ISSN 2224-087X. Електроніка та інформаційні технології. 2024. Випуск 25. С. 30–41
Electronics and information technologies. 2024. Issue 25. P. 30–41

UDC 004.4 DOI: https://doi.org/10.30970/eli.25.3

STUDY OF PARALLEL MODEL OF ARTIFICIAL BEE COLONY ALGORITHM

O. Sinkevych, V. Boyko, L. Monastyrsky, B. Sokolovsky

Radioelectronic and Computer Systems Department,
Ivan Franko National University of Lviv,
50 Drahomanova St., UA–79005 Lviv, Ukraine
oleh.sinkevych@lnu.edu.ua

The paper presents the exploration of artificial bee colony optimization algorithm for multi-dimension functions which is implemented in Python 3. Due to increase of the searching complexity, the high dimensional applied problems require development of the concurrent computing approaches which involve either parallel or asynchronous programming. The latter causes growth of general runtime and becomes bottleneck of single-processing implementation. In order to deal with the raised issue, we propose the parallelization algorithms based on multiprocessing library. The developed parallel approaches have shown the significant increase of the performance and allow taking into account multi-dimensional character of optimization problems. Despite the limitation of Python 3 multi-threading capabilities and the computational cost of creating execution processes, the proposed and explored approaches have demonstrated their efficiency for a number of benchmark functions.

Keywords: artificial bee colony; meta-heuristics; swarm intelligence; numerical optimization; multi-processing.

Introduction. Evolutionary computations, swarm intelligence and other meta-heuristics have earned recognition as simple and at the same time effective family of algorithms for solving complex and combinatorial problems [1]. The most of them are based on a biological evolution principle, where each generation (candidate solution) is developing during the outer conditions and interaction with each other. Different kinds of reproduction, mutation, recombination, and selection define the processes of their computational behavior [2].

Among the vast group of evolutionary methods, nature-inspired meta-heuristics are of considerable interest, in particular, because of solid performance in many of engineering problems [3]. In general, the core principle of such approaches is the reproduction of behavioral patterns of collective living beings aimed at solving everyday problems. For instance, artificial bee colony (ABC) [4] mimics the process of seeking good food sources which is applied to diverse optimization routines; artificial ant colony (AAC) that simulates pheromone communication [5] is usually used in routing tasks; gray wolf optimization which is based on complex social hierarchy (GWO) [6] can be applied to feature selection in machine learning pipelines.

The major advantages of the mentioned techniques are good scalability to the higher dimensional problems, ease of parallelism, robustness to noise and flexibility of customization [7]. Despite the important drawback, which consists in significant computational costs, namely in a large number of calculations of the fitness function, swarm meta-heuristics can be a suitable choice, if the possibility of parallelization of calculations is available. For example, in [8] parallel ABC algorithm which utilizes MPI programming environment has been proposed and studied; in [9] authors provide brief overview of GPU-based meta-heuristics; in [10] the implementation of high-level parallel AAC is described and explored; in [11] a comprehensive survey on parallel particle swarm optimization (PSO) algorithms is presented along with their parallelization strategies and applications.

In this article, an attempt is made to analyze the results of the implementation of ABC algorithm, both in single process mode and in their multi-process version. Typically, the application of ABC algorithm for real-world problems require a huge number of the computational agents, hence the parallelization techniques are preferable to speedup general performance. There are a bunch of studies dedicated to parallelization of ABC: in [12] author propose the parallel ABC based on colony division and mutation mechanics during the employed bee phase; in [13] asynchronous ABC was examined; in [14] the traditional emigrant selection strategy was replaced with a novel cooperative model, which was then examined using the widely adopted ring neighborhood topology. Interesting modification was proposed in [15] where author suggest using swapping mechanism to improve migration and solution sharing between sub-colonies.

The motivation of this study is determined by the prospect of parallel application of this algorithm in the tasks of neural network fine-tuning and its deployment within the framework of a multi-agent system. Multi-process implementation is based on Python programming language and multiprocessing library [16]; this choice is justified by the prevalence of the Python stack in machine learning and allows the integration of the ABC algorithm into existing deep learning frameworks. To investigate the implementations, we considered the minimization problems of seven benchmark functions with variable dimensions.

Sequential ABC algorithm. The first appearance of the algorithm dates back to 2005, when D. Karaboga proposed the use of honey bee swarm concept for solving optimization problems [17]. Since that time, numerous variants and modifications of ABC have been widely studied. The reason of this is the decent effectiveness in solving many applied and engineering problems. Like many other swarm approaches, in the ABC agents (bees) are trying to find the best solution in predefined region being communicating, sharing information and performing specific roles.

Let's consider the typical unconstrained minimization problem:

$$\boldsymbol{x}^* = \arg\min f\left(\boldsymbol{x}\right),\ \ \boldsymbol{x} \in \boldsymbol{R}^D, \tag{1}$$

where $\boldsymbol{x}$ is the D-dimensional vector, f is the objective function and $\boldsymbol{x}^*$ is the global minimum.

The ABC algorithm aims to solve this problem by performing several simple steps, which are determined by three search phases: employed bees, onlooker bees and scout bees. Before these phases start, it is necessary to set up the initial state of the routine. Firstly, SN vectors (candidate solutions) are generated as follows

$$x_{i,j} = x_j^{lb} + \text{rand}\left(0,1\right)\left(x_j^{ub} - x_j^{lb}\right), \tag{2}$$

where $i \in [1, SN]$, SN is the parameter of ABC which specifies the number of searching bees, $x_{i,j}$ is the *j*-th component of the vector $\boldsymbol{x}_i$, x_j^{lb} and x_j^{ub} are the lower and upper bounds of each component, $\mathrm{rand}(0,1)$ returns the random value in range [0, 1]. Number of bees is usually set equal to candidate solutions.

After the initialization is done, *employed bees* phase starts. During this step each employed bee generates the *i*-th solution using the next formulae:

$$v_{i,j} = x_{i,j} + \phi\left(x_{i,j} - x_{k,j}\right), \tag{3}$$

where ϕ is the random value from uniformly distributed range in [-1, 1], $x_{k,j}$ is the neighbor of $x_{i,j}$. For each vector $\boldsymbol{v}$, fitness function is calculated as:

$$fit(\boldsymbol{v}_i) = \left[1 + \left|f(\boldsymbol{v}_i)\right| : f(\boldsymbol{v}_i) \geq 0 \ or \ 1 \big/ \left(1 + f(\boldsymbol{v}_i)\right) : f(\boldsymbol{v}_i) < 0\right]. \tag{4}$$

Depends on its value, i.e., if $fit(\boldsymbol{v}_i) > fit(\boldsymbol{x}_i)$, then a new candidate solution $\boldsymbol{v}_i$ replaces previous $\boldsymbol{x}_i$. If the candidate solution has not been improved, it remains the same.

The *onlooker bees* phase helps to investigate the new solutions in the neighborhood of each $\boldsymbol{x}_i$ after the employed phase has been performed. It starts with the probabilities calculation via

$$p(\boldsymbol{x}_i) = fit(\boldsymbol{x}_i) \Big/ \sum_{n=1}^{SN} fit(\boldsymbol{x}_n). \tag{5}$$

As soon as probabilities have been obtained, each bee i, $i \in [1, SN]$ works on improving the solution in the neighborhood based on the condition $rand(0,1) < p(\boldsymbol{x}_i)$. If it is satisfied, then bee uses (3) and (4) to update the solution. If not, then the next $i+1$ bee makes the same step until all bees meet the condition $\mathrm{rand}(0,1) < p(\boldsymbol{x}_i)$. The purpose of this step is to enhance exploration process.

The last phase is *scout bees*. At the beginning, a trial counter is assigned to each solution $\boldsymbol{x}_i$. It signifies how many times that solution was not improved via employed bees and onlooker bees phase. Each scout bee checks corresponding trial counter and re-initialize solution via (2), if the trials reach predefined limit. It is recommended to set the limit value as $D*SN$ or $D*SN//2$.

To sum up, one iteration of the sequential ABC algorithm works as follows:

- solutions and bees initialization;
- employed bee phase;
- calculating the probabilities via (5);
- onlooker bees phase;
- scout bees phase.

The proposed Python 3 sequential implementation consists of two separate modules. The first one (fig. 1, 2) holds **Agent** class which represents the bee object and **UnconstrainedSwarm** class with the implementation of the algorithm. The second one consists of the main ABC phases implemented as the separate functions (fig. 3). Each of them implements the respective behavior as well as assures modularity and code flexibility.

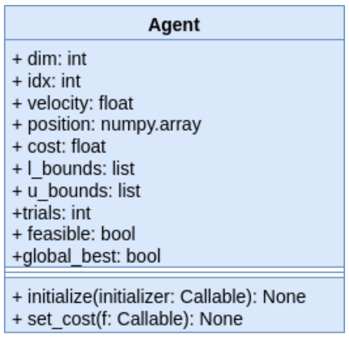

Fig. 1. UML diagram of Agent class

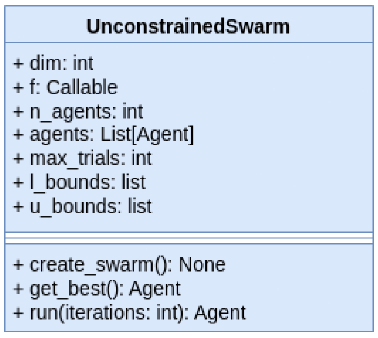

Fig. 2. UML diagram of UnconstrainedSwarm class

To increase the performance, all array data were implemented using Numpy package, which allows vectorizing computations.

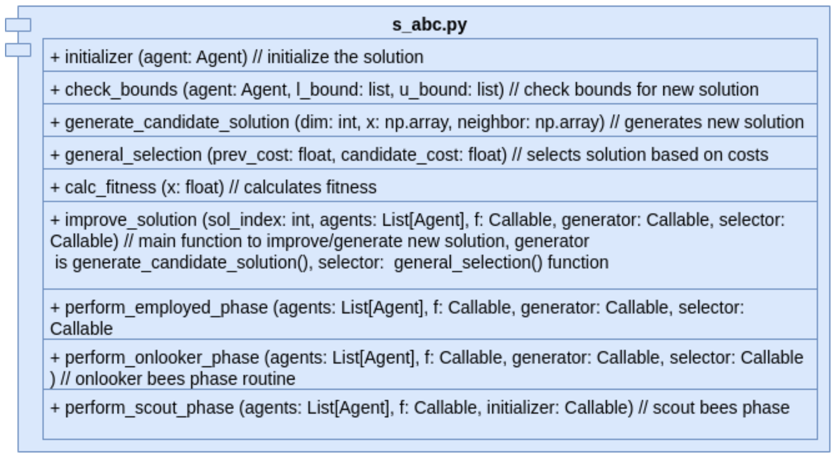
s_abc.py
+ initializer (agent: Agent) // initialize the solution
+ check_bounds (agent: Agent, l_bound: list, u_bound: list) // check bounds for new solution
+ generate_candidate_solution (dim: int, x: np.array, neighbor: np.array) // generates new solution
+ general_selection (prev_cost: float, candidate_cost: float) // selects solution based on costs
+ calc_fitness (x: float) // calculates fitness
+ improve_solution (sol_index: int, agents: List[Agent], f: Callable, generator: Callable, selector: Callable) // main function to improve/generate new solution, generator is generate_candidate_solution(), selector: general_selection() function
+ perform_employed_phase (agents: List[Agent], f: Callable, generator: Callable, selector: Callable
+ perform_onlooker_phase (agents: List[Agent], f: Callable, generator: Callable, selector: Callable) // onlooker bees phase routine
+ perform_scout_phase (agents: List[Agent], f: Callable, initializer: Callable) // scout bees phase

Fig. 3. UML module diagram for main ABC implementation

Parallel ABC algorithm. Regardless of numerous tasks, where the application of a sequential single-process/thread ABC algorithm demonstrates high efficiency, complex and multidimensional real-world problems require the parallelization of algorithm to speed up calculations. For instance, the industry routing or mechanical engineering tasks require a huge amount of computations which are time consuming and inefficient in the case of one core/thread implementation. With this in mind, it is necessary to develop, to research and to improve parallel ABC algorithms to speed up performance.

Since the development of concurrent algorithms quite often requires deep modification of their sequential counterparts, such a task can be outlined by the following statements:

• parallel or asynchronous implementation performance should not be worse than sequential;

• with the increase of complexity, parallel implementations should gain more and more effectiveness compared to the sequential approaches.

The most common idea for the parallelization of the ABC algorithm is to divide the colony of bees into a set of sub-colonies that correspond to a separate process or thread [19]. Hence, each colony works on improving the solution separately from others. After the parallel (or in some cases asynchronous) work has been completed, sub-colonies communicate with each other and exchange information about their local solutions [20]. Actually, the details of the last stage are the most complex and important when designing this kind of algorithms.

To set up the starting point of the study, in this work we consider a type of parallel ABC implementation, namely where force max or fmABC technique. Since the standard Python 3 (CPython implementation) is, generally speaking, limited in the sense of multithreading due to GIL blocking, we used the multi-processing module for the development [21].

We propose the simple parallel fmABC algorithm (fig. 4) which works as follows:

1) the whole array of solution SN is equally divided into sub-solutions (sub-colonies) and is assigned to separate process which holds isolated **UnconstrainedSwarm** and corresponding **Agent** objects;
2) inside each allocated process, sequential ABC performs default steps and produces recurring local results $\boldsymbol{X} = \boldsymbol{x}_i$, $1 = 1..SN/n\_proc$, where n_proc is number of machine processors, best solution $\boldsymbol{xBest}$ and fitness $\boldsymbol{fitBest}$ values;
3) after all processes complete their job, the best solution $\boldsymbol{xBest}^{*} = \arg\max\left(\boldsymbol{fitBest} \mid \boldsymbol{xBest}\right)$ among all processes is determined;
4) the best solutions in each of i-th **UnconstrainedSwarm** is replaced by $\boldsymbol{xBest}^{*}$.

Calculations are performed for a predetermined number of iterations, or until the condition regarding the accuracy of calculated solution is met. During the operations, the best solution is forced among all sub-colonies, which causes selection pressure on them. This makes sub-colony to reinforce a better solution. The disadvantages of this approach are spread of the same solution vector across all sub-colonies and the reduction of their diversity. However, as the numerical experiments in the next section will show, this technique demonstrates good efficiency in comparison with the sequential algorithm.

The core action part inside both implementations is sequential ABC which can be either modified to improve the performance. There are a lot of modifications which increase execution time and convergence to global optima; nevertheless it is hard to find one optimal version for all problems. Therefore, in this paper we consider the canonical version of the algorithm.

Each sub-colony that performs the standard steps of the ABC algorithm is a separate object of the **Swarm** class (method ***run()***), which is inherited from the **multiprocessing. Process** class and **UnconstrainedSwarm**. As soon as all processes complete their internal operations, the local **Swarm** objects are extracted from the corresponding processes through the **Queue** object, which allows processes to communicate with each other.

Python parallelization through the multiprocessing library have the drawbacks, which consist of the "costliness" of creating individual processes and a slowing factor such as the use of a queue object. In contrast to full-fledged multithreading, as in the C or C++ language, where threads have free access to a common object in memory, multiprocessing requires the creation of a communication mechanism between independent processes, which slows down the work and introduces additional computational complexity.

```
Algorithm 1 fmABC
  Input: $max\_cycles, n\_proc, f, SN$
  for $i = 1..n\_proc$ do
    Init. $Swarms = [Swarm_i(f, SN/n\_proc)]$ for each sub-swarm
  end for
  for $j = 1..max\_cycles$ do
    for $k = 1..n\_proc$ do
      $Swarms[k] \leftarrow ABC(Swarms[k])$
    end for
    $best\_solution = argmax(Swarm.best\_fitness)$
    $best\_idx = Swarms.best\_sol\_idx$
    for $k = 1..n\_proc$ do
      if $best\_idx! = k$ then
        $Swarms[k].best\_sol = best\_sol$
      end if
    end for
  end for
```

Fig. 4. Parallel ABC algorithm

Experiments and results. To carry out numerical experiments and to compare single process and multi-process implementation of the ABC algorithm we have considered seven benchmark functions [22], defined in Table 1.

Table 1. Benchmark functions

f	**Range**	**Formulae**
Sphere	[-100; 100]	$f(x) = \sum_{i=1}^{d} x_i^2$
Step	[-100; 100]	$f(x) = \sum_{i=1}^{d} (x_i + 0.5)^2$
Rastrigin	[-5.12; 5.12]	$f(x) = \sum_{i=1}^{d} (x_i^2 - 10\cos(2\pi x_i) + 10)$
Schwefel	[-500; 500]	$f(x) = -\sum_{i=1}^{d} (x_i \sin(\sqrt{\lvert x_i \rvert})) + 418.9829d$
Griewank	[-10; 10]	$f(x) = 1 + \frac{1}{4000}\sum_{i=1}^{d} x_i^2 - \prod_{i=1}^{d} \cos(\frac{x_i}{\sqrt{i}})$
Rosenbrock	[-10; 10]	$f(x) = \sum_{i=1}^{d-1} (100(x_{i+1} - x_i)^2) + (x_i - 1)^2)$
Ackley	[-32; 32]	$f(x) = -20\exp(-0.2\sqrt{\frac{1}{d}\sum_{i=1}^{d} x_i^2}) - \exp(\frac{1}{d}\sum_{i=1}^{d} \cos(2\pi x_i)) + 20 + e$

The first two are relatively simple unimodal functions with one global minimum. Rastrigin function is non-convex function with a large number of local minima and global minima in $x_i = 0$, $i = 1..D$. The hyper-surface of the Schwefel function has a rather complex shape with many local minimum and a global minimum at point $x_i = 420.9687$, $i = 1..D$, which unlike the previous ones, is shifted relative to the origin. Griewank function has a lot of widespread regularly distributed local minimums. In our opinion, as demonstrated by the following results, the Rosenbrock function is the most complex among the given seven functions because it has a rather narrow valley of complex curvature around the global minimum. The hyper-surface of the Ackley function is quite complex with one global minimum $x_i = 0$, $i = 1..D$. Almost all their minimization procedures are not trivial for classical gradient based algorithm.

Sequential ABC results. We have conducted a bunch of numerical experiments with the sequential ABC algorithm based on the following parameters:

- 50D functions: SN=60, max_cycles=2000, accuracy_eps = 1e-2, trials=d ∗ SN//2
- 100D functions: SN=60, max_cycles=10000, accuracy_eps = 1e-2, trials=d ∗ SN//2
- 200D functions: SN=60, max_cycles=14000, accuracy_eps = 1e-2, trials=d ∗ SN//2.

For each function and for each dimension we made ten experiments. The aforementioned hyper-parameters of the ABC algorithm were selected based on the analysis of relevant publications. All experiments conducted on 12 cores 12th Gen Intel(R) Core(TM) i5-1235U with 24 GB DDR4 RAM.

The results of experiments are depicted in Table 2-4. Here we point the best and worst computation times over ten runs of the program, as well as the smallest and largest number of iterations spent on the search.

Table 2. 50D functions ABC results

f	**Iterations**		**Time (sec.)**	
	Best	**Worst**	**Best**	**Worst**
Sphere	602	710	1.5	1.8
Step	603	710	1.5	2.1
Rastrigin	1380	2123	4.2	6.5
Schwefel	1907	2880	5.0	7.5
Griewank	355	472	1.3	1.7
Rosenbrock	1856	4893	10.3	17
Ackley	875	1037	3.6	4.2

Table 3. 100D functions ABC results

f	**Iterations**		**Time (sec.)**	
	Best	**Worst**	**Best**	**Worst**
Sphere	846	1015	2.4	3.0
Step	884	1002	2.6	2.9
Rastrigin	3076	4441	9.4	14
Schwefel	5201	7481	15.6	19.5
Griewank	718	978	2.8	3.9
Rosenbrock	5916	8145	22	30
Ackley	1937	2239	9.0	10.1

The application of the ABC algorithm to the minimization of 50-dimensional functions demonstrates a decent solution efficiency and relatively small execution time. Rosenbrock and Schwefel functions which are the most complicated in this list predictably require more time and iterations to be minimized.

Table 4. 200D functions ABC results

f	Iterations		Time (sec.)	
	Best	**Worst**	**Best**	**Worst**
Sphere	1894	2075	5.7	6.3
Step	1867	2237	5.8	6.3
Rastrigin	7253	9260	25	31
Schwefel	12649	13974	35	39
Griewank	1706	1818	8.0	8.5
Rosenbrock	10063	10063	84	84
Ackley	4099	4487	22	26

Compared with 100-dimensional minimization, for 200-dimensional functions the ABC algorithm is performed slower due to increase of solution space. Also, the most challenging Rosenbrock functions was optimized only once in ten times. The approximate slowness rate which is estimated as ratio between averaged the most quick and the most slow optimization times lies in range from 2.4 to 3 times against 100d functions depending on function's search complexity. Also, the number of iterations increases at least in 2 times. It indicates a need for the development of methods to accelerate the execution of calculations.

The high-dimensional 200d problems require a significant increase in calculation time compared to 50d or 100d problems, especially for the most complex functions. The complexity ratio r = ***MostSimpleFunc_execution_time/Rosenbrock_execution_time*** for each dimension equals $r_{50d} = 0.14$, $r_{100d} = 0.1$, $r_{200d} = 0.07$ and decreases approximately by 0.3-0.4 with the increase of problem dimension. This ratio may vary depending on the parameters of the algorithm, namely the number of bees and the trial parameter. An important problem in such calculations is the correct selection of hyper-parameters, which can be carried out using a meta-optimizer.

Parallel ABC results. The results of the single-process sequential implementation of the ABC algorithm clearly indicate the need for parallelization of calculations. One possible solution based on fig. 4 is outlined in this section.

To test fmABC algorithm we have chosen 200d problems and have set the following parameters: $n_{proc} = \{4,6\}$, $SN_{sub\_colony} = 60/n\_proc$, $\max\_cycles = 12000$. Here the same number of SN which has been used in sequential algorithm for the given dimension is divided among n_proc processes. That is, the number of agents involved in the work in the single-process version in this case is divided by executor processes that implement the work of the sequential algorithm independently of each other. Each process executes 10 inner iterations of ABC before the exchange with the best solution is done.

The results of experiments conducted for 4 and 6 cores are presented in Tables 5, 6. The proposed scheme of parallelization (fig. 4) reduces either the number of iterations and total execution time for both 4 and 6 cores spanning. Despite the computing load, which involves the creation of processes by the operating system, the developed parallelization scheme shows a significant increase in execution time.

Table 5. 200D functions parallel ABC results (4 cores)

f	**Iterations**		**Time (sec.)**	
	Best	**Worst**	**Best**	**Worst**
Sphere	187	197	4.2	4.8
Step	181	194	4.4	4.7
Rastrigin	358	390	9.2	10.3
Schwefel	704	878	15.7	19.9
Griewank	113	183	3.7	5.9
Rosenbrock	972	11691	30	350
Ackley	310	322	10.3	11.1

The average execution time (4 cores) except Rosenbrock function in case of 200d sequential runs equals 18.2 seconds. The parallel version for 200d problems requires in average 8.7 seconds and in fact is more than twice faster in comparison with the sequential implementation. Rosenbrock function was correctly optimized during the each of ten experiments, whereas the sequential approach was able to complete the task only once in a row. Also, the best result for Rosenbrock optimization was obtained in lower number of seconds during the parallel runs.

Table 6. 200D functions parallel ABC results (6 cores)

f	**Iterations**		**Time (sec.)**	
	Best	**Worst**	**Best**	**Worst**
Sphere	193	209	3.1	3.4
Step	197	205	3.3	3.6
Rastrigin	304	371	5.9	7.2
Schwefel	561	1100	9.1	17.9
Griewank	109	208	2.5	4.8
Rosenbrock**	1184	11767	24.3	242
Ackley	301	320	7.4	8.1

The parallel routine distributed in 6 cores results in average (except Rosenbrock function) 6.4 seconds. This result is even faster than 4 cores parallelism, which is actually obvious in the given circumstances. And although the number of iterations to achieve the result has not changed critically, the amount of time spent on optimization has been decreased. Also, in this particular case Rosenbrock function was successfully minimized only by 9/10 times**. However, the execution time is also better than 4 cores implementation. It is interesting that the reduction of the number of agents in 6 processes compared to 4 processes affects the search capacity and, in the case of a rather complex Rosenbrock function, reduces the minimization efficiency. This leads to the fact that it is necessary to study this aspect in the case of complex hyper-surfaces.

The strength of the proposed parallel approach is that the exchange mechanism between processes can be implemented in different ways, not only replacing the best solutions in each process with the best of all, but also experimenting with various schemes of inter-process migrations.

Conclusions. Since the Python 3 programming language is a standard in the development of both mathematical models and the implementation of machine learning algorithms, it is relevant to study parallelization taking into account its limitations. Swarm algorithms such as ABC being highly parallel in their nature can be actively used as the numerical optimization methods implemented specifically in the Python 3 ecosystem. Hence this work is the additional step forward developing the swarm-based Python 3 solutions.

Here we have studied a process of improving the standard ABC algorithm in a parallel form. Based on Python 3 implementation we have conducted several experiments with the sequential algorithm. Given seven benchmark 50, 100 and 200 dimensional functions of the different complexity we applied ABC method to minimize each of them and identified corresponding number of iterations and execution time to achieve the global minima. Because standard ABC approach has a lot of spots to make it more efficient and to speed up computations, we present a simple and effective parallelization scheme.

This scheme involves splitting the entire population of solutions into sub-groups across processes and running each of them in parallel. To force the search capabilities, we propose exchange mechanism: the best solution among all sub-groups (sub-swarms) replaces the best solution in each sub-group after some predefined number of local (inside each process) iterations. These iterations are just a standard sequential execution of employed, onlooker and scout bees phases. The comparative study between sequential and parallel forms of ABC revealed that proposed parallel scheme outperforms sequential in terms of execution time and number of iterations. Also, such a scheme can serve as a basis for further research of parallel swarm algorithms.

References

[1] A comprehensive survey: artificial bee colony (ABC) algorithm and applications / Dervis Karaboga [et al.] // Artificial Intelligence Review. – 2012. – T. 42, № 1. – C. 21–57. – DOI: https://doi.org/10.1007/s10462-012-9328-0 .

[2] *Hassanien A. E.* Swarm Intelligence: Principles, Advances, and Applications / Aboul Ella Hassanien, Eid Emary. Taylor & Francis Group, 2018. – 210 c.

[3] *Chopra D.* Swarm Intelligence in Data Science: Challenges, Opportunities and Applications / Deepti Chopra, Praveen Arora // Procedia Computer Science. – 2022. – Vol. 215. – P. 104–111. DOI: https://doi.org/10.1016/j.procs.2022.12.012 .

[4] Foundations of Fuzzy Logic and Soft Computing / ed. by P. Melin [et al.]. – Berlin, Heidelberg: Springer Berlin Heidelberg, 2007. DOI: https://doi.org/10.1007/978-3-540-72950-1.

[5] *Dorigo M.* Ant colony optimization theory: A survey / Marco Dorigo, Christian Blum // Theoretical Computer Science. – 2005. – Vol. 344, no. 2-3. – P. 243–278. DOI: https://doi.org/10.1016/j.tcs.2005.05.020 .

[6] *Mirjalili S.* Grey Wolf Optimizer / Seyedali Mirjalili, Seyed Mohammad Mirjalili, Andrew Lewis // Advances in Engineering Software. – 2014. – Vol. 69. – P. 46–61. DOI: https://doi.org/10.1016/j.advengsoft.2013.12.007 .

[7] *Boussaïd I.* A survey on optimization metaheuristics / Ilhem Boussaïd, Julien Lepagnot, Patrick Siarry // Information Sciences. – 2013. – Vol. 237. – P. 82–117. – Mode of access: https://doi.org/10.1016/j.ins.2013.02.041 .

[8] *Hong Y.* Research of Parallel Artificial Bee Colony Algorithm Based on MPI / Yingsen Hong, Zhenzhou Ji, Chunlei Liu // 2nd International Conference on Computer Science and Electronics Engineering (ICCSEE 2013), China, 22–23 March 2013. – Paris, France, 2013. DOI: https://doi.org/10.2991/iccsee.2013.339 .

[9] A comparative study of GPU metaheuristics for data clustering / Mario Santos [et al.] // 2021 IEEE International Conference on Systems, Man, and Cybernetics (SMC), Melbourne, Australia, 17–20 October 2021. – [S. l.], 2021. DOI: https://doi.org/10.1109/smc52423.2021.9658803 .

[10] High-Level Parallel Ant Colony Optimization with Algorithmic Skeletons / Breno A. de Melo Menezes [et al.] // International Journal of Parallel Programming. – 2021. DOI: https://doi.org/10.1007/s10766-021-00714-1 .

[11] A Survey on Parallel Particle Swarm Optimization Algorithms / Soniya Lalwani [et al.] // Arabian Journal for Science and Engineering. – 2019. – Vol. 44, no. 4. – P. 2899–2923. DOI: https://doi.org/10.1007/s13369-018-03713-6 .

[12] *Harikrishna Narasimhan*. Parallel artificial bee colony (PABC) algorithm / Harikrishna Narasimhan // 2009 World Congress on Nature & Biologically Inspired Computing (NaBIC), Coimbatore, India, 9–11 December 2009. – [S. l.], 2009. DOI: https://doi.org/10.1109/nabic.2009.5393726 .

[13] *Akay B.* Synchronous and asynchronous Pareto-based multi-objective Artificial Bee Colony algorithms / Bahriye Akay // Journal of Global Optimization. – 2012. – Vol. 57, no. 2. – P. 415–445. – DOI: https://doi.org/10.1007/s10898-012-9993-1 .

[14] *Karaboga D.* A new emigrant creation strategy for parallel Artificial Bee Colony algorithm / Dervis Karaboga, Selcuk Aslan // 2015 9th International Conference on Electrical and Electronics Engineering (ELECO), Bursa, 26–28 November 2015. – [S. l.], 2015. DOI: https://doi.org/10.1109/eleco.2015.7394477.

[15] *Aslan S.* A new emigrant utilization strategy for parallel artificial bee colony algorithm / Selcuk Aslan // Evolving Systems. – 2019. DOI: https://doi.org/10.1007/s12530-019-09294-5.

[16] Python Parallel Processing and Multiprocessing: A Rivew / Zina A. Aziz [et al.] // Academic Journal of Nawroz University. – 2021. – Vol. 10, no. 3. – P. 345–354. DOI: https://doi.org/10.25007/ajnu.v10n3a1145.

[17] *Karaboga D.* An Idea Based in Honey Bee Swarm for Numerical Optimization / Dervis Karaboga // Artificial Bee Colony (ABC) Algorithm Homepage. – Mode of access: https://abc.erciyes.edu.tr/pub/tr06_2005.pdf .

[18] *Slowik A.* Nature Inspired Methods and Their Industry Applications–Swarm Intelligence Algorithms / Adam Slowik, Halina Kwasnicka // IEEE Transactions on Industrial Informatics. – 2018. – Vol. 14, no. 3. – P. 1004–1015. DOI: https://doi.org/10.1109/tii.2017.2786782 .

[19] Research and implementation of parallel artificial bee colony algorithm based on ternary optical computer / Shuang Li [et al.] // Automatika. – 2019. – Vol. 60, no. 4. – P. 423–432. DOI: https://doi.org/10.1080/00051144.2019.1639118 .

[20] Parpinelli R. S. Parallel Approaches for the Artificial Bee Colony Algorithm / Rafael Stubs Parpinelli, César Manuel Vargas Benitez, Heitor Silvério Lopes // Adaptation,

Learning, and Optimization. – Berlin, Heidelberg, 2011. – P. 329–345. DOI: https://doi.org/10.1007/978-3-642-17390-5_14.

[21] *Arjona A.* Transparent serverless execution of Python multiprocessing applications / Aitor Arjona, Gerard Finol, Pedro García López // Future Generation Computer Systems. – 2022. DOI: https://doi.org/10.1016/j.future.2022.10.038 .

[22] *Jamil M.* A literature survey of benchmark functions for global optimisation problems / Momin Jamil, Xin She Yang // International Journal of Mathematical Modelling and Numerical Optimisation. – 2013. – Vol. 4, no. 2. – P. 150. DOI: https://doi.org/10.1504/ijmmno.2013.055204 .

ДОСЛІДЖЕННЯ ПАРАЛЕЛЬНОЇ МОДЕЛІ АЛГОРИТМУ ШТУЧНОГО БДЖОЛИНОГО РОЮ

О. Сінькевич, В. Бойко, Л. Монастирський, Б. Соколовський

кафедра радіоелектронних і комп'ютерних систем,
Львівський національний університет імені Івана Франка,
вул. Драгоманова, 50, 79005 Львів, Україна
oleh.sikevych@lnu.edu.ua

Ройові алгоритми, такі як штучний бджолиний рій (ABC), будучи паралельними за своєю природою, можуть використовуватися як чисельні методи оптимізації, реалізовані в екосистемі Python 3. Оскільки мова програмування Python 3 є стандартом у розробці як математичних моделей, так і реалізації алгоритмів машинного навчання, актуальним є вивчення розпаралелювання з урахуванням його обмежень. Дане дослідження є додатковим кроком у розробці рішень Python 3 на основі ройового інтелекту.

У роботі вивчається процес вдосконалення стандартного алгоритму ABC у паралельній формі. На основі реалізації Python 3 проведені чисельні експерименти з класичним послідовним алгоритмом. Враховуючи сім еталонних 50, 100 і 200-вимірних функцій різної складності, застосовано метод ABC для мінімізації кожної з них і визначено відповідну кількість ітерацій та час виконання для досягнення глобальних мінімумів. Оскільки стандартний підхід ABC має багато місць, щоб зробити його більш ефективним і прискорити обчислення, ми представляємо просту та ефективну схему його розпаралелювання. Ця схема передбачає розбиття всієї популяції рішень на підгрупи за процесами та виконання кожного з них паралельно. Щоб посилити можливості пошуку, ми пропонуємо механізм обміну: найкращий розв'язок серед усіх підгруп замінює найкраще рішення в кожній підгрупі після попередньо визначеної кількості локальних (всередині кожного процесу) ітерацій. Ці ітерації є лише стандартним послідовним виконанням фаз зайнятих бджіл, бджіл-спостерігачів і бджіл-розвідників. Порівняльне дослідження між послідовними та паралельними формами ABC показало, що запропонована паралельна схема перевершує послідовну з точки зору часу виконання та кількості ітерацій. Також, така схема може бути основою для подальших досліджень паралельних ройових алгоритмів.

Ключові слова: штучний бджолиний рій; метаевристика; ройовий інтелект; чисельна оптимізація; багатопроцесність.

Стаття надійшла до редакції 16.02.2024.
Прийнята до друку 23.02.2024.