Comparison of Different Implementations of Parallelization of Genetic Algorithms

Marat Zhaksilikov
Department of Computer Science
University of Nevada
Reno, NV 89557
zhaks@cs.unr.edu

Frederick C. Harris, Jr.
Department of Computer Science
University of Nevada
Reno, NV 89557
fredh@cs.unr.edu

Abstract

Developed by John Holland at the University of Michigan *genetic algorithms* are a heuristic search technique that can help to provide satisfactory results for hard optimization problems. But even using genetic algorithms the computations can require a pretty large amount of time. Since genetic algorithms are an abstraction of natural selection they are very easy to parallelize. There are different ways to parallelize genetic algorithms and each of them raise different issues. In this paper we try to compare the performance of a few different parallel implementations of genetic algorithms.

keywords: Genetic Algorithms, Parallel Processing

1 Introduction.

Genetic algorithms (GAs) are an effective search method that simulates the process of natural selection. The purpose of GAs is to provide satisfactory results for optimization problems that are hard to solve using exhaustive techniques. The simple GA consists of following steps. First, it is necessary to encode the parameter set of the problem into a string representation, called *a chromosome*. Chromosomes are usually binary strings. Then an initial population of chromosomes is randomly generated. The next step is *selection*, which is the process of choosing parent chromosomes for recombination. Usually this is done in some way to provide the opportunity for "better" chromosomes to be parents. What chromosome is better or worse is determined by a *fitness* function that is an objective function for the given problem. A *crossover* operator allows for the exchange of substrings between parent chromosomes in order to produce the children. Next *mutation* takes place. This is a random change of a bit position in a child chromosome. The last step is the *evaluation* of the child chromosome to determine the fitness of this individual. This sequence of events defines one iteration of genetic algorithm. After a certain number of iterations the algorithm can be stopped and the member of final population with the best fitness value will be the solution returned by the algorithm.

All of the presented operators can be implemented in different ways. For example selection can be completely random. In this work one of the popular methods of selection, called biased roulette wheel selection, was used. The roulette wheel is divided into a number of sectors equal to the size of population. The sector size is proportional to the fitness of the corresponding individual so that better chromosomes have a better chance of being chosen. In order to implement two parent recombination, one point crossover was used. In this case the random number generator produces a number k corresponding to one bit position, and then parents exchange substrings $[0; k-1]$. To provide the opportunity for more complete exploration of the search space the mutation operator was used. This takes place during recombination step and, like in nature, the probability of mutation is low.

In Section 2 the different approaches for GA parallelization and related issues are described. Section 3 contains the description of implemented parallel GAs and describes the results of performance comparison. Conclusions and future work are presented in Section 4.

2 Parallel GAs.

Because the natural selection is a parallel process, GAs as a model of natural selection are very suitable

for parallelization. [4] gives a classification of different approaches to GA parallelization.

2.1 Global Parallelization.

The first way to parallelize a GA is with *global parallelization*. In this model the genetic operators are applied to the single population in parallel. There are different approaches to implement this model. The first way is to parallelize just the evaluation process. In the case of a shared memory multiprocessor machine, each processor reads the assigned individuals, evaluates them, and stores results in the shared memory. In this method it is necessary to provide a synchronization barrier between generations. In a distributed memory environment, to simplify the implementation of algorithm, a master processor is used to store population and to apply the genetic operators. The master is also responsible for sending a subset of individuals to the slave processors for evaluation and collecting the results. We can see that performance of this approach strongly depends on communication overhead and performance of the parallel architecture used. Almost linear speedup can be achieved when computation time is much longer than communication time.

The second way to implement the global parallelization is to apply genetic operators in parallel. Because the genetic operators are very simple and the time spent in communication can be much longer then the time doing computations. It is possible for this method to take longer than the sequential.

2.2 Coarse Grained Parallelization.

The next way to parallelize GAs is a *coarse grained parallelization*. In this case the population is divided into a few subpopulations. Each subpopulation is assigned to a different processor and kept relatively isolated. To provide the way for information exchange the *migration* operator is used. There are two ways of migration implementation: island model and stepping stone model. The first model allows migration from one subpopulation to any other subpopulation, the second model restricts migration only to neighboring subpopulations.

The analysis of the literature shows that performance of a coarse grained parallel GA is controlled by migration related issues. [4] underlines following migration parameters: *topology* (defines the connection between subpopulations), *migration rate* (number of individuals to migrate), *migration interval* (how often migration takes place). Usually these parameters are tuned to get better performance. For example, [7] reports that if migration rate is too high, the population converges too fast and rarely produces a satisfactory result. But the absence of migration also leads to poor results. The topology issue is usually related to the connectivity of a network. Different papers report good results using both short and long diameter topologies [2, 3, 5]. In case of higher connectivity good solutions can spread faster across the network, but sparsely connected topology results in more isolated solutions that can be recombined later to form potentially better results. The number of migrants should be enough to provide the information exchange, but the large migration rate can probably be a waste of communication resources.

2.3 Fine Grained Parallelization.

In case of fine grained parallelization, the population is divided into large number of small subpopulations. The migration is provided by overlapping the subpopulations. The literature shows that the performance of fine grained parallel GAs depend on subpopulation size and interconnection topology. [9] and [10] show that performance usually degrades with an increase in the subpopulation size; while [1] and [8] show that a topology with a medium diameter usually provides better results.

2.4 Hybrid approaches.

Hybrid approaches combine features of different parallelization methods. For example the population may be divided as in case of coarse grained parallelization, and the migration operator may be used to exchange individuals, but evaluation of individuals may be handled in parallel.

3 Implementation and Performance Analysis.

Communication overhead is the biggest problem that restricts the possible speedup using multiple processors. Therefore, the goal is to construct algorithms where the increase of communication overhead with the increase in the number of processors is negligible when compared to the gain of computation speedup. This is all done while keeping the output of the parallel algorithm the same as for sequential version.

To compare the communication overhead, it is necessary to decrease the computation time as much as possible. For this purpose very simple selection,

crossover, mutation and evaluation functions were used. The genetic operators used were described in the previous section. The chosen evaluation function is x^2. Besides the decrease of computation time, it is a very easy to prognosticate the desirable result using this function.

These algorithms were implemented using Parallel Virtual Machine software [6] on a network of DEC stations. This is a distributed memory architecture and, as it was pointed out above, the easiest way to parallelize GA using this kind of architecture is to use a master/slave topology.

3.1 Global Parallelization.

In this sub-section we would like to address the question about different ways to implement the global parallelization of GAs. The question raised earlier was: Do you get better performance by applying genetic operators in parallel? To answer this question, it is necessary to compare the communication overhead of different implementations: 1) just evaluation in parallel (Algorithm 1); 2) application of genetic operators in parallel (Algorithm 2).

In the Algorithm 1 selection, crossover, and mutation are performed sequentially on the master processor, while slave processors evaluate individuals in parallel. The sequential algorithm performs evaluation for each new child during each iteration. When evaluation is done in parallel the master processor sends a portion of the children to each slave processor. The master processor the collects the results, the fitness of the individuals, and computes the global statistics.

Next Algorithm 2 was implemented. During the sequential algorithm, the generation step consists of $PopulationSize/2$ iterations. Each time two individuals are picked up and mated to produce two children, then fitness levels of the children are calculated. In the case of Algorithm 2 the master processor broadcasts the entire population to all slaves and each slave performs the described above sequence of steps $PopulationSize/2 * [Number\_Of\_Processors]$ times. Then the master processor collects the new population and calculates the global statistics. The necessary points are: 1) We can eliminate the broadcast of the first population by performing the population initialization on each slave processor using the same random seed number; 2) It is necessary to provide each slave processor with different random seed number to ensure that slaves will pick different individuals for mating[1].

[1] This can be eliminated by sending to each slave processor different part of the global population.

It can be noticed that communication overhead increases using Algorithm 2 because of the necessity to send the entire population back and forth. In the case of using the Algorithm 1, the slave processors return just the fitnesses of the individuals. So, the question is if the gain of doing selection, crossover, and mutation in parallel outperforms the increasing communication overhead.

In order to answer this question, it is necessary to compare the performance of the sequential GA and the different implementations of parallel GAs. The sequential algorithm found the optimal solution each time using a population size of 240. As can be expected, the performance of Algorithm 1 is the same because the basic genetic operators are applied sequentially. But Algorithm 2 showed worse performance, as shown on Table 1. Theoretically both implementations should have the same performance, because the Schema Theorem tells that the probability that the number of particular schemata increases or decreases in the next generation depends only on characteristics of this schemata and not on the sequence of picking individuals. But in practice, the selection of individuals strongly depends on the random number generator used and the implementation of the selection operator. It was observed that even using different random seed numbers the slaves often produced the same individuals. As a result, the search space was explored insufficiently and population sometimes converged to suboptimal results. It seems that performance of Algorithm 2 can be increased by sending different subpopulations to each slave.

No of Processors	Algorithm 1	Algorithm 2
8	1046529	1045916
20	1046529	1045098

Table 1: Output of Algorithms 1 and 2. The correct answer is 1046529.

The next step is to compare the dependence of execution time from the number of processors for different population sizes[2]. Because the evaluation step is so simple, the increase of evaluation time is mostly defined by communication overhead. So the speed of execution time increase with increasing of processor number is defined by increasing of the communication time. It was noticed that communication factor for Algorithm 2 is much larger than that for Algorithm 1.

[2] The population size defines the amount of data to be sent. Also the different chromosome length can be used for this purpose

When the number of processors is increasing the calculation part of the execution time should decrease. Therefore, if the difference in execution time decreases with increasing of processor number, then the communication overhead stays the same or increases slowly for increased amount of data to be evaluated. This feature is very important when we have to exchange a large amount of data between processors. Algorithm 1 showed a good performance with respect to the property described above, but in case of Algorithm 2 the communication overhead increases very fast. Also it is interesting to observe the dependence of execution time on the size of population. Even in the case of a very simple evaluation function, the difference in execution time between the sequential algorithm and Algorithm 1 remains the same with increasing population size. This means that an increase in communication overhead is compensated for by a decrease in the computational overhead. But in case of Algorithm 2 the communication overhead increases very fast and is not compensated for by lower computational time.

The situation just described with a short evaluation time is very unusual. In most cases the evaluation step requires a lot of execution time. In this case, the parallelization of the evaluation step can be very helpful. When we used Algorithm 1 for solving a robotics problem, speedup proved to be almost linear up to 20 processors.

3.2 Coarse Grained Parallelization.

When compared to the Global Parallelization Coarse Grained Parallelization seems to be more attractive. The division of the population into a number of subpopulations assigned to different processors results in decreasing the computation time, while the communication between processors is minimized.

Three different approaches to Coarse Grained Parallelization were implemented. The first approach (Algorithm 3) looks close enough to Algorithm 2, but in this case after a certain number of iterations the master processor collects only a few of the best individuals from each slave processor, sorts them and then broadcasts the bests from the bests to all of the slave processors. These individuals replace the worst individuals of each slave processor [7]. The second approach (Algorithm 4) allows migration only between neighboring slave processors that are connected in a chain. The advantage when compared to Algorithm 3 is that the communication time does not increase with the increasing of the number of processors[3]. The last approach (Algorithm 5) is a variant of the Algorithm 4, but in this case the individuals are assigned to each slave processor not randomly but according to the corresponding point in the search space. The search space is divided into a number of subspaces corresponding to the number of the slave processors, and each slave processor explores it's particular subspace. This makes it possible to decrease of the chromosome length. Migration is provided by overlapping of subspaces.

No of Processors	Algorithm 3			Algorithm 4			Algorithm 5		
	Migration interval			Migration interval			Migration interval		
	25	10	5	25	10	5	25	10	5
2	1046301.8	1046529	1046301.8	1046301.8	1046301.8	1046301.8	1046529	1046529	1046529
8	1046301.8	1046529	1046301.8	1046301.8	1046301.8	1046301.8	1042905.9	1042905.9	1042905.9
12	1046074.6	1046529	1046074.6	1046074.6	1046529	1046074.6	1046529	1046529	1046529
20	1044714.6	1046529	1044714.6	1044714.6	1044714.6	1044714.6	1046529	1046529	1046529

Table 2: Output of Algorithms 3, 4, and 5. The correct answer is 1046529.

In order to evaluate the performance of different implementations of Coarse Grained Parallelization a number of experements were carried out. Table 2 shows the dependency of the solution returned by algorithms from the number of processors for different intervals of migration. As can be seen, Algorithm 3 has the strongest dependency on the interval of migration. The frequency of migration above and below a certain level results in performance degradation. Moreover, the performance get worse with increasing number of processors probably because we are decreasing the number of individuals in a subpopulation. However, the right chosen interval of migration leads to the maximal performance for all considered number of processors. Algorithm 4 performs exactly the same way for migration intervals below and above a certain level, but in this case even when the best migration interval is chosen Algorithm 4 depends on number of processors. Our opinion is that the brodcasting of the best of the best individuals provides more information about the right direction of the search than the exchange of locally best individuals. Considering Algorithm 5, a strong performance degradation can be noticed when the number of processors equals 8. This degradation can be explained by the fact that the length of chromosome used in this experiment equals 10. In this case the search space is divided evenly between all slave processors without overlapping, eliminating the migration step. For all other considered numbers of processors and intervals of migration the algorithm finds the optimal solution. We think that reduced chromosome

[3] The possible increase of communication time can be result of the synchronization problem, but this is the common problem for both algorithms.

length makes easy the exploration of the search space.

While experimenting with the dependency of the execution time on the number of processors for different intervals of migration it was observed that the implemented algorithms showed a shorter execution time for all considered numbers of processors and intervals of migration even for the very simple evaluation function. This is the biggest advantage when comparing to Global Parallelization. Moreover in case of the Algorithms 4 and 5 the increasing problem of synchronization is compensated for by the decreasing of the evaluation time with increasing of the number of processors. This is also true for Algorithm 3 until the number of processors reaches a certain value. After this threshold, the necessity to collect data from the increased number of processors makes the execution time longer.

4 Conclusions and Future Work.

The comparison of different implementations of global parallelization of genetic algorithm shows that in case of usage very simple crossover, selection, and mutation operators, the application of genetic operators in parallel decreases the performance. The communication overhead increases much faster as the number of processors increases or the amount of data to be processed increases. But in case of more complicated genetic operators it may be helpful.

As was proposed, the use of Coarse Grained Parallelization looks more attractive for the purpose of decreasing the execution time. It is necessary to experiment with parameters of the algorithm to increase the performance. The right values of these parameters result in the same performance as for the sequential algorithm. Algorithm 5 can be very useful for a large number of processors, but in case of relatively small number of processors it just makes it more complicated to implement the particular problem.

In the future, it will be interesting to compare the performance of the various algorithms discussed for different types of problems. We also will continue the experiments with different parameters of Coarse Grained Parallelization.

References

[1] J. Anderson, E. and M.C. Ferris. A genetic algorithm for assembly line balancing problem. Technical Report Technical Report TR 926, Computer Science Department, University of Wisconsin-Madison, 1990.

[2] R. Bianchini and C. M. Brown. Parallel genetic algorithms on distributed-memory architectures. Technical Report 436, Computer Science Department, University of Rochester, 1992.

[3] R. Bianchini and C. M. Brown. Parallel genetic algorithms on distributed-memory architectures. In S. Atkins and A. S. Wagner, editors, *Transputer Research and Applications 6*, pages 67–82. Amsterdam: IOS Press, 1993.

[4] E. Cantu-Paz. A summary of research on parallel genetic genetic algorithms. Technical Report IlliGAL Report No. 95007, The Illinois Genetic Algorithm Laboratory, University of Illinois at Urbana-Champaign, 1995.

[5] E. Cantu-Paz and M. Majia-Olvera. Experemental results in distributed genetic algorithms. In *International Symposium on Applied Corporate Computing*, pages 99–108. Monterey, Mexico, 1994.

[6] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine*. MIT Press, Cambridge, MA, 1994.

[7] J. Hines, J.T. Thorpe, K.B. Winiecki, Jr., and F.C. Harris, Jr. Solving quadratic assignement problems with parallel genetic algorithms. In S. Louis, editor, *Proc. of the ISCA Int. Conf. on Intelligent Systems*, pages 11–16, San Francisco, CA, June 1995. ISCA.

[8] M. Schewehm. Implementation of genetic algorithms on various interconnection networks. *Parallel Computing and Transputer Applications*, pages 195–203, 1993.

[9] P. Spiessen and B. Manderick. A genetic algorithm for massively parallel computers. *Parallel Processing in Neural Systems and Computers, Dusseldorf, Germany*, pages 31–36, 1990.

[10] P. Spiessen and B. Manderick. A massively parallel genetic algorithm: Implementation and first analysis. In Richard K. Belew and Lashon B. Booker, editors, *Proceedings of the Fourth International Conference on Genetic Algorithms*. Morgan Kauffman, San Mateo, CA, 1991.