Technical University of Crete
School of Electronic and Computer Engineering

A dependency-aware task distribution extension to the BOINC framework for volunteer computing

Pissadakis Emmanouil

ADVISOR: *Prof. Dionisios Pnevmatikatos*

COMMITTEE: Assist. *Prof. Vasilis Samoladas*

Dr. Dimitris Theodoropoulos

Acknowledgments

Many people helped and inspired me in order to complete this thesis. Firstly, I would like to thank my professor Dionisios Pnevmatikatos for giving me the opportunity to work in the field that I am interested in. Secondly, special thanks to Dimitris Theodoropoulos for the cooperation and his constant technical support during this thesis, as well as to Gregorios Chrisos, Iosif Koidis and George Charitopoulos for their helpful advices. Additional thanks to my friends Manos, Spyros and Antonis for their vital help. Finally, I am indebted to my family for their mental support during my studies.

Abstract

Volunteer Computing is one of the distributed computing paradigms that has gained attention in recent years. It is used by a number of scientific researchers to perform scientific projects such as climate prediction, search for extraterrestrial live and protein structure prediction. It is an innovative approach to high performance computing that relies on volunteers who donate their personal computers' unused resources to a computationally intensive research project, as well as provides scientists with the necessary means for performing projects that require huge resources. BOINC (Berkeley Open Infrastructure for Network Computing) is an open-source framework for solving large-scale computational problems by means of volunteer computing. In contrast to massive parallel computing, applications are distributed into a large number of heterogeneous client computers connected by the Internet where each computer is assigned individual tasks that can be solved independently without the need of communication upon the clients. A BOINC-based project provides its own servers. Hosts download application's executables and data files from servers, carry out tasks (by running applications against specific data files) and upload the output files. However various problems exist while deploying applications over these heterogeneous machines using BOINC. The tasks of each application had to be independent due to the lack of communication between the clients, otherwise it is not compatible. Furthermore porting application to BOINC middleware is a very complex process. Several server daemons had to be implemented to achieve that. To resolve these issues, this thesis proposes a framework based on the Boinc infrastructure, the mCluster software framework. MCluster adopts a task-based programming model designed to resolve the existed dependencies. Finally, a source to source translator is included in this framework in order to transform this application into BOINC compatible tasks, ready to be executed from the available clients by implementing all the appropriate daemons that BOINC requires.

Keywords: Volunteer Computing, BOINC middleware, task based programming model

Table of contents

List of Figures

List of Tables

Chapter 1

Introduction

Over the past few years, distributed computing has become more and more popular. That occurs because the processing power in the computers has greatly increased, as well the internet became available to hundreds of millions of users. Distributed computing is an efficient way to solve large scale problems that require a lot of processing power. These problems are divided into smaller chunks (tasks) which are going to be distributed to machines over the internet. Those remote machines will execute them and return the individual results to a central server. Due to the increased demands of the researchers, this field of computer science mostly consists of scientific projects. Volunteer computing that is going to be analyzed in next sections, so as grid computing are forms of distributed computing.

Grid computing [6,9] is related to volunteer computing, but there are also some distinctions. A grid is usually a set of computers or clusters that are owned by universities, companies or organizations. Furthermore, users gain access to computing resources even though they don't know where they are located [6]. On the other hand, volunteer computing [2,3] (also known as Public Resource Computing) focuses on utilizing the available resources in the personal computers of individual volunteers rather than large networks of computers with persistent network connections and long uptimes. The increasing number of connected machines over the internet provides more computational power and storage than large clusters or grids. Grid computing is more reliable, because tasks are usually running in clusters always available for processing, in contrast to volunteer computing in which anyone can become a volunteer regardless his location and without giving information about his reliability.

There are several types of volunteer computing frameworks [4,7] but the most popular and reliable middleware is the Berkeley Open Infrastructure for Network Computing (BOINC) [24,25,26], attached to several scientific projects, where participants can volunteer their available resources. Most of these projects [1,23] have hundreds of thousands even millions of participants, so the performance of each project is based on the way that data are distributed to them.

These available resources are mostly personal PCs but the increasing performance of mobile devices trends them to a similar performance resource. Nowadays, mobile devices such as smartphones and tablets are becoming increasingly powerful and rising quickly in popularity. Mobile

devices follow us everywhere, allowing us to work and entertain ourselves at any venue. Thus, they are replacing desktops as our personal computers. We already have signs of smartphones becoming more popular than traditional desktop computers. A recent survey of users reveals that email, Internet access and a digital camera are the three most desirable features in a mobile phone while the consumers wanted these features to be as fast as possible. The increasing sales of more powerful phones also indicate consumer demand for more efficient mobile devices.

The mobile devices themselves could be the source of computing power. Recent technological advances have greatly improved the performance of smartphones/tablets in terms of CPU/GPU speed, memory size and storage space. A smartphone or tablet can provide a considerable amount of computing power that may be even comparable to the computing power of a desktop computer. In addition, unlike personal computers, mobile devices are rarely powered off, even when the owners are sleeping, which translates into hours of unutilized computing resources. There is great potential if we can make use of these idle computing resources.

1.1 Motivation

BOINC is based on the client-server architecture suited to most of those scientific projects, but its implementation requires project developers to be very familiar with this architecture. An application must be divided into two sides, the client's and the server's one, which is very difficult due to the lack of BOINC tools and documentations. BOINC is a very useful framework mainly for scientists because it can endure large scale computational requirements.

In order for a project to take advantage of volunteer computing, it must be able to be broken down into smaller parallel tasks and be compatible with the client-server architecture, where the client executes them, and the server gains the individual results. Furthermore, another important disadvantage of BOINC is its inability to support dependencies between the tasks of each project. If a project has to be divided in tasks that have dependencies among them, it cannot be ported to BOINC [25].

Considering all above-mentioned, this thesis initial goal is to implement an easy-to-use by the developers task based programming model that will resolve any task dependencies an application may have.

As a case study this thesis aims to implement this programming model into the BOINC framework in order to resolve any task dependencies that might occur in a project. Finally, a specific mechanism has to be designed to convert a simple non BOINC-compatible application, into a ready-to-be-executed BOINC project.

1.2 Contributions

The contributions of this thesis are

- The design of a task-based programming model targeting distributed systems. According to this, programmers via annotations can determine which piece of code they want to be executed in parallel. Moreover, any task dependencies will be resolved, as well as data hazards in task out-of-order execution.

- The implementation of the task-based programming model over the BOINC infrastructure. Via this model, projects can be divided into client and server side. The server side is responsible for creating tasks (workunits), distributing them to the clients, which are going to execute them, upload the result to the server which gathers and further analyzes them. This requires the implementation of a framework (source to source translator) which transforms a sequential application to a BOINC compatible project application. To achieve this, BOINC required daemons have to be specified and implemented according to application requirements. Likewise, it has to initialize all BOINC project configuration parameters.

- Extension in BOINC architecture in order to support dependencies between different project tasks. BOINC does not offer any mechanism that is able to resolve task dependencies [15]. Exploiting the properties of the specific programming model in combination with structural changes in BOINC architecture, an inter-BOINC mechanism will be designed for handling this problem.

1.3 Thesis Outline

- **Chapter 2-Background**

  This chapter presents related work to distributed computing. It describes how computers are organized in a distributed system and how volunteer computing became a useful tool for processing large scale problems.

- **Chapter 3-The mCluster task based programming model**

  In this chapter we present the mCluster task based programming model. We describe the application programming interface, its structure and how task dependencies are resolved in a way to be capable of recognizing any data hazards that might occur between tasks.

- **Chapter 4-Implementation of mCluster using the BOINC infrastructure**

  This chapter analyzes the most popular and efficient infrastructures used in distributed computing BOINC [21] and Hadoop [33] as well as their limitations regarding mCluster requirements. Moreover we present the implementation of mCluster using the BOINC infrastructure. Initially, a mechanism was designed to transform the sequential application to BOINC–compatible. Furthermore, BOINC was redesigned according to mCluster features in order to support task dependencies between the tasks of a project.

- **Chapter 5 Performance evaluation**

  This chapter presents the performance evaluation of the mCluster in a specific environment. Four separate applications benchmarked in order to gain the available information for the evaluation.

- **Chapter 6 Conclusions and future work**

  This chapter summarizes our work and provides directions for future work**.**

Chapter 2

Background

2.1 Distributed computing

The growing popularity of the internet, the availability of powerful computers, the high-speed networks, as low-cost commodity components, are changing the way we do computing. Distributed computing has been an essential component of scientific computing for decades. It consists of a set of processes that cooperate to achieve a specific common goal. It is widely recognized that Information and Communication Technologies (ICTs) have revolutionized our everyday activities. Social networks represent a stepping stone in the on-going process of using the internet to enable the social manipulation of information and culture. Mostly social network sites are implemented on the concept of large distributed computing systems.

Various definitions of distributed systems have been given in the literature, none of them satisfactory and in no agreement with any of the others, so as it is sufficient to give a loose characterization [45]:

A distributed system is a collection of independent computers that appears to its users as a single coherent system

There are many aspects of this definition. Firstly, the components of a distributed system are autonomous in the sense that each of them has its own local memory. Secondly the users think that they are participating in a single system. According to the assumptions above, the components of a distributed system have to collaborate. The establishment of the collaboration is the main aspect of distributed computing. An important reference here is that the way that the components of those systems (which may be from a computer or a mobile device to a high performance supercomputer) communicate is mostly hidden from users. Likewise, the way they interact is in a uniform way, regardless of the time and place the interaction occurs.

2.1.1 Goals of distributed computing

Building a distributed system is not always a good idea and may not be so efficient. In order to worth the effort, a distributed system should meet some important goals that are going to be discussed below.

- Making resources accessible: The main goal of a distributed system is to make it easy for the users to access remote resources and share them in a controlled and efficient way.

- Distribution transparency: A distributed system that is able to be presented to its users as a unified computing platform is said to be transparent.

- Scalability: Scalability is one of the most important design goals for developers of distributed systems and is achieved when components don’t change when the scale of a system increases.

- Openness: An open distributed system is a system that offers services according to standard rules that describe the syntax and semantics of those services.

- Fault tolerance: Distributed systems must maintain availability even if a hardware, software or network fail occurs. It is usually achieved by recovery, redundancy and replication.

Computers in a distributed system can have different roles. A computer's role depends on the goal of the system and the computer's own hardware and software properties. There are two predominant ways of organizing computers in a distributed system. The first is the client-server architecture and the second is the peer-to-peer architecture.

2.1.2 Client server architecture

The client-server architecture is a way to dispense a service from a central source as shown in figure 2.1 (left). The basic idea of this architecture is that clients request job from the central server, in order to perform some task whereas the server is responsible for handling those requests and provide the appropriate data to them. A server can be determined as a simple unit that provides a service, possibly to multiple clients simultaneously and a client is a unit that consumes the service. It is not required for the clients to know extensive information about the service that server provides, as well

as the server has no obligation knowing in which way the data is going to be processed.

This architecture is commonly known to consist of different machines but even a single machine can have this architecture. For example, signals from input devices on a computer need to be generally available to programs running on the computer. The programs are clients, consuming mouse and keyboard input data. The operating system's device drivers are the servers, taking in physical signals and serving them up as usable input.

Despite the advantages that this architecture provides, such as the integration of services, inter-operation of data, unaware data processing location and the easy maintenance, it also introduces some disadvantages. The major drawback of this architecture is that the server is a single point of failure. The server is responsible for the job relating to the clients. If this server is down, the whole system will be down as well. There is no communication between the clients meaning that a failure of the server leads to the loss of all processed data.

2.1.3 Peer to peer architecture

The client-server model is appropriate for service-oriented situations. However, there are other computational goals for which a more equal division of labor is a better choice. The term peer-to-peer [8] is used to describe distributed systems in which labor is divided among all system components, illustrated in figure 2.1 (right). Any node can initiate a connection, despite the client server-architecture where the connection is always initialized from the client. In peer-to-peer architectures, clients behave as both servers and routers, where each node is autonomous, leading to the creation of a dynamic network in the sense that nodes enter and leave the network frequently. Another important characteristic is that nodes collaborate directly with each other and not through servers.

The most common applications of peer-to-peer systems are data transfer and data storage. For data transfer, each computer in the system contributes by sending data over the network. If the destination computer is in a particular computer's neighborhood, that computer helps send data along. For data storage, the data set may be too large to fit in any single computer, or too valuable to store on just a single computer. Each computer stores a small portion of the data, and there may be multiple copies of the same data spread over different computers. When a computer fails, its lost data can be restored from other copies. Due to the absence of a central point of failure, the major drawback of the client server architecture is solved, at the expense of an increased number of active connections.

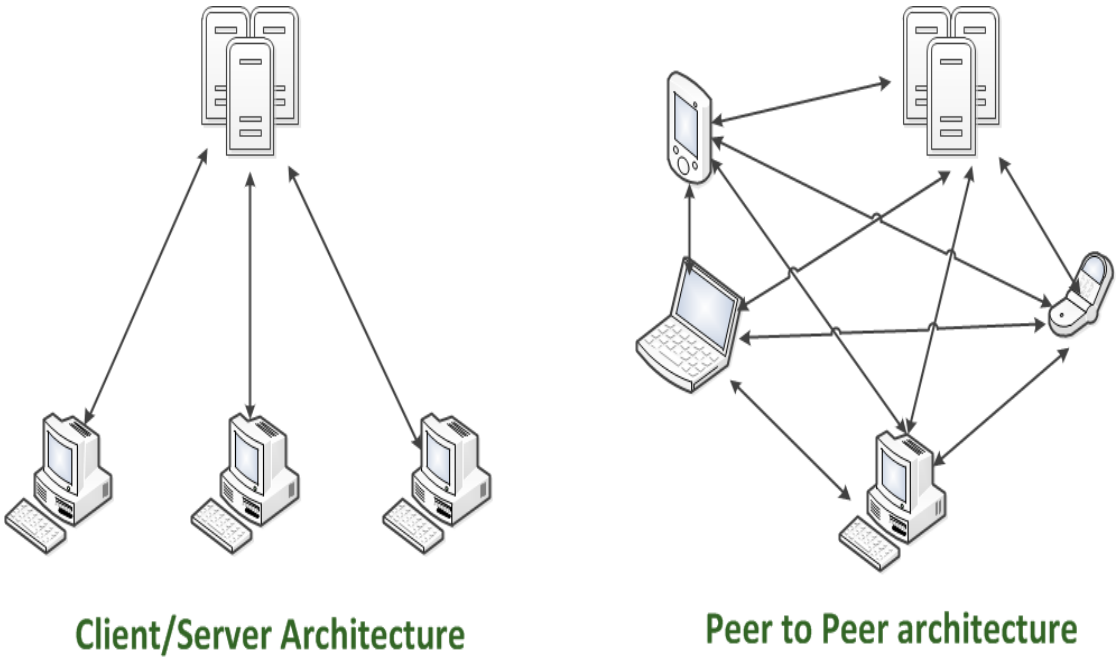


Figure 2.1: Client server architecture in contrast to peer to peer architecture

2.2 Volunteer computing

Volunteer Computing [2] is a form of distributed computing that allows volunteers to donate their computers' idle CPU cycles to a given application, or project. Recently, there has been a rapidly growing interest in volunteer computing. People worldwide can share their available computer resources in order to solve large parallel problems. The advantage of volunteer computing is that it is easy to use and accessible from anyone. Volunteers are typically members of the general public who own internet-connected personal computers. Organizations such as schools and businesses may also volunteer the use of their computers.

Volunteer computing is very important because a huge number of volunteers can supply more processing power that can be used from scientists. Most of these projects require a lot of processing power. Setting up supercomputer networks may increase prohibitive costs to research institutions. A research project that has limited funding but large public appeal can get huge computing power. In contrast, traditional supercomputers are extremely expensive, and are available only for applications that can afford them. Furthermore, it attracts more people in the field of science. Nowadays more than one billion of personal computers are used in volunteer computing supplying about 10 PetaFLOPS of computing power [41].

Although volunteer computing is growing fast, it has also to deal with some important issues. Volunteers are anonymous, which make them unreliable. The results they return, for some reasons,

(overclocking) may be incorrect as they intentionally return incorrect results, or claim excessive credit for results. Regarding the participant side, volunteer computing leads to more power consumption. Usually they donate their resources when they are idle so they consume more electricity. While the computer is in use and in parallel it executes a volunteer application, the usage of CPU, CPU cache and local storage increases. As a result the performance of the participant machines is reduced. Volunteer computing systems provide mechanisms for resolving those issues, which are mainly available in the client software.

2.3 Related work

2.3.1 Grid computing projects

Condor

Condor [39] is one of the oldest middleware systems used for distributed computing. It is a specialized batch system for managing compute-intensive jobs. Additionally, it provides a job querying mechanism, task policy, priority scheme, resource monitoring and management. In the past years, Condor usually operated in a workstation environment. The system aims to maximize the utilization of workstations with as little interference as possible between the job it schedules and the activities of people who own workstations. It uses a Condor pool in order to schedule the available jobs according to clients' demands. It is one of the most reliable systems because when an owner of a workstation resumes activity at a station, Condor checkpoints the remote job running on the station and transfers it to another workstation.

Nowadays Condor is a very useful tool for volunteer computing because of its excellent performance in environments where other systems are weak. In collaboration with its checkpoint mechanism, it provides fault tolerance by efficiently utilizing the available resources. Moreover, it can use all available resources even if they are not 100% available as a node, which keeps information for further computation. In order to achieve this, Condor uses the ClassAd language, a useful framework for matching the requests with the offers, migration and check pointing, as well as remote system calls (RPC). This process can preserve the local execution environment in order to make data inaccessible by remote workstations.

XtremWeb

XtremWeb[4] is a P2P project developed at University of Paris-Sud, France. It was originally designed to study execution models in the general framework of Global Computing and now there are distributions for Linux, Windows and Mac OS. It particularly focuses on multi-parameter applications, which need to be executed several times from different machines, in order to gather the appropriate information for further analysis. XtremWeb system is based in three entities, the coordinator, the workers and the clients.

The coordinator is responsible for the task management. It manages the tasks provided by the clients, schedules and distributes them to the available workers. According to this, communication is always started from workers. A worker can enter or leave the system unpredictably so the coordinator is responsible for distributing the tasks in an efficient way. To deal with this, the coordinator keeps information (CPU time, memory size operating system, history of worker) each time a worker request computation to provide an efficient way for distributing them.

Clients are responsible for submitting tasks to the coordinator. They connect to the coordinator to submit tasks or to fetch any previous submitted ones. Workers are usually volunteer entities like PCs that accept tasks for execution according to their characteristics. During the execution, they periodically communicate with the coordinator to update about the computation status. That way, if a failure occurs, coordinator knows about the "percentage" of execution that has already be done and attaches the task to the worker again from the last checkpoint, to resume task execution. When the task is completed, they send the result to the coordinator and then request new tasks.

2.3.2 Volunteer computing projects

Entropia

Entropia [40] is a distributed system which aggregates the raw desktop resources into a single logical resource. According to this assumption, the logical resource is reliable and predictable although any resource can be turned off or rebooted on any time (unreliable) and be heavily used by the users, so the available resources are not always the same (unpredictable). Moreover, it can provide high performance for applications and can be managed from a single administrative console. The Entropia system architecture is composed of three separate layers, the physical node management, the resource task and the job management layer as shown in figure 2.2.

Physical node management provides basic communication, naming, security, resource

management and application control. It is responsible for providing reliability to the system. If a computer is disconnected for a long time, when connected again, it has to pass through firewalls so as its ip address is changed. These are low-level reliability issues that this layer deals with. Furthermore, it monitors each client's available recourses and reports them to the master node, which are used during Resource Task Layer.

A distributed computing application often involves large amounts of computation jobs which has to break into smaller individual subunits, to be scheduled and run on a client machine. The Job Management layer is responsible for this decomposition in order to provide access to the status and the results of the generated subunits and results. The Resource Task layer takes the units of computation from the job management layer, matches them to the appropriate clients and schedules them for execution. The jobs are scheduled to the clients according the information that are gained from the Physical Node management layer which may not always be reliable. To deal with this problem Entropia supports multiple instances of heterogeneous schedulers.

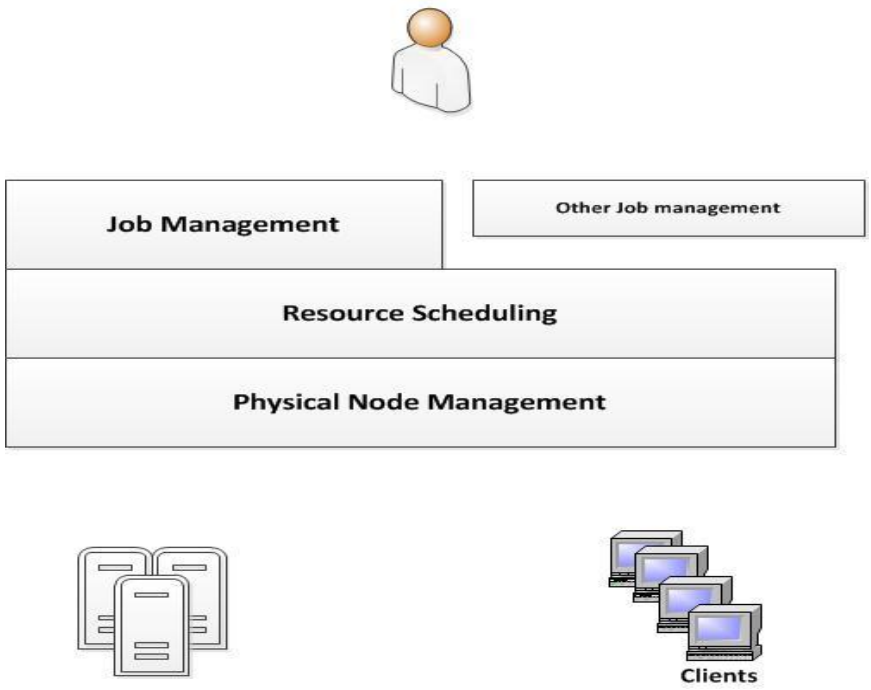


Figure 2.2 Entropia architecture

Hadoop

Hadoop [33] is a framework that allows distributed processing of large data sets across clusters of computers using simple programming models. It is designed to scale up from single servers to thousands of machines, each offering local computation and storage. Hadoop is an ideal tool for processing large amounts of data through the fault tolerance mechanism, which makes it remarkably reliable. The core of Apache Hadoop consists of a storage part (Hadoop Distributed File System (HDFS)) and a processing part (MapReduce). The Hadoop distributed file system is designed to have fault tolerance in an environment with thousands of nodes in which some of them may fail and has high throughput for data accessing. As shown in figure 2.3, according to HDFS [34] metadata and application data are stored separately. Similarly to other distributed systems, it stores metadata in a separate server named namenode. Application's data are stored in nodes named datanodes. It is a master/slave architecture. A HDFS cluster consists of a namenode and a master server managing the namespace of the system and regulates access to files from the clients. In contrast, there are many datanodes managing the storage they are connected and run. Hadoop MapReduce is a software framework for easily writing applications which process vast amounts of data (multi-terabyte data-sets) simultaneously on large clusters of commodity hardware in a reliable, fault-tolerant manner. A MapReduce job usually splits the input data-set into independent chunks which are processed by the map tasks in a completely parallel manner. This framework sorts the outputs of the maps, which are then input to the reduce tasks. Typically, both the input and the output of the job are stored in a file-system.

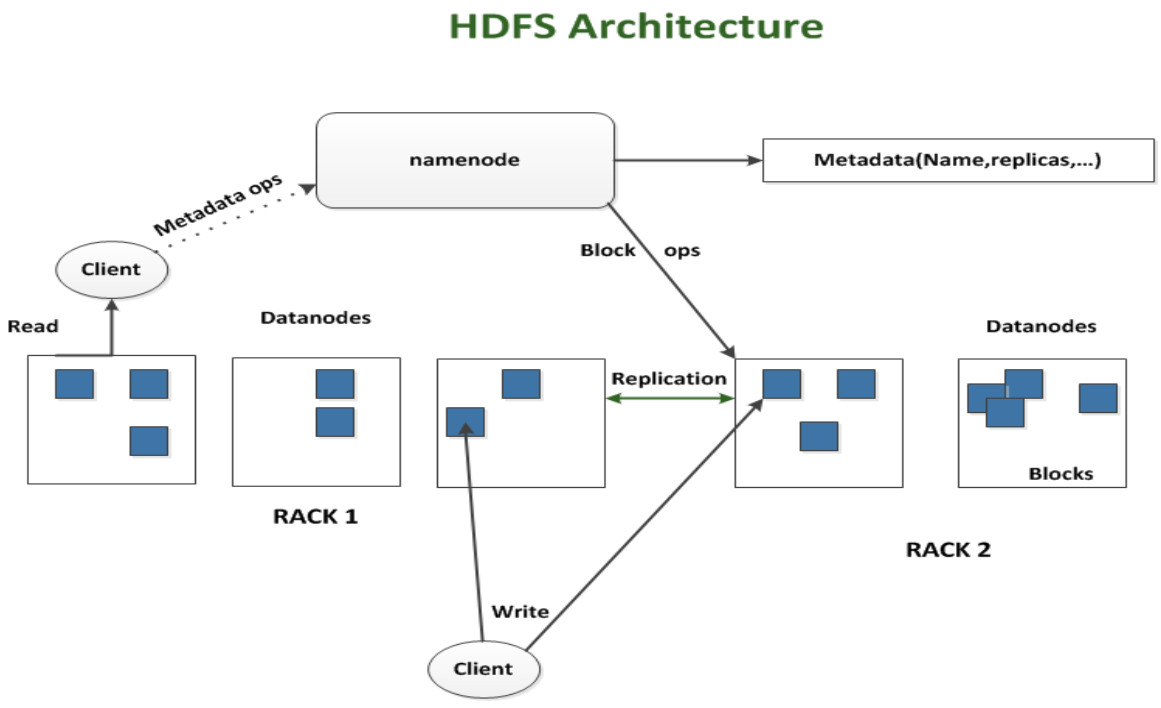


Figure 2.3: The Hadoop distributed file system architecture

BOINC

BOINC [21] is a platform for Volunteer Computing which is developed by the team that developed the popular project SETI@home [1]. BOINC follows the client-server paradigm in which each project has a server which manages all the communication and work distribution to the clients. Clients on the other side, download all of the application files and software, compute the results and send the results back to the server. BOINC enables the solution of large scale and complex computational problems. It supports diverse applications, including those with large storage or communication requirements. The target platforms or devices are not limited. BOINC-based applications can be executed on various target devices with different software environments, e.g. Windows, Linux, Mac, and mobile devices based on Android or iOS. Applications have to be specially made for running BOINC by using the BOINC API. BOINC is going to be further analyzed in section 4.

V-BOINC

V-BOINC [37] is the virtualized version of BOINC allowing users to avoid the drawbacks of BOINC and take advantage of virtualization. According to this framework virtual machine images are distributed to the clients. These, as well the BOINC core client are managed from the V-BOINC client, a downloadable package encapsulating a modified BOINC client and a GUI with the purpose of communicating with the BOINC and a modified server called V-BOINC. The main difference between the other projects that also implemented virtual machines over BOINC is the dependency handling. To achieve that, a separate Virtual Disk Image (DepDisk) containing the application's dependencies also is required. This .vdi file is created from the developers and downloaded from the clients. If the application is found to have dependencies, V-BOINC client attaches the DepDisk. On the other hand, an empty disk is created to mount the executable. DepDisk that contain the dependencies had to be in a specific location in the volunteer machine. This way, it is going to be forwarded to only one host in order to be executed and not be distributed to many of them because of the different configurations that they may have. In order to create the smallest usable virtual machine image possible, they used the VirtualBox Fixed Disk Image (FDI) type as opposed to the Dynamic Disk Image (DDI). This size had to be as small as possible because that way the transfer time will be also reduced. Afterwards, that image as well as an instantiation script is downloaded from the client. Via this script, BOINC is incorporated in the V-BOINC client. In conclusion V–BOINC performs checkpointing and recovery so as to achieve better performance and reliability in the job processing.

2.3.3 Parallel programming models

Openmp [22] is the most popular parallel programming model which focuses on loop-level parallelism for shared memory systems. Via its Application programming interface (API), a sequential code can easily be parallelized. The directives are added as an indicator to the compiler of the presence of a region to be executed in parallel, along with some instruction on how that region is to be parallelized. Openmp considers about nested tasks but the programmer is responsible to avoid races so as to use barriers for synchronization.

Wool [43] is a library that supports the nested independent task parallel programming model. It aims to obtain low overhead while achieving task-based parallelism. Wool provides synchronization and dependencies among different threads via its API and is designed to test the limits of a low overhead task management.

MPI [24] allows the implementation of parallel programs to distributed machines. MPI applications are composed of a set of processes with separate address spaces that perform computation on their local data and use communication primitives to share data when necessary. Achievable performance and portability are two of the most important advantages of this model due to the optimized libraries that offers compatible with a wide range of machines despite the lack of fault tolerance and the demand of substantial resources such as memory and network.

Cilk [26] is also based on a task based programming model similar to those referred above. It is based on the identification of tasks with the spawn keyword and the sync statement is used to wait for spawned tasks. Cilk does not have a mechanism for recognizing data dependencies between tasks, so additional synchronization points are required. While Cilk only supports parallel tasks, Cilk++ also supports parallel loops.

ClusterSs [27] is a task based programming model for clusters, based on starSs[42].At execution time, the user-selected methods are automatically replaced by runtime calls that create the tasks. The runtime analyses data dependencies between tasks, building a task dependency graph. Tasks that have no dependencies are immediately scheduled to available resources. These resources could be grid, cluster or multiprocessors. ClusterSs uses an Asynchronous Partitioned Global Address Space (APGAS) model over starSs to benefit from the performance portability of it. As a result, an asynchronous execution model is created based on a master-worker architecture, where nodes can either generate tasks or execute them and a data model where data is automatically distributed among the nodes according to the computation needs. Workers can exchange data, bypassing the main node and multiple replicas of mutable or immutable data can coexist.

Chapter 3

The mCluster task based programming model

3.1 Introduction

According to current trends, the rate of cores in a chip is rapidly increasing. This makes the shared memory system sufficient accessible by most developers. Nevertheless, it also means that developers must resort to multi-threaded programming to benefit from this type of system.

Task based programming models allow developers to take advantage of the computing power of each multiprocessor, because it can achieve great parallelism without dealing with many barriers. It gives the opportunity to programmers to express and share calculations in tasks and not manually in threads. Tasks that are dynamically created, allow the program to execute concurrently like having infinite number of processors. It also prevents deadlocks and races between threads. As a result, the system is responsible for minimizing the overheads in order to execute the program effectively, despite the fact that it may has only a few processors.

Most programming models so far, are used in systems where they had a shared memory, which allow indirect communication and synchronization. According to this, distributed memory systems cannot be efficiently used like a cluster of computers. There are architectural differences between shared memory systems and distributed memory systems. In a shared memory multiprocessor, different processors can access the same variables. This makes reference to data stored in memory similar to traditional single-processor programs, but adds the complexity of data. The common processors communicate with the shared address space. Its main features are the ease of programming and the lower communication overhead. The major disadvantage of the system is the synchronization access. This has been enough of a concern that many multiprocessor architects have augmented the basic shared-memory communication model with additional synchronization mechanisms.

3.2 A Task based programming model for distributed systems

Our goal is the design of a task based programming model for distributed memory systems and especially mobile devices. Large scale problems that needs multiprocessing maybe difficult to be executed from a single machine, because of the limitations of the memory space and resources. In order to deal with it, we had to split the major problem to smaller units so as to decrease the computation requirements. Data available in a server database, are downloaded by clients, executed and then sent back to server which will inform the database. The work will be distributed appropriately according to the requirements of each user. This way we will benefit from the large number of the volunteered resources to execute large scale computations without the restriction of the memory space.

In order to achieve this, the original problem should be able to be broken into independent parts so that each processing element can execute its part of the algorithm simultaneously. Parallel programs offer that ability. In this case, we can easily attach the independent tasks to each target device instead of attaching them to another source such as a computer core. Basic requirement for our assumption is the ability of decomposition of our major problem into smaller chunks.

Based on this, our next step is the creation of the appropriate programming model. In that way, parallel applications which computation can be divided in tasks will take the advantage of the multiple resources that target devices offer, favoring applications with large data volumes.

Specifically, according to figure 3.1, a sequential application via specific directives that will be given by the programmer, will generate the appropriate tasks that are going to be executed simultaneously from the available nodes of the distributed system. The application programming interface that will be provided to the programmers could perform the desirable parallelization in the application. This way, the suitable tasks are going to be created. Available resources could be related to computers or clusters but according to the mCluster model, target resources are mostly mobile devices like tablets and smartphones.

The explosive growth of smartphones over the past couple of years has been unprecedented. Almost half the phones now sold to consumers are smartphones. Additionally, these smartphones are becoming increasingly more powerful. If multiple mobile devices like smartphones can be linked together to perform processing, that would become a very useful tool for applications that have large-scale requirements.

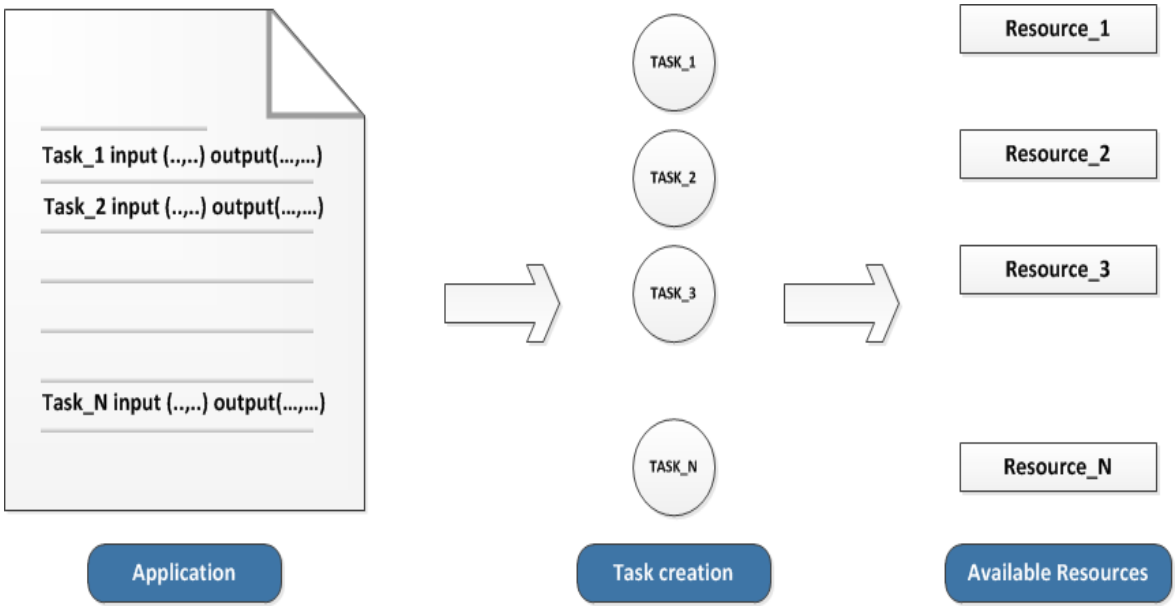


Figure 3.1. Basic idea of the mCluster programming model

3.3 mCluster Application programming interface (API)

Writing a distributed application often requires the programmers to deal with many kinds of complexities. To alleviate these issues, we have designed an API that enables programmability, and portability of user applications. In each program must be clearly specified which piece of code may be executed separately from a node in the distributed system.

A source code with directives is a sequential program annotated with pragmas that identify functions in the code that are candidates to be executed simultaneously. We call those functions tasks. Initially, the user defines what specific inputs and outputs each task has. Each input and output of the task necessarily contains the variable name followed by size. Each annotation must always be followed by the function call that corresponds to it. The inputs of each task in combination with the function implementation are going to be distributed to the clients for processing. The output definition of each task is useful for resolving dependencies.

For example, if we want to annotate the function of matrix addition, we have to initiate the inputs and outputs in the form of the example 1 as follows. Tasks are specified using the **#pragma mCluster** directive followed by its inputs and outputs. After this, the matrix implementation function follows, which is equivalent to the task body. This function will be transformed to the executable that is going to be distributed to the target devices.

Example 1: Pseudocode of the sequential program

```
int * matrix_addition  (int * table1,int length1,int * table2, int* length2)
        {
        ….
        }
Main()
        {
        ..
        #pragma mCluster input(int * table1,10,int *table2,10) output ( int * table3)
        matrix_addition  ( table1,10,table2,10)
        ..
        }
```

In the above example, we define a task that has as inputs two tables, table1 and table 2 with 10 and 20 elements length respectively. It implements the function matrix addition with table 3 as its output. This way, we get all available inputs in order to perform each task.

The length of each table in the annotations is necessary, because we can split a big table into smaller chunks in order to be executed separately by different nodes and not pass the whole table that could cost much computation time to a single node. This makes our program more flexible for further processing suitable for distributed memory systems.

3.4 Dependencies between tasks

Task parallelism is a natural model for expressing dependencies. Task parallelism can have a different execution path per unit of parallel work. According to this assumption, it does not imply uniform workloads because of the amount of work that can be executed simultaneously and the dependencies that might exist between them at any given time are irregular. Tasks allow us to express dependencies on a higher level. Our motivation is to offer a simple programming model, easily programmable by any developer, which with specific annotations will easily express task dependencies.

Any algorithm that is formalized and expressed in tasks in any programming language may contain some kind of dependence between them. Programmers generally pay little attention to the dependence. This may have implications during the program execution. On the other hand, in many cases reducing the number of dependence, leads to direct reductions in a program's running time.

Parallelism is achieved through hints given by the programmer in form of pragmas that

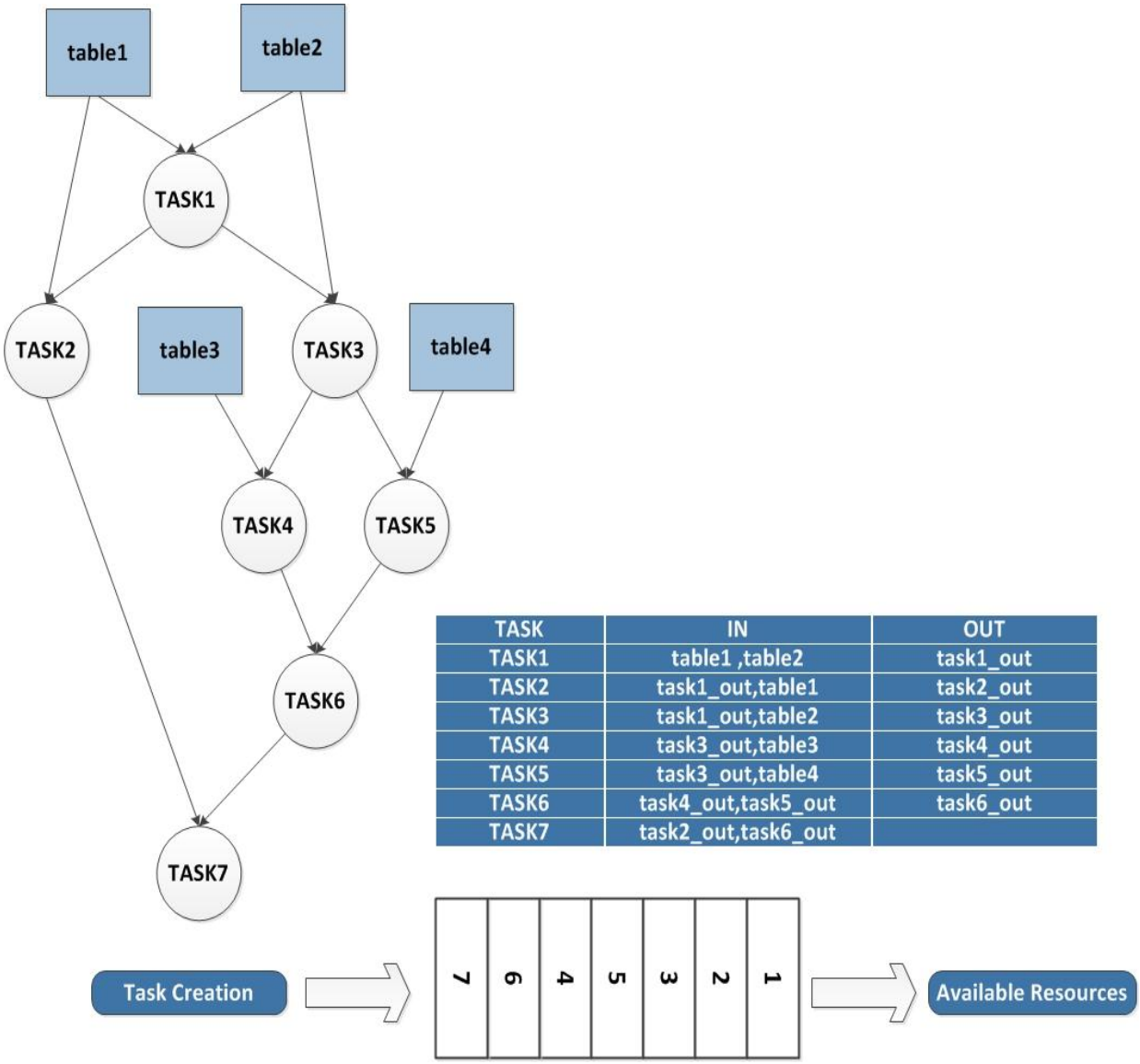
identify the part of the code that operates over a set of parameters. These parts of the code are encapsulated in the form of functions. With these hints, we can detect the task calls and their dependencies. If a task depends on another, it should not start its execution until the other task finishes its execution. So an additional work that should be implemented involves finding the dependencies between tasks. A task-graph is dynamically generated and run in parallel from the available resources.

As shown below in example 2, we have 7 tasks that have to be executed. Task 1 produces as output *task1_out*. On the other hand, task 2 has as input *task1_out*. With this restriction, task 2 cannot start its execution until task 1 is marked as finished, because if it starts before task 1 is completed, the results will be wrong. Similarly, this happens with the other tasks of the example.

Example 2 : Pseudocode of the sequential application representing task dependencies

```
Main()
{
1.  #pragma mCluster input(int * table1,10,int *table2,10) output ( int * task1_out )
2.  matrix mul ( table1,10,table2,10)
3.  #pragma mCluster input(int * task1_out ,int *table1,10) output ( int * task2_out)
4.  matrix add ( task1_out,10,table1,10)
5.  #pragma mCluster input(int * task1_out,10,int *table2,10) output ( int * task3_out)
6.  matrix sub  ( task1_out,10,table2,10)
7.  #pragma mCluster input(int * task3_out,10,int *table3,10) output ( int * task4_out)
8.   matrix sub  ( task3_out,10,table3,10)
9.  #pragma mCluster input(int * task3_out,10,int *table4,10) output ( int * task5_out)
10. matrix add (task3_out,10,table4,10)
11. #pragma mCluster input(int *  task4_out,10,int * task5_out,10) output ( int * task6_out)
12. matrix mul ( task4_out ,10,  task5_out,10)
13. #pragma mCluster input(int * task1_out,10,int *task6_out,10) output ( int * task7_out)
14. matrix add (task1_out,10,task6_out,10)
}
```

In Order to deal with it as shown in figure 3.2, a graph with task dependencies is created, representing dependencies of tasks towards each other. Afterwards a queue of tasks is created depending on each task depth in the graph. Tasks are stored in a queue, in the correct order so as when the clients are going to request work, the task they select will have no previous dependencies.



TASK	IN	OUT
TASK1	table1 ,table2	task1_out
TASK2	task1_out,table1	task2_out
TASK3	task1_out,table2	task3_out
TASK4	task3_out,table3	task4_out
TASK5	task3_out,table4	task5_out
TASK6	task4_out,task5_out	task6_out
TASK7	task2_out,task6_out	

Figure 3.2: A set of tasks (circles) and their dependencies (arrows). The arrows indicate the direction of the dependency, i.e. an arrow from task 1 to task 2 indicates that task 2 depends on task 1. Task 1 has no unsatisfied dependencies and can therefore be executed. Once task 3 is completed, tasks 4 and 5 become available, and task 6 only becomes available once both tasks 4 and 5 have been completed. The right table shows all graph's dependencies.

3.5 Data Hazards in task execution

Out-of-order execution [15] is well-known in the domain of computer architecture for a long time, originating in early work by Tomasulo /scoreboard [35]. Its basic idea is to execute a sequential instruction stream of a usual architecture in data-flow order, thereby establishing more parallelism and better load balancing of available functional units. To this end, the data dependencies between tasks

are analyzed and tracked by a couple of data structures. Data hazards are an implicit problem in tasks out-of-order execution.

Traditional out-of-order pipelines provide programmers with a sequential interface, yet internally execute instructions in parallel, based on dynamic analysis of data dependencies. According to this purpose, out-of-order execution may cause problems in the result each task generates.

In example 3, TASK 3 has as input table3 which outcomes from TASK 1 and TASK 2. The right value must be obtained from TASK 2.

Example 3: Pseudocode representing data hazards

```
TASK1   1.#pragma mCluster input(int * table1,10,int *table2,20) output ( int * table3)
        2.  matrix mul ( table1,10,table2,20)
TASK2   3.#pragma mCluster input(int * table4,10,int *table2,20) output ( int * table3)
        4.  matrix add ( table4,10,table2,20)
TASK3   5.#pragma mCluster input(int * table3,10,int *table2,20) output ( int * table4)
        6.  matrix mul ( table3,10,table2,20)
```

Task renaming in mCluster

```
TASK1   1.  matrix mul ( table1,10,table2,20)
TASK2   2.  matrix mul ( table4,10,table2,20)
TASK3   3.  matrix mul ( table3_1,10,table2,20)
```

If TASK 1 finishes its execution before TASK 2, the value that is going to be attached to TASK 3 is the outcome of TASK 1, which is not the right one. Dealing with this, if more than one tasks produces same result, it must be renamed internally. In our example, we rename the output of TASK 2 to table3_1 and then set this value as the input of TASK 3. This way, Task 3 will not execute until TASK 2 finishes resolving any data dependencies that may occur.

3.6 Limitations

If not planned properly, a distributed system can decrease the overall reliability of computations. Troubleshooting and diagnosing problems in a distributed system can also become more difficult, because the analysis may require connecting to remote nodes or inspecting communication between nodes.

Many types of computation are not well suited for distributed environments, typically owing to the amount of network communication or synchronization that would be required between nodes. If bandwidth, latency, or communication requirements are too significant, then the benefits of distributed computing may be negated and the performance may be worse than a non-distributed environment.

Chapter 4

mCluster implementation using the BOINC infrastructure

4.1 Introduction

As mentioned, many distributed-computing infrastructures are nowadays available. They are focused on efficient job distributing using a great variety of scheduling policies, different communication protocols and based on different architectures. Nevertheless, BOINC and Hadoop are the most popular and widely used programming frameworks. Each of them provides different mechanisms for job transmission, having their advantages and drawbacks. In this section we are going to choose the most efficient for the mCluster implementation.

4.2 Hadoop limitations

As presented in chapter 2 the basic idea of mCluster was the implementation for a task based programming model for distributed systems. Initially, the infrastructure studied for the implementation of mCluster was the Hadoop. The most important difference between volunteer computing frameworks and other computing frameworks such as Apache Hadoop is the vast diversity between resources available to the system. This diversity includes not only the speed, number of processors, memory and disk space at each resource, but also the operating system and hardware installed on the machine. Furthermore, different resources have different levels of availability and reliability, which can change unpredictably over time.

Hadoop seemed to be the ideal framework for mCluster because of the fault tolerance mechanism and its ability to run applications on systems with thousands of nodes involving thousands of terabytes. Despite the advantages referred above, Hadoop has also many drawbacks which are not consistent with mCluster's requirements.

These drawbacks are:

- The MapReduce engine: There are certain cases which MapReduce is not a suitable choice. Initially, it is not always easy to implement a program according to this programming model. The program has to be divided in the three phases so if the problem cannot be structured in that exact way it will not be compatible with Hadoop.

- Not Fit for Small Data: Due to its high capacity design, the Hadoop Distributed File System (HDFS) lacks the ability to efficiently support the random reading of small files. Moreover since Hadoop is architected to accommodate very large data files by splitting the file into small chunks over many worker nodes, it does not perform well if many small files are stored in the distributed file system. If only one or two nodes are needed for the file size, there is a large overhead of managing the distribution.

Dealing with mobile devices, Hadoop is not suitable because of the disadvantages referred to:

- Resource limitations: Hadoop was built in a way that data have to be always in client storage in order to be directly executed and not stored in a central server. It is based on the assumption that the available system executes attempts to process a great amount of data which is scattered throughout the nodes without the presence of a central node. This restriction, as well the fact that we had to deal with devices that may not have the desirable resources, like free memory space and computational power certainly enhances our disbelief regarding Hadoop's suitability. If a mobile device is attached as client, its storage is automatically going to be used for data processing. The way that the Hadoop File System is structured, does not require information about each client characteristics. It distributes the data without the knowledge of the available space which is suitable for computers with lots of available space and not for mobile devices with restricted memory.

- Network connectivity: Hadoop is structured in order to be mostly used in a local network. Each device which will be connected as a client, according to Hadoop requirements, has to be represented with a static ip address. Before the HDFS attempts to distribute its data, a list of the available clients is created, represented with their ip address. Mobile devices usually connect to the internet via Wi-Fi or mobile data. Every time they are reconnected, these values are changed so they don't belong to the previous lists and will be shown as not

available, in contrast with our requirements. Summarizing, Hadoop architecture is based to deal with static clients and not clients that dynamically attach to the system, in consideration of the restrictions that referred above.

4.3 Berkeley Open Infrastructure for Network Computing (BOINC)

In our attempt to deal with the drawback of the Hadoop framework, we are going to study BOINC, which is currently considered as one of the most popular volunteer computing platforms. It is designed to support applications that have large computation requirements, storage requirements or both. It is a framework for solving large scale computational problems by splitting them into smaller units which will be executed by volunteer computers. It consists of many independent projects in which each client can take part by downloading workunits and send the result back to a central server. In this way, the computational effort is distributed into many clients having their own computational resources. Each computer works on its own workunits independently from each other and sends back its result to a project server.

There are quite a few BOINC-based projects in the world. Installing, configuring and maintaining a BOINC based project however is a highly sophisticated task. Scientists and developers need a great deal of experience regarding the underlying communication and operating system technologies, even if only a handful of BOINC related functions are actually needed for most applications. This limits the application of BOINC in scientific computing although there is an ever growing need for computational power in this field. In this thesis, we present a new approach for model-based development of BOINC projects based on the specification of a high level abstraction language as well as a suitable development environment.

4.3.1 BOINC in contrast grid computing

Volunteer computing and Grid computing share the goal of the utilizing existing computing resources. Both are forms of distributed computing which try to fully utilize existing resources. However, as shown in figure 4.1, they differ in several essential respects. Grid computing involves organizationally-owned resources such as supercomputers and clusters. These resources are very reliable because they are usually managed from universities and other research institutions using high-bandwidth network links in order to minimize the probability of system failure. On the other hand, volunteer computing is based on individual participants owning computers, tablets, smartphones

connected to the internet most of the times using low-bandwidth networks which sometimes may not even be free. Furthermore, volunteer computing is based on the "pull" model in which clients periodically request for job from a central server despite the "push" model used in grid computing where the request for a given transaction is initiated by the publisher or central server.

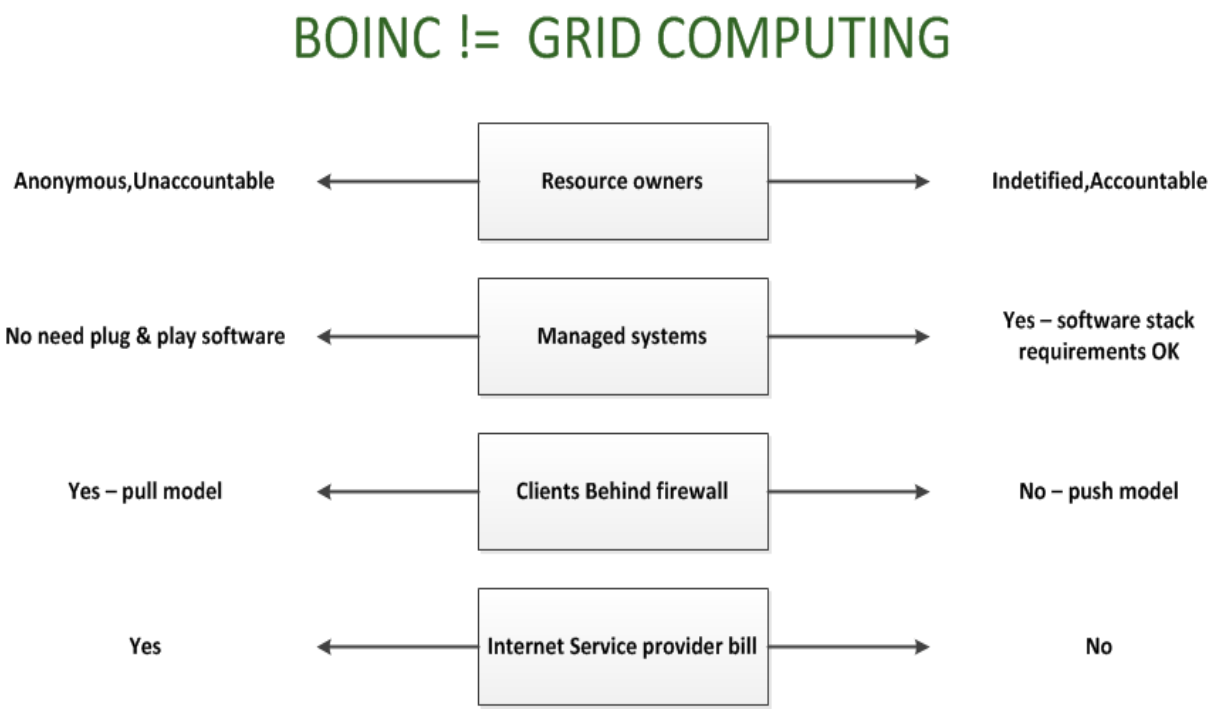


Figure 4.1 : Differences between BOINC and grid computing systems

4.3.2 Goals of BOINC

- **Reduce the barriers of entry to public-resource computing**: BOINC allows scientists to create a public resource project, which can be run from a single computer running open source software.

- **Share resources among autonomous projects**: Each BOINC project is independent from the others, has its own servers and can be run with no restrictions. In this way any participant can join many projects and perform tasks from more than one. That means that any computer, when the projects are temporarily down can use its computational resources to the others, in order to improve the resource utilization.

- **Support diverse applications**: BOINC support multi language applications (C,C++,FORTRAN) and offers various data distributions mechanisms.

- **Reward participants**: In order to attract participants BOINC offers credits to them who offer their computations resources and offers a great variety of graphical representation.

4.3.3 BOINC Features

1) **Redundant computing:** BOINC supports redundant computing, a mechanism for recognizing errors at client's workunit execution. For each workunit can be specified that N results should be created. Once M of these N have been distributed and completed, BOINC server daemons are called (transitioner, validator, assimilator, file deleter) to compare the results and possibly select a canonical result. If some of the results fail, BOINC creates new results for the workunit, and continues this process until either a maximum result count or a timeout limit is reached. In figure 4.2, a redundant computing example is presented.

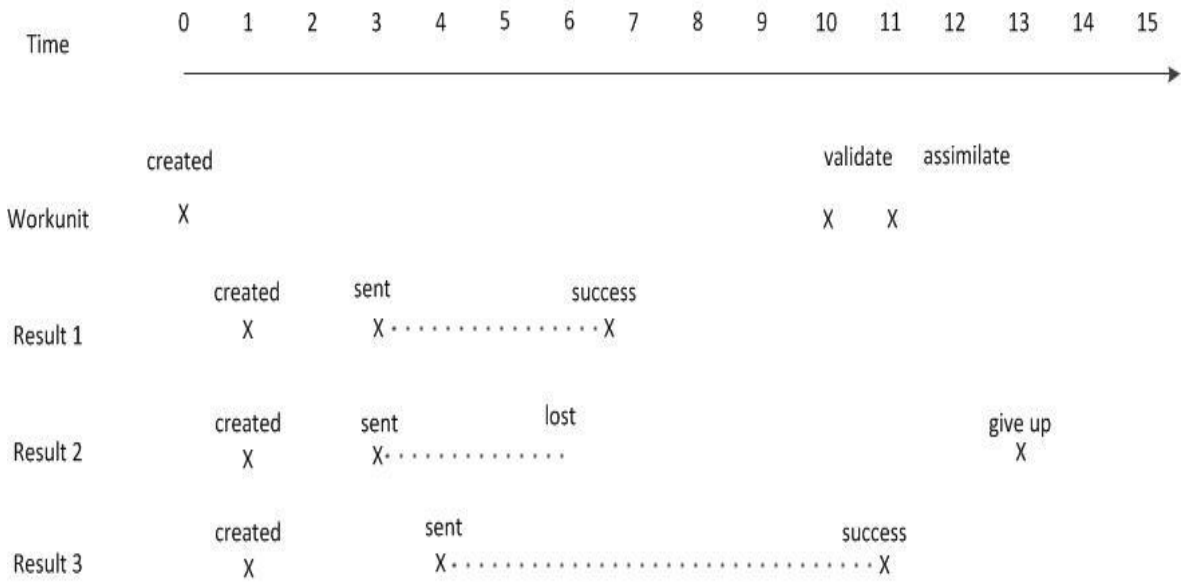


Figure 4.2: Redundant computing example. In this example, result 2 is lost (i.e., there's no reply to the BOINC scheduler). When result 3 arrives a consensus is found and the work unit is assimilated. At timestep 13 the scheduler 'gives up' on result 2 (this allows it to delete the canonical result's output files, which are needed to validate late-arriving results).

2) **Homogeneous redundancy:** Due to differences among the participants (different operating system different architecture, compiler), results for a given workunit might depend on them, so they may be different. To deal with this and in order to handle divergent applications (applications in which small numerical differences lead to unpredictably large differences in the final output) BOINC provides this feature. In that way hosts are divided into numerically equivalent classes. The BOINC scheduler will send results for a given workunit only to hosts in the same class. There is an option that hosts are separated according to the CPU type or the operating system. It can be manually enabled or disabled in BOINC configuration file.

3) **Security:** BOINC offers a mechanism to prevent server and client from various attacks like theft of project files, result falsification, malicious executable distribution, intentional abuse of participants' hosts by projects and other malware-type attacks.

4) **Fault tolerance and multiple servers:** Each BOINC project may have hundreds of thousands of participants. If all of them are trying to connect in the same time to a specific server will lead to its overload. BOINC has a number of mechanisms to prevent this. All client/server communication uses exponential back off in case of failure. According to this, if a client server communication fails for some reason the client will not try to connect to the project immediately. Instead it will retry to connect with some delay based to the number and the cause of failure.

5) **Local task:** Regarding the maximization of the resource usage when each workunit is sent to a client, a deadline is used for the execution and when it overcomes the workunit is send to another host. In order to minimize each project running time, the BOINC core client decides locally when to get work, from what project and what tasks to attach to specific clients.

6) **System monitoring tools:** BOINC offers a web based system, where any host can obtain information about the daemons that are running on each projects. Additional they can be informed about the tasks and their deadlines, the available tasks for downloading, the complete tasks and the numbers of applications that belongs to the projects or application's specific database tables.

7) **Participant preferences:** Computer owners generally participate in distributed computing projects if only they incur no significant convenience, cost, or risk by doing so.

8) **Open and extensible architecture:** Porting application to BOINC infrastructure is not easy but may not need much configuration. BOINC offers a variety of application programming interfaces APIs so programmers can implement any application compatible with BOINC.

4.3.4 BOINC architecture

BOINC [32] is based on client-server architecture. The basic idea is that clients are continuously requesting for services and the central server provides them. As shown in figure 4.3 the server side consists of three individual servers, the task server, the data server and the web interface server. The requests are made by clients using remote procedure calls (RPCs). Clients are running each BOINC application executable, linked with a specific runtime system which function incudes the process control for the workunit.

A client communicates with the project's task server. Client gets a set of instructions from the project's task server. The instructions depend on the computers that are going to be used. For example, the server won't assign them work that requires more than their available space. The instructions may include multiple pieces of work. Projects can support several applications and the server may send work to the computer from any of them. The request comes in the form of an XML document which describes the host's hardware and other characteristics and a request for a certain amount (expressed in terms of CPU time) of additional work. The reply message includes a list of new jobs with each one described by an XML element that lists the application, input and output files, including a set of data servers from which each file can be downloaded). The client downloads the executable and input files from the project's data server. If the project releases new versions of its applications, the executable files are downloaded automatically by the client to the computer, which runs the application programs, producing output files and then uploads the output files to the data server. Later (up to several days later, depending on user preferences) the client reports the completed results to the task server, and gets instructions for more work.

Furthermore, a web interface server is responsible for giving hosts the ability for plenty of operations via a web interface. Each project has a server status page where each host can gain information about the daemons that are running, the number of tasks that have already executed, the number of results that are not valid, as well to have access to a list of project applications with a summary of job throughput for each. Moreover, this page contains information related to hosts' personal profile, like the credit that they have been awarded. In addition, gives the ability to them for

editing their profile preferences. Each project has also a project management web page, not visible to the public, that lets project administrators to handle each project's properties. They can also create and edit application and application versions, cancel workunits, view recent results and analyze them in order to have access to the project database to perform the appropriate operations. Finally, as above referred the data server is used for downloading input files and their executable and uploading output files. The task server and the client side are going to be analyzed in detail in next sections.

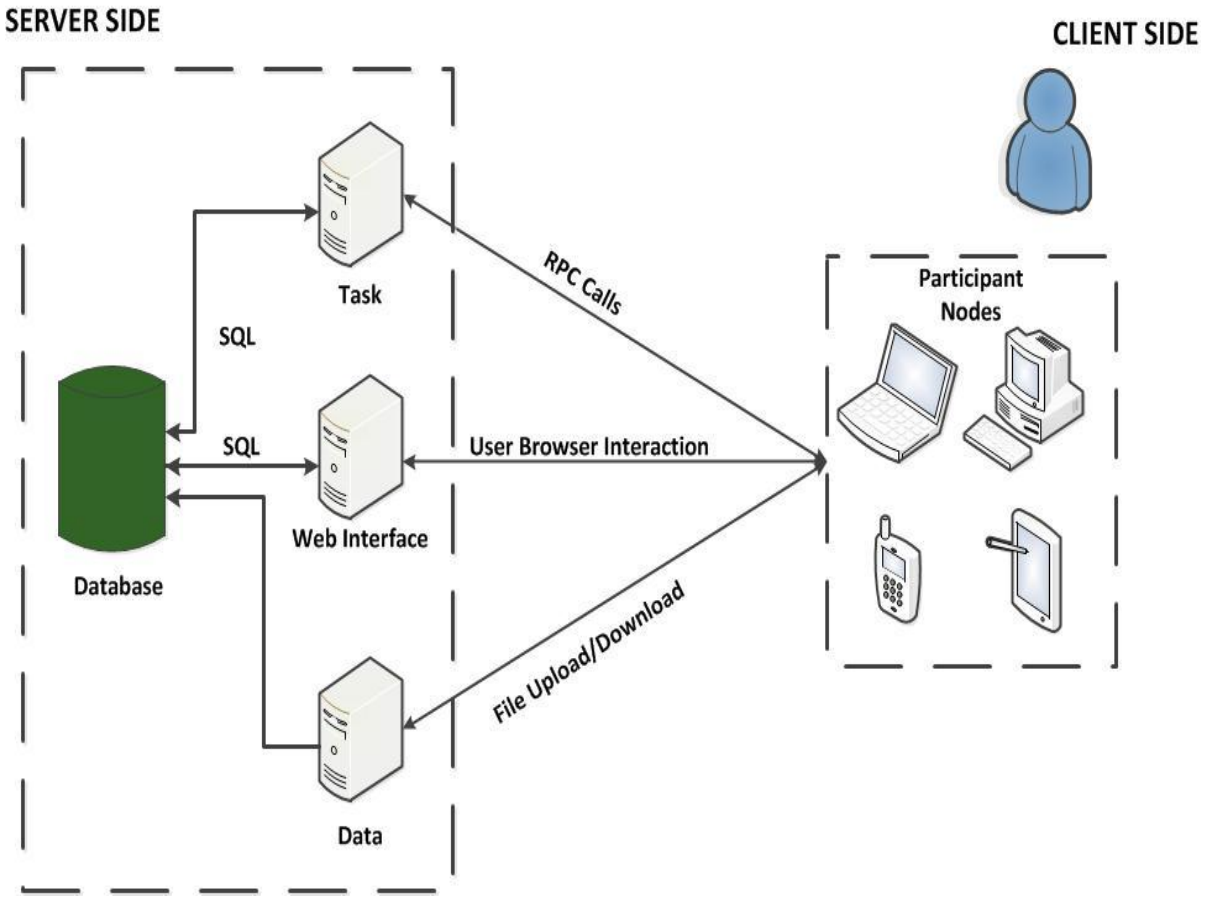


Figure 4.3: BOINC client-server architecture

4.3.5 BOINC client

BOINC [35] client shown in figure 4.4 is structured into a number of separate applications. These intercommunicate using the BOINC remote procedure call (RPC) mechanism. These applications are:

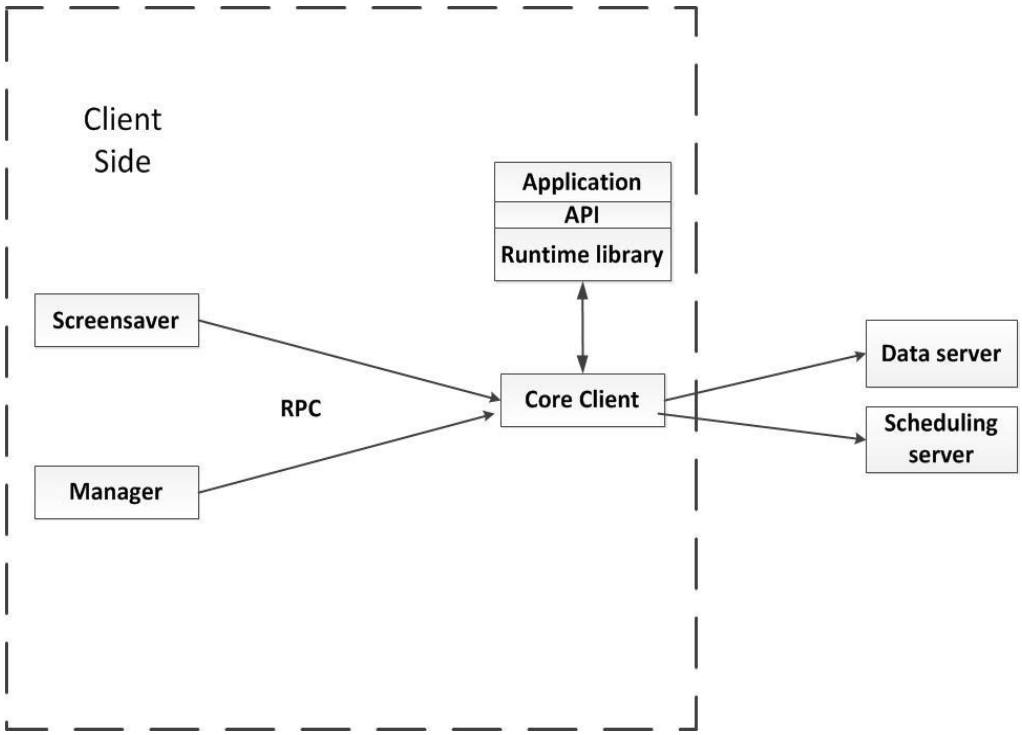


Figure 4.4: Client components

Core client: The process that actually does the main work. It makes the network communication with the servers, executes and monitors applications and enforces preferences. It consists of several depended FSMs responsible for network transfers. In addition, it is responsible for managing the way that one or more applications are going to be executed. It uses round robin mechanism in the way that the client executes them in order to dispatch the available resources evenly. Moreover, each task has its own requirements like memory or disk usage. If these requirements are greater than those that the client has, the core client aborts that task.

Manager: Provides a graphical interface for the users to able to perform multiple operations related to the project. It offers a variety of options to users like detaching/attaching to a project, updating a project to get new instructions, suspending computation for a project. Additionally, it informs users about the percentage of execution of each task and keeps statistics about the disk usage that each application has allocated.

Screensaver: BOINC display full screen graphics depending on application.

API: Via client's API, core client is informed about useable functionalities about the program execution.

Applications: Large-scale applications relating physics, mathematics and other fields of science.

Runtime library: BOINC runtime system is based on shared memory message passing. The bidirectional communication between core client and application is implemented by the core client which creates a shared memory in which messages are transmitted telling application what to do, like suspend and abort.

The client embodies two related scheduling policies the task scheduling policy and the work-fetch policy.
The goals of these policies are:

1) Tasks should be completed and reported by their deadline (results reported after their deadline may not have any value to the project and may not be granted credit).
2) All processors should be kept busy.
3) At any given point, the computer should have enough work so that its processors will be busy for at least (min buffer) days and not much more than (max buffer) days.
4) Project resource shares should be honored over the long term.
5) If a computer is attached to multiple projects, execution should rotate among projects on a frequent basis to improve the volunteer experience.

4.3.6 BOINC task server

BOINC task server [25] consists of several individual programs that are running separately in order to achieve efficient job scheduling. These programs, called daemons, are presented in figure 4.5 and are analyzed in this section. BOINC task server is the most important part of the server side because it is in charge of most processes that contribute to the BOINC framework. Processes related to:

Creation and distribution of workunits. Workunit describes how the experiment must run by the clients (the name of the binary, the input/output files and the command line arguments). Among workunits, MD5 files are created so as the client can recognize if the workunit has the right format.
Validation of the received results. Clients' results that are sent on the server database have to be validated first before they are stored. Different architectures among clients, which are rather common due to the vast diversity of devices, usually lead to different results for the same workunit. The server has to validate these results and store the valid ones in the database.

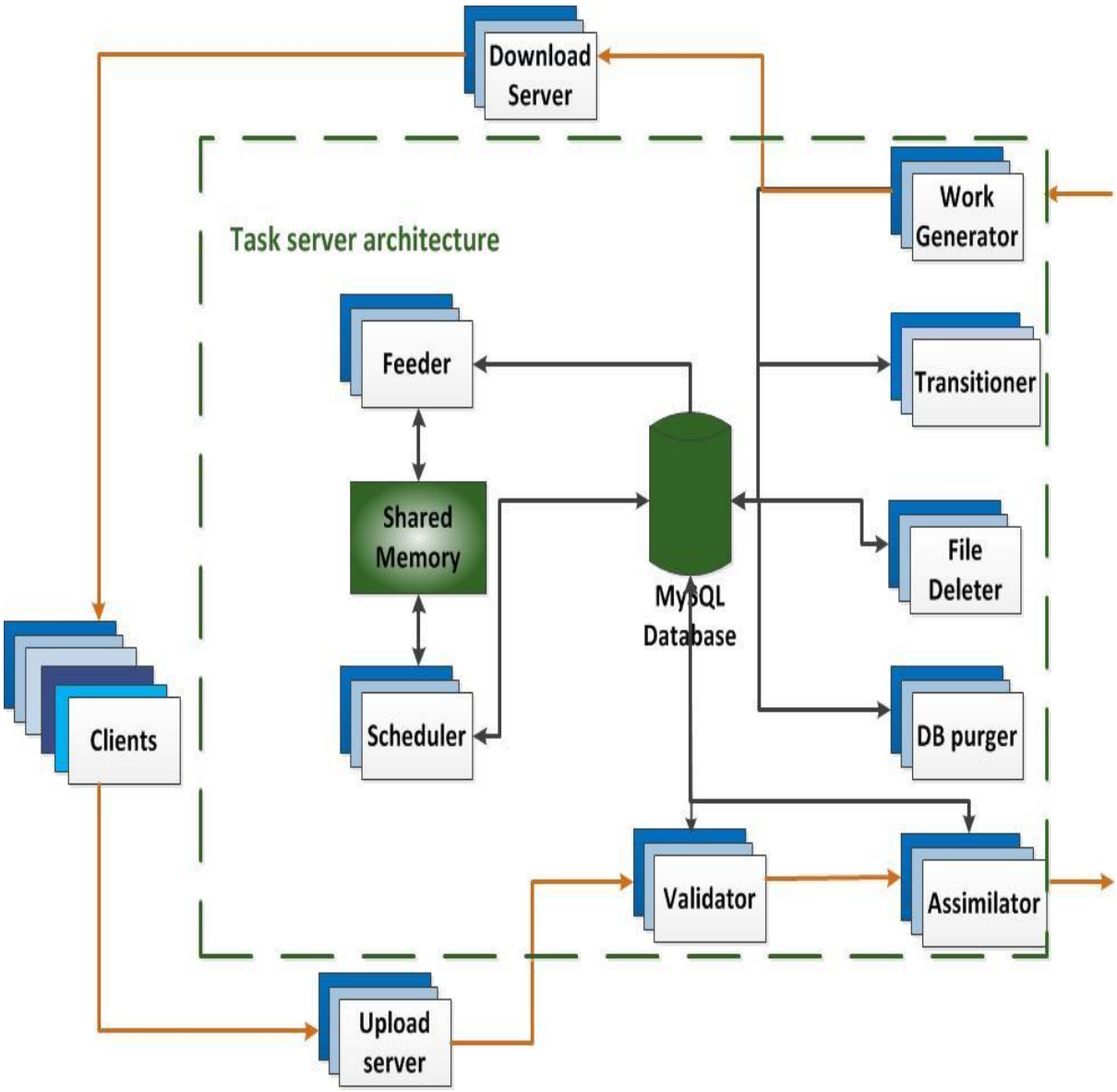


Figure 4.5: Task server components

Mysql database:

The data back-end of a BOINC project is a MySQL database. This database stores all information related to BOINC. Also the server may have more databases, where results are stored or have the replicas of the workunits if they are created. This database is a collection of tables and indexes that hold information about workunit's, results, applications and all information about specific projects. The database schema for BOINC is very complicated and the table above offers a high-level view of the table structure. Database table contents are shown in table 4.1.

Table name	Description
app	This table stores information about Applications
app_version	Contains different versions of the project's applications and incudes a URL for executables and the MD5 checksum
result	Contains iformations about the state of the workunits (whether the result has been dispatched).
host	Informations about the clients that are connected to the project
workunit	Includes counts of the number of results linked to this workunit, and the numbers that have been sent,succeded or failed
Platform	Compilation targets of the core client and/or applications
user	Descibes the users name, password, email address and Account key

Table 4.1 BOINC database schema

Scheduler: Coordinates the work that is issued to make the best use of the computers available to process the work that is ready to use. Each host that is requesting for work is defined by some characteristics like number of CPU and CPU frequency. A client request also includes information about the completed instances except the work request. According to these, the scheduler dispatches the available workunits so as to achieve the most efficient way that will lead to better execution time. In order to handle a request scheduler performs multiple database operations such as reading and updating the records for each user account instances and workunits.

Feeder: Fills up the ready to send queue with work units ready to be sent. Feeder streamlines the scheduler's database access. For better performance on the workunits access maintains a shared memory segments containing workunit relevant information:

1) Static database tables such as scientific applications, platforms, and application versions.

2) A fixed-size cache of unsent instance/job pairs. The Scheduler finds instances that can be sent to a particular client by scanning this memory segment.

Its runtime can be configured during the execution time. Each workunit can be marked to be sent in priority or random order depending on each project requirements.

Transitioner: Handles the state transitions of work units and results. It is responsible for giving each workunit a specific state e.g. a client has finished its workunit execution so the state that will be given is ready for report. If a workunit result is not valid will be marked as aborted, if cannot be downloaded its state will be marked as permanent error. Depending on its state, it will be also marked for validation or assimilation.

File deleter: Removes the workunit's data files and result data files that are no longer needed. This daemon helps to keep disks as clean as possible.

Validator: A back-end program that performs validation and credit granting. Validation consists of two parts. At first it performs a syntax check by verifying which output files are present on the server having the correct format. After that, a replication check follows. If the job is replicated, these replicas are compared, if a strict majority are found to be equivalent, those replicas are masked as valid and the rest as invalid.

BOINC provides three standard validators:

1) **Sample trivial validator** Marks a job as valid if its output files are present. It is usually used if all hosts are trusted.
2) **Sample substr validator** Marks a job as valid if its error output includes a string specified by the "stderr_string "command-line arg. If a specific flag is enabled, the logic is inverted: a job is valid if its error does not include the string.
3) **Sample bitwise validator** Output files are equivalent if they agree byte for byte through the comparison of MD5. This can be used if an application generates exactly matching results (either because it does no floating-point arithmetic, or because it uses homogeneous redundancy). Apart from those, BOINC offers the ability to project developer to implement his own validator. More specifically, in order to create a validator only three functions provided by BOINC had to be supplied. In most of scientific projects, building a validator could be very useful because every time each scientific application has certain requirements.

Assimilator: Completed jobs that are handled by programs called assimilators. These are generally application-specific: If the workunit has a canonical result, the output files from the BOINC upload directory are copied to a permanent location, or the output files might be parsed and placed in the project master science database for later analysis. On the other hand if an error occurs in the workunit, a message is written to a log or an e-mail is sent to the project administrator. It is performed once for each workunit. Assimilated workunits are stored in a specific folder, while the others that are written in an error file.

Work generator: Creates workunits that are going to be issued to the participants. Each application has its own work generator. It is responsible for creating the input files with the right data. It is the most important part of the server daemons, because the job that is going to be executed is exported from it. The work generator sleeps if the number of unsent instances exceeds a threshold, limiting the amount of disk storage needed for input files. Many projects have an essentially infinite supply of work. This can be handled by a 'flow-controlled work generator' that tries to maintain a constant number of unsent jobs (typically a few hundred or thousand). It does this by periodically querying the BOINC database to find the number of unsent jobs.

DB purger: This daemon is responsible for removing jobs and database entries that are old and no longer needed which bounds the size of this table in order to manage better performance in database operations.

4.3.7 BOINC suitable applications

BOINC is designed in order to be suitable for applications that require large computational power and heavy disc usage. A project may gain access to Teraflops of computation power and Terabytes of disk usage. Because of BOINC restrictions applications should have some specific properties in order to be used in an efficient way.

Application independent task parallelism: Each application's task must be independent because BOINC does not provide a mechanism to deal with this. This is the most important disadvantage of BOINC which are going talk about in the next section.

Low data/compute ratio: Data between server and client are transferred through internet connections, which may be expensive and sometimes slow. In that case, an application requires more than one gigabyte per day of CPU time, so it would be more efficient to use in-house cluster computing rather than BOINC [21].

Fault tolerance: BOINC provides mechanism to deal with this (redundant computing) but the error probability may be not always be equal to zero.

4.3.8 Application porting to BOINC

In order to port an application to BOINC, it has to be split and written in two parts. The server side which is responsible for creating the work units and the client side, which consist of the executable that is going to be processed from the client. The process to transform applications that are running on single machine, to be compatible with BOINC is very difficult and not straight forward.

Porting application to BOINC may become a very difficult process. BOINC provides various APIs for different purposes. As shown in the figure 4.6 the basic APIs that are mostly used for this reason in the BOINC API and the DC –API.

BOINC API is the application programming interface provided with BOINC. Includes a diagnostics API used to enable data collection for debugging and a graphics API for allowing the application to render graphics to the user's screen. It consists of a set of C and C++ functions in order

to implement 4 separate applications (work generator, assimilator, validator, worker) which makes it a very complex porting process.

In contrast, DC-API (Distributed Computing Application Programming Interface) [44] allows easy implementation and deployment of distributed applications on Grid environments. It's an API that supports master–worker programming model, designed to be easy to use and to hide the details of BOINC.

As shown in figure 4.6, in order to implement an application using the BOINC API source code had to be written for the client application, the work generator, the validator and the assimilator. Each of them consists of specific functions and libraries provided by BOINC API in combination with the applications source code. On the other hand, via DC-API must to be written only the master application and the validator. The master application includes simplified functions which implement WU generator's and Assimilator's functionalities.

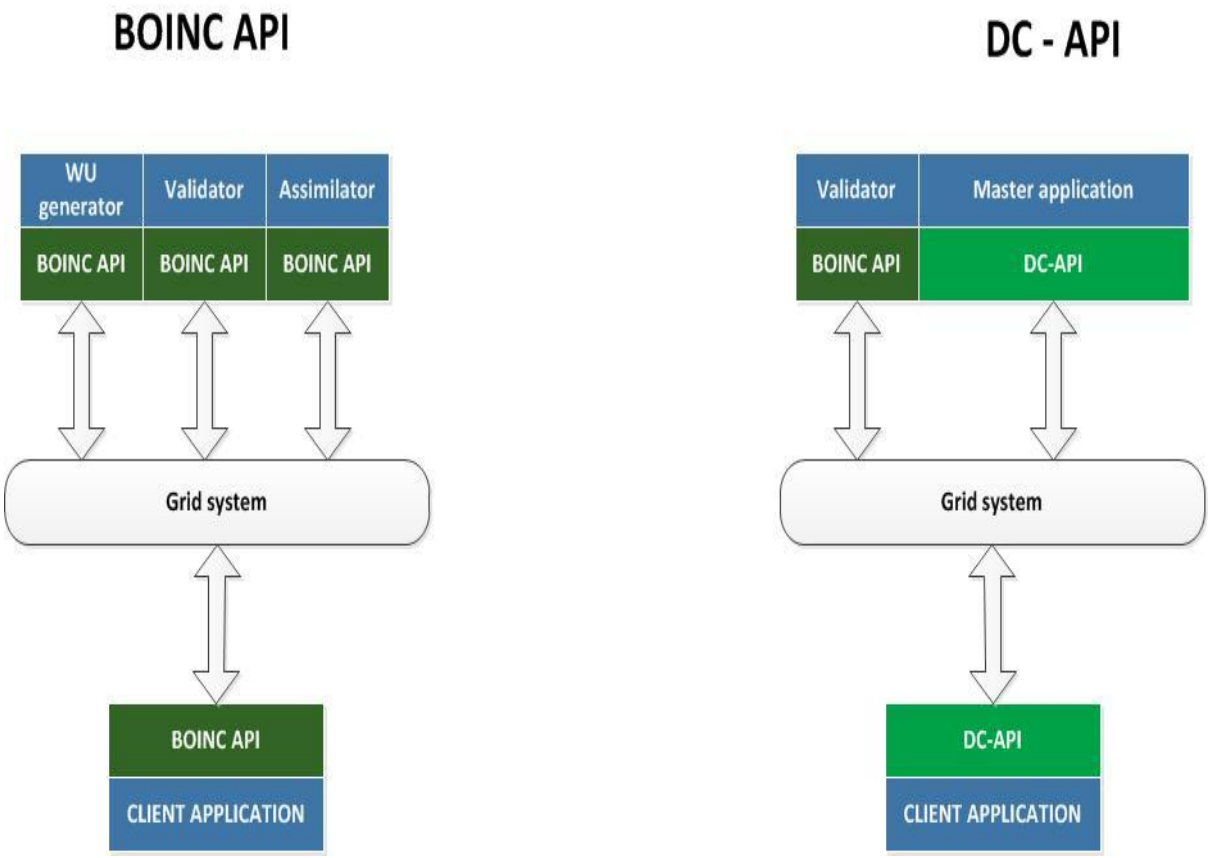


Figure 4.6: BOINC application programing interfaces

4.4 Aim development of mCluster

BOINC may evolve into a very useful framework, due to the fact that it is developed in a way that it solely consumes each client's free resources. Some of its advantages like the homogeneous redundancy, the fault tolerance mechanism and the security that provides as analyzed in section 4.3.3 make it as a very efficient programming model for distributed computing. On the other hand, the lack of resolving dependencies between tasks constitutes an important disadvantage, which makes a large scale of applications incompatible with the framework. The main goal of mCluster is to improve BOINC features by designing an additional functionality related with this restriction. Furthermore porting application to BOINC is a very difficult process and requires building many programs not familiar to most programmers because of the difficult-to-understand BOINC application programming interface.

4.5 Source to source translator

The first step is the implementation of the source to source translator, a program which will gain the original sequential application and generate code compatible with the runtime system of BOINC. The main goal is to create all the available programs BOINC requires for job transition and execution according to each application. In order to achieve this, application must be able to be broken in units in order to be distributed to the available clients. Programmers according to mCluster application interface (mCluster API) must define these units in order to be in the form that source to source translator supports.

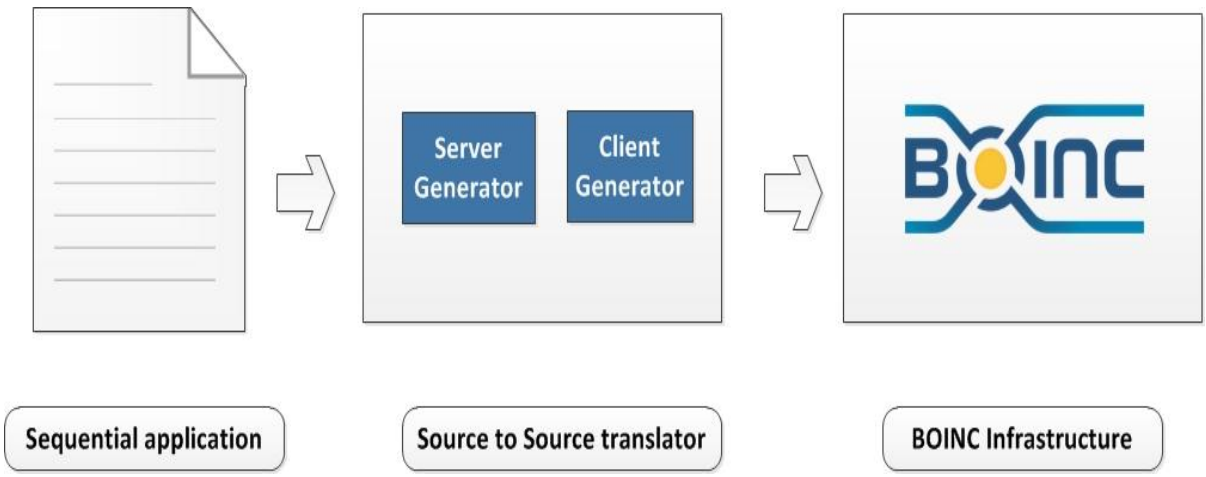


Figure 4.7: Representation of source to source translator environment. Source code with directives via source to source translator creates application ready to be executed by each distributed node's runtime system.

As shown in figure 4.7, having a program written in C or C++ programming language, our source to source translator will read the relevant directives and the arguments which inserted by the user and convert them into function calls that are going to be implemented by BOINC.

Finally, if a program executes in a shared memory system, the data will be taken from local memory. Now in distributed computing, data must be stored in the appropriate file location where the application executable is also placed, in order to be accessible from the available clients to download and then start its execution. Source to source translator is responsible to create and place these temporary files in the BOINC file system.

4.6 Source to source translator architecture

The main goal of source to source translator is to fully transform a sequential application to ready to be executed tasks by any user via the BOINC framework. As mentioned in section 4.3.8 in order to do this, source code had to be written for both client and server side. Source to source translator consists of two parts. The server generator and the client generator. BOINC daemons mode of operation was analyzed in section 4.3.6 and in section 4.3.8 the two programming interfaces for building BOINC daemons was also presented.

Combining the requirements extracted from these sections and dealing with mobile devices, it is concluded that the suitable API for applications compatible with the mCluster framework is the BOINC API. This mainly happens due to the incompatibility of DC-API libraries with the android devices. Dealing with BOINC API presupposes the implementation of three BOINC daemons (work generator, assimilator and validator) as well as the implementation of the client application. Finally, in order to handle dependencies between tasks that are not yet supported from BOINC, the task coordinator daemon is also created, which creates the dependency graph and the task Scheduler which inserts/updates elements in the queue order, generated from the graph.

4.6.1 Server Generator

The first and most important part of the source to source translator is the creation of the daemons that are required for the job processing. As shown in figure 4.8, the assimilator, the validator and work generator are created to be compatible with the BOINC API. Task coordinator is the new daemon responsible for task dependencies.

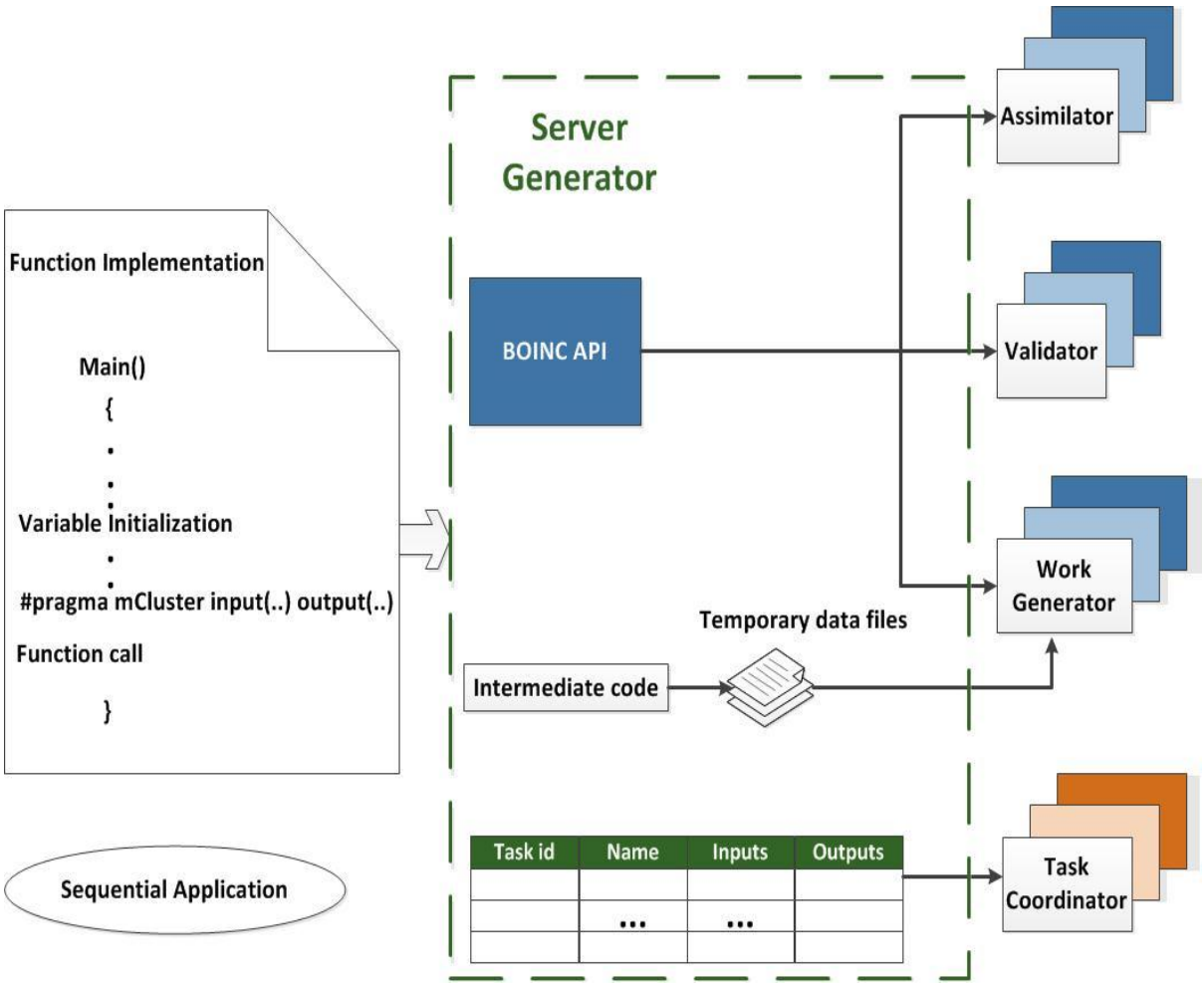


Figure 4.8: Components of the server generator

Assimilator and validator as analyzed in section 4.3.6 are daemons responsible for handling the result of a workunit. Any BOINC project cannot be started until these daemons start running. Task coordinator is the new daemon that is created from the server generator in order to handle data dependencies.

At first, server generator creates a table with each task's characteristics. The task id, name, inputs and outputs are stored in this table. In accordance with the inputs and the outputs of each task, the task coordinator creates the appropriate graph where dependencies between tasks are represented. This graph is going to be transformed from the job scheduler to a FIF0 of tasks. The tasks that have no dependencies are inserted with highest priority in a manner of being executed first and the others are going to be inserted based on their depth in the graph.

The work generator, in order to create a workunit, has to be initialized with the appropriate values. These values are usually stored in the BOINC file system. This assignment belongs to the server generator which reads each tasks inputs and parses them to temporary data files. After that

work generator's source code must be edited in consonance with the task requirements. BOINC API is the programming interface which work generator is based on. As show in example 5.1, the code is based on a specific template which allows easy modification.

Example5.1 Work generator pseudo code

```
main ()
{
    1. Parse BOINC configuration file to get project characteristics
    2. Parse the template files contain information about the workunits
    3. Make a unique name for the job and its input file
    4. Create workunits
    5. Put it in the wright place in the download directory hierarchy
    6. Fill the job parameters in order to be stored in database
    7. Register the job with BOINC
    8. Wait for the jobs to be transitioned
    9. Terminate BOINC
}
```

In the section of code that the workunits are going to be created, as shown in the example above, server generator replaces it with the appropriate values in accordance with the source application. Data values are loaded from the temporary files and assigned to the workunits. Furthermore it has to create the template files according each application requirements that are going to be loaded from the work generator.

In order for an application to run, its inputs and outputs must be described via xml template files. These files must be stored in a specific directory in BOINC project hierarchy. They are two template files for each application. The first one is responsible for the input files (workunits) and the other one for the output files (results). Input template file usually consists of three variables that had to be initialized. These values are referred to the number of input files each workunit has, the name of the workunits and flags relating to its requirements. These variables are initialized from server generator according to the directives that have been given from the programmer in the application source code.

In the same way, output template variables that had to be initialized are referred to the maximum number of bytes that each result must have, the name of the workunit result and flags related to validation and deletion of the results. Input and output templates must be placed in a specific directory in the BOINC project hierarchy. When a job is created, the name of its output template file is stored in the database. The file is read when instances of the job are created, which may happen days or weeks later. Thus, editing an output template file can affect existing jobs so a new template file had to be created.

4.6.2 Client generator

The client generator is responsible for creating a ready to be executed application by any BOINC client. According to the annotations that are given by the programmers, client generator transforms the appropriate functions in the sequential application to client executable files. An executable file contains the code that corresponds to the function implementation in the sequential program. The example below represents how the client generator encapsulates the task function in the client source code.

Example 5.2 Client's pseudo code

Main
{
1. *Initialize BOINC*
2. *Open the input file (resolve logical name first)*
3. *Get the size of input file*
4. *See if there's a valid checkpoint*
5. *If so seek input file and truncate output file*
6. *Main loop*
7. *Create checkpoint (If required)*
8. *Create the output file*
9. *Copy there the appropriate results*
10. *Terminate BOINC*

}

main ()
}
Initialize BOINC
Open the input file (resolve logical name first
Get the size of input file
See if there's a valid checkpoint
If so seek input file and truncate output file
Function implementation
Create checkpoint (If required)
Create the output file
Copy there the appropriate results
Terminate BOINC
}

The main goal of the mCluster was the task dependency resolving. As referred above, the client generator transforms a function implementation in a sequential application to a BOINC executable. Any executable that is downloaded from the clients is processed according to the input data files that are also given. The workunits that are created from BOINC are these data files. Dealing with BOINC terms an executable corresponds to an application. That means that each task in mCluster is a separate application. A project can consist of many applications. In brief, the sequential application using BOINC terms is referred to the project and the tasks are the applications which compose it.

Task dependencies are resolved according to the below manner. Workunits that are the input data for an executable are not created until the tasks that have dependence on the corresponding task finish their execution. In this way, the appropriate executable which are created and dispatched to the clients without having the input data cannot start the processing. Task scheduler, which will be thoroughly analyzed in the next section is responsible for handling the completed results and creating the appropriate workunits in agreement with the dependencies they have.

4.7 mCluster architecture

In the previous section we referred in detail about BOINC, its features, architecture and the mechanisms that are provided. Also identified its' main disadvantages as well as the kind of applications that are suitable with its framework. Applications that have task dependencies cannot be resolved because there isn't any mechanism covered from BOINC for recognizing them so as there is no communication between clients during the execution in order to handle the dependencies. Dealing with this, the mCluster programming model was implemented on the top of BOINC.

According to this programming interface and in order to support dependencies between tasks that are not handled from BOINC new components had to be added in its architecture.

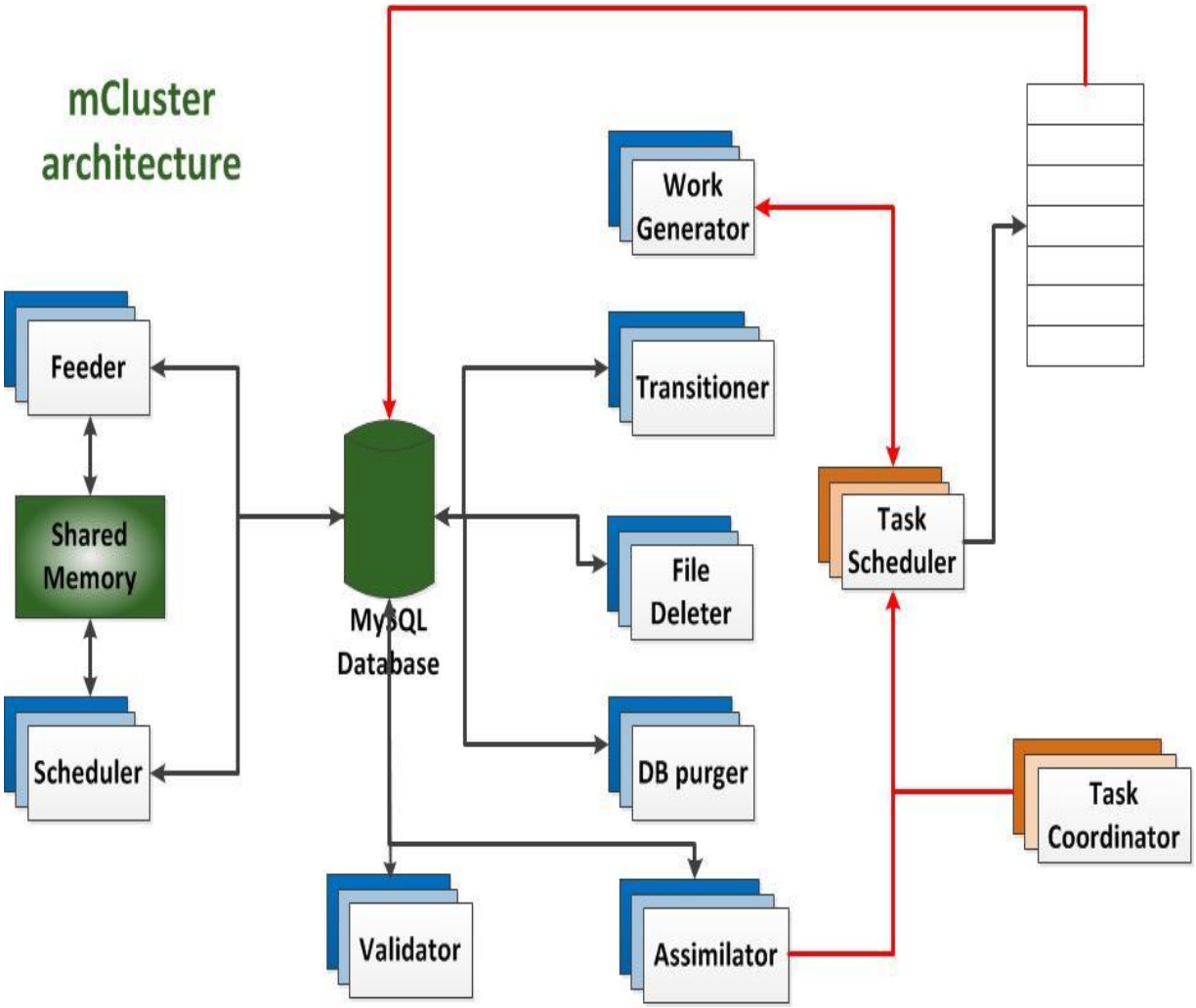


Figure 4.9: mCluster architecture

As shown in figure 4.9 the architectural differences between BOINC and mCluster are the Task Coordinator, Task scheduler programs that are going to be used for dependency handling. In this way, the tasks are going to be stored in the queue in the correct order before they are going to be distributed. These daemons are running alongside the others managing the tasks. Specifically:

Task coordinator: This daemon is responsible for creating the dependency graph based on the programmer's annotations. Source to source translator creates the appropriate tables which contain information about each task. Afterwards the task coordinator according to these creates the appropriate graph representing the task dependencies. Tasks that have no dependencies are the parent nodes. Task coordinator only creates that dependency graph, according to which the task scheduler is going to create the queue order.

Task scheduler: It is the daemon responsible for creating the appropriate workunits that belong to each task. In order to resolve the generated dependencies, the creation of the workunits is extremely important. When a device downloads the executable, it cannot start the execution until the input data is arrived. Task scheduler takes advantage of that and creates the workunits only when the results from the task that they depend on are finished. It communicates with the assimilator in order to check when a result is assimilated in order transfer it to the workunit which depends on. It additionally schedules the work generator. Workunits must be created if only the task they belong to has no dependencies. Finally, it performs queue operations like insert/delete tasks or update their position in the queue order.

Queue order: The queue order that used to handle task dependencies acts as a temporary buffer where tasks are stored. If a client downloads a task, it will not start the execution until the equivalent workunit is downloaded. Dealing with no dependencies, work generator creates the workunits with no restrictions which are going to be stored in the database. On the other hand, workunits must not be created until task scheduler decides to create them according the appropriate task priority. In that way workunits are created according to the position of the task in the queue and then are stored in the database.

In figure 5.4 the flowchart represents the life-time of a job in the mCluster framework. Firstly, the tasks coordinator creates the appropriate dependency graph according to the annotation that the programmers give. Next the task scheduler creates the tasks queue order. Tasks that have no dependencies are immediately marked as ready to be created. The workunits of these tasks are created and forwarded to the clients so as to execute them. When a result arrives after the validation and the assimilation the task scheduler checks if it is dependent on other ones. If not, this task is going to be deleted. On the other hand, updates, the queue order according the dependencies that this task has. Finally, the task scheduler informs the work generator to create the appropriate workunits that belong to tasks that have no more dependencies.

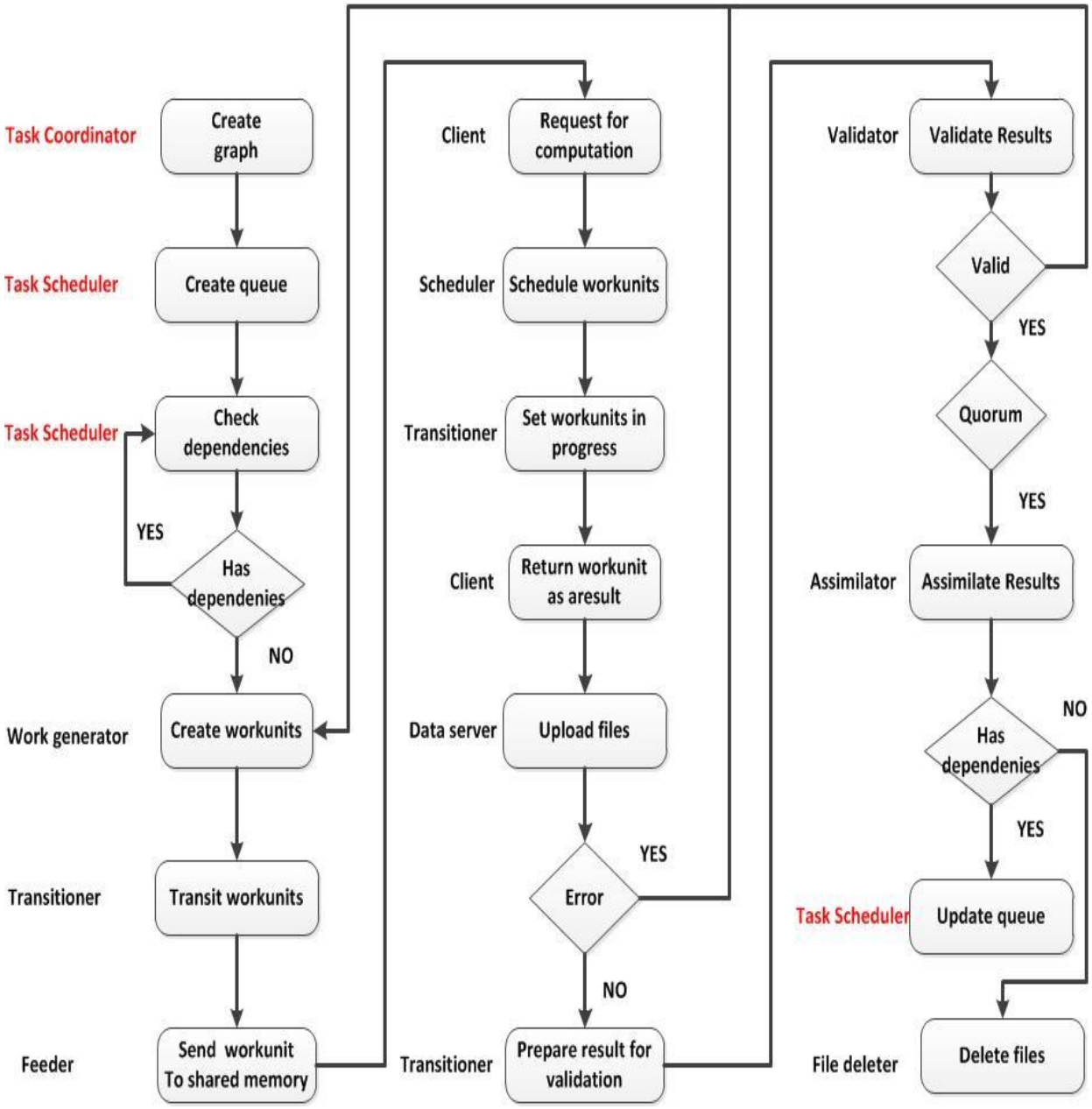


Figure 4.10: mCluster work life-time

Section 5

Performance evaluation

5.1 Experimental setup

We evaluate the performance of the mCluster by executing four separate applications with different level of parallelism. For the evaluation, we used six client devices. Table 5.1 shows each device specifications, all running the android operating system. Moreover, for collecting more accurate experimental results each device was set to use only one of the available cores. The main motivation behind the evaluation of mCluster in a mobile computing system is its ability to achieve performance similar to personal computers with less energy consumption. The table below displays each device characteristics such as the chipset, the CPU and the internal memory. All the results are measured according the log file that BOINC provides.

Table 5.1 Device specifications

Device	Operating system	Chipset	CPU	Internal memory
Asus Google Nexus 7	Android , v4.3 (Jelly Bean),	Qualcomm Snapdragon S4Pro	Quad-core 1.5 GHz Krait	16 GB ROM, 2 GB RAM
2X Asus Google Nexus 7	Android , v4.4.2 (Kitkat),	Qualcomm Snapdragon S4Pro	Quad-core 1.5 GHz Krait	16 GB ROM, 2 GB RAM
1X Asus Google Nexus 7	Android , v5.1.1 (Lollipop),	Qualcomm Snapdragon S4Pro	Quad-core 1.5 GHz Krait	16 GB ROM, 2 GB RAM
2X Samsung I9505 Galaxy S4	Android , v5.1.1 (Lollipop),	Qualcomm APQ8064T Snapdragon 600	Quad-core 1.9 GHz Krait 300	16 GB ROM, 2 GB RAM

5.2 Experimental results

Matrix-processing application

Firstly, in order to evaluate the mCluster, an application that consists of seven dependent tasks executing vector processing benchmarked. TASK1, TASK3 and TASK4 perform matrix addition, while TASK2, TASK5 and TASK6, TASK7 perform matrix subtraction. The dependencies between tasks are shown in figure 5.1.

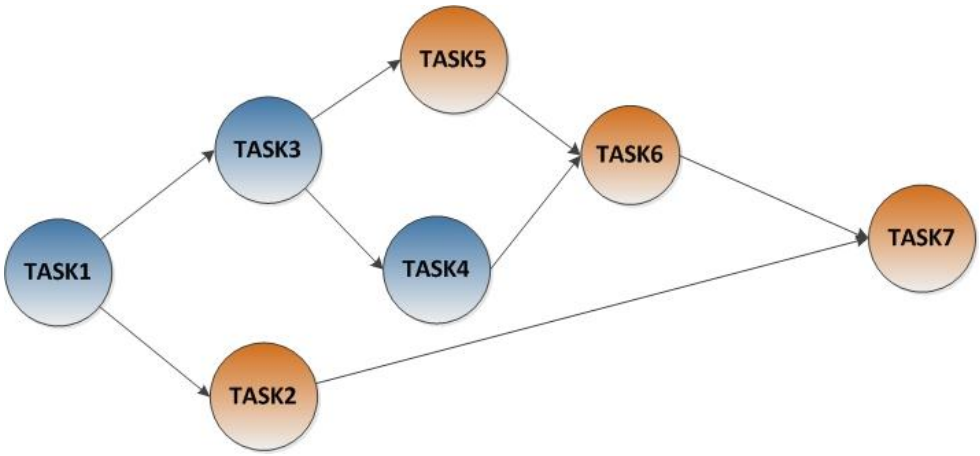


Figure 5.1: Task dependency graph

Initially, for the evaluation each matrix consists of 1000 elements. As shown in figure 5.2, the time of execution, upload and download processes is equivalent to 1 second. In addition, due to the existing dependencies between them, task scheduler lasts 0.10 seconds for each task to create the right output that is attached to the next task.

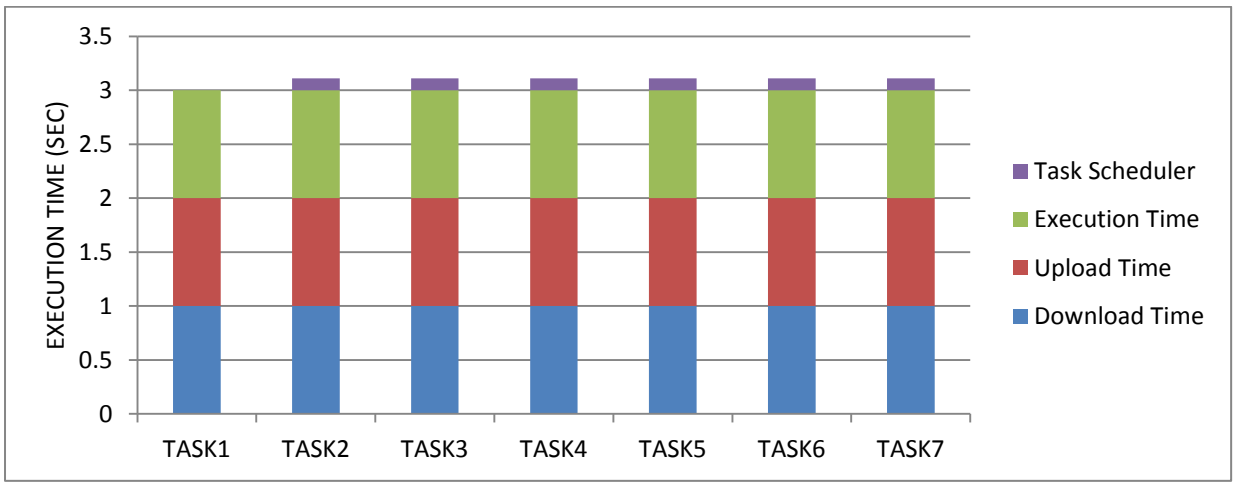


Figure 5.2: Download, upload, execution time of matrices consisting of 1000 elements

In the next experiment, the length of the matrices changed to 10000. As shown in figure 5.3, the download and upload time remained the same as in figure 5.2, while the execution time doubled. Task scheduler time was 0.15 second for each task. That happened due to the increase of data amount that had to be processed.

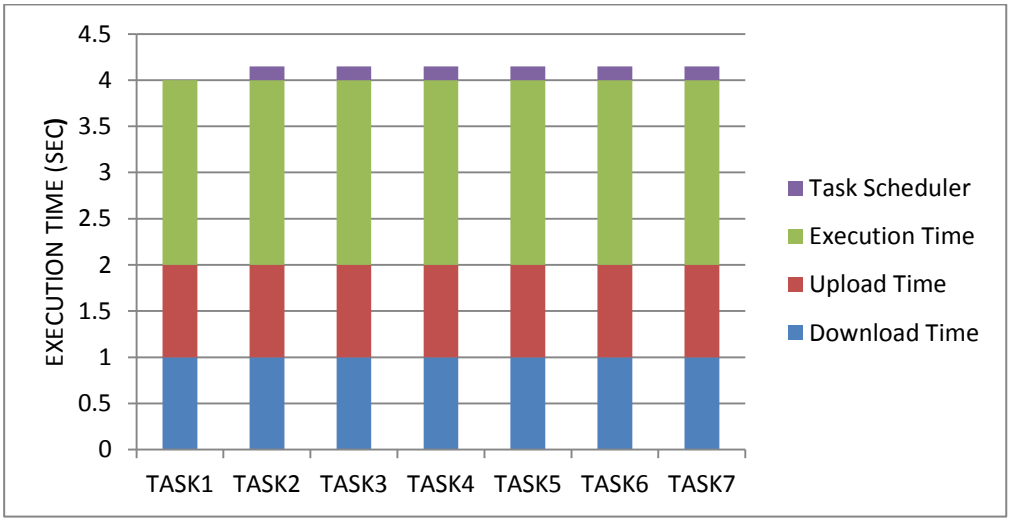


Figure 5.3: Download, upload, execution time of matrices consisting of 10000 elements

Summarizing, we had to refer that the download, and upload time for each task depend on the internet bandwidth. Moreover, this application takes 2.1 seconds to run on a single machine when the matrix consists of 10000 elements and 0.5 sec when the size is 1000 elements. That time is considerably smaller compared to the total mCluster's time. That occurs because in our case we had to deal with extra overhead (about 3.5% of the total time) , related to data transfer time, as the time needed for reading and writing to data files. That overhead may be gradually decreased while the number of the participant devices increases. As a result, the parallel task execution using an increased number of available resources leads to better performance.

In figures 5.2 and 5.3, we presented the time required for our distributed system to execute, download and upload each task. In figure 5.4, we represent the total execution time of the application using different number of devices. At first, by executing all the tasks in a single device, we did not take the advantage of the parallelization that occurs in this application. As we can see, TASK2 and TASK3 as well TASK4 and TASK5 can execute concurrently. Dealing with that, one more device used to benefit from the parallelization. As shown in figure 5.4, the total execution time using two devices in comparison with one device, regardless the number of elements is about 1.27 times faster. In this example, only two tasks could be executed simultaneously so increasing the number of the available devices will not speed up the execution time. On the other hand, in applications which have more parallelization, increasing the number of the available devices will increase the total performance of the system.

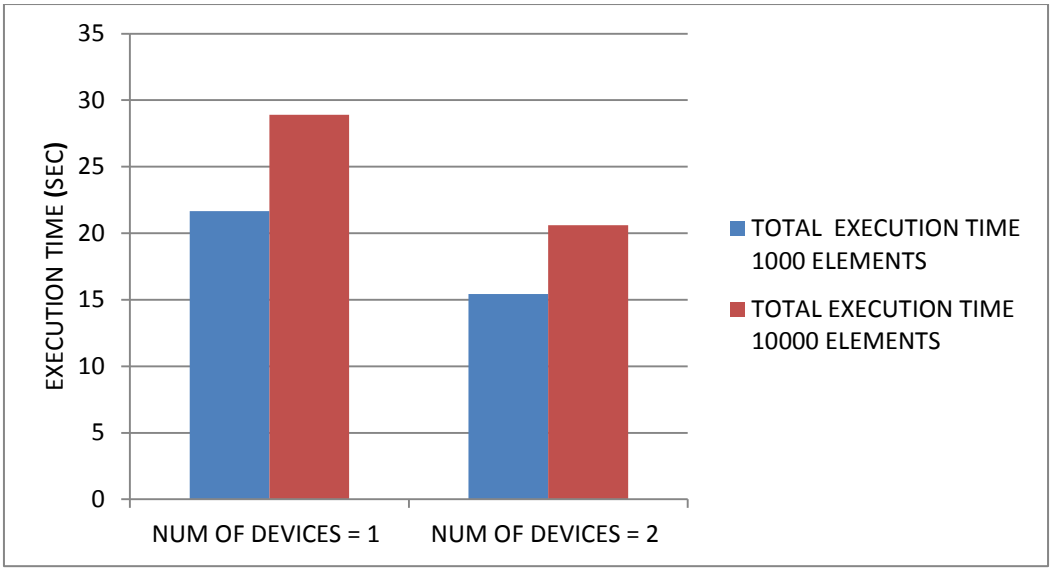


Figure 5.4: Total execution time using one and two devices.

The "Inferior-Oliver Application"

The cerebellum is one of the most complex areas of the brain and plays an important part on motor control and learning [46]. It does not initiate movement but influences the motor control region of the brain in order to guarantee coordination, accurate timing and precision on the body's activities. It also plays an important role in the sensing of rhythm, enabling the understanding of concepts such as music or harmony. The Olivocerebellar circuitry is a relatively well-charted region of the brain. The model of the system reveals that the brain structure in the area is highly repetitive and basically consists of the granule-cell layer (GCL), Purkinje-cell layer (PC), deep-cerebellar-nuclei (DCN), and inferior-olive (IO) nuclei. The inferior olivary nucleus provides one of the two main inputs to the cerebellum: the so-called climbing fibers. Activation of climbing fibers is generally believed to be related to timing of motor commands and/or motor learning. The models of this biological circuit as shown in figure 5.5 will be used as a proof-of-concept application for the SERFER paradigm and specifically an Inferior Olive neuron model [47]. The distributed and massively dataflow nature of such models lends itself naturally to the SERFER paradigm.

Biologically accurate brain simulation, such as the one in this application, is a highly relevant topic for neuroscience for a number of reasons [48]. The main goal is accelerated brain research by the creation of more advance research platforms. Even though a significant amount of effort is spent in understanding brain functionality, the exact mechanisms in most cases are still only hypothesized. Fast and real-time simulation platforms can enable the neuroscientific community to more efficiently

test these hypotheses. Better understanding of brain functionality could potentially lead to a number of critical practical applications: (1) Brain rescue: If brain functions can be simulated accurately enough and in real-time, this can lead to robotic prosthetics and implants for restoring lost brain functionality. (2) Advanced A.I.: It is believed that greater understanding of biological systems and the richer computational dynamics of their models, can lead to more advanced, artificial-intelligence and robotic applications. (3) New architectural paradigms: Alternatives to the typical Von-Neumann architectures.

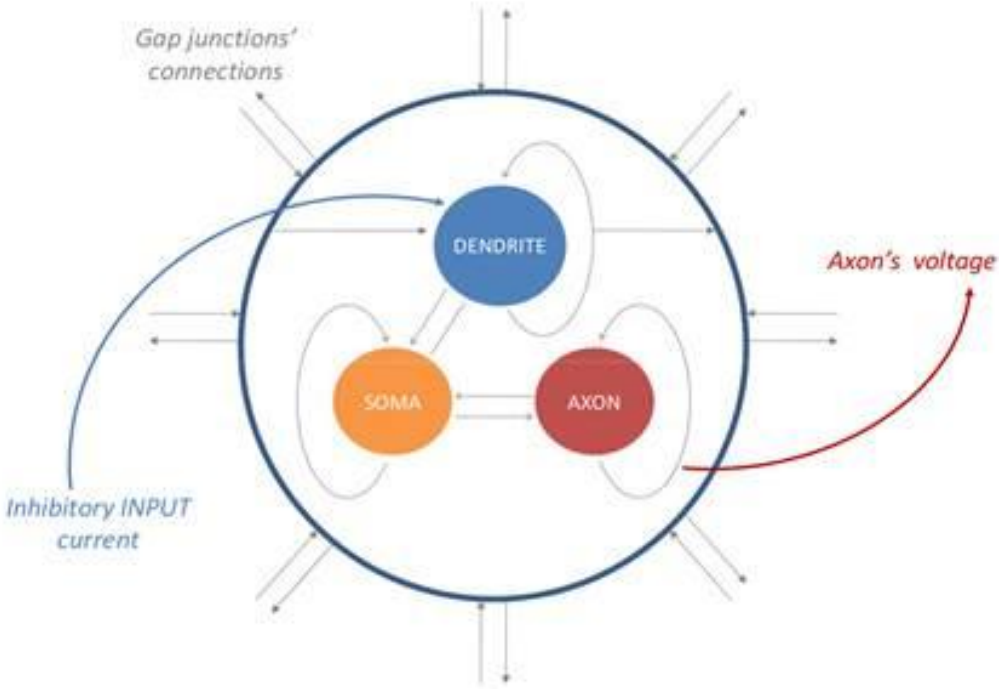


Figure 5.5: The Inferior Olive Model Structure. Visible are the I/O of the cell and its internal computational stages (Dendrite, Soma and Axon)

Figure 5.6 shows the application execution procedure. According to this, the Inferior-Oliver application can execute in a particular number of simulation steps (Simsteps). In every simulation step each task calculates a cell's state within the 2-d cell grid with dimensions Dim_x and Dim_y. All the internal tasks in each Simstep can be executed concurrently. Furthermore, each Simstep depend on the previous one. Dealing with that, all of the previous cells that belong to it had to be finished in order to start the next one its execution.

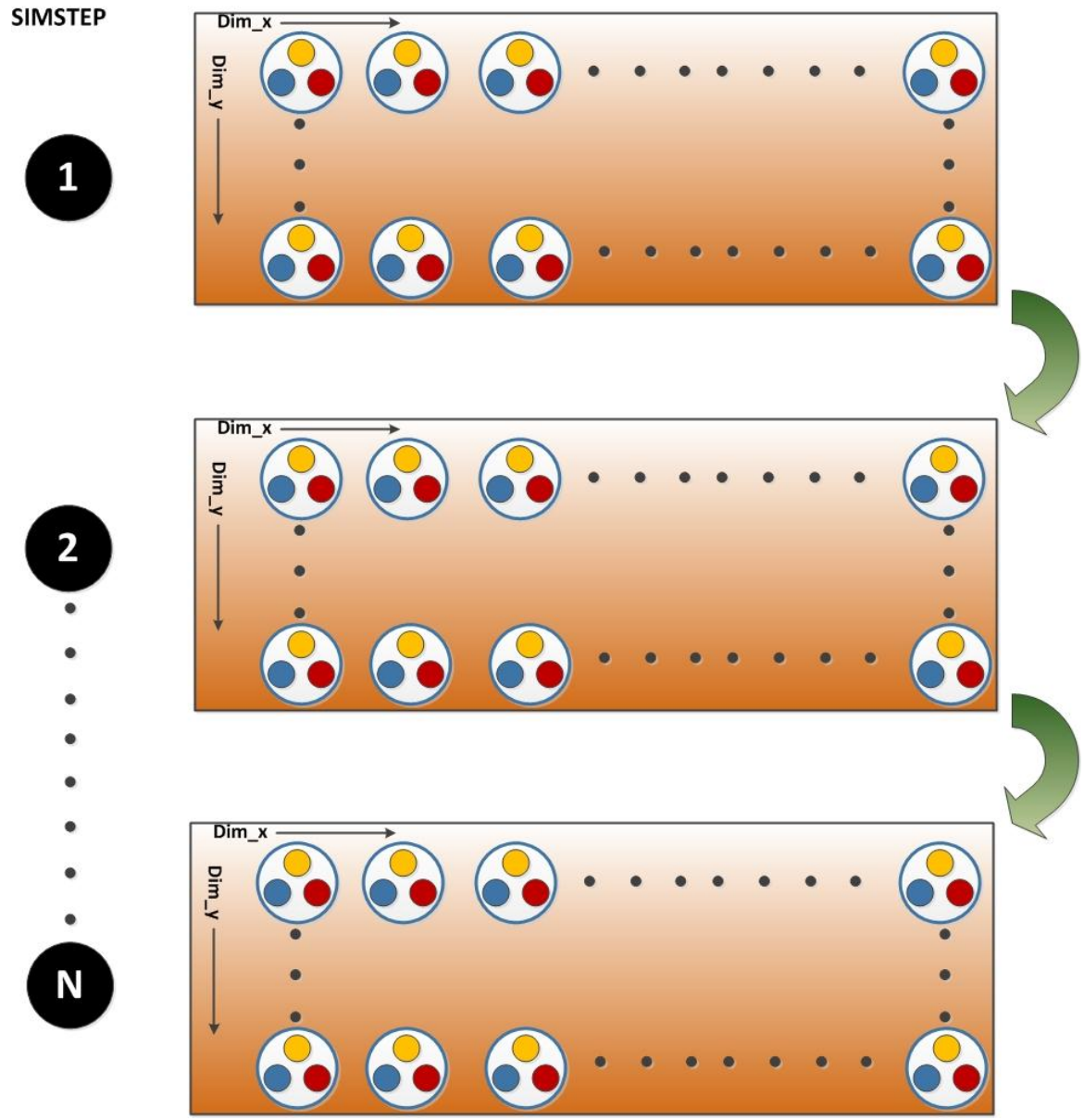


Figure 5.6: Inferior-Oliver Application execution procedure

In the figure 5.7 below, we can see the execution, upload and download time for different grid sizes. Each task of the Simstep is executed from any device according to BOINC scheduling policies. For each device, we are able to specify the number of the tasks that are going to be sent in order to achieve better performance. BOINC provides a variety of customizations depending on each application's requirements in order to dispatch and execute the available tasks in an efficient way.

For our example, they undertake maximum 16 tasks per request. This way, all the available devices execute the same number of tasks. In Figures 5.7 and 5.8 only one device used for the evaluation in order to be used for further analysis.

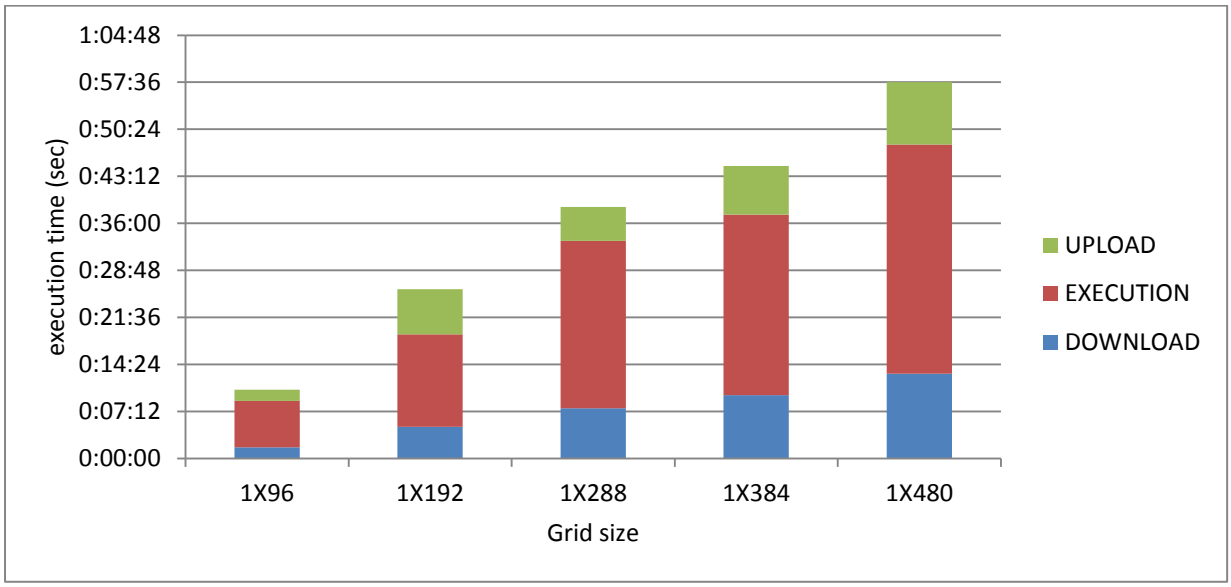


Figure 5.7: Download, upload, execution time for first Simstep for grid size 1X96, 1X192, 1X288, 1X384, 1X480 using one device

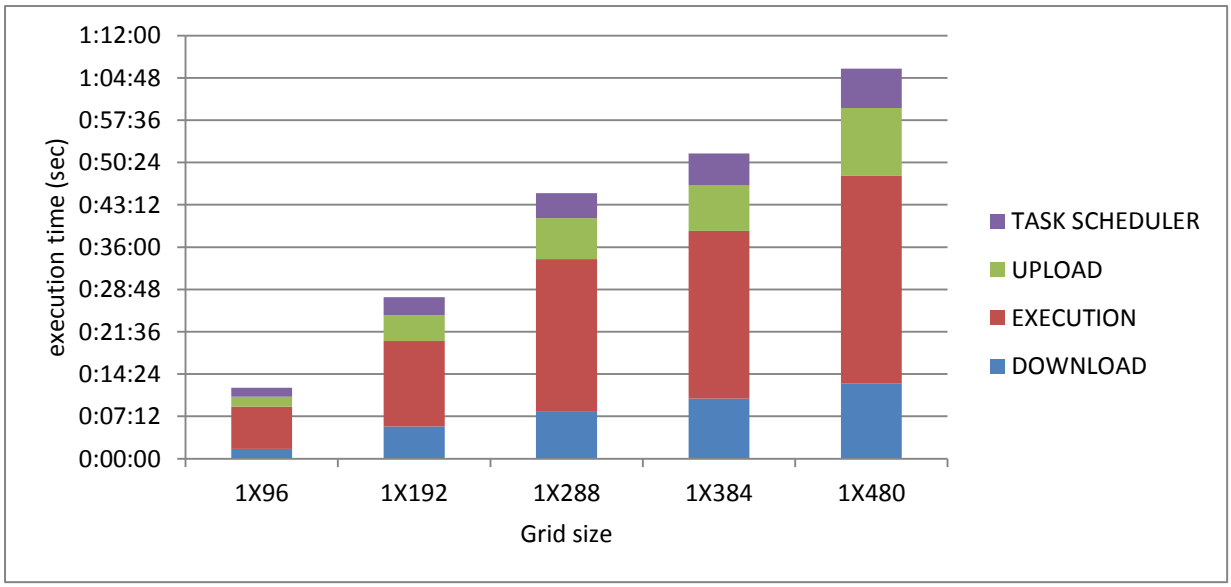


Figure 5.8: Download, upload, execution time for second Simstep for grid size 1X96, 1X192, 1X288, 1X384, 1X480 using one device

As shown in figure 5.8, in order to start the execution of the second Simstep, the first one should have finished. According to this, task scheduler counts the number of the assimilated results in the database and transfers the results to this Simstep. The overhead that task scheduler has, is related to the time it needs to read and write the right data values from each tasks of the Simstep in order to be attached to the next one. Furthermore, we had to point out that task scheduler (server work) time increases according to the grid size.

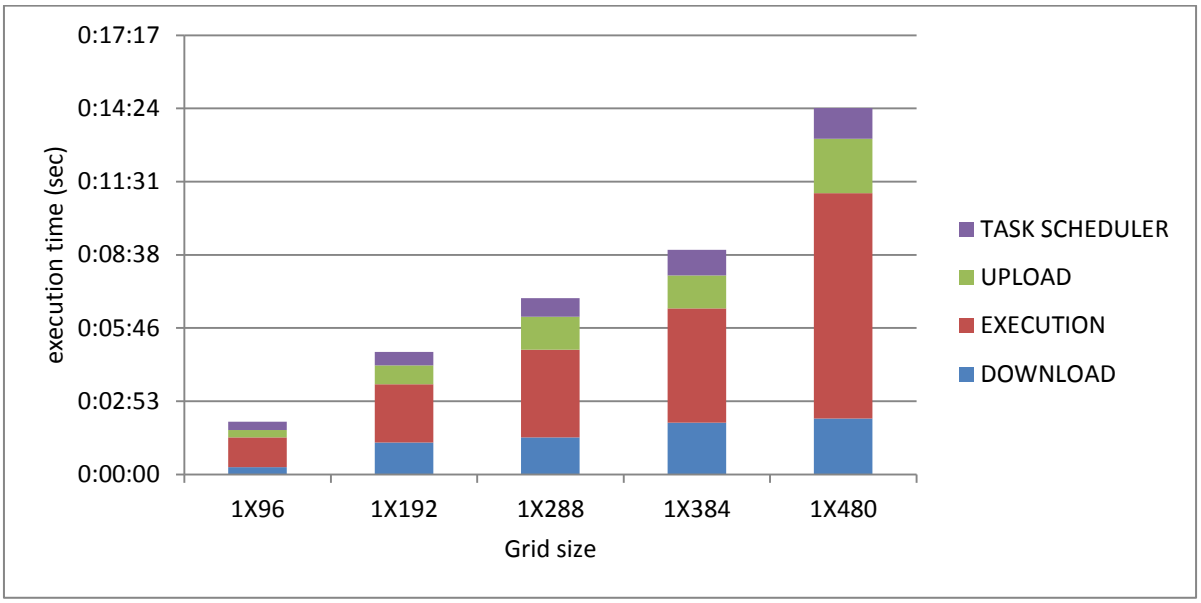


Figure 5.9: Download, upload, execution time for second Simstep for grid size 1X96, 1X192, 1X288, 1X384, 1X480 using 6 devices

As above mentioned, in figures 5.7 and 5.8 only one device is used for the evaluation. On the other hand in figure 5.9 used the processing power of both six devices. The total time spent in order to execute this Simstep is about 5 times less compared to figure 5.8. According to these, we come to the conclusion that the increased number of the available resources in combination with the high level of parallelization in the Inferior-Oliver application, leads to better performance. Furthermore we can point that the execution time for each task compared to the next one is not exactly the same. That happens because each target device's has different execution time. Several parameters are responsible for this result such, as the usage of the CPU and the processes that are simultaneously running in the device. Usually in volunteer computing systems, the participant devices have a great variety of CPUs, different internal storage size and other characteristics.

As noticed above, in this application all the tasks in the same Simstep can execute concurrently. We can take the advantage of this situation by using more devices for the evaluation. As shown in figure 5.10 while the number of the available devices increases, the total execution time of the application is decreased. In particular, if dealing with one device, the total the total execution time for grid size 1X96 is 10 minutes. On the other hand, by using 96 volunteer devices with each of them executing one task, the elapsed time could be reduced even to 2 minutes, time about 5 times faster in comparison with executing all tasks in a single device. Similarly, in bigger grid sizes, which have higher level of parallelism, more devices lead to better performance. In conclusion, because of the most devices nowadays consists of more than one processor the number of devices could be equal to the number of processors each of them have.

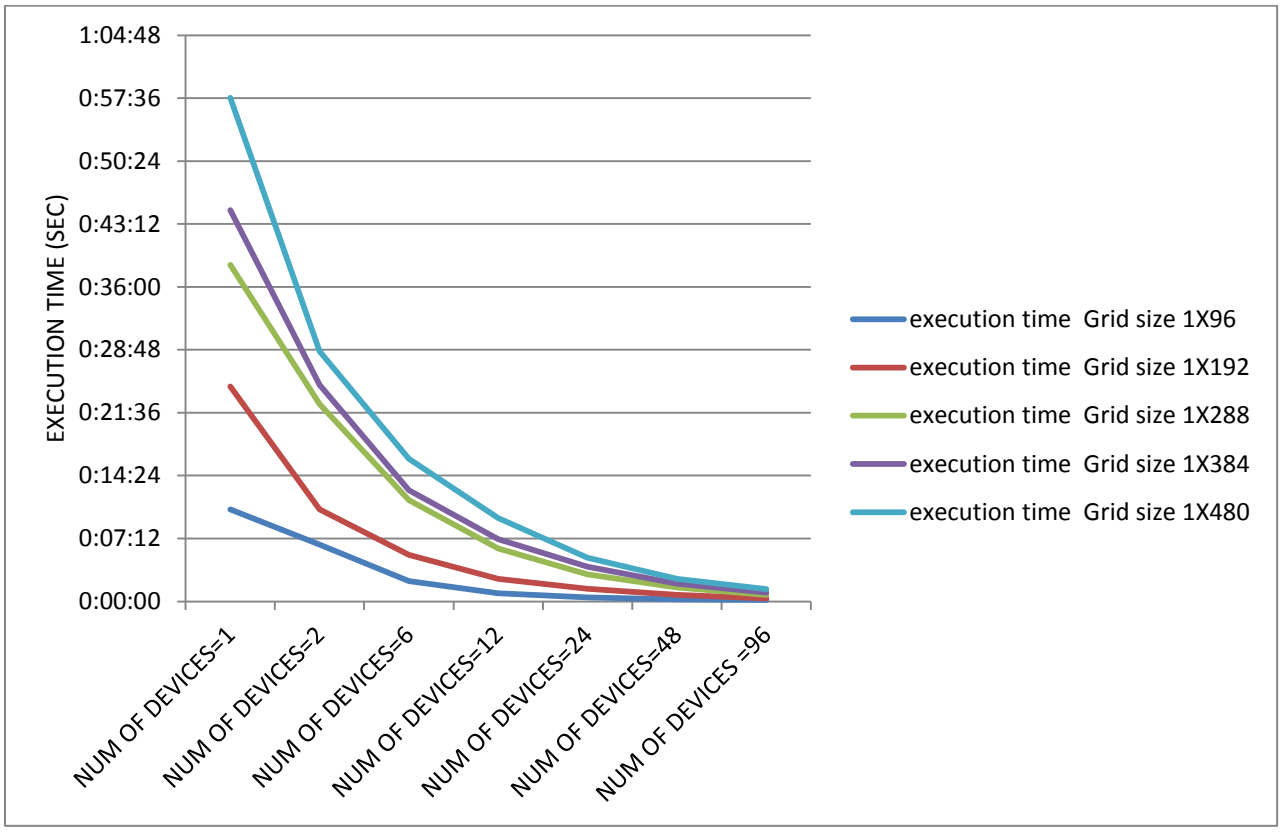


Figure 5.10: Total execution time in for grid sizes 1X96, 1X192, 1X288, 1X384, 1X480 using an increasing number of volunteer devices.

The "Black-Scholes formula"

The Black–Scholes or Black–Scholes–Merton model is a mathematical model of a financial market containing derivative investment instruments. From the model, one can deduce the Black–Scholes formula, which gives a theoretical estimate of the price of European-style options. The formula led to a boom in options trading of the Chicago Board Options Exchange and other options markets around the world. It is widely used, although in many cases are adjustments and corrections, by options market participants. Many empirical tests have shown that the outcome price is "fairly close" to the observed prices.

The Black-Scholes model is used to calculate the theoretical price of European put and call options, ignoring any dividends paid during the option's lifetime. While the original model did not take into consideration the effects of dividends paid during the life of the option, the model can be adapted to account for dividends by determining the ex-dividend date value of the underlying stock. The figure 5.11 shows the Black-Scholes pricing formula for put and call options.

call option:

$$c = SN(d_1) - Xe^{-rT}N(d_2)$$

put option:

$$p = Xe^{-rT}N(-d_2) - SN(-d_1)$$

where

$$d_1 = \frac{\ln(S/X) + (r + \sigma^2/2)T}{\sigma\sqrt{T}}, \qquad d_2 = d_1 - \sigma\sqrt{T}$$

S: Stock price.
X: Strike price of option.
r: Risk-free interest rate.
T: Time to expiration in years.
σ: Volatility.
$N(x)$: The cumulative normal distribution function.

Figure 5.11: The Black-Scholes pricing formula

The below figure shows the execution, upload, download and scheduler time spent calculating the call and put option for different size of input buffers.

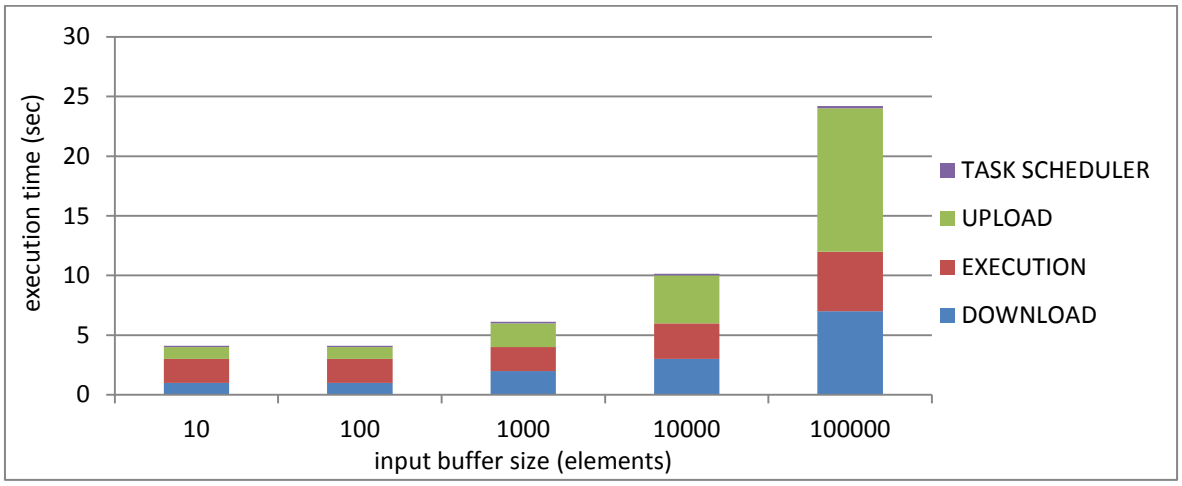


Figure 5.12: Task execution, download, upload and task scheduler time for different input buffer size using 2 devices

As we can see in the figure above large input buffer size leads mainly to more download and upload time. Specifically when the buffer size expanded into 100000 elements, the download time increased approximately 10 times compared to the initial price. Taking the advantage of volunteer computing which offers the ability for a large number of volunteer devices we divided the buffer size in 100 smaller tasks each of them consists of 100 elements. In the figure below we can see the total upload download and execution time when these tasks are dispatched into 1,10,20,50 and 100 devices.

This way the transfer time will shared to the available devices which are going to download, execute and upload the available tasks simultaneously. As a result the increasing number of the available devices outcome less total transfer time.

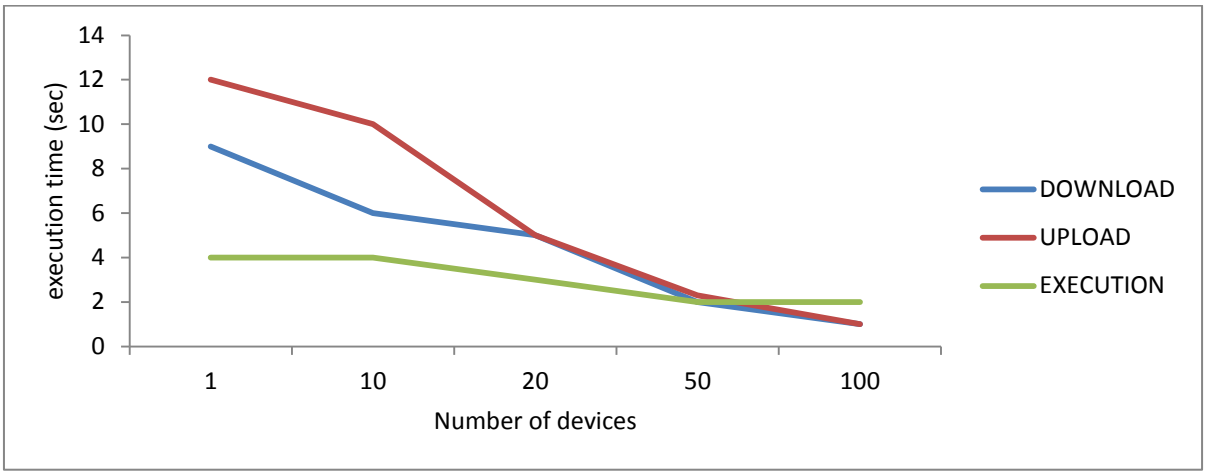


Figure 5.13: Download execution and upload time for 100000 elements input buffer size when divided into 100 smaller tasks using an increasing number of volunteer devices.

"Image processing application"

The other application used for the evaluation deal with image processing. Via this application the original image is divided into smaller pieces that are afterwards applied the following filters.

Grayscale: The grayscale filter transforms an RGB image to one in which the only colors are shades of gray.

Gaussian blur: In image processing, a Gaussian blur (also known as Gaussian smoothing) is the result of blurring an image by a Gaussian function. It is a widely known effect in graphics software, typically used for image noise and detail reduction.

Laplacian: The Laplacian is a commonly used filter for edge detection in digital images. It is often applied to an image that has first been smoothed with something approximating a Gaussian smoothing filter in order to reduce its sensitivity to noise, and hence the two variants will be described together here. The operator normally takes a single gray level image as input and produces another gray level image as output.

Threshold: This filter transforms an image into a binary image by converting each pixel according to whether it is inside or outside a specified range.

Initially, this application consists of 4 tasks each of them is attached to one of the below filters. In order to reduce the total transfer time for each part of the image we reduced the number of tasks by embodying two filters in one task. As shown in the figure 5.14 where the image has divided into 16 equivalent pieces, each part of the image consists of two depended tasks. The first one applies the grayscale and the Gaussian blur filter and the second one the edge Laplace and the threshold filter.

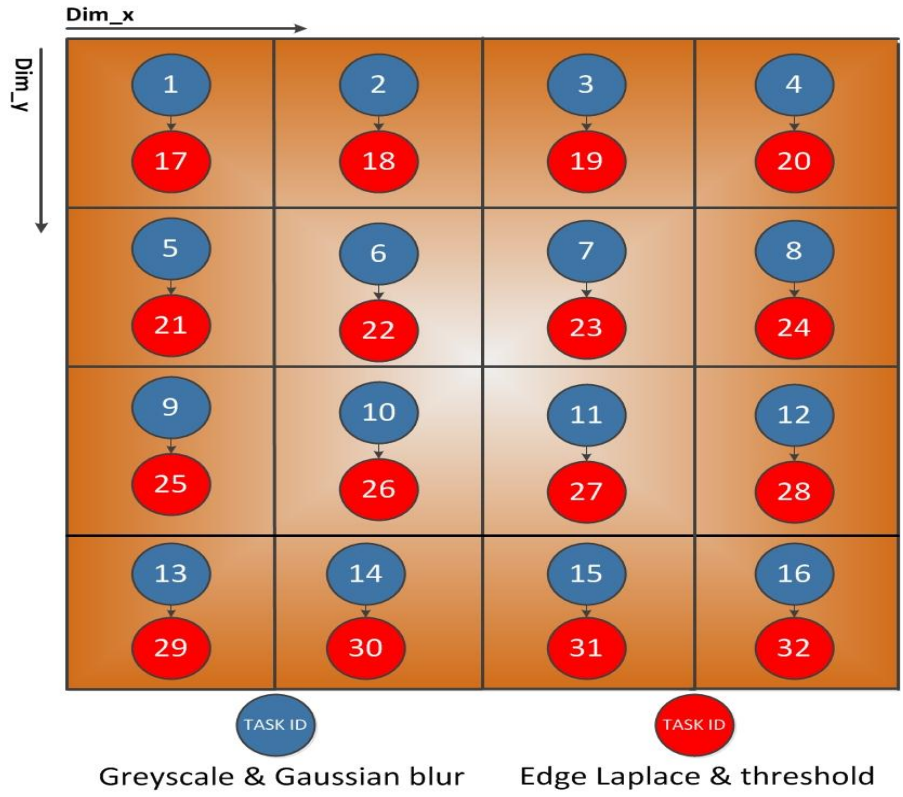


Figure 5.14: Application execution procedure

In order to evaluate the application in our framework, an image of 1024 pixels width and 916 pixels height is used. The image for our example is divided in 16 pieces of 256 pixels width and 228 pixels height each. This way, 32 tasks are going to be executed from the clients. The outcome is shown in the figure 5.15 below.

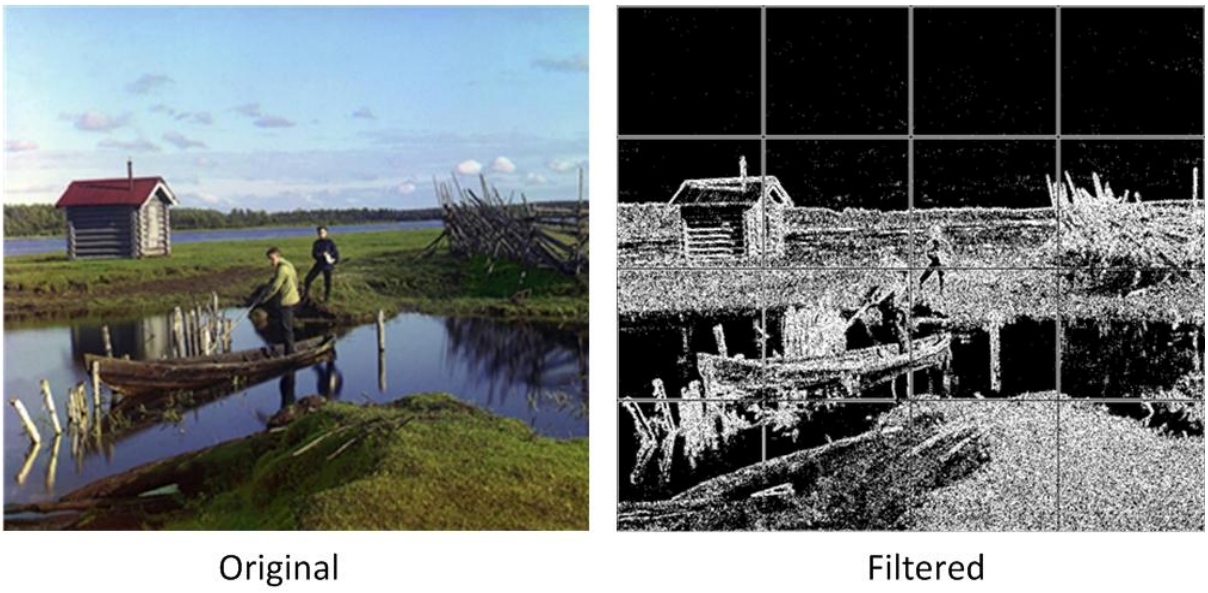


Figure 5.15: Image before and after applying filtering

The figure 5.16 shows the upload, download execution and task scheduler time for each of the created tasks. The first 16 tasks apply the grayscale and Gaussian blur filter and the rest of them the edge Laplace and threshold filter. The input for the tasks 17-32 is the output of tasks 1-16. As we can see below in these tasks there is an extra overhead related to the task scheduler which is responsible for handling the dependence and create the right input when any of the corresponding tasks finish.

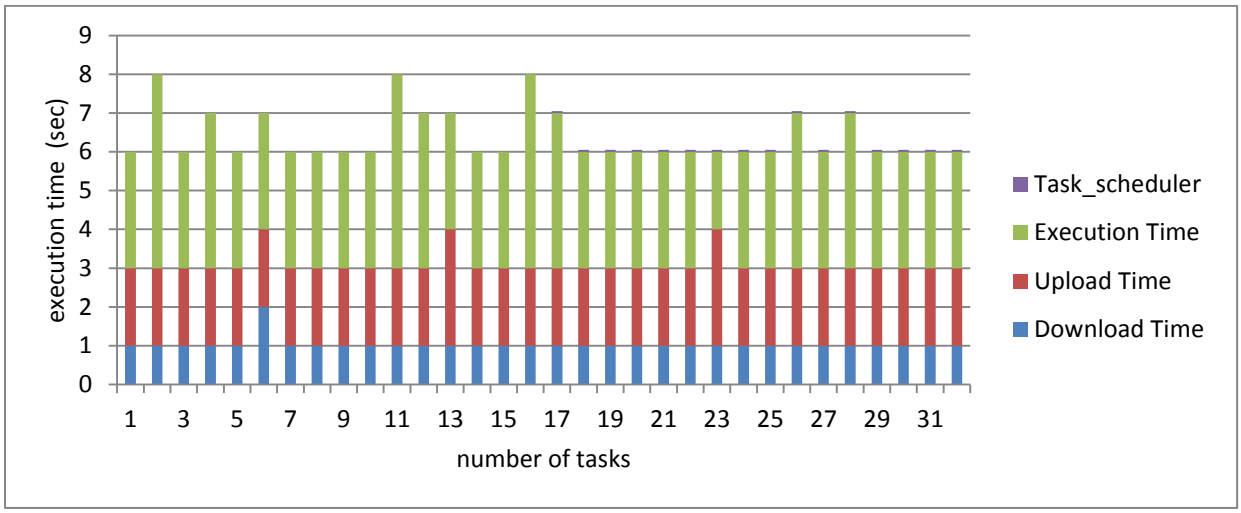


Figure 5.16: Task scheduler, upload download and execution time for every task using 2 devices

The figure 5.17 below shows the average task execution, download and upload time while splitting the image into 4, 16, 32, 64 equal pieces. When the image splits into 4 pieces, the average task execution time is about 4 seconds. Increasing the number of the pieces leads to less input data size for the task, as well as less execution time, about 2 seconds for 64 pieces. In the same way the download and the upload time decrease, as we can see below. To conclude, according to this application better performance can be achieved by splitting the initial image into more pieces.

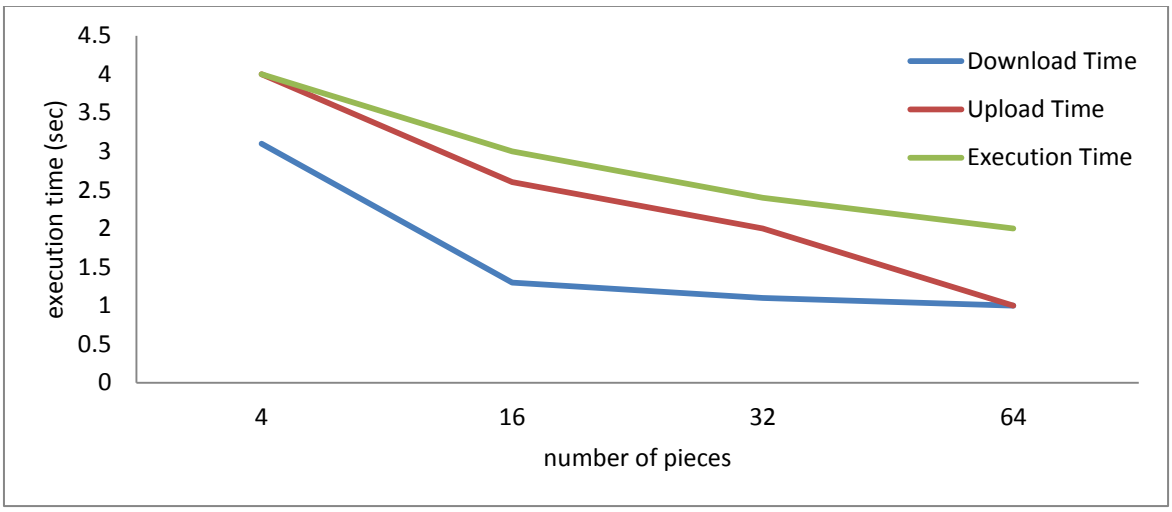


Figure 5.17: Upload, download and execution time for different task size using 2 devices

Section 6

Conclusions and future work

6. 1 Conclusion

Volunteer Computing is a new paradigm of distributed computing where the ordinary computer owners volunteer their computing power and storage capability to scientific projects. The increasing number of Internet-connected PCs allows Volunteer Computing to provide more computing power and storage capacity than what can be achieved with supercomputers, clusters and grids.

This thesis presented a task based programming model based on BOINC infrastructure which provides the ability to transform a sequential application to ready to be executed tasks from volunteer devices, via the appropriate annotations. BOINC is a composite of several distinct applications, some of which have been created by the developers and others that are developed separately by each public resource computing project. Creating a BOINC project is a very difficult process because of the level of comprehension required for the interactions between its components. Dealing with that, a mechanism that creates these components according to each application's properties had to be implemented, that can become a very useful tool for the developers. Finally in order to support task dependencies that BOINC does not, structural changes on the infrastructure needed.

In conclusion, mCluster provides solutions to the drawbacks of regular BOINC. Moreover it helps application developers by providing an easy–to-use application programming interface. Nevertheless the most important contribution of mCluster is that improves BOINC functionalities by allowing applications with dependencies to run under it.

6.2 Future work

Currently, client and server scheduling are not well integrated. The server sends jobs to clients without having important information related to the services that they can provide. Since the server has no information about work queued or in progress on the client, it can send jobs that will cause deadlines to be missed. Furthermore, information related to network connection (location, type) are also required , as well as by having client record statistics about periods when the host is powered off or not connected, would increase the total reliability and performance of the system.

BIBLIOGRAPHY

[1] David P. Anderson, Jeff Cobb,Eric Korpela, Matt Lebofsky and Dan Werthimer.SETI@home.An Experiment in Public-Resource Computing.

[2] David P. Anderson and Gilles Fedak.The Computational and Storage Potential of Volunteer Computing.

[3] Derrick Kondo David P. Anderson and John McLeod VII.Performance Evaluation of Scheduling Policies for Volunteer Computing.

[4] Franck Cappello,Samir Djilali,Gilles Fedak,Thomas Herault,Frédéric Magniette,Vincent Néri and Oleg Lodygensky. Computing on large-scale distributed systems: XtremWeb architecture, programming models, security,tests and convergence with grid.

[5] Samir Djilali. P2P-RPC: Programming Scientific Applications on Peer-to-Peer Systems with Remote Procedure Call.

[6] Ian Foster,Carl Kesselman and Steven Tuecke.The Anatomy of the Grid.Enabling Scalable Virtual Organizations.

7] O. Lodygensky,G. Fedak,F. Cappello,V. Neri,M. Livny and D. Thain. XtremWeb & Condor : sharing resources between Internet connected Condor pools.

[8] Dejan S. Milojicic,Vana Kalogeraki,Rajan Lukose,Kiran Nagaraja,Jim Pruyne,Bruno Richard,Sami Rollins and Zhichen Xu.Peer-to-Peer Computing.

[9] Christian Ulrik Søttrup and Jakob Gregor Pedersen. Developing Distributed Computing Solutions,Combining Grid Computing and Public Computing.

[10] Nagarajan Kanna,Jaspal Subhlok,Edgar Gabriel,Eshwar Rohit and David Anderson.A Communication Framework for Fault-tolerant Parallel Execution.

[11] Derrick Kondo,Bahman Javadi,Paul Malecot,Franck Cappelloand David P. Anderson.Cost-Benefit Analysis of Cloud Computing versus Desktop Grids.

[12] Ian Foster,Yong Zhao,Ioan Raicu and Shiyong Lu.Cloud Computing and Grid Computing 360-Degree Compared.

[13] Dimitris Theodoropoulos,Polyvios Pratikakis and Dionisios Pnevmatikatos. Efficient Runtime Support for Embedded MPSoCs.

[14] Hans Vandierendonck,Polyvios Pratikakis and Dimitrios S. Nikolopoulos.Parallel Programming of General-Purpose Programs Using Task-Based Programming Models.

[15] Yoav Etsion,Felipe Cabarcas,Eduard Ayguade,Alejandro Rico,Jesus Labarta,Alex Ramirez,Rosa M. Badia and Mateo Valero.Task Superscalar: An Out-of-Order Task Pipeline.

[16] Josep M. Perez,Rosa M. Badia and Jesus Labarta.A Dependency-Aware Task-Based Programming Environment for Multi-Core Architectures.

[17] Bradley Charles Goldsmith.DISTRIBUTED COMPUTING AND COMMUNICATION IN PEER-TO-PEER NETWORKS.

[18] Sangho Y, Emmanuel Jeannot,Derrick Kondo and David P. Anderson.Towards Real-Time,Volunteer Distributed Computing.

[19] Daniel Lombra˜na Gonz´alez,Francisco Fern´andez de Vega,Leonardo Trujillo,Gustavo Olague,Lourdes Araujo,Pedro Castillo,Juan Juli´an Merelo and Ken Sharman.Increasing GP Computing Power for Free via Desktop GRID Computing and Virtualization.

[20] M. Taufer,D. Anderson,P. Cicotti and C.L. Brooks.Homogeneous Redundancy: a Technique to Ensure Integrity of Molecular Simulation Results Using Public Computing.

[21] Linda Ponta.An overview on public resources computers:BOINC.

[22] Ayon Basumallik, Seung-Jai Min, Rudolf Eigenmann. Programming Distributed Memory Sytems Using OpenMP

[23] Ant´onio Amorim,Jaime Villatey and Pedro Andradez.HEP@HOME - A distributed computing system based on BOINC.

[24] William Gropp, Ewing Lusk, Nathan Doss and Anthony Skjellum. Hhigh-performance, portable implementation of the MPI message passing interface standard

[25] David P. Anderson. BOINC: A System for Public-Resource Computing and Storage.

[26] Robert D. Blumofe ,Christopher F. Joerg ,Bradley C. Kuszmaul, Charles E. Leiserson ,Keith H. Randall and Yuli Zhou.Cilk: An Efficient Multithreaded Runtime System

[27] Enric Tejedor ,Montse Farreras,David Grove, Rosa M. Badia, Gheorghe Almasi and Jesus Labarta.ClusterSs: A Task-Based Programming Model for Clusters

[28] David P. Anderson,Eric Korpela and Rom Walton.High-Performance Task Distribution for Volunteer Computing.

[29] David P. Anderson,Carl Christensen and Bruce Allen.Designing a Runtime System for Volunteer Computing.

[30] J. Dean and S. Ghemawat.MapReduce: Simplified Data Processing on Large Clusters.

[31] Pieter Bellens,Josep M. Perez,Rosa M. Badia and Jesus Labarta.CellSs: a Programming Model for the Cell BE Architecture.

[32] Christian Benjamin Ries. UML for BOINC: A Modelling Language Approach for the Development of Distributed Applications based on the Berkeley Open Infrastructure for Network Computing.

[33] Apache Hadoop. http://hadoop.apache.org/

[34] Konstantin Shvachko, Hairong Kuang, Sanjay Radia,and Robert Chansler. The Hadoop Distributed File System.

[35] https://en.wikipedia.org/wiki/Tomasulo_algorithm

[36] Trilce Estrada,Michela Taufer and David P. Anderson.Performance Prediction and Analysis of BOINC Projects:An Empirical Study with EmBOINC.

[37] Gary A. McGilvar, Adam Barker, Ashley Lloyd and Malcolm Atkinson.V-BOINC: The Virtualization of BOINC

[38] Gilles Fedak,C´ecile Germain,Vincent N´eri and Franck Cappello.XtremWeb : A Generic Global Computing System.

[39]Michael J.Litzkow,Miron Livny and Matt W. Mutka. Condor –A Hunter of Idle Worksatatios

[40] Andrew A. Chien.Architecture of a Commercial Enterprise Desktop Grid: The Entropia System

[41] David P. Anderson Space Sciences Laborator. Volunteer computing: the ultimate cloud.

[42] Rosa M. Badia.Easy Programming Heterogeneous systems

[43] https://www.sics.se/~kff/wool/

[44] http://www.desktopgrid.hu/index.php?page=24

[45] Andrew S. Tanenbaum, Maarten van Steen. DISTRIBUTED SYSTEMS : Principles and Paradigms

[46] Zhenyu Gao, Boeke J. van Beugen & Chris I. De Zeeuw, Distributed synergistic plasticity and cerebellar learning, Nature Reviews Neuroscience 13, 619-635 (September 2012)

[47] P. Bazzigaluppi, J. R. De Gruijl, R. S. Van Der Giessen, S. Khosrovani, C. I. De Zeeuw, and M. T. G. De Jeu. Olivary subthreshold oscillations and burst activity revisited. *Frontiers in Neural Circuits*, 6(91), 2012.

[48] National Academy of Engineering Grand Challenges for Engineering, 2010.