# A THREAD BUILDING BLOCKS BASED PARALLEL GENETIC ALGORITHM

**Erkan Bostanci *, Yilmaz Ar & Sevgi Yigit-Sert**
SAAT Laboratory, Computer Engineering Department, Ankara University, Golbasi Campus, Ankara, Turkey
{ebostanci, ar, syigit}@ankara.edu.tr

**ABSTRACT**
Genetic Algorithms are biologically-inspired computational methods commonly used for many different optimization problems in various domains. They are also known to require significant computational time to produce optimal solutions. This paper presents a genetic algorithm library that encompasses the main genetic operations such as selection, recombination and mutation. The library can be run in serial or parallel form using Intel's Thread Building Blocks which is a cross-platform multi-threading library. An evaluation with different benchmark functions has shown that the parallel implementation can achieve up to 3:1 speed-up over the serial implementation on a dual core computer with ordinary specifications.

**Keywords:** G*enetic Algorithm, Thread Building Blocks, Parallel Computation, Performance*

## 1. INTRODUCTION

Genetic Algorithms (GA) have been used in many areas to reach optimal solutions for various kinds of optimization problems [1, 2]. In many domains, the number of genes in chromosomes that represent the individuals (i.e. candidate solutions) of a population would become very large. The processing times required for these algorithms to find optimal solutions may not scale well, when the chromosome sizes and/or the number of individuals increase.

The performance also depends on the complexity of the fitness function. In order to overcome this problem as well as the one mentioned above, parallelism need to be utilized to compute fitness values concurrently. This embedded parallel calculation will reduce the computational time in GAs. This will allow GAs to be employed in a wider range of areas that need large numbers of genes in their individuals or have complex fitness functions. There are library solutions [3, 4] that employ Graphics Processing Unit's (GPU) threading features (e.g. CUDA), however, using them requires having a modern graphics card which can be still considered as expensive in the market.

Regardless of whether serial or parallel computation is used, the GA community needs a framework that the researchers can easily download and use in their applications. The software given in this study provides an easy to use, compact, yet complete GA framework with both serial and parallel computation options of fitness values.

## 2. SOFTWARE DESCRIPTION

The GA framework presented here is written in C++ language. The class features of this language allowed following a modular, user-friendly and reusable design. Parallelism was implemented using Intel's Thread Building Blocks (TBB) [5] which has superior features over its alternatives such as OpenMP or native threads. These features include concurrent data structures or independence of compiler support. The former advantage was utilized in the design that computed fitness values concurrently for each sample over a population which is a list structure. The latter advantage is that the TBB library can be used along with any compiler for different platforms.

The following sections will elaborate the framework design and how the parallelism was incorporated into this design.

### 2.1. Classes

The developed framework has the following class structure:

#### 2.1.1. GeneModel

This class has a single data member for storing a single piece of information, a gene, and a function to perform mutation on this gene. The example code uses a Boolean data as the gene, however the `GeneModel` class allows including user-defined data types for different problems.

**2.1.2. Sample**
The Sample class represents the chromosome of an individual which is a candidate solution the optimisation problem. The class has data members to store gene information stored as collection of `GeneModel` objects, sample identifier, the fitness value for this sample along with normalized and accumulated values for the fitness value. The latter two values are used for the selection process in the algorithm.

This class includes a calling method for the `GeneModel` class' mutate method. Note that random selection of the genes for mutation operation is performed here before the actual mutation is carried out on the `GeneModel`.

**2.1.3 Population**
This class serves as the container for all the individuals in the population. The class employs Intel's TBB for computing the fitness values for all individuals concurrently using a `blocked range<int>` construct where the integer value designates the index in the collection storing the Sample objects.

Service methods for adding new samples to the population, sorting based on fitness values and cleaning up the population (i.e. removing the samples with lows fitness values from the population) also reside in this class.

**2.1.4. Algorithm**
The core algorithms in GA were implemented in this class. The class definition shows the general outline of GA with call hierarchy of the methods. The first method is the one that checks the GA run parameters which defines some logical constraints such as the cut length size cannot exceed the size of the chromosome.

A GA run is performed by the method with the same name with two input parameters namely the population and the number of generations. This method calls the `evolvePopulation` method to perform the following operations inherent to any GA:

1. Selection
2. Recombination
3. Mutation
4. Fitness evaluation and clean up

**Selection** This operation requires the initial fitness values of the individuals to be computed for two implemented selection approaches, roulette-wheel and tournament [6]. The former method uses the accumulated fitness values to perform selection. In this approach, the selection probability of an individual is directly proportional with its fitness value. The latter selection method creates specified number of tournaments of specific tournament size. The fittest individuals of all tournaments are selected, creating a new tournament and finally yielding a selected individual. Both selection methods fill a mating-pool which will be used for creating new individuals to be added to the population.

**Recombination** This phase correspond to the `exploitation' stage [7] where the current genetic information is crossed-over to produce siblings. The library offers several different forms of recombination including one-point, two-point, uniform and whole-arithmetic recombination for both integer (or binary) and real-valued gene information.

**Mutation** The `exploration' stage in GA aiming to avoid local optima is the mutation operation. Here, the genetic information in the chromosome is altered in a random fashion. The mutation in the algorithm is applied to every individual in the population, except for the elite individuals i.e. ones with high fitness values. The reason behind this is that the population should keep the best-so-far individuals to keep the overall fitness of the population in a non-decreasing trend.

**Fitness evaluation and clean up** The final evaluation of the individual fitness values are carried out at this stage in order to take the effect of the mutation operation into account for the overall population fitness. The last operation here is sorting the individuals in descending order of fitness values and removing the ones with lower fitness values so that the population sizes will remain under the limit.

#### 2.1.5. Benchmark and test functions

The developed library employs a number of test functions in a benchmark form used for evaluating the performance of the parallel implementation in comparison with the serial version. Table 1 depicts the most common benchmark functions used in the evaluation.

**Table 1** All functions are for f($x_1$ … $x_n$)

| Function | Formula | Domain |
|---|---|---|
| OneMax | $\sum_{i=1}^{N} x_i$ | $x_i \in \{0,1\}$ |
| Sphere | $\sum_{i=1}^{n} x_i^2$ | $-5.12 \leq x_i \leq 5.12$ |
| Ackley | $-20exp(-0.2\sqrt{\frac{1}{n}\sum_{i=1}^{n} x_i^2}) - exp(\frac{1}{n}\sum_{i=1}^{n} cos(2\pi x_i)) + 20 + e$ | $-32 \leq x_i \leq 32$ |
| Schwefel | $\sum_{i=1}^{n}(-x_i sin(\sqrt{\lvert x_i \rvert})) + 418.98 \cdot n$ | $-512 \leq x_i \leq 512$ |
| Rastrigin | $10n + \sum_{i=1}^{n}(x_i^2 - 10cos(2\pi x_i))$ | $-5.12 \leq x_i \leq 5.12$ |
| Rosenbrock | $\sum_{i=1}^{n-1}(100(x_i^2 - x_{i+1})^2 + (1 - x_i)^2)$ | $-2.048 \leq x_i \leq 2.048$ |
| Griewank | $1 + \frac{1}{4000}\sum_{i=1}^{n} x_i^2 - \prod_{i=1}^{n} cos(\frac{x_i}{\sqrt{i}})$ | $-512 \leq x_i \leq 512$ |

The test functions are indexed in the `GenericHeader.h` with a constant value. These indices were used to access a dynamic function pointer array to call these test functions. The mentioned header file also includes compiler directives to include the collection classes such as the `vector`.

#### 2.1.6. Utility

This class is the container class for all GA run parameters involving the initial and maximum population sizes, tournament size, mutation rate, recombination type, etc. Two utility methods for random number and index generation also reside as the `static` members of this class.

## 3. ILLUSTRATIVE EXAMPLE

This section presents examples to run a minimal GA. The first step in the algorithm is modelling the problem with a suitable `GeneModel`. This is done by changing the data variable in this class for a single gene. The changes should be performed for the complete chromosome structure. Before proceeding, it is also imperative to define the algorithm specific parameters which include the initial and maximum population sizes, mutation rates, number of elite individuals as well as the selection method to be used.

```
Sample sample;
vector<GeneModel> sampleGene;
for ( int i =0; i<Utility :: GENE SIZE; i++)
{
double value = Utility::randomValueGenerator();
GeneModel geneModel;
geneModel.data = value;
sampleGene.push_back(geneModel) ;
}
```

The created chromosome structure can then be assigned to an individual in the GA using

```
Sample.setSample(sampleGene);
```

A population of individuals are then created by adding the created samples to the population:

```
Population population;
population.addSample(sample);
```

Note that this population comprises of a single individual after the previous code snippet. Any number of sample can be added to the population according to the problem definition using the same `addSample` function.

Once a population of individuals are created, then the GA can be run using the static run function of the Algorithm class for a specified number of generations for a given test function

```
Algorithm::run(population, NumberOfGenerations, ONEMAX);
```

The GA can be run in parallel mode by setting the `RUN_PARALLEL` variable true before calling the run function.

## 4. IMPACT AND PERFORMANCE EVALUATION

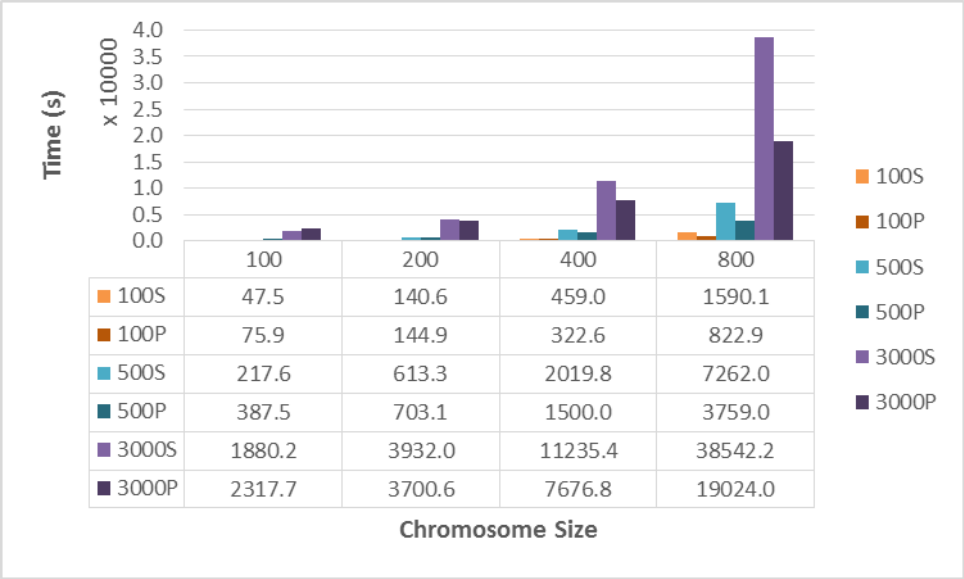
In order to evaluate the performance of the parallel implementation we conducted an experiment for optimizing various benchmark functions. Visual Studio 2012 was used as the compiler and development environment on a machine that has dual core Intel Core i5-3470 processor (3.20Ghz) with 4GB RAM and running 32 bit Windows 7. It is worth reiterating here that both the source code and the TBB library can be run on any operating system.

A t-test [8] was applied to compare the serial and parallel implementations. Table 2 shows the achieved performance using parallelism. Results indicate that there are statistically significant differences between serial and parallel execution modes. t-Stat values smaller than the t Critical values for both one and two tailed test point to significance of the results with P(T≤t) very close to zero as a measure of the confidence on the results.

**Table 2** t-test for demonstrating the performance difference in GA execution in serial and parallel modes

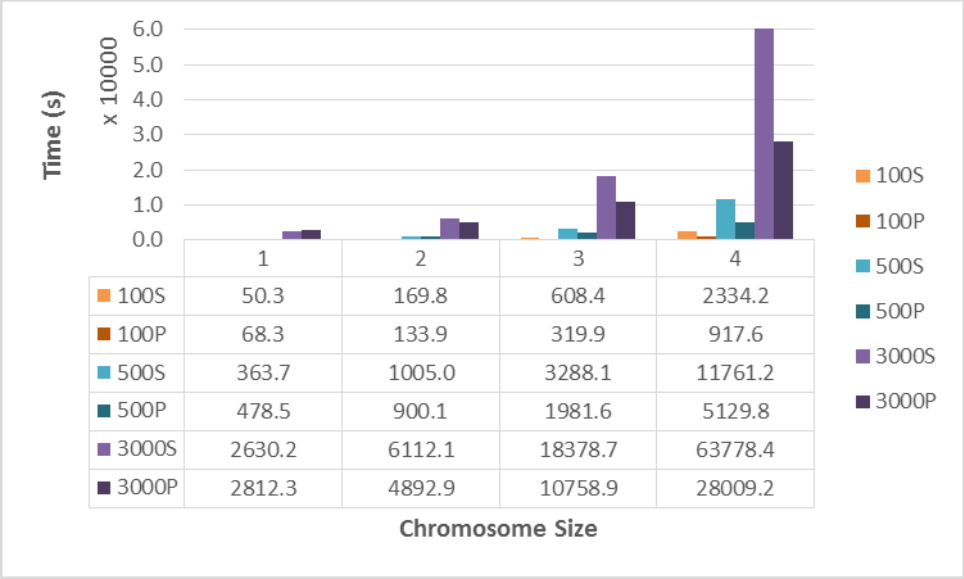
| | **Serial** | **Parallel** |
|---|---|---|
| Mean | 14111.2239 | 5816.9587 |
| Variance | 906459545.6 | 1.11E+08 |
| Observations | 84 | 84 |
| Pearson Correlation | 0.9864 | |
| Hypothesized Mean Difference | 0 | |
| df | 83 | |
| t Stat | 3.8446 | |
| P(T≤t) one-tail | 0.0001 | |
| t Critical one-tail | 1.6634 | |
| P(T≤t) two-tail | 0.0002 | |
| t Critical two-tail | 1.9890 | |

Figs.1 to 7 show individual performance results for different chromosome and population sizes for the tested seven benchmark functions of varying complexity where the performance differences can also be observed.



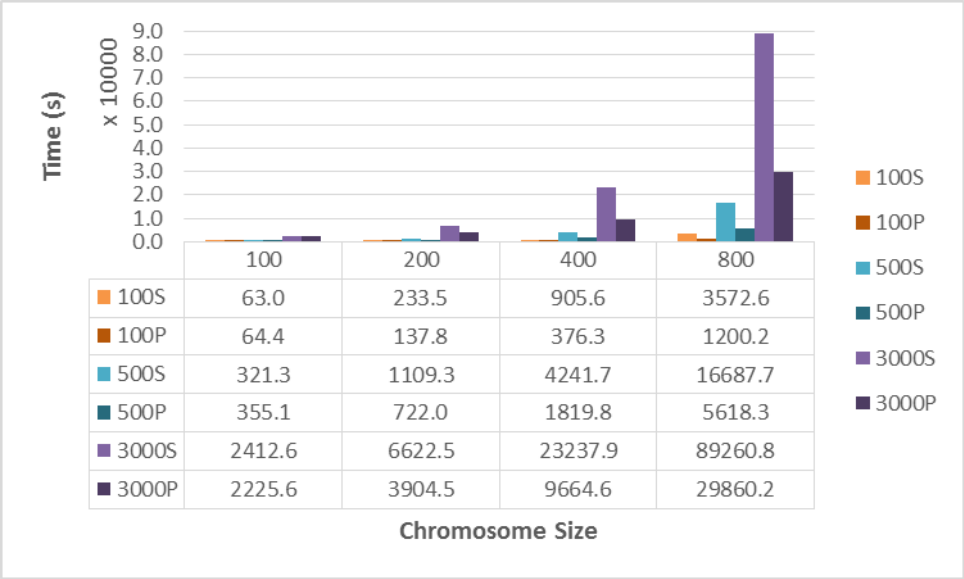

| | 100 | 200 | 400 | 800 |
|---|---|---|---|---|
| 100S | 47.5 | 140.6 | 459.0 | 1590.1 |
| 100P | 75.9 | 144.9 | 322.6 | 822.9 |
| 500S | 217.6 | 613.3 | 2019.8 | 7262.0 |
| 500P | 387.5 | 703.1 | 1500.0 | 3759.0 |
| 3000S | 1880.2 | 3932.0 | 11235.4 | 38542.2 |
| 3000P | 2317.7 | 3700.6 | 7676.8 | 19024.0 |

*Figure 1 Performance of the serial and parallel versions of the genetic algorithm for OneMax function. S:Serial, P: Parallel*



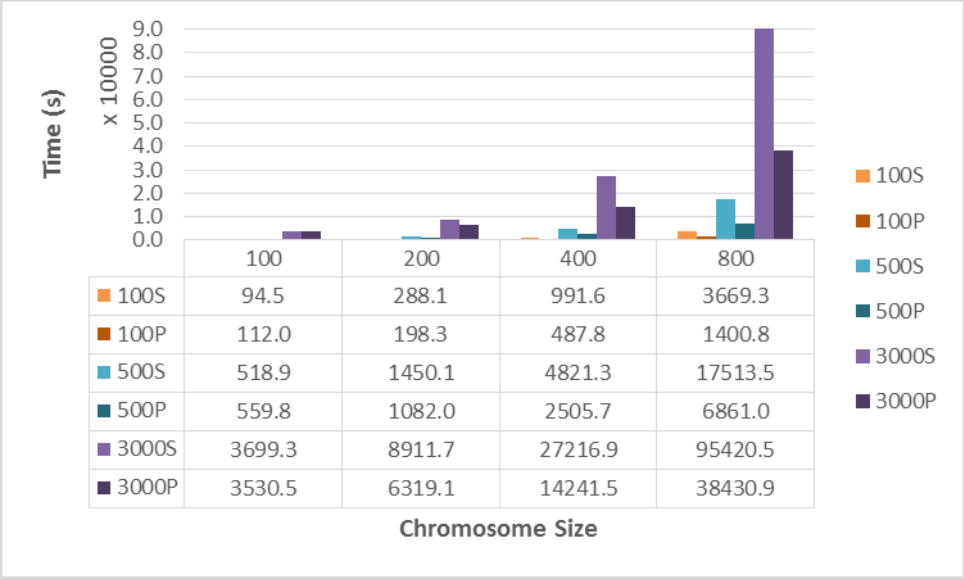

| | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 100S | 50.3 | 169.8 | 608.4 | 2334.2 |
| 100P | 68.3 | 133.9 | 319.9 | 917.6 |
| 500S | 363.7 | 1005.0 | 3288.1 | 11761.2 |
| 500P | 478.5 | 900.1 | 1981.6 | 5129.8 |
| 3000S | 2630.2 | 6112.1 | 18378.7 | 63778.4 |
| 3000P | 2812.3 | 4892.9 | 10758.9 | 28009.2 |

*Figure 2 Performance of the serial and parallel versions of the genetic algorithm for Sphere function. S:Serial, P: Parallel*



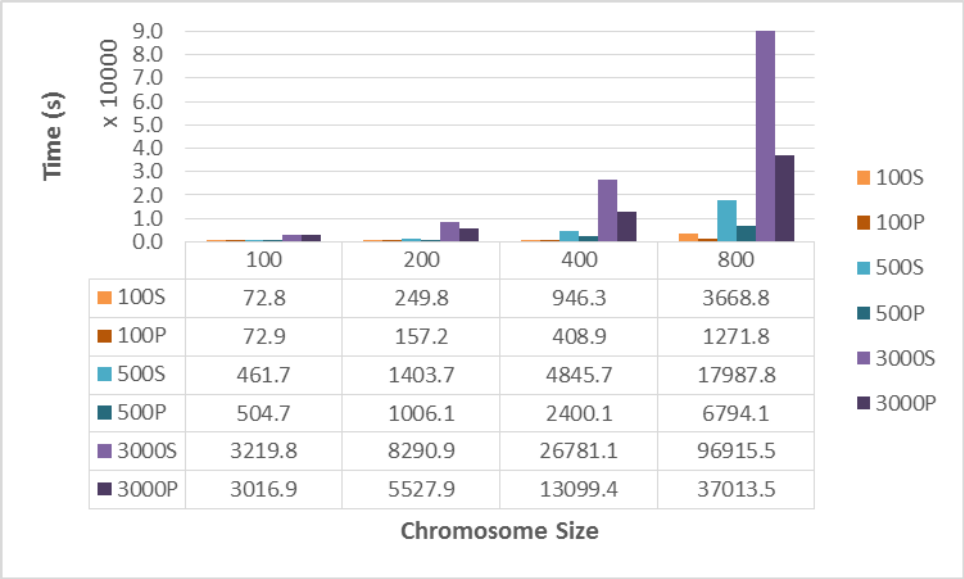

| | 100 | 200 | 400 | 800 |
|---|---|---|---|---|
| 100S | 63.0 | 233.5 | 905.6 | 3572.6 |
| 100P | 64.4 | 137.8 | 376.3 | 1200.2 |
| 500S | 321.3 | 1109.3 | 4241.7 | 16687.7 |
| 500P | 355.1 | 722.0 | 1819.8 | 5618.3 |
| 3000S | 2412.6 | 6622.5 | 23237.9 | 89260.8 |
| 3000P | 2225.6 | 3904.5 | 9664.6 | 29860.2 |

*Figure 3 Performance of the serial and parallel versions of the genetic algorithm for Ackley function. S:Serial, P: Parallel*



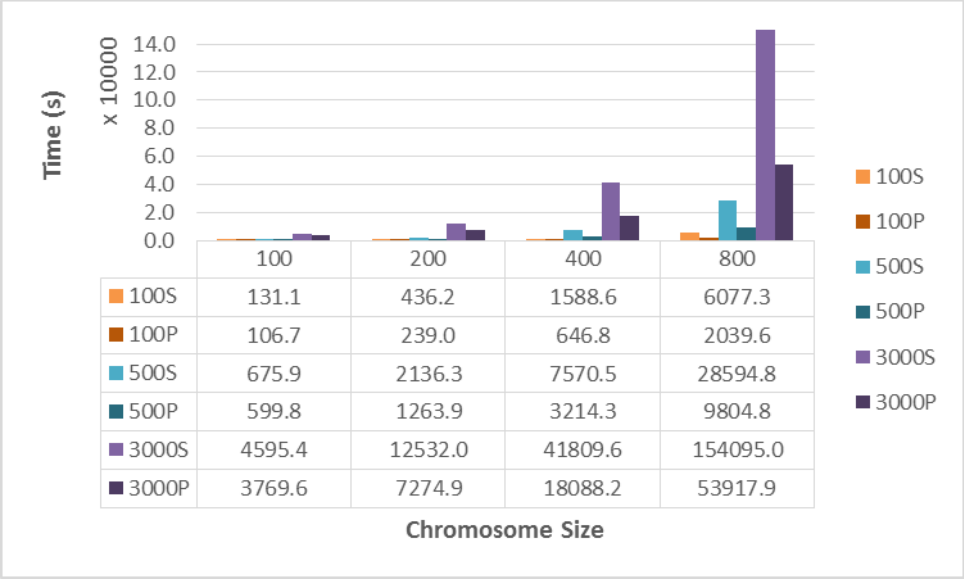

| | 100 | 200 | 400 | 800 |
|---|---|---|---|---|
| 100S | 94.5 | 288.1 | 991.6 | 3669.3 |
| 100P | 112.0 | 198.3 | 487.8 | 1400.8 |
| 500S | 518.9 | 1450.1 | 4821.3 | 17513.5 |
| 500P | 559.8 | 1082.0 | 2505.7 | 6861.0 |
| 3000S | 3699.3 | 8911.7 | 27216.9 | 95420.5 |
| 3000P | 3530.5 | 6319.1 | 14241.5 | 38430.9 |

*Figure 4 Performance of the serial and parallel versions of the genetic algorithm for Schwefel function. S:Serial, P: Parallel*



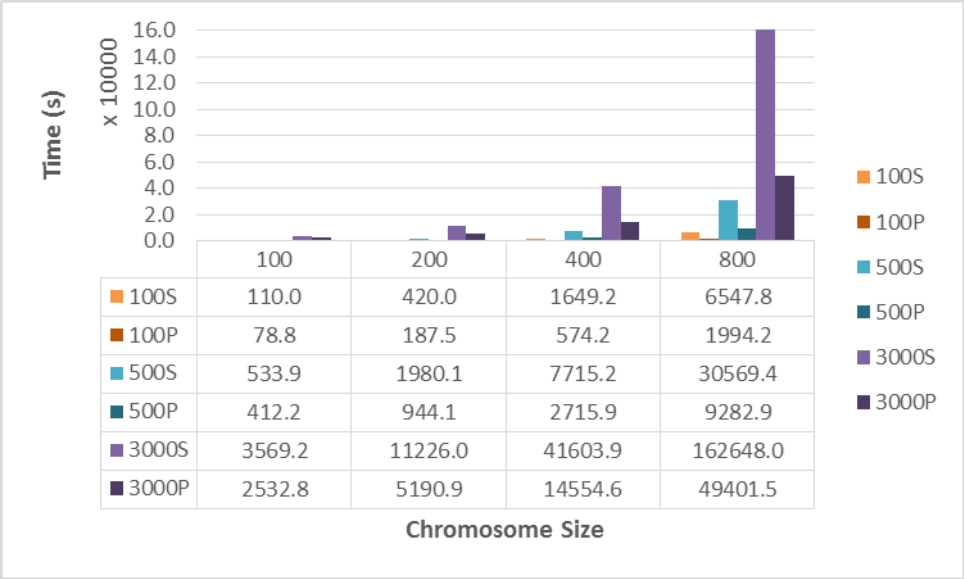

| | 100 | 200 | 400 | 800 |
|---|---|---|---|---|
| 100S | 72.8 | 249.8 | 946.3 | 3668.8 |
| 100P | 72.9 | 157.2 | 408.9 | 1271.8 |
| 500S | 461.7 | 1403.7 | 4845.7 | 17987.8 |
| 500P | 504.7 | 1006.1 | 2400.1 | 6794.1 |
| 3000S | 3219.8 | 8290.9 | 26781.1 | 96915.5 |
| 3000P | 3016.9 | 5527.9 | 13099.4 | 37013.5 |

*Figure 5 Performance of the serial and parallel versions of the genetic algorithm for Rastrigin function. S:Serial, P: Parallel*




| | 100 | 200 | 400 | 800 |
|---|---|---|---|---|
| 100S | 131.1 | 436.2 | 1588.6 | 6077.3 |
| 100P | 106.7 | 239.0 | 646.8 | 2039.6 |
| 500S | 675.9 | 2136.3 | 7570.5 | 28594.8 |
| 500P | 599.8 | 1263.9 | 3214.3 | 9804.8 |
| 3000S | 4595.4 | 12532.0 | 41809.6 | 154095.0 |
| 3000P | 3769.6 | 7274.9 | 18088.2 | 53917.9 |

*Figure 6 Performance of the serial and parallel versions of the genetic algorithm for Rosenbrock function. S:Serial, P: Parallel*




| | 100 | 200 | 400 | 800 |
|---|---|---|---|---|
| 100S | 110.0 | 420.0 | 1649.2 | 6547.8 |
| 100P | 78.8 | 187.5 | 574.2 | 1994.2 |
| 500S | 533.9 | 1980.1 | 7715.2 | 30569.4 |
| 500P | 412.2 | 944.1 | 2715.9 | 9282.9 |
| 3000S | 3569.2 | 11226.0 | 41603.9 | 162648.0 |
| 3000P | 2532.8 | 5190.9 | 14554.6 | 49401.5 |

*Figure 7 Performance of the serial and parallel versions of the genetic algorithm for Griewank function. S:Serial, P: Parallel*

## 5. CONCLUSION

This paper presented a GA framework for optimisation problems. The framework includes various selection, recombination and mutation operators commonly used in such evolutionary algorithms. It is well known that evolutionary algorithms may take a significant amount of time to yield optimal results. An important feature of the developed framework is that it is based on the Intel's TBB library allowing parallelism in order to improve the execution performance of GA. The library is cross-platform, hence can be used on different operating systems.

The framework was tested on several benchmark functions, results have revealed that the parallel mode has significantly lower execution time. A statistical test was also used to confirm this outcome. It has been shown that up to three-fold speed-up can be achieved on a dual core computer.

The authors believe that the framework developed in this study will be beneficial to a wider range of users when freely available online for both research and education communities.

## 6. ACKNOWLEDGEMENTS

The framework presented in this paper can be downloaded from
http://comp.eng.ankara.edu.tr/arastirma/laboratuvarlar/saat/

## 7. REFERENCES

[1] Z. Michalewicz, Genetic Algorithms + Data Structures = Evolution Programs. Artificial intelligence, Springer, 1996.
[2] A. E. Eiben and J. E. Smith, Introduction to Evolutionary Computing. SpringerVerlag, 2003.
[3] S. Zhang and Z. He, \Implementation of parallel genetic algorithm based on cuda," in Advances in Computation and Intelligence, pp. 24-30, Springer, 2009.
[4] P. Pospichal, J. Jaros, and J. Schwarz, “Parallel genetic algorithm on the cuda architecture," in Applications of Evolutionary Computation, pp. 442-451, Springer, 2010.
[5] C. Pheatt, Intel®; threading building blocks," J. Comput. Sci. Coll., vol. 23, no. 4, pp. 298-298, 2008.
[6] D. E. Goldberg and K. Deb, “A comparative analysis of selection schemes used in genetic algorithms," Foundations of genetic algorithms, vol. 1, pp. 69-93, 1991.
[7] D. E. Goldberg, Genetic Algorithms in Search, Optimization and Machine Learning. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1st ed., 1989.
[8] Student, “The probable error of a mean," Biometrika, vol. 6, no. 1, pp. 1-25, 1908.