A FEW IMPLEMENTATIONS OF PARALLEL GENETIC ALGORITHM

Marin Golub, Domagoj Jakoboviæ
Faculty of Electrical Engineering and Computing, University of Zagreb
Department of Electronics, Microelectronics, Computer and Intelligent Systems
Unska 3, 10000 Zagreb, Croatia
e-mail: marin.golub@fer.hr, domagoj.jakobovic@fer.hr

Abstract: *In this paper we describe the parallel genetic algorithm implementation using multithreading. The genetic algorithm was extended to deal with several genetic operators over binary vectors and floating-point numbers. Particularly, the possibilities of parallelisation of generational, steady-state and tournament selection are examined. The tournament selection appears to be the most suitable for parallel implementation.*

Key words: genetic algorithm, multithreading, tournament selection, adaptive mutation

1. Introduction

Genetic algorithm is a heuristic random search method based on natural evolution which requires considerable amount of CPU time. Since the optimisation problem has to be solved in given computing and time constraints, parallel genetic algorithm is an attempt to speed up the program without interfering with other properties of the algorithm.

The parallel genetic algorithm (PGA) can be implemented using several threads. The main benefits that arise from multithreading are: better program structure (any program in which many activities do not depend upon each other can be redesigned so that each activity is executed as a thread) and efficient use of multiple processors (numerical algorithms and applications with a high degree of parallelism, such as matrix multiplication or, in this case, genetic algorithm, can run much faster when implemented with threads on a multiprocessor) (SunSoft, 1994).

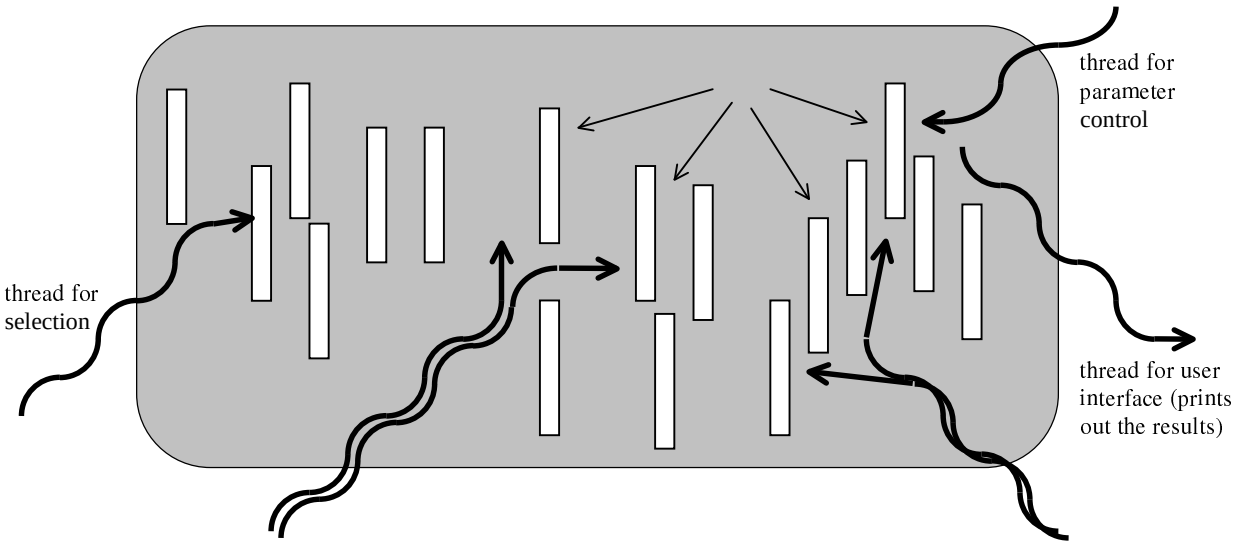

Fig. 1 Parallel genetic algorithm realised with multiple threads

For every algorithm that we want to execute in multiple threads, first we have to identify independent parts and assign to each a thread. One or more threads can be assigned to each

genetic operator (selection, crossover and mutation - Fig. 1). Additionally, we can assign a thread for user interface, a thread for parameter control, a thread for results comparison with other methods (e.g. we can implement a completely random search mechanism and compare its results with the genetic algorithm), etc.

2. The choice of selection

The steady-state selection has one parameter *M* - the number of new chromosomes to create. Generational replacement is the spatial case of steady-state selection in which parameter *M* equals the size of the population (Davis, 1991). Similarly, the tournament selection is the spatial case of steady-state selection too, in which parameter *M* equals 1 (Table 1).

Table 1 Steady-state reproduction and parameter *M*

Parameter *M*	Type of selection	Description
M=POP_SIZE	Generational selection	replaces whole population
1≤M≤POP_SIZE	Steady-state selection	replaces *M* individuals
M=1	Tournament selection	replaces only one individual (the worst of three chosen) with child of two survived parents

The steady state reproduction replaces *M* individuals in each iteration of evolution process. Let us divide that algorithm into three independent parts and assign to each one a thread. First thread executes only the crossover and creates new individuals. The second one performs the selection and deletes selected individuals. The third one does only the mutation. In that case, without any synchronisation, the population size will not be constant. If the thread for deletion is faster then the thread for creation, after some time, the population will consist of one individual (the last individual can't be deleted because it is the best one at the same time). Crossover operator needs two individuals to create a child and it waits for the other individual creation forever, because nobody will create it. That is the deadlock. The other possibility is memory overflow if the creation thread is faster than elimination.

That problem can be solved by simple synchronisation mechanism: if the population size is to small, the elimination thread waits; if the population size is to big the creation thread waits. The change of variable *POP_SIZE* must be assigned to a critical section to prevent multiple threads from simultaneously changing it. Few experiments showed that the parallel genetic algorithm described above spends more computation time for synchronisation than for optimisation, and the parallel program is even slower than the sequential one.

For steady-state selection with parameter *M=1* the roulette-wheel bad individual selection is not a good choice. As for the each turn the selection probabilities for whole population have to be calculated, roulette-wheel selection slows down the algorithm. In that case solution is tournament bad individual selection. The tournament bad individual selection in each step of the evolution chooses with equal probability three individuals from the mating pool. Then, it eliminates the weakest one of those three individuals. The survived two individuals are parents of a child which will replace the eliminated one.

3. Genetic operators as independent parts of genetic algorithm

The parallel steady-state genetic algorithm with tournament bad individual selection was implemented. In this implementation, genetic algorithm consists of two threads: one performs tournament selection and crossover and the other mutation (Fig. 2).

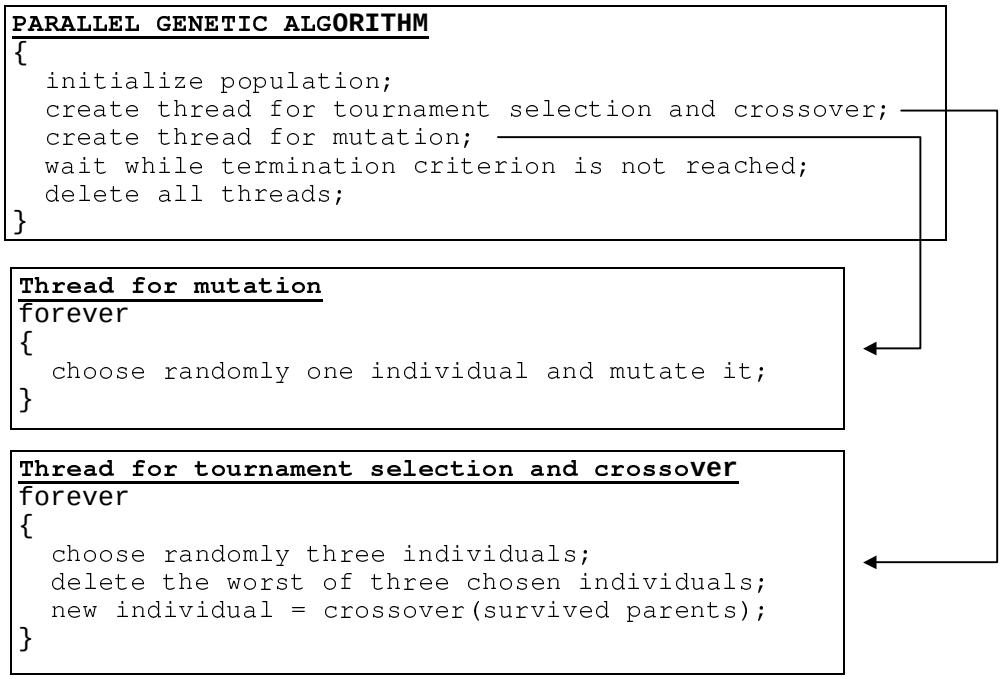

Fig. 2 The structure of simple parallel genetic algorithm (SPGA)

The major problem of that simple parallel implementation is that it has no control over mutation probability. The consequence is a very bad algorithm behavior. The results are slightly better than random search, but also useless.

If the threads are left to parallel execution without any control, one of two threads can waste some time on waiting for processor time. Then, one of two possibilities can happen:

A) The mutation thread works and the thread for selection and crossover waits.

This is completely random search, i.e. the mutation probability is set to one. If the elitism is not applied, the best individual achieved in past iterations will be lost. So, in the mutation thread the elitism must be added (Fig. 3).

```
Thread for mutation
forever
{
  choose randomly one individual;
  if(this individual is not the best) mutate it;  //elitism
}
```

Fig. 3 Thread for mutation extended with elitism

B) The thread for selection and crossover works and the mutation thread waits.

The mutation probability is set to zero. In several hundred iterations genetic algorithm produces a uniform population (the population consists of one individual with *POP_SIZE-1* copies). Even if we control the mutation probability, during the run of genetic algorithm about half the chromosomes created are duplicates (Davis, 1991). If the population is more similar, then the mutation probability must increase. The control of mutation probability can be easily solved with some synchronisation techniques such as MUTual EXclusion locks (MUTEX), condition variables or semaphore synchronisation, but, any of these synchronisation mechanisms spends too much of CPU time. As the goal of parallelisation is speeding up the algorithm, the synchronisation must be avoided if possible.

The other possibility is implementation of adaptive mutation probability. Before the crossover is performed, the parents have to be checked. If the parents are equal, then mutate one of them and produce their child completely randomly (Fig. 4).

```
Thread for tournament selection and crossover
forever
{
  choose randomly three individuals;              // tournament bad
  delete the worst of three chosen individuals;   // individual selection
  if(survived individuals are equal)
  {
    mutate one of the equal individuals; // adaptive mutation probability
    create new individual randomly;      //     and duplicate elimination
  }
  else
  {
    new individual = crossover(survived parents);
  }
}
```

Fig. 4 Thread for tournament selection and crossover extended with duplicate elimination and adaptive mutation probability

These two extended threads can be parallely executed without any synchronisation. Experiments have shown that 69,4% of the optimisation time consumes the thread for tournament selection and crossover and the rest of optimisation time (30,6%) spends the mutation thread. On the two processor system, the whole optimisation time is about 30% shorter than on one processor system.

Described extended parallel genetic algorithm (EPGA_1) divided into only two threads is suitable for execution on the two processor system.

4. Parallel genetic algorithm with equal threads

If we want to make a good use of multiprocessor system with more than two processors, the genetic algorithm has to be divided into more than two threads. The idea is in dividing genetic algorithm into required number of equal and independent parts (Fig. 5).

This is the same algorithm like described EPGA_1, but it is divided in a different manner.

```
PARALLEL GENETIC ALGORITHM
{
  initialize population;
  create several equal evolution threads;
  wait while termination criterion is not reached;
  delete all threads;
}
```

```
Evolution thread
forever
{
  perform tournament selection and delete one individual;
  perform crossover and replace deleted individual;
  perform mutation;
}
```

Fig. 5 The structure of parallel genetic algorithm with equal threads (EPGA_2)

Each thread performs all genetic operators over the whole population like nonparallel genetic algorithm. In each iteration, one thread operates on only a part of the population, because the tournament selection works over only three chromosomes. The other thread can work over the same chromosomes (one, two or all three) at the same time without any synchronization. This kind of parallel algorithm works the same with one or more threads.

5. Experimental results

The parallel genetic algorithm was tested on several multidimensional problems. Table 2 shows the results of the optimisation of 38 dimensional approximation problem (Schoeneburg, 1995). The global minimum of that problem is equal or greater than 0 (the smaller solution value is better solution). 100.000 iterations were made for each experiment and for each algorithm 20 experiments were done. The size of population is set to 50.

Table 2 Random search and parallel genetic algorithm comparison

	Random search	SPGA	EPGA_1	EPGA_2
Total CPU time in seconds	135	302	350	350
Optimization time for $N_P \geq 2$ (N_P - number of processors)	$\frac{135}{N_P}$	212	245	$\approx \frac{350}{N_P}$
the worst solution	25 019	19 455	109.0	
average solution	**21 798**	**16 881**	**49.1**	
the best solution	14 826	8 250	16.5	

The optimisation time is equal to duration of *longest* thread, if the number of processors are equal or greater than number of threads. The optimisation time for SPGA and EPGA_1 on a two processor system is equal to duration of the thread for crossover and selection (that is about 70% of total CPU time). On a three or more processor system the program isn't faster because the algorithm is divided into only two threads. The EPGA_2 is divided into $N_T=N_P$ threads. The optimisation time for EPGA_2 is equal to optimisation time for the same non-parallel genetic algorithm divided by number of processors.

6. Concluding remarks

By dividing the genetic algorithm into threads we achieved several benefits:

- the algorithm is faster (the optimisation time is shorter than for the nonparallel genetic algorithm),
- the code can be easily adapted and extended with new genetic operators and
- we can simultaneously execute two or more methods and compare the results at the same time.

It is desirable that the parts obtained by dividing the genetic algorithm are independent, i.e. the critical sections must be avoided, because the synchronisation mechanisms slow down the parallel program.

The simple parallel genetic algorithm (genetic algorithm without elitism, duplicate elimination and adaptive mutation probability) is like a random search: the results of both algorithms are useless (Table 2).

Steady-state genetic algorithm with tournament bad individual selection, elitism, duplicate elimination, adaptive mutation probability and uniform crossover achieved the best results. It is divided into threads in two ways:

1. genetic operators are implemented as threads (EPGA_1) and
2. genetic algorithm is divided into equal threads - each thread performs all genetic operators (EPGA_2).

Table 3 shows positive and negative properties of parallel genetic algorithm with extended threads.

Table 3 Advantages and disadvantages of EPGA

	EPGA_1	**EPGA_2**
Advantages	•code can be easily adapted and extended with new genetic operators	•there is no harm if one thread waits for CPU time because each thread has all elements of genetic algorithm •optimisation time is N_P times shorter than for sequential genetic algorithm
Disadvantages	•fixed number of threads •if the number of processors is greater than the number of threads, the optimisation time is not shorter	•if we cannot avoid critical sections when changing or adding new genetic operators then some synchronisation mechanisms must be implemented (it slows down the algorithm)

Acknowledgement

This work was carried out within the project 036-014 *Problem-Solving Environments in Engineering*, funded by Ministry of Science and Technology of the Republic of Croatia.

References:

1. Davis, L. (1991), *Handbook of Genetic Algorithms*, Van Nostrand Reinhold, New York.
2. Goldberg, D.E. (1989*), Genetic Algorithms in Search, Optimization and Machine Learning*, Addison-Wesley
3. Golub M., "Evaluating The Use of Genetic Algorithms for Approximation Time Series", M.S. Thesis, Zagreb, 1996. (in Croatian)
4. Jelenkoviæ, L, Omrèen-Èeko, Goran (1997), "Experiments with Multithreading in Parallel Computing", Proceedings of the 19th International Conference ITI'97, Pula, pp. 451-456.
5. Michalewicz, Z. (1994), *Genetic Algorithms + Data Structures = Evolution Programs*, Springer-Verlag, Berlin
6. Schoeneburg, E., Heinzmann, F., Feddersen. S., (1995) *Genetische Algorithmen und Evolutionsstrategien,* Addison-Wesley
7. Srinivas, M., Patnaik, L.M. (1994), "Genetic Algorithms: A Survey", Computer, Vol. 27-6, pp.17-26, June 1994.
8. SunSoft, (1994), *Solaris 2.4: Multithreaded Programming Guide,* Sun Microsystems, Mountain View, California