2011 IEEE International Parallel & Distributed Processing Symposium

A Shared-Memory ACO-Based Algorithm for Numerical Optimization

Peter Korošec, Jurij Šilc
Computer Systems Department
Jožef Stefan Institute
Ljubljana, Slovenia
{*peter.korosec, jurij.silc*}*@ijs.si*

Marian Vajteršic, Rade Kutil
Department of Scientific Computing
University of Salzburg
Salzburg, Austria
{*marian, rkutil*}*@cosy.sbg.ac.at*

***Abstract*—Numerical optimization techniques are applied to a variety of engineering problems. The objective function evaluation is an important part of the numerical optimization and is usually realized as a black-box simulator. For efficient solving the numerical optimization problem, new shared-memory approach is proposed. The algorithm is based on an ACO meta-heuristics, where indirect coordination between ants drives the search procedure towards the optimal solution. Indirect coordination offers a high degree of parallelism and therefore relatively straightforward shared-memory implementation. For the communication between processors, the Intel-OpenMP library is used. It is shown that speed-up strongly depends on the simulation time. Therefore, algorithm's performance, according to simulator's time complexity, is experimentally evaluated and discussed.**

***Keywords*-numerical optimization; parallel computing; bio-inspired method; speed-up; experimental performance evaluation**

I. Introduction

Numerical optimization problems are an important field of research and have a wide number of applications, from the mathematical optimization of functions to the real-world engineering problems. Many engineering problems involve choosing the best configuration of a set of parameters to achieve a specified objective. Numerical optimization refers to the case when these parameters take continuous values, as opposed to combinatorial optimization, which deals with discrete values.

In this paper we consider the following numerical minimization problem:

$$\min(f(\vec{x})),\ \vec{l} \leq \vec{x} \leq \vec{u},$$

where $\vec{x} = (x_1, x_2, \ldots, x_n)$ is the variable vector in $\mathbb{R}^n$, $f(\vec{x})$ denotes the objective function to minimize and $\vec{l} = (l_1, l_2, \ldots, l_n)$, $\vec{u} = (u_1, u_2, \ldots, u_n)$ represent, respectively, the lower and the upper bound of the variables, such that $x_i \in [l_i, u_i]$.

Usually, numerical optimization problems are black-box optimizations (see Figure 1), in which the objective function's form as well as its derivatives are unknown. Normally, this occurs when the objective function is computed using a complex simulation about which the optimization algorithm has no information. Executing a black-box simulation in order to evaluate a candidate solution is usually very expensive and can take up to several minutes or even hours. This is particularly problematic because optimization algorithms for black-box problems are necessarily blind search algorithms that must repeatedly sample points in a solution space, evaluate them by running the simulation, and apply various heuristics in order to choose the next points to sample.

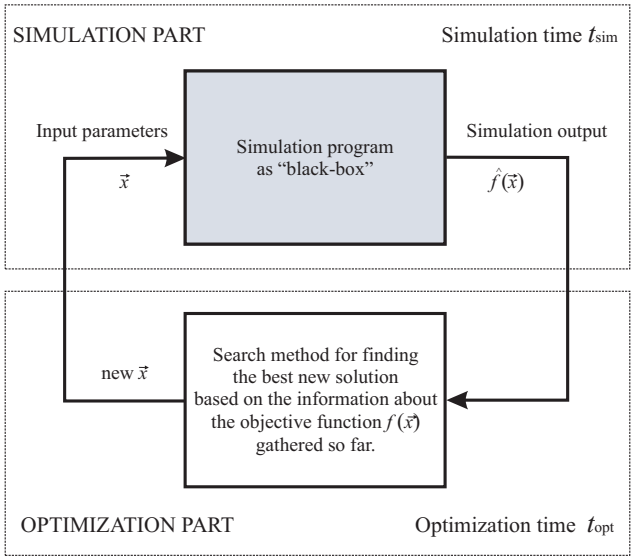

Figure 1. The Black-Box Optimization

In the past two decades, many bio-inspired optimization algorithms have been proposed to solve this kind of optimization problem, e.g., real-parameter genetic algorithm [1], evolution strategies [2], differential evolution [3], particle swarm optimization [4], immunological algorithm [5], continuous ant-colony optimization [6], etc. These algorithms have been used to solve problems in several research fields to the facts that do not require previous considerations regarding the problem to be optimized and offer a high degree of parallelism.

Although ACO-based optimization has been proven to be one of the best meta-heuristics in some combinatorial optimization problems [8], [7], its application to the nu-

1530-2075/11 $26.00 © 2011 IEEE
DOI 10.1109/IPDPS.2011.176

merical optimization appears more challenging, since the pheromone laying method is not straightforward. There are several possibilities to use ants for numerical optimization. We can use simplified direct simulation of real ants' behavior or we can extend the method to explore continuous spaces. This extension can be done by suitable discretization of a search space or by probabilistic sampling.

ACO has a high degree of inherent parallelism [9], [10], so several parallel [11] and distributed [12], [13] ACO-based algorithms for a combinatorial optimization were already proposed.

As far as we know, there are only few attempts of ACO-based algorithm parallelization for numerical optimization problems [14], [15]. One of the first distributed-memory implementations was the Distributed Multilevel Ant-Stigmergy Algorithm [15]. Its main flaw was a limited accuracy of solutions, which originates from the Multilevel Ant-Stigmergy Algorithm itself. Therefore, we recently implemented the sequential Differential Ant-Stigmergy Algorithm [16], which is limited only by the computer's floating-point arithmetics.

Motivated by possible computation speed-up and improved accuracy, we decided to propose a new shared-memory version of the Differential Ant-Stigmergy Algorithm.

The remainder of this paper is organized as follows. Section II presents the Differential Ant-Stigmergy Algorithm and its shared-memory version. Section III describes an experimental work, where an algorithm analysis according to the parallel execution-time is performed. Finally, our conclusions and some possible paths for future research are provided in Section IV.

II. The Differential Ant-Stigmergy Algorithm

To solve a numerical optimization problem, we created a fine-grained discrete form of continuous domain. With it we were able to represent this problem as a graph. This enabled us to use the ACO-based approach for solving numerical optimization problems.

A. The Fine-Grained Discrete Form of Continuous Domain

Let x'_i be the current value of the i-th parameter. During the search for the optimal parameter value, the new value, x_i, is assigned to the i-th parameter as follows:

$$x_i = x'_i + \omega\delta_i. \tag{1}$$

Here, δ_i is called the *parameter difference* and is chosen from the set $\Delta_i = \Delta_i^- \cup \{0\} \cup \Delta_i^+$, where

$$\Delta_i^- = \{\delta_{i,k}^-|\ \delta_{i,k}^- = -b^{k+B_{li}-1}, k = 1, 2, \ldots, d_i\},$$

$$\Delta_i^+ = \{\delta_{i,k}^+|\ \delta_{i,k}^+ = b^{k+B_{li}-1}, k = 1, 2, \ldots, d_i\}$$

and $d_i = B_{ui} - B_{li} + 1$. Therefore, for each parameter x_i, the parameter difference, δ_i, ranges from $b^{B_{li}}$ to $b^{B_{ui}}$, where b is the *discrete base*, $B_{li} = \lfloor \log_b(\epsilon_i) \rfloor$ and $B_{ui} = \lfloor \log_b(u_i - l_i) \rfloor$. With the parameter ϵ_i, the maximum precision of the parameter x_i is set. This precision is limited by the computer's floating-point arithmetics. To enable a more flexible movement over the search space, the random integer weight $\omega = \texttt{Rnd\_Integer}(1, b-1)$ is added to Eq. 1.

B. Search Graph Construction

From all the sets Δ_i, $1 \leq i \leq D$, where D represents the number of parameters, a *differential graph* $\mathcal{G} = (V, E)$ with a set of vertices, V, and a set of edges, E, between the vertices is constructed. Each vector Δ_i is represented by the set of vertices, $V_i = \{v_{i,1}, v_{i,2}, \ldots, v_{i,2d_i+1}\}$, and $V = \bigcup_{i=1}^{D} V_i$. Then

$$\Delta_i = \{\delta_{i,d_i}^-, \ldots, \delta_{i,d_i-j+1}^-, \ldots, \delta_{i,1}^-, 0, \delta_{i,1}^+, \ldots, \delta_{i,j}^+, \ldots, \delta_{i,d_i}^+\}$$

corresponds to

$$V_i = \{v_{i,1}, \ldots, v_{i,j}, \ldots, v_{i,d_i+1}, \ldots, v_{i,d_i+1+j}, \ldots, v_{i,2d_i+1}\},$$

where $v_{i,j} \xrightarrow{\delta} \delta_{i,d_i-j+1}^-$, $v_{i,d_i+1} \xrightarrow{\delta} 0$, $v_{i,d_i+1+j} \xrightarrow{\delta} \delta_{i,j}^+$, and $j = 1, 2, \ldots, d_i$.

Each vertex of the set V_i is connected to all the vertices that belong to the set V_{i+1}. Therefore, this is a directed graph (see Fig. 2), where each path $\vec{p}$ from the starting vertex, $v_1 \in V_1$, to any of the ending vertices, $v_D \in V_D$, is of equal length and can be defined with v_i as

$$\vec{p} = (v_1, v_2, \ldots, v_i, \ldots, v_D),$$

where $v_i \in V_i$, $1 \leq i \leq D$.

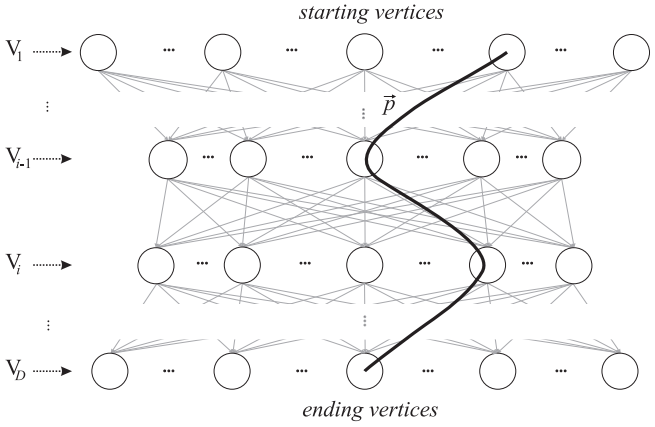

Figure 2. The Differential Graph

C. The Algorithm

The pseudocode of the Differential Ant-Stigmergy Algorithm (DASA) in sequential form is formulated in Algorithm 1. The optimization consists of an iterative improvement of the best solution, $\vec{x}^{\,b}$, by constructing an appropriate path $\vec{p}$. By applying $\vec{p}$ to $\vec{x}^{\,b}$ new solutions are produced (Eq. 1). First a solution $\vec{x}^{\,b}$ is randomly chosen and evaluated

(see Algorithm 1, lines 1 and 2). Then a search graph is created and an initial amount of pheromone is deposited on the search graph.

Algorithm 1 The DASA

```
1: x⃗^b = Rnd_Solution()
2: y^b = f(x⃗^b) {Black-box simulation}
3: G = Graph_Initialization(ε⃗)
4: Pheromone_Initialization(G)
5: while not ending condition met do
6:    for all m ants do
7:       p⃗_i = Find_Path(G) {path of the i-th ant}
8:       ω = Rnd_Integer(1, b − 1)
9:       x⃗_i = x⃗^b + ωδ(p⃗)
10:      y_i = f(x⃗_i) {Black-box simulation}
11:   end for
12:   if y_i < y^b then
13:      x⃗^b = x⃗_i
14:      y^b = y_i
15:      s = Update_Scales(s_global, s_local)
16:      Pheromone_Redistribution(p⃗_i, s)
17:   else
18:      Update_Scale(s_global)
19:   end if
20:   for all vertices in G do
21:      Pheromone_Evaporation(ρ)
22:   end for
23: end while
```

To simulate pheromone behavior in nature, we have decided to deposit pheromone on a differential graph according to the Cauchy probability density function

$$C(z) = \frac{1}{s\pi(1 + (\frac{z-l}{s})^2)},$$

where l is the location offset and $s = s_{\text{global}} - s_{\text{local}}$ is the scale factor. Here, factor s_{global} determines the amount of exploration and factor s_{local} enables a temporary exploitation, as will be shown later in the section. For an initial pheromone distribution, the standard Cauchy distribution ($l = 0$, $s_{\text{global}} = 1$, and $s_{\text{local}} = 0$) is used and each of the parameter vertices are equidistantly arranged between $z = [-4, 4]$. With the function `Graph_Initialization` (line 3) we create a search graph and with the function `Pheromone_Initialization` (line 4) we deposit initial pheromone on a search graph.

In the first **for** loop (lines 6–11), there are m ants in a colony, which simultaneously explore the differential graph (in sequential manner). An ant begins exploring from a starting vertex and uses a probability rule to determine which vertex will be chosen next.

Concretely, ant a in step i moves from a vertex in set V_{i-1} to the vertex $v_{i,j} \in V_i$ with a probability given by:

$$\text{prob}(a, v_{i,j}) = \frac{\tau(v_{i,j})}{\sum_{1 \le k \le 2d_i+1} \tau(v_{i,k})},$$

where $\tau(v_{i,k})$ is the amount of pheromone in vertex $v_{i,k}$.

The ants repeat this action until they reach the ending vertex. For each ant i, path $\vec{p_i}$ is constructed with function `Find_Path` (line 7). With it a new solution $\vec{x_i}$ is constructed (lines 8 and 9) and evaluated with a calculation of objective function $f(\vec{x_i})$ (line 10).

All solutions y_i are compared to the currently best solution y^{b} (line 12). If a solution is found that is better than the currently best solution, then information $\vec{x_i}$ and $\vec{y_i}$ regarding this solution is replaced as the current best one (lines 13 and 14). In this case, s_{global} is increased according to the algorithm parameter called global scale-increasing factor, s_+:

$$s_{\text{global}} \leftarrow (1 + s_+)s_{\text{global}}.$$

The value of s_{local} is set to

$$s_{\text{local}} = \frac{1}{2}s_{\text{global}}.$$

This is implemented with function `Update_Scales` (line 15). The pheromone amount is redistributed according to the associated path $\vec{p_i}$ with function `Pheromone_Redistribution` (line 16). If no better solution is found, s_{global} is decreased according to the algorithm parameter called global scale-decreasing factor, s_-:

$$s_{\text{global}} \leftarrow (1 - s_-)s_{\text{global}}.$$

These changes are implemented with the function `Update_Scale` (line 18).

The second **for** loop (lines 20–22) takes care of pheromone evaporation, where we need to simulate evaporation on all vertices in the graph $\mathcal{G}$. Pheromone evaporation is defined according to the predetermined percentage ρ. The Cauchy probability density function is changed in the following way:

$$l \leftarrow (1 - \rho)l$$

and

$$s_{\text{local}} \leftarrow (1 - \rho)s_{\text{local}}..$$

These changes are implemented with function `Pheromone_Evaporation` (line 21). Here we must emphasize that $\rho > s_-$, because otherwise we might obtain a negative scale factor. The whole procedure is then repeated until some termination condition is met (**while** loop, lines 5–23).

D. The Shared-Memory Implementation

The pseudocode of the Shared-Memory DASA (PDASA) is presented in Algorithm 2. Due to the stochastic nature of the algorithm we are forced to use a function that returns random numbers. If we would decide to use already implemented functions found in the programming language, we would run into the problem of sharing one global variable, `seed`, among all the threads, which is very time consuming. For this reason, we prepared an array of seeds

for all the threads $numOfThreads$, by calling function `Initialize_Rnd_Seed`($numOfThreads$) (line 5). We also implemented a simple random function, which can use values from this array as a seed to calculate new random numbers. In this way, we have eliminated the problem of sharing the global seed variable. All this is done in sequential form. Before we start with the **while** loop, we initialize all the threads (line 6). These threads will be used inside the loop to parallelize the main two **for** loops.

Algorithm 2 The PDASA

```
 1: x⃗^b = Rnd_Solution()
 2: y^b = f(x⃗^b) {Black-box simulation}
 3: G = Graph_Initialization(ε⃗)
 4: Pheromone_Initialization(G)
 5: Initialize_Rnd_Seed(numOfThreads)
 6: #pragma omp parallel {Creation of threads}
 7: while not ending condition met do
 8:    #pragma omp for schedule (static)
 9:    for all m ants do
10:       p⃗_i = Find_Path(G) {path of the i-th ant}
11:       ω = Rnd_Integer(1, b − 1)
12:       x⃗_i = x⃗^b + ωδ(p⃗)
13:       y_i = f(x⃗_i) {Black-box simulation}
14:    end for
15:    #pragma omp single
16:    if y_i < y^b then
17:       x⃗^b = x⃗_i
18:       y^b = y_i
19:       s = Update_Scales(s_global, s_local)
20:       Pheromone_Redistribution(p⃗_i, s)
21:    else
22:       Update_Scale(s_global)
23:    end if
24:    #pragma omp for schedule (static)
25:    for all vertices in G do
26:       Pheromone_Evaporation(ρ)
27:    end for
28: end while
```

In the first **for** loop (lines 9–14), there are m ants in a colony, where $numOfThreads$ of ants begin concurrently from the starting vertex in each own thread. With the command at line 15 we force the algorithm to run lines 16–23 in one thread, because it turned out to be the quickest and simplest way. The second **for** loop (lines 25–27), that takes care of pheromone evaporation, is also parallelized.

These are all the changes that were needed to be made in order the sequential algorithm can run under the shared-memory paradigm.

III. Experimental work

A. Environment

The computer platform used to perform the PDASA experiments was a dual processor motherboard with Intel Xeon DP E5345 2.33 GHz quad-core CPU and 16 GB of RAM, and the CentOS (Red Hat) 64-bit Linux operating system. The code of PDASA was written in C language. The Intel OpenMP 3.0 [17] was used for inter-thread communication.

The PDASA includes six parameters. The settings were the following: the number of ants $m = 32$, the maximum parameter precision $\varepsilon = 10^{-15}$, the discrete base $b = 10$, the pheromone dispersion factor $\rho = 0.2$, the global scale-increasing factor $s_+ = 0.01$, and the global scale-decreasing factor $s_- = 0.02$.

For testing purposes we have measured the execution time

$$t_{\text{exe}} = t_{\text{opt}} + t_{\text{sim}},$$

where t_{opt} is time of optimization and t_{sim} is time of black-box simulation. To simulate different degrees of simulation complexities, we have used a objective function which is run as black-box simulation. Its complexity is denoted by factor k. At $k = 1$ a single black-box simulation takes $t_{\text{sim}_1} = 0.85\ \mu$s for the PDASA.

B. Results

All results presented in this section are averages of 20 runs. We have compared execution time, relative speed-up, and processors utilization for different k values. Then the simulator's time for one simulation is

$$t_{\text{sim}_k} = k\ t_{\text{sim}_1}$$

and

$$t_{\text{sim}} = t_{\text{sim}_k} \times \#\text{ black-box calculations}.$$

The relative speed-up, $S_r(n)$ is defined as

$$S_r(n) = \frac{t_{\text{exe}}(1)}{t_{\text{exe}}(n)},$$

where $t_{\text{exe}}(1)$ is time to solve a problem with the parallel code with one processors and $t_{\text{exe}}(n)$ is a time to solve the same problem with the parallel code on n processors. Remark: Our primary goal is verification of proposed algorithms' scalability. Since the difference between the DASA and the PDASA code is very small, which resulted in almost equal execution times (i.e., $t_{\text{seq}} \approx t_{\text{exe}}(1)$), we decided to use only relative speed-up.

The efficiency $E(n)$ is defined as

$$E(n) = \frac{S_r(n)}{n}.$$

Table I shows performances of the PDASA when 1, 2, and 4 processors were used.

Here we can see that the efficiency is well over 50% on all instances. This shows that the PDASA is also applicable to very computationally simple black-box simulations.

Figures 3, 4, and 5 show average execution time, average relative speed-up, and average efficiency, respectively, over 20 runs for different number of processors when 500.000 black-box calculations were executed.

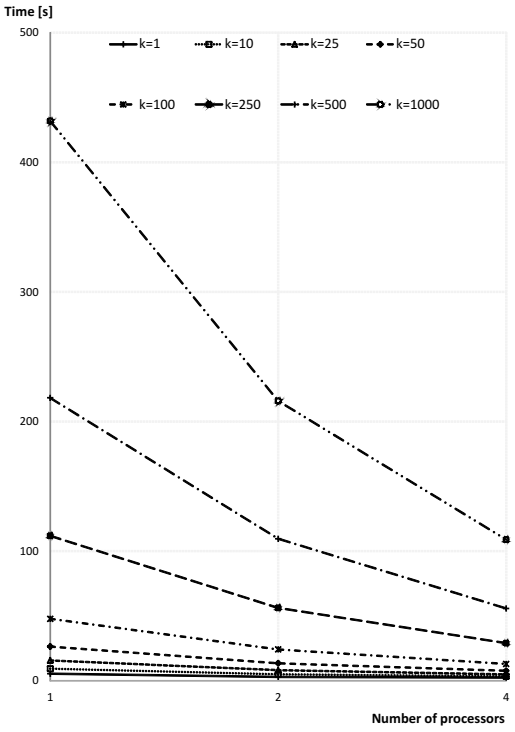

Figure 3. Average Execution Time

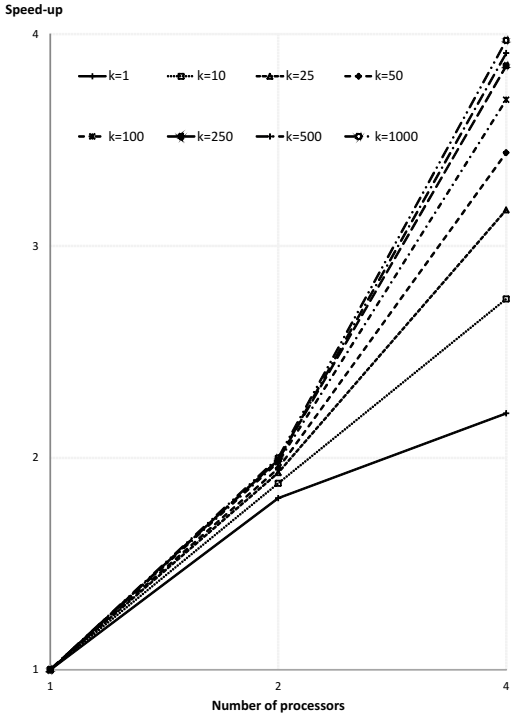

Figure 4. Average Relative Speed-up

Table I
THE PDASA AVERAGE EXECUTION TIME IN SECONDS, AVERAGE RELATIVE SPEED-UP, AND AVERAGE PROCESSORS EFFICIENCY IN PERCENTAGES ACCORDING TO DIFFERENT NUMBER OF PROCESSORS AND FACTOR OF BLACK-BOX SIMULATION COMPLEXITY

n	k	t_{exe} [s]	$S_r(n)$	$E(n)$ [%]
1	1	5.447	1.00	100.0
1	10	9.306	1.00	100.0
1	25	15.728	1.00	100.0
1	50	26.368	1.00	100.0
1	100	47.787	1.00	100.0
1	250	111.789	1.00	100.0
1	500	218.200	1.00	100.0
1	1000	431.827	1.00	100.0
2	1	3.006	1.81	90.5
2	10	4.960	1.88	94.0
2	25	8.133	1.93	96.5
2	50	13.505	1.95	97.5
2	100	24.157	1.98	99.0
2	250	56.197	1.99	99.5
2	500	109.527	1.99	99.5
2	1000	215.873	2.00	100.0
4	1	2.462	2.21	55.2
4	10	3.388	2.75	68.7
4	25	4.955	3.17	79.2
4	50	7.673	3.44	86.0
4	100	12.974	3.69	92.2
4	250	29.023	3.85	96.2
4	500	55.764	3.91	97.7
4	1000	108.859	3.97	99.2

IV. CONCLUSIONS

In this paper we have presented a shared-memory implementation of the Differential Ant-Stigmergy Algorithm (DASA) that is applied to the numerical optimization problems. The usefulness and efficiency of the algorithm, in its sequential form, to solve optimization problems has already been shown in previous work [18], [19]. Our aim was to decrease the execution time of the sequential algorithm.

We have shown the main issues that had to be addressed during the parallelization process. The results have shown that it was possible to implement efficient shared-memory version of the DASA. For the shared-memory version we used OpenMP, which is a portable, parallel application programming interface (API) for shared-memory multiprocessors. With it we were able to sufficiently speed-up even the simplest (computationally non-demanding) black-box simulations.

From the experimental results we can conclude that speed-up strongly depends on simulation time. We were able to achieve efficiency over 50% for the shared-memory algorithm even for very fast simulations, i.e., when the simulation time was less than 50 μs.

For future work, we plan to run scalability test on higher number of tightly-coupled processors, since processors with more than four cores are on its way.

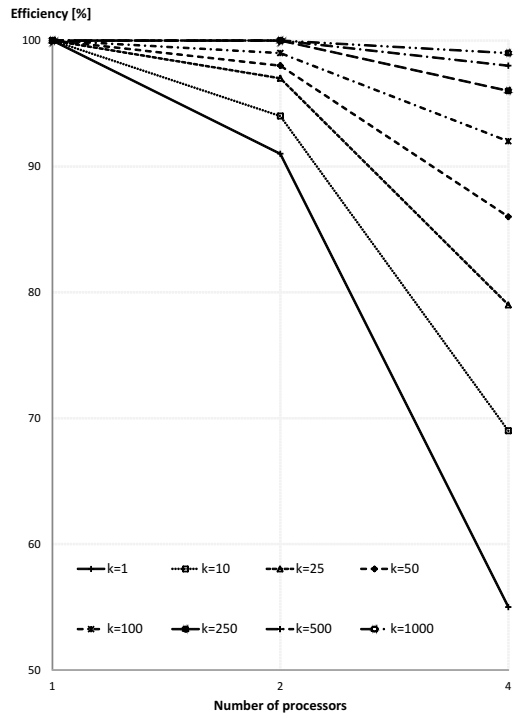

Figure 5. Average Efficiency

REFERENCES

[1] A.H. Wright, "Genetic algorithms for real parameter optimization," in *Foundations of Genetic Algorithms - 1*, G. J. E. Rawlins, Ed., Morgan Kaufman, 1991, pp. 205–218.

[2] K. Deb, A. Anand, and D. Joshi, "A computationally efficient evolutionary algorithm for real-parameter optimization," *Evolutionary Computation*, vol. 10, no. 4, pp. 371–395, 2002.

[3] R. Storn and K. V. Price, "Differential evolution – a fast and efficient heuristic for global optimization over continuous spaces," *Journal of Global Optimization*, vol. 11, no. 4, pp. 341–359, 1997.

[4] J. Kennedy and R. C. Eberhart, "Particle swarm optimization," in *Proceedings of the IEEE International Conference on Neural Networks*, New York, USA, November-December 1995, vol. 4, pp. 1942–1948.

[5] V. Cutello, G. Narzisi, G. Nicosia, and M. Pavone, "An immunological algorithm for global numerical optimization," *Lecture Notes in Computer Science*, vol. 3871, pp. 284–295, 2006.

[6] G. Bilchev and I. C. Parmee, "The ant colony metaphor for searching continuous design spaces," *Lecture Notes in Computer Science*, vol. 993, pp. 25–39, 1995.

[7] C. Blum, "Ant colony optimization: introduction and recent trends," *Physics of Life Reviews*, vol. 2, no. 4, pp. 353–373, 2005.

[8] M. Dorigo, E. Bonabeau, and G. Theraulaz, "Ant algorithms and stigmergy," *Future Generation Computer Systems*, vol. 16, no. 9, pp. 851–871, 2000.

[9] P. Delisle, M. Gravel, M. Krajecki, C. Gagné, and W. L. Price, "Comparing parallelization of an ACO: message passing vs. shared memory," *Lecture Notes in Computer Science*, vol. 3636, pp. 1–12, 2005.

[10] T. Stützle, "Parallelization strategies for ant colony optimization," *Lecture Notes in Computer Science*, vol. 1498, pp. 722–741, 1998.

[11] M. Randall and A. Lewis, "A parallel implementation of ant colony optimization," *Journal of Parallel and Distributed Computing*, vol. 62, no. 9, pp. 1421–1432, 2002.

[12] E.-G. Talbi, O. Roux, C. Fonlupt, and D. Robillard, "Parallel ant colonies for the quadratic assignment problem," *Future Generation Computer Systems*, vol. 17, no. 4, pp. 441–449, 2001.

[13] K. Taškova, P. Korošec, and J. Šilc, "A distributed multilevel ant colonies approach," *Informatica*, vol. 32, no. 3, pp. 307–317, 2008.

[14] Y. Lin, H.-C. Cai, J. Xiao, and J. Zhang, "Pseudo parallel ant colony optimization for continuous functions," in *Proceedings of the Third International Conference on Natural Computation*, Haikou, China, August 2007, vol. 4, pp. 494–500.

[15] P. Korošec and J. Šilc, "A distributed ant-based algorithm for numerical optimization," in *Proceedings of the 2009 ACM/IEEE International Conference on Automatic Computing*, Barcelona, Spain, June 2009, pp. 37–44.

[16] P. Korošec, J. Šilc, and B. Filipič, "The differential ant-stigmergy algorithm," *Information Sciences*, doi:10.1016/j.ins.2010.05.002, 2011.

[17] B. Chapman, G. Jost, and R. van der Pas, *Using OpenMP Portable Shared Memory Parallel Programming*, MIT Press, 2007.

[18] P. Korošec and J. Šilc, "The differential ant-stigmergy algorithm applied to dynamic optimization problems," in *Proceedings of the IEEE Congress on Evolutionary Computation (CEC 2008)*, Trondheim, Norway, May 2009, pp. 407–414.

[19] P. Korošec, J. Šilc, and K. Tashkova, "The differential ant-stigmergy algorithm for large-scale global optimization," in *Proceedings of the IEEE World Congress on Computational Intelligence (WCCI 2010)*, Barcelona, Spain, July 2010, pp. 4288–4294.