Learning and Nonlinear Models - Journal of the Brazilian Society on Computational Intelligence (SBIC), Vol. 17, Iss. 1, pp. 39-47, 2019

A TOOL FOR CREATING PARALLEL SWARM ALGORITHMS AUTOMATICALLY ON MULTI-CORE COMPUTERS

João Pedro Augusto Costa
Programa de Pós-Graduação em Engenharia da Computação e Sistemas - PECS
Universidade Estadual do Maranhão - UEMA
Cidade Universitária Paulo VI, s/n, São Cristovao, São Luis, MA, Brasil
jpac1207@gmail.com

Omar Andres Carmona Cortes
Departamento de Computação - DComp
Instituto Federal de Educação, Ciência e Tecnologia do Maranhão - IFMA
Av. Getulio Vargas, 04, Monte Castelo, São Luis, MA, Brasil
omar@ifma.edu.br

Abstract – Meta-heuristics are usually bio-inspired algorithms (based on genes or social behaviors) that are used for solving optimization problems in a variety of fields and applications. The basic principle of a meta-heuristic, such as genetic algorithms, differential evolutions, particle swarm optimization, etc., is to simulate the pressure that the environment applies to individuals resulting in the survival of the best ones. Regardless of which meta-heuristic is being used, the more complex the problem, the more time consuming the algorithm. In this context, parallel computing represents an attractive way of tackling the necessity for computational power. On the other hand, parallel computing introduces new issues that the programmers have to deal with, such as synchronization and the proper exploration of parallel algorithms/models. To avoid these problems, and at the same time, to provide a fast development of parallel swarm algorithms, this work presents a tool for creating parallel code using Parallel Particle Swarm Optimization (PSO) Algorithms in Java. The generator considers three models of parallelism: master-slaves, island and hierarchical. Experiments in the created code showed that a speedup of 5.3 could be reached in the Island model with 2000 iterations using Griewank's function. Moreover, using a cost estimation model (COCOMO) we showed that our tool could save from 4.4 to 14.5 person/month on programming effort.

Keywords – Particle Swarm Optimization, Parallel Computing, Automatic Code Generation, Programming Effort.

1. INTRODUCTION

Multi-core computers have been popularized in the past years. In fact, even cell phones are equipped with multi-core processors. In one hand, this scenario has provided high computation power to regular users that now can run applications efficiently, even in portable devices. On the other hand, this new technology increases the complexity of developing software, introducing issues such as different programming models and dealing with synchronization. Synchronization regards to how multiple processes or threads can communicate with each other; and, a programming model concerns how the program will behave during its execution in multiple processor units. Taking these problems into account, we can infer that the experience in developing parallel applications might impact directly on the software performance, increasing the effort on programming this kind of software [1].

In order to try making the development of parallel programs less complicate, Jaquie [2] presents one of the few environments to produce parallel code for Parallel Virtual Machine (PVM) and Message Passing Interface (MPI) [3] at the end of 90's. Since then, some works have been done, like a visual approach for creating distributed applications in Java [4] and a parallel code generator for C-MPI [5] and [6]. Then Xiang [7] creates parallel code for Open64 [8] compilers. The main drawback of Passini's work [5] is to demand to learn the Object-Based Graph Grammars (OBGG) language, consequently, increasing the learning curve to produce parallel code. Also, regardless of the parallel approach, the evolutionary and swarm algorithms have to be implemented nonetheless. Further, some attempts bring on parallel code for Graphic Processor Units (GPU) such as [9], [10], [11], and [12] works. Furthermore, Hou's work [13] creates efficient parallel code for a vectorized application, specifically, for sorting algorithms.

Despite the evolution in creating parallel code, all these attempts deal only with general aspects of parallel programming such as synchronization and communication, *i.e.*, programmers still have to implement their swarm-based applications manually, or in the best case scenario, they use an evolutionary or swarm framework to do so. The first work on doing parallel evolutionary algorithms automatically appeared in [14], which presents an automatic code generator for Genetic Algorithms and Evolution Strategies in Java. Then an analysis of the savings regarding the programming effort was made in [15]. At the best of our knowledge, there are no other initiatives to generates parallel evolutionary or swarm algorithms automatically.

Thus, we proposed and implemented an extension of Silva's work [14] by creating parallel versions of the particle swarm optimization (PSO) algorithm, providing three main benefits: (i) rapid software development; (ii) keeping consistency, avoiding

errors introduced by programmers; and, (iii) one point for gathering knowledge by using templates, *i.e.*, changes made in a template propagate through all source files [16]. This tool is particularly interesting for industry applications, in which users need to prototype new parallel applications rapidly. Alternatively, the tool can be used for creating hybrid evolutionary/swarm algorithms adding only parts of code. This is radically different from a framework because, in frameworks, we usually do not have access to the code. As Karasek's work states: there are not many tools in optimization field that allows the researchers to implement own code, modify existing code or compare different algorithms [17]. Also, if the code is not available, it is impossible to adapt the algorithm to specific problems. Therefore, the main contributions of this work are: (i) to offer a tool that provides parallel evolutionary/swarm algorithms code, ready to compile and execute, to the users in an easily and instantly way; (ii) to provide parallel code that is portable; (iii) a tool that extends algorithms in three different parallel models, which can be combined with a small programming effort; and (iv) we present an analysis on how much programming effort is saved using COCOMO [18] model.

In this context, this work is divided as follows: Section 2 shows the PSO algorithm; Section 3 details how the generator was extended; Section 4 presents a test on a famous benchmark function, proving that the generated code is ready to compile, run, and obtain a reasonable speedup. Also, the saved programming effort is shown using the COCOMO model; Section 5 describes the conclusions and future work.

2. Particle Swarm Optimization

The PSO was firstly proposed by Kennedy and Eberhart in 1995 [19]. The algorithm consists of particles that are real-coded vectors placed in a search space. Each particle moves along the search space combining some aspects of its historical position, and the best global position found up to the current iterations. All in all, particles travel around the search space and probably the swarm moves toward the potential optimum on each iteration.

A particle is represented by a pair of vectors $x = (x_1, x_2, x_3, ..., x_i)$ and $v = (v_1, v_2, v_3, ..., v_i)$, in which x is the position of the particle in the search space, v represent the velocity of the particle, and i is the dimension of the problem. The velocity is the main component used to determine the new position of a particle using the Equations 1 and 2, in which w is the inertia weight, c_l and c_2 are acceleration constants, r_l and r_2 are random numbers in the range $[0, 1]$, p_k is the best position known by the particle in the k iteration, and g is the global optima of the swarm at the instant k.

$$v_{(k+1)} = wv_k + c_1r_1(p_k - x_k) + c_2r_2(g - x_k) \tag{1}$$

$$x_{(k+1)} = x_{(k)} + v_{(k+1)} \tag{2}$$

Eventually, Equation 2 takes a particle to outside boundaries. In such case, Clerc [20] proposes to saturate $x_{(k+1)}$ in its respective boundary and change its velocity according to $v_{(k+1)} = -0.5 \times v_{(k+1)}$, meaning that the particle goes in the opposite direction regardless which boundary constraint has been violated. The complete pseudo code is shown in Algorithm 1.

Algorithm 1 - PSO Pseudo Code

```
S ← generate_swarm(n,d)
P ← S
V ← init_velocity(n,d,V_min,V_max)
fit ← evaluate_(S)
fitP ← fit
fitg ← best(fit)
g ← locate_best_particle(S^k,fitg)
while (Stop Criterion is FALSE) do
    for i = 1 to #swarm_size do
        V_i ← wV_i + c_1 r_1(P_i − S_i) + c_2 r2(g − S_i)
        S_i ← S_i + V_i
        fit_i ← evaluate(S_i)
        if (fit_i < fitP_i) then
            swap(fitP_i,fit_i)
            swap(P_i,S_i)
        end if
        if (fit_i < fitg) then
            swap(fitg,fit_i)
            swap(g,S_i)
        end if
    end for
end while
```

According to the algorithm, initially, the PSO distributes all particles around the search space randomly, initiates the P matrix (the browsing history of each particle) and g (the best particle). Then, while the stop criterion is not reached, the process

of updating velocities and positions continues. On each iteration, particles must test if their new positions are the best they have identified (P). If so, they update their browsing history. Further, each particle must test itself against the best position g. If the new position is better than g, then the algorithm swaps them.

3. The Parallel Code Generator

3.1 Parallel Models

The main idea behind parallel computing is to divide a problem into smaller pieces and then solve them independently. A programming model is a high level of abstraction that describes a parallel computer system in terms of the semantics of the programming language [21]. Consequently, a parallel programming model specifies how the programmer will code the application. So, as we can notice, this idea is particularly interesting when we are talking about evolutionary, and swarm algorithms due to their inherent parallelism [22]; therefore, several forms of dividing the problem come in mind. Nonetheless, four models are the most popular in the area of parallel evolutionary/swarm algorithms: master-slave, island, cellular and hierarchical [23].

- **Master-slave**: A master process maintains the swarm, delegating either the evaluation function or the operators to the slaves. Usually, the implementation of this model is based on the distribution of the population, *i.e.* swarm in our case, through the slaves to run the evaluation function. Figure 1 shows how the master-slave model works.

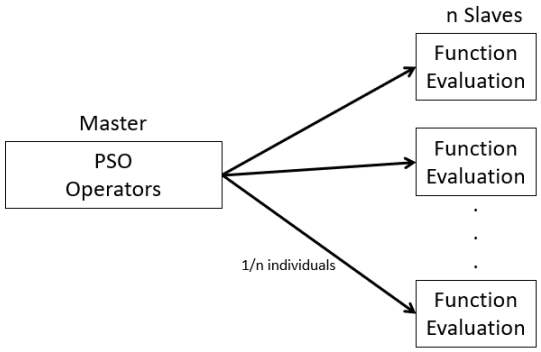
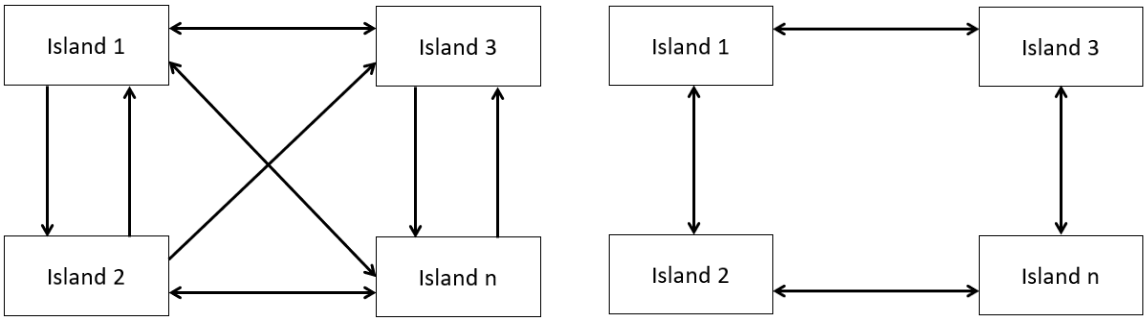


Figure 1: Parallel PSO: Master-Slave Model

- **Island**: Different swarms act independently on each processor unit, and eventually, they exchange information among themselves sending one or more particles to other swarms in a process called migration. In this model, different architectures of communications, also called topologies, can be implemented. Typically, the communication occurs in the form of a ring or all-to-all (similar to a broadcast of particles). It is important to notice that the user must inform both the frequency in which the migration happens and the number of particles being exchanged. Figure 2 illustrates the two types of the island architecture. In the first one, all islands communicate each other. In the second one, the ring architecture, only neighborhood islands can communicate with each other.



(a) All-to-All (b) Ring

Figure 2: Island Model

- **Cellular**: A grid is created in which each intersection receives a particle. Each particle can only interact with its neighborhood. This model is particularly interesting for computers whose architecture is fine-grained. Thus, if executed in a multi-core computer the cost of synchronization would lead to a poor speedup. Probably, the parallel version would be slower than the sequential one. Figure 3 presents a cellular architecture formed by nine particles.

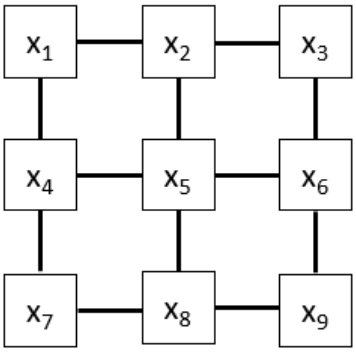


Figure 3: Cellular

- **Hierarchical**: It is a model that is similar to the cellular one; however, each intersection holds a group of particles. In shorts, this approach is a coarse-grained implementation of the cellular model allowing multi-core computers get speedup advantages of the cellular model. Figure 4 shows an example of a 2×2 hierarchical grid with 6 particles on each intersection. Thus, the particles that belong to the same intersection can communicate with each other and only with their neighborhood intersections.

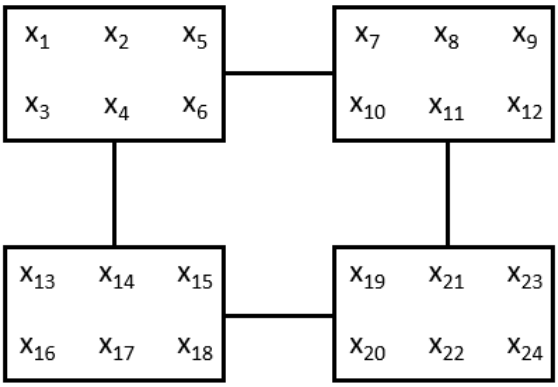


Figure 4: Hierarchical

Regardless of the model being implemented, the communication between processor units can be synchronous or asynchronous. In the first one, all communication is blocked, *i.e.*, while the communication is not completed on both sides, the execution of each processor unit does not carry on. In the asynchronous mode, the information is sent, and the processor unit continues to run. In this work, all communication is synchronous. Moreover, even though different names might be used for those models, their characteristics and implementations remain the same [15].

3.2 Implementation

The architecture of the parallel generator is shown in Figure 5. The *TemplateProcessor* class is responsible for retrieving the proper template from the template database according to the specifications defined in the user interface. In other words, the template is filled up based on user inputs and preferences (*AEConfiguration* class). Afterward, the parallel code is returned to the user through a download file.

Our templates are set to create Java code since a significant part of the code is similar in all of three meta-heuristics that can be generated (Genetic Algorithms, Evolution Strategies, and PSO). In fact, a template is a predefined piece of software, *i.e.*, it is an unfinished code that will be completed using variables [24]. Thus, the use of templates allows the programmer to easily extend the application in order to create new parallel meta-heuristics in any language. Moreover, similarities are also found in the different parallel models, enabling to implement the different parallel models, especially when they are the island and master-slave models. In this context, the template approach permits to modify any part of the generated code changing only the proper template, therefore, becoming easier the software maintainability [15].

A framework called Apache Velocity [25] processes the templates using a language named *Velocity Template Language* (VTL). The main advantage of using *Velocity* is to process templates for any textual language. Hence, it is possible to extend the parallel algorithms to any other language as previously mentioned. Figure 6 shows an instance of a VTL code, in which directives that start with the character "#" are processed by the framework. Further, variables begin with the character "$" will be filled out according to the settings done in the user interface previously. So, in this particular example, we are creating a PSO using the master-slave model. These configurations are done by using variables such as "$Parallelism" and "$EA".

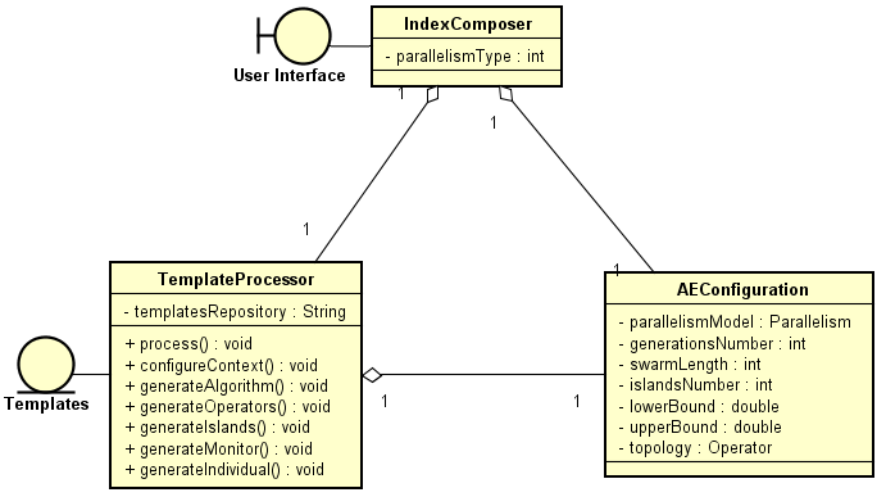


Figure 5: Generator Architecture

```
#if($Parallelism == "Island")
    public Individual execute() {
#end
    #if($EA =="PSO")
     double pBest = 0, particleBest[];
    #end
    #if($Parallelism == "MasterSlave")
        public void execute() {
    #end
        population.initPopulation(random);
        #if($Parallelism != "PSO")
            avaliator.evaluatePopulation(population);
        #end
```

Figure 6: Velocity Template

The code is devised by a method $execute()$ that is responsible for running the parallel PSO, and the method $evaluatePopulation()$ that belongs to an object $evaluator$ that distributes the population between the slaves. The remainder of the code is the implementation of the pseudo-code presented in Algorithm 1. The complete code can be seen in the link *https://github.com/jpac1207/Parallel-PSO*.

4. Results

In this section, we aim to show that the application creates code ready to compile and run, *i.e.*, we are not comparing which model presents the best speedup because it also depends on the granularity of the evaluation function. Also, because the obtained code aims to execute in multi-core computers, the cellular model was not presented due to its fine granularity and poor performance. All experiments ran on an 8-core Intel Xeon 2.53 GHz, 3.86 GB of RAM, and Linux 2.6.32-431.11.2.el6.x86_64.

4.1 Speedup

To show the generator performance, we present an experiment using the Griewank function [26], which is presented in Equation 3, where x is within in range $[-600; 600]$ and $n = 30$, representing the dimension, *i.e.*, the size of each individual.

$$f(x) = \sum_{i=1}^{n} \frac{x_i^2}{4000} - \prod_{i=1}^{n} cos(\frac{x_i}{\sqrt{i}}) \tag{3}$$

All the meta-heuristics are configured to use a swarm size of 360 particles, and the stop criteria are 1000 and 2000 generations. In the island topology, we used the all-to-all topology, the migration rate was set to 5 individuals per migration, and a frequency of 5 iterations. In the hierarchical and island models, each node/island has $360/n$ individuals, in which n is the number of threads. The accelerations constants are $c_1 = c_2 = 2.0$ and $w = 0.9$. Each model was executed 50 times; therefore, the time is the average between executions. The standard deviation is small in all tests. Consequently, we are not presenting it.

The speedup is computed by equation 4, in which 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 [1] because the code is exactly the same regardless the number of threads used in the execution. The efficiency is computed by equation 5, in which S_p is the speedup and $\#p$ is the number of threads.

$$S_p = \frac{T_s}{T_p} \tag{4}$$

$$E_f = \frac{S_p}{\#p} \tag{5}$$

Table 1 presents the speedup and the efficiency for each parallel model of the PSO algorithm using 1000 iterations. As we can notice, the Island topology reaches the bests results using four and eight threads. After that number of threads, the efficiency decreases because the communication and the overhead of having more threads than physical cores start growing up. The overhead is more noticeable using 16 threads in the master-slave model using, in which the performance got worse than the execution in one thread. In the island and hierarchical model, the speedup increases up to eight threads, even though the efficiency decreases as the number of thread grows. Then the overhead caused by 16 threads leads to a small improvement less than using four cores.

Table 1: 1000 iterations

Master-Slave			
#nt	Time (ms)	Sp.	Ef.
1	4771.4	-	-
2	2602.8	**1.8331**	**91.65**%
4	1700.6	**2.8057**	**70.14**%
8	1386.7	**3.4408**	**43.01**%
16	11170.4	0.4271	0.02%
Island			
#nt	Time (ms)	Sp.	Ef.
1	2375.1	-	-
2	1244.3	**1.9087**	**95.43**%
4	728.9	**3.2584**	**81.46**%
8	515.2	**4.6100**	**57.62**%
16	856.2	2.7740	17.33%
Hierarchical			
#nt	Time (ms)	Sp.	Ef.
1	4337.6	-	-
2	2572.2	**1.6863**	**84.31**%
4	1844.1	**2.3521**	**58.80**%
8	1694.4	**2.5599**	**31.99**%
16	2103.2	2.0623	12.88%

Table 2: 2000 iterations

Master-Slave			
#nt	Time (ms)	Sp.	Ef.
1	9492.5	-	-
2	5200.1	**1.8254**	**91.27**%
4	3371.5	**2.8155**	**70.38**%
8	2718.6	**3.4916**	**43.64**%
16	23091.0	0.4110	2.56%
Island			
#nt	Time (ms)	Sp.	Ef.
1	4481.2	-	-
2	2325.0	**1.9273**	**96.36**%
4	1246.7	**3.5944**	**89.86**%
8	833.8	**5.3744**	**67.18**%
16	932.6	**4.8050**	**30.03**%
Hierarchical			
#nt	Time (ms)	Sp.	Ef.
1	8632.2	-	-
2	5147.3	**1.6770**	**83.85**%
4	3683.6	**2.3434**	**58.58**%
8	3396.1	**2.5417**	**31.77**%
16	4254.0	2.0291	12.68%

Table 2 illustrates the speedup and the efficiency for each parallel model of the PSO algorithm using 2000 iterations. The results are quite similar to previous ones; however, the gain in terms of speedup is higher because the granularity of the task increases as well.

Figure 7 presents how speedup evolves as we increase the number of threads. As we can observe, the speedup increases up to eight threads then start decreasing showing that communication and overhead start affecting the performance. In the island model, the gain tends to be better as we increase the number of iterations because the granularity of the task also increases.

4.2 Programming Effort

According to da Silva et al. [15], we can compute the saved programming effort using a popular measure called Constructive Cost Model (COCOMO) [27–29]. COCOMO is a general regression-cost model as presented in Equations 6, in which PM represents the effort (person/month), A is a calibration factor, $KLOC$ is the number of lines of code (in terms of 1000 lines or K), B depicts a scale factor, and ME is the product of all predefined efforts involved in the project. There are fifteen predefined efforts as we can see in [30].

$$PM = A \times (KLOC)^B \times \prod_{i=1}^{n}(ME_i) \tag{6}$$

Thus, taking into account that this kind of code is intermediate and organic, as done in [15], the resulting equation is $PM = 3.2 \times (KLOC)^{1.05} \times \prod_{i=1}^{n}(ME_i)$. Then, considering that the following ME were used: RELY, CPLX, ACAP, AEXP, PCAP, LEXP, MODP, and SCED, we obtain the following MEs according to the experience of the developers regarding parallel computing and particle swarm optimization:

- Experts - $ME = 0.7$

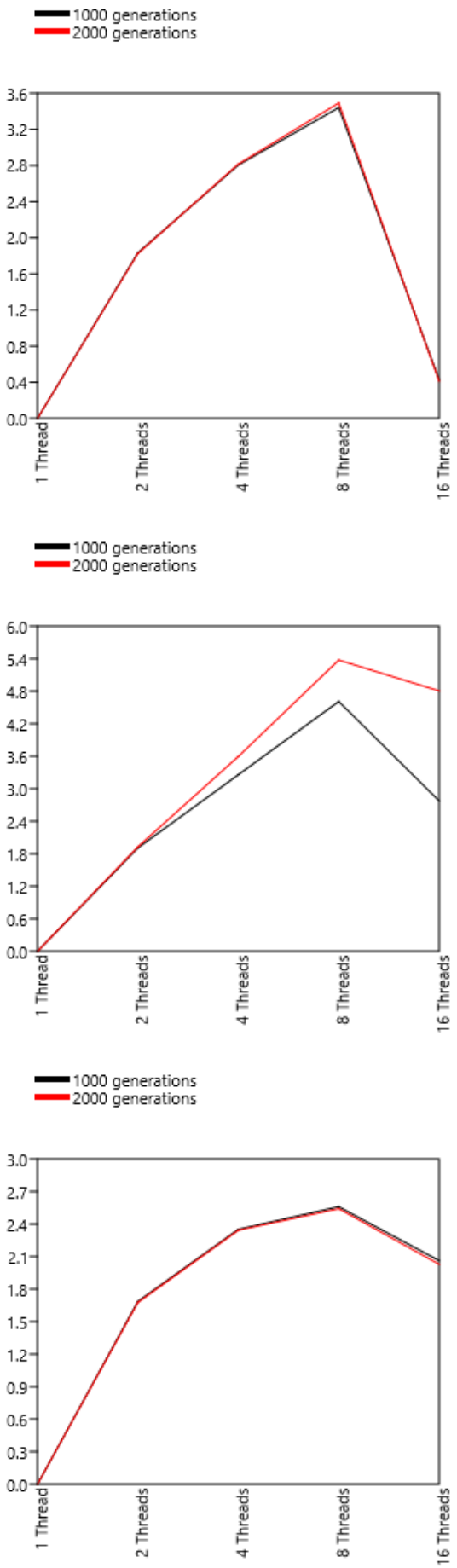


Figure 7: Speedup of the Master Slave, Island and Hierarchical for 1000 and 2000 iterations

- Beginners - $ME = 2.4$

Therefore, because of the Parallel PSO is composed by 1913 lines, summing up all parallel models, the final effort is $PM = 4.4$ person/month for an expert team and $PM = 14.5$ for a team with no expertise. In other words, the saved effort is within the interval [4.4, 14.5] (pearson/month) depending on the expertise of the team. Having this results, we can compute the time required to implement all parallel models ($time$) and the number of people (p) necessary to implement them as shown in Equations 7 and 8, respectively.

$$time = C \times PM^D \tag{7}$$

$$p = \frac{PM}{time} \tag{8}$$

The constants to compute time are $C = 2.5$ and $D = 0.38$, resulting in $time = 2.5 \times 4.4^{0.38} = 4.39$ months and $time = 2.5\ times14.5^{0.38} = 6.9$ months, for an expert team and for a beginner team, respectively. In other words the time required to implement all parallel models is in the range $[4.39, 6.9]$ months, depending on the team expertise. Hence, the number of people required to the task is $p = \frac{4.4}{4.39} = 1$ and $p = \frac{14.5}{6.9} = 2.1$. Table 3 summarizes the saved effort. All in all, according to the COCOMO model, an expert person might develop the entire code, the 1913 lines of code, in 4.39 months and a team of beginners might implement all models in 6.9 months using two people.

Table 3: Summarized Effort

Team	Beginners	Expert
Effort	14.5 person/month	4.4 person/month
Time	6.9 months	4.39 months
People	2 people	1 pearson

5. Conclusions

In this article, we showed the development and test of an automatic code generator for the Parallel Particle Swarm Optimization algorithm using three different parallel models: master-slave, island, and hierarchical. Results showed that all three approaches reach a good speedup taking into account the computer where tests have been held, the granularity of the tasks, and the overhead caused by creating more threads than physical cores. In our experiment, we reached a speedup of 4.61 using 8 cores on the island model and 1000 iterations. Also, we reached a speed up of 5.3 with 8 cores and 2000 iterations, indicating that the more the iterations, the better the speedup.

Moreover, we estimate the minimum and maximum effort saved by our tool using a model named COCOMO. As a result, we estimated that an effort between 4.4 and 14.5 (person/month) could be saved depending on the experience of the team who would develop all the parallel models. Indeed, we can save between 4.39 and 6.9 months of work depending on the team.

Future work includes: tests using computational intense evaluations functions; an adaptive approach for updating the acceleration constants ($c_1 and c_2$) and the inertia weight (w) in the parallel version; the implementation of parallel differential evolution algorithms; the development of parallel multi-objective algorithms; the generation of asynchronous code; and, the generation of code for execution on General Purpose Graphic Unit Processors (GPGPUs) using C-CUDA and OpenCL.

REFERENCES

[1] E. Alba and M. Tomassini. "Parallelism and evolutionary algorithms". *IEEE Transactions on Evolutionary Computation*, vol. 6, no. 5, pp. 443–462, 2002.

[2] K. R. L. Jaquie. "Extensão da Ferramenta de Apoio à Programação Paralela (FAPP) para ambientes paralelos virtuais". Ph.D. thesis, Universidade de São Paulo, 1999.

[3] W. Gropp, E. Lusk and R. Thakur. *Using MPI-2: Advanced features of the message-passing interface*. MIT press, 1999.

[4] J. Malacarne. "Ambiente Visual para Programação Distribuída em Java". *Porto Alegre: PPGC da UFRGS*, 2000.

[5] F. Pasini and F. L. Dotti. "Code generation for parallel applications modelled with object-based graph grammars". *Electronic Notes in Theoretical Computer Science*, vol. 184, pp. 113–131, 2007.

[6] D. P. Playne and K. A. Hawick. "Auto-generation of parallel finite-differencing code for mpi, tbb and cuda". In *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, pp. 1168–1175. IEEE, 2011.

[7] Y. Xiang, C. Chen, H. Wang and Z. Zhou. "An improved automatic MPI code generation algorithm for parallelizing compilation". In *2017 IEEE 2nd Advanced Information Technology, Electronic and Automation Control Conference (IAEAC)*, pp. 1623–1626, March 2017.

[8] A. D. Central. "x86 Open64 Compiler Suite". [Online; accessed 16-October-2018].

[9] A. W. O. Rodrigues, F. Guyomarc'h, J.-L. Dekeyser and Y. Le Menach. "Automatic multi-GPU code generation applied to simulation of electrical machines". *IEEE Transactions on Magnetics*, vol. 48, no. 2, pp. 831–834, 2012.

[10] K. A. Hawick and D. P. Playne. "Automated and parallel code generation for finite-differencing stencils with arbitrary data types". *Procedia Computer Science*, vol. 1, no. 1, pp. 1795–1803, 2010.

[11] M. Steuwer, T. Remmelg and C. Dubach. "LIFT: A functional data-parallel IR for high-performance GPU code generation". In *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pp. 74–85, Feb 2017.

[12] K. Hou, H. Wang, W. Feng, J. S. Vetter and S. Lee. "Highly Efficient Compensation-Based Parallelism for Wavefront Loops on GPUs". In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 276–285, May 2018.

[13] K. Hou, H. Wang and W. Feng. "A Framework for the Automatic Vectorization of Parallel Sort on x86-Based Processors". *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 5, pp. 958–972, May 2018.

[14] J. A. da Silva, O. A. C. Cortes and E. J. "A java-based code generator for parallel evolutionary algorithms". In *1st BRICS Countries Congress on Computational Intelligence and 11th CBIC Brazilian Congress on Computational Intelligence*, 2013.

[15] J. A. da Silva, O. A. C. Cortes, E. J. V. SÃ¡ and A. Rau-Chaplin. "An Automatic Code Generator for Parallel Evolutionary Algorithms: Achieving Speedup and Reducing the Programming Efforts". In *The Ninth International Conference on Advanced Engineering Computing and Applications in Sciences (ADVCOMP)*, pp. 36–41, 2015.

[16] J. Herrington. *Code generation in action*. Manning Publications Co., 2003.

[17] J. Karasek, R. Burget, M. K. Dutta and A. Singh. "Java evolutionary framework based on genetic programming". In *2014 International Conference on Signal Processing and Integrated Networks (SPIN)*, pp. 606–612, Feb 2014.

[18] B. W. Boehm, C. Abts, A. W. Brown, S. Chulani, B. K. Clark, E. Horowitz, R. Madachy, D. J. Reifer and B. Steece. *Software Cost Estimation with COCOMO II*. Prentice Hall Press, Upper Saddle River, NJ, USA, first edition, 2009.

[19] J. Kennedy. "Particle swarm optimization". In *Encyclopedia of machine learning*, pp. 760–766. Springer, 2011.

[20] M. Clerc. "Standard particle swarm optimisation". 2012.

[21] T. Rauber and G. Rünger. *Parallel Programming Models*, pp. 93–149. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[22] J. Yao. "Analysis of Scalable Parallel Evolutionary Algorithms", 2006.

[23] E. Alba and M. Tomassini. "Parallelism and evolutionary algorithms". *IEEE Transactions on Evolutionary Computation*, vol. 6, no. 5, pp. 443–462, Oct 2002.

[24] D. Lucrédio. "Uma abordagem orientada a modelos para reutilização de software". *INSTITUTO DE CIÊNCIAS MATEMÁTICAS E DE COMPUTAÇÃO UNIVERSIDADE DE SÃO PAULO*, p. 37, 2009.

[25] J. D. Gradecki and J. Cole. *Mastering Apache Velocity*. John Wiley & Sons, 2003.

[26] M. Locatelli. "A note on the Griewank test function". *Journal of global optimization*, vol. 25, no. 2, pp. 169–174, 2003.

[27] B. Boehm, B. Clark, E. Horowitz, C. Westland, R. Madachy and R. Selby. "Cost models for future software life cycle processes: COCOMO 2.0". *Annals of Software Engineering*, vol. 1, no. 1, pp. 57–94, 1995.

[28] T. Sharma. "Analysis of Software Cost Estimation using COCOMO II". *International Journal of Scientific & Engineering Research*, vol. 2, no. 6, pp. 1–5, 2011.

[29] C. Thirumalai, R. R. Shridharshan and R. Ranjith. "An Assessment of Halstead and COCOMO Model for Effort Estimation". In *International Conference on Innovations in Power and Advanced Computing Technologies*, pp. 1–4, 2017.

[30] B. W. Boehm. *Software cost estimation with COCOMO II*. Prentice Hall, Upper Saddle River, NJ, 2000.