Parallel Implementation of the Particle Swarm Optimization Algorithm on a Multiprocessor Embedded System with Network-on-Chip

Alberto de Carvalho Passos[1], Luiza de Macedo Mourelle [2], Nadia Nedjah [3]

[1]*Electronics Engineering Postgraduate Programme*
albertopassos.rj@gmail.com
[2]*Departmet of Systems Engineering and Computation*
ldmm@eng.uerj.br
[3]*Department of Electronics Engineering and Telecommunications*
nadia@eng.uerj.br
State University of Rio de Janeiro
524 São Francisco Xavier St., 20550-900, Maracanã, Rio de Janeiro, RJ, Brazil

Abstract. In recent years, with technological advancements, the need to solve complex problems has emerged in various areas of knowledge, such as data mining, combinatorial optimization, power systems, signal processing, pattern recognition, machine learning, and robotics. The key characteristic of these problems is their computational intensity, particularly in terms of execution time. In order to accelerate the problem-solving process, bio-inspired algorithms have been developed, which aim to simulate the behavior found in biological systems, such as living organisms and ecosystems, to efficiently solve complex problems. Examples of these algorithms include Particle Swarm Optimization, Ant Colony Optimization, Artificial Bee Colony, and Cuckoo Search. This work aims to obtain a parallel implementation of the Particle Swarm Optimization algorithm using a Multiprocessor Embedded System with Network-on-Chip. The parallelization strategies we employ are based on the Parallel Particle Swarm Optimization and Cooperative Parallel Particle Swarm Optimization algorithms, using master-slave, ring, and 2D grid topologies. Based on the execution time obtained by each parallel algorithm and each employed topology during the simulations, it will be possible to identify which parallelization strategy provides the best performance, as well as the number of processors required. Currently, the results, when compared to the serial version of the Particle Swarm Optimization algorithm, are promising.

Keywords: Particle swarm optimization, Parallel algorithms, Multiprocessor embedded system, Network-on-chip

1 Introduction

Particle Swarm Optimization (PSO), introduced by Kennedy and Eberhart [1], is one of the most popular bio-inspired optimization algorithms [ref]. It is based on a swarm of particles distributed over a search space, looking for the optimum solution. In general, the execution of PSO is computationally intensive due to the need for handling vast amounts of data when solving problems involving numerous particles or dimensions. The performance achieved by PSO, in these instances, can be impacted as the processing time for the outcome might increase significantly. In an attempt to seek improvement in the performance of the PSO, various parallelization approaches emerged. One idea is to enable the main code to be divided into smaller tasks and executed on multiple cores, resulting in more efficient processing. Parallelization also allows for solving previously infeasible problems sequentially or accelerating the processing of much larger volumes of data through distributed processing. Implementations in CUDA, OpenMP, MPI provide the necessary means to obtain a parallel implementation running in computers.

An embedded system provides an optimized implementation of a complete system in a single chip, which may consist of processors, memories, input/output controllers, coprocessors. The resulting component offers reduced area, low energy consumption, low response time. A multiprocessor embedded system, on the other hand, employs a set of processors in order to provide parallelism. The problem now is the communication between processors,

which can not be done via shared bus. In order to deal with this problem, the concept of network-on-chip, as introduced by Benini and De Micheli [2], emerged as an alternative to provide for scalability.

In this work, we intend to investigate a parallel version of the PSO, running in a multiprocessor embedded system. Our aim is to identify the relation between the number of processors and the execution time, in order to obtain an implementation that minimizes area and execution time. The communication between the processors is implemented via a network-on-chip, which provides the necessary means for task communication. Here, we identify another challenge, since increasing the number of processors can lead to communication overhead.

This paper is organized into seven sections. Section 2 presents related works. Section 3 defines the PSO algorithm. Section 4 describes the platform used in for the simulations. Section 5 introduces the parallelization strategies and the algorithms employed. Section 6 presents the obtained results and analysis of the data. In Section 7, we present some conclusions and draw some suggestions for future work.

2 Related Works

In Lalwani et al. [3], a comprehensive overview of the PSO algorithm and its parallel forms, such as star, ring, broadcast, and scatter, is presented.The Master-Slave, Island Model, and Cellular Model strategies are also discussed, along with the platforms used for implementing the applications. According to the author, to harness parallel computational resources, we can utilize one or multiple machines. When using a single machine, we can make use of a processor with multiple cores or even multiprocessors, in addition to the possibility of utilizing Graphics Processing Units (GPUs) with several specialized cores. Other parallelization techniques are also discussed, using CPUs such as Hadoop MapReduce, MATLAB Parallel Computing Toolbox, R Parallel package, Julia: Parallel for and MapReduce, the Parallel Computing module in Python, and OpenMP with C++.In [4], the technique of distributed computing using multiple machines through the Message Passing Interface (MPI) is introduced. Using MPI, the algorithm is divided into subcodes that are distributed to various computational units which execute portions of the code, and then send the results back to the main computer.

In Calazan [5], a solution is also discussed that utilizes a hardware and software-based architecture, employing co-design to implement the PSO algorithm. The module for updating particle velocity and position is implemented in hardware, while the module for the objective function is implemented in software.

In Venter and Sobieszczanski-Sobieski [6], a parallel PSO can be implemented by simply adding the calculation of the objective function in its algorithm, which will be performed in parallel by other processors without altering the algorithm's logic. In this synchronous model, there is a waiting limitation for processing in all slaves before the calculation can proceed, leading to low efficiency.

In Koh et al. [7], the idea is to avoid the need for synchronization at the end of each iteration, preventing the halt in the calculation of the objective function and thereby enhancing efficiency. In terms of implementation, synchronous and asynchronous parallel PSO only differ in the optimization step. Because there's no synchronization point, only a global calculation order is needed before the optimization procedure begins, whereas in synchronous algorithms, synchronization occurs at the start of each iteration.

In Hung and Wang [8], a problem with 100 dimensions and 65,536 (16×212) particles is solved. The GPU-accelerated PSO (GPSO) achieved accelerations of up to 280X and 83X on an NVIDIA Tesla C1060 GPU with a clock speed of 1.30 GHz compared to a single-core and quad-core Intel Xeon-X5450 central processing unit running at 3.00 GHz. The key point to achieving efficiency is harnessing the power of the multi-core multiprocessors and optimizing memory bandwidth on the GPU. According to the findings of this study, GPSO can reduce computational time, achieve high parallel efficiency and utilize numerous particles to discover improved optimal solutions.

In Tewolde et al. [9], parallelism is employed to execute PSO using scalable hardware modules. The objective is to further accelerate the algorithm's execution, with one of the main applications being its use in small embedded systems. In this model, the swarm is divided into a certain number of sub-swarms that operate simultaneously.

3 PSO algorithm

PSO employs a swarm of particles that, through social and cognitive behaviors, seek the solution according to the defined objective. Particles move through the search space in pursuit of the optimal solution. Each one represents a candidate solution, having an associated position and velocity. The algorithm leverages collaboration and interaction among particles to discover improved solutions over time. The movement of each particle is governed by eqs. (1) and (2):

$$v_{k+1} = wv_k + c_1 r_1 (pbest_k – x_k) + c_2 r_2 (gbest – x_k), \tag{1}$$

$$x_{k+1} = x_k + v_{k+1}. \tag{2}$$

In eq. (1), v_{k+1} is the particle's velocity in the next iteration. w defines the inertia factor, that controls the influence of the previous velocity, with values between 0.4 to 1.0. c_1 and c_2 are the learning coefficients, that control the influence of the individual best position $pbest$ and the global best position $gbest$, respectively, determining the influence of individual and global exploration in the optimization process. r_1 and r_2 are random values between 0 and 1. x_k represents the current position of the particle. $pbest$, which initially holds the same values as the initial particle coordinates, refers to the best position found by the particle so far, corresponding to the cognitive component. $gbest$ represents the best position found by any particle in the swarm up to that point, corresponding to the social component. The final position of the particle is determined by its current position plus the displacement value as per eq. (2). Algorithm 1 outlines the steps of the PSO. The number of particles is defined by $npart$. The fitness function, corresponding to the function to be optimized, is applied to each particle, at every iteration, while $pbest$ and $gbest$ are updated accordingly. The stop condition can be based on the maximum number of iterations, a precision limit or a minimum improvement in the solution.

Algoritmo 1 PSO

initialization w, c_1, c_2, r_1, r_2, $npart$;
Create a swarm of particles;
repeat
  for $i := 1 \rightarrow npart$ **do**
    Calculate particle fitness;
    if $fitness < pbest_i$ **then**
      Update $pbest_i$
    end if
    if $pbest_i <= gbest$ **then**
      Update $gbest$;
    end if
    Update particle velocity using Eq. (1);
    Update particle position using Eq. (2);
  end for
until Stop condition
return $gbest$

4 Simulation platform

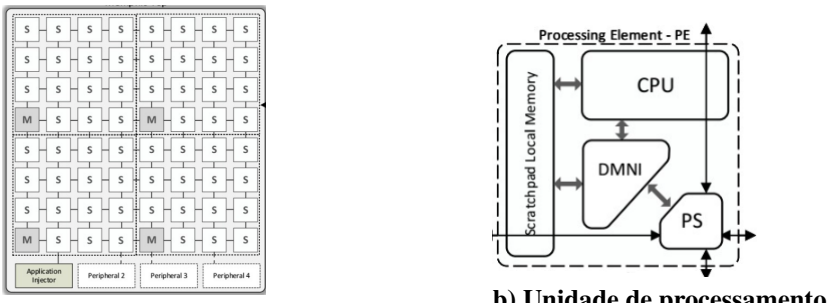
The chosen platform for the practical experiments in this work was MEMPHIS (Many-core Modeling Platform for Heterogeneous SoCs), as described by Ruaro et al. [10], shown in the Fig. 1. This platform was designed for modeling many Processing Elements (PEs), which are integrated into a single integrated circuit known as a Multiprocessor System-on-Chip (MPSoC). The platform operates using the LINUX operating system and the C programming language for task execution, alongside a specific messaging protocol for communication among PEs.

The PEs, referred to as S (Slave Processor), are used for task execution, while those, referred to as M (Cluster Manager), handle task distribution among the Ss, performing the task mapping onto the PEs, which can be done statically or dinamically. During task execution, the PEs communicate through message passing, transffering packages in analogy to a computer network. Associated to each PE, there is a switch that performs the routing function, based on the XY routing algorithm.

We selected MEMPHIS primarily due to its scalability. The number of processors can be chosen prior to the experiment. The platform can provide timing information, such as communication time between processors, according to specifications of the user. The communication time can be used to analyze the impact of the number of processors used and the mapping strategy adopted in the execution time of the application.

5 Parallelization strategies

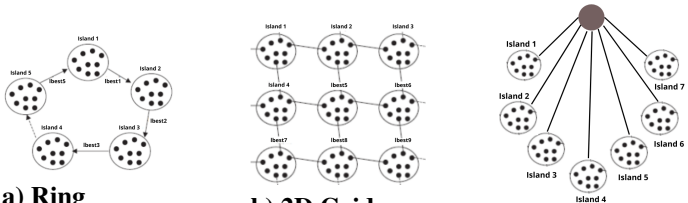
In an effort to enhance the performance of PSO execution, the proposition of parallelization emerged. The idea is to harness parallel processing capabilities, which, among other advantages, facilitate updating multiple particles simultaneously, resulting in a shorter time to reach a solution. Through parallelism, it is also possible



a) Plataforma MEMPHIS **b) Unidade de processamento**

Figure 1. Overview of the MEMPHIS hardware model,and a view of an isolated processing unit

to conduct a more efficient search by dividing the search space into smaller areas. Consequently, particles can be distributed into sub-swarms, called islands Lalwani et al. [3], allowing them to explore these spaces simultaneously. These advantages enable dealing with a larger number of particles, which is highly useful in solving complex problems requiring substantial computational power and computationally intensive fitness evaluations. In this work, we investigate the performance offered by three parallel topologies: ring (Fig. 2a), 2D grid (Fig. 2b) and master-slave (Fig. 2c).



a) Ring **b) 2D Grid** **c) Master-slave**

Figure 2. Parallelization strategies

In the ring topology, shown in Fig. 2a by Waintraub et al. [11], sub-swarms are allocated to neighboring processors in our simulation environment and will exchange information to benefit from the best results of the processors located to the right and left, defined respectively by $lbest_r$ *and* $lbest_l$. Algorithm 2 describes the PSO steps related to the ring strategy, wherein parameter $npart$ corresponds to the total number of particles that will be used by each processor in its subspace of search.

Algoritmo 2 Parallel PSO ring topology

1: **initialization** c_1, c_2, $npart$, $pbest$;
2: **Create** a Sub-swarm with $npart$ particles;
3: **repeat**
4: **Calculate** PSO ;
5: **Receive** $lbest$ values of neighbors on the right and left $lbest_l$ *and* $lbest_r$;
6: **if** $lbest_l < lbest$ **then**
7: $lbest = lbest_l$;
8: **end if**
9: **if** $lbest_r < lbest$ **then**
10: $lbest = lbest_r$;
11: **end if**
12: **until** Stop condition;
13: **return** $gbest$.

In the 2D grid topology, shown in Fig.2b by Waintraub et al. [11], there is an exchange of information between four sub-swarms. As a result, the algorithm can gather more information, covering a larger area with potential solutions for the problem. In our simulation environment, this is the case where neighboring processors are located to the right, left, above, and below each of the processors that are part of the topology. This broader communication exchange enables each processor to benefit from the results found by the others in the topology. This algorithm is similar to the one described for the ring strategy and only differs in the number of processors

involved in the communication exchange. Particles have access to information about the best solution from the sub-swarms to their right ($lbest_r$), left ($lbest_l$), up ($lbest_u$) and down ($lbest_d$), as described in alg. 3. In this topology, there are 3 types of possible configurations depending on the number of processors located around for information exchange. For this step in the algorithm, the boolean variables $proc_l$, $proc_r$, $proc_u$, and $proc_d$ have been included to check the presence or absence of the processors that are around.

Algoritmo 3 Parallel PSO with 2D grid topology

initialization c_1, c_2, $npart$, $pbest$;
Create a sub-swarm with $npart$ particles;
repeat
  Calculate PSO;
  Receive $lbest_l$, $lbest_r$, $lbest_u$ *and* $lbest_d$;
  if $proc_l = True$ **then**
    if $lbest_l < lbest$ **then**
      $lbest = lbest_l$
    end if
  end if
  if $proc_r = True$ **then**
    if $lbest_r < lbest$ **then**
      $lbest = lbest_r$;
    end if
  end if
  if $proc_u = True$ **then**
    if $lbest_u < lbest$ **then**
      $lbest = lbest_u$;
    end if
  end if
  if $proc_d = True$ **then**
    if $lbest_d < lbest$ **then**
      $lbest = lbest_d$;
    end if
  end if
until Stop condition
return $gbest$.

In the master-slave topology, depicted in Fig. 2c, the master is responsible for the overall algorithm coordination, while the slaves carry out the computations of the particles. This approach is beneficial when there is a substantial number of particles and parallel processing is needed to handle the computational load. In Alg. 4, we present the steps of the master process, while that for the slaves processes is outlined in Alg. 5.

Algoritmo 4 Master processor

initialization c_1, c_2, $npart$;
Send Slaves c_1, c_2, $npart$;
Receive $lbest_1$;
gbest = $lbest_1$;
for $i := 2 \rightarrow nproc$ **do**
  Receive $lbest_i$;
  if $lbest_i < gbest$ **then**
    $gbest = lbest_i$;
  end if
end for;
return $gbest$.

Algoritmo 5 Slave processor

Receive c_1, c_2, $npart$;
Create a Sub-swarm with $npart$ particles;
repeat
  Calculate PSO;
until Stop condition;
Send $lbest$ to Master processor.

6 Experimental results

The performance of the parallel implementation of the PSO is based on the speedup achieved, which is expressed by the relation between the serial and the parallel execution times. In order to do so, we used the Rosenbrock and Sphere test functions. The Rosenbrock function, represented by the eq. (3), presents the minimum value of 0, at the coordinate (1,1,1), and the Sphere function, represented by the eq. (4), presents a minimum value of 0, at the coordinates (1,1,1):

$$f(x) = \sum_{i=1}^{d-1} [100(x_{i+1} - x_i^2)^2 + (x_i - 1)^2], \tag{3}$$

$$f(x) = \sum_{i=1}^{d} x_i^2. \tag{4}$$

For the comparison, we used the same configurations when simulating the serial and parallel scenarios. The inertial factor w was defined as 1, the learning coefficients c_1 and c_2 were defined as 2, each. We employed 500 particles and a 3-dimensional search space, defined in the interval [0, 50] in each dimension. The particles were randomly generated within the search space and repositioned within this space, if they exceeded the upper or lower limits. As an additional test, all simulations, in the parallel version, were also conducted using 200 particles, in order examine the implications of this change on the results. The results obtained for the serial version are presented in Table 1 in ms and the stop criterion is the minimum value of the functions.

Table 1. Serial execution times

500 particles	*Rosenbrock*	*Sphere*
Number of iterations	190	138
Execution time (ms)	140,64	102,45

For all parallel simulations, we used 10 PEs. The particles were randomly generated within the search space defined for each processor, as shown in Table 2, in order to cover the same total search space as the serial version.

Table 2. Search space for each processor

Topologies	PE_1	PE_2	PE_3	PE_4	PE_5	PE_6	PE_7	PE_8	PE_9	PE_{10}
Master Slave	[0,5]	[5,10]	[10,15]	[15,20]	[20,25]	[25,30]	[30,35]	[35,40]	[40,45]	[45,50]
Ring	[25,30]	[5,10]	[10,15]	[15,20]	[20,25]	[0,5]	[30,35]	[35,40]	[40,45]	[45,50]
2D grid	[45,50]	[40,45]	[35,40]	[30,35]	[25,30]	[20,25]	[15,20]	[10,15]	[05,10]	[0,5]

The results for the parallel execution, using the master-slave strategy, are shown in Table 3. Speedup is the result of the relation between the serial and the parallel execution times, which expresses the performance of the parallel implementation, and Communication presents the percentage of execution time consumed with intertask communication, considering that each task is allocated to a different PE. Table 4 and Table 5 provide the performance results for the ring strategy and 2D grid strategy, respectively.

Table 3. Results of parallelism with master-slave strategy

	Rosenbrock	*Rosenbrock*	*Sphere*	*Sphere*
Number of particles	500	200	500	200
Iterations	1	92	1	29
Execution time (ms)	0,47	4,01	0,47	1,39
Speedup	299	35	218	74
Communication	53%	6,23%	53%	17,98%

7 Conclusion

The aim of this work was to obtain a parallel implementation of PSO in a Multiprocessor System-on-Chip with Network-on-Chip and analyze the speedup that this implementation could yield. In order to do so, we used a

Table 4. Results of parallelism with ring strategy

	Rosenbrock	*Rosenbrock*	*Sphere*	*Sphere*
Number of particles	500	200	500	200
Iterations	1	11	1	5
Execution time (ms)	0,58	1,98	0,58	1,05
Speedup	242	71	176	97
Communication	4,3%	2,5%	4,3%	2,3%

Table 5. Results of parallelism with 2D grid strategy

	Rosenbrock	*Rosenbrock*	*Sphere*	*Sphere*
Number of particles	500	200	500	200
Iterations	1	68	1	44
Execution time (ms)	0,57	4,14	0,54	2,70
Speedup	246	34	189	38
Communication	12,2%	8,4%	12,9%	10,3%

simulation platform, named MEMPHIS, which provides the possibility to determine the number of processors to be used, returns the execution time and the communication time between tasks. For these experiments, we used 10 processors and the search space was divided among them, in order to locate the minimum of the Rosenbrock and the Sphere functions. We applied three parallel strategies: master-slave, ring and 2D grid. The results obtained are very promising. For the Rosenbrock function, the best speedup obtained is similar between the master-slave and the 2D grid strategies. For the Sphere function, the best speedup obtained is for the 2D grid topology. The division of the search space into smaller areas, distributed among the processors, helped significantly in reducing the computational effort required by each one.

There is some work still to be done. One of our main goals is to identify the minimum number of processors required to optimize a function using PSO, since this is related to the cost of the implementation. Along the way, we will be able to identify the minimum number of particles required as well. Another analysis to be done is to identify the relation between the parallel strategy and the function complexity.

Authorship statement. The authors hereby confirm that they are the sole liable persons responsible for the authorship of this work, and that all material that has been herein included as part of the present paper is either the property (and authorship) of the authors, or has the permission of the owners to be included here.

References

[1] J. Kennedy and R. Eberhart. Particle swarm optimization. In *Proceedings of ICNN'95 - International Conference on Neural Networks*, volume 4, pp. 1942–1948 vol.4, 1995.
[2] L. Benini and G. De Micheli. Networks on chips: A new soc paradigm. *IEEE Computer*, vol. 1, pp. 70–78, 2002.
[3] S. Lalwani, H. Sharma, S. Satapathy, K. Deep, and J. Bansal. A survey on parallel particle swarm optimization algorithms. *Arabian Journal for Science and Engineering*, vol. 44, 2019.
[4] O. T. Altinoz, A. E. Yilmaz, and G.-W. Weber. Improvement of the gravitational search algorithm by means of low-discrepancy sobol quasi random-number sequence based initialization. *Advances in Electrical and Computer Engineering*, vol. 14, n. 3, pp. 55–63, 2014.
[5] R. M. Calazan. Otimização por enxame de partículas em arquiteturas paralelas de alto desempenho, 2013.
[6] G. Venter and J. Sobieszczanski-Sobieski. Particle swarm optimization. *AIAA Journal*, vol. 41, pp. 1583–1589, 2002.
[7] B.-I. Koh, A. D. George, R. T. Haftka, and B. J. Fregly. Parallel asynchronous particle swarm optimization. *International journal for numerical methods in engineering*, vol. 67, n. 4, pp. 578–595, 2006.
[8] Y. Hung and W. Wang. Accelerating parallel particle swarm optimization via gpu. *Optimization Methods and Software*, vol. 27, n. 1, pp. 33–51, 2012.
[9] G. S. Tewolde, D. M. Hanna, and R. E. Haskell. A modular and efficient hardware architecture for particle swarm optimization algorithm. *Microprocessors and Microsystems*, vol. 36, n. 4, pp. 289–302, 2012.
[10] M. Ruaro, L. Caimi, V. Fochi, and F. Moraes. Memphis: a framework for heterogeneous many-core socs generation and validation. *Design Automation for Embedded Systems*, vol. 23, 2019.
[11] M. Waintraub, R. Schirru, and C. Pereira. Parallel particle swarm optimization algorithms in nuclear problems. In de A. B. Energia Nuclear (ABEN), ed, *Proceedings of the International Nuclear Atlantic Conference (INAC 2009)*, volume 1, pp. 1–12. ABEN, 2009.