DEVELOPMENT OF A GRID-AWARE MASTER WORKER FRAMEWORK FOR ARTIFICIAL EVOLUTION

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

AHMET KETENCİ

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
COMPUTER ENGINEERING

DECEMBER 2010

Approval of the thesis:

DEVELOPMENT OF A GRID-AWARE MASTER WORKER FRAMEWORK FOR ARTIFICIAL EVOLUTION

submitted by **AHMET KETENCİ** in partial fulfillment of the requirements for the degree of **Master of Science in Computer Engineering Department, Middle East Technical University** by,

Prof. Dr. Canan Özgen Dean, Graduate School of **Natural and Applied Sciences**	________________
Prof. Dr. Adnan Yazıcı Head of Department, **Computer Engineering**	________________
Dr. Cevat Şener Supervisor, **Computer Engineering Dept., METU**	________________

Examining Committee Members:

Prof. Dr. Göktürk Üçoluk Computer Engineering Dept., METU	________________
Dr. Cevat Şener Computer Engineering Dept., METU	________________
Asst. Prof. Dr. Erol Şahin Computer Engineering Dept., METU	________________
Asst. Prof. Dr. Tuğba Taşkaya Temizel Information Systems Dept., METU	________________
Dr. Onur Tolga Şehitoğlu Computer Engineering Dept., METU	________________

Date: ________________

ABSTRACT

DEVELOPMENT OF A GRID-AWARE MASTER WORKER FRAMEWORK FOR ARTIFICIAL EVOLUTION

Ketenci, Ahmet
M.Sc., Department of Computer Engineering
Supervisor : Dr. Cevat Şener

December 2010, 66 pages

Genetic Algorithm (GA) has become a very popular tool for various kinds of problems, including optimization problems with wider search spaces. Grid search techniques are usually not feasible or ineffective at finding a solution, which is good enough. The most computationally intensive component of GA is the calculation of the goodness (fitness) of candidate solutions. However, since the fitness calculation of each individual does not depend each other, this process can be parallelized easily.

The easiest way to reach high amounts of computational power is using grid. Grids are composed of multiple clusters, thus they can offer much more resources than a single cluster. On the other hand, grid may not be the easiest environment to develop parallel programs, because of the lack of tools or libraries that can be used for communication among the processes.

In this work, we introduce a new framework, GridAE, for GA applications. GridAE uses the master worker model for parallelization and offers a GA library to users. It also abstracts the message passing process from users. Moreover, it has both command line interface and web interface for job management. These properties makes the framework more usable for developers even with limited parallel programming or grid computing experience. The performance of GridAE is tested with a shape optimization problem and results show that the framework is more convenient to problems with crowded populations.

Keywords: Genetic Algorithms, Grid Computing, Master Worker Model

ÖZ

YAPAY EVRİM İÇİN GRİD TABANLI USTA İŞÇİ ORTAMI GELİŞTİRİLMESİ

Ketenci, Ahmet
Yüksek Lisans, Bilgisayar Mühendisliği Bölümü
Tez Yöneticisi : Dr. Cevat Şener

Aralık 2010, 66 sayfa

Genetik algoritmalar (GA), geniş arama uzayları olan optimizasyon problemleri de dahil birçok çeşit problem için çok popüler bir araç halini aldı. Izgara arama yöntemleri, yeterince iyi bir sonuç bulmada genellikle uygun yada verimli değildir. GA'nın en yoğun hesaplama gereken parçası aday çözümlerin iyiliğinin hesaplanmasıdır. Ancak, her bir bireyin iyiliğinin hesaplanması diğerlerinden bağımsız olduğundan, kolaylıkla paralelleştirilebilir.

Yüksek miktarda hesaplama gücüne ulaşmanın en kolay yolu gridi kullanmaktır. Gridler birçok bilgisayar kümesinin birleşiminden oluştuğu için, tek bir bilgisayar kümesinden çok daha fazla kaynak sunarlar. Öte yandan, grid, koşut programlar geliştirmek için, işlemler arasında haberleşmede kullanılabilecek araç veya kütüphane eksikliği sebebiyle, en kolay ortam olmayabilir.

Bu çalışmada GA uygulamaları için GridAE adında yeni bir uygulama geliştirme ortamı sunuyoruz. GridAE koşutlaştırma için usta işçi modelini kullanıyor ve kullanıcılara bir GA kütüphanesi sunuyor. Ayrıca, mesaj geçiş işlemini de kullanıcıdan soyutluyor. Bununla birlikte, iş yönetimi için hem komut satırı hem de ağ arayüzlerine sahiptir. Ortamın bu özellikleri onu, hiç koşut programlama yada grid hesaplama tecrübesi olmayan kullanıcılar için bile daha kullanışlı kılıyor. GridAE'nin performansı bir şekil iyileştirme problemi ile denendi ve sonuçlar gösteriyor ki ortam, kalabalık popülasyonlu problemlere daha uygundur.

Anahtar Kelimeler: Genetik Algoritma, Grid Hesaplama, Usta İşçi Modeli

aileme...

ACKNOWLEDGMENTS

First of all, I would like to thank my supervisor, Dr. Cevat Şener who always believed in me, sometimes even more than I believe in myself. I have managed to overcome many obstacles thanks to his guidance.

I am more than grateful to my friends Çelebi, Serter, Ferhat, Gökcan and Dilan who gave their hands without asking. I could not have finished this thesis without you.

I should also thank all the staff of the Computer Engineering Department of METU for their enjoyable company during my 8 years spent there and TR-GRID staff of TÜBİTAK ULAKBİM who have managed to find answers to all of my questions.

This work is funded by SEE-GRID2 project which is a part of European Union's Framework Project 6.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ALGORITHMS

CHAPTER 1

INTRODUCTION

The need for computational power is constantly increasing and computer clusters are becoming more affordable for research institutions. Therefore, parallel programming is gaining more importance and attraction. There are many standards, tools and libraries available for parallel application developers. These provide higher level methods for communication among parallel threads and/or processes.

There exist various problems where a single cluster sometimes does not provide necessary computational power. Therefore, the number of parallel processes that should be executed for these problems, is generally quite high. On the other hand, the number of processors that a single user can use simultaneously is limited. For some problems, where it is very difficult (if not impossible) to find the best possible solution, using higher number of processors increases the quality of solution. Therefore, computer grids are significantly more suitable than clusters for these kind of problems.

Computational grids are formed by multiple clusters which are connected by high speed networking means. A computer grid has three major components, namely computing elements (CEs), storage elements (SEs) and user interface machines (UIs). CEs are the components that carry out the computations. Each CE can be considered as a cluster in the grid. They have a master node that accepts jobs and distributes them to the worker nodes. Moreover, grids are used for sharing large amounts of data. Researchers working on the same subject can store their data on SEs and use simultaneously. The storage of data produced by Large Hadron Collider (LHC) of CERN (European Organization for Nuclear Research) is an example of this kind of usage. The collider will produce 15 petabytes of data that scientists can reach every year [4].

In order to use the grid resources, one should acquire a grid certificate from a Certificate Authority and have a user account on a grid UI machine. From a UI, the user can submit a job on a CE or upload a file to a SE. A job consists of an executable and a file which contains information about the job, written in Job Description Language (JDL). The job management is done through the middleware which is a special software installed on grid components managing the coordination of these resources which are usually heterogeneous.

Genetic Algorithm (GA) is a search technique inspired by Darwinian evolution. Natural selection and survival of the fittest paradigms are used in the algorithm. Candidate solutions of a problem constitute a population, each of them being an individual in it. The solution is encoded in a data structure called the chromosome. And, each meaningful component of the chromosome is called a gene. For instance, if we have an optimization problem at hand, a set of parameters constitutes a chromosome and each parameter is a gene in that chromosome.

The algorithm starts with generating the first set of individuals which forms the first generation randomly, which is called initialization of the population. The goodness (fitness) of these candidate solutions (individuals) are measured with a fitness function which usually generates a numeric value for each individual, so that they can be compared. Obviously, the fitness function is always specific to the problem that the algorithm deals with. With the fitness values at hand, some of the individuals are selected to be the parents of the next generation. The selection can be performed in many ways, but it should favor the better individuals, because better individuals are more likely to produce better offsprings. The reproduction is performed by exchanging genes between chromosome pairs, and this exchange is called crossover. After the crossover, some of the genes of the offsprings can be modified randomly by simulating mutation. Like selection, crossover and mutation can be performed in many different ways. The offsprings constitute the next generation which is to be evaluated by the fitness function. This process continues until a predefined desired fitness value or a specific generation count is met.

Depending on the problem, the fitness calculation of an individual may take considerable amount of time. Most of the real world search problems fall into this category. This is one of the reasons that GA is preferred to grid search techniques. Moreover, since these calculations are not dependent on each other, the fitness calculation can be performed parallel on multiple processors .

There are three common parallelization models in GA. The first one is called the island model. The population is divided into subpopulations each of which is assigned to a different processor and these subpopulations evolve independently for the island model. However, there exist transfers of individuals between subpopulations during the evolution. The second one is the master worker model. In this model there is a single population and the master process distributes the individuals to workers at each generation. Then, the workers calculate the fitnesses and return back the results. The final model is a hybrid of the first two in which each subpopulation uses master worker model for fitness calculations. The island model is more suitable to parallel programming since the communication among the processes is limited. However, this model does not give the best results for all problems.

The research on parallel GA can be classified in three categories according to the architecture of the computational resource and network connecting them. Peer to peer (P2P) parallel GA usually uses dedicated processing power of personal computers. The Internet is used as the network channel for communication. Others use a single cluster to achieve higher speeds for communication. There are well known GA frameworks that can run on clusters like Distributed BEAGLE [17] and Parallel GAlib [1]. There are also examples on grid, which are generally implemented to solve specific problems and require a software to be installed on CEs. Moreover, researchers usually form their own grids from clusters that are dedicated to them.

As stated above, there are good parallel GA frameworks for cluster computing. However, for many real life problems, resources that a single cluster can offer to a single user may not be enough. Moreover, the amount of computational power can be very important for the quality of the solution where grid is a better alternative for these kind of problems.

There are two major problems that stand between the application developers and grid. The first one is the difficulty of parallel programming. The management of the communication between the processes can be very difficult, bothersome and time consuming especially for the developers who are inexperienced in parallel programming. This process is even more difficult on grid since there are not any tools or libraries for the interprocess communication. The second factor is the unfamiliarity with grid. In order to use grid resources, the developers should be familiar with the interface of the middleware and the JDL.

With this motivation, a C++ framework, GridAE, is developed aiming to eliminate these factors and make grid more reachable for GA developers. The framework adapts master worker model and uses the CEs to run the workers. The resources of South East European Grid (SEE-GRID) and Turkish Grid (TR-Grid) are used. GridAE offers both command line interface (CLI) and web interface ported on P-GRADE Portal.

In GridAE, the master initializes the population, distributes the individuals as tasks to workers, performs genetic operations (selection, crossover and mutation) and stops evolution according to the criteria specified by the user. The workers are only responsible for the fitness calculation of the individuals they receive. They continue waiting tasks until master instructs them to stop. Users are provided with high level functions that carry out the communication process which is performed by means of TCP/IP sockets, internally. The interfaces are designed to address the issues of users unfamiliar to grid computing and eliminate the need to communicate with the middleware.

GridAE also offers a GA library that includes common genetic operations (roulette-wheel selection, rank selection and tournament selection as selection methods, one point, two point and uniform crossover as crossover methods and finally point and swap mutation as mutation methods) and supports array chromosomes of integer and real number genes. The users can also implement and use their own operators. If these operators are suitable for the application of the user, then the only thing user should provide is the fitness function.

The user should provide two source files to GridAE written in C++ for the master and the worker processes. In the master's source file, the type of chromosome should be specified and the population should be initialized. The functions that GridAE is going to use for selection, crossover and mutation should be set and *evolve* method of the library has to be called in the main function of the master. Furthermore, the worker's source code has to contain the fitness function which should be specified in the main, before the *do_calculate* method of the library is called. Besides, the framework uses a parameter file to configure the GA which contains parameters like population size, target fitness and worker number that should also be provided by the user. The details about these three files will be given in the Chapter 3.

This chapter will be followed by background and literature survey in which more details about GA, parallel programming, grid computing and P-GRADE Portal will be given in addition to selected works on parallel GA. The next chapter introduces the framework in detail with its features, architecture and usage. After that, a chapter is reserved for implementation details like the class hierarchy of the GridAE library and implementation approaches for the communication process. The following chapter explains a missile shape optimization problem which is used as a test case for GridAE. The problem, its solution and results are discussed in that chapter. Finally, the last chapter concludes this thesis.

CHAPTER 2

BACKGROUND AND LITERATURE SURVEY

Nowadays, most of the engineering problems are stated in the form of an optimization problem. Therefore, lots of optimization algorithms take place to solve these optimization problems. Genetic algorithm is one of the most popular stochastic optimization algorithms. The most important property of GA is to use wide search space instead of the deterministic optimization techniques. In this chapter information on genetic algorithms, parallel programming and grid computing are briefly introduced. In addition to that, the usage and general description of P-GRADE Portal also takes place here, since web interface of GridAE is ported on P-GRADE Portal. At the end of this chapter some of the selected examples on literature about parallel genetic algorithms are presented. Overall, this chapter is intended to help the reader to get a better understanding of this thesis.

2.1 GENETIC ALGORITHMS

Genetic algorithms (GA), introduced in 1975 by Holland [19], is a member of evolutionary algorithms class in which natural evolution paradigm is adapted to computation. Concepts like natural selection, crossover (exchanging genes) and mutation are all simulated and applied over some data structures representing individuals. The algorithm is used in many different areas where the problem can be specified as an optimization or a search problem.

In GA, a candidate solution of the problem is generally represented with an array of numbers or a graph-like data structure which is called a chromosome. Initially, a collection of chromosomes is randomly generated and it forms the first generation of the so called population. Then, the population starts to evolve until the best individual(s) is suitable to be taken as the desired solution. At each generation the following steps are performed to build up the new generation. The goodness of each chromosome is calculated with the help of a fitness function which is specific to the problem and the individuals are sorted according to the fitness values. In order to produce the offsprings some of the individuals are selected and there are many selection techniques available. The most popular selection techniques are rank selection, tournament selection and roulette wheel selection. All of these techniques favor the better individuals like natural selection. From the selected two individuals, two new individuals are produced with crossover and mutation operations. Crossover is performed between the individuals and involves exchanging respective parts of their chromosomes. On the other hand, mutation operation is applied on a single individual and is usually less destructive, like flipping a value in a boolean array or swapping two values in a permutation array. The aim of applying crossover and mutation is to enlarge the search space. Both of these operations are carried out in predefined probabilities.

It was not Holland who first thought of adapting natural evaluation to computation. The idea first appeared during the late 1950's and early 1960's [9] [10]. In these years, this idea was used to solve optimization problems by researchers.

In 1965, German computer scientist Ingo Rechenberg developed a method and named it as ”evolution strategies” to solve aerodynamic design problems [27]. In this method, he used only one candidate solution in the population and performed only a mutation operator on this candidate and kept the better one (among the original and mutated ones) for the next generation.

The next important step came from Fogel, Owens, and Walsh (1966) [13]. They introduced the concept of evolving computer programs. Individuals were finite state machines and they were mutated to generate new individuals. This technique was called ”evolutionary programming”. Again, at the beginning, the population consisted of a single individual. Today, both evolution strategies and evolutionary programming are active research areas.

The introduction of crossover as a genetic operation along with mutation began with Holland and in his book, ”Adaptation in Natural and Artificial Systems”, he also developed a theoretical basis with schemata theory for genetic algorithms. Therefore, he is referred as the inventor of genetic algorithms [24]. After that, genetic algorithms are used in a wide range of areas and proved to be very powerful.

To solve a problem using genetic algorithms, the following items should be stated:

- the representation of the candidate solutions
- the selection method
- the crossover method and the crossover percentage
- the mutation method and the mutation percentage

The following subsections give preliminary information about determination of the items above.

2.1.1 REPRESENTATION OF INDIVIDUALS

Representation of individuals in genetic algorithms is a very crucial step. Different choices may lead to unexpected success rates. The genotype of an individual is stored in its chromosome. Each piece of information of chromosome comes from a gene. Therefore one should first decide what a gene in the problem is. For instance, each parameter should be a gene for an optimization problem. Numeric values can be used to show each gene, thus a floating point variable in the problem can be assigned to represent this gene.

Once the representation of genes is determined, the data structure to represent the chromosome which holds the gene should be determined. Again, for an optimization problem each gene is represented with a floating point number, thus a floating point array is probably a good choice. However, representation of genes and chromosomes is not always easy like this, therefore it is better to introduce the most common examples before deciding on one, starting with the following paragraph.

Bit strings

Representing individuals with binary encoding is the simplest of all representation methods. Holland used this representation for developing the theory behind genetic algorithms [24]. Most of the well known genetic operators (crossover and mutation) are convenient with this representation. Because of these, bit strings are very suitable for educational purposes. On the other hand, for many problems, this representation is inadequate. An example of a bit string chromosome can be seen in Figure 2.1.

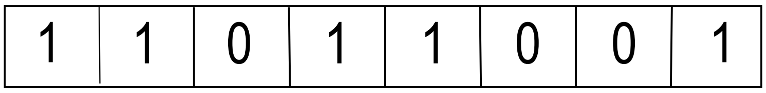


Figure 2.1: An 8-gene bitstring chromosome

Integers and floating point numbers

Most optimization problems are focused on finding the best values of the variables which usually take integer or real values. Therefore bitstrings are not suitable for this kind of problems. An array of numbers can represent an individual, where each gene can be integer or a floating point number. Each gene should be able to take their values from their own range of numbers. Figure 2.2 demonstrates this kind of chromosome. Range of each gene also has to be restricted, because it is crucial for the mutation operation. For the example chromosome in Figure 2.2, range of each gene can be determined as in the Table 2.1.

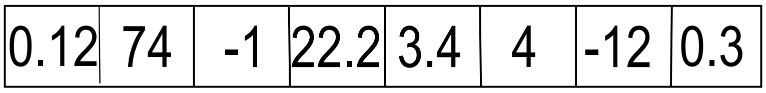


Figure 2.2: An 8-gene chromosome consisting of integers and floating point numbers

Table 2.1: The minimum and maximum values for each gene

Gene	*min*	*max*
Gene 1	0	1
Gene 2	1	100
Gene 3	-1	1
Gene 4	10.5	24.2
Gene 5	2	4.5
Gene 6	1	12
Gene 7	-100	0
Gene 8	0	1

Other data structures

Sometimes candidate solutions are more structured in their nature to be represented by pure numeric values. Evolving computer programs is an example of that kind of problem, for which representing an individual with a tree like data structure is more convenient. The genetic operators have to be selected properly for the representation.

The theoretical work is mainly on the representation types mentioned here. Also, many well known genetic operators are suitable for these too. Yet, each particular problem may have its unique representation which will be easier to adopt and yield better results.

2.1.2 SELECTION METHODS

During the evolution, each generation will leave its place to the next one. This takes place by generating offsprings from the current population. One should decide which individuals are going to be used for this process. The natural evolution offers to use the fittest individuals as much as possible. On the other hand, this may decrease the diversity and the population may stick at a local maxima. Therefore, the selection method of the individuals has to be determined carefully. The popular selection methods are briefly discussed below.

Roulette-wheel selection and stochastic universal sampling

In roulette-wheel selection method, probability of an individual to be selected is proportional to its fitness. The roulette wheel is divided into pieces. The number of the pieces is equal to the population size, and each of them represents an individual. Though, unlike a fair roulette wheel, the sizes of these pieces are proportional to the fitnesses of the individuals. Hence, the individual with the best fitness value has the biggest piece on the wheel and a higher probability to be selected.

Although roulette-wheel selection seems to be fair and solid, a problem may be encountered. After an unlucky set of spins of the wheel, the selected individuals may be the ones with worse fitness values. This more likely takes place when the population size is relatively small. Stochastic universal sampling is introduced to overcome this problem. The wheel only spins once in the extended method but there are multiple pointers to select the individuals. The number of pointers is equal to the number of desired individuals to be selected, and the pointers are evenly placed on the wheel. With this method, actual selection frequency of an individual is nearer to its expected value, compared to the roulette-wheel selection.

Rank selection

Rank selection method is similar to roulette-wheel selection except that the selection probabilities of individuals are proportional to their rank among the others in the generation rather than their pure fitness values. Therefore the differences between fitness values of individuals do not matter.

The advantage of this method is that it does not let extremely better individuals dominate the next generation in the early generations and lead to premature convergence. Moreover, in the later generations, the differences between the fitness values of individuals may decrease. When using roulette-wheel selection, this situation decreases the effect of better individuals. Conversely, rank selection preserves importance of better individuals.

Tournament selection

Tournament selection method is quite different from the methods mentioned up to now. In this method n (called *tournament size*) individuals are randomly selected from the population. The best one according to their fitness values is selected to be a parent for the next generation. Increasing n will decrease the chance of bad individuals to be selected.

There is also an alternative implementation of this algorithm, where $n = 2$ and another parameter (say k) is introduced. k should be chosen between 0 and 1. For each tournament a random number which is also between 0 and 1 is taken. If this number is smaller than k, the better of the individuals is selected, and otherwise the worse one. Setting $k = 1$, will lead to the same algorithm described above, where $n = 2$.

Elitism

Elitism is actually not a selection method by itself. However, it can be used with selection methods to improve the quality of the next generation. Elitism lets some of the best individuals be carried to the next one without being subjected to crossover or mutation. Although elitism improves the performance for most of the problems, the amount of individuals that will be carried the next generation should be decided carefully. Too much elite individuals may decrease diversity.

2.1.3 CROSSOVER METHODS

After selection of the individuals, the next generation should be formed by reproduction. The individuals are coupled to be the parents of the new offsprings. The offsprings take their chromosomes from their parents. The parents exchange some parts of their chromosomes to produce new chromosomes. This process is called crossover. With crossover, new individuals are created without losing the genes yielding better fitness values of the selected individuals. Therefore, it is one of the most important parts of genetic algorithm.

There are many ways to perform crossover. The most popular methods on array like chromosomes are one point crossover, two point crossover and uniform crossover.

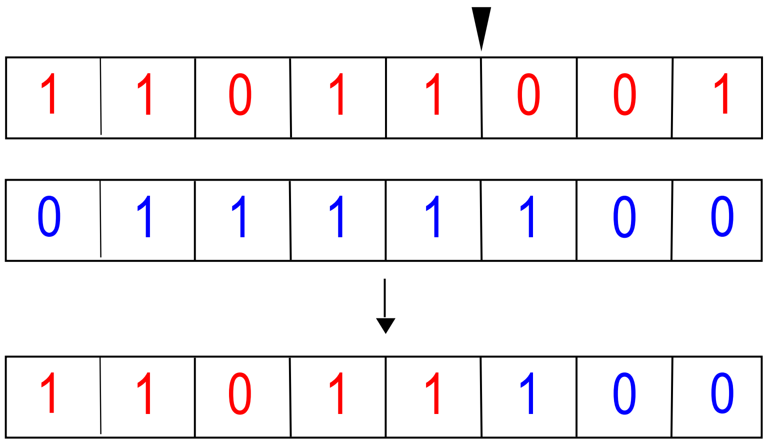


Figure 2.3: One point crossover

In one point crossover, position of a gene on the chromosome is randomly selected and the offspring gets the genes before that position from one of the parent and the rest come from the second one as shown in Figure 2.3. The number of random positions is two for two point crossover. The offspring inherits the genes between these positions from one of the parent and the rest from the second one (Figure 2.4). Finally, uniform crossover takes the genes with odd numbered position from one of the parent and even numbered from the other one as illustrated in Figure 2.5.

If the chromosome is encoded with a tree structure, then the methods mentioned up to now will not work. One of the common crossover methods for trees is exchanging branches. Two

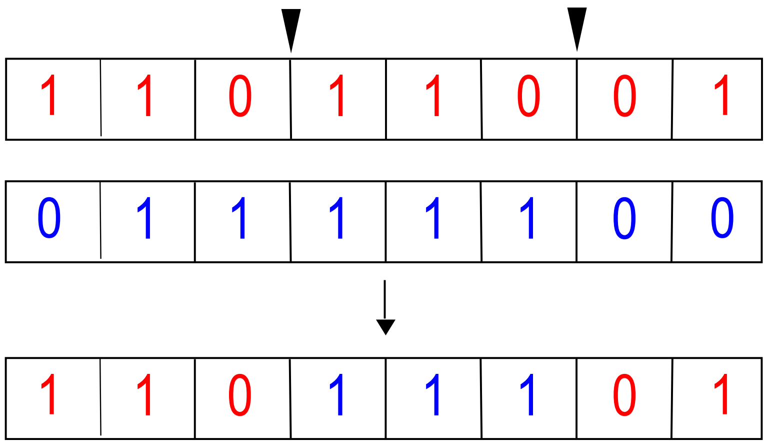


Figure 2.4: Two point crossover

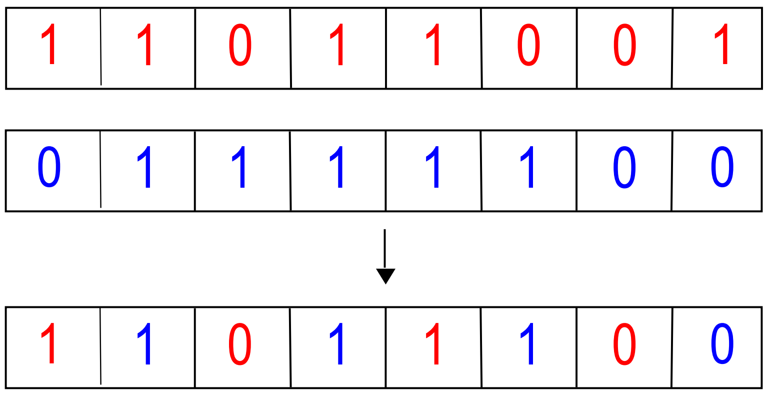


Figure 2.5: Uniform crossover

random branches are selected from the parents and one of them is replaced with the other one as shown in Figure 2.6 to form the new individual.

The crossover method has a significant effect on the performance of the genetic algorithm. However, it is not easy to determine which method gives better performance. Different methods may lead different results for a problem, even if the same representation is used. It may be helpful to observe the evolution in the early generations with different methods and decide which method will be used afterwards, if the whole process takes too much time.

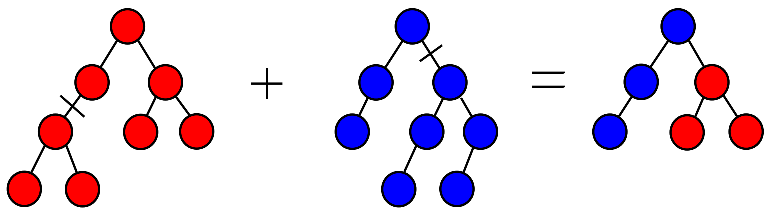


Figure 2.6: A crossover method on trees

2.1.4 MUTATION METHODS

Using crossover as the only genetic operation is usually not enough. Most of the crossover methods only change the gene combinations. If the initial population does not have a particular value on a gene, then it is not possible to see this value on the final solution. Mutation operation lets the algorithm to introduce new genes to the evolution process. It also decreases the probability of being stuck around a local maximum with jumps to another place on the search space.

Mutation is performed on a single chromosome and usually affects a small number of genes. It can be done in different ways on different chromosome representations. On a bitstring, inverting the value of a randomly selected gene is the most common way as shown in Figure 2.7. For integer and floating point array representations, the value of a gene can be changed with another randomly generated number.

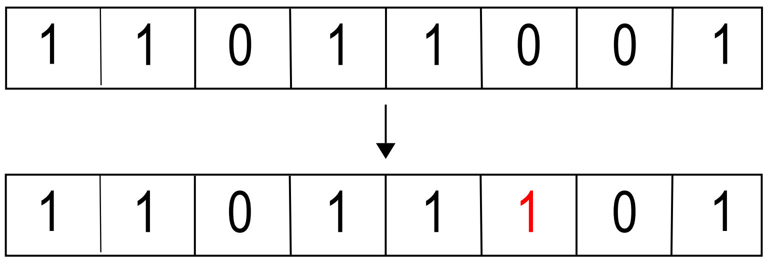


Figure 2.7: Point mutation

2.2 PARALLEL PROGRAMMING

In order to shorten the running time of a program, one should either use a faster processor or multiple processors simultaneously. Multiple processors can only be used, when the running program is designed and implemented by taking this fact into consideration. Therefore, it seems easier to run a program on a faster computer but this is not the case. The technological development of microprocessors is hindered by physical limitations such as the sizes of transistors and other electronic units and overheating. On the other hand, multi-core technology progressed rapidly, even more personal computers include two or four core microprocessors nowadays.

Since it is getting more affordable to acquire a computer cluster, the importance of parallel programming is increasing. The libraries, tools and APIs are developed in order to help users parallelize their programs. Some of them, such OpenMP and POSIX threads, are useful for thread level parallelism to utilize multiple cores of a CPU. Other libraries like PVM and MPI are developed to benefit from multiple CPUs on different computers within the same network.

There are mainly two different parallelization types, namely data parallelism and task parallelism. In data parallelism, data is fragmented and distributed to the processors to be processed. The parallel processes usually do the same thing on the data they receive. Therefore, this type of parallelism is relatively easier to implement. A GA implementation which distributes the fitness calculation of individuals is a good example of data parallelism. In order to achieve task parallelism, the program should be divided into subtasks which does not depend on the results of each other; but it is usually very difficult (if not impossible) to achieve this.

When discussing the performance of a parallel program, two concepts are usually referred. First one is speedup which is defined by the following formula:

$$S_n = \frac{T_1}{T_n} \tag{2.1}$$

where n is the number of processors, T_1 is the running time of program on a single processor and T_n is the running time of the program with n processors. It is desirable to achieve $S_n = n$ as n increases, which is called *linear speedup* but it is a very difficult goal to achieve. The second one is efficiency which is given by:

$$E_n = \frac{S_n}{n} \tag{2.2}$$

where n is again the number of processors and S_n is the speedup with n processors. Efficiency is between 0 and 1 and gives an idea about the utilization of the processors.

2.3 GRID COMPUTING

In this section, an introduction to grid computing is given. The grid structure is summarized with additional information. The usage of grid is also addressed in this section.

2.3.1 DEFINITION AND GENERAL INFORMATION

Computer clusters are formed by connecting computers using a high speed network. These computers are very close to each other, usually in the same room. A cluster has a single interface for its users. Through that interface, it operates like a single computer with many processors. A computer grid can simply be described as a cluster of clusters. It is constituted by geographically distributed computer clusters and a network connecting them. A more formal definition of grid comes from Kesselman and Foster [23]: ”A computational grid is a hardware and software infrastructure that provides dependable, consistent, pervasive, and inexpensive access to high-end computational capabilities.”

Foster stated a three point checklist to clarify which computing systems should be called a grid system and which ones should not [15]. According to that checklist a grid system should have the following properties:

- The administration of the resources are distributed which distinguishes grid from local management systems.
- Standard and open protocols are used which distinguishes grid from the systems that are specific to an application.
- Nontrivial qualities of service is delivered.

Middlewares are special softwares used in grid computing that aim to maintain the consistency among the resources that are heterogeneous and managed by different local organizations. They also provide user interfaces for job and data management. Globus Toolkit, gLite, UNICORE and ARC are major grid middlewares. Grid systems also include information services and resource brokers. The resources of grid are registered to grid information services and grid resource brokers use these information services in order to answer the users' needs when they submit a job with requirements on resources.

The grid structure has three main components. First one is the computing elements (CE). CEs accept jobs from the job manager and runs the jobs on its working nodes (WN). A CE can be identified as a cluster which is a part of grid. The second one is the storage elements (SE). SEs have large amounts of disk space and store data of the users. SEs allow grid to be used not only for computational purposes, but also for sharing large amounts of data. Multiple users can have permission to access the data stored by a user. Thus with the help of the grid, researchers from different locations can work on the same data together. The last one is the user interface (UI) machines where grid users have accounts on. The users can connect to these machines remotely and carry out job and data management.

Grids allow many computationally complex problems in different scientific areas including medicine, astronomy, economics and physics to be solved. Current projects include BIRN (Biomedical Informatics Research Network) which aims to improve the diagnosis and treatment of human diseases like schizophrenia, depression and brain cancer [22]. There also exist large scaled projects in order to predict climate changes, better understand the magnetic fusion and simulate earthquakes.

This project was a part of SEE-GRID2 (South East European Grid) project [6]. Therefore, during every phase of this research, SEE-GRID resources are used. SEE-GRID2 has eleven partners, including, Greek Research and Technology Network (GRNET), European Organization for Nuclear Research (CERN), Institute for Parallel Processing (IPP, Bulgaria), Roder Boskovic Institute (RBI, Croatia), Faculty of Electrical Engineering Banja Luka (UoBL, Bosnia and Herzegovina), Parallel and Distributed Systems Laboratory (SZTAKI, Hungary), Academy of Sciences-Institute of Informatics and Applied Mathematics (ASA-INIMA, Albania), National Institute for Research and Development in Informatics (ICI Bucharest, Romania), Ss Cyril and Methodius University in Skopje (UKIM, Macedonia), Research and Educational Networking Association of Moldova (RENAM), University of Montenegro (UOM), University of Belgrade (UOB, Serbia) and The Scientific and Technological Research Council of Turkey - Turkish Academic Network and Information Center (TUBITAK ULAKBIM).

2.3.2 HOW TO USE GRID

The usage of grid can be described in two parts.

Preliminaries

First of all, the user should introduce himself/herself to the grid community, and prove that he/she is who he/she claims to be to a certification authority (CA). TUBITAK ULAKBIM is one of these authorities. The full list of CAs can be found here [7]. The user will be given an X.509 certificate by the CA. The certificate is usually installed the user's web browser and it should be exported. The exported certificate usually has *.p12* extension. From this file the user should generate two files (namely *usercert.pem* and *userkey.pem*) using *openssl* [5].

The next step is acquiring a user account from a grid UI machine. When declaring the request to use grid, the administrators guide the prospective user on how to get an account. After getting the account, the files *usercert.pem* and *userkey.pem* should be transferred to the UI machine under the *.globus* directory which is in the home directory of the user.

After that step, the user should get a membership from a *virtual organization (VO)*. A VO is actually representing a group of people or institutions sharing resources of grid. These resources are not only computers but also can be software, data, file and other resources. The resource sharing has to be defined clearly between the members of the VO and controlled strictly. The things that can be shared and by whom have to be determined. The purpose of this sharing and constitution of a VO is usually to solve a problem emerged in industry, science or engineering [14]. By being a member of a VO, the user can use the resources reserved for this VO. An example of VOs is the SEE-GRID VO which includes the developers of the projects that are part of the SEE-GRID2 project.

Finally, a user proxy should be created on the UI machine with *voms-proxy-init* command. Although the certificates usually have one year lifetime, the proxies last for twelve hours and should be renewed when it dies.

Job management

In order to run a job on a CE of the grid, first of all a file that includes information about the job has to be prepared. This file has its own syntax and called JDL (Job Description Language) file. It has *.jdl* extension and is filled with the lines in the following format:

```
attribute = expression;
```

Here is an example of a jdl file:

```
Executable = "myprog.sh";
Arguments = "inputfile 10";
StdOutput = "out";
StdError = "err";
InputSandbox = {"myprog.sh", "inputfile"};
OutputSandbox = {"out", "err"};
```

From this JDL file, it can be understood that the files that are going to be transferred to the CE are *myprog.sh* and *inputfile* and the executable is *myprog.sh* and it will be run with two arguments, *inputfile* and *10*. Moreover, it is indicated that the standard output and error streams will be redirected to files named *out* and *err* respectively and they should be transferred back to the UI machine. Some of the most common JDL attributes can be found in the Table 2.2. For a complete list and more information please see here [2].

Once the JDL file is ready, the job can be run on a CE. Grid middleware interfaces are used for job management and gLite middleware is installed on SEE-GRID. The gLite commands used for basic job management are detailed in Appendix A.

2.4 P-GRADE PORTAL

P-GRADE Portal is a web based framework for job and data management on grid [21]. The user can also relate the jobs to each other by defining workflows. It is a free tool to use but the user should have an account on the portal. The portal has a user-friendly GUI developed with Java. Before submitting a job, the certificate files of the user have to be uploaded to a

Table 2.2: Most common JDL attributes and their meanings

Attribute	Meaning
Executable	The name of the executable
Arguments	The arguments of the executable
InputSandbox	The files that will be sent to CE
OutputSandbox	The files that will be received by UI
StdOutput	The name of the output file
StdError	The name of the error file
Environment	The environment variables and their values
VirtualOrganisation	The VO of the user
Requirements	Specifies the attributes that the CE should sustain
RetryCount	The number of times that the job should be resubmitted if it fails after starting running
ShallowRetryCount	The number of times that the job should be resubmitted if it fails before starting running

myproxy server and a proxy has to be created through the interface. Certificate files should be ready on the local machine of the user and can be uploaded in the following way:

1. Go to *Certificates* tab
2. Click the *Upload* button
3. Click *Browse*
4. Find your *userkey.pem* file
5. Write your certificate password
6. Click *Browse* again
7. Find your *usercert.pem* file

A proxy can be created from the same tab with clicking *Download* button. The certificate files will be on the server until their lifetime expires. Therefore, file uploading is only required when the portal is used for the first time.

2.5 PARALLEL GA

Parallel GA is classified into three main categories [11]:

Single Population Master Worker GA

In this type of GA, there is only one population and all individuals are in it. The parallelism is used for fitness calculations of individuals. These calculations are carried out by the workers and selection and recombination are performed by the master. The master is also responsible for task distribution.

Single Population Fine Grained GA

This type of GA is mostly suited for massively parallel computers. Each processor usually has one individual and selection and recombination operations are performed with the neighbor individuals to achieve higher speed.

Multi Population (Deme) Coarse Grained GA (Island Model)

The population is divided into subpopulations (demes) in this type of GA and each subpopulation is evolved on a single processor. The subpopulations usually exchange individuals which is called *migration*. This type is the most popular among researchers, mostly because of the fact that the *computation/communication* ratio is bigger in this type of GA than the other types.

The examples of parallel GA applications can be classified in three categories according to the distribution types of the resources used: (1) Peer to peer (P2P), (2) Cluster and (3) Grid.

2.5.1 EXAMPLES OF PARALLEL GA WITH P2P COMPUTING

There are two notable examples in this category. Both of them are frameworks for distributed evolutionary algorithms. The first one is named DREAM (Distributed Resource Evolutionary Algorithm Machine) [8]. The framework is placed on another framework designed for distributed applications [20] which uses P2P computing methodology to distribute the computation. It offers a graphical user interface (GUI) to its inexperienced users. With the help of this interface the users specify the properties of their GA program and then the framework transforms this information into Java codes. It also lets experienced Java programmers to implement their GA in Java using offered libraries.

The second example is named Paladin-DEC (Distributed Evolutionary Computing) and uses the Internet as the communication mean [28]. The software is Java based and uses Java Message Service (JMS) for information exchanging and Java Genetic Programming (JGProg) Package. The island model is used and communication is required during migrations.

One of the drawbacks of these P2P frameworks is that they need other users to donate their computational power to the ones who need it. Moreover, the denoted computational power will mostly come from personal home / work computers with relatively slow CPUs and network connections compared to grid resources.

2.5.2 EXAMPLES OF PARALLEL GA ON A SINGLE CLUSTER

First example of this type of parallel GA is Distributed BEAGLE [17]. It is an extension to Open BEAGLE [16] framework. Open BEAGLE is a C++ evolutionary computation framework which offers many classes and methods for its users. Although the Distributed BEAGLE uses master slave paradigm, it implements the island model. There are two kinds of workers in the architecture: (1) evolver carries on the selection and reproduction process of a subpopulation. The number of evolvers is equal to the number of subpopulations. (2) Evaluator only performs fitness calculation of individuals. The number of evaluators are much more than the number of evolvers and can be up to the number of individuals in the population. The communication is handled with TCP/IP sockets and the framework performs well on Beowulf clusters.

Another example comes with GAlib (Genetic Algorithm Library) [1]. GAlib is a free C++ library that supports most of the common representation types and includes many genetic operators. It also allows users to define their own chromosome type and methods for selection, crossover and mutation. The library also supports parallel evolution with the help of PVM. Both the single population master worker model and the island model can be implemented. However, the parallel programing part is not abstracted from the user. Therefore, initialization and message passing have to be dealt with by the user using PVM function calls.

2.5.3 EXAMPLES OF PARALLEL GA ON GRID

Grid and Parallel GA were used together to solve different problems in the past. For instance, Nebro *et al.* used GA and solved large DNA fragment assembly problems in feasible times on grid [25]. A DNA sequence of 77292 base pairs and 773 fragments, successfully concatenated within hours with the software (GrAE) they developed. GrAE is a grid based software for DNA assembly problem. The researchers constructed a grid of up to 150 computers. The computers in the grid were owned by different people and they had Condor software [29] installed. Condor lets the owners to contribute to the grid when the CPUs of their computers are idle. The communication is handled by a library called Master Worker (MW) [18] This library is installed on top of Condor and offers message passing routines for the developers who use master worker paradigm for their applications.

Durille *et al.* formed a grid of 330 processors from the Computer Science Department of the University of Malaga to investigate the performances of different parallel GA approaches on multi objective problems [12]. MOPs do not have a simple unique fitness function and the goodness of individuals depends on multiple objective functions. A better individual according to one of the objectives may be worse than the others when other objectives are considered. Since the authors have administrative privileges on the computers, they could have installed their software on them. A software tool called Sporrow is used for task management. This tool is similar to MW except that it is based on Java.

In these two examples, the computers were direct members of the grid rather than being member of clusters that constitute the grid. However, in this final example, the grid is formed by clusters. Ng *et al.* developed a grid based parallel GA framework to solve aerodynamic airfoil design optimization problems which is based on Globus [26]. They have used the island model as the design pattern. The architecture of the framework can be summarized as follows: The subpopulations are assigned to clusters forming the grid. The server communicates to single nodes of each cluster on which selection and reproduction of subpopulations are carried out. These master nodes distribute the fitness evaluation task to the other nodes on the same cluster. Fitness evaluation is performed by software services that run on working nodes. These services are specific to the problem, therefore they should be provided by the users of the framework. Since there does not exist a connection among these master nodes, the migration occurs through the server. The authors had three clusters to use, provided by the Nanyang Technological University. The master nodes had an agent process running on them, which handled the job management and genetic operations.

These examples show that parallel GAs are suitable to grid and can be very efficient. However, in all of these examples, the researchers were able to manage the computers in the grid; therefore, they could install necessary software on them. When we look at the users' perspective, it is very difficult to find this many resources to set up a grid for a specific purpose.

CHAPTER 3

PROPOSED SOLUTION

In this chapter, GridAE is described in detail, beginning with its features. These features point out the differences between the most similar studies discussed in the previous chapter and GridAE. Furthermore, the architecture of the framework is detailed and the operation principle of GridAE is revealed with a section in this chapter. The last section of this chapter is dedicated for the prospective users, explaining the usage of framework with examples.

3.1 FEATURES

While most of the parallel GA applications and frameworks use the island model, GridAE uses the master worker model. The island model is usually preferred because of its higher *computation / communication* ratio. However, not all problems give good results with this model. Moreover, most of the serial GA applications use single population evolution. Adapting those applications to the island model requires additional effort. GridAE is suitable for these serial GA applications. Additionally, the framework also achieves good scalability in spite of using the master worker model (see Chapter 5).

GridAE library uses an array of genes to represent the chromosome of an individual. Each gene may take an integer or a real number value. The genes do not have to be the same type (integer or real number). The types of all genes and their minimum and maximum values should be specified by the user. This feature brings flexibility and allows many problems to be adopted without much effort.

The library also implements most of the common genetic operations, including roulette-wheel selection, rank selection and tournament selection as selection methods, one point, two point and uniform crossover as crossover methods and finally point and swap mutation as mutation methods. If these methods are not sufficient or not suitable for the user, new genetic operations can be implemented by the user.

Although parallel programming is gaining more and more popularity, it may still be difficult for most of the application developers. An important feature of GridAE is that it does not require parallel programming experience from its users. The communication layer is shielded from the users. The users only have to specify what their program is going to do with the data both master and worker receive.

GridAE is placed on top of glite middleware. The middleware has a command line interface as summarized in Chapter 2. Job management is not very complicated when dealing with a single job. However, it may be difficult to deal with multiple jobs simultaneously as GridAE requires. The framework also abstracts the job management process form the user with the help of its interfaces. The GridAE user does not need to know any of the glite commands. These properties allow GridAE to be used with ease.

Two of the examples discussed in the previous chapter, that are similar to GridAE with respect to the usage of grid are application specific softwares. Therefore, they are not genuine alternatives to GridAE. On the other hand, the last software mentioned, which is developed by Ng *et al.* [26], is a framework that uses grid architecture to solve GA problems. Its most distinguishing distinction from GridAE is that the island model is used for GA. The framework enforces that the fitness calculations have to be carried out by NetSolve software services that reside on computing nodes. Since this service has to be implemented or provided by the user, the users who are not very familiar with grid computing may find it difficult which is a negative factor on the adaptability of the framework to different problems. The component which is responsible for performing genetic operators on subpopulations is also implemented as a software service and predefined methods are used. Therefore, users can not use different methods for genetic operations. These two restrictions are the main disadvantages of the framework compared to GridAE.

3.2 ARCHITECTURE

GridAE has 6 components, namely server, master, worker, instant messaging (IM) layer, web interface and command line interface (CLI). These components can be seen in Figure 3.1 and they will be explained in the remaining of this section.

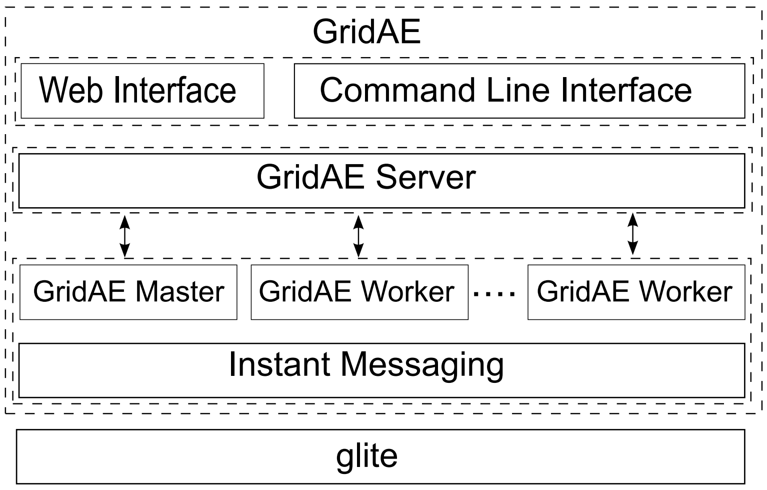


Figure 3.1: Architecture of GridAE

3.2.1 SERVER

The main task of the server is job management. It runs on a UI machine and is composed of several shell scripts. These scrips are responsible for:

- Preparing the jdl files of the workers: The IP address of the UI on which the master runs has to be passed as an argument to the worker processes. Therefore, server modifies the jdl files to include the IP in the *Arguments* field of the jdl. Moreover, user may need to send other files with worker that may be necessary for fitness calculation. The names of these files should be mentioned in the jdl file in the *InputSandbox* field. This task is also performed by the server.
- Starting the master: The master should start running on the UI before the workers are submitted. Server runs the master as a daemon process so that it can start submitting the workers.

- Submitting the jobs to grid: Using *glite-wms-job-submit* command, server submits the jobs and stores the job identifiers in a file. This file is used while the statuses of the jobs are checked.

- Checking the statuses of the jobs: This process is needed twice. The first one is right after the jobs are submitted to determine the jobs that fail to start in a reasonable time (5 minutes). If a job is still not running after 5 minutes, it is canceled with *glite-wms-job-cancel* command and resubmitted. The file that stores the job identifiers is also modified. The identifiers of the resubmitted jobs are changed in the file. The second check is done to realize the termination of the jobs. The status check is done with *glite-wms-job-status* command.

3.2.2 MASTER

The master runs as a daemon process on a UI machine. The server starts the master before submitting the workers to grid. The master first, reads the parameter file, configures the GA according to it and initializes the population with random individuals. Then, the tasks for the workers are prepared by the master and it waits for connections from the workers. The tasks are composed of individuals whose fitnesses are going to be evaluated. When a connection is established, the master sends the tasks with a message starting with the character ’1’ for that worker. ’1’ indicates that the message is a task message containing individuals. After sending all the tasks, the master waits for the results. When the results are ready, the master collects them, performs selection, crossover and mutation with user defined methods to create the new generation. This cycle is repeated until the stopping criteria is met. This criteria can be a desired fitness value or a limit on the generation number. Afterwards, the master instructs the workers to stop, through the same communication channel by sending ’0’ character as a message. The pseudocode of the master can be found in Algorithm 3.1.

The master is also responsible for keeping logs. At each generation, the average and the best fitness is logged in addition to the chromosome of the best n individuals. n is specified by the user in the parameter file. On the other hand, if user does not wish to see the evolution of the problem (setting $n = 0$), only the final fitness and best individual is printed along with the execution time of the whole process.

```
 1: setSelectionFunction
 2: setCrossoverFunction
 3: setMutationFunction
 4: readParameters
 5: initializePopulation
 6: initializeConnection
 7: **repeat**
 8:    performSelection
 9:    performCrossover
10:    performMutation
11:    prepareTasks
12:    **repeat**
13:       acceptConnection
14:       sendTask
15:    **until** all tasks are sent
16:    **repeat**
17:       receiveResult
18:       closeConnection
19:    **until** all results are received
20:    sortPopulation
21:    logInfo
22: **until** $bestFitness < targetFitness$ **or** $genCount < maxGen$
23: **repeat**
24:    acceptConnection
25:    sendStopMessage
26:    closeConnection
27: **until** all workers are shut down
28: logInfo
```

ALGORITHM 3.1: Pseudocode of Master

3.2.3 WORKER

```
1: setFitnessFunction
2: initializeConnection
3: repeat
4:     requestConnection
5:     receiveTask
6:     calculateFitness
7:     sentResult
8:     closeConnection
9: until stop message is received
```

ALGORITHM 3.2: Pseudocode of Worker

The worker's job is simpler compared to the master's. The IP address of the master is given as an argument to the workers and all workers try to establish a connection with the master, when they start running. When connection is secured, they either receive their task or a message to stop from the master. If a task message composed of individuals is received, the fitness values of these individuals are calculated and the results are sent back. The fitness functions are specific to the problems and supplied by the user. If the message arrived is a stop message, it means that evolution is over and the worker stops (see Algorithm 3.2).

3.2.4 IM LAYER

IM layer is responsible for the communication between the master and the workers. The master sends their tasks to workers and results are sent back by the workers, when they are finished. There are two types of messages that master can send, namely *task message* and *stop message* (Figure 3.2). There are 3 fields in the task message. The first field contains the character '1' indicating the type of the message. The second field also contains an integer *n* specifying the number of individuals in that task. The remaining of the message contains the chromosomes of the individuals. The stop message consists of a single character '0'. The message that the master sends can be identified by reading the first character of that message by the worker. A *result message* that contains the amount of fitness values in it with these fitness values is sent back to master for each task message. Implementation details of this layer is given in Chapter 4.

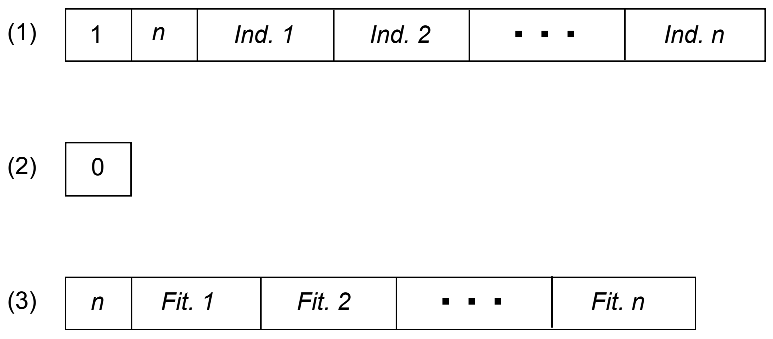


Figure 3.2: Message types in IM Layer: (1) Task Message, (2) Stop Message, (3) Result Message

3.2.5 COMMAND LINE INTERFACE

GridAE includes two script files written in shell script for those users who prefer to use the framework through the command line. Since the scripts use glite commands, they should run on UI machines with glite middleware installed. First one is *submitJobs.sh* which takes at least 2 arguments:

```
submitJobs.sh ip numberOfWorkers [file]...
```

The first argument is the IP address of the computer that the master runs on. The number of workers going to be submitted is given as the second argument. If the worker process needs additional files, the names of these files have to be added at the end. The script submits the jobs and stores the job identifiers in the file named *.address*.

The second script is named *checkJobs.sh* which takes only one argument that can be either ’r’ or ’d’:

```
checkJobs.sh [r|d]
```

If ’r’ (resubmit) is given as the argument, the script checks the statuses of the jobs in five-minute intervals and if a job fails to start running it is canceled and resubmitted. When all jobs start running, the script terminates. If the argument is ’d’ (done), the statuses of the jobs are checked until all of them are done. The job identifiers are read from the file named *.address* and this file is modified if a resubmission occurs.

3.2.6 WEB INTERFACE

Grid has users from many different disciplines that may find it difficult to use the CLI. Therefore a web interface is developed for the users who are more comfortable with GUIs. The interface is implemented as a Java portlet on P-GRADE Portal which is introduced in the previous chapter. P-GRADE Portal is selected because it offers a nice GUI and already has many users that are familiar with it. This choice facilitates the usage of GridAE for those users. Moreover, as explained earlier, it also has an interface for certificate management that can be used by other portlets like GridAE.

P-GRADE Portal assigns roles like *user*, *developer* and *administrator* to its users. A new role *gridae* is defined for the users of GridAE and only these users can see the GridAE tab among other tabs. These roles can only be assigned by the administrators of the portal. Therefore, prospective users should initially contact and fill the application form on the web page of the application [3]. Successful applicants' GridAE accounts are created afterwards.

GridAE portlet has 5 pages, each of them is designed to manage a step of the process. The first page is for the *project management* step in which a new project can be created or an existing one can be loaded. The files of each project is kept in different directories. If an existing project is loaded, the source, parameter and input files of that project can be used without the requirement of reuploading. The next step is for the *parameters* step which is used for specifying the parameters that is necessary for the master. The parameter file is prepared by GridAE according to these user specified values. After the parameters step, the source files *master.cpp* and *worker.cpp* can be uploaded on the *file management and compilation* page. As well as the source files, if there is any input file for the workers, they should also be uploaded on this page. The name of the output file that the master produces has to be specified at the same page. Moreover, there exists a compilation button on this page and clicking that button ensures the compilation of the source files provided by the user. The source files are compiled and linked with the GridAE library by the framework and the output of the compilation is presented to the user. The source files can be uploaded multiple times, but the last one always replaces the previous one. Therefore, if compilation results with errors, the source files can be modified, uploaded again and recompiled. On the next page, the job submission and job status monitoring can be performed. The framework uses the scripts prepared for CLI to perform these operations. At the end, on the *output* page, the output file produced by the master can be downloaded to user's local computer.

3.3 USAGE

The usage of the framework can be summarized in 3 steps. Preparation of the parameter file that the master requires is the first step. The source files, *master.cpp* and *worker.cpp* are prepared at the second step. At the last step, grid is used via one of the interfaces. This section tries to clarify these steps.

3.3.1 PARAMETER FILE

GridAE uses a file to store GA parameters, so that users do not have to change the source code and recompile every time a parameter is needed to be modified. These parameters are used by the master process. The parameter file should be a text file and the syntax of the file is quite simple. It includes lines of *attribute value* pairs. There should be at least one blank space between an attribute and its value and the sequence of these pairs is not important. The following 7 attributes (parameters) can be specified in the file:

- ***ngen*** the maximum generation limit that GA will continue until (a positive integer)
- ***tfitness*** the target fitness that GA will try to achieve (a real number)
- ***popsize*** the number of individuals in the population (a positive integer)
- ***nelite*** the number of elite individuals that will be in the population(0 or a positive integer less than *popsize*)
- ***pmut*** the mutation percentage (a real number between 0 and 1)
- ***verbose*** the verbose level determining the amount of information that will be logged during the process (an integer between -1 and *popsize* which specifies the number of best individuals that will be logged at each generation. 0 means none and -1 means whole population)
- ***workernum*** the number of worker processes that will be used (a positive integer)

If both *ngen* and *tfitness* parameters are specified, GA will stop if one of them is achieved. An example of a parameter file can be found in Figure 3.3.

```
ngen      50
popsize   100
nelite    3
pmut      0.6
verbose   3
workernum 20
```

Figure 3.3: A parameter file example

3.3.2 PREPARING SOURCE CODES

Two C++ source files have to be supplied to GridAE by the user for master and worker processes. These files should be named *master.cpp* and *worker.cpp*. In *master.cpp*, basically the selection, crossover and mutation functions are specified and the population is initialized and in *worker.cpp*, the fitness function is specified.

```
 1: #include "individual.h"
 2: #include "parameters.h"
 3: #include "population.h"
 4: #include "ga.h"
 5: #include "functions.h"
 6:
 7: int main () {
 8:   GA myga;
 9:   Population mypop;
10:   myga.setCrossover_fun(&onePointCrossover);
11:   myga.setMutation_fun(&swapMutation);
12:   myga.setSelection_fun(&rouletteWheel);
13:   mypop.setGa(myga);
14:   mypop.readParameters("parameters.txt");
15:   mypop.randomFill(5, 5, 0, 10, 0.0, 1.0);
16:   mypop.evolve();
17: }
```

Figure 3.4: An example of *master.cpp*

Figure 3.4 gives an example of *master.cpp* file. The first 5 lines include the necessary header files and the rest of the file has the main function. Two objects of type *GA* and *Population* are needed in the master and they are declared on lines 8 and 9 respectively. *setCrossover_fun*, *setMutation_fun* and *setSelection_fun* are all methods of *GA* class and take a pointer to a function as an argument. The prototypes of these functions are given in Figure 3.5.

```
(1) Individual *selectionSample(Individual *, int, int)
(2) int crossoverSample(Individual *, Individual *)
(3) int mutationSample(Individual *)
(4) double fitnessSample(Individual *)
```

Figure 3.5: Prototypes of (1) selection, (2) crossover, (3) mutation and (4) fitness functions

Selection functions take three arguments, the first one being an array of individuals from which the selection will occur. The second argument is the size of this array and the last one determines the number of individuals that will be selected. The array containing the selected individuals is returned by the function. Crossover and mutation functions take references to individuals and return either 0 or -1 indicating success or error respectively. *onePointCrossover*, *swapMutation* and *rouletteWheel* are predefined functions in the GridAE library. *GA* object is a member of *Population* class and set on line 13. Then, parameters are read with the method of *Population* class. The population is then initialized with *randomFill* method that takes 6 arguments, respectively *number of integer genes*, *number of real number genes*, *minimum limit for integer genes*, *maximum limit for integer genes*, *minimum limit for real number genes* and *maximum limit for real number genes*. Finally, *evolve* method which carries on the evolution process is called.

An example of the second source file, *worker.cpp* can be found in Figure 3.6. Lines 4–14 contain the definition of sample fitness function that adds up all the gene values and lines 16–20 contain the main function. The main function of the worker is simpler than the master's. It takes one argument which is the IP address of the computer that the master runs on. An object of type *GA* is needed and declared on line 17. Then, using the *setFitness_fun* method of *GA*, fitness function is specified. Fitness functions take a pointer to the individual whose fitness will be calculated and return a double precision number (see Figure 3.5). GridAE aims to achieve higher fitness values. Therefore, a fitness function which returns higher numbers for better individuals has to be supplied by the user. On the final line in main, *do_calculate*

```
#include "individual.h"
#include "ga.h"

double myfitness_func(Individual *myind) {
  Gene *g = myind->getGenes();
  double fitness = 0;
  for(int i = 0; i < myind->getGene_count(); i++) {
    if(g[i].int_value)
      fitness += g[i].value_i;
    else
      fitness += g[i].value_d;
  }
  return fitness;
}

int main (int argc, char *argv[]) {
  GA ga;
  ga.setFitness_fun(&myfitness_func);
  ga.do_calculate(argv[1]);
}
```

Figure 3.6: An example of *worker.cpp*

method is called with the IP address given as the only argument.

3.3.3 USING COMMAND LINE INTERFACE

Once the source files are ready, they should be compiled and linked with the GridAE library. The object files of GridAE are transfered into a dynamic library named *libgridae.a*. This step can be carried out with the following commands:

```
g++ -c master.cpp
g++ -c worker.cpp
g++ -o master -lgridae master.o
g++ -o worker -lgridae worker.o
```

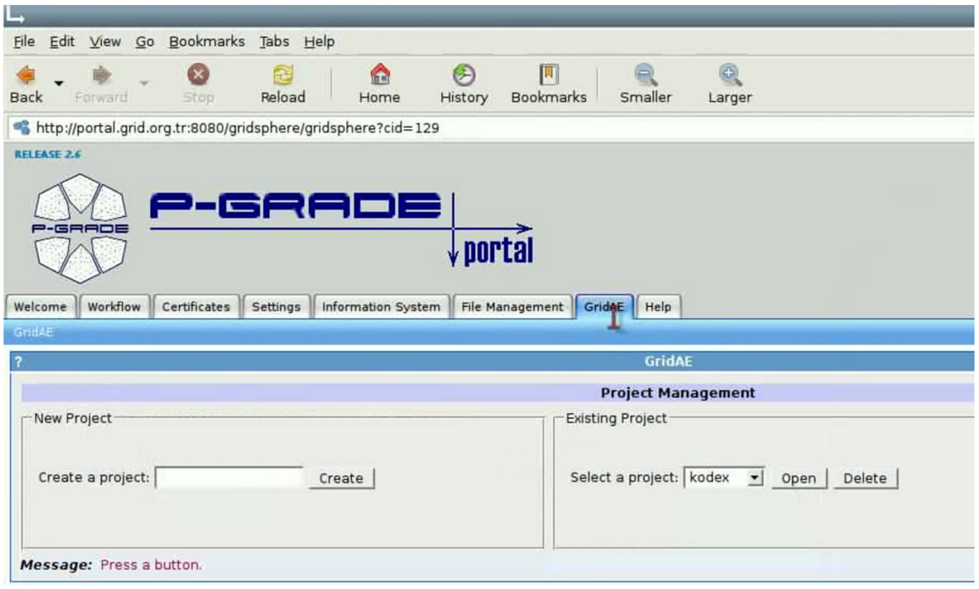
A successful compilation produces two executable files, *master* and *worker*. Before submitting the workers to grid, the master should start running. Master can be executed with the following command on a terminal:

```
./master
```

Then, on another terminal the workers should be submitted using *submitJobs.sh* script:

```
./submitJobs.sh ip numberOfWorkers [file]...
```

It should be noted that the number of workers given here should be consistent with the *workernum* parameter in the parameter file. After submission, it is recommended to run the *checkJob.sh* with argument *r* to detect the jobs not starting running and resubmit them automatically.


File Edit View Go Bookmarks Tabs Help
Back Forward Stop Reload Home History Bookmarks Smaller Larger
RELEASE 2.6
P-GRADE portal
Welcome Workflow Certificates Settings Information System File Management GridAE Help
Open Delete Create

http://portal.grid.org.tr:8080/gridsphere/gridsphere?cid=129

GridAE

Project Management

New Project

Create a project:

Existing Project

Select a project: kodex

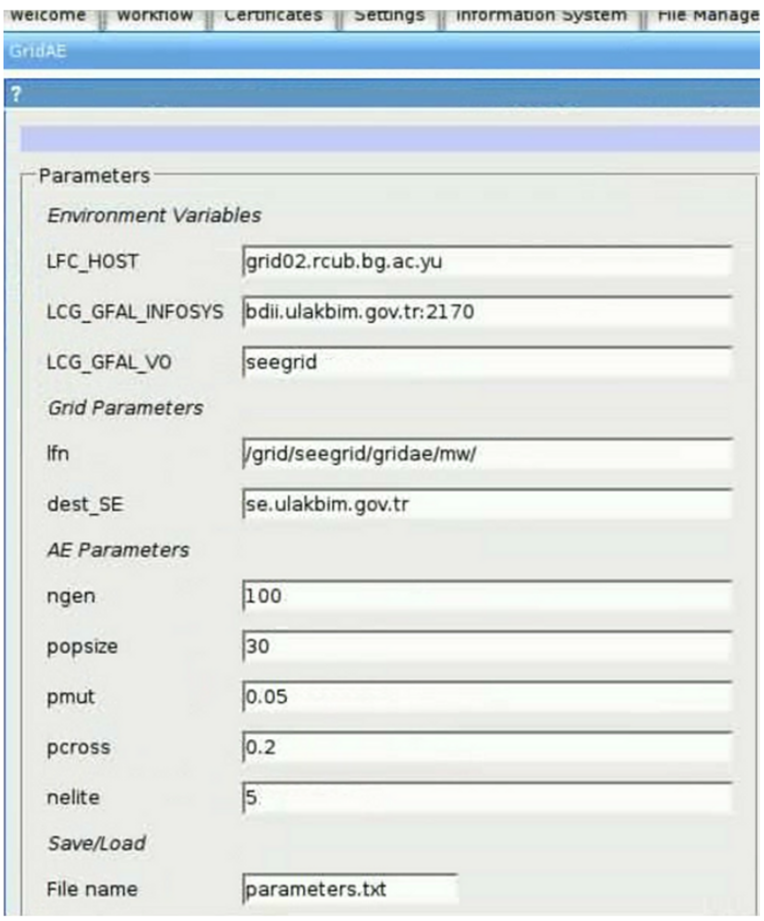
Message: Press a button.

Figure 3.7: Project page on GridAE

3.3.4 USING WEB INTERFACE

After signing in the P-GRADE Portal, GridAE users will be able to see the ”GridAE” tab among the other tabs (see Figure 3.7). Before starting to use the framework, user should create a proxy as explained in Chapter 2. The first page on the GridAE tab is the *project* page that user can create a project (on the left in Figure 3.7) or select an existing one (on the right) to open. Existing projects can also be deleted on this page by clicking ”Delete” button. After clicking ”Create” or ”Open” buttons, the next page of the framework loads.

The next page is the *parameters* page 3.8 and the GA parameters is specified on this page. After filing the form, a file name has to be given and ”Save” button should be clicked. It should be noted that the name of the file should be consistent with the file name given as a parameter to the *readParameters* method in the *master.cpp*. If a previously created project is opened and a parameter file has already been saved during previous usages of GridAE, than without filling the form, the name of the file can be written in the same text box and ”Load” button can be clicked to use the same parameters. After loading the parameters, they can also be modified and saved by the user. However, the file that is lastly loaded or saved will be used by the framework. If nothing is left to be done on this page, the ”Next Step” button at the bottom of the page can be clicked to go to the next page.



Parameters

Environment Variables

LFC_HOST	grid02.rcub.bg.ac.yu
LCG_GFAL_INFOSYS	bdii.ulakbim.gov.tr:2170
LCG_GFAL_VO	seegrid

Grid Parameters

lfn	/grid/seegrid/gridae/mw/
dest_SE	se.ulakbim.gov.tr

AE Parameters

ngen	100
popsize	30
pmut	0.05
pcross	0.2
nelite	5

Save/Load

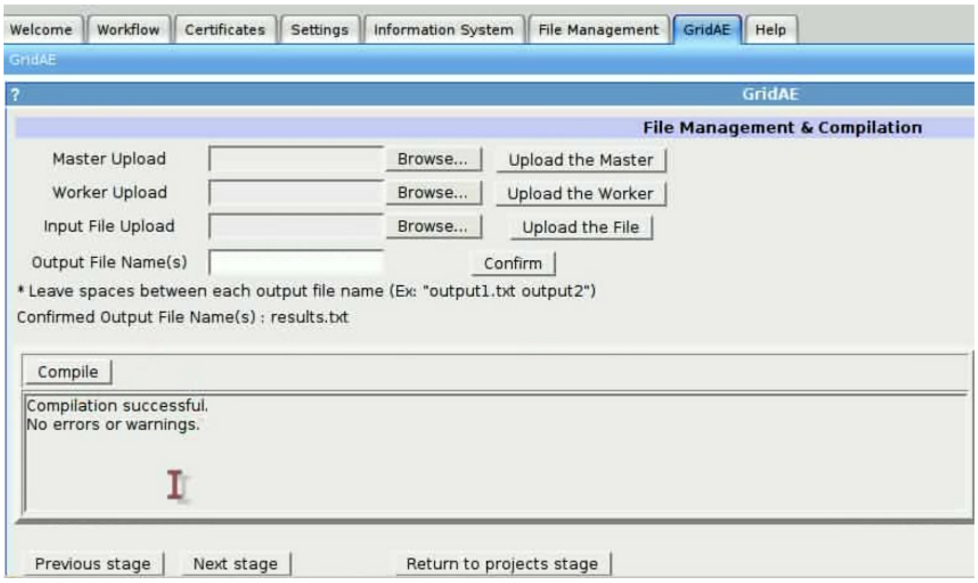
File name	parameters.txt

Figure 3.8: Parameters page on GridAE

On *file management and compilation* page, the source and input files can be uploaded. The files can be selected from the user’s local computer with ”Browse” button and uploaded with

”Upload” buttons. If the *master* produces any output files, they have to be specified on ”Output File Name(s)” area with one blank space between them.

The compilation has to be done with the ”Compile” button. The output of the compilation appears on the area below that button. The source files can be uploaded and the compilation can be performed multiple times. After a successful compilation the user can proceed with the next step.



* Leave spaces between each output file name (Ex: "output1.txt output2")

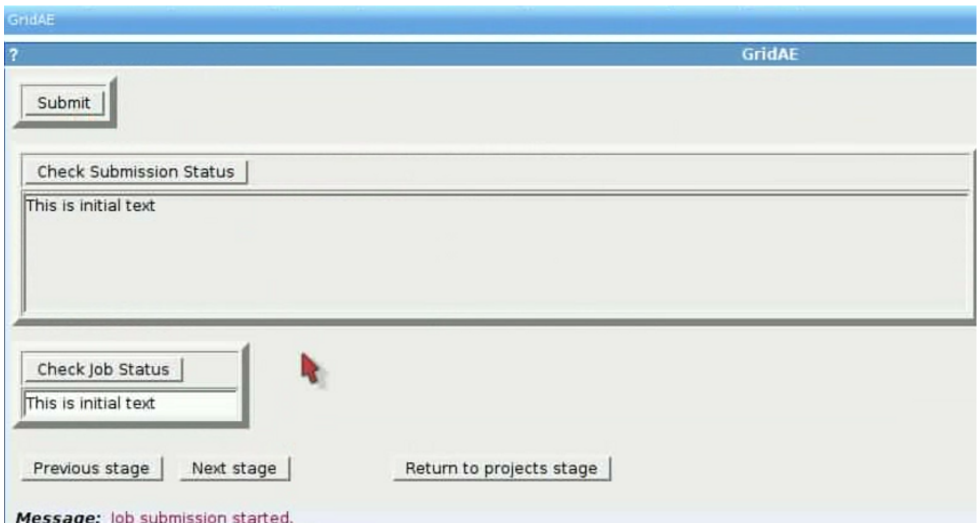
Confirmed Output File Name(s) : results.txt

```
Compilation successful.
No errors or warnings.
```

Figure 3.9: File management and compilation page on GridAE

The next step after the compilation is the submission step (see Figure 3.10). The *workers* have to be submitted with the ”Submit” button. After the submission, ”Check Submission Status” button is used to check the statuses of the jobs and resubmit automatically, if any of them fails to start running. When all of the jobs start running, it will be stated on the area below this button. The final button on this page is the ”Check Job Status” button which is used for monitoring the statuses of the running jobs to be noticed when they finish executing.

The final page is designed for the outputs of the master process (see Figure 3.11). With the ”Prepare” button the output files which are specified by the user are packed in a single tar ball. And with the ”Download” button, this file can be transferred to the user’s local computer.

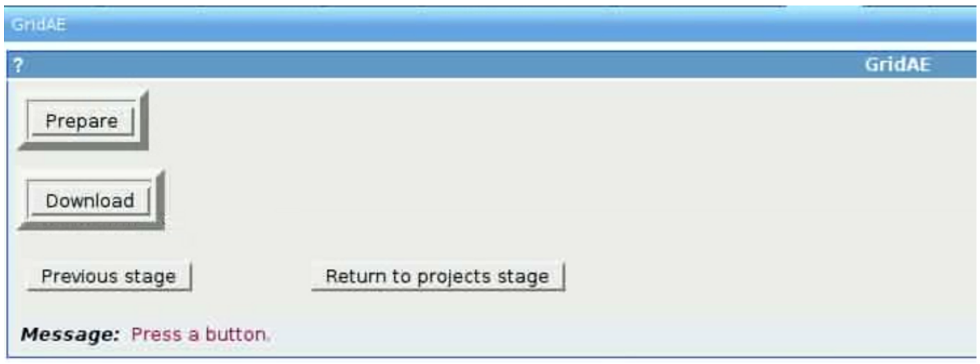


This is initial text

This is initial text

Message: Job submission started.

Figure 3.10: Submit page on GridAE



Message: Press a button.

Figure 3.11: Output page on GridAE

3.4 EXECUTION TIME ANALYSIS

In order to compare the performance of the framework with that of serial GA, the execution time of both options should be analyzed. In a typical GA, the time consuming steps can be summarized in 3 items:

1. Initialization
2. Fitness calculation of individuals
3. Genetic operations (selection, crossover, mutation)

Initialization is only performed once and its running time is a function of population size ($I(populationsize)$). The second and third items are performed at each generation and running time of performing genetic operations is also a function of population size ($G(populationsize)$). Assuming that the fitness calculation of each individual takes the same amount of time, then the total running time of a serial GA application and GridAE are given by the following formulas respectively:

$$T_{serial} = I(p) + ngen(fp + G(p)) \tag{3.1}$$

$$T_{GridAE} = S(w) + I(p) + ngen(wccost + \frac{fp}{w} + G(p)) \tag{3.2}$$

where, p is the population size, $ngen$ is the number of generations, f is the fitness calculation time of an individual, w is the number of workers, $ccost$ is the communication cost of each task and $S(w)$ is the elapsed time between the starting time of submission of the workers and the time they all start running which depends on the number of workers.

In order to achieve better performance results with GridAE compared to the serial implementation of the same problem, the following inequality should hold:

$$T_{GridAE} < T_{serial} \tag{3.3}$$

Using Equations 3.1 and 3.2, Equation 3.3 yields to:

$$S(w) + I(p) + ngen(wccost + \frac{fp}{w} + G(p)) < I(p) + ngen(fp + G(p)) \tag{3.4}$$

After simplifying Equation 3.4, we have the following inequality:

$$ccost < \frac{fp(w-1)}{w^2} - \frac{S(w)}{wngen} \tag{3.5}$$

Communication cost is the required time to transfer a task message from the server to the worker and the respective result message from the worker to the master. The length of the task message and the distance of the worker from the server are the two main factors determining the communication cost. The length of the task message depends on the representation of an individual and the number of individuals sent within a single task.

Equation 3.3 shows that if population size or fitness calculation time of a single individual increases, GridAE is more likely to perform well. Moreover, the number of generations decreases the effect of the starting time of the workers. Total execution time is proportional to the number of maximum generations and the starting time required for GridAE is less effective when the whole process takes more time. The number of workers is another factor determining the performance. However, the effect of worker number is more complicated since it both affects the right hand side of the inequality and the communication cost. When other variables are fixed, increasing the number of workers decreases the communication cost of a single task. Because the number of individuals in each task decreases.

We can also drive speedup and efficiency formulas with respect to number of workers (w) for GridAE from Equations 2.1, 2.2, 3.1 and 3.2:

$$S_w = \frac{I(p) + ngen(fp + G(p))}{S(w) + I(p) + ngen(wccost + \frac{fp}{w} + G(p))} \tag{3.6}$$

$$E_w = \frac{I(p) + ngen(fp + G(p))}{w[S(w) + I(p) + ngen(wccost + \frac{fp}{w} + G(p))]} \tag{3.7}$$

CHAPTER 4

IMPLEMENTATION DETAILS

In this chapter, details of GridAE library are introduced. Firstly, the class structure is explained in detail, because the users should know the relations between these classes to make good use of the library. The latter section describes two approaches in the design of the IM Layer from the developer's perspective.

4.1 CLASSES

GridAE library has 5 classes, as summarized in Figure 4.1, which will be explained in detail one by one in this section.

***Gene* class**

This class does not have any methods and all its members are public. Therefore, it is like a container. Each instance holds the data of a single gene in a chromosome. The gene holds either an integer or a real number value. If the boolean variable *int_value* is true then the value is integer and stored in *value_i*; otherwise it is a real number and stored in *value_d*. The other members *min* and *max* hold the minimum and the maximum limits that the gene value can take, respectively.

***Parameters* class**

Parameters class holds the GA parameters *ngen* (maximum generation limit), *tfitness* (target fitness), *popsize* (population size), *nelite* (number of elite individuals that will be carried to next generation) and *pmut* (probability of mutation) as well as *workernum* (number of workers that will be used) and *verbose* (verbose level). These parameters are read from a text file with the *read* method.

Individual class

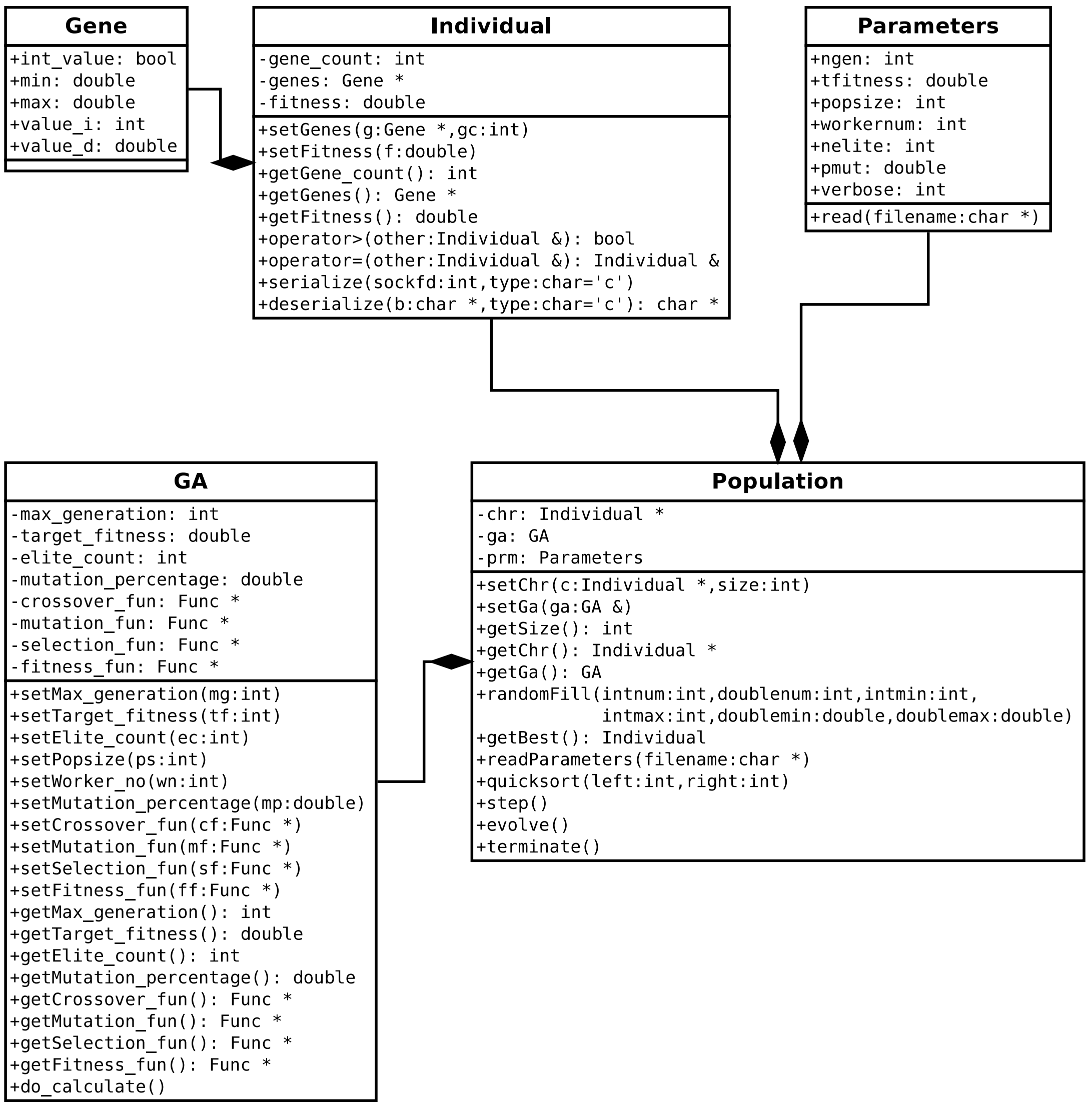
Each individual is stored in an instance of the *Individual* class. Apart from the self explanatory *set* and *get* methods, the class implements comparison operators and the *equal to* operator. The comparison operators compare the two individuals according to their fitness values. (Only the *bigger than (>)* operator is shown in Figure 4.1 for simplicity.) If *'c'* (chromosome) is given as the second argument, the *serialize* method takes the socket descriptor and encodes the chromosome to socket and *deserialize* takes a character pointer holding the address of the beginning of the encoded chromosome and decodes it. The other option is *'f'*, where instead of the chromosome, only the fitness value is written or read respectively by these methods.

GA class

The *GA* class holds all the necessary information about the GA that is going to be used. The prototypes of the functions for genetic operations are not given in Figure 4.1, this information can be obtained from Figure 3.5. All of the members and methods of the GA class are self explanatory except the *do_calculate* method. This method is called in the worker after the initializations and does the whole work. It establishes the connection and gets the task. It uses the *serialize* method of the *Individual* class during the communication and calculates the fitness values of the individuals, and finally sends the results using the *deserialize* method. This cycle continues inside the *do_calculate* method until the master sends the stop message.

Population class

A *Population* object holds instances of the *Parameters*, *Individual* and *GA* classes. This class implements the usual *set* and *get* methods and also the *getBest* method, which returns the individual with the best fitness value. The *readParameters* method uses the *read* method of the *Parameters* class. The *randomFill* method can be used to initialize the population. To easily reach the best individuals, the population is sorted after each generation and the *quicksort* method is used for this purpose. The master process calls the *evolve* method after the initializations. This method is responsible for initializing the connection and calling the *step* method until the best individual reaches the target fitness or the generation number reaches the maximum generation limit. When this condition occurs, *evolve* calls *terminate* that sends the stop message to the workers.



Gene
+int_value: bool
+min: double
+max: double
+value_i: int
+value_d: double
Individual
-gene_count: int
-genes: Gene *
-fitness: double
+setGenes(g:Gene *,gc:int)
+setFitness(f:double)
+getGene_count(): int
+getGenes(): Gene *
+getFitness(): double
+operator>(other:Individual &): bool
+operator=(other:Individual &): Individual &
+serialize(sockfd:int,type:char='c')
+deserialize(b:char *,type:char='c'): char *
Parameters
+ngen: int
+tfitness: double
+popsize: int
+workernum: int
+nelite: int
+pmut: double
+verbose: int
+read(filename:char *)
GA
-max_generation: int
-target_fitness: double
-elite_count: int
-mutation_percentage: double
-crossover_fun: Func *
-mutation_fun: Func *
-selection_fun: Func *
-fitness_fun: Func *
+setMax_generation(mg:int)
+setTarget_fitness(tf:int)
+setElite_count(ec:int)
+setPopsize(ps:int)
+setWorker_no(wn:int)
+setMutation_percentage(mp:double)
+setCrossover_fun(cf:Func *)
+setMutation_fun(mf:Func *)
+setSelection_fun(sf:Func *)
+setFitness_fun(ff:Func *)
+getMax_generation(): int
+getTarget_fitness(): double
+getElite_count(): int
+getMutation_percentage(): double
+getCrossover_fun(): Func *
+getMutation_fun(): Func *
+getSelection_fun(): Func *
+getFitness_fun(): Func *
+do_calculate()
Population
-chr: Individual *
-ga: GA
-prm: Parameters
+setChr(c:Individual *,size:int)
+setGa(ga:GA &)
+getSize(): int
+getChr(): Individual *
+getGa(): GA
+randomFill(intnum:int,doublenum:int,intmin:int,
intmax:int,doublemin:double,doublemax:double)
+getBest(): Individual
+readParameters(filename:char *)
+quicksort(left:int,right:int)
+step()
+evolve()
+terminate()

Figure 4.1: Class diagram of GridAE library

4.2 IMPLEMENTATION OF THE IM LAYER

Communication is the backbone of parallel applications. There are tools like MPI and PVM that offer efficiency and reliability for cluster computing. However, there are no such tools that can be used on a grid platform. Therefore, the IM Layer is implemented without help of such a tool. Our research led us to a C library called *lcg_util* which is used for file management through SEs. Despite being quite reliable, this library's performance was under acceptable limits. Due to this failure, TCP/IP sockets were used to achieve a better performance. The details of these two approaches will be given in this section.

4.2.1 LCG UTIL LIBRARY

LCG Data Management Tool (*lcg_util* library) is used for file transfer between SEs and other elements of grid. To be able to use this library in GridAE, each message is written into a file and the master and the workers copy this file to an SE with a specific name and the receiver downloads that file. Three functions, namely *lcg_cp* (copy), *lcg_cr* (copy and register) and *lcg_del* (delete), are frequently used in the library. The prototypes of these functions are given in Figure 4.2 and their arguments are explained in the Table 4.1.

```
int lcg_cr (char *src_file, char *dest_file, char *guid, char *lfn,
            char *vo, char *relative_path, int nbstreams,
            char *conf_file, int insecure, int verbose,
            char *actual_guid);

int lcg_cp (char *src_file, char *dest_file, char *vo, int nbstreams,
            char *conf_file, int insecure, int verbose);

int lcg_del (char *file, int aflag, char *se, char *vo,
             char *conf_file, int insecure, int verbose);
```

Figure 4.2: Prototypes of frequently used *lcg* functions

There are more than one way to refer to the files in the grid. Grid Unique Identifiers (GUID) and Logical File Names (LFN) are two of them. A GUID is a 41 byte randomly generated string which uniquely identifies a file. LFN is in a more understandable and hierarchical format:

Table 4.1: The arguments used in *lcg* functions

Argument	Meaning
src_file	The source file name
dest_file	The destination
guid	The grid unique identifier
lfn	The logical file name associated with the file
vo	The virtual organization the user belongs to.
relative_path	The path relative to the SARoot for the given VO
nbstreams	The number of parallel streams
conf_file	Currently ignored
insecure	Currently ignored
verbose	Verbose level
actual_guid	Buffer for actual grid unique identifier
file	The logical file name
aflag	Non-zero if all replicas will be deleted
se	Only the replica on this SE will be deleted

```
lfn:/grid/<VO>/<directory>/<filename>
```

The LCG File Catalog holds the links between the *lfn* addresses and actual files. A file may have more than one copy (replica) on different SEs. When the file is required, the file on the nearest SE is downloaded.

GridAE used the *lfn:/grid/seegrid/gridae/* directory to store message files. The workers were enumerated and these numbers were used in the filenames to distinguish the tasks and results. For instance, the task message for the first worker was in the file named *t_1* and respective result message was in file *r_1*.

However, even when the nearest SEs were used, the performance of the library was still disappointing. Transmission of a single message could take more than 3 minutes. Therefore, an alternative method needed to be used.

4.2.2 TCP/IP SOCKETS

TCP/IP sockets enable interprocess communication through the Internet among processes which may run on different computers. Therefore, they are suitable for GridAE. In GridAE implementation, the master behaves like a *server* and *workers* act like *clients*. The master uses the following code segment to initialize the socket for getting ready to accept connections:

```
struct sockaddr_in serv_addr;
bzero((char *) &serv_addr, sizeof(serv_addr));
serv_addr.sin_family = AF_INET;
serv_addr.sin_addr.s_addr = INADDR_ANY;
serv_addr.sin_port = htons(PORTNO);

sockfd = socket(addr->sin_family, SOCK_STREAM, 0);
bind(sockfd, &serv_addr, sizeof(serv_addr));
listen(sockfd, workernum);
```

Since *SOCK_STREAM* type sockets are used, the connection should be established before any data transfer. The *listen* function call tells the system that the program is ready for connections. The second argument of *listen* specifies the number of maximum waiting connection requests that the system should hold. The rest of the communication code of master is here:

```
struct sockaddr_in cli_addr;
int newsockfd[workernum];
for(int i = 0; i < workernum; i++) {
  newsockfd[i] = accept(sockfd, (struct sockaddr *) &cli_addr,
                        sizeof(cli_addr));
  send(newsockfd[i], buffer, strlen(buffer), 0);
  shutdown(newsockfd[i], SHUT_WR);
}
for(int i = 0; i < workernum; i++) {
  int n, m = 0;
  while ((n = recv(newsockfd[i], recvbuffer, 20000, 0)) != 0) {
    strncpy(buffer+m, recvbuffer, n);
    m+=n;
  }
  shutdown(newsockfd[i], SHUT_RDWR);
  close(newsockfd[i]);
}
```

The *accept* function returns a file descriptor specific to that connection and they are stored in an array. After the task message is sent, the write end of the socket is closed. This lets the receiver understand that no more data will come and the respective *recv* function returns 0. Since, *recv* may not get the whole message at once, it is called in a *while* loop. When the result message is received the socket is closed.

The respective *worker* code is similar. The main difference is that it does not use the *listen* and the *accept* functions; the *connect* function is called instead.

CHAPTER 5

TEST CASE

This chapter aims to give an example of how GridAE can be used and to prove that it is suitable for many problems and performs well. The test problem introduced below is a shape optimization problem from aeronautical science. The problem is described in the first section. Its solution and adaptation process to GridAE are given in the next one. Finally, the results obtained in terms of fitness and execution time are discussed in the final section.

5.1 PROBLEM: SHAPE OPTIMIZATION OF A MISSILE

In this problem, the aim is to obtain an airframe geometry for a missile that provides the longest possible flight. It is assumed that propulsion, warhead and instruments sections are previously defined for this experiment. In addition to that, the dimensions of the body and the nose shape are already determined. Thus, the properties related to the body are taken as constants, and are not considered in the optimization process. A typical airframe geometry, with design parameters is given in Figure 5.1.

The configuration is built with a body which has a circular cross section and two sets of fins. The first fin set (wing section) consists of two panels and the second set (tail section) consists of cross-oriented four fins. The following parameters are selected as design variables on cross-sectional plane of each fin set:

- Leading edge sweep angle (Λ_l)
- Root chord length (c_r)

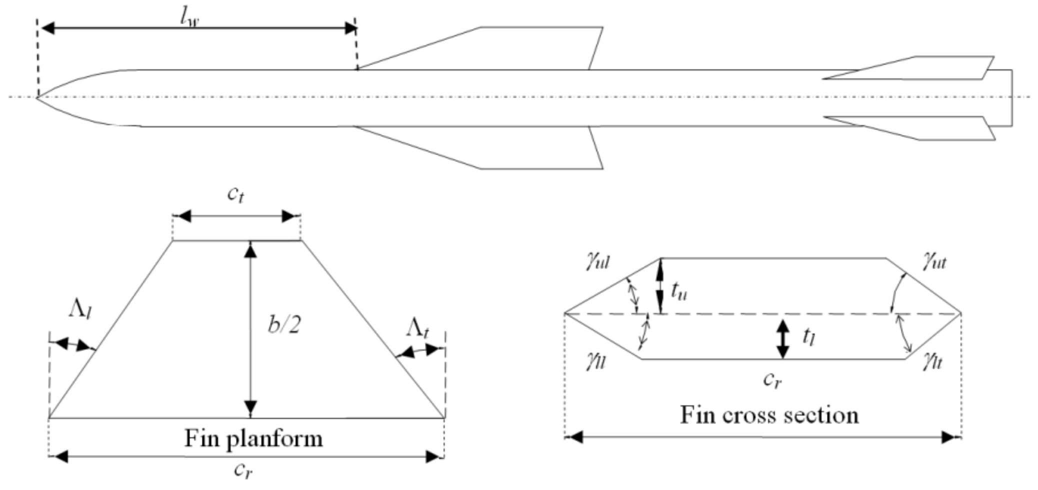


Figure 5.1: Airframe geometry

- Aspect ratio ($A = b^2/2S, S:$ plan-form area)
- Taper ratio ($\lambda = c_t/c_r$)

Each plane has a double wedged cross section. It is assumed that thickness-to-chord ratios are constant along the half span. On the cross section, the following parameters determine the geometry of each fin set:

- Thickness-to-chord ratios of wedges (t/c)
- Cone angles for leading edges of wedges (γ_l)
- Cone angles for trailing edges of wedges (γ_t)

The wedges are assumed to be symmetric with $t_u = t_l$, $\gamma_{ul} = \gamma_{ll}$ and $\gamma_{ut} = \gamma_{lt}$.

There is one final parameter determining the design:

- Position of leading edge on the body (l_w)

To summarize, total of 15 parameters on the missile geometry should be optimized to achieve the longest flight in terms of distance.

5.2 SOLUTION APPROACH

GA and GridAE are used to solve this optimization problem. Each parameter given in the previous section becomes a gene and a parameter set (containing 15 parameters) represented in a chromosome of an individual. The fitness of an individual is calculated with a simulation of the flight. This simulation is carried out with a simple aerodynamics simulation software which uses files for input and output.

Master specifies the genetic operators which are *tournament selection uniform crossover* and *one point mutation* for the experiments. Since, these methods already exist in GridAE library, the master code is simple. Each gene (parameter) has different minimum and maximum limits given in the Table 5.1. The initialization is done according to these limits.

Table 5.1: The minimum and maximum values for each parameter

Parameter	*min*	*max*
l_w	0.9	1.6
c_r (wing & tail)	0.7	1.2
Λ_l (wing & tail)	30.0	60.0
A (wing & tail)	0.185	0.7
λ (wing & tail)	0.6	0.8
t/c (wing & tail)	0.02	0.04
γ_l (wing)	7.0	9.0
γ_l (tail)	8.0	10.0
γ_t (wing & tail)	9.0	11.0

In the worker, the fitness function is implemented so that it prepares the input file for the simulation software, runs the software with a system call and reads the output file it produces. This software simulates the flight with the parameters carried with the individual and determines the distance that the missile travels, in meters. GridAE CLI is used for managing the worker jobs. There is one additional file that should be sent with *worker* to CE which is the simulator executable. Therefore, *submitJobs* script is used as follows, ”simulator” being the name of the executable:

```
submitJobs.sh ip numberofworkers simulator
```

The CEs that the worker processes run on are assigned by grid job scheduler and selected among 10 sites (given in Table 5.2), according to their availability. Therefore, multiple clusters are used at each experiment. An unbiased job distribution among these clusters is not observed during the experiments.

Table 5.2: The sites that the experiments are performed on

Sites
ce.seua-cluster.grid.am
cream01.athena.hellasgrid.gr
ce01.athena.hellasgrid.gr
ce.ysu-cluster.grid.am
ce.ysu-cluster2.grid.am
ce01.marie.hellasgrid.gr
ce64.ipb.ac.rs
kalkan1.ulakbim.gov.tr
cr1.ipp.acad.bg
grid01.elfak.ni.ac.rs

5.3 RESULTS AND DISCUSSION

The experiments are performed with different configurations of *population size*, *maximum generation limit* and *number of workers*. Since all of these parameters can be configured in the parameter file with the attributes *popsize*, *ngen* and *workernum*, the code need not to be changed and recompiled. Each experiment is repeated three times and the average of the results is used in the charts.

The objective of the experiments was to reveal the relation between the execution time and other parameters. Moreover, the effect of generation number and population size on fitness is investigated. For these experiments, in addition to the best fitness, average fitness of the population is viewed too. However, during the evolution some individuals represent unstable geometries and this is penalized by fitness function. These individuals have 0 fitnesses values as a result. These unstable individuals affect the population's fitness average significantly. Therefore, another ”average” is calculated among the non-zero fitnesses and named *non0 average*.

In all of the experiments, elitism is used. At every generation, the best individual of the previous one is carried to the next generation.

The first experiment is performed to see the convergence of fitness in generations. The evolution is carried until the 200th generation and obtained fitness values at each 10th generation is recorded. The population size is fixed at 40 for the experiment. The results can be seen in Figure 5.2. Although the *best fitness* continues to improve until the end of the evolution, after the 50th generation the improvement is not very significant. As it can be seen from Figure 5.2, the *average fitness* does not give much information about the convergence. It is continuously alternating without any correlation with the *best fitness*. The *average non0 fitness* is more parallel with *best fitness*, although it is maximum at the 130th generation.

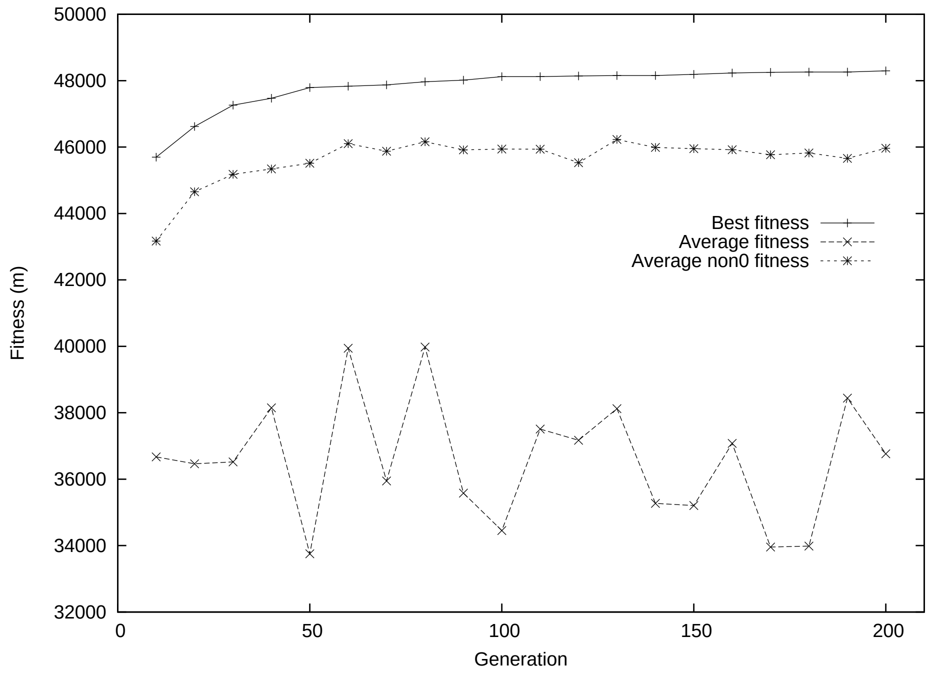


Figure 5.2: Fitness vs. generation (*popsize* = 40)

Among all the experiments the best fitness for this problem is achieved during this experiment. In one of the runs, we could get 49147.578 as the final best fitness. The best individual is corresponding to the parameters in the Table 5.3.

The next thing investigated is the effect of population size on the fitness when maximum generation is fixed. The population size is 40, 60, 70 and 80 in each run and the evolution continues till the 100th generation. In Figure 5.3, it can be seen that the increase in the

Table 5.3: The values of variables with the best individual

Parameter	value (wing)	value (tail)
l_w	1.593	
c_r	0.7	0.7
Λ_l	54.9	60.0
A	0.612	0.422
λ	0.676	0.704
t/c	0.0204	0.0202
γ_l	7.56	9.98
γ_t	10.9	9.78

population size has a positive effect on the best fitness. The best value is obtained when the population size is the biggest.

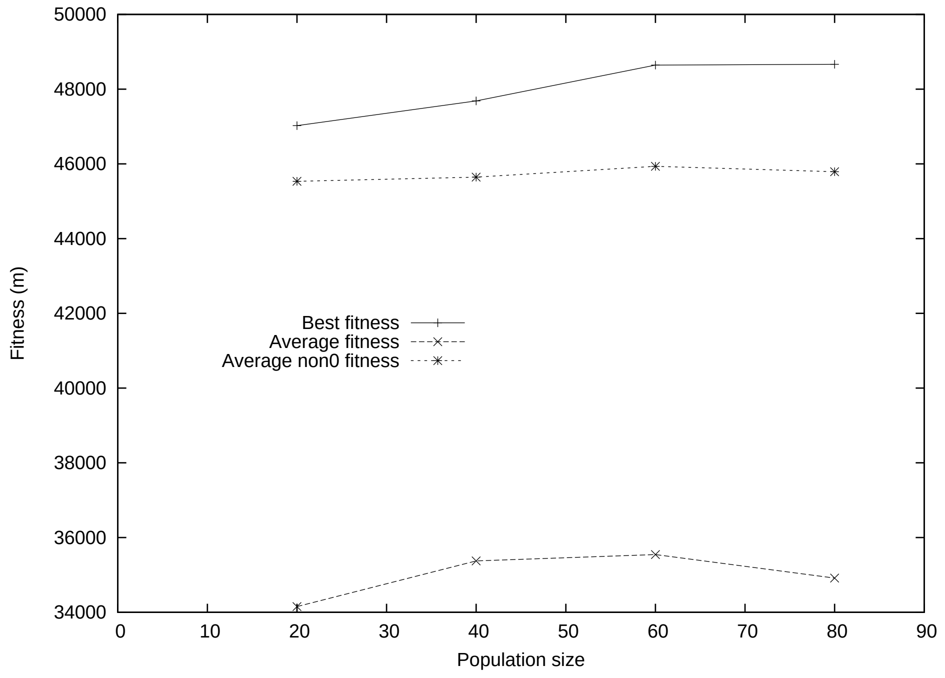


Figure 5.3: Fitness vs. population size ($ngen$ = 100)

However, an increase in the population size means more fitness calculation and longer running programs. The execution time of these experiments are given in Figure 5.4. 20 workers are used for these experiments and as it can be seen from Figure 5.4, running time significantly increases when the population gets bigger.

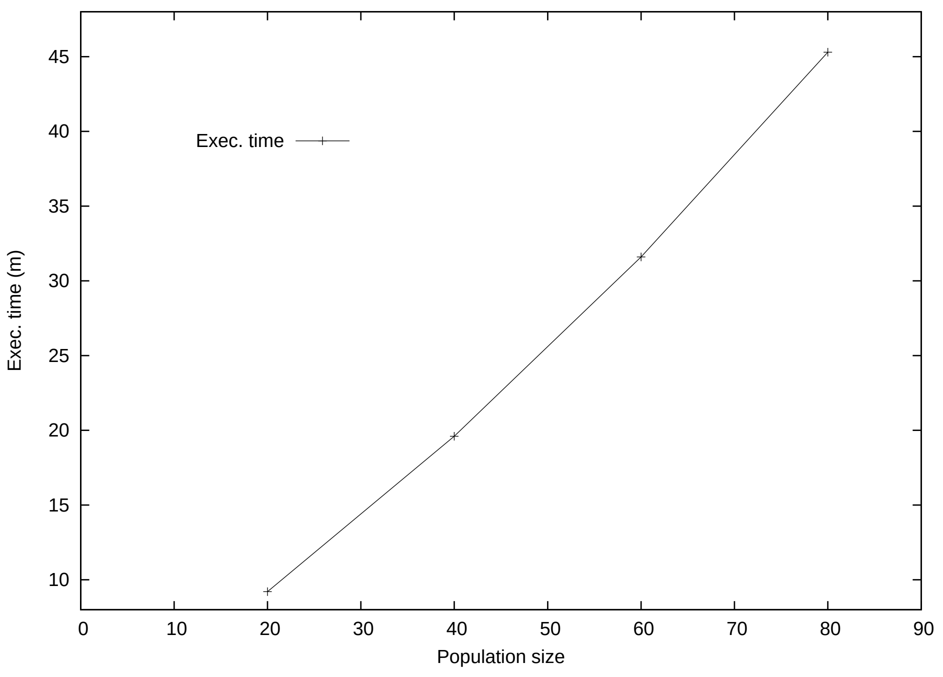


Figure 5.4: Execution time vs. population size (*workernum* = 20)

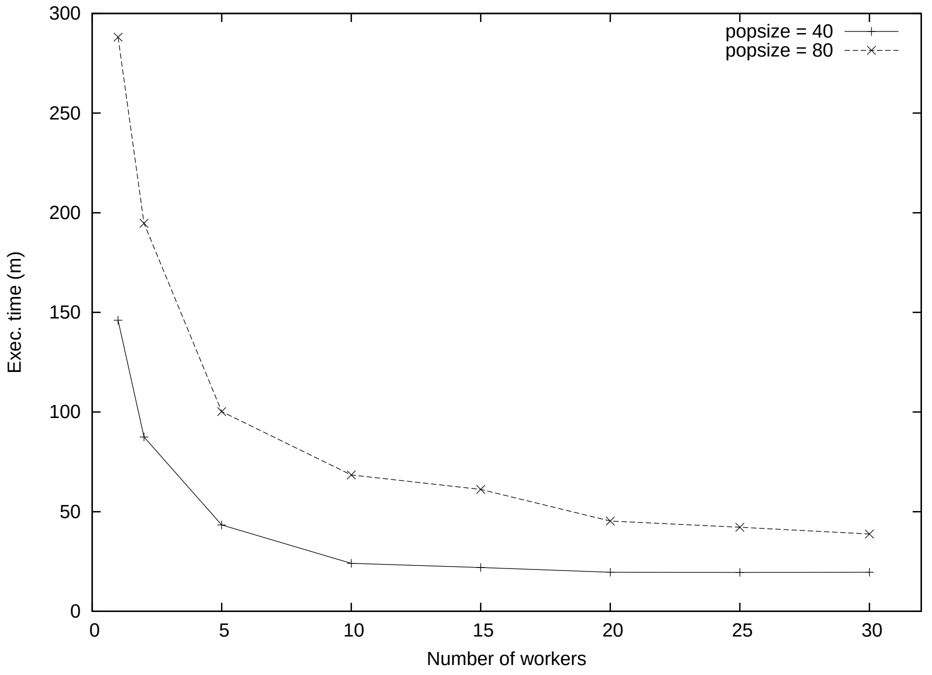


Figure 5.5: Execution time vs. number of workers (*ngen* = 100)

With GridAE, longer running times with crowded populations can be shortened by using more workers. Grid offers much more computing power than a single cluster. The power of GridAE comes from this fact. In Figure 5.5, the execution times of two populations with population sizes 40 and 80 respectively with respect to number of workers used are given. When there are 40 individuals in the population, the framework scales well up to 10 workers (where each worker deals with 4 individuals at each generation). There is a slight improvement with doubling the number of workers to 20. However, after that, increasing the number of workers does not have a visible effect on the running time. On the other hand, with 80 individuals in a population, significant improvements are seen up to 20 workers.

The experimental speedup and efficiency values are illustrated in Figures 5.6 and 5.7 respectively. The charts show that when the population size is 40 both speedup and efficiency are better up to the 30 workers which is expected, since the genetic operations take more time when populations are more crowded. On the other hand, as the number of workers increases the distance between these two lines lessens. This happens because the number of individuals per worker drops below 4 when the population size is 40 and the communication cost becomes more dominant. Since the individuals are not shared fairly when there are 15 workers, we experience worse performance compared to the impression given by the charts.

The charts in Figures 5.6 and 5.7 are also consistent with the Equations 3.6 and 3.7. For instance, when number of generations is 100, population size is 40 and number of workers is 10 we have the following for speedup from Equation 3.6:

$$S_{10} = \frac{I(40) + 100(40f + G(40))}{S(10) + I(40) + 100(10ccost + 4f + G(40))} \tag{5.1}$$

And, when the population size is 80, Equation 3.6 becomes as follows:

$$S_{10} = \frac{I(80) + 100(80f + G(80))}{S(10) + I(80) + 100(10ccost + 8f + G(80))} \tag{5.2}$$

If we ignore I, and S in Equations 5.1 and 5.2, we have the following equations respectively:

$$S_{10} = \frac{40f + G(40)}{10ccost + 4f + G(40)} \tag{5.3}$$

$$S_{10} = \frac{80f + G(80)}{10ccost + 8f + G(80)} \tag{5.4}$$

When the population size is 80, the number of individuals in a single task message is twice the number in the case when the population size is 40. Therefore, if we assume the communication cost doubles with the population size, all the factors in the second equation doubles except G which is the cost of genetic operations and sorting. Since, quicksort has $O(nlogn)$ complexity, the elapsed time increases more than twice, when the population size doubles. Therefore, we have simply $\frac{x}{y}$ and $\frac{2x+a}{2y+a}$ for speedups of population sizes 40 and 80 respectively. Doubling both numerator and denominator does not change the value of a fraction, but adding a number to both numerator and denominator decreases the value of the fraction if it is bigger than 1, which is the case here. Therefore, we expect the speedup (and consequently the efficiency) to be higher when the population size is 40, which is the case in Figure 5.6 (and in Figure 5.7).

Although, the speedup and efficiency give important information about the performance of the framework, from the user's perspective, even for one minute decrease in the total execution time, increasing the number of workers significantly may be preferred. Therefore, Figure 5.5 can be more important than Figures 5.6 and 5.7 for the user.

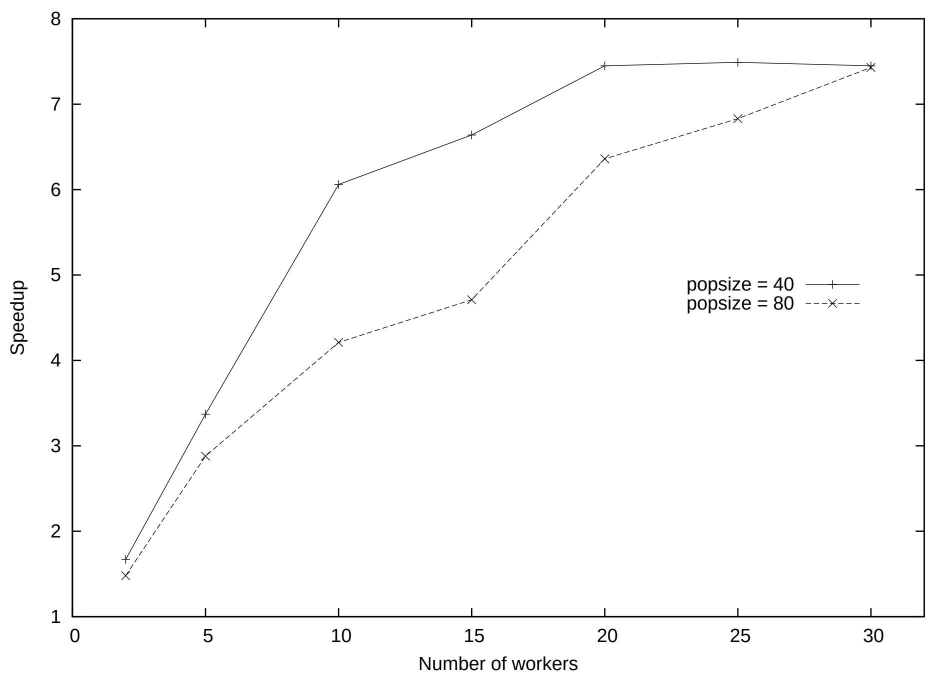


Figure 5.6: Speedup vs. number of workers (*ngen* = 100)

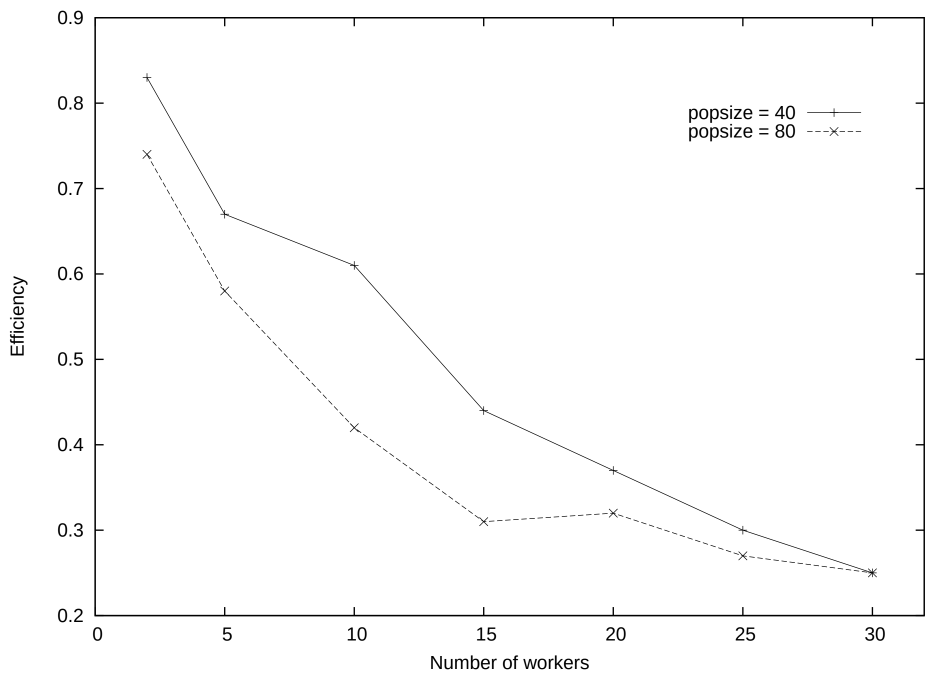


Figure 5.7: Efficiency vs. number of workers (*ngen* = 100)

CHAPTER 6

CONCLUSION

GA is a very popular and successful tool for search and optimization problems. However, for most of the real life problems the fitness calculation of an individual takes too much time, Luckily, the fitness calculation of the individuals does not depend on each other, so it is readily parallelizable.

Since the computational resources on the grid are distributed far from each other, the communication costs are more than the ones on a cluster. Therefore, the research on parallel GA is mainly focused on cluster computing rather than grid. The examples on grid are usually designed to solve specific problems and have their own constructed grids in which the computers in the grid are much more nearer to each other than in an ordinary grid. This gives the researchers the ability to manage and configure the components in their grid. Moreover, it also improves the performance with reducing the latency.

In this work, we introduced a framework aiming to make it easier for developers using GA to port their applications to grid that can provide more computational resources for a single user than a cluster. With this framework, users that are not familiar with either parallel programming or grid computing can enjoy the power of parallel programming. Because, GridAE abstracts the message passing from the user and manages it internally. The users also do not need to be familiar with the middleware running while using the framework.

GridAE also provides a library that supplies required data structures to represent individual's chromosomes. Commonly used genetic operations for selection(roulette wheel and tournament), crossover (one-point, two-point and uniform) and mutation (point, flip and swap) are already implemented. If these operations are not suitable for an application, the developer can implement her/his methods and use them. The user should also define the fitness function that the application is going to use.

The framework is tested with a problem from aerospace science. GridAE is used to find the optimum parameters that define the shape of a missile which can travel the longest distance. The performance of the framework is examined and shown to be rather high. It also shows that the applications which have bigger population sizes are more convenient with GridAE.

REFERENCES

[1] *GALib: a C + + Library of Generic Algorithm Components*. http://lancet.mit.edu/ga/. Last visited in 02/12/2010.

[2] *gLite User Guide*. https://edms.cern.ch/file/722398/1.3/gLite-3-UserGuide.pdf. Last visited in 02/12/2010.

[3] *GridAE: A Grid-based Framework for Artificial Evolution Applications*. http://gridae.ceng.metu.edu.tr/. Last visited in 02/12/2010.

[4] *The Large Hadron Collider*. http://public.web.cern.ch/public/en/lhc/LHC-en.html. Last visited in 02/12/2010.

[5] *OpenSSl*. http://www.openssl.org/. Last visited in 02/12/2010.

[6] *South East European Grid Project 2*. http://www.see-grid.eu/. Last visited in 02/12/2010.

[7] *Worldwide LHC Computing Grid*. http://lcg.web.cern.ch/LCG/digital.htm. Last visited in 02/12/2010.

[8] M. G. Arenas, P. Collet, A. E. Eiben, M. Jelasity, J. J. M. Guervós, B. Paechter, M. Preuß, and M. Schoenauer. A framework for distributed evolutionary algorithms. In *Proceedings of the 7th International Conference on Parallel Problem Solving from Nature*, PPSN VII, pages 665–675, London, UK, 2002. Springer-Verlag.

[9] George E. P. Box. Evolutionary operation: A method for increasing industrial productivity. *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, 6(2):81–101, 1957.

[10] H. J. Bremermann. Optimization through evolution and recombination. *Self-Organizing Systems*, 1962.

[11] E. Cantu-Páz. A survey of parallel genetic algorithms. *Calculateurs Paralleles*, 10, 1998.

[12] J. J. Durillo, A.J. Nebro, F. L., and E. Alba. A study of master-slave approaches to parallelize NSGA-II. In *IPDPS*, pages 1–8, 2008.

[13] L.J. Fogel, A.J. Owens, and M.J. Walsh. *Artificial intelligence through simulated evolution*. Wiley, Chichester, WS, UK, 1966.

[14] Ian T. Foster. The anatomy of the grid: Enabling scalable virtual organizations. In *Proceedings of the 7th International Euro-Par Conference Manchester on Parallel Processing*, Euro-Par '01, pages 1–4, London, UK, 2001. Springer-Verlag.

[15] Ian T. Foster. What is the grid? a three point checklist. 2002.

[16] C. Gagné and M. Parizeau. Open BEAGLE: A new versatile C++ framework for evolutionary computation. New York, NY, USA, 2002.

[17] C. Gagné, M. Parizeau, and M. Dubreuil. A robust master-slave distribution architecture for evolutionary algorithms. *Late Breaking Papers of GECCO 2003*, 2003.

[18] J. Goux, S. Kulkarni, M. Yoder, and J. Linderoth. An enabling framework for master-worker applications on the computational grid. In *Proceedings of the 9th IEEE International Symposium on High Performance Distributed Computing*, HPDC '00, pages 43–50, Washington, DC, USA, 2000. IEEE Computer Society.

[19] J. H. Holland. *Adaptation in Natural and Artificial Systems*. University of Michigan Press, Ann Arbor, MI, USA, 1975.

[20] M. Jelasity, M. Press, and B. Paechter. A scalable and robust framework for distributed applications. volume 2 of *CEC '02*, pages 1540–1545, Los Alamitos, CA, USA, 2002. IEEE Computer Society.

[21] P. Kacsuk and G. Sipos. Multi-grid, multi-user workflows in the P-GRADE portal. *Journal of Grid Computing*, 3:221–238, 2005.

[22] David B. Keator, J. S. Grethe, D. Marcus, B. Ozyurt, S. Gadde, Sean Murphy, S. Pieper, D. Greve, R. Notestine, H. J. Bockholt, and P. Papadopoulos. A national human neuroimaging collaboratory enabled by the biomedical informatics research network (BIRN). *IEEE Transactions on Information Technology in Biomedicine*, 12 (2):162–172, 2008.

[23] C. Kesselman and I. Foster. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers, San Fransisco, CA, USA, 1998.

[24] Melanie Mitchell. *An introduction to genetic algorithms*. MIT Press, Cambridge, MA, USA, 1996.

[25] A. J. Nebro, G. Luque, F. Luna, and E. Alba. DNA fragment assembly using a grid-based genetic algorithm. *Computers and Operations Research*, 35:2776–2790, 2008.

[26] H. Ng, D. Lim, Y. Ong, B. Lee, L. Freund, S. Parvez, and B. Sendhoff. A multi-cluster grid enabled evolution framework for aerodynamic airfoil design optimization. In *Advances in Natural Computation*, volume 3611 of *Lecture Notes in Computer Science*, pages 1112–1121. Springer Berlin / Heidelberg, 2005.

[27] I. Rechenberg. *Evolutionsstrategie : Optimierung technischer Systeme nach Prinzipien der biologischen Evolution*. Problemata. Frommann-Holzboog, Stuttgart-Bad Cannstatt, 1973.

[28] K. C. Tan, A. Tay, and J. Cai. Design and implementation of a distributed evolutionary computing software. *IEEE Transactions on Systems, Man, and Sybernatics, Part C*, 33:325–338, 2003.

[29] T. Tannenbaum, D. Wright, K. Miller, and M. Livny. Condor – a distributed job scheduler. In Thomas Sterling, editor, *Beowulf Cluster Computing with Linux*. MIT Press, October 2001.

APPENDIX A

JOB MANAGEMENT WITH GLITE

In order to perform any job management action the users should communicate with the gLite middleware installed on UIs. This middleware has a command line interface (CLI) for the users. To submit a job, the following command is used:

```
> glite-wms-job-submit -a myjob.jdl
```

The output of the command will be similar to this:

```
Connecting to the service https://wms.ulakbim.gov.tr:7443/glite_wms_wm
proxy_server



=================== glite-wms-job-submit Success ====================

The job has been successfully submitted to the WMProxy
Your job identifier is:

https://wms.ulakbim.gov.tr:9000/44CBrXjXvuGDdwVTodwwvg

======================================================================
```

The job identifier is a unique string and useful for getting information about the job from the system with the following command:

```
> glite-wms-job-status https://wms.ulakbim.gov.tr:9000/44CBrXjXvuGDdwV
```

```
Todwwvg

===================== glite-wms-job-status Success ===================
BOOKKEEPING INFORMATION:

Status info for the Job : https://wms.ulakbim.gov.tr:9000/44CBrXjXvuGD
dwVTodwwvg
Current Status:     Running
Status Reason:      Job successfully submitted to Globus
Destination:        ce64.ipb.ac.rs:2119/jobmanager-pbs-seegrid
Submitted:          Sun Nov 14 18:50:52 2010 EET
======================================================================
```

Here, we see that our job is running. After some time, if nothing goes wrong we will get the following output from the *glite-wms-job-status* command:

```
===================== glite-wms-job-status Success ===================
BOOKKEEPING INFORMATION:

Status info for the Job : https://wms.ulakbim.gov.tr:9000/44CBrXjXvuGD
dwVTodwwvg
Current Status:     Done (Success)
Logged Reason(s):
    -
    - Job terminated successfully
Exit code:          0
Status Reason:      Job terminated successfully
Destination:        ce64.ipb.ac.rs:2119/jobmanager-pbs-seegrid
Submitted:          Sun Nov 14 18:50:52 2010 EET
======================================================================
```

Now that our job is finished, we can retreive the output files:

```
> glite-wms-job-output --dir $PWD https://wms.ulakbim.gov.tr:9000/44CB
```

```
rXjXvuGDdwVTodwwvg

Connecting to the service https://wms.ulakbim.gov.tr:7443/glite_wms_wm
proxy_server

======================================================================

JOB GET OUTPUT OUTCOME

Output sandbox files for the job:
https://wms.ulakbim.gov.tr:9000/44CBrXjXvuGDdwVTodwwvg
have been successfully retrieved and stored in the directory:
/home_eymir/aketenci/aketenci_44CBrXjXvuGDdwVTodwwvg

======================================================================
```

With $--dir$ argument, we specify the directory that the output files will be copied to. If the user wishes to cancel a job that is not finished yet, the following command should be used:

```
> glite-wms-job-cancel https://wms.ulakbim.gov.tr:9000/44CBrXjXvuGDdwVT
odwwvg
```

These four gLite commands are usually enough for job management. If the job fails after beginning to run, the output can still be retrieved. If this situation occurs, the error file might be useful to find the source of the problem.