A Parallel Genetic Algorithm Framework for Cloud Computing Applications

ELENA APOSTOL, IULIA BĂLUŢĂ, ALEXANDRU GORGOI and VALENTIN CRISTEA, University Politehnica Bucharest

Genetic Algorithms(GA) are a subclass of evolutionary algorithms that use the principle of evolution in order to search for solutions to optimization problems. Evolutionary algorithms are by their nature very good candidates for parallelization, and genetic algorithms do not make an exception. Moreover, researchers have stated that genetic algorithms with larger populations tend to obtain better solutions with faster convergence. These are the main reasons why they can benefit from a MapReduce implementation. However, research in this area is still young, and there are only a few approaches for adapting genetic algorithms to the MapReduce model.

In this article we analyze the use of subpopulations for the GA MapReduce implementations. MapReduce naturally creates subpopulations, and if this characteristic is properly explored, we can find better solutions for genetic algorithm parallelization. In this context, we propose new models for two well know genetic algorithm implementations, namely island and neighborhood model. Our solutions are using the island model, with isolated subpopulations, and the neighborhood model, with overlapping subpopulations. We incorporate these solutions in a framework, that makes the development of Cloud applications using Genetic Algorithm easier.

Categories and Subject Descriptors: C.2.4 [**Distributed Systems**]: Cloud Computing—*Genetic Algorithms*

Additional Key Words and Phrases: Cloud Applications, Map-Reduce, Parallel Genetic Algorithms, sub-populations

1. INTRODUCTION

In the past few years, the increase in the information available on the Internet and the large volumes of information captured by complex scientific equipment in domains like High Energy Physics or Astronomy have determined researchers to explore the domain of data intensive computing. Data-intensive frameworks were developed to deal with these situations. Among them, the MapReduce framework and Hadoop - its open source implementation, are widely used in research these days.

The power of the MapReduce framework comes from the fact that it splits the data into smaller blocks that can be processed in parallel by the mappers and then transmitted to the reducers for merge. This approach is similar to SIMD processors. For the user of the framework, the process is translated into two functions: a map function and a reduce function. The framework takes care of splitting the data into chunks and passing the results from the mappers to the reducers. The intensive parallel processing nature of this framework makes it a good candidate for execution of parallel algorithms.

Genetic algorithms are a subclass of evolutionary algorithms and are based on the darwinian principles of evolution and natural selection. A genetic algorithm searches for a solution to a problem by evolving a population of individuals towards fitness maximization. The essential aspects of any genetic algorithm are: how to represent a solution to the problem as an individual (encoding), how to evaluate how good an individual is (fitness) and how to evolve better individuals from the existing ones (selection, crossover).

Evolutionary algorithms are by their nature very good candidates for parallelization, and genetic algorithms do not make an exception. Moreover, researchers have stated that genetic algorithms with larger populations tend to obtain better solutions with faster convergence [D. Huang 2010], [Witt 2008]. These are the main reasons why they can benefit from a MapReduce implementation. However,

research in this area is still young, and there are only a few approaches for adapting genetic algorithms to the MapReduce model [Verma 2010], [Xavier Llora and Goldberg 2010], [D. Huang 2010], [C. Jin]. These approaches, however, explore but a small part of the parallelization approaches existing in the field. Moreover, they do not discuss the use of subpopulations in their MapReduce implementations. This is an important discussion, as MapReduce naturally creates subpopulations, and if we manage to properly explore and exploit this characteristic we can find better solutions for genetic algorithm parallelization. We propose two alternative solutions to the implementation suggested in [Verma 2010]: the island model, with isolated subpopulations, and the neighborhood model, with overlapping subpopulations. We proposed and implemented these methods as part of a framework for genetic algorithms in Hadoop. The purpose of the framework is to make the development of genetic algorithms easier and to enhance in this way the research in this area. The framework has two different approaches for adapting genetic algorithms to MapReduce: a coarse grained approach, that follows the island model and a distributed fitness evaluation approach, with three possible models: global population, island model and neighborhood model.

The rest of the paper is structured as follows. In section II we present some relevant related work. In section III we present the architecture of the framework. In section IV we describe the methods of applying map reduce. We then describe the distributed fitness evaluation mechanism and the three models that we developed using this method: the global population model, the island model and the neighborhood model. In the fifth section, we present the experiments that we have conducted for the distributed fitness evaluation method. We then evaluate and interpret those results, and, in the last section, we provide conclusions and some ideas for future work.

2. RELATED WORK

Using map reduce for running genetic algorithms has been a subject of research in the last few years. In [C. Jin], an adapted model of map reduce with an additional reduce step, has been proposed in order to deal with iterative algorithms like evolutionary algorithms. However, there were some drawbacks regarding that method, mainly the fact that there was a lot of serial execution time and the fact that this model does not use the benefits that map reduce offers but instead forces a new model.

In [Verma 2010], it is argued that there is no need for adapting the map reduce model for genetic algorithms, but instead we should try to adapt the genetic algorithms to fit into the model. The author succeeds in doing just that and introduces a new model of fine-grained parallel genetic algorithm adapted for MapReduce. In this model, each iteration of the algorithm is transformed into a map reduce job. He also extends his model for two classes of GAs: compact and extended compact genetic algorithms.

This work becomes the starting point of other projects, among which the most notable is [D. Huang 2010]. In this article, the job shop scheduling problem is tested using the model in [Verma 2010], slightly adapted and tuned. The main merit of this work is that it introduces the idea that map reduce might be fit to work with very large populations, that, correctly handled, can lead to faster convergence with very good solutions. It is common for map reduce to work with big data, so this approach suits it.

Our work differs from these previous works in that it incorporates these approaches in a more generic implementation, inside a framework for genetic algorithms in Hadoop. Another contribution that we brought was the discussion regarding the imminent separation of the global population into subpopulations handled by reducers, and the way that we could use this to implement some well-known models for parallel genetic algorithms. We note that implementing these models involve no additional costs, but provides good results for some classes of problems.

3. THE PROPOSED GA FRAMEWORK ARCHITECTURE

In order to create a distributed genetic algorithm we must consider the independent elements of the algorithm which will be executed in parallel. Because on Hadoop each mapper takes independent tasks we considered the multiple population coarse grained GA model. Each mapper takes as input a subpopulation and for each individual computes the fitness function and takes care of crossover and mutation. The Reducer's job is to migrate individuals from one subpopulation to another. This way we enlarge the solution space by bringing novelty to populations. We denoted our framework as IGAF, which stands for "Improved Genetic Algorithm Framework".

IGAF is designed on three different levels regarding to the accessibility and configuration from the user's point of view. It consists of several interconnected modules, as depicted in Figure 1.

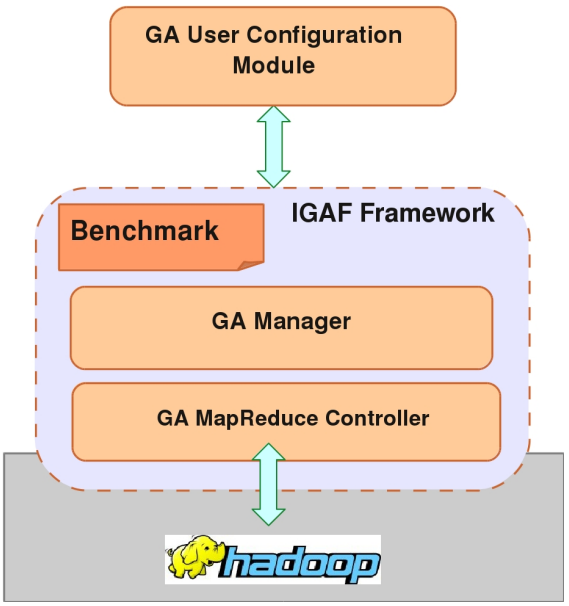

Fig. 1. **IGAF Framework Architecture**

At the upper layer stays the *GA User Configuration* module, which is the interface that the user has towards the configuration of the algorithm implementation. Here, the user can set the parameters of the algorithm for the next two levels of the implementation. The parameters that can be set using the configuration model can also be spread into two categories: genetic algorithms parameters, map reduce implementation parameters. The first category consists of the genetic parameters, such as: mutation, crossover, selection and other parameters needed for the tuning of the genetic algorithms: mutation rate, stop criteria, population size, etc. In the second category there are parameters specific to the chosen map reduce implementation: migration percentage, retain best individuals, distributed model, partitioner implementation, etc.

The next level is represented by the *GA Manager*, which contains the logic for the evolutionary algorithms. This module controls the different types of behaviors that can be impose to the Map Reduce stage, such as the *'distributed fitness evaluation'* or the *'island model'*.

In the third level, and the closest to the Hadoop core, there is the *GA MapReduce Controller* which contains two sub-modules. The first sub-module manages the mappers, the reducers and the parti-

tioner. It provides different behaviors for those components and an interface in order to dynamically add new types of behaviors. The second sub-module manages the Map Reduce pipeline, verifies the stop criteria and writes the result in the HDFS at the end of the algorithm.

When dealing with Evolutionary and in particular with Genetic Algorithms there are many parameters to be taken into consideration in order to obtain the best results. These parameters may refer to how the initial populations will be generated, what genetic algorithm model to be used, the migration frequency and rate, the percentage of individual to be removed for the next generation and so on. The Benchmark module gives the user the possibility to adjust these parameters in order to achieve a certain performance regarding the execution time or the solution accuracy.

4. FRAMEWORK FUNCTIONALITY

In this section we will describe the possible behaviors of the framework, based on the user configuration: the island behavior or the distributed fitness evaluation behavior. We adapted these models to better suit data-intensive Cloud applications. We will present a series of implementation details and the modifications we added to these models.

The Improved Island Model Coarse-grained Implementation

The island model works at subpopulation level. Each subpopulation will evolve independently, without any kind of interaction. After all the subpopulations evolved, a migration process will start. The migration process is responsible of diversifying each subpopulation with individuals from other subpopulation.

The framework will pass a subpopulation to each mapper and then, each mapper will evolve its subpopulation. This approach can be seen as a number of genetic algorithms running in parallel and solving the same problem. Initially the solution space of each mapper is different. After the number of pipeline steps increases the solution space of the mappers begins to resemble and to converge.

In a pipeline step the mapper tries to evolve the subpopulation and after the maximum number of generations is reached or the subpopulation cannot evolve anymore, it writes the output (the evolved subpopulation) for the reducers. Each subpopulation has a specific identifier. The reducer decodes the subpopulations and starts the migration process. The default migration process consists of selecting the best $p\%$ chromosomes of the subpopulation i and replacing the worst $p\%$ chromosomes of the subpopulation $i+1$. This behavior can, however, be changed by the user through the configuration section.

Depending on the genetic problem the migration frequency may vary. For example, a genetic problem can have the migration frequency in such way that the migration will be performed in each pipeline step, in random pipeline steps or in every k step.

When migration is complete the reducers write all the subpopulations on the HDFS file system as a final result or for the next pipeline step. The framework analyzes the output received from the reducers and decides if the solution satisfies the input conditions. If more work is required the next pipeline step is prepared and launched on Hadoop.

Since each mapper process starts with a different subpopulation, generic drift will tend to drive these populations into different directions. By introducing migration, the island model is able to exploit differences in various subpopulations; this variation represents a source of genetic diversity. However, migrating a large number of individuals too often may lead to destroying the global diversity (the islands will be less different). On the other hand, if migration doesn't occur often, it may lead to premature convergence of the subpopulations. So when dealing with the island model some aspects need to be considered:

—each reducer must choose the right subpopulations when exchanging individuals

—the migration frequency - how often individuals are exchanged
—the migration rate - the number of individuals exchanged between subpopulations
—the individuals chosen for exchange
—the individuals removed after the new individuals are received [Cantu-Paz 1998]

The user can use the Benchmark module in order to obtain a set of configuration parameters that will better suit his/her implementation.

An Efficient Distributed Fitness Evaluation

This approach is suitable when the fitness function is hard to compute and a lot of data is required for processing. The framework splits the entire population equally to mappers and each mapper will compute the fitness for every individual.

For the implementation of this model, we followed the work of A. Verma [Verma 2010], and added some improvements.

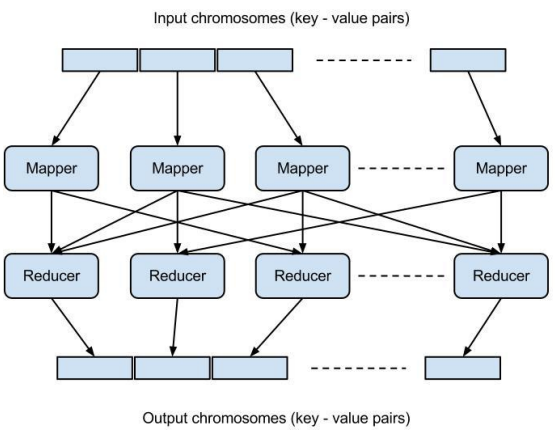

Fig. 2. **Distributed fitness evaluation - global population**

The main idea is that each iteration of the algorithm will be transformed in a MapReduce job. The mappers will evaluate the time consuming fitness function for each individual, and then the reducers will perform selection, crossover and mutation to produce a new population for the next generation. Figure 2 depicts how the populations are distributed among the mappers and the reducers. Besides these general lines, the mapper and reducer can have additional improvements, depending on the implemented problem.

Because our framework is meant to be generic enough to capture any problem and to allow the user to implement a genetic algorithm in his specific way, much of the mapper's functionality can be configured and changed without the user modifying the specific mapper class.

The Mapper. Each mapper receives a list of individuals and its main purpose is to evaluate the fitness function of each of them and to pass to the reducers the individuals with the fitness value set. Ideally, one mapper per individual would be used, and the calculated fitness should be enough computationally intensive to make the overhead of creating the mappers a small fraction of the total processing time.

Fitness evaluation is generally the first step performed in an iteration of the genetic algorithm, and it is also the part of the algorithm that is most suited for parallelization.

Besides the fitness evaluation, the mapper can include some local search mechanism in order to improve the chromosome that it receives. Also, the mapper can keep track of the best individual it has seen in order to treat it separately. In the implementation in [D. Huang 2010], the best individuals are assigned a separate key, in order to be handled by a specific reducer that would determine the best individual at each generation. In our framework, we wanted to leave the decision regarding the handling of the best individuals to the user, by a parameter in the Configuration class indicating that the best individuals should be stored separately.

The Reducer. The reducer has the main purpose of selecting the chromosome for crossover and then performing crossover and mutation to obtain a set of individuals to pass to the next generation.

The selection mechanism, as well as crossover and mutation operators are up to the user. In other implementations of genetic algorithms that we have seen, variants of tournament selection seem to be the top choices. The user of the framework can choose to use this method or selection, as well as other methods with a default implementation offered by the framework, or to implement his own selection method. The same case also happens for the genetic operators of crossover and mutation.

The Partitioner. The partitioner is the one that assigns a certain (key-value) pair representing the output of a mapper to a reducer. By default, the partitioner in Hadoop assigns tasks according to the hash of the key.

In [Verma 2010] it is argued that this implementation hurts the genetic algorithm, as it can make it converge slower or not converge at all. The reason for this is that the default partitioner sends all the individuals with the same key to the same reducer, creating in this way isolated partitions instead of simulating a global population. The proposed solution was to rewrite the partitioner to distribute randomly individuals from mappers to reducers, not taking in consideration the key assigned by the mapper. In our opinion, if a global population is wanted, then it can be achieved by making each mapper randomly generate a key for each chromosome from a list of keys with different hash values and leaving the default partitioner on. This requires that each mapper will know how many reducers are created, something that we can obtain in our mapper implementation from the Configuration object. Also, another reason why we can take this approach is that we considered IDs as keys, that are meaningful solely for the map-reduce process, as all the information about the individual is retained in the value field attached to the key.

However, the partitioner can be modified in order to handle the best individuals received from each mapper. In [D. Huang 2010], the best individual of each mapper was assigned a special key: *null*. The partitioner wouldn't have known how to handle this key, so a new partitioner was implemented, that would group all the individuals with *null* keys to the same reducer, that would evaluate them. We can also note that in this implementation, the authors considered sending the best individuals to a reducer along with other individuals normally assigned by the partitioner to that reducer based on their hash value. However, other variations are possible: a reducer can handle only best individuals or they can be spread uniformly through the reducers. In order to know which way is better, one would have to consider all the possible ways of handling the best individuals and carefully select the one that would best suit its problem.

Population Initialization. The initialization of the population can be a real performance issue if it is done in a serial manner. This is the reason why in parallel implementations of genetic algorithms it is common to also parallelize this part of the algorithm. In the other map reduce implementations that we have study, the solution that was used was to make the initialization in a separate map reduce job,

with the mappers generating the initial chromosomes. In this case the reduce phase is skipped and the output from the mappers is the final one. However, this approach creates a map reduce job just for initialization, which means additional overhead. We consider that this section can be incorporated in the first map reduce job. In this case, the mapper will receive a null value instead of an already generated chromosome. Also, before the evaluation of fitness, the mapper can generate a new individual that it will then evaluate normally.

Working with Subpopulations

The approach we presented so far follows the fine-grained model of parallel genetic algorithms. The original algorithm treats each individual independently and does not evolve subpopulations in isolation.

However, working at subpopulation level can be achieved by ensuring that at each generation, the mappers handling specific individuals and their offspring will always map the results to the same key, thus sending them to the same reducer for selection and mating. The exchange between the subpopulations can be done synchronously after a specific number of generations or asynchronously, each mapper at a random generation.

Isolated Subpopulations. In order to keep subpopulations isolated, we added a key attribute to all individuals, that will be also written in the representation in HDFS. Each reducer associates the key it receives to all individuals in its subpopulation and to all the offspring it creates from parents in the subpopulation. In this way, all individuals in the same subpopulation will be recognized by the mappers and always sent together to a common reducer. The mapper randomly generates a key in the interval $[0, number\_of\_reducers - 1]$ for each individual in the first generation. In the next generations, since the individual already has a key, it uses that key to associate the individual with calculated fitness to a certain reducer.

Migration can happen, as we already mentioned, synchronously or asynchronously. Migration means that a mapper decides (based on a migration probability or at a certain generation) to send some individuals to another reducer than the one indicated by their associated key. In this case, the mapper generates another random key for a migrated individual and resets the key in its representation.

Migration causes certain reducers to receive more individuals than the normal configured subpopulation size. In this case, the individuals with the lowest fitness values are dropped, to maintain a constant subpopulation size (we can say that the newly received individuals replace the weaker ones in the existing subpopulation). Other reducers might receive less individuals than the subpopulation size. In this case, the receiver produces more offspring to compensate the migrated individuals.

Overlapping Subpopulations. Also, based on this approach, another well-known model of parallel GAs can be obtained: the neighborhood model, also known as fine grain model, with small overlapping subpopulations. This can be obtained by allowing each mapper to have a specific set of keys, thus communicating with a number of reducers common to other mappers in the neighborhood. In this way, each mapper will communicate through the reducers with only a fraction of the other mappers, the neighborhood of the mappers overlapping in such a way that would allow evolution in all population.

We keep the idea presented in the island model, and we give to each individual a subpopulation id. The set of keys for each mapper is constructed by creating a neighborhood of consecutive ids derived from the subpopulation id of the individual. So, for each individual, there are a number of reducers that will receive it. In this respect, the subpopulation id can also be seen as a node id, if we think of a grid representation.

Following this analogy, each node in the grid is represented as a mapper and a reducer. This means that the number of mappers will be equal to the number of reducers in this implementation. The

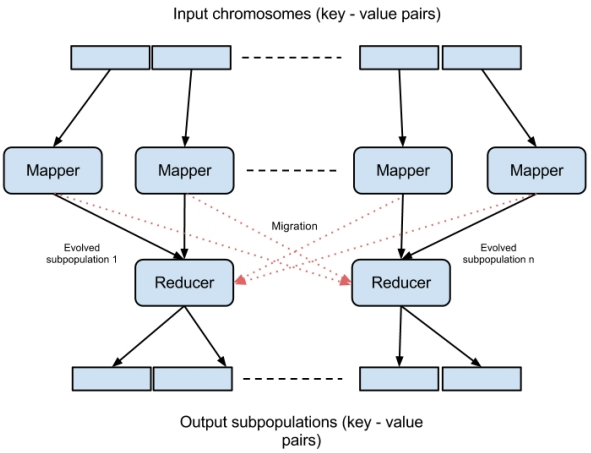

Fig. 3. **Distributed fitness evaluation - isolated populations**

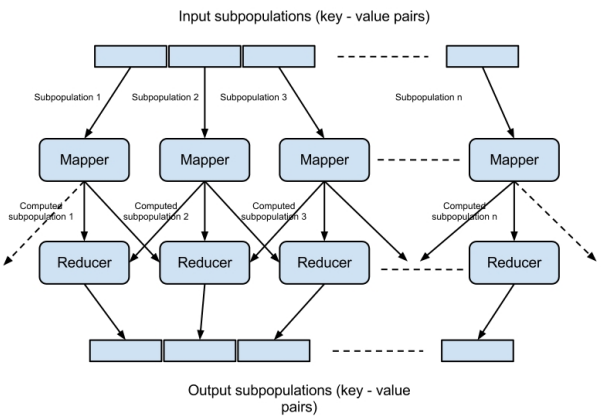

Fig. 4. **Distributed fitness evaluation - overlapping subpopulations**

mapper will handle the phase of spreading the value of the node in the neighborhood. The reducer will have the role of selecting among the individuals from the subpopulation represented by chromosomes that are locally stored in the node and those received from the neighborhood. After selection, crossover and mutation are performed and, in the end, only a number of individuals equal to the size of the node are kept for the next generation.

In the next generation, based on its id, each individual will be sent to the right reducers, so that the neighborhood relationships are kept. In the first iteration, the mapper is responsible for setting the subpopulation id of the newly generated chromosomes.

This technique of overlapping subpopulations can be efficient in some situations, as it follows the fine grained model of parallel genetic algorithms.

Parametrization. There are a few important decisions to take when implementing a genetic algorithm and when using distributed fitness evaluation. First of all, there are design decisions common

for a genetic algorithm: choosing a chromosome representation, implementing a fitness function, determining the genetic operators: selection, crossover, mutation. The genetic operators must ensure the convergence of the algorithm, while also allowing diversity inside the population for a continuum improvement of the solution from one generation to another and thus to ensure the evolution of the population. Then there are some parameters that can be setup to improve the algorithm: stop criteria, mutation probability, chromosome sampling, population size or the use of a local optimizer.

Population size is especially important when using the distributed fitness evaluation approach, as the individuals of the population will be spread across the mappers for fitness evaluation and then grouped in subpopulations on each reducer for selection, crossover and mutation. It is important that each subpopulation has enough individuals to ensure that the individuals selected for reproduction will be good enough to produce offspring that will improve the overall population. So, population size has to be carefully selected together with the number of mappers and the maximum number of reducers that will be created at each generation. The number of mappers and reducers are also dependent on the environment on which the algorithm is deployed.

Another important configuration option is the model that will be used in order to spread the individuals on the reducers. Naturally, the map reduce model creates subpopulations in the reduce step. However, the way subpopulations are used in the three models presented in the previous section are different and can impact the performance and convergence of the algorithm. In the results section, we try to compare the performances of these models on two different problems. However, choosing the best model depends on each problem and on the number of machines available in the environment.

Other decisions that can be taken in order to improve the algorithm for the distributed fitness evaluation approach are: to keep the best individuals on each mapper separately, to override the default partitioner in order to send the best individuals to one reducer or spread them to all reducers.

5. EXPERIMENTS

5.1 Test Problems

In this section we will present the experiments that we conducted in order to test our framework. We will present the problems we choose for testing, the configurations for those problems and the results that we obtained. We tested our framework on two problems: the traveling salesman problem and the job shop scheduling problem. Both of them are hard problems that are frequently used for benchmarking of genetic algorithm implementations.

The traveling salesman problem consists of finding the minimum path across N cities, starting from one city, visiting all cities exactly once and returning to the starting point. This is equivalent to finding a minimum Hamiltonian cycle in a complete graph with N nodes. For each instance of the problem, it is given the number of cities (N) and the distance between each two cities (or the coordinates of the cities).

For our tests, we used an instance of the problem with 38 cities, corresponding to the state of Djibouti, taken from [TSP.Gatech.Edu 2014].

The Job Shop Scheduling problem aims at scheduling N jobs, each with M tasks on M machines so that the makespan is minimized. The maskespan of a schedule is the time needed for all jobs to complete their execution, or more specific, the time that passes from the beginning of the first scheduled task until the completion of the last running task. Each task requires to be deployed on a specific machine. The tasks of a job must be executed in a specific order.

For testing we used a classical instance of the job shop scheduling problem: $FT10$. It consists of 10 jobs, each with 10 tasks that must be scheduled on 10 machines. The input data is taken from the *OR library* [Beasley 1990].

5.2 Conducted Experiments

We deployed our solution on the *Grid 5000's* distributed environment [Grid5000.Fr 2014], using *Hadoop version 1.0.1*. We used *OpenNebula Cloud toolkit* [OpenNebula.Org 2014] for deploying a Cloud infrastrucure on the *Grid 5000's* site.

We conducted several experiments in order to compare the three models for handling subpopulations: the global population model, the island model and the neighborhood model. We wanted to compare them in terms of result quality. In execution time we cannot see any significant variation, because the operations done by the mappers and reducers are almost the same, the only difference being the mechanism of handling subpopulation, which does not result in significant overhead.

The experiment consisted of varying population size while keeping the number of mappers and reducers constant, with all three models. We used 50 mappers and 10 reducers, and subpopulations varying from 1000 to 10000 individuals for the first two models. For the neighborhood model, we need to have the same number of mappers and reducers, so we used 50 mappers and 50 reducers, with an overlapping window of 2. This means that a mapper will send its results to two reducers. Overall, the neighborhood model will have more reducers with smaller subpopulations, with the same population size.

The results we obtained are depicted in Figure 5. We can see that, for all three models, the tendency is to obtain better results with bigger populations. Also, we can see that both the island model and the neighborhood model outperform the global population model. Moreover, the neighborhood model obtains with a population of 1000 better results than the other models obtain with a population of 7000, and the time therefore is considerably smaller. Besides that, we observed that the neighborhood model tends to converge faster than the other two models. The tests were made for 50 generations, and if the other two models were not converging after 50 generations, the neighborhood model was converging in 35 to 45 generations, which again was improving the time needed to run. On the other hand, a faster convergence presents the risk of converging to a local optima.

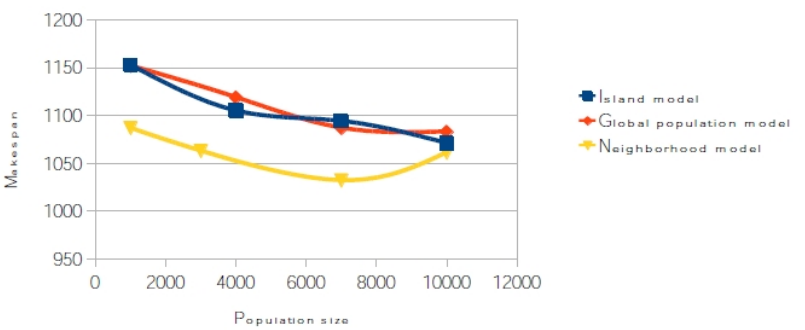

Fig. 5. **Island model vs. Global population model vs Neighborhood model for job shop scheduling problem**

In order to properly evaluate the difference between the neighborhood model and the other two models, we tested the neighborhood model and the global population model for a more complex problem. We chose the instance of TSP with 38 cities.

We used 10 mappers and 5 reducers for the global population model and 10 mappers and 10 reducers for the neighborhood model. We kept the overlapping window at 2. The results are depicted in Figure 6.

We can observe that in this case, too, the neighborhood model outperforms the global population model, and the difference is even bigger than for the previous test. Moreover, we can see that the global population model does not show much progress for larger populations, while the neighborhood model has a higher evolution rate.

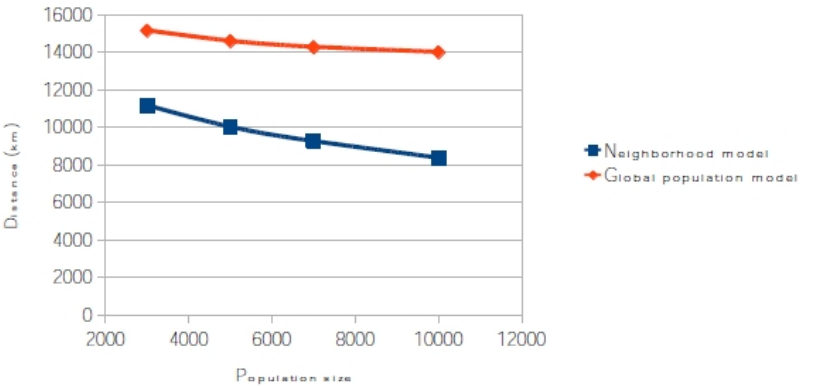

Fig. 6. **Global population model vs Neighborhood model for TSP**

6. CONCLUSIONS

Due to its powerful mechanism of handling large amounts of data and its massive parallelization technique, the MapReduce framework can be successfully used for running parallel algorithms. Genetic algorithms, due to their parallel nature and the need of using large populations to solve complex optimization problems, are one of the classes of algorithms that can benefit from this method of parallelization.

Research in the area of genetic algorithm parallelization using Hadoop is still in the beginning, and only a few works in adapting genetic algorithms to this model exist. The most convincing method used so far was to make each generation a MapReduce job. In this article, we used the same concept, but we enhanced it with different methods of handling subpopulations. We observed that is natural for the MapReduce model to form subpopulations, and thus the handling of subpopulations can be done with insignificant additional overhead. We used this characteristic to develop two alternative models: the island model and the neighborhood model. Both models rely on existing parallelization techniques for genetic algorithms, our contribution being to adapt these techniques to the MapReduce model and to add some modifications in order to improve the results.

Overall, we implemented three models for distributed fitness evaluation, with three methods of handling subpopulations: with a global population, isolated subpopulations and overlapping subpopulations. We tested all three implementations on two different problems: the job shop scheduling problem and the traveling salesman problem. There was no significant difference in execution time between the three models, but the quality of the solution was definitely higher for the neighborhood model over the other two models. The island model also outperformed the global population model. Moreover, as the complexity of the problem grows, the difference between the neighborhood model and the other models grows as well. This proved the fact that correctly handling the subpopulations formed by MapReduce can significantly improve the obtained results.

In our tests, we also measured the increase in execution time while increasing the size of the population. Also, we considered the variation in execution time when the number of mappers is increasing. We found that the number of mappers influences the execution time per iteration. The optimum number of mappers must be properly determined through tests. This is an important matter, as a too small number of mappers might get overloaded and a too large number of mappers might lead to additional overhead.

We implemented these models as part of a framework designed specifically for running genetic algorithms in Hadoop. The implementations are generic, so that each new problem could be easily implemented, with their specific parameters and genetic operators. We developed this framework having in mind the fact that, with a starting point and an easier way to develop genetic algorithms in Hadoop, research will grow and new improved models will be implemented.

As future work, we believe it is worth researching ways to reduce the overhead between mappers and reducers, by compressing the representations of individuals and thus reducing the I/O overhead.

7. ACKNOWLEDGMENTS

The research presented in this paper is supported by the following projects: "*SideSTEP - Scheduling Methods for Dynamic Distributed Systems: a self-* approach*", (PN-II-CT-RO-FR-2012-1-0084); "; *CyberWater* grant of the Romanian National Authority for Scientific Research, CNDI-UEFISCDI, project number 47/2012.

REFERENCES

J. Beasley. 1990. OR-Library: Distributing Test Problems by Electronic Mail. *Journal of the Operational Research Society* 41, 11 (Nov. 1990). DOI:http://dx.doi.org/41(11):1069-1072

R. Buyya C. Jin, C. Vecchiola. MRPGA: An Extension of MapReduce for Parallelizing Genetic Algorithms. (????).

Erick Cantu-Paz. 1998. A Survey of Parallel Genetic Algorithms. 10, 2 (1998), 141–171.

Jummy Lin D. Huang. 2010. Scaling Populations of a Genetic Algorithm for Job Shop Scheduling Problems using Mapreduce. *Cloud Computing Technology and Science* (2010).

Grid5000.Fr. 2014. Grid5000 Home Page. (2014). Retrieved May, 2014 from http://www.grid5000.fr

OpenNebula.Org. 2014. Open Nebula Home Page. (2014). Retrieved May, 2014 from http://opennebula.org

TSP.Gatech.Edu. 2014. The Traveling Salesman Problem Official Site. (2014). Retrieved May, 2014 from http://www.tsp.gatech.edu/world/countries.html

Abhishek Verma. 2010. *Scaling simple, compact and extended compact genetic algorithms using MapReduce*. Master's thesis. University of Illinois at Urbana-Champaign.

Carsten Witt. 2008. Population size versus runtime of a simple evolutionary algorithm. *Theoretical Computer Science* 403, 1 (Aug. 2008), 104–120.

Roy H. Campbell Xavier Llora, Abhishek Verma and David E. Goldberg. 2010. When Huge is Routine: Scaling Genetic Algorithms and Estimation of Distribution Algorithms via Data - Intensive Computing. *Parallel and Distributed Computational Intelligence SCI 269* (2010).