Parallel Artificial Bee Colony (PABC) Algorithm

Harikrishna Narasimhan
Department of Computer Science and Engineering
College of Engineering, Guindy, Anna University
Chennai – 600 025, Tamil Nadu, India
nhari88@gmail.com

***Abstract*—The artificial bee colony (ABC) algorithm is a metaheuristic algorithm for numerical optimization. It is based on the intelligent foraging behavior of honey bees. This paper presents a parallel version of the algorithm for shared memory architectures. The entire colony of bees is divided equally among the available processors. A set of solutions is placed in the local memory of each processor. A copy of each solution is also maintained in a global shared memory. During each cycle, the set of bees at a processor improves the solutions in the local memory. At the end of the cycle, the solutions are copied into the corresponding slots in the shared memory and made available to all other bees. It is shown that the proposed parallelization strategy does not degrade the quality of solutions obtained, but achieves substantial speedup.**

Keywords- Artificial bee colony (ABC) algorithm; numerical optimization; swarm intelligence; parallel metaheuristics; shared memory;

I. INTRODUCTION

Metaheuristics are high-level strategies for exploring search spaces [1]. Many metaheuristic algorithms that have been inspired from nature are efficient in solving numerical optimization problems. One such algorithm is the Artificial Bee Colony (ABC) algorithm motivated by the intelligent foraging behavior of honey bees.

The ABC algorithm was proposed by Karaboga [2] in 2005 for unconstrained optimization problems. Subsequently, the algorithm has been developed by Karaboga and Basturk [3], [4], [5], [6] and extended to constrained optimization problems [7]. Improvements to the performance of the algorithm [8], [9] and a hybrid version of the algorithm [10] have been also been proposed.

The ABC algorithm is a swarm-based algorithm good at solving unimodal and multimodal numerical optimization problems. It is very simple and flexible when compared to other swarm based algorithms such as Particle Swarm Optimization (PSO). It does not require external parameters like mutation and crossover rates, which are hard to determine in prior. The algorithm combines local search methods with global search methods and tries to attain a balance between exploration and exploitation [2].

Researchers have come up with several real-world applications for the ABC algorithm. One such application is in training artificial neural networks [11], [12]. Applications in graph theory include solving leaf constrained spanning trees [13] and the travelling salesman problem [14]. The algorithm has also been used to solve the generalized assignment problem [15]. Recently many engineering applications have come up for the algorithm including minimization of power loss in distribution networks [16].

A lot of research has gone into parallel metaheuristics with the aim of obtaining more efficient optimization algorithms. Several parallelization strategies have been proposed for genetic algorithm (GA). One popular strategy, which is also used in parallelizing other metaheuristic algorithms, is the master-slave configuration, where the evaluation of fitness is distributed across the available processors. This approach does not alter the behavior of the algorithm. Other approaches that modify the behavior of the algorithm can be seen in fine grained parallel GAs and multiple-deme parallel GAs [17], [18].

In general, parallel architectures may use either a shared memory or a message passing mechanism to enable communication between the multiple processing elements. Parallel metaheuristic algorithms have been developed for both these kinds of architectures. There have been several publications on parallel ant systems and parallel particle swarm optimization for different applications. However, to the best of my knowledge, a parallel implementation for the artificial bee colony algorithm does not exist in literature.

In this paper, a parallel model has been proposed for the ABC algorithm on shared memory architectures. A similar implementation can be seen in global GAs that use shared memory [17]. Abramson and Abela [19] implemented a GA on a shared memory multiprocessor for solving the school timetabling problem. Shared memory GAs can also be seen in [20] and [21]. Shared memory implementations for other metaheuristic algorithms include an implementation of ant colony optimization using OpenMP to solve the travelling salesman problem [22].

The rest of the paper is organized as follows. Section II provides a description of the ABC algorithm. The proposed parallel implementation has been explained in Section III. The experiments and results have been presented in Section IV. Section V contains some conclusions.

II. ARTIFICIAL BEE COLONY (ABC) ALGORITHM

The Artificial Bee Colony (ABC) algorithm uses a colony of artificial bees. The bees are classified into three types: 1. Employed bees, 2. Onlooker bees, and 3. Scout bees. Each employed bee is associated with a food source, which it exploits currently. A bee waiting in the hive to

978-1-4244-5612-3/09/$26.00 ©2009 IEEE

choose a food source is an onlooker bee. The employed bees share information about the food sources with onlooker bees in the dance area. A scout bee, on the other hand, carries out a random search to discover new food sources.

In a robust search process, exploration and exploitation must be carried out together. In the ABC algorithm, the scout bees are in charge of the exploration process, while the employed and onlooker bees carry out the exploitation process [4].

In the algorithm, one half of the population consists of employed bees and the other half consists of onlooker bees. The number of food sources equals the number of employed bees. During each cycle, the employed bees try to improve their food sources. Each onlooker bee then chooses a food source based on the nectar amount available at that food source. An employed bee whose food source is exhausted becomes a scout bee. The scout bee then searches for a new food source.

The position of a food source represents a solution for an optimization problem. The nectar amount of the food source is the fitness of the solution. Each solution is represented using a D-dimensional vector. Here, D is the number of optimization parameters. Initially, SN solutions are generated randomly, where SN equals the number of employed bees. Let MCN be the maximum number of cycles that the algorithm would run.

During each cycle, the employed and onlooker bees improve the solutions through a neighborhood search. A new solution v_i in the neighborhood of an existing solution x_i is produced as follows:

$$v_{ij} = x_{ij} + \varphi_{ij}(x_{ij} - x_{kj}), \quad (1)$$

where $k \in \{1, 2, 3, \ldots, \mathrm{SN}\}$ and $k \neq i$, φ is a random number in the range [-1,1] and $j \in \{1, 2, 3, \ldots, D\}$. k and j are chosen randomly. A greedy selection is then performed between x_i and v_i.

The onlooker bees are placed on food sources using the roulette wheel selection method [23]. An onlooker bee thus chooses a food source at position x_i with a probability p_i calculated as follows:

$$p_i = \frac{fit_i}{\sum_{n=1}^{\mathrm{SN}} fit_n}. \quad (2)$$

Here, fit_i is calculated using the following equation:

$$fit_i = \begin{cases} \frac{1}{1+f_i} & \text{if } f_i \leq 0 \\ 1 + abs(f_i) & \text{if } f_i > 0 \end{cases}, \quad (3)$$

where f_i is the fitness value of the solution.

A solution representing a food source is abandoned by an employed bee if it cannot be improved for a predetermined number of trials given by the parameter "limit". The employed bee then becomes a scout bee and randomly produces a new solution replacing the existing solution. The value of limit is generally chosen as SN × D.

The basic algorithm is sketched below:

1. Generate the initial solutions (positions of food sources) randomly and evaluate them.
2. For each solution x_i, determine a neighbor v_i using (1) and perform a greedy selection between x_i and v_i.
3. Calculate the probabilities for the solutions using (2).
4. Use the roulette wheel selection method to place the onlookers on the food sources and improve the corresponding solutions (as in step 2).
5. Determine the abandoned solution (if any) and replace it with a new randomly produced solution.
6. Record the best solution obtained till now.
7. Repeat steps 2 to 6 until MCN cycles are completed.

III. Parallel ABC Algorithm

It can be observed that many parts of the Artificial Bee Colony algorithm can be run in parallel. One way to have a parallel implementation would be to evaluate the fitness for each solution on a different processor. Alternatively, we could distribute the bees among the various processors and allow them to improve the solutions independently. However, this approach would be hindered by the dependencies between the bees that arise in three steps of the algorithm:

- Neighborhood search
- Probability calculation
- Roulette wheel selection

When a bee carries out a neighborhood search, it needs to know the position of another randomly chosen solution. Also, the probability for a solution cannot be calculated without knowing the fitness of other solutions. Clearly, the roulette wheel selection method also introduces dependencies.

A parallel implementation of the algorithm is designed for a shared memory architecture, which overcomes these dependencies. The entire colony of bees is divided equally among the available processors. Each processor has a set of solutions in a local memory. A copy of each solution is also maintained in a global shared memory. During each cycle the set of bees at a processor improves the solutions in the local memory. At the end of the cycle, the solutions are copied into the corresponding slots in the shared memory overwriting the previous copies. The solutions are thus made available to all the processors.

Let p be the number of processors and SN_p be the number of solutions in the local memory of each processor. Then, SN_p = SN / p. The number of employed bees and the number of onlooker bees per processor equals SN_p.

During a neighborhood search, a bee improves a solution in the local memory by randomly picking another solution from the shared memory. Let v_i^r be the new solution in the neighborhood of an existing solution x_i^r in the local memory M_r of processor P_r. If x_k^s is a randomly chosen solution from

TABLE I. NUMERICAL BENCHMARK FUNCTIONS

Function		Ranges	Minimum Value
Sphere	$f_1(\vec{x}) = \sum_{i=1}^{n} x_i^2$	$-100 \le x_i \le 100$	$f_1(\vec{0}) = 0$
Rosenbrock	$f_2(\vec{x}) = \sum_{i=1}^{n-1} 100(x_{i+1} - x_i^2)^2 + (x_i - 1)^2$	$-50 \le x_i \le 50$	$f_2(\vec{1}) = 0$
Rastrigin	$f_3(\vec{x}) = \sum_{i=1}^{n} x_i^2 - 10\cos(2\pi x_i) + 10$	$-5.12 \le x_i \le 5.12$	$f_3(\vec{0}) = 0$
Griewank	$f_4(\vec{x}) = \frac{1}{4000}\left(\sum_{i=1}^{n} {x_i}^2\right) - \left(\prod_{i=1}^{n} \cos\left(\frac{x_i}{\sqrt{i}}\right)\right) + 1$	$-600 \le x_i \le 600$	$f_4(\vec{0}) = 0$
Schaffer	$f_5(\vec{x}) = 0.5 + \frac{\sin^2\left(\sqrt{x_1^2 + x_2^2}\right) - 0.5}{(1 + 0.001(x_1^2 + x_2^2))^2}$	$-100 \le x_i \le 100$	$f_5(\vec{0}) = 0$

the shared memory *S*,

$$v_{ij}^r = x_{ij}^r + \varphi_{ij}^r(x_{ij}^r - x_{kj}^S), \quad (4)$$

where $k \in \{1, 2, 3, \ldots, SN\}$ and $k \neq i$, φ is a random number in the range [-1,1] and $j \in \{1, 2, 3, \ldots, D\}$.

The calculation of probabilities is done locally as follows:

$$p_i = \frac{fit_i}{\sum_{n=1}^{SN_p} fit_n}, \quad (5)$$

where fit_n is the fitness of a solution in the local memory. An onlooker bee at a processor picks up a solution from the local memory based on its probability.

Thus, a global neighborhood search and a local roulette wheel selection allow a parallel implementation for the algorithm. The parallel ABC algorithm is sketched below:

1. Generate SN initial solutions randomly and evaluate them. Place them in the shared memory *S*.
2. Divide the solutions equally among *p* processors by copying SN_p solutions to the local memory of each processor.
3. Steps 4 to 10 are carried out in parallel at each processor P_r.
4. For each solution x_i^r in the local memory M_r of processor P_r, determine a neighbor v_i^r using (4) and perform a greedy selection between x_i^r and v_i^r.
5. Calculate the probabilities for the solutions in M_r using (5).
6. Use the roulette wheel selection method to place the onlookers on the food sources in M_r and improve the corresponding solutions (as in step 4).
7. Determine the abandoned solution (if any) in M_r and replace it with a new randomly produced solution.
8. Record the best local solution obtained till now at P_r.
9. Copy the solutions in M_r to the corresponding slots in *S*.
10. Repeat steps 4 to 9 until MCN cycles are completed.
11. Determine the global best solution among the best local solutions recorded at each processor.

It has to be noted that the solutions in the shared memory are read concurrently, but have to be written exclusively.

IV. EXPERIMENTS AND RESULTS

The proposed parallel algorithm was tested on five classical benchmark functions as in [5]. The functions used along with their parameter ranges and global minimum have been listed in Table I. The Sphere function f_1 and the Rosenbrock function f_2 are unimodal, whereas Rastrigin f_3, Griewank f_4, and Schaffer f_5 functions are multimodal.

For the experiment, the size of the bee colony was chosen as 160 with 50% employed bees and 50% onlooker bees (SN = 80). A maximum of one scout bee is produced per cycle. The dimension is 30 for all functions except f_5 (D = 2) and limit = SN × D. The ABC algorithm was run for 2000 iterations for functions f_1 and f_5 and 5000 iterations for f_2, f_3, and f_4. The algorithm was simulated for different number of processors. The mean best values and the standard deviation (SD) over 30 runs for 1, 4, and 16 processors have been tabulated in Table II. The progress of the mean best values is shown in Fig. 1-5.

From the simulation results it is evident that the performance of the proposed algorithm does not degrade with increase in number of processors. In the case of functions f_2, f_3, and f_4, the mean best values plotted follow a similar curve for different number of processors. In the case of the Sphere function (f_1), increasing the number of processors gives lower mean best values in the initial cycles, but gives better performance in the later cycles. Eventually, the parallel and sequential versions converge to almost the same value. Only in the case of Schaffer (f_5), the number of cycles taken to converge changes significantly for different

TABLE II. RESULTS OBTAINED FOR THE ABC AND PABC ALGORITHMS ON DIFFERENT BENCHMARK FUNCTIONS

f	*D*	MCN	ABC		PABC			
					p=4		p=16	
			Mean	*SD*	*Mean*	*SD*	*Mean*	*SD*
f_1	30	2000	2.647711e-16	2.115732e-17	2.492479e-16	4.043068e-17	**2.467389e-16**	4.100729e-17
f_2	30	5000	2.306847e-02	3.120865e-02	**2.182352e-02**	3.250047e-02	2.282869e-02	2.585128e-02
f_3	30	5000	**1.865695e-16**	5.152662e-17	1.946071e-16	4.615336e-17	1.931904e-16	5.386725e-17
f_4	30	5000	**4.654841e-18**	5.501250e-19	4.896980e-18	7.036649e-19	4.756034e-18	6.995602e-19
f_5	2	2000	2.827780e-17	1.344715e-17	**2.418765e-17**	1.549690e-17	2.932405e-17	1.634437e-17

TABLE III. RESULTS OBTAINED FOR THE PABC ALGORITHM BY VARYING THE PARAMETER LIMIT

f		*p*=4			*p*=8			*p*=16		
		limit=SN×D	*limit=SN×D/p*	*limit=0.25×SN ×D×p*	*limit=SN×D*	*limit=SN×D/p*	*limit=0.25×SN ×D×p*	*limit=SN×D*	*limit=SN×D/p*	*limit=0.25×SN ×D×p*
f_1	Mean	**2.492479e-16**	2.544817e-16	2.527993e-16	2.653557e-16	2.554872e-16	**2.449317e-16**	**2.467389e-16**	3.127052e-16	2.491927e-16
	SD	4.043068e-17	3.357172e-17	4.651466e-17	1.228011e-17	3.582793e-17	4.345551e-17	4.100729e-17	5.191544e-17	3.756103e-17
f_2	Mean	**2.182352e-02**	4.915440e-02	3.219451e-02	**2.826200e-02**	4.713457e-02	6.012437e-02	**2.282869e-02**	5.782222e-02	3.655643e-02
	SD	3.250047e-02	8.423102e-02	5.789224e-02	4.547282e-02	6.228730e-02	9.873277e-02	2.585128e-02	7.373245e-02	4.684016e-02
f_3	Mean	1.946071e-16	1.983078e-16	**1.783296e-16**	**1.716509e-16**	2.415602e-16	1.739928e-16	1.931904e-16	2.877617e-16	**1.793415e-16**
	SD	4.615336e-17	5.577592e-17	4.722549e-17	4.209958e-17	4.388070e-17	4.559255e-17	5.386725e-17	5.069267e-17	4.801074e-17
f_4	Mean	4.896980e-18	4.727121e-18	**4.405475e-18**	**4.331388e-18**	4.830121e-18	4.407282e-18	4.756034e-18	2.671655e-17	**4.625929e-18**
	SD	7.036649e-19	8.078822e-19	6.075296e-19	7.669469e-19	1.027740e-18	6.135010e-19	6.995602e-19	6.671409e-17	6.804514e-19
f_5	Mean	**2.418765e-17**	7.243618e-06	2.598742e-17	2.822088e-17	2.642529e-04	**2.437196e-17**	2.932405e-17	2.759761e-03	**2.868889e-17**
	SD	1.549690e-17	6.595380e-06	1.699398e-17	1.670526e-17	2.647448e-04	1.435020e-17	1.634437e-17	2.779701e-03	1.432422e-17

number of processors. However, this change can also be observed in different runs of the sequential algorithm and is not related to the number of processors used. Even here, the algorithm converges to almost the same value for different number of processors.

Another aspect that was investigated is the effect of scout production on the performance of the proposed algorithm. Scout production is controlled by the parameter limit. Experiments conducted by Karaboga and Basturk [5] revealed that the algorithm performed best for limit = SN × *D*. It was tested whether this holds true for the parallel ABC algorithm. In the sequential version, there can be at most one scout bee produced per cycle. In the case of the PABC algorithm, *p* scout bees can be produced during each cycle. Hence, the value chosen for limit may depend on the number of processors used.

The algorithm was simulated on the benchmarks functions for different number of processors (p=4, 8, and 16). Except for limit, the parameters used were same as those used previously. The mean best values obtained over 30 runs have been tabulated in Table III. For limit = SN_p × *D* (i.e., limit = SN × *D* / *p*), the performance of the algorithm degraded with increase in number of processors. When we compare the performance of the algorithm for limit = SN × *D* and limit = 0.25 × SN × *D* × *p*, it is seen that sometimes the latter gives marginally better results.

The proposed algorithm was implemented in C using OpenMP [24] for a shared memory multicore architecture. We now describe the implementation. The population of bees is divided equally among *m* threads. Each thread independently improves the solutions in its private memory and copies them to the shared memory at the end of each cycle. The synchronization among threads required at the end of each iteration is a pure overhead. Therefore, to achieve reasonable speedup, we need to ensure that the work load in a single iteration is substantially high.

An Intel Core 2 Quad Q6600 machine, which has four processor cores, was used for the experiment. The parallel implementation was run on the Griewank (f_4) function with the following parameters: *D* = 1000, SN = 900, and MCN = 5000. The execution time for different number of threads was recorded. As given in [25], the speedup of was calculated using

$$\text{Speedup} = \frac{\text{Execution time on one processor core}}{\text{Execution time on } m \text{ processor cores}} \quad (6)$$

and the efficiency (normalized speedup) was calculated using

$$\text{Efficiency} = \frac{\text{Speedup}}{m}. \quad (7)$$

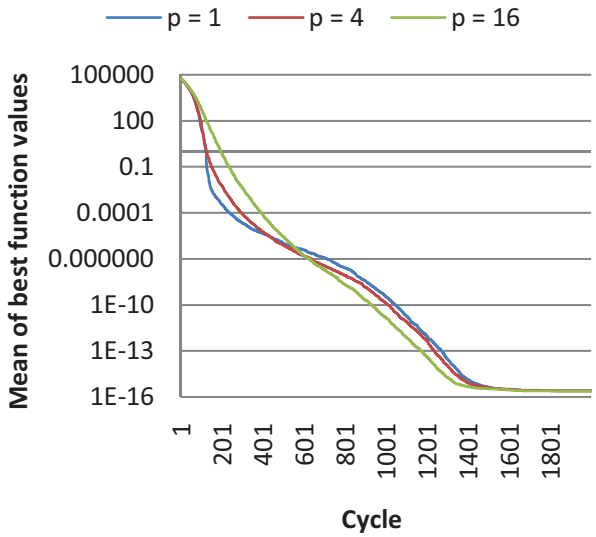

Figure1. Evolution of Mean Best Values – Sphere (f_1)

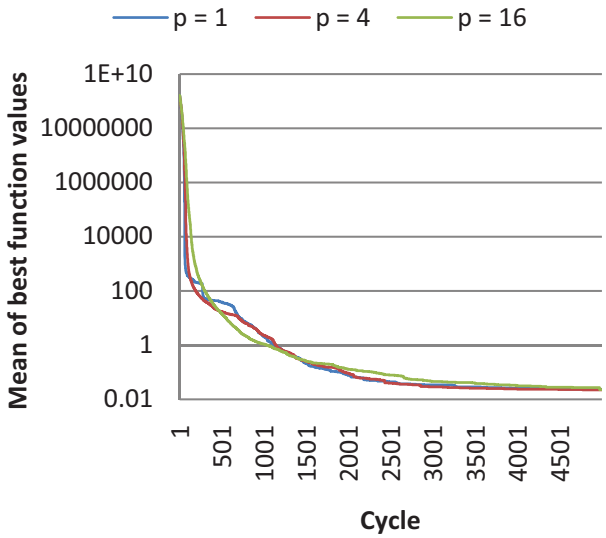

Figure 2. Evolution of Mean Best Values – Rosenbrock (f_2)

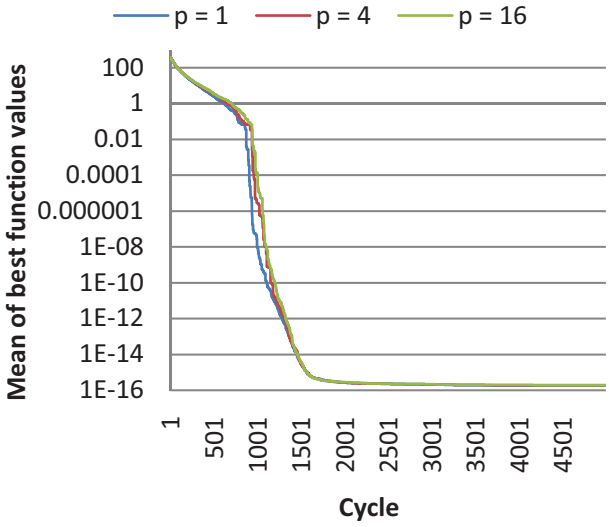

Figure 3. Evolution of Mean Best Values –Rastrigin (f_3)

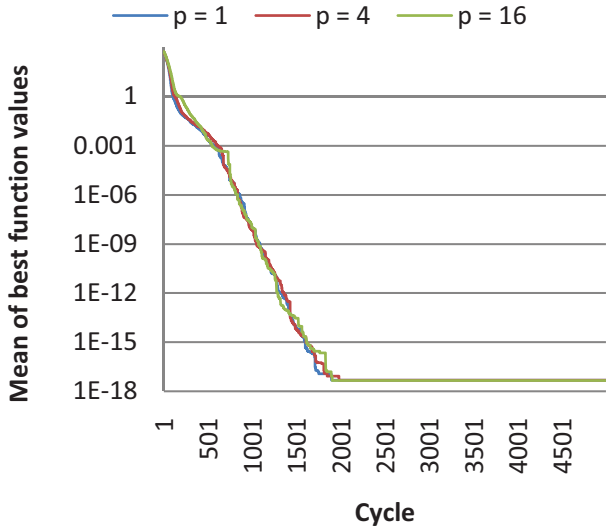

Figure 4. Evolution of Mean Best Values – Griewank (f_4)

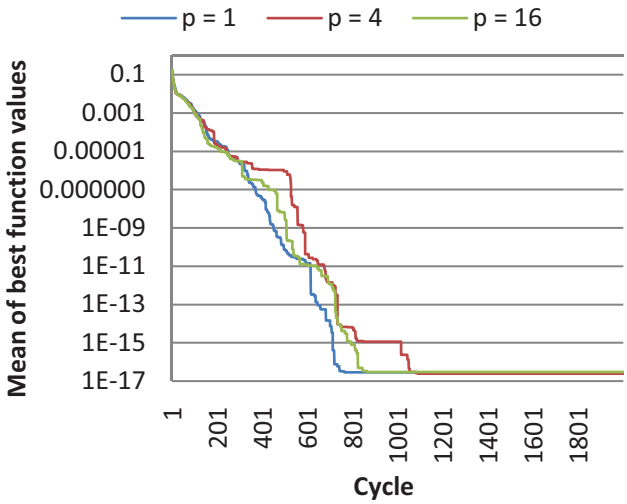

Figure 5. Evolution of Mean Best Values – Schaffer (f_5)

The results have been presented in Table IV. Clearly, the proposed implementation achieves a near linear speedup with efficiency close to 1. The speedup of the algorithm with a large number of processors will be investigated soon.

TABLE IV. SPEEDUP AND EFFICIENCY FOR GIVEN NUMBER OF THREADS

No of Threads	Execution Time (sec)	Speedup	Efficiency
1	991.675	-	-
2	498.278	**1.990**	**0.995**
3	334.484	**2.965**	**0.988**
4	252.067	**3.934**	**0.983**

V. CONCLUSIONS

In this paper, a parallel implementation has been proposed for the artificial bee colony algorithm on shared memory architectures. Through simulation results on a set of benchmark functions, it has been shown that the solutions produced by the parallel ABC algorithm are as good as those produced by the sequential ABC algorithm. Experimental results also showed that the algorithm can achieve a substantial speedup.

Future direction of work would be to test the performance of the proposed algorithm on real world applications and also to come up with a parallel implementation of the ABC algorithm for a message passing architecture.

ACKNOWLEDGMENT

I would like to thank Dr. K.S. Easwarakumar for his valuable guidance throughout the research work.

REFERENCES

[1] E. Alba, ed., "Parallel metaheuristics: a new class of algorithms," Wiley, 2005.

[2] D. Karaboga, "An idea based on honey bee swarm for numerical optimization," Technical Report TR06, Computer Engineering Department, Erciyes University, Turkey, 2005.

[3] B. Basturk, D. Karaboga, "An artificial bee colony (ABC) algorithm for numeric function optimization," Proc. IEEE Swarm Intelligence Symposium, Indianapolis, IN, USA, May 2006.

[4] D. Karaboga and B. Basturk, "A powerful and efficient algorithm for numerical function optimization: artificial bee colony (ABC) algorithm," Journal of Global Optimization 39, 2007, pp. 459–471.

[5] D. Karaboga, B. Basturk, "On the performance of artificial bee colony (ABC) algorithm," Applied Soft Computing 8, 2008, pp. 687–697.

[6] D. Karaboga and B. Akay, "A comparative study of artificial bee colony algorithm", Applied Mathematics and Computation, 214, 2009, pp. 108-132.

[7] D. Karaboga and B. Basturk, "Artificial bee colony (ABC) optimization algorithm for solving constrained optimization problems," Lecture Notes in Artificial Intelligence 4529, 2007, Springer-Verlag, Berlin, pp. 789–798.

[8] F. Qingxian and D. Haijun, "Bee colony algorithm for the function optimization," Science Paper Online, http://www.paper.edu.cn/en/paper.php?serial_number=200808-448.

[9] H. Quan and X. ShiOn, "On the analysis of performance of the improved artificial-bee-colony algorithm," Proc. Fourth International Conference on Natural Computation (ICNC'08), Jinan, China, Oct. 2008.

[10] F. Kang, J. Li, and Q. Xu, "Structural inverse analysis by hybrid simplex artificial bee colony algorithms," Comput. Struct. 87, 13-14, Jul. 2009, pp. 861-870.

[11] D. Karaboga and B. Basturk Akay, "Artificial bee colony algorithm on training artificial neural networks," Proc. Signal Processing and Communications Applications, 2007, SIU 2007, IEEE 15th, Jun. 2007, pp. 1-4, doi:10.1109/SIU.2007.4298679.

[12] D. Karaboga, B. Basturk Akay, and C. Ozturk, "Artificial bee colony (ABC) optimization algorithm for training feed-forward neural networks," in LNCS: Modeling Decisions for Artificial Intelligence, Vol: 4617/2007, Springer-Verlag, 2007, MDAI 2007, pp. 318-319.

[13] A. Singh, "An artificial bee colony algorithm for the leaf-constrained minimum spanning tree problem," Appl. Soft Comput. J., 2008, doi:10.1016/j.asoc.2008.09.001.

[14] L. Fenglei, D. Haijun, F. Xing, "The parameter improvement of bee colony algorithm in TSP problem," Science Paper Online, http://www.paper.edu.cn/en/paper.php?serial_number=200711-226.

[15] A. Baykosoglu, L. Ozbakir, and P. Tapkan, "Artificial bee colony algorithm and its application to generalized assignment problem," Swarm Intelligence: Focus on Ant and Particle Swarm Optimization, Austria: Itech Education and Publishing, 2007, pp. 532-564.

[16] R. Srinivasa Rao, S. V. L. Narasimham, and M. Ramalingaraju, "Optimization of distribution network configuration for loss reduction using artificial bee colony algorithm," International Journal of Electrical Power and Energy Systems Engineering (IJEPESE), Volume 1, Number 2, Spring 2008.

[17] E. Cantu-Paz, "A Summary of research on parallel genetic algorithms," IllGAL Report 95007, University of Illinois, Jul. 1995.

[18] N. Nedjah, E. Alba, L. M. Mourelle, "Parallel evolutionary computations," 3-540-32837-8, Springer-Verlag, 2006.

[19] D. Abramson and J. Abela, "A parallel genetic algorithm for solving the school timetabling problem," Proc. Fifteenth Australian Computer Science Conference (ACSC-15), 1992.

[20] R. Hauser, R. Manner, "Implementation of standard genetic algorithm on MIMD machines," in Parallel Problem Solving from Nature (PPSN III), Springer-Verlag (Berlin), Y. Davidor, H.-P. Schwefel, R. Manner, Eds., 1994, pp. 504-513.

[21] T. Maruyma, A. Konagaya, and K. Konishi, "An asynchronous fine–grained parallel genetic algorithm," Parallel Problem Solving from Nature 2, 1992, pp. 563–572.

[22] P. Delisle, M. Krajecki, M. Gravel, and C. Gagne, "Parallel implementation of an ant colony optimization metaheuristic with OpenMP," Proc. 3rd European Workshop on OpenMP (EWOMP'01), International Conference on Parallel Architectures and Compilation Techniques, Barcelona, Spain, Sep. 2001.

[23] D. E. Goldberg and K. Deb, "A comparative analysis of selection schemes used in genetic algorithms," in Foundations of Genetic Algorithms, Morgan Kaufmann, San Mateo, California, G. J. E. Rawlins, ed., 1991, pp. 69-93.

[24] OpenMP Architecture Review Board [2008], OpenMP Application Program Interface Version 3.0, http://www.openmp.org/.

[25] E. Alba and G. Luque, "Evaluation of parallel metaheuristics," Proc. Parallel Problem Solving from Nature (PPSN-EMAA'06), Reykjavik, Iceland, Sep. 2006, pp. 9-14.