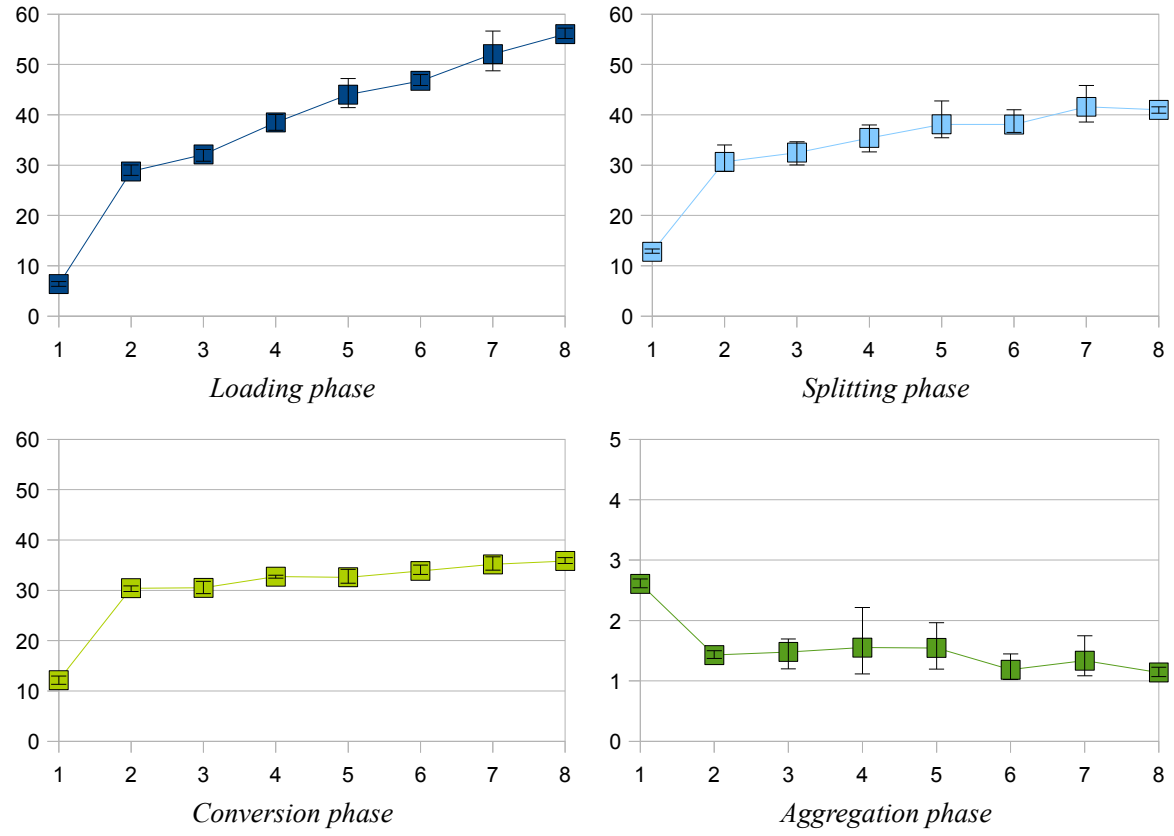


Figure 33: Coherence with Asynchronous backend ETL test, 128,000 log lines sample. Note the different scale of aggregation phase.

30 http://coherence.oracle.com/display/COH34UG/async-store-manager (last accessed 31 Jan 2010)

Hazelcast shows a different behaviour; as shown in Figure 34. The complete test required about the same time independently by the number of nodes (except in the case of a single node).

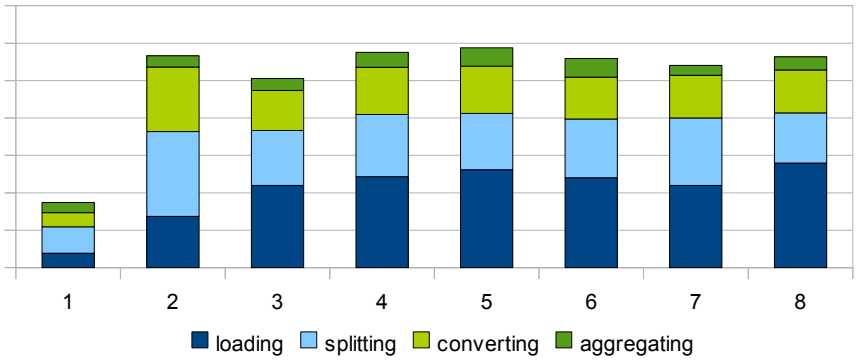


Figure 34: Hazelcast ETL test, using 64,000 log lines sample

Variation in the phases are more likely due to erratic network and host conditions, as visible in the details graphs in Figure 35.

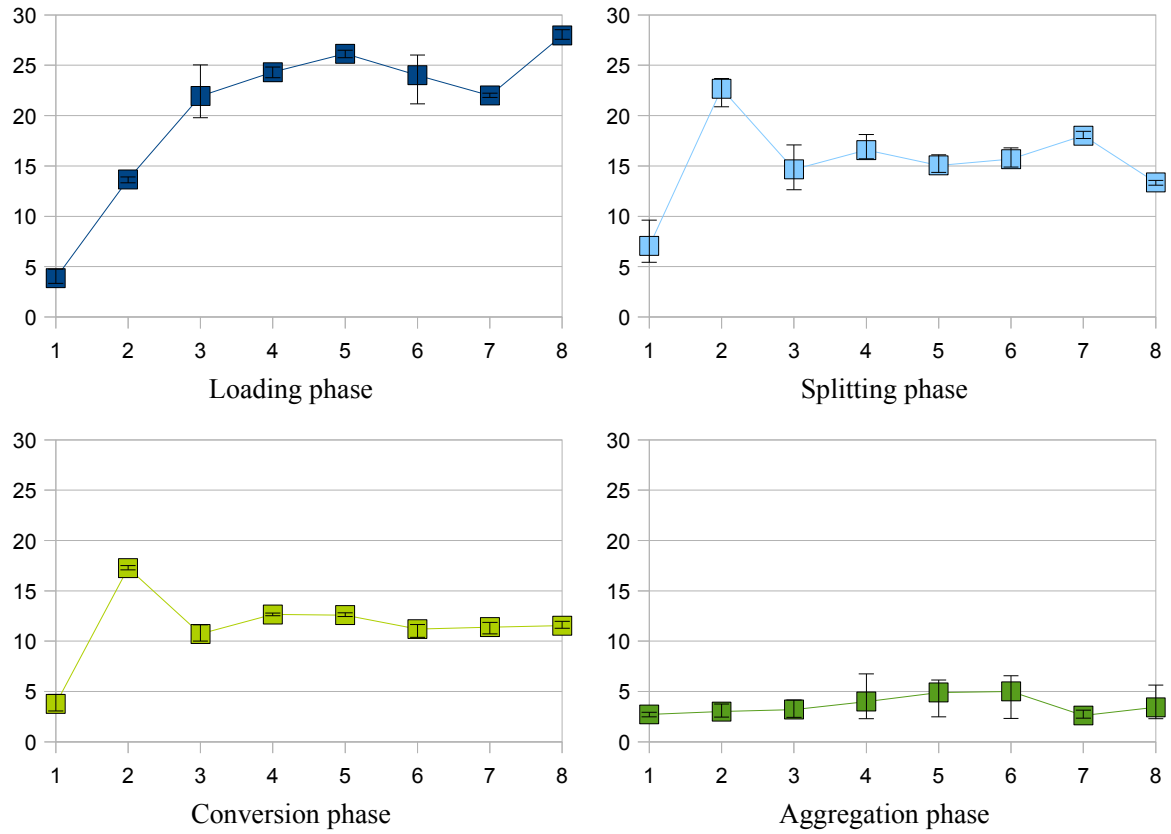


Figure 35: Hazelcast ETL test, 64,000 log lines sample, details.

This behaviour is due to the synchronous architecture of Hazelcast; both load/store operations and distributed task executions are blocking and the algorithms cannot exploit concurrent execution to increase performances. Obviously, in other contexts such as Web Application clustering, this behaviour, which does not depend on the number of nodes, might be an advantage.

4.6 Some details on the Simple Data Grid Façade framework

To test the different Data Grid implementations, we developed the *Simple Data Grid Façade* (SDGF), a lightweight abstraction library which offers an uniform API to many Data Grids. SDGF is inspired to other well known libraries with similar goals, such as SLF4J (*Simple Logging Façade for Java*) [slf4j] which serves as an abstraction for various logging frameworks.

This library is by far from having a production-grade quality, however its use provided some suggestions. SDGF works at a low level and aims to be used as a connection provider. Upper level abstractions can be found by other projects, such as Cascading [casc] or Granules.

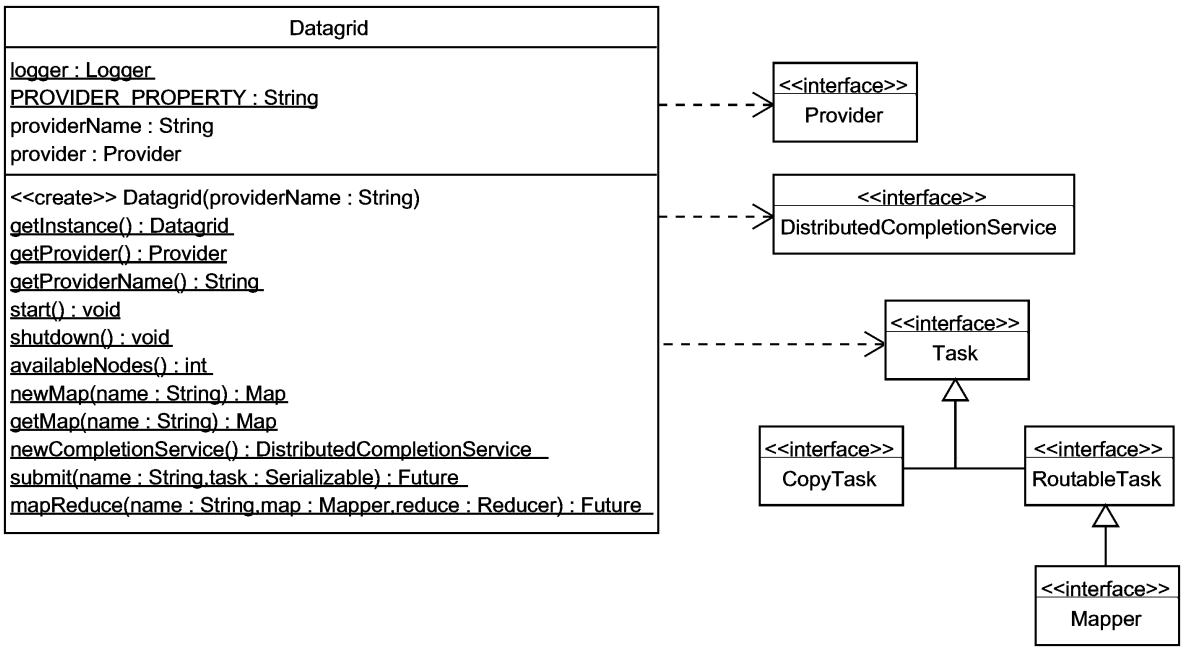


Figure 36: Class diagram of SDGF

4.6.1 Instantiation pattern

The Data Grid is accessed through the `it.unige.disi.sdgf.Datagrid` class, which implements a singleton, handles the lifecycle of a driver class, and mediates the access to the backend. Drivers implement the `i.u.d.s.Provider` interface, exposing a *Service*

Provider Interface (SPI) and wrapping the IMDG with a specific adaptor. A local implementation based on `j.u.c.ConcurrentMap` and `j.c.u.ExecutorService` is provided for both testing and fallback purposes.

Since many IMDGs do not offer a suitable execution layer, in such cases the missing features are provided integrating GridGain. Support for other computational grid implementations, such as JPPF, is possible.

The concrete `Provider` is specified using a system property. This design is close to many other Java services, such as JAXP [CW02]. Although, this is not a OSGi-friendly solution, the effort to assembly a service bundle was out of the scope of this work.

4.6.2 Task as serializable Callable

In SDGF, all tasks implement a specialized interface derived from `Task`. This interface has the same role of `Callable` in the package `j.u.c`. The main method is *`execute(key, value)`*, which operates on an entry. The code must be *reentran* and can raise exceptions; this design is close to Oracle Coherence and IBM WebSphere eXtreme Scale, while other implementations, such as GemStore GemFire or GigaSpaces XAP, use a model similar to `Runnable`. Unfortunately, `Runnable` does not help the programmer in preventing the perils of concurrent programming. It presents several dangerous points that a programmer needs to face with: returning a value require some attention to shared resources, there is not an obvious way to deal with exceptions, *`start()`* and *`join()`* must be explicitly invoked, etc... Moreover, the API should make transparent to the programmer the underlying distribution, but using class member as both input arguments and the returning value could lead to some confusion, since the programmer could perceive both as local variables and not as copy or proxy of remote shared data. Thus, this model is not supported in SDGF.

`Task` is not used as-is: each implementation has specific strong point to exploit, while other usage patterns are not directly supported and required adaptors. For this reason, SDGF offers several specialized subinterfaces, each one mapping to some specific feature of the backend. Missing features are emulated.

4.6.3 Built in Completion service

Many IMDGs support a grid-aware `ExecutorService`, but no one could be integrated with the standard `ExecutorCompletionService`. In fact, this component has hidden dependencies from `FutureTask`, at least in Java 6, which is not serializable [Fer09]. SDGF offers an advanced support to a grid-aware completion service.

4.6.4 Unsupported features and future directions

SDFG is in a preliminarily stage. Many features are not supported because they were not need for any specific test in this work. At the moment, for example, event notification is unsupported. Other features are not implemented by design. An example is the resource injection using annotations. This technique is applied by many frameworks, such as GigaSpaces XAP, GridGain, and WebSphere eXtreme Scale, but it seems still unfamiliar to many Java programmers. Moreover, it requires the dynamic discovery at runtime, which could degrade the overall performances.

5 Conclusions and future work

Let us come back to our original question asking whether In-Memory Data Grid is a technology suitable for distributed Data Mining. The answer cannot be neither a definitive yes nor a definitive no since lights and shadows has emerged.

Concerning their use as distributed storage systems, IMDGs seem fitting the architecture of typical data mining applications. In fact, the storage model based on familiar `Map` is very similar to the internal data structure used by most of recent data mining programs. Older applications could rely on an array representation of data, often inherited by the core libraries of machine learning algorithms, which would require more work to be adapted. Instead, libraries more focused on ETL operations might integrate seamlessly with this new technology. From this viewpoint, Data Grids could offer to distributed Java applications a scalable storage space, capable to handle objects as-is without *impedance mismatch*, with an integrated checkpointing facility, and fault tolerance mechanisms in Desktop Grid architectures. Although not standardized yet, storage APIs exposed by different products appear as similar and, with a tiny adaptation layer, almost all implementations seem interchangeable. Some issues are still open, such as the behaviour when the memory usage is near its limit, which results in an application crash instead of providing a graceful degradation.

A promising usage scenario seems that of permanent infrastructures dedicated to continuous analysis of streaming data coming from remote systems and having an authoritative storage, that can be found for example in fraud detections, anomalies searching in system logs, or spam fighting. In this use case, the absence of ACID properties is no crucial, and older samples could be discharged, since the goal is detecting abnormal behavioural pattern exploring recent data and comparing them with existent models.

Storage features have been considered as the primary APIs of Data Grids, but this technology is proposed as a support to move computational tasks where the data are stored. Unfortunately, the current status of the majority of the products resulted still immature. The tests discussed in Chapters 3 and 4, have shown that often these products do note scale as expected or can not handle faults in cluster nodes.

Another major problem is the difficult in the translation of the MapReduce programming model when it is ported to Java. Every implementation presents a different API, often in-

cluding some specific (an not-well documented) idiosyncrasy, thus making the code developed for one platform not easily portable onto another one. This introduces several costs. A programmer has to restart from scratch to switch to a different product, and a project is stuck to the original choice once a specific product has been adopted. This recalls the “old days” when each RDBMS required its specific libraries to be used. Technologies such as ODBC, JDBC, etc... have greatly simplified the life of programmers who do not have to worry anymore about the database until the deployment time. Also database producers have benefit from API-level interoperability, since new products will have no opportunity to emerge if switching to them is too costly. We hope such a situation could emerge for In--Memory Data Grid as well.

5.1 Open source full-featured IMDG

From the experience learned during our tests, Oracle Coherence emerged as a stable and performing solution. Unfortunately, Coherence as well as many other platforms cannot really be used in academic studies because of their license. Commercial solutions, in fact, offer a full support and a very stable implementation, but pose serious limitations to the usage in Desktop Grids within academic infrastructures because the licenses are expensive, and usually set limits in the number of CPUs or in the network boundary. Moreover, often they are too complex to manage without a skilled technical staff, easier to find in a stable infrastructure rather than in a “deploy on the fly” scenario.

It becomes evident the need of an open source solution, even with limited functionalities, but ready and easy to use. Hazelcast is a possible candidate but at the moment it offers strong capabilities on the storage side, while it has a poor distributed model for computation. JBoss Infinispan is a promising project, but its distributed execution capabilities are far to be implemented. In the roadmap, their are estimated for release 5.1.0.

We hope our Simple Data Grid Façade framework, once mature for a stable release, could help in this process, allowing the programmers to abstract from the underlying Data Grid implementation.

6 Appendix: A Java Data Grids survey

In-Memory Data Grids are currently a hot topic in the business and web-oriented developer community, but have received a little attention from the developers of scientific oriented applications. Most of the main features of Data Grids are not new and often were already present in “distributed cache” engines. In this survey, we will consider applications that define themselves “data grid”, or “distributed cache” or “distributed tuple space”, providing they have a native binding for Java.

Data Grid solutions are available as both commercial products such as Oracle Coherence, GemStone GemFire Enterprise, GigaSpaces XAP, or IBM WebSphere eXtreme Scale and open source projects, such as JBoss Infinispan and its predecessor JbossCache, Hazelcast, or Terracotta Ehcache.

In order to provide an overview of production-grade products suitable for Desktop Grid installations, the following implementations, in some cases only loosely identifiable as Data Grids, have been left out for their status at the end of 2009:

- Memcached (http://memcached.org/), because it has a client/server API access
- OSCache (http://www.opensymphony.com/oscache/), since it appears not more supported with the last release date back to 2007 and the last bug fixed in the January
- SwarmCache (http://swarmcache.sourceforge.net/), XSTM (http://www.xstm.net/), Whirlycache (https://whirlycache.dev.java.net/), and cache4j (http://cache4j.sourceforge.net/) because they are stick on beta or first released and not actively developed for more than 3 years.
- JBoss cache, since it is being to be replaced by Infinispan
- Cacheonix because it is licensed per-processor (http://www.cacheonix.com/) and the unavailability of publicly released documentation
- FKcache (http://jcache.sourceforge.net/) and SHOP.COM Cache System (http://code.google.com/p/sccache/) because, although actively maintained, they are completely new cache-oriented projects, with a very small team, currently in beta release

Besides the functional features enumerated in the Section 2.3, other characteristics are observed:

- General description, licensing, status, architecture and base libraries
- Eviction policy, multilevel cache, fail-over support
- Configuration mechanisms
- Networking features
- Querying and indexing capabilities

Features not reported here could be available in additional plugins or could be present but officially undocumented or unsupported.

These results is provided "as is" without warranty of any kind, either express or implied, including, but not limited to, the implied warranties of non-infringement, merchantability or fitness for a particular purpose.

6.1.1 Oracle Coherence

http://www.oracle.com/technology/products/coherence/

Latest release 3.5.3, January 2010

Oracle defines Coherence "*A JCache-compliant in-memory distributed data grid solution*". Oracle acquired Tangosol Inc. in March 2007. Coherence was the top product of Tangosol and is now part of the commercial offer of Oracle.

Coherence is commercially available in three edition: Standard, Enterprise and Grid. Grid Edition is also available for free download for testing and developing prototypes[31].

Every edition is distributed as a single JAR file of approximatively 4.5 Mbyte.

6.1.2 IBM WebSphere eXtreme Scale

http://www.ibm.com/software/webservers/appserv/extremescale/

Release 7.0.0, December 2009.

IBM claims "*WebSphere eXtreme Scale is an in-memory grid*".

Commercial license based on Processor Value Units (PVU). A free trial version is available, with the limitation of 8 hours of continual use. After that, the instance must be restarted.

Previously named ObjectGrid, until 2008, it is an ORB based product.

31 Read the OTN License carefully before download

6.1.3 JBoss Infinispan

http://www.jboss.org/infinispan

Latest release 4.0.0 Release Candidate 4, 2 February 2010 (for commercial reasons, the first release is directly numbered as 4.x, to not overlap with previous JBoss Cache releases).

JBoss defines Infinispan as "*an extremely scalable, highly available data grid platform - 100% open source, and written in Java*".

JBoss, a subsidiary of Redhat , Inc., developed a popular clustered caching library named JBoss Cache since 2003. In April 2009, JBoss announced JBoss Cache will be discontinued and it introduced a new product, named Infinispan. It requires Java 6.

Infinispan is released under GNU LGPL license.

6.1.4 Hazelcast

http://www.hazelcast.com/

Latest release 1.8, 15 December 2009

The web site announces "*Hazelcast Provides In-Memory Data Grid*" and "*Hibernate Second Level Cache*". It consists of one only jar file, sizing 750 kbyte, without any external dependencies.

Hazelcast is released under Apache open source license.

6.1.5 Ehcache

http://ehcache.org/

Latest release 1.7.2, 11 January 2010

In origin, Ehcache was "*an open source, standards-based cache*" only. Ehcache has recently merger with Terracotta in August 2009, hence there is an integration plan of the two products which generate a version more similar to a In-Memory cluster solution than a simple cache.

In the current release, Ehcache has few dependencies on external library and a modular architecture. Ehcache is released under Apache Software License Version 2.0.

6.1.6 Gigaspace XAP

http://www.gigaspaces.com/xap

Latest release 7.0.2, Decempber 2009.

GigaSpaces says "*XAP In-Memory Data Grid delivers an in-memory cache for fast data access, and an advanced distributed cache for extreme performance and scalability.*" XAP is one of the few products exposing the JavaSpaces API.

GigaSpaces XAP is offered in a variety of license models, including perpetual, annual subscription, and pay per use. Free developer and academic editions available.

6.1.7 GemStone GemFire Enterprise

http://www.gemstone.com/products/gemfire

Latest release 6.0.1, April 2009.

GemFire Enterprise is defined as an "*in-memory distributed data management platform*".

Production licenses are usually node-locked and limited to a fixed number of CPUs, but other licensing models can be negotiated. Evaluation expiring evaluation licenses available upon request. A development licenses for development and testing only.

6.1.8 Jakarta JCS

http://jakarta.apache.org/jcs/

Latest formal release 1.3, June 2007; snapshot 1.3.3.2, June 2009.

Jakarta JCS (Java Caching System) is "*a distributed caching system*". Although the term "Data Grid" in not used to describe the product, it share many features with IMDG. JCS has a dependency from Doug Lea's `concurrent` package [Lea04].

JCS is released under Apache Software License Version 2.0.

6.2 Data storage

API: from the programmers viewpoint, the API are the most visible point. Main API style are illustrated in Section 2.3, as `java.util.Map`, JSR-107 JCache, and JavaSpace/TupleSpace. In addition, an implementation might support the `j.u.c.ConcurrentMap` API, an extension of `Map` which add the atomic non-locking operations based on the *Compare-and-swap* (CAS) technique *`putIfAbsent(key, value)`*, *`remove(key, value)`* , *`replace(key, value)`* and *`replace( key, oldValue, newValue)`*. This interface avoid using locks to execute common procedure such as checking the presence of a key value before to save a new entry.

Entry container name: almost all the products use the term "entry" to identify the key-object pair as elementary storable element, but there is not accordance on the name to assign to the elementary container of entries, which has the same conceptual role of table in relational databases.

Serialization: in order to be transmitted over the wires, Java object must be *serialized*. Many Data Grid implementations also store in the partition a serialized version of the object instead of an handle to its representation in the heap memory. Java includes a standard serialization mechanism, but it is know as inefficient [vNMH+02], thus many implementation uses a custom serialization format. In this case, programmers can advantages of this implementing externalization method as preferred by the implementation.

Pluggable serialization: the serialization mechanisms can be customized using plugin.

Passivation: the capability of deactivate an entry by moving it from memory to backend storage when certain condition are meet (usually analogues to the eviction policy).

Multimap: a MultiMap is a specialized map where a key can be associate with multiple values. The benefit of specialized implementation of MultiMap instead of using something such as `Map<K, List>` in Data Grids is tied to serialization: adding or removing an element form a `List` stored as an entry of a `Map` requires the whole list being deserialized then serialized and stored again after the update. For long list, this process is very expansive. MultiMap avoid this problem associating whit the key a list of the serialized version of elements.

Query and indexing: when data partitions are spreads across the cluster, iterate over the entries and look for ones meeting some criteria is possible but very inefficient, especially in those implementation which not offers parallel distributed execution. Many Data Grids im-

plement some support for queries. SQL-like, XPath, LDAP, etc... No current products full supports SQL, specifically multi-table operations are not available.

Unique id generator: a function to generate an identifier unique within the whole cluster.

	Oracle Coherence	IBM eXtreme Scale	JBoss Infinispan	Hazelcast	Ehcache	Gigaspace XAP	GemStone GemFire	Jakarta JCS
API style	Map	1)	ConcurrentMap	ConcurrentMap 2)	3) JSR107	Map Jini JDBC	Map	custom
Entry container name	cache, named cache		cache	map	cache	space	data region 4)	cache
Serialization			River	custom		Custom 5)	custom	
Pluggable serialization	X							X
Passivation					X			
Multimap				X			6)	
Query and indexing	filters	OGQL, 4)		SQL-like, JPA		SQL-like, XPath	OQL	Index, pattern
Unique id generator				X				

Notes:

1) WebSphere eXtreme Scale exposes two API, `ObjectMap` and `EntityManager`, similar to Map but not compatible with it.
2) Hazelcast also implements the distributed version of other Java standard data structures, specifically `java.util.Set`, `java.util.List`, `java.util.Queue`, and their concurrent versions.
3) Implements a proprietary API in which entries (`Element`) must be create explicitly.
4) GemFire documentation define as “cache” the whole entry set handled by a node, which can include several regions; regions can be nested, which produces similar effect of MultiMaps.
5) XAP can control the serialization mode using different mechanisms.
6) WebSphere eXtreme Scale supports the Object Grid Query Language (OGQL) which is similar to JPQL.

6.3 Networking

Multicast discovery: in most of implementation, IP multicast is the default protocol to the discovery process. During discovery, nodes locate each other and negotiate to join the distributed system. The membership and discovery facility keeps also track of the membership list and makes the members aware of the identities of the other members in the distributed system.

Client/server: in this configuration, clients and servers are organized into separate distributed system, usually communicating through network. The cluster could also support non-Java clients or standard protocols.

P2P: in this context, a In-Memory Data Grid provide a Peer-to-Peer (P2P) architecture if the client and the server might run in the same JVM exchanging data as intra-JVM object. From the programmers perspective, this means that an application moved on another host without a network connection work as-is.

Super peer: or "super client", or "lite client" model, where client application is cluster member with no storage. The client may employ an near cache.

Hub: in case of complex networks, in which not all the nodes are mutually reachable, a node can act as store&forward gateway to another remote node over segmented LAN or WAN.

JGroups: it is a toolkit for reliable multicast communication over different protocol stack, not only IP Multicast. It has NAT-traversal capability, joining and leaving handling, notification about joined/left/crashed members, point-to-multipoint and point-to-point messaging. It is used as base library by many projects. Thus all that projects have similar configurations syntax and support similar capabilities. Development of JGroups has been started by Bela Ban during his post-doc at Computer Science Department at Cornell University, in 1999. More information on http://www.jgroups.org/ (last accessed 07 Feb 2010)

JMS: a *Java Message Service* (JMS) [jms], usually in a Publish/Subscribe configuration, can be used to pushes changes between nodes or from nodes to client near cache.

	Oracle Coherence	IBM eXtreme Scale	JBoss Infinispan	Hazelcast	Ehcache	Gigaspace XAP	GemStone GemFire	Jakarta JCS
Multicast discovery	X		1)	X	X		X	X
Client/Server	C++ .NET	native REST	REST		SOAP REST	C++ .NET	C++ .NET	RMI
P2P	X	2)	X	X			X	X
Super peer	3)			X		X	X	
Hub						X	X	
JGroups			X		4) 5)			X
JMS		X			4)	6)		

Notes:

1) Based on JGroups.
2) eXtreme Scale can be configured in a P2P topology regarding the nodes partition stores, but still requires a catalog server.
3) Partitioned configuration can disable local storage, resulting in a client-only node.
4) Optional.
5) Also Terracotta clustering.
6) Including server features.

6.4 Data distribution

Distributed cache and In-Memory Data Grids could adopt several topologies to spread the data across the nodes. Replication has been studied extensively and different distributed distribution strategies have been proposed in the literature:

Data replication: all data are fully replicated to all cluster members. for availability (and performance)

Data partitioning: the full key-space is divided in a fixed number of virtual *blocks* (or *segments* or *shards*). Then each block is assigned to to a node (*owner* of the block); in case of fault tolerant partitioning, one or more *backup* (or *replica*) copies of the block are assigned to other nodes. Each entry is store in the node owning the corresponding key plus in all the nodes hosting the backups. A common assignment strategy is set the block count to a prime number, then having a *coordinator* or *catalog* node (usually the oldest) which assign the ownership of each block and backups to the nodes. In pure Java implementations, entries belong to the block resulting form a computation such as `key.hashCode() % blockCount` or similar.

Near caching: (or *Local*, or *L1*): a local view proxy maintains a subset of the partitions data, allowing the client to read distributed data without any remote operations.

Memory Management and Eviction Policy: when memory is about to exhausting, the Data Grid could discharge (*evict*) some entries to make room for newer ones. Common policies are *perpetual* (never discharge), well known *FIFO*, *LRU* (*Least Recently Used*), *LFU* (*Least Frequently Used*), …. Usually eviction policy can be configured per-region, in some case, also per-entry. Some implementations allow to plug-in a custom provider for the eviction policy. Products which not implement a perpetual eviction strategy are not included in this survey, since are not suitable as Data Grid.

Moreover, also the mechanism to propagate changes in data could be:

- Data copy: new elements placed in a cache and element removals are replicated are replicated to others nodes
- Data destroy: an entry is removed completely from the cache with a distributed synchronous the operation
- Data invalidation: the entry value is set to null. Being entry is invalid, a subsequent `get()` causes the cache to retrieve the value from the original source
- Write-Through and Write-Behind

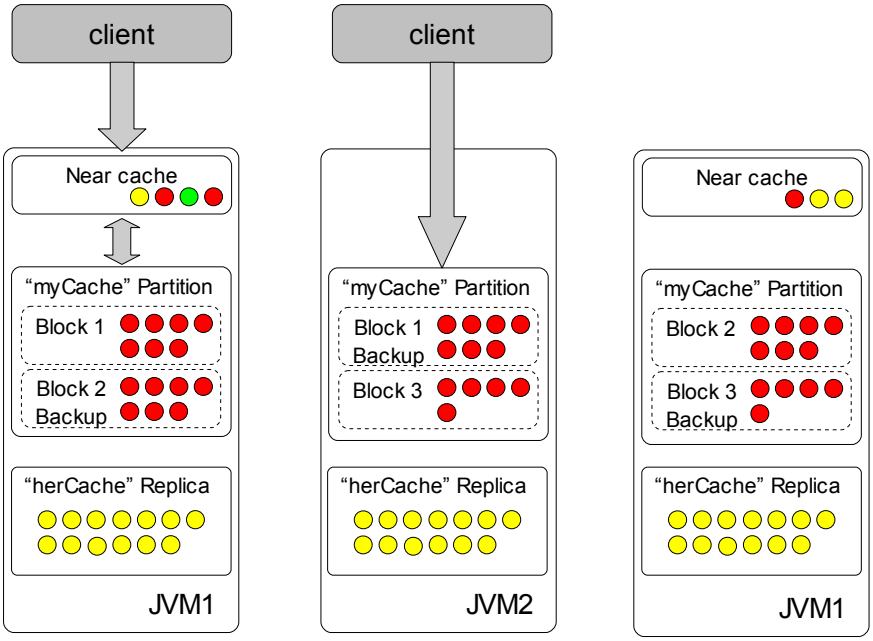


Figure 37: Possible data distribution topologies in Data Grid

Batch Write: especially in conjunction with a persistent backend, the ability of bundling into a single operation many different updates. In some implementations, the changes might also coalesced into a single backend operation if occurred to the same piece data. E.g. in case two successive update of the same entry happened before the Write-Behind delay, only the latest could be apply to the backend storage.

Asynchronous replication: to guarantee the consistency among distributed cache, all the products support synchronous operations. These are a serious bottleneck, since all are supported between partitions of the distributed cache (primarily used to synchronize partitions with their backup copies on other machines).

Persistent backend storage: allows to load and store the map entries from and to a persistent storage such as relational database. When an entry is retrieve via its key and it does not exist in-memory, the engine will try to load the entry from the storage. Similarly, when an entry is stored on the cache, the engine will also store it into the backed. Storing can be perform synchronously (write-through) with no-delay or asynchronously (write-behind) after a configurable delay.

Cache initialization: or "warm startup" is the capability of read the initial state from an external source. This is a distinct feature from persistent backend, since the data are only read, and not saved on the backend, and all the data are read, not only the entries occurring in cache miss. This feature speed up cache initialization because usually skip any lock control.

	Oracle Coherence	IBM eXtreme Scale	JBoss Infinispan	Hazelcast	Ehcache	Gigaspace XAP	GemStone GemFire	Jakarta JCS
Data replication	X		X	1)	X	X	X	X
Data partitioning	X		X	X		X	X	
Near cache	X	X	X	X	X	X	X	
Eviction policies	LRU LFU	TTL LRU LFU memory			LRU LFU FIFO	LRU FIFO All-In-Cache memory	LRU Heap limit TTL	LRU
Per-entry policies					X			X
Pluggable policy	X	X			X			
Write-Through	X			X		X		
Write-Behind	X			2)		X		
Batch Write	X		3)			X	X	
Asynchronous operations	4)		X		X	X	X	
Persistent backend	Disk Database		many	plugin	Disk	Hibernate		disk
Cache initialization			X		X	X		JDBC plugin

Notes:

1) The same result can be obtained setting the backup replica number to `Integer.MAX_VALUE`
2) Only writing on the persistent backend
3) Coalesced changes also supported
4) As backend for distributed partitions.

6.5 Transactions and database integration

Distributed locks: are usually acquired on a key. Usual algorithms are pessimistic, optimistic and *Multiversion concurrency control* (MVCC) locking. Cluster wide (*eagerly*) must be supported, in some cases also supported local-only locking associated to cluster wide transactions.

Time-bound locks: sometimes associated to deadlock detection.

Transactions isolation: advanced implementation offers other transaction isolation levels besides the *serializable* one provided by locks. Some Data Grid provides *repeatable read*, in which keyset and entryset cannot change once selected from the map, or *read committed*, in which data retrieved by a selection may be modified by some other transaction and became visible when it commits.

Transaction manager: support to cooperate in distributed transaction manager

Trigger: allows to validate, reject or modify mutating operations against a map.

Cache plugins: many data access frameworks include the support for a pluggable second cache manager to be used below the persistence layer and completely transparent to the application. The Data Grid could be include plugins for frameworks such as Hibernate and JPA.

	Oracle Coherence	IBM eXtreme Scale	JBoss Infinispan	Hazelcast	Ehcache	Gigaspace XAP	GemStone GemFire	Jakarta JCS
Distributed locks	X		X	X				
Time-bound locks	X			X		X		
Transactions isolation	Repeatable-read Read committed		Repeatable-read Read committed	Read committed		1)	Repeatable-read	
Transaction manager	JTA JCA XA	J2EE	JTA	J2EE JCA	JTA	JTA Spring	J2EE JTA	
Triggers	X		2)				3)	
Cache plugin		Hibernate JPA	Hibernate	Hibernate	Hibernate			

Notes:

1) Provided by Spring Framework.
2) Interceptors.
3) Used in authorization.

6.6 Data affinity, data routing and fault recovery

Data affinity: IMDG providing *data affinity*, or *co-location*, could ensure that entries related in the same group is contained within the same data partition. For example, in a master-detail pattern such as an "Order-Item", the entire collection of Item objects belonging to an Order may be co-located in the same data partition of the Order object.

Data affinity ensures that all relevant data is managed on a single primary cache node. In some implementations, affinity may span multiple partitions managed by the same host. Usually, data affinity is specified in terms of keys, not values.

Main benefit of data affinity are:

- only a single node is required to manage queries and transactions against a set of related items;
- all concurrency operations can be managed locally, avoiding the need for clustered synchronization.

Zones: allows for rules-based block allocation, enabling optimized topology for Grids spanning across physical locations. As an example, nodes in the same zone could have a synchronous replication and nodes in different zoned could be asynchronously replicated.

Fault recovery: mechanisms for automatically recovery from cluster errors.

	Oracle Coherence	IBM eXtreme Scale	JBoss Infinispan	Hazelcast	Ehcache	Gigaspace XAP	GemStone GemFire	Jakarta JCS
Data affinity	X		X				1)	
Zones		X	2)			X		
Fault recovery	X					X		X

Notes:

1) Data regions (maps) can themselves be nested and contain child data regions.
2) In compatibility mode with JBoss 3, will be discontinued.

6.7 Event and messaging

Cluster events: listeners for membership events are notified when members added or leave the cluster

Partition events: listeners for partition events are notified when primary partitions, replicas, or block, is created, moved or disposed.

Map/cache events: listeners for map/cache events are notified when a storage has finished pre-loading, or an entry is stored or evicted.

Entry events: listeners for entry events are notified when a specific entry is modified.

Local entries events: notifications are provided for events occurring in the local partition only; some implementations can emulate this feature using a cluster-wide listeners, which receive all events, and a filter to ignore not local ones.

Continuous query: with the continuous query facility, the clients application registers a listener associated to a query expressions. Then, events are sent to client listeners any time a change in the data cluster satisfies the query.

	Oracle Coherence	IBM eXtreme Scale	JBoss Infinispan	Hazelcast	Ehcache	Gigaspace XAP	GemStone GemFire	Jakarta JCS
Cluster events	X			X			X	
Partition events		X					X	
Map/cache events	X	X	X	X	X	X	X	X
Entry events						X	X	
Local entry events						X		
Continuous query	X					X	X	

6.8 Distributed and data-aware execution

Task name: there is not accordance on the name of the elementary execution unit. Common choices are *function*, *job*, or *task*. The latter two are cause of confusion, since system providing a way to split the work unit into smaller parts, also named them jobs or tasks. As example, for GridGain "Grid *task* gets split into *jobs* when `GridTask.map(List, Object)` method is called" [32], while `JPPFJob` class in JPPF "represent a JPPF submission and hold all the required elements: *tasks*, execution policy, task listener, data provider"[33]. In this dissertation, a basic indivisible work unit is identify as *task*, and an collection of correlate tasks is called *job*. Sometimes also the term "agent" is used, but normally the *agent* is the service component running in daemon mode on remote host which handle the incoming requests.

Distributed executor: distributed executors automatically span a single task submission among all or specified nodes.

ExecutorService: the system exposes an API compatible with the interface `java.util.concurrent.ExecutorService`,

Task cancellation: the remote task can be cancelled.

Execution callbacks: even in products where the task execution is asynchronous in nature, it is possible that a call to get a result is a blocking (either indefinitely or for a specific timeout). Products supporting callbacks adds the ability to register an listener which will be executed once a result arrives.

Targeted execution: the ability of execute an agent (or task) against an entry in any map of data, sending it to the Grid node owning the entry and then executing it locally at that node. In many cases, it is much more efficient to move the serialized form of the agent (usually a few hundred bytes) than moving the data to the execution host, handling distributed concurrency control, coherency and data updates. The agent can be routed according to the key of the entry, or according to a query result, or to a specific node.

Parallel execution: if the agent is targeted to multiple key, owned by different nodes, some IMDGs allow the parallel execution of the task on each node. To be parallel, it is required the execution could be truly asynchronous: some framework, such as Hazelcast, present an

32 http://www.gridgainsystems.com/wiki/display/GG15UG/Grid+Tasks+And+Grid+Jobs (last access 4 Aug 2009)

33 http://www.jppf.org/api/org/jppf/client/JPPFJob.html (last access 4 Aug 2009)

asynchronous interface to the programmer, based on `java.util.concurrent.Future`, but internally handle the task synchronously, not exploiting parallelism.

MapReduce: some IMDG includes a framework or template classes for the MapReduce programming model.

MapReduce generics: Map and Reduce classes or parameters are defined using Java generics.

Fault tolerance: when a task is invoked from a client and the original request fails due to a server-side issue (the node owning the target key goes down, or a network partition), the client or another peer automatically retries the function execution. Since the task execution service is usually not transactional, this can result in multiple execution or partially executed results.

Integration with computational grids: some products offers out-of-the-box can integrate with computational oriented grids, such as GridGain or JPPF. Integration support could be offered by the computational grid.

	Oracle Coherence	IBM eXtreme Scale	JBoss Infinispan	Hazelcast	Ehcache	Gigaspace XAP	GemStone GemFire	Jakarta JCS
Task name	Agent, Entry Processor	agent		Callable Task		Processing Unit, task	function	
Distributed executor	X			X		X	X	
ExecutorService				X		X		
Task cancellation	X			X		X		
Execution callbacks	X			X		X		
Targeted execution	Key based, node based, query based	Query based		Key based, node based		Key based	Map based Node based Query based	
Parallel execution	1)	X				X	X	
MapReduce	X	X				X	X	
MapReduce generics						X		
Fault tolerance	X					X	X	
Computation grid integration	GridGain		2)			GridGain JPPF		

Notes:

1) Parallel execution is available with Enterprise and Grid editions only
2) GridGain has an adapter for JBoss cache

6.9 Security

Authenticated join to the cluster: a node must present a valid token to join the cluster; mostly used in P2P architectures

Autenticated access and user management: a client application must present a valid token to access and operate in the Data Grid; mostly used in client/server architecture.

Authorization level or roles: client can be authorized to specific operations; as example, clients can be authorized or not to perform insert, read, update, invalidate, or delete operations on cache or to perform queries, etc... Administration authorization are a different topic. Many P2P implementation, once allowed the access using a token, have no limitation on the action the client can perform. Authorization is often delegated to a standard access manager, such as *Java Authentication and Authorization Services* (JAAS).

Traffic encryption: supported protocols.

	Oracle Coherence	IBM eXtreme Scale	JBoss Infinispan	Hazelcast	Ehcache	Gigaspace XAP	GemStone GemFire	Jakarta JCS
Authenticated join	X	shared keys		password		X	shared keys SSL LDAP	
Authentication	X	Password SSL certificate Kerberos				X	Password SSL certificate LDAP	
Authorization	JAAS	JAAS Tivoli Access Manager				X	X	
Traffic encryption	shared keys certificate	TLS/SSL		SSL		SSL	SSL	

6.10 Management

Demo client: the toolkit includes a demo client for testing purpose.

Admin console: description of the management console.

JMX: supports Java Management Extensions (JMX) Agent.

	Oracle Coherence	IBM eXtreme Scale	JBoss Infinispan	Hazelcast	Ehcache	Gigaspace XAP	GemStone GemFire	Jakarta JCS
Demo client	CLI		GUI	CLI				
Admin console	JMX	web	JOPR		web	CLI GUI	CLI API	
JMX	X		X	1)	X	X	X	

Notes:

1) Partial support

7 Appendix: Amendment One to the OTN License

AMENDMENT ONE
TO THE
ORACLE TECHNOLOGY NETWORK DEVELOPER LICENSE TERMS
("Oracle Coherence")
BETWEEN
MARCO FERRANTE
AND
ORACLE AMERICA, INC.

This document ("Amendment One") amends the Oracle Technology Network Developer License Terms (the "Agreement"), between Marco Ferrante ("Ferrante") and Oracle America, Inc. ("Oracle"). This Amendment shall become part of and subject to the terms and conditions of the Agreement which, except as modified herein, remains unchanged and in full force and effect. In the event of any conflict between the terms of this Amendment and the Agreement, the terms of this Amendment shall govern.

All terms not otherwise defined herein shall have the same meaning ascribed to them in the Agreement.

WHEREAS, the parties desire to amend the provisions of the Agreement;
NOW, THEREFORE, in consideration of the foregoing recitals, and for good and valuable consideration, the receipt and sufficiency of which are hereby acknowledged, intending to be legally bound hereby, the undersigned parties agree as follows:

1. Notwithstanding anything to the contrary in the Agreement, Oracle consents to Ferrante's disclosure of benchmark tests related to the "Oracle Coherence" program, provided, however, any such disclosure shall be limited to Ferrante's Ph.D. candidacy and final dissertation in Computer Science, focusing on "In-Memory Data Grids", at the University of Genoa, Italy.

2. This Amendment One may be executed on facsimile copies in two counterparts, each of which shall be deemed an original and all of which together shall constitute one and the same Amendment One. Notwithstanding the foregoing, each party shall deliver original executed copies of this Amendment One to the other party as soon as practicable following execution thereof.

Subject to the modifications herein, the Agreement shall remain in full force and effect.

The Effective Date of this Amendment One is the later date as signed below.

MARCO FERRANTE

By: _____

Name: M. Ferrante

Date: 17th May 2010

ORACLE USA, INC.

By: _____

Name: Michael Poplack

Title: VP. Legal + Assoc. GC.

Date: 6/17/10

8 References

[ABCD+08] Brian Amedro, Vladimir Bodnartchouk, Denis Caromel, Christian Delbe, Fabrice Huet, and Guillermo L. Taboada. *Current State of Java for HPC*, INRIA Technical Report RT-0353, 2008. http://hal.inria.fr/inria-00312039/en (last access 23 July 2009)

[ABJ05] Gabriel Antoniu, Luc Bougé, and Mathieu Jan. "JuxMem: An Adaptive Supportive Platform for Data Sharing on the Grid", in *Scalable Computing: Practice and Experience*, vol. 6, pp.45-55, Sep 2005. See also http://juxmem.gforge.inria.fr/ (last accessed 26 Nov 2009)

[ABN09] Holger Arndt, Markus Bundschus, and Andreas Nägele. "Towards a Next-Generation Matrix Library for Java", *33rd Annual IEEE International Computer Software and Applications Conference* (COMPSAC) 2009.

[AC04] Cosimo Anglano and Massimo Canonico. "A Comparative Evaluation of High-Performance File Transfer Systems for Data-intensive Grid Applications", *13th IEEE International Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE-2004), Emerging Technologies for Next-Generation Grid Track (ETNGRID)*, Modena, Jul 2004.

[ACGB+08] Cosimo Anglano, Massimo Canonico, Marco Guazzone, Marco Botta, Sergio Rabellino, Simone Arena, Guglielmo Girardi. "Peer-to-Peer Desktop Grids in the Real World: The ShareGrid Project", *Cluster Computing and the Grid CCGRID '08*, 2008.

[ACKL+02] David P. Anderson, Jeff Cobb, Eric Korpela, Matt Lebofsky, and Dan Werthimer. "SETI@home: An Experiment in Public-Resource Computing", *Communications of the ACM*, vol. 45, n. 11, Nov 2002. Available on line http://setiathome.berkeley.edu/sah_papers/cacm.php (last accessed 18 Dec 2009)

[AM02] Christophe Ambroise and Geoffrey McLachlan, "Selection bias in gene extraction on the basis of microarray gene-expression data", *Proceedings of the National Academy of Sciences*, vol. 99, n. 10, p. 6562, 2002.

[And04] David P. Anderson. "BOINC: A System for Public-Resource Computing and Storage", *in Proceedings of 5th IEEE/ACM International Workshop on Grid Computing*, Nov 2004.

[app] Accelrys Pipeline Pilot, http://accelrys.com/products/scitegic/ (last accessed 11 Jan 2010)

[Arn09] Holger Arndt. "The Java Data Mining Package – A Data Processing Library for Java", *33rd Annual IEEE International Computer Software and Applications Conference (COMPSAC)*, 2009.

[BBBC+06] Rosa M. Badia, Olav Beckmann, Marian Bubak, Denis Caromel, Vladimir Getov, Ludovic Henrio, Stavros Isaiadis, Vladimir Lazarov, Maciek Malawski, Sofia Panagiotidi, Nikos Parlavantzas, and Jeyarajan Thiyagalingam. *Lightweight Grid Platform: Design Methodology*, CoreGRID Institute on Systems, Tools, and Environments, technical report TR-0020, Jan 2006.

[BCDG+08] Michael R. Berthold , Nicolas Cebron, Fabian Dill, Thomas R. Gabriel, Tobias Kötter, Thorsten Meinl, Peter Ohl, Christoph Sieb, Kilian Thiel, and Bernd Wiswedel. "KNIME: The

Konstanz Information Miner", *Data Analysis, Machine Learning and Applications*, pp. 319-326, Springer Berlin Heidelberg, 2008. see also KNIME - Konstanz Information Miner. http://www.knime.org/ (last access 6 Jan 2010)

[BDLK] Kanishka Bhaduri, Kamalika Das, Kun Liu, and Hillol Kargupta. *The Distributed Data Mining Bibliography*, http://www.cs.umbc.edu/~hillol/DDMBIB/ (last access 31 Jan 2010)

[BF09] Annalisa Barla and Marco Ferrante. "Deployment of a Regularized Feature Selection Framework on an Overlay Desktop Grid", *International Workshop on High Performance Computational Systems Biology HiBi '09*, Trento (I), pp. 103-104, 2009.

[BFSV09] Curzio Basso, Marco Ferrante, Matteo Santoro, and Alessandro Verri. "Automatic annotation of 3D multi-modal MR images on a Desktop Grid", in *Proceedings of the MICCAI-Grid 2009 Workshop*, London (UK), 2009. Available online http://www.i3s.unice.fr/~johan/MICCAI-Grid09/MICCAI-Grid_2009_Proceedings_FINAL.pdf (last accessed 7 Nov 2009)

[BHKM+04] Rosa Badia, Robert Hood, Thilo Kielmann, Christine Morin, Stephen Pickles, Massimo Sgaravatto, Paul Stodghill, Nathan Stone, Heon Y. Yeom. "Use-Cases for Grid Checkpoint and Recovery", Global Grid Forum Request for Comments GWD-XXX-00x-7, Nov 2004.

[Bre00] Eric Brewer, *Towards Robust Distributed Systems*, keynote at ACM Symposium on Principles of Distributed Computing PODC, 2000. Available online http://www.cs.berkeley.edu/~brewer/cs262b-2004/PODC-keynote.pdf (last accessed 15 Aug 2009)

[Bul01] Mark Bull, Lorna Smith, Lindsay Pottage, and Robin Freeman. "Benchmarking Java against C and Fortran for Scientific Applications", *Proceedings of the 2001 joint ACM-ISCOPE conference on Java Grande*, Palo Alto, California, pp. 97-105, 2001

[cacm] Special issue "End-user development", *Communications of the ACM*, vol. 47, issue 9, September 2004. Available under subscription http://cacm.acm.org/magazines/2004/9 (last accessed 27 Oct 2009)

[Cap07] Franck Cappello, "3rd generation desktop grids", in *Proceedings of the 1st XtremWeb Users Group Workshop* (XW'07), Hammamet, Tunisia, Feb 2007.

[casc] Cascading, http://www.cascading.org/ (last accessed 10 Feb 2010)

[CBKB+08] Sungjin Choi, Rajkumar Buyya, Hongsoo Kim, Eunjoung Byun, Maengsoon Baik, Joonmin Gil, and Chanyeol Park. *A Taxonomy of Desktop Grids and its Mapping to State-of-the-Art Systems*, Technical Report GRIDS-TR-2008-3, Grid Computing and Distributed Systems Laboratory, The University of Melbourne, Australia, Feb. 20, 2008.

[CBSA+03] Walfredo Cirne, Francisco Brasileiro, Jacques Sauvé, Nazareno Andrade, Daniel Paranhos, Elizeu Santos-Neto e Raissa Medeiros. "Grid Computing for Bag of Tasks Applications", *Third IFIP Conference on E-Commerce, E-Business and E-Goverment*, São Paulo, Sep 2003.

[CFKS+01] Ann Chervenak, Ian Foster, Carl Kesselman, Charles Salisbury, and Steven Tuecke. "The Data Grid: Towards an Architecture for the Distributed Management and Analysis of Large Scientic Datasets", *Journal of Network and Computer Applications*, v. 23, pp. 187–200, 2001.

[CG89] Nicholas Carriero and David Gelernter. *How to write parallel programs: a guide to the perplexed*, ACM Computing Surveys, vol. 21, issue 3, pp. 323-357, Sep 1989.

[CKLY+06] Cheng-Tao Chu, Sang Kyun Kim, Yi-An Lin, YuanYuan Yu, Gary Bradski, Andrew Y.

Ng, and Kunle Olukotun. “Map-Reduce for Machine Learning on Multicore”, in P*roceedings of NIPS 19*, 2006.

[CKV98] Denis Caromel, Wilfried Klauser, and Julien Vayssiere. “Towards Seamless Computing and Metacomputing in Java”, in *Concurrency: Practice and Experience* (editor Geoffrey C. Fox), vol. 10, pp.1043-1061, John Wiley & Sons, Ltd., Sep.-Nov. 1998.

[CLHK+06] Jeffrey C. Carver, Lorin M. Hochstein, Richard P. Kendall, Taiga Nakamura, Marvin V. Zelkowitz, Victor R. Basili, and Douglass E. Post. “Observations about Software Development for High End Computing”, *CTWatch Quarterly*, vol. 2, n. 4A, Nov 2006. http://www.ctwatch.org/quarterly/articles/2006/11/observations-about-software-development-for-high-end-computing/ (last accessed 26 Oct 2009)

[CM02] Sebastian Celis and David R. Musicant. *Weka-Parallel: Machine Learning in Parallel*, Carleton College, Technical Report, 2002.

[Coh09] Uri Cohen. *Inside GigaSpaces XAP*, technical paper, 2009. See also http://www.gigaspaces.com/wiki/display/XAP7/7.0+Documentation+Home (last accessed 9 Nov 2009)

[cohe] Oracle Coherence. http://www.oracle.com/technology/products/coherence/index.html (last accessed 12 Feb 2010)

[CSGA07] Kelvin Cardona, Jimmy Secretan, Michael Georgiopoulos, and Georgios Anagnostopoulos. *A Grid Based System for Data Mining Using MapReduce*, Technical Report TR-2007-02, The AMALTHEA REU Program, 2007. Available on line http://www.amalthea-reu.org/pubs/amalthea_tr_2007_02.pdf (last accessed 04 Jan 2010)

[CW02] Robert C. Seacord and Lutz Wrage. *Replaceable Components and the Service Provider Interface*, Technical Note CMU/SEI-2002-TN-009, Jul 2002.

[Deu94] Peter Deutsch. *The Eight Fallacies of Distributed Computing*, 1994, see also http://blogs.sun.com/jag/resource/Fallacies.html (last accessed 5 Aug 2009)

[DG04] Jeffrey Dean and Sanjay Ghemawat. “MapReduce: Simplified Data Processing on Large Clusters”, *OSDI'04: Sixth Symposium on Operating System Design and Implementation*, San Francisco, Dec 2004.

[DMDV+08] Augusto Destrero, Sofia Mosci, Christine De Mol, Alessandro Verri, and Francesca Odone. "Feature selection for high dimensional data", *Computational Management Science*, 2008. Available on line ftp://ftp.disi.unige.it/person/MosciS/PAPERS/CompManSc.pdf

[DMTV09] Christine De Mol, Sofia Mosci, Magali Traskine, and Alessandro Verri. "A regularized method for selecting nested groups of relevant genes from microarray data", *Journal of Computational Biology*, vol. 16, pp. 1-15, Apr 2009.

[DvNH06] Niels Drost, Rob V. van Nieuwpoort, and Henri E. Bal. “Simple locality-aware co-allocation in peer-to-peer supercomputing”, in *GP2P: Sixth International Workshop on Global and Peer-2-Peer Computing*, Singapore, May 2006.

[DWS08] David DeWitt; Michael Stonebraker. "MapReduce: A major step backwards" blogpost, Database Column, http://databasecolumn.vertica.com/database-innovation/mapreduce-a-major-step-backwards/ (last accessed 8 Feb 2010)

[DZLC04] Janez Demšar, Blaž Zupan, Gregor Leban, and Tomaz Curk. "Orange: From Experimental Machine Learning to Interactive Data Mining", in *Proceedings of the 8th European*

Conference on Principles and Practice of Knowledge Discovery in Databases PKDD 2004, Pisa, Italy, Sep 2004. See also http://www.ailab.si/orange/ (last accessed 21 Jan 2010)

[Eat02] John W. Eaton. GNU Octave Manual, Network Theory Limited publisher, 2002. See also http://www.octave.org/ (last access 8 Feb 2010)

[eclp] The Eclipse Project. http://www.eclipse.org/ (last access 5 Oct 2009)

[ehc] Terracotta Ehcache. http://ehcache.org/ (last accessed 26 Nov 2009)

[Fer00] Marco Ferrante. "Accedere ai servizi di directory con LDAP"*, Computer Programming*, n. 93, pp. 78-81, Jul 2000.

[Fer08] Marco Ferrante. "JXTA: a dead end way to Grid?", *JM4Grid 2008*, Genoa, 21 Feb 2008.

[Fer09] Marco Ferrante. "Grid semplificato con Java", *Computer Programming*, n. 17, vol. 4, pp. 48-55, Oct 2009.

[FL09]Marco Ferrante and Laura Lo Gerfo. Annotazione automatica di immagini con sistemi desktop grid, DISI Technical Report DISI-TR-09-04, Sep 2009.

[Flo99] Martin Fowler. *Refactoring. Improving the Design of Existing Code*, Addison-Wesley, 1999.

[Fos05] Ian Foster. "Globus Toolkit Version 4: Software for Service-Oriented Systems", *IFIP International Conference on Network and Parallel Computing*, Springer-Verlag LNCS 3779, pp 2-13, 2005.

[GCCC85] David Gelernter, Nicholas Carriero, Sarat Chandran e Silva Chang. "Parallel Programming in Linda", in *Proceedings of the IEEE International Conference on Parallel Processing*, Aug 1985.

[gem05] GemStone Systems, Inc. *GemFire Enterprise Data Fabric Facilitates Agile Risk Management at a Financial Service Provider*, Sep 2005

[jsdl] Global Grid Forum. *Job Submission Description Language (JSDL) Specification, Version 1.0*, Nov 2006. Available on line http://www.gridforum.org/documents/GFD.56.pdf (last accessed 26 Nov 2008)

[ggp] GridGain platform. http://www.gridgain.com/ (last accessed 9 Nov 2009)

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994.

[GL02] Seth Gilbert and Nancy Lynch. "Brewer's Conjecture and the Feasibility of Consistent Available Partition-Tolerant Web Services", *ACM SIGACT News*, vol. 33, Jun 2002.

[Goe05] Brian Goetz. "Java theory and practice: Urban performance legends, revisited", *IBM developerWorks*, Sep 2005, http://www.ibm.com/developerworks/java/library/j-jtp09275.html (last accessed 1 Dec 2009)

[Goe06] Brian Goetz. *Java Concurrency in Practice*. Addison-Wesley, 2006.

[gsg] *GemStone GemFire*. http://www.gemstone.com/products/gemfire/ (last accessed 26 Nov 2009)

[Ham00] Howard Hamilton. *C4.5 Tutorial*, 2000 http://www2.cs.uregina.ca/~dbd/cs831/notes/ml/dtrees/c4.5/tutorial.html (last accessed 1 Jan 2010)

[Ham04] Graham Hamilton. *Multithreaded toolkits: A failed dream?* blogpost http://weblogs.java.net/blog/kgh/archive/2004/10/multithreaded_t.html (last access 31 Jan 2010)

[haz] *Hazelcast*. http://www.hazelcast.com/ (last accessed 26 Nov 2009)

[HKY93] Tim Howes, Steve Kille, and Wengyik Yeong. *Lightweight Directory Access Protocol*, IETF 1777, 1993.

[ibm] IBM WebSphere eXtreme Scale. http://www-01.ibm.com/software/webservers/appserv/extremescale/ (last accessed 27 Nov 2009)

[IG96] Ross Ihaka and Robert Gentleman. “R: A Language for Data Analysis and Graphics”, *Journal of Computational and Graphical Statistics*, v. 5, n. 3, pp. 299-314, 1996.

[Ime08] Slava Imeshev. *Distributed Cache Latency and JVM Heap Size* blogpost, Aug 09, 2008, http://www.jroller.com/imeshev/entry/effect_of_jvm_heap_size (last accessed 10 Jan 2010)

[jama] The MathWorks and the National Institute of Standards and Technology (NIST). *JAMA: A Java matrix package*, 2005. http://math.nist.gov/javanumerics/jama/ (last accessed 1 Nov 2009)

[jbi] JBoss Infinispan. http://www.jboss.org/infinispan (last accessed 26 Nov 2009)

[jini] JINI. http://www.jini.org/wiki/Main_Page (last access 30 April 2009)

[jms] Sun Mycrosystems. Java Message Service (JMS) specification, http://java.sun.com/products/jms/ (last access 28 Jan 2010)

[jppf] Java Parallel Processing Framework. http://www.jppf.org/ (last access 20 May 2009)

[jsr107] *JSR 107: JCACHE - Java Temporary Caching API*. http://jcp.org/en/jsr/detail?id=107 (last accessed 26 Nov 2009)

[kep] *Kepler Project*. http://kepler-project.org/ (last access 15 Nov 2009)

[knc] KNIME Cluster Execution. http://www.knime.com/products/knime-cluster-execution (last accessed 21 Jan 2010)

[KZK04] Rinat Khoussainov, Xin Zuo, and Nicholas Kushmerick. "Grid-enabled Weka: A Toolkit for Machine Learning on the Grid", *ERCIM News*, n. 59, Oct 2004.

[Lau03] Francis C. M. Lau. “Towards a Single System Image for High-Performance Java”, *Parallel and Distributed Processing and Applications*, pp. 95-126, 2003

[Lea04] Doug Lea. *Package util.concurrent*, http://gee.cs.oswego.edu/dl/classes/EDU/oswego/cs/dl/util/concurrent/intro.html (last accessed 13 Mar 2009)

[LFGL01] Gregor von Laszewski, Ian Foster, Jarek Gawor, and Peter Lane. “A Java Commodity Grid Kit”, *Concurrency and Computation: Practice and Experience*, vol. 13, n. 8-9, pp. 643-662, 2001.

[LLM88] Michael Litzkow, Miron Livny, and Matt Mutka. “Condor - A Hunter of Idle Workstations”, in *Proceedings of the 8th International Conference of Distributed Computing Systems*, pp. 104-111, Jun 1988. See also http://www.cs.wisc.edu/condor/ (last accessed 6 Nov 2009)

[matl] The MathWorks, Inc. MATLAB®, http://www.mathworks.it/ (last access 8 Feb 2010)

[MSKW+06] Ingo Mierswa, Martin Scholz, Ralf Klinkenberg, Michael Wurst, Timm Euler. “YALE: Rapid Prototyping for Complex Data Mining Tasks", in *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD-06)*, 2006.

[MZC08] Cristian Mateos, Alejandro Zunino, and Marcelo Campo. “A survey on approaches to

gridification", *Software - Practice and Experience*, n. 38, pp. 523-556, John Wiley & Sons, Ltd., 2008. DOI: 10.1002/spe.847

[ora08] Oracle Corporation. *Oracle Coherence: Providing Extreme Performance, Predictable Scalability, and Continuous Availability for Mission-Critical Java Applications*, Dec 2008.

[osgi] *OSGi Service Platform, Core Specification, Release 4*, Version 4.1, OSGi Alliance, 2007

[p2pmpi] P2P-MPI http://grid.u-strasbg.fr/p2pmpi/ (last accessed 12 Oct 2009)

[PEF08] Shrideep Pallickara, Jaliya Ekanayake, and Geoffrey Fox. "An Overview of the Granules Runtime for Cloud Computing", *Proceedings of the IEEE International Conference on e-Science*, Indianapolis, Dec 2008. See also http://granules.cs.colostate.edu/ (last accessed 16 Nov 2009)

[PBGP+01] Michael Philippsen, Ronald F. Boisvert, Vladimir Getov, Roldan Pozo, José E. Moreira, Dennis Gannon, and Geoffrey Fox. " Javagrande - high performance computing with Java ", in *PARA'00: Proceedings of the 5th International Workshop on Applied Parallel Computing, New Paradigms for HPC in Industry and Academia*, pp. 20-36, Springer-Verlag, 2001.

[proactive] OASIS Research Team. *ProActive Programming: A Comprehensive Solution For Multithreaded, Parallel, Distributed, And Concurrent Computing*, version 4.0.2, 27 Oct. 2008.

[Qui93] Ross Quinlan. *C4.5: Programs for Machine Learning*, Morgan Kaufmann Publishers, San Mateo, CA, 1993.

[rmi] Sun Microsystems, Inc. *Java Remote Method Invocation Specification*, May 1996, Revision 0.9.

[SBDM+09] Tiziana Sanavia, Annalisa Barla, Barbara Di Camillo, Sofia Mosci, Gianna Toffolo. "Function-based analysis of microarray data via l1-l2 regularization", *2009 ISMB/ECCB Conference*.

[Seg07] Judith Segal. "Some Problems of Professional End User Developers", *IEEE Symposium on Visual Languages and Human-Centric Computing* (VL/HCC 2007), 2007.

[Seg08] Judith Segal and Chris Morris "Developing Scientific Software", *IEEE Software*, vol. 25, n. 4, pp. 18-20, Jul/Aug 2008.

[SHSS+07] Hermes Sengera, Eduardo R. Hruschkaa, Fabrício A. B. Silvaa, Liria M. Satob, Calebe P. Bianchinib, and Bruno F. Jeroscha. Exploiting idle cycles to execute data mining applications on clusters of PCs, *Journal of Systems and Software*, vol. 80, is. 5, Elsevier, May 2007.

[SHW98] Luis F. G. Sarmenta, Satoshi Hirano, Stephen A. Ward. "Towards Bayanihan: Building an Extensible Framework for Volunteer Computing Using Java". *ACM 1998 Workshop on Java for High-Performance Network Computing*, Palo Alto, California. Also published in *Concurrency: Practice and Experience*, vol. 10, pp. 1015-1019, 1998. See also http://groups.csail.mit.edu/cag/bayanihan/ (last accessed 10 Oct 2009)

[slf4j] *Simple Logging Facade for Java (SLF4J)*, (last accessed 11 Feb 2010)

[SMB07] Christoph Sieb, Thorsten Meinl, and Michael R. Berthold. "Parallel and distributed data pipelining with KNIME", *The Mediterranean Journal of Computers and Networks*, Vol. 3, No. 2, 2007

[spss] IBM SPSS Modeler, http://www.spss.com/software/modeling/modeler/ (last accessed 11 Jan 2010)

[SS98] Luís Moura Silva and João Gabriel Silva. “System-Level versus User-Defined Checkpointing”, in *Proceedings of the Seventeenth IEEE Symposium on Reliable Distributed Systems*, 1998.

[ssp] The SemiSpace project, http://www.semispace.org/semispace/ (last access 1 Feb 2010)

[sun96] Sun Microsystems, Inc. *Java Object Serialization Specification*, May 1996, Revision 0.9.

[Sut66] William R. Sutherland. *The on-line graphical specification of computer procedures*. Massachusetts Institute of Technology. Dept. of Electrical Engineering. Ph.D Thesis. 1966.

[tav] Taverna workbench. http://taverna.sourceforge.net/

[terr] Terracotta. http://www.terracotta.org/ (last accessed 23 Nov 2009)

[tri] *The Triana Project*. http://www.trianacode.org/

[tspace] IBM TSpaces, http://www.almaden.ibm.com/cs/TSpaces/ (last access 1 Feb 2010)

[TTV05] Domenico Talia, Paolo Trunfio, and Oreste Verta. "Weka4WS: a WSRF-enabled Weka Toolkit for Distributed Data Mining on Grids", in *Proceedins of the 9th European Conference on Principles and Practice of Knowledge Discovery in Databases (PKDD 2005)*, Porto, Oct 2005.

[VC08] Monica Vlădoiu and Zoran Constantinescu. “A Taxonomy for Desktop Grids from Users' Perspective”, in *Proceedings of The World Congress on Engineering 2008*, pp. 599-604.

[vNMH+02] Rob V. van Nieuwpoort, Jason Maassen, Rutger Hofman, Thilo Kielmann, and Henri E. Bal. “Ibis: an efficient Java-based grid programming environment”, in *Joint ACM Java Grande-ISCOPE 2002 Conference*, Seattle, Nov. 2002.

[vNMK+05] Rob V. van Nieuwpoort, Jason Maassen, Thilo Kielmann, and Henri E. Bal. “Satin: Simple and efficient Java-based Grid programming”, *Scalable Computing: Practice and Experience*, vol. 6, pp. 19-32, Sep 2005.

[WF05] Ian H. Witten and Eibe Frank. *Data Mining: Practical machine learning tools and techniques,* 2nd Edition, Morgan Kaufmann, San Francisco, 2005.

[WFTH+99] Ian H. Witten, Eibe Frank, Len Trigg, Mark Hall, Geoffrey Holmes, and Sally Jo Cunningham. "Weka: Practical Machine Learning Tools and Techniques with Java Implementations", in P*roceedings of the ICONIP/ANZIIS/ANNES'99 Workshop on Emerging Knowledge Engineering and Connectionist-Based Information Systems*, pp. 192-196, 1999.

[wfmc] Workflow Management Coalition. *Terminology & Glossary*. Technical Report Document Number WFMC-TC-1011, 1999, Issue 3.0.

[WN08] Piotr Wendykier and James G. Nagy. *Image Processing on Modern CPUs and GPUs*, Emory University Technical Report TR-2008-023, submitted for publication. http://mathcs.emory.edu/technical-reports/techrep-00148.pdf (last access 27 April 2009)

[Woj08] Marcin Wojnarski. "Debellor: A Data Mining Platform with Stream Architecture", *Transactions on Rough Sets IX, Lecture Notes in Computer Science*, vol. 5390, pp. 405-427, Springer-Verlag, 2008.

[WRW96] Ann Wollrath, Roger Riggs, and Jim Waldo. “A Distributed Object Model for the Java System”, In *Proceedings of the USENIX 1996 Conference on Object-Oriented Technologies*, Toronto, June 1996.

[WWWK94] Jim Waldo, Geoff Wyant, Ann Wollrath, and Sam Kendall. *A Note on Distributed Computing*, Sun Microsystems technical report TR-94-29, 1994. Available online

http://research.sun.com/technical-reports/1994/smli_tr-94-29 (last accessed 19 Aug 2009)

[ZH05] Hui Zou and Trevor Hastie. "Regularization and variable selection via the elastic net", *Journal of the Royal Statistical Society*, Series B, Jan 2005. Available on line http://www.blackwellsynergy.com/doi/abs/10.1111/j.1467-9868.2005.00503.x (last accessed 24 Apr 2009)