Concurrent Implementation of Differential Evolution

KIYOHARU TAGAWA and TAKASHI ISHIMIZU
School of Science and Engineering
Kinki University
3-4-1 Kowakae, Higashi-Osaka, 577-8502
JAPAN
tagawa@info.kindai.ac.jp

Abstract: Recently, general-purpose multi-core processors have been introduced widely into personal computers. In order to utilize the additional cores to execute costly application programs such as Evolutionary Algorithms (EAs), concurrent implementations of them are demanded. Even though EAs including various Differential Evolutions (DEs) are naturally prone to parallelism, Sequential DE (SDE) is especially suited for concurrent programming. Therefore, a concurrent implementation of SDE, which is based on the map and reduce framework, is proposed. Through the numerical experiment, the speedup of SDE due to the use of multiple cores is demonstrated. Furthermore, it is shown that the concurrent programming of SDE is efficient, simple, portable and scalable.

Key–Words: Evolutionary Algorithm, Differential Evolution, Concurrent Programming

1 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 functions[2, 3]. Furthermore, because of its simple but powerful searching capability, DE has got numerous real-world applications[3, 4].

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[5]. 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”[6]. 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 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.

Recently, a new DE based on the above continuous generation model is proposed[6, 7, 8]. The new DE is sometimes called “Sequential DE” (SDE)[8]. According to the continuous generation model, SDE uses only one population. 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. On the other hand, if the target vector included in the population is worse than the trial vector, the target vector is replaced by the trial vector immediately. Then the excellent newborn individual, namely, the trial vector, can be used soon to generate offspring. As a result, it can be expected that SDE finds good solutions faster than the classic DE[6].

General-purpose multi-core processors have been introduced into personal computers lately. In order to utilize the additional cores to execute costly application programs such as EAs, concurrent implementations of them are demanded[9]. Furthermore, since EAs maintain a lot of individuals, or tentative solutions, manipulated competitively, EAs have a parallel and distributed nature intrinsically. Therefore, various parallelization techniques of EAs have been proposed[10]. Parallel DE is also implemented by using Parallel Virtual Machine (PVM)[11].

In this paper, a concurrent implementation of SDE is proposed. The survival selection of DE that compares the trial vector only to the target vector enables us to manipulate all individuals included in the population independently. Furthermore, SDE need not synchronize the manipulations of all individuals in each generation for replacing the old population by the new one. Therefore, SDE is especially suited for concurrent programming. The proposed concurrent implementation of SDE is based on the map and reduce framework[9]. Incidentally the map and reduce framework has been attracting attention as a modern
algorithmic framework that can be executed concurrently. Through the numerical experiment on a personal computer, the speedup of SDE due to the use of multiple cores is demonstrated. Furthermore, it is shown that the proposed concurrent programming of SDE is efficient, simple, portable and scalable.

2 Differential Evolution (DE)

2.1 Representation

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

$$\begin{align*}
\text{minimize} & \quad f(x) = f(x_0, \cdots, x_{D-1}) \\
\text{subject to} & \quad \underline{x}_i \leq x_i \leq \overline{x}_i, \quad j = 0, \cdots, D-1.
\end{align*} \quad (1)$$

Differential Evolution (DE)[1] is used to solve the optimization problem shown in (1). As well as conventional real-coded GAs[12], each tentative solution is represented by a real-parameter vector and called an “individual”. Furthermore, DE holds $N_P$ individuals within the population for each generation. Therefore, the $i$-th individual $x_i$ $(i = 0, \cdots, N_P - 1)$ arranged in the population is represented as,

$$x_i = (x_{0,i}, \cdots, x_{j,i}, \cdots, x_{D-1,i}) \quad (2)$$

where, $\underline{x}_j \leq x_{j,i} \leq \overline{x}_j$, $j = 0, \cdots, D-1$.

2.2 Strategy of DE

Differential mutation is a unique genetic operator of DE. Furthermore, a set of three genetic operators, namely, reproduction selection, differential mutation and crossover, is called the strategy of DE[1]. SDE also uses the strategy of DE[6, 7, 8]. Even though various strategies have been proposed for DE[3], a basic strategy named “DE/rand/1/exp” is described and used in this paper. That is because the basic strategy has relatively good compatibility with SDE[13].

For each individual $x_i$ $(i = 0, \cdots, N_P - 1)$ of the population, which is also called the target vector, three different individuals, say $x_{r1}, x_{r2}$ and $x_{r3}$ $(i \neq r1, r2 \neq r3, r3)$, are selected randomly from the population. Then a new individual $u_i$ called the trial vector is generated from them as shown in (3).

$$\begin{align*}
\left[ \begin{array}{c}
\underline{j}_r = \text{rand}[0, \ D-1]; \\
\text{do} \\
\quad u_{j;i} = x_{j;r1} + S_F (x_{j;r2} - x_{j;r3}); \\
\quad j = (j + 1) \% D;
\text{while} (\text{rand}_j[0, 1] \leq C_R \vee j \neq j_r) \\
\quad u_{j;i} = x_{j;i}; \\
\quad j = (j + 1) \% D;
\end{array} \right] \quad (3)
\end{align*}$$

where, $u_i = (u_{0,i}, \cdots, u_{j,i}, \cdots, u_{D-1,i})$.

If an element $u_{j,i}$ comes out of the range $[\underline{x}_j, \overline{x}_j]$ by using the strategy, it is returned to the range as,

$$u_{j;i} = (\overline{x}_j - u_{j,i}) \text{rand}_j[0, 1] + \underline{x}_j. \quad (4)$$

In the strategy of DE shown in (3), the subscript $j_r \in \{1, D-1\}$ is selected randomly. Therefore, the trial vector $u_i$ will be different from the target vector $x_i$ at least one element. rand$_j[0, 1]$ is the random number generator that returns a uniformly distributed random number from within the range between $0$ and $1$. Besides the population size $N_P$, the scale factor $S_F \in (0, 1+]$ and the crossover rate $C_R \in [0, 1]$ are control parameters specified by the user in advance.

2.3 Procedure of DE

The procedure of the classic DE[1] can be described by using the following pseudocode. Since the classic DE is based on the discrete generation model, two populations, namely, the old one $x_i \in P_{old}$ and the new one $\tilde{x}_i \in P_{new}$, are used. As the stopping condition for DE, the generation $g$ is limited to the maximum $G_M$.

[ pseudocode for DE ]

Randomly generate $x_i \in P_{old}$
for ($i = 0; i < N_P; i++$) {
    Evaluate $f(x_i)$;
}
for ($g = 0; g < G_M; g++$) {
    for ($i = 0; i < N_P; i++$) {
        Generate $u_i$ from (3) and (4);
        Evaluate $f(u_i)$;
        if ($f(u_i) \leq f(x_i)$) $\tilde{x}_i = u_i$;
        else $\tilde{x}_i = x_i$;
    }
    \// Update population
    for ($i = 0; i < N_P; i++$) {
        $x_i = \tilde{x}_i$;
    }
}
Output the best $x_i \in P_{old}$;
3 Sequential DE (SDE)

The procedure of SDE[8] can be described by using the following pseudocode. Since SDE is based on the continuous generation model, only one population $x_i \in P$ is used. If a newborn trial vector $u_i$ is excellent, it is added to the population immediately. Therefore, in case of SDE, the excellent trial vector $u_i \in P$ can be used soon to generate succeeding trial vectors.

\[
\begin{align*}
\text{Randomly generate } & x_i \in P; \\
\text{for } (i = 0; i < N_P; i++) & \\
\text{Evaluate } & f(x_i); \\
\text{for } (g = 0; g < G_M; g++) & \\
\text{for } (i = 0; i < N_P; i++) & \\
\text{Generate } & u_i \text{ from (3) and (4);} \\
\text{Evaluate } & f(u_i); \\
\text{if } (f(u_i) \leq f(x_i)) & x_i = u_i; \\
\end{align*}
\]

Output the best $x_i \in P$.

Figure 1 illustrates the procedure of DE, while Figure 2 illustrates the procedure of SDE. Comparing the procedure of SDE in Fig. 2 with that of the classic DE in Fig. 1, we can confirm that SDE obviously saves both the memory space for one population and the processing time spent for replacing the old population $x_i \in P_{old}$ by the new one $x_i \in P_{new}$.

4 Concurrent DE (CDE)

4.1 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. The difference between these definitions is the phrase in progress. The concurrent program performs multiple tasks in parallel if it 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 modeled by the Parallel Random Access Machine (PRAM)[9]. Figure 3 shows the configuration of PRAM. PRAM has multiple cores, or Central Processing Units (CPUs), attached to an unlimited memory that is shared among all the CPUs. PRAM uses a shared bus connecting memory and CPUs, where details of the connection mechanism between them are usually ignored. The threads that are executing on CPUs are assumed to be advancing in lockstep fashion. Also, the threads are assumed to have the same access time to the memory locations regardless of the number of CPUs.

CPUs included 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. Therefore, PRAM need to be specified how threads executing on individual CPUs will be able to access memory for both reading and writing. In this paper, we employ the Concurrent Read and Exclusive Write (CREW)[9] memory access for PRAM. In accordance with CREW, multiple threads executing on CPUs may read from the same memory locations.
location at the same time and only one thread may write to a given memory location at any time.

4.2 Procedure of CDE

From the results of the numerical experiment and the statistical test, a significant difference between SDE and DE has not been confirmed[6, 13]. On the other hand, comparing to DE, SDE is more suitable for the concurrent programming. That is because SDE doesn’t have the procedure for replacing the old population by the new one at a time. Therefore, the concurrent implementation of SDE is proposed in this paper. From now on, the proposed concurrent implementation of SDE is called Concurrent DE (CDE).

In the proposed CDE, the population \( x_i \in P \) is divided into \( N_T \) regions called chunks. Therefore, each chunk is regarded as a subpopulation including \( N_P/N_T \) individuals. Then 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. The procedure of CDE can be described by using the following pseudocode.

[Pseudocode of CDE]

```plaintext
Randomly generate \( x_i \in P \);
// Map
for all \( n \) in parallel do {
    for \( (n = 0; n < N_T; n++) \) {
        Thread\( (n) \);
    }
}
Barrier();
// Reduce
Output the best \( x_i \in P \);
```

[Figure 4: Procedure of Concurrent DE (CDE)]

The above CDE is based on the map and reduce framework[9]. First of all, initial population is generated randomly. Then, in the map phase, the procedures of Thread\( (n) \) \( (n = 0, \cdots, N_T - 1) \) are executed in parallel. Each Thread\( (n) \) is assigned to one thread and contracts the task for updating the individuals in the chunk. Barrier\( () \) denotes the object that waits until all the procedures of Thread\( (n) \) are completed. Finally, in the reduce phase, the best individual is selected from the final population. Figure 4 shows the procedure of CDE in case of \( N_T = 3 \).

4.3 Inspection of CDE

The goodness of a concurrent program is usually examined with respect to four categories, namely, efficiency, simplicity, portability and scalability[9]. Therefore, we will examine the proposed implementation of CDE with respect to the above categories.

First of all, EAs spend the majority of the computational time for evaluating the objective function value in real-world applications. The proposed CDE evenly distributes the objective function evaluations to \( N_T \) threads Thread\( (n) \). Each individual in the population \( x_i \in P \) can be overwritten by a unique Thread\( (n) \) that satisfies the condition: \( n = i \% N_T \). On the other hand, every individual can be red from every Thread\( (n) \) \( (n = 0, \cdots N_T) \) at the same time. Since the mutual exclusion is guaranteed naturally in the proposed CDE, any thread need not to synchronize with the other threads. Therefore, CDE can be regarded as an efficient program.

Comparing the procedure of CDE with that of SDE, there is not so much difference between these procedures. Therefore, the proposed CDE is simple as well as SDE. Furthermore, CDE can be translated into any program languages that support multiple threads. Therefore, the proposed CDE is portable. Finally, the proposed CDE is also scalable. That is because the population \( x_i \in P \) can be divided into chunks of an arbitrary number within the range: \( 1 \leq N_T \leq N_R \).
5 Numerical Experiment

5.1 Benchmark Problems

In order to evaluate the performance of CDE, the following four benchmark problems are employed. \( f_1 \) and \( f_2 \) are unimodal functions, while \( f_3 \) and \( f_4 \) are multimodal functions. All the benchmark problems have \( D = 30 \) dimensional real-parameters. Besides, the objective function values of their optimal solutions \( x^* \) are known as \( f_p(x^*) = 0 \ (p = 1, \ldots, 4) \).

- Sphere function
  \[
  f_1(x) = \sum_{j=1}^{D} x_j^2 \\
  -100 \leq x_j \leq 100, \ j = 1, \ldots, D
  \]

- Schwefel’s Ridge function
  \[
  f_2(x) = \sum_{j=1}^{D} \left( \sum_{k=1}^{j} x_k \right)^2 \\
  -100 \leq x_j \leq 100, \ j = 1, \ldots, D
  \]

- Rosenbrock function
  \[
  f_3(x) = \sum_{j=1}^{D-1} (100 (x_{j+1} - x_j^2)^2 + (x_j - 1)^2) \\
  -30 \leq x_j \leq 30, \ j = 1, \ldots, D
  \]

- Rastrigin function
  \[
  f_4(x) = \sum_{j=1}^{D} (x_j^2 - 10 \cos(2\pi x_j) + 10) \\
  -5.12 \leq x_j \leq 5.12, \ j = 1, \ldots, D
  \]

5.2 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 serial execution time to parallel execution time.

In order to evaluate the performance of a concurrent program executed on a multi-core processor, the above conventional definition of the speedup has to be modified. In the multiple-core processor, the executions of multiple threads, which are generated by the concurrent program, are assigned automatically to respective cores by operating system. Although the programmer can specify the number of threads, he can not specify the number of cores executing his program. Furthermore, in order to evaluate the performance of the proposed CDE, a single execution of CDE is not statistically significant. That is because CDE is a stochastic algorithm as well as the other EAs[10]. Therefore, a new speedup is defined as,

\[
S_m(N_T) = \frac{T_m(1)}{T_m(N_T)}
\]

where, \( T_m(1) \) 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.

5.3 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™ i7 @3.33[GHz]; OS: Microsoft Windows XP). The multi-core processor has four cores that can respectively manipulate two threads at a time.

In order to evaluate the speedup defined in (5), SDE and CDE are applied to the four optimization problems \( m = 10 \) times respectively. For making a comparative study, the classic DE is also applied to the respective optimization problems in the same way. During the experiments, the following control parameters of every DE are fixed: the population size \( N_P = 120 \), the scale factor \( S_F = 0.5 \) and the crossover rate \( C_R = 0.9 \). As the stopping condition, the maximum generation is specified as \( G_M = 10^6 \).

Figure 5 shows the speedup curves evaluated about the four benchmark problems. Although the
speedup achieved by CDE does not exceed the number of threads, the speedup generally increases as the number of threads increases for all the benchmark problems except Rastrigin function $f_1$ (□).

Table 1 shows the computational times spent by the classic DE, SDE and CDE ($N_T = 8$) for the four benchmark problems. Similarly, Table 2 shows the objective function values of the best solutions obtained by the respective DEs. The results shown in Table 1 and Table 2 are averaged over 10 runs.

From Table 1, we can confirm that CDE actually reduces the computational time for every benchmark problem by using multiple cores. From Table 2, CDE is superior to the classic DE. However, there is no difference between CDE and SDE in the quality of solutions. Therefore, we can also confirm that the algorithm of CDE is equivalent to that of SDE.

6 Conclusion

In order to utilize the recent general-purpose multi-core processor, a concurrent implementation of DE named CDE was proposed. CDE divided the population into multiple chunks. Then CDE assigned the task of each chunk to a thread statically. From the numerical experiment, it was confirmed that the speedup increased as the number of threads increased.

In our future work, we would like to apply the proposed CDE to costly real-world applications.

Acknowledgment

The research was supported by the Grant-in-Aid for Scientific Research (C) (No. 21560432) from JSPS.

References:


