IEEE Computational Intelligence Society
Transactions on Evolutionary Computation

A fast genetic algorithm model for shared memory parallel computing

Journal:	*Transactions on Evolutionary Computation*
Manuscript ID:	TEVC-00182-2009
Manuscript Type:	Regular Papers
Date Submitted by the Author:	03-Jun-2009
Complete List of Authors:	Nematizadeh, Fariborz; Tehran University of Medical Sciences, School of Medicine Cai, Ximing; University of Illinois at Urbana-Champaign, Department of Civil and Environmental Engineering
Keywords:	Genetic algorithms, Shared memory systems, Parallel algorithms, Random number generation

A fast genetic algorithm model for shared memory parallel computing

Banafsheh Zahraie[1] Fariborz Nematizadeh[2], Ximing Cai[3]

Abstract

Parallel computing has been used in different scientific fields for high performance computing (HPC), but interest in it has been growing rapidly due to physical constraints that prevent the increase in clock frequency of Central Processing Units (CPUs). In the past, parallel computing was only possible through the use of multiple CPUs but the vast availability of multi-core CPUs in PCs that are widely used today makes parallel computing a promising way of harnessing this additional computational power to shorten execution time of algorithms. In this study, an innovative approach has been developed for parallel implementation of the Genetic Algorithm (GA) optimization technique. In the Parallel GA (PGA) Model, a pipeline has been designed in which a task-based parallelization of evolution operators (selection, mutation, and crossover) and evaluation of chromosomes has been implemented. In this model, the total parallelization of the GA's computational burden has been implemented and near-linear scalability with the number of cores has been achieved. Another problem addressed in this study is the inability of some of pseudo-random number generators (RNGs) to produce true uniformly distributed random

[1] Associate Professor and member of the Center of Excellence on Infrastructure Engineering and Management, University of Tehran, Tehran, Iran (bzahraie@ut.ac.ir)
[2] Water Institute, University of Tehran (fariborz.nz@gmail.com)
[3] Assistant Professor, University of Illinois at Urbana-Champaign (xmcai@illinois.edu)

numbers. Non-uniformly distributed random numbers are a major problem in GA models because truly uniform distributions are required to keep the diversity in the population. A uniform distributor (UD) implemented in the PGA model for use with a thread-safe set of Mersenne Twister RNGs provides dramatic convergence improvements compared with simple random initialization. Multiple tests were conducted on double- and quad-core computer systems. The issue of scalability with the number of cores is discussed and the effects of proposed UD on the convergence speed of the model are evaluated.

Keywords: Multi-core systems, Parallel Genetic Algorithms, uniform random numbers, parallel computing, genetic algorithms

1. Introduction

Historically, designers and manufacturers of Central Processing Units (CPUs) have increased processing power of their products mainly through increasing processor frequency but recently they have not been able to elapse the 4-5 GHz barrier due to heat dissipation and other limitations imposed by laws of physics. Instead, they have decided to increase processing performance by incorporating more than one independent Execution Unit (Core) inside a single physical CPU. Through this approach, they have been able to reach near-linear scalability of processing power while keeping the CPU frequency below the aforementioned margin. However, the idea of using more than one processing unit to conduct complex computations is not new. High-end Motherboards for servers and workstations have been constructed with multiple (usually 2 or 4) sockets that accepted multiple physical CPUs on the same board. Graphic Processing Units (GPUs) are another well-known example of parallel processing units.

The problem with parallel processing is that software has to be specifically written to run in multiple threads, as it does not offer an immediate increase in performance in the same manner that increased clock speeds do. A single-threaded application can only make use of one core. Multi-core computers give the opportunity to solve high-performance applications more efficiently by using parallel computing (multi-threaded applications). Genetic Algorithms (GA) and other evolutionary optimization techniques need a lot of computing power to solve a problem, especially when they have a large number of decision variables. Before the development of multi-core CPUs, the most practical way to speed up the process of finding the optimal solution was to use very expensive multi-CPU computers or multicomputers (e.g. cluster) [1]. Chai et al. [2] show some of the advantages of multi-core architecture while Dongarra et al. [3] present several applications of parallel computing. The only previous study on the development of parallel GA for multi-core systems of which the authors are aware is a study in which Serrano et al. [4] employed the Distributed Computing Toolbox of MATLAB 7.4a.

The vast availability of multi-core CPUs in today's commonly available PCs makes parallel computation a promising way of harnessing this additional computational power to shorten the execution time of many numerical models, including GAs. This paper examines the benefits that the Parallel Genetic Algorithm (PGA) model can provide in terms of reduction in execution time The performance of the proposed PGA model is assessed with tests on 12 arbitrarily chosen benchmark functions of low to relatively high dimensions (up to 100). The tests were made on double- and quad-core computers and a comparison of the reduction in execution time in relation to the number of cores is shown.

Another issue addressed in this paper is the non-uniformity of pseudo random numbers that RNGs produce. For over 40 years, stochastic modelers have been using computerized random

number generators (RNGs) that are expected to produce random numeric sequences that fit a specified statistical distribution [5]. RNGs are needed for many computer applications, such as simulation of stochastic systems, probabilistic algorithms, and evolutionary optimization algorithms to name a few. These so-called random numbers may come from a physical device but are more often produced with a deterministic function (hence they are called pseudo-random number generators) or algorithmic RNGs. Since they have a deterministic and periodic output, it is clear that they do not produce independent random variables and, consequently, cannot pass all statistical tests of uniformity and independence [6]. In fact, several popular RNGs, some of which are available in commercial software, fail very simple tests [6, 7]. The scientific literature is also filled with examples of inappropriate RNGs. A bad RNG can completely ruin a researcher's analysis [8]. Coddington [9, 10] and Meyer et al. [5] discuss the impacts of this weakness in the fields of physics and hydroclimatology, respectively.

The convergence speed of GAs is highly dependent on their initialization. Usually, GAs are initialized randomly. When a perfect RNG (produces a truly uniform distribution) is used, this promotes maximum diversity in the initial population, which can help counter suboptimal convergence to local optima, as discussed in [11]. However, due to weaknesses of RNGs that are used for random initialization of initial population in GAs, different adjusted, biased initialization techniques have been suggested in previous studies [12, 13, 14, 15]. In [14], the diversity of the initial population was maximized by ensuring that the initial population was uniformly distributed across the search space. Chou and Chen [14] suggested splitting the search space into sub-spaces for one-dimensional problems, but for multi-dimensional cases, the diagonal linear subspaces of the search space that appeared to contain important information were considered for biased initialization. Multiple processors were used to implement the

optimization in [15], in which the search space was divided into regions assigned to each processor, and each set of individuals in the initial population was biased to search the corresponding region.

In this study, we have used Mersenne Twister RNG developed by Matsumoto and Nishimura [16], which is a highly regarded RNG for achieving better randomness. We also employed a biased initialization scheme and assessed its effects on the convergence of the PGA model. For this purpose, we have proposed a Uniform Distributor (UD) suitable for high-dimensional problems to be used along with the Mersenne Twister RNG to improve the diversity of the initial population produced in GA in a way that it covers the whole search space uniformly.

The rest of this paper is organized as follows. First, the PGA model algorithm is discussed in Section 2.1. The uniform distributor (UD) which has been proposed for biased initialization and experiments used to evaluate model performance are presented in Sections 2.2 and 2.3, respectively. Finally, the results of testing the model to find the optimal solutions for 12 benchmark functions are discussed in Section 3.

2. Methodology

2.1. PGA Model

First, a hierarchy of classes for the objects in the evolutionary PGA model was designed for thread-safety operation and concurrent access and execution of properties and methods. The parallelism of the PGA model was achieved with concurrent execution of evolutionary operators including selection, crossover, and mutation, as well as the evaluation of chromosomes throughout the evolving population.

A task-based pipeline was designed based on Intel Threading Building Blocks (TBB). Each step of the evolutionary algorithm in GA was mapped into a set of parallel tasks and the sequence of these steps was incorporated into the pipeline. A master thread executing the pipeline creates a thread pool based on the number of available cores and dynamically allocates a subset of tasks to each thread for execution.

The application is designed as a standard ANSI C++ template-based hierarchy of classes with portability and ease of extendibility in mind. The class model was developed using Microsoft Windows Vista SP1 32 bit using Intel C++ compiler v10.1 for Windows. We also used the following libraries: Boost library v1.38 [17], TBB v2.1, [18], STLport v5.21. [19] and SFMT [20]. All of these libraries are open source and are available across platforms. The main challenges that we tried to address were: 1) Generation of true random numbers with concurrent calling of random number generator function (the standard C++ rand() command is not thread safe), and 2) Concurrent access to data.

The first challenge can be a major issue in a GA model because almost all different types of selection, crossover and mutation operators are highly dependent on random numbers generated in the model. If the same random number generator is called concurrently by different threads, then the results are unpredictable. Usually, the same number is returned to all threads, which hinders the evolution process significantly. The second challenge concerns issues with racing conditions in which the separate threads of execution depend on a shared state or sequence of events initiated by other threads, mutual access to shared resources, validity of concurrently modified data, and avoiding deadlocks and thread collision. To overcome the aforementioned obstacles, the following design was chosen:

- A set of mutually synchronized Mersenne Twister RNGs, each with the period of 2^{19937}-1

- Stateless class design with synchronization on currently accessed members

2.2. Uniform Distributor (UD)

Since GAs became popular through the work of John Holland [21] and Davis Goldberg [22], modelers have used computerized RNGs to produce uniform random numeric sequences for different GA operators. As explained earlier in the paper, most of these RNGs do not satisfactorily produce independent and uniformly distributed random numbers. Non-uniformly distributed short sequences of random numbers are a major problem in GA models because truly uniform distributions are required to preserve the diversity in the population. RNGs are used in the first step of the evolution process in generation of initial population to make sure that all different parts of the search space are uniformly covered by the randomly generated solutions. If the randomly generated population does not follow a uniform distribution in each dimension, then some parts of the search space might remain untouched and it may take a long time for the model to reach those regions with crossover and mutation operators. In that case, a modeler's choice of mutation and crossover operators among different available techniques might become very influential, making the whole modeling procedure highly dependent on the modeler's experience.

In this study, Mersenne Twister RNG is used because it combines speed with good statistical properties and an extremely long periodicity It is used in Goose, SPSS, EViews, GNU R, VSL and in many other software packages [7]. Along with using Mersenne Twister RNG, a UD is also proposed for biased initialization to ensure enough diversity in the initial population to cover all different parts of the search space in every dimension. The main idea behind the proposed approach is to make sure that the initial population is produced in a way that the whole

search domain is covered as much as possible and as quickly as possible through the evolutionary process.

To understand this concept, consider a search space with n dimensions. The following equation has been used in this study for estimation of population size (adopted from Yu et al. [23]):

$$PS = 30n \times \ln(n) \quad (1)$$

The search domain for each decision variable is discretized into m equal non-overlapping classes. A random number is generated for each dimension in each class. In order to have all different combinations of the discretized values in the initial population (having one solution in each sub-region of the search space as also discussed in [14]), m^n chromosomes should be generated. Assume a sample case of 100 decision variables with $m = 10$. The number of chromosomes in the initial population for this case will be 10^{100}, which requires a huge number of fitness function evaluations for the GA model to find the optimal/near-optimal solutions. This population size can make the GA model as inefficient as dynamic programming techniques when it faces the curse of dimensionality. On the contrary, it is not necessary to consider all of the combinations of the discretized values in GAs because applying crossover operators over selected individualsyields different combinations of discretized values during the evolution process if simple one point, two point, or uniform crossover operators are chosen.

Fig. 1 shows a simple example of a two-dimensional problem in which the search domain in each dimension is only discretized to two classes. As it is shown in this figure, to have one chromosome in each sub-region of the search space, 2^2 chromosomes should be generated in the initial population. However, in the figure, it is shown that by generating only two chromosomes, the two other combinations can be generated by applying a single point crossover operator.

Therefore, in the proposed biased initialization, it is only necessary to ensure that, a random number is generated in the initial population for each dimension and each class. For the aforementioned case, $m \times n$ random numbers should be generated. Based on Eq. (1), m can be estimated as follows:

$$m \times n = 30n \times \ln(n) \quad \Rightarrow \quad m = 30\ln(n) \tag{2}$$

For example, if the number of decision variables or dimensions is equal to 100, 138 classes can be considered for random population initiation. It should be mentioned that, when using the mutation operator, the search will be expanded to different values in each class. The pseudo code for the biased initial population generation is as follows:

Begin Initialization;

Read number of decision variables (genes);

Estimate population size using Eq. (1);

Estimate number of classes using Eq. (2);

For each decision variable $\left(x_i\right)$ **do**

Read search domain $\left(x_i^{\min}, x_i^{\max}\right)$;

Classify search domain to *m* non-overlapping classes;

For each class **do**

Generate a random number;

Assign the random number to the i^{th} gene of a chromosome in the initial population;

End for;

End for;

End Initialization;

The results from tests of the proposed PGA model with and without using UD on 12 benchmark functions is presented in Section 3 of the paper.

2.3. Experiments to Evaluate PGA Performance

The PGA model was tested on two computer systems with the following configurations:

(1) Quad-Core System: CPU Intel Q6600 2.4 GHz FSB 1066 MHz– 8 MB cache memory - 3 GB DDR2 of main memory running Microsoft Windows XP Professional SP3 32 bit

(2) Dual-Core System: CPU Intel P9500 2.53 GHz FSB 1066 MHz– 6 MB cache memory - 4 GB DDR2 of main memory running Microsoft Windows Visa Business SP1 32-bit

Numerical experiments were conducted to test the effectiveness and efficiency of the PGA. Twelve benchmark functions from two categories [17] were selected, covering a broad range for the purpose of demonstrating the robustness and reliability of this algorithm. Table 1 lists the 12 test functions and their key properties. These functions can be divided into two categories based upon their complexity. $f_1 - f_7$ are unimodal functions, which are relatively easy to optimize, but the difficulty increases as the problem dimension becomes higher. Meanwhile, $f_8 - f_{12}$ are multimodal functions with many local optima, which represent the most difficult class of problems for many optimization algorithms. As an example, Fig. 2 shows the surface landscape of f_8 when the dimension is set to two. Some functions possess rather unique features. For example, f_6 is a discontinuous step function with a single optimum.

Usually, convergence rates are the main of attribute of interest for unimodal functions, since optimizing these functions to a satisfactory accuracy is not a major issue. For multimodal functions, however, the quality of the final results is more crucial since it reflects the PGA's ability to escape from deceptive local optima and locate the desired near-global solution.

3. Results

The PGA model developed in this study is a real-coded GA with uniform crossover, uniform mutation, and deterministic tournament selection with a tournament size of three. The convergence criteria which have been used in the model are (1) no improvement of more than 10^{-4} and (2) no worsening of the best fitness value in 50 consecutive generations. Test results of the PGA model on the benchmark functions are presented in the next two subsections of the paper. These two subsections are aimed to show the performance of the model and how it correlates with the number of cores on the system and UD effects on the convergence speed.

3.1. Scalability with number of cores

To test the scalability of the PGA model, execution times per 100 generations were compared for 10 runs of the model on a quad-core and a dual-core computer. The hardware and software specifications of these computers were presented in the previous section of the paper. It is well-known that the number of generations needed for GA models to reach optimal/near-optimal solutions varies from run to run. Therefore, in this study, the execution time per 100 generations has been chosen for the purpose of comparison.

Fig. 3 shows the comparison between execution time of the model on the dual-core computer with single and double-core configurations. As it can be seen in Fig. 3-a, when using

two cores instead of one core, a 39% reduction in the average execution time of the PGA model for the 12 benchmark functions was achieved. Fig. 3-b shows that the percent of decrease in execution time is different for the benchmark functions, as more mathematically complex functions require larger numbers of CPU cycles. For example, functions 1 and 6 need very few execution cycles to reach the optimum so the overhead of creating and managing threads is evident. On the contrary, Function 11 needs lots of CPU cycles and the parallel overhead p is -overcame. Function 7, despite a simple definition, uses the random number generator, which is mathematically intensive. Therefore, PGA will be more advantageous for problems with mathematically complex fitness functions.

The same test was carried out on the quad-core computer, the results of which are shown in Fig. 4. The utilization of four cores instead of two resulted in more than a 38% reduction in the average execution time of the PGA model for the 12 benchmark functions.

3.2. UD effects on convergence speed

To evaluate the effectiveness of the proposed UD, the PGA model was run 10 times for each of the benchmark functions with and without biased initialization using UD. The Mersenne Twister RNG was used in both sets of runs. Table 2 shows a comparison between execution time and the number of generations required for the PGA model to converge with and without using uniform distributor (UD). As it can be seen in this table, except for the function, f_7, the decrease in execution time of the model due to UD ranged from 86 to 95.7 percent with the lone exception of function, f_1. This shows a very significant improvement in PGA model efficiency in finding the optimal/near-optimal solutions. Adding UD did not result in improved model performance because the PGA model was already capable of finding the optimal solution of the function, f_7,

without using UD. The average improvement in execution time of the whole set of 12 benchmark functions was 86%. The number of generations required to reach the optimal solution was also reduced by 84.4% r (average for 12 benchmark functions) when UD was used in the model. Figures 5 and 6 show a comparison between execution time and number of generations required for convergence for different benchmark functions. As it can be seen in Figs. 5-b and 6-b, almost the same level of improvement in convergence speed of the PGA model was seen across the board except for function f_7. The PGA model was able to find the optimal solution for this function in less than 50 generations, which is lower than the minimum threshold of the number generations that the convergence criteria require. Therefore, adding UD does not show an improvement for this specific function.

It should be noted that using UD did not add any major computational burden to the PGA model. Fig. 7 shows the 100-generation execution times of the PGA model with UD. The average execution time of the PGA model when UD was employed was only 0.1% more than the execution time when only Mersenne Twister RNG was used.

4. Conclusions

In this study, a Parallel GA (PGA) model featuring task-based parallelization of evolution operators (selection, mutation, and crossover) and evaluation of chromosomes has been implemented. In this model, the total parallelization of GA computational burden and near-linear scalability with the number of cores has been achieved. Doubling the number of cores reduced the execution time of the model by nearly 40 percent. The proposed biased initialization has also resulted in the significant reduction of the number of generations needed for convergence. Improvements between 87 to 98 percent for twelve different benchmark functions

that were tested with the model were observed. In addition, a dramatic reduction execution time between 86 to 96 percent in the PGA model execution time was achieved.

The very high quality libraries which are available as open source contributed significantly to the implementation of the PGA model. This experience also showed that recent versions of compilers and libraries provide effective support for multi-core development. Future development of engineering software can benefit from this well-established balance between hardware and software in the field of parallel computing.

Finally, general purpose computation on graphics processing units (GPGPU) has received a lot of attention in the parallel computing community. GPUs have a parallel architecture with up to hundreds of cores, with each core capable of running thousands of threads simultaneously. They offer impressive performance benefits in certain parallel coded applications that are suited for this architecture. We plan to adapt the PGA model for GPGPUs to benefit from their massively parallel execution engines. Conditional and unconditional branching does not harm parallel processing over CPUs, but on GPGPU, it might cause the execution grid to be run as serial sequences. Therefore, the PGA model will need to be modified to benefit from GPGPU parallelization.

5. Acknowledgement

This study has been partially supported by a grant from the Islamic Bank of Development, an agency that offers widespread support in the development of science and engineering in member countries.

6. References

[1] Alba, E., Luna, F., Nebro, A.J.: Advances in Parallel Heterogeneous Genetic Algorithms for Continuous Optimization. International Journal of Applied Mathematics and Computer Science 14, 317–333 (2004)

[2] Chai, L., Gao, Q., Panda, D.K.: Understanding the Impact of Multi-Core Architecture in Cluster Computing: A Case Study with Intel Dual-Core System. In: The 7th IEEE International Symposium on Cluster Computing and the Grid (CCGrid 2007) (2007)

[3] Dongarra, J., G. Fox, K. Kennedy, L. Torczon, W. Gropp: Sourcebook of Parallel Computing. Morgan Kaufmann Publishers, San Francisco (2003).

[4] Serrano R., J. Tapia, O. Montiel, R. Sepúlveda and P. Melin (2008), "High Performance Parallel Programming of a GA Using Multi-core Technology", published in O. Castillo et al. (Eds.): Soft Computing for Hybrid Intel. Systems, SCI 154, pp. 307–314, Springer-Verlag Berlin Heidelberg.

[5] Meyer, C. R., C. S. Renschler, and R. C. Vining: Implementing quality control on a random number stream to improve a stochastic weather generator. Hydrological Processes, 22, 1069–1079 (2008).

[6] L'Ecuyer C. and R. Simard: TestU01: A C Library for Empirical Testing of Random Number Generators. ACM Transactions on Mathematical Software, 33(4), Article 22 (2007).

[7] L'Ecuyer, P.: Software for uniform random number generation: Distinguishing the good and the bad. In Proceedings of the Winter Simulation Conference. IEEE Press, 95–105 (2001).

[8] McCullough B. D.: A Review of TESTU01. Journal of Applied Economics, 21, 677–682 (2006).

[9] Coddington P. D.: Analysis of random number generators using Monte Carlo simulation. International Journal of Modern Physics C 3, 547–560 (1994).

[10] Coddington P. D.: Tests of random number generators using Ising model simulations. International Journal of Modern Physics 7(3), 295–303 (1996).

[11] Burke, E., S. Gustafson, and G. Kendall: Diversity in genetic programming: An analysis of measures and correlation with fitness. IEEE Trans. Evol. Comput., 8(1), 47–62 (2004).

[12] Dechaine, M. D. and M. A. Feltus: Fuel management optimization using genetic algorithms and expert knowledge. Nuclear Science and Engineering, 124(1), 188-196 (1996).

[13] Lloyd, L. D., R. L. Johnson, and S. Salhi: Strategies for increasing the efficiency of a genetic algorithm for the structural optimization of nanoalloy clusters. Journal of Computational Chemistry, 26(10), 1069-1078 (2005).

[14] Chou C. H. and J. N. Chen: Genetic algorithms: Initialization schemes and genes extraction. in Proc. 9th IEEE Int. Conf. FUZZ, 2, 965–968 (2000).

[15] Yao, X., Y. Liu, and G. M. Lin: Evolutionary programming made faster. IEEE Trans. Evol. Comput., vol. 3, pp. 82–102, July 1999.

[16] Matsumoto, M. and T. Nishimura: Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. ACM Transactions on Modeling and Computer Simulation (TOMACS), Volume 8, Issue 1, 3-30 (1998).

[17] Boost library site: http://www.boost.org/.; Version 1.38.0; February 8th, 2009; last visited on May 1, 2009.

[18] TBB site: http://www.threadingbuildingblocks.org/; Version 2.1; Submitted June 7, 2008; last visited on May 1, 2009.

[19] STLport site: http://www.stlport.org/; Version 5.2.1; Last update Dec 10, 2008; last visited on May 1, 2009.

[20] SFMT site: http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/SFMT/; dSFMT-Version 2.1; Apr 18, 2008; last visited on May 1, 2009

[21] Holland, J. H.: *Adaptation in Natural and Artificial Systems*. University of Michigan Press: Ann Arbor, MI (1975).

[22] Goldberg, D.: *Genetic algorithms in search, optimization, and machine learning*. Addison-Wesley: Reading, MA (1989)

[23] Yu, T.-L., K. Sastry, D. E. Goldberg, and M. Pelikan: Population Sizing for Entropy-based Model Building in Discrete Estimation of Distribution Algorithms, Proceedings of GECCO'07, July 7–11 (2007).

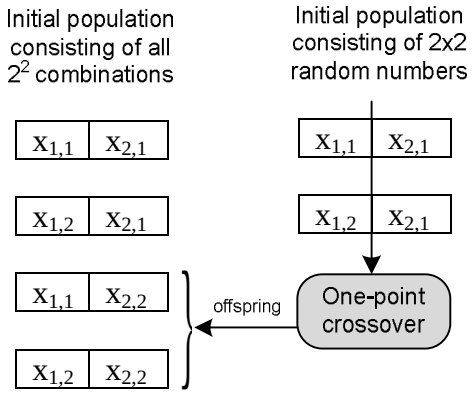


Figure 1. Example of random generation of initial population and biased initialization with a one-point crossover ($x_{i,j}$ shows the random number generated for the j^{th} class of the i^{th} decision variable (gene))

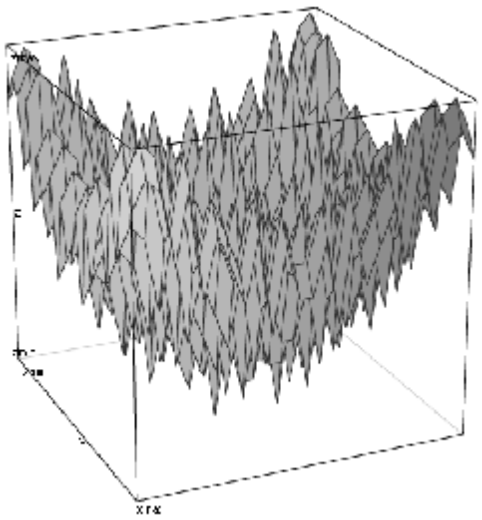
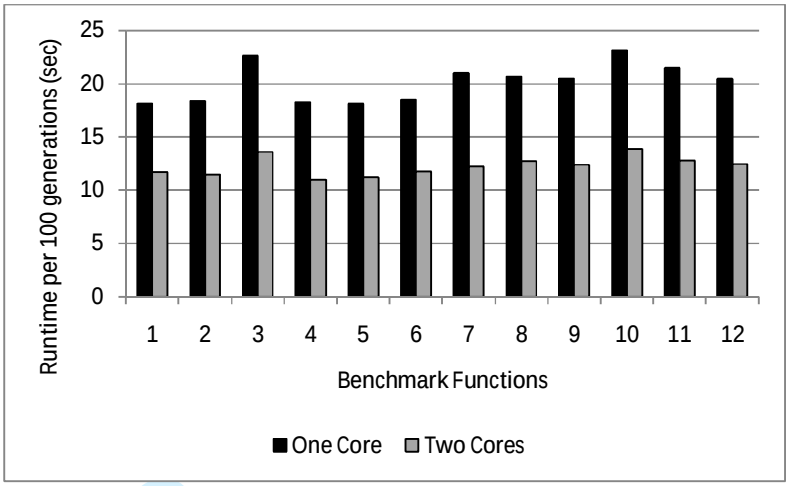
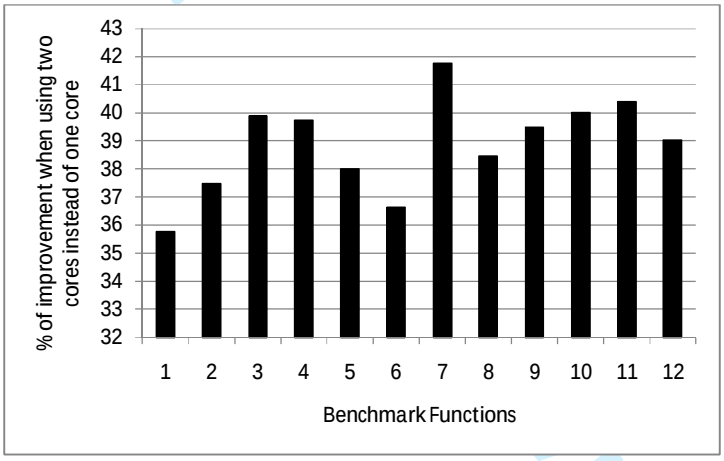


Figure 2. Graph of f_8 with a dimension of 2.



(a)



(b)

Figure 3. Execution time of the PGA model on the dual-core computer when using single- or dual-core configurations

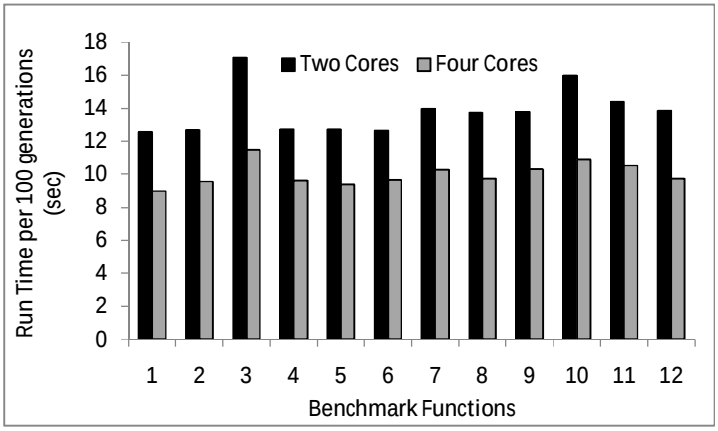
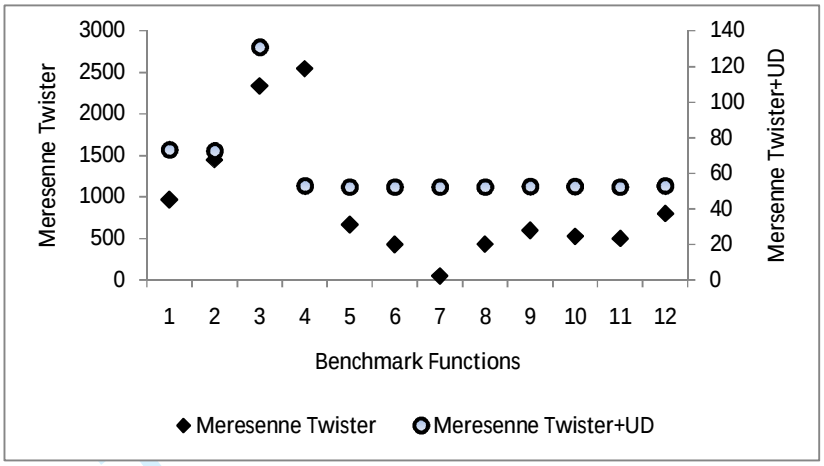
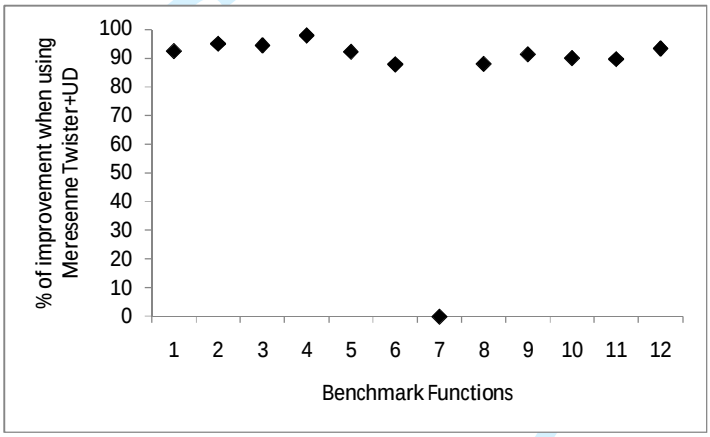


Figure 4. Execution time of the PGA model on the quad-core computer when using double- or quad- core configurations

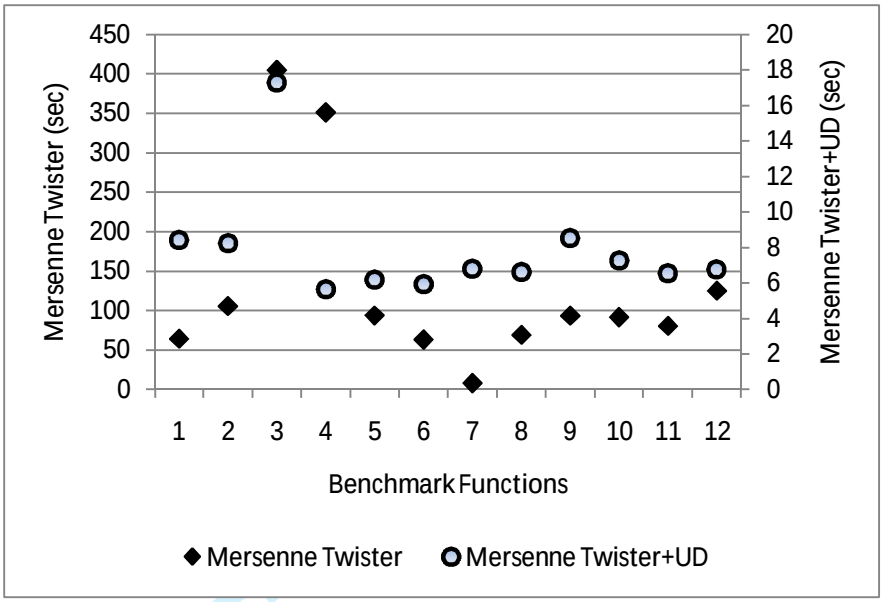


(a)

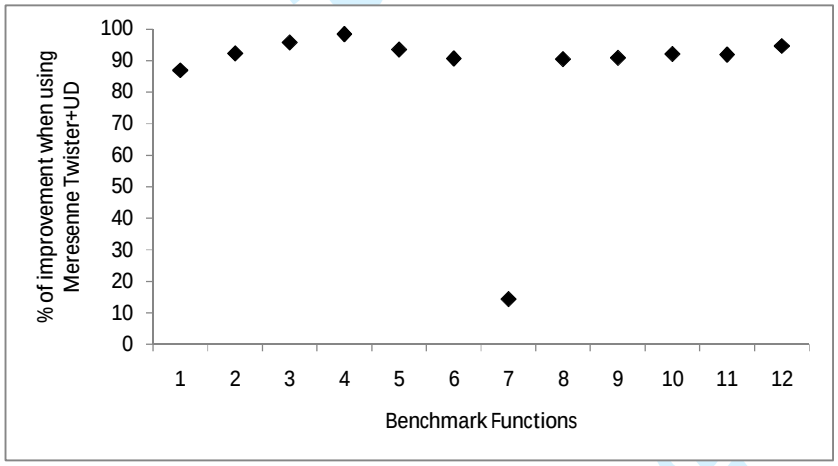


(b)

Figure 5. Comparison of the number of generations required for the PGA model to converge when using (1) Mersenne Twister RNG and (2) Mersenne Twister RNG + UD
(Note: Scaling on the vertical accesses for the two sets of results is different)



(a)



(b)

Figure 6. Comparison between execution time of the PGA model when using (1) Mersenne Twister RNG and (2) Mersenne Twister RNG + UD
(Note: scaling on the vertical accesses for the two (1) and (2) sets of results are different)

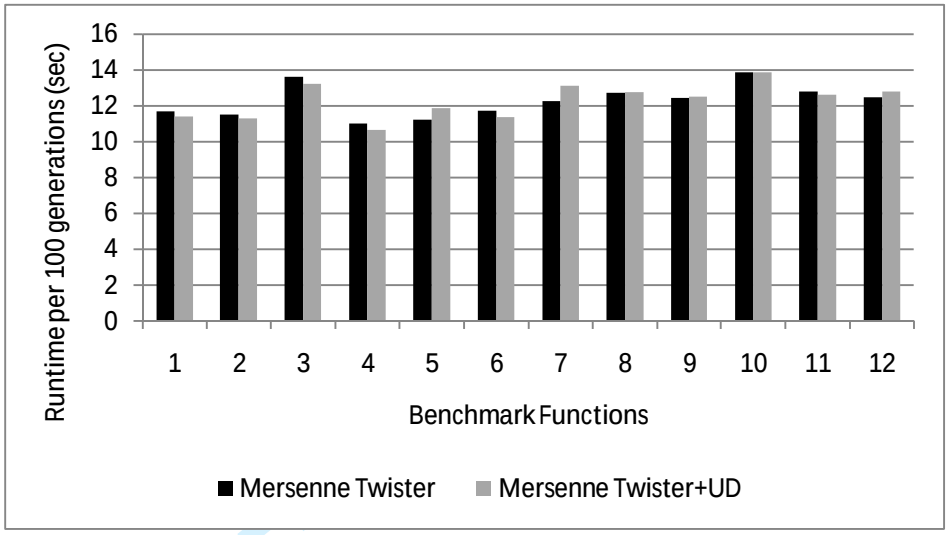


Figure 7. Comparison between execution time of the PGA model in 100 generations when using (a) Mersenne Twister RNG and (b) Mersenne Twister RNG + UD

Table 1. Benchmark functions

(N = dimensions, SD = search domain, $f_{\min}$ = minimum function value)

Test Function	N	SD	$f_{\min}$
$f_1(x)=\sum_{i=1}^{n} x_i^2$	100	$[-100,100]^n$	0
$f_2(x)=\sum_{i=1}^{n} \lvert x_i\rvert+\prod_{i=1}^{n} \lvert x_i\rvert$	100	$[-10,10]^n$	0
$f_3(x)=\sum_{i=1}^{n}\left(\sum_{j=1}^{i} x_j\right)^2$	100	$[-100,100]^n$	0
$f_4(x)=\max_i\left(\lvert x_i\rvert, 1\le i\le n\right)$	100	$[-100,100]^n$	0
$f_5(x)=\sum_{i=1}^{n-1}\left[100\left(x_{i+1}-x_i^{\,2}\right)^2+\left(x_i-1\right)^2\right]$	100	$[-30,30]^n$	0
$f_6(x)=\sum_{i=1}^{n}\left(\lfloor x_i+0.5\rfloor\right)^2$	100	$[-100,100]^n$	0
$f_7(x)=\sum_{i=1}^{n} -ix_i^4+random[0,1)$	100	$[-1.28,1.28]^n$	0
$f_8(x)=\sum_{i=1}^{n}\left[x_i^2-10\cos(2p\,x_i)+10\right]$	100	$[-5.12,5.12]^n$	0
$f_9(x)=-20\exp\left(-0.2\sqrt{1/30\sum_{i=1}^{n} x_i^2}\right)-\exp\left(1/30\sum_{i=1}^{n}\cos 2p\,x_i\right)$	100	$[-32,32]^n$	0
$f_{10}(x)=\frac{1}{4000}\sum_{i=1}^{n}\left(x_i^2\right)-\prod_{i=1}^{n}\cos\left(\frac{x_i}{\sqrt{i}}\right)+1$	100	$[-600,600]^n$	0
$f_{11}(x)=\frac{p}{n}\left\{10\sin^2(p\,y_1)+\sum_{i=1}^{n-1}(y_i-1)^2\left[1+10\sin^2(p\,y_{i+1})\right](y_n-1)^2\right\}$ $y_i=1+\frac{1}{4}(x_i+1)$	100	$[-10,10]^n$	0
$f_{12}(x)=0.1\left\{10\sin^2(3p\,x_1)+\sum_{i=1}^{n-1}(x_i-1)^2\left[1+3\sin^2(3p\,x_{i+1})\right]\right.$ $\left.+(x_n-1)^2\left[1+\sin^2(2p\,x_n)\right]\right\}$	100	$[-5,5]^n$	0

Table 2. Comparison between execution time and number of generations required for the PGA model to converge with and without using uniform distributor (UD)

Function Number	No. of Generations		Runtime		% of reduction in no. of generations	% of reduction in runtime
	Without UD	With UD	Without UD	With UD		
1	969.2	73.3	64.3	8.4	92.4	86.9
2	1448.4	72.5	105.5	8.3	95.0	92.2
3	2339.2	131.0	405.1	17.3	94.4	95.7
4	2546.0	53.0	351.5	5.6	97.9	98.4
5	669.4	52.3	93.8	6.2	92.2	93.4
6	430.2	52.3	63.1	5.9	87.9	90.6
7	52.0	52.0	8.0	6.8	0.0	14.4
8	434.2	52.0	69.0	6.6	88.0	90.4
9	600.0	52.3	93.4	8.6	91.3	90.8
10	528.4	52.5	91.6	7.3	90.1	92.1
11	502.0	52.0	80.5	6.6	89.6	91.9
12	802.6	53.0	125.1	6.8	93.4	94.6