Parallelizing Modified Cuckoo Search on MapReduce Architecture

Chia-Yu Lin, Yuan-Ming Pai, Kun-Hung Tsai, Charles H.-P. Wen, and Li-Chun Wang

***Abstract*—Meta-heuristics typically takes long time to search optimality from huge amounts of data samples for applications like communication, medicine, and civil engineering. Therefore, parallelizing meta-heuristics to massively reduce runtime is one hot topic in related research. In this paper, we propose a MapReduce modified cuckoo search (MRMCS), an efficient modified cuckoo search (MCS) implementation on a MapReduce architecture ― Hadoop. MapReduce particle swarm optimization (MRPSO) from a previous work is also implemented for comparison. Four evaluation functions and two engineering design problems are used to conduct experiments. As a result, MRMCS shows better convergence in obtaining optimality than MRPSO with two to four times speed-up.**

***Index Terms*—Cuckoo search, MapReduce, meta-heuristics, particle swarm optimization.**

1. Introduction

Meta-heuristics such as particle swarm optimization (PSO) and cuckoo search (CS) are widely used in engineering optimization. PSO was inspired by foraging social behavior of birds and fishes[1]. At the beginning, the species have no idea about the food location and thus search according to their experience and intuition. Once an individual finds the food, it informs other individuals of such location. Accordingly, others adjust their flight. Bird/fish foraging behavior is a concept of socially mutual influence, which guides all individuals to move toward the optimum. PSO is prevailing because it is simple, requires little tuning, and is found effective for problems of wide-range solutions.

Moreover, cuckoo search (CS), an optimization algorithm was proposed in 2009[2]. The cuckoo eggs mimic the eggs of other host birds to stay in their nests. This phenomenon leads to the evolution of egg appearance towards optimal disguise. In order to improve the performance of cuckoo search, a modified cuckoo search (MCS) was later proposed in 2011[3] and successfully demonstrated good performance. Based on [3], we parallelize MCS to propose a MapReduce modified cuckoo search (MRMCS) in this work. As a result, our MRMCS outperforms previously proposed MapReduce particle swarm optimization (MRPSO)[4] on all evaluation functions and two engineering design problems in terms of both convergence of optimality and runtime.

MapReduce[5] is a widely-used parallel programming model in cloud platforms and consists of mapping and reducing functions inspired by dividing and conquering. Mapping and reducing functions execute the computation in parallel, combine the intermediate result, and output the final result. Independent data are suitable for the MapReduce computing. For example, in the *k*-means algorithm, each data node computes the distance from itself to all central nodes and thus the work[6] proposed its parallelized version on MapReduce in 2009. Similarly, particle swarm optimization (PSO) addresses that each data node computes its own best value by itself and thus were also successfully parallelized on a MapReduce framework[4].

Since PSO can be successfully parallelized into MRPSO[4], we are motivated to parallelize MCS on a MapReduce architecture and compare the performance of MRMCS with that of MRPSO. However, two critical issues are worth pointing out when parallelizing MCS on a MapReduce architecture: 1) job partitioning (i.e. which jobs go to the mappers and which jobs go to the reducers) needs to be decided in MRMCS; 2) the support of information exchange is critical during evolution in MCS. However, an original MapReduce architecture like Hadoop cannot support this and thus need proper modification. Therefore, this work is motivated to deal with these two problems to enable good parallelism on MRMCS.

The rest of the paper is organized as follows. Section 2 introduces the fundamentals of MCS, and Section 3 describes the MapReduce architecture in detail. MRMCS is proposed and elaborated in Section 4. Section 5 shows several optimization applications with MRMCS and

Manuscript received October 31, 2012; revised December 25, 2012.

C.-Y. Lin is with the Institute of Communications Engineering, National Chiao Tung University, Hsinchu (Corresponding author e-mail: sallylin0121@gmail.com)

Y.-M. Pai, K.-H. Tsai, C. H.-P. Wen, and L.-C. Wang are with the Department of Electrical and Computer Engineering, National Chiao Tung University, Hsinchu (e-mail: paiming0728@hotmail.com; moonape1226@gmail.com; opwen@g2.nctu.edu; lichun@g2.nctu. edu.tw).

Digital Object Identifier: 10.3969/j.issn.1674-862X.2013.02.002

compares their performance and runtime with MRPSO. Finally, Section 6 concludes the paper.

2. Modified Cuckoo Search

CS was proposed for optimization problems by Yang *et al.* in 2009[2]. Later, in order to improve the performance of the baseline CS, Walton *et al.* in 2011 added more perturbations to the generation of population and thus proposed MCS in [3]. In this work, we further parallelize MCS on a MapReduce architecture and propose MRMCS.

The original CS was inspired by the behavior of cuckoo laying eggs. Cuckoos tend to lay eggs in the nests of other host birds. If the host birds can differentiate cuckoo eggs from their own eggs, they may throw away cuckoo eggs or all eggs in the nest. This leads to the evolution of cuckoo eggs mimicking the eggs of local host birds. Yang *et al.*[2] conducted the three following rules from the behavior of cuckoo laying eggs for optimization:

- Each egg laid by one cuckoo is a set of solution coordinates and is dumped in a random nest at a time.
- A fraction of the nests containing the eggs (solutions) with best fitness will carry over to the next generation.
- The number of nests is fixed and there is a probability that a host can discover such alien egg. If this happens, the host can either discard the egg or the nest, resulting in building a new nest in a new location.

Besides the three rules stated above, the use of Lévy flight[7] for both the local and global search is another important component in CS. The Lévy flight, also frequently used in other search algorithms[7], is a random walk in which the step lengths have a probability distribution with heavy tails. The egg generated by Lévy flight compares its fitness value with that of the current egg. If the fitness value of the new egg is better than the current one, the new egg takes place of the position. The random size of Lévy flight is controlled by a constant step size α where α can be adjusted according to the problem size of target applications. The fraction of nests to be abandoned is the only one parameter which is needed to be adjusted during the CS evolution.

In order to speed up the convergence of evolution, Walton *et al.*[3] proposed MCS. There are two modifications over CS. The first change is that the step size α_1 is no longer a constant and can decrease as the number of generation increases. Adjusting α_1 dynamically leads to faster convergence on optimality. At each generation, a new step size of Lévy flight is

$$\alpha_1 = A/\sqrt{G}$$

where A is initialized as 1 and G is the generation number. This setting is used for deciding the fraction of nests to be abandoned.

The second change is the information exchange between eggs. In MCS, eggs with the best fitness values are put in the top-egg group. Every top egg will be paired with a randomly-picked egg. During the selection process, if the same egg is picked, a new egg is generated with the step size

$$\alpha_2 = A/G^2 .$$

Otherwise, a new egg is generated from two top eggs using the golden ratio

$$\phi = \left(1+\sqrt{5}\right)/2 .$$

The fraction of nests to be abandoned and the fraction of nests to generate next top eggs are two adjustable parameters in MCS. Algorithm 1 shows the details of MCS as follows.

Algorithm 1. MCS Algorithm in [3]

1: A←MaxLévyStepSize
2: ϕ←GoldenRatio
3: Initialize a population of n host nests x_i ,(i=1,2,⋯,n)
4: **for** all x_i **do**
5: Calculate fitness $F_i = f(x_i)$
6: **end for**
7: Generation number G←1
8: **while** NumberObjectiveEvaluations<MaxNumberEvaluations **do**
9: G←G+1
10: Sorts nests by order of fitness
11: **for** all nests to be abandoned **do**
12: Calculate position x_i
13: Calculate Lévy flight step size $\alpha_1 \leftarrow A/\sqrt{G}$
14: Perform Lévy flight from x_i to generate new egg x_k
15: $x_i \leftarrow x_k$
16: $F_i \leftarrow f(x_i)$
17: **end for**
18: **for** all of the top nests **do**
19: Calculate position x_i
20: Pick another nest from the top nests at random x_j
21: **if** x_i=x_j **then**
22: Calculate Lévy flight step size $\alpha_2 \leftarrow A/G^2$
23: Perform Lévy flight from x_i to generate new egg x_k
24: $F_k = f(x_k)$
25: Choose a random nest l from all nests
26: **if** $F_k > F_l$ **then**
27: $x_l \leftarrow x_k$
28: $F_l \leftarrow F_k$
29: **end if**
30: **else**

31: $dx = \left| x_i - x_j \right| / \phi$
32: Move distance dx from the worst nest to the best nest to find x_i
33: $F_k = f(x_k)$
34: Choose a random nest l from all nests
35: **if** $F_k > F_l$ **then**
36: $x_l \leftarrow x_k$
37: $F_l \leftarrow F_k$
38: **end if**
39: **end if**
40: **end for**
41: **end while**

3. MapReduce Architecture

MapReduce[5] is a patented software framework introduced by Google to support distributed computing on large data volumes on clusters of computers. MapReduce can also be considered as a parallel programming model and it aims at processing large datasets. A MapReduce framework consists of mapping and reducing functions which are inspired by dividing and conquering. The map function, which is also known as the mapper, parallelizes the computation on large-scale clusters of machines. The reduce function, which is also called the reducer, collects the intermediate results from the mappers and then outputs the final result. In the MapReduce architecture, all data items are represented as the form of keys paired with associated values. For example, in a program that counts the frequency of occurrences for words, the key is the word itself and the value is its frequency of occurrences. Applications with independent input data or computation are suitable to be parallelized on the MapReduce framework. For example, for PSO, each data node can finish computing its own best value without acquiring information from other nodes. Therefore, PSO is a good candidate that can be parallelized on the MapReduce framework to save runtime greatly. Such idea was termed MRPSO and realized in [4].

3.1 Map Function (Mapper)

A MapReduce job usually splits the input data set into many independent chunks which are processed by the map function in a completely parallel manner. The map function takes a set of (key, value) pairs and generates a set of intermediate (key, value) pairs by applying a designated function to all these pairs, that is,

$$\text{Map: } (k_1, v_1) \rightarrow \text{list}(k_2, v_2).$$

3.2 Reduce Function (Reducer)

Before running the reduce function, the shuffle and sort functions are applied to the outputs from the map function. Then the new outputs become the input to the reduce function. The reduce function merges all pairs with the same key using a reduction function:

$$\text{Reduce: } (k_2, \text{list}(v_2)) \rightarrow \text{list}(k_3, v_3).$$

The input type and output type of a MapReduce job are illustrated in Fig. 1, respectively. The data which is a (key, value) pair is the input to the mapper. The mapper extracts meaningful information from each record independently. The output of the mapper is sorted and combined according to the key and passed to the reducer where the reducer performs aggregation, summarization, filtering, or transformation of data and writes the final result.

3.3 MapReduce Example

An example of the overall Map/Reduce framework is shown in Fig. 2. This is a program named "WordCount" used for counting the frequency of occurrences for different words. The input data is partitioned into several files and sent to different mappers to count occurrences of one target word. The input key is ignored but arbitrarily set to be the line number for the input value. The output key is the word under interest, and the output value is its counts. The shuffle and sort functions are performed to combine key values output from the mappers. Finally, the reducer merges the count value of each word and writes out the final result (i.e. the frequency of occurrences).

3.4 MapReduce Implementation

Google has published its MapReduce implementation in [5], but has not yet released the system to the public. Thus, the Apache Lucene project developed Hadoop, a Java-based platform, as an open-source MapReduce implementation. Hadoop[8] was derived from Google's MapReduce architecture and the Google file system (GFS). Data-intensive and distributed applications can work on Hadoop which can support up to thousands of computing nodes. In this work, we referred to [4] and implemented PSO and MCS into MRPSO and MRMCS, respectively, on the Hadoop platform.

(input) $<k_1, v_1>$ → **map** → **combine** → **reduce** → $<k_3, v_3>$ (output)

Fig. 1. Input and output types of a MapReduce job.

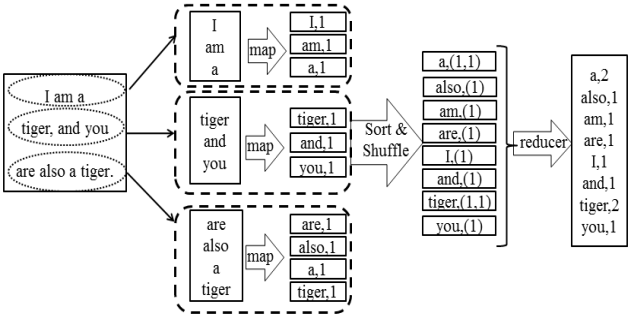

Fig. 2. Example of a MapReduce framework.

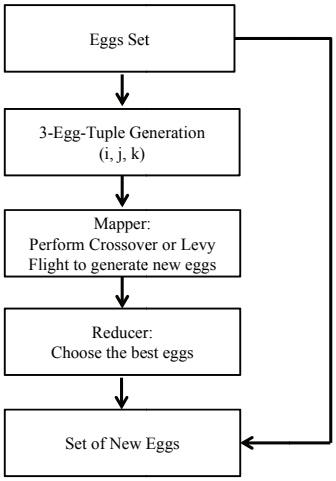

Fig. 3. Overall flowchart of MRMCS.

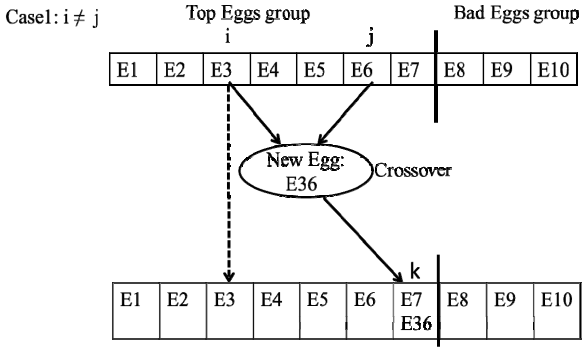

Fig. 4. One of the map function in MRMCS.

4. MRMCS

Parallelizing MCS on a MapReduce architecture is elaborated in this section. Two major problems remain to be solved. First, we have to determine a strategy for job partitioning. In other words, we need to decide jobs that mappers and reducers need to take care, respectively. Second, information is enabled to exchange in MRMCS, and computing nodes need to communicate with each other in the MapReduce architecture, which was not supported originally. Therefore, we propose 3-egg-tuple transformation to facilitate exchange information between eggs.

Fig. 3 shows the overall flow of MRMCS. The 3-egg-tuple transformation function outputs a new sample composed of original samples (i, j, k) for mappers, where i denotes the index of current egg, j is the index of a randomly-picked egg to be paired with egg i, and k is the index of the nest for putting the new egg after evolution. After the 3-egg-tuple transformation process, mappers perform gold-ratio crossover or Lévy flight to generate a new egg. Later, each reducer chooses the best eggs as the descendant sample among all candidates of its own. Details of three steps stated above are further discussed as follows.

4.1 3-Egg-Tuple Transformation

In MCS, eggs are separated according to top-egg groups and bad-egg groups. The egg picked from one top-egg group and the other one randomly picked from another top-egg group are first paired and then MCS performs the crossover operation over the pair to generate a new egg. If two eggs are picked from the same top-egg group coincidently, the Lévy flight is used instead to generate the new egg. Since the egg information is not preserved on different mappers in the MapReduce architecture, we combines information from three eggs into one and such function is called 3-egg-tuple transformation. The outputs of 3-egg-tuple transformation function are sets of (current egg index, randomly-picked egg index, target-nest index for putting the new egg) denoted as (i, j, k). Each 3-egg-tuple (i, j, k) is sent to a mapper for generating a new egg.

4.2 MRMCS Mappers

One key challenge of parallelizing MCS on a MapReduce platform is job partitioning. We have to decide which jobs go to mappers and which jobs go to reducers. The general rule is that mappers take charge of independent jobs and reducers are responsible for combining the results. Since operations of crossover and Lévy flight for new egg generation are independent among all samples, mappers are assigned to perform the new-egg generation. The 3-egg-tuples are the input to new-egg generation in mappers and each new-egg generation can be divided into three cases.

• Case 1: The top egg x_i and top egg x_j are not drawn from the same nest. The egg x_i is first duplicated and placed at the nest n_i for the next generation. Then the egg x_i and egg x_j are further used to perform the crossover operation and generate a new egg to be placed at the nest n_k. Fig. 4 shows an example for this case.

• Case 2: The top egg x_i and top egg x_j are drawn from the same nest. The egg x_i is first duplicated and placed at the nest n_i for the next generation. The Lévy flight operation is performed on the egg x_i instead and a new egg is generated to be placed at the nest n_k. Fig. 5 shows an example for Case 2.

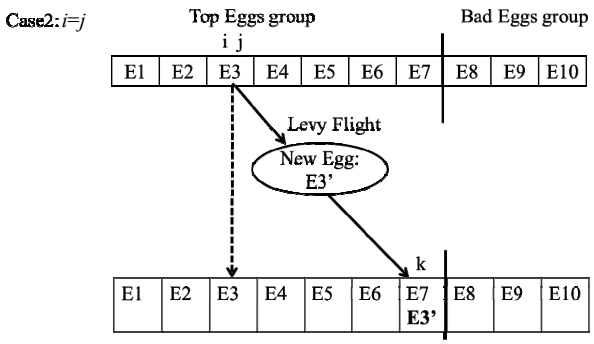

Fig. 5. Case two of the mapper function in MRMCS.

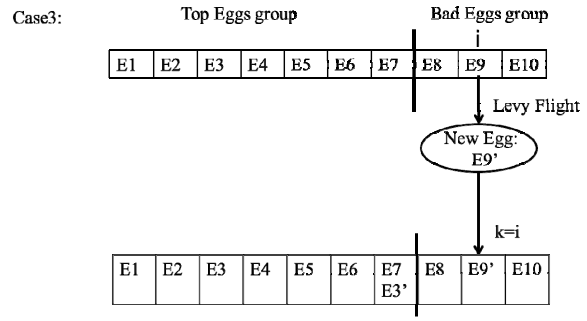

Fig. 6. Case three of the mapper function in MRMCS.

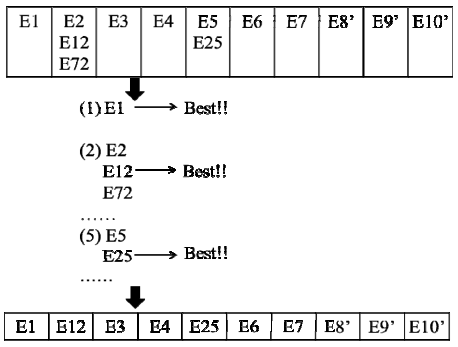

Fig. 7. Example on the reduce function.

• Case 3: The Lévy flight operation is performed on the bad egg x_i directly and a new egg is generated to be placed at n_k=n_i, as shown in Fig. 6.

4.3 MRMCS Reducers

Reducers are responsible for combing the intermediate results from mappers. In MRMCS, reducers determine the next-generation egg of the nests, respectively. Fig. 7 shows an example for the reducer operation. After mappers generate new eggs, every nest may contain more than one egg. Each reducer finds the best value from all eggs in one nest and uses the egg with the best value as the next generation. The results of reducers are used as the input to the next MRMCS generation.

Algorithms 2 and 3 summarize the details of mapper operations including three cases stated above and the reducer operations in MRMCS.

Algorithm 2. MRMCS on Map

1: A←MaxLévyStepSize
2: ϕ←GoldenRatio
3: $f(x_i)$ ← The fitness of x_i
4: **definition**: Mapper (key, value)
5: **input**: (Last iteration Fitness value, *S*),
S:{(x_1, x_j, x_k),…,(x_n, x_j, x_k)}, a set of (the current egg, a random egg, the nest for putting new egg).
6: **if** Bad nest **then**
7: Pick the nest n_i
8: Calculate Lévy flight step size $\alpha_1 \leftarrow A/\sqrt{G}$
9: Perform Lévy flight from x_i to generate new egg x_k
10: $x_i \leftarrow x_k$
11: $F_i \leftarrow f(x_i)$
12: **end if**
13: **if** Top nest **then**
14: Pick the nest n_i
15: Randomly pick another nest n_j from another top nest
16: **if** $i = j$ **then**
17: Calculate Lévy flight step size $\alpha_2 \leftarrow A/G^2$
18: Perform Lévy flight from x_i to generate new egg x_k
19: $F_k = f(x_k)$
20: **else**
21: $dx = |x_i - x_j| / \phi$
22: Move distance *dx* from the worst nest to the best nest to find x_k
23: $F_k = f(x_k)$
24: **end if**
25: **end if**

Algorithm 3. MRMCS on Reduce

1: **definition:** Reducer (key, valuelist):
2: **input**: (Last iteration fitness value, a population of *n* host nest F_i, i=1, 2,…, n)
3: **for** all F_i **do**
4: Find the best value x_{best} of F_i
5: Calculate fitness $F_i = f(x_{best})$
6: **end for**

5. Evaluations and Applications

In our experiments, we implemented both serial and parallel versions for MCS and PSO on Hadoop. The serial MCS generated the new egg of every nest and replaced the old egg with the better one serially. The process of MRMCS was similar to the serial MCS. However, instead of performing MCS sequentially, in order to exchange information on Hadoop, the 3-egg-tuple transformation proceeded before executing the mapping and reducing functions. The output of 3-egg-tuple transformation was the input to the MapReduce operation.

Hadoop carried out a sequence of MapReduce operations, each of which evaluated a single iteration of MCS. In each MapReduce operation, Hadoop called the mapping function (as in Algorithm 2) and the reducing function (as in Algorithm 3). Mappers in Hadoop generated the new egg of every nest through the crossover or Lévy flight operation in parallel and reducers chose the best egg from all candidates of every nest, respectively. The output of each MapReduce operation represented the best egg of each nest. MRPSO was also implemented according to [4]. Various evaluations of MRMCS and MRPSO in terms of performance and runtime were compared in the following sections.

Experiments were conducted on a computer with an AMD FX(tm)-8150 eight-core processor and 12 GB

memory. Eight virtual machines (VMs) were run on the physical machine. A 10 G disk and a 1 G memory were allocated to each VM. Hadoop version 0.21 in Java 1.7 was used as the MapReduce system for all experiments. The input dataset (containing 1000 data nodes) was generated by Latin hypercube sampling[9] with respect to different applications. Here four evaluation functions and two engineering optimization applications[10] with their experimental results are presented as follows, respectively.

5.1 Function Griewank

The Griewank function can be expressed as

$$f_A(x) = (1/4000)\sum_{i=1}^{d} x_i^2 - \prod_{i=1}^{d} \cos\left(x_i/\sqrt{i}\right) + 1$$

where in our experiment, dimension d is set as 30, x_i is a random variable, $x_i \in [-600, +600]$, and i is their index from 1 to d. Fig. 8 compares the performance of MRMCS and MRPSO for Griewank. As a result, MRMCS and MRPSO found the minimum values at the scale of 10^{-2} and 10^{-1} after 3000 times of iteration evolution, demonstrating that MRMCS shows a faster convergence than MRPSO does.

Fig. 9 compares the runtime of MRMCS and MRPSO on Griewank using 1, 2, 4, and 8 virtual machines, respectively. As you can see, MRMCS run faster than MRPSO. Such phenomenon can be attributed to two reasons: 1) MRPSO in [4] did not use fitness values as keys. As a result, in each iteration, searching the optimal value among all samples requires more time for additional comparison operations. 2) MRPSO requires an extra file for the dependent list as its input data. However, such a large file incurs more processing time to the total runtime. More specifically, in Fig. 9, we can also observe that the runtime of MRMCS decreases when VM increases. Although the runtime reduction is not linear, MRMCS still runs more efficiently than MRPSO does on Hadoop.

5.2 Function Rastrigrin

Define the second evaluation function—Rastrigrin as

$$f_B(x) = 10d + \sum_{i=1}^{d} [x_i^2 - 10\cos(2\pi x_i)]$$

where in our experiment, dimension d is set as 30, x_i is a random variable, $x_i \in [-5.12, +5.12]$, and i is their index from 1 to d. The performance comparison of MRMCS and MRPSO for Rastrigin is shown in Fig. 10. Surprisingly, the minimum value found by MRMCS is much smaller than that found by MRPSO after 3000 times of iteration evolution. Again, MRMCS demonstrates a better convergence than MRPSO does. Runtime comparison between MRPSO and MRMCS is presented in Fig. 11. Similarly, thanks to two reasons stated above, MRMCS uses much shorter runtime than MRPSO under various numbers of VM in use. The total runtime used by MRMCS also decreases when the number of VM in use increases.

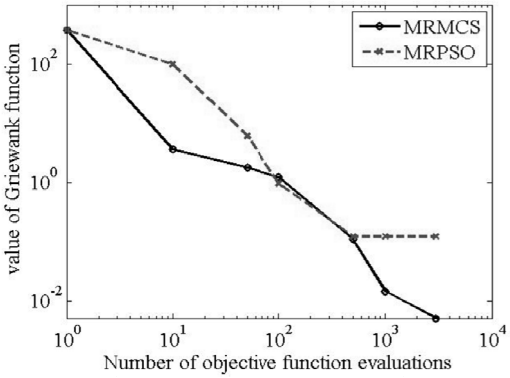

Fig. 8. Performance of MRMCS and MRPSO on Griewank.

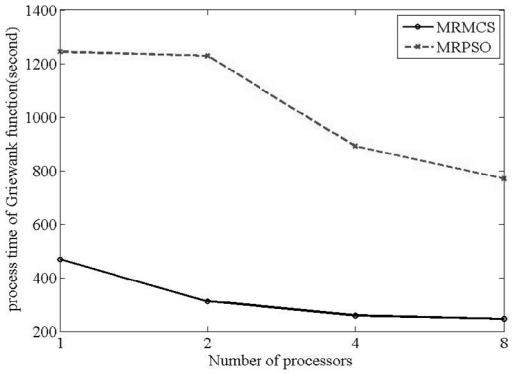

Fig. 9. Runtime of MRMCS and MRPSO on Griewank.

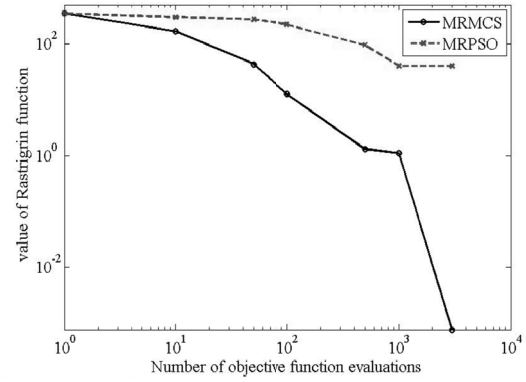

Fig. 10. Performance of MRMCS and MRPSO on Rastrigrin.

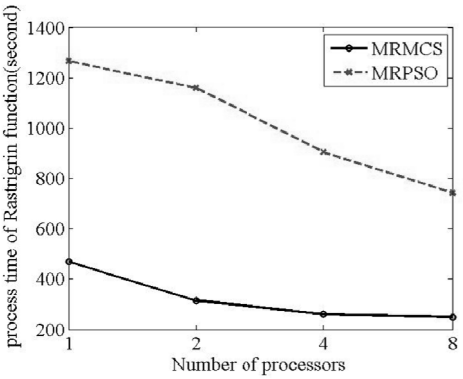

Fig. 11. Runtime of MRMCS and MRPSO on Rastrigrin.

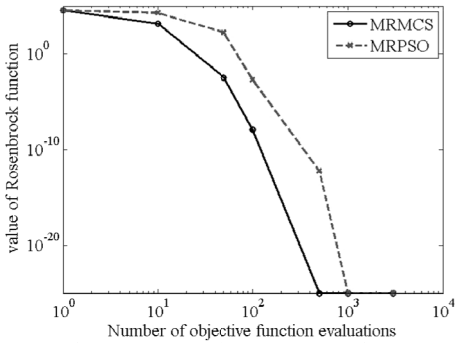

Fig. 12. Performance of MRMCS and MRPSO on Rosenbrock.

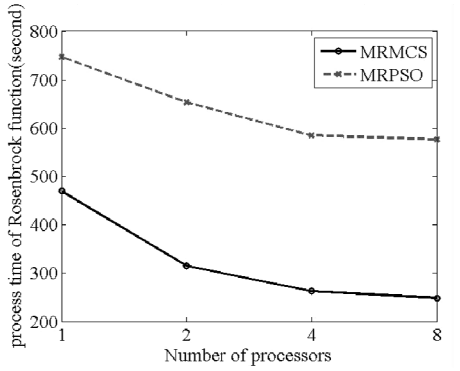

Fig. 13. Runtime of MRMCS and MRPSO on Rosenbrock.

5.3 Function Rosenbrock

The third evaluation function, Rosenbrock, is define as

$$f_C(x)=\sum_{i=1}^{d}\left[(1-x_i)^2+100(x_{i+1}-x_i^2)^2\right]$$

where in our experiment, dimension d is set as 30, x_i is a random variable, $x_i\in[-100,+100]$, and i is their index from 1 to d. Fig. 12 compares the performance of MRMCS and MRPSO for Rosenbrock. In this case, MRMCS and MRPSO can find the minimum value of the same quality after 3000 times iteration evolution. However, MRMCS converges during the 500th iteration where MRPSO converges during the 1000th iteration. Therefore, MRMCS is more efficient than MRPSO in finding the optimality for Rosenbrock.

Fig. 13 compares the runtime of MRMCS and MRPSO on Rosenbrock using 1, 2, 4, and 8 virtual machines, respectively. As you can see, MRMCS run faster than MRPSO. Again, MRPSO uses 2 to 3 times of runtime than MRMCS does, demonstrating that MCS is more suitable to be parallelized on the MapReduce architecture than PSO for the function Rosenbrock.

5.4 Function Sphere

The expression of the Sphere function is

$$f_D(\mathrm{x})=\sum_{i=1}^{d}x_i^2$$

where in our experiment, dimension d is set as 30, x_i is a random variable, $x_i\in[-5.12, +5.12]$, and i is their index from 1 to d. Fig. 14 compares the performance of MRMCS and MRPSO for Sphere. Unlike previous cases, in the middle of the search process, MRPSO once found a better value than MRMCS during around the 400th iteration. However, it cannot make any advancement for the rest of 2600 iterations. MRMCS, on the other hand, keeps polishing it solution. Before the end of our experiment, we have not yet concluded if the optimal value found by MRMCS is the true minimum value.

As to the runtime, Fig. 15 compares it of MRMCS with MRPSO to the Sphere function. Following the same trend as previous evaluations, MRMCS runs faster than MRPSO does, maintaining a 3 times speed-up.

5.5 Application of Spring Design

Tensional and/or compressional springs are used widely in engineering. There are three design variables in the spring design problem: the wire diameter w, the mean coil diameter d, and the length (or number of coils) L. The goal is to minimize the weight of the spring with the limitation of the maximum shear stress, minimum deflection, and geometrical limits. The details of spring design problem are described in [11] and [12].

This overall problem can be formulated as

$$\min f_E(x)=(L+2)w^2d$$

subject to

$$g_1(x)=1-\left(d^3L\right)\Big/\left(74785w^4\right)\le 0$$

$$g_2(x)=1-\left(140.45w\right)\Big/\left(d^2L\right)\le 0$$

$$g_3(x)=2(w+d)/3-1\le 0$$

$$g_4(x)=\left[d(4d-w)\right]\Big/\left[w^3(12566d-w)\right]+1\Big/\left(5108w^2\right)-1\le 0$$

where

$$0.05\le w\le 2.0,\ 0.25\le d\le 1.3,\ 2.0\le L\le 15.0.$$

Fig. 16 and Fig. 17 show the comparison in terms of performance and runtime of MRMCS and MRPSO, respectively, on the spring design application. It is clear that MRMCS and MRPSO can find the same optimal value but MRMCS runs much faster than MRPSO does, maintaining a 4-times speed-up.

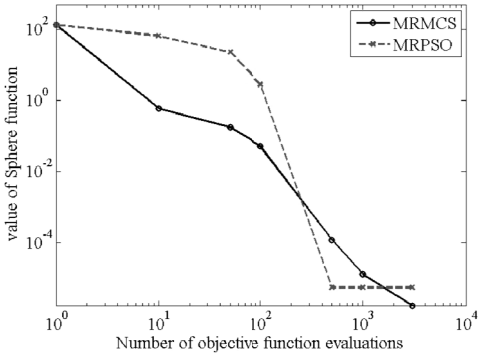

Fig. 14. Performance of MRMCS and MRPSO on Sphere.

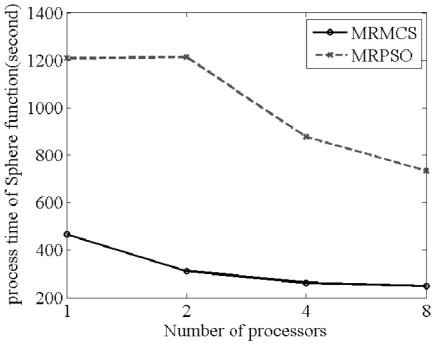

Fig. 15. Runtime of MRMCS and MRPSO on Sphere.

5.6 Application of Welded-Beam Design

The Welded-beam design problem comes from the standard test problem for constrained design optimization[12],[13]. There are four design variables in this problem: the width w and length L of the welded area, the depth d and thickness h of the main beam. The objective is to minimize the overall fabrication cost, under the appropriate constraints of the shear stress τ, bending stress σ, buckling load $P(x)$, and maximum end deflection δ.

This overall problem can be formulated as:

$$\min f_F = 1.1047w^2L + 0.04811dh(14.0 + L)$$

subject to

$$g_1(x) = \tau(x) - 13000 \le 0$$
$$g_2(x) = \sigma(x) - 30000 \le 0$$
$$g_3(x) = w - h \le 0$$
$$g_4(x) = 0.10471w^2 + 0.04811hd(14 + L) - 5 \le 0$$
$$g_5(x) = 0.0125 - w \le 0$$
$$g_6(x) = \delta(x) - 0.25 \le 0$$
$$g_7(x) = 6000 - P(x) \le 0$$

where

$$\sigma(x) = 504000 \Big/ \left(hd^2\right)$$
$$\delta(x) = 2.1951 \Big/ \left(h^3 d\right)$$
$$P(x) = 64746.022(1 - 0.0282346d)dh^3$$
$$\tau(x) = \sqrt{\alpha^2 + \alpha\beta L/D + \beta^2}$$
$$\alpha = 6000 \Big/ \left(\sqrt{2}wL\right)$$
$$\beta = QD/J$$
$$Q = 6000(14 + L/2)$$
$$D = 0.5\sqrt{L^2 + (w+d)^2}$$
$$J = \sqrt{2}wL\left[L^2/2 + (w+d)^2/2\right].$$

Fig. 18 and Fig. 19 show the performance and runtime comparisons of MRMCS and MRPSO on the welded-beam design application, respectively. Similarly as the spring design optimization, MRMCS and MRPSO achieve the solutions of comparable quality whereas MRMCS only takes a quarter of runtime than MRPSO does.

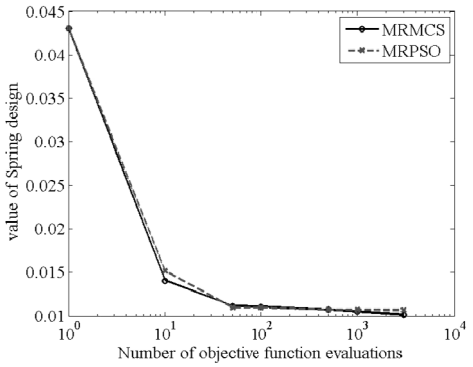

Fig. 16. Performance of MRMCS and MRPSO on spring design.

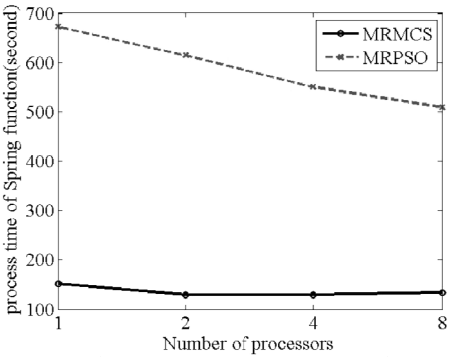

Fig. 17. Runtime of MRMCS and MRPSO on spring design.

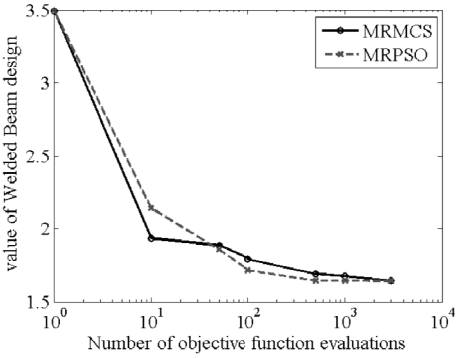

Fig. 18. Performance of MRMCS and MRPSO on welded-beam design.

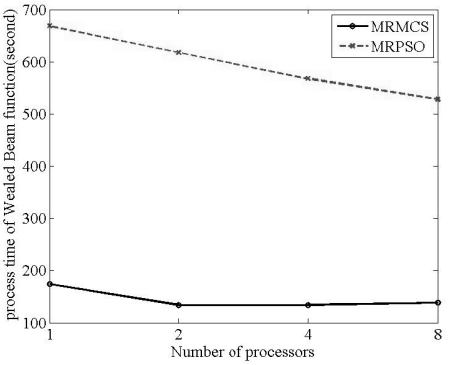

Fig. 19. Runtime of MRMCS and MRPSO on welded-beam design.

6. Conclusions

Meta-heuristics as a search strategy for optimization has been extensively studied and applied to solve many engineering problems. Most of them suffer from long runtime and thus parallelizing them to improve their efficiency is a thriving topic in research. Recently, PSO has been successfully implemented on the MapReduce platform. Therefore, in this paper, we parallelize MCS on a MapReduce platform and propose MRMCS. Problems of job partitioning and information exchange are solved by modification on the MapReuce architecture and 3-egg-tuple transformation. As a result, MRMCS outperforms MRPSO on four evaluation functions and two engineering design optimization applications. Experimental results show MRMCS has better convergence than MRPSO does. Moreover, MRMCS also brings about two to four times speed-ups for four evaluation functions and engineering design applications, demonstrating superior efficiency after parallelization on the MapReduce architecture (Hadoop).

References

[1] J. Kennedy and R. Eberhart, "Particle swarm optimization," in *Proc. of IEEE Int. Conf. on Neural Networks*, Perth, pp. 1942–1948, 1995.

[2] X. Yang and S. Deb, "Cuckoo search via Lévy flights," in *Proc. of IEEE World Congress on Nature & Biologically Inspired Computing*, Coimbatore, 2009, pp. 210–214.

[3] S. Walton, O. Hassan, K. Morgan, and M. Brown, "Modified cuckoo search: a new gradient free optimisation algorithm," *Chaos*, *Solitons & Fractals*, vol. 44, pp. 710–718, Sep. 2011.

[4] A. McNabb, C. Monson, and K. Seppi, "Parallel PSO using mapreduce," in *Proc. of IEEE Congress on Evolutionary Computation*, Singapore, 2007, pp. 7–14.

[5] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[6] W. Zhao, H. Ma, and Q. He, "Parallel *k*-means clustering based on mapreduce," *Lecture Notes in Computer Science* vol. 5931, 2009, pp. 674-679.

[7] I. Pavlyukevich, "Lévy flights, non-local search and simulated annealing," *Journal of Computational Physics*, vol. 226, no. 2, pp. 1830–1844, 2007.

[8] *Hadoop: The Definitive Guide*, O'Reilly Media, 2012.

[9] R. L. Iman, "Latin hypercube sampling," in *Encyclopedia of Statistical Science Update*, New York: Wiley, 1999, pp. 408–411.

[10] X. Yang and S. Deb, "Engineering optimisation by cuckoo search," *Int. Journal of Mathematical Modelling and Numerical Optimisation*, vol. 1, no. 4, pp. 330–343, 2010.

[11] J. S. Arora, *Introduction to Optimum Design*, Waltham: Academic Press, 2004.

[12] L. Cagnina, S. Esquivel, and C. Coello, "Solving engineering optimization problems with the simple constrained particle swarm optimizer," *Informatica*, vol. 32, no. 3, pp. 319–326, 2008.

[13] K. Ragsdell and D. Phillips, "Optimal design of a class of welded structures using geometric programming," *ASME Journal of Engineering for Industries*, vol. 98, no. 3, pp. 1021–1025, 1976

Chia-Yu Lin received the B.S. and M.S. degrees from National Chiao Tung University (NCTU), Hsinchu in 2010 and 2012, respectively, all in computer science. She is currently working toward the Ph.D. degree with the Institute of Communications Engineering, NCTU. Her current research interests include VM resource estimation and load balancing in cloud data centers.

Yuan-Ming Pai was born in Taiwan in 1991. He is currently pursuing the B.S. degree with the Department of Electrical and Computer Engineering, NCTU. His research interests include parallel computing and cloud computing.

Kun-Hung Tsai was born in Taiwan in 1990. He is currently pursuing his B.S. degree with the Department of Electrical and Computer Engineering, NCTU. His research interests include parallel computing and cloud computing.

Charles H.-P. Wen received his Ph.D. degree in VLSI verification and test from University of California, Santa Barbara in 2007. He is an associate professor at NCTU and a specialist in computer engineering. Over the past few years, his work has been focused on applying data mining and machine learning techniques on SoC design (especially on statistical soft error rates and circuit diagnosability in nanometer technologies) and cloud computing (especially on performance analysis and architecture design of large-scale data centers).

Li-Chun Wang received the B.S. degree from NCTU in 1986, the M.S. degree from National Taiwan University in 1988, and the Ms.Sci. and Ph.D. degrees from the Georgia Institute of Technology, Atlanta in 1995 and 1996, respectively, all in electrical engineering. From 1990 to 1992, he was with the Telecommunications Laboratories of the Ministry of Transportations and Communications in Taiwan (currently the Telecom Labs of Chunghwa Telecom Co.). Since August 2000, he has been an associate professor with the Department of Communication Engineering, NCTU. His current research interests include adaptive/cognitive wireless networks, radio network resource management, cross-layer optimization, and cooperative wireless communication networks.