An Automatic Code Generator for Parallel Evolutionary Algorithms: Achieving Speedup and Reducing the Programming Efforts

Omar A. C. Cortes
and Eveline de Jesus V. Sá

Instituto Federal do Maranhão
Informatics Department
São Luis, MA, Brazil
Email: {omar,eveline}@ifma.edu.br

Jackson A. da Silva

Computer Engineering Department
Universidade Estadual do Maranhão
São Luis, MA, Brazil
Email: jackson.amarals@gmail.com

Andrew Rau-Chaplin

Dalhousie University
Faculty of Computer Science
Halifax, NS, Canada
Email: arc@cs.dal.ca

***Abstract*—Building parallel applications is not a trivial task, especially when these applications involve different kinds of evolutionary computation because two different fields have to be mastered. In order to overcome this problem, we propose an automatic code generator that automatically creates Java code for parallel evolutionary algorithms considering four models of parallelism: master-slave, island, cellular, and hierarchical. Furthermore, two evolutionary algorithms can be created: genetic algorithms and evolution strategies. A speedup experiment on a parallel genetic algorithm showed that good performance can be achieved using our generator. Moreover, we applied COCOMO and COCOMO II models in order to demonstrate that the cost for programming this kind of application can be considerably reduced in terms of effort, time and people when the generator is used. According to COCOMO II model, the generator may save about 6 months using 2 people.**

***Keywords*–Parallel Computing; Evolutionary Computation; Programming Effort; Code Generator.**

I. INTRODUCTION

The popularity of multicore computers has increased the importance of building parallel applications. In fact, nowadays even cell phones take the benefits from parallel computing using multi-core architectures. Despite so, the parallel computing creates new challenges to programmers such as synchronization and the proper exploration of parallel algorithms. In addition, the lack of experience in developing parallel applications might impact directly in the software productivity, increasing significantly the effort on programming this kind of software.

One of many fields that can obtain advantages from parallel computing is the evolutionary computation. Doing so, we can combine the high performance environment provided by parallel architectures with the ability of solving complex problems using Evolutionary Algorithms (EA). Actually, making EAs faster by using parallel implementations has been one of the most promising choices in the area [1].

Evolutionary algorithms are search algorithms based on natural evolution [2]. They can be parallelized by using different models resulting in Parallel Evolutionary Algorithms (PEA). Even though other classifications might exist, they have been separated in four main categories: master-slave, island, cellular and hierarchical. In master-slave, there is a single population and the evaluation of fitness is distributed through the slaves. An island EA consists of two or more independent populations with occasional migration of individuals between sub-populations. In the cellular model, a predefined structure is held, such as a grid in a 2D case, then genetic operators are limited to a small neighborhood. Finally, the hierarchical model, also known as hybrid model, is a mix between island and either master-slave or cellular.

As we can see, each model introduces new complexities in developing a PEA, consequently increasing the effort of programming a parallel system based on EA. In this context, we built a tool called Java Parallel Evolutionary Algorithm Generator (JPEAG) which can generate both Parallel Genetic Algorithms (PGAs) and Parallel Evolution Strategies (PESs) in the four parallel models aforementioned. Moreover, from the user perspective, we show how much effort this tool can save. In order to do so, we applied two popular algorithm approaches named Constructive Cost Model (COCOMO) and COCOMO II [3], calculating metrics such as effort (people/month), time (months) and people.

To best of our knowledge, there are no other works which generate code for PAEs in a complete automatic way. There are some efforts toward creating parallel code such as the following authors: Passini [4], Hawick [5] and Rodrigues [6]; however, these works are based on general aspects of parallelization (synchronization and communication) where the programmer has to implement both his application and the PEA manually. Further, they do not show metrics for quantifying how much effort is saved. Also, our proposal is different from a framework because JPEAG can provide the programmers a entire runnable code. In terms of metric for measuring programming effort, COCOMO and COCOMO II have been used in many works such as [7], [8] and [9], for example. Furthermore, COCOMO II has been the bottom line for many researches whose aim is to improve the quality of the effort measurements such as [10][11][12].

For this sake, our paper is divided as follows: Section II introduces basic concepts on evolutionary algorithms, how JPEAG was built, how templates are used to produce the code, and how design patterns are presented in the generated code; Section III shows how COCOMO and COCOMO II compute and assess effort, time and people; Section IV presents the results considering the development of all PEAs; finally, Section V draws the conclusions of this work.

II. The Code Generator

A. Evolutionary Algorithms

Basically, an evolutionary algorithm processes a population of individuals or solutions in a particular search space. All movements along the search space are done through genetic operators on either many iterations or until reaching some other stop criteria such as: there is no more evolution within the population; the algorithm reaches the known optimal; or, the solution is sufficiently close to the optimum considering an small error (commonly the error is 1×10^{-6}). Figure 1 shows a basic structure of an EA.

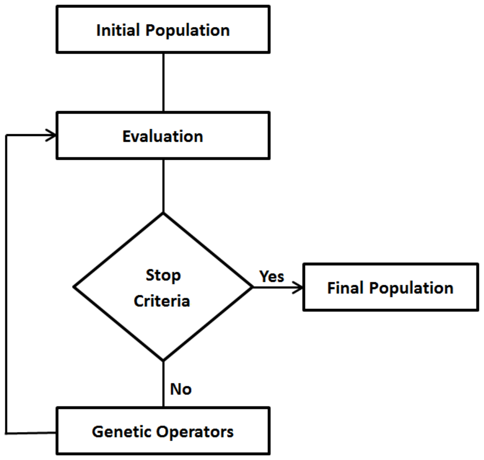


Figure 1. Basic structure of a EA

In the first step, the population is initialized at random normally using a uniform distribution. Then the population is evaluated in order to determine how each individual fits to the problem. The better the fitness, the stronger the individual within the population, thus might be higher the probability of an individual to be selected to undergo a genetic operators and, consequently, to go to the next iteration (generation). In fact, who goes to the next iteration depends on which kind of EA is being run. When the stop criteria is reached the final population containing the best solution is presented.

Typically, the genetic operators are selection, crossover and mutation. Selection is the process of choosing individuals to undergo a genetic operators or to go to next generation. Parents exchange information (genes) between themselves in order to create one or more offspring, in the crossover operator. Ideally, when two strong individuals exchange genes, in theory, the offspring is stronger than its parents [13], thus, strong individuals tend to spread its genes to next generations. On the other hand, this behavior can lead to a premature convergence of the solution because the population can end up trapped into a local optima. The mutation operator has the purpose of avoiding the premature convergence applying some modifications to one or more genes. In other words, the process of using genetic operators tends to improve the solution's quality as new generations carry on [14]. Taking those operators (crossover and mutation) into account, we can notice that several EAs share similar features. For instance, genetic algorithms and evolutionary strategies may use all those genetic operations. However, the sequence that these operators are used is different. Evolutionary strategy applies firstly the genetic operators then it uses the selection operator, whereas genetic algorithms select the individuals and afterward perform genetic operators. In addition, the individual representation can be different between EAs. For instance, evolutionary strategies need two vectors for representing an individual instead of only one of genetic algorithms. Details about these EAs can be seen in Herrara [13], Michalewicz [14] and Cortes [15].

B. Parallel Evolutionary Algorithms (PEA)

The main idea behind the parallel computing is to divide a problem into smaller pieces and solve them using different processing units. In this particular case, EAs are good candidates to be parallelized because they have an intrinsic parallelism [16], which can be explored in many different ways such as: to find out distinct solutions for the same problem, to explore several points in the search space at the same time, to distribute the evaluation function between two or more processors/threads, and to reduce the computation time to get a solution [26].

Regardless what it is being parallelized into an EAs, as previously mentioned, there are four basic models to explore the parallelism of EAs: master-slave, cellular, island and hierarchical. Different names might be used for these models; however, their characteristics remain the same.

Concerning the master-slave model, a PEA maintains a population in a master processor that delegates the function evaluation or the applications of genetic operators to slaves. Commonly, the parallelization is done distributing only the evaluation function among the available slaves. In the cellular model, all processors work on the same population, where each individual is placed into a grid, thereby genetic operators can be done only with their neighborhood in the grid. Independent populations are processed at the same time in the island model, introducing the concept of migration, where one or more individuals can be frequently exchanged between populations. This model also introduces new parameters such as the number of individuals being exchanged, how frequent the migration has to be done and the island topology which represents how islands can communicate each other. Researches such as Cantú-Paz [1] and Sakuray [18] indicate that the proper migration process contributes in the population diversity and enhance the quality of solutions. The cellular multi-individual is a hierarchical approach where each cell can contain two or more individuals, therefore being a combination between both cellular and island model.

In any parallel model, all communication between processor can be either synchronous or asynchronous. In the first one, if a processor wants to communicate with one or more processors, it has to wait until all of them be ready. On the other hand, in the asynchronous communication, the execution and the communication do not depend on the other processors, *i.e.*, if a processor wants to communicate with another one it sends the information and continues with its own execution. In our implementation of JPEAG, all communications are synchronous.

C. The Code Generator

Code generation can be defined as the technique in which we write programs that create another programs. According to

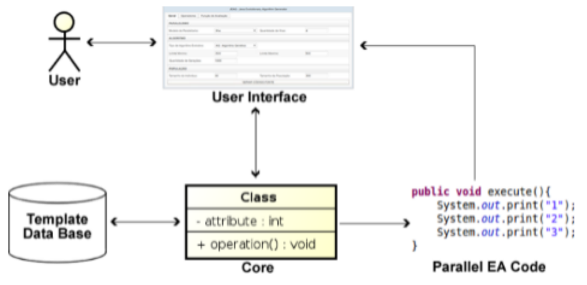
Herrington [19], creating code presents many benefits such as:

- Agile software development code generators go toward completion faster than a hand-made code, reducing the cost of development as well.
- Consistency code generators might maintain the standard in both design patterns and code conventions, avoiding breaches introduced by programmers.
- One point for gathering knowledge a change made in a definition file can be propagated to all files created previously, whereas programmers have to do so file-by-file in a hand-made process.

There are some strategies in order to implement a code generator. We are focused on templates which represent a predefined piece of software, *i.e.*, an unfinished code that may be completed using variables [20]. In other words, a software replaces some elements presented in a template file [21], where the substitution has to be performed by a template processor considering a set of inputs.

The approach based on templates was chosen because a significant amount of code for GAs and ESs are similar. Moreover, we also noticed some similar characteristics in parallel models, especially in the island and master-slave models. Thus, when these similarities were identified we could write the templates containing the common parts. In this context, the template approach allows to modify any part of the generated code changing only the proper template, therefore, becoming easier the software maintainability.

Figure 2 shows how components communicate each other in JPEAG. Its operation is described as follows: the interface is a web-based interface where users can configure all features of the PEAs, including the evaluation function (the programmer has to provide the evaluation function in terms of Java code); when all parameters are set, the user sends it to the core, which is responsible for selecting the proper template in the templates data base and use it to create the parallel EA code. The use of templates permit to create PEAs in different languages without the necessity of changing the application code; then, the parallel code is packed into a zip file and sent it back to the user via download because is a web-based application.



```
public void execute(){
    System.out.print("1");
    System.out.print("2");
    System.out.print("3");
}
```

Figure 2. Code Generator Architecture

It is important to mention that the templates are processed by a framework called Apache Velocity [22], which implements an engine for template processing and defines a Velocity Template Language (VTL) for creating it. The main advantage of the Apache Velocity is to provide methods for processing templates and creating code for any textual language. Taking this into account, our templates are Java code mixed with VLT, where VLT instructions indicate where the JPEAG has to fill up the code according to the parameters previously defined in the graphic interface.

Figure 3 shows an example of a VLT code, where directives start with the character "#" and are executed when the template is processed.

```
#if ($parallelismModel == "Island")
     ${EAModel}${parallelismModel} ea = new
     ${EAModel}${parallelismModel}($islandNumber,
     $migrationRate, $migrationFrequency);
#end
```

Figure 3. Example of VLT code

The *#if* directive in the aforementioned example means that this part of code will be processed only if the user chose the island model. Variables begin with the character "$" and will be filled up according to the configuration done by the user in the graphic interface. As a result, the user will receive a pure Java code such as "$GAIslandea = newGAIsland(4, 5, 10);$", where the parameters in this particular example are the number of islands (4), the migration rate (5) and the migration frequency (10).

D. Design Patterns on PEA

The code received by the user via download as explained in the previous section, is built using design patterns that play an important role in the reuse of software because it tends to impact in programmer productivity, specially due to the similarity between operators in evolutionary algorithms. In our case, the code produced by the generator contains mainly two design patterns: strategy and observer.

The strategy pattern defines a group of algorithms encapsulating them by means of interfaces, allowing variations regardless it uses in the clients. This pattern is used, for instance, to hide the implementation of genetic operators and to assure that any changes in the operators do not affect other parts of the code [23]. Further, this pattern permits that genetic operators may be used in other kinds of EAs, for instance, in creating parallel hybrid algorithms. The stop criteria was implemented by using the strategy pattern in this work, as well.

The observer pattern defines a $1 - to - n$ relationship between objects. This relationship consists of one object being observed by many other ones with low coupling. When an observed object has its status modified all observers receive a notification being automatically updated [23]. This pattern is used for synchronization purposes between objects. Being specific, a process receives information about other processes, thus it can decide whether it have to wait for other processes, in case of migration for example, or carry on with its own execution.

Figure 4 presents a model with an example of the master-slave model for a parallel genetic algorithm, where we can see clearly how the observer interface is implemented by the *ParallelismMonitor* class, consequently controlling the communication between islands, while the strategy pattern interfaces are used to control the genetic operators and the stop criteria (classes which implement the pattern are omitted due to the lack of space).

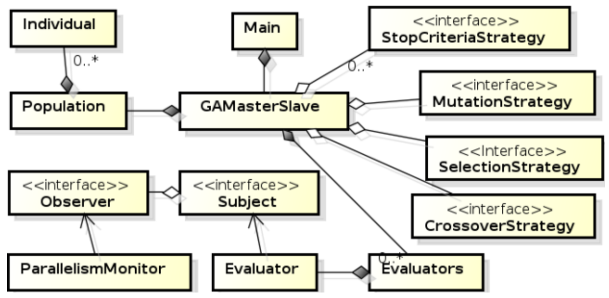


Figure 4. Master-Slave Patterns

III. Computing Programming Effort

A. COCOMO

COCOMO is a regression-based model used to estimate the programming effort in a software development project. The model can be divided into three sub-models: basic, intermediary and advanced. The basic one is presented in (1), where PM means Person/month (effort), A is a calibration factor, $KLOC$ represents the number of lines of code (in terms of 1000 lines or K) and B is a scale factor.

$$PM = A \times (KLOC)^B \tag{1}$$

$$time = C \times PM^D \tag{2}$$

$$p = \frac{PM}{time} \tag{3}$$

Also, the basic model can compute the required time for developing the software in months (2) and the number of required people (3) as well, where C and D are constants. The constants A, B, C, and D, which are originally from the model, can be seen in Table I, where $Project$ indicates the following features: (i) Organic is a project involving a small team with good experience and less than rigid requirements; (ii) Semi-detached is a project with a medium team with mixed experiences and a combination of rigid and less than rigid requirements; (iii) Embedded involves a set of tight constraints, being a combination of the organic and semi-detached project.

TABLE I. CONSTANTS FOR COCOMO - BASIC

Project	A	B	C	D
Organic	2.4	1.05	2.5	0.38
Semi-detached	3.0	1.12	2.5	0.35
Embedded	3.6	1.20	2.5	0.32

The intermediary model is similar to the basic one; however, it considers a multiplier effort (ME) as we can see in (4), where $n \in [1, 15]$ according to 15 different multipliers. In fact, ME is the product of all efforts that might be involved in the project.

$$PM = A \times (KLOC)^B \times \prod_{i=1}^{n}(ME_i) \tag{4}$$

Basically, what we do with the ME is multiply all of it according to the levels we have in the team. Which one we use depends on the requirement we have in the project. For example, a project might demand a high level of programming capability and a very high level of analyst capability. So, in this particular case the ME is computed as follows: $ME = 0.86 \times 0.71 = 0.6106$. All values in ME are predefined and can be seen in [24] and [25]. On the other hand, the constants A and B are different for the intermediary model as shown in Table II, while C and D remain the same from Table I.

TABLE II. CONSTANTS FOR COCOMO - INTERMEDIARY

Project	A	B
Organic	3.2	1.05
Semi-detached	3.0	1.12
Embedded	2.8	1.50

The advanced model is similar to the intermediary, however the calculation is done on each step of development, being adequate only for big projects.

B. COCOMO II

The main differences between COCOMO and COCOMO II are: (i) the first one uses 15 multiply efforts, whereas the last one uses 17 [27]; and, (ii) B can be computed based on a Scale Factor (SF) as presented in (5), where β is a constant equals to 0.91 as suggested in [24] and i varies from 1 to 5 according to predefined SFs. All these values can be obtained from COCOMO II manual and in [24].

$$B = \beta + 0.01 \times \sum_{i=1}^{5} SF_i \tag{5}$$

IV. Results

A. Speedup

In order to demonstrate that the software is usable and performs in parallel, we present an experiment based on the Griewank function [28], which is shown in (6), where x_i belongs to the range [-600,600] and n is equal 30, representing an individual with 30 real-coded genes.

$$f(x) = \sum_{i=1}^{n} \frac{x_i^2}{4000} - \prod_{i=1}^{n} cos(\frac{x_i}{\sqrt{i}}) \tag{6}$$

The experiment was conducted in an Intel i5 2.3 Ghz, 4GB of RAM with two physical cores and hyper threading (4 logical cores), using Ubuntu Linux, JDK 1.7.04. A Parallel Genetic Algorithm was built using the following configuration: selection operator is tournament; tournament size equals 10; heuristic crossover operator; probability of crossover equals to 0.8; uniform mutation; probability of mutation equals to 0.01; population size sets to 360; stop criteria equals to 2000 iterations. For the island model parameters are: migration rate sets to 5; migration frequency equals to 200 iterations; and topology is ring. We did not generate a cellular PEA because we did not have the proper architecture for its execution. The speedup is computed by $S_p = \frac{T_s}{T_p}$ where T_s represents the time for running the code in 1 thread and T_p is the required time for executing in p threads. This metric is known as weak speedup and was proposed by Alba [26] because the code is exactly the same regardless the number of threads.

Table III presents the speedup and the efficiency achieved by the Parallel Genetic Algorithm where we can observe that the island model got the best speedup being close to the ideal one. In terms of efficiency, the best one is reached using 2 threads as expected because the overhead caused by the parallel synchronization is smaller.

TABLE III. SPEEDUP AND EFFICIENCY ACHIEVED BY A GA

Master-Slave			
Threads	Time(ms)	Speedup	Efficiency
1	3071.774	-	-
2	2318.774	1.325	66.237
3	2158.129	1.423	47.445
4	2152.774	1.427	35.6728
Island			
Threads	Time(ms)	Speedup	Efficiency
1	2939.935	-	-
2	1516.29	1.939	96.945
3	1069.871	2.748	91.598
4	866.484	3.393	84.824
Hierarchical			
Threads	Time(ms)	Speedup	Efficiency
1	2940.42	-	-
2	1554.87	1.891	94.555
3	1101.58	2.67	88.975
4	942.29	3.12	78.01

B. COCOMO

The first result regards COCOMO basic considering the development of all possible outputs, *i.e.*, all parallel models and evolutionary algorithms summing up 5507 lines, can be seen in Table IV.

TABLE IV. NUMBER OF LINES OF CODE (LOC) PER MODEL AND ALGORITHM

Parallel Model/Algorithm	GA	EE
Master-Slave	602	589
Island	859	848
Cell	635	641
Cell Multi-Individual	675	658
Sub-Total	2771	2736
Total	**5507**	

Taking into account that the project is organic and using (1), (2) and (3), we can estimate the following results: $MP = 14.37$ (effort), $time = 6.9$ months, $p = 2.1$. In other words, developing all available models and algorithms would require 15 people/month of effort, 7 months of time and 3 people according to COCOMO basic model.

In the intermediary model, the following multiplication effort are considered as essential for developing all parallel models and evolutionary algorithms: RELY, CPLX, ACAP, AEXP, PCAP, LEXP, MODP and SCED [25]. Then, using the predefined values (4), and also taking into account the non-defined values as nominal inputs, we can calculate the values presented in Table V for different levels. For instance, considering parameter as very low level, a project would take 46.8 people/month, 11 months and 5 people to complete, whereas considering the level as very high those values are reduced to 13 people/month, 7 months and 2 people.

C. COCOMO II

In COCOMO II, we considered the following efforts: RELY, CPLX, RUSE, PVOL, ACAP, PCAP, AEXP, PLEX,

TABLE V. COMPUTING THE EFFORT FOR THE COCOMO INTERMEDIARY MODEL

	Very Low	Low	Nominal	High	Very High
Total ME	2.4	1.5	1.0	0.8	0.7
PM	46.8	28.7	19.2	15.3	12.8
Time	10.8	8.9	7.7	7.1	6.6
p	4.3	3.2	2.5	2.2	1.9

LTEX, TOOL and SCED [24]. Further, we also considered all scale factors for computing B values using (5) as illustrated by Table VI.

TABLE VI. COMPUTING B AND ME FOR THE COCOMO II INTERMEDIARY MODEL

	Very Low	Low	Nominal	High	Very High	Extra High
B	1.23	1.16	1.10	1.04	0.97	0.91
ME	2.7	1.5	1.0	0.8	0.6	0.9

Taking into consideration that now we have six different Bs, we can calculate the table of effort for each one as presented in Table VII, where we can observe that as we increase both the level of scale factors and the multiplier factors, the effort tends to be lower, which is an expected behavior. It is important to notice that the extra high level is not completely filled up which causes an increase in the effort parameters, which is not desirable.

TABLE VII. COMPUTING EFFORT FOR COCOMO II INTERMEDIARY MODEL PER B

	Very Low	Low	Nominal	High	Very High	Extra High
ME	2.7	1.5	1.0	0.8	0.6	0.9
B = 1.23						
PM	71.02	38.05	25.88	20.11	15.90	22.26
Time	12.63	9.96	8.61	7.82	7.15	8.13
p	5.62	3.82	3.01	2.57	2.22	2.74
B = 1.16						
PM	63.74	34.15	23.23	18.05	14.27	19.98
Time	12.12	9.56	8.26	7.51	6.87	7.80
p	5.26	3.57	2.81	2.40	2.08	2.56
B = 1.10						
PM	57.24	30.67	20.86	16.21	12.82	17.95
Time	11.64	9.18	7.93	7.20	6.59	7.49
p	4.92	3.34	2.63	2.25	1.94	2.40
B = 1.04						
PM	51.39	27.54	18.73	14.55	11.51	16.11
Time	11.17	8.81	7.61	6.92	6.33	7.19
p	4.60	3.12	2.46	2.10	1.82	2.24
B = 0.97						
PM	46.14	24.72	16.81	13.06	10.33	14.46
Time	10.72	8.46	7.31	6.64	6.07	6.90
p	4.30	2.92	2.30	1.97	1.70	2.10
B = 0.91						
PM	41.43	22.19	15.10	11.73	9.28	12.99
Time	10.29	8.12	7.01	6.37	5.83	6.62
p	4.02	2.73	2.15	1.84	1.59	1.96

D. Discussion on Effort

As previously stated, we consider the software as an organic one, since it does not have a huge amount of lines. Thus, the result for COCOMO basic may be considered as an interesting estimation of effort, time and people if the team has some previous experience in the subjects. These results are similar to those one given by COCOMO II using a high level of multiplier efforts which would represents the necessity of learning the aforementioned subjects.

In terms of COCOMO II, we can observe that as we increase the level of the team we also decrease the effort parameters (people/month, months and pearson), as expected. Also, it gives a more precise measure due to the use of both multiplier efforts and scale factors, excepting when the extra high level is used in Multiplier Efforts (ME) because this particular level is not fully-filled, producing a little increase in the parameters. On the other hand, we can notice that parameters are more susceptible to the scale factors and multiplier effort making the measures more accurate in the all levels.

Doing an analysis of the person who developed the system, we can consider the following multiplier for COCOMO: RELY = high, CPLX =high, ACAP =high, AEXP =nominal, PCAP = high, LEXP = high, MODP = =high and SCED =nominal; and the following one for COCOMO II: RELY = high, CPLX = high, RUSE = high, PVOL = low, ACAP = high, PCAP = high, APEX = high, PLEX = high, LTEX = high, TOOL = high and SCED = Nominal. Either, we considered all scale factor as being in the maximum level.

TABLE VIII. ANALYSIS OF THE PERSON WHO DEVELOP THE PARALLEL EVOLUTIONARY ALGORITHMS

	COCOMO Basic	COCOMO Intermediary	COCOMO II (B = 0.91)
ME	-	0.76	0.67
PM	14.4	14.6	10.08
time	6.88	6.93	6.02
p	2.09	2.11	1.68

Thus, the results are shown in Table VIII, where COCOMO II presented a more precise estimation, considering that all knowledge was acquired during the disciplines in the master degree.

V. CONCLUSION

This paper presented a software whose main purpose is to reduce the programming effort when programming parallel evolutionary algorithms. A speedup test showed that it is possible to achieve a good speedup using the parallel models. Moreover, the models COCOMO and COCOMO II were used in order to support our hypotheses of saving effort. All in all, the JPEAG can save around one year of the time using 4 people if a low level of knowledge is held, or about 6 months and 2 people if the programmers has a high level of previous knowledge.

REFERENCES

[1] E. Cantú-Paz, “A Survey of Parallel Genetic Algorithms”, Department of ComputerScience and Illinois Genetic Algorithms Laboratory, Universi-tyof Illinois at Urbana-Champaign, 1998.

[2] A. E. Eiben and Smith, J. E., “Introduction to Evolutionary Computing”, Berlin: Springer Verlag, 2003.

[3] B. Boehm, B. Clark, E. Horowitz, and C. Westland, “Cost Models for Future SoftwareLife Cycle Processes: COCOMO 2.0”, Annals of Software Engineering, 1, 1995, pp. 57–94.

[4] F. Pasini and L. Dotti, “Code Generation for Parallel Applications Modelled with Object-Based Graph Grammars”, Electronic Notes in Theoretical Computer Science, v. 184, 2007, pp. 113–131.

[5] K. A. Hawick and D. P. Playne, “Automated and parallel code generation for finite-differencing stencils with arbitrary data types”. Procedia Computer Science, v. 1, n. 1, 2010, pp. 1795–1803.

[6] A. Rodrigues, F. Guyomarch, J-L. Dekeysera, and Y. Menach, “ Automatic Multi-GPU Code Generation applied to Simulation of Electrical Machines”, IEEE Transactions on Magnetics, v. 48, n. 2, 2012, pp. 831–834.

[7] W. Jiamthubthugsin and D. Sutivong, “Portfolio management of software development projects using COCOMO II”. In Proceedings of the 28th international conference on Software engineering (ICSE). ACM, New York, NY, USA, 2006, pp. 889–892.

[8] T. N. Sharma. “Analysis of Software Cost Estimation using COCOMO II”, International Journal of Scientific & Engineering Research, v. 2, Issue 6, 2011.

[9] L. L. Minku and X. Yao, “How to make best use of cross-company data in software effort estimation?”. In Proceedings of the 36th International Conference on Software Engineering (ICSE). ACM, New York, NY, USA, 2014, pp. 446–456.

[10] L. V. Patiland, R. M. Waghmode, S. D. Joshi, and V. Khanna, “Generic model of software cost estimation: A hybrid approach”, IEEE International on Advance Computing Conference (IACC), 2014, pp. 1379–1384.

[11] Z. Dan, “Improving the accuracy in software effort estimation: Using artificial neural network model based on particle swarm optimization”, IEEE International Conference on Service Operations and Logistics, and Informatics (SOLI), 2013, pp. 180–185.

[12] L. V. Patil, N. M. Shivale, S. D. Joshi, and V. Khanna, “Improving the accuracy of CBSD effort estimation using fuzzy logic”, International on Advance Computing Conference (IACC), 2014, pp. 1385–1391.

[13] F. Herrera, M. Lozano, and J. L. Verdagay, “Tack-ling Real-Coded Genetic Algorithms: Operators and Tools for Behavioural Analisys”, Artificial Intelligence Review, 4(12), 1998, pp. 265-319.

[14] Z. Michalewicz, “Genetic Algorithms + DataStructure = Evolution Programs”, Springer-Verlag, New York, 3 edition, 1999.

[15] O. A. C. Cortes, R. H. C. Santana, M. J. Santana, and O. R. S. Mendez, “Anáise de Operadores de Recombinao em Estratgias Evolutivas Aplicados no Refinamento de um Sistema Nebuloso”, In: Simpósio Brasileiro de Automática Inteligente, 2005.

[16] J. Yao, “Analysis of Scalable Parallel Evolutionary Algorithms”, IEEE Congress on Evolutionary Computation Sheraton Vancouver Wall Centre Hotel, Vancouver, BC, Canada. July, 2006.

[17] E. Alba and M. Tomassini, “Parallelism and Evolutionary Algorithms”, IEEE Transactionson Evolutionary Computation, Vol. 6, No. 5, October, 2002.

[18] M. Sakuray, “Estudo da Influência dos Parâmetros de Algoritmos Paralelos da Computao Evolutiva no seu Desempenho em Plataformas Multinúcleos”, PhD Thesis, Universidade Federal de Uberlândia, 2014.

[19] J. Herrington, “Code generation in action”. Greenwich: Manning Publications, 2003.

[20] D. Lucrédio, “Uma Abordagem Orientada a Modelos para Reutilizao”, Phd Thesis USP, So Carlos, SP, Brazil, 2009.

[21] D. Manolescu, M. Voelter, and J. Noble. Pattern Languages of Program Design 5. Reading: Addison-Wesley Professional, 2006.

[22] Apache Velocity Project Site - http://velocity.apache.org - accessed: Sep-9th-2014.

[23] H. Feng, K. Li-shaff, and C. Yu-ping, “A Generic Design Model for Evolutionary Algorithms”, Wuhan University, *Journal of Natural Sciences*, v. 8, n. 1b, 2003, pp. 224–228.

[24] B. W. Boehm, “Software cost estimation with Cocomo II”. Prentice Hall, Upper Saddle River, NJ, 2000.

[25] Y. Miyazaki and K. Mori, “COCOMO evaluation and tailoring”. In Proceedings of the 8th International Conference on Software engineering, IEEE Computer Society Press, Los Alamitos, CA, USA, 1985, pp. 292–299.

[26] E. Alba, “Parallel Evolutionary Algorithms Can Achieve Super-Linear Performance”, Information Processing Letters, v. 82, 2002, pp. 7-13.

[27] B. Steece and B. Boehm, “A constrained regression technique for cocomo calibration”, Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement, 2008, pp. 213–222.

[28] Locatelli, M., ”A Note on the Griewank Test Function“, Journal of Global Optimization, v. 25, n. 02, 2003, pp. 169-174.