INSTITUTO
SUPERIOR
TÉCNICO

UNIVERSIDADE TÉCNICA DE LISBOA
INSTITUTO SUPERIOR TÉCNICO

New Environments for Parallel Execution and Simulations

João Nuno de Oliveira e Silva

Supervisor: Doctor Luís Manuel Antunes Veiga

Co-Supervisor: Doctor Paulo Jorge Pires Ferreira

Thesis specially prepared to obtain the PhD Degree in
Computer Science and Engineering

Draft

July 2011

Resumo

Uma nova classe de utilizadores tem emergido na área da computação paralela, tendo, no entanto, requisitos diferentes daqueles dos utilizadores tradicionais de sistemas de computação de alto desempenho. A maioria dos projectos destes novos utilizadores enquadram-se no paradigma *Bag-of-Tasks* (problemas compostos por tarefas independentes). As semelhanças entre o software e hardware (no que se refere a instalações especializadas e computadores pessoais), somado ao elevado desempenho dos computadores pessoais actuais, pode levar a concluir-se que estes novos utilizadores têm acesso aos recursos computacionais necessários. Apesar de existirem, a disponibilidade destes recursos é baixa, devido à inexistência de ferramentas e mecanismos adaptados a estes novos utilizadores.

Esta dissertação tenta encurtar esta barreira, tendo-se desenvolvido trabalho em três áreas relevantes: Submissão e criação de trabalhos, ambientes para execução de tarefas, e descoberta e avaliação de recursos.

No que respeita à criação de trabalhos, nenhum dos sistemas existentes é adequados à nova classe de utilizadores com poucos ou nenhum conhecimentos de programação. Assim, foi desenvolvida uma nova *interface* para criação de *Bag-of-Tasks* e uma plataforma para paralelização automática de código.

Também novos ambientes de execução de tarefas foram desenvolvidos. A plataforma BOINC foi estendida de modo a tornar-se verdadeiramente um sistema de Computação Distribuída Pública, uma plataforma que permita qualquer utilizador submeter trabalhos para serem executados. Outra fonte de poder computacional contemplada neste trabalho foi a Nuvem, com o desenvolvimento de uma nova heurística para a optimização da execução de *Bag-of-Tasks* em sistemas de computação utilitária (*Utility computing*).

Finalmente, de modo a garantir eficiente alocação de recursos e execução de tarefas, foi desenvolvida uma nova álgebra. Esta álgebra é extensível (ao nível dos recursos avaliados) e permite uma mais precisa e flexível avaliação e emparelhamento de recursos. Aos utilizadores é permitida a definição de funções de utilidade não lineares, mas intuitivas, podendo o próprio sistema, avaliar um mesmo requisito de modo diferente dependendo da classe do utilizador.

Abstract

A new class of users is emerging in parallel programming area, having different requirements from the traditional high performance computing users, since most projects deployed by the new users classes fit into the Bag-of-Tasks problems (mainly composed of independent tasks, thus embarrassingly parallel). The similarity of software and hardware (in specialized infrastructures and desktop home computers), added to the high performance of today's personal computers, may lead to the notion that these new common users (hobbyists or those with few computational requirements) may have easy access to the necessary computational power. Although there are new sources of computational power, their availability to common users is still low, due to the inexistent tools and mechanisms adapted to this new user class.

This thesis tries to address these gaps, with developed work in the three relevant layers: Job submission; Job deployment computational infrastructures, and Resource Integration and discovery.

With respect to job submission, the available systems and programming methodologies do not offer solutions to the new user classes with low (or even none) programming knowledge. So a new user interface for Bag-of-tasks' creation and a platform for automatic code parallelization were developed and evaluated.

New environments for the execution of Bag-of-Tasks were also envisioned and developed. BOINC was extended in order to make it a truly public Distributed Computing system (one that allows any user to submit work and execute other users' tasks). Another source of computing power targeted was the Cloud, with the development of an heuristic for the efficient execution of Bag-of-Tasks on utility computing infrastructures.

Finally, in order to guarantee an efficient allocation of resources and execution of tasks, a new resource evaluation algebra was developed. A more precise, flexible, and extensible resource evaluation and requirement matching mechanisms was developed. Users are now allowed to define non linear, yet intuitive, utility functions for the resources being evaluated, while the middleware can assign different resource evaluation rules, depending on the user class.

Palavras Chave

- Computação Paralela
- Computação Distribuída
- Computação na Nuvem
- Partilha de Ciclos
- *Bag-of-Tasks*
- Escalonamento de Tarefas
- Descoberta de Recursos

Keywords

- Parallel Computing
- Distributed Computing
- Cloud and Utility Computing
- Cycle Sharing
- Bag-of-Tasks
- Task Scheduling
- Resource Discovery

Publications

The work and results presented in this dissertation are partially described in the following peer-reviewed scientific publications:

International Journals

1. A2HA - Automatic and Adaptive Host Allocation in Utility Computing for Bag-of-Tasks. **João Nuno Silva**, Paulo Ferreira, and Luís Veiga. Journal of Internet Services and Applications (JISA), 2(2), pp. 171-185, Sep. 2011, Springer. (Ranked in the IST CCAD[1] *A Journals* list)

International Conferences ranked by IST CCAD

1. Service and resource discovery in cycle-sharing environments with a utility algebra. **João Nuno Silva**, Paulo Ferreira and Luís Veiga. 2010. In 2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS). IEEE. (Ranked in the IST CCAD[1] *B Conferences* list, ranked A in the CORE[2] Conference Ranking)

Book Chapters

1. Peer4Peer: E-science Communities for Overlay Network and Grid Computing Research. Luís Veiga, **João Nuno Silva**, João Coelho Garcia. Chapter on

[1] Conselho Coordenador de Avaliação dos Docentes, https://fenix.ist.utl.pt/ccad/

[2] Computing Research and Education Association of Australasia, http://core.edu.au/

the book "Guide to e-Science: Next Generation Scientific Research and Discovery", Springer. 2011. ISBN-10: 0857294385.

International Conferences / Workshops

1. Mercury: a reflective middleware for automatic parallelization of Bags-of-Tasks. **João Nuno Silva**, Luís Veiga, and Paulo Ferreira. In Proceedings of the 8th International Workshop on Adaptive and Reflective Middleware (ARM '09), collocated with ACM/IFIP/Usenix Middleware 2009. ACM. 2009.

2. SPADE: scheduler for parallel and distributed execution from mobile devices. **João Nuno Silva**, Luís Veiga, and Paulo Ferreira. In Proceedings of the 6th international workshop on Middleware for pervasive and ad-hoc computing (MPAC '08), collocated with ACM/IFIP/Usenix Middleware 2008. ACM. 2008.

3. Heuristic for resources allocation on utility computing infrastructures. **João Nuno Silva**, Luís Veiga, and Paulo Ferreira. In Proceedings of the 6th international workshop on Middleware for grid computing (MGC '08), collocated with ACM/IFIP/Usenix Middleware 2008. ACM. 2008. (ranked C in the CORE[2] Conference Ranking)

4. nuBOINC: BOINC Extensions for Community Cycle Sharing. **João Nuno Silva**, Luís Veiga, and Paulo Ferreira. In Second IEEE International Conference on Self-Adaptive and Self-Organizing Systems Workshops. IEEE. 2008

Other publications related to this thesis

Although not described in this document, because of the scope of this thesis, one more publication was produced during the course of this PhD.:

- Transparent Adaptation of e-Science Applications for Parallel and Cycle-Sharing Infrastructures. João Morais, **João Nuno Silva**, Paulo Ferreira and Luís Veiga.
  11th IFIP International Conference in Distributed Applications and Interoperable Systems, DAIS 2011, LNCS, Springer. 2011.(ranked B in the CORE[2] Conference Ranking)

Acknowledgments

First I would like to thank my advisers, for all the support given during the length of this PhD. To Professor Luís Veiga, the adviser, for all the energy and enthusiasm and encouragement given during the development and writing process of this thesis. To Professor Paulo Ferreira, my co-adviser for the initial support and for believing in the work being carried out.

To all the members of the Distributed Systems Group of INESC-ID, for the dynamic and fruitful work environment provided.

To the member of the DEEC Management Bodies, for caring about the progress of my work.

To all the colleagues of DEEC with whom I taught, with whom I learnt how to teach.

To João Garcia, for putting up with me all these years as an office mate and friend, and David Matos, for the offered friendship.

To all my friends, for being around and patiently waiting for the conclusion of this thesis.

To my Parents and Brothers. To Xana.

Lisboa, October 28, 2011

João Nuno Silva

To my parents and brothers

☯

Because of my students

Contents

List of Figures

List of Tables

Listings

1 Introduction

In recent years there has been a trend on the upper scale *supercomputers*: instead of dedicated, specialized hardware, these new supercomputers use (close to) off-the-shelf hardware in their construction [BG02, MSDS10]. The more powerful supercomputers are built as a collection of independent computers, linked by high speed network connections. Although these computers are much more powerful than desktop personal computers (in terms of processing power, storage capacity and reliability), their architecture and running software are essentially the same.

This similarity between commodity hardware and specialized high performance computing nodes, and its wide availability, has lead to the idea of using available personal computers as processing nodes, either locally or on a larger, more diffuse, distributed computing infrastructure.

On the user level, it can also be seen a higher demand for computational power. In the areas traditionally demanding computational power (e.g. physics and mechanics), scientists still require numerical simulation, but new areas are emerging also requiring the simulation of processes (such as economy, computational biology).

In emergent research areas, tools to process data are being developed, and users are increasingly generating more data to be processed. Furthermore, users with some programming knowledge are now able to develop, and efficiently execute, their simulations and data analysis.

Besides these new knowledgeable users, also on the commoner side there are new requirements for high levels of processing power to solve some of their problems. Domestic and professional users now have more digital data to be processed. For instance, designers have digital models that need to rendered, hobbyists want to generate animation films, and even common users have batches of photos to process for enhancement.

Most of these users' work fits into the Bag-of-Tasks programming model, where a computational job is composed only of independent tasks. Researchers from the statistical or economy areas may require to perform batches of simulations without strict requirements, as the simulations are independent and decoupled among themselves. The rendering of frames of films can also be easily parallelized as independent tasks. Domestic or graphical industry users' needs also fit into this model: the rendering of an image or processing of batches of images can be easily decomposed in independent tasks.

The tools used by this new user class are mostly off-the-self applications (such as statistical analysis software or image renders). Although most of such tools are not designed to take advantage of the the available parallel execution environments, a task decomposition of the problems can take advantage of these unmodified applications.

These new users have different requirements from the traditional high performance computing users. While traditional users require a dedicated computational infrastructures (due to the necessary synchronization between tasks) and have easy access to such infrastructures, other users do not. Although, most projects created by the new users classes fit into the Bag-of-Tasks problems (mainly composed of independent tasks, thus embarrassingly parallel), they can neither afford nor access existing parallel computing infrastructures.

The similarity of software and hardware (in specialized infrastructures and desktop home computers) added to the high performance of personal computers may lead to the idea that common users (hobbyists or those with few computational requirements) may have easy access to the necessary computational power.

Although there are new sources of computational power, their availability to common users is still low, due to the inexistence of tools and mechanisms adapted to this new user class.

The most accessible sources of computing power are the desktop computers owned by each user. In some cases, when connected by a local area network, it is possible to use them as a cluster capable of speeding up existing computations.

The ubiquitous connection of these computers to the Internet also makes them a good source of computing cycles to remote users, as demonstrated by the various successful Distributed Computing projects [BOI11].

Recently, a new source for computing power has appeared. Different from other cloud services, Amazon offers the possibility to dynamically switch on or off computer instances on their cluster. The user can install any operating system and run on those machines any service, also allowing the execution of computing intensive tasks.

In the remaining of this chapter, it is presented in detail the available solutions for the execution of Bag-of-Tasks problems, with respect to available infrastructures, tools, and programming methodologies. Next, the deficiencies of the available solutions are presented when applied to the focus populations (i.e. user with limited or null programming knowledge), and the objectives and contributions of the developed work are listed.

Although the concepts to be presented next are related to the current taxonomies [Fly72, Ski88, Fos95, Bar10], in this section a higher level approach is taken, presenting a view of parallel computing resources closer to the end user or programmer, and to the problems being solved.

1.1 User classes

The increase of the execution speed of the Personal Computer has lead to the idea of using it as a data processing platform. Added to the wide availability of Personal Computers, more users have started to develop their computing intensive code, and take off-the-shelf application to their performance limits.

Up until a few years ago the only users that could take advantage of parallel computing infrastructures where those performing numerical simulations. These users were required to have knowledge of parallel programming, as they had to develop the jobs completely: processing code, task creation, and communication between tasks.

Today, new classes of users requiring fast processing of data emerged. Some actually have limited programming knowledge (but not necessarily with proficiency on parallel programming) but most do not. These only know how to work with a limited set of tools, mostly by invoking them with command line arguments.

These changes can be related to the phenomenon that occurs since the 70's: end-user computing [RF83, SK86, RH88, Pan88, CK89, MK03b, Gov03]. The replacement of dumb main-frame terminal by personal computers has lead to the rise of computer related training and formation: users have become partially responsible for the maintenance of the computers, and some of the now generic applications (such as word processing, spread-sheets) require extensive training plans.

In parallel with this evolution, also the application development area has been changing. End-users can now develop some applications, allowing them to immediately solve some of their problems without the intervention of the IT department personnel.

The level of computer literacy varies from users without any computing knowledge to the IT personnel, but with a finer distribution in between.

MCleans [McL74] originally proposed a simple characterization of computer users existing in a organization:

- Data Processing professionals
- Data Processing amateurs
- non Data Processing trained users

While it is possible to match these classes to users currently using and requiring computing cycles to execute data processing and simulations, Rockart [RF83] provides a finer grained, more useful classification of end-users:

- Non-programming end-users
- Command level users
- End-user programmers
- Functional support personnel
- End-user computing support personnel
- Data processing programmers

For the purposes of the developed work presented in this Thesis, such a fine-grained classification is not required. A broader classification, that can still be matched to the one developed by Rockart, should be used:

- Expert HPC[1] users (**with** parallel programming knowledge)
- Programmers (**without** specific parallel programming knowledge)
- Tool users and Hobbyists

Non-programming users and *end-user computing support personnel* are of no interest to this work. The former do not have computing needs, and the latter only develop applications to help support the operation, not to process data.

In this classification *Expert HPC users* encompass *data processing programmers* and *Functional support personnel*. These are users having extensive programming knowledge, and have a deep understanding of the available infrastructure. All these users are researchers or engineers requiring large infrastructures to execute their simulations. The codes developed simulate physical phenomena, namely on the physics, chemistry or mechanical engineering area. The developed code runs on hundreds of nodes and, due to their nature, requires communication among them. These users, when developing their simulation code, have to take into account the inter-process communication and the synchronization between processes.

[1] High Performance Computing

These programmers are considered end-users as their primary field of knowledge and study is not computer science or programming. They are engineers or scientist that, due to the environment, have to program the solutions to their problems.

Another class of users (*programmers*) knows how to program and can develop sequential applications. This class matches the characteristics of *end-user programmers* proposed by Rockart. However, they can only develop, albeit often sophisticated, single threaded simulation or analysis code, that nowadays can be executed efficiently on personal computers. Some of the problems being solved require the execution of the same code with different parameters. These users can be found in research laboratories working in the area of biology, statistics, or even computer science (e.g. network protocols simulation).

Nowadays, *tool users* can also be a target population for parallel execution environments. These users know how to parametrize a limited set of tools (such as image rendering software, or data analysis), thus fitting into Rockart's *command level users*. Nonetheless, at some points of their work, these users need to execute repetitive tasks over a large set of data. These users usually have access to personal computers, and their computer knowledge is often limited. Hobbyists also fit in this category of users, in case of processing batches of images or rendering images.

1.2 Current tasks execution environments

Up until a few years ago, the only source for usable cycles for the execution of timely simulations or other data processing was found in datacenters. The available computers were either mainframes, or supercomputers, depending on the target population. Today most of the techniques developed for supercomputers are available in the most ordinary desktop computers, such as floating point and vector processing units, and even dedicated processors. Furthermore the performance of the currently available systems is on par with the supercomputers of the past at a much lower cost [AB08].

The sources for computing cycles are now more diverse and closer to the common user present at the edges of the Internet. In this section, the available sources (from the classics to the most recent ones) are presented along with a evaluation of their target population and possibility to be used by a common user (*programmer* or *tool user*). The list of available processing powers sources is as follows:

- Institutional HPC infrastructures (Cluster/Supercomputer)
- Grids
- Personal Clusters
- Shared Memory Multiprocessors
- Internet Distributed Computing systems,
- Cloud / Utility computing

High Performance Computing (HPC) infrastructures [Ste09, MSDS10] are composed of hundreds of computing nodes and several TBytes of storage space, all connected with dedicated high speed network links. This allows the fast execution of any demanding computing jobs. These systems are usually owned and managed by an entity the enforces strict access and usages polices: only users belonging to that organization are allowed to run jobs, or the access requires a previous contract or grant. So, the access to these systems is restricted to users with a continuous and high demand for computing cycles.

Grid [FK99, FKT01, Fos02] infrastructures ease the remote access to HPC computing infrastructures to users outside the owner organization (e.g. in the context of a virtual organization). Furthermore, Grid initiatives and infrastructures allow the aggregation of scattered resources. In the same manner as with classical HPC infrastructures, it is necessary for a user, in order to use it, to have an institutional relationship with a grid initiative participant.

The construction of a small scale computing *cluster* [SSBS99, BB99] is nowadays easy. The available commodity-off-the-shelf (COTS) hardware is powerful enough to be used in complex computations, and the speeds attained with a simple Ethernet Gigabit switch are sufficient to the user needs. Although the hardware is available, the existing tools are not targeted at all the user classes. Intensive computing applications may also take advantage of multicore machines, but the integrated execution of multiple distributed computers is not easy to the common user.

The current development of microprocessor technology has lead to the development of multi-core processors [cor08, SCS$^+$08, Dev10, Int11]. These fit into the same package distinct processing cores connected to the external memory and devices by a shared bus. In most multi-core architectures, each core fits a individual Arithmetic, Logic Unit and Register Sets. Depending on their implementation, caches and Control Units can be private or shared among cores. In commodity processors (from Intel and AMD), each core fits a different and private Control Unit rendering that multicore processor similar to a *shared memory multiprocessor* implementation. On these systems, the operating systems see them as multiprocessors, employing multiprocessor thread and process scheduling. Also on the parallel application level, most existing programming methodologies can be applied as if a multiprocessor was being used.

The processing power of desktop personal computers has increased in the last years. Adding to this fact, these personal computers, besides being idle a large part of the time, have increasingly faster Internet network connections. Taking advantage of these facts, a new infrastructure for parallel computing has emerged: *Internet Distributed computing* [ACK$^+$02, And04]. Nowadays several research projects and organizations [Pea11] take advantage of this novel computing environment: researchers develop applications that perform a certain simulation to be executed on the personal computers owned by the the users at the edges of the Internet. Data is transferred to the donor computer when the user is connected to the Internet, allowing later execution of the simulation (when, otherwise, the computer would be idle).

Recently, private data center owners with a presence in the Internet have begun to offer computing services to remote users, promoting the emergence of the *Cloud* [Wei07, AFG+09, AFG+10, WvLY+10]. One of such services, coined *Utility Computing* allows computer resources such as virtual machines, disk space to be rented by the end-user. Amazon is one of the the most successful providers of Utility Computing, offering in its portfolio the Elastic Computing Cloud [Ama11] service. Such services allow users to install ordinary versions of regular operating systems and, on demand, launch instances of those operating systems. The charged value depends on the execution time and on the hardware characteristics (number of processors, memory). The access to these computer instances is straightforward, as the user only has to sign a simple contract and provide a valid credit card to be later charged. Other *cloud* services allow the run-time upload of code to a remote infrastructure for later parallel execution of locally invoked functions [PiC10, Lon11].

1.3 Current parallel programming paradigms

Depending on the problems being solved, different programming paradigms, work organization and decomposition fit them better. Next, the main programming paradigms for the development of parallel jobs, and the various ways to organize the tasks comprising a job are presented.

1.3.1 Work organization

The kind of problem being solved determines the best suited work organization to be used, being one of the following:

- Bag-of-Tasks
- Master-slave

- Recursive
- Decomposition with inter-task communication

These different work organizations are tightly related to the way data can be partitioned and the way that data is to be processed: i) if it can be processed in a single or multiple stages, ii) if it can be split *a priory* or dynamically split by each of executing tasks, or iii) if each task needs data and results from other tasks.

Bag-of-Tasks problems are composed of independent tasks [CBS$^+$03]. These tasks can be launched independently of each other. Before execution, each computing node receives the code and the data to be processed and, after the execution of the code, the results are returned. In this kind of jobs, there is no interaction between the launching code and each task, furthermore after each task execution (data processing) the launching code (job management) does not further process any results. At the end of the execution of all the tasks, the user has a set of data that must be later processed in other tools.

When using the master slave paradigm [SV96], there is some interaction between the main process (master) and the tasks (slaves), but still there is not any communication among tasks. The master, besides launching each task, also performs some data pre-processing and results aggregation. In the simplest form, the master only interacts with the slaves at the beginning of each task (to invoke them and send data) and when tasks finish, to retrieve results. More complex interactions involve the master to wait for the slaves (using barriers) in the middle of their execution, in order to retrieve partial results, process them and distribute new versions of data.

If tasks can spawn themselves in order to further distribute work, a recursive work distribution [BJK$^+$95, RR99] is present. The data assigned to a task is split and part of it is assigned to the newly created child task. A task can only terminate (and return its result) after completion of all its children and aggregation of their partial results. Although of a simple implementation, due to the natural recursion in data, a high level of parallelism is easily attained.

The most complex problems require data to be decomposed and distributed among processes and, during execution, communication between the various tasks, such as in Particle-in-Cell [Daw83] or Finite Elements [FWP87] simulations. When, in order for a task to make progress in its computation, it needs data from another tasks (i.e. a sibling or neighbour task, w.r.t data being processed), communication between computing nodes needs to be carried out. When developing tasks' code, it is necessary to guarantee the correct synchronization so that dead locks or inconsistencies do not happen. Although a master-slave solution is always possible, where a single process centralizes all communication, the number of tasks, and the amount of transmitted data may render such solution unusable. The decentralized communication between tasks eliminates a bottleneck (the master) and reduces communication delays.

1.3.2 Programming model

In order to implement a parallel project and its tasks, some sort of programming tools and libraries must be used; these are mainly used to initiate computation and to perform communication between intervening components.

These libraries or tools must allow the implementation and deployment of the previously described work organization, follow some known model, and fit into at least one of the following classes:

- Message passing communication
- Shared memory communication
- Explicit programmed task creation
- Declarative task definition

Of the previous list, the first two items state how communication between components is programmed and carried out [GL95]. The other two define how tasks are created by the programmer/end user.

When the work organization requires communication between components (*master-slave* or *decomposition with inter-task communication*), the programmer either uses an explicit *Message Passing* API or uses *Shared Memory* paradigm.

With Message Passing [GBD$^+$94, Mes94], the programmer explicitly invokes functions to send and receive data. During execution, the executing processes (tasks or master) synchronize at these communication points, with the receiver of the message waiting for the sender to transmit the data. In a master-slave problem, usually the master waits for messages from all the tasks, processes the received data and transmits back a new set of data for a new parallel interaction. When there is inter-task communication, tasks must know the identification of those others where some data must be sent to, in order to periodically communicate with them.

Another way to transmit information among tasks or between a master and its slaves is using variables residing in a shared memory space. These variables are shared among the participant processes, and its access is performed as if it was a simple global variable, by using the common programming language assignments, accesses and expressions. As this data can be concurrently accessed by various processes, it is necessary to guard these accesses: synchronization routines must be explicitly used when reading or writing values on the shared memory. The underlying hardware or system software must guarantee some level of consistency when accessing shared data [AG96, Gha96].

Shared memory [Ste99, Boa08] is mostly used when executing the concurrent tasks in a computing infrastructure offering shared memory hardware support, since the available memory space is naturally shared among all the processes or threads, and transparently accessed by the application. If a central physical memory is accessed by all processors, the use of a shared memory programming model, incurs no additional performance penalty when performing communication, because there is no data transmission between private memory spaces.

When the tasks are to be executed on a cluster of computers connected by a System Area Network (SAN) (such as Gigabit Ethernet [IEE11], Myrinet [Myr09] or Infiniband [Ass10]), a *Software Distributed Shared Memory* [LH89] (DSM) library can be used. Although not all of the data can or should be shared among tasks, a communication buffer can be shared and accessed as a local variable. In order to reduce data transfer these DSM systems implement relaxed data consistency protocols that guarantee that: from all the shared data, only the necessary one (the one recently modified) is transmitted [GLL$^+$90, KCZ92]. This data selection is performed without the user knowledge or intervention and transparently reduces the overall communication [SG00, Sil03].

In terms of task creation, the user either creates them explicitly (in inter-task communication, master-slave or recursive problems) or relies on the infrastructure to create them. Using a task creation API (*Explicit programmed task creation*), the programmer must define when tasks are to be created: in the beginning of the program or during the execution of jobs.

If the problem fits into the *Bag-of-Tasks* category, it is only necessary for the programmer to define the code to be executed by each task, possibly by means of *declarative task definition* mechanisms. The underlying system will be responsible for launching the tasks (transmitting the code and the input data of each task) and retrieve the partial results after each task completion. The code to be executed can be enclosed in either a function (or class method) or a self-contained application. In both scenarios the underlying system is responsible for the execution of that code.

1.4 Target population characterization

The following tables systematize the relations between each class of users and the infrastructures, tools and methodologies previously presented. These relations are important in order to know and understand the focus and target population of available systems, and what are the requirements common users have.

Table 1.1 shows the types of computational work the different classes of users usually have to perform.

	Experts	Programmers	Tool users
Bag-of-Tasks	✓	✓	✓
Master-slave	✓	✓	✗
Recursive	✓	✓	✗
Inter-task communication	✓	✗	✗

Table 1.1: Usual work organization by user class

When referring to scientific parallel computing, *expert users* are usually tied to to development and execution of *inter-task communication* problems, as their problems are usually best solved this way. All other types of work, is also possible to be deployed by a *expert user*, depending on the problem being solved.

Master-slave work division can be performed when the problem being solved requires several iterations of the same code, with a global state being evaluated, but without communication between the concurrent tasks. Depending on the supplied infrastructure, both *expert users* or *programmers* can implement and deploy jobs with such work organization.

Both *programmers* and *tool users* can develop solutions to problems fitting the *Bag-of-Tasks* category. All the problems dealing with the independent processing of batches of data or files fit into this category. These users, due to the unavailability of resources, may solve this kind of problems executing sequentially several instances of serial applications (or use pre-existing tools).

Recursive work organization naturally arises from a recursively organized data. As both *expert* users and *programmers* may need to handle such data, these user classes may encounter problems that can be easily partitioned in a recursive manner.

This evaluation only states the possibility for a user to encounter a specific work organization model. The possibility to implement it and efficiently execute such solutions is presented in the following tables.

The next table (Table 1.2) presents the possibility for a particular user to access and use the currently existing computing resources. Here, only the access to such infrastructures is evaluated, not taking into account the knowledge needed to configure and use such resources.

	Experts	Programmers	Tool users
HPC Clusters / Supercomputer	✓	✗	✗
Grid	✓	✗	✗
Shared Memory Multiprocessors	✓	✓	✓
Personal Cluster	✓	✓	✓
Distributed Computing	✓	✓	✗
Cloud / Utility Computing	✓	✓	✓

Table 1.2: User access to the available infrastructures

As expected, *expert users* usually have access to any of the presented infrastructures. If the institution he is working at has a *HPC Cluster* or *Supercomputer* or belongs to one of the existing *Grid* initiatives, the access to those resources is possible. The other two classes of users, because they do not have the necessary institutional links, are deprived of these resources.

Today any kind of user, independently of his programming skills can build, with off-the-shelf components (computing and network hardware), a small *personal cluster*. This cluster can also aggregate old and slower hardware. Although small, these can speed some of the usually lengthy jobs.

Shared memory multiprocessors are also widely available, as most entry level personal computers nowadays can fit a multi-core processor. At least two cores are available on most desktop computers currently sold.

The access to *utility computing* is also possible to any of the presented user classes, as it is only necessary to sign a simple contract with the service provided, and provide a valid payment method.

Distributed Computing may seem a solution usable by any kind of users, but that is actually not the case. Any user with a personal computer and an Internet connection can donate cycles for solving others' problems, but not everybody can take advantage of them. Today it is necessary to have programming knowledge to write the code to be remotely executed. Furthermore it is necessary to be well known and to have some media coverage to gather enough donors. These issues are further detailed in Chapter 2.

Another relevant issue is how well a user is proficient in the required tools to execute parallel jobs, Table 1.3 shows this relation.

	Experts	Programmers	Tool users
Message Passing	✓	✗	✗
Shared Memory	✓	✗	✗
Task creation	✓	✓	✗
Task definition	✓	✓	✓

Table 1.3: Ease of use of various programming models and tools

As expected *expert users* are able and can use any kind of tool or API to develop their jobs.

Users with limited programming skills may not be able to use *message passing* (such as MPI) or *shared memory* APIs to develop their jobs, but are able to design and develop self contained tasks that are created using a simple *task creation* API or using *task definition* tools. *Message passing* or *shared memory* require an architectural knowledge that a non *expert* programmer may not have.

On the other end of the spectrum, lie *tool users* that can not use any system requiring programming knowledge. These users are limited to the declarative *task definition*.

Observing the previous tables a conclusion can be drawn: although *expert users* can take advantage of the full spectrum of systems, work organizations and resources, this is not true to the other two classes of users.

Programmers with limited parallel programming knowledge can only solve *bag-of-tasks* and simple *master-slave* problems, execute their jobs on *personal clusters*, *Distributed computing* infrastructures and *utility computing* systems, and develop them using simple *task creation* APIs or *task definition* tools.

Ordinary *tool users* are further limited: the resources are limited to *personal clusters* and *utility computing*, and they can only develop their *Bag-of-Tasks* jobs using *task definition* tools.

Besides these natural restrictions, these two user classes face further limitations due to the inadequacy of the available tools to these new users and their usage.

Although it is possible to easily build *personal clusters* and use *multiprocessors*, the tools to deploy work on them are still the same as the ones available to *HPC infrastructures*, making them impractical to low knowledge users. With *utility computing* the same happens: it is possible to create on-demand clusters but the tools to easily deploy work on them do not exist. Furthermore, nowadays it is impossible to know exactly how many machines should be allocated for the intended performance, so that the execution cost is minimal.

The Internet *distributed computing* may seem the optimal source for resources, but users with short term jobs, or with small visibility, can not take advantage of this cycles source. The following chapter (Chapter 2) presents a taxonomy for Internet based Distributed Computing and applies it to existing systems in order to infer the reasons for success of such systems.

1.5 Objectives

The main objective of the developed work is to allow non *HPC experts* to easily and efficiently use available computing resources for the parallel execution of tasks. The target population of this work belongs to the *programmers* and *tool users*, presented in Section 1.1. Due to their limited knowledge, these users are not capable of creating and deploying their processing tasks on the available infrastructures (Section 1.2).

The developed work focuses on solutions for the execution of *bag-of-Tasks* problems, created by users with limited or no knowledge on parallel programming, and without access to conventional parallel execution environments.

In order to increase resource availability and ease of access, three specific lines of work were explored, addressing different aspects, and ranging from high level task creation to lower level resource discovery:

- job submission methods
- resource integration and discovery
- job deployment environments

  These lines of work can be easily matched to different layers of the overall generic architecture, presented in Figure 1.1.

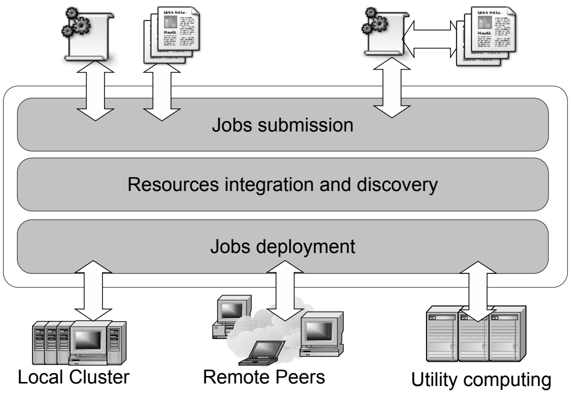


Figure 1.1: Overall architecture

On par with the work line presented, also the main objectives can be further detailed:

- allow a simple, more efficient job definition to the proposed target population
- allow an efficient resource usage
- allow the use of more diverse and widely available execution environments

1.5.1 Contributions

As expected each of the individual contributions also lies in at least one of the layers presented.

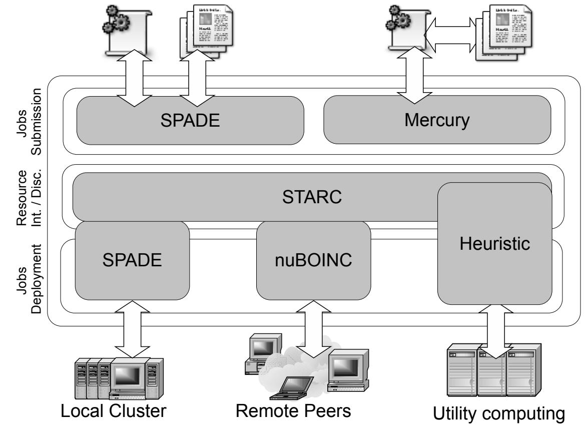


Figure 1.2: Overall architecture with highlighted contributions

Figure 1.2 clearly presents the developed work and its division, and where its contributions fit in the overall architecture. The following paragraphs present briefly the developed work, along with its main contributions and results.

Jobs submission Although the Bag-of-Tasks are the easiest problems to develop, the existing tools are not the best suited to the target population. Users are required to either write complete scripts or applications, from where tasks are created. This fact limits the users able to use available resources, not due to the access difficulties, but due to complexity of job creation.

The developed work and mechanisms target efficiency (in terms of tasks definition/development time) and expressiveness in task creation, allowing the submission of jobs and their tasks based on independent method invocations (Mercury [SVF09]) or independent applications (SPADE [SVF08c]).

Jobs deployment As the target population does not have access to HPC infrastructures, the use of more accessible resources is fundamental. It is necessary to develop job deployment mechanisms that allow less knowledgeable users to use existing infrastructures, and to provide them with access to novel resources.

Two new execution environments are addressed: utility computing infrastructures (Heuristic [SVF08a, SFV11]) and Internet distributed computing environments (nuBOINC [SVF08b]). Furthermore part of this work allows easier employment of available personal clusters (SPADE [SVF08c]) in the execution of

Bag-of-Tasks.

Resources integration and discovery With new users having jobs to be executed, with more resources being available for use, the resources integration is fundamental for efficient usage. Without these efficient mechanisms for resource discovery, aggregation, integration and usage, any work related to job submission or deployment is of no use.

To tackle this issue, a new algebras for a more flexible and precise evaluation of Internet scattered resources was developed (STARC [SFV10]). The heuristic [SVF08a, SFV11] also allows an efficient use of an Utility Computing infrastructures as a source of processors for the execution of Bag-of-Tasks.

1.5.2 Scientific Publications

All the developed work was published and presented in the context of peer reviewed international scientific journals, conferences and workshops, and partially described in a book chapter. The list of these publications is as follows:

- A2HA - Automatic and Adaptive Host Allocation in Utility Computing for Bag-of-Tasks. [SFV11]

  João Nuno Silva, Paulo Ferreira, and Luís Veiga.

  Accepted for publication on JISA - Journal of Internet Services and Applications, Springer.(Ranked in the IST CCAD[2] *A Journals* list)

- Service and resource discovery in cycle-sharing environments with a utility algebra. [SFV10]

  João Nuno Silva, Paulo Ferreira and Luís Veiga. 2010.

  In 2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS). IEEE. (Ranked in the IST CCAD[2] *B Conferences* list)

  Object Identifier: 10.1109/IPDPS.2010.5470410

- Peer4Peer: E-science Communities for Overlay Network and Grid Computing Research. [VSG11]

[2] Conselho Coordenador de Avaliação dos Docentes, https://fenix.ist.utl.pt/ccad/

Luís Veiga, **João Nuno Silva**, João Coelho Garcia. 2011.
Chapter on the book "Guide to e-Science: Next Generation Scientific Research and Discovery", Springer.
ISBN: 0857294385

- Mercury: a reflective middleware for automatic parallelization of Bags-of-Tasks. [SVF09]
  João Nuno Silva, Luís Veiga, and Paulo Ferreira. 2009.
  In Proceedings of the 8th International Workshop on Adaptive and Reflective MIddleware (ARM '09), , collocated with ACM/IFIP/Usenix Middleware 2009. ACM.
  Object Identifier: 10.1145/1658185.1658186.
- SPADE: scheduler for parallel and distributed execution from mobile devices. [SVF08c]
  João Nuno Silva, Luís Veiga, and Paulo Ferreira. 2008.
  In Proceedings of the 6th international workshop on Middleware for pervasive and ad-hoc computing (MPAC '08),collocated with ACM/IFIP/Usenix Middleware 2008. ACM.
  Object Identifier: 10.1145/1462789.1462794
- Heuristic for resources allocation on utility computing infrastructures. [SVF08a]
  João Nuno Silva, Luís Veiga, and Paulo Ferreira. 2008.
  In Proceedings of the 6th international workshop on Middleware for grid computing (MGC '08), collocated with ACM/IFIP/Usenix Middleware 2008. ACM.
  Object Identifier: 10.1145/1462704.1462713
- nuBOINC: BOINC Extensions for Community Cycle Sharing. [SVF08b]
  João Nuno Silva, Luís Veiga, and Paulo Ferreira. 2008
  In Second IEEE International Conference on Self-Adaptive and Self-Organizing Systems Workshops, Colocated with SASO 2008. IEEE.
  Object Identifier: 10.1109/SASOW.2008.66

- Transparent Adaptation of e-Science Applications for Parallel and Cycle-Sharing Infrastructures.

  João Morais, **João Nuno Silva**, Paulo Ferreira and Luís Veiga.

  In 11th IFIP International Conference in Distributed Applications and Interoperable Systems, DAIS 2011, LNCS, Springer. 2011.

1.6 Document Roadmap

The rest of the document is organized as follows.

In the next chapter the developed taxonomy is presented, along with its use in the discovery of the requirements for a successful Distributed Computing system: usable by users of all conditions, and providing efficient and correct results.

The following chapters substantiates the developed work, presenting its relevance, contributions and results. The novel job definition mechanisms are presented in chapters 3 and 4.

Chapters 5 and 6 deal with the job deployment layer. The former allows the use of Internet scattered resources, while the latter presents work related to the use of Utility computing infrastructures. Chapter 7 presents an algebra for the evaluation of the available resources.

The document concludes with Chapter 8 discussing the proposed solutions, and where an overview of the possible integration of the envisioned solutions and future work is presented.

2

Distributed Computing Systems

The use of a Internet based Distributed Computing infrastructure may seem the panacea for common users with computational needs. The true fact is that there are several systems that, in theory, could be used by common users scattered over the Internet, but in fact they are not. This reality was briefly motivated in the previous chapter, but must be further detailed.

This chapter presents a new taxonomy for the characterization of Distributed Computing Systems. For each relevant characteristic identified, the possible alternative approaches are highlighted and presented how existing systems implement it. This taxonomy includes the more usual architectural characteristics but also those more tied with the user experience and often overlooked: efficiency of job execution, security, and the availability and nature of the mechanisms for development and creation of jobs. Using the presented taxonomy, the most relevant systems developed up to date are also characterized.

With the detailed characterizations, made following this new taxonomy, it will be possible to understand the deficiencies of current approaches, what makes a system successful, and the directions a Distributed Computing system must take to be widely used.

2.1 Introduction

The Internet is a good source of computational power for the execution of parallel tasks: most of the connected computers are idle some of the time and, even when busy, are perfectly capable of executing most available jobs.

Taking this into account, in recent years there has been a significant development and research on Distributed Computing Systems to allow the public execution of Bag-of-Tasks jobs on the Internet. Existing systems present and employ novel mechanisms, ranging from the definition of adequate programming techniques, to new network architectures, or even to more efficient scheduling techniques.

In general, the goals of these systems are twofold: i) to allow anyone with parallel jobs to deploy them to be executed on remote computers, and ii) to attract owners of connected computers to donate processing time to those jobs that need it.

The presented goals are to be achieved with minimal burden to those that take part in the process (client programmers and donors), and should allow any user (independently of their computing knowledge) to take advantage of existing resources.

Programmers should have minimal work parallelizing the applications to be executed on the Internet, and should gain from the parallel execution of their tasks. Issues such reliability of the returned values and security of data and code should also be handled by the system. Furthermore, after submitting the work, it is expected that the tasks execute with a certain speedup.

On the other hand, the donor should be disturbed to a minimum, when installing the system, and executing the parallel code: i) the installation should be straightforward, ii) the security should not be compromised, and iii) the overhead incurred from downloading and executing the code should be minimal.

The way each system handles and solves the previous issues is fundamental to its widespread adoption as a valid solution to the execution of lengthy Bag-of-Tasks problems. In order to define or find the best design and architectural decisions, it is necessary to study and characterize existing systems.

In the previous chapter, Internet Distributed Computing Systems were presented as a possible and viable source of computing cycles, but it was also stated that they were essentially unusable by the target population of this work: common users, or programmers with infrequent computational needs. To demonstrate this statement, an evaluation of the available systems must be performed.

Although currently various systems exist, and employing different technologies, few are able to attract a suitable user base. The only Distributed Computing platform with widespread use is BOINC [And04] and its derivatives. All other systems are in effect not widely used.

To study this phenomenon and determine the reasons behind it, it is necessary to define a suitable taxonomy. Some taxonomies already exist, but are focused on architectural characteristics, not dealing with those more more tied with the user experience: efficiency of job execution, security and mechanisms for development and creation of jobs.

The main focus of this chapter are systems that are public, in the sense that they are generic enough to allow any Internet connected computer owner to create projects or jobs, and that do not require complex administrative donor admission. Such systems are generic, not being tied to a specific problem, and should allow users to create jobs (by providing the processing code and data), or just create tasks (by submitting the data to be processed by previously developed and/or deployed code).

In the systems targeted in this chapter, users can be either clients or donors. Donors are gathered from the Internet, not being required to belong to an organization (as seen in Enterprise Desktop Grids). These donor users are only required to install some simple software module that will execute the code submitted by the clients. The client users must have lengthy problems easily parallelizable: preferably using the Bag-of-tasks paradigm.

The following section (Section 2.2) describes the developed taxonomy. Its presentation and description is split in three distinct sets of characteristics: architecture, security and reliability, and user interaction. For each of these sets, each characteristic is explained and analysed on how it is implemented by the currently available systems.

Section 2.3 describes some previous works that tried to systematize the characteristics of distributed and parallel computing systems. Some of this work also presents taxonomies for the characterization of Desktop Grid systems that, only at some points, intersects with the study here presented. These intersection areas are also highlighted.

Section 2.4 presents the characteristics more relevant to user adhesion and describes the optimal design and implementation decisions. In this chapter, BOINC is presented along with its design, architecture and implementation decisions and how they affect the installed user base.

This chapter closes with a summary of the conclusions that rose from the development and use of the taxonomy.

2.2 A Taxonomy for Cycle-Sharing Systems

Three important factors have impact on the user experience one can have while using a Distributed Computing System, either when submitting work or executing it: i) the architecture, ii) the security and reliability, and iii) how the user interacts with the systems.

The first class of factors limits the overall performance of the systems when the number of users (providing and using resources) scales to hundreds, thousands, or more, and how efficiently the executing hosts are selected (taking into account the jobs' requirements). Delays before the start of each task should be minimum and fairness should be pursued when tasks from different users compete for the same resource.

Security is an important feature, as those donating resources do not want to become vulnerable to attacks, and all of the participants require minimal levels of privacy. The implementation of reliability mechanisms is fundamental to promote both liveness and correctness.

The last class affects how users create jobs and what kind of work can be submitted. A system difficult to configure will not attract donors nor the clients will be able to efficiently deploy and execute their jobs.

As all these factors directly affect users (either as a work creator or donor), they are fundamental to ensure that a system gathers a large user base to be useful to those submitting work to be executed.

This section is split in three parts, each dealing with one class of characteristics. There, the different characteristics are presented and then applied to the various Distributed Computing Systems considered in this study.

The use of non dedicated remote network connected machines for the execution of parallel jobs has been initiated with Condor [LLM88]. This infrastructure allows the execution of independent tasks on remote computers scattered on a LAN. As the original target computing environment was composed of commodity workstations, Condor only scheduled work when these computers were idle. Due to the homogeneity of the environment (all aggregated workstations shared a similar architecture and operating system) compiled applications could be directly used.

The applicability of the concept inaugurated by Condor to the Internet was limited by the natural heterogeneity of this new environment. There was no guarantee that the available workstations shared the same architecture, or operating systems, and no widely available portable language existed. The development and success of the JAVA language and Virtual Machine partially solved these problems.

This new language was ported to most existing architectures, allowing the execution of a single application version on distinct and heterogeneous devices, thus allowing the development of the first generic Distributed Computing Systems [BBB96]. The second half of the nineties witnessed an increased development rate, with the application of novel techniques to solve the presented problems. This evolution can be seen in the following illustrative yearly work distributio of papers and their related systems:

- **1995** : ATLAS [BJK+95]
- **1996** : ATLAS [BBB96] and ParaWeb [BSST96]
- **1997** : IceT [GS97], SuperWeb [SSP+97], JET [PSS97, SPS97], Javelin [CCI+97, CCNS97] and Charlotte/KnittingFactory [BKKK97],
- **1998** : Java Market [AAB98b, AAB98a], POPCORN [NLRC98] and Charlotte/KnittingFactory [Kar98, BKKK98],
- **1999** : Javelin [NBK+99], Bayanihan [SH99, Sar01] and Charlotte/Knitting-Factory [BKKW99]

In the 21st century the development and research on Internet Distributed Computing continued, seeing the advent of the peer-to-peer architectures for resources discovery and work distribution:

- **2000** : Javelin [NPRC00], MoBiDiCK [DH00] and XtremWeb[GNFC00]
- **2001** : Bayanihan [Sar01] and XtremWeb[FGNC01]
- **2002** : JXTA-JNGI [VNRS02, Mik05], P^3 [OLS02] and CX [CM02]
- **2003** : G2-P2P [MK03a, MK05]
- **2004** : CCOF [ZL04b]
- **2005** : Personal Power Plant [STS05], CompuP2P [Sek05], G2DGA [Ber05], Alchemi [LBRV05b, LBRV05a] and BOINC [AKW05]
- **2006** : YA [CA06] CompuP2P [GS06], and BOINC [AF06]
- **2007** : Leiden[Som07, Lei], Ginger [VRF07b], and BOINC [And07]
- **2008** : NuBoinc [SVF08b]
- **2010** : Ginger [RRV10, EVF10]
- **2011** : Ginger [MSFV11, OFV11]

As stated before, this list only presents Internet based Distributed computing systems. Although having some similar problems, Enterprise Desktop Grids (such as Entropia [CCEB03]) were not addressed, because their are not free.

Also in the area of institutional clusters and the Grid, there has been some work on gathering remote donors. For instance, in Albatross [BPK+99, KBM+00] most work has been done in the development of an infrastructure for the consolidated use of distributed clusters. Albatross optimizes work distribution taking into account communication latency between clusters. LiveWN [KG06] allows ordinary users to donate cycles to a Grid, by executing Grid middleware inside a virtual machine. Although users can easily donate cycles, a pre-existent Grid infrastructure (with all complex security configurations) must be set-up.

These systems are of no interest to this study since some sort of prior organized and administered infrastructure should exist (clusters in Albatross and a Grid in LiveWN), making them impractical, or otherwise inaccessible and/or unusable to a regular home user.

Although SETI@home [ACK+02] was a predecessor of BOINC, it is not listed due to the fact that it is tied to a single specific problem, not being generic and not allowing the creation of different projects.

2.2.1 Architecture

Although the CPU speed of the donating hosts is the most important factor to determine individual task execution time, other architectural decisions have a fundamental impact on the overall system performance, such as jobs speedups, fairness on the selection of tasks, improvement or optimization of resource utilization.

With respect to architectural characteristics, it is necessary to include different implemented network topology and organizations, how resources are evaluated, what scheduling policies are used and how work distribution is performed.

2.2.1.1 Network Topology

Different systems employ different network topologies, with different organization of the various involved entities (donors, clients, or servers).

This architectural decision affects the kind of participating entities, their interaction, the overall complexity and the system efficiency. These can be, in order of increasing flexibility and decoupling:

- Point-to-point
- Single server
- Directory server
- Hierarchical servers
- Replicated servers
- Peer-to-peer

Point-to-point In a point-to-point topology: the user having work to be executed must own and operate his own server. The creation of jobs must be performed in that server. At a later instant, computers owned by the donors contact directly the servers owned by the clients (those needing processing time and creating work to be done).

As it is the responsibility of the user submitting the jobs to install and maintain all the hardware and software necessary to provide work to the donors, this simple solution adds burden to the client user. Furthermore, there is no reuse of previously installed infrastructures.

On the donor side, this solution is not very flexible. This user has to exactly know the identification of the server where certain jobs are hosted, and it is impossible to load balance the requests to distinct computers.

Single Server By decoupling the place where jobs are created (client computer) and the server where they are stored, it becomes possible for a single server to hosts projects and jobs from users geographically dispersed. This solution allows the reuse of the infrastructure, and the reuse of the processing code. Users can now submit different data to be processed by previously developed code.

This architecture requires the existence of remote job creation mechanisms. The existence of a simple user interface is enough.

This architectural solution still requires, as in a point-to-point architecture, the donor to exactly know the identification of the server hosting the jobs to be executed. Furthermore, both these architectures have a single point of load and failure.

Directory Server With the addition of a directory server, the problems with the point-to-point and single server approaches, with respect to server identification and location, are solved. The donors contact a directory server that redirects requests to one of the servers supplying jobs. Although the donor still has to know the identification of a computer (the directory server), this server will act as an access point to various work sources.

The architecture of the underlying server may follow the patterns presented earlier (single server or point-to-point). Either the servers are shared among different clients by allowing the creation of jobs from a remote location, or a server can only be used by its owner. Furthermore, all communication is still performed directly between the donor computer and the server where jobs are stored.

Even with a directory server, scalability and availability problems continue to exist. For each job, there is still only one server hosting its tasks. All donors wanting to execute such tasks contact the same server, that may not have enough resources to serve all requests, in particular to transfer all data efficiently. Also, in case of failure there are no recovery options, being impossible for any donor to execute tasks from jobs hosted on failed servers.

Hierarchical Servers In order to tackle scalability issues, the simplest way is to partition data among several servers. In the case of Distributed Computing systems, this division can be made at different granularity levels: at the job level (different jobs hosted on different servers) or at the task level (different tasks on different hosts).

In order to manage a set of systems, the simplest solution is to organize them in a tree structure. This hierarchical structure can range from the simple two tier architecture (with one coordinator and a set of servers), to a deeper organization.

The most evident use of a hierarchical infrastructure is to balance server load.

Taking this into account, when jobs (or tasks) are created, the server responsible for them can be the one least loaded. In this case, a job (and all its tasks) can be hosted on a server or have its tasks distributed among several servers.

Systems that allow the creation of recursive tasks (allowing tasks themselves to spawn new tasks) also fit well in a hierarchical architecture. When tasks are created, a server to host the tasks is selected. In order to maintain information about dependencies, it is necessary that each server maintains a list of the servers hosting the sub-tasks. This requires a tree-based architecture, reflecting the task dependencies.

Since each server stores parts of the jobs, some level of load balancing is possible and scalability is attained. Nonetheless, even with this solution, availability is still an issue, since, if a server crashes, the work stored there becomes unreachable.

Replicated Servers The replication of task data among several servers addresses both availability and scalability issues.

In systems with replicated servers, the information about a single task is stored in more than one server. When submitting work, the client contacts one of the available servers. The way the storage of the data is performed is transparent to the user, without having any hint about what server will host his tasks. Later, when donors contact the system in search of work, one copy of a task is retrieved.

Replication can be performed on two different levels: server and task. In the first case, all the data stored in a server is replicated on other servers. If the replication is at the task level, there is no strict mapping between the information stored on the server: replicas of tasks from the same jobs can be stored on different servers.

In the case of failure of a server, as replicas of that data are stored elsewhere, the system continues operational, maintaining all task information accessible. Furthermore, with this solution, donors do not have to contact the same set of servers to retrieve tasks from a given job, which also tends to balance load across servers.

The algorithm to distribute tasks among servers is not the only issue this new topology raises: the management of the correct execution of those tasks is also a complex issue. Since multiple copies of the same task exist in different servers, the multiple execution may raise global issues of inconsistency of jobs state.

If tasks are idempotent, the multiple execution of the same task causes no harm; thus if the system only allows such type of tasks the replication implementation is straightforward. On the other hand, if tasks are not idempotent, it is necessary to guarantee that before starting a task, no replica of it was previously started. These verification mechanisms and the communication overhead may hinder the gains from replication, in the case of shorter tasks.

There is no strict relation between replicated storage and replicated execution of tasks. In section 2.2.2 it is presented how replicated execution of tasks is performed and its uses, an what problems it solves.

Peer-to-peer The most distinct characteristic of a peer-to-peer architecture is that any intervening computer (peer) can perform at least two simultaneously, yet distinct roles: i) a peer is a donor, by executing tasks submitted by others, and ii) acts as a server because it stores data, results, and helps on work distribution. Furthermore, an intervening computer can also act as a client, from where jobs and tasks are created.

While on other architectures, dedicated servers are used to store data, in a peer-to-peer system, the data is on the edge of the Internet on often insecure, unreliable computers. This problem requires more resilient solutions than on other systems: i) the distribution of job data into different nodes is essential not to overload a single computer, and ii) replication of data should be implemented to guarantee that the (highly probable) failure of a peer does not compromise the execution of a job.

With respect to donors' high volatility, the problems on a peer-to-peer infrastructure are similar to those observed in systems relying on dedicated servers.

Also, in this architecture, there are often no guarantees about the offered quality-of service. These issues will be described in Section 2.2.2.

The entry point for a peer-to-peer network can be any of the already participating nodes. Furthermore, any peer may only know (and contact with) a limited number of peers. These issues affect normal operation of the system: the way discovery and selection of tasks is performed, how remote resources are discovered, and how information is exchanged between the peer acting as client and the one acting as donor. These issues will be detailed in the following sub-sections.

2.2.1.2 Resource Evaluation

Tasks require various kinds of resources to be efficiently executed. Due to the heterogeneity of the donors, it is necessary to keep track of the characteristics of the donors to perform resource usage efficiently, and to optimize individual tasks and global jobs execution.

The way the characteristics of available resources are monitored, evaluated, and information about them is kept, is also an important factor for the overall system performance. There are three different approaches:

- Centralized database
- Decentralized database
- Polling

Centralized Database The use of a centralized database is the simplest of the resource evaluation methods. When a donor registers for the first time on the system, information about the available resources is stored in a database.

This solution has a few important drawbacks: only a single server is allowed, and the rate with which resource availability changes may not be too high.

Even if several servers are used to store task information, the use of a centralized database overrides any gain from the use of multiple servers (as described previously). There is one central failure point, reducing both availability and scalability.

If the resources available on the donor change with a high frequency, the rate at which the database has to be updated may not scale to large number of donors.

If none of the previous issues are problematic, the use of a single database is effective and offers efficient resource discovery, and subsequently more efficient job execution and resource usage.

Decentralized Database In the same way as data can be split among several computers to increase availability and scalability, so the information about donors (characteristics and available resources) can be stored in decentralized database.

The use of a decentralized database for storing the available resources information, not only solves some of above problems but also fits nicely on peer-to-peer systems, or those using multiple servers. When using a distributed database, the burden of selecting a donor is distributed, and the failure of a server does not become catastrophic.

On systems with a single database, update frequency can become a problem due to the network traffic it may generate. With a distributed database, although information still needs to be transferred, it can be done to a server closer to the donor: the bottleneck of having a single server is avoided, and the messages have to transverse fewer network links.

The information stored in the distributed database, can be either replicated or partitioned. If the information is replicated, some inconsistency among replicas must be tolerated: one drawback is the possible selection of non-optimal hosts to execute a task; another one is the need to contact a second server to retrieve information about a suitable donor.

Polling If it is impossible to maintain a database (either centralized or decentralized) the only solution to discover the resource characteristics at a given moment, is by directly polling donors to gather information about their available resources.

With an efficient donor discovery infrastructure, polling those computers is a straightforward step, but requiring extra care to guarantee close to optimal answers without too much network bandwidth consumption. To select the optimal donor for a task it would be necessary to poll every donor available, a solution that is impractical.

While with a database, it is possible to have an overall vision of the available resources, allowing the selection of the best donors; with polling that becomes difficult. A complete vision of available resources requires a full depth search, impossible for large systems. Thus, only a partial view of the system is possible at a given moment, and the selection of the donors may not yield the optimal answer.

So, the use of polling to evaluate remote resources guarantees that the information about contacted hosts is up-to-date (when donors can be contacted), but does not guarantee that the best available host is used to execute every task. The use of firewalls may limit the connection to donors, requiring different strategies to overcome this fact (for instance piggybacking of requests).

2.2.1.3 Scheduling Policies

Along with the resource discovery and evaluation, the way jobs are scheduled is also important to the performance of the system. The available policies range from the simpler one (eager) to a complete heuristic matchmaking between available resources and tasks requirements:

- Eager
- Resource aware
- Heuristic
- Market oriented
- User priority

Eager When an eager scheduling policy is used, the selection of hosts to execute available tasks is blind, neither taking into account the characteristics of the computer nor the possible task execution requirements. This is only possible if a uniform execution environment exists: either all hosts have the same architecture and operating systems, or the applications execute inside a common virtual machine.

Whenever a donor host is idle and requests some work, the system assigns it a task: randomly selecting it or using a FIFO policy.

Some level of fairness can be guaranteed (by first assigning previously created tasks) but this offers no guarantee about the optimal resource usage. This solution is the simpler to implement since no global information about available resources is needed.

Resource Aware In order to optimize the execution of tasks, a resource aware scheduling policy assigns tasks to the more capable machines before assigning tasks to other less capable.

This solution still picks tasks to be executed either randomly or in a FIFO manner, but then, for the task in question, it selects a machine with enough resources (e.g. memory or CPU speed). This selection is made taking into account task requirement information.

To implement this solution, it is required the existence of a database (where servers query for the best donor), or polling donors to discover their resources.

This donor selection method guarantees that every task's requirements are met and that each task is executed close to minimum possible execution time. On the other hand, this method does not minimize the overall makespan of a set of tasks or job, since different donor assignments (with delay on a task starting), and task execution orders, could lead to a best overall performance.

Heuristic When selecting execution hosts, previous policies select tasks by a predefined order (or randomly) and just take into account each task's individual resource requirements.

With a complete knowledge of tasks requirements and available resources it would be possible to schedule tasks to the best hosts in order to minimize every job makespan. In a highly dynamic environment, this feat is impossible: i) it is difficult and costly to maintain updated information about all available resources, and ii) the exact execution times for each task is often impossible to predict.

To reduce a job's makespan it is necessary to use heuristics. These empirical rules, although not providing the optimal solution, allow better job execution times than blind or just resource-aware scheduling.

Market-Oriented Previous scheduling policies resort to the information about available computational resources for assigning tasks to donors. This limits any user intervention on the selection of the hosts to execute tasks.

This host and task matching can be changed and manipulated by both clients and donors if a resource market exists.

In this new market-driven environment, the selection of the tasks to be executed is made taking into account some bidding mechanism and using some sort of currency earned by executing other peoples' jobs.

Clients state how much they are willing to pay for the execution of their tasks, donors decide the cost of their resources, and a matching algorithm (implementing available bidding mechanisms) matches the tasks with the donors.

Buya et al [BAV05] present an overview of market oriented mechanisms presented on current Grid systems. In this document, the presented auction types are those currently in use: i) English auction, ii) Dutch auction, and iii) Double auction. The efficiency of each one of the auction mechanisms is also evaluated and presented.

While the English auction method is the mostly 0used mechanism (where buyers increase the value they are willing to pay), its straightforward implementation in a wide area distributed system has a high communication overhead. The Vickrey [LR00] method is an efficient auction methodology that can replace the English auction, thus reducing the communication overhead. In a Vickrey auction, buyers bid the product by stating the highest value they are willing to pay, the winner of the auction is the one that bid with the highest value, but only pays the value of the second highest bid plus one monetary unit.

When using a Dutch auction mechanism it is the responsibility of the seller to try to match the price of the resource to the buyer's expectation. In this case, the seller decreases the price until a buyer is willing to accept it. Although still requiring communication between the buyer and the seller, it has a lower overhead than the English auction method. This iterative method can be replaced by sealed first-price auction [MW82], where the buyer places a sealed bid and the one with the highest value is chosen.

In a Double auction [PMNP06, WWW98], both buyer and seller "bargain" the price of the resource: the seller asks for a price and the buyer bids for it. This can be a single step or repeated until the asked price is equal or lower than the bid one. This is the equilibrium price that is reached when, after a few iterations, supply equals demand.

User Priority Although the selection of the hosts to run a task is important, another fundamental issue to scheduling efficiency is task selection. Besides matching tasks to suitable hosts, on the server side, it is necessary to select which users will have their tasks executed first.

Several users can have tasks that compete for the same resources and the system should have means to prioritize those tasks. Heuristic policies can decide what task is to execute next, but this selection can be dependent only on the user that submitted the work, assigning each user a different priority. This priority mechanisms can be static, where each user has a predefined priority, or they can vary with time. Furthermore, users can be grouped in classes according to, for instance, resources donated or reputation.

2.2.1.4 Work Distribution

Both the network architecture and scheduling policies affect the way work distribution (from servers to donors) is performed. Work distribution can be characterized in two independent axes: i) who initiates the process, and ii) whether it is brokered by a third party:

- Direct pull
- Direct push
- Brokered pull
- Brokered push

Pull vs Push If it is the donor that initiates the request for a new task, the system uses a pull mechanism. After completion of a task, the donor contacts one server in order to be assigned more work.

In the case of push, it is the initiative of the server storing information about a task to initiate it. A donor is selected, contacted, and the task's data is transferred.

The pull mechanism is more efficient when the donor also performs some sort of local scheduling: by selecting the servers to contact and projects to execute. Depending on the execution time spent on the various jobs, it is the donor that selects where the new tasks come from. Furthermore, there is no need for a database to store resource information: the donor, when requesting for work, may inform the server about its available resources.

If there is a centralized database with the resources available on the various donors, the push mechanism can be easily implemented. The server hosting the resource information database knows the characteristics of the donors and with that information, it can easily assign them new tasks.

Direct vs Brokered If the system only has one server, the distribution mechanism is necessarily direct, independently of the direction (pull or push). When starting a task, the donor contacts (or is contacted by) a server that can provide task data without the intervention of any other server.

In the case of a peer-to-peer architecture, or when using several servers, the donor may not contact directly the machine storing the task's data.

In the case of a brokered pull, if the job or project is not owned by the server (or peer) contacted, or has no tasks capable of being executed, it is the responsibility of that server (or peer) to discover a suitable task. The server (or peer) forwards the request and, after receiving suitable data, delivers it to the donor.

The brokered push mechanism is mostly used in peer-to-peer architectures. The peer storing a task contacts neighbouring peers trying to find a suitable host to execute it. If these contacted peers are incapable (because they are unsuitable or are not idle), they forward the request to other peers.

2.2.1.5 Analysis

Table 2.1 summarizes the various combinations of architectural characteristics of the available Distributed Computing systems described in this section. Some classifications are not present due to the unavailability of information and their description in the scientific literature.

	Network Topology	Resource Evaluation	Scheduling Policies	Work Distribution
ATLAS	Hierarchical servers	-	Eager	Direct pull
ParaWeb	Multiple servers Directory server	Polling	-	Direct push
Charlotte KnittingFactory	Single server Directory server	Centralized DB -	Eager -	Direct pull Brokered pull
SuperWeb	Single server	Centralized DB	-	Direct push
Ice T	-	-	-	-
JET	Hierarquical servers	-	Eager	Direct pull
Javelin	Hierarquical servers	Decentralized DB	Eager	Brokered pull
Java Market	Single server	Centralized DB	Market oriented	-
popcorn	Single server	Centralized DB	Market oriented	-
Bayanihan	Hierarchical servers	Decentralized DB	-	-
MoBiDiCK	Single server	Centralized DB	Resource aware	-
XtremWeb	Single server	Centralized DB	Resource aware	Brokered pull
JXTA-JNGI	Replicated servers	Decentralized DB	Eager	Brokered pull
P^3	Peer to Peer	Decentralized DB	-	Brokered pull
CX	Multiple servers	Decentralized DB	Eager	Direct pull
G2-P2	Peer to Peer	Polling	Eager	-
CCOF	Peer to Peer	Decentralized DB	Resource aware	Brokered push
Personal Power Plant	Peer to Peer	Polling	Eager	Direct pull

Table 2.1 (Continues on next page)

	Network Topology	Resource Evaluation	Scheduling Policies	Work Distribution
CompuP2P	Peer to Peer	Polling	Market oriented	Brokered push
Alchemi	Single server	Polling	Eager	-
YA	Peer to Peer	Decentralized DB	Resource aware	Brokered push
BOINC	Point to Point	Centralized DB	Eager Resource aware	Direct pull
Leiden	Point to Point	Centralized DB	Eager	Direct pull
Ginger	Peer to Peer	Decentralized DB	Resource Aware	Brokered pull
NuBoinc	Point to Point	Centralized DB	Eager	Direct pull

Table 2.1: Internet Distributed Computing systems architectural decisions

Network Topology The late nineties saw the development of many systems, each one with a different approach to its architecture. While on some systems the architecture was not a fundamental issue (using simply point-to-point or a single server), for others the architecture was fundamental (due to efficiency issues and due to the proposed programming model).

The simpler point-to-point architecture was initially implemented by ParaWeb, while the use of a different server from the client computer was implemented by Charlotte, SuperWeb, Java Market, POPCORN and MoBiDiCK.

In order to balance load across servers, JET uses a two layer hierarchical architecture, where there is a single JET server and a layer of JET Masters, that interact with the donors. Bayanihan uses a similar approach to tackle network limitations, taking advantage of communication parallelism and locality of data.

In Atlas and Javelin, any task information is stored in one of the available servers. Each of these servers also manages a set of clients. Furthermore, as will be presented in section 2.2.3.10, these systems allow any task to create and launch new ones.

These two characteristics allow a simple load balancing mechanism. Whenever a new task is created, its data is stored in the least loaded server. Since previously executing tasks depend on new ones, it is necessary to maintain connections between the several servers hosting related (child and parent) tasks, creating a tree-based structure.

Furthermore, both ATLAS and Javelin allow work stealing as a mean to distribute work. Whenever a donor is free to do some work, it contacts a server. If that server has available work, it assigns it immediately to a donor. In the opposite case, the initially contacted server needs to finds a server with available work. This way, a hierarchical and recursive architecture also emerges.

Most of the later projects present a peer-to-peer architecture, where donors also act as servers with management tasks (storage of work and results, distribution of work, . . .) and clients: XtremWeb, P^3, G2-P2, CCOF, Personal Power Plant, CompuP2P and YA.

JXTA-JNGI clearly makes a distinction between donors, clients and servers but uses a peer-to-peer infrastructure to manage communication between replicated servers.

Ginger implements a peer-to-peer network topology, with distinction between regular and Super Nodes. These special nodes store information about the reputation of a set of regular nodes.

Other recent projects implement simpler architectures: CX, Alchemy, BOINC, Leiden, and nuBOINC. CX allows the existence of multiple servers (each managing distinct sets of work), Alchemi uses onlys a single server, and BOINC, Leiden and nuBOINC rely on the simple point-to-point architecture for the distribution of work.

Resource Evaluation The systems that require up-to-date knowledge of the donors characteristics perform polling whenever work is sent to the donor. In the case of G2-P2, Personal Power Plant, and CompuP2P, whose architecture is peer-to-peer, it is natural that polling is required to know the exact characteristics of the donors. In the case of ParaWeb and Alchemi, no information about donors is stored in the server, so it is necessary to poll them whenever work is to be executed.

All other systems, independently of their architecture, use some sort of database. If the architecture relies on multiple servers (hierarchical, replicated or peer-to-peer) the database is decentralized.

Scheduling Policies The simplest of the scheduling policies (Eager) is also the most adopted. In systems using the eager scheduling policy, users do not have to specify any execution requirement. This, not only eases job creation, but also reduces task scheduling implementation. In these systems, the architecture of the donor is not an issue, since these systems require clients to use particular interpreted languages for the development of the task's code (as will be seen in Section 2.2.3.7). The only requirement is the availability of a uniform execution environment (such as Java). Although donors have different execution power (available memory and execution speed), on systems using eager scheduling, these characteristics are not taken into account when selecting the host to run a task.

The systems that are resource-aware assign tasks taking into account donors' characteristics (such as processor, operating systems) or availability. For instance, MoBiDiCK donors specify the time slot when their computers can donate cycles: this information is used when scheduling tasks. CCOF uses the same idea of optimal resource availability and automates the execution and migration of tasks. Here, users do not have to specify the availability slots, since CCOF uses current time and assumes computers are only available during the night. Tasks are scheduled taking into account donors' current time, selecting donors that are available, and also imposing the migration of executing tasks at dawn. In XtremWeb, BOINC, and Leiden, clients must develop one executable for every architecture, so it is necessary to match the donor's architecture (processor and operating system) with the suitable executable. Ginger goes one step further, as the matching of clients with donors also takes into account the reputation, and historic data, of the donors.

Java Market, CompuP2P, and POPCORN use a market-oriented approach, by matching a value offered by the client with the one required by the donor. In

Java Market, this matching is performed automatically, after the client submits the required resources, and the value it is willing to pay back for the execution of the task in certain amount of time. In the case of concurrent execution of tasks, those that maximize benefits for the donors are chosen, this solution is close to a sealed first-price auction. CompuP2P uses Vickrey auctions to assign tasks to the less expensive donor, after donors state the cost of the resources. POPCORN offers three different auction types: Vickrey, a sealed-bid double auction (where both parties define a lower and higher bound for the price of the resource), and a repeated double auction.

Work Distribution The way work distribution is performed is partly dependent on the system's architecture. Systems that rely on a single server necessarily perform direct task distribution, either pulling or pushing work.

In systems with more complex architectures, the distribution can be brokered so that the donor does not have to know, and contact, directly the server or peer owning the task (KnittingFactory, Javelin, XtremWeb, JXTA-JNGI, P^3, CCOF CompuP2P YA and Ginger). In other systems, the various servers (or peers) are used to find the interlocutor, but task transfer is performed directly between the donor and the server that stores the task information (JET, CX and Personal Power Plant).

The work distribution method (pull or push) is also related to the scheduling policies. Systems that have an eager scheduling policy use a pull mechanism: when idle, the donor contacts a server or another peer and receives the work to be processed. The distribution can be both direct or indirect, as previously explained.

2.2.2 Security and reliability

The second class of relevant characteristics of Distributed Computing systems are those related to security and reliability, either on the donor side and on the client side: i) privacy of the data, code, and client identity, ii) result integrity against malicious donors, iii) reliability against donor and network failure, and iv) protection of the donor computer against attacks.

2.2.2.1 Privacy

Some of the work that can be deployed and executed on the Internet can have sensitive information: the data being processed or even the code to be executed. The identity of the user submitting the work should, in some cases, be kept secret.

So, on the client side, these Distributed Computing systems should guarantee the following kinds of privacy:

- Code
- Data
- Anonymity

Code / Data The developer of the code to be executed can have concerns about privacy guarantees of both the code and the data. The algorithms used can be proprietary as well as the data.

The mechanisms to guarantee code privacy are similar to those of data.

By encrypting the communication between the server and the donor it is possible to enssure that, from an external computer, it is not possible to access the downloaded information.

During the execution of the tasks, on the donor's computer, the privacy of the data and code should also be preserved. Executing the task inside a sandbox and not storing any information in files outside it can promote the guarantee that no external malicious process running in the donor can access the code. This solution does not fully prevent the access to the information, as as compromised/modified execution environment (sandbox or virtual machine) has access to whole task state.

Anonymity Another information that can be hidden from the donor is the identity of the client, guaranteeing some sort of anonymity. Guaranteeing anonymity is important since, with information about the owner of the tasks, it is possible to infer the the purpose of the work.

On the other hand, donors may also want to guard its identity, not allowing others to know what work they are executing.

In any architecture employing servers, it is their responsibility to enforce and guarantee this two-way anonymity. Although the server must know who created and executed each task, it may not disclose it to third parties. In a peer-to-peer architecture, the distributed architecture may ease the implementation of anonymity mechanisms.

2.2.2.2 Result Integrity

Since most machines that execute tasks on Distributed Computing systems are not under the control of the user submitting jobs, it is fundamental to verify result correctness after the conclusion of a task. Only if that happens, is it possible to guarantee that those results are the same as the ones that would be obtained in a controlled and trusted environment.

There are several techniques to verify if the results are not tampered with nor forged:

- Executable verification
- Spot-checking
- Redundancy
- Reputation

Executable Verification The simplest and less intrusive method to verify if the results were produced by a non-tampered application is to perform some sort of executable verification.

The middleware installed at the donor computer calculates the checksum of the executable being executed and compares it with checksum of the correct program (a value previously calculated and stored in the server or obtained from a trustable source).

This method just guarantees that the executed code is the one provided by the client, and that the output produced is correct. On the other hand, this method does not guarantee that the result received by the client is the one obtained: it is still possible to tamper with and modify the result transmitted by the donor.

Spot-checking With spot-checking, what is verified is not the correctness of the results, or the executable producing them, but the reliability of the donor.

Systems that use spot-checking generate quizzes enclosed in dummy tasks, whose results are previously known. These tasks are periodically sent to donor computers, that treat them as regular tasks. Comparing the returned result with the expected one, it is possible to know if results returned by that host are to be trusted.

This method still does not guarantee that the results transmitted from the donors are correct: only that the dummy tasks were executed correctly, and that, with a high probably, the donors are reliable, provided that they cannot identify dummy tasks among normal ones.

In the case of discovery of a donor with incorrect behavior, the server can take the appropriate measures about the results previously returned by that donor, or about future work to be sent to it.

Redundancy With redundancy, several instances of the same task are executed on different hosts. The results returned by each execution are then compared and used to decide which result is considered correct. Usually, a simple voting is performed and the result obtained by the majority of the executions is the one considered correct (quorum). If a decision is not reached, more instances of that task are launched.

Redundancy can only be used if tasks are idempotent; only in this case is it safe to execute several times the same task. Furthermore, special care should be taken if the system is market driven, since the execution of multiple instances of the same task has a higher cost to the client.

If a compromised donor executes several replicas of the same task, the result can be forged. Thus, the execution of replicas of the same task must be performed by distinct donors, and, sometimes, a task replica (a sample) can be executed by the client itself to screen donors.

Reputation In addition to the use of any of the previous techniques, it is possible to implement a reputation scheme, where information about incorrect results is used on the calculation of a donor's reputation.

This reputation value can then be used when scheduling tasks, by assigning tasks to the most reliable donors, or after receiving the result, to know whether some additional result verification is necessary.

Although a reputation scheme may not fully guarantee the result integrity, it may help increase system performance by allowing the execution of tasks on more reliable, and non-compromised donors.

2.2.2.3 Reliability

Even if donors are not malicious and execute all tasks correctly, they can fail or get disconnected from the Internet. It is still necessary to guarantee that, in case of failure, the job gets executed and results produced. This can be accomplished using one of the following techniques:

- Redundancy
- Restarting
- Checkpointing

Redundancy Besides result integrity checking, redundancy can also be used in order to increase the reliability of a system.

By launching several instances of the same task, the probability that at least one task concludes increases (even in the case of donor failure). With redundancy, the number of tasks to launch is fixed. Even if the first task concludes in the expected time frame, redundant tasks (in this case, unnecessary) were still created and partially executed. This drawback is overridden if redundancy is also used to guarantee result integrity.

When using redundancy, slower computers may not be rewarded for the execution of tasks, since before these slower computers can return the results, a faster machine has already done so.

Restarting To solve the problem of blind execution of replicas of the same task, a small modification to the replica launch decision mechanism can be made.

Only after all tasks have been launched at least once, and when some results are still due, the system decides to relaunch tasks: those not finished are to be restarted.

Again, as in redundancy, either the previously started instance or the new one will eventually finish, yielding the expected result: either the first task was executed on a slow computer, or that donor went off line or failed.

Checkpointing Both redundancy and restarting have problems when tasks are not idempotent or there is some form of payment for the execution of tasks, or in the case of long-running tasks: i) non-idempotent tasks can not be executed multiple times, and ii) users may not be willing to pay more than the real execution time (improbable feat if tasks are restarted or replicated).

To solve these issues, it is necessary to guarantee that no piece of a task is executed twice. So, if a donor crashes, the tasks need to be restarted from the last correctly executed instruction. To do so, some sort of checkpointing should be periodically performed.

Besides the idempotence and payment concerns, by using checkpointing, the conclusion of restarted tasks is faster, since restarting a task does no require it to be executed from its first instruction.

This method requires periodic saving of each tasks' state and recording of executed instructions since that point. The use of the donor's local non-volatile memory to store the checkpoint reduces checkpointing time, but does not allow the efficient restart in case of complete donor failure. This event requires the state to be saved in a remote non-volatile medium.

2.2.2.4 Execution Host Security

On the donor's side, some security precautions should be taken into account in order to prevent malicious code from executing and cause any harm. To avoid such threats a few solutions are possible:

- User trust
- Executable inspection
- Sandboxing

User Trust The more lax mechanism is the simple trust on the users submitting the tasks. No external verification mechanisms re needed, as donors believe in the integrity of the programmers that developed the downloaded code, and that no harm comes from executing it.

This requires that owner of the server, the task creator, and also the purpose of the code to be well know by the potential donors.

Executable inspection In order to execute tasks in a non modified environment, the code must be inspected and audited to guarantee that it does not contain harmful instructions.

This can be performed off-line in a trusted server that performs an executable verification, and signs the application to prove that it was verified and considered harmless.

This technique allows the safe execution of applications on a non-modified environment, but requires the extensive study and analysis of the tasks' code.

Sandboxing If it is impossible to guarantee the correctness of the downloaded code, it is necessary to isolate its execution. In the case of the execution of a malicious instruction, no harm should come to the host computer.

With sandboxing, the downloaded code is executed in a restricted environment or virtual machine, not allowing harmful operations to interfere with the host operating system. The execution environment also intercepts accesses to the host file system, preventing access to system files. The execution environments can be either application-level or system-level virtual machines.

High-level languages language (such as Java, or .NET) execution environments fall in the category of application-level virtual machines. Although these environments allow the execution of applications with full OS access, it is possible to define security policies to restrict access to some resources.

System-level virtual machines mimic a fully functional computer, but isolate the host operating system. Any application executing inside such environments will see and execute within a fully functional guest operating system without affecting, modifying, or compromising the host computer.

Adding to this advantage, the use of a virtual machine eases the development of portable code providing an uniform execution environment. As long as users can install the virtual machine, the virtualized execution environment (hardware characteristics and software versions) is similar in all donors.

2.2.2.5 Analysis

The following table (Table 2.2) briefly presents how each system solves the security and reliability issues.

	Privacy	Result Integrity	Reliability	Execution Host Security
ATLAS	Anonymity	-	Checkpointing	User trust Sandboxing
ParaWeb	-	-	-	-
Charlotte KnittingFactory	-	-	Redundancy	Sandboxing

Table 2.2 (Continues on next page)

	Privacy	Result Integrity	Reliability	Execution Host Security
SuperWeb	-	-	-	Sandboxing
Ice T	-	-	-	User trust Sandboxing
JET	-	-	Checkpointing	Sandboxing
Javelin	-	-	Redundancy	Sandboxing
Java Market	-	-	-	Sandboxing
popcorn	-	-	-	Sandboxing
Bayanihan		Redundancy Spot-checking	Redundancy	Sandboxing
MoBiDiCK	-	-	Redundancy	-
XtremWeb	-	Executable verification	Restarting	User trust
JXTA-JNGI	-	-	-	-
P^3	-	-	Checkpointing	-
CX	-	-	Redundancy	-
G2-P2	-	-	Checkpointing	-
CCOF	-	Spot-checking Reputation	-	Sandboxing
Personal Power Plant		Redundancy Voting	-	-
CompuP2P	-	-	Checkpointing	-
Alchemi	-	-	-	Sandboxing
YA	-	-	-	-
BOINC	-	Redundancy Reputation	Restarting Checkpointing	User trust
Leiden	-	Redundancy Reputation	-	User trust
Ginger	-	Spot-checking Reputation	-	Sandboxing
nuBOINC	-	Redundancy Reputation	-	User trust

Table 2.2: Internet Distributed Computing systems security concerns

Privacy Privacy of the code and data are difficult to guarantee, thus almost never being referred to in the literature. Although some systems use virtual machines (mostly Java or .Net virtual machines) to execute the code, after the data and code have been downloaded, malicious processes running on the donor can access that information.

Although anonymity could be enforced by most of the available systems, only the papers describing ATLAS explicitly refer client anonymity. The authors state that a donor cannot obtain the identity of the client that submitted the work. Although not stated by other systems' authors, any of those not having a point-to-point architecture (with distinctions between server and client computers) can easily guarantee anonymity.

Other systems (BOINC, ParaWeb, Charlotte/KnittingFactory, POPCORN) rely on the knowledge of the identity of the client. In these systems the user donating resources explicitly chooses the jobs and projects to be allowed to run on the donor computer.

Result Integrity Most of the studied systems do not present any solution to guarantee result integrity in the presence of malfunction or malicious donors.

From those systems that implement a solution to prevent result corruption, redundancy is the one adopted by the majority (Bayanihan, Personal Power Plant, BOINC, Leiden, and nuBOINC). These systems launch several identical tasks and, after receiving the results, decide about the correct answer. Most of the systems resort to a vote counting mechanism on the server side: a result is considered valid if a majority of computers returned that value. Personal Power Plant, due to its peer-to-peer architecture, must use a distributed voting algorithm.

Bayanihan, CCOF, and Ginger use spot-checking to verify the correctness and trustiness of a donor, issuing dummy tasks with previously known results.

XtremeWeb donor software calculates the checksum of the downloaded executable before starting the tasks. By comparing it with the checksum calculated on the server it confirms that there was no tampering with the executable. Besides executable verification, XtremeWeb offers no other mechanism to verify if the results transmitted from the donor were the ones calculated by the verified executable.

CCOF, BOINC, Leinden, Ginger and nuBOINC implement reputation mechanisms to guarantee that non-trusted donors do not interfere with the normal activity of the system. CCOF resorts to the results of spot-checking while BOINC, Leiden, and nuBOINC use previously submitted results to classify donors in different trust levels. Ginger uses both results from spot-checking and real tasks to calculate the reputation of a donor.

Reliability Reliability, guaranteeing that a task eventually completes, is not tackled by all of the studied systems.

As expected, CompuP2P (a system that is market-oriented) implements checkpointing. For a market-oriented system, any other method would increase the value to pay for the execution of a task: redundancy requires the launch of several similar tasks, even if there is no need, and with task restarting the work done until the donors failure would be wasted. For the other market-oriented systems (Java Market and POPCORN), there is no information about the mechanisms to ensure some level of reliability.

Five more systems implement checkpointing with recovery of the tasks on a different donor: ATLAS, JET, P^3, G2-P2 and BOINC, with different levels of abstraction.

G2-P2 periodically saves the state of the application and records all posterior messages. These checkpoints can either be saved on the donor local disk or on another peer. The first method is more efficient, but requires the faulty donor to recover to restart the task.

Checkpointing is the only method that can be used with tasks that have side effects. The other methods, which are also the simplest, require all tasks to be both independent and idempotent. Three systems implement the restarting of tasks when only some results are missing to complete a job: XtremWeb, BOINC, and Leiden.

In systems where resources are truly free, redundancy can be used without further cost to the client. Although redundancy can be used to guarantee result integrity and system reliability, there are some systems that do not take advantage of redundancy to tackle both issues. As a means to guarantee result integrity, BOINC only uses redundancy when requested by the user, while Charlotte, Javelin, Bayanihan, MoBiDiCK and CX do not perform any result verification.

Execution Host Security Although systems that use an high-level language to implement tasks execute the code within a virtual machine, not all of these are capable of guaranteeing donor host integrity against malicious code.

In the case of systems that use Java, only those that have tasks implemented as applets (ATLAS, Charlotte, SuperWeb, IceT, JET, Javelin, Java Market, POPCORN, Bayanihan, CCOF) execute the code inside a sandbox. Ginger also executes its tasks inside a virtual machine. Alchemi uses the .NET virtual machine that allows the definition of sandboxes.

In ATLAS, IceT, XtremWeb, BOINC, Leiden and nuBOINC donors simply trust the developers of the tasks, believing that the code will not harm the computer.

2.2.3 User interaction

The way users interact with the available Distributed Computing Systems may also be relevant to the popularity of such systems. This section presents issues related with the user interaction: how work is organized, the possible user roles, the mechanisms to create jobs and tasks, and the incentives for donating cycles to the community.

2.2.3.1 Work Organization

Independently of the terminology used by each author, the work submitted to a Distributed Computing System follows a predetermined pattern, with a common set of entities and corresponding layers.

The work can be organized around the following entities, of increasing complexity:

- Task

- Job
- Project

Systems organizing work in different layers use more complex entities, not only to encase other simpler entities, but also to reduce the effort to create work.

Task Tasks are the smallest work unit in a cycle sharing system. Each is composed of the code to be executed and the data to be processed.

Although some systems allow tasks to spawn themselves in order to split their work among their children, in most systems tasks are simple entities: the code is usually single threaded, running on a single processor, and tasks do not interact with each other (neither for synchronization nor message passing). In these simple cases, tasks have similar code, but process different data.

When submitting tasks, users have to define what code will be executed on the donor and the data that will be processed. The methods to define the code and the data differ between systems, as will be seen later in this section.

Job Tasks executed to solve a common problem are grouped in jobs. These tasks, belonging to the same job, execute similar code, but have different input parameters.

A job must encase all the code necessary for its tasks' execution: initialization code, task launch, tasks' code, and result retrieval. Furthermore, when creating a job, the user must define the input data for each task.

When defining a job, the user must supply the code that will be executed by each task, and how the overall data will be split among different tasks. The creation of tasks can be automatically handled by the middleware, or programmed by the user.

Project Some jobs share the same code, only differing on the data sets processed. Without a higher level entity, every user would always have to submit the same processing code, whenever he wanted to submit a job.

A project can be seen as a job template, where a user only defines the processing code. Later, the project is instantiated to create a job.

In systems with the concept of project, when creating jobs, the user only has to supply the dataset to be processed and the tasks' input parameters. The job's code has been previously defined when creating the project.

2.2.3.2 User Roles

A regular user located at the edge of the Internet can have several roles in a Distributed Computing System:

- Donor
- Client Job creator
- Client Project creator

A user, with enough computer and network administration knowledge, can install any of the available systems, being capable of creating and submitting work to be executed. In this classification there is a distinction between the administrator of the computer hosting the work and the user submitting it. If it is necessary for a user to be the administrator (because of the inexistence of user level tools) to create jobs, it is not considered possible for a regular user to create them.

Some systems allow a single user to have several of the presented roles.

Donor A donor is the owner of a computer where tasks from others are executed. The donor must install the necessary software to execute tasks, and configure it. Before the execution of other users' work, the donor must select either the server where tasks will come from or select what kind of work he is willing to execute.

Client Project Creator If a system support projects, someone has to create them: develop the processing code, and register it in the system.

If there are tools for the easy creation of projects, then any client user can be a project creator. Later, the same user (or others) can create jobs that will be executed within the same project.

Other systems require administrators to create projects. In such cases client users are limited to being Job Creators.

Client Job Creator Job creators are those users that submit work to be executed. Although the simplest work unit is the task, clients merge their work into jobs (even with only one task). If the user is allowed to submit data and code, he acts as a Job creator.

Using the tools provided by the system, these users only submit the information about the job: tasks' code and input data. If the system supports projects they just need to select the project and prepare the data.

2.2.3.3 Job Creation Model

Job creation is always composed of two steps: definition of the processing code, and definition of each task's input parameters and data. In systems with projects, these two steps are independent.

The definition of processing code can be performed in three different ways. Users can take advantage of off-the-shelf applications or develop their own processing code:

- (COTS) Executable assignment
- Module development
- Application development

The way job code is defined or developed affects the way tasks are created. This relationship will be further described later in the analysis.

(COTS) Executable Assignment If the system allows external commodity-off-the-shelf (COTS) executable assignment, to create a job the user has only to provide that application, either uploading it to the server or copying it to a suitable directory. In this step the user should also provide information about the executable: type of input/output data or the format of the data.

In the development of the executable, normally no use of special API is necessary. The application should only follow simple interface requirements imposed by the underlying system. Although a regular application can be used, to take advantage of some services (e.g. checkpointing) it may be necessary to use a supplied API, or link with special libraries.

In systems accepting executable assignment in the job definition step, the selected application just contains the code executed by every task. This application, being generic, can not contain any task creation code, and is simply invoked on donor computers.

Module Development Some other systems require users to explicitly develop the processing code. In these systems, the task's code should be encased in a module (or class if using some object-oriented language).

The developed modules should comply with a pre-defined interface and can use a helper API in order to take advantage of available services.

The user is freed from developing task launching code, since this step is transparently handled by the systems.

Application Development Other systems require the explicit programming of the task launching step. Here, this step must be inside a complete application, that can also contain the data splitting among tasks. It is responsibility of the client user to develop any pre and post-processing steps.

A system that requires the complete application development must also supply the API for task launching and any other offered services. Using that, the user develops a complete application containing:

- environment initialization
- pre-processing code
- data partition code
- tasks' execution code (in a function or method)

- tasks' launch invocation
- result retrieval
- post-processing code

Although setting up a job this way is a complex feat, the set of problems solvable in such ways is vaster.

2.2.3.4 Task Creation Model

To create a job, it is necessary to define the various independent tasks that will compose it. Depending on the system, the development and definition of those tasks can be made in different ways:

- Data definition
- Data partitioning
- Code partitioning

Data Definition The easiest mode to define a task is by simply performing data definition. The user only states (declaratively, in a configuration file or with a user interface) what data each task is to process. These systems offer no auxiliary means to split the data, they just offer simple assignment mechanisms.

Data definition is used on systems where job creation relies on either executable assignment or module development. Then, the midleware just executes the executable (or module) on the donor computer, with the supplied input data.

Data Partitioning With data partitioning, a script, or even the master code, splits the data that will later be implicitly assigned to tasks. The system offers both the means to split the data and to add the partial data to a pool. This addition can be performed programatically (by means of a API) or declaratively (trough a user interface or configuration file). In this case, the underlying system is responsible for the creation and invocation of the various independent tasks, and to assign each task its arguments and input data.

Code Partition With data partition, as no explicit launch of tasks is performed, the user does not have the possibility to control task execution. On the other hand, when job creation resorts to code partition, the user must explicitly launch each task by using a specific API. The user must develop a complete application, as explained in the previous section, with all the necessary components.

Since each task must process different data, the main program must also perform data partition, and data assignment (when creating a task). With code partitioning the parallelism arises, not from independent data, but from the explicit creation of tasks.

2.2.3.5 Task Execution Model

During the execution of a job, there are different ways to start and execute the various tasks, leading to different organizations of these tasks:

- Bag-of-tasks
- Master-slave
- Recursive

Bag-of-tasks In a Bag-of-Tasks job all tasks are created simultaneously and execute the same code.

The main process only splits the data, creates the tasks, waits for their conclusion, and performs minimal processing of the results. Being the simplest parallel programming model, this reduces the programming effort to create such jobs. The user only has to develop the processing code, and rely on the middleware for launching each task and execute that code.

Further data processing (of the input and results) must be performed off-line with external tools.

Master-slave In a master-slave execution model, the main process creates the various tasks in a computational loop and blocks its own execution until the end of all of them. The tasks may be different (in terms of input data and code), started at different times, and have data dependencies between them.

In this kind of computation, where there is a main program from where all tasks are launched, the user can either create Bag-of-tasks or more complex workflows. The main loop creates tasks, waits for the results, that can be used as input of following tasks.

Recursive In a master-slave computation, only the main program can create tasks. Some problems, however, require that running tasks can themselves create and launch others. Naturally recursive problems fit in this model, as well as problems where tasks also generate new or intermediate data to be processed.

In a recursive execution, the main application creates the first level of tasks (as in the master-slave model). These running tasks can also create other tasks when needed.

The programmer must develop the whole application: the main program, data splitting, and the tasks. In this model, task creation is explicit, requiring the use of a supplied API.

2.2.3.6 Offered Services

Although the programmer must always develop the code that is to be executed on the remote hosts, the available systems can provide different support tools or services. For the definition and deployment of parallel jobs, the offered services can fit into:

- Launching Infrastructure
- Monitoring Infrastructure
- Programming API

Some sort of software layer (middleware) must always exist in order to coordinate all available resources, independently of the offered support services. The infrastructure referred to in this section does not deal with this concern, but solely with the launching and definition of jobs and tasks.

Launching Infrastructure For Bag-of-Tasks problems, the simplest launching mechanism is by means of a launch infrastructure. The user supplies the application or module corresponding to tasks, defines the data to be processed, and the middleware becomes responsible for launching the necessary tasks.

The use of a launching infrastructure fits well and is the obvious solution to: i) Bag-of-Tasks problems, ii) task creation based on data definition, and iii) job creation based on executable assignment and module development.

Even when the application development job creation model is used, a launching infrastructure is convenient. In these cases, the middleware configuration can be performed programatically (inside the main application) or declaratively (outside the application and through a user interface). In such cases, a launching infrastructure can be used to define the job application, and configure the middleware.

Monitoring Infrastructure Independently of the way jobs are created and tasks launched, a status monitoring infrastructure can be provided. The user can monitor the execution of the job and the status of the running tasks by means of a user interface. These monitoring infrastructures interact with the middleware to provide a global view of the system: available resources and status of the submitted work. In addiction, they can include the following: tasks not yet launched, tasks currently executing, terminated tasks, and results received (validated, not yet validated, or corrupted). Based on this information, the client can easily decide to abort executing jobs, or predict termination times.

Programming API In order to take advantage of some offered services, it is necessary to use some programming API, for instance, when programming the tasks, an API can be offered to provide checkpointing.

When the user needs to develop a complete application, the use of an API to define tasks is fundamental. These APIs are needed to register the data to be processed by each task or to explicitly create the tasks.

The existence of an API for code development is orthogonal to the existence of any kind of infrastructure.

Existing API are normally used to launch new tasks, control their termination, or provide checkpointing to the code.

2.2.3.7 Programming Languages

The programming languages used for the development of projects and tasks, not only limits the type of problems one can solve, but also affects how easily development is carried out.

In the different phases of the development and execution of tasks several kind of languages, can be used:

- Declarative
- Compiled
- Interpreted
- Graphical

Some of the presented systems used different languages for the development of projects and definition of tasks.

Problems needing more complex execution models (such as recursive, or Master-Slave) require the use of compiled or interpreted languages, since all others languages do not have enough expressiveness. On the other hand, the less expressive languages require less expertise and are easier to learn.

Declarative When using a declarative task creation paradigm, the user can only state the input data of each task, either using a supplied user interface or by creating a configuration file.

This is mostly applicable when the user only needs to make an executable assignment and data definition when creating jobs and tasks.

Compiled To develop the processing code, it is necessary to use an interpreted or compiled language.

The use of a compiled language allows full flexibility on the development of task code. The systems can even provide some APIs so that tasks can use some offered services.

Due to difficulties in making compiled code mobile, systems that use compiled languages require tasks to be complete applications (with the main function) instead of being simple modules. The middleware uploads one complete application to the donor computer and starts it there. This architectural characteristic limits theses languages to, mostly, Bag-of-Tasks problems.

In order to have tasks running on different architectures, it is required that the user compiles the developed code to those architectures.

Despite these difficulties, languages targeting high-performance computing can be used (Fortran, for instance) allowing tasks to execute at full speed (much faster than if using a interpreted language).

Interpreted The use of interpreted or JIT-ed languages overcomes some of the issues raised with compiled languages.

The development of tasks as modules is straightforward and the deployment of that code becomes more practical. Most interpreted or JIT-ed languages (Java, C#, python) allow introspection, mobility and dynamic code loading, easing the development of a Distributed Computing infrastructure. Even from a complete application it becomes easy to isolate tasks' code (usually a function or class), and upload and invoke it on remote donors.

The use of a high-level byte-code language, along with its execution environment, increases the portability of the code, and donor availability. The client only needs to develop one version of the code, and the donors are only required to have the language execution environment installed. All required libraries can be uploaded to the donor along with the task. The added ease of setting up a donor can also increase the number of users willing to donate cycles.

The only drawback of using an interpreted language is its possible lower execution speed, that can be overcome by the gains of having a large donor base and ease of development.

Graphical Graphical languages should be used in conjunction with a more traditional programming language (either interpreted or compiled). In a Distributed Computing system, graphical languages can be used on the definition of jobs, in the following aspects:

- Task code selection
- Data splitting
- Workflow creation

This may increase the number of users submitting work, making these systems more visible to the common computer user.

2.2.3.8 Project Selection

Some of the available systems can host several projects, so there is a need for donors to select the projects they want to donate cycles to. Independently of the mechanisms to select the projects, the selection process fits into one of these classes:

- Explicit
- Implicit
- Restricted (topic-based)

Explicit The user is obliged to contact the server and make an explicit selection of the projects or jobs he will donate cycles to.

Even on systems where a server just hosts a single well known project, the user, when connecting, is explicitly selecting a project.

This explicit selection can be in the form of choosing a URL (of the server or the project/job) or by means of a user interface. The explicit selection of the project requires the identity of the client creating the job, or the focus of the work, to be disclosed to the donors.

Implicit With an implicit job selection scheme, the donor does not have the means or possibility to select the jobs that will be executed. The donor just connects to a server hosting various jobs, or to the peer-to-peer network, and the selection of the work is made transparently by the system.

Restricted (topic-based) In a intermediate level, users may be able to choose the subjects of the work they are willing to execute. In this case, the donor defines interest topics or a set of keywords allowing the servers (or the peer-to-peer network) to select the work to deliver to each donor.

This type of work selection does not require the donor to know the identity of the work creator, nor the precise purpose of the work, in order to match tasks with donors.

2.2.3.9 Donation Incentives

To gather donors, each system must give back some reward for the donated time. Available systems, after the completion of task assigned the donor, may reward the donor with one of the following:

- User ranking
- Processing time
- Currency

In any of the presented incentives, for the complete execution of a task some reward is awarded. This reward can be just a point/credit, the right to use another CPU, or a virtual monetary value (currency) to use in other services.

In either of the three cases, the reward should take into account the processing power employed in the execution of each task. In these scenarios, the processing time is not a good measure, as a slow machine takes longer to process a task.

User Ranking The easiest incentive to implement is user rankings. The ordering of donors depends on the quantity of donated resources to the community. When the donor correctly finishes a task the systems assigns points to it. To add a sense of accomplishment and boost the competitiveness between users, a ranking of the more meritorious users is publicly posted.

Processing Time On the other hand, after successful completion of a task, the donor can be rewarded with the right to use available resources to execute his tasks (by means of processing time).

In this case, when a donor becomes a client, the rewarded time can be used to his benefit. Later, when scheduling tasks to be executed, the system should be able to use the rewarded execution time in a fair manner.

Currency In a similar way, a virtual currency can be awarded after the completion of a task. In systems with a market-oriented scheduling policy, the collected values can be used to buy the processing time.

The awarded value can be used to bid for the latter execution of tasks on remote computers.

2.2.3.10 Analysis

The way the various systems implement and handle the user interaction issues is presented in two distinct tables: Table 2.3 (specifying the various possible user roles) and Table 2.4 (presenting programming and usage alternatives).

	Work organization	User Roles	Job creation Model	Task creat. Model	Task exec. Model
ATLAS	Job Task	Donor Job creator	Application dev.	Code part.	Recursive
ParaWeb	Job Task	Job creator	Application dev.	Code part.	Bag-of-Tasks
Charlotte KnittingFactory	Job Task	Donor Job creator	Application dev.	Code part.	Master-slave
SuperWeb	Task	Donor Task creator	Module dev.	Data def.	Bag-of-Tasks
Ice T	Job Task	Donor Job creator	Application dev.	Code part.	Master-slave
JET	Job Task	Donor Job creator	Module dev.	Data part.	Bag-of-Tasks
Javelin	Job Task	Donor Project creator	Application dev.	Code part.	Recursive

Table 2.3 (Continues on next page)

	Work organization	User Roles	Job creation Model	Task creat. Model	Task exec. Model
Java Market	Task	Donor Task creator	-	Data def.	Bag-of-Tasks
popcorn	Job Task	Donor Job creator	Application dev.	Code part.	Bag-of-Tasks
Bayanihan	Project Job Task	Donor Project creator Job creator	Application dev.	Data part.	Master-slave
MoBiDiCK	Project Job Task	Donor	Application dev.	Data part.	Bag-of-Tasks
XtremWeb	Project Job Task	Donor	Application dev.	Data part.	Bag-of-Tasks
JXTA-JNGI	Project Job Task	Job creator Donor	Application dev.	Code part.	Master-slave
P^3	Job Task	Donor Job creator	Module dev.	Data part.	Bag-of-Tasks
CX	Job Task	Donor Job creator	Application dev.	Code part.	Recursive
G2-P2	Job Task	Donor Job creator	Application dev.	Code part.	Master-slave
CCOF	-	-	-	-	Bag-of-Tasks
Personal Power Plant	Job Task	Donor Job creator	Application dev.	Code part.	Master-slave
CompuP2P	Job Task	Donor Job creator	Executable def.	Data part.	Bag-of-Tasks
Alchemi	Job Task	Donor Job creator	Application dev. Executable def.	Code part. Data def.	Master-slave Bag-of-Tasks
YA	-	-	-		
BOINC	Project Job Task	Donor	Application dev. Executable def.	Data part.	Bag-of-Tasks
Leiden	Task	Donor Task creator	Application dev.	Data def.	Bag-of-Tasks
Ginger	Job Task	Donor Job creator	Executable def.	Data def.	Bag-of-Tasks
nuBOINC	Job Task	Donor Job creator	Executable def.	Data part.	Bag-of-Tasks

Table 2.3: Internet Distributed Computing systems user roles

Work Organization Most of the evaluated systems divide the work in both jobs and tasks. In these systems, the work submission unit is a job composed of tasks with similar code but different input data.

SuperWeb, Java Market, and Leiden do not have the job concept. User can only submit individual tasks that have no relation between them.

On the other hand, Bayanihan, MoBiDiCK, XtremWeb, JXTA-JNGI and BOINC have the project entity. Before any job creation, it is necessary to define a project. The responsibility for the project creation differs in these systems, as will presented later.

User Roles As expected, all systems allow users scattered on the Internet to donate their cycles on a simple, and some times anonymous, way. What distinguishes most systems is the ability of ordinary users (those that can also donate cycles) to also create work requests (tasks, jobs) and make them available to be executed.

On BOINC, XtremeWeb, and MoBiDiCK, only the administrator can create work. On all other systems, the creation of jobs (and tasks) is straightforward without requiring any special privileges.

Job Creation Model and Task Creation Model The way jobs and tasks are created is tightly linked in Distributed Computing systems.

In three systems, the user that has work to be done has to explicitly assign the data to each task: SuperWeb, Java Market, and Leiden. These are also the systems that do not have the concept of job, here every task is truly independent of the others. In SuperWeb, the programmer develops a module whose code will be executed by each task, while in Leiden, the tasks correspond to a independent application that is executed on the remote host.

It is possible to observe that the other systems (JET and P^3) that require the sole development of a module (the code of the tasks), only require data partitioning when creating tasks. Although the user is required to develop the code to split the data, no explicit creation of tasks is necessary.

In MoBiDiCK, XtremWeb, CompuP2P, Alchemi, and BOINC each task is encased in a complete application that executes on the remote hosts. This application must be specially developed.

In Ginger and nuBOINC the user also assigns a executable to the job, but does not have to implement it. These are commodity-off-the-self applications that are widely available (or installed) on the donor computers.

As in these systems, each task corresponds to the execution of a complete application, the creation of tasks is simply made by assigning to each one its input data. In these systems, with the exception of Alchemi, the user must program the data partition. In Alchemi the user defines each task input data by means of XML file. In BOINC and Alchemi it is also possible to use a pre-existing application as task code (close to the concept present on nuBOINC and Ginger).

In Bayanihan, the user develops the main application that is responsible for data partition. In this system, tasks are not created explicitly. The data to be processed is programatically stored in a pool, by means of a supplied API. The system then picks data from that pool and assigns it to the tasks, without programmer intervention. In Ginger the data is transparently split, before tasks are initiated. Ginger includes a data partition engine that automatically created task's input with a XML data description.

In all other systems, the developed application must contain, as a function or class, each task's code. Data partitioning and task creation are performed explicitly and internally in the developed application.

Task Execution Model With the exception of Bayanihan, in all systems whose tasks are created by means of data partitioning, the parallel jobs fall in the Bag-of-Tasks category. In these systems, it is impossible to have some sort of workflow (where results are used as input of other tasks) and there can only be some minimal pre-processing of the data and post-processing of the results.

Although in Bayanihan tasks are created by data partition, the programmer can both control and interact with the running tasks and chain them to get complex data computations and workflows. Bayanihan main application, after partitioning data and storing it in the data pool, waits for the availability of the results. This application can assign the results of a previous step to new tasks, allowing complex workflows.

In the two systems that require the explicit declarative definition of data (Java Market and Leiden), the solvable problems obviously fall in the Bag-of-Tasks category.

All other systems can be used to solve master-slave problems, and consequently also Bags-of-Tasks.

ATLAS, Javelin, and CX also allow the execution of recursive problems, by allowing tasks to spawn themselves to create new tasks.

	Offered Services	Programming Language	Project Selection	Donation Incentives
ATLAS	Programming API	Interpreted	Implicit	-
ParaWeb	Programming API	Interpreted	Implicit	-
Charlotte KnittingFactory	Programming API	Interpreted	Explicit	-
SuperWeb	Infrastructure	Interpreted	Implicit	Processing time
Ice T	Programming API	Interpreted	-	-
JET	Monitor infrastructure Programing API	Interpreted	-	-
Javelin	Launch infrastructure Programming API	Interpreted	Implicit	-
Java Market	Infrastructure	Interpreted	-	Currency
popcorn	Programming API	Interpreted	Explicit	Currency
Bayanihan	Monitor infrastructure	Interpreted	Implicit	-
MoBiDiCK	Launch infrastructure Programming API	Compiled	Explicit	-
XtremWeb	Infrastructure	Interpreted	Implicit	-
JXTA-JNGI	Programming API	Interpreted	Implicit	-
P^3	Programming API	Interpreted	-	-
CX	Programming API	Interpreted	Implicit	-
G2-P2	Programming API	Interpreted	Implicit	-
CCOF	-	-	-	-
Personal Power Plant	Launch infrastructure Programming API	Interpreted	Explicit	-

Table 2.4 (Continues on next page)

	Offered Services	Programming Language	Project Selection	Donation incentives
CompuP2P	Infrastructure	Interpreted Declarative	Explicit	Currency
Alchemi	Infrastructure Programming API	Interpreted Declarative	Implicit	-
YA	-	-	Implicit	
BOINC	Monitor infrastructure Programming API	Compiled	Explicit	User ranking
Leiden	Infrastructure	Declarative	Explicit	User ranking
Ginger	Launch Infrastructure	Declarative	Explicit	Currency User ranking
nuBOINC	Infrastructure	Declarative	Explicit	User ranking

Table 2.4: Internet Distributed Computing systems programming and usage

Offered Services Some of the analyzed systems only provide a programming API for the development and launching of applications. Programmers write one application, launch it and wait for the resulting tasks to finish. In such systems, there is no way for the owner of the work to control the various tasks: i) terminate the tasks, ii) check for their status, or iii) observe the intermediate results.

SuperWeb, Java Market, XtremeWeb, CompuP2P, Alchemi, and Leiden, on the other hand, offer a full infrastructure for the deployment and launching of jobs and for observing tasks' execution status.

Other systems also have some sort of infrastructure either for launching jobs (Javelin, MoBiDiCK and Personal Power Plant, Ginger, Leiden and nuBOINC) or for monitoring tasks (JET, Bayanihan, BOINC, Leiden and nuBOINC).

Of the available systems, some require the programmer to use a programming API to develop the application, and use the infrastructure to launch the jobs or monitor the task. Examples of such systems are JET and BOINC (with a programming API and monitoring infrastructure), and Javelin, MoBiDiCK, and Personal Power Plant (with a programming API and launching infrastructure).

Programming Language The large majority of the available systems use interpreted languages for the development and execution of tasks, namely Java. Mo-

BiDiCK and BOINC require the development of a compiled application.

In CompuP2P, Alchemi, Leiden, Ginger, and nuBOINC, the creation of tasks resorts to the declaration of their arguments. In Alchemi, the client defines them in a XML file, while in the other systems (CompuP2P, Leiden, Ginger and nuBOINC) the user uses a supplied user interface.

Project Selection In most systems the user has no way to select which projects he will donate cycles to. In these systems, it is the server that selects what tasks to send to the clients, and, since each server hosts several projects, the donor does not know to which job or project the task belongs to.

In systems where a server only hosts one project, the project selection is explicit. The donor contacts a pre-determined server, knowing exactly what is the purpose of the tasks being run.

In BOINC, the donor contacts one particular server, but can select the projects that he wants to donate cycle to. From the various hosted projects, the server selects tasks from the ones the user has previously registered.

In Ginger and nuBOINC, the selection is also explicit as the donor selects what off-the-self application can be used to execute remote tasks.

Donation Incentives Most of the studied systems do not have any sort of reward to users donating cycles. In the market-oriented systems (Java Market, POPCORN and CompuP2P), the reward for executing correctly a task is a currency value that can later be used to get work done remotely. In SuperWeb, instead of using a generic currency, the donor is rewarded a certain amount of processing time to be spent later. In Ginger, the user is rewarded a generic currency, that can be later used to exchange for processing time, but his reputation (user ranking in the table) is also modified. The overall user ranking is used when assigning work to donors.

The donors in BOINC, Leiden, and nuBOINC are only rewarded execution points, that serve no other purpose than to rank the donors. These points cannot be exchanged for work and are only used for sorting the users and the groups they belong to.

2.3 Related Work

Although there is some work trying to systematize the characteristics of distributed computing infrastructures [CKBH06, CKB^{+}07, CBK^{+}08], the main focus has been on network and software architectural decisions, not including the various security aspects and user level interaction models. Furthermore, these surveys also describe systems not usable in a public infrastructure (with admission of donors or clients), while this chapter has focused the Internet based free access systems. The range of systems presented here is much broader (presenting more systems, and from a wider temporal range) than those presented in existing documents.

Other documents also describe some of the characteristics dealt with in this survey. Marcelo Lobosco et al. [LdAL02] present a series of systems and libraries that allow the use of Java in High Performance Computing scenarios. The document is mostly focused on the description of programming paradigms available for the development of parallel applications in Java. The described systems are, however, mostly targeted to cluster environments.

Koen Vanthournout et al. [VDB05] describe currently available resource discovery mechanisms. This work describes the different available peer-to-peer network topologies and service discovery frameworks. While the presented mechanisms can be targeted to distributed computing systems, the presented application examples are mostly restricted to the file sharing category.

2.4 Evaluation

The previous section, not only presented a new taxonomy for Distributed Computing systems, but also characterized them in the light of it. A more profound study can be made to find out what are the characteristics that have a greater impact on user adhesion to systems and the donation of cycles to other users. First, the characteristics that seem more relevant are presented and then evaluated on how are implemented by the more successful systems.

A system that optimizes (by using the most efficient technique) these characteristics should be the most capable of gathering donors. These characteristics will fall under the previously presented classes:

- **Security** - Execution host security
- **Architecture** - Network topology and scheduling policies
- **User interaction** - User roles, project selection and donation incentives

One of the fundamental issues that may prevent users from donating cycles to others is the security of their machines. There should be some sort of guarantee that the downloaded code will not harm the donor's machine. The use of some sort of sandbox (either application or system-level) is the best approach to guarantee this. By isolating the downloaded code, even if it is malicious, the host computer cannot be compromised.

Of the proposed Network Topologies, peer-to-peer is the one that more easily allows users to adhere to a system, continue using it, and donate cycles to others. In a peer-to-peer infrastructure, no complex configurations are needed. The peer configuration (usually stating a network access point) is much simpler than configuring a client to connect do different servers. After this initial configuration the donor automatically is in position to donate cycles to any client, making resources (in this case, cycles) more available than on client-server systems.

Besides simplicity of configuration, current peer-to-peer systems (mostly on file sharing) offer other highly appreciated characteristics. In these systems, anonymity is usually preserved, but still allowing the aggregation of users around common interests. Furthermore, the inexistence of central servers (even if more than one) would increase the reliability and scalability of the system. The preservation of these characteristics should also increase user adhesion.

Another factor that can increase user participation (by donating cycles), is the possibility for donors to take advantage of the system by becoming clients. If users are able to execute their work (having the role of job creators) they are more willing to donate cycles later. Furthermore, there should be some fairness on the access to the available cycles. The selection of tasks to be executed must take into account the amount of work the task owner has donated to others. This issue can be handled by the scheduling algorithm when selecting the tasks to be executed.

The use of a market-oriented approach can also introduce a level of fairness. Users receive a payment for the execution of work, and use that amount to pay for the execution of their own tasks.

Necessary for fair task execution are the rewards given for the execution of work. If, when scheduling tasks, user sorting or market mechanisms are used, it is necessary to use some differentiating values (either points or some currency). These reward points or currency are given after each task's completion and can later be used to sort the tasks or for bidding for resources.

If users know the available projects, and what problems are being solved by the tasks, they may be more inclined to donate cycles. By knowing what the tasks do, users can create some sort of empathy, and donate cycles to that particular project. To guarantee this, it is necessary that job selection is explicit.

BOINC is undoubtedly the only successful (in terms or user base) Distributed Computing system. Most of the existing public computing infrastructures rely of this infrastructure [Sch07] and with more than two million [BOI11] aggregated donors.

Because of these facts it is relevant to study and know how BOINC implements the more important characteristics. The following presents how BOINC implements the most relevant characteristics:

- **Execution host security** - User trust
- **Network Topology** - Point-to-point
- **Scheduling Policies** - Eager / Resource aware

- **User Roles** - Donor
- **Project Selection** - Explicit
- **Donation Incentives** - User ranking

The first two issues, those more related with the system architecture, present sub-optimal solutions. There is no automatic mechanism to guarantee the security of the donor, and the network topology may not efficiently handle a large amount of users.

In order to host a single project BOINC uses a single server. The experiments carried out [AKW05] prove that a single computer can handle more requests than those possible in a real usage environment. The absence of any security guarantee mechanism is much more difficult to justify. Users just trust the code they are downloading from a server, because they know who developed it and what is the purpose of the work being executed. This is only possible because project selection is explicit.

In BOINC, users donating cycles cannot submit their own work. Their sole function is to execute tasks created by the project managers. With such usage, the scheduling policies are not relevant to the satisfaction of the users. To promote cycle donation, BOINC employs user ranking: one of the rewards is to be seen as the better donor, the one executing more tasks.

The two fundamental decisions for the success of BOINC as an infrastructure for Distributed Computing are: the explicit selection of projects and user rankings.

The explicit selection of projects allows user to donate cycles to those projects they think are more useful, thus leveraging the altruistic feelings users may have. As most projects have results with great impact to society, e.g. medicine, it is easy for them to gather donors and become successful.

On the other hand, the ranking of users based on the work executed promotes competitive instincts. Although donating cycles to useful causes, the ranking (of users and teams) increases the computing power a user is willing to donate to a cause.

These two factors greatly overcome the deficiencies of BOINC (security, and user roles) and make BOINC arguably the most successful Distributed Computing system.

2.5 Conclusions

This chapter presented a taxonomy to evaluate and describe the various characteristics of Distributed Computing platforms. This taxonomy is split in three different classes of aspects: i) Architecture, ii) Security and reliability, and iii) User interaction.

This taxonomy was also applied to Distributed Computing systems developed up to now, presenting the design decisions taken by each system.

Although all presented characteristics are relevant to the global and local performance of the system, some of those seem fundamental for the system success in gathering users: i) Execution host security, ii) Network topology, iii) Scheduling policies, iv) User roles, v) Project Selection, and vi) Donation incentives. For each characteristic, the optimal and most efficient solution are presented: those that would make a system successful and widely used.

Finally, these optimal solutions were compared with those implemented in BOINC and conclude that BOINC does not implemented most of them. The great success of BOINC does not come from the use of optimal solutions but from the use of two natural human reactions: empathy with the problem being solved, and competitiveness among users.

3 Graphical Bag-of-Tasks Definition

Of the previously presented task creation methods (Chapter 2), the easiest one to use is the declarative definition of tasks. This method only requires the user to assign the code to be executed and to define the data to be processed by each task. Although some graphical tools exist to do that, they are not suitable to the tool users or hobbyists (defined in Section 1.1) that require the processing of several files using an available application. The existence of suitable tools, not only reduces the user burden, but can also leverage the use of available Internet based Distributed Computing systems, and allow the efficient use of personal computers and clusters.

This chapter presents SPADE, a system that allows the easy definition of tasks, simply by declaratively stating the processing code to be used, and how the data is to be split among tasks. The developed system will resort to an user interface, where the user assigns each task a piece of the data to be processed. In order to define the requirements a task definition method must comply to, it is necessary to evaluate the user needs in terms of possible data splitting ways.

In order to leverage existing personal clusters, the work here presented is also composed of a simple LAN based efficient task scheduling infrastructure.

3.1 Introduction

Even the less knowledgeable users have computational jobs that can be executed in pools of computers. Most of these jobs are lengthy, and composed of independent similar tasks, thus fitting into the Bag-of-Task model and being easy to parallelize. The typical jobs, usually require the processing of sets of files, or the execution of a program several times, each time with with different parameters.

Furthermore these users can not use existing tools, either to take advantage of existing resources, or to parallelize their work. These tools were designed for users with some system administration expertise and programming knowledge.

So, in order to take advantage of any of the existing resources presented in Section 1.2, it is necessary to develop mechanisms to easily allow the creation of jobs composed of multiple tasks and tools to deploy them on pools of computers.

Due to the target user group, and in order to increase usability, the most efficient mechanism to create these jobs is declaratively and by means of suitable user interfaces.

The development of a user interface for job submission targeted at users without large computing experience, requires a previous study of the possible problems to be solved. These fit, obviously, into the Bag-of-Tasks problems, where the differences between each task fall into one of the following:

- each task processes a different input file;
- each tasks receives different values as arguments;
- each task processes a different part of a (uni or bidimensional) set.

Most of the usual jobs will fit into one of these classes. The batch processing of photos or data file analysis requires each task to get as input a file with a different name. The user should be able to easily state how many tasks to create and what file is to be processed by each task.

If what distinguishes tasks is a numerical scalar argument or a set of such arguments, the user should define the tasks arguments based on the task identifier, either directly or by means of a simple transformation function. Jobs such as image rendering, that can be split assigning to each task, a part of a bidimensional area, are also usual. In this case, the user should only be required to state the limits of the set and how many tasks to create.

In parallel to this need to execute lengthy repetitive tasks by common users, there is also a somehow high availability of wasted resources, since most of these users own multiple computers (although with different processing power) or computers with multiple cores (equivalent to multiprocessors). Although these users have enough resources at their reach, task scheduling infrastructures (designed for enterprise clusters), either require complex configuration or do not have suitable job creation mechanisms. The installation of available task scheduling tools requires system administration knowledge, and administrative right along with the edition of multiple configuration files, and in order to launch parallel jobs, the user is required to edit some scripts.

In SPADE, jobs are composed of simple tasks that will be executed on remote computers. The user states what files should be processed by each task, what application will be launched and defines the application arguments. By using SPADE any user can, for instance, accelerate a batch image manipulation by using otherwise idle remote computers, while releasing the mobile device for other tasks.

In order to make SPADE usable by a wide set of computer users, three ideas were carried out: i) the execution code is a commodity piece of software already installed on the remote computers (such as image processing applications), ii) the definition of the data sets to be remotely processed is done in a simple and intuitive way, and iii) easy maintenance and use of LAN connected cluster.

SPADE is a tool that allows the remote execution of common computing tasks on several remote computers, otherwise executed sequentially on a single computer. Users should be able to speed the execution of tasks such as image manipulation by taking advantage of idle computers. SPADE is also composed of a low-level layer responsible for the scheduling and deployment of tasks on a personal cluster.

To accomplish this, techniques similar to the ones used in Grid and cluster computing are used and extended, while taking into account the particularities of home users, and those with low computing knowledge.

On the area of scientific computing, the offloading of computations to more powerful remote computers is a good and proven solution to speed lengthy jobs. Users either develop their own parallel application or use specific applications previously developed and installed. To enable access to these systems, cluster and Grid technologies have emerged.

On a desktop environment, a user owning several computers can connect them using a high speed switch, and install any of the available job scheduling software. As these systems were developed to specific target populations (system administrators and programmers), these are not suitable for different user classes. Even systems targeted at personal home clusters (such a XGrid) required a high proficiency.

Remote execution of generic software packages may seem infeasible and without practical use, but there are several applications whose data can be easily partitioned and several instances of the application executed in parallel. The scope of commodity software that can be executed in a more powerful remote environment (single computer or cluster) ranges from ray tracing systems to video or image processing, or even data processing packages target at more specialized populations.

For instance, batch image manipulations using available software packages can be executed concurrently on multiple computers, as long as each computer has the same software installed. The user defines the arguments to apply to each image conversion, SPADE uploads the images to a remote computer and invokes the installed application there, conveying the arguments defined by the user. The resulting files are downloaded after the conversion is complete.

In the remaining of this chapter, SPADE, a system that facilitates the execution of commodity software on remote idle computers in order to jobs completion, is presented. These jobs should be executed by commodity applications, invoked from a command line. They can be composed by multiple tasks whose parameters differ on only numerical values or input file. The owner of the personal cluster registers their applications (such as R, or POV-Ray) on the SPADE system, so that later, it can be transparently executed. When there is a job to be executed on remote computers, the user supplies SPADE with the different input files and the parameters that should be provided to each task.

SPADE is responsible for uploading the necessary input files to the remote hosts, executing the selected application and downloading the result file. The user only needs to supply a generic command line that is parameterized to each task. Such command lines can differ on the input file or on a numerical value.

The definition of usable applications and jobs, and how they are possibly divided, is made using a simple user interface, allowing any user with minimum computer knowledge to do it. The selection of the available hosts, where remote tasks are executed, is performed in a easy transparent way.

The next section presents currently available systems that allow the execution of jobs composed of independently tasks (Bag-of-Tasks) in pools (or clusters) of computers, describing their characteristics and limitations to the target population. In section 3.3 the requirements a system such a SPADE must comply to are enumerated. SPADE architecture, its implementation and execution are presented in Section 3.4, while in the last ones, SPADE is evaluated and conclusions presented.

3.2 Related Work

In the area of cluster computing, some developments have been made on middleware to allow easy deployment of applications over several computers. One of those tools is XGrid [App05, App07]. XGrid allows the use of several computers as a cluster, providing a tool to launch and distribute parallel computations. The main characteristic of XGrid is its integration with the Mac OS X operating system, making it easier to create a cluster of computers. Even though XGrid allows an easy creation of clusters there is no easy to use user interface to create and launch jobs. Users are required to use command line tools to launch individual tasks or define configuration files that define the arguments of each task. On the other hand Apple supplies a set of API [App07] that allows the development of applications that programmatically interact with the XGrid, in order to create tasks and retrieve results.

On a Grid, schedulers like Condor [LLM88] or GLOBUS [Fos05] allow launching of lengthy jobs on remote computers, but users have to write a launching definition file, while the middleware is responsible for selecting the best available hosts to execute the tasks. Furthermore, in order to use remote infrastructures users are required to have the necessary authorizations and credentials.

Grid infrastructures have strong authentication mechanisms that are required on a public conventional installation, having users with different requirements and roles. In a simple environment, where all computers are owned and administered by the same user, and are located on the same LAN, such a heavy solution is over demanding. Implementing a simpler authentication method instead, security would still be guaranteed with limited overhead.

When dealing with clusters of computers, besides Condor, the most used task creation mechanisms are the ones provided by message passing programming libraries, such as MPI [Mes94] or PVM [GBD$^+$94]. The use of such mechanisms requires users to develop their own tasks, or write a task launching application, which is out of reach for the target population.

Most existing solutions for cluster management and task launching require either programming or system administration knowledge, that most of the target users of this work do not have.

3.2.1 Task launching mechanisms

Although existing systems are not suitable, some complementary tools have been developed to tackle these systems' limitations. It is relevant not only to study such task launching tools, but also to evaluate more complex task launching mechanisms, in order to extract some requirements for a fully usable job definition user interface.

As stated earlier, in order to use XGrid or Condor, it is necessary to invoke command line utilities responsible for launching tasks. Part of the XGrid command line utility is as follows (Listing 3.1):

```
xgrid -job run   [-gid grid-identifier] [-si stdin] [-in indir]
                 [-so stdout] [-se stderr] [-out outdir] [-email email-address]
                 [-art art-path -artid art-identifier] [-artequal art-value]
                 [-artmin art-value] [-artmax art-value] cmd [arg1 [...]]
% xgrid -job batch [-gid grid-identifier] xml-batch-file
```

Listing 3.1: XGrid command line utility manual (fragment)

With this application, a user is capable of invoking the command **cmd** on a remote computer with the suitable arguments (**arg1 [...]**). It is possible to define different arguments for each task, as well as assign different input or output files to each task. To accomplish that, the user must parametrize the arguments **-so** and **-si**. After each argument the user must supply a file name. In the case of **-si** the contents of supplied file will be used as keyboard input (stdin) of the remote tasks. The output written to *stdout* by the task, will be redirected to the file supplied after **-so**.

Although it is possible to supply different input files to each task, the executed application must read all input from the *stdin*. This is limitative as applications that read from files defined in the command line, more so if from two or more files, are impossible to be used with XGrid.

The invocation of the previous command creates a single task. In order to create multiple tasks, with different arguments, it is necessary to write a shell script that invokes multiple times this application.

In Condor, task's parameters are defined in configuration files. Examples of such files are presented in Listings 3.2 and 3.3.

```
1 executable = foo
2 input      = test.data
3 output     = loop.out
4 error      = loop.error
5 Log        = loop.log
6
7 Initialdir = run_1
8 Queue
9
10 Initialdir = run_2
11 Queue
```

Listing 3.2: Condor Configuration files

```
executable = foo
error      = err.$(Process)
Input      = in.$(Process)
Output     = out.$(Process)
Log        = foo.log
arguments  = $(Process)

Queue 150
```

Listing 3.3: Condor Configuration files

In order to submit such jobs, the user invokes the command **condor_submit** with the configuration file as argument. In the first example two tasks are created, that will run the **foo** application. One of the tasks has its input and output files in the directory **run_1** and the other tasks will have those files in the **run_2** directory. In a similar way as with XGrid, Condor replaces the *stdin* of the task with the file test.dat (stated in the line 2), and will redirect the **stdout** to the file **loop.out**.

The second example (Listing 3.3) will create 150 tasks, all executing the same application (**foo**), but with different arguments and input/output files. The **condor_submit** application, will replace the placeholder **$(Process)** with the actual process identifier.

In these examples it is impossible to use more than one file as task input, and it is required that the task code reads it from the *stdin*. In order to overcome this limitation, it is required that there exists a networked file system serving and providing the same directory structure to all cluster nodes. In a institutional cluster the existence of a Network File System is fundamental and easily implemented, but on a home cluster the same does not happen.

In order to ease the submission of jobs to cluster and Grid infrastructures, several graphical user interfaces were developed.

Nimrod [ASGH95] is composed of a set of tools to allow the the execution of numerical simulations on multiple computers. It is composed of a configurable user interface for jobs creation and a low-level layer responsible for task deployments. Although it offers an easy to use interface for job submission, it is not generic: previously to its use, it is necessary to develop a TCL/Tk script responsible for the creation of the user interface for a specific application. From this moment on, users are presented with an interface where they can parameterize their simulations using check boxes or sliders. The user can also define their own experiments writing a plan [NIM00] of the experiment. This plan file includes a parameter part, where the user states what parameters exist, their range, and how they vary between each task.

Ganga [EKJ$^+$05, MBE$^+$09] is a Python application that allows the definition of the tasks to be executed on remote clusters. Although Ganga was primarily developed to allow the data analysis within the ATLAS [A$^+$08] and LHCb [C$^+$08] experiments in High Energy Physics, its modular architecture allows its integration with other simulation and execution environments. This system allows the development of modules to interact with different back-end execution environments. Besides the programmatically definition of jobs, it is also possible to use a Graphical User Interface to parametrize jobs. These user interfaces are also modular, and specific to each simulation experiment. When using Ganga, the user is only allowed to either define a single lengthy task (processing application and parameters), or to define all tasks that compose the jobs. When defining several tasks, users must supply different parameters to each one, since there is no way to define a generic set of arguments and parameterize them for each task.

Although there are tools providing easy means to create parametric execution of jobs, none is generic, all requiring previous programming of the GUI.

Distributed Computing systems (presented in Section 2) were developed to take advantage of Internet scattered resources, and, in part, to allow those users to take advantage of the available remote computers to execute their tasks. The last assumptions may lead to the conclusion that these systems provide easy to use job definition mechanisms.

However, of the presented systems, only three offer the possibility to declaratively define jobs: CompuP2P, Alchemy and Leiden. All other systems require the client user (the one submitting work) to program, both the processing and tasks launching code.

Leiden [Lei, Som07] is the only system to offer a graphical user interface, for the launching of jobs, and retrieval of the results. As each job is only composed of a task, this submitting interface does note provide any data splitting mechanism: users simply supply a configuration file, that will be processed.

CompuP2P [Sek05] and Alchemy [LBRV05a] allow the creation of jobs in a declarative manner, but using a configuration file. CompuP2P configuration files includes for each task a XML element. This element describes the input and output files, along with the executable arguments. In Alchemy, the user must write explicitly the copy commands necessary to guarantee that the required files exist at the donor, but in order to define the number of tasks and their arguments, it employs a strategy similar to Nimrod. The user states what are the tasks' parameters, their limits and how they are split among tasks.

In face of available systems for job creation and deployment on remote hosts, presently, users with low computing and programming knowledge have no means to efficiently user their LAN connected computer to execute parametrized jobs. Proper installation of the execution environment is difficult and requires high levels of expertise, added to the fact that it is difficult for them to define the jobs to execute.

3.3 Requirements

As a tool to offload computations to remote computers, SPADE is most useful to those who have lengthy tasks to perform: lengthy single task jobs, repetitive data analysis, or long processing of large batches of files. Users with such requirements range from scientists who need to process batches of data, to the hobbyist who needs to create thumbnails for his collection of photos, or to designers who need to generate a detailed image using ray tracing tools. All these users have in common the fact that they are literate in some sort of tool (statistical package, image manipulation tool or image generator) but are not proficient in programming parallel or distributed applications or even system administration.

The problems to be accelerated, by executing them on remote machines, should have data easily partitionable, to which does not need extra processing and transformation: for instance, the scientist and the hobbyist have their data or images in different files, while the artist knows how to tell the ray tracer to generate only a small sub-set of the picture, or to generate a few frames of an animation.

With minimal effort, users should be able to deploy the execution infrastructure over their LAN connected computer. There should be neither complex configuration files to edit nor complementary services (such as distributed file systems) to install and configure.

3.3.1 Applications

One of the most relevant factors for the success of a system like SPADE is the amount of applications that can be executed on it, and the number of users that can use them. These applications are typically used to process batches of data and exist in distinct areas: statistical analysis, simulation, image processing, image or video generation. In order to allow an easy integration with a execution environment, it is required that such applications can be configured using command line arguments.

In the area of statistical analysis, existing systems resort to scripts written in a proprietary language, that, either process different data sets (located in multiple files), or process the same data with different parameters. One example of such systems is R [R F11]. It is composed of a execution environment for statistical computing and graphics generation, executing scripts written in the R language.

SPADE is useful in situations where a user has several scripts to execute in the R environment or has a script that must evaluate several data sets. In the R command line interface, a user can define the input and output files as well as other R variables, allowing the easy interaction of the SPADE system with the R program.

ImageMagick [Ima11] packages is a set of image manipulation tools that run with the same interface (command line arguments) on most desktop operating systems. When a user wants to apply the same transformation (for instance, color correction, or border generation) to a batch of images, one of the ImageMagick applications would have to be executed for each file. A non-expert user would sequentially process every image while SPADE would execute each image transformation on different hosts in parallel.

POV-Ray [Per08] is a ray tracing image generator that processes text files describing 3D scenes and generates the corresponding image. POV-Ray may also generate a sequence of images that can be used to create a movie. It runs on a variety of operating systems delivering the same results on every one and has shell commands for the definition of the viewport of an image or the timestep of an animation to be rendered. Image rendering may take advantage of a system such as SPADE, by distributing the rendering of small parts of the final image on different computers, while a movie generation could also be accelerated by rendering each frame in parallel. Obviously, other tools such as audio or video processing tools can also be safely used with SPADE.

Although Commodity-Off-The-Shelf applications are good candidates to be executed in personal clusters, also sequential custom built applications can be executed in a personal clusters as well. Any application developed in an interpreted language (JAVA or Python, for instance) can take advantage of SPADE. In this case the application to be executed would be the virtual machine, while part of the data would be the code to be executed.

3.3.2 Input Data Definition

Besides the definition of jobs with only one task, SPADE user interface allows the definition of jobs with multiple tasks, each one possibly processing different data. In order to define these simpler jobs, the user should select one of the available applications, select the input files, and define the command line arguments to use when invoking the application. All other steps should be transparent: hosts selection, transmission of the data and files, and invocation of the application.

In order to allow the execution of jobs comprising multiple tasks, SPADE should allow the easy partition of available data (files and parameters):

- Each task processes a different input file, also generating a different output file. The name of this file should contain a numerical prefix that should match the task identifier. This category includes batch image processing or statistical analysis of different data sets.

- Each task has a numerical argument ranging from 0 to N. This numerical value can be a task identifier or its transformation. In the case of a film generation with POV-Ray, this numerical argument may represent the timestep of the frame being generated within the film.

- Each task processes a range of an input data set or output space, receiving as arguments the limits of the assigned partition. The rendering of a complex image can be parallelized giving to each task the responsibility to generate a piece of the output image. For instance, a task would render a view from $(0, 0)$ to $(1023, 511)$ while another task would generate the view from $(0, 512)$ to $(1023, 1023)$. This example represents a two-dimensional space sweep but a one-dimensional space sweep should also be possible.

Any combination of these examples could be used to define a job task.

Besides these simple, but expressive data partition mechanisms, SPADE should also allow the traditional redirection of the *stdin*/*stdout* to predefined files.

3.3.3 Architecture

Independently of the underlying technologies deployed, SPADE should hide from the user most of the details related to configuration of cluster:

- The file transmission should not rely on distributed file systems, since it would require its configuration.
- The creation of the pool of available computers should be transparent. As long as a computer has the SPADE system running it should be able to discover and be discovered by others.
- The configuration of the available application and how to use it should be simple, resorting only to the identification of the application location on the disk.
- The selection of computers where to run tasks should be hidden from the user.

3.4 Architecture

In order to comply with the previously presented architectural requirements, SPADE should be modular, allowing easy development of the necessary functionalities, and future integration with existing systems. The overall SPADE architecture is presented in Figure 3.1.

SPADE is composed of a *Client* application, where the user creates and submits the jobs to be executed and *Daemons*, running on remote computers. Although logically separated, a computer running the *Client* module, also executes the *Daemon* code.

As stated earlier, SPADE relies on commodity software installed on the cycle providing computers. It is the responsibility of the *Application Manager* to keep a record of the software accessible by remote SPADE clients. This information is registered locally in each *Daemon*, but transferred transparently to each *Client*.

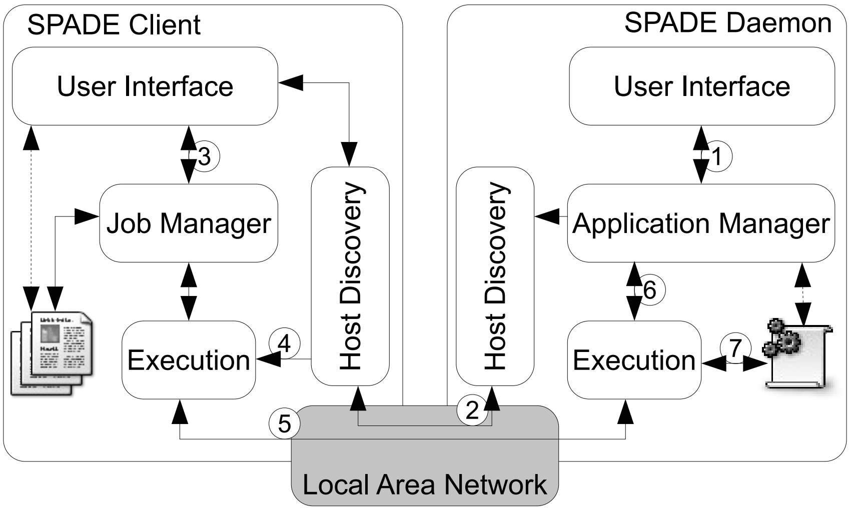


Figure 3.1: SPADE System Architecture

The registration step (step 1 in Figure 3.1) must be performed before any job is created. The user should only provide the path to the application and its well know identifier (povray, or povray2.5, for instance). Whenever there is a task to be executed, this module is contacted to provide the path to the executable.

During normal operation, *Clients* maintain updated information about what *Daemons* are running and what applications are available there (step 2). *Clients* only need to know the well known identifier of the remotely installed applications.

When submitting a job, the user provides the *Job Manager* module with all information regarding the job and tasks to be executed over the network (step 3). This information includes the application's well-known name, its arguments and the files (input or output) that should be transmitted over the network. The user must also state how many tasks comprise the complete job. Then, this module submits each task to the *Task Execution* module and keeps track of their state (*not started*, *started* or *finished*). The *Task Execution* module chooses where to execute each task from the information provided by the *Host Discovery* module (step 4).

After receiving information about a task to be executed (step 5) The *Daemon*, contacts the Application Manager, retrieves the exact location of the application to be executed (step 6), and executes it in a directory containing the input files (step 7). Result files are then transmitted back to the client.

3.4.1 Implementation

Although it wold be possible to implement in part the described architecture by combining already existing systems (service discovery, task scheduling on clusters) the resulting system would not comply with the requirements the target users pose: users would have to configure the required systems, and submit tasks using complex languages.

In order to make SPADE compatible with the largest number of hardware architectures and operating systems, it was developed in Python. The use of Python also reduced development time. Communication between Client and Daemons was implemented using a XML-RPC and the user interface uses the wxPython framework [wxp]. Any other programming language that provides some sort of RPC mechanism could have been used, and any module implemented can be replaced or complemented with any existing system.

Figure 3.2, presents the UML class diagram of an executing daemon.

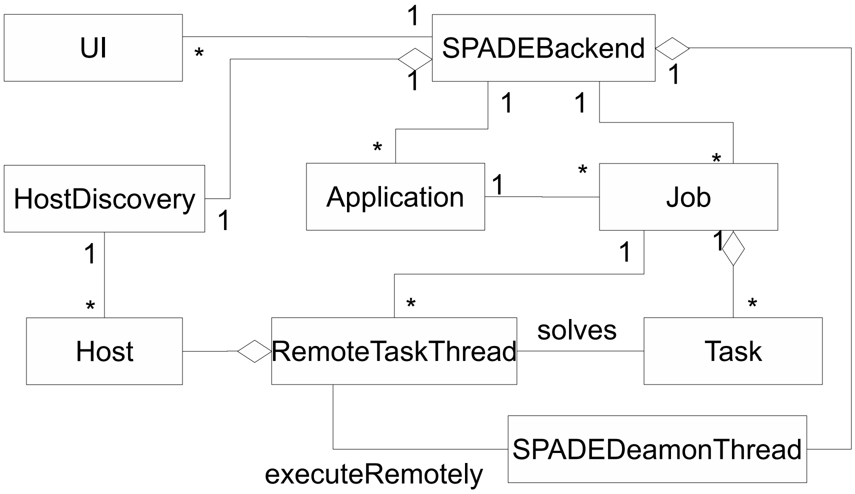


Figure 3.2: SPADE UML class diagram

SPADEBackend is the class that encapsulates all SPADE daemon behaviour. It receives requests from the user interface module to register new applications and new jobs. *SPADEBackend* has a list of *Applications* containing the well known name of each application and the path to the corresponding executable.

Each *Job* object stores the corresponding application and the set of tasks that make it. Each *Task* object contains all the information necessary to its local or remote execution: arguments, input files data and output file name.

The *UI* class handles all user interaction, its implementation and use will be presented in Section 3.5.

On every participating computer, the *Host Discovery* module is responsible for the maintenance of the list of available computers (and the application installed there) located on the same network. In order for a Daemon to announce its existence, it sends periodically an broadcast message containing its identification, along with the list of installed and registered applications. Clients listen to that broadcast port and, when receiving an announcement, store the identity of the daemons (and the software installed there) for latter deployment of tasks.

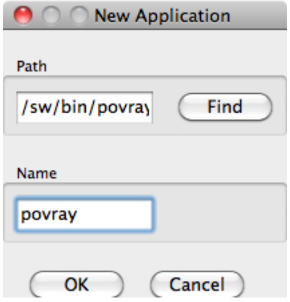
In order to allow parallel execution of several tasks and control the interaction with several remote Daemons, for every Daemon that has the required application, a remote *RemoteTaskThread* thread is created. Each *RemoteTaskThread* contacts the corresponding SPADE daemon, delivering the input files and the shell command and receiving the result file.

3.5 Job Submission

One of the requirements of SPADE is to allow the execution and creation of jobs without complex configurations. To accomplish that, the user interface should allow the creation of tasks in a simple and straightforward manner.

Before the creation of any task, the only configuration step is the registration of available applications. At each remote computer, the user should provide the path to the application and its well know identifier, as presented in Figure 3.3.

The name given to the application should be uniform among all configured



Path
/sw/bin/povray

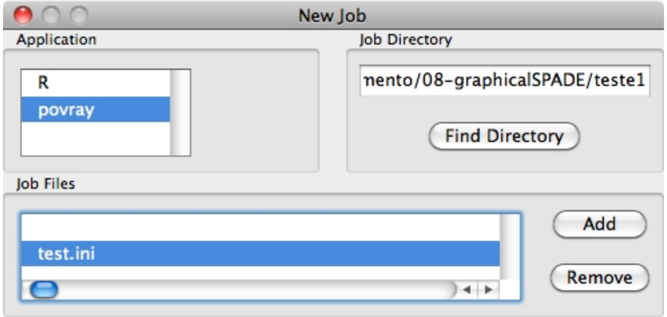
Name
povray

Figure 3.3: SPADE application registration

computers, otherwise it would not be possible to assign tasks to all available computers.

In order to define the jobs that will be executed, the user must first select the necessary application, the location of the input files and what files should be transmitted to the remote hosts, and define the command line arguments of each task. All these steps are performed using a graphical user interface, split in three different areas: i) execution environment setup, ii) parameters definition, and iii) command line definition.

Figure 3.4 shows the fields necessary to the definition of the execution environment.



Application
R
povray

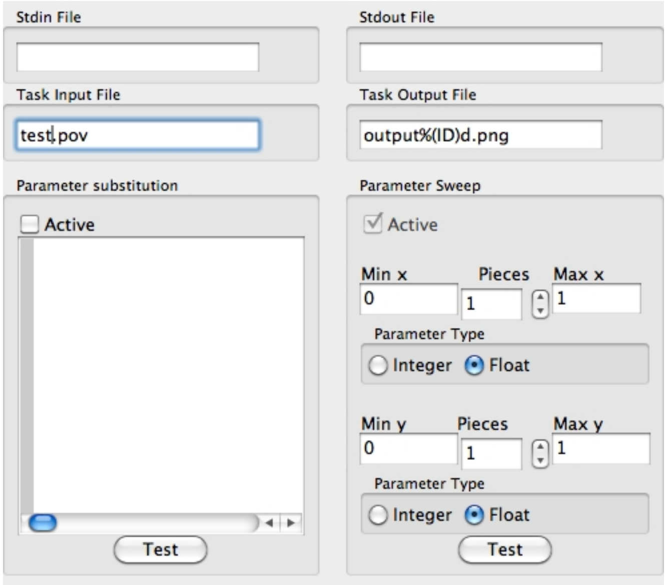
Job Directory
nento/08-graphicalSPADE/teste1

Job Files
test.ini

Figure 3.4: SPADE job submission user interface, execution environment setup

The user must choose, from the available processing applications, the one to use, define the directory where input files are presented and where results are copied to, and select what files are to be read by every task. All information is constant and similar to every task.

In order to define different input or output files, or different parameters for each task, the user must fill the parameters definition part of the user interface (shown in Figure 3.5).



Stdin File

Stdout File

Task Input File: test.pov

Task Output File: output%(ID)d.png

Parameter substitution

□ Active

Parameter Sweep

☑ Active

Min x: 0 Pieces: 1 Max x: 1

Parameter Type: ○ Integer ● Float

Min y: 0 Pieces: 1 Max y: 1

Parameter Type: ○ Integer ● Float

Figure 3.5: SPADE job submission user interface, parameters definition

The upper fields allow the definition of the input and output of the tasks. In the fields *Stdin File* and *Stdout File* the user defines the name of the files to emulate the keyboard and console. The file which name is inside the *Stdin File* field will be used as the keyboard input of each task, while the output written in the screen will be redirected to the *Stdout File*.

The fields *Task Input File* and *Task Output File* allow the user to specify, to each task, the individual files to be copied to/from the remote host. The file referred in the *Task Input File* will be copied to the remote computer along with those specified in the *Job files* list. At the end of each task, the *Task Output File* is copied back from the remote computer.

In order to allow different tasks to process, read, generate, and write file with different name, the user should use the `$(ID)d` placeholder in their definition. When generating the actual file names, SPADE will replace that placeholder with the actual task identifier (integer higher or equal than zero). `%(ID)d` is similar in meaning and function to the `$(Process)` placeholder from Condor.

Although in Figure 3.5, the `%(ID)d` placeholder is only used in the *Task Out-*

put File field, it can be used in the other three if necessary: each task processes different input files, reads different information from the keyboard, or writes different information to the screen.

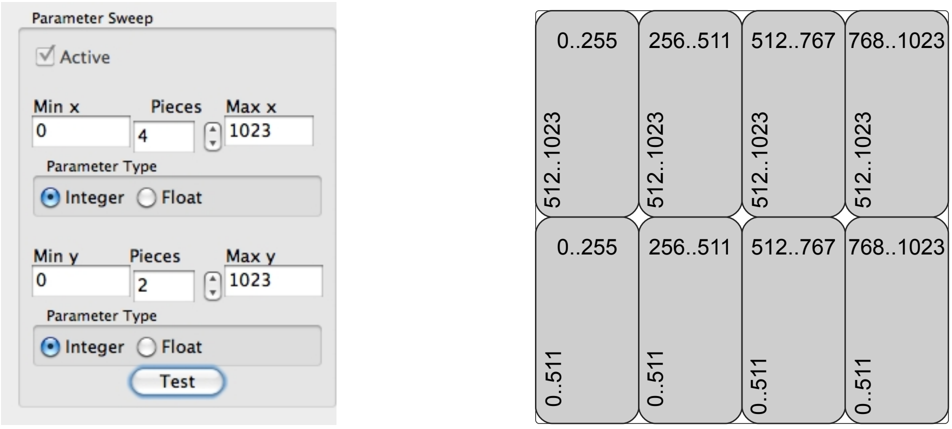
In the previews example, every task receives as input the files `test.ini` (shown on Figure 3.4) and `test.pov`, while generating different output files:

- output0.png, by task 0
- output1.png, by task 1
- ...

Although the use of the the `%(ID)d` placeholder allows some differentiation between tasks, it is still too limitative. The only value that varies between tasks is their identifier and the input/output files.

In order to allow more complex task differentiation, the user can fill the *Parameter Substitution* and *Parameter Sweep* areas.

In the *Parameter Sweep* area, the user defines how a uni or bidimensional range of values can be split among tasks. The user defines the limits of the set (setting the minimal and maximal values) and selects the number of partitions to be created from that set. For each task, SPADE assigns one of the produced partitions.



Parameter Sweep

☑ Active

Min x: 0 | Pieces: 4 | Max x: 1023

Parameter Type: ● Integer ○ Float

Min y: 0 | Pieces: 2 | Max y: 1023

Parameter Type: ● Integer ○ Float

0..255 512..1023	256..511 512..1023	512..767 512..1023	768..1023 512..1023
0..255 0..511	256..511 0..511	512..767 0..511	768..1023 0..511

a) b)

Figure 3.6: SPADE value range splitting: a) user interface, b) result division

Figure 3.6 presents the fields needed to be filled in order to define the subsets to be processed by each task. The user defines the minimum and maximum of the X and Y sets, along with the number of divisions to be performed at each axis.

The resulting partitioning is presented in Figure 3.6.b). For each task, SPADE generates four boundary values that can be used in the definition of the input/output files names or on the command line arguments. Each of the produced limits is assigned to a specific placeholder: `%(MINX)s`, `%(MAXX)s`, `%(MINY)s` and `%(MAXY)s`. These placeholders can be used in the definition of the input/output files in the same manner as the `%(ID)d` placeholder.

In the *Parameter Substitution* field, the user can write in Python a simple function with a `transFunc(i, nTasks)` header, where the parameter `i` will be replaced by the actual task identifier and the `nTasks` is the total number of tasks. Also in this case, the calculated value can be used in the definition of file names and command line arguments, by means of its specific placeholder (`%(NEWPARAM)`).

Up until now the user only defined the files to be processed, and how the data set can be split. Now it is still necessary to define the command line to be used when executing each task. This is performed filling the *Command Line* field (presented in Figure 3.7).

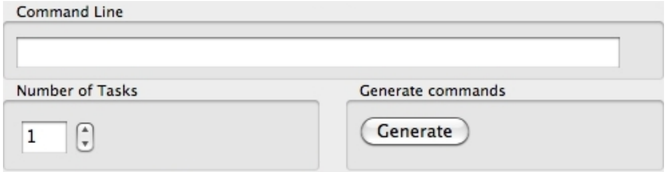


Figure 3.7: SPADE job submission user interface, Command line

When defining the command line arguments for each task, the user can and should use the values defined earlier: task identifier, name of files (input and output), and limits of the input data set. Besides the already presented `$(ID)d` placeholder, all other calculated values (including the names of the input/output files) are also available by means of their specific placeholders.

All the available placeholders are presented in Table 3.1.

Placeholder	Data Type	Description
$(ID)d	Integer	Numeric identifier of current task. Value ranging from 0 to number of tasks minus one.
%(MINX)s %(MAXX)s	Real / Integer	Limits of the X data set of a partition. Calculated from the values inserted in the *Parameter sweep* fields.
%(MINY)s %(MAXY)s	Real / Integer	Limits of the Y data set of a partition. Calculated from the values inserted in the *Parameter sweep* fields.
%(NEWPARAM)s	any	Value calculated for each task in the function *transfunc*.
$(INFILE)s	String	Actual name of the input file. Calculated expanding the expression written in field *Input File*.
$(OUTFILE)s	String	Actual name of the output file. Calculated expanding the expression written in field *Output File*.

Table 3.1: SPADE task definition placeholders

When using these placeholders, the user is modifying the actual command line argument used in the creation of tasks, since each placeholder is replaced by a valued calculated taking into account the actual task identifier.

Using the configuration stated in Figures 3.5 and 3.6.a), the user can write the following command line:

%(INFILE)s -O**%(OUTFILE)s** -SC**%(MINX)s** -EC**%(MAXX)s** -SR**%(MINY)s** -ER**%(MAXY)s**

This example would render 8 tasks each one with one of the following command lines arguments:

- test.pov -Ooutput0.png -SC0 -EC255 -SR0 -ER511
- test.pov -Ooutput1.png -SC256 -EC511 -SR0 -ER511
- test.pov -Ooutput2.png -SC512 -EC767 -SR0 -ER511
- test.pov -Ooutput3.png -SC768 -EC1023 -SR0 -ER511
- test.pov -Ooutput4.png -SC0 -EC255 -SR512 -ER1023
- test.pov -Ooutput5.png -SC256 -EC511 -SR512 -ER1023
- test.pov -Ooutput6.png -SC512 -EC767 -SR512 -ER1023

- test.pov -Ooutput7.png -SC768 -EC1023 -SR512 -ER1023

In the generation of the actual command line, SPADE first calculates results s returned from the invocation of `transFunc`, and the data set partition. With these values and the task identifier, the input files are defined next.

After building the file names, it is possible to replace all placeholders present in the command line.

The user can define the total number of tasks to create, by setting the correct value in the *Number of Tasks* field (figure 3.7). If the user performs data sets partition using the *Parameter Sweep* fields (Figure 3.6) this value is automatically set with the correct number of tasks.

3.5.1 Job Execution

When a job is created by means of the form presented in Figures 3.4, 3.5 and 3.7, an object of class *Job* is created and populated with all its tasks' information. When creating this *Job* object, all the values defined by the user (except the *Command Line*) are expanded by replacing the placeholders with their actual values. This newly created *Job* object will contain the execution information for each of the composing tasks.

Figure 3.8 presents a Job execution UML sequence diagram, of the steps necessary for the execution of a Job, after its creation.

After the *Job* object creation and initialization, a thread is created. This thread will get a list of hosts (`gethosts()`) that have SPADE installed and the corresponding job application . Then, for each host discovered, a *RemoteHostThread* is created. Each one of these threads will fetch tasks (`getNextTask()`) and send the necessary data to the remote host (`executeRemotely()`).

In order to execute the task, the remote host must receive the following information: the name of the input and output files, the actual content of the input files, the task identification and the command line without the placeholders re-

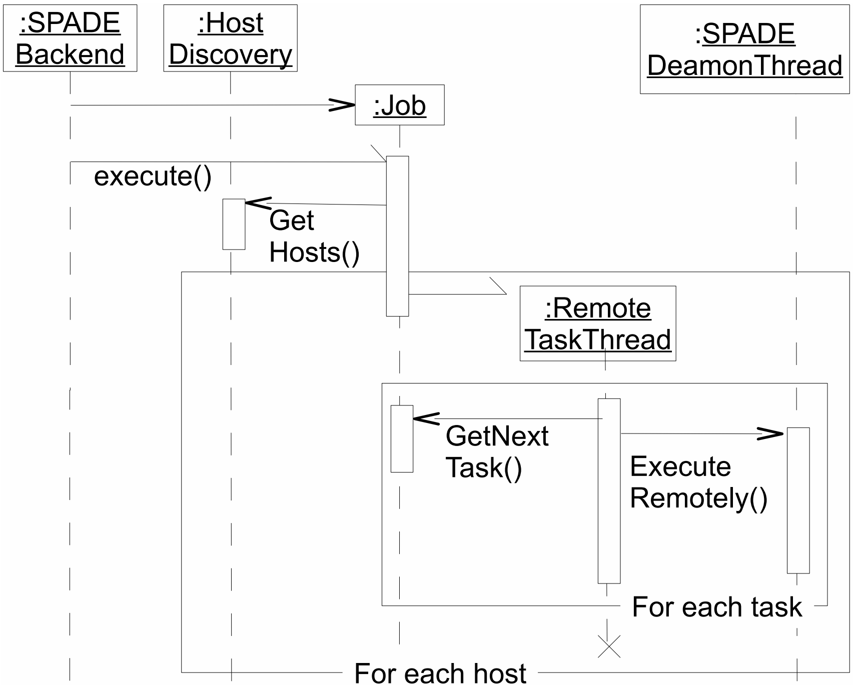


Figure 3.8: SPADE job execution UML sequence diagram

placed. Besides this, the actual contents of the input file should also be transferred to the remote host.

After receiving all information and data, the *SPADEDeamonThread* creates a temporary directory and copies the input file there. The actual command line is built taking into account the executable location and the temporary directory where the input file is. The placeholders `%(INFILE)s` and `%(OUTFILE)s` are replaced with the concatenation of the temporary directory and the files names. After the generation of the actual command line the task is executed, the result output file is read and its contents returned to the cycle consumer host.

Every task may be in one of three possible states: *not started*, *started* and *finished*. When a task is executed, its state changes from *not started* to *started*. Until receiving the result of its execution, the state of a task remains in the *started* state, changing to *finished* after the reception of its results. In order to guarantee that all tasks are executed, after all tasks are initiated (no more tasks in the *not started* state) the previously started tasks are sent to available remote hosts. This way, duplicate results may be received, but there is the added guarantee that all tasks will finish.

3.6 Evaluation

SPADE is mainly composed of two components, a user interface and LAN task scheduler. So, it is necessary to evaluate the speedups attained from its use and study what classes of problems and jobs can be created using it.

In order to evaluate SPADE performance and the possible speedups, SPADE was used to deploy large jobs composed of multiple tasks and execute them on various hardware configurations. The experiment was composed of the rendering of 120 frames to create a computer-generated film. POVRay was used to render each frame.

The first original execution was performed on a desktop computer with a 3.06GHs Intel(R) Core(TM) 2 Duo processor running Mac OS X. Using a single core and rendering each frame sequentially, it took 4400 seconds (about 72 minutes) to generate the movie. This result in presented in the first column (**Original 1/1**) in Figure 3.9.

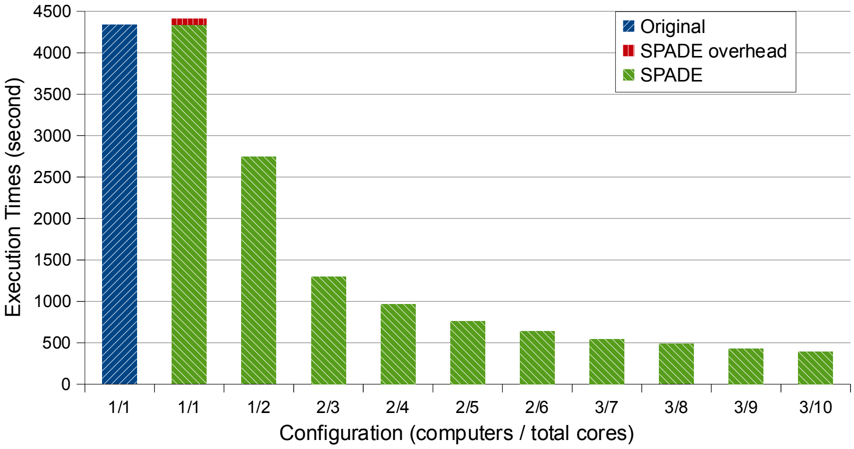


Figure 3.9: SPADE based film rendering parallelization execution times

The second column presents the time taken to execute the same job, on the same computer, but from within SPADE. Due to the SPADE additional steps (task deployment, execution and result retrieval) it presents some overhead. For this experiment, the total overhead time is 75 seconds (less than 2% of the total execution time). On average, each task incurs less than 1 second from being deployed by SPADE.

The following experiment deployed tasks concurrently on multiple processors or cores. The experiment **SPADE 1/2** executed all tasks on the same computer, but took advantage of the two available processing cores. Since no processor affinity was defined, and these cores were also shared with the operating systems, a suboptimal speedup was observed. Nonetheless the speedups are evident.

The experiments with more than one computer resorted to additional computers with a 2.40GHz Intel(R) Core(TM)2 Quad processors running Ubuntu 9.04 Linux. As expected, the total execution times decrease with the addition processing cores.

In order to evaluate if there are considerable gains even from using additional slower computers, the previous experiment was executed concurrently on the original computer (3.06GHs Intel(R) Core(TM) 2 Duo) and on a Netbook with a 1.6GHz Intel(R) Atom(TM) processor and running Windows(R) XP. These results are presented in Figure 3.10.

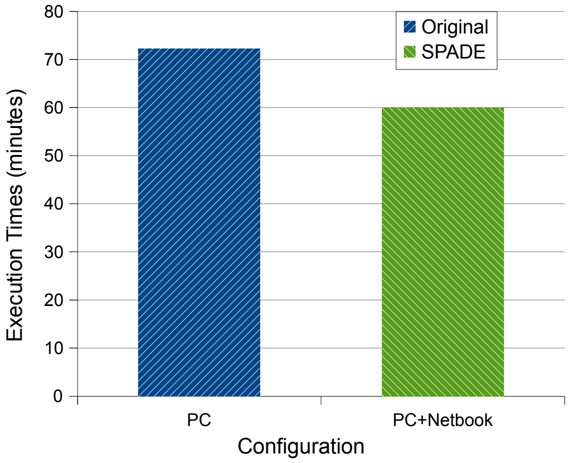


Figure 3.10: SPADE based film rendering parallelization speedup

The first column represents the time to execute the movie rendering on the original computer (with only one core participating), while in the second column the Netbook also takes part in the computation. As expected there are gains that, in this example, are higher than 10 minutes: a job taking 72 minutes was executed in 60. The presented gains are high enough to justify the time necessary to setup the jobs using SPADE.

With respect to usability factors, SPADE should be capable to fulfil the requirements presented in Sections 3.3.3 and 3.3.2.

Although simple, the job creation interface is expressive enough to allow the creation of most jobs a common user may have. The user is capable of defining jobs, where each task receives as parameter the task identifier, and where different files are processed by each task, as well as easily define a one or two-dimensional space partitioning. The writing of the Parameter substitution function requires some programming knowledge, but by providing some sample transformation function, any user can adapt them to its needs.

As an example, Table 3.2 ilustrates the configurations necessary to fulfil each of the classes of jobs described in Section 3.2:

- different arguments for each task
- different input and output files for each task
- different partitions of a data set for each task

Task differentiator	UI field	Assigned Value
≠ arguments	Task Input	test.pov
	Task Output	result%(ID)d.png
	application	povray
	Command Line	%(INFILE)s -Initial_Frame=%(ID)d -O%(OUTFILE)s
≠ arguments	Task Input	simulation.class
	Task Output	result%(ID)d.txt

	Parameter Substitution	def trans(i,n):
		    return 1.0*i/1000

	application	java
	Command Line	simulation %(NEWPARAM)s

Table 3.2 (Continues on next page)

Task differentiator	UI field	Assigned Value
≠ files	Task Input	image%(ID)d.png
	Task Output	result%(ID)d.png
	application	convert
	Command Line	%(INFILE)s -resize 102x76 %(OUTFILE)s
≠ data sets	Task Input	test.pov
	Task Output	output%(MINX)s-%(MAXX)s-%(MINY)s.png
	Min X	0
	Pieces	4
	Max X	1023
	Min Y	0
	Pieces	120
	Max Y	8
	application	povray
	Command Line	%(INFILE)s -K%(MINY)s -O%(OUTFILE)s %-SC%(MINX)s -EC%(MAXX)s

Table 3.2: SPADE configuration examples

Any user with jobs similar to the ones presented in the table, and already capable of executing them in a serial manner, will be able to fill correctly the required fields.

After being installed, the only necessary configuration is to state where processing applications are located on each computer. This straightforward step is performed by means of the dialogue box presented in Figure 3.3. Furthermore SPADE executes unmodified and with a similar user interface in any Desktop computer running Windows, Linux or Mac OS X.

3.7 Conclusions

This section presents a possible solution for the creation of Bags-of-Tasks. As presented earlier, none of the available systems allows an easy creation of tasks by users without extensive programming knowledge: users were required to program all steps of any parallel job wanting to execute.

SPADE offers a simple, yet expressive mechanism to partition data and deploy parallel tasks on any execution environment. The User Interface allows the submission of jobs where each task processes a different file, or the creation of more complex data partitioning: parameter sweep or data sets splitting. Most of the jobs the target users of the work presented in this dissertation will fit into these classes of jobs.

SPADE can be used to create jobs to be executed on a cluster or on the Internet. Although many Internet Based Distributed Computing systems exist (as presented in Chapter 2), none provided suitable means for the creation of jobs by the majority of their users. These systems could have been be envisioned to be used by common users, but nonetheless are unsuited to them.

SPADE can also be used as a User Interface for a job scheduler on a cluster of computers. The experiments shown were attained on a small scale cluster composed of four desktop computers with a total of ten processing cores. The experiments showed that even with a small number of computers it is possible to observe speedups.

Besides the data creation User Interface, another contribution to the ease of execution of parallel jobs, was the creation of a cluster-based task scheduler. The underling execution environment requires no configuration. It is only necessary to have a Daemon running on the computer where tasks are to be executed. As it is to be executed on privately owned home LAN, security requirements can be relaxed. Furthermore, the user explicitly defines what application (those that can not cause harmful side effects) are to be remotely executed.

With these two contributions (a User Interface and an underlying task execution systems) users with parallel lengthy tasks, but without the knowledge to use existing systems, can take advantage of idle computers to speed their tasks execution.

4

Bag-of-Tasks Automatic Parallelization

The previous chapter shown mechanisms to simplify the creation of Bag-of-Tasks where the processing code is a complete application, and the data splitting is external. Although useful, it does not address all Bag-of-Tasks creation possibilities. If the data splitting needs to be performed inside the developed application, SPADE is of no use.

Today, the development of Bag-of-Tasks, i.e. embarrassingly parallel applications, for execution on multiprocessors or clusters requires the use of APIs. The user modifies a serial version by including the task creation calls. Besides the obvious need to change source code (with the possible creation of error), users are required to learn a new API, and after transformation, the code becomes tied to a single architecture.

So, in order to address presented issues, it is necessary to provide tools to automatically create tasks from a serial application.

Mercury provides a platform for the transformation of serial applications into parallel Bag-of-Tasks. It simply reads a configuration file stating what methods and classes should be parallelized, loads the application and, in run-time, transforms it so that the specified methods are executed concurrently. This transformation is performed without user intervention. Its modular design allows the integration of Mercury with different parallel execution environments.

The experiments done show that the overhead is minimal, and that it is possible to take advantage of parallel processing environments (such as multiprocessors/multicores, or clusters) without the use of complex APIs.

4.1 Introduction

Besides the repetitive invocation of an application with different command line arguments, it is also possible to develop serial applications that internally and independently process different arguments. In order to take advantage of available multi-processors it is necessary to use specialized libraries, either with parallel implementation of lengthy functions (for instance BLAS[1] parallel implementations [FC00]), or to create parallel tasks to be executed concurrently.

Even if the problems fit into the Bag-of-Tasks category, for such applications to be parallelized the user is required to modify the application and use some sort of API to create tasks.

For instance, the use of MPI [Mes94] to create tasks is possible, but may be an overkill, since the user is required to learn a new API. In general, for the execution of a pure Bag-of-Tasks, the use of the above mentioned approaches should be avoided, as it requires the use of specialized APIs to perform explicit parallelization, synchronization and data transmission.

As a matter of fact, the gains achieved may not be enough to justify the extra complexity the programmer has to manage to parallelize his application. Moreover, a user with limited coding skills may not be able to handle such complexity. Furthermore the transformed application is tied to a particular API and system, and requires extensive and complex debugging in order to guarantee its correctness, and extra effort to port it to different API of another system.

As an example, consider the effort of parallelizing the application shown in Listing 4.1 (allowing all `processData()` methods to execute concurrently).

```
for i in range(1000):
    inputData = getIterationInput(i)
    objList[i] = processinObject()
    objList[i].processData(inputData)
    outputResult = objList[i].getResult()
```

[1] Basic Linear Algebra Subprograms

```
    process(outputResult)
```

Listing 4.1: Typical serial Bag-of-Tasks pseudo-code

Using MPI, or any other parallel execution middleware; the following modifications have to be done: i) identification of master and slave tasks, ii) transmission of data to slaves, and iii) receiving the results.

A possible parallelization of the previous application would be the presented in Listing 4.2. In this example each of the iterations of the original application is transformed in a distinct task.

```
1  if is_master:
2      for i in range(1000):
3          inputData = generateTaskInput(i)
4          tasks[i] = CreateTask(taskCode)
5          sendData(tasks[i], inputData)
6      waitForCompletion(tasks)
7      for t in tasks:
8          outputResult = ReceiveData(t)
9          process(outputResult)
10 if is_slave:
11     inputData = ReceiveData(master)
12     outputdata = processData(inputData)
13     sendData(master, outputData)
```

Listing 4.2: Typical Parallelization of a Bag-of-Tasks

Since a message passing was used to communicate among tasks, it was necessary to explicitly include communication primitives on the master process and on all tasks: the transmission of the input data is performed in lines 5 and 11, and the reception of the results in lines 13 and 8. Furthermore, it is necessary to include the creation of all tasks (line 4).

This direct transformation of a loop iteration in tasks may not be viable: it is necessary to have proficiency on the used parallel programming API, and it is necessary to guarantee correctness of the new code. As already stated, the new code is tied to a single API, and to a limited set of parallel execution environments.

With respect to computational efficiency, the number of created tasks may be excessive, leading the system to thrashing. It is still necessary to enforce a limited number of tasks (related to the number of available processors), being each task responsible for a series of iterations.

In order to solve previews issues, a more generic approach could be as follows (Listing 4.3).

```
class Task:
   def init(inf_limit, sup_limit):
      ...
   def executeTask():
      for i in range(self.inf_limit, self.sup_limit):
            processData(i)
...
for n in (nTasks):
   task[n] = Task(1000/nTasks * n, 1000/nTasks *(n+1)-1)
   task[n].executeTask()
waitForCompletion(tasks)
for n in (nTasks):
   outputResult = task[n].getResults(t)
   process(outputResult)
```

Listing 4.3: Typical Parallelization of a Bag-of-Tasks using objects

In this example, task creation is explicit, but communication is not. Input data is transmitted when creating the task and results should be coded in the method `getResults`. Nonetheless the presented code is neither tied to any specific API nor architecture, and can easily be transformed to include parallel primitives, but is close to a serial version.

In this example each task would execute part of the iterations (each one with a parameter between `inf_limit` and `sup_limit`). Each task partial result would contain the results of several iterations. Although the task code is encased in a class, it could have been simply stated as a function, that would receive as arguments the limits of the data set to be processed and return the result. This model is presented by Jon Nilsen et al [NCHL10], along with possible parallelization using pypar [Nie11] MPI implementation.

The code structure modification generates a small impact on the complexity of the development of a parallel version. It is the integration with a parallel execution runtime that requires the largest programming effort, as a new API must be used, and new error-prone code is introduced, thus reducing programmer's productivity, as well as applications robustness. Furthermore, from the moment additional calls are inserted, debugging the application requires a parallel environment (even with a single processor), becoming more difficult to debug than the serial version.

4.1.1 Objectives

Thus, the goal of the work here presented is to minimize the programmer's modification effort necessary to transform a sequential application, as the example presented in Listing 4.1, into a parallel Bag-of-Tasks. For that purpose, Mercury was developed. Mercury is a middleware that allows the parallelization of independent object methods, allowing their concurrent execution on different local threads or different remote computers.

This solution transparently transforms and parallelizes sequential applications containing long independent iterations. This transformation and execution on a parallel environment are done with no user intervention.

These lengthy tasks are executed on different threads, either locally, in the case of a multiprocessor (or multicore) computer, or on remote computers.

In order to use Mercury, the programmer still has to adhere to very simple coding requirements: the sequential application must follow the structure and form presented in Listing 4.3, and state in a XML configuration file the class and methods that can be executed concurrently.

The code to parellelize must be composed of a worker class (`Task` class in Listing 4.3) and an initialization code following this structure:

- loop with creation of worker class instances and invocation of the processing code (lines 8 to 10)

- loop with the retrieval and processing of the results (lines 12 to 14)

The primitive from line 11 is not necessary.

The Mercury middleware is responsible for spawning the necessary threads, and synchronize the invocation of the methods. In the previous example, only after the conclusion of `processdata`, the corresponding `getResult()` method can be executed.

Mercury also allows different execution environments for the parallelized methods. A set of adaptation classes allows the execution of the methods on different threads on the same multiprocessor/multicore computer or the remote execution of faster computers. The indication of what parallel execution environments are available is also made in a declarative way. The selection of the best location to execute the parallel code is made during run-time, and takes into account both the code requirements and the computing resources availability.

With run-time code adaptation, the programmer only needs to develop a sequential application, test it and verify that the results are as expected. In order to take advantage of available parallel resources, all the code to perform task creation and work distribution is automatically and transparently weaved in the correct places during application execution. Besides requiring only one version of the application, the sequential one, it does not become tied to any specific execution environment.

The proposed solution uses metaclasses, allowing the modification of the code to be done in run-time, without any need to transform or recompile the source code. The developed metaclass intercepts all class creations and modifies the implementation of those that are to be made parallel, without any user coding intervention: the user must only state what classes have methods that can be executed concurrently with the rest of the code.

The next section presents some technologies and systems that address similar issues as work here presented (parallel execution and reflection). Sections 4.3 and 4.4 describe the architecture and implementation (respectively) of Mercury.

Finally performance and functional evaluations are presented along with the conclusions.

4.2 Related work

The most evident way to transform a sequential application into a parallel one is using some sort of task creation and data communication API. These API depend mostly on the programming model used (Section 1.3.2).

The most widely used API is MPI [Mes94], which independently of the underlying architecture (shared or distributed memory), allows the programmer to define the synchronization and communication points between tasks. Any parallel application developed with this API, is dependent of its availability in the target architecture and operating system, and requires programming proficiency.

To solve this dependency of the code on a specific programming API, it is possible to delegate the responsibility of modifying the code (by inserting the communication, and synchronization primitives) to a special compiler.

OpenMP [Boa08] requires the user to annotate the code to be parallelized with special comments. These comments are processed by a special compiler generating a new executable, with the required and suitable calls. The way data is transferred between tasks and how synchronization is performed depends on the target execution architecture.

Using a similar idea, aspect oriented [KLM+97] systems allow the separation of the functionality of the application, from complementary aspects or concerns such as distribution or concurrency.

With respect to concurrency [SCM07], it is possible for the programmer to state what methods can and should be executed concurrently and delegate all code changes to a specific code transformation tool. This tool is responsible for the insertion of task creation, synchronization and data transmission function calls.

Later developments [CS07] allowed mixing of code annotation (as seen in openMP) with the use of aspect oriented programming. The classes and methods to be parallelized are annotated (inside comments), which are later treated by special compilers. These special compilers transform the application inserting the necessary code to guarantee that the final application version delivers the required behaviour.

Although reducing the programmer's generated code and programming effort, these systems still required the execution of a transforming tool (or special compiler) and the maintenance of multiple versions of the application: the sequential one, and the parallel ones targeted at distinct execution environments.

In order to avoid the existence of multiple executables for multiple architectures, and reduce the programming burden, a suitable middleware is to be used. Middleware overcomes non-functional concerns related to system issues (e.g., distribution, communication, parallelism) and frees developers from repetitive and/or intricate code (unproductive and error-prone), in the application execution phase.

To improve middleware and application behaviour, the former provides inspection and reconfiguration abilities (reflective middleware [KCBC02]) that may be invoked by applications, or configured by declarative policies. This configuration capability is only available with the use of special compilers or off-line application transformers.

Thus, middleware may even be monitored and triggered automatically, according to surrounding environment and execution context [BCD+97], and by autonomic systems (e.g., cluster and cloud computing management). Other non-functional concerns can be addressed using techniques similar to aspect-oriented programming for clean software design and reuse, but handled by the underlying execution middleware [GTLJ07].

Nonetheless, middleware should not impose new or heterodox programming models, languages, APIs as it reduces developers' productivity and applications portability, and increases error proneness. Thus, meta-object protocol approaches should be preferred instead [BK88].

The research work more related to the one presented here includes middleware projects that attempt to improve application performance by providing support for object distribution, deployment, remote execution, or thread parallelism (and parallelization). This entails that process/thread enrolment, synchronization, scheduling and resource management must be also addressed by the middleware and execution environment, or left for the programmer to resolve.

A natural direction for a middleware development is to extend the built-in distributed invocation mechanism in Java VM (RMI) via middleware with enhanced semantics. In the middleware described by Laurent Baduel [BBC02], group communication services are provided as a new set of Java classes and methods with multicast and parallel execution with *wait-by-necessity* semantics (futures). Sven De Labey proposes another type system extension [LS07] but employs communication qualifiers as attributes associated to objects to be invoked with different strategies. Such approaches either lack transparency as they impose new APIs or are unable to allow automatic parallel method execution. Furthermore, both are unable to dynamically configure the number of participating processes.

The work described in [TS02] and [JS05] treats applications as bundles of components that may be dynamically deployed in different processes of a distributed system. In [TS02], application byte-codes (and comprising class types) are transformed to allow application partition and component distribution following hints by the developer. In [JS05], the Java Isolate API is employed. Although able to deploy code using native types and allowing lazy creation of distributed objects, they do not provide transparency since a component-based application architecture is implied. Thus, automatic parallelization, as proposed, is not allowed.

A different approach is to transparently join multiple virtual machines running on a cluster or distributed infrastructure, in a *larger* JVM. This allows the increase of resources available to existing applications. This is acomplished using a single system image with a shared global object space [AFT99, AFT$^+$00], and with global thread scheduling. A minimal implementation just ensures global identities for distributed threads, offering safe synchronization semantics [HMRT03], and supporting thread migration [ZWL02]. Although providing transparency and code portability, these approaches do not extract parallelism automatically as they only manage threads explicitly created by developer's code.

The work in [Kam07] aims at supporting parallelism and distribution in Java by offering a new API fusing the semantics of shared memory (OpenMP) and message-passing (MPI). Parallel code is defined inside lambda constructs passed as arguments to API methods. Synchronization is explicitly defined in code and scheduling is based on a task queue. Transparency to the programmer is not intended, and the number of processes involved must be known *a priori* by participants.

In Map-Reduce [DG04], programmers develop large parallel applications by providing functions to split input and aggregate results, tailoring it to a specific API and programming model. For efficiency, a distributed shared storage (e.g., distributed file system, table, tuple-space) is assumed, which requires some form of coordination and membership management among participating nodes. Existing applications cannot be parallelized transparently on Map-Reduce. Data splitting and task creation are also defined programmatically, not automatically handled by middleware.

The ProActive [AG03, BBC$^+$06, BCHM07] offers multicast invocation semantics and futures for Java, task distribution, and deal with distributed synchronization and cooperation semantics. It does not aim at transparency, initially imposing a specific API and type hierarchy, then evolving for a component model combining a set interfaces, active objects, introspection and byte-code enhancement. No adaptability nor transparency is sought for scheduling and parallelization.

Globally addressed and analysed, none of the proposed approaches in the literature combines the necessary transparency (do not impose new class hierarchy, API or programming model, using reflective approach instead), parallelization for performance improvement (transform local method invocations into distributed ones and execute sequential method invocations in parallel with automatic synchronization enforcement), and flexibility (able to adapt and configure middleware behavior w.r.t. scheduling, and being able to dynamically spawn one or more parallel tasks to processes currently available with no need for prior knowledge on group membership).

4.3 Architecture

The architecture of Mercury is closely mapped to its main functionality: class adapters loading, classes information loading, classes transformation, and transformed classes adaptation to the available execution environments.

The modules that compose Mercury and the way they interact are shown in Figure 4.1.

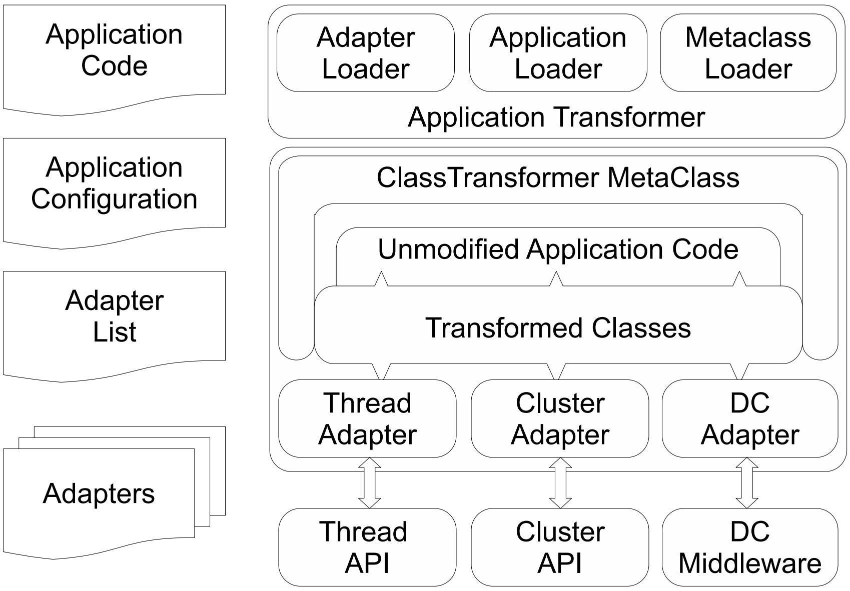


Figure 4.1: Mercury Architecture

The *Application Transformer* module replaces the entry point of the program being transformed, loads the transformation code and initiates the transformation of the original application. This module starts by loading and creating the

Adapter Loader, *Metaclass Loader* modules, then initializes the necessary data structures and starts the *Application Loader*.

The *Adapter Loader* reads a configuration file (the *Adapters List* in Figure 4.1) stating the available *Adapters*, loads all necessary code, and creates and registers the corresponding class. For each available computational resource (threads and other parallel execution environments with suitable middleware) there is one *Adapter*. Each of these *Adapters* are responsible for the creation and termination of the concurrent tasks on one infrastructure.

The *Metaclass Loader* loads the *ClassTransformer Metaclass* code, instatiates it and registers it for use when the *Aplication Loader* starts loading and processing the *Application Code*, taking into account the name of the parallel classes and methods stated on the *Application Configuration*.

During application execution, the organization of classes and objects is the one shown in Figure 4.2.

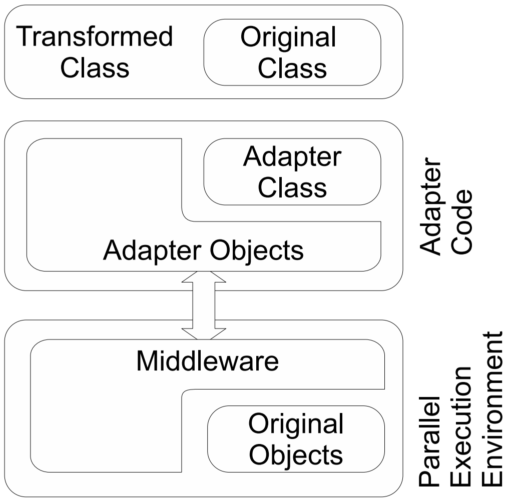


Figure 4.2: Mercury transformed classes organization

After loading the application, besides the original classes (and their objects), all auxiliary classes are created and instantiated, as described next.

The *Transformed Class* is a wrapper for the *Original Class*, to which it has one reference. When an instance of the *Original Class* is about to be created, it is responsibility of the *Transformed Class* to decide what kind of *Adaptation Object* to create. No instances of this *Transformed Class* exist during execution as it serves as an object factory: only instances of the *Original Class* (locally or on remote computers) and instances of the *Adaptation Classes* exist.

The *Adapter Objects* serves the purpose of handling all particularities of the underlying middleware parallel execution mechanisms: threads, or processes on remote computers. These objects act as proxies, being responsible for redirecting all calls to the original objects (instances of the *Original Classes*), and handling all synchronization issues.

During execution, other unmodified application objects only know the *Original Class* and its interface, but interact with *Adapter Objects* as if they were *Original Objects*.

4.3.1 Code loading and transformation

The code loading process and initialization of a transformed application is performed as shown in Figure 4.3.

Before any code is loaded, it is necessary to set up the execution environment. First the supplied metaclass is loaded and registered for later use (when the information about the parallel classes is read). The supplied *Application Configuration* files must contain the name of the classes to be transformed, and what methods can be executed concurrently. This information is stored and will be used when loading the application classes, and when invoking their different methods, as will be shown in the next section.

The *Adapter Loader* module is responsible for loading the *Adapter List* file and create the *Adaptation classes* referred in that file. This configuration file contains the list of externally available classes. From those names, the *Adapter Loader* obtains the name of the file with the class implementation and imports it. From this moment forward, the corresponding class is available for use. The names of all *Adaptation Classes* are stored in a list.

The last step before the application starts executing is the loading of the appli-

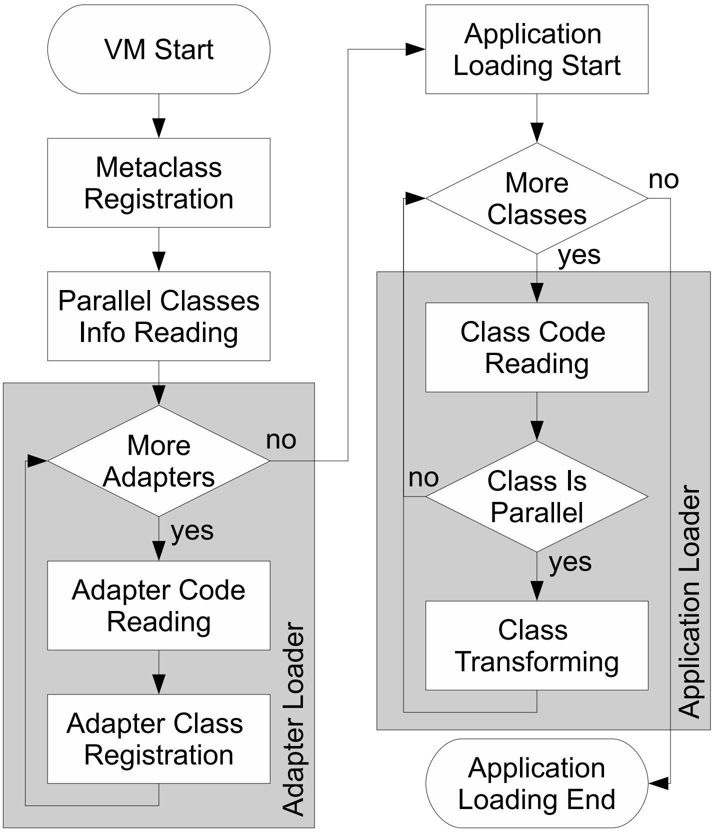


Figure 4.3: Mercury application start flowchart

cation bytecode from disk and its transformation. Inside the *Application Loader*, whenever a class is loaded from disk it is verified if its name was present in the *Application Configuration* file. If the name was present in the configuration file, a class transformation should occur, on the other hand the original classes (whose code was read) is created normally.

The code loading interception, and later transformation, either requires an intermediate step (as in Aspect Oriented Programming), or requires an execution environment that allows introspection and run-line code modification.

Due to its architectural characteristics, availability, and wide use in the area of scientific computing [CLM05, LD07, SSRP09, MB11, SCI11], Python was chosen as the language (and environment) to be used in the implementation of Mercury. The Python mechanisms used in the interception of the class loading and the class transformations are presented in the next section.

4.4 Implementation

The previous section presented an overview of how the application transformation is made. The implementation details and how the presented steps are carried out are now presented: i) interception of class loading, ii) class transformation and iii) adapter implementation.

All this transformation is performed when the application is executed, not requiring previous steps or multiple executables.

4.4.1 Class loading interception

Python allows the installation of a custom metaclass that intercepts every class loading, but due to the fact that some Python libraries require and install their own metaclasses, it is not possible to use a global metaclass. So for every loaded class, it is necessary to check if it is going to be transformed requiring the intervention of the custom metaclass. To perform this step, it is necessary to intercept, not the class creation (responsibility of the metaclass) but the code importing. Before starting reading the application code a custom import function is installed and executed.

Whenever a Python file is included, it is this modified import function that is executed. Its pseudo-code is shown in Listing 4.4.

```
1 def new__import__(fileName):
2     mod = __old__import__(fileName)
3     for name, object in mod:
4         if isClass(object) and  name in paralelClasses:
5                 object.assign_metaclass(classTransformer)
6 return mod
```

Listing 4.4: Mercury custom file import

On line 2, the original *import* function loads the file code , which is stored in the `mod` variable.

Then, in line 4, for every loaded object (constants, function, and classes), Mercury checks if it is a class and if that class was referred to in the *Applica-*

tion Configuration file (its name is stored in paralelClasses). In a affirmative case, the metaclass responsible for the actual creation of that class is replaced by classTransformer metaclass (line 5).

After all code being verified, and assigned a new metaclass, the list is returned (line 6). The returned object (mod) contains all loaded code, that will later be used to create the actual objects (variables, classes or methods).

4.4.2 Class transformation

As stated earlier, the transformation of the *Original Classes* is performed by a metaclass. In Python, both classes and metaclasses are first-class objects, and, as such, have the ordinary class methods: __new__ where the instances are actually allocated, and __init__ used to initialize the state of its instances.

As it is necessary to intervene on the actual creation of the classes, the *classTransformer* metaclass must inherit from type (to become a metaclass) and have the __new__ method defined (to intercept actual classes creation).

The actual implementation of the *classTransformer* metaclass is shown in Listing 4.5.

```
1 class classTransformer(type):
2     def __new__(cls, name, bases, dct):
3         oldclass = newClass(name+"old", cls)
4         proxyclass = newClass(name,transformedClass)
5         proxyclass.originalClass = oldclass
6         return proxyclass
```

Listing 4.5: Mercury metaclass pseudo-code

In line 3, a copy of the original class is created but assigned a different name. This is necessary, since later, instances of this class will be created. From this point forward, the original class can be accessed globally by its new name (the original name followed by the old suffix, or locally through the oldclass variable.

In the following lines, a copy of the transformedClass pre-existent class is made (line 4), this new class stores in it the *Original Class* (line 5). The __new__ method concludes returning a reference to a copy of the *transformedClass* class.

In the original code, although the original class was supposed to be created, the retuned class is a copy of the *transformedClass*. Now, whenever the programmer wants to create an instance of the original class, the object creation will be handled by the newly created copy of the *transformedClass* class.

4.4.3 Object creation

After all code was loaded and all classes created, the application state consists of unmodified classes, the original classes (with different names) and copies of the *transfomedClass* replacing the original classes.

The *transformedClass* is an object factory, as actually no instances of this class will ever exist. When trying to create an instance of this class, the returned objects will belong to one of the *Adaptation Classes*. In this case, its is also the method `__new__` (shown in Listing 4.6) that is executed.

```
1 class transformedClass (object):
2    def __new__ (cls, *args):
3       adapterClass = selectAdapterClass()
4       proxyObj =
5           adapterClass(cls.originalClass, paralelClasses[originalClassName], *args)
6       return proxyObj
```

Listing 4.6: Mercury *transformedClass* object factory

On line 3, the most suitable *Adapter* is selected. This has to be done to take into account the various resources available and optimize the allocation of available resources (such as processors, memory), taking into account the object's computational requirements. A possible mechanism and algebra to evaluate the available resources will be presented in Chapter 7. There is also a need of different Adapters depending on the available compuitng resources due to the different mechanisms to create tasks (such as local vs remote threads, mutiple task creation API).

On the next lines (4 and 5), an instance of the selected *Adaptation Class* is created. The constructor receives as parameters the *Original Class*, the list of *Parallel Methods* (those from the *Original Class* that can be executed concurrently) and the original arguments that are to be passed to the *Original Class* constructor.

This method returns a proxy class (instance of *Adaptation Class*), that will become responsible for the creation of the actual *Original Objects* and forwarding of all method calls to them.

During normal program operation, different instances of the same *Adaptation Class* exist concurrently, each one responsible for a synchronization and parellelization of the execution of single *Original object*.

4.4.4 Adapter implementation

Each adapter class is responsible for handling different computational resources (such as shared memory multiprocessors or clusters). As such, each *Adapter class* is responsible for a series of management activities:

- selection of the execution target;
- creation of *Original Objects* on the target platform;
- interception and proxying of the method invocation;
- synchronization of invoked methods.

The fundamental *Adaptation Class* is the one that takes advantage of local multiprocessors/multicores to allow the efficient concurrent execution of several *Original Classes*, each on a differnte thread: a *ThreadAdapter*. Besides allowing the concurrent execution of parallel methods, this *Adaptation Class* must also block other method invocations until parallel methods terminate.

Other *Adaptation Classes* can extend this class to accomplish the same objectives, but creating the *Original Objects* on different computing infrastructures.

Listing 4.7 presents the generic code common to any *Adapter*. This listing shows the constructor (`__init__`) of the *Adapter Object* and the method calls interception code (`__getattr__`).

```
1  class ThreadAdapter:
2     _lock = threading.Lock()
3     def __init__(self, originalClass, _paralelMethods, *args):
4        self._proxiedObject=originalClass(*args)
5        self._paralelMethods = _paralelMethods
6     def __getattr__(self, attr):
7        if type(attr) is MethodType:
8           self._name.append(attr)
9           if attr in self._paralelMethods:
10             return self.__invokeParalel__
11          else:
12             return self.__invokeSerial__
13       else:
14          ThreadAdapter._lock.acquire();
15          ret = getattr(self._proxiedObject, attr)
16          ThreadAdapter._lock.release();
17          return ret
18    ...
```

Listing 4.7: Mercury *Thread Adapter* - initialization and method selection

Since each *parallel method* is to be executed on a different thread, it is necessary to guarantee that other methods (executing on the main thread) do not execute until all *parallel methods* have finished. This barrier is accomplished using a single lock that is stored in a class attribute (created on line 2, at class loading and creation).

When creating instances of this class it is necessary to generate the actual instance on the **Original Class** that contains all functional code. These objects are created in line 4 and will be responsible for actually executing all functional code, althoug all interactions are perfomed via the **Adapter**. In this example the *proxiedObject* is created locally.

The *_parallelmethods* attribute is necessary, in order to store which methods should be executed in separate private thread.

From the moment of their creation, instances of this *Adapter* replace and encapsulate the *original objects*. Instances of the *Adapters* will not have any functional code, delegating its responsibility to the actual *Original objects*.

In order to parallelize *original objects* method calls and synchronize its execution, it is necessary for the *adapter objects* to intercept those calls and inject the necessary code. The method `__gettattr__` is allays and transparently executed when invoking methods or accessing attributes from this class's instances. Since this method selects and returns the methods or attributes to be later executed or accessed, it is necessary to modify this method to perform the required interception.

If the access is to an attribute (the condition in line 7 is false), the access is forwarded to the original object: the code in line 15 retrieves the value from the *original object* to be returned right after (line 17). The access to the *original object* is guarded with a lock, to guarantee that this access is only performed after the completion of all the *parallel methods*.

If it is a method call, two cases are possible: execution of a parallelizable method or not. This is evaluated in line 8. In both cases, the returned method object does not belong to the **Original Object** but to the **Adapter** (lines 10 and 11). These methods still belong to the *Adapter object* and invoke the original methods along with all the complementary code: thread creation and synchronization. Before returning the references to these methods (`__invokeSerial__` or `__invokeParallel__`), the name of the called method is pushed to a stack (line 8).

The code of these methods is presented in Listing 4.8, along with the actual execution of the original methods.

```
class ThreadAdapter:
    ...
    def __invokeSerial__(self, *vargs):
        methodName = self._name.pop()
        meth = getattr(self._proxiedObject, methodName)
        ThreadAdapter._lock.acquire();
        ret = meth(*vargs)
        ThreadAdapter._lock.release()
        return ret
    def __invokeParallel__(self, *vargs):
        if ThreadAdapter.nThread == 0:
```

```
12            ThreadAdapter._lock.acquire()
13        ThreadAdapter.nThread ++
14        self._thread = threading.Thread(self.__paralelCode__, vargs)
15        self._thread.start()
16    def __parallelCode__(self, *args):
17        methodName = self._name.pop()
18        meth = getattr(self._proxiedObject, methodName)
19        meth(*vargs)
20        ThreadAdapter.nThreads —
21        if ThreadAdapter.nThread == 0:
22            ThreadAdapter._lock.release()
```

Listing 4.8: Mercury Thread Adapter - method execution

After `__getattr__` method returning, the program execution continues inside either `__invokeSerial__` or `__invokeParallel__` methods.

The `__invokeSerial__` method first retrieves the name of the method being called (line 4) and then obtains the actual method to be executed from the **Original Object** (line 5). In order to guarantee that no there is no other concurrent method invocation, the invocation of the actual method (line 7) is guarded by a lock.

Since all parallel methods must execute in different threads, it is the responsibility of the `__invokeParallel__` method to to start them. This is performed in lines 14 and 15.

In order to guarantee correct method calls synchronization, it is necessary to acquire the lock before starting the thread (lines 11..13) (first lines of the `__invokeParallel__` methods) and release it before the thread completion (last lines of the `__parallelCode__` methods).

The thread code can not be solely the *original object* method, due to the need to insert the correct synchronization code. The method `__parallelCode__` executes on its own thread, gets the name of the method (line 17), retrieves the actual method (line 18) and executes it (line 19). The code in lines 20 to 22 releases the lock.

The lock initialized when creating the *Adapter* class is used to guarantee that only *parallel methods* execute concurrently. To accomplish that, this lock is acquired and released in the following events:

- acquired before accessing attributes and released right after (lines 14 and 15 in Listing 4.7
- acquired before executing non parallel methods and released right after (lines 6 and 8 in Listing 4.8
- aquired before the execution of the first parallel method (lines 11 to 13 in Listing 4.8
- released after the execution of the last parallel method (lines 20 to 22 in Listing 4.8

The various acquires and releases of the lock guarantee that the resulting execution of a transformed objects is similar to the unmodified serial version execution.

For simplicity, the presented code lacks mechanisms for efficient functioning when there are less processors than *parallel objects*. If more threads are created than the number of processors, their concurrent execution can be penalized. When creating the *Thread Adapter*, it is possible to state how many processors are available. Later, it is possible to guard the execution of *parallel methods* with a semaphore, initialized with the number of processors or cores. The semaphore is acquired before starting the execution threads (leading to a optimal number of simultaneous threads) and released at their completion.

4.4.4.1 Adapter extensions

The development of **Adapters** to allow the use of other parallel execution architectures can be made taking as base the **Thread Adapter**. The synchronization mechanisms are the same, only differing in the **Original Object** creation mechanism.

In order to allow the creation and execution of objects in different computers, the work previously developed and described in [SF04] was used. The described system allows the transparent creation of objects in remote hosts, resorting to similar code transformation mechanisms as the ones presented early in this chapter, but always creating objects in remote hosts.

The new *Adapter class* only differs in one line of code and has one more method. When creating the actual *Original object* (line 4 in Listing 4.7) the *Cluster Adapter* invokes the method that creates the object in a remote host, named `newRemoteObject` and shown in Listing 4.9.

```
1 class ClusterAdapter:
2     ...
3  def newRemoteObject (self, originalClass, *args):
4      URI = getURI()
5      proxy = Pyro.core.getAttrProxyForURI(URI)
6      cdURI = proxy.createObject(package(originalClass), str(originalClass), args)
7      proxyObj = Pyro.core.getProxyForURI(cdURI)
8      return cdProxy
```

Listing 4.9: Mercury Cluster Adapter - remote object creation

The PYRO [Jon11] (PYthon Remote Objects) system was used in order to allow an easy interaction with remote servers and creation of the remote objects. Each of the remote computers executes a server that accepts remote method invocations, creates objects and exposes them as new services.

In order to create an *original object* on a remote host, the adapter just finds a suitable object creation server (identified by an URI [BFM05]), creates a proxy for it and invokes the remote method (lines 4 to 6). The remote method is called along with the name of the package containg the *original class*, the name of class and the constructor arguments.

The `createObject` method (executed on the remote host) creates an *original object* and returns the URI of the newly created object. The proxy created in line 7 will intercept all method calls and redirect them to the *original object* living in a remote computer.

Part of the code of the `ObjectGenerator`, running in the remote computers and responsible for the actual creation of the *original objects*, is presented in Listing 4.10.

```
1 class objectGenerator(Pyro.core.ObjBase):
2 ...
3     def createObject(self, classPackage, className, *args):
4         import(classPackage)
5         exec("class␣"+className+"("+classPackage+"."+className+",␣Pyro.core.ObjBase):pass")
6         newObject = eval(className+str(args[0]))
7         Pyro.core.ObjBase.__init__(newObject)
8         uri = self.daemon.connect(newObject)
9         return uri
```

Listing 4.10: Mercury Cluster Adapter - remote object creator server

The first step is the import of the *Original class* code. When detecting that the imported code is not locally available, PYRO automatically downloads it from the host where the remote call was initiated.

The next step is the creation of a new class that inherits from *Original class* and from `Pyro.core.ObjBase` (line 5). This will allow the creation of objects whose methods can be remotely invoked.

The following instructions (lines 6 to 7) create the actual object and execute the PYRO initialization. Finally a daemon is created and its URI returned.

4.4.5 Execution environment

The Mercury code should be injected into the Python virtual machine before the loading of the application. As a last resort, the source code of the virtual machine could have to be changed. The code manipulations performed by Mercury do not oblige that.

The transformation code just needs to be loaded before the application code. For instance if, in order to start one application, the user would invoke it by writing the command `python app.py`, to take advantage of Mercury the command would be replace by `python mercury.py app.py`. Alternatively the Python virtual machine configuration file could be edited, so that with every execution the **mercury.py** would be loaded.

This **mercury.py** file contains the code responsible for loading the configuration files, loading and creating the metaclasses, changing the include function, and executing the modified code as described in this section.

4.5 Evaluation

The evaluation of Mercury is twofold: i) quantitative, where the overhead incurred by using Mercury is shown, and ii) functional, developing sample applications and executing on different environments.

The first evaluation consisted on executing the function integration code presented in Listings 4.11 and 4.12, sequentially (the original and modified versions) and within Mercury (the modified version).

```
sum = 0.0;
i = 0
print i;
while i < trials:
    xr = random.uniform(x1, x2)
    sum = sum + xr * xr - 3 * xr
    i +=1
```

Listing 4.11: Original $x^2 - 3x$ integration code

```
sum = 0.0;
nTasks = trials/trialsPerObj
for trial in range(nTasks):
    objs[trial] = parallelObj(x1, x2)
    objs[trial].execCode(trialsPerObj)
for trial in range(nTasks):
    sum += objs[trial].getResult()
```

Listing 4.12: Modified $x^2 - 3x$ integration code

The modified version presents and outer extra loop, used to partition data, and allow the same number of `parallelObjects` and number of used threads. The method (`execCode` is equivalent to the loop in the original code.

This evaluation was performed on 3.06GHs Intel(R) Core(TM) 2 Duo processor running Mac OS X. In all experiments the total number of iteration (trials) was 50 million, but on the modified version multiple quantities of `parallelObjects` were used. The results can be observed in Figure 4.4.

The time to execute the serial original version of the code (about 55 seconds)

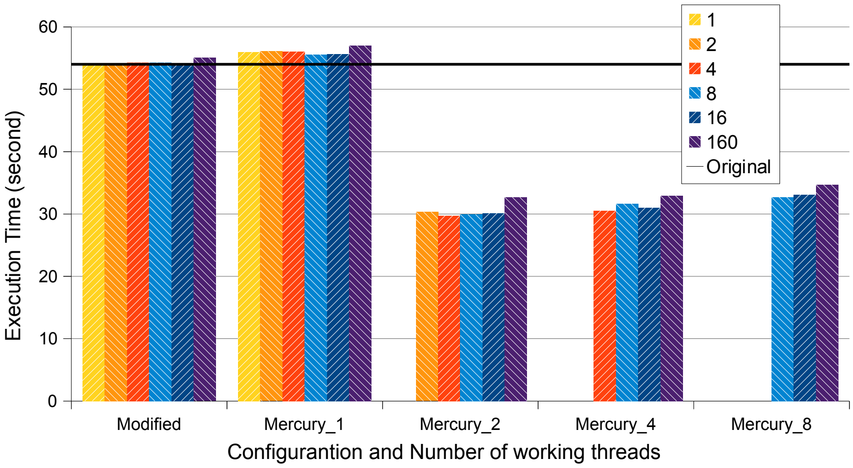


Figure 4.4: Mercury based function integration parallelization execution times

is represented as a black line, while each of the coloured columns present the execution times with different number of threads (x axis) and different number of `parallelObjects` (from 1 to 160).

When executing the modified version outside Mercury (**Modified**), only with 160 `parallelObjects` the increase on the execution time is noticeable. This is due to the fact that more instructions are actually executed: more objects are created, and more calls (to `execCode` and`getResult`) are performed.

When executing the code within Mercury with only one available thread (set of columns labelled **Mercury_1**), there is an added overhead. The reason is in the code transformation and objects synchronization: the code had to be rewritten, new objects were created, new method calls invoked, and `getresult` methods had to be synchronized. Nonetheless the overhead is minimum. When 160 `parallelObjects` were created, the total overhead was about 3 seconds, rendering and overhead of about 20ms per `parallelObjects`.

With two available threads (**Mercury_2**), the gains are, as expected, close to optimal. With the number of threads higher than the number of processing core there is a degradation of the execution times. In this case the contention and synchronization on the access to the CPU adds some overhead.

Using Mercury, it was possible to parallelize a Monte-Carlo computation to integrate one function. Instead of treating each random value in a serial way, each task was responsible for obtaining part of the solution. In order to use Mercury, the definition of a class was necessary. Although a simpler serial version could be developed (as presented in Listing 4.11) Mercury could not parallelize it.

In order to evaluate the modifications necessary to parallelize a pre-existent application, a simple ray-tracer was executed inside Mercury. The original ray tracer (yopypar[2]) was executed inside Mercury with minimal code modification.

The original application (with 390 source code lines, 9 classes and 3 functions) suffered the following modifications:

- modification of 2 functions' headers, since these functions access a global variable and no shared memory space exist among tasks;
- modification of previous functions calls, to receive as argument the removed global variable;
- addiction of a *parallel class*, responsible for the rendering of part of the image;
- modification of the main function, to allow the creation of the *parallel objects*, invocation of the parallel code and retrieving of results.

The total number of modified code lines was 9 plus the transformation of the `main` function similar to the transformation presented in Listings 4.11 and 4.12.

With the new modified code, it was possible to execute it in a correct sequential way, and take advantage of the multiple cores existing in the test platform. The same code base is used to execute the sequential version, the one using local cores, and another one taking advantage of remote computers.

Using the *ClusterAdapter* earlier described, it was possible to execute the presented ray-tracer in multiple processors distributed in various hosts. The results of these executions (sequential, on local cores and on remote computers) is shown in Figure 4.5.

The first executions (**Original**, **Modified**, **Mercury_1** and **Mercury_2**) were all executed on the computer with a 3.06GHs Intel(R) Core(TM) 2 Duo processor

[2] http://code.google.com/p/shedskin/source/browse/trunk/examples/yopyra.py?r=1320

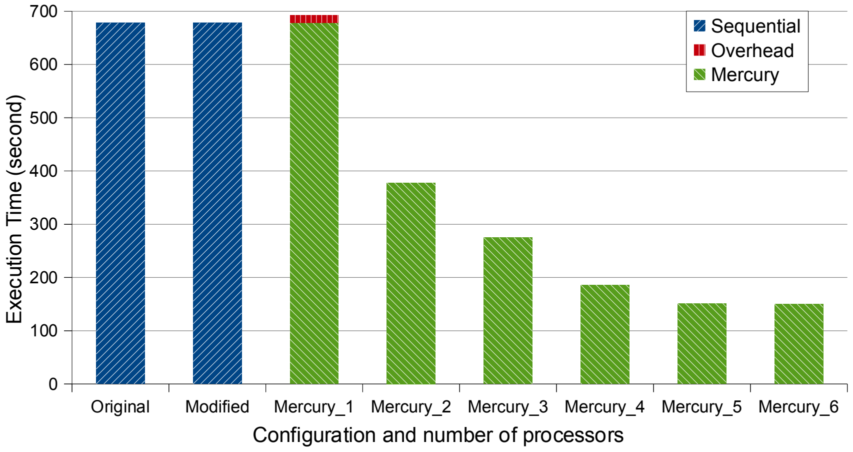


Figure 4.5: Mercury based ray tracing application parallelization execution times

and running Mac OS X, while the last four also resorted to a computer having a 2.40GHz Intel(R) Core(TM)2 Quad CPU, running Ubunt 9.04.

The overhead for using Mercury is low: the execution time difference between the **Original** and **Modified** versions is imperceptible, while the overhead for executing the modified version within Mercury (**Mercury_1**) is only 15 seconds on a job longer than 11 minutes.

With one thread, the overhead is due to the amount of data to transmit at task creation (4 Kbyte) and at task completion (700 Kbyte). These values correspond to the transmission of the scene file and reception of the resulting image.

With more processors, the speedups attained are in line with the expected ones.

4.6 Conclusion

With Mercury it is now possible to automatically and transparently produce parallel Bag-of-Tasks. The user needs only to program a serial version of its job (following simple restrictions), define what objects and methods should be executed concurrently and execute that application within Mercury.

The developed application must have objects that are responsible for the execution of lengthy tasks. The serial version must have one initialization part (where objects are created and data partitioned), one or more execution parts (where interactively and independently each object executes its task) and a data recollection phase. This structure matches most of Bag-of-Tasks problems. Mercury will transparently execute each of the parallel methods on a different processor or computer.

From the experiments made, it is possible to conclude that Mercury can handle most Bag-of-Tasks. The presented experiments shown the parallelization of a simple implementation of a Monte-Carlo simulation as well as the execution of an externally developed ray-tracer. The overhead incurred by Mercury is low and easily overdue by the gains on programming ease and speedups.

It was possible to parallelize applications allowing the execution of concurrent threads on two different environments: i) multicore computers, executing each concurrent thread on a different core, and ii) cluster of computers, using an object distribution systems to leverage different computers scattered on a local network.

5 Off the Shelf Distributed Computing

With suitable tools, the Internet offers close to unlimited processing power usable to solve users' problems. Chapter 2 presented an overview of the available solutions to access Internet scattered computational resources, along with an assessment of their real use.

From the presented evaluation, it is possible to conclude that currently, cycle sharing over the Internet is a one-way deal. Computer owners only have one role in the process: to donate their computers' idle time. This is a consequence of being difficult for an ordinary user to install the required infrastructure, develop the processing applications, or even gather enough computer cycle donors.

These three reasons must be overcome in order to allow a system to be fully successful.

This chapter presents mechanisms to allow users to create and submit jobs that are executed on remote computers taking advantage of donated cycles.

The best class of jobs to be executed in a Distributed Computing infrastructure is Bag-of-tasks, since it does not require communication between intervening computers, nor complex scheduling algorithms.

To allow an easy task development, these jobs must be processed by commonly available software (e.g. programming language interpreters or virtual machines, statistical software, signal/video processing tools, ray-tracers) that is installed on the remote donating computers, resorting to task creation mechanisms similar to those presented in Chapter 3.

In order to submit their jobs, users only have to provide the input files, select the processing application and define the command line to provide to that application. Later, users of the same software packages will contact the server, receive a set of jobs, and process them using the already installed commodity application. These users can later take advantage of other people's computer cycles.

It is still necessary to gather enough donors in order to obtain representative gains. This is accomplished by allowing users to donate cycles to causes or projects they are aware of: the donors select the applications that are authorized to execute on their computers.

5.1 Introduction

Chapter 2 presented currently available systems allowing the use of Internet scattered resources. Of these, the most successfully and widely deployed infrastructure is BOINC [AF06] While lacking features that would allow an efficient use by a broader user base, of the available systems, it is the one that more closely matches the requirements presented earlier in this document.

BOINC is a successful platform for distribution of parallel jobs to be executed on remote computers. Researchers create and publicize projects that require solving a complex problem, by installing a BOINC server, developing the data processing applications and creating the data sets to be processed. Later, users willing to donate their personal computers' idle cycles register themselves and start processing data with the applications downloaded from the server and previously developed for such projects.

This greatly limits the scope of users that can create projects to be remotely executed. Projects must have large visibility in order to attract enough donors and be composed of hundreds of individual tasks. Project creators must also have extensive knowledge on C++ or Fortran programming.

Projects from users that do not satisfy the previous characteristics can not take advantage of available remote cycles. Even if the user has enough programming knowledge to create a project, if the project is of short length, or not capable of attracting enough donors, the gains will be low.

Some computer user communities can take advantage of remote idle cycles to speed their jobs, but do not have the skills to efficiently use BOINC. These users range from hobbyists or designers, that use ray tracing software to render movies or complex images, to researchers that use statistical software packages to process very large data sets. Researchers who develop applications in Java or Python can also take advantage of remote cycles to speed up their jobs.

To allow these new users to create jobs in a cycle-sharing system, they should be allowed to use the applications or programming languages they are literate on, and there should be enough cycle donors to speed even short spanning jobs.

The use of commodity software as a job execution environment reduces the code development cost, as users do not have to learn a new programming language (C, or Java), being allowed to use the most efficient of the existing tools for the task. It would be infeasible to install all these commodity applications remotely, but allowing the use of widely available software would increase the potential number of users that already had them installed. Furthermore, the use of previously installed software, creates a sense of community among users, as each user is willing to donate his computer's idle cycles to solve a job similar to the ones he will later submit.

The sense of usefulness that comes from the use of the same software as other users may increase the participation on such projects. After donating cycles, a computer owner will also take advantage of remote cycles to speed his jobs. This compels users to provide more and more cycles to others.

This can be promoted if, instead of just donating them, users can actually lend them and expect to employ them later in return. When needed, the credits received by executing tasks from others, will be exchanged by processing time on remote computers. This new relationship within the system will increase the number of users and the amount of time each user is able to share resources with remote users.

The remaining of this chapter presents a set of BOINC extensions that allow efficient execution of user submitted jobs and allowing any user to have two complementary roles: owner of the jobs that are executed on remote computers and owner of the computers where jobs will be executed. Any user that lends his idle cycles to the execution of other users' jobs, will also be able to take advantage of their remote computing resources.

In order to accomplish this, it was necessary to modify both the BOINC client and server software, and developed a custom BOINC project application. The data processing code used by these jobs comprises commodity applications that are installed in the remote computers, only after their owners have allowed their use.

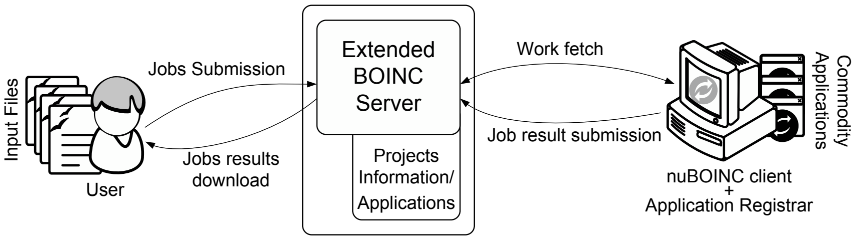


Figure 5.1: Extended BOINC (nuBOINC) usage

As shown in Figure 5.1 there are two roles while interacting with the extended BOINC server: BOINC clients that execute the jobs and users that submit them. To submit and create new jobs, users must: i) select the commodity application that should be used to process the data, ii) provide the input files (data or code, as scripts), and iii) define the number of jobs to create, the name of the output files and the arguments that should be used to invoke the commodity application. Then, the server waits for requests from BOINC clients to distribute each job. When contacting the server, the extended BOINC client sends the identifiers of the commodity applications the client owner has already installed. The server then selects the jobs to send according to this information. After receiving each job information (input files and arguments), the BOINC client invokes the correct commodity application to process the input files. After each job completion, the

BOINC client submits the output to the server.

The next section presents existing cycle-sharing platforms and how they relate to the proposed solutions, following with the presentation of how the job submission is to be made (section 5.3). Section 5.4 will present the extensions made to BOINC, and their evaluation is presented in Section 5.5. Finally, the conclusions are presented.

5.2 Related work

BOINC is the best known platform for the creation and execution of distributed computing projects, providing all the data storage, communication and client management infrastructure. The project manager has to develop a C++ or Fortran application that will be executed on the client computers to process data. Even though it is straightforward to install a hosting infrastructure, two issues arise: it is necessary knowledge on C++ or Fortran to program the applications and, in order to have some speedup gains, it is necessary to publicize the project in order to attract clients.

BOINC wrappers [oC07] allow the use of legacy applications as processing code in a BOINC project. Project developers implement a simple wrapper and define the configuration file where it is stated how the legacy application will be executed. Both the wrapper and the legacy application are downloaded by the client, and when executed, the wrapper only invokes the legacy application. Even with this solution, short length projects, or without the ability to raise cycle donors, can not take advantage of BOINC.

XtremWeb [GNFC00] and the Leiden Grid Infrastructure [Som07] (LGI) are distributed computing projects that allow registered users to submit their jobs, as opposed to plain BOINC installations where only the system administrator creates jobs. These systems allow the execution of jobs on institutional clusters and on BOINC based Internet scattered clients.

In XtremWeb users provide the input files and define the command line arguments used to invoke the application, while in LGI users are required to define a configuration file that is used when submitting work. In neither case, users are allowed to install a new data processing application to solve their problems.

The first project to use LGI was Leiden Classical [Lei]. Leiden classic is a regular BOINC project (with custom built processing code, but featuring a task creation user interface). Users can submit a single input file (with the simulated experiment configuration), that is processed in a donor computer.

Existing P2P cycle sharing infrastructures still do not allow an easy and efficient job submission by users, as shown in Chapter 2. For instance, JXTA-JNGI [VNRS02] provides an API for the development of distributed cycle sharing systems. It is possible for any one to develop of a proprietary cycle sharing system, where the owner defines all the code and data communication, but to make it efficient over the Internet it is necessary to publicize and gather cycle donors.

Currently, other generic infrastructures (such as POPCORN [NLRC98] or P2P-G2 [MK05]) exist but they still require the explicit development of the code to be remotely executed. These solutions free the user from programming the distribution and communication protocols, but still require writing (in Java or C#) the code to be executed remotely. Furthermore, these solutions would still require a lot of publicity to get enough donors.

The use of commodity data processing applications as a remote processing environment has also been proposed in NetSolve [SYAD05], but it requires the commodity application (e.g. Mathematica, Matlab, Octave) to be extended in order to allow work distribution. Furthermore, users should also adapt their scripts or applications to distribute lengthy functions.

The required user interface is closer to Nimrod [ASGH95], in the way the data distribution is defined: the user defines the input files, the type of parameters and how they vary. Nimrod then generates all parameter combinations and assigns each parameter combination to a task. Even though Nimrod helps with the combination of all parameters, the user must still have some programming knowledge, because the processing application must be coded and the data type of each parameter must be defined.

The mechanisms to submit work and define tasks' input and output should be close to those presented in Chapter 3.

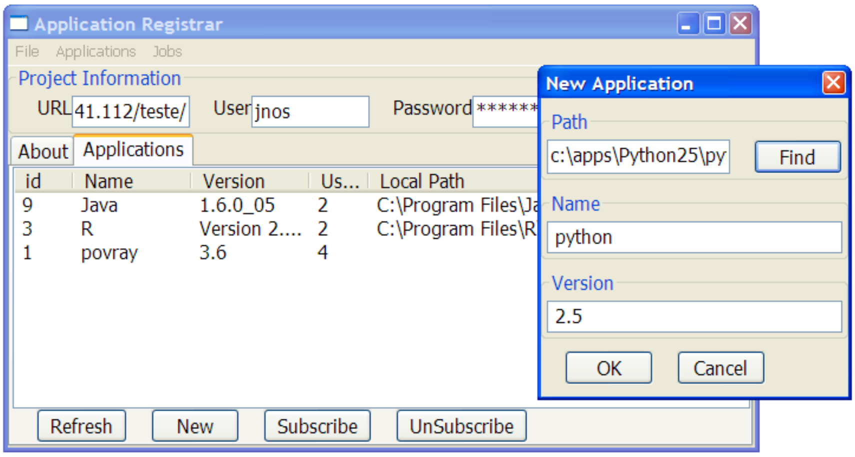
5.3 Usage

With the modifications to BOINC presented in this chapter, users are now capable of executing tasks, but also to create their own tasks.

5.3.1 Cycles Donation

To share his computer idle processing cycles, a user only has to download the *nuBOINC client* and the commodity *application registrar*. The first step is similar to the donation of cycles to any regular BOINC project: the user downloads the regular BOINC client, creates an account on a nuBOINC server, and declares willingness to donate cycles to nuBOINC projects.

After installing the BOINC client, it is necessary to define which commodity applications are authorized to be executed. This step also defines what class of work is to be executed.



URL 41.112/teste/ User jnos Password ******

id	Name	Version	Us...	Local Path
9	Java	1.6.0_05	2	C:\Program Files\Ja
3	R	Version 2....	2	C:\Program Files\R
1	povray	3.6	4	

Path: c:\apps\Python25\py
Name: python
Version: 2.5

Figure 5.2: nuBOINC Application Registrar user interface

Figure 5.2 shows the user interface. Both the user interface and the registration steps are similar to those presented in Chapter 3, Figure 3.3.

Before defining what applications are available, the user must connect to the nuBOINC server, stating its URL, and inserting the *user name* and *password*. A list of available applications is fetched from the nuBOINC server and displayed. This list contains all commodity applications that other users have registered.

The user can either *Subscribe* to a previously registered application or define a new one. In either case, the user selects the executable (that can be executed from the command line) to be used to execute tasks. This file location can be assigned to a previously registered application, such as **povray** in the previous figure, or to a new application.

When assigning an executable to a new application, the user must define a well known name and insert its version number, as shown in the *New Application* dialogue box.

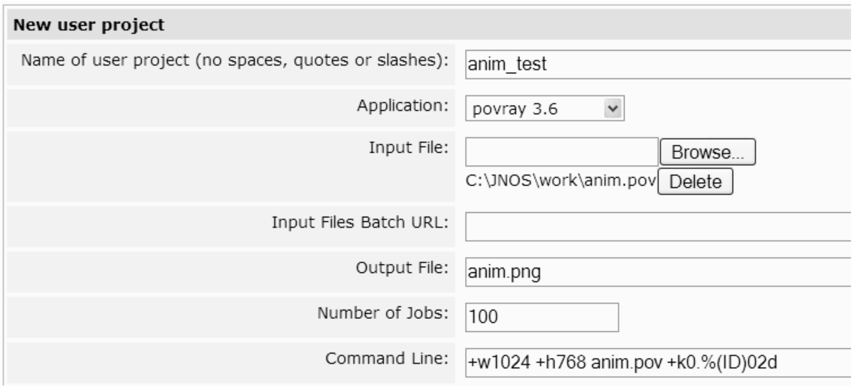
From this moment on, the computer where the *Application Registrar* was executed is authorized and configured to execute jobs requiring the defined applications.

5.3.2 Job Creation

In order to create a job comprised of multiple tasks it is necessary to use a simple user interface. Chapter 3 describes such user interface requirements (Section 3.3.2) and a possible user interface implementation (Section 3.5).

Although the user interface presented in Chapter 3 could have been integrated with BOINC, for the sake of simplicity, a simple prototypical web form was developed (Figure 5.3).

Here, the user connects to the server hosting nuBOINC, identifies himself and is presented with the previous user interface. The mechanisms to create a job and its tasks, and to define input/output file and parameters is similar to those presented in Chapter 3.



New user project

Name of user project (no spaces, quotes or slashes):	anim_test
Application:	povray 3.6
Input File:	C:\JNOS\work\anim.pov
Input Files Batch URL:	
Output File:	anim.png
Number of Jobs:	100
Command Line:	+w1024 +h768 anim.pov +k0.%(ID)02d

Figure 5.3: nuBOINC User project submission interface

Nonetheless, the user must name its project/jobs and select the application to process the data. The list of available commodity application only includes those previously registered (using the steps described in Section 5.3.1) by the user creating a job.

In the example, the user wants to process a file (`anim.pov`) with the POVray ray tracer and generate a movie with 100 frames. For each frame (corresponding to one time instant), one different image will be generated. One hundred jobs will be created and, on each one, the POVray executable will be invoked with the following command line: `+w1024 +h768 anim.pov +k0.%(ID)02d`; `anim.pov` is the input file and `%(ID)02d` will be replaced with the job identifier (00, 01, . . . , 98, 99).

5.4 BOINC Extensions

In order to handle these new user interactions, the BOINC server and client must suffer some modifications. Of these modifications, the one with more impact is the possibility for any user to create jobs: a suitable user interface is needed, and the creation and storage of tasks is different from the original BOINC.

The architecture of the developed infrastructure (shown in Figure 5.4) closely matches the one of a regular BOINC installation [AKW05]. This figure also shows the ordered interactions between the different components (circled numbers).

The *Application registrar* and *RPC Interface* components do not exist on regular BOINC installations, while all others maintain the same functionality. The *web interface* was modified to allow the creation of tasks and the retrieval of results.

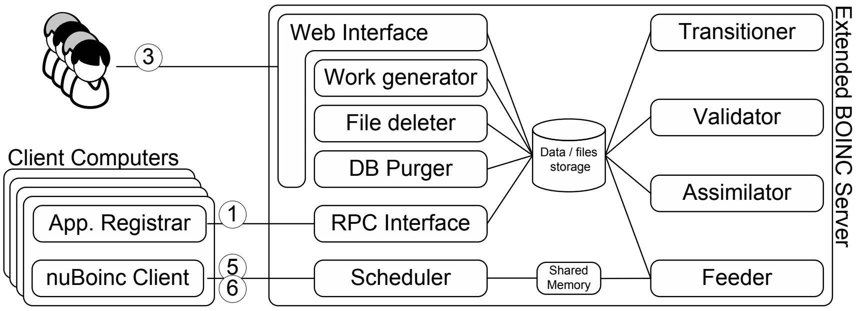


Figure 5.4: nuBoinc architecture: Detailed server view

In order to allow the execution of user submitted jobs, modifications to BOINC (server and client) were required.

To allow users to register the applications (those allowed to be executed on the donor computer), an auxiliary application (*Application Registrar*) was developed. This application interacts with the server by means of XML-RPC calls. On the server side, these calls are handled by a specially developed service.

The original BOINC *web interface* was modified in order to allow job submission and result retrieval. The modules responsible for job creation and deletion (*Work Generator*, *File Deleter* and *DB Purger*) are now invoked by the users by means of the *web interface*. The modules that verify the validity and correction of job execution (*Validator*), and that replicate or change job state in case of erroneous or successful execution (*Transitioner* and *Validator*), were not modified. The *feeder* and *scheduler* modules (responsible for delivering work to clients) were modified in order to allow the matching between the required *commodity applications*, and the those installed on the remote computers.

With respect to job information organization within the BOINC server, only one modification was made. In regular BOINC installations, work is grouped into projects, and all jobs from the same project are executed by the same application. With the presented extensions, all user submitted jobs are processed within the same BOINC project (*nuBOINC project*) but belong to different *user projects*: sets of jobs, submitted by one user, that are to be processed by a given commodity application. This new work division had to be mapped to the database structure. All other internal data organization remains unchanged. For each job there is one workunit (input files and execution parameters) and several replicas of each workunit, called results. As in any BOINC installation, results are sent to remote clients to be processed; after the processing of all results associated to a workunit, the valid or erroneous execution outcome, and output are stored in the corresponding workunit.

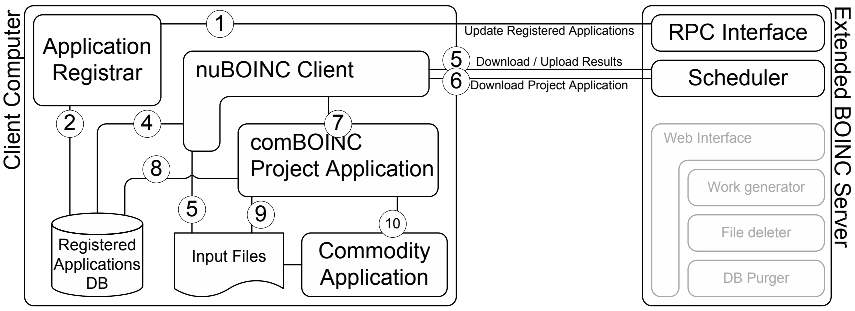


Figure 5.5: nuBoinc architecture: Detailed client view

On the client side some modifications were also made. The new client architecture is shown in Figure 5.5. Besides the inclusion of the new *Application Registrar*, the processing of the data received from the server is not performed directly by the downloaded *Project Application*, but by a previously installed *Commodity Application*.

Some of the interactions between the client computers and the BOINC server are also different:

- All *Application Registrars* must inform the server about the registered applications;
- Users can submit jobs;
- BOINC clients must inform the server about available *commodity applications*

The architectural modifications and the new interactions are presented in the remaining of this section.

5.4.1 Application registrar

After signing in at the BOINC server, the user must define what applications he allows to be used by remote users. The user must execute the supplied *application registrar* tool on every owned computer. Only after registering commodity applications with the *application registrar*, such applications are made available for execution on behalf of other users. Furthermore, each user is only allowed to create tasks that use commodity applications previously registered by him.

After filling the *nuBOINC project* and user information (Figure 5.2), this tool fetches from the BOINC server a list of applications (name and version) that other users made available, and presents it (step 1 in Figure 5.4). If any of the presented applications is the one the user wants to register, it is only necessary to insert the corresponding executable disk location (screen shot not shown). If the user wants to register a new application, he must fill in the *New Application* form. The user will supply the path of the executable, and the name and version of the application. This information will later be presented to other users wishing to register their applications.

The information about the path of the executable is stored locally on each client (step 2 in Figure 5.5), on its local *Registered Application DB*. This information will later be necessary when fetching and executing the jobs.

5.4.2 Job submission user interface

The submission of the jobs to be executed on remote computers is also made in a straightforward way. A web browser is used to supply the input files and to define each job's parameters.

After logging in (at the *ComBOINC project*), a web page is supplied with a user interface similar to the one presented in Figure 5.3. Here, the user must define the total number of jobs, the name of the output file and the command line to supply to each job. There are also means to supply the input files. The user can upload files that will be accessed by every job or provide an URL pointing to a directory containing several files. Each one of the files present in that URL will be fed to a different job. The information submitted by the browser (step 3 in Figure 5.5) is handled by a PHP script: for each defined job a workunit is created, containing the information about input files, output files, commodity application to be used and command line parameters. After the input files have been uploaded to the server and the workunit templates created, the *Work generator* is invoked in order to store relevant information in the database.

The workunit creation process is similar to the one presented developed in SPADE (Chapter 3). With suitable modifications to SPADE and nuBOINC, both could be integrated.

After the processing of a workunit, its output files are made available to the user who created it, also by means of PHP scripts. These scripts allow each user to inspect their workunit state (waiting to be processed, valid, erroneous) and to download completed results.

5.4.3 Database Tables

The relational model of the new information to be stored is presented in Figure 5.6. Shaded entities were already present in the original data model.

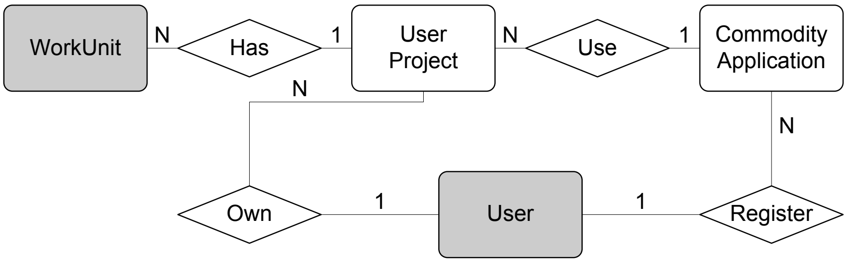


Figure 5.6: BOINC database aditional information

In order to accommodate the new information related to the registered applications and user submitted jobs, it was necessary to modify the original BOINC database.

The fundamental entity is *user Project*. This entity is related to the workunits that compose a project, to the application used to process its data, and to the owner.

The *Commodity Application* entity and *Register* relationship were added to accommodate the names and versions of the commodity applications available on remote hosts. This way, it is possible to know and store what applications are needed by certain Projects (*Use* relationship) and what application a user has registered on the client computers.

In order to restrict user access to workunits, results and their execution information, it was also necessary to store the workunits ownership information on the database.

5.4.4 nuBOINC Client

Besides all the information regular BOINC clients send to the server, the modified client (nuBOINC client) also sends the identification of the commodity applications allowed to be used. This information is required for the selection of the suitable jobs to be executed on that client.

Before sending any request to any BOINC server, the *nuBOINC client* tries to find information about the previously registered commodity applications (step 4 in Figure 5.5), stored by the *application registrar* on the local *Registered Applications DB*. If the information regarding registered *commodity applications* is found, a list of those applications is sent to the server along with a regular work request. The answer to this request contains the input files to be processed (step 5).

Apart from the *commodity applications* list attached to the work requests, the interaction between the nuBOINC client and the BOINC servers is unchanged. After receiving a workunit to be processed (step 5) the client verifies if the required *comBOINC application* exists. If this *comBOINC application* is not present on the client computer, it is downloaded from the server (step 6), and executes it (step 7).

The *comBOINC application* performs the following steps:

- retrieves the location of the commodity application (previously registerd by the user), step 8
- sets up the commodity application execution environments, by copying the downloaded input files, step 9
- executes the commodity application, step 10

After completion of the workunit processing, the output is sent to the server.

With the exception of the work request (that includes a list of commodity applications) all other steps are common to any other BOINC client. The similarities between the modified client and a regular one make it compatible with any regular server. This way the nuBOINC client can process work from any *nuBOINC project* or from regular BOINC projects. The registration to these projects is the same as with regular BOINC clients.

In this version of the client, the time slicing between regular jobs and *user jobs* follows the original BOINC scheduling rules.

5.4.5 Scheduler and Feeder

When contacted by an extended *nuBOINC client*, the extended BOINC server should be able to return a workunit suitable to any of the commodity applications installed on the client. In order to accomplish this, the scheduler and feeder modules had to be modified.

When requesting more jobs, the client can include the identification of the installed commodity applications. In this case, the scheduler selects workunits that can be handled by any one of those applications. This way, work is sent only to hosts that can handle it.

As shown in Figure 5.4, the *scheduler* does not interact with the database, it only accesses a shared memory segment. This shared memory segment is populated with the not yet processed results. The feeder module had to be adapted to handle the changes in the database and to communicate to the *scheduler* the *commodity application* associated with the task.

If the extended BOINC server hosts several projects, all other ordinary requests are handled in the same manner as in a regular server, guaranteeing the compatibility with all clients.

5.4.6 nuBOINC Project application

If the downloaded workunit belongs to the *nuBOINC project*, and consequently requiring the *Project Application*, the processing of the input files is different from the regular cases. In a regular BOINC project, its *project application* has all the code to process the data, while in a *nuBOINC project*, its *project application* only handles the invocation of the correct *commodity application* that is needed to process the input files.

The parameters of the *nuBOINC project application* invocation (step 7) include the identifier of the *commodity application*, the names of the input files and the parameters to be used when invoking the *commodity application*.

The *nuBOINC project application* first starts by finding the location of the required *commodity application* (step 8). Then, it creates a temporary directory, copies the input file there (step 9) and invokes the *commodity application*. Upon *commodity application* completion, the output files are copied from the temporary directory. Then, the client returns them to the server in the normal way.

5.4.7 Commodity applications

Currently, the projects allowed to be solved fit either in the parameter sweep category or in batch file processing, whose execution applications are already installed in the remote personal computers. These applications should either be parameterized through the command line or receive a script or configuration files as input. They should also easily generate output files and print error to the standard output.

The usable applications include a large set of applications: ray tracing software (POVray, YafRay), image or video processing (convert package, ffmpeg), computer algebra (Maxima, Sage), statistical software and data analysis (S-PLUS, R) or more general numerical computation applications (Matlab, Octave). Any other interpreted/managed language execution environment (Java, Python, Lisp) can also be used. All these packages are used by a large community and are available on the largest families of operating systems: Windows, Unix and its derivatives (Linux, Mac OS X).

Some of these *commodity applications* (specifically the programming languages interpreters) may be used to attack and abuse the cycles donor remote computer. To reduce those risks, the nuBOINC client and the *commodity applications* should be executed on a restricted environment: a virtual appliance, with the necessary applications, running inside a virtualization platform (e.g. Virtual PC, VMware or QEMU), with a lightweight operating system installation can be used [BOI09, Rey10, Seg10].

5.5 Evaluation

In order to evaluate the usability and performance gains, the nuBOINC modified server was deployed to allow the execution of jobs on several clients.

The first experiment performed consisted on using the povRay[Per08] ray tracer to generate an animation with 100 frames. The times for the execution of these jobs on several computers, shown in Figure 5.7, were measured with identical computers connected by a 100 Mbit/s local network. On a Pentium 4 running at 3.2GHz with Linux, each frame took between 3 and 100 seconds, giving a total rendering time of about 127 minutes.

This experiment was made on a local network but, although not representing a real usage, it allows the evaluation of maximal speedups and the adequacy

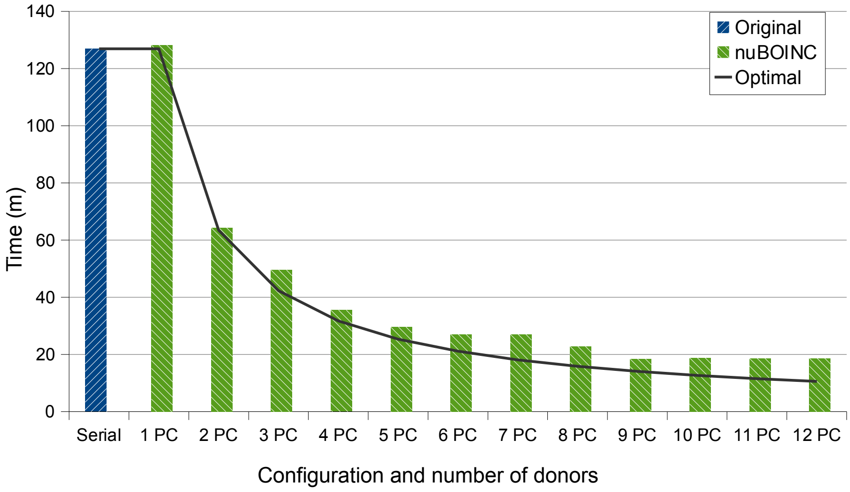


Figure 5.7: Movie rendering times

of the standard BOINC scheduling policies to environments where users are allowed to create jobs.

The graph presents, along with the time to execute the job on several computers, the time to execute the jobs serially on one computer (both locally and by means of the BOINC infrastructure).

As expected, the speedups are in line with the number of cycle donor hosts. The overhead incurred by using nuBOINC job distribution platform is minimal, only 2 minutes. This is caused by the job submission and *nuBOINC client* startup. With the participation of another host, even during a small period, this overhead is not noticeable. On an wide area network, or with larger input files, this overhead is larger, but is easily surpassed with the contribution of another user.

With high number of participant computers, the difference between the optimal and the measured times increases. Besides the expected incurred overhead, the phenomenon presented in Figure 5.8 can also explain the difference between the optimal and the measured execution times.

Figure 5.9 presents the instant each of the 100 taskds finished, when executed on 6 computers.

It is possible to observe that from minute 22 on, only 3 hosts continue processing results. This effect is a consequence of the task distribution algorithms.

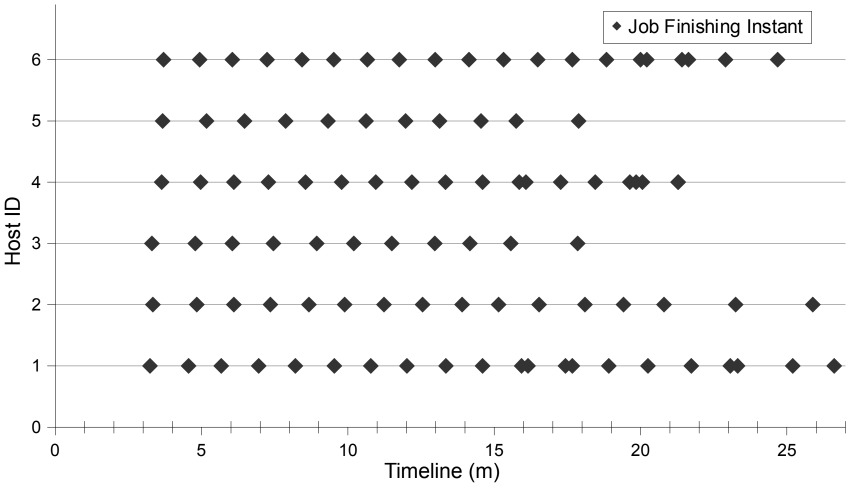


Figure 5.8: Jobs finishing instants

In order to reduce communication overhead, BOINC delivers tasks to donors in batches. When delivering first tasks, this has a beneficial effect, but when few tasks remain to be executed, this task distribution policy may starve some idle donors. As these delivered batches maintain the same size during all execution, close to the finish one computer receives a batch whose jobs could be distributed equally to other clients, leading to a unbalanced job distribution.

Even though task distribution in batches affects speedups, in some cases (for instance, 2 and 9 PCs) the speedups are close to optimal.

In a system such as nuBOINC, gains from the use of remote computers are also dependent on the amount of time a remote computer is available to process tasks and the frequency these computers contact the server to get new work.

In order to optimize network usage, between unsuccessfully work requests, BOINC clients introduce a delay (*back-off*). This delay is exponentially increased whenever a request is not fulfilled by the server, such as in the case of not existing more results to process, or in case of network failure.

The Figure 5.9 shows the influence of those delays. In each of the experiments, it was measured the delay between the job creation and the first task fetch for different client idle times. The x-axis presents the time a client was contacting the server and not receiving any tasks.

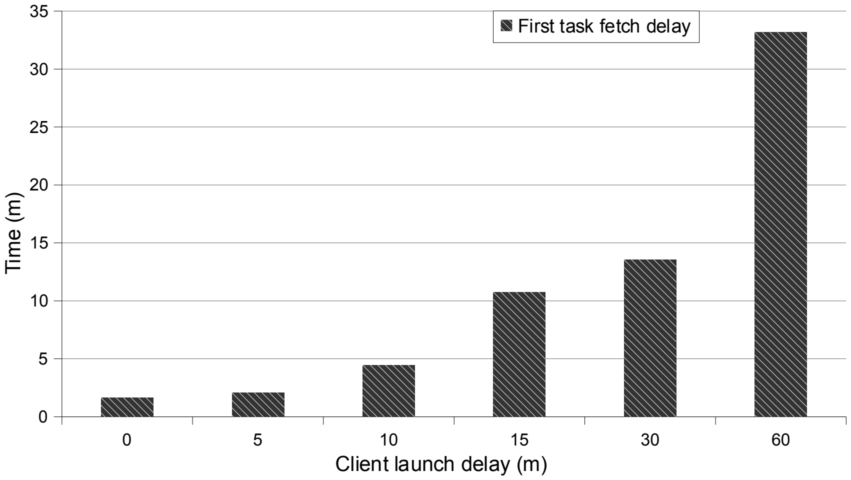


Figure 5.9: Computations start delay times

It is possible to conclude that the clients, after a high idle time, will have a large *back-off* and may be idle even when some newly created projects have data to be processed. If the projects are short termed or the *back-offs* large, these projects may not take advantage of these idle clients.

Although the deployment of jobs that used different applications (Java virtual machine and R statistical software) was experimented, the developed user interface proved limitative. Its replacement or integration with SPADE (Chapter 3) is a viable solution.

5.6 Conclusions

With the minimal modifications made to the original BOINC infrastructure it was possible to extend it into a system that allows the efficient execution of user submitted jobs on donor computers scattered over the Internet. These jobs must fit in the parameter-sweep or bag-of-tasks categories.

It was possible to integrate a job submission user interface into BOINC, allowing the definition of each job's input files and parameters without any programming knowledge. By allowing the execution of commodity applications as the data processing tools, users do not have to develop the BOINC applications to be executed on remote computers. Users only have to define the parameters or configuration files to the applications required to execute the jobs.

These commodity applications are well known to the users and are already installed in remote computers.

As these applications have a large user base, it is very likely to get high gains from the execution of some jobs on remote computers, as lots of remote users will be able to donate their CPU cycles. Organized user groups or communities can deploy a task distribution infrastructure to be used effortlessly by its members, independently of their computer expertise.

BOINC original scheduling mechanisms may not fit for the new BOINC usage. The evaluation of the scheduling mechanisms shows that the batch delivery of tasks and the back-off may reduce the obtained speedups.

Furthermore, since the owner of a client may have jobs to be executed, the scheduling algorithm should be adapted so that a client process his jobs first. To take this fact into account, and guaranteeing some fairness, the way scheduling is performed, on the client and on server should be modified.

6 Task scheduling on the cloud

The previous section presented nuBOINC, a system to allow the execution of Bags-of-tasks over the internet on donor computers, but, for the same target population, another source for computing power exists. For those without access to institutional cluster or Grid infrastructures, it is possible to create a cluster of machines on on-demand public utility computing infrastructures such as Amazon Elastic Compute Clouds [Ama11] (EC2). With the suitable middleware (SPADE, for instance) Bag-of-Tasks can be easily deployed over such infrastructures.

In contrast with Internet Distributed Computing Systems or privately owned clusters, in these new infrastructures following the utility computing paradigm, it is necessary to pay for the rented processing time, usually charged by one hour units. In order to optimize the number of allocated machines (thus reducing payment while maximizing the speedups) it is necessary to reduce the wasted idle processing time.

In Bag-of-tasks problems, the number of concurrent active tasks does not need to be fixed over time, because there is no communication among them. To execute tasks on utility computing infrastructures, the number of allocated virtual computers is relevant both for speedups and cost. If too many are created, the speedups are high but this may not be cost-effective; if too few are created, costs are low but speedups fall below expectations. Determining this number is difficult without prior knowledge regarding processing time of each task and job totals. This is usually impossible to determine reliably in advance.

This chapter presents an heuristic to optimize the number of allocated computers on utility computing infrastructures. This heuristic maximizes speedups taking into account a given budget. This heuristic was simulated and evaluated against real and theoretical workloads, w.r.t. ratios among allocated hosts, charged times, speedups and processing times. The results show that the heuristic allows speedups in line with the number of allocated computers, while being charged approximately the predefined budget and removing from the user the burden of deciding in advance how many machines to allocate.

6.1 Introduction

Grid and cluster infrastructures have become the most widely used architectures to execute lengthy and heavy computational jobs. However, as shown in Chapter 1, to take advantage of them, a user needs membership or institutional relationship with the organization, possibly virtual, controlling the computing resources. In this scenario, scientists or even home users lacking either the resources or the incentives, or the institutional links to take advantage of such infrastructures, are left without practical and viable options.

An alternative available to these users is provided by utility computing infrastructures such as Amazon Elastic Compute Clouds (EC2). These computing infrastructures provide basic mechanisms and interfaces for users to create virtual computers, where operating system, middleware, and job application code is left to be defined by users, frequently assembled in *virtual appliances* with associated system and disk images. With a careful setup, virtual computational clusters can be created easily. The creation of such machine pools is performed programmatically by means of an API, with the allocation and management of the actual physical resources completely hidden from the user. Furthermore, allied to easy creation of virtual clusters, utility computing infrastructures employ a simple subscription and payment model, with users required only to pay for the processing time used. Another benefit is the easy service subscription, where each user only needs to sign a simple contract and pay for the processing time used.

By using virtual machines with the necessary operating system and software, a computational cluster can be easily created. If these computers run a suitable middleware (such as SPADE, presented in Chapter 3), jobs composed of independent tasks can be easily executed on them. These virtual computers are easily initiated and managed, allowing the creation of real clusters of virtual computers running the operating systems and software previously provided by the users.

Bag-of-Tasks problems developed by users left out of cluster and Grid infrastructures may be solved by leveraging on-demand creation of virtual machines in order to provide the necessary computing cycles. Once set-up with tasks' execution environment and launched, each virtual machine may contribute by solving one or more tasks. In the specific case of Bag-of-Tasks problems, orchestration of the participating virtual computers is rather straightforward due to the absence of requirements for inter-task communication.

To be both efficient and cost-effective, the middleware in charge of launching virtual computers (i.e., scheduling) has to be able to predict the number of necessary computers and instruct their creation to the utility computing infrastructure. Naturally, this number varies among applications and data workloads. To deal with this, determining the optimal number of machines must take into account both the time necessary to complete the job as well as the minimum time unit subject to payment. This optimal number obtains the best speedups possible within a defined budget. Therefore, one must be able to avoid two relevant, because inefficient, boundary situations described next.

If the tasks are much shorter than the minimum time unit charged (normally one hour), allocating as many computers as there are tasks, will produce a very low ratio between processing time used and charged. This stems from the fact that only a small fraction of the time used by the virtual computers while running was actually used to solve the problem. Thus, when a large number of computers is allocated, the outcome will be the maximum possible speedup, but this may not be financially feasible. If instead, a very low number of (or just one) virtual computers are allocated, the speedup will be very small, but as each virtual computer will have to run for a long time, there may be no additional financial savings.

The necessary trade-off between these two extreme cases is specially difficult to decide in those situations where the expected processing time for each task is unknown beforehand and the unit of charge is much larger in comparison with the former. In order to address this problem, the remaining of this chapter presents an heuristic and scheduling algorithm capable of dynamically allocating computing resources for Bag-of-Tasks computations, maximizing speedups, while ensuring that users pay close to an initially predefined amount.

The need for an heuristic can be further illustrated by the following example. For instance, if in order to solve a job comprising 300 tasks (each one with a completion time of about 5 minutes), 300 computers are allocated, then, the maximal speedup is achieved, but at the expense of very high charged time as well. If the minimum time unit charged is one hour, the user will be charged for 300 hours for only 25 hours ($300 * 3\ minutes$) of total processing time used.

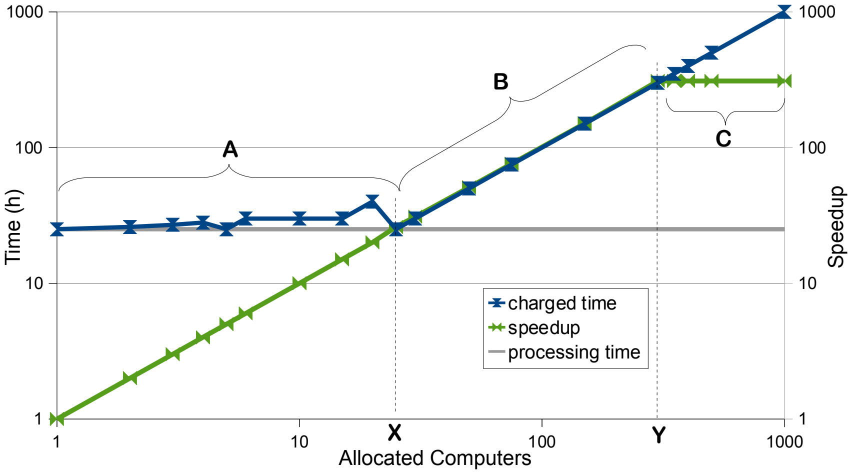


Figure 6.1: Evaluation of cost and speedups

Figure 6.1 shows the evolution of charged time and speedups, regarding this example, considering different numbers of allocated remote computers. The grey horizontal line represents for how long allocated computers are executing the tasks (in this example 25 processing hours).

The example presented in Figure 6.1 clearly shows three distinct areas, with different relations between charged time and the speedups.

When allocating a number of machines within **region A** (in this example between 1 and 25) the charged time is constant: the user always pays about 25 hours. If only one machine is allocated, it will be executing for the whole 25 hours, while if 25 machines are created, each one will only execute for one hour. It is also possible to observe that in this region, along with the increase of the number of allocated machines, the speedup also increases.

In **region B**, if more machines are created and added to the cluster, the speedup increases, along with the total payment. Although the computers are not executing for a complete hour, they are charged as if processing data for a complete hour, since the minimum charged value for each machine is one hour. With optimal task scheduling all computers process data during the same time, rendering a speedup equal to the charged time. If the number of allocated machines equals the number of tasks, each machine only executes one task. In this case the speedups are maximal, with a total payment (300 hours) much higher than the total processing time actually required (25 hours).

The third region (**region C**) corresponds to a setup where the number of machines is higher than the number of tasks. In this case, some machines are idle, not processing any tasks, but are still charged for one hour. The speedups remain constant, with only an increase of the charged value.

In the previous graphic two important points are also evident:

- **X** - represents the ideal number of machines to attain the maximal speedup when aiming for paying the minimal possible amount.
- **Y** - represents the ideal number of machines to attain the minimal payment when aiming for the the maximal possible speedup.

If the user needs the maximal speedup, all tasks should be executed concurrently. This is attained allocating as much computers as independent tasks. This corresponds to the point **Y** in the graph. Although most machines will be idle for a long time, the speedups will be maximal.

If the user is not willing to pay more than the total executing time, it is necessary that none of the allocated machines present any idle time, while guaranteeing that the maximal number of machines execute concurrently. This value is attained initially by simply dividing the total execution time with the charging unit:

$optimal\_hosts = total\_execution\_time/charging\_Unit$

If the user is willing to pay more than the minimum possible, it is necessary to allocate a number of machines between **X** and **Y**. The user pays more than the minimum, but there is a linear increase on the speedups.

If the number of machines is outside this interval, there is a waste of either processing power, or money:

- if less than **X** machines are allocated, it would have been possible to attain best speedups while paying the same amount;
- if more than **Y** computers are allocated, it would have been possible to pay less while attaining the same speedup.

Domestic users or even scientists without Grid or cluster access would be willing to execute their jobs with substantial speedups while within constrained budgets, instead of paying for the maximum speedup possible. By varying the number of allocated hosts inside the interval one can get higher or lower speedups, depending on the available budget, but guaranteeing that one gets the maximum performance that value could pay.

Determining the appropriate number of computers to create is made easier if the time each task takes to complete is known beforehand (e.g., resorting to bin packing algorithms [CJGJ78]), and elementary if this time is equal for all tasks. Small run-time adjustments make it possible to obtain good speedups while paying the minimum amount possible.

However, when the time to complete each task and its variation are not known *a priori*, which often occurs (e.g., ray tracing, BOINC projects, and most parameter sweep problems), the number of participating computers should be decided and possibly adapted during run-time. In this case, during job execution the only available information is for how long the completed tasks have executed. This partial information is the only one available to predict the total job execution time, and consequently the optimal number of computers to allocate.

Another option could be considering users' estimates of task execution times. However, frequently these are incorrect [MF01], and it has been found that proneness to estimation error is higher with less knowledgeable users [LSHS05]. Therefore, if user estimates fall below the actual task completion times, suitable run-time adjustments should be made. If however, estimates are above completion times, more hosts than necessary will have been allocated already. Moreover, this burden should be avoided by automating task running time prediction.

In the next section, other distributed computing infrastructures are presented along with their requirements and scheduling strategies, and why they do not apply to the proposed target environment. The following sections present the target resource and application model, intended target applications, and the heuristic and algorithm proposed. Finally, the algorithm is evaluated against workloads, traces and conclusions are presented.

6.2 Related Work

The scheduling of computational problems on available resources is a fundamental problem in order to optimize both program execution and infrastructure usage and availability. In this context, scheduling algorithms and related heuristics aim at ensuring that requests are handled with a specified quality of service, and that underlying resources usage is optimized.

Typically, MPI [Mes94] applications require a fixed predetermined number of hosts to cooperate in order to solve a problem, thus simplifying the decision on the resources to allocate. The processors allocated to each process can be used exclusively or shared with other requests. In *gang scheduling* [FR95], only one application is executing on a site with all tasks executing simultaneously, while in *co-scheduling* [Ous82] there is not the requirement different for tasks all tasks to execute simultaneously, thus allowing some tasks to be idle while other make the program progresses. Some hybrid techniques such as those presented by Bouteiller [BBH$^+$06] try to conciliate the best of the these approaches.

The access to Grid infrastructures usually requires the user to define the characteristics of the application to execute. These characteristics must state how many processors (or hosts) are necessary, their architecture, the operating system and the maximum duration of each task. In order to reduce the timespan of parallel applications, Grid schedulers employ heuristics that try to take into account the expected task duration, and the speed and availability of the selected hosts [CLZB00]. In the case of workflow applications, besides host selection heuristics, tasks are also ordered with the goal of reducing the job's total timespan [KA99].

In the case of Bag-of-Tasks problems, the number of concurrent processes is not previously known and may vary. Current cycle-sharing systems, such as BOINC [AF06], use a greedy approach to allocate remote computers: all available computers are used to solve part of the problem. BOINC clients participate in the selection of tasks to be executed [And07]. It is the BOINC client that is responsible for guaranteeing an even distribution of work among different projects the user is donating cycles to. The user states the share of idle time to give to the different projects, and after completion of tasks, it contacts different servers to retrieve tasks. Some improvements have been made, such as in CCOF [ZL04a], in order to add some resource efficiency to remote host selection algorithms.

Tasks comprised in Bag-of-Tasks problems are by definition data-independent, therefore subject to parallelization and simple restart. This allows easy scheduling and deployment on remote grids and clusters. Furthermore, these tasks can be executed when queues are empty, to take advantage of the resource idle times. For instance, Transparent Allocation Strategy [NCS+05] allows the allocation of processing power to execute tasks parameterized with the number of requested processors (p), and duration (tr). Smaller p and tr allow a better fit of the requests, while larger tr accommodates a wider range of tasks. The cluster resource manager tries to satisfy each request, but when processors are necessary for higher priority jobs, tasks allocated using Transparent Allocation Strategy are killed to free resources. Later, these tasks can be restarted on other available computers.

As a solution to the difficulty of determining p and tr, another strategy is proposed. The Explicit Allocation Strategy [RFC+08] presents an adaptive heuristic allowing, during run-time, the definition of both p and tr for each request, with information gathered by a resource scheduler managing space-shared resources. This heuristic takes into account free time slots available on the cluster and the estimated task duration time to generate the first request. If the tasks included in such request are successfully executed, the execution time of the longest task will be used in subsequent requests; if the requested time is not enough, the estimated task execution time will be multiplied by an integer factor. Even though some estimation is performed w.r.t. task execution time, this solution tries neither to obtain average task processing times, nor to reduce the unused idle time by the requests.

Existing utility computing infrastructures, (e.g. Amazon EC2 [Ama11], Enomalism [Eno08], or Eucalyptus [NWG+08]) provide means for the management of pools of computers, via deployment and execution of virtual machines. Such machines are created from disk images containing an operating system and necessary applications. Images are provided by the users, employing an API to launch and terminate the various instances of the machines.

In currently available utility computing infrastructures, resource allocation and scheduling problems are hidden at a lower level. When a user creates a virtual machine, the middleware managing the infrastructure is responsible for assigning a physical computer that can deliver the contracted quality of service. There is no need to know the total execution time for each virtual machine beforehand, as it is only used, after termination, to calculate the amount to charge. Furthermore, in commercial infrastructures, the charged time unit is large, usually one hour, which requires guarantees that machines are idle for a minimum amount of time.

The notion of computing clouds [FDF03] providing virtual clusters has been employed before and emerged as a natural step for designers of Grid infrastructures. In the work described in [FFK$^+$06], a number of previously configured Xen-based [BDF$^+$03] virtual machines, communicating via MPI, together with information regarding resource (CPU, memory) description and management, are considered as an aggregate virtual workspace. Complete aggregate workspaces are the basic unit of scheduling. Workspace deployment resorts to Globus Toolkit services when enough resources are available to schedule a given workspace.

In [SKF08], the basic unit of scheduling are individual virtual machine instances and their usage is compared against Grid-based scheduling for both performance, overhead, flexibility and overall system utilization. VM technology allows task deployment, activation and suspension. Resource management revolves also around VM instances that may be subject to leasing (best-effort approach subject to pre-emption) or advance reservation (with timing guarantees). The study shows that despite the inherent overhead of virtualization technology w.r.t. native execution, the ability to suspend VMs allows better overall performance (shorter total execution time and job delays) and system utilization, when compared to Grid-based schedulers without preemption. In the case of pre-emptive Grid schedulers, performance is only slightly worse compensated by greater flexibility and portability (neither need to modify OS, nor code targeting checkpointing libraries).

Computing clouds have become more and more used in thus called e-science problems, such as scheduling workflows of astronomy applications [HMF$^+$08] comprised of large numbers of small tasks. This approach was compared, with encouraging results regarding virtual clusters, in four different environments: combinations of virtual machines and virtual clusters deployed on the Nimbus science cloud versus a single local machine and a local Grid-based cluster. The work described in [EH08] performs calculations of MPI-driven ocean climate models on Amazon EC2 using 12 processes, each one running on a virtual machine inside a virtual cluster. They study the cost-effectiveness of the two main classes of architectures provided at the time by EC2 w.r.t. this type of applications (*m1-standard*, i.e., single-core and *c1-high-cpu*, i.e., multi-core *virtual* Opteron/Xeon processors). They have similar price-performance ratio even though the claims of almost five-fold performance increase in *c1-high-cpu* are not met experimentally. The authors conclude that it is feasible to run such applications on EC2, despite significant overhead penalties regarding bandwidth and latency of memory and I/O. A virtual cluster created on-demand can perform on par with low-cost cluster systems, but comparatively with high-end supercomputers, performance is much lower.

Job Scheduling can also be driven by utility functions following an economic or market-driven model [BAGS02, BAV05, CL06, LL07]. These systems employ utility function to analytically optimize system throughput while fulfilling user requirements. They can also make use of pricing models that are auction-based to achieve supply-demand equilibrium.

Taking these characteristics into account, current scheduling approaches, regardlessly of targeting cluster, Grid, or cloud computing scenarios, do not address the problem of optimizing the number of hosts to allocate in such utility computing infrastructures, employing quantum-based pricing schemes. Most of them do not take into account hard currency paid for the computing power used, and some of them even apply a totally counter productive greedy approach at machine allocation. They are mostly job-oriented and not task-oriented, which is more fine-grained. They assume the ability to preempt jobs or to suspend virtual machines, in order to uphold reservations; in utility computing infrastructures, processing time is paid by the hour and the middleware must take the most of it actually executing tasks, not suspending their execution which will not bring any savings. Finally, these approaches consider only a fixed (pre-configured or pre-calculated) number of participating hosts, unable to dynamically adapt the computing power engaged. This is rather inflexible and therefore unsuitable for many problems where task completion times are variable and unknown beforehand.

6.3 Resource / Application Model

Recently, hardware vendors started offering solutions to create truly on-demand providers of utility computing resources. Such providers own pools of computers and offer computing power in the form of virtual machines through simple launching mechanisms. Users first register their virtual machines' images, with suitable operating systems and software, and then launch them.

These new computing infrastructures also provide means to allow virtual machines within the same pool to discover each other and communicate among them. Moreover, it is also possible to programmatically create a new machine from another virtual machine already running in the infrastructure. All this allows the easy creation of a computational cluster and its self-management and tuning (addition or removal of machines) during run-time.

Load distribution between available physical computers is not user's responsibility. The virtual machine management software handles the creation of virtual machines and schedules them to suitable computers in order to guarantee a minimum of quality of service (e.g., virtual machine processing speed).

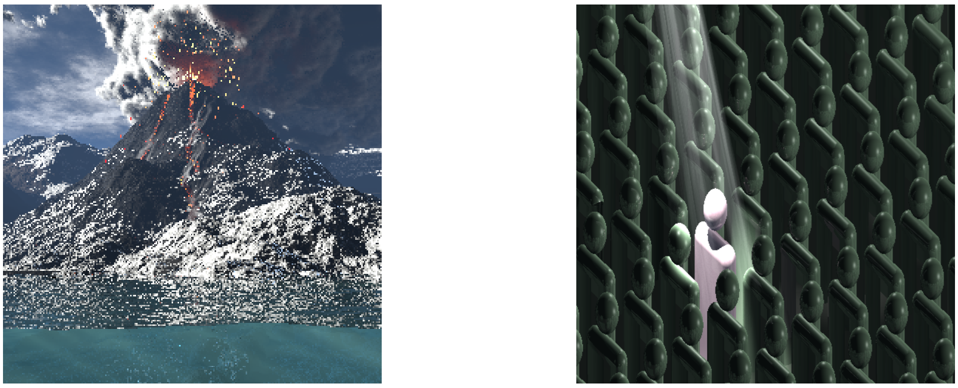
Users need not to reserve processing time slots. As with any other utility provider, accounting of chargeable time is performed after termination of the virtual machines using some predefined time unit (e.g., one hour per instance). The user will then pay for the time used.

Although not dealing with the explicit allocation of the physical resources, it still is responsibility of the user to decide how many machines to allocate.

The work here presented proposes an heuristic to dynamically decide how many machines to allocate for a Bag-of-Tasks application, where no communication occurs between tasks and where there is no need for all tasks to run simultaneously. The number of tasks should be high, on the order of hundreds, requiring several concurrent computers to process them all within an acceptable time frame.

An heuristic to determine the optimal number of hosts to solve a Bag-of-Tasks becomes a requirement when task execution times are neither constant nor known in advance. This category of problems includes some scientific simulations, data analysis, or even the parallel rendering of large images, where each task renders a small view-port of the final image, or each individual frame if rendering an animation.

As an example from the proposed application model, two submissions from the Internet Ray Tracing Competition (http://www.irtc.org/) were picked, each with different levels of complexity, in terms of graphical objects quantity and their distribution on the scene.



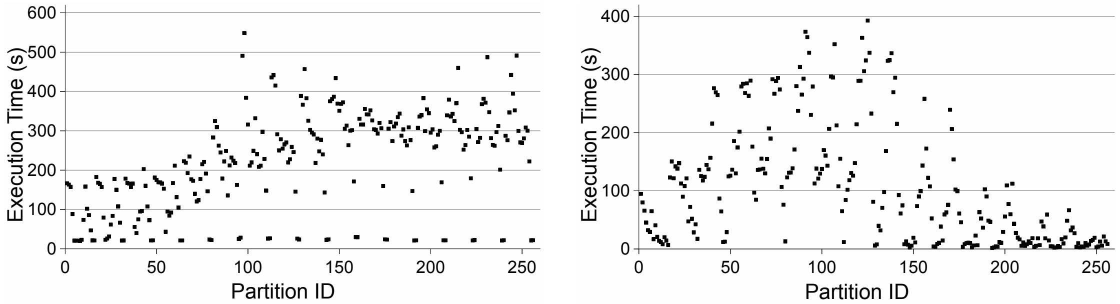
a) Example 1

b) Example 2

Figure 6.2: POV-RAy rendered example images

Although the output of both images is of identical size and resolution, the complexity of these examples is substantially different. Thus, the rendering times are different and so are the local complexity patterns in the images.

In order to assess this complexity and their difference, each scene was partitioned in a 16x16 rectangular grid, allowing each partition to be individually and independently rendered. These partitions were rendered using POV-Ray [Per08] on a 3.2GHz Pentium 4 PC running Windows.



a) Example 1 b) Example 2

Figure 6.3: Image partitions rendering executing times

Figure 6.3 shows the time each partition takes to be rendered for the examples in Figure 6.2.

If each partition is rendered in a different task, as the previous graphics show, it is impossible to predict how long will the next task will take to execute, even if executed in the order presented. Each task execution time depends on the local complexity of the viewport being rendered and can change substantially from neighbouring partitions.

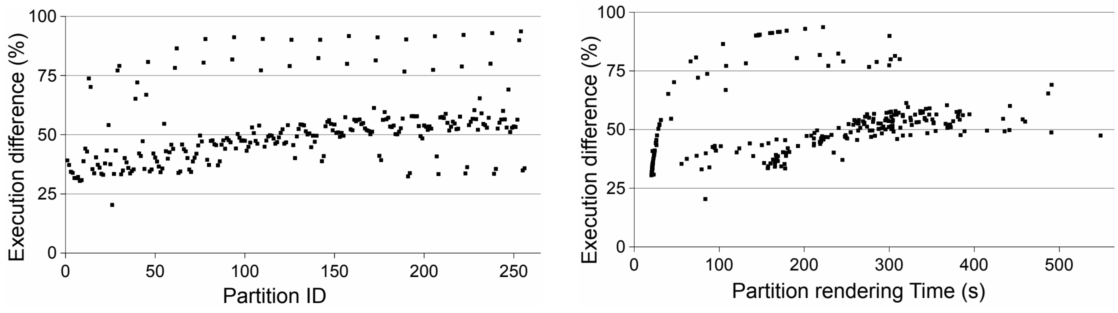
On the Example 1, occasionally a few tasks take about 25 seconds to complete, a value much lower than the average execution time (about 200 seconds), furthermore it is also possible to observe that as the rendering approaches its end, tasks take longer to execute.

Example 2 exhibits a different pattern, middle tasks take, on average, more time to execute than initial and final ones. The last partitions are of substantially shorter duration than all others.

These variations depend on the problem being solved by each task and result in different complexity (and execution time) patterns. These patterns can be difficult to predict and depend on the data being processed and the used code and configurations.

It is also possible to observe an unpredictable variation of the execution times, if a similar task is executed with different configuration parameters (at least to a large extent without intricate parameter analysis). It is also clear that the times to render the previous partitions with different output quality (different parameters) render multiple execution time differences: not all partitions or tasks suffer the same percentage variation.

Figures 6.4 shows, for each partition, the difference in execution times for the rendering of the Example 1 with two different configuration settings.



a) b)

Figure 6.4: Difference between distinct example 1 rendering: a) time difference depending on the Partition ID, b) time difference depending on original partition rendering time

In Figure 6.4.a) one can see that most partitions suffer a difference in execution time higher than 25% (with most partitions around the 50%) but some partitions exhibit a drift higher than 75%. From the partition identifiers, it is also impossible to predict how the execution time will vary.

In Figure 6.4.b) it is possible to conclude that the observed variation does not depend on the original partition rendering time. Simpler partitions exhibit a drift between 25 and 50%, but those suffering a higher drift (higher than 75%) had an original rendering time between 100 and 350 seconds.

The previous figures clearly show that even knowing a previous job execution time, it is impossible to predict future executions of the same job with different configuration settings. It is possible (if there is special knowledge regarding the impact of each parameter) to predict whether tasks will take longer to execute, but the exact difference for each task is not.

Similar observations can be performed in different Bag-of-tasks classes. One such example is software unit testing. Such jobs are composed of multiple independent tests, where each test verifies a delimited functionality, its code and dependencies.

As expected, the time to execute each task depends only of the complexity of the code being tested, and does not have any strict relation with the execution time of the previous and next test.

Figure 6.5 shows the execution time for the Outsystems Agile Platform testing, a software firm that provides a platform for rapid software development (RAD). Each point represents the average execution time of each test across all the runs during a week, where developers make changes to some of the platform's code.

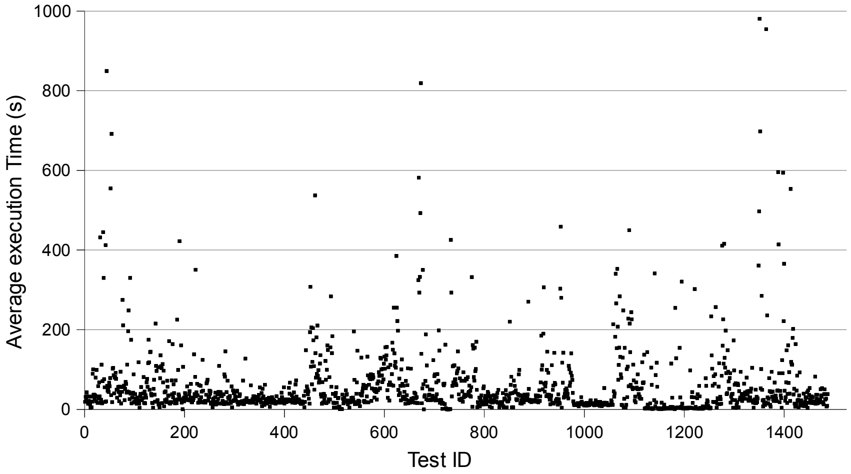
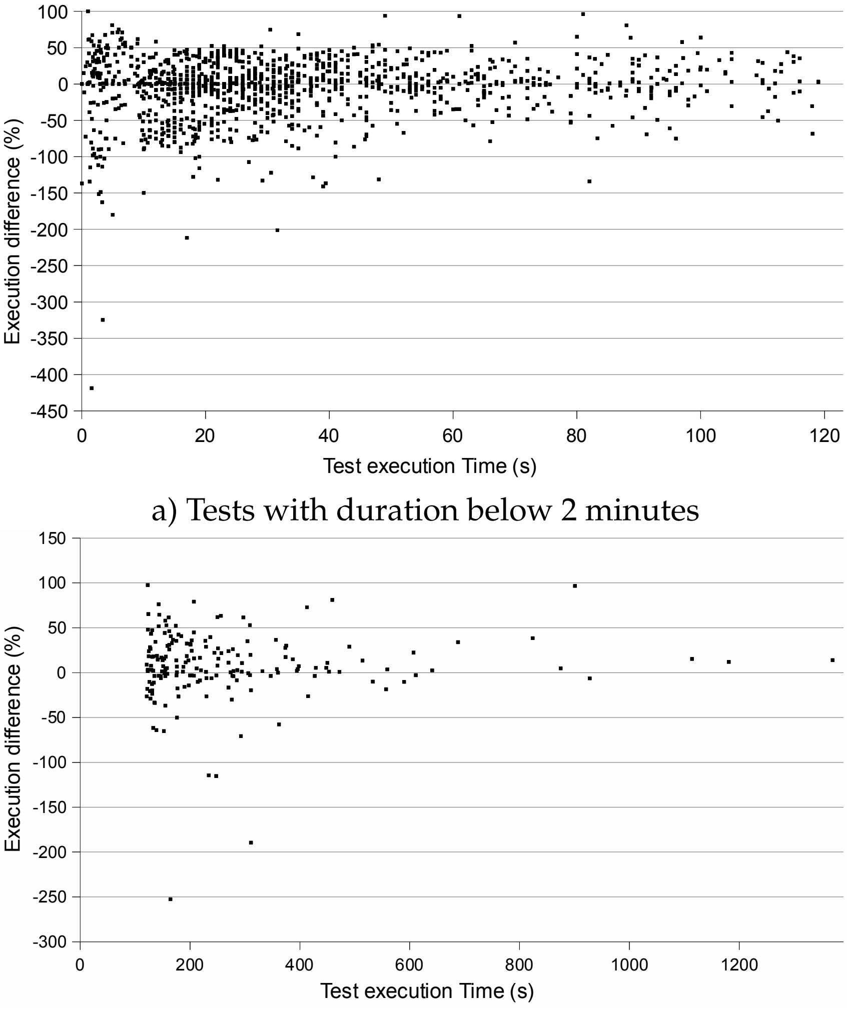


Figure 6.5: Unit testing executing times

The conclusion drawn from the image rendering examples can also be applied to this example. Tests (and tasks) execution times vary greatly and depend solely on the code being tested. In some areas of the test space, it is possible to observe some similarity of the execution times between consecutive tests, but many discontinuities also exist.

As in each test batch, new functionality, new code, and new interactions between modules are being tested, it is still more difficult to predict execution time of future tests. Since new code is being tested, error conditions may occur, drastically reducing the test execution time or, on the the other hand, longer loops can emerge, increasing the test time. This unpredictability between multiple executions of the same test can be observed in Figure 6.6. These graphs present the difference between two different Outsystems Platform unit tests (performed in consecutive days), showing how each individual test varies.



a) Tests with duration below 2 minutes

b) Tests with duration above 2 minutes

Figure 6.6: Difference between executions two batches of the Outsystems Platform test

It is possible to observe that there is no evident correlation between test length and the attained difference. In the case of shorter tests, it is possible to observe diverse variations, both negative (with a reduction of the execution time) and positive. It is also possible to observe that some tasks have much shorter execution times in the second batch, this is due to the possible errors in the code tested, or new code organization that significantly modifies code execution flow.

Most of the lengthier tests have small percentage variations, but due to the original length, they may have a great impact on the overall execution time. While the lengthier tests change little (in relative terms), shorter ones present a large diversity of variations. Also, in longer tasks, it is impossible to predict a trend on the overall variation of task execution time.

From the previous examples it is possible to conclude that in a single job (batch of independent tasks), several variation patterns exist: increase/decrease of execution time, execution time similarity between "neighbour" tasks and periodical/non-periodical variations. These variations patterns require a generic scheduling algorithm, that takes into account or handles all classes of workloads.

Furthermore, past execution times can not be used as reference of future tasks' execution times. Modifications in the task code, data or parameters, may lead to changes of the execution times. These time differences may be not linear, not allowing the clear extrapolation of a task's new execution time from others of the same batch or job.

6.4 Heuristic for task scheduling

The proposed heuristic allows the definition of the number of machines to allocate on a pool of computers, in order to ensure that the charged time fits within a predefined budget, while obtaining the maximum speedup possible with the allocated machines.

If a user wants to pay the minimum possible, each machine should execute and solve tasks during the whole duration of the minimum charging unit (one hour, for instance). If too many machines are created, each machine will have some idle time that will be charged anyway; if too few machines are allocated, the job timespan will increase, with no extra savings.

The optimal value for the number of machines to allocate is calculated using the simple expression:

$optimal\_hosts = n\_tasks * task\_average\_time/charging\_Unit.$

6.4.1 Virtual machine allocation

In order to define how many machines are needed, it is necessary to know how long a task will take to be executed (or the job total execution time). As presented earlier, for the target work class it is impossible to know this beforehand. Thus, during a job execution the only available information is the duration of the already completed tasks.

Using this information, it is possible to calculate the average task execution time of the concluded tasks. This value can be used to calculate an estimate for the optimal number of necessary computers.

Taking this into account, the developed algorithm, works broadly as follows. During normal job execution, whenever a tasks terminates, the measured execution time is stored and used to calculate the average task execution or processing time. This value is used to predict the optimal number of hosts needed to solve the remaining tasks.

Listing 6.1 shows the code executed whenever a task concludes.

```
remainingTasks —
finishedTasks ++
totalProcessingTime += concludedTask.processingTime
tasksAverageTime = totalProcessingTime/finishedTasks

possibleTasks = 0
for each runningComputer:
    possibleTasks += runningComputer.possibleTasks(tasksAverageTime)
necessaryComputers = (remainingTasks − possibleTasks) * tasks_Average_Time /
                            hostProcessingTime
```

Listing 6.1: Heuristic pseudocode executed when a tasks conclude

Lines 1-4 calculate the average time that was necessary to process the tasks completed thus far. This value will later be used to determine the number of necessary hosts, and how many tasks currently running hosts are still capable of executing while they are running.

In the following lines of code (lines 6-8), each host predicts how many tasks it will be able to process until the end of its `charging unit` interval. Both the remaining time each host still has left, and task average time, are used in this prediction. Since it is known how much remaining time each host still has left, using the task average time previously calculated, it is possible to estimate how many tasks a host is still able to execute till the end of the charging unit.

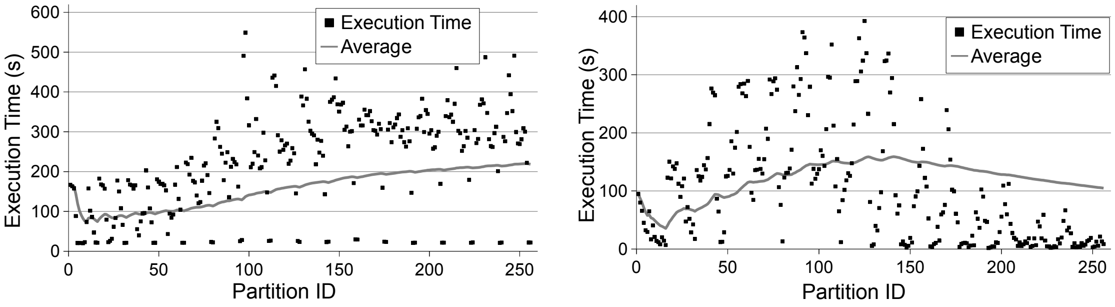
Lines 9-10 calculate how many additional hosts are needed to execute the remaining tasks. Since some tasks can be executed in the currently available hosts only $remainingTasks - possibleTasks$ tasks are taken into consideration. Here, instead of using the charging unit, the $hostprocessingTime$ value that used. This value is calculated subtracting each host startup time from the charging unit, and reflects the time actually available for a host to execute tasks. Furthermore, if the user is willing to attain higher speedups, $hostProcessingTime$ can be lower than the charging unit. For instance, if each host is only willing to execute half the charging unit, twice the number of hosts are needed, doubling the speedup, but also increasing the total payment.

6.4.2 Task selection criteria

Although not being easily predictable, task execution times can still present some visible trends. For instance, the first tasks can be executed substantially faster that the average, or even the opposite.

For the examples previously presented, it is possible to draw the average execution time (calculated with the execution times of completed tasks) and observe those trends.

Figure 6.7 shows the average execution times, for the previously presented examples, if tasks are executed with two predefined orders, exhibiting two different behaviours.



a) Example 1

b) Example 2

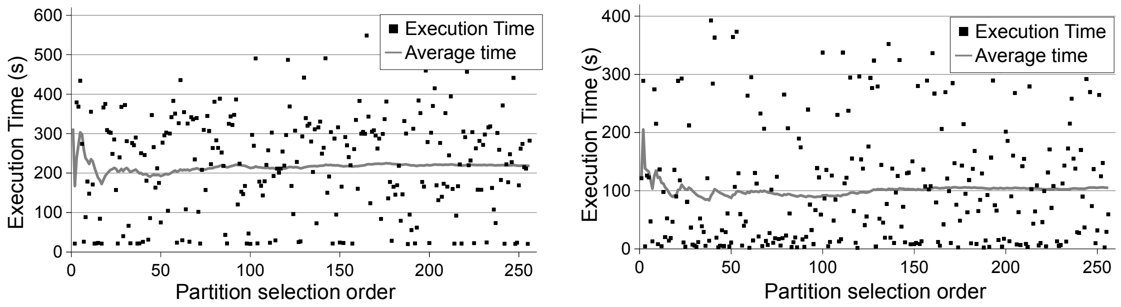
Figure 6.7: Image partitions rendering executing times and average of completed partitions execution time

In the case of Example 1, the average execution time increases as the tasks terminate, only approaching the final value when the last tasks terminate. If the this value is used, the number of allowed host in the beginning of the jobs would be lower than optimal. Furthermore, near the end of the job, new hosts would be allocated, because the previously allocated hosts were not sufficient. This would render a number of allocated hosts higher than the necessary, since the last created hosts would be idle after the end of the job. Despite the high number of allocated hosts, the job would take longer than required to conclude, because only close to the end, when few tasks remained, more computing power was added.

Figure 6.7.b) shows a different behaviour. At a given time, in the middle of the job, the average task processing time reaches its highest point, decreasing till the end of the job. In this example, when reaching the highest value, more machines are created, although not all completely necessary for the remaining of the tasks. Since remaining tasks are shorter than the calculated average value, most machines will present a high idle time. In this case, the speedups could be higher but the payment would also be higher.

To solve the problems resulting from such variability in these trends, it is necessary to guarantee that a good approximation to the average value (only actually known at the end of the job) is obtained as close as possible to the beginning of the job. To contribute for this, the heuristic must ensure that tasks are executed in a random order. In fact, by randomly selecting tasks, some possible locality differences in processing time are levelled, guaranteeing that the average task processing time will converge more rapidly.

Figure 6.8 shows how the average execution time varies, when tasks are selected randomly.



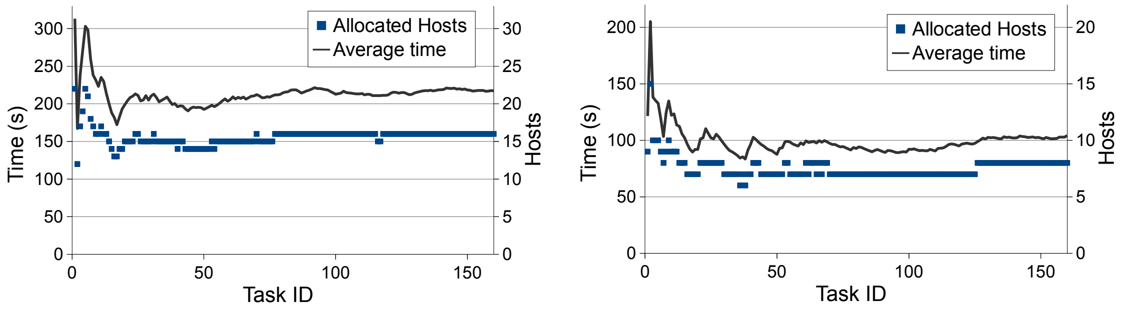
a) Example 1 b) Example 2

Figure 6.8: Example of random selection of partitions: executing times and average of completed partitions execution time

It is evident that after a few tasks conclude (close to 20) the final average values are attained, or very closely approximated. This fact allows the early definition of the optimal number of hosts to allocate. When other random task execution ordering is used, the final average value is attained after a similar number of tasks.

With a now stable average execution time, due to the random selection of the tasks to execute, it is possible to predict earlier the optimal number of hosts to allocate. The code that executes when a task terminates is the one presented in Listing 6.1. For the previous examples the evolution of the number of execution machines is presented in 6.9.

The optimal number of machines (16 and 8, for the Examples 1 and 2) are attained and stabilize after the completion of about 15 tasks. Nonetheless too many hosts can still be initially allocated. This is due to the fact that some of the first tasks can have an execution time higher than the average. In the previous examples, in the beginning of the each job, 23 (Example 1) and 15 (Example 2)



a) Example 1 b) Example 2

Figure 6.9: Example of random selection of partitions: executing times and average of completed partitions execution time

hosts would have been created, resulting an overallocation of 40% and 100% more machines.

6.4.3 Overallocation prevention

In order to reduce the effect, on task allocation, of having longer tasks executed in the beginning of the job, it is necessary to attenuate the calculated average execution time.

This attenuation is performed by means of a `creationRatio` factor that is applied to the calculated value in Listing 6.1. Initially `creationRatio` is lower than 1.0, so that the number of initially allocated computers is conservatively lower than the calculated value.

Since the calculated average time becomes stable and increasingly closer to the final one (as seen on Section 6.4.2), the attenuation should be mitigated. As new tasks terminate, and the average time converges to the final value, `creationRatio` may also converge to 1.0. The variation of the `creationRatio` depends on another value (`increaseRatio`), as shown in Figure 6.10.

In the previous example, the initial value of `creationRatio` is 0.5, meaning that after the first tasks terminates, only half of the necessary computers calculated are indeed created. The value of `creationRatio` is also updated, taking into account `increaseRatio`. The higher the `increaseRatio` the faster `creationRatio` converges to 1.

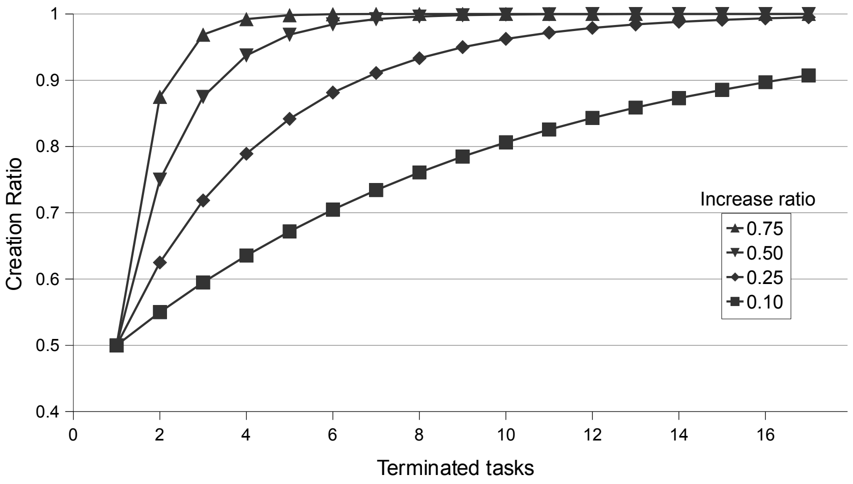


Figure 6.10: Unit testing executing times

If the user knows in advance that tasks do not have a high variation execution times, the value of `increaseRatio` can be high (expressing higher confidence that the average values calculated initially are close to the actual average), thus guaranteeing that `creationRatio` increases rapidly to 1.

The code for the calculation of the number of machines to allocate, and the update of the value of `creationRatio`, is presented in Listing 6.2.

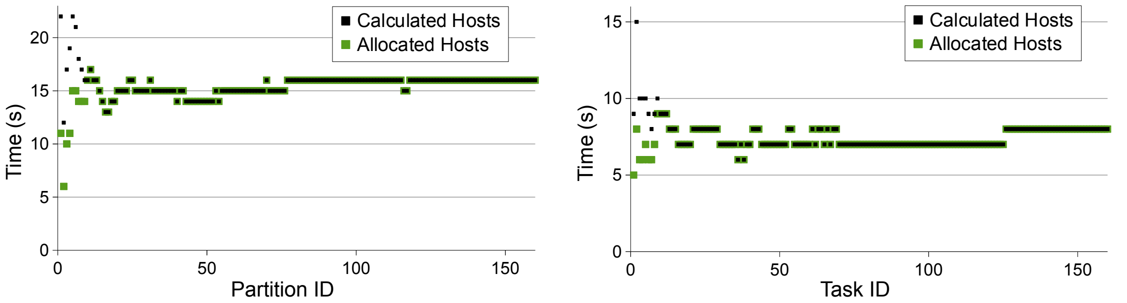
```
1 if (necessaryComputers>0):
2    computersToCreate = ceil(necessaryComputers*creationRatio)
3    for i in range(computersToCreate):
4       allocateNewComputer()
5    creationRatio = creationRatio+(1-creationRatio)*increaseRatio
```

Listing 6.2: Heuristic pseudocode executed when a task concludes

If more hosts are necessary (line 1), it is necessary to apply to it the creation ratio. Next, in lines 3 to 4, the necessary hosts are created. Finally the `creationRatio` is updated, taking into account the value of `increaseRatio`.

The value of `creationRatio` is used to guarantee that the initially predicted numbers of hosts are not higher than necessary. As this value converges to 1, so does the value of the calculated average of task processing time converges naturally to the final value. This effect is evident in Figure 6.11.

In Figure 6.11, it possible to observe the effect of `creationRatio` on the



a) Example 1

b) Example 2

Figure 6.11: Result of `creationRatio` usage: calculated number of hosts and actually created hosts

number of hosts to allocate. When the first tasks terminate, there is a difference between the calculated value and the one used to create hosts. When the execution times average converges, this difference disappears.

The first prediction points to 23 hosts, but after applying a `creationRatio` of 0.5, the actual number of hosts allocated is only 11. After about 10 completed tasks, the `creationRatio` has the value 1.0. Also from this point forward, the calculated execution time average is also close to its final value. Since this happens, the number of allocated hosts is the one calculated.

By varying the initial value of `creationRatio` and `increaseRatio`, it is possible for the user to decide if the heuristic has an aggressive (i.e., speedup oriented) or more conservative (i.e., savings oriented) behaviour, by allocating, in the beginning of the job, more or less hosts (launching virtual machines). The influence of the initial `creationRatio` and `increaseRatio` parameters will be evaluated in the next section.

6.4.4 Periodic update

Since the presented heuristic relies only on information gathered when tasks terminate, if the first processed tasks take too long to terminate, only after a long time, can new hosts be allocated to the computation. This reduces the speedups, since at the beginning of the job, not all possible hosts are executing.

To solve this problem, periodically, the code shown in Listing 6.3 is executed.

It calculates the average processing time of the currently executing tasks and, if greater than the previously found average time, it preemptively creates new hosts.

```
1 for each runningComputer:
2    runningTasksProcessingTime += runningComputer.currentTask.processingTime
3 runningTasksAverageTime = runningTasksProcessingTime/runningComputers
4
5 if runningTasksAverageTime > tasksAverageTime:
6    currentTasksAverageTime = (totalProcessingTime + runningTasksProcessingTime) /
7                              (finishedTasks + runningComputers)
```

Listing 6.3: Heuristic pseudocode executed periodically (partial)

In Listing 6.3, lines 1-3 calculate for how long currently executing tasks have been running, and their average execution time (up to that moment). If this value is greater than the average execution time of the previously finished tasks (line 5), the `currentTasksAverageTime` is calculated (lines 6-7). This new value is then used to decide how many hosts are necessary, in a similar manner as the code presented in Listing 6.1 (lines 7-10) and Listing 6.2 (lines 2-4). The values for `tasksAverageTime` and `finishedTasks` were previously calculated (when tasks terminated), furthermore these values, along with `creationRatio`, are not modified here.

All this code is periodically executed locally on the computer responsible for the coordination of all virtual machines. Since this coordinator machine has all the information regarding every virtual computer creation times, and all running tasks, no extra communication is necessary.

6.4.5 Host termination

In order to guarantee that the time each host is running is close to the minimum charging unit, so that wasted time is reduced, every host must be terminated just before reaching the minimum charging time, usually one hour. If a running task is prematurely terminated, it can and will be restarted on another host, with no side effects but with some processing time waste. This solution only works if all tasks are smaller than the charging unit, as it is oblivious of the following two cases: i) the average task processing time is close to or higher than the charging unit, and ii) a long task starts close to virtual machine shutdown.

The first case is easily tackled by instructing several machines to execute multiples of the charging unit, delaying their finishing deadline. This way, one virtual machine can solve several lengthy tasks, without wasting idle cycles nor interrupting running tasks.

From the number of tasks still pending, the number of hosts running and the average task processing time, the coordinator (the machine where tasks are launched from) must decide what machines should continue executing after the charged time unit, and for how long.

This step is performed just before the creation of new hosts. If the additional number of machines outperforms half the number of still pending tasks, some hosts are allowed to execute for one more charging unit. The selected hosts should be those that are closer to termination (with less remaining processing time) so that the number of prematurely terminated tasks is minimal and to guarantee that no machine terminates right after this decision.

When dealing with hosts that are allowed to execute longer, the previously presented remains the same: the only value dependent on the virtual machine completion time is the discovery of the number of tasks each host can run (line 8 in Listing 6.1). The method `possibleTasks` takes into account the new deadline when calculating how many tasks the host can complete.

When shutting down virtual machines, the problem of knowing if the task running on that machine should be allowed to terminate or not is more complex. When dealing with this problem, it should be taken into account the average task processing time, and for how long that particular task was running.

If the task was running for a short period of time, no harm comes from restarting it later on another computer. If the task's running time surpassed the average task running time, probably it should be allowed to terminate normally. In this case, the user would pay for more execution time, but will be able to use the remaining of the virtual machine time to solve other tasks.

6.4.6 Handling of long tasks

Besides the issue of host termination (dealt with in Section 6.4.5), tasks with long execution also raise additional concerns when deciding how many machines to allocate.

Independently on the number of allocated hosts, the length of the bigger tasks limits the maximum speedup. In an extreme case, where one machine is allocated for each task, the speedup would not be equal to the number of machines, as it would be limited by the duration of the longest tasks, since most of the hosts would be idle (because their shorter tasks had already terminated), while that last task was still executing.

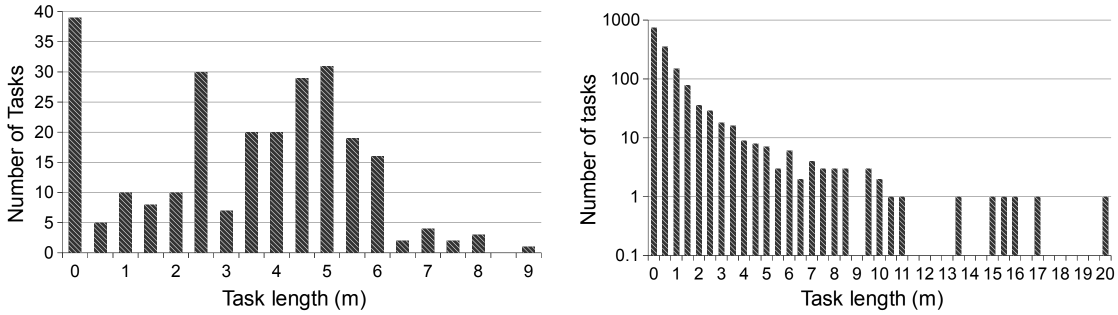
In order to solve this issue, one possibility is to allocate less machines and have them execute for a longer period. The speedups would not be so high but the cost would be lower. For instance, if only half of the needed machines were allocated, but allowed to execute for more than the charging unit, the costs would still be low but with a slight decrease of the speedup. In one case, the timespan would be equal to the length of the longest task (with most hosts presenting high idle times), while with this solution the timespan would be slightly longer (guaranteeing however that the payment was close to optimal).

To tackle this issue, in the presented heuristic, before creating the necessary machines (line 3 in Listing 6.2, and after the code in Listing 6.3), a simple evaluation of the number of machines to create is performed: if the number of necessary hosts is greater than half the number of remaining tasks to execute, instead of creating more virtual machines, the running time of those already executing is increased. This causes that those on-line machines will execute twice the charging unit.

If, however, the number of hosts is close to the number of remaining tasks, it means that there is a high probability that such tasks are long running and thus, incapable of running on the available processing time. This adaptation preemptively increases the execution time in available hosts, instead of deciding later whether hosts are allowed to execute (as presented in Section 6.4.5). Furthermore, this adaptation can be continuously applied, increasing the hosts' running time, when observing that the running time still available is not enough for task completion.

6.5 Evaluation

To evaluate the algorithm and the heuristics described in the previous section, three different representative examples of Bag-of-Tasks jobs were used: i) a synthetic job with a set of tasks whose processing times have a normal distribution, ii) traces of a real image rendering job (example 1 presented in Section 6.3), and iii) the OutSystems software testing job (also mentioned in Section 6.3). With these execution traces, it is possible to study how the heuristic behaves with different classes of jobs, the attained speedups and the total payment. To study how the heuristic responds to jobs with very long tasks, a specific synthetic job was also used.



a) Example 1 image rendering b) Outsystems software testing

Figure 6.12: Tasks's execution time distribution

Figure 6.12 shows the tasks execution time distribution for the image rendering and Outsystems job. The image rendering job has 256 tasks (illustrated in Figure 6.12.a), with an average task execution time of 220 seconds and standard deviation of 125 seconds. In the task execution times histogram (Figure 6.12.a)), it is possible to observe that there is a significant number of tasks with short execution time. The OutSystems job (Figure 6.12.b) is composed of 1480 tasks, where half of the tasks have an execution time lower than 30 seconds, and the average execution time is 1 minute. There are few long running tasks with execution times longer than 15 minutes. Regarding the synthetic job, it has 256 tasks with execution times following a normal distribution, with an average of 150 seconds and a standard deviation of 30 seconds.

In this evaluation, the proposed algorithm and heuristics were evaluated against 200 different randomly generated orderings of tasks execution scenarios.

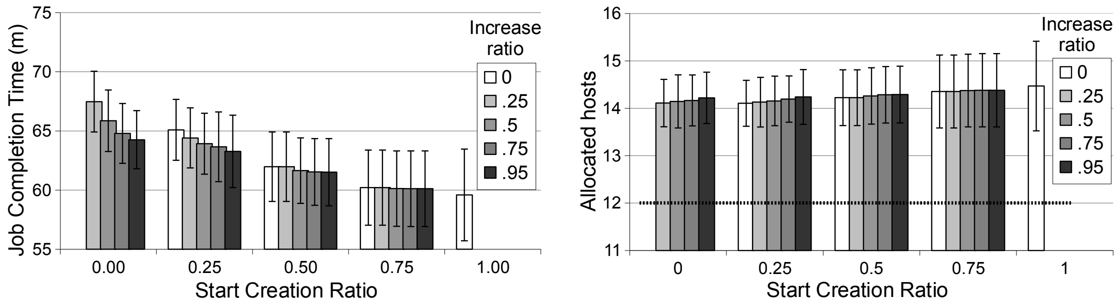
In all experiments, the first 5 minutes of every host are spent on creating the virtual machine and launching the operating system and the necessary middleware. Since the charging unit used was 60 minutes, this delay renders the usable initial execution time unit to only 55 minutes (instead of a full 1 hour slice). This value is used by the heuristic, namely when making the calculation of the optimal number of hosts.

These tests allow the evaluation of the behaviour of the algorithm and heuristics with different classes of jobs, all comprised of many tasks whose duration is variable, not known in advance and with multiple distributions. The algorithm and heuristics were coded in Python [Pyt08], and simulated using the SimPy [Sim09] discrete-event simulator.

6.5.1 Impact of creationRatio and increaseRatio

The following graphs show the effectiveness of the algorithm and heuristics previously described, for each of the three jobs presented. Each one shows the average time to complete each of the jobs previously mentioned, and the number of created hosts, for different values of `creationRatio` and `increaseRatio`.

Figure 6.13 shows the results when applying the heuristic to the synthetic workload.



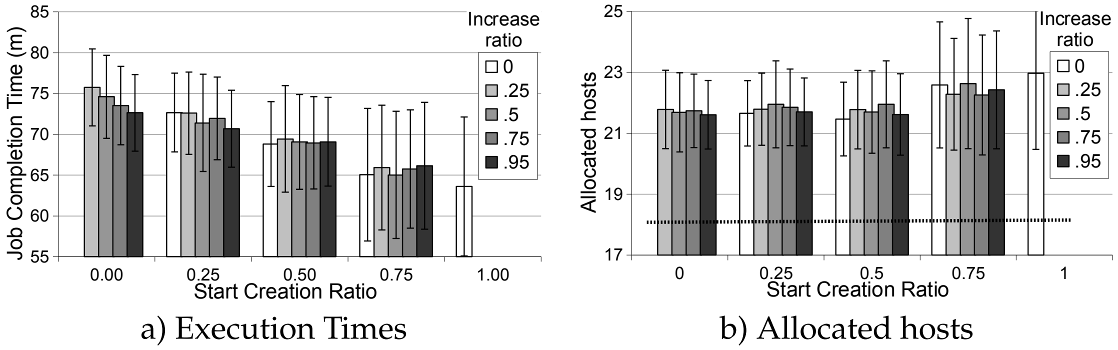
a) Execution Times b) Allocated hosts

Figure 6.13: Evaluation of the impact of `creationRatio` and `increaseRatio` on the job completion time, and number of hosts created for the synthetic job with a normal distribution of tasks processing time.

The first possible observation is that the attained execution times are close to the target: jobs' execution times range between 60 and 70 minutes, and the number of allocated computers (and subsequently payment) is close to the minimal (dashed line in Figure 6.13.b)). Note that the graphs show in detail only a fraction of the result space (e.g., between 11 and 16).

In this example, it is evident the impact of the values of `creationRatio` and `increaseRatio`. As the initial `creationRatio` increases, the job timespan is reduced, but, on the other hand, with an increase on the number of allocated hosts. A similar trend is noticeable with the variation of `increaseRatio`.

Figure 6.15 presents the results for the ray tracing example shown in Figure 6.2 and described in Figure 6.12.a).



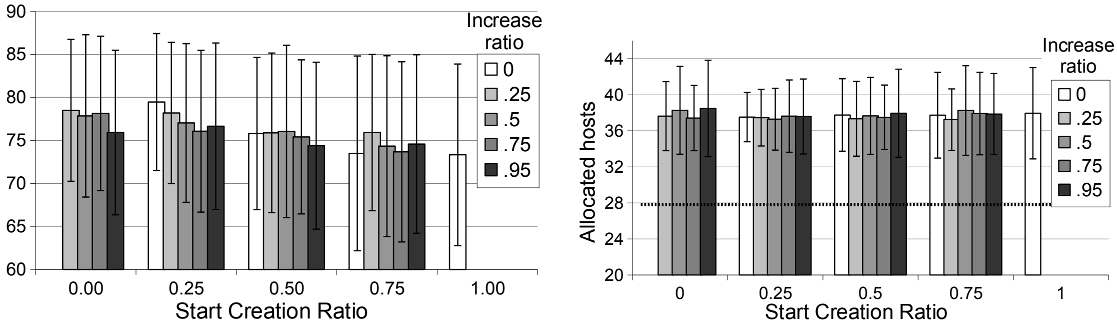
a) Execution Times b) Allocated hosts

Figure 6.14: Evaluation of the impact of `creationRatio` and `increaseRatio` on the job completion time and number of hosts created for the image rendering job (Example 1).

The trend observed in the case of the synthetic workload are also presented in these graphs with a reduction of the execution time, as both ratios increase along with a slight increase of the number of allocated computers.

In this example, the difference between the attained times and the charging unit, and between the allocated hosts and its minimum possible value, is higher on the synthetic case. This is due to the execution times distributions. Since most tasks had execution times much lower than the average, it becomes highly probable that the first tasks are short running, affecting the initial number of allocated hosts.

The same effect but at a higher scale is present in Figure 6.15.



a) Execution Times b) Allocated hosts

Figure 6.15: Evaluation of the impact of `creationRatio` and `increaseRatio` on the job completion time and number of hosts created for the software testing job from OutSystems.

Since the tasks execution times have a noticeable bias in this example (Figure 6.12.b)), the difference between the optimal and attained values (time and hosts) is much higher. Nonetheless the heuristic, without any *a priori* knowledge about the job being executed, is still able to limit the expenses to about 135% of the minimal value.

From these graphics, it is possible to observe that the algorithm and heuristics previously presented, with a suitable selection of `creationRatio` and `increaseRatio` parameters, can predict the number of hosts to create (and respective bill) with a difference between 20% (synthetic and image rendering) and 35% (OutSystems software testing), with respect to the optimal number of hosts.

Nonetheless, it is important to note that the presented optimal number of allocated hosts are indicative lower bound values. In order to attain the expected execution times with these number of hosts, it would be necessary to schedule all the tasks of each job in such a way so that no host would ever have any idle time.

From Figures 6.13 and 6.14, it is possible to see that with a `creationRatio` of 1, the standard deviation (solid vertical lines) of both the total job execution time and the number of allocated hosts is higher. As the execution times of the first tasks are used without any correction, these initial values have great impact on the number of allocated hosts. The reason for that lies in the fact that during job execution, any variation on the average task execution time (and consequent calculated average time) makes the number of created hosts to change rapidly. In this case, the algorithm and heuristics respond very rapidly to modifications of the average task execution time. Therefore, a value of `creationRatio` lower than 1 is needed to attenuate the influence of such task execution time variations.

However, when choosing a too low initial value for `creationRatio`, the heuristic does not yield good results either: only after too many terminated tasks, enough hosts are effectively created to execute the job's tasks.

In the case of the OutSystems job (Figure 6.15), as the majority of the tasks are short termed, it is highly probable that the initial tasks render a low average execution time. In this case, the `creationRatio` increases rapidly, not being possible to observe a distinct difference between the various combinations of both ratios: `creationRatio` and `increaseRatio`.

The influence of `increaseRatio` can be easily observed if an initial `creationRatio` of 0 is used: higher values of `increaseRatio` reduce the total job execution time, while increasing the number of created hosts, as well as the standard deviation of both the total job execution time and number of allocated hosts. When using a higher initial `creationRatio` value, these observations are not so evident but are still present. Thus, the user must decide the values for these two variables according to a more conservative or aggressive posture, as described in the next section.

6.5.2 Speedup and Allocated Hosts

To further evaluate the effectiveness and efficiency of the presented algorithm and heuristics, it is necessary to compare directly the speedup with the number of allocated hosts for distinct values of `creationRatio` and `increaseRatio`. Figure 6.16 presents what speedups are obtained when a varying number of hosts were allocated for the two of jobs already mentioned (image rendering and synthetic).

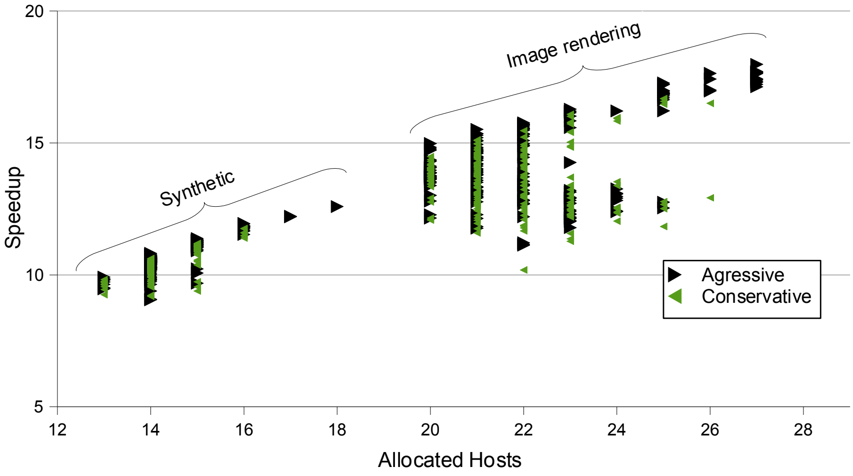


Figure 6.16: Speedup evolution with the number of created hosts for two scenarios.

Each of the points depicted represents one job execution with different ratios: conservative (the value of both `creationRatio` and `increaseRatio` is 0.5) and aggressive (the value of both `creationRatio` and `increaseRatio` is 0.75).

In both scenarios, conservative and aggressive, the number of created hosts and the speedup do not coincide, not yielding an 100% efficiency. The first reason is the fact that 5 minutes out of one hour were not used to process tasks (virtual machine setup time), limiting the maximal efficiency to about 90% (more precisely 91,7%) . Other reasons lie in the fact that only after a few tasks terminate, it is possible to have an accurate prediction of the average execution time. Furthermore, if multiple hosts are executing tasks, the shortest ones complete first, affecting the first calculated averages. Close to the termination of the charging unit, some hosts are allowed to execute for more than a charging unit. This may result in some waste, since this decision was taken close to the job completion, and probably all tasks terminate before these new deadline expires.

One key difference distinguishes the aggressive from the conservative approach: the aggressive approach offers higher speedups. One of the reasons is the amount of allocated hosts. It is possible to observe that in the case of the aggressive approach, the range of allocated hosts is broader than when using conservative values for both ratios. In the case of the conservative approach, most of the results have less than 16 hosts (in the case of the synthetic workload), and less than 24 in the case of the image rendering, while with the aggressive approach the maximum number of allocated hosts increases to 18 and 28, respectively. With the increase of the number of hosts, it is possible to observe also an increase on the speedups.

It is noticeable that, for the same amount of allocated hosts, better speedups are attained with an aggressive behaviour. This is due to the fact that higher values are calculated earlier with the aggressive behaviour, and thus those hosts are recruited sooner to enrol in the computation.

Comparing the number of allocated machines with the speedups attained, it is possible to conclude the efficiency of the heuristic. As stated earlier, with a virtual host setup time of 5 minutes, the maximal individual contribution to the speedup is 90% (almost 10% of the charged time is wasted, not being used to execute tasks). The proposed heuristics presents efficiency levels higher than 50%, reaching in some executions 75%.

It should be noted again that the 90% efficiency is only attainable if the order by which tasks are executed, is such that no host presents any idle time. Only knowing the exact execution time, in advance, of each task would it be possible to schedule them in a way to guarantee minimal waste.

In conclusion, the user can vary the values of `creationRatio` and `increa` seRatio between 0.5 and 0.75, obtaining different variations on the number of created hosts (and charged values), but always with efficiencies higher than 50%.

This difference between the number of allocated hosts and the speedups has two reasons: some hosts are allocated close to the completion of the job, leading to an excess of wasted processing time, or the heuristic maintains them running leading to the charging of an additional charging unit.

6.5.3 Long-Running Tasks

To evaluate how the presented algorithm and heuristics handle jobs with long-running tasks, another synthetic job was used. This job was composed of 256 tasks with a execution time fitting into a normal distribution with a mean value of 1.5 hours and a standard deviation of 20 minutes.

The results on trying to decide how many machines to allocate are presented in Table 6.1. Since these tests were performed with 200 different combinations of the random values, the table presents the average of the measured values along with the standard deviation (inside parentheses).

	Allocated hosts	Wall Time (h)	Speedup (h)	Payment (h)
Serial	1	384 (5)		**384** (5)
1 host per task	256	**2.43** (0.14)	**158** (8.5)	512 (5.8)
Heuristic (100 initial hosts)	179 (2.9)	3.70 (0.16)	104 (4.2)	463 (6.3)
Heuristic (50 initial hosts)	154 (0)	3.90 (0.15)	99 (3.7)	452 (6.6)
Heuristic (1 initial host)	129 (5.9)	4.14 (0.17)	93 (3.6)	450 (5.9)

Table 6.1: Evaluation of scheduling with long running tasks

Table 6.1 presents the results for five different settings:

- serial execution in only one host
- allocation of one host per task
- three simulations using the presented heuristic with different initial hosts

From the first two experiments, it is possible to retrieve the three important values that limit any possible results (shown in bold):

- Minimal payment (384 hours) - This value corresponds to the payment if only one host is allocated.
- Minimal wall time (2.43 hours) - This value is the one measured when execution one task per host and corresponds to the longest task
- maximal speedup (158) - Attained when executing one task per host.

Independently of the number of initially hosts, the heuristic allows a payment saving, when compared with the allocation 256 hosts. Related with this reduction on the payment, it is also evident a decrease on the speedup. If more hosts are initially allocated, the results are better, the speedups increase more than the payment: the speedups increases 11% while the payment only increases 3%.

Although for large tasks the allocation of one host per task, may seem a viable solution, using the proposed heuristic, the number of allocated hosts is lower (essential if there exists a limit to concurrent hosts), and the total payment is also lower, while attaining comparable execution time and speedups.

6.6 Conclusion

In a Utility Computing infrastructure, hosts can be easily allocated on-demand to execute jobs comprised of independent tasks. Only after the computation, the user will be charged for the time each host ran. The heuristic presented in this chapter efficiently determines the number of hosts to allocate on such a computing infrastructure, when used to solve Bag-of-Tasks problems, whose task execution times are not known before their execution. This removes the burden form the user to decide how many hosts to allocate.

For jobs with short termed tasks, the results show that the heuristic determines the number of necessary hosts to guarantee that the charged time is close to the desired value. The number of allocated hosts is close to the value that would be found, if the user knew for how long each task would execute; the speedups accomplished are close to the number of allocated hosts, therefore achieving cost-efficiency.

Furthermore, the heuristic can provide distinct behaviours: i) a conservative one, where the charged values are lower, and ii) a more aggressive one, where the charged values are higher, with a proportional increase of the speedups. The user can select within the spectrum of these two behaviours, by varying both the `creationRatio` and the `increaseRatio` parameters. This enables either reducing the charged time (with longer timespan), or reducing the job timespan with an increase in payment.

The heuristic is also able to handle jobs with long tasks, tasks taking longer than the charging unit. The speedups remain high, while guaranteeing a lower payment than if one host per task was allocated.

If the user has a guess on the task processing times, this information can be used to launch several computers when the job starts. The number of initially launched computers should also be corrected with the `creationRatio`, to avoid the allocation of too many machines.

7

Utility Algebra for Resource Discovery

Previous chapters shown systems to allow the execution of parallel jobs by a new class of users. These new users not only have different requirements from the current ones, but are able to use and leverage new sources of computing power.

The new available computing sources range from private clusters, utility or cloud computing infrastructures, Internet Distributed Computing systems to peer-to-peer overlay networks. Furthermore, the available resources are becoming more and more heterogeneous: multiple distinct infrastructures exist, and the Internet shared computers have multiple configurations and characteristics.

Existing resource discovery protocols have a number of shortcomings for the current variety of cycle sharing scenarios. They either i) were designed to return only a binary answer stating whether a remote computer fulfils the requirements, ii) rely on centralized schedulers (or coherently replicated) that are impractical in certain environments such as peer-to-peer computing, iii) they are not extensible as it is impossible to define new resources to be discovered and evaluated, or new ways to evaluate them.

This chapter presents a novel, extensible, expressive, and flexible requirement specification algebra and resource discovery middleware. Besides standard resources (memory, network bandwidth,...), application developers may define new resource requirements and new ways to evaluate them. Application programmers can write complex requirements (that evaluate several resources) using fuzzy logic operators. Each resource evaluation (either standard or specially coded) returns a value between 0.0 and 1.0 stating the capacity to (partially) fulfil the requirement. This value is calculated considering a client defined utility depreciation (i.e., *partial-utility* functions that define a downgraded measure of how the user assesses the available resources) and policies for combined utility evaluation. By comparing the values obtained from the various hosts, it is possible to precisely know which one best fulfils each client's needs, regarding a set of required resources.

7.1 Introduction

Currently, there is increasingly greater offer of computing cycles and computing resources in general, whether paid or free. There is a myriad of alternative approaches and technologies encompassing: computing clusters (possibly virtualized), idle time of computers in private LANs, academic and scientific institutional grids creating virtual organizations with coordinated scheduling, utility and cloud computing infrastructures, distributed computing on opportunistic or desktop grids, and peer-to-peer (fully decentralized or federated) cycle-sharing topologies. At the same time, and also motivated by that, there are more and more users taking advantage of these infrastructures with different applications capable to take advantage of such available resources.

Regardless of the underlying cycle-providing infrastructure and the applications, resource discovery (e.g., CPU, memory, bandwidth, specific hardware installed) is a key enabling mechanism. It serves a double purpose of enabling suitable resources to be discovered for execution, as well as systems utilization optimization (on the user and provider sides).

Today, there are many systems supporting the discovery of resources; e.g., Grid infrastructure schedulers [CKK$^+$99, RLS99, FTL$^+$02], resource monitoring systems [ZS05] with possible resource aggregation [SPM$^+$06, CGM$^+$03, CT10], service discovery protocols [GCLG99, Gut99], and web service discovery [CGMW03]. These protocols and systems have a number of shortcomings for the wide variety of present cycle sharing scenarios. They either i) were designed to return only a binary answer stating whether a remote computer fulfils the requirements, ii) rely on centralized schedulers (or coherently replicated) that are impractical in certain environments such as peer-to-peer computing, iii) they are not extensible as it is impossible to define new types of resources to be discovered and evaluated, or iv) they are inflexible in the sense that users and administrators are unable to set alternative ways to evaluate and assess available resources.

Therefore, a middleware platform (STARC) was designed in order to make resource discovery more adaptive via extensibility and with increased flexibility, highlighting:

- ability to incorporate the evaluation of new resources;
- expressiveness in requirement description;
- ability to evaluate partial resources fulfilment;
- employment of fuzzy-logic to combine multiple requirements

STARC is able to interface with different network topologies e.g., LAN, peer-to-peer (P2P). STARC extensibility stems not only from allowing the evaluation of most usual resources (e.g., CPU speed and number of cores, memory) but also from the dynamic inclusion of new characteristics (presence of specific applications, services, libraries, licenses or hardware) to be evaluated.

Regarding flexibility, STARC uses XML files to describe application requirements stating, with logic operators, the relation between the several resource characteristics that are relevant, with associated value ranges, and utility depreciations in the case of only partial fulfilment being possible (i.e., sets of resource availability ranges and associated utility depreciation or *partial-utility*). Hosts may return information ranging from no availability (0.0), to requirements fully met (1.0). If requirements are only partially met, a value between 0.0 and 1.0 is returned (partial-utility), taking into account the partial fulfilment ranges and associated utility depreciations functions (not necessarily strict linear mappings), as well as evaluation policies provided by the client. Utility depreciations are also applicable to individual resource alternatives not enclosed within ranges (e.g., OS family).

Next section presents some available systems that allow the evaluation of remote computer resources, compares their shortcoming. In Section 7.3, the STARC architecture is presented, along with its components, and resource discovery protocol. The algebra that aggregates the utility of each individual resource included in a resource discovery request is also presented in this section. Sections 7.5 and 7.6 describe STARC implementation and evaluation. This chapter closes with the conclusions.

7.2 Related Work

Currently, there are a number of systems that allow the discovery of resources in network-connected computers. Such systems fall into the following categories: cycle sharing systems, resource management in Grid infra-structures, service discovery protocols, or utility-based scheduling. This section describes these systems and present their limitations regarding the intended flexibility, expressiveness and extensibility.

7.2.1 Cycle sharing systems

Currently available cycle sharing systems allow the development of parallel applications that execute in remote computers. Projects like BOINC [AF06] provide a centralized infrastructure for code distribution and result gathering, while G2:P2P [MK03a, MK05] and CCOF [ZL04b] provide a truly P2P access to computing cycles available remotely, employing advertisement propagation, expanding ring search, and rendezvous supernodes.

In either set of systems, the processing power or other relevant resources are not taken into consideration. In projects such as BOINC or CCOF, only the processor state is relevant when selecting the remote host that will execute the code. This solution is easy to implement, fair to the owner of the remote computer, but may slow the overall application, while being restrictive by not considering other resources and partial fulfilment of the requested resources.

7.2.2 P2P based resource discovery

In the area of P2P overlay networks the work more related to resource discovery has been trying to optimize the number of messages necessary to discovery a host that has resources that fulfil a complex requirement.

Some systems resource to to specific network topologies to accomplish that. Mercury [BAS04], for instance, builds multiple logical groups (one for each resource), in order to allow the representation of the multitude of available resources. When performing resource discovery, Mercury, first finds the corresponding logical group, and then finds the most suitable host. The evaluation of multiple requirements requires the simultaneous navigation on multiply logical groups.

Other group of systems use the available resources as keywords of the data to be stored in the overlay network. Min Cai [CFCS03, CH07] proposes the use of a Chord [SMK$^+$01] based P2P overlay network to store the information about the resources available on each host. When a host enters the overlay, it registers its available resources on available P2P nodes. For a certain resource a_i with value v_i, the selected peer to store its identity and characteristics is is $n = successor(Hash(v_i))$. Later, range queries (for instance when searching for a host having a resource in $[k, l]$), will start verifying the existence of any host information in the peers between $successor(Hash(k))$ and $successor(Hash(l))$.

More recent works try to optimize the number of exchanged messages, necessary in order to acomplish a multiple resource query, by coding all available resources in bit-arrays. FRDT [KK11] organizes peers in a tree-based structure, where each node knows if a certain resource is available on its children.

PIRD [She09] represents available resources as vectors in multidimensional spaces and stores each host information in bit-arrays. These arrays are also used to define what peer from a Chord like overlay will store that information (namely, the name of the hosts, and resources there available). When searching for a host with a determined set of resources, the returned one is the one with the closes bit-array, by means of a locality sensitive hashing.

Although these systems allow the discovery of the available resources, none of the available systems takes care of the actual evaluation of the resources, since available systems only allow the specification of its capability or existence, and the simple matching of resources with requirements. Furthermore no distinction between users is possible, since all these systems ignore any possible user classing and treat all queries in the same manner.

The management of information about multiple resources is not optimized, in Mercury, multiple groups are expensive, while the work of Min Cai, may suffer from uinbalance of information. Although optimizing the search for a resource, both FRDT and PIRD only allow the storing of boolean information about the existence of a certain resource, but not its real capability. Furthermore, it is necessary to previously know the what resources can be searched for.

7.2.3 Grid Resource Management

In order to optimize the use of Grid resources it is necessary to choose the hosts that best answer to the resource requirements of applications. The work described in [ZFS07] offers a study on the performance and scalability of most widely deployed approaches to resource monitoring in existing Grid middleware (MDS, R-GMA and Hawkeye). They are found to offer similar functionality and performance with good scalability behaviour w.r.t. the number of clients making simultaneous requests to the system. Next, the most important features of resource discovery in Grid infrastructures are presented along with their limitations.

MDS4 [SPM$^+$06] describes a Resource Aggregator Framework that could allow implementations to calculate combined utility of a set of resources. None of them implements the utility algebra proposed in this chapter and described in Section 7.3.

Both Condor [LLM88], Condor-G [FTL$^+$02] and Legion [GW96] provide mechanisms to state what requirements must be met in order to execute efficiently some code. Legion uses Collections [CKK$^+$99]: repositories of information describing the resources of the system, that are queried to find the host having the required resources. Collection queries are built with the usual relational and logical operators. Condor, Condor-G and Hawkeye use Classad [RLS99, FTL$^+$02, CT10] objects in order to describe computing nodes' resources (providers) and define requirements and characteristics of programs (customers). Classads are dictionaries where each attribute has a corresponding expression. These attributes describe the characteristics of the resource provider (the owner of the Classad). Matchmaking of requirements and resources is accomplished using customers' and providers' Classads. Such systems allow the description of application requirements and the discovery of a computer that supports its efficient execution. However, if a host does not completely satisfies a requirement, it is simply considered as not eligible. Although the rank attribute allows ordering customers and providers with matching attributes, this is only used to order providers that fully satisfy the requirements. Furthermore, aggregation of several Classad with configurable weights by users is not available.

Current Globus GRAM[All11] implementations, as described in [CFK$^+$98], suffer from the same problems: usually, a fixed infrastructure is needed to collect and monitor resource information. Although this information may be replicated for higher reliability and throughput, this is often done resorting to additional institutional servers, frequently co-located. The set of resources to monitor, evaluate, and aggregate, although configurable, is pre-determined within a virtual organization. This renders Grid Resource Management systems inadequate to dynamic environments, where there is no centralized infrastructure and the resources to evaluate are highly variable.

In the work described in [HML04], clients are able to select hosts based on user-provided ranking expressions, in order to evaluate resources considering peak and off-peak periods, sustained availability over a period of time, time of day. In [ODDG03], an adaptive resource broker estimates job completion time based on periodically sampling available resources (CPU) in order to trigger job migration. However, both systems are designed to speedup execution and trigger job migration, bearing no information on how multiple resource requirements (besides CPU) and their partial fulfilment could be described and evaluated. Furthermore, in the scenario targeted by STARC (embarrassingly parallel applications), tasks can be made much smaller, can be easily restarted and job migration stops being a requirement.

7.2.4 Service Discovery Protocols

Service discovery protocols, such as SSDP [GCLG99] and SLP [Gut99], allow a client computer to search and discover other computers that are service providers. These protocols allow a service provider to advertise its presence. In this advertisement the service provider sends a description of the exported service. A client receiving such a message compares the service description with the desired service requirement and, if the requirements match the service description, the client can start using the service. The discovery of the services can also be initiated by the client, by broadcasting a message with the requirements the service provider must comply to. Every service provider that has a service that matches the requirements answers with its identification.

SLP [Gut99] also allows the existence of Directory Agents, central servers that are contacted by the client when looking for services and by the Service Providers to publicize its services. Web service discovery [CGMW03] can be employed in distributed computing scenarios to find locations where remote functionality may be executed. Even if it is possible to find devices and hosts that have a certain service, it is not possible to easily evaluate the available resources, since these protocols were designed to ease the discovery (i.e., mere presence) of services. The requirements are tested against the static characteristics of the service, not allowing the evaluation of available resources at the computer. Whenever several versions of the same service could be used by the client, these protocols still do not allow the association of utility depreciations to outdated versions (i.e., accepting some outdated versions while preferring current one), therefore being inflexible.

7.2.5 Utility-based Scheduling

There are systems that perform scheduling based on utility functions [BAGS02, ABG02, CL06, LL07] aiming at maximizing overall system utility or usefulness. However, these solutions suffer from a number of drawbacks. They assume there is a coordinator or scheduler node that receives and process all resource discovery requests and therefore is able to employ complex pricing models that globally optimize resource usage as well as maximize user utility. The way these systems incorporate utility functions is rather inflexible as they either: i) consider only complete requirement fulfilment and award it an utility (possibly weighted), or ii) they assume a nearly constant elasticity model where the lack of availability of one resource can be supplanted by a surplus of others for equal utility (clearly a system-centric approach and not user-driven), or iii) provide no support for hints regarding resource or service adaptation in case of partially unfulfilled requirements.

7.2.6 Network Management Protocols

Although of a lower level, network management protocols can be used to query remote computers about its resources, thus allowing the evaluation of the surrounding environment. These protocols may not help locating the computer on a network, but after the initial communication setup, allow a finer evaluation of the available resources. Currently two major standards exist: SNMP and WBEM/CIM.

SNMP, Simple Network Management System, is a protocol used in remote management of computer networks. The original RFC [CFSD90] defines the architecture, the representation of the information managed and the operations supported on the managed information. Subsequent RFCs define the managed information base (MIB). For instance, the MIB$-$II [Ros90] specification defines the data that an agent (i.e., a resource provider) must export to management applications. Other MIB defines what information related to storage, processes, memory usage is exported through this protocol.

The Distributed Management Task Force is developing several standards related to the management of computing resources. These standards range from the definition of APIs the hardware should comply to, to the development of protocols of remote network management.

The CIM [Dis10b] specification defines a management information conceptual model and the set of operations to manage the information. The Web Based Enterprise Management [Dis10a] (WBEM) has three main components: CIM, that defines the structure of the managed data, the XML encoding specification for the representation of CIM classes and instances, and a protocol of CIM operations over HTTP, enabling the management of remote computers using the CIM exported information.

All these protocols allow the querying of the resources available in a remote computer, but all the network communication and resource comparison must be explicitly coded by the application programmer. Even though, these protocols may help define what computer resources should be evaluated and provide an interface to locally access the state of the resources.

Besides the communication primitives used to query a remote host, these protocols also define the information and characteristics that can be accessed remotely. This information, although dynamic in value, is statically defined, depending on the class of the host.

7.3 STARC architecture

Applications need to discover, evaluate and select the resources present and available in remote computers as well as services and software provided by them. This is achieved via a middleware platform (STARC), an assemblage of components (STARC daemons), that execute both in clients and in resource providers, all regarded as peers. Each daemon handles all requirement evaluation requests: those generated from a local application and remote requests generated by other remote daemons. The architecture of STARC is presented in Figure 7.1 and is described next.

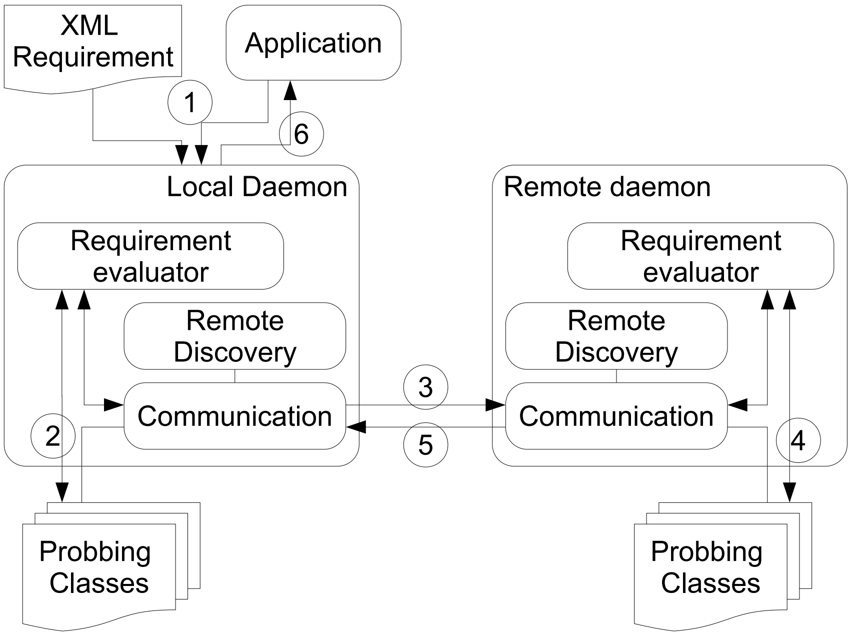


Figure 7.1: STARC Middleware Architecture

To use STARC, an application, must provide to the local STARC daemon an **XML Requirement File** containing a logical description of the hardware and software requirements and alternatives (Step 1 in Figure 7.1). The **STARC daemon** reads the requirement and executes the relevant **Environment Probing Classes** (Step 2) in order to know how the resources fulfil the requirements. The logical descriptions are evaluated against the values returned by the Environment Probing Classes according to specified partial-utility functions and combined utility evaluation policies defined by an utility algebra (more details in Section 7.4.3).

After local resource evaluation, the STARC daemon contacts remote daemons (Steps 3) by means of the **Communication** component. Each contacted daemon evaluates the requirements (Step 4) and returns the resulting value (Step 5). The contacted hosts are those found by the The **Remote Host Discovery** module. This component abstracts the middleware from different network topologies. It interfaces with the rest of the middleware uniquely by providing for each request a list of available hosts. These hosts are later contacted by the **Communication** component. Further details on remote host discovery when addressing different network topologies (e.g., LAN, coordinator/scheduler-based grids or virtual organizations, peer-to-peer overlays) are addressed in Section 7.5.

After receiving the evaluation of all contacted hosts, a lists of results is returned, so that the user can choose the most appropriate one.

7.3.1 Probbing Classes

In existing systems, all evaluation code is statically linked to the underlying middleware, reducing extensibility: the addition of new resources to be evaluated (or new ways to evaluate existing resources) requires the recompilation and redistribution of the complete system.

STARC proposes the use of dynamically loaded classes to perform evaluation of all types of resources. These probing classes are dynamically loaded at runtime depending of the resource to be evaluated: from the name element of the resource being evaluated, STARC dynamically loads the correct class and executes it.

The dynamic loading can be performed from the local disk (step 2 in Figure 7.1) or from another STARC daemon (step 4). Each class must follow a predetermined interface: the constructor (that initializes the evaluation environment) and a resource evaluation method.

Each **Environment Probing Class** evaluates one specific resource, comparing its state with a parameter received from the **Requirement Evaluator** (a XML snippet whose contents are to be presented in Section 7.4). The way the STARC daemon interacts with the **Environment Probing Classes** is further detailed in Section 7.5.

7.4 Requirement Specification and Evaluation

In order to use STARC, the user or programmer writes a XML file stating what are the requirements and feeds it to a locally running STARC daemon. This is done by a single method invocation, in order to make the evaluation of the XML requirement files transparent to the programmer. On completion of discovery, the application will receive a list identifying available hosts and how fit they are to fulfil those requirements.

The XML requirement file syntax is presented in Listing 7.1.

```
<! ELEMENT requirement          (resource | and | or | not ) >
<! ATTLIST requirement policy   (priority | strict | balanced | elastic) "strict" >
<! ATTLIST requirement weight   CDATA "1.0" >
<! ELEMENT and                  (requirement+) >
<! ELEMENT or                   (requirement+) >
<! ELEMENT not                  (requirement) >
<! ELEMENT resource             (config+) >
<! ATTLIST resource    name     CDATA #REQUIRED >
<! ELEMENT config               (#PCDATA) >
```

Listing 7.1: XML requirement DTD

These requirements can be physical resources (e.g., available memory, processor speed, GPU installed), installed software (e.g., certain operating systems, virtual machines, libraries, helping applications) or availability of specific services (e.g., logging, checkpointing, specific web services).

A simple example requirement is presented in Listing 7.2 and will be used as a base for the description of the features of the proposed algebra.

```
<requirement>       <and>
  <requirement>  <resource  name="CPU"  >
    <config>
      <processorCores>          16          </processorCores>
    </config>
  </resource>         </requirement>
  <requirement>  <resource  name="CPU"  >
    <config>
      <processorSpeed>         3000         </processorSpeed>
    </config>
  </resource>         </requirement>
  <requirement>       <or>
    <requirement> <resource  name="NumPy">
      <config>
        <version>              4            </version>
      </config>
    </resource>  </requirement>
    <requirement>  <resource  name="PyGPU">
    </resource>     </requirement>
  </or>  </requirement>
</and>  </requirement>
```

Listing 7.2: Prototypical example of XML requirements description

A requirement can refer a simple resource (CPU or the libraries NumPY or PyGPU in Listing 7.2) or it can be a complex composition of other requirements using logical operators. The composition of requirements is accomplished by using usual logical operators (and, or, not) whose evaluation will be described later in the Section.

In order to precisely define a required resource, it is necessary to state its name and, if necessary a XML snippet configuring how the resource will be evaluated: in the previous example only the minimal required value is stated. This config element (e.g. processorCores, processorSpeed, version in Listing 7.2) will be fed to the adequate **Probbing Class** during resource evaluation. The **Probbing Class** taking into consideration the config XML snippet is able to evaluate the resource and return an adequate value.

Using the prescribed syntax it is possible to write complex requirements with the conjunction or disjunction of the characteristics of the resources. For instance, it is possible to state that a program needs a certain number of processor cores, with specified speed, and either one of two libraries, as illustrated in Listing 7.2. In this example the user wants to know if a certain computer has an ideal configuration of at least 16 available processor cores preferably with speed of at least 3000 MHz, and has either the `NumPy` (at least version 4) or the `PyGPU` library installed.

To state this information the user employs the `or` and the `and` logical operators to three different resources. Inside the `CPU resource` element the user states the required configuration for `processorCores` and `processorSpeed` (an implicit conjunction). Although the information inside the `config` XML tag can follow a predetermined pattern, the information inside it depends on the resource, its possible characteristics and the code of the responsible probing class.

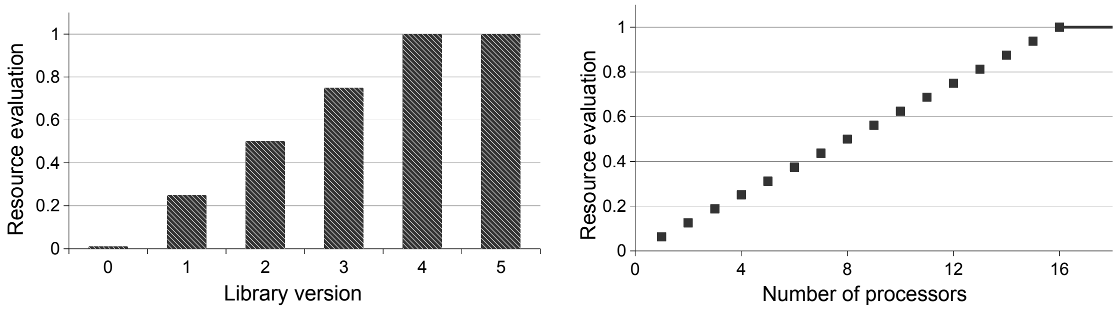
In addition to specifying requirements, XML files allow users to employ an utility algebra, described next, comprised of: i) ranges of resource values with associated client-specific utility depreciations (partial-utility), ii) alternative resource options with associated utility depreciation, and iii) policies for combined satisfaction evaluation.

7.4.1 Partial utility resource evaluation

All the systems presented in Section 7.2 return only boolean answers to the queries. If the host fulfils the stated requirements, true is returned, otherwise return false. If a host can completely fulfil the requirement, it returns the highest possible value, meaning, it is among those that best fit the requirement. However, this kind of binary answer to the evaluation of requirements is rather inflexible, since a host that cannot fulfil a requirement completely, it should not necessarily be considered as having utility 0.0. Although not the optimal resource, nonetheless the user may be willing to use it to solve his tasks.

STARC solves this issue by allowing any requirement to be evaluated to a Fuzzy value. The returned value fits into the $[0.0, 1.0]$ interval: if the host fulfils the requirement, 1.0 is returned; otherwise, a value between 0.0 and 1.0 is returned.

This behaviour can be observed in Figure 7.2 for the number of processors and the NumPy library.



a) NumPy library evaluation b) CPU evaluation

Figure 7.2: Resource evaluation (examples from from Listing 7.2)

If the host being evaluated exceeds the minimum required value the **Probing Class** returns 1, while if (in the case of the NumPy library) no resource exists, the returned value is 0.0.

In the intermediate cases, the returned value (between 0.0 and 1.0) is proportional to the existing resources: in this example, if the host only contains 8 processors (half the required) its evaluation returns 0.5.

Since this resource evaluation returns a numerical value stating the partial capacity to meet the requirement, the user can use a collection of values (produced in multiple hosts) to decide which hosts to use and what is the threshold for acceptance.

On a different perspective, the same host configuration (processor cores and speed) and the same CPU evaluation code will render different partial fulfilment values, according to the specifics of each client's request. Thus, when receiving requests with different configurations, a host can also select the tasks that more efficiently use the provided resources. Moreover, XML requirement files issued by clients contain additional information specifying alternatives that can also be used to drive resource and service adaptation at contributing hosts. This way, the overall system is rendered more flexible and assurance of requirement satisfaction is improved.

7.4.2 Non-linear Partial-Utility Criteria

Although offering higher flexibility and expressiveness than existing systems, simple stating one limit value for the intended resource characteristics is not flexible enough. Users may not want to have resources to "depreciate" in a linear fashion. For instance in the example of Listing 7.2, a host offering less than four processor cores should be excluded, its evaluation yielding a value of 0.0.

In such cases, the configuration of the resource evaluation should be more expressive, allowing the user to define ranges of resource characteristics with its limiting partial-utility evaluation. These partial utility ranges must follow the format defined in Listing 7.3 and appear inside the resource characteristics elements (processorCores, processorSpeed, and version in Listing 7.2).

```
<!ELEMENT range        ((minnumber | maxnumber ), util) >
<!ELEMENT maxnumber    (#CDATA)>
<!ELEMENT minnumber    (#CDATA)>
<!ELEMENT option       (value, util) >
<!ELEMENT value        (#CDATA)>
<!ELEMENT util         (#CDATA)>
```

Listing 7.3: XML requirement DTD

The `range` element is used to define utility values for well known characteristics of the resource: inside the `maxnumber` (or `minnumber`) the limit of the range and inside the `util` element appears the well known utility value.

The `minnumber` elements should be used to create a monotonically increasing utility function (such as the example in Figure 7.3): the `util` element contains the utility of the lower limit of the range and should increase when the `minnumber` increases.

The user only needs to define three points $(2, 0.5)$, $(3, 0.5)$ and $(5, 1.0)$ using the `minnumber` and `range` elements. From these points STARC interpolates the

```
<range>
  <minnumber> 2 </minnumber> <util> 0.5 </util>
</range>
<range>
  <minnumber> 3 </minnumber> <util> 0.5 </util>
</range>
<range>
  <minnumber> 5 </minnumber> <util> 1.0 </util>
</range>
```

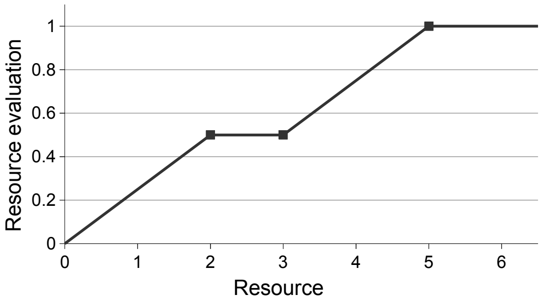


Figure 7.3: Increasing Utility Function definition

remaining of the function. Resources with values lower that 2 will fit into the line between $(0, 0.0)$ and $(2, 0.5)$, while resource values higher than 5 will yield an evaluation of 1.0.

In a similar manner `maxnumber` should be used to create a monotonically decreasing utility function (Figure 7.4), e.g., referring to price, queue length, waiting time, concurrent jobs being executed.

```
<range>
  <maxnumber> 2 </maxnumber> <util> 1 </util>
</range>
<range>
  <maxnumber> 4 </maxnumber> <util> 0.3 </util>
</range>
<range>
  <maxnumber> 10 </maxnumber> <util> 0.0 </util>
</range>
```

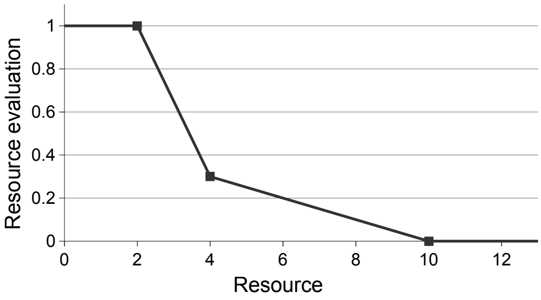


Figure 7.4: Decreasing Utility Function definition

Using the `range` elements, the example from Listing 7.2 can be further detailed, as show in Listing 7.4

Furthermore, as illustrated in the example in Listing 7.4 (lines 19-20), users can also assign the utility depreciation (partial-utility) of the individual alternatives available. In this example the user requires the version 4 of the the NumPy library, but is partially satisfied with version 3. In hosts with any other installed version the evaluation of this resource will yield 0.0.

This individual assignment is made by means of the `value` element defined in Listing 7.3, lines 5-6, and is particularly useful for resources whose characteristics can not be represented numerically (for instance, operating systems).

```
<requirement policy="strict"> <and>
  <requirement> <resource name="CPU" >
    <config> <processorCores>
      <range> <minnumber> 16 </minnumber> <util> 1.0 </util>  </range>
      <range> <minnumber> 10 </minnumber> <util> 0.5 </util>  </range>
      <range> <minnumber>  4 </minnumber> <util> 0   </util>  </range>
    </processorCores> </config>
  </resource> </requirement>
  <requirement> <resource name="CPU" >
    <config> <processorSpeed>
      <range> <minnumber> 3000 </minnumber> <util> 1.0 </util> </range>
      <range> <minnumber> 2600 </minnumber> <util> 0.5 </util> </range>
      <range> <minnumber> 2400 </minnumber> <util> 0   </util> </range>
    </processorSpeed> </config>
  </resource> </requirement>
  <requirement> <or>
    <requirement> <resource name="NumPy">
      <config> <version>
        <range>  <value> 4 </value> <util> 1.0 </util> </range>
        <range>  <value> 3 </value> <util> 0.5 </util>  </range>
      </version> </config>
    </resource> </requirement>
    <requirement> <resource name="PyGPU">
    </resource> </requirement>
  </or> </requirement>
</and> </requirement>
```

Listing 7.4: XML requirements description

7.4.3 Policies for Combined Satisfaction Evaluation

Given a XML requirement file with a set of resources and service requirements, it is necessary to select the host best fit for it. This selection must take into account not only the partial utility of each requirement (the evaluation of each individual resource) but also an evaluation of the combined satisfaction of requirements as a whole. This is accomplished applying the presented operators (lines 4-6 in Listing 7.1), following a pre-determined algebra.

As discussed earlier in Section 7.2, current resource management and job scheduling in grids do not consider partial requirement satisfaction, employing the simple Boolean logic operators to the evaluation of complex requirements.

On the other hand, since STARC allows and represents the partial fulfilment of a requirement (resorting to fuzzy logic values) the operators should take that into consideration. In the same manner as the **Environment Probing Classes** return values between 0.0 and 1.0, these logical operators should return values in the same range, indicating how capable a host is to satisfy the requirements. Using these operators and comparing the value returned from the evaluation of a requirement on different computers, it is possible to find the one(s) more capable: the one(s) with the highest requirement evaluation value(s).

$$\text{and}\ (A, B) = \min\{A, B\}$$
$$\text{or}(A, B) = \max\{A, B\}$$
$$\text{not}(A) = (1 - A)$$

Figure 7.5: Zadeh Logical operators

The use of Zadeh logical operators (Figure 7.5) is a possible solution to the evaluation of the complex requirements.

These operators closely match the usual gradation mechanisms:

- The `or` operator always returns the largest operand. The combined utility is equivalent to the one of the resource providing the highest utility.
- With the `and` operator, the user wants all requirements to be met, so the result of this operator corresponds to the minimum value.
- The operator `not` allows a user to rate a resource negatively. This may happen when a user wants to specify an undesired resource, one that contributes negatively to the final utility.

When compared with ordinary boolean logic operators, the use of fuzzy logic operators allows more information to be coded in the requirement evaluation answer, it is possible to infer the requirement fulfilment level. Nonetheless, it still becomes limitative to use one single set of operators.

STARC proposes an utility algebra that takes user-perceived values of partial-utility into account in a flexible and expressive way. To accomplish that, STARC allows users to select how combined requirements (operator `and`), alternatives (operator `or`), and disapproval (operator `not`) are evaluated. These evaluation rules are coded in different policies (`priority`, `strict`, `balanced`, and `elastic`) described next. Each policy is inspired by a specific scenario and targets a class of intended users (summarized in Table 7.1).

Policy	Intended users	Calculation	AND (aggregation)	OR (adaptation)	NOT (disapproval)
priority	administrators	boolean logic	all fully met	at least one fully met	fail if partial fulfilment
strict	SLA users	Zadeh fuzzy logic	min	max	complement (1-x)
balanced	registered members	geometric average-based	sqrt(product)	max	complement (1-x)
elastic	best-effort	arithmetic average	average	max	complement (1-x)

Table 7.1: STARC aggregate utility evaluation policies

A policy embodies a specific aim or goal on how to combine available resources to fulfil resource discovery requirements. This goal is implemented in the way aggregate utilities are calculated in order to evaluate how a set of available resources satisfies a given request (i.e., a numeric measure of combined satisfaction).

The greatest difference between policies resides in the way conjunction of requirements is calculated. This difference in the `and` operator rises from the fact that the same requirement applied to the same combination of resources has different evaluations depending on the Policy and user class. This stems from the fact that different users are satisfied in different levels from the same resources (even if they only partially fulfil the requirements). This is illustrated in Figure 7.6.

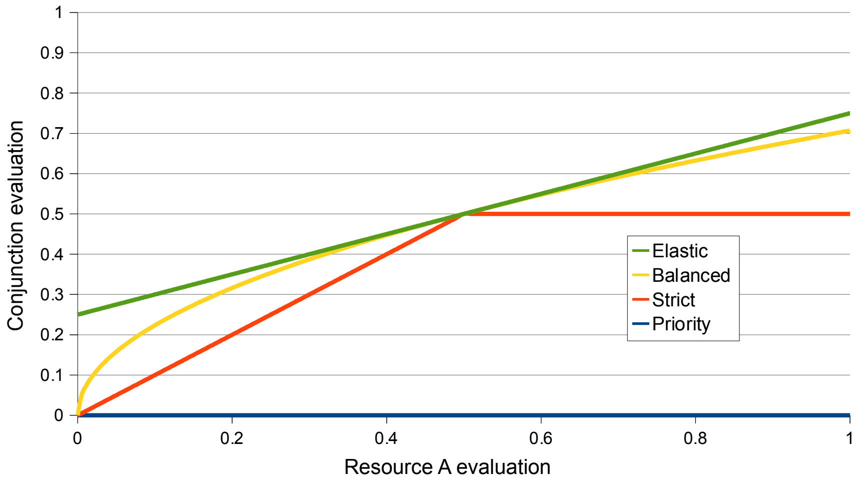


Figure 7.6: Evaluation of a conjunction of two resources (A and B) using multiple Policies (Resource B evaluated with 0.5)

What should be noted in this figure, is the fact that for a certain combination of resources, the evaluation made by an elastic policy is higher than by any other stricter policy. In the other end, in this case, the Priority policy renders 0.0 (the lowest possible value).

The description of the rational behind each policy is in the following paragraphs:

- **Priority:** This policy enforces the complete fulfilment by the resources of the user-supplied requirements. It aims at satisfying requirements that can not be not representable by utility functions, and when a user is not willing to accept hosts where one or more of the required resources are only partially fulfilled.

  In this case the `and` operator has the semantic of its boolean counterpart: The answer is 1.0 only if all operands are fully met (evaluated with the value 1.0). Requirements combined with operator `or` will be tried in sequence by order of appearance (priority) in the XML file, and will return 1.0 when at least one of the operands is fully met. Operator `not` results in the rejection of any host where the resource is available, even partially.

  This is the most rigid of evaluation policies and should only be used in service-critical scenarios, and is equivalent to the policies available in other systems.

- **Strict:** This policy aims at minimizing dissatisfaction on every resource requirement, but still representing the utility resources composing the requirement. Thus for a given set of resources, when complete fulfilment is not possible, the resulting utility will use directly the partial utility of the resource by means of the fuzzy logic Zadeh [Zad65] operators depicted in Figure 7.5 . In the case of the conjunction of requirements, the combined evaluation will result in the lowest of the combined operands, meaning that all other resources have a partial utility higher than the returned value. The operator `or` returns the maximal partial utility in a set of requirements, and the operator `not` returns the complementary utility (1-x), in a way to invert a combined utility.

  The use of these operators allows not only a precise representation of partial fulfilment. Also, from the complex requirement and the results, the user can infer the levels of fulfilment of the resources, and easily compare multiple hosts.

  A common scenario for this policy would be the case of users with service-level agreements that, when a single requirement is partially unmet, impose penalties in the aggregate utility (once again due to `and` returning the minimum).

- **Balanced:** This policy aims at giving higher requirement evaluations to the most balanced hosts, that is, those whose combination of available resources satisfies user requests in a more favourable and balanced manner. Although the `or` and `not` operator follow the Zadeh definition (Figure 7.5), in order to calculate the conjunction, this policy is based on geometric averages: the operator `and` will return the square root of the product of all involved partial utilities.

  This way, the more a resource requirement is far from 1.0 (complete fulfilment) the higher is (proportionally) the decrease on the result combined utility. With this policy, hosts that can fully satisfy certain requirements but can only produce a very poor alternative in others, will not be so highly considered. Hosts that partially satisfy all requirement at a good value will be preferred instead. Of course, hosts with well balanced yet low availability in most/all resources should still be awarded lower utility.

A common scenario for this policy would be the case of registered users that, when a resource is partially unmet, are imposed a penalty in to the aggregate utility, but taking into account that the returned value is higher than if evaluated using the strict policy. Typical users will be regular members of a virtual organization such as in a Grid.

- **Elastic:** This policy aims at maximizing engagement of resources by the system, employing an eager-like approach assuming full elasticity or resources: any non-satisfied resource can be compensated by a good utility in another resource. This is usually not the case since resources are not interchangeable (e.g., CPU for memory and vice-versa). Thus, this policy simply offers a best-effort approach to requirement fulfilling, that may be suitable for non-registered users in a peer-to-peer cycle sharing scenario.

  Combined evaluation is based on simple arithmetic averages. Operator `and` will return the averaged sum of all partial utilities and operator `not` its opposite (additive inverse). With this approach, the system can take advantage of any resource because unfulfillment of a requirement can never bring down the utility value resulting from a combined requirement evaluation. Nonetheless, hosts with better resources will always render better requirement evaluation.

Policy	Calculation	0.5 and 1.0	0.5 and 0.5	0.9 and 0.2

Table 7.2 (Continues on next page)

Policy	Calculation	0.5 and 1.0	0.5 and 0.5	0.9 and 0.2
Priority	Boolean logic	0	0	0
Strict	Zadeh logic	0.5	0.5	0.2
Balanced	Geometric	0.70	0.5	0.42
Elastic	Arithmetic	0.75	0.5	0.55

Table 7.2: STARC evaluation policies for distinct resources

Table 7.2 show how different resources are evaluated by the different policies.

As expected, with respect to the priority policy, none of the examples returns a positive value, since in none of the examples all the requirements are completely met. When using a Strict policy, it is possible to observe that the first two examples return the same evaluation value. This comes from the fact that, of the required resources, the lowest one has the same value (0.5). Using this policy, while evaluating a conjunction, the highest partial requirement is not taken into consideration. This is evident in the last example: the utility of the conjunction of a good and a bad resource is low. In the case of the Balanced policy it is possible to observe that the host with the more balanced resources (0.5 and 0.5) is better evaluated than the one with low quality resources (third example). The Elastic policy, as described earlier, evaluates resources as interchangeable, thus evaluating better a host from the last column than the one from the fourth.

It is also possible to observe that the ordering of the evaluations remains. For each example, the evaluation by different policies is sorted as follow:

$Evaluation(request, Priority) \leq Evaluation(request, Strict) \leq$

$Evaluation(request, Balanced) \leq Evaluation(request, Elastic)$

This is backed by the intuitive notion that, for users with lower or less rights, the evaluation of a resource renders a higher value meaning that that resource, even of low quality, is perceived as of more value to this less privileged user (i.e., he is less demanding). In a similar manner, a resource that, evaluated with different requirement, renders the same evaluation is of more use to the user with the more strict requirements.

When writing requirements, the user can use the optional policy attribute shown in line 2 from Listing 7.1. Nonetheless, a daemon running at a host may override the selected policies offering different quality-of-service considering user information (e.g., user class, rank, reputation) while notifying the requesting user.

Any requirement can also be weighted to denote a user preference. Using the attribute `weight` (line 3 from Listing 7.1), the user can assign a weight (in $]0.0, 1.0]$). After the evaluation of such requirement, the attained utility value is multiplied by the provided weight.

7.5 Implementation

A prototypical implementation of STARC was developed in Python, using its standard XML processing library to parse the requirement files and generate an internal representation of the logical expressions. All communication is performed by means of the Pyro [Jon11] remote object invocation library. To simplify current implementation, any interaction between an application and its local STARC daemon is also made by means of a Pyro invocation. Although Python was used, any other language that supports dynamic code loading, remote method invocation, and remote code loading could have been used.

In order to ease STARC extensibility, the reflective mechanisms and class loading provided by the Python runtime were used. The reflective mechanisms not only allow the run-time loading of classes but also ease its implementation, since Probing Classes do not require any complex inheritance to be dynamically loaded and executed. This allows any programmer to define new proprietary, user-specific, or compound resources to be evaluated, and software or services to be discovered simply by developing a new Environment Probing Class.

All Environment Probing Classes implement a predefined interface composed of a constructor with no arguments and a method called `evalResource`. This method is invoked by the Requirement Evaluator and receives as a parameter the config XML snippet described in Section 7.4.

The name of Environment Probing Classes is obtained from the `name` element present in the XML (Listing 7.1). This name is used to dynamically find the corresponding module from disk, load it and create the Probbing Class. If not present in the remote host, the Resource Probing Classes can be transferred from the local computer that initiated the requirement evaluation and executed in a restricted safe environment, allowing the evaluation of specific user requirements. This code transfer is backed by the dynamic code service for Pyro. If for some reason the Environment Probing Class associated to a resource can not be executed, the evaluation of that resource requirement will return 0.0, meaning the computer does not have the resources to meet that requirement.

A set of standard Environment Probing Classes were developed. These should be present in every host running STARC allowing the evaluation of a number of most used resources: CPU family, cores and frequency, cache size, memory size, available memory, network link speed available. In the implementation of these classes, as a low level layer, WBEM [Dis10a] (in Windows) and the **/proc** file system [FG91] (in Linux) were used.

7.5.1 Remote Host Discovery

The discovery and maintenance of a set of working hosts is out of the scope of this work, nonetheless it is possible to identify three main network topologies of increasing scale and membership flexibility/variation: i) clusters and LAN; ii) Grid-based virtual organizations, and iii) peer-to-peer cycle-sharing desktop grids.

Cluster/LAN Scenarios In the LAN implementation, the Remote Host Discovery module finds remote computers in the same sub-network, by means of UDP broadcasts. Any other computer discovery protocols could have been used (e.g., Jini, UPnP). When performing a requirement evaluation, the local computer contacts all known remote computers, sends them the requirements and waits for an answer. Each host evaluates the requirements against its resources and returns the resulting partial-utility value. These are combined with host identification in an ordered list by the local STARC daemon. The application requesting the query needs only pick the first in list or iterate it for parallel scheduling.

Grid Infrastructures STARC can be easily modified to become a module within the framework of MDS4 [SPM$^+$06], along with a set of Resource Providers (namely resorting to Hawkeye [CT10] for monitoring node resources and availability). The proposed utility algebra can be integrated into the MDS4 Aggregator Framework (including partial-utility evaluation and policy enforcement).

As mentioned in the related work section, this approach and architecture have been previously evaluated and determined scalable [ZFS07]. Therefore, it is only necessary to ensure that the algebra evaluation does not impose excessive overhead. A similar approach could be pursued for integration with R-GMA [CGM$^+$03] or other meta-schedulers, such as Condor-G [FTL$^+$02].

P2P Cycle Sharing Regarding cycle sharing peer-to-peer infrastructures [TT03], the set of hosts made available to STARC to evaluate should be reduced and not the entire P2P population as this would preclude scalability.

Therefore, STARC can be integrated as an additional service on a peer-to-peer cycle sharing system [VRF07a], in order to evaluate only two sets of hosts: direct neighbours in routing tables, and those returned by a lower-level resource discovery mechanism seeking CPU, memory and bandwidth.

7.5.2 Security

The dynamic loading of probing classes has two different usage scenarios. One where the probing classes are stored in a central repository, but managed and created by one trusted entity. This case is no worst that the use of locally installed probing classes. In the second scenario, middleware users create probing classes to be uploaded. In this case, it is necessary to execute them in a restricted environment, such as a virtual machine, similar to where the scheduled jobs would be deployed.

STARC middleware has a small code footprint (below 500 KB) and can be executed at host nodes within the boundaries of a virtual machine. Therefore, its access to local resources (such as file system or communication) can be limited and configured. This extends to the vast majority of Environment Probing Classes. If one needs to access the native system directly, it can be subject to individual configuration and will not be executed without user's authorization. In order to guarantee a timely response from the probing classes and to prevent denial of service attacks, timeouts are enforced on probing class execution (returning 0.0).

7.6 Evaluation

This section describes the evaluation of STARC, resorting to a qualitative analysis and micro-benchmark performance results.

In qualitative terms, STARC provides a number of advantages when compared to the related work. It allows a more expressive, flexible description of resource requirements for jobs, since it implements a series of new ideas:

- the notion of partial-utility;
- user-specified ranges for non-linear partial-utility;
- multiple requirement evaluation policies (selected by the user or enforced by the middleware)

Existing systems either rely on binary decision on requirement fulfilment (as service-level agreements) not considering partial utility, or adopt solely a system-centric approach by employing complex, pre-defined (not user-specified) utility functions in order to optimize request scheduling based on budget, deadlines, and a binary requirement fulfilment that are not adequate for the intended scenarios.

Regarding performance, a micro-benchmark was designed to evaluate the incurred overhead brought when evaluating a XML requirement file. The measurements consider a LAN setting as a worst-case scenario where the relative overhead of requirement evaluation w.r.t. communication is the highest. In order to measure the requirement evaluation times, a 3.2 GHz Pentium 4 personal computer with 1 Gb of memory was used as a resource provider where all resource evaluation were made. The client computers, where all remote evaluations were initiated is an Apple Ibook with a 800MHz PowerPC processor, 640Mb of memory and the Linux operating system. All computers were connected by a 100Mb switched Ethernet network.

In order to measure a requirement evaluation time, a **simple** requirement (presented in Listing 7.5) stating a minimum necessary amount of memory. A **complex** requirement (a conjunction of 20 **simple** requirements) was also used in this evaluation.

```
<requirement> <resource name="Memory" >
    <config> <physicalMemory>
       <range> <minnumber> 4096 </minnumber> <util> 1.0 </util> </range>
     </physicalMemory> </config>
 </resource> </requirement>
```

Listing 7.5: STARC micro-benchmark XML requirements description

The probing class for this resource was developed resorting to the WBEM for access to the characteristics of the memory.

The first micro-benchmark measurements comprised the evaluation of the simple XML requirement file (Listing 7.5), locally using STARC and a similar evaluation using WBEM [Dis10a] and SNMP [CFSD90]. These results are presented in Table 7.3

	Evaluation time (ms)
Bootstrap	38.2
STARC (loopback)	40.1
WBEM	36.3
SNMP (loopback)	110.0

Table 7.3: STARC local resource evaluation comparison (ms)

From this table it is possible to infer the time to bootstrap and process the requirement XML, as well as the time to actually retrieve the resource information. The value in the first line (Bootstrap) corresponds the bootstrap time before an evaluation: parsing of the XML, creation of the evaluation tree and loading of the necessary probing classes. This time is only about $38.2ms$ and remains in the same order of magnitude when increasing the complexity of the requirement. Since WBEM was used to retrieve the hardware information, it is necessary to compare the time to evaluate the available memory using STARC and WBEM. The incurred overhead is only about $4ms$, an order of magnitude lower that the actual time to retrieve the resource information. STARC is noticeable better that SNMP.

Table 7.4 shows how STARC behaves when executing evaluations on remote computers and how it scales with the complexity of the requirements.

	Local loopback	Remote no upload	Remote upload
STARC (simple)	40.1	41.0	110.0
STARC (complex)	795.3	799.0	878.8
SNMP (simple)	110.1	320.7	
SNMP(complex)	165.5	610.7	

Table 7.4: STARC performance comparison (ms)

The results allow to conclude that STARC scales w.r.t the size of the requirement files (**simple** versus. **complex**) and regarding local and remote evaluation.

The difference between a local and remote evaluation is only evident when code upload is necessary (last column), nonetheless the time to upload a complete probing class is only about $70ms$. Naturally, other Environment Probing Classes will take different time to upload.

Since in a requirement evaluation, the majority of the time is spent accessing the hardware information, it is natural that the time to evaluate a complex request is proportional to the number of resources evaluated. In this case a complex requirement takes close to $800ms$, 20 times more than the simple requirement.

In the case of complex requirements, STARC is noticeable slower than SNMP when performing local evaluations, but this difference disappears if a remote evaluation is required. The difference between the last two columns derives from the necessity to transmit the probing class code to the remote host. In this case, it only takes about $70ms$.

To evaluate how STARC behaves under load, the time to evaluate series of **complex** requirements concurrently was measured. These results are shown on Figure 7.7.

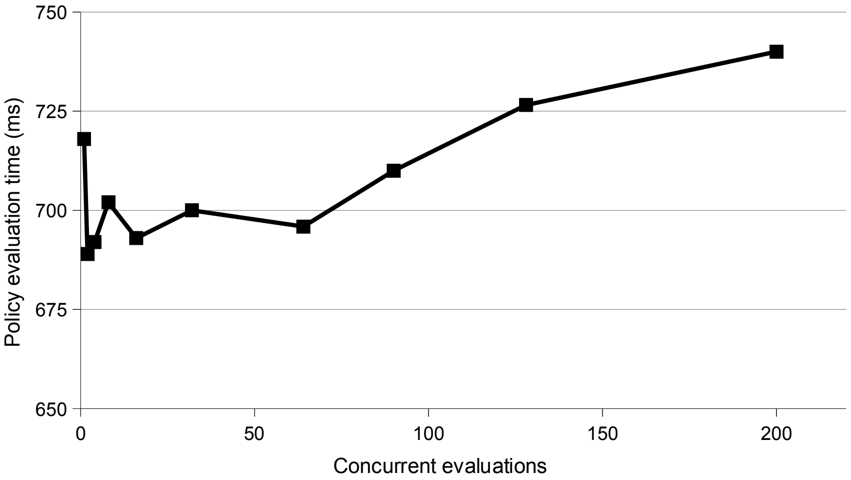


Figure 7.7: Concurrent requirements evaluation overload

The graph clearly shows that until about 100 concurrent evaluations there is no performance degradation. Furthermore, the interleaving of the various processes evaluating requirement policies leads to a reduction of the average processing time. This is naturally due to the operating system being able to better utilize the CPU, by executing some threads while others wait for I/O due to virtual memory.

From that point onwards (about 100 concurrent evaluations) the scheduling and process swapping costs are greater than the gains from the interleaving of the parallel execution. Nonetheless, the middleware maintains scalable behaviour since with 200 concurrent processes performing complex policy evaluations, there is only an increase of about 10% on the average requirement evaluation time. To access the WBEM service, a Pyro RPC server was implemented, which has a limit of 200 active network connections, hence the maximum value concurrent policies tested. It is possible to conclude that requirement evaluation in STARC scales well to large numbers of concurrent clients based on the same conclusions of previous studies [ZFS07].

Since the actual point-to-point communication overhead is low, STARC can also scale to large numbers of participating nodes, when used a P2P infrastructure [VRF07a].

7.7 Conclusions

STARC is capable of evaluating and comparing different resource providers with respect to client specific resource requirements, considering ranges of partial fulfilment and utility depreciation. It is able to evaluate a set of standard resources in different computer architectures: number and speed of CPU cores, available memory, available disk space, network speed, among others.

All these evaluations can be parameterized according to different user profiles and needs. With the proposed utility algebra and corresponding XML DTD, it is possible to define any kind of requirement a module or a complete application may have, and that a resource provider must satisfy.

All resources are evaluated to a satisfiability level between 0.0 and 1.0, in contrast to the usual boolean answers: completely satisfies the requirement or not. By using these numerical results the information about the partial fulfilment of a requirement (partial-utility) can be used in complex requirement definitions.

Although the complex requirements are written using the usual logical operators, their evaluation is not limited to the usual boolean logic. Each user class (Administrator, SLA users, registered members, or best-effort) employs a different complex requirement (conjunction, disjunction and negation) evaluation rule.

By using these operators, the result of the evaluation returns a numeric value that clearly states how the requirements are totally or partially fulfilled, useful in the comparison of multiple hosts.

The architecture of this system allows its extensibility by allowing the definition of new types of resources to be discovered and evaluated. The code to evaluate a resource can be dynamically installed, without system compilation, and reused on behalf of many clients.

8 Conclusion

This document presents a series of contributions to ease the use of parallel programming to commoners. These are users that either have some limited programming knowledge (development of modular sequential code), or use and know how to parameterize some sort of data processing tools (such as image processing, image rendering, statistical packages).

Independently of the computing proficiency, this class of users normally requires the execution of lengthy jobs, but do not have access to existing computational infrastructures. Nonetheless, there are plentiful sources of computational resources to these users. The developed work aims at bridging the existing gap between common users and the flexible and seamless use of available computing sources: personal clusters, the cloud and the whole Internet.

One of the existing sources of computing power is the Internet, and all idle computers that could be used to process data, by means of cycle sharing or Distributed Computing systems. Many systems were developed (most of them presented in Chapter 2), but only one currently exists successfully (BOINC).

From the taxonomy specially developed (described in Chapter 2), and its application to developed and existing systems, it was possible to infer two of the reasons for the failure of most of the systems, and the success of only one: explicit selection of projects and user rankings. In BOINC, users explicitly select the jobs they are willing to donate cycles to, furthermore, users are rewarded symbolic points for the execution of tasks. These two characteristics leverage both the altruistic and competitive nature of human feelings. Users donate cycles to projects they think are worthy and at the same time publicize its donors.

The problem with the definition of tasks by less knowledgeable users was tackled and described in Chapter 3. The work presented in this chapter takes as an assumption, the use of off-the self applications as task execution environment. A simple user interface was developed to help the parametrization of the different tasks using simple mechanisms. In a declarative way, the user is capable to define what files are to be processed by each task, along with the tasks' parameters. The developed user interface allows the definition of the usual data partitioning schemes existing in Bag-of-Tasks problems: different file per task, different arguments per task (parameter sweeping), partition of a numerical data set (uni or bidimensional).

These task definition mechanisms were integrated with SPADE, a simple scheduler of bag-of-tasks to LANs. SPADE allows the execution of tasks whose execution engine is a previously installed application in remote computers scattered in a LAN. With respect to configuration needs, SPADE is lightweight, since it is to be executed in a restricted environment. Remote computer announcements are accomplished by means of periodic broadcast, eliminating any directory service. The processing software are off-the-shelf applications that were previously installed, and no security mechanisms are required, since all the infrastructure is controlled by the owner of the LAN.

Another form of bag-of-tasks creation, is to programmatically define the code to be executed by each of the tasks. As seen in Chapter 2, some programming models are presented, none of which is suitable to users only capable of writing simple sequential code. In Chapter 4, a transparent parallelization mechanism is presented. Users develop sequential applications, with lengthy processing loops, and the developed middleware parallelizes them, and executes each independent tasks in a thread or remote computer. The system just requires a simple configuration file stating what methods are to be executed concurrently.

This bag-of-tasks definition mechanism requires users to have some programming knowledge, but reduces the burden of developing the parallel version. Furthermore, the user only writes one version of the applications, independently of the execution environment: uniprocessor, shared memory processor (with tasks executed in different threads), or cluster (with tasks executed on different remote computers). The idea of transparently transforming sequential applications can also be applied to other interpreted/managed languages (such as Java, R, Matlab) to help users take advantage of available resources.

One good source for computing power is the Internet, and its characteristics make BOINC a good a starting point for a truly universal Distributed Computing infrastructure, where users donate cycles (already possible) but also create work that is to be executed by others. To achieve that, nuBOINC was developed (Chapter 5).

nuBOINC extends BOINC to allow any user to create jobs that are later executed in remote computers. The main contribution of this work is the use of off-the-shelf applications as execution environments for the tasks. Users are only required to define the input files, and configuration parameters of those off-the-shelf applications. Later, nuBOINC transfers those input files to the donor computers, executes those existing applications, and returns the results.

The use of off-the shelf applications as processing engines has a series of advantages. These are the tools users already use, therefore knowing how to configure them, exist in most areas of knowledge, and allow donors to know for what they are donating cycles to. In nuBOINC, donors are required to have the off-the-shelf application previously installed. Most probably, users already had it installed, use it frequently, and explicitly have chosen to donate cycles to projects using it. This goes in line with the altruistic feeling, since users are tempted to donate cycles to those that have similar problems.

Up until this moment only solutions to the use of LAN clusters and Internet scattered resources were presented. Another available source for computational resources is the cloud, by means of utility computing infrastructures. Currently, several providers offer scalable "pay-as-you go" compute capacity (virtual machine instances). The use of these infrastructures requires only their setup (out of the scope of this work), and any cluster-based job creator and scheduler (such as those presented in Chapters 3 and 4). One relevant distinction of this new resource lies in the fact that all allocated processing time in a utility computing infrastructure is required to be payed, in contrast to the obviously free nature of locally owned clusters or processing power donated in Internet based Distributed Computing systems. Added to this is the fact that the payment units are usually one hour, and that any idle time will also be charged.

In order to use a utility computing infrastructure more efficiently, it is necessary to ensure that the idle time is minimum (so that the charged time is close to the actual processing time) but attaining good speedups. This is possible if users know for how long their jobs will execute, which is improbable. To address this issue, a new heuristic was developed (Chapter 6), that during job execution, decides the optimal number of virtual machines to allocate on the cloud, always without any user intervention.

The existing mechanisms for resource discovery and evaluation (a requirement for the optimal execution of jobs and optimal resource management) have become unfit due to the new user classes, and execution environments. Multiple classes of users may try to use the same resource, the offered resources are more diverse in terms of characteristics, and users are requesting more, and more diverse resources.

In order to allow a more flexible and precise resource and requirement evaluation, a new resource evaluation algebra and middleware was developed (Chapter 7), offering some benefits not currently available, as presented next. The developed middleware (STARC) dynamically loads the resource evaluation code (coded in classes following a predetermined format), thus allowing its extensibility and evaluation of any type of resource. This resource evaluation code returns, not a boolean answer (that only states if the requirement is fulfilled by the resource), but a value between 0.0 and 1.0. This value states the utility of the resource with respect to a requirement: 1.0 completely fulfils, $]0.0, 1.0[$ partially fulfils, 0.0 does not provide the resource (not even partially). Since any particular resource evaluation returns a value in $[0.0, 1.0]$, so should any complex requirement evaluation (expressed in terms of *and*, *or* and *not* operators). This is accomplished using operators distinct from the usual boolean ones.

Furthermore, four different user classes were defined (administrator, SLA user, registered user, best-effort user), and assigned each one a different complex requirement evaluation expression policy, thus allowing for higher expressiveness, since the same resource is evaluated in different manners depending of the user class.

8.1 Future Work

Although each part is *per se* a valid contribution to the parallel and distributed computing research area, they can be further developed, integrated with existing technologies or even more thoroughly evaluated.

8.1.1 Complementary evaluation

The work presented in Sections 3 and 6 can be further evaluated, in order to perceive more usages.

The developed user interfaces presented in Chapter 3, although empirically evaluated, should be submitted to a more formal evaluation, in order to study its real potential, and proposed changes. The nature of the presented work (user interaction) is affected by various factors that only usability tests are capable of detecting and exhaustively demonstrate.

Although the developed heuristic (Chapter 6) presented good results for the selected workloads, it is desirable to evaluate the heuristic with other workloads from different areas. Not only can the heuristic be further validated, but also tasks execution time patterns can be discovered. By evaluating different workloads, with such patterns, it can help to increase of the heuristic efficiency.

8.1.2 Complementary development

Although nuBOINC is capable of hosting parallel jobs, it is still far from full production level. It is still necessary to develop client software to other architectures (MAC OS X and Windows) and add to it a new job definition interface. The work of SPADE can be easily integrated with nuBOINC, in order to create a fully functional truly public Distributed Computing infrastructure. nuBOINC should also be deployed in the Internet to allow users to create their own jobs. This way a real infrastructure can be used to validate and help develop new scheduling algorithms. nuBOINC can also be integrated with the automatic parallelization framework presented in Chapter 4, in order to allow the easy creation of BOINC tasks from Python code.

8.1.3 New research topics

The work presented in Chapter 4 used as inspiration a mobile code infrastructure [SF04]. This previous work can be extended with the help of the utility algebra described in the Chapter 7. A more expressive resource evaluation will allow a more precise evaluation of the resources existent in different hosts, allowing a more efficient code execution. The Utility algebra can be used to select the host where to execute a piece of code, but also to select the class or object more fit to execute on a host.

Bibliography

[A$^+$08] G. Aad et al. The ATLAS Experiment at the CERN Large Hadron Collider. *JINST*, 3:S08003, 2008.

[AAB98a] Yair Amir, Baruch Awerbuch, and Ryan S. Borgstrom. A cost-benefit framework for online management of a metacomputing system. In *ICE '98: Proceedings of the first international conference on Information and computation economies*, pages 140–147, New York, NY, USA, October 1998. ACM.

[AAB98b] Yair Amir, Baruch Awerbuch, and Ryan S. Borgstrom. The java market: Transforming the internet into a metacomputer. Technical report, Johns Hopkins University, 1998.

[AB08] Joel C. Adams and Tim H. Brom. Microwulf: a beowulf cluster for every desk. *SIGCSE Bull.*, 40:121–125, March 2008.

[ABG02] David Abramson, Rajkumar Buyya, and Jonathan Giddy. A computational economy for grid computing and its implementation in the Nimrod-G resource broker. *Future Generation Computer Systems*, 18(8):1061–1074, 2002.

[ACK$^+$02] David P. Anderson, Jeff Cobb, Eric Korpela, Matt Lebofsky, and Dan Werthimer. Seti@home: an experiment in public-resource computing. *Commun. ACM*, 45(11):56–61, 2002.

[AF06] David P. Anderson and Gilles Fedak. The computational and storage potential of volunteer computing. In *IEEE/ACM International Symposium on Cluster Computing and the Grid*, May 2006.

[AFG$^+$09] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy H. Katz, Andrew Konwinski, Gunho Lee, David A. Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. Above the clouds: A berkeley view of cloud computing. Technical Report UCB/EECS-2009-28, EECS Department, University of California, Berkeley, Feb 2009.

[AFG$^+$10] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. A view of cloud computing. *Commun. ACM*, 53:50–58, April 2010.

[AFT99] Y. Aridor, M. Factor, and A. Teperman. cJVM: a Single System Image of a JVM on a Cluster. In *International Conference on Parallel Processing*, volume 411, 1999.

[AFT$^+$00] Yariv Aridor, Michael Factor, Avi Teperman, Tamar Eilam, and Assaf Schuster. Transparently obtaining scalability for java applications on a cluster. *J. of Parallel and Distributed Computing*, 60(10), 2000.

[AG96] Sarita V. Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. *Computer*, 29:66–76, December 1996.

[AG03] Martin Alt and Sergei Gorlatch. Future-based RMI: Optimizing compositions of remote method calls on the grid. *Lecture notes in computer science*, pages 427–430, 2003.

[AKW05] David P. Anderson, Eric Korpela, and Rom Walton. High-performance task distribution for volunteer computing. In *E-SCIENCE '05: Proceedings of the First International Conference on e-Science and Grid Computing*, pages 196–203. IEEE Computer Society, 2005.

[All11] The Globus Alliance. Resource management (gram). *http://www.globus.org/toolkit/docs/2.4/gram/*, 2011.

[Ama11] Amazon Web Services LLC. Amazon elastic compute cloud (amazon ec2). *http://aws.amazon.com/ec2*, January 2011.

[And04] David P. Anderson. Boinc: A system for public-resource computing and storage. In *GRID '04: Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing*, pages 4–10, Washington, DC, USA, 2004. IEEE Computer Society.

[And07] David P. Anderson. Local scheduling for volunteer computing. *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–8, 26-30 March 2007.

[App05] Apple Computer, Inc. Xgrid - the simple solution for distributed computing. *http://apple.com/macosx/features/xgrid/*, 2005.

[App07] Apple Inc. Xgrid programming guide. *http://developer.apple.com/library/mac/documentation/MacOSXServer/Conceptual/X-grid_Programming_Guide/Xgrid_Programming_Guide.pdf*, October 2007.

[ASGH95] David Abramson, Roc Sosic, Jon Giddy, and B. Hall. Nimrod: A tool for performing parametised simulations using distributed workstations. In *The 4th IEEE Symposium on High Performance Distributed Computing*, August 1995.

[Ass10] InfiniBand Trade Association. Introduction to infiniband^{TM} for end users. *http://members.infinibandta.org/kwspub/Intro_to_IB_for_End_Users.pdf*, 2010.

[BAGS02] Rajkumar Buyya, David Abramson, Jonathan Giddy, and Heinz Stockinger. Economic models for resource management and scheduling in Grid computing. *Concurrency and Computation: Practice and Experience*, 14(13-15):1507–1542, 2002.

[Bar10] Blaise Barney. Introduction to parallel computing. *https://computing.llnl.gov/tutorials/parallel_comp/*, December 2010.

[BAS04] Ashwin R. Bharambe, Mukesh Agrawal, and Srinivasan Seshan. Mercury: supporting scalable multi-attribute range queries. *SIGCOMM Comput. Commun. Rev.*, 34:353–366, August 2004.

[BAV05] Rajkumar Buyya, David Abramson, and Srikumar Venugopal. The Grid Economy. *Proceedings of the IEEE*, 93(3):698–714, 2005.

[BB99] Mark Baker and Rajkumar Buyya. Cluster computing: the commodity supercomputer. *Softw. Pract. Exper.*, 29:551–576, May 1999.

[BBB96] J. Eric Baldeschwieler, Robert D. Blumofe, and Eric A. Brewer. Atlas: an infrastructure for global computing. In *EW 7: Proceedings of the 7th workshop on ACM SIGOPS European workshop*, pages 165–172, New York, NY, USA, 1996. ACM.

[BBC02] L. Baduel, F. Baude, and D. Caromel. Efficient, flexible, and typed group communications in java. In *Proc. of the ACM-ISCOPE conf. on Java Grande*, pages 28–36. ACM, 2002.

[BBC$^+$06] Laurent Baduel, Françoise Baude, Denis Caromel, Arnaud Contes, Fabrice Huet, Matthieu Morel, and Romain Quilici. *Grid Computing: Software Environments and Tools*, chapter Programming, Deploying, Composing, for the Grid. Springer-Verlag, January 2006.

[BBH$^+$06] Aurelien Bouteiller, Hinde Lilia Bouziane, Thomas Hérault, Pierre Lemarinier, and Franck Cappello. Hybrid preemptive scheduling of mpi applications on the grids. *In Int. Journal of High Performance Computing Special issue*, 20:77–90, 2006.

[BCD$^+$97] GS Blair, G. Coulson, N. Davies, P. Robin, and T. Fitzpatrick. Adaptive middleware for mobile multimedia applications. In *Network and Operating System Support for Digital Audio and Video, 1997., Proceedings of the IEEE 7th International Workshop on*, pages 245–254, 1997.

[BCHM07] F. Baude, D. Caromel, L. Henrio, and M. Morel. Collective interfaces for distributed components. In *CCGRID 2007, IEEE Int. Symposium on Cluster Computing and the Grid,*, 2007.

[BDF$^+$03] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Timothy L. Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In Michael L. Scott and Larry L. Peterson, editors, *SOSP*, pages 164–177. ACM, 2003.

[Ber05] Johan Berntsson. G2dga: an adaptive framework for internet-based distributed genetic algorithms. In *GECCO '05: Proceedings of the 2005 workshops on Genetic and evolutionary computation*, pages 346–349. ACM, 2005.

[BFM05] Tim Berners-Lee, Roy Fielding, and Larry Masinter. *Uniform Resource Identifier (URI): Generic Syntax*. Network Working Group, 2005.

[BG02] Gordon Bell and Jim Gray. What's next in high-performance computing? *Commun. ACM*, 45:91–95, February 2002.

[BJK$^+$95] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: an efficient multithreaded runtime system. *SIGPLAN Not.*, 30(8):207–216, 1995.

[BK88] D.G. Bobrow and G. Kiczales. The common Lisp object system metaobject kernel: a status report. In *Proceedings of the 1988 ACM conference on LISP and functional programming*, pages 309–315. ACM New York, NY, USA, 1988.

[BKKK97] Arash Baratloo, Mehmet Karaul, Holger Karl, and Zvi M. Kedem. Knittingfactory: An infrastructure for distributedweb applications. Technical Report TR 1997ñ748, Courant Institute of Mathematical Sciences - New York University, November 1997.

[BKKK98] Arash Baratloo, Mehmet Karaul, Holger Karl, and Zvi M. Kedem. An infrastructure for network computing with Java applets. *Concurrency: Practice and Experience*, 10(11–13):1029–1041, 1998.

[BKKW99] A. Baratloo, M. Karaul, Z. M. Kedem, and P. Wijckoff. Charlotte: metacomputing on the web. *Future Gener. Comput. Syst.*, 15(5-6):559–570, 1999.

[Boa08] OpenMP Architecture Review Board. Openmp application program interface. *http://www.openmp.org/mp-documents/spec30.pdf*, May 2008.

[BOI09] Running apps in virtual machines. *http://boinc.berkeley.edu/trac/wiki/VmApps*, 2009.

[BOI11] Boincstats. *http://boincstats.com/stats/project_graph.php*, 2011.

[BPK$^+$99] Henri E. Bal, Aske Plaat, Thilo Kielmann, Jason Maassen, Rob van Nieuwpoort, and Ronald Veldema. Parallel computing on wide-area clusters: the albatross project. In *In Extreme Linux Workshop*, pages 20–24, 1999.

[BSST96] Tim Brecht, Harjinder Sandhu, Meijuan Shan, and Jimmy Talbot. Paraweb: towards world-wide supercomputing. In *EW 7: Proceed-*

ings of the 7th workshop on ACM SIGOPS European workshop, pages 181–188. ACM, 1996.

[C$^+$08] The LHCb Collaboration et al. The LHCb detector at the LHC. *Journal of Instrumentation*, 3(08):S08005, 2008.

[CA06] Javier Celaya and Unai Arronategui. Ya: Fast and scalable discovery of idle cpus in a p2p network. In *GRID '06: Proceedings of the 7th IEEE/ACM International Conference on Grid Computing*, pages 49–55. IEEE Computer Society, September 2006.

[CBK$^+$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, February 2008.

[CBS$^+$03] Walfredo Cirne, Francisco Brasileiro, Jacques Sauvé, Nazareno Andrade, Daniel Paranhos, Elizeu Santos-neto, Raissa Medeiros, and Federal Campina Gr. Grid computing for bag of tasks applications. In *In Proc. of the 3rd IFIP Conference on E-Commerce, E-Business and EGovernment*, 2003.

[CCEB03] Andrew Chien, Brad Calder, Stephen Elbert, and Karan Bhatia. Entropia: architecture and performance of an enterprise desktop grid system. *J. Parallel Distrib. Comput.*, 63:597–610, May 2003.

[CCI$^+$97] Bernd O. Christiansen, Peter Cappello, Mihai F. Ionescu, Michael O. Neary, Klaus E. Schauser, and Daniel Wu. Javelin: Internet-based parallel computing using java. *Concurrency: Practice and Experience*, 9:1139–1160, 1997.

[CCNS97] P. Cappello, B. Christiansen, M. Neary, and K. Schauser. Market-based massively parallel internet computing. In *Third Working Conf. on Massively Parallel Programming Models*, pages 118–129, Nov 1997.

[CFCS03] Min Cai, Martin Frank, Jinbo Chen, and Pedro Szekely. Maan: A multi-attribute addressable network for grid information services. In *Proceedings of the 4th International Workshop on Grid Computing*, GRID '03, pages 184–, Washington, DC, USA, 2003. IEEE Computer Society.

[CFK$^+$98] K. Czajkowski, I. Foster, N. Karonis, C. Kesselman, S. Martin, W. Smith, and S. Tuecke. A resource management architecture for metacomputing systems. *Lecture Notes in Computer Science*, 1459:62–82, 1998.

[CFSD90] J. Case, M. Fedor, M. Schoffstall, and J. Davin. *RFC 1157: The Simple Network Management Protocol*. Internet Activities Board, 1990.

[CGM$^+$03] A. Cooke, A.J.G. Gray, L. Ma, W. Nutt, J. Magowan, M. Oevers, P. Taylor, R. Byrom, L. Field, S. Hicks, et al. R-GMA: An information integration system for grid monitoring. *Lecture Notes in Computer Science*, pages 462–481, 2003.

[CGMW03] R. Chinnici, M. Gudgin, J.J. Moreau, and S. Weerawarana. Web services description language (WSDL) version 1.2 part 1: Core language. *World Wide Web Consortium, Working Draft WD-wsdl12-20030611*, 2003.

[CH07] M. Cai and K. Hwang. Distributed aggregation algorithms with load-balancing for scalable grid resource monitoring. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1 –10, march 2007.

[CJGJ78] EG Coffman Jr, MR Garey, and DS Johnson. An application of bin-packing to multiprocessor scheduling. *SIAM Journal on Computing*, 7:1, 1978.

[CK89] William W. Cotterman and Kuldeep Kumar. User cube: a taxonomy of end users. *Commun. ACM*, 32:1313–1320, November 1989.

[CKB$^+$07] SungJin Choi, HongSoo Kim, EunJoung Byun, MaengSoon Baik, SungSuk Kim, ChanYeol Park, and ChongSun Hwang. Characterizing and classifying desktop grid. *Cluster Computing and the Grid, 2007. CCGRID 2007. Seventh IEEE International Symposium on*, pages 743–748, May 2007.

[CKBH06] Sungjin Choi, Hongsoo Kim, Eunjung Byun, and Chongsun Hwang. A taxonomy of desktop grid systems focusing on scheduling. Technical Report KU-CSE-2006-1120-01, Department of Computer Science and Engeering , Korea University, November 2006.

[CKK$^+$99] Steve J. Chapin, Dimitrios Katramatos, John Karpovich, Andrew Grimshaw Karpovich, and Andrew Grimshaw. Resource management in Legion. *Future Generation Computer Systems*, 15(5-6):583–594, 1999.

[CL06] L. Chunlin and L. Layuan. QoS based resource scheduling by computational economy in computational grid. *Information Processing Letters*, 98(3):119–126, 2006.

[CLM05] Xing Cai, Hans Petter Langtangen, and Halvard Moe. On the performance of the python programming language for serial and parallel scientific computations. *Sci. Program.*, 13:31–56, January 2005.

[CLZB00] H. Casanova, A. Legrand, D. Zagorodnov, and F. Berman. Heuristics for scheduling parameter sweep applications in grid environments. *Heterogeneous Computing Workshop, 2000. (HCW 2000) Proceedings. 9th*, pages 349–363, 2000.

[CM02] Peter Cappello and Dimitrios Mourloukos. Cx: A scalable, robust network for parallel computing. *Scientific Programming*, 10(2):159–171, 2002.

[cor08] *ARM11 MPCore^{TM} Processor Technical Reference Manual*. ARM Ltd., 2008.

[CS07] C. A. Cunha and J. L. Sobral. An annotation-based framework for parallel computing. In *Proceedings of the 15th Euromicro International Conference on Parallel, Distributed and Network-Based Processing*, PDP '07, pages 113–120, Washington, DC, USA, 2007. IEEE Computer Society.

[CT10] University of Wisconsin-Madison Condor Team, Computer Sciences Department. Hawkeye a monitoring and management tool for distributed systems. *http://www.cs.wisc.edu/condor/hawkeye*, 2010.

[Daw83] John M. Dawson. Particle simulation of plasmas. *Rev. Mod. Phys.*, 55(2):403–447, Apr 1983.

[Dev10] Advanced Micro Devices. Family 10h amd phenom^{TM} ii processor product data sheet. *http://support.amd.com/us/Processor_TechDocs/46878.pdf*, April 2010.

[DG04] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI*, pages 137–150, 2004.

[DH00] M. Dharsee and C.W.V. Hogue. Mobidick: a tool for distributed computing on the internet. *Heterogeneous Computing Workshop, 2000. (HCW 2000) Proceedings. 9th*, pages 323–335, 2000.

[Dis10a] Distributed Management Task Force, Inc. DMTF CIM operations over http. *http://www.dmtf.org/standards/wbem*, 2010.

[Dis10b] Distributed Management Task Force, Inc. DMTF CIM specification. *Phttp://www.dmtf.org/standards/cim*, 2010.

[EH08] Constantinos Evangelinos and Chris N. Hill. Cloud computing for parallel scientific hpc applications: Feasibility of running coupled atmosphere-ocean climate models on amazon's ec2. In *Proceedings of Cloud Computing and its Applications*, http://www.cca08.org, October 2008.

[EKJ$^+$05] U. Egede, K.Harrison, R.W.L. Jones, A. Maier, J.T. Moscicki, G.N. Patrick, A. Soroko, and C.L. Tan. Ganga user interface for job definition and management. In *Proc. Fourth International Workshop on Frontier Science: New Frontiers in Subnuclear Physics*, Italy, September 2005. Laboratori Nazionali di Frascati.

[Eno08] Enomaly Inc. Enomaly : Elastic computing. *http://enomalism.com*, 2008.

[EVF10] S. Esteves, L. Veiga, and P. Ferreira. Gridp2p: Resource usage in grids and peer-to-peer systems. In *Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, April 2010.

[FC00] Salvatore Filippone and Michele Colajanni. Psblas: a library for parallel linear algebra computation on sparse matrices. *ACM Trans. Math. Softw.*, 26(4):527–550, 2000.

[FDF03] R.J. Figueiredo, P.A. Dinda, and J.A.B. Fortes. A case for grid computing on virtual machines. *Distributed Computing Systems, 2003. Proceedings. 23rd International Conference on*, pages 550–559, May 2003.

[FFK$^+$06] Ian T. Foster, Timothy Freeman, Katarzyna Keahey, Doug Scheftner, Borja Sotomayor, and Xuehai Zhang. Virtual clusters for grid communities. In *CCGRID*, pages 513–520. IEEE Computer Society, 2006.

[FG91] R. Faulkner and R. Gomes. The process file system and process model in unix system v. In *USENIX Winter*, pages 243–252, 1991.

[FGNC01] G. Fedak, C. Germain, V. Neri, and F. Cappello. Xtremweb: a generic global computing system2001. *Cluster Computing and the Grid, 2001. Proceedings. First IEEE/ACM International Symposium on*, pages 582–587, 2001.

[FK99] Ian Foster and Carl Kesselman. *The Grid: Blueprint for a New Computing Infrastructure*, chapter 2 - Computational Grids. Morgan-Kaufman, 1999.

[FKT01] Ian Foster, Carl Kesselman, and Steven Tuecke. The anatomy of the grid: Enabling scalable virtual organizations. *The International Journal of Supercomputer Applications*, 15(3), 2001.

[Fly72] Michael J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, 21(9):948–960, September 1972.

[Fos95] Ian Foster. *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[Fos02] Ian Foster. What is the grid? a three point checklist. *GRID today*, 1(6):32–36, 2002.

[Fos05] I. Foster. Globus toolkit version 4: Software for service-oriented systems. In *IFIP International Conference on Network and Parallel Computing*, pages 2–13. Springer-Verlag, 2005.

[FR95] Dror G. Feitelson and Larry Rudolph. Parallel job scheduling: Issues and approaches. In *Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*, pages 1–18, London, UK, 1995. Springer-Verlag.

[FTL$^+$02] J. Frey, T. Tannenbaum, M. Livny, I. Foster, and S. Tuecke. Condor-G: A computation management agent for multi-institutional grids. *Cluster Computing*, 5(3):237–246, 2002.

[FWP87] Charbel Farhat, Edward Wilson, and Graham Powell. Solution of finite element systems on concurrent processing computers. *Engineering with Computers*, 2:157–165, 1987. 10.1007/BF01201263.

[GBD$^+$94] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. S. Sunderam. Pvm: Parallel virtual machine. a users' guide and tutorial for networked parallel computing. Cambridge, MA, USA, 1994. MIT Press.

[GCLG99] Yaron Goland, Ting Cai, Paul Leach, and Ye Gu. *Simple Service Discovery Protocol/1.0 Operating without an Arbiter*. Internet Engineering Task Force, 1999.

[Gha96] Kourosh Gharachorloo. *Memory consistency models for shared-memory multiprocessors*. PhD thesis, Stanford, CA, USA, 1996. UMI Order No. GAX96-20480.

[GL95] W.W. Gropp and E.L. Lusk. A taxonomy of programming models for symmetric multiprocessors and smp clusters. In *Programming Models for Massively Parallel Computers, 1995*, pages 2 –7, October 1995.

[GLL$^+$90] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. *SIGARCH Comput. Archit. News*, 18:15–26, May 1990.

[GNFC00] Cécile Germain, Vincent Néri, Gilles Fedak, and Franck Cappello. Xtremweb: Building an experimental platform for global computing. In *GRID '00: Proceedings of the First IEEE/ACM International Workshop on Grid Computing*, pages 91–101. Springer-Verlag, 2000.

[Gov03] Chittibabu Govindarajulu. End users: who are they? *Commun. ACM*, 46:152–159, September 2003.

[GS97] Paul A. Gray and Vaidy S. Sunderam. Icet: Distributed computing and java. *Concurrency - Practice and Experience*, 9(11):1161–1167, 1997.

[GS06] Rohit Gupta and Varun Sekhri. Compup2p: An architecture for internet computing using peer-to-peer networks. *IEEE Trans. Parallel Distrib. Syst.*, 17(11):1306–1320, 2006.

[GTLJ07] Paul Grace, Eddy Truyen, Bert Lagaisse, and Wouter Joosen. The case for aspect-oriented reflective middleware. In *ARM '07: Proceedings of the 6th international workshop on Adaptive and reflective middleware*, pages 1–6, New York, NY, USA, 2007. ACM.

[Gut99] Erik Guttman. Service location protocol: Automatic discovery of ip network services. *IEEE Internet Computing*, 3(4):71–80, July 1999.

[GW96] Andrew S. Grimshaw and William A. Wulf. Legion - A view from 50, 000 feet. In *HPDC '96: Proceedings of the High Performance Distributed Computing (HPDC '96)*. IEEE Computer Society, 1996.

[HMF$^+$08] Christina Hoffa, Gaurang Mehta, Tim Freeman, Ewa Deelman, Kate Keahey, Bruce Berriman, and John Good. On the use of cloud computing for scientific workflows. *eScience, IEEE International Conference on*, 0:640–645, 2008.

[HML04] E. Huedo, R.S. Montero, and I.M. Llorente. Experiences on adaptive grid scheduling of parameter sweep applications. *Parallel, Distributed* *and Network-Based Processing, 2004. Proceedings. 12th Euromicro Conference on*, pages 28–33, Feb. 2004.

[HMRT03] B. Haumacher, T. Moschny, J. Reuter, and WF Tichy. Transparent distributed threads for java. In *Parallel and Distributed Processing Symposium, 2003. Proceedings. International*, page 7, 2003.

[IEE11] IEEE-SA Standards Board. 802.3: Csma/cd (ethernet) access method. *http://standards.ieee.org/about/get/802/802.3.html*, January 2011.

[Ima11] ImageMagick Studio LLC. Imagemagick. *http://www.imagemagick.org/*, 2011.

[Int11] *Intel® 64 and IA-32 Architectures Software Developer's Manual - Volume 1: Basic Architecture*. Intel Corporation, 2011.

[Jon11] Irmen de Jong. Pyro - python remote objects. *http://xs4all.nl/ irmen/pyro3*, 2011.

[JS05] Mick Jordan and Christopher Stewart. Adaptive middleware for dynamic component-level deployment. In *ARM '05: Proceedings of the 4th workshop on Reflective and adaptive middleware systems*, New York, NY, USA, 2005. ACM.

[KA99] Yu-Kwong Kwok and Ishfaq Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Comput. Surv.*, 31(4):406–471, 1999.

[Kam07] A. Kaminsky. Parallel Java: A Unified API for Shared Memory and Cluster Parallel Programming in 100% Java. In *IEEE International Parallel and Distributed Processing Symposium, 2007. IPDPS 2007*, pages 1–8, 2007.

[Kar98] Mehmet Karaul. *Metacomputing and Resource Allocation on the World Wide Web*. PhD thesis, New York University, May 1998.

[KBM$^+$00] Thilo Kielmann, Henri E. Bal, Jason Maassen, Rob van Nieuwpoort, Ronald Veldema, Rutger Hofman, Ceriel Jacobs, and Kees Verstoep. The albatross project: Parallel application support for computational grids. In *In Proceedingof the 1st European GRID Forum Workshop*, pages 341–348, 2000.

[KCBC02] Fabio Kon, Fabio Costa, Gordon Blair, and Roy H. Campbell. The case for reflective middleware. *Commun. ACM*, 45(6):33–38, 2002.

[KCZ92] Pete Keleher, Alan L. Cox, and Willy Zwaenepoel. Lazy release consistency for software distributed shared memory. *SIGARCH Comput. Archit. News*, 20:13–21, April 1992.

[KG06] Giannis Kouretis and Fotis Georgatos. Livewn, cpu scavenging in the grid era. *http://arxiv.org/abs/cs/0608045*, arXiv.org,, August 2006.

[KK11] Leyli Mohammad Khanli and Saeed Kargar. Frdt: Footprint resource discovery tree for grids. *Future Gener. Comput. Syst.*, 27:148–156, February 2011.

[KLM$^+$97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Aksit and Satoshi Matsuoka, editors, *ECOOP'97 — Object-Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer Berlin / Heidelberg, 1997. 10.1007/BFb0053381.

[LBRV05a] Akshay Luther, Rajkumar Buyya, Rajiv Ranjan, and Srikumar Venugopal. Alchemi: A .net-based enterprise grid computing system. In *Proceedings of the 6th International Conference on Internet Computing (ICOMP'05)*. CSREA Press, Las Vegas, USA, June 2005.

[LBRV05b] Akshay Luther, Rajkumar Buyya, Rajiv Ranjan, and Srikumar Venugopal. *High Performance Computing: Paradigm and Infrastructure*, chapter Peer-to-Peer Grid Computing and a .NET-based Alchemi Framework. Wiley Press, USA, June 2005.

[LD07] Piotr Luszczek and Jack Dongarra. High performance development for high end computing with python language wrapper (plw). *Int. J. High Perform. Comput. Appl.*, 21:360–369, August 2007.

[LdAL02] Marcelo Lobosco, Claudio Luis de Amorim, and Orlando Loques. Java for high-performance network-based computing: a survey. *Concurrency and Computation: Practice and Experience*, 14(1):1–31, 2002.

[Lei] University of Leiden. Leiden classical. *http://boinc.gorlaeus.net/*.

[LH89] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Trans. Comput. Syst.*, 7:321–359, November 1989.

[LL07] C. Li and L. Li. Utility-based QoS optimisation strategy for multi-criteria scheduling on the grid. *Journal of Parallel and Distributed Computing*, 67(2):142–153, 2007.

[LLM88] Michael Litzkow, Miron Livny, and Matthew Mutka. Condor - a hunter of idle workstations. In *Proceedings of the 8th Intl.Conf. of Distributed Computing Systems*. IEEE Computer Society, June 1988.

[Lon11] J.D. Long. Segue for r: Parallel r in the cloud two lines of code! *http://code.google.com/p/segue/*, January 2011.

[LR00] David Lucking-Reiley. Vickrey auctions in practice: From nineteenth-century philately to twenty-first-century e-commerce. *Journal of Economic Perspectives*, 14(3):183–192, Summer 2000.

[LS07] Sven De Labey and Eric Steegmans. A type system extension for middleware interactions. In *Proceedings of the 1st workshop on Middleware-application interaction: in conjunction with Euro-Sys 2007*, pages 37–42. ACM New York, NY, USA, 2007.

[LSHS05] Cynthia Bailey Lee, Yael Schwartzman, Jennifer Hardy, and Allan Snavely. Are user runtime estimates inherently inaccurate? In *Job Scheduling Strategies for Parallel Processing, 10th International Workshop, JSSPP 2004*. Springer, 2005.

[MB11] Stefano Masini and Paolo Bientinesi. High-performance parallel computations using python as high-level language. In Mario Guarracino, Frédéric Vivien, Jesper Träff, Mario Cannatoro, Marco Danelutto, Anders Hast, Francesca Perla, Andreas Knüpfer, Beniamino Di Martino, and Michael Alexander, editors, *Euro-Par 2010 Parallel Processing Workshops*, volume 6586 of *Lecture Notes in Computer Science*, pages 541–548. Springer Berlin / Heidelberg, 2011.

[MBE$^+$09] J.T. Moscicki, F. Brochu, J. Ebke, U. Egede, J. Elmsheuser, K. Harrison, R.W.L. Jones, H.C. Lee, D. Liko, A. Maier, A. Muraru, G.N. Patrick, K. Pajchel, W. Reece, B.H. Samset, M.W. Slater, A. Soroko, C.L. Tan, D.C. van der Ster, and M. Williams. Ganga: A tool for computational-task management and easy access to grid resources. *Computer Physics Communications*, 180(11):2303 – 2316, 2009.

[McL74] E.R McLean. End users as application developers. In *Guide/Share Application Development Symposium*, October 1974.

[Mes94] Message Passing Interface Forum. MPI: A message-passing interface standard. Technical report, University of Tennessee, Knoxville, TN, USA, 1994.

[MF01] Ahuva W. Mu'alem and Dror G. Feitelson. Utilization, predictability, workloads, and user runtime estimates in scheduling the ibm sp2 with backfilling. *IEEE Trans. Parallel Distrib. Syst.*, 12(6):529–543, 2001.

[Mik05] Henri Mikkonen. Enabling computational grids using jxta-technology. In *Peer-to-peer technologies, networks and systems - Seminar on Internetworking*. Helsinki University of Technology, 2005.

[MK03a] Richard Mason and Wayne Kelly. Peer-to-peer cycle sharing via .net remoting. In *AusWeb 2003. The Ninth Australian World Wide Web Conference*, 2003.

[MK03b] Tanya McGill and Chris Klisc. End user development and the world wide web. Research working paper series IT/03/01, Murdoch University. School of Information Technology, 2003.

[MK05] Richard Mason and Wayne Kelly. G2-p2p: a fully decentralised fault-tolerant cycle-stealing framework. In *ACSW Frontiers '05: Proceedings of the 2005 Australasian workshop on Grid computing and e-research*. Australian Computer Society, Inc., 2005.

[MSDS10] Hans Meuer, Erich Strohmaier, Jack Dongarra, and Horst Simon. Top 500 supercomputers sites. *http://www.top500.org/*, November 2010.

[MSFV11] João Morais, João Silva, Paulo Ferreira, and Luís Veiga. Transparent adaptation of e-science applications for parallel and cycle-sharing infrastructures. In Pascal Felber and Romain Rouvoy, editors, *Distributed Applications and Interoperable Systems*, volume 6723 of *Lecture Notes in Computer Science*, pages 292–300. Springer Berlin / Heidelberg, 2011.

[MW82] Paul R Milgrom and Robert J Weber. A theory of auctions and competitive bidding. *Econometrica*, 50(5):1089–1122, September 1982.

[Myr09] Myricom. Myrinet overview. *http://www.myricom.com/myrinet/overview/*, October 2009.

[NBK+99] Michael O. Neary, Sean P. Brydon, Paul Kmiec, Sami Rollins, and Peter Cappello. Javelin++: scalability issues in global computing. In *JAVA '99: Proceedings of the ACM 1999 conference on Java Grande*, pages 171–180. ACM, 1999.

[NCHL10] Jon K Nilsen, Xing Cai, Bjørn Høyland, and Hans Petter Langtangen. Simplifying the parallelization of scientific codes by a function-centric approach in python. *Computational Science & Discovery*, 3(1):015003, 2010.

[NCS+05] M.A.S. Netto, R.N. Calheiros, R.K.S. Silva, C.A.R. De Rose, C. Northfleet, and W. Cirne. Transparent resource allocation to exploit idle cluster nodes in computational grids. *e-Science and Grid Computing, 2005. First International Conference on*, December 2005.

[Nie11] Ole Nielsen. pypar parallel - programming for python. *http://code.google.com/p/pypar/*, 2011.

[NIM00] High performance parametric modeling with nimrod/g: Killer application for the global grid? In *Proceedings of the 14th International Symposium on Parallel and Distributed Processing*, pages 520–, Washington, DC, USA, 2000. IEEE Computer Society.

[NLRC98] N. Nisan, S. London, O. Regev, and N. Camiel. Globally distributed computation over the internet - the popcorn project. In *ICDCS '98: Proceedings of the The 18th International Conference on Distributed Computing Systems*, page 592. IEEE Computer Society, 1998.

[NPRC00] Michael O. Neary, Alan Phipps, Steven Richman, and Peter R. Cappello. Javelin 2.0: Java-based parallel computing on the internet. In *Euro-Par 2000, Parallel Processing, 6th International Euro-Par Conference, Munich, Germany, August 29 - September 1, 2000, Proceedings*, 2000.

[NWG$^+$08] Daniel Nurmi, Rich Wolski, Chris Grzegorczyk, Graziano Obertelli, Sunil Soman, Lamia Youseff, and Dmitrii Zagorodnov. The eucalyptus open-source cloud-computing system. In *Proceedings of Cloud Computing and Its Applications*, http://www.cca08.org, October 2008.

[oC07] University of California. BOINC WarapperApp – legacy applications. *http://boinc.berkeley.edu/trac/wiki/WrapperApp*, 2007.

[ODDG03] A. Othman, P. Dew, K. Djemamem, and I. Gourlay. Adaptive grid resource brokering. *Cluster Computing, 2003. Proceedings. 2003 IEEE International Conference on*, pages 172–179, Dec. 2003.

[OFV11] Pedro Oliveira, Paulo Ferreira, and Luís Veiga. Gridlet economics: resource management models and policies for cycle-sharing systems. In *Proceedings of the 6th international conference on Advances in grid and pervasive computing*, GPC'11, pages 72–83, Berlin, Heidelberg, 2011. Springer-Verlag.

[OLS02] Licínio Oliveira, Luís Lopes, and Fernando Silva. P^3: Parallel peer to peer - an internet parallel programming environment. *Lecture Notes in Computer Science*, 2376, 2002.

[Ous82] John Ousterhout. Scheduling techniques for concurrent systems. In *Proc. 3rd International Conference on Distributed Computing Systems*, pages 22–30, June 1982.

[Pan88] Raymond R. Panko. *End user computing: management, applications, & technology*. John Wiley & Sons, Inc., New York, NY, USA, 1988.

[Pea11] Kirk Pearson. Distributed computing. *http://www.distributedcomputing.info/*, January 2011.

[Per08] Persistence of Vision Raytracer Pty. Ltd. Persistence of vision raytracer. *http://www.povray.org/*, 2008.

[PiC10] PiCloud, Inc. Picloud | cloud computing. simplified. *http://www.picloud.com/*, 2010.

[PMNP06] Simon Parsons, Marek Marcinkiewicz, Jinzhong Niu, and Steve Phelps. Everything you wanted to know about double auctions but were afraid to (bid or) ask. Technical report, Department of Computer Science, Graduate School and University Center, City University of New York, 2006.

[PSS97] Hernâni Pedroso, Luis M. Silva, and João Gabriel Silva. Web-based metacomputing with JET. *Concurrency: Practice and Experience*, 9(11):1169–1173, 1997.

[Pyt08] Python Software Foundation. Python programming language. *http://python.org/*, 2008.

[R F11] R Foundation. The r project for statistical computing. *http://www.r-project.org/*, 2011.

[Rey10] Christopher J. Reynolds. Distributed video rendering using blender, virtualbox, and boinc. In *The 6th BOINC Workshop*, 2010.

[RF83] John F. Rockart and Lauren S. Flannery. The management of end user computing. *Commun. ACM*, 26:776–784, October 1983.

[RFC$^+$08] César A. F. De Rose, Tiago Ferreto, Rodrigo N. Calheiros, Walfredo Cirne, Lauro B. Costa, and Daniel Fireman. Allocation strategies for utilization of space-shared resources in bag of tasks grids. *Future Gener. Comput. Syst.*, 24(5):331–341, 2008.

[RH88] Suzanne Rivard and Sid L. Huff. Factors of success for end-user computing. *Commun. ACM*, 31:552–561, May 1988.

[RLS99] Rajesh Raman, Miron Livny, and Marvin H. Solomon. Matchmaking: An extensible framework for distributed resource management. *Cluster Computing*, 2(2):129–138, June 1999.

[Ros90] J. Rose. *RFC 1158: Management Information Base for network management of TCP/IP-based internets: MIB-II*. Internet Activities Board, 1990.

[RR99] Radu Rugina and Martin Rinard. Automatic parallelization of divide and conquer algorithms. *SIGPLAN Not.*, 34:72–83, May 1999.

[RRV10] P.D. Rodrigues, C. Ribeiro, and L Veiga. Incentive mechanisms in peer-to-peer networks. In *Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, pages 1–8, April 2010.

[Sar01] Luis F. G. Sarmenta. *Volunteer Computing*. PhD thesis, Massachusetts Institute of Technology, June 2001.

[Sch07] Bertil Schmidt. A survey of desktop grid applications for e-science. *International Journal of Web and Grid Services*, 3(3):354–368, 2007.

[SCI11] Python for scientific computing conference. http://conference.scipy.org/scipy2011/talks.php, Austin, texas, USA, July 2011.

[SCM07] João Sobral, Carlos Cunha, and Miguel Monteiro. Aspect oriented pluggable support for parallel computing. In Michel Daydé, José Palma, Álvaro Coutinho, Esther Pacitti, and João Lopes, editors, *High Performance Computing for Computational Science - VECPAR 2006*, volume 4395 of *Lecture Notes in Computer Science*, pages 93–106. Springer Berlin / Heidelberg, 2007. 10.1007/978-3-540-71351-7_8.

[SCS$^+$08] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, and Pat Hanrahan. Larrabee: a many-core x86 architecture for visual computing. *ACM Trans. Graph.*, 27:18:1–18:15, August 2008.

[Seg10] Ben Segal. Boinc-vm and volunteer cloud computing. In *The 6th BOINC Workshop*, 2010.

[Sek05] Varun Sekhri. Compup2p: A light-weight architecture for internet computing. Master's thesis, Iowa State University, Ames, Iowa, 2005.

[SF04] João Nuno Silva and Paulo Ferreira. Remote code execution on ubiquitous mobile applications. In *Ambient Intelligence: Second European Symposium, EUSAI 2004, Eindhoven, The Netherlands, November 8-11, 2004. Proceedings*, pages 172–183. Springer-verlag, 2004.

[SFV10] João Nuno Silva, Paulo Ferreira, and Luís Veiga. Service and resource discovery in cycle-sharing environments with a utility algebra. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1 –11, 2010.

[SFV11] João Nuno Silva, Paulo Ferreira, and Luís Veiga. A2ha - automatic and adaptive host allocation in utility computing for bag-of-tasks. Accepted for publication on JISA - Journal of Internet Services and Applications, Springer, 2011.

[SG00] João Nuno Silva and Paulo Guedes. Ship hull hydrodynamic analysis using distributed shared memory. In *Applied Parallel Computing, New Paradigms for HPC in Industry and Academia, 5th International Workshop, PARA 2000 Bergen, Norway, June 18-20, 2000 Proceedings*, volume 1947 of *Lecture Notes in Computer Science*, pages 366–372. Springer, 2000.

[SH99] Luis F. G. Sarmenta and Satoshi Hirano. Bayanihan: building and studying Web-based volunteer computing systems using Java. *Future Generation Computer Systems*, 15(5–6):675–686, 1999.

[She09] Haiying Shen. A p2p-based intelligent resource discovery mechanism in internet-based distributed systems. *J. Parallel Distrib. Comput.*, 69:197–209, February 2009.

[Sil03] João Nuno Silva. Optimização e avaliação de aplicações de simulações em sistemas paralelos. Master's thesis, Instituto Superior Técnico, 2003.

[Sim09] SimPy Developer Team. Simpy homepage. *http://simpy.sourceforge.net/*, 2009.

[SK86] Mary Sumner and Robert Klepper. End-user application development: practices, policies, and organizational impacts. In *Proceedings of the twenty-second annual computer personnel research conference on Computer personnel research conference*, SIGCPR '86, pages 102–116, New York, NY, USA, 1986. ACM.

[SKF08] Borja Sotomayor, Kate Keahey, and Ian T. Foster. Combining batch execution and leasing using virtual machines. In Manish Parashar, Karsten Schwan, Jon B. Weissman, and Domenico Laforenza, editors, *HPDC*, pages 87–96. ACM, 2008.

[Ski88] David B. Skillicorn. A taxonomy for computer architectures. *Computer*, 21:46–57, November 1988.

[SMK$^+$01] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer commu-*

nications, SIGCOMM '01, pages 149–160, New York, NY, USA, 2001. ACM.

[Som07] M.F. Somers. Leiden grid infrastructure. *http://fwnc7003.leidenuniv.nl/LGI/docs/*, 2007.

[SPM+06] J.M. Schopf, L. Pearlman, N. Miller, C. Kesselman, I. Foster, M. DíArcy, and A. Chervenak. Monitoring the grid with the Globus Toolkit MDS4. In *Journal of Physics: Conference Series*, volume 46, pages 521–525. Institute of Physics Publishing, 2006.

[SPS97] Luis M. Silva, Hernâni Pedroso, and João Gabriel Silva. The design of jet: A java library for embarrassingly parallel applications. In A. Bakkers, editor, *Parallel programming and Java: WoTUG-20*, pages 210–228. IOS Press, 1997.

[SSBS99] Thomas L. Sterling, John Salmon, Donald J. Becker, and Daniel F. Savarese. *How to build a Beowulf: a guide to the implementation and application of PC clusters.* MIT Press, Cambridge, MA, USA, 1999.

[SSP+97] Klaus E. Schauser, Chris J. Scheiman, Gyung-Leen Park, Behrooz Shirazi, and Jeff Marquis. Superweb: Towards a global web-based parallel computing infrastructure. In *IPPS '97: Proceedings of the 11th International Symposium on Parallel Processing*, pages 100–106. IEEE Computer Society, 1997.

[SSRP09] William R. Scullin, James B. Snyder, Nick Romero, and Massimo Di Pierro. Python for scientific and high python for scientific and high performance computing. Tutorial at 2009 International Conference on High Performance Computing, Networking, Storage and Analysis - SC09, 2009.

[Ste99] W. Richard Stevens. *UNIX network programming, volume 2 (2nd ed.): interprocess communications*, chapter Part 4. Shared Memory. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1999.

[Ste09] Aad J. van der Steen. *Overview of recent supercomputers*. Dutch National Computer facilities Foundation, 2009.

[STS05] K. Shudo, Y. Tanaka, and S. Sekiguchi. P3: P2p-based middleware enabling transfer and aggregation of computational resources. *Cluster Computing and the Grid, 2005. CCGrid 2005. IEEE International Symposium on*, 1:259–266, May 2005.

[SV96] Sartaj Sahni and George Vairaktarakis. The master-slave paradigm in parallel computer and industrial settings. *Journal of Global Optimization*, 9:357–377, 1996. 10.1007/BF00121679.

[SVF08a] João Nuno Silva, Luís Veiga, and Paulo Ferreira. Heuristic for resources allocation on utility computing infrastructures. In *Proceedings of the 6th international workshop on Middleware for grid computing, held at the ACM/IFIP/USENIX International Middleware Conference*, MGC '08, pages 9:1–9:6, New York, NY, USA, 2008. ACM.

[SVF08b] João Nuno Silva, Luís Veiga, and Paulo Ferreira. nuboinc: Boinc extensions for community cycle sharing. *IEEE International Conference on Self-Adaptive and Self-Organizing Systems Workshops*, 0:248–253, 2008.

[SVF08c] João Nuno Silva, Luís Veiga, and Paulo Ferreira. Spade: scheduler for parallel and distributed execution from mobile devices. In *Proceedings of the 6th international workshop on Middleware for pervasive and ad-hoc computing, held at the ACM/IFIP/USENIX International Middleware Conference*, MPAC '08, pages 25–30, New York, NY, USA, 2008. ACM.

[SVF09] João Nuno Silva, Luís Veiga, and Paulo Ferreira. Mercury: a reflective middleware for automatic parallelization of bags-of-tasks. In *Proceedings of the 8th International Workshop on Adaptive and Reflective MIddleware, held at the ACM/IFIP/USENIX International Middleware Conference*, ARM '09, pages 1:1–1:6, New York, NY, USA, 2009. ACM.

[SYAD05] K Seymour, A YarKhan, S Agrawal, and J Dongarra. Netsolve: Grid enabling scientific computing environments. In *Grid Computing and New Frontiers of High Performance Processing*. Elsevier, 2005.

[TS02] E. Tilevich and Y. Smaragdakis. J-orchestra: Automatic java application partitioning. In *ECOOP 2002-Object-Oriented Programming, 16th European Conference, Malaga, Spain, June 10-14, 2002, Proceedings*, volume 2374, pages 178–204, 2002.

[TT03] D. Talia and P. Trunfio. Toward a Synergy Between P2P and Grids. *Internet Computing*, 7(4):51–62, 2003.

[VDB05] Koen Vanthournout, Geert Deconinck, and Ronnie Belmans. A taxonomy for resource discovery. *Personal Ubiquitous Comput.*, 9(2):81–89, 2005.

[VNRS02] Jerome Verbeke, Neelakanth Nadgir, Greg Ruetsch, and Ilya Sharapov. Framework for peer-to-peer distributed computing in a heterogeneous, decentralized environment. In *GRID '02: Proceedings of the Third International Workshop on Grid Computing*, pages 1–12. Springer-Verlag, 2002.

[VRF07a] L. Veiga, R. Rodrigues, and P. Ferreira. Gigi: An ocean of gridlets on a "grid-for-the-masses". *Cluster Computing and the Grid, 2007. CCGRID 2007. Seventh IEEE International Symposium on*, pages 783–788, May 2007.

[VRF07b] Luís Veigas Veiga, Rodrigo Rodrigues, and Paulo Ferreira. Gigi: An ocean of gridlets on a "grid-for-the-masses". In *7th IEEE International Symposium on Cluster Computing and the Grid (CCGrid07)*, May 2007.

[VSG11] Luís Veiga, João Nuno Silva, and João Coelho Garcia. *Peer4Peer: E-science Communities for Overlay Network and Grid Computing Research*. Springer, 2011.

[Wei07] Aaron Weiss. Computing in the clouds. *netWorker*, 11:16–25, December 2007.

[WvLY$^+$10] Lizhe Wang, Gregor von Laszewski, Andrew Younge, Xi He, Marcel Kunze, Jie Tao, and Cheng Fu. Cloud computing: a perspective study. *New Generation Computing*, 28:137–146, 2010. 10.1007/s00354-008-0081-5.

[WWW98] Peter R. Wurman, William E. Walsh, and Michael P. Wellman. Flexible double auctions for electronic commerce: theory and implementation. *Decision Support Systems*, 24(1):17 – 27, 1998.

[wxp] wxpython. *http://www.wxpython.org/*.

[Zad65] Lotfi Zadeh. Fuzzy sets. *Information and Control*, (8):338–353, 1965.

[ZFS07] X. Zhang, J.L. Freschl, and J.M. Schopf. Scalability analysis of three monitoring and information systems: MDS2, R-GMA, and Hawkeye. *Journal of Parallel and Distributed Computing*, 67(8):883–902, 2007.

[ZL04a] D. Zhou and V. Lo. Cluster computing on the fly: resource discovery in a cycle sharing peer-to-peer system. *Cluster Computing and the Grid, 2004. CCGrid 2004. IEEE International Symposium on*, pages 66–73, April 2004.

[ZL04b] Dayi Zhou and Virginia Lo. Cluster computing on the fly: Resource discovery in a cycle sharing peer-to-peer system. In *IEEE International Symposium on Cluster Computing and the Grid*, 2004.

[ZS05] S. Zanikolas and R. Sakellariou. A taxonomy of grid monitoring systems. *Future Generation Computer Systems*, 21(1):163–188, 2005.

[ZWL02] Wenzhang Zhu, Cho-Li Wang, and F.C.M. Lau. Jessica2: a distributed java virtual machine with transparent thread migration support. In *Cluster Computing, 2002. Proceedings. 2002 IEEE International Conference on*, pages 381–388, 2002.