Concurrent Differential Evolution Based on MapReduce

Kiyoharu Tagawa and Takashi Ishimizu

***Abstract*—**Multi-core processors, which have more than one Central Processing Unit (CPU), have been introduced widely into personal computers. Therefore, in order to utilize the additional cores, or CPUs, to execute various costly application programs, concurrent implementations of them have been paid to attention. MapReduce is a concurrent programming model and an associated implementation for processing and generating large data sets. This paper has been participated in plenary presentation at the conference of WSEAS and is presenting a further progress of a concurrent implementation of Differential Evolution (DE) based on MapReduce. Especially, through the numerical experiment conducted on a wide range of benchmark problems, the speedup of DE due to the use of multiple cores is demonstrated. Furthermore, the goodness of the proposed concurrent implementation of DE is examined and proved with respect to four categories, namely efficiency, simplicity, portability and scalability.

***Keywords*—**Algorithm, concurrent program, differential evolution, evolutionary algorithm, MapReduce, parallel program.

I. Introduction

DIFFERENTIAL Evolution (DE) is one of the most recent Evolutionary Algorithms (EAs) for solving real-parameter optimization problems [1]. Comparing with typical EAs such as Genetic Algorithm (GA), Evolutionary Strategy (ES), and Particle Swarm Optimization (PSO), it has been reported that DE exhibits an overall excellent performance for a wide range of benchmark problems [2], [3]. Furthermore, because of its simple but powerful searching capability, DE has been applied to numerous real-world applications successfully [4]–[6].

The procedure of EA for updating the individuals included in the population is called a “generation model” or a “generation alternation model”. EAs usually employ either of two types of generation models [7]. The first one is called a “generational model” or a “discrete generation model”, while the second one is called a “steady-state model” or a “continuous generation model” [8]. The classic DE proposed originally by R. Storn and K. Price has been based on the discrete generation model [1]. According to the discrete generation model, the classic DE holds two populations, namely the old one and the new one. Then, by using a particular strategy, the individuals of the new population are generated from those of the old one. After that, the old population is replaced by the new one at a time.

Inspired by the great success of the classic DE, a variety of revised DEs have been developed for solving different types of optimization problems such as noisy [9], constrained [4], and multi-objective optimization problems [10], [11]. Furthermore, self-adaptive DEs that have various learning mechanisms to choose appropriate strategies and control parameters [12], [13]. However, many of the conventional DEs have been also based on the discrete generation model as well as the classic DE.

Recently, a new DE based on the continuous generation model is proposed [14], [15]. The new DE is sometimes called “Sequential DE (SDE)” [14]. According to the continuous generation model, SDE holds only one population. Therefore, SDE renews the individuals of the population one by one. Exactly, SDE generates a new individual called the “trial vector” from an existing individual called the “target vector” in the same way with the classic DE. After that, if the target vector included in the population is not better than the trial vector, the target vector is replaced by the trial vector immediately. Since the excellent newborn individual, namely the trial vector, can be used soon to generate offspring, it can be expected that SDE finds good solutions faster than the classic DE [8].

Multi-core processors, which have more than one Central Processing Unit (CPU), have been introduced widely into personal computers. Therefore, in order to utilize the additional CPUs to execute costly application programs such as EAs applied to real-world applications, concurrent implementations of them have been paid to attention [16]. Because EAs including DE maintain a lot of individuals manipulated competitively in the population, EAs have a parallel and distributed nature intrinsically. Therefore, many parallelization techniques of various EAs have been reported [17], [18]. Actually, a parallel implementation of the classic DE has been also proposed by using Parallel Virtual Machine (PVM) [19]. However, the concurrent implementation of DE executable on a multi-core processor has not been reported yet.

In this paper, a concurrent implementation of DE is proposed. Although various DEs have been reported [11], SDE is chosen for the proposed concurrent implementation. That is because the survival selection of SDE comparing the trial vector only with the target vector enables us to manipulate all the individuals in the population independently. Besides, SDE need not synchronize the manipulations of the individuals in each generation for replacing the old population by the new one. Therefore, comparing with the classic DE, SDE is especially suited for concurrent programming. The proposed concurrent implementation of SDE is based on MapReduce. Incidentally, MapReduce is a modern concurrent programming model and an associated implementation for processing and generating large data sets [20]. Finally, through the numerical experiment conducted on various benchmark problems, the speedup of SDE due to the use of multiple CPUs is demonstrated. Besides, the goodness of the proposed concurrent implementation of SDE is examined and proved with respect to four categories, namely efficiency, simplicity, portability and scalability.

The rest of the paper is organized as follows. Section II describes the classic DE and a basic strategy of DE. The basic strategy is also used by SDE for generating a new individual. Section III describes SDE. For designing concurrent and parallel programs, Section IV presents a model of the multi-core processor and explains MapReduce briefly. The concurrent implementation of SDE, which is called Concurrent DE (CDE), is proposed in Section V. Comparing with SED and the classic DE, the performance of the proposed CDE is evaluated through the numerical experiment in Section VI. Finally, Section VII concludes this paper.

II. Differential Evolution (DE)

A. Representation

The optimal solution of the real-parameter optimization problem is represented by a D-dimensional real parameter vector $\mathbf{x}=(x_0,\cdots,x_{D-1})$ that minimizes the value of the objective function $f(\mathbf{x})$. Besides, the value of each decision variable $x_j \in \Re$ is usually limited to the range between the lower $\underline{x}_j$ and the upper $\overline{x}_j$ boundaries. Therefore, the real-parameter optimization problem can be formulated as

$$\left[\begin{array}{ll}\text{minimize} & f(\mathbf{x})=f(x_0,\cdots,x_{D-1}) \\ \text{subject to} & \underline{x}_j \le x_j \le \overline{x}_j, \quad j=0,\cdots,D-1.\end{array}\right. \tag{1}$$

Differential Evolution (DE) [1], [3] is used to solve the optimization problem shown in (1). As well as conventional real-coded GAs [21], each tentative solution is represented by a real-parameter vector and called an "individual". Furthermore, DE holds N_P individuals within the population. Therefore, an individual $x_i\ (i=0,\cdots,N_P-1)$ is represented as

$$\begin{array}{l}\mathbf{x}_i=(x_{0,i},\cdots,x_{j,i},\cdots,x_{D-1,i}) \\ \text{where, } \underline{x}_j \le x_{j,i} \le \overline{x}_j, \quad j=0,\cdots,D-1.\end{array} \tag{2}$$

The members of an initial population $\mathbf{x}_i \in P$ are generated randomly by using the random number generator that is denoted by $rand_j[0,1]$ and returns a uniformly distributed random number from within the range between 0 and 1 as

```
for ($i=0; i<N_P; i++$) {
    for ($j=0; j<D; j++$) {
        $x_{j,i}=(\overline{x}_j-\underline{x}_j)\,rand_j[0,1]+\underline{x}_j$;          (3)
    }
}
```

B. Strategy of DE

Differential mutation is a unique genetic operator of DE [1]. Besides, a set of three genetic operators, namely reproduction selection, differential mutation and crossover, is usually called the strategy of DE [1]. As we will describe in detail later, SDE is also uses the strategy of DE [8], [14], [15]. Even though various strategies have been contrived for DE [1], [3], a basic strategy named "DE/rand/1/exp" is described and used in this paper. That is because the experimental result shows that the basic strategy has relatively good compatibility with SDE [22]. Of course, any strategies can be used in SDE.

For each of the individuals $\mathbf{x}_i\ (i=0,\cdots,N_P-1)$ within the population, which is also called the target vector, three different individuals, say $\mathbf{x}_{r1}, \mathbf{x}_{r2}$ and $\mathbf{x}_{r3}\ (i\ne r1\ne r2\ne r3)$, are selected randomly from the current population. Then a new individual $\mathbf{u}_i=(u_{0,i},\cdots,u_{D-1,i})$ called the trial vector is generated from the above four individuals, namely $\mathbf{x}_i$, $\mathbf{x}_{r1}$, $\mathbf{x}_{r2}$ and $\mathbf{x}_{r3}$ as

```
$j_r = rand[0, D-1]$;
do {
    $u_{j,i}=x_{j,r1}+S_F\,(x_{j,r2}-x_{j,r3})$;
    $j=(j+1)\%D$;
} while ($rand_j[0,1]\le C_R \wedge j\ne j_r$)          (4)
while ($j\ne j_r$) {
    $u_{j,i}=x_{j,i}$;
    $j=(j+1)\%D$;
}
```

In the basic strategy of DE described in (4), a subscript $j_r\in[0,D-1]$ is selected randomly. Therefore, the trial vector $\mathbf{u}_i$ will be different from the target vector $\mathbf{x}_i$ at least one element that is specified by the subscript $j_r\in[0,D-1]$. Besides the population size $N_R\ (N_R\ge 4)$, the scale factor $S_F\in(0,1+]$ and the crossover rate $C_R\in[0,1]$, which also appear in (4), are the control parameters of DE decided by user in advance.

If an element of the trial vector $\mathbf{u}_i$ generated by the strategy comes out of the range shown in (2), it will be repaired as

$$u_{j,i}=(\overline{x}_j-\underline{x}_j)\,rand_j[0,1]+\underline{x}_j \tag{5}$$

C. Procedure of DE

The procedure of the classic DE [1] can be described by using the pseudo-code in Fig. 1. Because the classic DE is based on the discrete generation model, two populations, namely the old one $\mathbf{x}_i\in P_{old}$ and the new one $\mathbf{z}_i\in P_{new}$, are used. Then the members of the old population are replaced by those of the new one such as $\mathbf{x}_i$=$\mathbf{z}_i$. Furthermore, as the stopping condition, the generation g is limited to the maximum G_M.

III. Sequential DE (SDE)

The procedure of SED [14] can be described by using the pseudo-code shown in Fig. 2. Because SDE is based on the continuous generation model, only one population $\mathbf{x}_i\in P$ is used in Fig. 2. If a newborn trial vector $\mathbf{u}_i$ is not worse than the corresponding target vector $\mathbf{x}_i$, the trial vector $\mathbf{u}_i$ is added to the population immediately. Therefore, the excellent trial vector $\mathbf{u}_i$ can be used soon to generate succeeding trial vectors.

Fig. 3 illustrates the procedure of the classic DE, while Fig. 4 illustrates the procedure of SDE. Comparing the procedure of

SDE in Fig. 4 with that of the classic DE in Fig. 3, we can confirm that SDE obviously saves both the memory space for one population and the processing time spent for replacing the old population $\mathbf{x}_i \in P_{old}$ by the new population $\mathbf{z}_i \in P_{new}$.

```
Rendomly generate $\mathbf{x}_i \in P_{old}$;
for ($i = 0; i < N_P; i++$) {
    Evaluate $f(\mathbf{x}_i)$;
}
for ($g = 0; g < G_M; g++$) {
    for ($i = 0; i < N_P; i++$) {
        Generate $\mathbf{u}_i$ from (4) and (5);
        Evaluate $f(\mathbf{u}_i)$;
        if ($f(\mathbf{u}_i) \le f(\mathbf{x}_i)$)    $\mathbf{z}_i = \mathbf{u}_i$;
        else    $\mathbf{z}_i = \mathbf{x}_i$;
    }
    /*   Update population   */
    for ($i = 0; i < N_P; i++$) {
        $\mathbf{x}_i = \mathbf{z}_i$;
    }
}
Output the best $\mathbf{x}_i \in P_{old}$;
```

Fig. 1 Pseudo-code of the classic DE

```
Randomly generate $\mathbf{x}_i \in P$;
for ($i = 0; i < N_P; i++$) {
    Evaluate $f(\mathbf{x}_i)$;
}
for ($g = 0; g < G_M; g++$) {
    for ($i = 0; i < N_P; i++$) {
        Generate $\mathbf{u}_i$ from (4) and (5);
        Evaluate $f(\mathbf{u}_i)$;
        if ($f(\mathbf{u}_i) \le f(\mathbf{x}_i)$)    $\mathbf{x}_i = \mathbf{u}_i$;
    }
}
Output the best $\mathbf{x}_i \in P$;
```

Fig. 2 Pseudo-code of SDE

IV. Concurrent Program

A. Parallel Random Access Machine

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 can support two or more tasks executing simultaneously [23]. The difference between these definitions is the phrase in progress. A concurrent program may perform multiple tasks in parallel if the concurrent program is executed on a multi-core processor. Therefore, it can be expected that the execution time of an algorithm is reduced by using the concurrent program on the multi-core processor.

For designing parallel or concurrent programs, the multi-core processor is model by the Parallel Random Access Machine (PRAM) [16]. Fig. 5 shows a configuration of PRAM. PRAM has multiple cores, or CPUs, attached to an unlimited memory that is shared among all the CPUs. PRAM uses a shared bus connecting the memory and respective CPUs, where details of the connection mechanism between them are ignored. Therefore, all the threads running on CPUs are assumed to be advancing in lockstep fashion. Furthermore, all the threads running on CPUs are assumed to have the same access time to the memory locations regardless the number of CPUs.

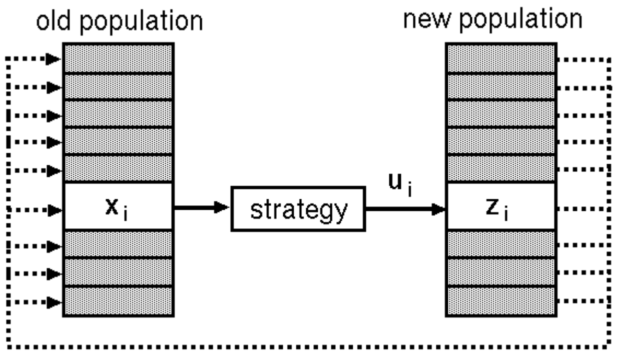

Fig. 3 Procedure of the classic DE

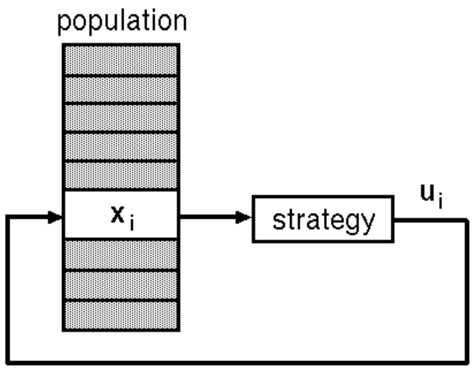

Fig. 4 Procedure of SDE

All the CPUs in PRAM will usually require some form of synchronization and communication in order to cooperate on a given application. However, PRAM makes no assumption about software or hardware support of synchronization objects available to a programmer [16]. Therefore, PRAM need to be specified how threads running on respective CPUs will be able to access memory for both reading and writing. In this paper, we introduce the Concurrent Read and Exclusive Write (CREW) [16] memory access into PRAM. In accordance with CREW, multiple threads running on respective CPUs 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.

B. MapReduce Framework

MapReduce is a modern concurrent programming model and an associated implementation for processing and generating large data sets [20]. Recently, MapReduce is used widely to realize various concurrent and parallel applications [16], [24]. Although many different implementations of MapReduce are possible, the right choice of them depends on both the application and the environment [16]. For example, one implementation of MapReduce may be suitable for a small shared-memory machine that can be modeled by the above PRAM, another for a cluster of commodity machines, and yet another for an even larger collection of networked machines.

MapReduce is a kind of the "divide and conquer" framework for handling a large data set. The procedure of MapReduce consists of two functions, namely "map" and "reduce". First of all, MapReduce running on the master divides a large data set into some blocks and distributes them to multiple workers that execute the map function respectively. A worker corresponds to a processor in the parallel program, but it corresponds to a thread in the concurrent program. The map function takes a set of input (*key*, *value*) pairs associated with the specified "*key*" and produces a set of intermediate (*key*, *value*) pairs. After that, MapReduce groups together all the intermediate pairs and passes them to the reduce function. Finally, the reduce function accepts the intermediate (*key*, *value*) pairs and merges together these values to form a possibly smaller set of values [20].

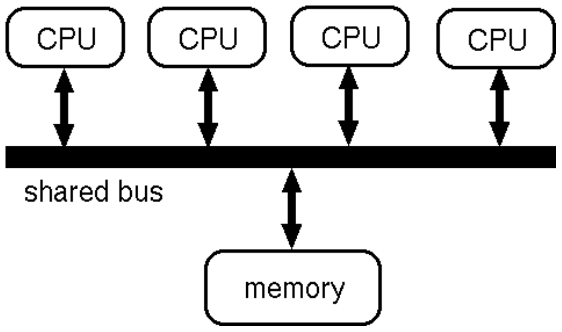

Fig. 5 Parallel random access machine (PRAM)

V. CONCURRENT DE (CDE)

A. Procedure of CDE

First of all, from the results of the numerical experiment and the statistical test conducted on various benchmark problems, there is not a significant difference between the classic DE and SDE in their performances [8]. On the other hand, comparing with the classic DE, SDE is more suitable for the concurrent programming. That is because SDE does not have the procedure replacing the old population by the new one at a time. Therefore, the concurrent implementation of SDE is proposed. As we have mentioned above, the proposed concurrent implementation of SDE is called Concurrent DE (CDE).

The implementation of CDE is based on MapReduce. The input (*key*, *value*) pair corresponds to the structure of individual $\mathbf{x}_i \in P$ associated with the index $i \in [0, N_P - 1]$. Therefore, all the individuals $\mathbf{x}_i \in P$ are distributed to multiple workers, or threads, based on their indexes. At that time, the modulus function of the index tends to result in fairly well-balanced partition. Furthermore, the intermediate (*key*, *value*) pair, which is the output of the map function and the input of the reduce function, corresponds to the objective function value $f(\mathbf{x}_i)$ that is also associated with the index. Finally, the output of the reduce function is the best individual in the population associated with the minimum objective function value.

```
/***  Master  ***/
Randomly generate $\mathbf{x}_i \in P$;
/*  Map phase  */
for all $n$ in parallel do {
  for ($n = 0$; $n < N_T$; $n++$) {
    $Thread(n)$;
  }
}
$Barrier()$;
/*  Reduce phase  */
Output the best $\mathbf{x}_i \in P$;

/***  Worker  ***/
$Thread(n)$ {
  for ($i = 0$; $i < N_P$; $i++$) {
    if ($i \% N_T == n$) {
      Evaluate $f(\mathbf{x}_i)$;
    }
  }
  for ($g = 0$; $g < G_M$; $g++$) {
    for ($i = 0$; $i < N_P$; $i++$) {
      if ($i \% N_T == n$) {
        Generate $\mathbf{u}_i$ from (4) and (5);
        Evaluate $f(\mathbf{u}_i)$;
        if ($f(\mathbf{u}_i) \le f(\mathbf{x}_i)$)  $\mathbf{x}_i = \mathbf{u}_i$;
      }
    }
  }
}
```

Fig. 6 Pseudo-code of CDE

The procedure of CDE can be described by using the pseudo-code shown in Fig. 6. At the beginning of the procedure of CDE, an initial population is generated randomly. Then the population $\mathbf{x}_i \in P$ is divided into N_T blocks called chunks. Therefore, each chunk is regarded as a subpopulation holding

N_P/N_T individuals. The task for updating the individuals included in one chunk is assigned to one thread statically. As a result, N_T tasks are executed concurrently by N_T threads.

In the map phase of CDE shown in Fig. 6, the procedures of $Thread(n)$ $(n = 0, \cdots, N_T - 1)$ may be executed in parallel. Each $Thread(n)$ is assigned to one thread object and contracts the task for updating the individuals included in the chunk. Besides, $Barrier()$ denotes the object that waits until all the procedures of $Thread(n)$ are completed. Finally, in the reduce phase of CDE shown in Fig. 6, the best individual is selected from the final population. Even though the procedure of the reduce phase is described sequentially in Fig. 6, the procedure can be also executed in parallel by using several threads. Fig. 7 shows the procedure of the proposed CDE in case of N_T=3 threads.

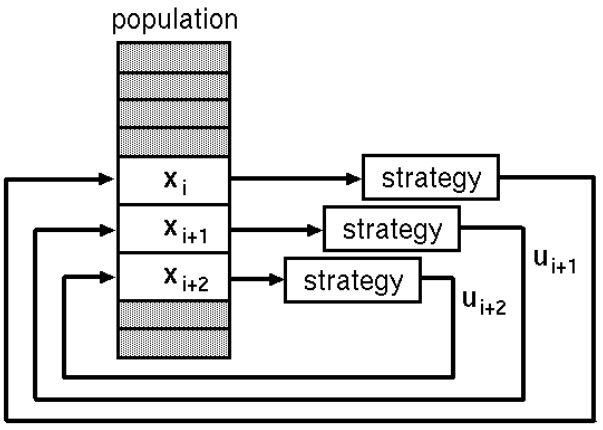

Fig. 7 Procedure of CDE

B. Inspection of CDE

The goodness of a concurrent program is usually examined with respect to four categories, namely efficiency, simplicity, portability and scalability [16]. Therefore, we will examine the proposed CDE with respect to the above four categories.

First of all, EAs that are applied to real-world applications spend the majority of the computational time for evaluating the objective function values. The proposed CDE evenly distributes the objective function evaluation to N_T threads by using the modulus function. Next, each individual in the population $\mathbf{x}_i \in P$ is overwritten only by a unique thread, or $Thread(n)$, that satisfies the condition: $n == i \,\%\, N_T$. On the other hand, every individual can be red from every $Thread(n)$ $(n = 0, \cdots, N_T - 1)$ at a time. Since the mutual exclusion is guaranteed naturally in CDE, any thread need not to synchronize with the other threads. Consequently, the proposed CDE can be regarded as an efficient program.

Comparing CDE with SDE, there is not so much difference in their procedures. Therefore, the proposed CDE is simple as well as SDE. Furthermore, the pseudo-code of CDE can be translated into any program languages that support multiple threads. Therefore, the proposed CDE is portable. Finally, the proposed CDE can be also regarded as scalable. That is because the population $\mathbf{x}_i \in P$ can be divided into multiple chunks of an arbitrary number N_T within the range: $1 \le N_T \le N_P$.

VI. Numerical Experiment

A. Benchmark Problems

In order to evaluate the performance of CDE, the following six benchmark problems are employed. Functions f_0 and f_1 are unimodal, while f_2, f_3, f_4 and f_5 are multimodal. All the benchmark problems have D=30 dimensional real-parameters. Besides, the objective function values of their optimal solutions $\mathbf{x}^*$ are known as follows: $f_p(\mathbf{x}^*) = 0 \quad (p = 0, \cdots, 5)$.

- Sphere function

$$f_0(\mathbf{x}) = \sum_{j=0}^{D-1} x_j^{\,2}$$
$$-100 \le x_j \le 100, \quad j = 0, \cdots, D-1.$$

- Schwefel's Ridge function

$$f_1(\mathbf{x}) = \sum_{j=0}^{D-1} \Big(\sum_{k=0}^{j} x_k\Big)^2$$
$$-100 \le x_j \le 100, \quad j = 0, \cdots, D-1.$$

- Rosenbrock function

$$f_2(\mathbf{x}) = \sum_{j=0}^{D-2} \left(100\,(x_{j+1} - x_j^{\,2})^2 + (x_j - 1)^2\right)$$
$$-30 \le x_j \le 30, \quad j = 0, \cdots, D-1.$$

- Rastrigin function

$$f_3(\mathbf{x}) = \sum_{j=0}^{D-1} \left(x_j^{\,2} - 10\cos(2\,\pi\, x_j) + 10\right)$$
$$-5.12 \le x_j \le 5.12, \quad j = 0, \cdots, D-1.$$

- Ackley function

$$f_4(\mathbf{x}) = -20\exp\left(-0.2\sqrt{\frac{1}{D}\sum_{j=0}^{D-1} x_j^{\,2}}\right) - \exp\left(\frac{1}{D}\sum_{j=0}^{D-1}\cos(2\,\pi\, x_j)\right) + 20 + e$$
$$-32 \le x_j \le 32, \quad j = 0, \cdots, D-1.$$

- Griewank function

$$f_5(\mathbf{x}) = \frac{1}{4000}\sum_{j=0}^{D-1} x_j^{\,2} - \prod_{j=0}^{D-1}\cos\left(\frac{x_j}{\sqrt{j}}\right) + 1$$
$$-600 \le x_j \le 600, \quad j = 0, \cdots, D-1.$$

B. Performance Metric

In order to evaluate the performance of a parallel program executed on a multi-processor machine, the speedup of the parallel program is usually used. The well-known definition of the speedup is the ratio of the serial execution time spent by a single processor to the parallel execution time spent by a set of multiple processors [17]. However, in order to evaluate the performance of a concurrent program executed on a multi-core processor, the conventional definition of the speedup has to be modified [16]. In case of the multiple-core processor, the execution of multiple threads, which are invoked from the concurrent program, is distributed automatically to respective CPUs by the operating system. Although the programmer can specify the number of the threads within his program, he cannot decide the number of CPUs that execute his program.

Furthermore, in order to evaluate the performance of CDE, a single execution of CDE is not statistically significant. That is because CDE is a stochastic algorithm as well as the other EAs. Therefore, the speedup of CDE is defined as follows:

$$S_m(N_T) = \frac{T_m(1)}{T_m(N_T)} \tag{6}$$

In the definition of the speedup in (6), $T_m(1)$ denotes the execution time of SDE averaged over m runs, while $T_m(N_T)$ denotes the execution time of the proposed CDE achieved by using N_T threads. $T_m(N_T)$ is also averaged over m runs. If the speedup $S_m(N_T)$ is close to the number of threads N_T, most threads are executed in parallel through multiple CPUs. On the other hand, if $S_m(N_T)$ is close to one, most threads are executed sequentially. Hence, it is desirable that $S_m(N_T)$ is close to N_T.

C. *Experimental Results*

SDE, CDE and the classic DE are coded by Java language, which is a very popular language supporting multiple threads, and executed on a personal computer equipped with a multi-core processor (CPU: Intel® Core^{TM} i7 @3.33[GHz]; OS: Windows XP). The multi-core processor has four cores that can respectively manipulate two threads at the same time.

In order to measure the speedup defined in (6), SDE and CDE are applied to the six benchmark problems m=20 times respectively. For making a comparative study, the classic DE is also applied to the six benchmark problems in the same way. During the experiments, the control parameters of every DE are fixed as the population size N_P=160, the scale factor S_F=0.5, and the crossover rate C_R=0.9. These values are decided considering the results of our preliminary experiments about SDE [8]. For the stopping condition of every DE, the maximum generation is specified as G_M=10^2, 10^3, and 10^4.

Table I shows the objective function values of the best solutions obtained by CDE, SDE and the classic DE with the maximum generation G_M=10^3. For the proposed CDE, the number of threads is chosen as N_T=2, 4, 8 and 16. Similarly, Table II shows the minimum objective function values obtained by the three DEs with the maximum generation G_M=10^4. The results in Table I and Table II are averaged over 20 runs.

Table III shows the computational times spent by CDE, SDE and the classic DE for obtaining the results shown in Table I. Similarly, Table IV shows the computational times spent by the three DEs for obtaining the results in Table II. The results in Table III and Table VI are also averaged over 20 runs.

The speedup carves achieved by the proposed CDE for the six benchmark problems are plotted in from Fig. 8 to Fig. 13 respectively. As we have mentioned above, in order to evaluate the speedup achieved by CDE, the number of threads is chosen as N_T=2, 4, 8 and 16. Furthermore, three different maximum generations, namely G_M=10^2, 10^3, and 10^4, are specified for the stopping condition of CDE in every benchmark problem.

Table I Objective function value (G_M=10^3)

f_p	CDE (N_T)				SDE	DE
	(2)	(4)	(8)	(16)		
f_0	0.0	0.0	0.0	0.0	0.0	0.0
f_1	53.6	47.9	48.7	135.8	51.2	55.8
f_2	18.5	18.5	18.3	23.8	18.5	19.4
f_3	24.8	23.1	25.2	24.9	24.4	25.2
f_4	0.0	0.0	0.0	0.0	0.0	0.0
f_5	0.0	0.0	0.0	0.0	0.0	0.0

Table II Objective function value (G_M=10^4)

f_p	CDE (N_T)				SDE	DE
	(2)	(4)	(8)	(16)		
f_0	0.0	0.0	0.0	0.0	0.0	0.0
f_1	0.0	0.0	0.0	0.0	0.0	0.0
f_2	0.0	0.0	0.0	0.0	0.0	0.0
f_3	0.0	0.0	0.0	0.0	0.0	0.0
f_4	0.0	0.0	0.0	0.0	0.0	0.0
f_5	0.0	0.0	0.0	0.0	0.0	0.0

Table III Computational time [msec] (G_M=10^3)

f_p	CDE (N_T)				SDE	DE
	(2)	(4)	(8)	(16)		
f_0	93.7	71.8	52.3	47.6	156.2	185.9
f_1	135.1	88.3	66.4	62.5	244.5	278.1
f_2	97.6	71.1	53.1	50.8	162.5	192.9
f_3	216.4	123.4	89.0	92.1	417.9	446.9
f_4	189.0	111.7	82.8	78.9	356.2	385.1
f_5	207.0	120.3	82.0	88.3	392.9	424.2

Table IV Computational time [msec] (G_M=10^4)

f_p	CDE (N_T)				SDE	DE
	(2)	(4)	(8)	(16)		
f_0	1038.2	869.5	590.6	427.3	1618.7	1807.0
f_1	1432.0	972.6	646.8	564.0	2431.3	2758.5
f_2	1084.3	801.5	567.2	456.2	1678.1	1897.6
f_3	1767.2	1066.4	753.1	648.4	3185.9	3457.8
f_4	1589.8	1083.6	752.3	603.1	2762.5	2962.5
f_5	1960.9	1146.1	792.9	710.2	3657.0	3903.9

D. *Discussion of Experimental Results*

From Table I, there is not a significant difference between CDE and SDE in the quality of solutions except one benchmark problem: f_1. In the benchmark problem: f_1, the quality of solutions obtained by CDE depends on the number of threads N_T. In case of SDE, the target vectors $\mathbf{x}_i$ $(i = 0, \cdots, N_P - 1)$ ordered within the population are selected sequentially to generate the trial vectors $\mathbf{u}_i$. However, in case of CDE, the number of threads N_T changes the original order of the target vectors $\mathbf{x}_i$ selected to generate the trial vectors $\mathbf{u}_i$. Therefore, the number of threads N_T

may have an effect on the performance of CDE in some sorts of optimization problems.

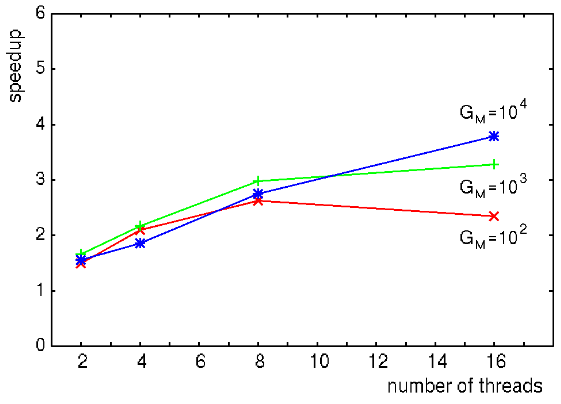

Fig. 8 Speedup by CDE on Sphere function: f_0

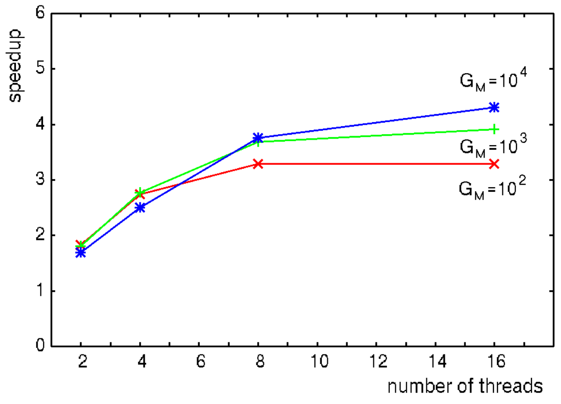

Fig. 9 Speedup by CDE on Schwefel's Ridge function: f_1

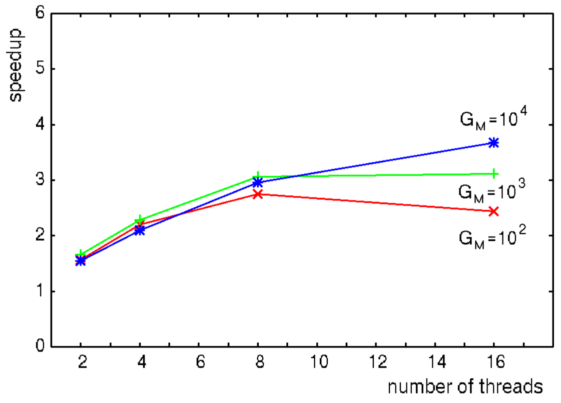

Fig. 10 Speedup by CDE on Rosenbrock function: f_2

From Table II, we can see that every DE has found the optimal solutions for all the benchmark problems. Therefore, the proposed CDE has an advantage over other DEs only in the computational time, because CDE can utilize multiple CPUs.

From Table III and Table IV, we can confirm the advantage of the proposed CDE in the computational time. From Table III, the most efficient number of threads is either N_T=8 or N_T=16 depending on the benchmark problems. On the other hand, from Table IV, CDE is the most efficient with N_T=16 threads for all the benchmark problems. Therefore, the desirable number of threads N_T depends not only on the benchmark problem but also on the maximum generation.

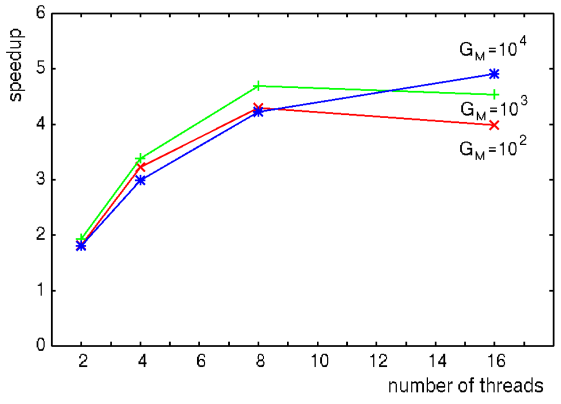

Fig. 11 Speedup by CDE on Rastrigin function: f_3

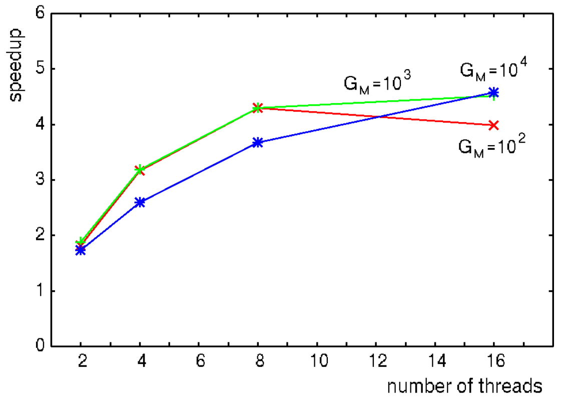

Fig. 12 Speedup by CDE on Ackley function: f_4

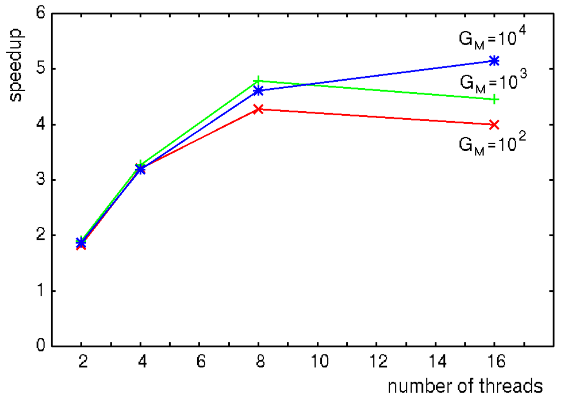

Fig. 13 Speedup by CDE on Griewank function: f_5

From the speedup carves shown in from Fig. 8 to Fig. 13, we can confirm that the speedup is larger than one in every instance. Therefore, we can say that the proposed CDE reduces the computational time that has been spent by SDE apparently. Especially, in case of the large maximum generation G_M=10^4, the speedup achieved by CDE increases as the number of threads increases in all the benchmark problems. Even though the multi-core processor guarantees the parallel processing of

N_T=8 threads at the maximum, we can observe the increase of the speedup until N_T=16. On the other hand, in case of the small maximum generation G_M=10^2, the speedup achieved by CDE increases steady until N_T=8 threads but decreases with N_T=16 in all the benchmark problems. However, we can say that CDE utilizes all resources provided by the multi-core processor.

From the results of the numerical experiment, the speedup achieved by CDE actually depends on the type of benchmark problems. Exactly, for expensive benchmark problems that require the calculation of costly functions such as trigonometric function and exponential function for evaluating their objective function values, namely f_3, f_4 and f_5, the speedup achieved by CDE is kept in high. Consequently, we can expect that the proposed CDE is useful specifically for solving the real-world applications that spend the majority of the computational time for evaluating their objective function values.

VII. Conclusion

In order to utilize the recent multi-core processor efficiently, a concurrent implementation of DE named Concurrent DE (CDE) was proposed. The proposed CDE was based on a modern concurrent programming model called "MapReduce". CDE divided the population into multiple chunks. Then CDE assigned the task for updating the individuals included in each chunk to a thread statically. The multi-core processor executed multiple threads in parallel by using multiple CPUs. Therefore, we could expect that the computational time was reduced by using the proposed CDE on the multi-core processor. From the numerical experiment conducted on a variety of benchmark problems, it was confirmed that the speedup achieved by CDE generally increased as the number of the threads increased.

In our future work, we need to study the effect of the number of threads on the performance of CDE. Besides, we would like to utilize CDE to solve expensive real-world applications.

Acknowledgment

The research reported in this paper was supported in part by the Grant-in-Aid for Scientific Research (C) (No. 21560432) from Japan Society of the Promotion of Science (JSPS).

References

[1] R. Storn and K. Price, "Differential evolution – a simple and efficient heuristic for global optimization over continuous space," *Journal of Global Optimization*, vol. 11, no. 4, pp. 341–359, 1997.
[2] J. Vesterstrom and R. Thomson, "A comparative study of differential evolution, particle swarm optimization, and evolutionary algorithms on numerical benchmark problems" in Proc. *IEEE Congress on Evolutionary Computation*, 2004, pp. 1980–1987.
[3] K. V. Price, R. M. Storn, and J. A. Lampinen, *Differential Evolution – A Practical Approach to Global Optimization*. Springer, 2005.
[4] R. Storn, "System design by constraint adaptation and differential evolution," *IEEE Trans. Evolutionary Computation*, vol. 3, no. 1, pp. 22–34, 1999.
[5] K. Tagawa, "Multi-objective optimum design of balanced SAW filters using generalized differential evolution," WSEAS Trans. System, Issue 8, vol. 8, pp. 923–932, 2009.
[6] R. Oonsivilai and A. Oonsivilai, "Differential evolution application in temperature profile of fermenting process," WSEAS Trans. System, Issue 6, vol. 9, pp. 618–628, 2010.
[7] G. Syswerda, "A study of reproduction in generational and steady-state genetic algorithms," *Foundations of Genetic Algorithms 2*, Morgan Kaufmann Publ., 1991, pp. 94–101.
[8] K. Tagawa, "A statistical study of the differential evolution based on continuous generation model," in Proc. *IEEE Congress on Evolutionary Computation*, 2009, pp. 2614–2621.
[9] S. Rahnamayan, H. R. Tizhoosh, and M. M. A. Salama, "Opposition-based differential evolution for optimization of noisy problems," in Proc. *IEEE Congress on Evolutionary Computation*, 2006, pp. 6756–6763.
[10] S. Kukkonen and J. Lampinen, "GDE3: The third evolution step of generalized differential evolution," in Proc. *IEEE Congress on Evolutionary Computation*, 2005, pp. 443–450.
[11] U. K. Chakraborty, *Advances in Differential Evolution*. Springer, 2008.
[12] A. K. Qin and P. N. Suganthan, "Self-adaptive differential evolution algorithm for numerical optimization," in Proc. *IEEE Congress on Evolutionary Computation*, 2005, pp. 1785–1791.
[13] J. Zhang and A. C. Sanderson, "JADE: Adaptive differential evolution with optional external archive," IEEE Trans. Evolutionary Computation, vol. 13, no. 5, pp. 945–958, 2009.
[14] V. Feoktistov, *Differential Evolution in Search Solution*. Chapter 6, Springer, 2006.
[15] K. Tagawa and H. Takada, "Comparative study of extended sequential differential evolutions," in Proc. *the 9th WSEAS International Conference on Applications of Computer Engineering*, 2010, pp. 52–57.
[16] C. Breshears, *The Art of Concurrency – A Thread Monkey's Guide to Writing Parallel Applications*, O'Reilly, 2009.
[17] E. Alba and M. Tomassini, "Parallelism and evolutionary algorithms," *IEEE Trans. Evolutionary Computation*, vol. 6, no. 5, pp. 443–462, 2002.
[18] L. F. Bic and M. B. Dillencourt, "Advantages of self-migration for distributed computing," *International Journal of Computers*, Issue 3, vol. 2, pp. 320–329, 2008.
[19] D. Zaharie and D. Petcu, "Parallel implementation of multi-population differential evolution," *Concurrent Information Processing and Computing*, ISO Press, 2005, pp. 223–232.
[20] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," in Proc. *6th Symposium on Operating Systems Design and Implementation*, 2004, pp. 137–149.
[21] L. J. Eshelman and J. D. Schaffer, "Real-code genetic algorithms and interval-schemata," *Foundations of Genetic Algorithms 2*, Morgan Kaufmann Publ., 1993, pp. 187–202.
[22] K. Tagawa, "A comparative study of distance dependent survival selection for sequential DE," in Proc. *IEEE International Conference on System, Man, and Cybernetics*, 2010, to be published.
[23] K-Y. Wong, Y-M. Choi, and S-W. Lam, "The design, implementation and application of the software framework for distributed computing," *International Journal of Computers*, Issue 3, vol. 1, pp. 109–116, 2007.
[24] J. Wan, W. Yu, and X. Xu, "Design and implementation of distributed document clustering based on MapReduce," in Proc. *the 2nd Symposium on International Computer Science and Computational Technology*, 2009, pp. 278–280.

Kiyoharu Tagawa received his M.E. and Ph.D. degrees from Kobe University Japan, in 1993 and 1997, respectively. From 2005 to 2007, he served as an Associate Professor of the Faculty of Engineering, Kobe University. He is currently a Professor of the School of Science and Engineering, Kinki University Japan. His research interests include evolutionary computation, concurrent programming, and real-world applications.

Takashi Ishimizu received his M.E. and Ph.D. degrees from Nara Institute of Science and Technology (NAIST), in 1997 and 2000, respectively. He is now an Assistant Processor of the School of Science and Engineering, Kinki University Japan. His main researches are parallel algorithms and parallel complexity theory.