Tampereen teknillinen yliopisto. Matematiikan laitos. Tutkimusraportti 95
Tampere University of Technology. Department of Mathematics. Research Report 95

Jarno Rantala & Robert Piché

Software Systems for Distributed Computing

TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

Tampereen teknillinen yliopisto. Matematiikan laitos. Tutkimusraportti 95
Tampere University of Technology. Department of Mathematics. Research Report 95

Jarno Rantala & Robert Piché

Software Systems for Distributed Computing

Tampereen teknillinen yliopisto. Matematiikan laitos
Tampere 2009

ISBN 978-952-15-2245-1 (printed)
ISBN 978-952-15-3302-0 (PDF)
ISSN 1459-3750

ABSTRACT

The computing power of present desktops is mostly unused under general office usage. These many computers can be unified into one grid, such that unused cycles can be scavenged to solve one computing task. Two software systems to build such grids are compared in this work: BOINC and Techila. These systems are critically compared through the computation of two different scientific tasks, making observations of each simultaneously.

BOINC is designed for volunteer computing where anyone can donate his or her computer to the joint computation via an Internet connection. The system has been designed to facilitate millions of client computers, from which the results can not be blindly trusted. In addition, the system holds volunteers' interest in the computational work by giving credits for the processing donated. Also available is a screen saver that is related to the computational workload.

Techila is designed for the internal use of organizations that use scientific computing. In this case only the trusted, organization owned computers are used. The system makes it easy for many users to simultaneously add computation projects. Moreover, the user gets the results and error messages for his or her computations directly to their own computer. This makes it possible to run applications that are still under development in a grid.

This work concludes that BOINC is preferable for use in public projects where there is enough computation for thousands of client computers for months or even years. Techila is better suited for environments where many users want to use the grid for minor computation projects at the same time. Moreover, maintenance can be handled centrally with a web interface. BOINC does not provide such a tool.

CONTENTS

1. Introduction 1
2. DISTRIBUTED COMPUTING 4
2.1 General 4
2.2 Distributing scientific computing to desktop computers 5
2.3 Distributability of computing 6
2.3.1 Distributability in general 6
2.3.2 Well-distributable computing 7
2.3.3 Computing that is not well-suited for distribution 9
2.4 Volunteer computing 11
2.5 Intra-organisation distributed computing 11
2.6 Criteria for the comparison 12
3. BOINC 15
3.1 General 15
3.1.1 Background 15
3.1.2 Structure 16
3.1.3 Job timing 18
3.2 Using BOINC 19
3.2.1 Installation 19
3.2.2 Modifying and running a project 20
3.2.3 Error situations 22
3.2.4 Combining the results 22
3.3 Data security 22
3.4 Maintenance 23
4. Techila 24
4.1 General 24
4.2 Using the Techila system 25
4.2.1 Installation 25
4.2.2 Modifying and running a project 26
4.2.3 Error situations 28
4.2.4 Combining the results 29
4.3 Data security 30
4.4 Maintenance 30
5. THE TESTS 32
5.1 Pricing of options 32
5.1.1 Introduction of the problem 32
5.1.2 The gridification of the problem 33
5.1.3 BOINC 34

5.1.4 Techila 34
5.2 Garfield 35
5.2.1 Introduction of the problem 35
5.2.2 The gridification of the problem 36
5.3 Comparison 37
6. CONCLUSIONS 42
References 45
A.Napoleon MATLAB code 48
B.Computing Napoleon in Techila Grid with peach function 50
C.Garfield input file 54
D.MATLAB main program for running Garfield on Techila 55
E.Program to create Napoleon jobs 61

1. INTRODUCTION

Scientific computing has become a part of modern research alongside traditional experimental and theoretical research. Computing is used, for example, in solving optimisation tasks, visualising results, and simulating experiments [1]. Furthermore, scientific computing has become an important part of product development, because costly and time-consuming experiments can be replaced to a great extent with computer simulations [2]. Computing power of processors has continuously increased, which has made it possible to solve more and more demanding computing tasks. Yet, we need more computing power, as solving present-day computing tasks may take months or even years.

For example, the ClimatePrediction.net project [3] aims to forecast climate changes until the year 2080. This is done by running the same climate change simulations thousands of times with small changes in the parameters. The result is an understanding of how small changes can affect the climate during this century. Running one simulation takes months, so completing the whole project will take hundreds of years of computing time.

Nowadays the computing power of a single processor is harder to increase, which is why the trend is towards the distributing of computing tasks to several processors at once. Modern supercomputers consist of thousands of regular, inexpensive processors, such as those used in normal personal computers. For example, the Cray XT4/XT5, CSC's new supercomputer, has 9424 processor cores and its quad-core processors run at 2,3 GHz [4]. In addition to processors, video (graphic coprocessor) cards are also beginning to be used in scientific computing because they are inexpensive and efficient. The use of video cards has become more widespread as the required libraries have been made available for high-level programming languages such as C++ [5].

As the computing power has increased, so has the speed of the network connections. This makes it possible for the computing nodes to be spread far apart physically, as long as there is a working network connection between them. A system where several computers are grouped together to solve an extensive scientific computing task is called a grid. For example, the Akaatti cluster at the Tampere University of Technology is part of the national M-grid network [6]. In addition, there are international grid networks like DEISA [7], NorduGrid [8], EGEE [9], and

Worldwide LHC Computing Grid [10].

Purchasing and maintaining clusters and supercomputers is very expensive. For example, the fastest supercomputer in Great Britain, HECToR (The High-End Computing Terascale Resource), cost 59.4 million pounds to build, and the maintenance costs are 8.2 million pounds per year [11]. One must also take into consideration that the maintenance costs are dependent on the energy market, as supercomputers require vast amounts of energy for cooling. At HECToR's completion, the yearly costs were estimated to be 5.4 million pounds, but as electricity prices went up, the expenditure rose by almost three million pounds [11]. It is therefore clear that all organisations that need computing resources do not have the financial means to set up and maintain clusters. An alternative is needed to obtain more computing power.

Modern personal computers are already so powerful that most of their resources remain unused in daily use. When the user is reading e-mail or using a text editor, the processor is mostly idle. Utilising this idle time ("cycle scavenging") makes it possible to create grid networks with personal computers. There are many kinds of distributed computing software available for this purpose: BOINC [12], Techila [13], Condor [14], MATLAB Distributed Computing Server [15], XtremWeb [16], Fura [17], and UnivaUD [18].

The aim of this work is to study and compare two such systems: BOINC and Techila. BOINC is an open source software that has been used for years in large international projects such as SETI@home [19], Climateprediction.net [3], LHC@home [20], and the World Community Grid [21]. Techila is a commercial product developed by Techila Technologies Ltd. which has been in use at the Tampere University of Technology (TUT) since 2006. At TUT, the Techila system has been used by professor Ulla Ruotsalainen's research group from the Department of Signal Processing, docent Matti Lindroos from the Department of Physics, researcher Juho Kanniainen from the Department of Industrial Management, and professor Samuli Siltanen from the Department of Mathematics. TUT, CERN, and Techila Technologies Ltd. have financed this study. The features that will be compared in this study are: usability, maintainability, performance, data security, reliability, and invisibility. The comparison is carried out by solving two scientific computing tasks utilising both systems, and making observations on both systems.

The structure of this work is as follows. Chapter 2 will concentrate on describing distributed scientific computing. Issues such as distributability and different kind of grid systems are discussed. The latter part of this chapter will be dedicated to the introduction of the criteria used in the comparison. Chapters 3 and 4 will introduce BOINC and Techila, respectively. The general characteristics and the use of these systems will be described. Chapter 5 will introduce the computing tasks to be used

in the comparison and explain how the tasks have been distributed. Also in this chapter, the observations will be introduced, and the systems compared. The work closes with a brief chapter of Conclusions.

2. DISTRIBUTED COMPUTING

2.1 General

Scientific computing means solving, with the aid of computers, a mathematical model of some aspect of a scientific or technological problem. In practice, this can mean using a piece of software to solve a specific problem. The most common approach, however, is to use general computing software to solve some specific mathematical model, a linear system of equations for example. In this work, scientific computing is simply referred to as computing.

Traditionally, computing has been what is called serial computing where instructions are executed sequentially one after another. By distributing the task among several computing cores to be computed simultaneously, the computing time can be significantly reduced. For this kind of approach, there are several different types of systems. If the cores are within a single computer, it is called *parallel computing*. In *distributed computing*, the cores are in separate computers that are connected via a network connection. This makes it possible for the computing cores to be spread far apart physically. Basically, the difference between parallel computing and distributed computing is the manner of connection between the cores. Whereas parallel computing uses a common memory area, distributed computing has to resort to network connections that in comparison are very slow. For this reason, parallel computing is not always distributable, but distributable computing can always be done also in parallel. This work concentrates on distributed computing.

The advantage in distributed computing is the larger amount of available resources. This makes it possible to run computing tasks that would not be feasible with the resources of a single computer. Based on kind of resources required, distributed computing can be divided into *computation intensive* and *data intensive* computing. When there is a pool of computers that can be used, and the computation intensive task can be divided into non-interdependent pieces, the computing time can be significantly reduced. For example, the time required to complete the Napoleon algorithm, introduced later in Chapter 5, was reduced from three hours to a few minutes. In data intensive computing, tasks require so much data that it is not possible to run the project on a single computer. It is then quite possible that the data can be split to several computers. However, in this work the focus is on computation intensive computing.

Distributed scientific computing can be run on two different kinds of systems: *clusters* and *grids*. In clusters, the computers have the same operating system and the computers are connected via a high-speed local network [22, p. 17]. In grids, the computers can have different software platforms, devices, and network connections [22, p. 17]. In addition, in clusters the computers are dedicated only for scientific computing. This is not the case with grids, where the computers can be in different kinds of use.

2.2 Distributing scientific computing to desktop computers

Modern personal computers are already so powerful that in normal use, such as word processing, most of their resources are unused most of the time. Public computers, such as computers used in PC classes, are only in use for a certain part of the day, and even when the classes are full, the full computing capacity of each computer is in use for only a small portion of the time. Over the years, a number of systems, called computing grids, have been developed to exploit this idle time. One potential advantage of such systems is that increased computing power can be obtained without large-scale investments, as only existing machine capacity is being taken into use. The computational power obtained from these grids is not entirely free; for one thing, the energy consumption increases, because these computers will consume more electricity when the processor is computing. In this work, the term grid refers to a computing grid, although there are also data grids and application grids.

Computing grids usually have a server-client architecture. Figure 2.1 illustrates a typical grid structure. A computing project is created on the server. The project comprises of several computing jobs, which need to be computed in order to solve the original problem. The server's role is to distribute jobs to clients, which in turn send the results back to the server. In this work, two grid systems, Techila and BOINC, will be studied.

Distributing computing tasks to desktop computers has its own problems that are not present in clusters or supercomputers. The computing environment needed by each computer program has to be made available on each client computer. This can present problems with larger pieces of software, because it is not always clear what is needed for the computing project. For example, in computing a MATLAB code, the code in question has to be first compiled into a program using MATLAB's own compiler. In addition, the program and the MATLAB runtime library have to be sent to the client, and environmental variables need to be defined. Even this may not be enough, because in Windows computers, the MATLAB runtime library is dependent on Microsoft's VC++ library, which is not included by default in Windows installations. In these cases, also the VC++ library needs to be sent to the clients.

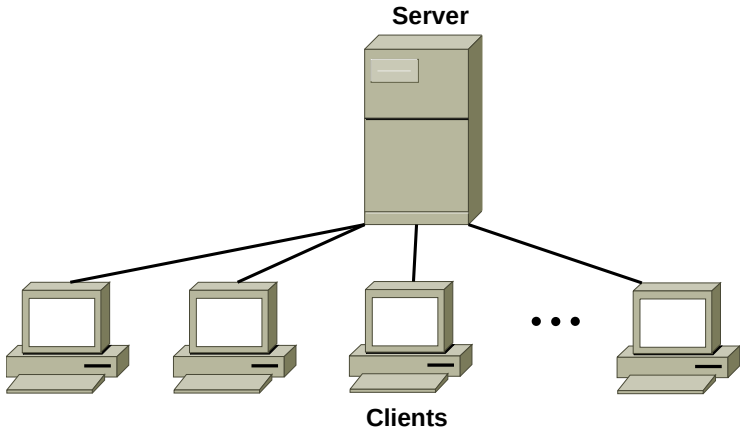

Figure 2.1: The structure of a computing grid

Another problem is that desktop computers may be running different operating systems. In order for the project to use all the computers in the grid, each operating system needs to have a version of the program. For example, Fortran is often used in Linux and Unix environments in scientific computing. Offices and public desktop computers, on the other hand, often use Windows operating systems. This is why the Fortran code should be compilable in Windows, and this can be difficult.

2.3 Distributability of computing

2.3.1 Distributability in general

In this section, the distributability on desktop computers of different types of computing tasks will be studied. This differs from cluster-based computing, for example in that desktop computers are primarily intended for other uses. The computing should not disturb the interactive user of the computer. Also, the data transmission is usually slower than in clusters.

In practice, no computing task can be completely distributed — there is always part of the computing task that needs to be done in serial computing. Let p be the distributable part of the computing, and n the number of computing cores. Then, according to Amdahl's law, the speed of computing can be improved by a maximum factor of s:

$$s = \frac{1}{(1-p) + \frac{p}{n}}. \tag{2.1}$$

The *scalability* of computing, on the other hand, describes how much shorter the computing time becomes when the number of the computing cores n is increased. The better the computing scales, the better it is suited to be computed with mul-

tiple cores, because there is sufficient benefit obtained from increasing the number of cores. For example, one is only allowed to run parallel computing on CSC's supercomputers if the computing task passes a set of scalability tests [23].

Amdahl's law and scalability can only describe how distributable the computing task is. There is no attempt here to define the distributability of the problem itself. Scientific problems can be solved in many different ways — some are distributable and others are not. In other words, a computing task that cannot be distributed might be replaceable with a distributable computing task. Also, the significance of scalability is dependent on the grid used. If the grid is in heavy use, then certain scalability limits may be sensible. In practice, all computing tasks have a limit into how many parts they can be distributed. This means that if there are enough computing cores, scalability can be zero or even a negative value. Scalability is negative when the time needed for data transfer grows more than the time needed for computing a single job is reduced by distributing it.

2.3.2 Well-distributable computing

A computation is well-distributable when it can be easily divided into parts that do not relay information to other parts during the computation. For example, loops where the result of the previous round is not needed on the subsequent round, as in algorithm 2.1, are well-distributable.

Algorithm 2.1: A well-distributable algorithm

for $i = 0, 1, 2, \ldots, n$ **do**
    $x_i = T(y_i)$;
end

The computing task also needs to be distributable into reasonably sized parts. The jobs should not be too small, or the time required for transferring the data will grow to be too significant. The jobs should not be too big either, because computers may be restarted in the middle of the computation. This problem can be avoided by taking snapshots of the job at set intervals. The grid used needs to support the use of snapshots. The BOINC and Techila solutions studied in this work both support snapshots. The computing task too has to be of a type that can be written to a file from time to time, and it should be possible to resume when the file is read.

Computing based on Monte Carlo methods are usually well-distributable [24, p. 358]. In the Monte Carlo method, the simulation is run several times and only the value of a starting value of a random number generator changes. This can be used to compute integrals or study the statistical properties of a phenomenon. The methods are often very computing-intensive, because a lot of simulations are needed to get a

result that is accurate enough. The general rule is that the number of simulations needs to increase by N^2 in order to obtain an N-fold improvement in accuracy; for example, an additional decimal of accuracy requires one hundred times more computation. Monte Carlo methods are used in many different areas, for example in physics and finance.

In scientific computing, the visualisation of results is very important. With visualisation, complex phenomena and vast amounts of data can be turned into a form that is easier to understand. Computing the data for a single image can be very time-consuming. If computing a single pixel takes minutes, computing the whole image can even take months. If it is possible to distribute the pixels into independents jobs, the computing task is easy to distribute. Each client is given a set of pixels that are to be computed. This makes it possible to draw the full picture much faster. If needed, the resolution of the picture can be increased by increasing the number of pixels. The Techila grid has been used at TUT for this type of computing by docent Matti Lindroos studying superconductors, and by Harri Pölönen for quantifying the intensity of fluorescence microscopy.

Optimisation algorithms are basic methods in scientific computing. Sometimes the problem itself is to find the optimal solution, sometimes optimisation algorithms are just a part of the computation. Optimisation is used, for example, in physics, mechanics, finance, and different kinds of technologies. Some optimisation algorithms are distributable. For example in genetic algorithms, computing the objective and constraint functions for members of a population can be distributed to several computers. This approach is needed especially when computing the objective function or constraint conditions requires intensive computation. For example in optimisation problems related to structural mechanics, the computation of stresses using a finite element method can take hours or even months. The size of a population can also be increased when the computing is distributed. However, the effect this will have on the effectiveness or accuracy of the computation depends on the optimisation task.

Optimisation algorithms do not usually guarantee that a global optimum is found, instead they return some local optimum. Gradient algorithms, for example, behave like this. Algorithm 2.2 introduces the operational principle of such algorithms on a general level.

Algorithm 2.2: A gradient-based optimisation algorithm

for $i = 0, 1, 2, \ldots$ **do** `// objective function` f
  $\mathbf{g} = \nabla f(\mathbf{x}^i)$;
  $\mathbf{d} = new\_direction(\mathbf{g})$;
  $\alpha^* = \min_\alpha \ f(\mathbf{x}^i + \alpha\mathbf{d})$;
  $\mathbf{x}^{i+1} = \mathbf{x}^i + \alpha^*\mathbf{d}$;
end

If a gradient-based algorithm is used to solve a problem that has several local minimums, the choice of the starting point decides which the result will be. This means that the same optimisation task will need to be run with different starting points to find a solution that is accurate enough. The computation can be distributed to several computers, and the results will be obtained much faster. A similar approach can be used with other optimisation algorithms as well. For example, heuristic algorithms have several parameters and it is difficult to guess what the suitable values could be when faced with a new problem. A grid can be used to run the computation using different values.

For example, Harri Pölönen, a researcher at TUT, used a grid to estimate 22 000 parameters for analysing PET (Positron Emission Tomography) scans. A gradient-based optimisation algorithm was used in this case. In this particular problem, the computation of the gradient was intensive, and the optimal length of a step was unknown. The solution was to use the 100 computers of the grid to start at the same point, but to randomly vary the length of the step. After fifteen minutes, the best resulting point was identified and the computation continued from that particular point. Using the method described, the required computing time was reduced from two months to two weeks [25]

2.3.3 Computing that is not well-suited for distribution

Not all computing tasks are well-distributable. In some cases it may even be impossible to distribute a computing task. The most important requirement is that the individual jobs must not be interdependent. Otherwise the computing must be done in a specific order using serial computing. For example, the gradient-based optimisation algorithms themselves are hard to distribute if the aim is to run a single optimisation run. This is due to the fact that the result of the previous iteration decides the point from where the next iteration's gradient will be computed. However, if one intends to run the optimisation problem several times with a number of different parameters that are determined beforehand, the computations can run independently and the distribution would be easy.

A gradient-based optimisation algorithm is one example of iterative algorithms. In iterative algorithms, a variable is updated by performing the same operation several times until a certain stopping criterion is satisfied. For example, if the change in the value of the variable is lower than some predetermined limit, it can be considered that the solution has been found and the iteration is stopped. Algorithm 2.3 describes an iterative algorithm on a general level.

These types of algorithms are difficult to distribute, but it is not necessarily impossible. The book [26] discusses iterative algorithms and how they can be distributed. There are two types of distributions for iterative algorithms: synchronised

Algorithm 2.3: Iterative algorithm

for $i = 0, 1, 2, \ldots$ **do**
    $x_{i+1} = T(x_i)$;
end

and non-synchronised. In synchronised distributions, it is necessary, from time to time, to wait for the completion of the computation. This means that the slowest computing job defines how long computing will take. In non-synchronised computing, each client can run for the whole duration of the project. A non-synchronised solution requires that messages be sent between the clients. In [26] non-synchronised solutions are recommended for grids that include different types of computers and network connections. The systems studied in this work are grids of this type, but non-synchronised solutions are not feasible because the systems do not support relaying messages between the clients. For the same reason, some parallel algorithms require relaying messages and so are not distributable with these systems.

The computing run on a desktop computer grid should not significantly affect the other processes of the client. For this reason, the computing should not be very memory-intensive. Also, due to slower network connections, it is not advisable to transfer large amounts of data in a grid.

Thomas Mühlenstädt from Dortmund University of Technology has written an algorithm to compute an experimental design where the computation of a single iteration can be distributed to several computers [27]. The problem with this approach is that each client needs to know all the points that have been found to date in the iteration, and the number of these points increases with every iteration. Therefore, the memory required to store the information becomes the limiting factor. Distributing would still reduce the time needed, so in this case using a cluster could be an option because clusters usually have more memory than desktop computers, and faster network connections.

Because clients are used for other purposes, they may drop out of the grid in the middle of a project. A grid may also suspend computing when the interactive user of the computer is perceived as active. Clients may also be shut down or restarted in the middle of the computation. For these reasons, the computation of a single job cannot be too long, otherwise there is a risk that the project will never finish: clients running the job may never remain in continuous operation long enough to complete it! This problem can be avoided by taking snapshots of the job at set intervals. A snapshot contains the state of the computation at the time the snapshot is taken. This snapshot can then be used to continue the job from the same point. For example, Tarmo Äijö from Tampere University of Technology Department of Signal Processing had to use snapshots when distributing an algorithm used for estimating

the parameters a gene control network, because the computing time for a single job was longer than a day [28]. Another example where snapshots have to be used is the ClimatePrediction.net project, mentioned in the Introduction, where one job takes months to complete.

2.4 Volunteer computing

In volunteer computing, anyone can contribute to a computing project by adding their computer to a grid, provided that a network connection is available. These kinds of grids can grow to be very large. For example, the SETI@home project runs on 2.2 million computers in 200 countries [29].

The volunteers provide their computer time free of charge to an organisation that needs the computing power. The computing projects are not completely free. Marketing is needed to get more volunteers to participate. A popular research topic is a good asset for the project. The World Community Grid, for example, only runs computing projects that are deemed beneficial for humanity as a whole [21]. Also, the volunteers have to be kept interested in the project. BOINC offers its volunteers a project-specific screensaver and points for completed jobs. This allows users to compete with each other.

However, the results from volunteer computing should not be blindly trusted. Computers connected to the grid may be overclocked (which introduces computing errors) or the user may maliciously attempt to skew the results. For this reason, computations need to be repeated on several computers, and results are only accepted when they are replicated. The comparison must also accommodate small differences arising from computations being run on different computers running different hardware and operating systems. Another feature of volunteer computing is that the organisation running the project cannot decide when the computing should be run and how much resources are to be used. These factors are up to the volunteers. The computing projects cannot include confidential information because the data cannot be hidden from the volunteer.

2.5 Intra-organisation distributed computing

Organisations like universities and companies that need scientific computing may build their own grids using existing computers. For a grid to be a viable option, the organisation needs to have computers that have computing cores that are not in continuous active use. The cores also need constant access to energy and network connections. For example, the computers in computer classrooms at universities and office computers meet these criteria.

2.6 Criteria for the comparison

To compare two different grid systems, one has to consider the different roles related to running a grid. These roles are grid administrator, client computer administrator, interactive user of the client computer, computing project owner, and the gridifier. Figure 2.2 shows a grid and the different roles. The roles are not necessarily different people. For example, the project owner may handle the gridification of their own code, or a client computer administrator may also be the grid administrator. The criteria analysed in this study are: usability, maintainability, performance, data security, reliability, and invisibility.

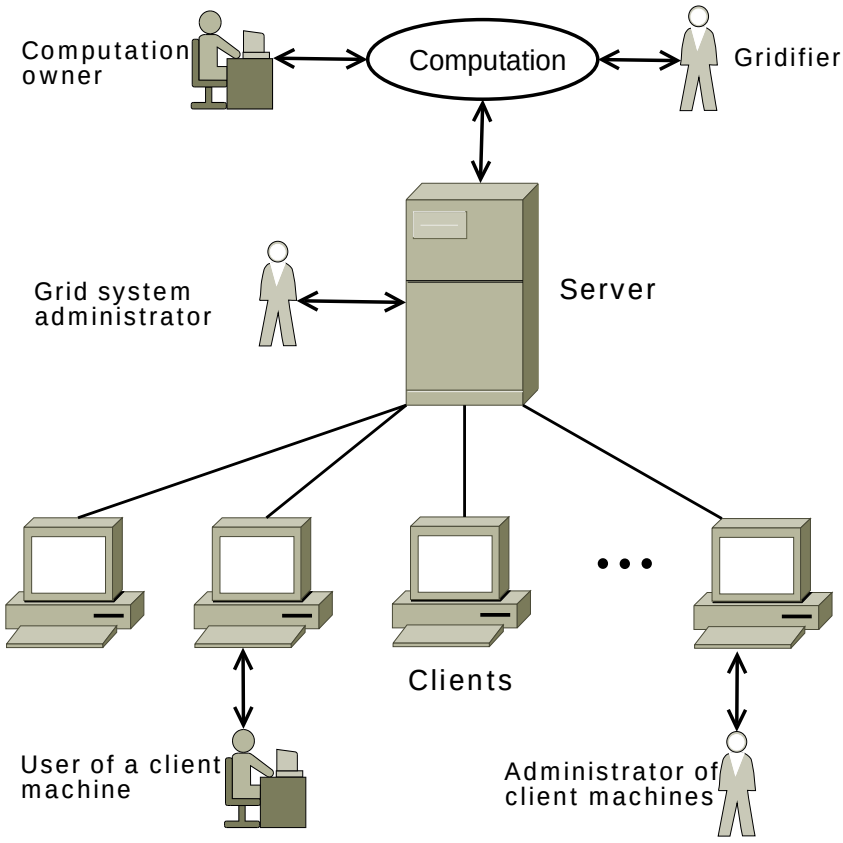

Figure 2.2: The roles in a grid

Usability. Grids are used to quickly solve scientific computing projects. This means that gridification should not take too long. Otherwise the computing task could be solved in the time it takes to gridify a project. Also, installing a new system can take considerable time. The installation of the system should be easy and the system should be easy to learn to use. Gridification should be possible using several programming languages, so that the person responsible for the gridification

does not need to learn to program in a new language. The person who runs the computing project should be able to run computing tasks in the system smoothly and the results should be readily available. In error situations, the system should give sensible feedback to the user.

Maintainability. In modern organisations, IT administration has their hands full of work even without the added layer of a grid. As the use of grids is not yet widespread, few people have a profound understanding of the systems, or experience in using them. This means that the systems should be as easy to maintain as possible. Updating the grid software, adding users, and monitoring resources should be simple.

Performance. The object in grid computing is to get a project finished as fast as possible. Computing jobs should reach client computers as quickly as possible. When the number of jobs is of the same magnitude as the number of the client computers, the way in which jobs are distributed to clients becomes very important. This situation is encountered at the later stages of the project at the latest. Optimising the distribution of jobs is no simple task because grids contain computers of varying efficiency and computers may be needed for some other use in the middle of a computation. Another factor taken into consideration when comparing the performance is the amount of computing power required by the system itself. If the grid requires part of the computing power for running its internal processes, the extra effort will be multiplied by all computers in the grid. The system should be able to efficiently use the resources provided by the pool of client computers.

Data security. Grids have two users whose information needs to be secure: the person running the computing and the user of the client computer. Computing tasks sent to a grid may contain sensitive information — for example, a researcher may not want others to have access to files before the results have been published. On the other hand, the user of a client computer needs to know that a grid user has no access to his or her files on the client computer. The grid must not open the client computer to potential attacks from third parties. In essence, a grid system must be able to prevent the modification or reading of grid files from the users of the client computer or outsiders. Similarly, the client software running on the client must have no access to other files on the client.

Reliability. The most important criterion for a good grid is reliability. The computing project owner must be able to trust the grid to deliver a result on all jobs that are sent to the grid. The system must therefore be able to handle situations where a client is dropped out of the system in the middle of a job. In an error situation the system should provide sensible feedback and return the results of successful computing jobs. It would be very unfortunate if a month's worth of computing was wasted because of a single erroneous job. In addition, the system

should be available at all times.

Invisibility. The computing should not disturb the interactive user of the computer — the system should be completely invisible to the interactive user. This means that the system should be able the pause the computing job if it seems possible that running the computation would otherwise slow the client computer too much. When the client needs to be suspended depends on the computing power of the client. On slower computers, the computing has to be paused whenever the user is logged in, whereas the most powerful computers can always run computing jobs. The system should be able to define, for each computer, when the computing needs to be paused. In addition, the system should not use too much of the client computer's hard drive space or memory.

3. BOINC

3.1 General

3.1.1 Background

BOINC (Berkeley Open Infrastructure for Network Computing) is a software package developed for use in volunteer computing. It provides tools for creating and joining volunteer computing projects. BOINC was developed for SETI@home project, which still uses the software. Nowadays BOINC is used in dozens of public computing projects. For example, ClimatePrediction.net [3], World Community Grid [21], Folding@home [30], Einstein@home [31], and GPUgrid.net [32] use BOINC.

The development of BOINC is based on the experiences gained from running the SETI@home project. The main aim has been to utilise the resources available on the internet. The software has also been developed in a way that lowers the threshold to take up volunteer computing. Other goals have been the ability to attach a client in several projects, support for different kinds of applications, and rewarding volunteers. [33, p.2]

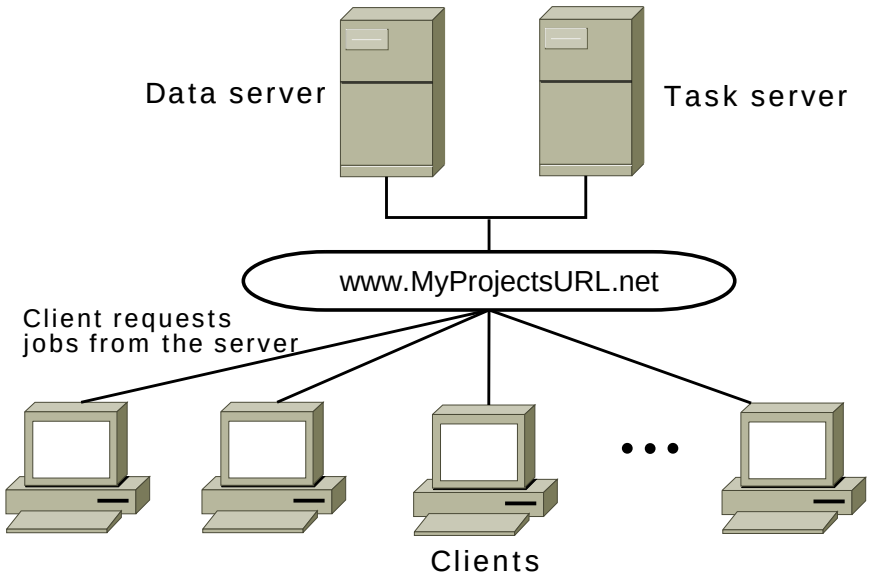

Figure 3.1: A general overview of the BOINC system

3.1.2 Structure

BOINC comprises client and server software. The structure is shown in figure 3.1. The client contacts the server using a remote procedure call (RPC). At the same time, the client reports on completed jobs and asks for more jobs for a set minimum time period. [34, p.3]. Credits are also awarded for the volunteer at this point.

In BOINC the computation is always attached to a BOINC project that can contain one or several applications that are to be run. The client software contacts the desired BOINC project using the project's URL (uniform resource locator). On a Web browser, the URL also points to the home page of the BOINC project, which is automatically generated when a project is created. Through the Web page, a volunteer can check their points, and the project administrator can monitor the project database. Thus, each BOINC project has a URL that is also used as a project identifier.

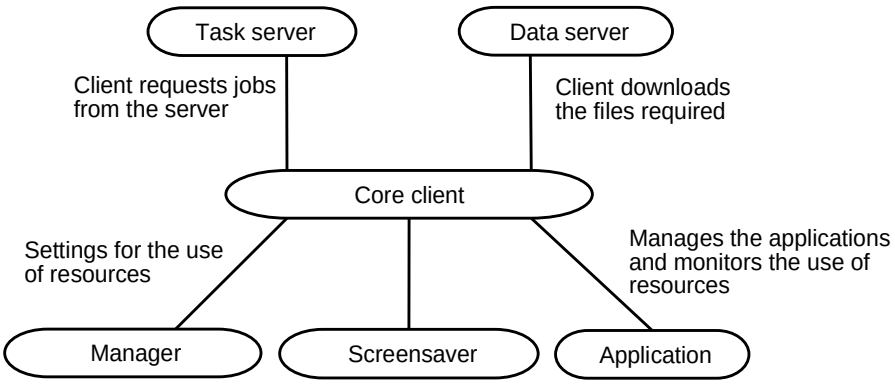

Figure 3.2: Components of the BOINC system

The server is divided into a task server and a data server, which can physically reside on the same server, but they can also be located on different servers if needed. The task server creates all the jobs for a BOINC project and sends them to clients for computation, and finally processes the completed jobs. The data server is used as a storage server for the data used by the computing project. The client downloads the required input files and software from this server. When a computing job is finished, the client transfers the result files on the data server. [34, p.1]

The task server can have several parallel programs running that use a common database. Only the ones that are the most relevant for adding new computations are mentioned here. The *work generator* creates the required jobs on the server. For example, the work generator used in the SETI@home project reads digital tapes and transfers the data contained therein to files, and creates jobs in the BOINC database [34, p.3]. The *validator* checks the correctness by comparing the results

received from different clients for the same job. It also awards credits for correct results to the computers and their owners. The *assimilator* processes the validated results. The assimilator can, for example, move the result files to a specific directory on the server, or process the results itself and store the results in a database. Figure 3.3 shows how the computing proceeds in a BOINC-based system.

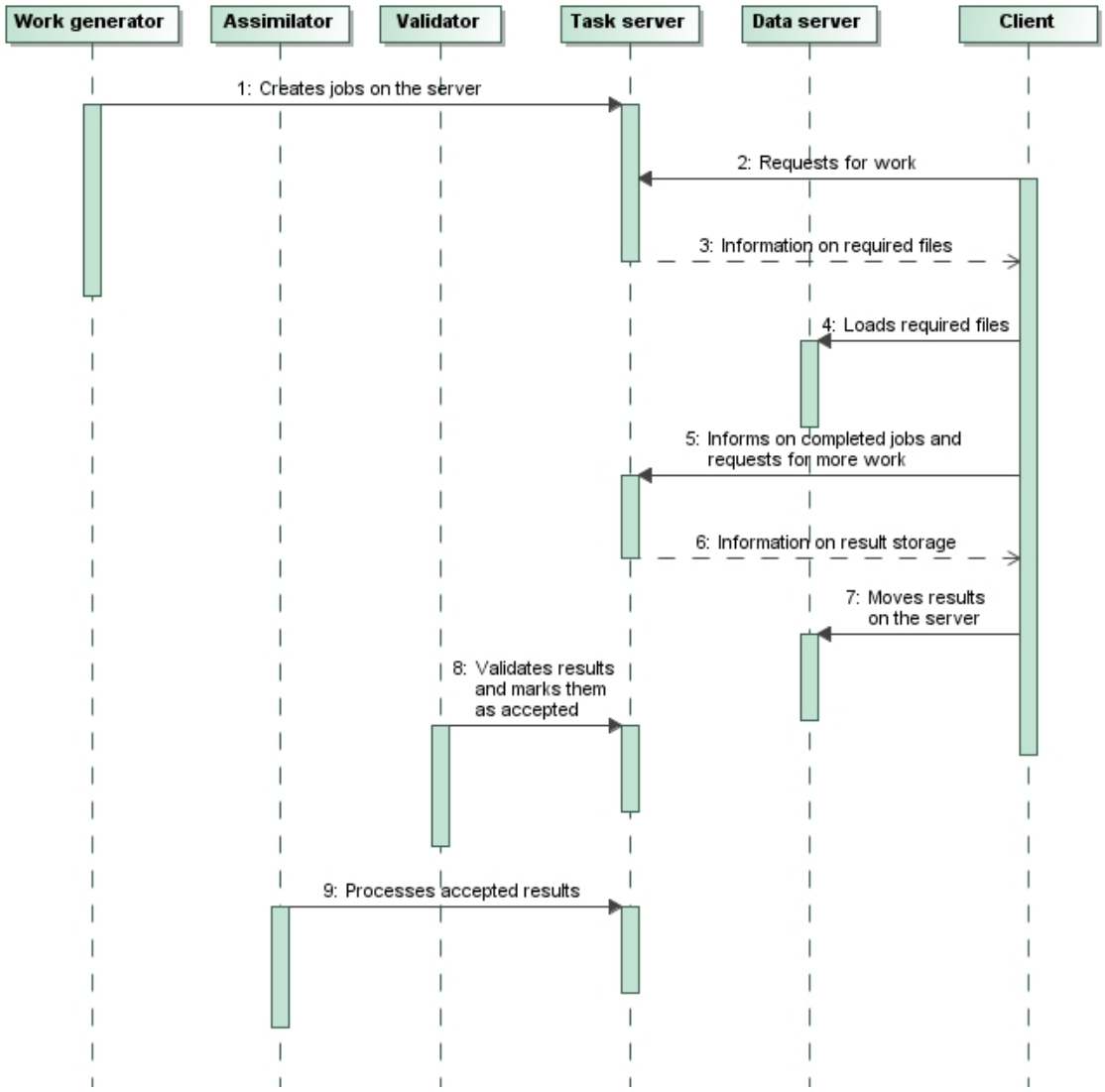

Figure 3.3: The computing process in BOINC

The client software comprises three components: the BOINC core client, the BOINC Manager, and the BOINC screensaver. These three programs, and the actual project software to be run, interact using methods offered by the BOINC runtime library. The core client communicates with the project server, and runs and coordinates the other applications. The Manager has a graphic user interface that the volunteers can use to configure the client program. A volunteer can, for example, decide what amount of resources are to be used by BOINC, and when the

client computer can be used by the computing project. The volunteer can also join new projects through the Manager and see how many points he or she has acquired. Figures 3.4 and 3.5 show views of the Manager graphical interface.

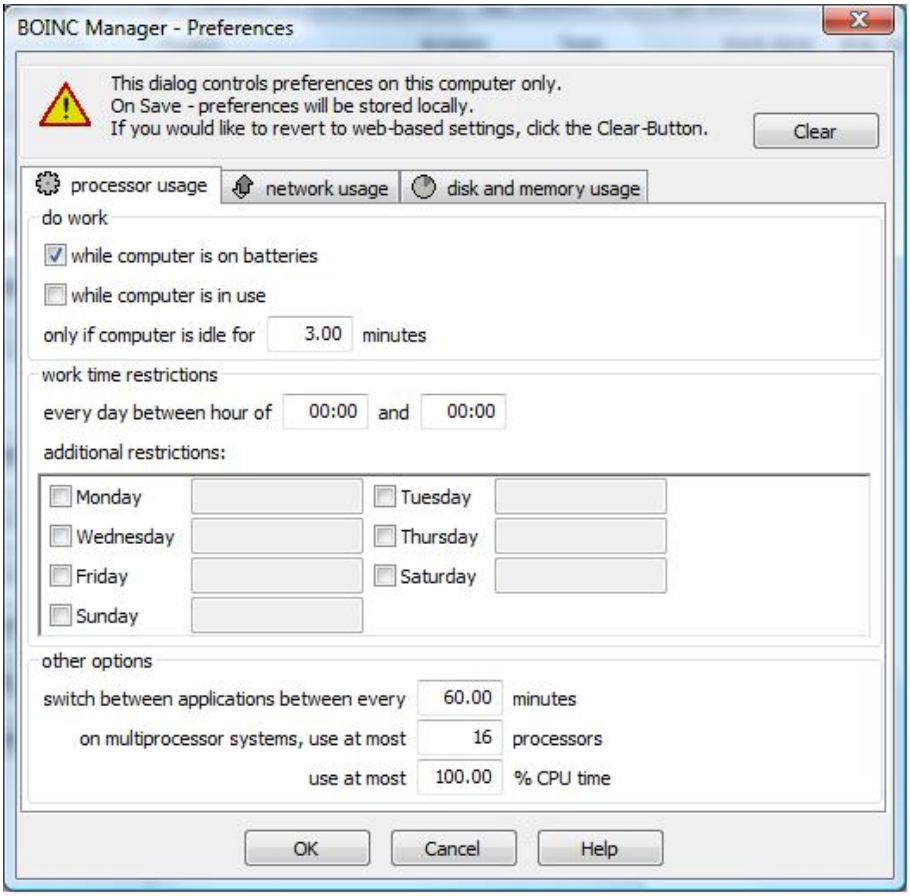

Figure 3.4: The volunteer can use the BOINC Manager to set the amount of resources used for computation

3.1.3 Job timing

Connections between the server and the client are not continuous. Instead, the client connects to the server at intervals. This means that the server cannot really optimise the distribution of the jobs. When the client connects, it informs the server how computation work (in time units) it requires. The server simply carries out this request. If there is not enough work to fulfill the client's request, the server sends as much work as it can. Sometimes a situation arises where some of the jobs require handling large files. In these situations, the BOINC server can be configured to give these jobs to a client that already has the required files. This reduces the load on the data server.

The server in turn informs the client when it can connect to the server the next

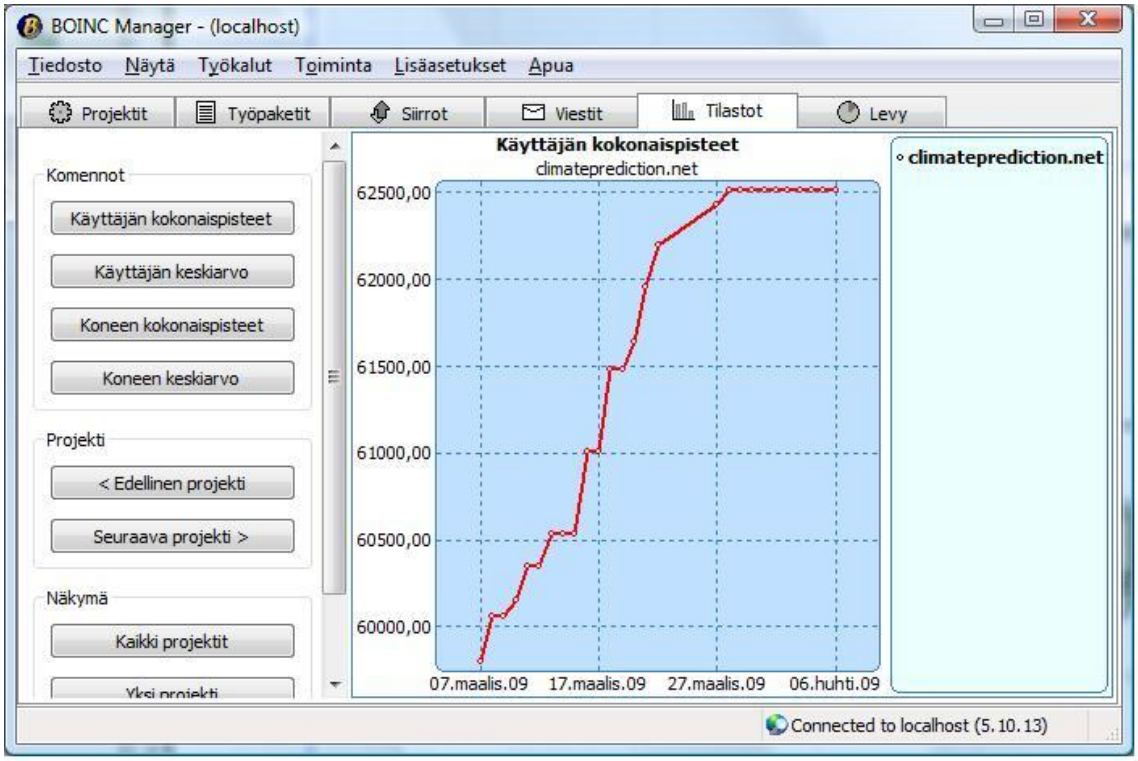

Figure 3.5: The Manager keeps a tally of the points awarded for the user

time. If the connection fails or the server has no jobs available, the client uses an exponential backoff [34, p. 6]. In exponential backoff, the time to the next connection attempt is doubled until a set threshold is reached This approach is needed to avoid congestion in a situation where the server is temporarily offline, or the project cannot offer computing jobs. This is essential because BOINC is designed for projects that have millions of volunteers.

3.2 Using BOINC

3.2.1 Installation

For testing purposes, the BOINC server can be installed on virtually any available computer, but if the the server is to be used in a larger scale project, it is recommended to use a server that has a powerful CPU; for example a dual core Xeon processor with at least 2 GB of memory and 40 GB of hard drive capacity [35]. The easiest way to install the server software is to load a virtual machine from the BOINC website, which comes with a pre-installed server [36]. The virtual machine can be run using a program such as VMware server [37] or VirtualBox [38]. This approach is sufficient for testing purposes, but in large-scale use, the client should be installed for that specific purpose. The BOINC web pages contain instructions

for the installation [35].

A project needs to be created on the server before it is possible to add resources. This is done by running an executable. The executable will create the database, directory structure, and configuration files used for the project settings, background processes, and repetitive tasks. [39]

Installing the client program is easy. This is a requirement, of course, because every volunteer should be able to install the client software without problems. To install the client, the volunteer downloads the software for example from the BOINC website, and runs the installer. There are versions available for Windows, Linux, and Mac OS environments. More detailed requirements can be found on the BOINC website [40]. When the installation is complete, the volunteer can join projects using the Manager. At this point, the software asks for the login ID to the project in question. The ID used can be an existing one, or a new ID can be created.

If BOINC is to be used for computing within an organisation, the default settings need to be changed because they are meant for volunteer computing. The first task is to create one or more user IDs that have the default settings. In this way the computers joining the project using these user IDs use the same resources and same instructions. When this is accomplished, the option to create any further user IDs has to be disabled from the project settings. To make sure that no 3rd party gains access to the project, the server's firewall must block all http protocol connections from computers that are not involved in the computing project. In addition to these changes, the server needs to be set to accept one result per job without validation, as the computers used in the project are trusted. No validation is necessary.

3.2.2 Modifying and running a project

The computing project can be modified and run by using ready-made scripts on the server. The scripts are created when the project is added. These scripts can be used to add the information on a new application to the database, and to start and pause the project. Project administrator rights are required for running the executables.

Adding a computing task starts by adding the name of the application to the database. The application and the required files are then moved under the `apps` folder of the project. The name of the folder where the files are added must include the name of the application, version number, and the operating system where the application is to be run. If the application, or the required files, are changed, a new folder needs to be created with an incremented version number. When the folder has been created, a script is run that adds the contents of the new folder into the project's `download` directory.

Before adding the computing task to the project, a new result template and a workunit template are to be created. The result template defines the files that

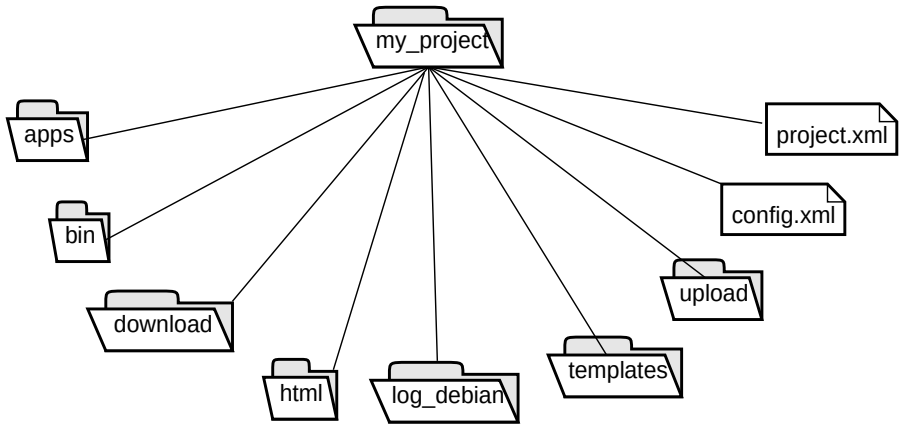

Figure 3.6: The file structure of a BOINC server

are to be transferred to the server when a job has been finished. The workunit template can be used to define the input file for a job, and command line and work attributes. It is possible to define attributes that specify how resources are to be used, and how jobs should be distributed. For example, it is possible to specify the maximum number of floating point operations, and how many instances of the job will be created.

When the application and the required templates have been added, new tasks can added to the server with the `create_work` function. In addition to the templates and attributes, it is possible to give to the function, as a parameter, the command line used for launching the application. This makes it possible to author a software that creates several jobs with different parameters. An example program can be found in Appendix E. It is also possible to give the jobs a separate input file. An important point to note is that the program must be authored with a programming language that can call C language functions. This is because the `create_work` function that creates new jobs is a C function.

The project database contains the information on all the jobs created and all their instances. In addition, it contains the information on the status of a given job: ready, sent, or not sent for computing. The easiest way to monitor the database and follow the progress of individual jobs is through the project web page.

BOINC assumes that the application informs the system about the amount of CPU time used and when a task is finished. This requires that the application has to be modified to use the required calls from the BOINC API library. In particular, the program should be written in a language that can call C language functions. However, it is possible to circumvent this requirement by using an example program called `wrapper` which is included in the BOINC source code. The program in question handles the communication with BOINC and the user can specify which

applications are to be computed. When the files are added to the `apps` directory, the files have to be placed in a folder named so that it mentions `wrapper` as the application that is to be run. To be able use this approach, there has to be an existing version of the `wrapper` program available for the required operating system. For example, for Linux it can be found in the directory of the server, but for Windows it has to be compiled separately. This means that a compiled BOINC API library is needed for Windows as well.

3.2.3 Error situations

If there is an error when running the application, the error can be seen on the project web page. The standard output of the application will not be visible on the page. Sometimes it may be necessary to see the output when troubleshooting an error. In this case the output needs to be written to a file, which needs to be sent to the server as a results file. If the error is due to a problem in the assimilator, the related log files can be found on the server.

If the error is caused by the distribution of the application, or resources related to the application, the error can be viewed on the project pages. This error message, however, is usually very brief. Usually the best option is to use the `debugging` mode on the BOINC client, locally on the client computer, which yields a much larger amount of output.

3.2.4 Combining the results

To combine the results, a separate program called the assimilator needs to be created. When a given job is finished and the validator has accepted the result, the assimilator assigned to the particular application processes the result file. The results are therefore not processed by the context of their creation, but rather by the application used for computation.

The assimilator handles one result at a time and does not know which result will be handled next. A unified result is therefore difficult to obtain at this phase. The assimilator is often used to transfer the result files to a specific folder on the server. Then the results can be processed later when all of the jobs have completed.

3.3 Data security

Any application can be sent to a grid network to be processed by the grid. Therefore it is advisable to use protected application execution when installing the client software. This runs the client software under a user name that has no privileges. It is possible, however, to circumvent these restrictions by exploiting a vulnerability in the operation system; therefore, this approach does not guarantee complete data

security. The setting is also available only in Macintosh version 5.5.4 and Windows version 6.0.0, or older. Otherwise, a user account with no rights has to be created and the client software must be run from this account. During the installation, it is also important to check that the client software folders are not accessible by other users. Otherwise any user may gain access to the files sent to the grid network. The administrator of the client computer does have the permissions needed to do this.

The BOINC manager has the functionality to control the core clients on the volunteer's computer over a remote connection. This is not enabled by default, and it is not advisable to enable this option. If a third party manages to acquire the password used for the remote connection, they will be able to change the core program settings freely. This is of critical importance, as a potential intruder could attach the core program to any project. Because projects can be created by anyone, an intruder could attach the core program to their own project, which could run any applications on the client. This application could be, for example, a virus, spyware, or a keylogger.

Programs sent to the grid network use a digital signature, so that an outsider cannot modify the files unless data security is compromised. However, the connections between the server and the client are not encrypted, so a third party may be able to intercept information by listening to network traffic. Files sent over the network cannot be arbitrarily changed, because the client software checks the file sizes. BOINC also protects the client from unintentional harm. Each new client must have set upper limits for disk space, memory use, and computing time. If these limits are exceeded, the client software stops the computation.

3.4 Maintenance

Updates to the client software need to be installed locally and separately by each client computer's administrator. Also the server needs to be updated separately. In addition to software updates, the administrator has to be involved when adding a new computing task. The administrator has to add the needed applications into the database and update the assimilator and the validator used by the application.

BOINC offers no centralised tools for monitoring the use of resources; resources can only be monitored locally. The administrator cannot remove single computers from the computing project either; this too needs to be done locally. Jarifa [41] is a tool to handle some of these maintenance tasks with BOINC but it is not a part of the original BOINC software and it does not, for example, offer possibility to do client software updates.

4. TECHILA

4.1 General

The Techila grid system is designed for intra-organisation computing. In Techila's solution, several users can run distributed computing projects simultaneously, and each user can add new computing tasks from their computers. The system is a commercial product developed by Techila Technologies Ltd. It is based on a prototype designed at the Tampere University of Technology, and the system is nowadays also used at other Finnish universities.

Techila is a Java-based software. The design drivers have been: extendability, maintainability, scalability, configurability, tolerance for errors, and usability. In addition, the aim has been to make as general and autonomous a system as possible. [42, p. 12]

Techila's grid network system consists of client and server software. Figure 4.1 describes the grid network structure on a general level. The user creates a computing project on the server, which in turn sends the jobs to client computers. There is a two-way connection between the server and the client. Computing software first contacts the server. The connection is acknowledged by identifying the computing software. If the computing software has an approved certificate, it is considered trusted. Then a command channel, which is always active, is opened between the server and the computing software. In this way, the server can better control the client computers. The server can autonomously decide which server receives computing tasks, and it can pause or stop the computation on certain clients. After a computing project has been created, the server can start sending computing jobs almost immediately. In addition to the command channel, a data channel used for fetching resource files and sending result files is created between the client software and the server.

To minimise the time required to complete the project, the server uses heuristic optimisation in issuing the jobs. The server has an estimate of the current performance capability of the clients. This estimate is based on a benchmark and the real-time load on the computing cores. This makes it possible to issue jobs first to the fastest computers. Also, at the later stages of a project, the unfinished jobs are redistributed. This ensures that there is no need to restart these jobs in case there is a failure or an interruption.

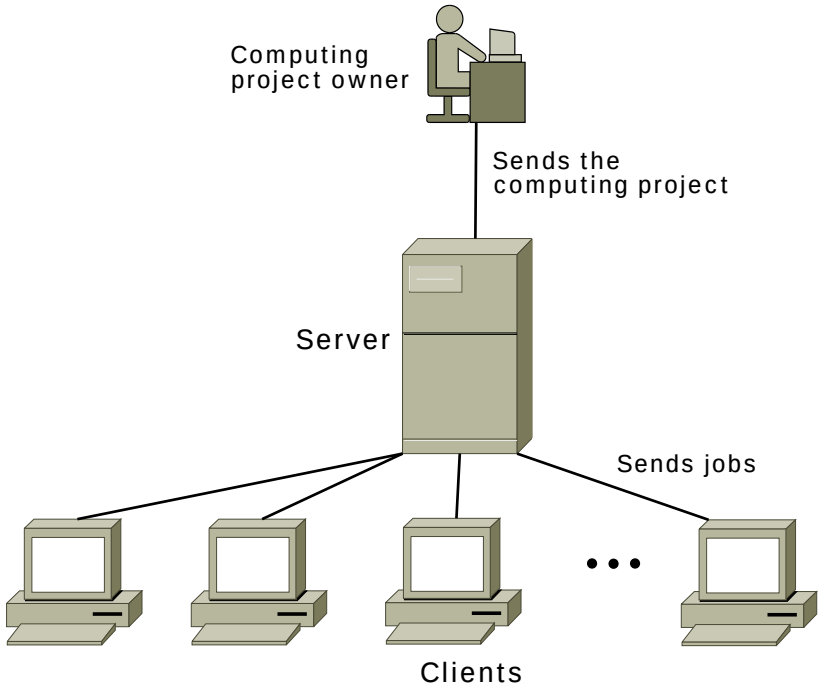

Figure 4.1: A general overview of the Techila system

4.2 Using the Techila system

4.2.1 Installation

Techila Technologies Ltd. usually delivers a pre-installed server — the client does not have to handle installation tasks. The servers are immediately capable to handle extensive use. Techila can also provide a virtual machine, preinstalled with the server software, for testing purposes.

The client software can be installed in environments that support Java such as Windows, Linux, MacOS and Sun Solaris operating systems. Installation files are available for these platforms. Before the installation, it is recommended that the user accounts be created that are used to install the client software. This restricts the rights of the computing software on the local computer. If the installation is done for a single computer, the user account creation can be handled by the installation package. If the installation is done under a domain identifier, the administration must create the user account in question. When the account is ready, the only required step is to run the installation file. During the installation, the installer asks for example for the user name, installation file, and the address and port of the server. The installation package can be modified to include these details. This approach is especially useful when a domain identifier is used. In addition to the actual computing software, Java is installed on the client computer so that the

computing can be run on it. The particular instance of Java is only usable by the Techila software.

4.2.2 Modifying and running a project

The user has to create their own program for creating a computing project. The program has to use the interface calls from the Techila Management Interface library (TMI). The library is currently available in the following programming languages: C, C++, Java, Fortran, APL, MATLAB, R, and Python. In other words, the user can write the program used for the project creation in the aforementioned languages, or other languages that can call functions written in these languages. New versions of the TMI are being created for other programming languages. For example, runs can also be executed using Excel. The TMI can be extended to all languages that can make use of Java calls or external libraries. Figure 4.2 shows how the computation proceeds in a Techila-based system.

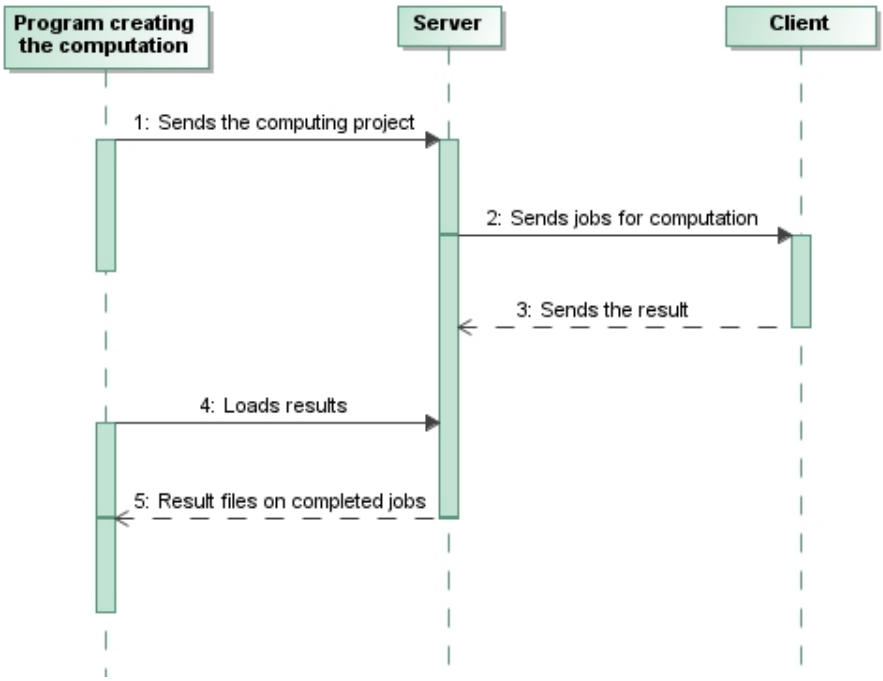

Figure 4.2: The computation process in the Techila system

Algorithm 4.1 shows on a general level how the program used to add computing tasks works. First the grid network software must be set up. The user's configuration file is read and an SSL connection is established with the server. Then the bundles required by the computing project are created. The binary bundle contains information required for running the computation, such as the name of the program, files required, and the command line. In addition to the binary bundle, a data bundle can be created that contains files processed by the computing project. When

the required bundles have been created, the computing project can be started.

Algorithm 4.1: A program for creating a computing project on the Techila grid

```
initialize();
create_binarybundle();
create_databundle() ; // if one is needed
create_project();
wait_for_completion();
download_results();
handle_results();
```

An alternative procedure for creating a computing project is to use a `peach`-function (parallel each), which is available for the following languages: MATLAB, R, APL, and Python. With the `peach` function, it is possible to compute a function several times using different parameter values. It will create the required bundles automatically according to the parameters given to it. `Peach` requires the name of the function, as a parameter, and the parameters of the function. Parameters can be divided into two classes: parameters common to all jobs and job-specific parameters. If required, the `peach` function can be given the names of the files required in the computation. It is also possible to give the `peach` function other parameters related to its operation. For example, the `peach` function can be instructed to compile a function each time it is run. As a default, a compilation is only done if the source code of the function has been modified.

Common parameters are given to the `peach` function as a table where one parameter has to be a `<param>` string. This parameter is replaced with a job-specific parameter when a client begins computing a certain function. Job-specific parameters are defined for the `peach` function in a table where each cell is distributed into a separate job for computation. The result returned by the `peach` function is a table that is the same size as the table that contains the job-specific parameters, and the cells are replaced with the return values of the function that was executed. For example in MATLAB, the tables are cell arrays. Thus it is possible to give the job as parameters, and the results can be of almost any kind of data type, or combination of different types such as scalars or strings.

When a project has been created, the progress can be monitored in a status bar that is created when the computing project is created. The status bar is shown in figure 4.3. A status bar is created only if the user has defined this in the settings. An optional way to monitor the project is to use the server web page where the users can see their compution.

Figure 4.3: A status bar that shows the progress of the project

4.2.3 Error situations

If there is an error when running a job, the output of the faulty job can be reviewed in the status bar. Figure 4.4 shows an example error message. It is also possible to view errors through the server web page. Figure 4.5 shows an example error message on the web page. If needed, the user can set up the system to write errors into a file on the local computer.

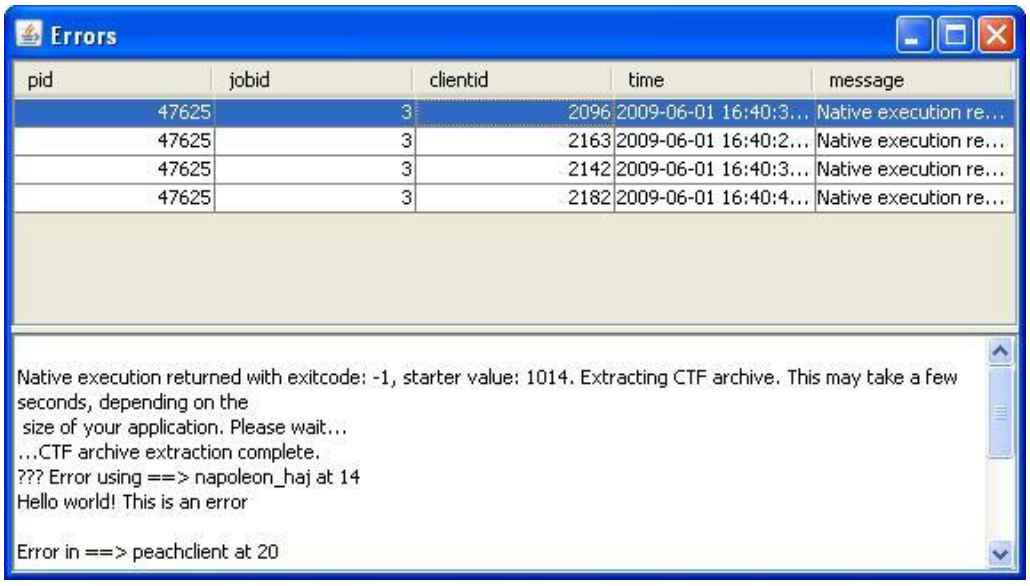

Figure 4.4: An error message where there has been a problem with the application being run in the grid

It is also possible that an error is encountered in the code that handles the distribution. A typical example would be a binary file missing from the data bundle. This kind of error will be noticed only when the code is executed. Figure 4.6 shows an error message related to distribution. In this particular case, the name of the result file has not been defined.

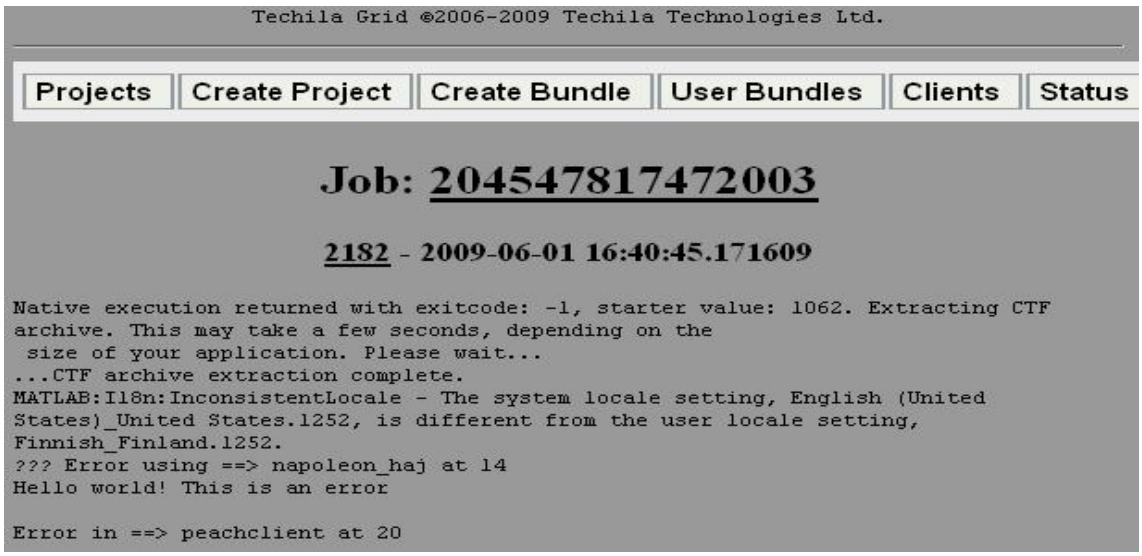

Figure 4.5: An error message on the web page when an error has occured in the application being run in the grid

The Techila system also logs all events in the grid management kit. The user can set up the system to show some or all of these events through the console. If it is not possible to troubleshoot an error from the status bar, or the error occured before computing could start, a log file is often helpful. For example, when a user has entered a wrong user name or password, this is easy to check in the log files.

4.2.4 Combining the results

The results can be combined in the same program that is used to create the computing project. When a project has been created, the first step is to wait for the project to complete. When the project is finished, the result files are loaded and processed. The results can stored on the computer for later use, for example, or the results contained in several files can be read and processed into a single result.

However, it is not necessary leave the program waiting for results. This would be impractical especially when the computing takes a long time, days for example. In this case the program that creates the project ends when it has created the project. Another program needs to be created to load the results. The project ID number is needed so that the program knows to fetch the correct results. It is also possible to load results from the server in real-time. This requires a program that polls the server for new results at set intervals. This is a useful feature for example when creating animations.

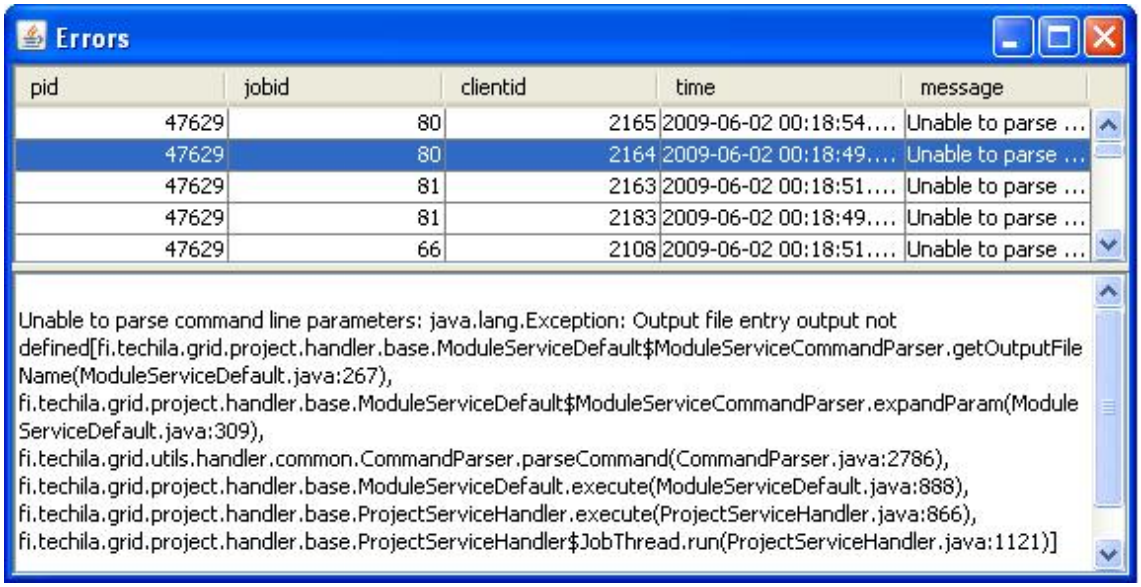

Figure 4.6: An error message where there has been a problem with the distribution

4.3 Data security

Any application can be sent to a grid network to be run by the clients. This can be considered as a security risk because a malevolent person could run malware on client computers. Although the client software has been set up to be run under a dedicated user account, the security may be breached by exploiting a vulnerability in the operating system. If a piece of malware is detected in the system, the sender of the program can be traced because the server database contains the information on all of the programs sent to the grid.

An application run in the grid can also attempt to crash a server by directing all the clients to simultaneously try to connect to the server. However, in the Techila's system this is not possible as network connections from the applications run in the grid have been blocked. Programs sent to the grid network use a digital signature, so that an outsider cannot modify the files unless data security is first compromised. In addition, the connections are always SSL encrypted; data sent over the network cannot be read by listening to the network traffic. Also at the client computer end, the files sent to the grid cannot be read without administrative rights.

4.4 Maintenance

The person maintaining the grid network updates both the server and client software centrally through the web user interface. Updates are therefore delivered directly to the server and all client computers connected to the server. Computers that at the time of the update are not connected to the grid will receive updates at the

beginning of their next connection attempt. Client software handles the update process autonomously. All this is done during runtime, which means that there is no downtime caused by an update. Also the version of Java specific for grid clients can be centrally updated. This ensures that all client computers have the correct Java version.

In addition to updates, maintenance creates new user accounts and accepts certificates for new client computers. When a new client computer is added, it has to be added to the client groups where it belongs within the organisation. For example, at the Tampere University of Technology, the Department of Physics has their own group. The default group for clients is the `All Clients` group. When a new user account is created, the account needs to be given access to the correct client groups. The grid administration also sets, through the web user interface, a suitable computing policy for the new client computers. The administrator can, for example, define when computing is paused, or set the time period when computing can be run on the client.

5. THE TESTS

5.1 Pricing of options

5.1.1 Introduction of the problem

Financial computing is one branch of scientific computing where distribution can be utilised. Distribution can be used because financial computing projects can be solved with the Monte Carlo method. A typical example would be computing projects related to different kinds of financial derivatives, such as stock and currency derivatives [43]. As mentioned in Chapter 2, these methods are well-distributable. For this reason, the pricing of Napoleon cliquet options, a problem that can be solved with the Monte Carlo method, has been chosen as one of the test problem.

A Napoleon cliquet is a collection of options where the annual profit is based on the sum of a fixed coupon $C > 0$ plus the worst monthly performance of an index. Let us assume that Napoleon consists of a portfolio that includes n stocks. Second, assume also that $S_i(s)$ is the price of stock i at the point of time s and $\mu^*_{i,j}$ is the most negative profit in year j.

$$\mu^*_{i,j} := \inf_{t_{j-1}<t_k<t_j} \frac{S_i(t_k) - S_i(t_{k-1})}{S_i(t_{k-1})} \tag{5.1}$$

Then the fixed annual coupon c_j can be calculated:

$$c_j = \left(C + \frac{1}{n} \sum_{i=1}^{n} \mu^*_{i,j} \right)^+ \tag{5.2}$$

The total value of the Napoleon cliquet is then:

$$\mathbb{E}\left[F \sum_{j=1}^{T} c_j e^{-rj} \right] = F \sum_{j=1}^{T} \mathbb{E}\left[c_j\right] e^{-rj}, \tag{5.3}$$

where T is the duration of the contract in years, and F is the nominal value.

The price of the underlying stocks will, as a martingale, satisfy the stochastic

differential equation

$$d\ln S_i(t) = \left(r - \frac{1}{2}v_i(t)\right)dt + \sqrt{v_i(t)}dW_i(t), \quad S_i(0) = S_{i0} > 0, \tag{5.4}$$

where r is the fixed interest rate, v_i the stochastic variance in the profit for the stock i, and W_i the standard Brownian motion.

The variance of the profit is assumed to be a mean-reverting process

$$dv_i(t) = \kappa_i\left(\bar{v}_i - v_i(t)\right)dt + \psi_i\sqrt{v_i(t)}dW_i^v(t), \quad v_i(0) = v_{i0} > 0, \tag{5.5}$$

where $\bar{v}$ is the reference variance, κ_i is the speed in the change of direction, ψ_i standard deviation of the volatility, and dW_i^v the standard Brownian motion with correlations ρ_{mn}, $dW_m dW_n = \rho_{mn}dt$ and $i, m, n = 1 \ldots n$.

The Monte Carlo solution for the problem is attained by discretising the differential equations. Equation (5.4) is discretised with the Euler method

$$\ln S_{i+1} = \ln S_i + \left(r - \frac{1}{2}v_i\right)\Delta t + \varepsilon_i\sqrt{v_i\Delta t}, \tag{5.6}$$

where $\{\varepsilon_i\}$ is a sequence of standard normal distributed random variables. The equation for the volatility (5.5) is discretised with the Milstein method

$$v_{i+1} = \left[v_i + \kappa(\bar{v} - v_i)\Delta t + \xi_i\psi\sqrt{v_i\Delta t} + \frac{1}{4}\psi^2\Delta t\left({\xi_i}^2 - 1\right)\right]^+. \tag{5.7}$$

Here $\xi_i = \rho\varepsilon_i + \sqrt{1-\rho^2}\varepsilon_i^*$, where $\{\varepsilon_i^*\}$ is a sequence of standard normal distributed random variables that are uncorrelated with variables $\{\varepsilon_i\}$.

The value of Napoleon is obtained by running M simulations on the price trajectories of the stocks. A single simulation computes the price and the volatility of the stock at 24 hour intervals. After a month, the profit for each stock is calculated. This is compared to the lowest annual profit to date. If the resulting value is lower, it is stored. In this way, the coupon c_j for year j is computed using the equation (5.2). When all years in T have been computed, a value for Napoleon is attained using equation (5.3). The price of the Napoleon cliquet is found by taking the discounted mean of M simulations:

$$\frac{1}{M}\sum_{k=1}^{M} F \sum_{j=1}^{T} \mathbb{E}\left[c_j\right] e^{-rj}. \tag{5.8}$$

5.1.2 The gridification of the problem

The computation is easy to break into smaller parts because the simulations are independent. Each client can do a fraction of the original M simulations. Let us

assume that we want to divide the assignment into k parts. Then an option is used where $M \mod k$ parts result in $\lfloor \frac{M}{k} \rfloor + 1$ simulations and remaining jobs in $\lfloor \frac{M}{k} \rfloor$ simulations. Thus, M simulations will be created as was the original intention:

$$M = k \left\lfloor \frac{M}{k} \right\rfloor + M \mod k. \tag{5.9}$$

In the computing project, the following fixed values have been used for parameters: $S_{i0} = 1$, $v_{i0} = 0.2$, $r = 0.05$, $\kappa_i = 0.1$, $\psi_i = 0.5$, $T = 10$, $N = 365$, and $M = 2 \cdot 10^5$, $i = 1 \ldots n$. Maximum value of the coupon is $C = 0.1$ and the nominal value $F = 1$. It is also assumed that the correlation between stocks i and j is 0.5 for all $i, j = 1 \ldots n$, $i \neq j$ and that the correlation between the profit from the stocks and the volatility is -0.5. On a single desktop PC (AMD Athlon(tm) 64 X2 Dual Core Processor 3800+), this computing task would take 3 hours.

The code used to compute the value of the Napoleon is shown in Appendix A. This code was compiled into a standalone program using MATLAB's own compiler. The results also need to be stored into a file so they can be easily sent over the network.

5.1.3 BOINC

The easiest way to run MATLAB computing on a BOINC system is to use the `wrapper` program. The required MATLAB runtime library was installed directly on the clients. An alternative to this would have been to send the files to the client with the executable program. This would have required an executable with an environment variable pointing to the location of the library. Also, the Microsoft VC++ would have had to have been sent to the client.

Job and result templates were created for the Napolen calculations. The example assimilator on the server was modified to store the results in the desired file. A validator was set up to accept the job if the time used to compute a result was greater than zero. A program had to be written for the server to create the required jobs. This was easily accomplished by modifying the example program found in the BOINC source code. The only difference in the jobs was the command line used to run the Napoleon calculation. The source code for the program can be found in Appendix E.

5.1.4 Techila

Napoleon was implemented using MATLAB, which meant that the easiest way to distribute the project was to use the `peach` function. The function in question automatically handles the creation of the required bundles and projects. In the

program calling the function, only a table needs to be created that includes the job-specific parameters.

In this case, each job was given the number of simulations and the integer used to initialise the random number generator. The same integer cannot be given to two different jobs because the result would be the same exact random number sequence, i.e. the result would be the same. Appendix B contains the MATLAB files that can be used to run the Napoleon calculation in the Techila grid using the `peach` function.

5.2 Garfield

5.2.1 Introduction of the problem

Garfield is a program used to accurately simulate a particle detector called a drift chamber [44]. A drift chamber is used to ascertain the path of a charged particle. There is no single type of a drift chamber; the geometry and the enclosed gas varies. For example, the muon detector in the ATLAS detector has drift tubes [45, p. 19], the cross-section of which is shown in figure 5.1.

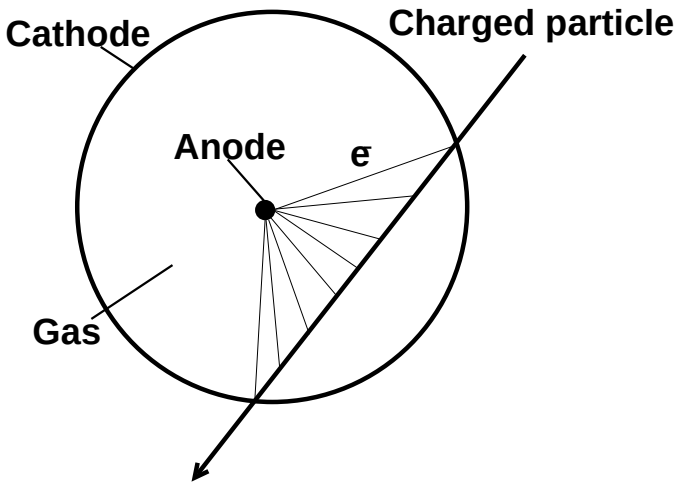

Figure 5.1: A cross-section of a drift tube

When a charged particle travels through the tube it ionises the gas inside the tube. The electrons created in this fashion drift into the positively charged anode inside the tube. When the time from ionisation to the impact is measured, it is possible to calculate the path of the particle that traveled through the tube. This requires, however, that the properties of the electrons, such as the drift velocity, are known for the gas used in the tube. In addition to the gas, these properties are affected by the electric and magnetic fields, and the angle between them. The

calculation of these properties is very intensive, especially when magnetic fields are involved. This calculation only needs to be done once because Garfield stores the results on a file for later use in computation.

5.2.2 The gridification of the problem

In the distributed project, the electron properties of the gas (Ar 93% / CO_2 7%) used in the muon detector of the LHC's ATLAS detector was computed using different magnetic field values, and different values for the angle between the magnetic and the electronic field. Altogether, there were 171 different pairs for the values. On a single desktop PC (AMD Athlon(tm) 64 X2 Dual Core Processor 3800+), this computing task would take approximately 1 hour 40 minutes. Computing all the jobs on a single computer would therefore have taken 12 days.

Garfield can be run in batches so that it executes the commands from an input file. This makes it easy to run the program in a distributed system with only the program and the input file. In order to compute the 171 variant jobs, an input file was created such that the magnetic field and the angle parameter values can be determined based on an integer between $[1, 171]$. This number was given as a parameter for the jobs. The input file is found in Appendix C.

The biggest problem in the Garfield gridification was that it was originally written for a Linux environment and the grid network used only had Windows computers. Garfield is also a big application that has been written in C and Fortran, and it is dependent on the Cernlib collection of libraries. Garfield and Cernlib were compiled in Windows using the Cygwin environment [46]. However, this was an exceedingly laborious operation. It took two weeks to get a Windows version of Garfield. After the compiling, it only took a few hours before the computing project was running on the Techila system.

BOINC. An older version of Garfield had already been run with BOINC in the LHC@home project. A lot of effort was put into getting Garfield modified so that it would communicate with the BOINC client software. This task was now avoided using the `wrapper` program. Garfield and the other needed files were added on the server and the input file was added into the `download` folder of the project. After this, the required templates and the work generator, for defining the command line for each job, were created. The work generator was similar to the one used in the Napoleon project.

Techila. In the gridification phase, the program for adding the computation was created in MATLAB. Any other languages with an existing TMI could have been used just as well. The uploaded result files were stored in MATLAB, under a new name, on the local computer. If the files would not have been stored, they would have remained in a temporary file where they would have been extracted after the

project had been finished.

5.3 Comparison

Usability. Both systems can be used to send any kind of program for execution, so the original program could have been written, in principle, in any programming language. If the programming language has a version of the TMI, the gridification can be done in the original code. The part that is to be gridified is replaced with TMI calls and possibly a `peach` function. After this, the results received from the grid are combined so that the original computation can be continued if needed. In BOINC, this approach to gridification is not easily doable because the work generator and assimilator functionality would need to be added to the original code. This means that the original code should be able to call C language functions. Also in this case, the original program should be such that it can be executed on the server. In Techila grid, the program adding the computation can be run from a local computer, as long as there is a network connection to the server. In the BOINC gridification, the command line files have to be run on the server to add information on the application on the server. Furthermore, for each new application, an assimilator needs to be written, job and result templates created, and information on the new application needs to be added to the configuration file. Gridification is more difficult in BOINC than in Techila because in addition to having to learn to modify the original computation and to use new functions, one also needs to learn the operations for adding a new task on the server.

Maintainability. The maintenance of the BOINC system is difficult because clients need to be updated locally. BOINC does not provide tools for centralised monitoring of clients and resources. If some of the clients need to be diverted into spesific computations, the only option is to create a separate BOINC project. In Techila's system this can be accomplished with little effort. Centralised updates can be pushed through the network. This ensures that all clients have the latest updates for the computing software and Java. By creating groups of clients it is also possible to give specific users access to only some of the computers. For example, at the Tampere University of technology, the computers of the Department of Physics are only available for the department employees. The same set-up is in use at other departments that use the grid. Furthermore, when a new computation is added in BOINC, the administrator must be present. In Techila's system this is not necessary as users can create their own computing projects on their local computers. Installation of the client software is very straightforward on both systems. However in BOINC, care must be taken to make sure that the settings are correct.

Performance. Tests were run on 15 computers with AMD Athlon(tm) 64 X2 Dual Core Processor 3800+ and approximately 1 GB of memory. All in all, 30

computing cores were in use. Tests were run several times, and only when the computers did not have interactive users using them. The progress of the computing runs was monitored. This made it possible to compare the performance of the systems. Figure 5.2 shows the median for the ten Napoleon computations on both systems. Figure 5.3 shows the median for the five Garfield runs. The figures show that Techila is faster in both cases. The reason for this is that Techila was able to get the computing started on all computers almost instantly. In BOINC, the computing started when the client computers had contacted the server. Because of the exponential backoff, this could take several hours on some of the clients. This is why, in Napoleon runs, BOINC was never able to utilise all clients, which explains the clear difference in time when compared with Techila.

The management used by Techila is clearly shown in the figures. When a client has finished computing, it will immediately proceed to the next job. The jags in the figures are due to the uniform size of the jobs in the tests. Furthermore, towards the end of the computing project BOINC waited for the results from the remaining jobs, which resulted in a slowdown. In the Garfield case, both systems worked equally fast in the middle of the computing project. This is explained by both systems having all computing cores in use. When this happens the progress of the computation is only dependant on the computing power of the clients and the amount of resources needed to run the client software. That is to say that optimising the job distribution only affects the progress in the later parts of the computation. As both systems progress at an equal speed, it is also possible to conclude that no significant amount of resources is expended in running the client software in either system.

A comparison closer to the reality, where students would be using the client computers in the computer labs, was not conducted. Otherwise the runs should have to have been repeated several times and at different hours for the results to be comparable. This approach would have been needed because the computers of the computer class used in the tests are used for very different kinds of uses, depending on the hour. Sometimes all computers are in use. Sometimes almost all of them are idle. Conducting repeat runs would not have brought new information into the comparison. Techila was able to manage the clients much more efficiently, so it is clear that it would have handled this situation better as well.

Data security. Both systems allow any kind of program to be run on the system, which can be considered as a data security risk if users cannot be trusted. In BOINC, this risk is lower as the grid administration needs to be present when a new computation is added. Furthermore, in both systems, the administrator of a client computer has access to the sent files. If the BOINC installation is not done correctly, any user at the client end can have access to the files.

The difference between the systems is how they handle connections between the

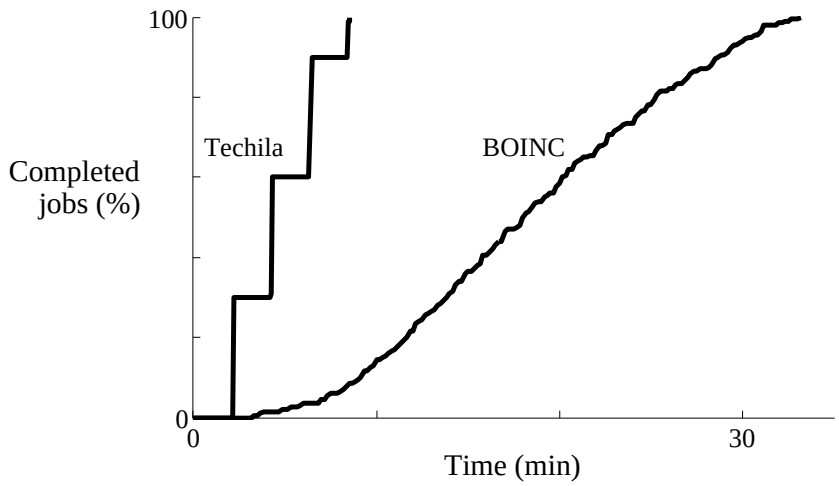

Figure 5.2: The progress of the Napoleon project in the systems. The figure shows a 10 run median

server and the clients. In Techila's system the connections are encrypted, whereas BOINC uses unencrypted connections. Unencrypted connections may allow the capture of information by someone listening to the network traffic. If the remote control in BOINC is taken into use, the risk can be high. If a third party gains control of the client software, they can attach it to their own project, which makes it possible to run any kind of software on the client.

Reliability. Both systems can handle situations where clients are dropped from the system in the middle of computing. In this case the job is forwarded to other clients. In BOINC this can take longer as the job is redistributed only when a reply has not been received in a given time frame. This means that in BOINC a situation may arise where there is only one unfinished job, but it is not given to free clients as the time limit has not yet been reached. Techila sends computing jobs immediately to other clients when they have finished their own computation.

It would be problematic if some of the finished jobs contained errors and others were correct. This situation can arise, for example, when a job is given an integer parameter that is used to index a table. An error can be made in the job creation, which causes some of the jobs to over-index the table. If this happens, the results of the successfully completed jobs need to be recovered, especially if the job duration is very long. In BOINC this is not a problem, because the server-side assimilator handles the results one-by-one when they are received. In Techila's system, the function that loads the results must be given a parameter that only allows the loading of results that have completed. Otherwise loading the results will fail.

Invisibility. Both systems can be set up in a way that the interactive user of the computer does not notice the computing software on the computer. In BOINC

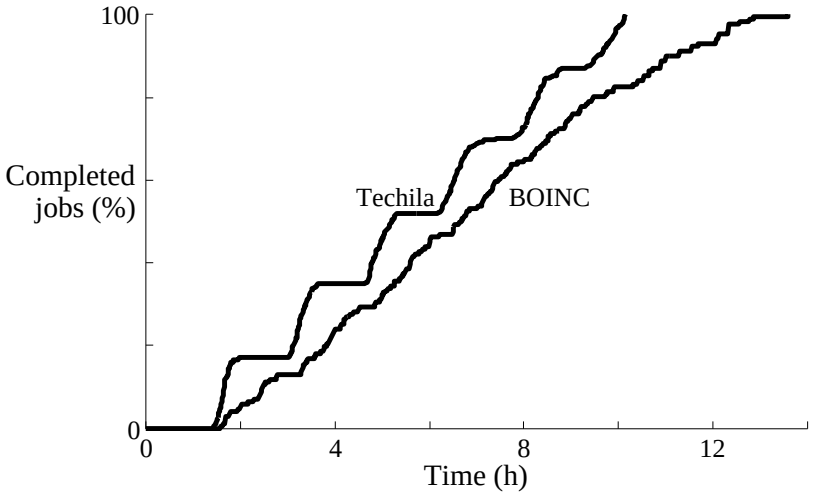

Figure 5.3: The progress of the Garfield project in the systems. The figure shows a 5 run median

this requires that the correct options are set during the installation. Otherwise a shortcut to the BOINC management software may be placed on the user's desktop and the user may have the rights to manage the client software. The client will also use the BOINC screensaver if this has been defined during the installation. The Techila computing software always runs on the background regardless of the installation.

Both systems allow the settings, in what situations the computation is to be paused, to be decided individually to each client. Both systems can therefore be set up in a way that they do not interfere with the interactive user. The Techila system has been installed in public computers used by the students at the Tampere University of Technology since 2006, yet very few students are aware of the system. BOINC is run on personal computers by millions of satisfied volunteers around the world.

During the test runs, it was noticed that the BOINC software does not remove all the application or other required files from the client computer. That is, the files which are added into the apps folder in BOINC server, are not removed. If an input file was defined in workunit template, that however is removed. This results in a situation where, in systems where several users run their own programs, BOINC may utilise the hard drive too much. The client software can be told how much it can use hard drive space, but when this quota has been used up, the client will not load new jobs. Therefore the BOINC project has to be reset, so that all the project files are removed. This must be done locally on the client.

Table 5.1: Comparison of the BOINC and Techila systems

	Techila	**BOINC**
Design	For intra-organisation computing.	For volunteer computing.
	Suitable for short and long computing projects.	Suitable for very long computing projects.
Distribution of jobs	Uses heuristic optimisation.	Does not use optimisation.
Usability	Computing tasks can be added from the user's computer.	Computing tasks are created on the server.
	TMI available for programming languages that support external libraries or Java calls	BOINC API available for programming languages that can call C functions
	Several users can add computing tasks simultaneously.	The BOINC grid administrator has to be present when a new application is added.
	Error messages are relayed to the person running the computing project.	Error messages sent to the server or the client.
	The user can load the results on their own computer.	Assimilator processes the results per each application.
Maintenance	Updates are run centrally from the web GUI.	Updates have to be done locally.
Data security	Encrypted connections.	Unencrypted connections.

6. CONCLUSIONS

Modern personal computers are already so powerful that in normal use their processor is mostly idle. This idle capacity can be exploited by combining computers into a grid where they are used to solve a computational problem. The aim of this work was to study two such grid systems, BOINC and Techila, and compare them. The comparison of the systems was carried out by running two computing tasks and making observations. It was noted that even the basis for designing the systems differed. BOINC has been originally designed for vast public projects where millions of volunteers can join in solving a computing project. This means that the system needs to accommodate congestion at the server side. Techila, on the other hand, has been designed for projects that utilise existing resources of the organisation that carries out the computing. In a situation such as this, the grid users should be able to easily add computing tasks, and the results of a computing project should be available as fast and easily as possible.

The biggest difference for the typical user, like a researcher at a university, is that the grid user can add computing tasks from their own computer in the Techila system, and the results of the project can be easily downloaded from the server. In BOINC this is not possible by default because only the grid administrator can add new applications. The results of a BOINC project are also handled by application. If an application has several users, this will lead to mixed up results. Furthermore, BOINC does not provide any built-in functionality for loading results from the server. From the grid administrator's point of view, the main difference is that the Techila system can be updated centrally through the web interface. In BOINC, the updates have to be run locally.

Techila uses heuristic optimisation in the job distribution. The Techila server is also able to initiate the computing project almost immediately at the client end. These features enable the Techila system to solve in minutes a computing project that could take hours. This is important especially when the results are needed within the same day. In BOINC this would take much more time, because it can take hours for the clients to contact the server.

It is our view that BOINC and Techila are not mutually exclusive — instead, they can be used in different situations. Both are designed for a specific type of computing, and it is our view that they should be used in the type of computing

that they were designed for. BOINC is excellent for very large public projects that can have up to millions of computers around the world. The project should also be big enough to keep volunteer clients busy for months or even years. Techila is not suited for these types of projects, but it is clearly better in environments where several users need to easily add their own computing tasks to the grid. Such an environment can be found at universities for example, where many researchers need significant computing resources. The Techila system also makes it possible to run computing projects where the code is still being developed, as the user running the computation can see the error messages on their computer. Troubleshooting can sometimes be very difficult in BOINC.

It is not possible to say that one system is better than the other — it all depends on the intended use. Comparing BOINC and Techila is like comparing volunteer computing and intra-organisation computing. It would be interesting to combine the two systems. This would be possible by creating a project to Techila system with the lowest priority that would run the BOINC client core. An organisation's could then run BOINC in some large public project, using the computing resources of the organisation's own computers when there is no computing running in the Techila system.

This work focused on comparing only BOINC and Techila. As these systems are intended for different kinds of projects, it would be interesting to compare Techila to another intra-organisation grid system, such as Condor. Such a comparison would make it possible to study the significance of the heuristic optimisation, as both systems would be better able to manage client computers. Furthermore, this work did not discuss the use of snapshots in detail. The reason for this omission is that when the test problems were chosen, the Techila Grid did not yet support snapshots.

Both systems could be improved with the use of virtual machines. This would enhance both usability and security. If the computations were done on a virtual machine, the person doing the gridification would not have to compile the computation separately for each operating system. For example, a Linux user could run computing tasks on a Windows computer using a Linux virtual machine. The virtual machine could also come with a built-in computing environment. The projects would no longer run directly on the computer, but on a virtual machine, so the client would be better protected from malicious software. Virtual machines have not yet been used in grid networks due to their large size and maintainability lackings.

Finally, the characterisation of computing tasks suitable for a grid network has not been studied much. It would be interesting to know, for example, when it is feasible to run iterative algorithms on a grid. Differences between synchronised and non-synchronised distributions for iterative algorithms have been studied in [26], but no comparison has been made between sequential and distributed computing.

REFERENCES

[1] Juha Haataja, toim. Alkuräjähdyksestä kännykkään - näkökulmia laskennalliseen tieteeseen. CSC - Tieteellinen laskenta Oy, 2002. Www-osoite http://www.csc.fi/oppaat/lask.tiede/.

[2] Juha Haataja, Jari Järvinen, Jari Koponen ja Peter Råback, toim. Laskennallinentuotekehitys: suunnittelun uusi ulottuvuus. CSC - Tieteellinen laskenta Oy, 2002. Www-osoite http://www.csc.fi/oppaat/tuotekehitys/.

[3] Climateprediction.net. http://climateprediction.net/ (28.5.2009)

[4] CSC. Cray XT4/XT5 -supertietokone. http://www.csc.fi/tutkimus/Laskentapalvelut/laskenta/palvelimet/louhi (28.5.2009)

[5] GPU computing. http://www.nvidia.com/object/GPU_Computing.html (7.7.2009)

[6] Akaatti. http://alpha.cc.tut.fi/akaatti/ (28.5.2009)

[7] DEISA. http://www.deisa.eu/ (28.5.2009)

[8] NorduGrid. http://www.nordugrid.org/ (28.5.2009)

[9] EGEE. http://public.eu-egee.org/ (28.5.2009)

[10] Worldwide LHC Computing Grid. http://lcg.web.cern.ch/LCG/ (28.5.2009)

[11] HECToR. http://www.guardian.co.uk/technology/2008/jan/02/computing.climatechange (2.2.2009)

[12] BOINC. http://boinc.berkeley.edu/ (28.5.2009)

[13] Techila. http://www.techila.fi/ (28.5.2009)

[14] Condor. http://www.cs.wisc.edu/condor/ (28.5.2009)

[15] MATLAB Distributed Computing Server. http://www.mathworks.com/products/distriben/ (28.5.2009)

[16] XtremWeb. http://www.xtremweb.net/ (28.5.2009)

[17] Fura. http://fura.sourceforge.net/ (28.5.2009)

[18] Univa UD. http://www.univaud.com/ (28.5.2009)

[19] SETI@home. http://setiathome.ssl.berkeley.edu/ (16.4.2009)

[20] LHC@home. http://lhcathome.cern.ch/ (28.5.2009)

[21] World Community Grid. http://www.worldcommunitygrid.org/ (28.5.2009)

[22] Tanenbaum A., Van Steen M. Distributed Systems: Principles and Paradigms, Second Edition - Pearson International Edition. Pearson, 2007.

[23] CSC. Skaalautuvuustestit. http://www.csc.fi/tutkimus/Laskentapalvelut/laskenta/palvelimet/louhen_skaalautuvuustestit (28.5.2009)

[24] Newman M., Barkema G. Monte Carlo Methods in Statistical Physics. Oxford University Press, 1999.

[25] Harri Pölönen. Biological signals and image processing computations on TUT's PC grid. slides of Experiences in grid computing at TUT -seminar. Tampere. 20.11.2008

[26] Bahi J., Contassot-Vivier S., Couturier R. Parallel Iterative Algorithms: From Sequential to Grid Computing. Chapman & Hall/CRC, 2007.

[27] Mühlenstädt T. Simplex Based Space Filling Designs. 2008. http://www.statistik.tu-dortmund.de/muehlenstaedt_en.html (30.6.2009)

[28] Äijö T. and Lähdesmäki H. Learning gene regulatory networks from gene expression measurements using non-parametric molecular kinetics. Bioinformatics, X(Y), Z1-Z2. 2009. (julkaisematon)

[29] BOINC stats. http://www.boincstats.com/ (30.1.2009)

[30] Folding@home. http://folding.stanford.edu/ (5.6.2009)

[31] Einstein@home. http://einstein.phys.uwm.edu/ (5.6.2009)

[32] GPUgrid.net. http://www.gpugrid.net/ (5.6.2009)

[33] Anderson D. BOINC: A System for Public-Resource Computing and Storage. Fifth IEEE/ACM International Workshop on (2004), pp. 4-10. http://boinc.berkeley.edu/grid_paper_04.pdf (5.6.2009)

[34] Anderson D., Korpela E., Walton R. High-Performance Task Distribution for Volunteer Computing. e-Science and Grid Computing, 2005. First International Conference on 1-1 July 2005 Page(s):8 pp. - 203

[35] BOINC server. http://boinc.berkeley.edu/trac/wiki/ServerIntro#cookbook-useraccount (16.4.2009)

[36] BOINC server virtual machine. http://boinc.berkeley.edu/trac/wiki/VmServer (16.4.2009)

[37] VMware server. http://www.vmware.com/products/server/ (16.4.2009)

[38] VirtualBox. http://www.virtualbox.org/ (16.4.2009)

[39] BOINC server components. http://boinc.berkeley.edu/trac/wiki/ServerComponents (16.4.2009)

[40] BOINC system requirements. http://boinc.berkeley.edu/wiki/System_requirements (16.4.2009)

[41] Jarifa. http://jarifa.unex.es/ (1.10.2009)

[42] Tammisto T. Java-pohjaisen hilalaskentajärjestelmän arkkitehtuuri. Diplomityö. Tampereen teknillinen yliopisto, 2006.

[43] Kanniainen, J., Piché, R. and Mikkonen, T. 2008. Use of distributed computing in derivative pricing. 5th International Conference on Computational Management Science, 26-28 March 2008, Imperial College London

[44] http://garfield.web.cern.ch/garfield/ (27.5.2009)

[45] Aleksa M. Performance of the ATLAS Muon Spectrometer. Väitöskirja. Technical University of Vienna, 1999.

[46] http://cygwin.com/ (27.5.2009)

[47] http://vmware.com/appliances/learn/ (13.1.2009)

A. NAPOLEON MATLAB CODE

```
function price=napoleon(S0,v0,r,RR,kappa,psi,C, F, T,M)

%We use the following parameter values:
%nn =30;                 % number of stocks
%S0= ones(nn,1);         % initial stock prices
%v0=0.2*ones(nn,1);      % initial stock price volatilities
%r = 0.05;               % risk-free interest rate
%kappa = 0.1*ones(nn,1); % volatility reversion rates
%psi = 0.5*ones(nn,1);   % volatility-volatilities
%C = 0.1;   % coupon base payoff
%F = 1;     % face value
%T = 10;     % contract duration (in years)

%Correlation matrix
%we suppose that the correlation between the returns of
%stock i and stock j is 0.5 for all i,j=1...n, i \neq j
%and that the correlation between stock returns and
%volatilities is -0.5. This can be executed for nn stocks
%as follows:
%a_1=0.5; a_2=0.5; a_3=0.5;
%RR= [ a_1*eye(nn)+a_2*ones(nn) -a_3*eye(nn)
%    -a_3*eye(nn)          eye(nn)     ];

%-----------------------------------------------
nn = length(S0)   % number of stocks
Dt=1/360;         % time step size (one day)
sRR = sqrtm(RR);
p=zeros(M,1);
coupon=zeros(1,T);
for m=1:M  % simulations loop
    v=v0;
    x=log(S0);
    for t=1:T % year loop
        mustar = inf(nn,1); % reset at start of year
        for k=1:12 % month loop
            x0=x;  % logprice at start of month
            zz=sRR*randn(2*nn,30);
            for d=1:30  % day loop
               z=zz(:,d);
```

```
                for i = 1:nn % stock loop
                    svdt=sqrt(v(i)*Dt);
                    x(i)=x(i)+(r-0.5*v(i))*Dt+svdt*z(i);
                    v(i)=max(0,v(i)+kappa(i)*(v0(i)-v(i))*Dt ...
                        +psi(i)*svdt*z(i+nn) ...
                        +0.25*psi(i)2*Dt*(z(i+nn)2-1));
                end % of stock loop
            end % of day loop
            mustar = min(mustar,exp(x-x0)-1); % least monthly return
        end % of month loop
        coupon(t) = max(0,C+mean(mustar));
    end  % of year loop
    p(m)=F*sum(coupon.*exp(-r*(1:T)));
end % of simulations loop
price=mean(p);
```

B. COMPUTING NAPOLEON IN TECHILA GRID WITH PEACH FUNCTION

Main Program:

```
%Function run_napoleon_grid is executed in the the local PC. It initializes
%the grid, compiles the computing intensive Asian routine in 'asian_haj.m'
%file, computes  the Asian routine in the grid and draws a graph.
%function run_napoleon_grid

fprintf('Execution started!\n');

% We use the following parameter values:
nn = 30;                % number of stocks
S0 = ones(nn,1);       % initial stock prices
v0 = 0.2*ones(nn,1);    % initial stock price volatilities
r = 0.05;               % risk-free interest rate
kappa = 0.1*ones(nn,1);% volatility reversion rates
psi = 0.5*ones(nn,1);   % volatility-volatilities
C = 0.1;                % coupon base payoff
F = 1;                  % face value
T = 10;                 % contract duration (in years)

% Correlation matrix, RR
% we suppose that the correlation between the returns of
% stock i and stock j is 0.5 for all i,j=1...n, i \neq j.
% The correlation between the return and the volatility of
% stock i is -0.5 for all i=1...n and the corelation
% between the return of stock i and volatility of stock j,
% i \neq j, is 0, i,j = 1...n.
% This can be executed for nn stocks
% as follows:
 a_1=0.5; a_2=0.5; a_3=0.5;
 RR = [ a_1*eye(nn)+a_2*ones(nn) -a_3*eye(nn)
    -a_3*eye(nn)          eye(nn)     ];

%The parameters of stock price diffusion
M = 200e3; %The number of trajectories
MaxMPerClient = 2000; % The maximum number of trajectories in each of the jobs
```

```
% Root directory of the Grid Management Kit
gmkRoot='../../../';

addpath([gmkRoot '/grid/Matlab']); %Add grid toolbox functions to the path

% Check that MaxMPerClient is not bigger than M
if (MaxMPerClient > M)
    MaxMPerClient = M;
end

% Number of jobs
jobs = ceil(M/MaxMPerClient);
% The number of trajectories in each of the jobs approximately
MPerClient = floor(M/jobs);

% The parameters of eachs job.
jobparams = cell(jobs,1);
for id=1:jobs
    if id <= mod(M,jobs)
        jobparams{id}=[MPerClient+1,id];
    else
        jobparams{id}=[MPerClient,id];
    end
end

% Execute the "napoleon_haj" function using peach (parallel each). The peach
% parameters are following:
% - the name of the function to be executed
% - the parameters for the function (<param> is special parameter which
%   gets an element from peachvector - each of the grid jobs gets different
%   element)
% - data files to be transferred to the clients, for example:
%   {{'data.m', 'c:\\temp\\data.m'}, {'data2.m', 'c:\\temp\\data2.m'}}
% - peach cell array containing the values to be delivered to the clients and
%   replacing <param>.
% The result of the peach is vector containing the results of each of the
% computations as its elements.

%-----------------
prices = peach('napoleon_haj', ...
    {S0, v0, r, kappa, psi, C, F, T, RR, nn,'<param>'}, ...
    {}, ...
    jobparams, ...
     'gmkRoot', gmkRoot);

prices = cell2mat(prices);
napoleon_price = sum(prices)/M;
```

Program executed on the grid clients:

```
%Function napoleon_haj is the computive intensive part of the program. It is
%the one that will be compiled and then executed in the grid clients.
function jobprice=napoleon_haj(S0, v0, r, kappa, psi, C, F, T, RR, nn, jobparams)

% The job specific parameters
jobparams = cell2mat(jobparams);
MPerClient = jobparams(1);
jobidx = jobparams(2);


%if jobidx == 3
%    error('Hello world! This is an error');
%end

% Initialize random number generator
% This is needed for all the different clients to generate different
% random number. If the random number generator is not seeded it will
% always produce the same sequence of numbers, which would mean that each
% job would produce the same result.
randn('state',jobidx);

%% Pricing loops

%-------------------------------------------------
Dt=1/360;

sRR = sqrtm(RR);
p=zeros(MPerClient,1);
coupon=zeros(1,T);
for m=1:MPerClient  % simulations loop
    v=v0;
    x=log(S0);
    for t=1:T % year loop
        mustar = inf(nn,1); % reset at start of year
        for k=1:12 % month loop
            x0=x;  % logprice at start of month
            zz=sRR*randn(2*nn,30);
            for d=1:30  % day loop
               z=zz(:,d);
               for i = 1:nn % stock loop
                   svdt=sqrt(v(i)*Dt);
                   x(i)=x(i)+(r-0.5*v(i))*Dt+svdt*z(i);
                   v(i)=max(0,v(i)+kappa(i)*(v0(i)-v(i))*Dt ...
                       +psi(i)*svdt*z(i+nn) ...
                       +0.25*psi(i)^2*Dt*(z(i+nn)^2-1));
```

```
            end % of stock loop
        end % of day loop
        mustar = min(mustar,exp(x-x0)-1); % least monthly return
    end % of month loop
    coupon(t) = max(0,C+mean(mustar));
end  % of year loop
p(m)=F*sum(coupon.*exp(-r*(1:T)));
end % of simulations loop

jobprice=sum(p)
```

C. GARFIELD INPUT FILE

```
// Retrieve the argument
Parse Arg job

// Vector of B-fields and angles
Global bfield = 0.25 + 0.05*row(9)
Global angles = 5*(row(19)-1)

// Define a B field
&MAG
comp 0 0 0.6 T

// Gas calculations
&GAS
If type(job)#‘Number‘ Then
   Say "The argument following -arg ({job}) is missing or not of type Number."
Elseif job<1 | job>size(bfield)*size(angles) Then
   Say "The job sequence number ({job}) is out of range."
Else
   Global job = job - 1
   Global ib = 1+entier(job/size(angles))
   Global ia = 1+job-size(angles)*(ib-1)

   pressure 3 bar
   temperature 25 C
   magboltz argon 93 co2 7 ...
      e-range 100 300000 n-e 25 ...
      b-field {number(bfield[ib])} ...
      angle   {number(angles[ia])} ...
      coll 5
   write "output.gas" "B{ib}A{ia}"
   &MAIN
Endif

// End of job
Say "Job completed."
```

D. MATLAB MAIN PROGRAM FOR RUNNING GARFIELD ON TECHILA

```
%Function run_garfield_grid is executed in the the local PC. It initializes
%the grid, computes the Garfield with commandline:
%       >> garfield-9.exe < input - arg i
%where i is an integer 1..numberOfJobs
% - cygwin1.dll, cygX11-6.dll and garfield-9.exe should be in the directory
%   to make the binarybundle
% - input (file) should be in the directory to make the databundle
function run_garfield_grid

fprintf('Execution started!\n');

jobs = 171; % number of jobs

% Root directory of the Grid Management Kit
gmkRoot='c:/TechilaGMK/gmk';

addpath([gmkRoot '/grid/Matlab']); %Add grid toolbox functions to the path

fprintf('Initializing grid!\n');

%Initialize the grid using toolbox function initgrid()
status = initgrid(gmkRoot);

%Defines the grid, grid bundle, and grid result management interface
%variables GRID, GRID_BUNDLES, GRID_PROJECTS and GRID_RESULTS as global,
%the management interfaces has been assigned in the initgrid() toolbox
%function.
global GRID;
global GRID_BUNDLES;
global GRID_PROJECTS;
global GRID_RESULTS;
global CLEANUP_MODE_ALL;

try %Try to catch block to trace back possible errors and their locations.

    %Checks the status of the initialization using toolbox function
    %checkStatusCode(). If any errors have occurred, the exception is
```

```
%thrown and the catch block will print out the error to the console.
checkStatusCode(status);

%Opens a project handle for a new grid project. The purpose for the
%handle is to be an identifier in the Management Interface. It is
%used to bind together bundle and project related information between
%function calls.
handle = GRID.open();

%Defines the name of the bundle which will contain the executable
%binary of the Garfield.
bundleName = '{user}.Garfield';

%The version of the binary bundle.
%  -If any changes are made to the Garfield-9.exe, -
%  -this version number has to be increased.  -
%The version number is in format x.y.z, where each of the parts (x,y,z)
%can be any whole number. The using of the parts is not fixed anyway,
%but the newer version of the bundle must have greater number.
%For example 0.1.0 is greater than 0.0.1000. Good way to use the format
%is to increase the value of z for minor changes, the value of y for
%major changes and the value of x for the bigger releases.
bundleVersion = '0.0.10';

%The method bundleVersionExist()is used to check if the binary bundle
%with the specified version is already compiled and committed to the
%grid server.
%If the bundle does not exist on the server, the separated helper
%function createBinaryBundleOfAsian() is called to compile the binary
%and create the binary bundle.
if ~GRID_BUNDLES.bundleVersionExists(bundleName, bundleVersion);
    createBinaryBundleOfGarfield(handle, bundleName, bundleVersion);
end

%Construct the framework internal name of the binary bundle, to be used
%in the import statement later.
binarybundlefullname = [ bundleName ';specification-version=' ...
    bundleVersion ];

%Define a unique name (using timestamp) for the data bundle which is
%used to deliver the data file. Version number can stay as 0.0.1,
%because the name of the bundle is different each time.
databundlename=[ 'data_' num2str(now*10000000000) ];
databundleversion='0.0.1';
databundlefullname = [ bundleName '.data.{user}.' databundlename ];

%Creates a databundle of the variables. The parameters are:
```

```
    %- project handle
    %- the prefix of the bundle name (binary bundle name is used in
    %  this case)
    %- the name of the bundle
    %- description for the bundle
    %- the bundle version
    %- the other bundles to be imported (binary bundle)
    %- thename the bundle is exported as (so other bundles may import it)
    %- category name (can be always {user})
    %- resource id used by the computation to find the right bundle for
    %  datafiles etc.
    %- the bundle which contains the binary to be executed
    %- extraparameters, in this case only ExpirationPeriod which tells how
    %  long (in milliseconds) the bundle is stored on the clients after the
    %  computation before it is deleted
    %- files to be included to the bundle
    createBundle(handle, [bundleName '.data'], databundlename, ...
'description', databundleversion, binarybundlefullname, ...
databundlefullname, '{user}', 'inputdata', binarybundlefullname, ...
    {{'ExpirationPeriod', '60000'}}, {{'input', 'input'}});

    %Binds the binary bundle to be the base for the project to be created.
    checkStatusCode(GRID_BUNDLES.useBundle(handle, databundlefullname));

    %Priority of the project executed in the grid. 4 is normal priority,
    %7 is the lowest and 1 is the highest.
    projectPriority = 4;

    projectDescription = 'Garfield calculation';

    %Create a new project based on the bundle using toolbox function
    %createProject().
    checkStatusCode(createProject(handle, projectPriority, ...
        projectDescription, {{'jobs', jobs}}));

    %Get the id of the created project and write it to the console. This
    %information is also used in the output filename.
    projectid = GRID_PROJECTS.getProjectId(handle);
    fprintf('Project created with id: %i\n', projectid);

    %Waits the completion of the project using toolbox function
    %waitAndRetrieveResults(). The toolbox function writes the progress of
    %the project as percent value to the console. When the project is
    %finished, the compressed result file is downloaded and extracted to
    %partial result files. The names of the result files are returned.
    resultfiles = waitAndRetrieveResults(handle);
```

```
    %Loops for the number of the result files and saves the files to local
    %machine with the name Ar_93_CO2_7_3bar_25C_B_partN, where N is the
    %number of job.
    for p=1:size(resultfiles,1)
        copyfile(char(resultfiles(p)), ...
            strcat('Ar_93_CO2_7_3bar_25C_B_part',num2str(p),'.gas'));
    end

    %Get and print some statistics about the project.
    cputime = GRID_PROJECTS.getUsedCpuTime(handle);
    realtime = GRID_PROJECTS.getUsedTime(handle);

    fprintf('CPU time used %i d %i h %i m %i s.\n', ...
        floor(cputime/3600/24), ...
        floor(cputime/3600 - floor(cputime/3600/24)*24), ...
        floor(cputime/60 - floor(cputime/3600)*60), ...
        floor(cputime - floor(cputime/60)*60));
    fprintf('Real time used %i d %i h %i m %i s.\n', ...
        floor(realtime/3600/24), ...
        floor(realtime/3600 - floor(realtime/3600/24)*24), ...
        floor(realtime/60 - floor(realtime/3600)*60), ...
        floor(realtime - floor(realtime/60)*60));

    %Removes the temporary files etc. created by the grid management
    %interface.
    GRID.cleanup(handle, CLEANUP_MODE_ALL);

    GRID.close(handle); %Closes the project handle.

%Catches any possible exceptions caused by any error during the
%execution and prints out the error message to console using toolbox
%function printError().
catch ME1
    printError(ME1);
end

%Unloads the grid management interface and removes the temp. directories.
GRID.unload(true, true);

end

%This function is a helper routine for making binary bundle of Garfield
%computations.
%Parameters are handle = the project grid handle, bundleName = the name of
%the binary bundle, bundleVersion = the version of the binary bundle
function createBinaryBundleOfGarfield(handle, bundleName, bundleVersion)

bundleDescription = 'Garfield binary';
```

```
%The name of the binary compiled above, the supported processor
%architecture and the supported Operating System are described here.
%In this example to different suported Operating Systems are defined:
%Windows XP and Windows Vista. They are separeted with comma (,).
natives = ['garfield.bat;osname=Windows XP;processor=x86,'...
           'garfield.bat;osname=Windows Vista;processor=x86'];

%extraparams are used as a definitions for compiled binary bundle.
%    Copy = Files included in the bundle to be copied to the execution
%directory. This contains the files included in the binary bundle which
%are needed for the execution of the binary. Only the files that are not
%mentioned in the Executable or Parameters are needed to be included
%to this definition. The files mentioned in the Executable or Parameters
%are copied automatically.
%    InternalResources = The files included in the binary bundle that
%can be referenced in the Executable or Parameters. This contains at
%least the binary executable, but optionally also input
%files etc. that can be used in Parameters.
%    Executable = The resource of the binary file to be executed on the
%grid clients. This file is copied automatically to the execution
%directory.
%    OutputFiles = The files that are written out from the binary and
%whose content should be sent to the server as the result.
%    Parameters  =  Command line arguments for the binary.
extraparams = {
    {'Copy', 'cygwin1.dll, cygX11-6.dll, garfinit'}, ...
    {'InternalResources', strcat('cygX11-6.dll, cygwin1.dll, garfield-9.exe,',...
                                 'garfield.bat, garfinit')}, ...
    {'ExternalResources', 'input;resource=inputdata;file=input'}, ...
    {'Executable', '%I(garfield.bat)'}, ...
    {'OutputFiles', 'output;file=output.gas'}, ...
    {'Parameters', '%I(garfield-9.exe) -batch -noterminal -arg %P(jobidx) %R(input)'}};

%fileparams are used to describe the resource names and locations of
%the files to be included in the binary. These are the same names that
%were defined in the InternalResources above. The file locations must
%contain the _full_path_ to the files if they are not located in the
%same directory as the Matlab file.
fileparams = {
    {'garfield.bat', 'garfield.bat'}, ...
    {'garfield-9.exe', 'garfield-9.exe'}, ...
    {'garfinit', 'garfinit'}, ...
    {'cygwin1.dll', 'cygwin1.dll'}, ...
    {'cygX11-6.dll', 'cygX11-6.dll'}};

%A new signed bundle is made of the compiled binary using toolbox
```

```
%method createSignedBundle() with the parameters define above.
createSignedBundle(handle, bundleName, bundleDescription, ...
    bundleVersion, natives, extraparams, fileparams);

end
```

E. PROGRAM TO CREATE NAPOLEON JOBS

```
#include <unistd.h>
#include <cstdlib>
#include <string>
#include <cstring>

#include "boinc_db.h"
#include "error_numbers.h"
#include "backend_lib.h"
#include "parse.h"
#include "util.h"

#include "sched_config.h"
#include "sched_util.h"
#include "sched_msgs.h"

#define REPLICATION_FACTOR  1

// globals
//
char* wu_template;
DB_APP app;
int start_time;
int seqno;

// create one new job
//
int make_job(int number, int sims, int remain) {
    DB_WORKUNIT wu;
    char name[256], command_line[256];
    const char* infiles[2] = {"napoleon_haj.ctf", "input_napoleon.mat"};
    int retval;

    // make a unique name (for the job)
    //
    sprintf(name, "napoleon_%d_%d", number, start_time);
    sprintf(command_line, "%d %d %d result.mat input_napoleon.mat",
number, sims, remain);

    // Fill in the job parameters
```

```
    //
    wu.clear();
    wu.appid = app.id;
    strcpy(wu.name, name);
    wu.rsc_fpops_est = 1e13;
    wu.rsc_fpops_bound = 3e13;
    wu.rsc_memory_bound = 1e8;
    wu.rsc_disk_bound = 1e8;
    wu.delay_bound = 3600;
    wu.min_quorum = REPLICATION_FACTOR;
    wu.target_nresults = REPLICATION_FACTOR;
    wu.max_error_results = REPLICATION_FACTOR*4;
    wu.max_total_results = REPLICATION_FACTOR*8;
    wu.max_success_results = REPLICATION_FACTOR*4;

    // Register the job with BOINC
    //
    return create_work(
        wu,
        wu_template,
        "templates/napoleon_result.xml",
        "../templates/napoleon_result.xml",
        infiles,
        2,
        config,
command_line
    );
}

void main_loop(int njobs, int sims) {
    int retval;
    int remain;
    int sims_wanted;

    // jobs with number of job less or equal to remain
    // does one extra simulations. This way the total number
    // of simulations is equal to the number of simulations wanted
    sims_wanted = sims;
    sims = floor(sims_wanted/njobs);
    remain = sims_wanted-sims*njobs;

    for (int j=1; j<=1; j++) {
        check_stop_daemons();

        log_messages.printf(MSG_DEBUG,
            "Making %d jobs\n", njobs
        );
```

```
        log_messages.printf(MSG_DEBUG,
            "%d total simulations\n", sims_wanted
        );
        log_messages.printf(MSG_DEBUG,
            "%d sims per job, %d remain\n", sims, remain
        );
        for (int i=1; i<=njobs; i++) {
            retval = make_job(i, sims, remain);
            if (retval) {
                log_messages.printf(MSG_CRITICAL,
                    "can't make job: %d\n", retval
                );
                exit(retval);
            }
        }
        // Now sleep for a few seconds to let the transitioner
        // create instances for the jobs we just created.
        // Otherwise we could end up creating an excess of jobs.
        sleep(5);
    }
}

int main(int argc, char** argv) {
    int i, retval, jobs;
    int sims;
    jobs = 0;
    sims = 0;

    for (i=1; i<argc; i++) {
        if (!strcmp(argv[i], "-d")) {
            log_messages.set_debug_level(atoi(argv[++i]));
} else if (!strcmp(argv[i], "-jobs")) {
    jobs = atoi(argv[++i]);
        } else if (!strcmp(argv[i], "-sims")) {
            sims = atoi(argv[++i]);
        } else {
            log_messages.printf(MSG_CRITICAL,
                "bad cmdline arg: %s", argv[i]
            );
        }
    }

    if (jobs < 1) {
        log_messages.printf(MSG_CRITICAL,
            "invalid number of jobs\n"
        );
        exit(1);
```

```
    }

    log_messages.printf(MSG_NORMAL, "sims %d\n", sims);

    if (sims < jobs) {
        log_messages.printf(MSG_CRITICAL,
            "number of simulations %d is less than number of jobs %d\n", sims, jobs
        );
        exit(1);
    }

    log_messages.printf(MSG_NORMAL, "sims %d\n", sims);

    if (config.parse_file("..")) {
        log_messages.printf(MSG_CRITICAL,
            "can't read config file\n"
        );
        exit(1);
    }

    retval = boinc_db.open(
        config.db_name, config.db_host, config.db_user, config.db_passwd
    );
    if (retval) {
        log_messages.printf(MSG_CRITICAL, "can't open db\n");
        exit(1);
    }
    if (app.lookup("where name='napoleon'")) {
        log_messages.printf(MSG_CRITICAL, "can't find app\n");
        exit(1);
    }
    if (read_file_malloc("../templates/napoleon_wu.xml", wu_template)) {
        log_messages.printf(MSG_CRITICAL, "can't read WU template\n");
        exit(1);
    }

    start_time = time(0);
    seqno = 0;

    log_messages.printf(MSG_NORMAL, "Starting\n");
    log_messages.printf(MSG_NORMAL, "total sims %d\n", sims);

    main_loop(jobs, sims);

    log_messages.printf(MSG_NORMAL, "Done!\n");
}
```

Tampereen teknillinen yliopisto
PL 527
33101 Tampere

Tampere University of Technology
P.O.B. 527
FI-33101 Tampere, Finland