Framework for Workflow Gridication of Genetic Algorithms in Java

Boro Jakimovski, Darko Cerepnalkoski, and Goran Velinov

University Sts. Cyril and Methodius,
Faculty of Natural Sciences and Mathematics,
Institute of Informatics,
Arhimedova bb, 1000 Skopje, Macedonia
{boroj,darko,goranv}@ii.edu.mk

Abstract. In this paper we present new Java framework for Gridification of Genetic Algorithms. The framework enables easy implementation of Genetic Algorithms and also enables researchers easy and stable usage of the Grid for their deployment. The design of the framework was based on principles that make it very open and extensible. The Grid components use pure Java implementation of Grid job submission and retrieval for the Glite grid middleware by using Web Services (WS). The framework was tested on the SEEGRID testbed. Using this framework we have developed a pilot application for optimizing data warehousing VIS problem.

1 Introduction

Evolutionary algorithms (EA) are a computational model inspired by the natural process of evolution. They have been successfully used for solving complex optimization problems. Genetic algorithms (GA), a subclass of EA, search for potential solution by encoding the data into a chromosome-like structure. The search is done over a set of chromosomes (population) with repetitive application of recombination, mutation and selection operators until certain condition is reached. One repetition is called a generation.

Usually the search for a solution using GA is a long and computationally intensive process. Fortunately the GA is easily parallelized using data partitioning of the population among different processes. This kind of parallelism ensures close to linear speedup, and sometimes super-linear speedup. Over the past years many variants of parallelization techniques are exploited for parallelization of GA [1][2].

Computational Grids [3] represent a technology that enables ultimate computing power at the fingertips of users. Today, Grids are evolving in their usability and diversity. New technologies and standards are used for improving their capabilities.

Gridified genetic algorithms have been efficiently used in the past years for solving different problems [4][5]. The Grid architecture resources are very suitable for GA since the parallel GA algorithms use highly independent data parallelism. Gridification and effective utilization of the powerful Grid resources represent a great challenge. New programming models need to be adopted for implementation of such

M. Bubak et al. (Eds.): ICCS 2008, Part III, LNCS 5103, pp. 463–470, 2008.

parallel algorithms. In this paper we present the Java GA Grid Framework that will make this utilization easier and more efficient.

The rest of this paper is organized as follows. In Section 2, we present the architecture of the Java Grid framework for GA (JGFGA). Section 3 describes the Grid components of the framework. Real usage of the framework is presented in Section 4 as we present the pilot application GROW. Finally, Section 5 concludes this paper and gives future development issues.

2 Java Framework for GA

In this section we will present the architecture of the new Java Grid framework for Genetic Algorithms responsible for implementation of GA. First we will start with the architecture design issues and later progress towards presentation of the framework components.

2.1 Designing the Framework

Various GA frameworks have been developed in the past years, implemented in different programming languages. The reason for developing new GA framework was not for introducing new model for parallelizing GA, but to enable easier implementation, better portability and grid execution of parallel GA. The framework is implemented in Java because of its platform independence and good OO properties.

The framework design is founded on the following concepts: *modularity* – the framework should be defined as collection of base components that enable modular composition when designing a solution; *extensibility* – one of the main aspects influencing the development of the framework is to provide base components that enable easy extension of the framework functionality, i.e. new algorithms or new chromosome type with new evaluation function can easily implement this by extending classes or implementing interfaces that are part of the framework; *flexibility* – of the way the framework is used either by simply using already defined classes or by implementing extensions, or by choosing parallel or serial execution and other similar aspects; and *adaptability* – a framework should be as adaptable as possible because the process of implementation of GA can be divided into several phases, so the framework should enable researchers to easily implement new or adapt existing solutions by changing phase implementation.

2.2 Framework Components

The Framework organization can be divided in two parts: GA implementation and GA gridification. The GA implementation part of the framework consists of components that give easy and custom implementation of GA optimizations. On the other hand the GA gridification components enable workflow grid parallelization of the execution of the implemented GA optimizations. We continue with more detailed description of both parts.

The framework is organized in three main packages: gridapp.grid, gridapp.ga and gridapp.util. The focus in this section will be the *gridapp.ga* package containing the core GA classes. Some of the classes are abstract classes, intended for further

implementation and specialization. Other classes are normal classes that implement some aspects of the GA execution, which are not designed to be extended. Most common implementations of the abstract classes can be found in *gridapp.ga.impl* package. The package *gridadd.util* contains utility classes used by many different classes in the framework. The last package *gridapp.grid* will be discussed in the next section. We continue with the presentation of the *gridapp.ga* and *gridapp.ga.impl* class design and available classes.

2.3 GA Classes

Fig. 1a presents the design of the core classes that implement or allow the implementation of GA in the framework. We will briefly describe this design.

The abstract class *Gene<T>* represents one gene. Every real implementation of a gene needs to inherit this class and implement needed methods. The reason why *Gene* class is defined as generic class of type *T* is because one gene is an array of alleles. *T* is the type of one allele. Available implementations of the *Gene<T>* class that can be found in the *gridapp.ga.impl* package are shown in Fig. 1b.

The abstract class *Chromosome<T extends Gene>* represents one chromosome. It can be seen that chromosomes are derived using one *Gene* type, and the main purpose of the *Chromosome* class is to act as a container of *Genes* and implement structure for gene organization. The only subclass included in the framework that implements the *Chromosome* class is the *ArrayChromosome<T>* class that organizes the genes into an array. If this is not sufficient the users can implement different structures for the chromosomes. The *Chromosome* class has several methods for accessing and manipulation of its genes. Mainly these are methods for accessing the genes using indexes, manipulation of the structure such as cloning or copying parts of the gene which later are used for recombination or mutation.

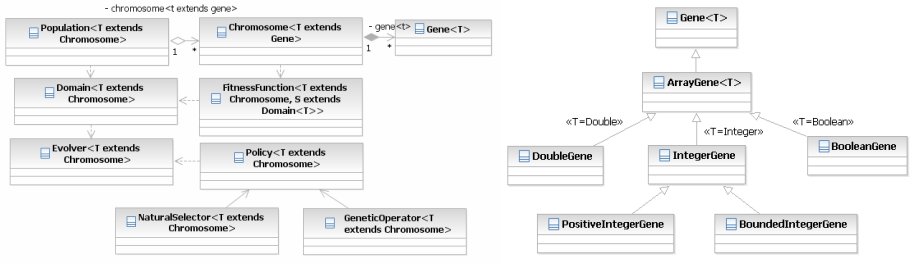

Fig. 1. a) Core Genetic Algorithm class design and connection; b) Gene<T> class hierarchy

The class *Population<T extends Chromosome>* represents single population of chromosomes. This class is not abstract since it is only used as a container of chromosomes, with standard methods for data access and manipulation.

The class *Domain<T extends Chromosome>* plays key role in GA execution. This class represents a structure that holds additional information on the chromosomes and genes needed for their interpretation. As shown previously, genes and chromosomes

carry only raw data organized in certain data structure. The *Domain* class adds order and meaning to this data, usually by storing additional data specific for the problem. The class is not abstract, since simple problems that do not need additional information can step over this class.

The class *FitnessFunction<T extends Chromosome, S extends Domain<T>>* is the class that implements the fitness function evaluation for a certain chromosome type and domain over that chromosome type. The *Domain* is critical since it holds the main information for interpretation of the chromosomes. The *Domain* class also holds a link towards its *FitnessFunction* and *Population*, and hence represents a description of the whole problem we are solving. This makes it one of the key ingredients to the class *Evolver* that implements the actual process of evolution of the population.

Another class that is used by the *Evolver* is the *Policy<T extends Chromosome>* class. The *Policy* class specifies the rules the *Evolver* will use to implement the optimization. In another words the *Policy* is a "program" that *Evolver* will follow. The policy uses the subclasses of the abstract classes *NaturalSelector* and *GeneticOperator* to specify which operators will be used to create new chromosomes, and which selectors will be used to select the new chromosomes.

The classes *NaturalSelector* and *GeneticOperator* are inherited in many classes from the *gridapp.ga.impl* package. Some of them are: *RouletteWheelSelector*, *TournamentSelector*, *LinearRankSelector*, *EliteSelector*, *RandomSelector*, *CrossoverOperator* and *MutationOperator*.

3 Gridification of the GA Framework

In this section we will present the Grid components of the Grid Java framework for GA. Gridification of GA optimizations can be divided in two aspects. The first aspect is the choice of parallel GA technique. The second aspect is concerned with the underlying Grid connectivity with the Glite grid middleware.

3.1 Parallel Genetic Algorithms

The Parallel Genetic Algorithms (PGAs) are extensions of the single population GA. The well-known advantage of PGAs is their ability to perform speciation, a process by which different subpopulations evolve in diverse directions simultaneously. They have been shown to speed up the search process and to attain higher quality solutions on complex design problems [6][7].

There are three major classes of PGA: Master-slave, Cellular and Island. Master-slave parallelization uses single population of chromosomes and parallelizes only the chromosome evaluation part of the optimization. This makes it suitable for usage where the parallel environments have shared memory. Cellular PGA also consists of single chromosome population, but the computation can be spatially structured. This is mostly suitable for massively parallel systems, consisting of large number of processing elements organized in a topology, which is followed by the PGA. Most widely used and most sophisticated PGA is the Island PGA, or in other words Multi-population PGA. This approach enables parallel nearly-independent execution of populations. The only connection between the populations is occasional migration of chromosomes. This PGA is suitable for message passing parallelism environments.

The nature of the Grid makes it best suited for Island PGA for achieving high performance parallelism. The available cluster resources can be used using MPI or similar parallelization mechanism. This approach is extended in the Grid-enable Hierarchical PGA (HPGA)[8] which uses two level of population distribution. The first level makes several independent islands distributed over several clusters. On each cluster several parallel jobs are started which take part of the island (sub-island) on which they run the GA. After several iterations the sup-islands are rejoined and mutation is done. This process is repeated. Another positive aspect of using several clusters in parallel is having bigger population and separation into islands might increase diversity and speed up convergence.

3.2 Grid Workflow Genetic Algorithms

The JGFGA implements the PGA by using workflow execution. A grid workflow is a directed acyclic graph (DAG), where nodes are individual jobs, and the vertices are inter-job communication and dependences. The workflow inter-job communication is implemented by input and output files per job. More precisely the first job outputs data into files, and after the job terminates the files are transferred as input files to the second job. gLite WMS service takes care of the job scheduling and file transfers.

The JGFGA enables implementation of PGA by menas of four classes (jobs): *Breeders*, *Migrator*, *Creator* and *Collector,* all members of the *gridapp.grid* package. Breeders take as input a domain file and policy file. The domain file is simply Java serialized Domain object, while policy file is a java serialized Policy object. Having a population and a policy the Breader calls the Evolver and iterates several generations. The resulting population is again serialized into a domain file. Migrators on the other hand take as input several domain files, execute the inter-population migrations and output one resulting domain. The Creator and Collector classes enable easy creation of new random populations and collect several populations into one.

Having this four classes, currently we have implemented the class *JobGraph<T extends Chromosome>* that enables automatic workflow generation. Generated workflows contain several iterations of Breeder and Migrator jobs where each node is mapped to a separate Grid job. One iteration of breeding is called an epoch. An example of a sample workflow is shown on Fig. 2. The same class enables users to generate JDL (Job Description Language) files specifying the workflow for gLite grid middleware. These files are later submitted using the Grid submission tools. The parameters that can be given to JobGraph in order to model the Grid execution are: number of islands, number of epochs and migration width. Additional parameters that specify the population and policy are number of iteration per epoch and size of a single population. Further parameters that define Grid execution characteristics are the Retry counts which make the workflow more resilient when some job hits problematic Grid site and fails to execute.

The scheduling and execution of the generated workflows are not controlled by the framework. The Grid mapping decisions are done by the gLite WMS service. We plan in later developments to introduce additional properties for defining constraints in the JobGraph class that will enable guided mapping of jobs to Grid resources.

It can easily be seen that JGFGA has superior flexibility than HPGA. Most important disadvantage is the restrictions of HPGA for inter island material exchange.

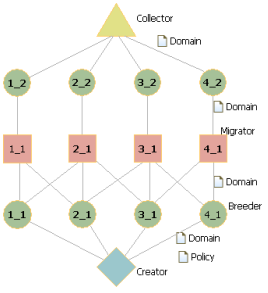

Fig. 2. Sample Grid Workflow GA. The circles are breeders and squares are Migrators.

3.3 Java Grid Framework Components

The Grid components of the framework that enable usage of the gLite Grid testbeds are based on the Workload Management System (WMS) and Logging and Bookeeping (LB) Web Services (WS). The framework is bundled with several libraries that support the WMSWS and LBWS connectivity. Additional functionality of the framework is Grid authentication by using VOMS-proxy-init. This makes the framework completely independent from the Glite UI installation and enables users to use it from any available platform.

In order for the framework to work, the user only needs to provide his/hers valid certificate, and put it in *.globus* directory in the his/hers home. For convenience we decided that it is best to provide the certificate in *.pkcs12* format, even though the framework will work if the certificate is in *.pem* format. Additionally the user needs to specify the directory with the CA certificates and vomses configuration and voms certificates. Best place for this directories are *.globus/certificate*, *.globus/vomses*, *.globus/vomsdir*.

For simplicity reasons the Grid components are organized in a single class called *GridServices*. The *GridServices* class offers the following methods: *buildProxy*, *isProxyValid*, *jobListMatch*, *jobSubmit*, *dagJobSubmit*, *getJobStatus*, *getJobOutput* and *jobPurge*. The constructor of the class requires for a *Properties* object that specify the Grid configuration parameters: *tmpDir*, *userGlobusDir*, *proxyFile*, *vomsesDir*, *vomsDir*, *caDir*, *delegationId*, *WMProxyURL* and *LBProxyURL*. If some of the parameters are not supplied the default values are assumed.

4 Implementation of Pilot Application

The pilot application that was successfully implemented using the Java Grid Framework for GA is the GRid Optimization for data Warehousing (GROW). The application problem area is VIS optimization of Data Warehouses. We choose this application as it was addressed in our previous research on GA optimizations [9][10].

The performance of the system of relational data warehouses depends of several factors and the problem of its optimization is very complex. The main elements of a system for data warehouse optimization are: definition of solution space, evaluation function and the choice of optimization method. The solution space includes factors

relevant for data warehouse. Previous research has shown performance and quality aspects of different approaches towards solving this problem. Some of them use genetic algorithms for the search for optimal result [10].

The focus on this paper is not to develop new optimization algorithm for the VIS problem, but to implement the problem using the Java Grid Framework for GA.

ViewGene.java

```
Public class ViewGene
  extends Gene<Boolean> {
  static Random rand = new Random();
  int size = 0;
  BitSet view;
  // Constructors …
  @Override
  public Boolean getAllele(int index) {
    return this.view.get(index);
  }
  @Override
  public void setAllele(Boolean allele, int index) {
    this.view.set(index, allele);
  }
  ...
  @Override
  public void mutate(int index) {
    this.view.flip(index);
  }
  @Override
  public int size() {
    return this.size;
  }
  public Iterator<Boolean> iterator() {
    return new BitSetIterator(size, this.view);
  }
}
```

ViewFunction.java

```
Public class ViewFunction
  extends FitnessFunction<ViewChromosome, ViewDomain> {
  ViewChromosome ch;
  double value;
  private void positiveEffect() { … }
  private void negativeEffect() { … }
  public double setFitnessValue(ViewChromosome chromosome) {
    ch = chromosome;
    positiveEffect();
    negativeEffect();
    chromosome.fitnessValue = value;
    return value;
  }}
```

Fig. 3. Implementation of GROW GA

The implementation of GROW follows the path of every GA implementation in JGFGA. It starts with the implementation of *Gene* and *Chromosome* and later continues with the implementation of the *FitnessFunction* class for the problem in conjunction with the *Domain* class. All classes need to be implemented as extensions of classes from the framework mentioned above.

The GROW application implements *ViewGene* using *BitSet* structure and overriding the required methods as shown in Fig. 3. The *ViewChromosome* is extension of the *ArrayChromosome* class and uses *ViewGene* objects. The evaluation *ViewFunction* uses the implemented *ViewChromosome* and *ViewGene* classes and is implements by defining positive and negative functions (Fig. 3). At the end the *Domain* is implemented to hold the *Population*, *ViewFunction* and additional information. In order to enable easier gridification prior to the implementation of the applications frontend the GROW GA was packed into a single jar archive.

The developed pilot application for data warehousing is currently running on the SEE-GRID testbed. We are currently using the application for our future research in the field of GA for Data Warehousing.

5 Conclusion and Future Work

In this paper we presented the new Java Grid Framework for GA. The framework represents a powerful tool in the hands of researchers, enabling easy creation of Gridified GA optimizations.

Our current research is oriented on testing different kinds of approaches for Grid paralellizations of GA, mainly focused on what kind of models of workflow parallelization (based on previously defined parameters) can be most effective when using the Grid infrastructure.

Further development of the JGFGA is oriented towards implementation of grid data management and job wrapping per island in order to overcome the problem with job failure. The job wrapping can be easily facilitated since the input and output files of each job are in the same format (Domain files). Hence in failed nodes we can skip the epoch iteration and just copy the input files as output files. This approach will decrease the convergence of the optimization process and might produce worse final results, but will increase Grid job success rate which is the biggest problem with gridified workflows nowadays.

References

1. Nowostawski, M., Poli, R.: Parallel Genetic Algorithm Taxonomy. In: Proceedings of the Third International conference on knowledge-based intelligent information engineering systems (KES 1999), Adelaide, pp. 88–92 (1999)
2. Cantu-Paz, E.: A Survey of Parallel Genetic Algorithms. Calculateurs Paralleles, Reseaux et Systems Repartis 10(2), 141–171 (1998)
3. Foster, I., Kesselman, C. (eds.): The Grid: Blueprint for a New Computing Infrastructure. Morgan Kaufmann Publishers, San Francisco (1999)
4. Herrera, J., Huedo, E., Montero, R.S., Llorente, I.M.: A Grid-Oriented Genetic Algorithm. In: Sloot, P.M.A., Hoekstra, A.G., Priol, T., Reinefeld, A., Bubak, M. (eds.) EGC 2005. LNCS, vol. 3470, pp. 315–322. Springer, Heidelberg (2005)
5. Imade, H., Morishita, R., Ono, I., Ono, N., Okamoto, M.: A Grid-Oriented Genetic Algorithm Framework for Bioinformatics. New Generation Computing 22(2), 177–186 (2004)
6. Cui, J., Fogarty, T.C., Gammack, J.G.: Searching databases using parallel genetic algorithms on a transputer computing surface. Future Generation Computer Systems 9(1), 33–40 (1993)
7. Sena, G.A., Megherbi, D., Isern, G.: Implementation of a parallel genetic algorithm on a cluster of workstations: travelling salesman problem, a case study. Future Generation Computer Systems 17(4), 477–488 (2001)
8. Lim, D., Ong, Y.-S., Jin, Y., Sendhoff, B., Lee, B.-S.: Efficient Hierarchical Parallel Genetic Algorithms using Grid computing. Future Generation Computer Systems 23(4), 658–670 (2007)
9. Velinov, G., Kon-Popovska, M.: Solving View and Index Selection Problem Using Genetic Algorithm. In: Proc. Second Balkan Conference in Informatics, BCI, pp. 180–192 (2005)
10. Velinov, G., Gligoroski, D., Kon-Popovska, M.: Hybrid greedy and genetic algorithms for optimization of relational data warehouses. In: Proc. of the 25th IASTED International Multi-Conference: artificial intelligence and applications, pp. 470–475 (2007)