Journal of Artificial Intelligence and Data Mining (JAIDM), Vol. 9, No. 1, 2021, 59-71.

Shahrood University of Technology

Journal of Artificial Intelligence and Data Mining (JAIDM)
Journal homepage: http://jad.shahroodut.ac.ir

Research paper

A Distributed Sailfish Optimizer Based on Multi-Agent Systems for Solving Non-Convex and Scalable Optimization Problems Implemented on GPU

Soodeh Shadravan [1], Hamid Reza. Naji [2]* and Vahid Khatibi [3]

1. Department of Computer Engineering, Kerman Branch, Islamic Azad University, Kerman, Iran.
2. Department of Computer Engineering, Graduate University of Advanced Technology, Kerman, Iran.
3. Department of Computer Engineering, Kerman Branch, Islamic Azad University, Bardsir, Iran.

Article Info

Article History:
Received 18 February 2020
Revised 06 June 2020
Accepted 24 June 2020

DOI:10.22044/jadm.2020.9389.2075

Keywords:
SailFish Optimizer, Distributed Sailfish Optimizer, Multi-agent System, Parallel Processing, Shared Memory, Graphic Processing Units, CUDA.

**Corresponding author:* h.naji@kgut.ac.ir *(H. Naji).*

Abstract

The SailFish Optimizer (SFO) is a metaheuristic algorithm inspired by a group of hunting sailfish that alternate their attacks on a group of prey. The SFO algorithm takes advantage of using a simple method for providing a dynamic balance between the exploration and exploitation phases, creating the swarm diversity, avoiding local optima, and guaranteeing a high convergence speed. Nowadays, multi-agent systems and metaheuristic algorithms can provide high performance solutions for solving combinatorial optimization problems. These methods provide a prominent approach to reduce the execution time and improve the solution quality. In this paper, we elaborate a multi-agent based and distributed method for sailfish optimizer (DSFO), which improves the execution time and speeds up the algorithm, while maintaining the optimization results in a high quality. The Graphics Processing Units (GPUs) using Compute Unified Device Architecture (CUDA) are used for the massive computation requirements in this approach. In depth of the study, we present the implementation details and performance observations of the DSFO algorithm. Also a comparative study of the distributed and sequential SFO is performed on a set of standard benchmark optimization functions. Moreover, the execution time of the distributed SFO is compared with other parallel algorithms to show the speed of the proposed algorithm to solve the unconstrained optimization problems. The final results indicate that the proposed method is executed about maximum 14 times faster than the other parallel algorithms and shows the ability of DSFO for solving the non-separable, non-convex, and scalable optimization problems.

1. Introduction

A combination of metaheuristics such as computational intelligence and operational research work is called hybrid metaheuristic. The main reason of using this technique is to obtain the high-quality solutions with a reasonable computing time [1,2]. The hybrid metaheuristic algorithms usually use some advanced strategies like multi-agent systems, decomposition of the search space, and parallel computation. However, taking a lot of time to solve NP complete and high-dimensional problems has become a challenge for the metaheuristic algorithms. Due to the independence of the metaheuristics components, their combination with parallel processing and multi-agent systems is a good option to reduce the computational time and to increase the quality of the solutions. In intelligent multi-agent systems, a set of proactive agent acts individually for solving the problems collaboratively [3]. They are used in the metaheuristic algorithms to solve hard

optimization problems with their own intelligent skills and cooperation of different agents by distribution of the problem among agents and they are organized and coordinated into a complete way. The agent systems have shown a good performance for solving problems in large-scale distributed and dynamic systems [4]. In addition, the agents can react to environmental changes for adaptations with unpredictable events; this characteristic is very important for the intelligent systems [5]. In fact, it is up to the designer to determine how information is exchanged between the agents, which agents can share search space information such as solutions, sub-problems and the status of the agents, and how to control the solution process to acquire a better performance in various strategies. Compared with the centralized optimization algorithms, the distributed optimization algorithms based on the multi-agent systems provide a robust way to solve the large-scale problems. These systems also have the ability to decompose the complex problems into simpler and smaller problems that are operated by various agents [6]. When the distributed algorithm is described as a multi-agent system, all agents operate asynchronously and use parallelization strategies for enhancing the efficiency and speed of the execution time compared to the centralized systems.

In this paper, we describe a new distributed version of SailFish Optimizer (SFO) [7] based on the multi-agent systems using the CUDA architecture. In this method, a large number of iterations is investigated to reach the high quality solutions to solve various optimization problems, and it has the ability to improve the SFO's processing speed. Also the impacts of parallelism in high dimensionality problems are investigated. In addition, a large number of search agents are analyzed in order to achieve the desired SFO's computational time compared to the sequential implementation and other previously metaheuristic algorithms on GPU using the CUDA architecture.

The rest of this paper is organized as what follows. Section 2 describes the SFO algorithm and sketches the basic concepts of the GPU computing and CUDA architecture. Section 3 presents implementation of the distributed version of SFO to solve the optimization problems. The experimental results and analysis are discussed in Section 4. Finally, some concluding remarks and future works are given in Section 5.

2. Related Works

In much of the literature, the agent technologies have provided the practical framework for metaheuristic algorithms. Previously, a couple of multi-agent systems were applied to metaheuristic algorithms for examination of their performance in a team coordinated [8,9]. Each one of these research works provides different benefits of tackling search and problem solving. In [10] and [11], the autonomous agents have been used in Asynchronous Teams (A-Teams) that are associated via shared memory. Also a hybrid metaheuristic algorithm has been presented in [12], where each agent acts independently in the search space and it has collaborated with other agents through the multi-agent environment. The results obtained show the reduction of cost function by using the cooperation agents. In [13], a multi-agent based Gravitational Search Algorithm (GSA) has been presented and it has compared the execution time of the multiple agent implementation with the original GSA sequential implementation. In this method, different agents handle the small and simple components, and these multiple agents are used to express the parallelism strategy. Currently, the researchers show how we can use agent-based techniques for solving complex problems quickly. This manner has the potential flexibility and expandability to enhance the computational systems and creates a strong approach to the traditional multi-agent systems [14, 15].

Nowadays, the distributed optimization algorithms based on multi-agent systems have drawn much attention of the researchers and several algorithms have been proposed in the recent years [16-19]. In most of these methods, each agent computes the whole global minimizer. Map-building and classification are specific distributed optimization problems that can be solved by these approaches. In these problems, the data is physically distributed among their agents and the number of decision variables is independent from the number of agents. Therefore, due to this independence, they are good options for solving with distributed optimization algorithms. In [20], another distributed heuristic algorithm has been presented for detecting the optimal route between the three-way and intersections. The packets are delivered based on the selected routes from a source to the destination in vehicular ad hoc network. The results show the superiority of this algorithm over similar algorithms. Also the MOEA/D-TS algorithm is a hybrid metaheuristic algorithm that is derived from Multi-Objective Evolutionary Algorithm based on Decomposition (MOEA/D) and Tabu Search (TS) [21]. This algorithm uses the neighborhood search authority of TS along with the parallel computing of MOEA/D to cover the totality of the Pareto front by uniformly distributed

solutions. According to the final results obtained, the MOEA/D-TS algorithm could produce fully satisfactory results and outperforms the previous algorithms.

Moreover, the graphics processing units (GPUs) have become a strong tool to implement the parallel execution of hundreds of threads for metaheuristic algorithms [22]. Many parallel algorithms have been implemented with different designs that cover the use of GPUs to implement nature-inspired metaheuristics [23], and they offer different parallelism strategies and communication patterns of metaheuristics on GPUs [24]. Due to the independence of the metaheuristics components, combination of these algorithms by distributed optimization algorithms based on multi-agent systems and parallel processing can provide high performance solutions to quickly solve the combinatorial optimization problems.

3. Preliminaries

The main inspiration of the SFO algorithm will be described in this section. Then the proposed algorithm and the mathematical model are discussed in details.

3.1. Sailfish Optimizer (SFO)

The SFO algorithm presents a novel nature-inspired metaheuristic optimization algorithm and mimics the strategy of the group of hunting sailfish [7]. This algorithm includes two types of population, the population of sailfish for intensification of the search space and the population of sardines for diversification of the search space. In order to describe the proposed algorithm, it is assumed that the positions of sailfish are the variables of all solutions, while the *i*th member at the *k*th search agent has a current position $SF_{i,k}$ in a *d*-dimensional search space. The position of all sailfish is saved to the matrix *SF* and the following matrix shows the fitness value for all solutions:

$$SF_{Fitness} = \begin{bmatrix} f(SF_{1,1}\ SF_{1,2}\ \cdots\ SF_{1,d}) \\ f(SF_{2,1}\ SF_{2,2}\ \cdots\ SF_{2,d}) \\ \vdots\ \ \vdots\ \ \vdots\ \ \vdots \\ f(SF_{m,1}\ SF_{m,2}\ \cdots\ SF_{m,d}) \end{bmatrix} = \begin{bmatrix} F_{SF_1} \\ F_{SF_2} \\ \vdots \\ F_{SF_m} \end{bmatrix} \quad (1)$$

where $SF_{i,j}$ shows the value of the *j*th dimension of the *i*th sailfish, f calculates the cost function and will be saved in the matrix SF_{Fit}, and m indicates the number of sailfish. Another significant incorporator is the group of sardines in the SFO algorithm. It is assumed that the school of sardines is also swimming in the search space and their positions will be saved to the matrix *S* so that their fitness values are utilized as follows:

$$S_{Fitness} = \begin{bmatrix} f(S_{1,1}\ S_{1,2}\ \cdots\ S_{1,d}) \\ f(S_{2,1}\ S_{2,2}\ \cdots\ S_{2,d}) \\ \vdots\ \ \vdots\ \ \vdots\ \ \vdots \\ f(S_{n,1}\ S_{n,2}\ \cdots\ S_{n,d}) \end{bmatrix} = \begin{bmatrix} F_{S_1} \\ F_{S_2} \\ \vdots \\ F_{S_n} \end{bmatrix} \quad (2)$$

where $S_{i,j}$ indicates the value of the *j*th dimension of the *i*th sardine, f calculates the cost function of each sardine and saves in the matrix S_{Fit}, and n is the number of sardines. Moreover, the position of sailfish will be updated during the optimization. The new position of sailfish $X^i_{new\_SF}$ updates at the *i*th iteration as follows:

$$X^i_{new_{SF}} = X^i_{elite_{SF}} - \lambda_i \times \left(rand(0,1) \times \left(\frac{X^i_{elite_{SF}} + X^i_{injured_S}}{2} \right) - X^i_{old_{SF}} \right) \quad (3)$$

where $X^i_{elite\_SF}$ and $X^i_{injured\_S}$ are the best positions of sailfish and the best positions of sardines, respectively, $X^i_{old\_SF}$ determines the current position of sailfish, $rand(0,1)$ is a random number between 0 and 1, and λ_i is generated as follows:

$$\lambda_i = 2 \times rand(0,1) \times PD - PD \quad (4)$$

Due to the decrease in the number of prey during the group hunting, the PD parameter is a significant parameter for updating the position of sailfish around the prey school and shows the number of prey at each iteration as follows:

$$PD = 1 - \left(\frac{N_{SF}}{N_{SF} + N_S} \right) \quad (5)$$

where N_{SF} and N_S are the number of sailfish and sardines at each iteration, respectively. Figure 1 shows a 2D position of sailfish after and before an alternative attack and encircling the prey during collaborative hunting. The proposed alternative attack and encircling mechanism create a circle-shaped neighborhood around the solutions for approaching the prey from different directions by hunters.

In addition, for mimicking to update the position of sardines at the *i*th iteration, it can be formulated as follows:

$$X^i_{new\_S} = r \times \left(X^i_{elite\_SF} - X^i_{old_S} + AP \right) \quad (6)$$

where r is a random number between 0 and 1, $X^i_{elite\_SF}$ is the best position of sailfish formed until now, $X^i_{old\_S}$ is the current position of sardines, and

the amount of sailfish's attack power will be saved in AP parameter that is generated as follows:

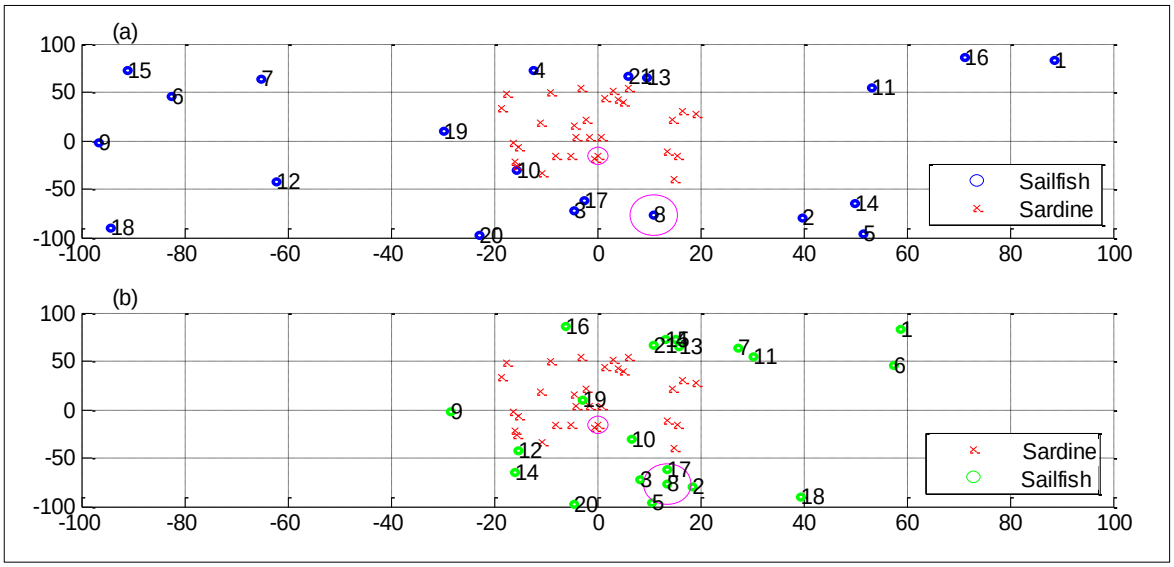

Figure 1. 2D position vectors of sailfish and sardines simulated by MATLAB: (a) before updating sailfish position, (b) after updating sailfish position. A big magenta circle is drawn around the elite sailfish and a small magenta circle is drawn around the injured sardine.

$$AP = A \times \left(1 - (2 \times Itr \times \varepsilon)\right) \quad (7)$$

where A and ε are the coefficients for decreasing the value of power attack linearly from A to 0. Using the AP parameter, the number of sardines that update their position (α) and the number of variables of problem (β) can be calculated as follows:

$$\alpha = N_S \times AP \quad (8)$$

$$\beta = d_i \times AP \quad (9)$$

where N_S indicates the number of sardines and d_i is the number of variables at the ith iteration. In order to show a conceptual model of position updating of a sardine in search space, figure 2 is illustrated. The horizontal axis shows only one dimension that is one variable or parameter of a given problem. However, the SFO algorithm can utilize all the variables of the problem. As it can be seen in this figure, the red multiplication signs are the possible positions that can be chosen as the next position of the sardine over the course of iteration. Finally, to increase the chance of hunting the new prey, the position of sailfish substitutes the latest position of the hunted sardine. The adaptive formula is as follows:

$$X_{SF}^i = X_S^i \quad if \ f(S_i) < f(SF_i) \quad (10)$$

where X_{SF}^i shows the current position of sailfish and X_S^i indicates the current position of sardine the ith iteration. The pseudo-code of SFO is

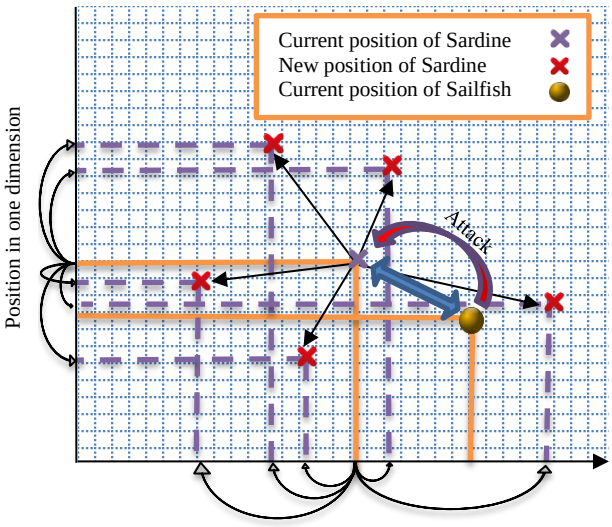

Figure 2. Sardine's position in one dimension with different r values stochastically.

summarized in table 1.

3.2. GPU Computing

Nowadays, the graphics processing units (GPUs) have obtained a high performance computing with a lower cost compared to the CPU-based architectures. Although GPUs have been mainly used to support the graphical applications, in the recent years, they have been employed for highly parallel computations.

This modern hardware has been successfully implemented in various fields such as image

processing [25], data mining [26], neural network computation [27], and data compression [28].
Also GPU computing is efficient in executing such synchronized parallel algorithms that involve data transfers and regular computations. Nevertheless, to benefit from these advantages, more complex programs are provided to solve problems with high arithmetic intensity and distributed architectures [29].

Table 1. Pseudocode for the SFO algorithm.

```
Initialize the population of sailfish and sardine randomly
Initialize parameters (A=4 and ε = 0.001).
Compute the fitness of sailfish and sardines.
Find the best sailfish and sardine and assume that they
 are as elite sailfish and injured sardine, respectively.
 While the termination conditions are not satisfied
    for each sailfish
       Calculate λ_i using(4).
       Update the position of sailfish using (3) and (10).
    end for
    Calculate AttackPower using (7).
       If AttackPower < 0.5
         Calculate α using (8).
         Calculate β using (9).
        Select a set of sardine based on the value of α and β.
  Update the position of the selected sardine by (6) and (10).
    else
      Update the position of all sardine by (6) and (10).
      end if
   Calculate the fitness of all sardine.
   If there is a better solution in sardine population     Replace a
     sailfish with injured Sardine using (10).
        Remove the hunted sardine from population.
        Update the best sailfish and best sardine.
     end if
  end while
Return best sailfish
```

3.3. An Overview of CUDA Architecture

Computing Unified Device Architecture (CUDA) is a multi-threaded programming model and parallel computing platform that has been developed by NVIDIA [30]. The CUDA architecture employs the multi-core parallel processing of a GPU and use C as a high-level programming language for solving complex computational problems. High quality of solutions and good scalability are effective advantages for implementation of an algorithm distributed on the CUDA platform. Using this method, the metaheuristic algorithms can scale the problem in a natural and decent way. Also it can steer the optimization with a few control variables.
The CUDA architecture is made up of an array of Streaming Multiprocessors (SMs) that have the ability to run several blocks in the kernel simultaneously. A kernel calls from CPU (named as host) and duplicates on the GPU (named as device), and is executed by a batch of threads. The Nvidia architecture supports several types of memory that the programmers can use to achieve a high execution speed in their kernels [31]. The largest memory is the global memory, and its content is visible to all threads of all launched kernels. However, the accessing global memory should be improved due to a high throughput and latency. Therefore, global memory is often used for moving data from one kernel to another one. Another type of memory is the constant memory that is a global memory with a special cache for efficient access. It often uses to provide the input value to kernel functions. Also registers and share memory are on-chip memories on GPU, and their variables can be accessed extremely fast and in parallel. For keeping the frequently accessed variables, a kernel will use the register memory, and the content of this memory is private to each thread. However, shared memory is allocated for threads within a block to collaborate to each other, and the contents of shared memory will be deleted after termination of a kernel. The GPU memory hierarchy is presented in figure 3.

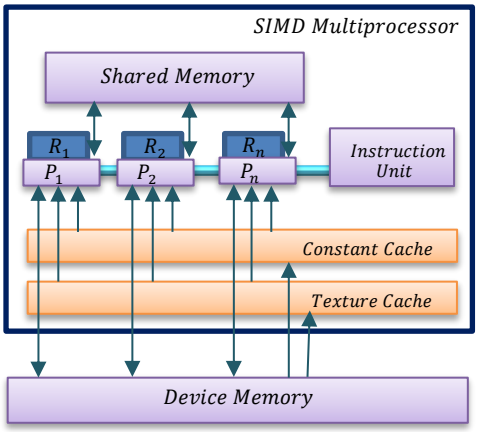

Figure 3. The GPU memory model

4. Implementing Distributed SFO using CUDA

In this section, the main parts of the Distributed SFO (namely, DSFO) based on multi-agent systems have been developed. A single agent in DSFO algorithm implements each group of operations. Then the agents act in parallel for gathering information from other agents or environments and return the results to the other agents or environments. The input information is a combination of the current position and best position of the search agents, and this information will be saved in the memory over the course of iteration during parallelization.
In order to reduce the consumed time and improve the optimization speed, DSFO is composed of the decision-maker agents, exploration agents, and exploitation agents. The implementation of

updating the sailfish's and sardine's positions is designed for the purpose of exploration and exploitation agents, respectively. In addition, the decision-maker agents calculate the cost of search agents and decide whether the current search agent in this iteration is the best or not.

The DSFO algorithm is expressed in a CUDA-based pseudo-code with six kernel functions. The first kernel generates random numbers on a GPU using the CURAND library [32]. The CURAND library is used to generate the high-quality random numbers for sailfish's and sardine's populations.

In this kernel, using k blocks of h threads, the position of sailfish and sardine will be utilized on a GPU. For this initialization, the matrices SF and S have been converted to the arrays to realize the coalescing memory access. Conversion of the matrix SF to the array SFd is shown in figure 4.

In this figure, $SF_1 \cdots SF_m$ represents the position of sailfish, where m and dim indicate the number of sailfish and the number of variables, respectively.

Due to the limited number of threads per block and to prevent the production of unused threads in each block, we assumed that the size of the block in our GPU architecture was equal to 900, and the random numbers were allocated to 900 decent threads in each block. As shown in figure 4, each sub-swarm of sailfish and sardines is associated with one block of threads and each dimension is mapped onto a distinct thread. Each block is also considered as an agent to calculate the cost function and determine the best search agent in each iteration.

The second kernel generates blocks of 900 threads to compute the objective functions for the sailfish's and sardine's populations, respectively. Calculation of the fitness value depends on the number of search agents so the time of calculation is proportional to the size of populations. In addition, in this kernel, the fitness value is calculated via shared memory. The reason for using the shared memory variables is the calculation of fitness value via global memory consuming a lot of time and will decrease the speed of optimization. Therefore, as shown in figure 5, the information of each block will be transferred to the shared memory variable with the same size and the fitness value of each search agent will be saved respect to a given dimension.

Through another shared memory variable with the shared memory variable with the same size and the fitness value of each search agent will be saved through another shared memory variable with respect to a given dimension. When the calculation of cost functions is finished, all of the fitness values will be sorted in the shared memory. Due to the sorting operation, thread 0 contains the best value in this stage, and it is also responsible for writing the result to the global memory.

After completing this process, the best value of each block will be transferred from the global memory to the share memory, and the current best values are compared with the previous values in the third kernel. Also using the --syncthreads() function, the tasks of thread will be finished in the shared memory before any of them moving on to the next iteration. In this case, none of the threads would load their contents too early and destroy the input value for other threads.

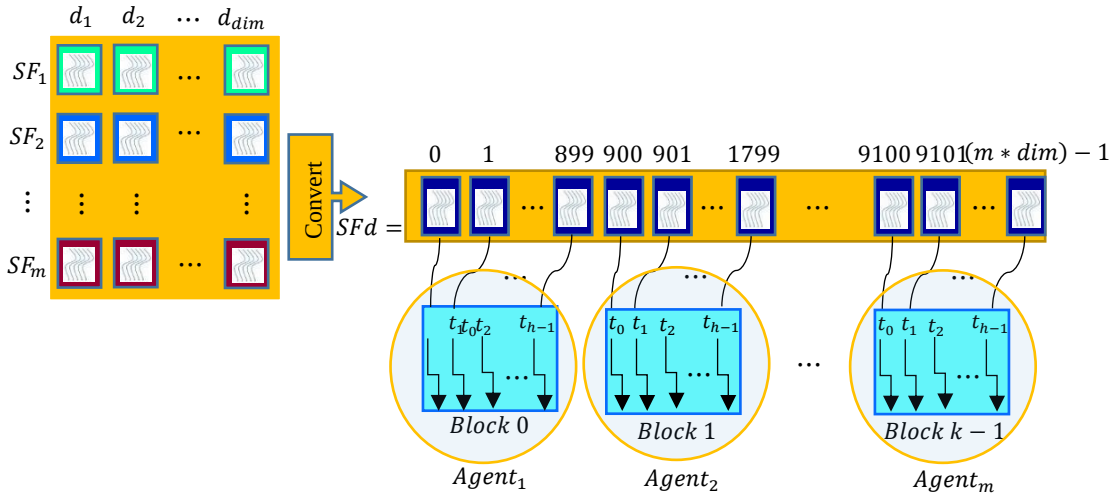

Figure 4. Conversion of the matrix SF to the array SFd and assigning random numbers to the threads and the existing agents.

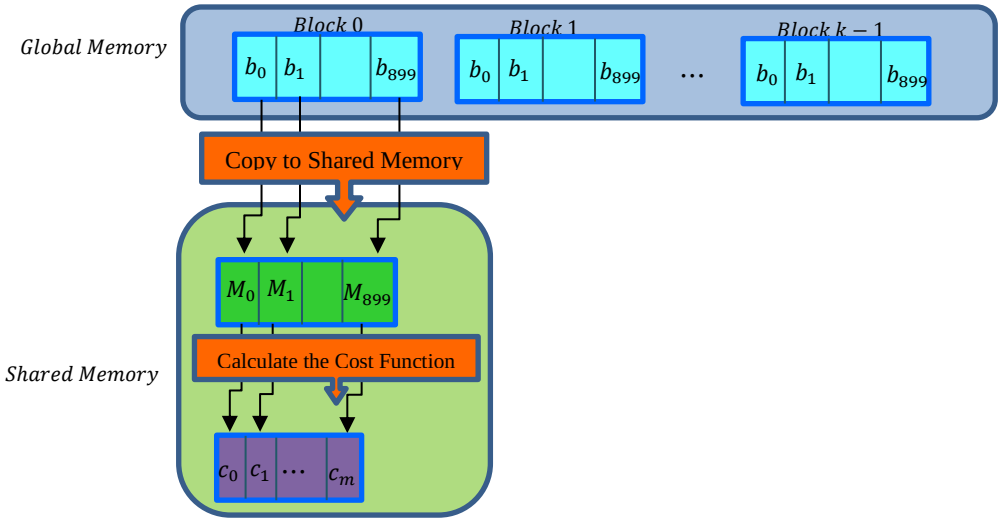

Figure 5. To transfer the information of blocks to shared memory variables for calculation of cost function.

In the third kernel, the decision-maker agents decide to update the current best value of the sailfish's population (called elite sailfish) and the current best value of sardine's population (called injured sardine) if they are smaller than the previous best values. In other words, the decision-maker agents record the high-quality solutions, which are discovered during the optimization in the shared memory. Throughout this process, the threads of this kernel accordingly update the coordinates of the best values obtained so for.

The fourth kernel updates the position of sailfish by the exploration agents that are independent from any other and act in parallel. If the exploration agents observe any improvement or insignificant improvement, the search will be stagnated and need to be diversified by updating the position of sailfish according to (3). In this kernel, the information of each block will be copied to the shared memory for decreasing the access latency and improving the performance of the algorithm.

The fifth kernel updates the position of sardine by the exploitation agents using (6). These agents cooperate with the exploration agents when a more intensified search is required during the optimization. The positions of sardines are allocated to the threads of block. Thereafter, for updating the position of sardines, the info of block transfers to the shared memory for time-consuming reduction due to the high speed of computing in the shared memory. In the last kernel, each exploration agent decides whether it needs to substitute the latest position of the injured sardine with the position of the current sailfish or it can continue its process without information exchange. If the fitness value of the injured sardine is fitter than the elite sailfish, their positions will be replaced together according to (10) and the position of the injured sardine will be removed from their population (it means that the injured sardine is hunted by the elite sailfish).

A flowchart of the DSFO implementation on CUDA is shown in figure 6. As shown in this figure, after generation of the initial populations, each swarm will be partitioned into separate sub-swarms. Then the information of sub-swarms will be transferred to the shared memory for computing the fitness value and updating the position of the search agents according to the best solution obtained so far. This process continues until a number of maximum generations is given.

5. Experimental Results and Analysis

This section presents the experimental results that have been obtained using sequential and distributed SFO algorithm. The results are compared with another previous implementation on two performance measures, including speedup and quality. These measures are evaluated on four classical benchmarks that are often used to evaluate parallel optimization algorithms as listed in table 2. In this table, F1 and F4 are unimodal functions that are suitable for benchmarking the exploitation phase because of having one global optimum and no local optima. However, F2 and F3 are multimodal functions that have a massive number of local optima and there are suitable for benchmarking the exploration phase. A successful metaheuristic algorithm should have a great ability to provide the dynamic balance between exploration and exploitation phases on a given optimization problem.

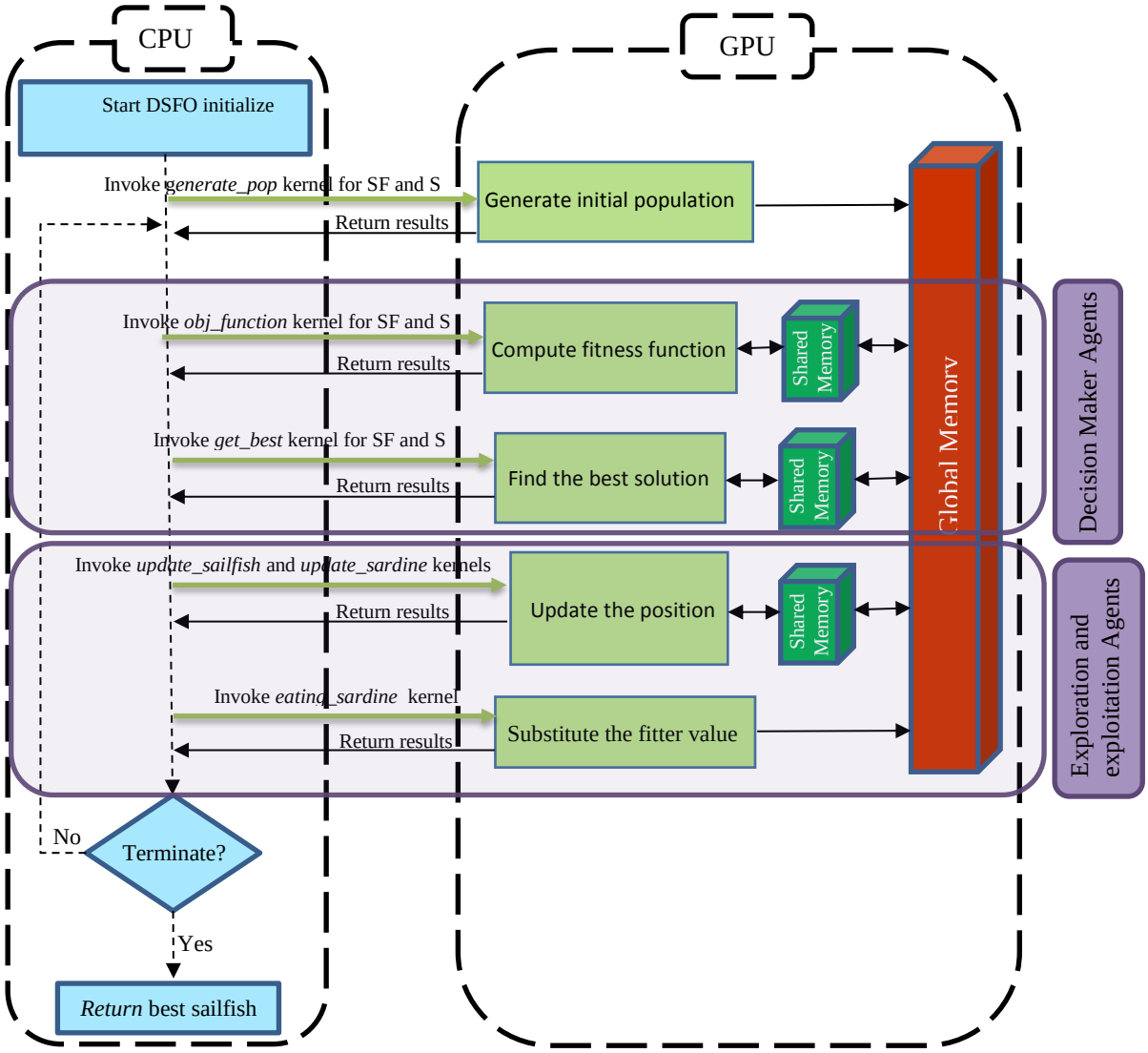

Figure 6. A flowchart of DSFO implementation on CUDA (SF is sailfish and S is sardine).

Table 2. Benchmark test functions.

	Name	Function	Characteristics	Range	Optimal
Unimodal	*sphere*	$f_1(x) = \sum_{i=1}^{n} x_i^2$	Convex, Scalable, Separable	[-100,100]	0
	Rastrigin	$F_2(x) = \sum_{i=1}^{n} [x_i^2 - 10\cos(2\pi x_i) + 10]$	Convex, Scalable, Separable	[-5.12,5.12]	0
Multimodal	Griewank	$F_3(x) = \frac{1}{4000}\sum_{i=1}^{n} x_i^2 - \prod_{i=1}^{n} \cos\left(\frac{x_i}{\sqrt{i}}\right) + 1$	Non Convex, Scalable, Non Separable	[-600,600]	0
	Rosenbrock	$f_4(x) = \sum_{i=1}^{n-1} [100(x_{i+1} - x_i^2)^2 + (x_i - 1)^2]$	Non Convex, Scalable, Non Separable	[-30,30]	0

Table 3. Comparison of optimization results between the SFO and DSFO algorithms for the four benchmark functions.

Function		SFO	DSFO
F1	Mean	6.55×10^{-13}	7.1×10^{-14}
	Std	1.11×10^{-12}	4.21×10^{-14}
F2	Mean	6.49×10^{-11}	6.49×10^{-11}
	Std	2.34×10^{-10}	5.76×10^{-11}
F3	Mean	2.31×10^{-14}	5.65×10^{-14}
	Std	6.46×10^{-14}	3.34×10^{-13}
F4	Mean	3.86×10^{-7}	1.42×10^{-6}
	Std	5.38×10^{-7}	4.04×10^{-6}

The sequential and distributed SFO (DSFO) and executed using the same number of search agents and dimensions. Also, our tests were conducted using an Intel(R) Core i7(TM) with 8 GB RAM and an NVIDIA GeForce GTX980 GPU. In order to compile the distributed version, Microsoft Visual Studio 2012 Professional Edition and CUDA 4.2 SDK was used and to compile the sequential version, MATLAB 2019 was used. The operating system was Windows 7 Professional SP1. To provide a fair comparison, every test function was solved with 30 candidate solutions on the 500 iterations and the average results were reported after 20 times of run for each experiment.

As mentioned earlier, the quality of solution is one of the most important issues to measure the performance of a distributed algorithm. The experimental results of the DSFO algorithm have proved that the quality of solutions is not sacrificed for the sake of speed-up. Table 3 presents the quality comparison of solutions for the CPU and GPU versions of the SFO algorithm. As it can be seen in this table, the DSFO algorithm evaluates the quality of the optimum points similar to the sequential version for all test problems (unimodal and multimodal functions). In fact, these results show that DSFO has been carefully explored during optimization. The reason of improving the exploration of search space and good quality of solutions is high diversity in the DSFO algorithm.

The multi-core parallel processing power of a GPU generates the high-quality random numbers using the CURAND library that provides more diversity in the population. Furthermore, in figure 7, as we expected, when the population size increased, the distributed SFO was more successful for searching the optimal point especially with high dimensions (above 32). The population size significantly affects finding the global optima, and increasing the problem dimension affects the number of warp switches in the GPU architecture.

Another metric that is used for compression between the sequential and distributed versions of algorithms is execution time. The execution time depends on the number of operators and types of function that are used inside a test function. In figure 8, the execution time of four benchmark functions are presented for a fixed number of iterations and dimensions. As shown in this figure, in the low population size of search agents such as the 30, 50 or 100 search agents, there is no significant difference between the execution time of the sequential and distributed SFO. In other words, the slope of the time-consuming curves is very steep in lower populations but gradually decreases with increase in the population size. In fact, GPU does not indicate an extraordinary performance in a small set of data. However, the execution time of GPU becomes less than the CPU version when the swarm population size is increased. The reason for this improvement is a single instruction in the GPU version works over a large block of data and all of them are applied in the same operation.

Certainly, working in blocks of data at the same time reduces the time consumption and overhead during the optimization. Moreover, the decision-maker agents, exploration agents, and exploitation agents in the DSFO algorithm can run in several processing cores simultaneously, and the execution time will be faster than using CPU definitely.

As demonstrated in figure 8, the DSFO algorithm is able to evaluate four test problems with a high population size (1000 to 8000). This ability is possible through the use of shared memory configuration and multi-agent system, and this algorithm can decompose the complex problems into simpler and smaller problems that are operated by the various agents.

We carried out another set of simulations to evaluate the execution time of DSFO compared to several algorithms such as the parallel GSA algorithm [33], GPU_sync and GPU_SPSO algorithms [34], and cuda DE_1 and FPDE algorithms [35] with different numbers of dimension and population size. Figure 9 shows the comparison of the computation time (second) of the DSFO and PGSA implementations across varying population sizes on the Rastrigin function of different dimensions (64, 128, and 256).

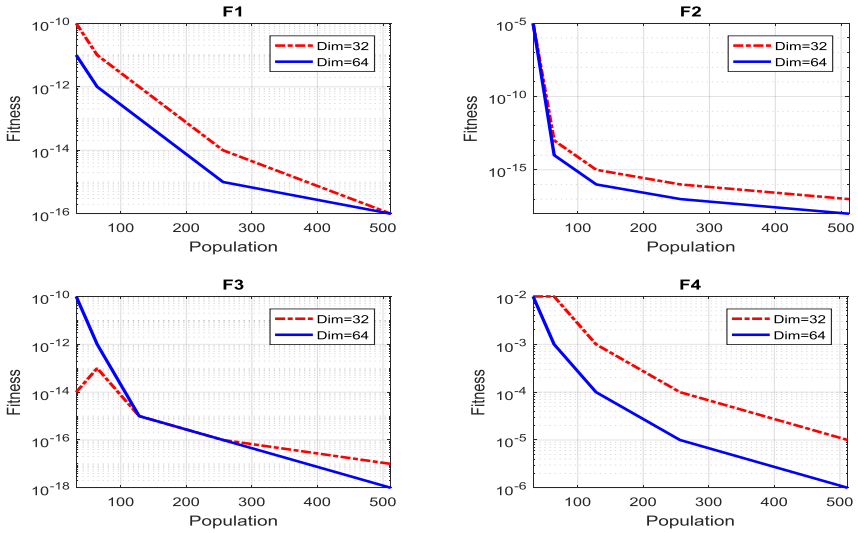

Figure 7. Fitness variation with population size for different dimensions.

We carried out another set of simulations to evaluate the execution time of DSFO compared to several algorithms such as the parallel GSA algorithm [33], GPU_sync and GPU_SPSO algorithms [34], and cuda DE_1 and FPDE algorithms [35] with different numbers of dimension and population size. Figure 9 shows the comparison of the computation time (second) of the DSFO and PGSA implementations across varying population sizes on the Rastrigin function of different dimensions (64, 128, and 256). Due to the limitation of the number of threads per block, data transfer between the shared memory and the global memory will increase, especially with increasing dimensions of the problems so it will increase the time consumed during optimization. However, as it can be seen in figure 9, the execution time of the DSFO algorithm is less than parallel GSA under the same number of iterations and it shows that DSFO is extremely effective at the unimodal and scalable functions according to table 2.

Another comparison illustrated in figure 10 is between the DSFO algorithm and the other two algorithms, i.e. GPU_sync and GPU_SPSO, that are parallel versions of the SPSO algorithm.

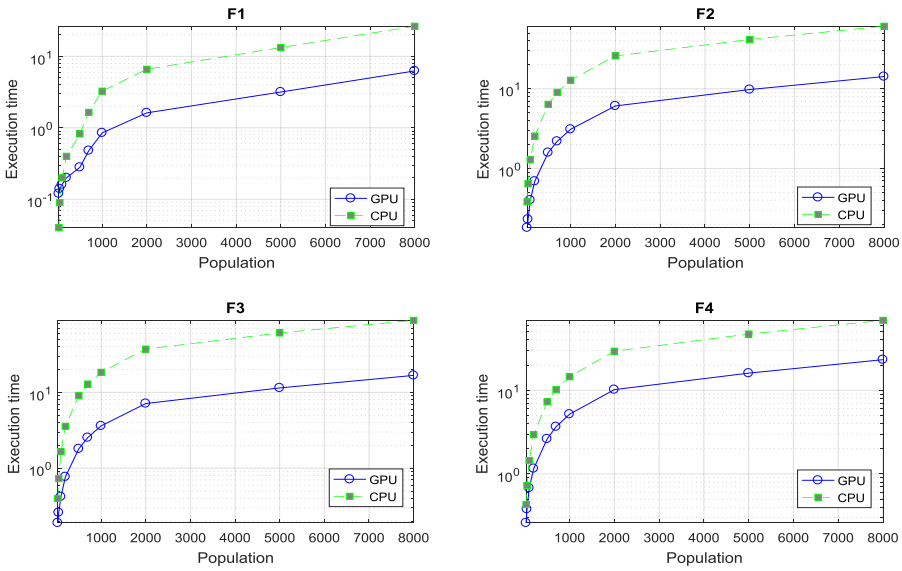

Figure 8. Execution time for distributed and sequential SFO implementation (iteration = 500 and dimension = 30).

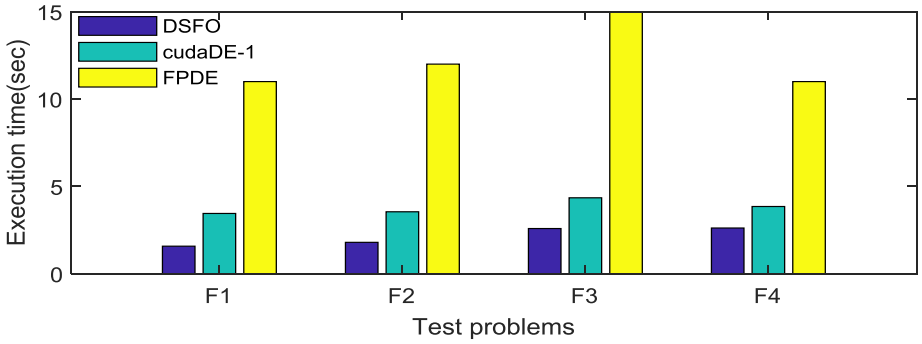

Figure 11. A comparison between DSFO and two other algorithms for four benchmark functions.

From figure 10, it is clear that the DSFO algorithm takes less execution time than the others. Also from this evaluation, it appears that the DSFO algorithm has a great ability for solving the computing arithmetic intensive functions like the Rosenbrock function in a short time.

Moreover, figure 11 shows that DSFO is very competitive compared with cuda DE_1 and FPDE. These results obtained indicate that the DSFO algorithm is suitable for the non-convex, scalable, and non-separable optimization problems. These types of problems are not relatively easy to solve and optimize because they have multiple local optimal points, and it takes a lot of time to find the global optima or sticks in the local optima.

Also some of these functions cannot divide into sub-objective functions, and this will make it more difficult to solve the functions during optimization. Another property of the test problems is scalability. This ability responds well when the dimension of the search space increases. In the previous evaluations, several multi-dimensional scalable test functions were evaluated and the results showed that the DSFO algorithm had a great ability to optimize the scalable problems as well. Overall, the experimental results show that the DSFO algorithm outperforms the other algorithms with a less time-consuming optimization. Such an improvement in the speed-up promises to solve the optimization problems with a higher speed.

5. Conclusions

This paper presented a multi-agent based distributed SFO algorithm (DSFO). Also the implementation of this algorithm on GPU using the CUDA architecture was presented. The practical implications of DSFO suggest a multiple kernels solution, and each dimension is mapped onto a distinct thread, and the blocks are employed for saving the position of the search agents. This strategy distributes the computational load by the search agents to calculate and determine the

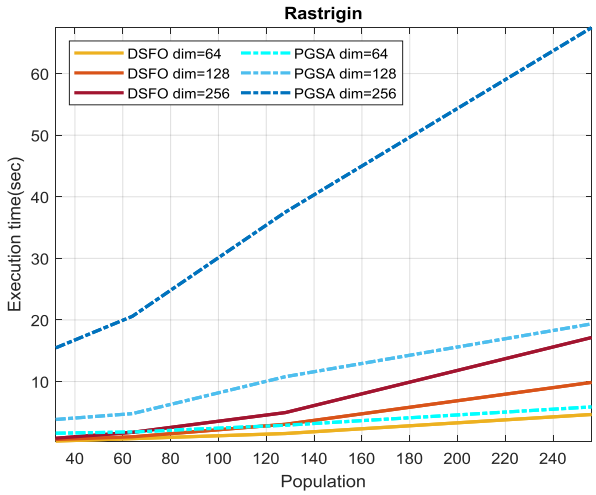

Figure 9. A comparison between distributed SFO (DSFO) and parallel GSA (PGSA) on Rastrigin function.

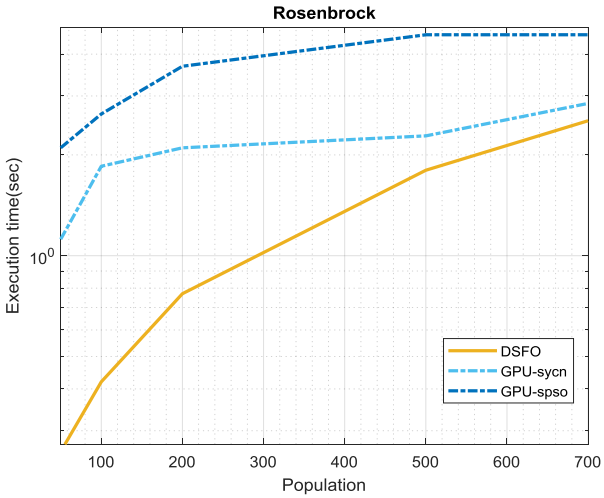

Figure 10. A comparison between DSFO and two other algorithms on the Rosenbrock function.

optimal point during optimization. It exploits all the available SMs and reduces the global memory access delay using the data shared memory among different kernels. The results obtained showed the reduced execution time of DSFO implementation

compared to the original sequential SFO implementation and other algorithms, while the optimizing quality was the same or even better.
Our future work will present on ameliorating distributed SFO by providing an approach using reconfigurable hardware for increasing the speed of optimization and improving the quality of the solutions.

References

[1] C. Blum *et al.*, "Hybrid metaheuristics in combinatorial optimization: A survey" *Applied Soft Computing*, vol.11, no. 6, pp. 4135 – 4151, 2011.

[2] I. Boussaid *et al.*, "A survey on optimization metaheuristics" *Information Sciences*, vol. 237, pp. 82–117, 2013.

[3] M.B. Ayhan *et al.*, "A multi-agent based approach for change management in manufacturing enterprises" *Journal of Intelligent Manufacturing*, vol. 26, no. 5, pp. 975-988, 2015.

[4] M.A. Hale and J. Craig, "Preliminary development of agent technologies for a design integration framework" *Proc. 5th Symp. Multidisciplinary Analysis and Optimization, Panama City, FL,* 1994.

[5] N. Jennings and M. Wooldridge, "Intelligent Agents: Theory and Practice" *The Knowledge Eng. Rev*, vol. 10, no. 2, pp. 115–152, 1995.

[6] J. M Vidal *et al.*, "Inside an Agent" *IEEE Internet Computing*, 2001.

[7] S. Shadravan *et al*., "The Sailfish Optimizer: A novel nature-inspired metaheuristic algorithm for solving constrained engineering optimization problem" *Engineering Applications of Artificial Intelligence*, vol. 88, pp. 20–34, 2019.

[8] M. E. Aydin, "Meta-heuristic agent teams for job shop scheduling problems" *Lecture Notes in Artificial Intelligence*, vol. 4659, pp. 185-194, 2007.

[9] M. Hammami and K. Ghediera, "COSATS, X-COSATS: Two multi-agent systems cooperating simulated annealing, tabu search and X-over operator for the K-Graph Partitioning problem" *Lecture Notes in Computer Science*, vol. 3684, pp. 647-653, 2005.

[10] S. Talukdar *et al.*, "Asynchronous teams: Cooperation schemes for autonomous agents*" Journal of Heuristics*, vol. 4, no. 4, pp. 295–321, 1998.

[11] S. Talukdar, S. Murthy and R. Akkiraju "Asynchronous teams. In Handbook of Metaheuristics, ser. International Series in Operations Research & Management Science" *Springer US*, vol. 57, pp. 537–556, 2013.

[12] Maria Amélia Lopes Silva *et al*. "A Multiagent Metaheuristic Optimization Framework with Cooperation" *Brazilian Conference on Intelligent Systems IEEE*, pp. 104-109, 2015.

[13] H. R. Naji, M. Sohrabi, and E. Rashedi, "A High Speed and Performance Optimization Algorithm Based on Gravitational Approach" *IEEE Journal of Computing in Science and Engineering*, vol. 14, no. 5, pp. 56-62, 2013.

[14] H. R. Naji, "Solving Large Computational Problems using Multi-Agents Implemented in Hardware" *Computing in Science and Engineering*, IEEE CS and American Institute of Physics, vol. 10, no. 5, pp. 54-63, 2008.

[15] H.R. Naji and B.E. Wells, " On incorporating multi-agents in combined hardware/software based reconfigurable systems, a general architectural framework" *Symposium on System Theory, Huntsville, AL,* 2002.

[16] G. Binetti *et al*., "Distributed consensus-based economic dispatch with transmission losses" *IEEE Trans. Power Syst*., vol. 29, no. 4, pp. 1711–1720, 2014.

[17] Z. Qiu, S. Liu and L. Xie, " Distributed constrained optimal consensus of multi-agent systems" *Automatica*, vol. 68, pp. 209–215, 2016.

[18] R. Carli *et al*., "Analysis of newton-raphson consensus for multi-agent convex optimization under asynchronous and lossy communications" *in IEEE 54th Annual Conference on Decision and Control (CDC). IEEE*, pp. 418–424, 2015.

[19] H. Zhang *et al*., "Adaptive consensus-based distributed target tracking with dynamic cluster in sensor network" *IEEE Trans. Cybern*., vol. 49, no. 5, pp. 1580–1591, 2019.

[20] R. Yarinezhad and A, Sarabi, "A New Routing Algorithm for Vehicular Ad-hoc Networks based on Glowworm Swarm Optimization Algorithm" *Journal of AI and Data Mining*, vol. 7, no. 1, pp. 69-76, 2019.

[21] Sh. Lotfi and F. Karimi, "A Hybrid MOEA/D-TS for Solving Multi-Objective Problems" *Journal of AI and Data Mining*, vol. 5, no. 2, pp. 183-195, 2017.

[22] M. Essaid *et al.*, " GPU parallelization strategies for metaheuristics: a survey" *International Journal of Parallel, Emergent and Distributed Systems*, vol. 34, no. 5, pp. 497-522, 2018.

[23] P. Krömer, J. Platoš and V. Snášel. "Nature-inspired meta-heuristics on modern GPUs: state of the art and brief survey of selected algorithms" *Int J Parallel Program*, vol. 42, no. 5, pp. 681–709, 2014.

[24] E. Alba, G. Luque and S. Nesmachnow, " Parallel metaheuristics: recent advances and new trends" *International Trans Oper Res*, vol. 20, no. 1, pp. 1–48, 2013.

[25] Z. Yang, Y. Zhu and Y. Pu, " Parallel image processing based on CUDA" *In: 2008 International Conference on Computer Science and Software Engineering*, pp. 198–201, 2008.

[26] W. Fang *et al.*, "Parallel data mining on graphics processors" *Technical Report HKUST-CS08-07*. Hong

Kong, China: Hong Kong University of Science and Technology, 2008.

[27] Z.W. Luo *et al.*, "Artificial Neural Network Computation on Graphic Process Unit" *IEEE International Joint Conference on Neural Networks*, pp. 622–626, 2005.

[28] R.A. Patel *et al*., "Parallel lossless data compression on the GP" *San Jose (CA): IEEE*, 2012.

[29] A. Brodtkorb *et al.*, "State-of-the-art in heterogeneous computing" *Sci. Program*, vol. 18, no. 1, pp. 1-33, 2012.

[30] NVIDIA, NVIDIA CUDA Programming version 6.0, 2014.

[31] DB. Kirk, WH. Wen-Mei, "Programming massively parallel processors: a hands-on approach" *Morgan kaufmann*, 2016.

[32] NVIDIA: CURAND Library 7.5., 2015. http://docs.nvidia.com/cuda/pdf/CURAND Library.pdf.

[33] A. Zarrabi *et al*., "Gravitational search algorithm using CUDA: a case study in high-performance metaheuristics" *Springer Science+Business Media New York*, vol. 71, no. 4, pp. 1277-1296, 2014.

[34] R.V. Krishna and S.S. Reddy, "Performance Evaluation of Particle Swarm Optimization Algorithms on GPU using CUDA" *I J C S S E I T*, vol. 5, no. 1, pp. 65-81, 2012.

[35] A.K. Qin *et al*., "An Improved CUDA-Based Implementation of Differential Evolution on GPU" *ACM New York, NY, USA*, pp. 991-998, 2012.

الگوریتم توزیع شده بادبان ماهی مبتنی بر سیستم های چند عامله جهت حل توابع غیر محدب و مقیاس پذیر و پیاده سازی آن توسط پردازشگرهای گرافیکی

سوده شادروان [۱]، حمید رضا ناجی [۲،*] و وحید خطیبی [۳]

[۱] دانشکده علوم کامپیوتر، دانشگاه آزاد اسلامی واحد کرمان ، کرمان، ایران.

[۲] دانشکده کامپیوتر، دانشگاه تحصیلات تکمیلی صنعتی و فناوری پیشرفته، کرمان، ایران.

[۳]دانشکده علوم کامپیوتر، دانشگاه آزاد اسلامی واحد بردسیر ، بردسیر، ایران.

ارسال ۲۰۲۰/۰۲/۱۸؛ بازنگری ۲۰۲۰/۰۶/۰۶؛ پذیرش ۲۰۲۰/۰۷/۲۴

چکیده:

الگوریتم بادبان ماهی یک الگوریتم فراابتکاری است که با الهام از شکار گروهی بادبان ماهی ها و از طریق حملات متناوب آنها به دسته ماهی های کوچکتر ابداع شده است. الگوریتم SFO با استفاده از یک روش ساده مزایایی همانند فراهم کردن تعادل پویا بین مراحل اکتشاف و بهره برداری، ایجاد تنوع در تولید جمعیت اولیه، جلوگیری از افتادن در دام بهینه محلی و تضمین همگرایی با سرعت بالا را تضمین نموده ست. امروزه ، سیستم های مبتنی بر چند عامل و الگوریتم های فراابتکاری برای حل مسائل بهینه سازی ترکیبی راه حل هایی با عملکرد بالا ارائه نموده اند . این روش ها توانسته اند با ادغام با الگوریتم های موازی سازی شده گزینه مناسبی برای کاهش زمان محاسباتی و افزایش کیفیت راه حل های محاسباتی محسوب گردند. در این مقاله، ما یک روش جدید مبتنی بر سیستم های چند عامله و یک روش توزیع شده از الگوریتم بادبان ماهی (DSFO) که باعث بهبود زمان اجرا و سرعت به همراه حفظ کیفیت راه حل های حل مسئله می باشد را ارائه داده ایم. در ادامه این مقاله، جزئیات پیاده سازی و ارزیابی عملکرد الگوریتم DSFO ارائه شده که نشان دهنده یک مطالعه مقایسه ای از الگوریتم بادبان ماهی توزیع شده بر روی مجموعه ای از توابع بهینه سازی محک استاندارد می باشد و همچنین با مقایسه آن با سایر الگوریتم های موازی سرعت بالای الگوریتم پیشنهادی برای حل مسائل بهینه سازی نامحدود به نمایش گذاشته است. نتایج ارائه شده حاکی از توانایی بالای این الگوریتم در حل مسائل بهینه سازی پیوسته، تجزیه ناپذیر، غیر محدب و مقیاس پذیر می باشد که همگی جزو مسائل دشوار بهینه سازی معرفی شده اند.

کلمات کلیدی: بهینه ساز بادبان ماهی، الگوریتم توزیع شده بادبان ماهی، سیستم‌های چند عامله، پردازش موازی، حافظه اشتراکی، پردازشگر گرافیکی موازی.