A Statistical Study of Concurrent Differential Evolution on Multi-core CPUs

Kiyoharu Tagawa

School of Science and Engineering,
Kinki University, Higashi-Osaka 577-8502, Japan
tagawa@info.kindai.ac.jp

Abstract. In order to utilize multi-core CPUs, a concurrent program of the latest evolutionary algorithm, i.e., Differential Evolution (DE), is described. The concurrent program of DE is based on a programming model known as "MapReduce" and named Concurrent DE (CDE). In this paper, two implementation techniques of CDE, namely CDE/D and CDE/S, are presented and compared in the quality of the solutions. Through the numerical experiments and the statistical tests conducted on two kinds of popular multi-core CPUs, it is shown that CDE/D is superior to CDE/S in the quality of the solutions obtained by CDE, however CDE/D is inferior to CDE/S in the execution time of CDE.

Keywords: concurrent program, parallel processing, multi-core CPU, evolutionary algorithm, differential evolution, non-parametric test

1 Introduction

Differential Evolution (DE), which was proposed originally by R. Storn and K. V. Price [1], is arguably one of the most powerful stochastic real-parameter optimization algorithms in current use. DE is regarded as a kind of Evolutionary Algorithm (EA). However, comparing with conventional EAs such as genetic algorithm, evolution strategy, and particle swarm optimization, it has been reported that DE exhibits an overall excellent performance for a wide range of benchmark problems [2]. Because of its simple but powerful searching capability, DE has gotten numerous science and engineering applications [2, 3].

Because EAs maintain a lot of individuals manipulated competitively in the population, EAs have a parallel and distributed nature intrinsically. Therefore, many parallelization techniques have been contrived for various EAs [4, 5]. These parallelization techniques of EAs can be introduced easily into DE variants [6]. Actually, the parallel implementations of DE variants using networked computers and computer clusters have been reported [7–10]. Besides, Graphics Processing Units (GPUs) designed originally to accelerate graphics applications with several hundreds of simplified cores have been also used to run a parallel program of DE consisting of many tasks realized by threads executable in parallel [11, 12].

Recently, multi-core CPUs, which have more than one processor (core), have been introduced widely into personal computers. Therefore, in order to utilize

the additional cores to execute costly application programs, concurrent implementations of them have been paid attention to [13]. Even though the number of available cores is not so large, the concurrent program executed on a multi-core CPU is the most simple and easy way to realize a parallelized DE. Therefore, in our previous paper [14], a concurrent program of DE, which is called Concurrent Differential Evolution (CDE), was proposed. Exactly the proposed CDE is a parallelized version of a neoteric DE based on the steady-state model.

The procedure of EAs for updating the individuals in the population is called "generation alternation model". Many EAs usually employ either of two types of generation alternation models [15]. The first one is called "generational model", while the second one is called "steady-state model". The original DE has been based on the generational model [1]. According to the generational model, DE holds two populations, namely current one and auxiliary one. After generating all individuals of the auxiliary population from those of the current population, the current one is replaced all together by the auxiliary one. On the other hand, a neoteric DE based on the steady-state model has been reported and studied lately [16–18]. The neoteric DE is sometimes called Sequential DE (SDE) [16]. According to the steady-state model, SDE holds only one population. Then each individual of the population is updated one by one. Comparing with the generational model, the steady-state model is usually suitable for parallelizing the procedures of EAs [19]. That is because EAs based on the steady-state model need not synchronize the manipulations of all individuals in the current population for replacing them by newborn individuals simultaneously.

In our previous paper [14], through the numerical experiment conducted on a multi-core CPU, it was demonstrated that the execution time of CDE was reduced as the number of threads increased. However, the quality of solutions obtained by CDE had not been considered. This paper focuses on the quality of solutions obtained by CDE as well as the execution time of CDE. In order to analyze the quality of solutions obtained by CDE, a non-parametric test is employed. The major contributions of this paper include the following:

1. Two implementation techniques of CDE, namely CDE/D and CDE/S, are presented. CDE/D allocates the manipulations of individuals to respective threads dynamically, while CDE/S allocates them to all threads statically.
2. The performances of CDE/D and CDE/S are compared in both the execution time and the quality of solutions by using two kinds of popular multi-core CPUs: Intel(R) Core(TM) i7 and AMD Phenom(TM) II X6.
3. Through the numerical experiment and the statistical test, it is shown that CDE/D is superior to CDE/S in the quality of solutions obtained by CDE, however CDE/D is inferior to CDE/S in the execution time of CED.

The remainder of this paper is organized as follows. Section 2 describes the procedure of SDE. Section 3 explains the concurrent program of SDE called CDE. The above two implementation techniques of CDE are also presented. Through numerical experiments, CDE/D and CDE/S are compared on two kinds of multi-core CPUs in Section 4. The results of the numerical experiments are analyzed by using statistical tests in Section 5. Section 6 concludes the paper.

2 Differential Evolution

2.1 Representation

The real-parameter optimization problem can be formulated as shown in (1). The optimal solution of the optimization problem is a D-dimensional real-parameter vector $\boldsymbol{x} = (x_1, \cdots, x_D) \in \mathbb{R}^D$ that minimizes the objective function value $f(\boldsymbol{x}) \in \mathbb{R}$. Furthermore, the value of each decision variable $x_j \in \mathbb{R}$ is limited to the range between the lower $\underline{x}_j$ and the upper $\overline{x}_j$ boundaries as

$$\begin{cases} \text{minimize } f(\boldsymbol{x}) = f(x_1, \cdots, x_D) \\ \text{sub. to } \ \underline{x}_j \leq x_j \leq \overline{x}_j, \ j = 1, \cdots, D. \end{cases} \tag{1}$$

DE is used to solve the optimization problem shown in (1). DE holds N_P tentative solutions of the optimization problem, which are called individuals, in the population $\mathbf{P}$. Therefore, the i-th individual $\boldsymbol{x}_i \in \mathbf{P}$ is represented as

$$\boldsymbol{x}_i = (x_{1,i}, \cdots, x_{j,i}, \cdots, x_{D,i}) \tag{2}$$

where, $\underline{x}_j \leq x_{j,i} \leq \overline{x}_j, \ j = 1, \cdots, D; \ i = 1, \cdots, N_P$.

2.2 Strategy of DE

In order to generate a candidate for a new individual of the population, DE uses a unique reproduction procedure called strategy. The strategy of DE is defined by a series of three genetic operators, namely reproduction selection, differential mutation, and crossover. Even though various strategies have been proposed for DE [2, 16], a basic strategy named "DE/rand/1/exp" is described and used in this paper. That is because our experimental studies have shown that the basic strategy has relatively good compatibility with Sequential DE (SDE) [18].

In the reproduction selection, a parent individual called "the target vector" is selected from the population $\mathbf{P}$ in turn. Besides, three different individuals, say $\boldsymbol{x}_{i1}$, $\boldsymbol{x}_{i2}$ and $\boldsymbol{x}_{i3} \in \mathbf{P}$ ($i \neq i1 \neq i2 \neq i3$), are selected randomly from $\mathbf{P}$.

By using the above three individuals, namely $\boldsymbol{x}_{i1}$, $\boldsymbol{x}_{i2}$ and $\boldsymbol{x}_{i3}$, the differential mutation generates a mutated vector $\boldsymbol{v} = (v_1, \cdots, v_j, \cdots, v_D)$ as

$$\boldsymbol{v} = \boldsymbol{x}_{i1} + F\,(\boldsymbol{x}_{i2} - \boldsymbol{x}_{i3}) \tag{3}$$

where, the scale factor $F \in (0, \ 1+]$ is a control parameter.

The exponential crossover between the mutated vector $\boldsymbol{v}$ and the target vector $\boldsymbol{x}_i$ generates a candidate for a new individual $\boldsymbol{u} = (u_1, \cdots, u_j, \cdots, u_D)$ called "the trial vector". Each component u_j of the trial vector $\boldsymbol{u}$ is inherited from either the mutated vector $\boldsymbol{v}$ or the target vector $\boldsymbol{x}_i$. The pseudocode in (4) gives the procedure of the exponential crossover combined with the differential mutation shown in (3). The subscript $j_r \in [1, \ D]$ in (4) is selected randomly, which ensures that the trial vector $\boldsymbol{u}$ differs from the target vector $\boldsymbol{x}_i \in \mathbf{P}$ at least one component u_{j_r}. Furthermore, rand[0, 1] in (4) denotes the random number

generator that returns a uniformly distributed random number from within the range between 0 and 1. As well as the scale factor F in (3), the crossover rate $C_R \in [0,\ 1]$ is a control parameter specified by the user in advance.

$$\left\lceil \begin{array}{l} j = j_r; \\ \text{do } \{ \\ \quad u_j = x_{j,i1} + F\,(x_{j,i2} - x_{j,i3});\ j = j\,\%\,D + 1; \\ \} \text{ while}(\text{rand}[0,\ 1] < C_R\ \wedge\ j \neq j_r) \\ \text{while}(j \neq j_r)\ \{ \\ \quad u_j = x_{j,i};\ j = j\,\%\,D + 1; \\ \} \end{array} \right. \tag{4}$$

If a component u_j of the trial vector $\boldsymbol{u}$ comes out of the range $[\underline{x}_j,\ \overline{x}_j]$ as the result of the strategy shown in (4), it is returned to the range as

$$u_j = \begin{cases} x_{j,r1} + \text{rand}[0,\ 1]\,(\underline{x}_j - x_{j,i1}) & \text{if}(u_j < \underline{x}_j) \\ x_{j,r1} + \text{rand}[0,\ 1]\,(\overline{x}_j - x_{j,i1}) & \text{if}(u_j > \overline{x}_j) \end{cases} \tag{5}$$

2.3 Procedure of SDE

Since the proposed CDE is a concurrent program of SDE, we explain SDE instead of the original DE. The procedure of SDE can be described as follows:

1: Randomly generate N_P individuals $\boldsymbol{x}_i \in \mathbf{P}$ $(i = 1, \cdots, N_P)$.
2: For each individual $\boldsymbol{x}_i \in \mathbf{P}$, evaluate the objective function value $f(\boldsymbol{x}_i)$.
3: Set the generation as $g = 0$.
4: For each $\boldsymbol{x}_i \in \mathbf{P}$, i.e., the target vector, execute from Step 4.1 to Step 4.3.
  4. 1: Randomly select $\boldsymbol{x}_{i1}$, $\boldsymbol{x}_{i2}$ and $\boldsymbol{x}_{i3} \in \mathbf{P}$ $(i \neq i1 \neq i2 \neq i3)$.
  4. 2: Generate the trial vector $\boldsymbol{u}$ from (4) and (5). Evaluate $f(\boldsymbol{u})$.
  4. 3: If $f(\boldsymbol{u}) \leq f(\boldsymbol{x}_i)$ holds then replace $\boldsymbol{x}_i \in \mathbf{P}$ by $\boldsymbol{u}$ and let $f(\boldsymbol{x}_i) = f(\boldsymbol{u})$.
5: If $g < G_M$ then set $g = g + 1$ and return to Step 4.
6: Output the best $\boldsymbol{x}_b \in \mathbf{P}$ with the minimum $f(\boldsymbol{x}_b)$ and terminate.

where, the maximum number of generation G_M is also a control parameter.

3 Concurrent Differential Evolution

3.1 Model of Multi-core CPU

A program is said to be concurrent if it can support two or more tasks in process at the same time. On the other hand, a program is said to be parallel if it support two or more tasks executing simultaneously. The difference between these definitions is the phrase in progress [13]. A concurrent program evokes multiple independent tasks, which are called "threads", at the same time. If the concurrent program runs on a multi-core CPU, the processing of each thread is assigned automatically to one core. Therefore, if the multi-core CPU has N_T $(N_T \geq 1)$ cores, N_T threads can be executed in parallel at the maximum.

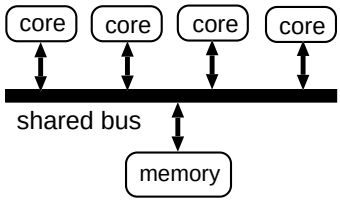


Fig. 1. Parallel Random Access Machine (PRAM)

For designing concurrent programs, Parallel Random Access Machine (PRAM) is often used to model a multi-core CPU [13]. Figure 1 illustrates a configuration of PRAM in which multiple cores attached to an unlimited memory that is shared among all the cores. Besides, Concurrent Read and Exclusive Write (CREW) is assumed to access the memory of PRAM. Therefore, multiple threads running on respective cores may read from the same memory location at the same time and only one thread may write to a given memory location at any time.

3.2 Main Routine of CDE

As stated above, CDE is a concurrent program of SDE. Except the detailed procedure of `Worker(n)`, the main routine of CDE is described as

1: Randomly generate N_P individuals $\boldsymbol{x}_i \in \mathbf{P}$ $(i = 1, \cdots, N_P)$.
2: Evoke `Worker(n)` $(n = 1, \cdots, N_T)$ in parallel.
3: Wait until every `Worker(n)` $(n = 1, \cdots, N_T)$ is completed.
4: Output the best $\boldsymbol{x}_b \in \mathbf{P}$ with the minimum $f(\boldsymbol{x}_b)$ and terminate.

CDE is based on a programming model known as "MapReduce" [20]. The programming model consists of two phase, namely Map-phase and Reduce-phase. In the main routine of CDE, Step 2 corresponds to the Map-phase, while Step 4 corresponds to the Reduce-phase. Each `Worker(n)` is realized by one thread. Therefore, N_T threads are evoked and executed concurrently in Step 2.

`Worker(n)` generates the trial vector $\boldsymbol{u}$ from the assigned target vector $\boldsymbol{x}_i$, evaluate $f(\boldsymbol{u})$, and updates the individual in the population. For assigning the target vector $\boldsymbol{x}_i \in \mathbf{P}$ to `Worker(n)`, we propose the following two techniques, namely the dynamic allocation and the static allocation of tasks.

3.3 Dynamic Allocation of Tasks

CDE with the dynamic allocation of tasks is named CDE/D. CDE/D allocates the target vector $\boldsymbol{x}_i \in \mathbf{P}$ to `Worker(n)` over time as CDE/D executes. `GetIndex()` denotes an exclusive function that returns an unique index at a time in ascending order such as $t = 1,\ 2,\ \cdots$. By using `GetIndex()`, `Worker(n)` gets an index of the assigned target vector dynamically. According to the access rule of CREW, `Worker(n)` has to overwrite $\boldsymbol{x}_i \in \mathbf{P}$ and $f(\boldsymbol{x}_i)$ in Step 3.4 under the exclusion control. The procedure of `Worker(n)` is described as

1: Get an index such as $t = \texttt{GetIndex()}$.
2: While $t \leq N_P$ holds, evaluate $f(\boldsymbol{x}_t)$ and let $t = \texttt{GetIndex()}$.
3: While $t \leq (G_M + 1)\, N_P$ holds, execute from Step 3.1 to Step 3.5.
3. 1 Designate the target vector such as $\boldsymbol{x}_i$ $(i = t\%N_P + 1)$.
3. 2 Randomly select $\boldsymbol{x}_{i1}$, $\boldsymbol{x}_{i2}$ and $\boldsymbol{x}_{i3} \in \mathbf{P}$ $(i \neq i1 \neq i2 \neq i3)$.
3. 3 Generate the trial vector $\boldsymbol{u}$ from (4) and (5). Evaluate $f(\boldsymbol{u})$.
3. 4 If $f(\boldsymbol{u}) \leq f(\boldsymbol{x}_i)$ holds then replace $\boldsymbol{x}_i \in \mathbf{P}$ by $\boldsymbol{u}$ and let $f(\boldsymbol{x}_i) = f(\boldsymbol{u})$.
3. 5 Get an index such as $t = \texttt{GetIndex()}$.

3.4 Static Allocation of Tasks

CDE with the static allocation of tasks is named CDE/S. First of all, the population $\mathbf{P}$ is divided into N_T sub-population $\mathbf{P}_n$ called "chunks" as

$$\mathbf{P} = \mathbf{P}_1 \cup \cdots \cup \mathbf{P}_n \cup \cdots \cup \mathbf{P}_{N_T} \tag{6}$$

CDE/S allocates each chunk $\mathbf{P}_n$ to `Worker(n)` statically. `Worker(n)` can read any individuals $\boldsymbol{x}_i \in \mathbf{P}$, but it may overwrite only the individuals $\boldsymbol{x}_i \in \mathbf{P}_n$. In other words, the task for updating the individuals in one chunk $\mathbf{P}_n$ is permitted only to `Worker(n)`. Therefore, the exclusion control among multiple threads is not necessary for CDE/S. The procedure of `Worker(n)` is described as

1: For each individual $\boldsymbol{x}_i \in \mathbf{P}_n$, evaluate the objective function value $f(\boldsymbol{x}_i)$.
2: Set the generation as $g = 0$.
3: For each $\boldsymbol{x}_i \in \mathbf{P}_n$, i.e., the target vector, execute from Step 3.1 to Step 3.3.
3. 1 Randomly select $\boldsymbol{x}_{i1}$, $\boldsymbol{x}_{i2}$ and $\boldsymbol{x}_{i3} \in \mathbf{P}$ $(i \neq i1 \neq i2 \neq i3)$.
3. 2 Generate the trial vector $\boldsymbol{u}$ from (4) and (5). Evaluate $f(\boldsymbol{u})$.
3. 3 If $f(\boldsymbol{u}) \leq f(\boldsymbol{x}_i)$ holds then replace $\boldsymbol{x}_i \in \mathbf{P}_n$ by $\boldsymbol{u}$ and let $f(\boldsymbol{x}_i) = f(\boldsymbol{u})$.
4: If $g < G_M$ then set $g = g + 1$ and return to Step 3.

4 Numerical Experiments

4.1 Benchmark Problems

The following two test functions are used as the objective function $f(\boldsymbol{x})$ in (1). Both benchmark problems have $D = 30$ dimensional real-parameters. The function values of the optimal solutions are known as $f_p(\boldsymbol{x}) = 0$ $(p = 1,\ 2)$.

- Sphere function (unimodal function):

$$f_1(\boldsymbol{x}) = \sum_{j=1}^{D} x_j^2$$
$$-100 \leq x_j \leq 100,\ j = 1, \cdots, D.$$

- Griewank function (multimodal function):

$$f_4(\boldsymbol{x}) = \frac{1}{4000} \sum_{j=1}^{D} x_j^2 - \prod_{j=1}^{D} \cos\left(\frac{x_j}{\sqrt{j}}\right) + 1$$
$$-600 \leq x_j \leq 600,\ j = 1, \cdots, D.$$

Table 1. Specifications for Personal Computers (PCs)

PC	CPU	OS	clock	memory
PC1	Intel(R) Core(TM) i7	WindowsXP	3.34GHz	2.99GB
PC2	AMD Phenom(TM) II X6	Windows7	3.20GHz	3.25GB

4.2 Experimental Results

The programs of SDE, CDE/D and CDE/S were coded by the Java language, which is a very popular language supporting multiple threads, and executed on two kinds of Personal Computers (PCs) equipped with different multi-core CPUs. Table 1 summarizes the specifications for the two PCs which are denoted by PC1 and PC2. The multi-core CPU in PC1 has four cores each of which manipulates two threads at the same time, while the multi-core CPU in PC2 has six cores. The control parameters of SDE, CDE/D and CDE/S were chosen as $F = 0.5$, $C_R = 0.9$, $N_P = 144$ and $G_M = 1000$. Then the three methods were applied respectively to the two benchmark problems 50 times.

Tables 2-5 show the results of the numerical experiments conducted on PC1. Table 2 shows the execution times of SDE and CDE/D averaged over 50 independent runs, where the standard deviation of the execution times also appear in parentheses. Table 3 shows the execution time of CDE/S in the same way with Table 2. On the other hand, Table 4 and Table 5 show the average and the standard deviation of the objective function values of the best solutions obtained by CDE/D and CDE/S respectively. Similarly, Tables 6-9 show the results of the numerical experiments about CDE/D and CDE/S conducted on PC2.

In order to evaluate the performance of CDE in the execution time, the speedup defined by (7) is used [14]. T_m denotes the execution time of SDE, while $T_m(N_T)$ denotes the execution time of CDE using N_T ($N_T \geq 1$) threads. Both T_m and $T_m(N_T)$ are averaged over $m = 50$ independent runs.

$$S_m(N_T) = \frac{T_m}{T_m(N_T)} \tag{7}$$

Figure 2 plots the speedup curves achieved by CDE/D and CDE/S on PC1, which are calculated from Table 2 and Table 3. Figure 3 also plots the speedup curves achieved by CDE/D and CDE/S on PC2, which are calculated from Table 6 and Table 7. From the speedup curves in Fig. 2 and Fig. 3, it can be confirmed that CDE/S utilizes multi-core CPUs more efficiently than CDE/D on both PCs because CDE/S doesn't spend any overhead for the exclusion control.

5 Non-Parametric Tests

Since CDE is a stochastic algorithm as well as the other EAs, it is desirable to verify the results of the numerical experiments by using statistical tests. Even though parametric tests such as the analysis of variance (ANOVA) have

Table 2. Execution times of CDE/D and SDE on PC1 [ms]

f_p	SDE	CDE/D				
		$N_T = 1$	$N_T = 2$	$N_T = 4$	$N_T = 6$	$N_T = 8$
f_1	146.26 (8.79)	146.86 (8.86)	128.74 (7.38)	94.63 (5.97)	80.30 (5.53)	85.62 (9.12)
f_2	361.58 (7.05)	362.50 (9.53)	214.06 (9.60)	128.44 (7.90)	99.68 (7.67)	90.00 (7.50)

Table 3. Execution times of CDE/S and SDE on PC1 [ms]

f_p	SDE	CDE/S				
		$N_T = 1$	$N_T = 2$	$N_T = 4$	$N_T = 6$	$N_T = 8$
f_1	146.26 (8.79)	145.62 (7.89)	100.62 (8.97)	62.50 (8.29)	54.68 (7.97)	46.24 (6.22)
f_2	361.58 (7.05)	361.26 (6.83)	199.06 (9.32)	111.26 (7.60)	92.18 (7.84)	74.70 (7.28)

Table 4. Objective function values obtained by CDE/D and SDE on PC1

f_p	SDE	CDE/D				
		$N_T = 1$	$N_T = 2$	$N_T = 4$	$N_T = 6$	$N_T = 8$
f_1	4.32E-10 (1.27E-10)	4.32E-10 (1.27E-10)	4.66E-10 (1.52E-10)	4.64E-10 (1.40E-10)	4.88E-10 (1.77E-10)	4.40E-10 (1.39E-10)
f_2	8.94E-8 (4.48E-7)	8.94E-8 (4.47E-7)	9.59E-9 (1.86E-8)	2.04E-8 (6.11E-8)	1.80E-8 (5.31E-8)	1.07E-7 (6.20E-7)

Table 5. Objective function values obtained by CDE/S and SDE on PC1

f_p	SDE	CDE/S				
		$N_T = 1$	$N_T = 2$	$N_T = 4$	$N_T = 6$	$N_T = 8$
f_1	4.32E-10 (1.27E-10)	4.32E-10 (1.27E-10)	4.53E-10 (1.33E-10)	1.20E-9 (2.50E-9)	9.78E-10 (5.33E-10)	8.09E-10 (3.31E-10)
f_2	8.94E-8 (4.48E-7)	8.94E-8 (4.48E-7)	2.06E-8 (6.13E-8)	8.99E-9 (1.29E-8)	3.47E-8 (5.38E-8)	1.78E-8 (4.18E-8)

Table 6. Execution times of CDE/D and SDE on PC2 [ms]

f_p	SDE	CDE/D				
		$N_T = 1$	$N_T = 2$	$N_T = 4$	$N_T = 6$	$N_T = 8$
f_1	196.40 (7.77)	195.18 (7.71)	205.18 (15.34)	146.78 (7.74)	179.56 (8.57)	187.98 (3.13)
f_2	401.24 (7.84)	399.26 (8.80)	273.72 (11.80)	178.94 (7.90)	169.26 (6.64)	181.12 (8.46)

Table 7. Execution times of CDE/S and SDE on PC2 [ms]

f_p	SDE	CDE/S				
		$N_T = 1$	$N_T = 2$	$N_T = 4$	$N_T = 6$	$N_T = 8$
f_1	196.40	192.34	175.18	94.92	70.88	65.58
	(7.77)	(7.96)	(18.28)	(9.91)	(9.58)	(6.29)
f_2	401.24	398.76	252.94	143.64	103.34	106.80
	(7.84)	(8.95)	(10.97)	(7.09)	(8.91)	(8.74)

Table 8. Objective function values obtained by CDE/D and SDE on PC2

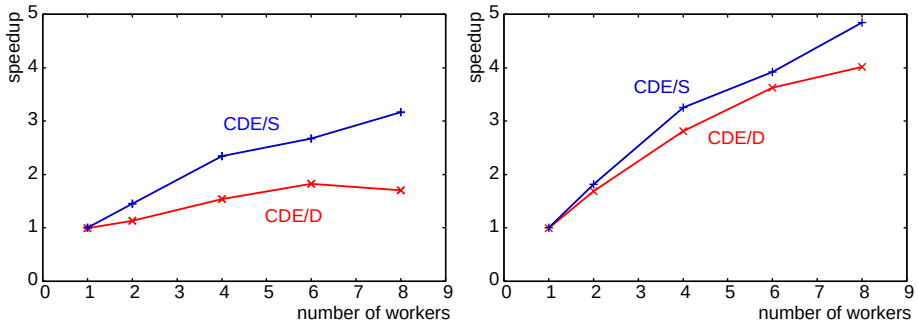
f_p	SDE	CDE/D				
		$N_T = 1$	$N_T = 2$	$N_T = 4$	$N_T = 6$	$N_T = 8$
f_1	4.32E-10	4.32E-10	4.92E-10	4.79E-10	4.95E-10	5.07E-10
	(1.27E-10)	(1.27E-10)	(1.31E-10)	(1.34E-10)	(1.72E-10)	(1.62E-10)
f_2	8.94E-8	8.94E-8	1.36E-8	1.42E-8	1.14E-8	2.29E-8
	(4.48E-7)	(4.47E-7)	(3.32E-8)	(2.44E-8)	(1.44E-8)	(5.39E-8)

Table 9. Objective function values obtained by CDE/S and SDE on PC2

f_p	SDE	CDE/S				
		$N_T = 1$	$N_T = 2$	$N_T = 4$	$N_T = 6$	$N_T = 8$
f_1	4.32E-10	4.32E-10	4.18E-10	4.09E-9	4.01E-9	3.22E-7
	(1.27E-10)	(1.27E-10)	(9.74E-11)	(6.39E-9)	(6.16E-9)	(2.23E-7)
f_2	8.94E-8	8.94E-8	1.45E-8	1.99E-8	2.46E-8	2.19E-6
	(4.48E-7)	(4.48E-7)	(2.10E-8)	(3.85E-8)	(4.40E-8)	(2.51E-6)

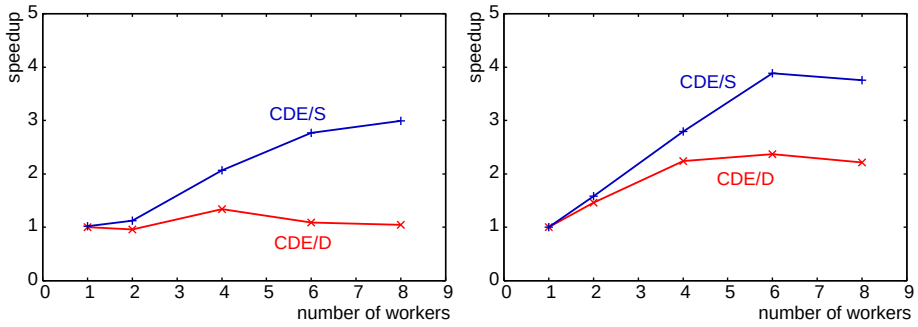
Table 10. Wilcoxon test between SDE and CDE (CDE/D or CDE/S) on PC1

f_p	N_T	1	2	4	6	8
f_1	CDE/D	50.50	52.88	54.04	54.52	50.14
	SDE	50.50	48.12	46.96	46.48	50.86
	P-value	1.00	0.41	0.22	0.16	0.90
	CDE/S	50.50	52.30	69.44	70.16	70.96
	SDE	50.50	48.70	31.56	30.84	30.08
	P-value	1.00	0.53	0.00	0.00	0.00
f_2	CDE/D	50.50	43.88	48.04	47.54	51.02
	SDE	50.50	57.12	52.96	53.46	49.98
	P-value	1.00	0.02	0.39	0.30	0.85
	CDE/S	50.50	45.70	46.02	57.28	45.92
	SDE	50.50	55.30	54.98	43.72	55.08
	P-value	1.00	0.09	0.12	0.01	0.11



(a) Sphere function: f_1 (b) Griewank function: f_2

Fig. 2. Speedup curves achieved by CDE/D and CDE/S on PC1



(a) Sphere function: f_1 (b) Griewank function: f_2

Fig. 3. Speedup curves achieved by CDE/D and CDE/S on PC2

been used widely in the comparison of the behavior of EAs [17, 21, 22], a non-parametric test known as Wilcoxon test is used to analyze the behavior of CDE. That is because non-parametric tests don't require to check any conditions such as independence, normality, and heteroscedasticity in the advance.

From the results of the numerical experiments shown in Tables 4, 5, 8, and 9, the objective function values of the best solutions, i.e., the quality of solutions, seem to depend on the number of threads. Furthermore, the quality of solutions seems to be different between CDE/D and CDE/S. Therefore, the quality of solutions was analyzed statistically by using Wilcoxon test. The null hypothesis of Wilcoxon test was that there was no significant difference between two objective function values obtained by SDE and CDE (CDE/D or CDE/S).

Tables 10-11 show the averaged ranks of the two methods and P-values calculated by Wilcoxon test. The lower the rank is, the smaller the objective function value is. If P-value is larger than 0.01 in Tables 10-11, we can say that there is no significant difference between SDE and CDE in the quality of solutions. In

Table 11. Wilcoxon test between SDE and CDE (CDE/D or CDE/S) on PC2

f_p	N_T	1	2	4	6	8
f_1	CDE/D	50.50	57.62	55.69	54.98	56.52
	SDE	50.50	43.38	45.31	46.02	44.48
	P-value	1.00	0.01	0.07	0.12	0.03
	CDE/S	50.50	49.47	75.44	75.46	75.50
	SDE	50.50	51.53	25.56	25.54	25.50
	P-value	1.00	0.72	0.00	0.00	0.00
f_2	CDE/D	50.50	46.88	47.34	49.90	48.58
	SDE	50.50	54.12	53.66	51.10	52.42
	P-value	1.00	0.21	0.27	0.83	0.50
	CDE/S	50.50	49.02	51.64	54.06	74.44
	SDE	50.50	51.98	49.36	46.94	26.56
	P-value	1.00	0.60	0.69	0.21	0.00

every case of CDE/D, the null hypothesis can't be rejected with the risk less than 0.01 ($P \geq 0.01$). However, in some cases of CDE/S, the null hypothesis can be rejected with the risk less than 0.01. In other words, the quality of solutions obtained by CDE/S depends on the number of threads on both PCs.

6 Conclusion

In this paper, two implementation techniques of CDE, namely CDE/D and CDE/S, were presented and compared in the execution time and the quality of solutions. Through the numerical experiment and the statistical test conducted on two kinds of multi-core CPUs, it was shown that CDE/D was superior to CDE/S in the the quality of solutions, while CDE/S was superior to CDE/D in the execution time. The concurrent program executed on a multi-core CPU can allocate tasks to threads directly but can't assign these threads to cores by itself. Besides, in addition to the concurrent program, several system programs including OS may be running on the same PC. Consequently, the performance of the concurrent program usually depends on the computational environment.

In our future work, we would like to develop a new implementation technique of CDE for maximizing the performance of CDE in both the execution time and the quality of solutions regardless of the computational environment.

References

1. Storn, R. and Price, K.: Differential evolution - a simple and efficient heuristic for global optimization over continuous space. Journal of Global Optimization, 4(11), 341–359 (1997)
2. Price, K. V., Storn, R. M., and Lampinen, J. A.: Differential Evolution - A Practical Approach to Global Optimization, Springer (2005)
3. Das, S. and Suganthan, P. N.: Differential evolution: a survey of the state-of-the art. IEEE Trans. on Evolutionary Computation, 15(1), 4–31 (2011)

4. Cantú-Paz, E.: Efficient and Accurate Parallel Genetic Algorithms. Kluwer Academic Publishers (2001)
5. Alba, E. and Tomassini, M.: Parallelism and evolutionary algorithms. IEEE Trans. on Evolutionary Computation, (6)5, 443–462 (2002)
6. Dorronsoro, B. and Bouvry, P.: Improving classical and decentralized differential evolution with new mutation operator and population topologies. IEEE Trans. on Evolutionary Computation, (15)1, 67–98 (2011)
7. Tasoulis, D. K., Pavlidis, N. G., Plagianakos, V. P., and Vrahatis, M. N.: Parallel differential evolution. In: Proc. of IEEE Congress on Evolutionary Computation, 2023–2029 (2004)
8. Zaharie, D. and Petcu, D.: Parallel implementation of multi-population differential evolution. Concurrent Information Processing and Computing, ISO Press, 223–232 (2005)
9. Zhou, C.: Fast parallelization of differential evolution algorithm using MapReduce. In: Proc. of Genetic and Evolutionary Computation Conference, 1113–1114 (2010)
10. Ishimizu, T. and Tagawa, K.: Experimental study of a structured differential evolution with mixed strategies. Journal of Advanced Computational Intelligence and Intelligent Informatics, 15(9), 1310–1319 (2011)
11. de Veronses, L and Krohling, R.: Differential evolution algorithm on the GPU with C-CUDA. In: Proc. of IEEE Congress on Evolutionary Computation, 1–7 (2010)
12. Krömer, P., Snášel, V., and Platoš, J.: Many-thread implementation of differential evolution for the CUDA platform. In: Proc. of Genetic and Evolutionary Computation Conference, 1595–1602 (2011)
13. Breshears, C.: The Art of Concurrency - A Thread Monkey's Guide to Writing Parallel Applications. O'Reilly (2009)
14. Tagawa, K. and Ishimizu, T.: Concurrent differential evolution based on MapReduce. International Journal of Computers, 4(4), 161–168 (2010)
15. Syswerda, G.: A study of reproduction in generational and steady-state genetic algorithms. Foundations of Genetic Algorithms 2, Morgan Kaufmann Publ., 94–101 (1991)
16. Feoktistov, V.: Differential Evolution in Search Solutions. Chapter 6, Springer (2006)
17. Tagawa, K.: A statistical study of the differential evolution based on continuous generation model. In: Proc. of IEEE Congress on Evolutionary Computation, 2614–2621 (2009)
18. Tagawa, K. and Ishimizu, T.: A comparative study of distance dependent survival selection for sequential DE. In: Proc. of IEEE International Conference on System, Man, and Cybernetics, 3493–3500 (2010)
19. Davison, B. D. and Rasheed, K.: Effect of global parallelism on a steady state GA. In: Proc. of Genetic and Evolutionary Computation Conference Workshops, Evolutionary Computation and Parallel Processing Workshop, 167–170 (1999)
20. Dean, J. and Ghemawat, S.: MapReduce: simplified data processing on large clusters. In: Proc. of 6th Symposium on Operating Systems Design and Implementation, 137–149 (2010)
21. Rojas, I., Gonzalez, J., Pomares, H., Merelo, J. J., Castillo, P. A., and Romero, G.: Statistical analysis of the main parameters involved in the design of a genetic algorithm. IEEE Trans. on Systems, Man, and Cybernetics, Part C: Applications and Reviews, 32(1), 31–37 (2002)
22. Czarn, A., MacNish, C., Vijayan, K., Turlach, B., and Gupta, R.: Statistical exploratory analysis of genetic algorithms. IEEE Trans. on Evolutionary Computation, 8(4), 405–421 (2004)