2014 IEEE Congress on Evolutionary Computation (CEC)
July 6-11, 2014, Beijing, China

A New Grouping Genetic Algorithm for the MapReduce Placement Problem in Cloud Computing

Xiaoyong Xu and Maolin Tang

Abstract— **MapReduce is a computation model for processing large data sets in parallel on large clusters of machines, in a reliable, fault-tolerant manner. A MapReduce computation is broken down into a number of map tasks and reduce tasks, which are performed by so called *mappers* and *reducers*, respectively. The placement of the mappers and reducers on the machines directly affects the performance and cost of the MapReduce computation. From the computational point of view, the mappers/reducers placement problem is a generation of the classical bin packing problem, which is NP-complete. Thus, in this paper we propose a new grouping genetic algorithm for the mappers/reducers placement problem in cloud computing. Compared with the original one, our grouping genetic algorithm uses an innovative coding scheme and also eliminates the inversion operator which is an essential operator in the original grouping genetic algorithm. The new grouping genetic algorithm is evaluated by experiments and the experimental results show that it is much more efficient than four popular algorithms for the problem, including the original grouping genetic algorithm.**

I. Introduction

MapReduce is a highly-popular programming model for big data processing, which has been widely applied to many commercial and scientific applications, such as data mining, bioinformatics, machine learning and web indexing. MapReduce has the capability of processing petabytes of data in a single job through parallelizing the job on a large-scale cluster of computing nodes.

Unlike in a traditional cluster, MapReduce is operated in a different way in cloud computing. Once an end user submits its MapReduce jobs, a dedicated cluster of virtual machines (VMs), rented from an Infrastructure-as-a-Service (IaaS) provider like Amazon EC2, is generated instantly, and then the jobs start running on the cluster. Once the jobs are completed, the cluster is dismissed and the end user pays for the usage of the VMs.

A MapReduce job is executed by a set of *mappers* and *reducers*. Mappers and reducers are respectively used to execute the map tasks and reduce tasks in a MapReduce job. Both of them are called *workers*. In the rest part of the paper, without specific instruction, the workers indicate the mappers and reducers. When executing different jobs submitted by an end user, the workers probably have different demands for the resources like CPU, memory and so on. For example, some workers may have higher demands for CPU when running WordCount jobs while some workers may have higher demands for memory when executing Terasort jobs. These workers need to be placed on VMs, such that they can acquire the resources provided by VMs to execute their jobs. An inappropriate worker placement usually leads to a poor match for the resource demands of the workers. Placing too many workers on the same VM probably results in resource competition, thus leading to performance degradation. In contrast, placing too few workers on the same VM lowers the resource utilization although the resource demands of the workers are met. However, most current works on the resource provision [1] [2] for MapReduce have not considered the worker placement.

Then, a new problem named *Mapper/Reducer Placement Problem* (MRPP) is raised and needs to be addressed. The objective of MRPP is to place all the workers for a MapReduce computation on the VMs, such that the costs of the VMs are minimized while the resource demands of the workers are met. MRPP can be seen as a type of the bin packing problem (BPP). But, compared with the classic BPP, MRPP has three special features: (1) multiple types of VMs (bins) with different costs are available to load the workers (items); (2) there are multiple resource constraints on the worker placement; (3) multiple workers probably have the same resource demands, since they apply the same operations on the input with the similar size. Thus, MRPP can be taken as a multi-constraint BPP with variable bin size.

Clearly, MRPP is NP-hard, since it is a generalization of the BPP, which has been proven to be NP-complete [3]. Some heuristics [4] [5] [6] [7] have been proposed to solve multiple variants of the BPP. Recently, the genetic algorithms (GAs) including the ordering GA (OGA) [7] and the grouping GAs (GGAs) [8] [9] [10] have been introduced to solve the problems due to their ability of searching global optimums. However, MRPP-like problems have rarely been studied.

Therefore, in this paper we studies MRPP, a new problem of the MapReduce in cloud computing, and proposes a new GGA to solve it. Compared with the original GGA, our GGA uses an innovative coding scheme that can significantly reduce the search space and a knowledge-based crossover that can use domain-specific knowledge to enhance its exploitation capacity. In addition, the inversion operator which is an essential operator in the original GGA is eliminated from our GGA. Furthermore, we also provide a flexible way of constructing MRPP instances with known optimal solutions, which can be used to test the quality of solutions. Finally, we evaluate the performance of five algorithms including two popular heuristics, a representative OGA, the original GGA and our new GGA on solution quality and computation time.

Xiaoyong Xu and Maolin Tang are with the School of Electrical Engineering and Computer Science, Queensland University of Technology, Brisbane, Australia 4000 (email: {x21.xu, m.tang}@qut.edu.au).

978-1-4799-1488-3/14/$31.00 ©2014 IEEE

The rest of the paper is organized as follows. Section II discusses the related work, Section III formulates the problem, Section IV describes the new GGA, Section V shows the way to construct MRPP instances with known optimal solutions, Section VI presents the evaluation and Section VII concludes the study.

II. Related Work

MRPP can be seen as a generalization of the BPP, an NP-complete problem [3]. Several greedy algorithms have been proposed to solve the BPP and its variants. For instance, in [4] and [5] several variants of the first-fit-decreasing (FFD) algorithm were proposed to address a multi-constraint BBP. In these works, several ways to calculate the surrogate weights were investigated. In addition, Kang and Park [6] presented an iterative FFD (IFFD) especially for the variable sized BPP. Besides these greedy algorithms, the work [7] proposed a set-covering heuristic (SCH) for the variable sized BPP. Through the comparative study of SCH and other greedy algorithms, it was proven that SCH had the best performance on solution quality.

Recently, GAs [7] [9] [10] were introduced to solve the problems. Compared with the above mentioned heuristics, GAs have the ability of searching global optimums. In term of coding schemes, they are divided into two classes. The first class is OGA where the individuals are encoded as an ordered list of items. A typical example is the GA developed by Haouari and Serairi [7], which was used to address the variable sized BPP. In this GA, they introduced an acyclic digraph to compute the shortest path as the fitness value. However the calculation of fitness values is computation-intensive, as it has to enumerate all possible paths in the acyclic digraph, and the number of paths will increase exponentially as the problem size increases.

Another class is GGA where the individuals are encoded as a number of item groups. Falkenauer [8] firstly proposed a GGA to solve BPP. The GGA adopted a coding scheme in which the items in the groups were represented by their identities rather than their types. Thus, when using the coding scheme to MRPP, it will result in a large search space as there will be a so-called redundant representation problem, which will be discussed in Section IV.A. Moreover, it adopted a special genetic operator called inversion to help the crossover operator select different combination of groups to exchange between two parents. In addition, Iima and Yakawa [9] and Wilcox et al. [10] respectively modified the original GGA and adopted it to solve the BPP with the same bin size. However, since there are multiple VM types (or bin sizes) in MRPP, the GGAs proposed in [9] [10] are hardly applicable to MRPP.

To the best of our knowledge, few works have studied MRPP, thus a new GGA will be proposed to solve it in this paper.

III. Problem Formulation

When a set of jobs are submitted by an end user concurrently, a cluster of VMs needs to be generated to execute the jobs. Assume that, the set of workers to execute the jobs is $\mathcal{W} = \{w_1, w_2, ...w_n\}$, and the i^{th} worker w_i has the requirement r_{ih} for the h^{th} resource. r_{ih} can be estimated through using a profiling tool to compact the upper bounds of the resource consumption of the workers from the past job running or the sample tests. In this work, two resources, CPU ($h = 1$) and memory ($h = 2$) are considered, while other resources like I/O will be discussed in the future work. Moreover, K VM types are available. For the VM of type k ($k = 1, 2, ..., K$), the capacity of the h^{th} resource is denoted as R_{kh}, and the cost for renting this VM per hour is C_k. Note that, K, R_{kh} and C_k are all constants. Assume that each type of VMs has an infinite amount. Additionally there exists at least one VM type that has enough resource capacity to load any worker in $\mathcal{W}$.

We give the following definitions:

Definition 3.1: A *placement pattern* is a combination of workers placed on one VM. Let W_j be the j^{th} placement pattern, and $W_j \subset \mathcal{W}$; let $\mathrm{k}(W_j)$ be the type of the cheapest VM (denoted as $\mathrm{V}(W_j)$) being able to provide enough resources to meet the demands of all the workers in W_j. Then, the placement pattern W_j represents that all the workers in W_j are placed on the VM $\mathrm{V}(W_j)$ of type $\mathrm{k}(W_j)$. Moreover, $\mathrm{V}(W_j)$ cannot load any worker not in W_j.

Definition 3.2: The placement pattern W_j is feasible if and only if there exists at least one VM type k^*, such that $\sum_{\forall w_i \in W_j} r_{ih} \leq R_{k^*h}, h = 1, 2$.

Then, MRPP is formulated as follows: having known K types of VMs with infinite amounts, given a set of workers $\mathcal{W}$, the objective of MRPP is to find a worker placement solution

$$P = \{W_j | j = 1, 2, ..., m\}$$

which minimizes

$$Z(P) = \sum_{j=1}^{m} C_{\mathrm{k}(W_j)} \tag{1}$$

subject to

$$W_j \text{ is a feasible placement pattern }, \forall W_j \in P \tag{2}$$

$$W_j \cap W_{j'} = \emptyset, j \neq j' \tag{3}$$

$$\bigcup_{j=1}^{m} W_j = \mathcal{W} \tag{4}$$

$$m \in \mathbb{N}^+ \tag{5}$$

In the above formulation, m is a variable, representing the number of the placement patterns in P. The first constraint indicates that the total resource demands of the workers cannot exceed the capacities of the VMs on which these workers are placed. Note that, although not all workers start running concurrently, there is an overlap between the execution. Therefore, a VM should satisfy the total resource demands of the workers on this VM, even they do not start together. The second constraint means that each worker only can be allocated to a single VM. The third constraint denotes that all the workers should be placed on the VMs.

IV. A New Grouping Genetic Algorithm

In order to solve MRPP efficiently, we propose a new GGA in which a new coding scheme and a knowledge-based crossover are applied.

A. The Coding

In MRPP, multiple workers probably have the same resource demands, and it is defined that the workers with the same resource demands are on the same type. Then, a set of workers $\mathcal{W} = \{w_1, w_2, ..., w_n\}$ is divided into D worker types. For the workers of type d $(d = 1, 2, ..., D)$, they all have the same demands r_{dh} for the h^{th} resource. Additionally, let $\mathrm{d}(w_i)$ be the type of the worker w_i.

Next, we will describe the coding scheme in our GGA. Firstly, consider a worker placement solution $P = \{W_j | j = 1, 2, ..., m\}$, for any placement pattern W_j in P, it is encoded as a super set of worker types, called *group*, $g = \{\mathrm{d}(w_i) | \forall w_i \in W_j\}$. Then, P is encoded as a set of groups $G = \{g_1, g_2, ..., g_m\}$, and g_q is the j^{th} group in G. We take G as an individual in the population $\mathcal{G}$ of our GGA. Note that there is no sequence in the set of groups unlike the original GGA [8].

By using our coding scheme, an individual G probably represents a large number of worker placement solutions, as a number of workers can be on the same type, and multiple combinations of workers could be mapped to the same set of worker types. For example, having known that $\mathcal{W} = \{w_1, w_2, ..., w_8\}$, $d \in \{1, 2, 3\}$, w_1 and w_2 are on type 1, w_3 and w_4 are on type 2, w_5, w_6, w_7 and w_8 are on type 3, and an individual G is

$$\big\{\{1, 2\}, \{1, 3, 3\}, \{2, 3, 3\}\big\}$$

The group $g_1 = \{1, 2\}$ represents a combination of one worker of type 1 and a worker of type 2; $g_2 = \{1, 3, 3\}$ represents a combination of one worker of type 1, two workers of type 3; $g_3 = \{2, 3, 3\}$ represents a combination of one worker of type 2 and two workers of type 3. Since there are respectively two workers of type 1 and 2, four different placement patterns, including $\{w_1, w_3\}$, $\{w_1, w_4\}$, $\{w_2, w_3\}$ and $\{w_2, w_4\}$, are encoded as the same group g_1; similarly, there are respectively 12 different placement patterns encoded as the same group g_2 and g_3. Consequently, $24 (= A_2^2 A_2^2 C_4^2 C_2^2)$ different worker placement solutions in total can be encoded the same individual G.

Therefore, by using our coding scheme, the search space of our GGA is greatly reduced. By contrast, in the original GGA [8], the elements in each group are represented by unique identities, an individual just represents one worker placement solution, thus the search space for the original GGA could be very large. In addition, by using our coding scheme, there is no sequence of the groups, the inversion operator which is an essential genetic operator in the original GGA will not be considered in our GGA, which makes our GGA simpler.

B. Fitness Function

If we take Eq. (1) as the fitness function, a needle-in-a-haystack problem arises, which means the fitness function lacks the capacity of guiding the algorithm in the search [8], because a very small number of optimal solutions probably are lost in a large number of sub-optimal solutions yielding the same cost.

To overcome this problem, we define a new fitness function by Eq. (6),

$$F(G) = Z(G) + c_{min}\left(1 - \frac{f(G)}{\sum_{\forall G_o \in \mathcal{G}} f(G_o)}\right) \qquad (6)$$

where c_{min} is the basic unit of VM price, G_o is an individual and $f(G) = \sum_{j=1}^{m} f_j$ where f_j is the *fill ratio* of the VM $\mathrm{V}(W_j)$ which loads all workers in W_j, an indicator of the resource utilization of $\mathrm{V}(W_j)$, and f_j is expressed as

$$f_j = \sum_{h=1}^{2} \left(\frac{\sum_{\forall w_i \in W_j} r_{ih}}{R_{\mathrm{k}(W_j)h}}\right)^{\alpha} \qquad (7)$$

where $R_{\mathrm{k}(W_j)h}$ is the capacity of the h^{th} resource of the VM $\mathrm{V}(W_j)$ $1 < \alpha \leq 2$, here we follow the suggestion in [8] and set $\alpha = 2$. After applying the new fitness function, within the individuals with same $Z(G)$, the well-filled one will be preferred.

Next, we will propose a theorem proving that the solution achieves optimal when $F(G)$ is minimized. According to this theorem, our GGA will try to find the individual G with the minimal $F(G)$, instead of that with the minimal $Z(G)$.

Theorem 4.1: If $F(G)$ is minimized, the individual G achieves optimal.

Proof: Consider an individual G and the minimal value Z_{min} of $Z(G)$, obviously,

$$G \text{ achieves optimal} \Leftrightarrow Z(G) = Z_{min}$$

Rewrite Eq. (6), let $F(G) = Z(G) + \epsilon(G)$, where $\epsilon(G) = c_{min}\big(1 - \frac{f(G)}{\sum_{\forall G_o \in \mathcal{G}} f(G_o)}\big)$ and $0 < \epsilon(G) < c_{min}$.

In addition, let G^* be the individual minimizing $F(G)$. Assume that there is an individual $G^{'}$ $(G^{'} \neq G^*)$, $Z(G^{'}) < Z(G^*)$. Obviously, $Z(G^{'}) \leq Z(G^*) - c_{min}$ and $F(G^{'}) \geq F(G^*)$. Then, $\epsilon(G^{'}) \geq \epsilon(G^*) + c_{min}$, of course, it is not true. Thus, $G^{'}$ does not exist; in other words, $Z(G^*) = Z_{min}$, then the individual P^* is proven to be optimal. ■

C. Initial Population

We generate an initial population $\mathcal{G}$ containing S individuals by randomly placing all workers in $\mathcal{W}$ on the VMs of all possible types. The detail process of generating the initial population is described in Algorithm 1: at iteration q $(q = 1, 2, ..., K)$, and let $k = q$, we select a subset of workers W_k from $\mathcal{W}$, all of which can be placed on the VM of type k without exceeding the resource capacities, then we apply a random FFD to place W_k on the VMs of type k. In detail, the random FFD firstly calculates the surrogate weight l_i for each worker $w_i \in W_k$. Here we use the expression of the surrogate weight in [5], and $l_i = \sum_{h=1}^{2} a_h r_{ih}$ where

$a_h = \frac{\sum_i^n r_{ih}}{nR_{kh}}$. Then, W_k is sorted by the surrogate weight in a descending order. Unlike the traditional FFD preferring to place the worker with the largest surrogate weight, the random FFD will randomly place the largest but different a workers with probability proportional to $(1-b)^a$ $(0 < b < 1)$ on the VMs of type k. Note that, these a workers have the distinct surrogate weights. The rest part of random FFD is the same as the traditional one. After that, the worker placement patterns generated by the random FFD will be encoded as a set of groups G_{qk}. We repeat above steps, until all workers in $\mathcal{W}$ are assigned by the random FFD. Next, we will combine all groups generated at iteration q into an individual G_q, and insert it into the population $\mathcal{G}$. Then, we will check if the population size achieves S, if not, we will go to next iteration. Specially, if $\mathcal{G} < S$ after K iterations completes, the iteration will roll back the first one.

Algorithm 1 The initial population generation procedure

1: **Input:**$\mathcal{W}$;
2: **Output:** $\mathcal{G}$;
3: $\mathcal{G} \leftarrow \emptyset$;
4: **for** $q = 1$ to K **do**
5: $k = q$, $\widetilde{W} \leftarrow \mathcal{W}$;
6: $W_k = \{w_i \in \widetilde{W} | r_{ih} \leq R_{kh} h = 1, 2\}$;
7: apply the random FFD to place the set W_k of the workers on the VMs of type k, generating a set of groups G_{qk};
8: $\widetilde{W} \leftarrow \widetilde{W} - W_k$;
9: **if** $\widetilde{W} \neq \emptyset$ **then**
10: $k = k + 1$, go to 6;
11: **end if**
12: $G_q = \bigcup_{k=q}^{K} G_{qk}$;
13: $\mathcal{G} \leftarrow \mathcal{G} \cup G_q$;
14: **if** $|\mathcal{G}| == S$ **then**
15: return $\mathcal{G}$;
16: **end if**
17: **end for**
18: **if** $Size(\mathcal{G}) < S$ **then**
19: go to 4;
20: **end if**

D. The Crossover

Our GGA adopts a knowledge-based crossover to enhance its exploitation capacity. The knowledge-based crossover can discover and use the good placement patterns which can make best use of resources. The good placement patterns are discovered using a VM-centric placement procedure and FFD. The crossover consists of three steps.

The first step is *insertion*. Fig. 1 describes an example of insertion. In this step, several groups are randomly selected from a parent G_2, and then inserted into a group set denoted as t_1. Meanwhile, another parent G_1 is firstly sorted by fill ratio in an ascending order, and the elements of G_1 also occurred in t_1 are removed by order, then the groups including the removed elements are inserted to another group set t_2 while the rest groups are copied to t_1.

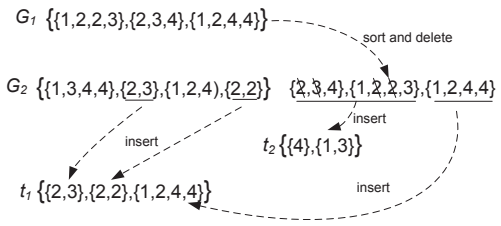

Fig. 1. Step 1: insertion

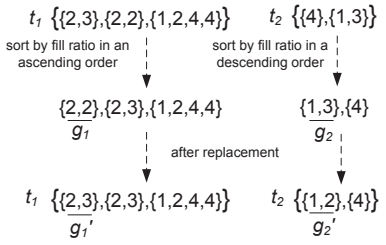

Fig. 2. Step 2: replacement

The second step is *replacement*. Fig. 2 describes an example of replacement. Firstly, t_1 and t_2 are sorted by fill ratio in ascending and descending orders respectively. After that, the first group in t_1 is replaced by the first three groups in t_2 one by one. In detail, the process of replacing a group (g_1) in t_1 by a group (g_2) in t_2 is described as follows: firstly, let $W_{1,2} = W_1 \cup W_2$, where W_1 and W_2 respectively represent the workers in one possible placement pattern decoded from g_1 and g_2 (recall that a group could represent multiple placement patterns), then let k_1 be the type of the cheapest VM being able to load W_1. Then, we place $W_{1,2}$ on the VMs of type k_1 by the VM-centric placement procedure (Algorithm 2). In this procedure, a VM will be used to load the worker with the largest surrogate weight in the current workers to be placed, until no more worker can be placed on this VM, then a new VM will be used. After applying Algorithm 2, a set of groups $G^{'}$ will be generated, if the fill ratio of g_1 increases, g_1 will be replaced by the first group $g_1^{'}$ in $G^{'}$ while g_2 will be replaced by the rest groups of $G^{'}$; otherwise, they will not. Once the first group in t_1 completes the replacement, t_2 will be sorted again, and then the next group in t_1 will repeat the previous procedures, until all groups in t_1 are replaced.

The last step is *combining*. As shown in Fig. 3, by regarding the worker set decoded from each group in t_1 and t_2 as a large worker, we apply the steps 3-13 in Algorithm 1 to place the large workers, but replacing the random FFD in Step 7 by a traditional FFD which adopts the same way to calculate the surrogate weight as the random FFD. Finally a set of individuals are returned. Then we choose the individual with the minimal fitness value in the set as the child. Through combining the groups, we not only reduce the number of groups in the child but also let the child inherit good building blocks from parents.

Algorithm 2 The VM-centric placement procedure

Input: $W_{1,2}$, k_1;
Output: $G^{'}$;
$q = 1$;
$G^{'} \leftarrow \emptyset$;
while $W_{1,2} \neq \emptyset$ **do**
  $g_q^{'} \leftarrow \emptyset$;
  select a new VM of type k_1, denoted as V_{k_1};
  while at least one worker in $W_{1,2}$ is able to be placed on V_{k_1} **do**
    place a worker w_i from $W_{1,2}$ with the largest surrogate weight l_i on V_{k_1} where $l_i = \sum_{h=1}^{2} a_h r_{ih} s_{k_1 h}$, $s_{k_1 h}$ is the rest space for the h^{th} resource on V_{k_1};
    insert the type of w_i into $g_q^{'}$;
    $W_{1,2} \leftarrow W_{1,2} - w_i$;
  end while
  $G^{'} \leftarrow G^{'} \cup g_q^{'}$ and $q = q + 1$;
end while

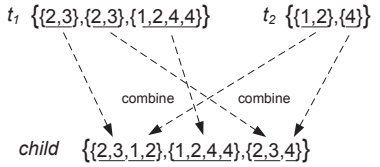

Fig. 3. Step 3: combining

E. The Mutation

Firstly, several groups are randomly generated from an individual G, and all workers decoded from these groups are moved to a temporary set $\mathcal{W}^{'}$. In addition, randomly select a VM type $k^{'}$ from all K VM types. Then, apply the steps from 6 to 12 in Algorithm 1, where $\mathcal{W} = \mathcal{W}^{'}$ and $q = k^{'}$. After that, a set of groups are generated. Combine these groups with the rest groups in G, and then an mutated individual comes out.

F. The Outline of Algorithm

With regard to the selection operator, the tournament selection will be used in our GGA. The outline of our GGA is presented in Algorithm 3, where F_{min} and G_{best} respectively denote the minimal fitness value and the best individual found by the algorithm, $Rate_{cro}$ and $Rate_{mut}$ respectively denote the crossover rate and mutation rate.

V. Construction of Test Instances with Known Optimal Solutions

In order to evaluate the quality of solutions for comparison, it is necessary to know the optimal solutions of the test instances. Here we present a procedure to construct test instances with known optimal solutions. Using this procedure, we can also construct test problems with various characteristics.

Algorithm 3 The grouping genetic algorithm for MRPP

Input: an initial population $\mathcal{G}$ containing S individuals;
Output: G_{best};
generate an initial population $\mathcal{G}$ containing S individuals;
while the termination condition is not true **do**
  the population of next generation, $\mathcal{G}^* \leftarrow \emptyset$;
  find the individual $P^{'}$ with the minimal fitness value from $\mathcal{G}$;
  if $F(P^{'}) < F_{min}$ **then**
    $F_{min} = F(G^{'})$;
    $G_{best} = G^{'}$;
  end if
  for $times = 1$ to S **do**
    select two parents G_1 and G_2 from $\mathcal{G}$ using selection operator;
    apply crossover operator with the probability of $Rate_{cro}$ on G_1 and G_2, generate a child G^*;
    apply mutation operator with the probability of $Rate_{mut}$ on G^*;
    $\mathcal{G}^* \leftarrow \mathcal{G}^* \cup P^*$;
  end for
  sort $\mathcal{G}$ by the fitness values of the individuals in a descending order, copy the first $\left(|\mathcal{G}| - |\mathcal{G}^*|\right)$ individuals in $\mathcal{G}$ to $\mathcal{G}^*$;
  $\mathcal{G} \leftarrow \mathcal{G}^*$;
end while

Firstly, consider a problem (denoted as P1), which minimizes

$$Z = \sum_{k=1}^{K} x_k C_k \tag{8}$$

subject to

$$\sum_{k=1}^{K} R_{kh} x_k \geq R_h^o, h = 1, 2 \tag{9}$$

$$x_k \in \mathbb{N} \tag{10}$$

where R_h^o is a constant, denoting the demands for the h^{th} resource.

P1 can be solved exactly by a MIP solver like CPLEX in a short time, since the number of variations, K, in the problem is very limited. Let $X^* = (x_1^*, x_2^*, ..., x_K^*)$ be an optimal solution to P1, which indicates the required number of each type VM to satisfy the resource demands R_h^o $(h = 1, 2)$ with the minimal costs. Additionally, let Z^* be the optimal solution value and $Z^* = \sum_{k=1}^{K} x_k^* C_k$.

Next, we present a theorem as follows:

Theorem 5.1: Consider a MRPP instance, denoted as $\mathcal{P}$, its input is a set of workers $\mathcal{W}$, and the total resource demands of the workers in $\mathcal{W}$ satisfy Eq. (11).

$$\sum_{\forall w_i \in \mathcal{W}} r_{ih} = R_h^o, h = 1, 2 \tag{11}$$

Let P be a solution to $\mathcal{P}$, and $P = \{W_j | j = 1, 2, ..., m\}$. Then, if Eq. (12) is satisfied, P is the optimal solution to $\mathcal{P}$,

unless it is infeasible.

$$|\mathcal{W}_k| = x_k^*, k = 1, 2, ..., K \tag{12}$$

$\mathcal{W}_k$ is the set of all placement patterns on the VM of type k, $\mathcal{W}_k = \{W_j | \mathrm{k}(W_j) = k, j = 1, 2, ..., m\}$, and $\bigcup_{k=1}^{K} \mathcal{W}_k = \bigcup_{j=1}^{m} W_j = P$.

Proof: If Eq. (12) is satisfied, according to Eq. (2), the objective value of P, $Z(P) = \sum_{k=1}^{K} |\mathcal{W}_k| C_k = \sum_{k=1}^{K} x_k^* C_k = Z^*$. Obviously, Z^*, the optimal solution value to P1, is the lower bound of $\mathcal{P}$. Therefore, once P is feasible, it must be an optimal solution to $\mathcal{P}$. ■

Then, we give the procedure of constructing a MRPP instance $\mathcal{P}$ with known optimal solutions (Algorithm 4). In steps 1-4, we solves the problem P1 and get the optimal solution $X^* = (x_1^*, x_2^*, ..., x_K^*)$. Then, in steps 5-7, we respectively determine the numbers of worker types and workers, D_k and n_k, for each VM type k ($k = 1, 2, ..., K$), and it is ensured that $\sum_{k=1}^{K} D_k = D$ and $\sum_{k=1}^{K} n_k = n$. In steps 8-12, we construct a set of feasible placement patterns on the VM of type k, $\mathcal{W}_k$ ($k = 1, 2, ..., K$), satisfying $|\mathcal{W}_k| = x_k^*$; meanwhile, the total resource demands of the workers in the placement pattern amount to the resource capacity of the VM of type k; moreover, D_k worker types and n_k workers are involved in $\mathcal{W}_k$. In step 13, a feasible worker placement solution $P = \{\mathcal{W}_1, \mathcal{W}_2, ..., \mathcal{W}_K\}$ is constructed, and the problem input is $\mathcal{W} = \{\mathcal{W}_1, \mathcal{W}_2, ..., \mathcal{W}_K\}$. Finally, in order to ensure Eq. (11) is satisfied, we apply step 14 to update $\mathcal{W}$.

Consequently, a MRPP instance $\mathcal{P}$, which has the input $\mathcal{W}$ satisfying Eq. (11) and a feasible solution P, has been constructed; meanwhile, Eq. (12) is satisfied. Then, according to Theorem 5.1, P is the optimal solution to $\mathcal{P}$ and Z^* is the optimal solution value.

Algorithm 4 The procedure of constructing a MRPP instance with known optimal solutions

1: **Input:** R_1^o, R_2^o, n, D;
2: **Output:** $\mathcal{W}$, P;
3: $\forall \mathcal{W}_k \leftarrow \emptyset$, $k = 1, 2, ..., K$;
4: solve the problem P1 using CPLEX, get the optimal solution $X^* = \{x_1^*, x_2^*, ..., x_K^*\}$;
5: $\mathcal{D} = \{D_1, D_2 ..., D_K\}$ where $D_k = \lceil x_k^* / \sum_{k=1}^{K+1} x_k^* \rceil$ $(k = 1, 2, ..., K)$;
6: randomly select a number $D_k > 1$ from $\mathcal{D}$, set $D_k = D_k - 1$, repeat this action until $\sum_{k=1}^{K+1} D_k = D$;
7: apply similar actions from steps 6-7 to construct $\mathcal{N} = \{n_1, n_2, ..., n_K\}$ and ensure $\sum_{k=1}^{K+1} n_k = n$;
8: **for** $k = 1$ to K **do**
9: if $x_k^* = 0$, set $k = k + 1$, then go to the next loop;
10: construct a set of placement patterns, W_{k1}, W_{k2}, ..., $W_{kx_k^*}$, and for the q^{th} placement pattern W_{kq} $(q = 1, 2, ..., x_k^*)$, $\sum_{w_i \in W_{kj}} r_{ih} = R_{kh}$, meanwhile, the total numbers of worker types and workers in these placement patterns respectively are D_k and n_k;
11: set $\mathcal{W}_k = \{W_{k1}, W_{k2}, ..., W_{kx_k^*}\}$;
12: **end for**
13: $P \leftarrow \{\mathcal{W}_1, \mathcal{W}_2, ..., \mathcal{W}_K\}$, $\mathcal{W} \leftarrow \{\mathcal{W}_1, ..., \mathcal{W}_K\}$;
14: randomly choose a worker type d^* involved in $\mathcal{W}$, assume that the number of workers of type d^* is n^*, $r_{d^*h} = (\sum_{i=1}^{n^*} r_{ih} - (\sum_{k=1}^{K} x_k^* r_{kh} - R_h^o))/n^*$, then the demand for the h^{th} resource of any worker of type d^* is updated to r_{d^*h}, if $r_{d^*h} \leq 0$, repeat this step until $r_{d^*h} > 0$;

VI. Evaluation

In this section, we will evaluate the performance of our new GGA (denoted as GGA-II) and compare it with IFFD [4] which adopts several ways given by [6] to calculate the surrogate weights, SCH [7], a representative OGA [7] and the original GGA (denoted as GGA) [8] on solution quality and computation time. Note that we will respectively run IFFD with different methods given by [6] for calculating surrogate weight, and compare the best results of IFFD with the results of the other algorithms. All these algorithms are coded in C#, and the integer programming problem involved in Section V is solved by CPLEX 12.5.1.0. The algorithms are implemented on a laptop with 4 cores (2.90 GHz Intel Core i7-3520M CPU) and 8 GB RAM.

The parameter settings for the three GAs are presented in Table I. In particular, for the parameter settings in OGA and GGA, we respectively follow the suggestions in [7] and [8], while for the parameter settings in GGA-II, we just choose the best ones from several tests due to the time limitation, but fine parameter tuning will be conduct in the future. Furthermore, both the tournament sizes in GGA and GGA-II are two. The termination condition of all the three GAs is that the number of consecutive non-improving generations before stopping is up to 50. Moreover, for the maximal time of solving the set-covering problem in SCH, we follow the configuration in [7], setting it to 30 seconds.

TABLE I
THE PARAMETER SETTINGS FOR THE GAS

Parameters	OGA	GGA	GGA-II
Population size	200	100	30
Crossover rate	0.9	0.5	0.9
Mutation rate	0.9	0.33	0.1

A. Test Instances

Firstly, eight VM types from Amazon EC2 (shown in Table II) are involved in the test instances. Note that the amounts of the h^{th} resource a type k VM possesses, shown in Table II, is not equal to its capacity R_{kh}, but equal to $R_{kh} + R_{kh}^o$, where R_{kh}^o is a constant, representing the h^{th} resource consumption of an idle k type VM. We set $R_{k1}^o = 0$ and $R_{k2}^o = 0.3$, where $k = 1, 2, ..., K$.

Then, we construct a group of MRPP instances to test the algorithms by Algorithm 4. All instances are divided into three sets:

- Set 1. The test instances are generated by Algorithm 4 in which the parameters are configured as follows:

TABLE II
THE VM TYPES FROM AMAZON EC2

VM Type	CPU (cores)	Memory (GB)	Price ($/hour)
m1 small	1	1.7	0.06
m1 medium	2	3.75	0.12
m1 large	4	7.5	0.24
m1 xlarge	8	14.7	0.48
m2 xlarge	6.5	17.1	0.41
m2 2xlarge	13	34.2	0.82
c1 medium	5	1.7	0.145
c1 xlarge	20	7	0.58

$R_1^o = 400$, $R_2^o = 800$, $D \in \{16, 24, 32\}$, $n \in \{150, 200, 250, 300\}$, and their optimal solution values are 21.675.

- Set 2. The test instances are generated by Algorithm 4 in which the parameters are configured as follows: $R_1^o = R_2^o = 600$, $D \in \{16, 24, 32\}$, $n \in \{150, 200, 250, 300\}$, and their optimal solution values are 23.455.
- Set 3. The test instances are generated by Algorithm 4 in which the parameters are configured as follows:$R_1^o = 800$, $R_2^o = 400$, $D \in \{16, 24, 32\}$, $n \in \{200, 250, 300, 350\}$, and their optimal solution values are 25.27.

B. Results and Discussion

We repeat running SCH, OGA, GGA and GGA-II on each test instances 10 times, while running IFFD just once since it is not stochastic.

Table III presents the gaps between the solution values found by the algorithms and the optimal solution values (Z^*). The gap can be expressed by $\frac{Z-Z^*}{Z^*} \times 100\%$, especially when the gap is 0, it means the solution is optimal. In Table III, the column D means the number of worker types and n represents the number of workers; the columns *Min* and *Avg* respectively stand for the minimal and average gaps.

As seen from Table III, the solutions obtained by GGA-II are much better than those found by the other algorithms. GGA-II is the only algorithm being able to find the optimal solutions and the gaps of the solutions found by GGA-II are much lower than those by the other algorithms. In addition, the quality of solutions found by GGA-II changes slightly as the problem size varies, showing the stability of GGA-II.

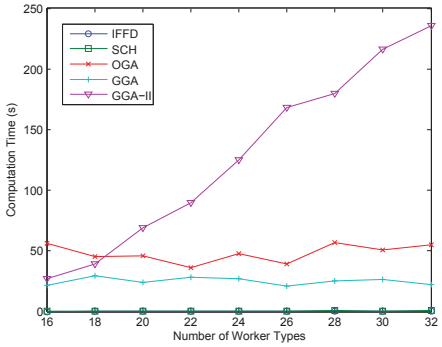

Fig. 4. The variations in the average computation time of the algorithms as the number of worker types increases

Fig. 4 displays the variations in the average computation time of the five algorithms as the number D of worker types increases from 16 to 32, when n is fixed at 150 and the test instances are in Set 2. Doubtlessly, IFFD is the fastest one while SCH ranks second. Among the three GAs, the fastest one is GGA. As D increases, the average computation times of the algorithms except GGA-II changes slightly, the reason is D is not a input involved in these algorithms. On the contrary, the average computation time of GGA-II increases linearly as D increases.

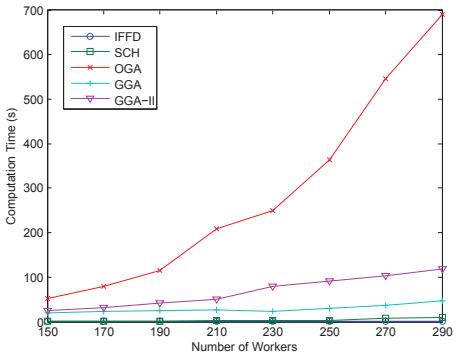

Fig. 5. The variations in the average computation time of the algorithms as the number of workers increases

Fig. 5 displays the variations in the average computation time of the five algorithms as the number n of workers increases from 150 to 290, when D is fixed at 16 and the test instances are in Set 2. Similarly, IFFD is the fastest one while SCH ranks second. Both the average computation times of GGA and GGA-II increase linearly as n increases, while the computation time of OGA increases much more dramatic than the other two GAs. Considering that the trends in the average computation times of the algorithms when the test instances are in set 1 and 3 are similar to that in Set 1, we will not illustrate them.

In summary, compared with the other four algorithms, GGA-II is much more efficient for MRPP. On the one hand, GGA-II is the only one being able to find the optimal solutions, and the gap of the solutions found by GGA-II is much lower than those found by the other algorithms, furthermore, GGA-II shows its stability as the problem size varies. On the other hand, on the term of computational time, although GGA-II is not the fastest one, it is much scalable as its computation time increase linearly when the problem size increases.

VII. CONCLUSION

In this paper, a new problem of the MapReduce in cloud computing called MRPP has been studied and a new GGA has been proposed to solve it. Different from the original GGA, our GGA uses an innovative coding scheme in which the items in the groups are represented by the worker types rather than the worker identities, which can greatly reduce the search space of the algorithm. It also applies a knowledge-based crossover which can find more promising groups in

TABLE III

THE COMPUTATIONAL RESULTS

Set	D	n	IFFD (%)	SCH		OGA		GGA		GGA-II	
				Min (%)	Avg (%)	Min (%)	Avg (%)	Min (%)	Avg (%)	Min (%)	Avg (%)
1	16	150	25.33	10.45	10.98	7.98	8.15	2.28	2.37	0.81	0.86
		200	40.99	25.47	27.88	11.84	13.44	3.39	4.08	2.28	3.15
		250	19.86	10.7	12.03	9.02	10.95	3.84	4.19	0.28	0.59
		300	15.87	9.57	11.04	8.76	10.13	3.4	3.92	0.28	0.42
	24	150	19.63	9.83	10.99	7.54	8.12	3.07	3.55	0.55	1.07
		200	23.69	14.81	15.82	10.4	12.31	2.73	3.41	1.06	1.61
		250	31.76	17.85	18.51	13.32	14.98	2.61	3.86	0.55	1.49
		300	22.08	12.3	14.37	11.08	12.56	3.73	3.88	0.95	1.2
	32	150	24.38	12.16	13.19	9.67	10.74	3.73	4.19	1.18	1.81
		200	18.85	12.53	13.53	8.85	10.28	2.81	3.27	1.59	1.81
		250	18.75	12.34	14.16	11.24	12.51	2.73	3.14	0.28	1.08
		300	19.77	12.9	14.57	9.76	10.6	2.38	3.98	0.83	1.25
2	16	150	41.21	11.94	13.11	10.85	12.32	3.69	4.24	0	0.53
		200	40.91	12.81	13.33	8.02	9.5	2.68	3.69	0.66	1.59
		250	43.68	30.7	32.42	15.24	17.67	3.73	4.52	1.24	1.48
		300	46.51	23.7	25.38	13.33	15.56	2.66	3.07	0	0.88
	24	150	53.31	17.69	20.2	9.02	10.35	3.13	3.33	0	0.54
		200	42.06	10.64	10.96	7.45	8.58	2.62	3.86	0.77	1.58
		250	46.45	13.62	14.89	9.62	10.45	2.41	3	1.02	1.95
		300	45.9	20.68	21.18	12.69	14.26	2.6	3.12	0	1.2
	32	150	56.58	15.35	16.07	13.56	15.9	3.75	4.11	0.26	1.98
		200	42.83	24.6	26.78	14.24	16.2	2.52	3.22	1.79	2.25
		250	47.94	17.42	18.18	12.68	14.21	4.11	4.78	1.79	2.89
		300	42.36	13.96	14.65	10.03	11.28	4.94	4.43	0.26	2.21
3	16	200	67.91	10.65	11.76	12.5	13.42	3.42	3.81	1.62	1.72
		250	61.14	22.44	23.59	16.98	18.29	6.33	7.57	2.37	3.31
		300	56	15.63	16.38	15.23	16.56	4.56	5.42	2.65	3.77
		350	65.65	19.57	20.4	12.75	14.28	2.83	3.28	0.47	0.88
	24	200	67.97	11.97	12.98	12.65	14.33	3.62	4.09	0.24	0.47
		250	63.59	23.72	24.69	14.82	17.9	4.1	4.87	1.72	2.46
		300	68.62	16.66	17.03	14.31	15.51	3.36	4.08	1.5	2.16
		350	66.92	21.29	22.25	10.56	12.65	3.03	4.12	0.47	2.01
	32	200	61.77	12.35	12.88	8.9	10.55	3.15	4.03	0.47	1.39
		250	67.39	20.89	21.94	13.45	14.92	2.47	3	0.47	1.45
		300	59.97	13.57	14.23	13.27	14.6	4.1	4.79	1.42	2.89
		350	57.02	8.09	8.36	7.14	9.14	3.18	3.92	0.47	1.56

the individuals so that its exploitation capacity is enhanced. Moreover, since the sequence of the groups is not significant, the inversion which is an essential genetic operator in the original GGA is removed from our GGA to make our GGA simpler.

We have also compared our GGA with IFFD, SCH, a representative OGA and the original GGA. The computational results have shown that our GGA is much more efficient than the other four algorithms for MRPP and the solutions found by our GGA are much better than those found by the other algorithms. Furthermore, our GGA is scalable, since its computation time increases linearly when the problem size increases.

ACKNOWLEDGMENT

This research was funded by the State Scholarship Fund of the China Scholarships Council (CSC) and the CSC Top-Up Scholarship of Queensland University of Technology, Australia.

REFERENCES

[1] A. Verma, L. Cherkasova, and R. H. Campbell, "Aria: automatic resource inference and allocation for mapreduce environments," in *Proceedings of the 8th ACM international conference on Autonomic computing*. ACM, 2011, pp. 235–244.

[2] X. Xu and M. Tang, "A comparative study of the semi-elastic and fully-elastic mapreduce models," in *Proceedings of the 2013 IEEE International Conference on Granular Computing (GrC)*. IEEE, 2013, pp. 380–385.

[3] M. R. Gary and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-completeness*. WH Freeman and Company, New York, 1979.

[4] A. Caprara and P. Toth, "Lower bounds and algorithms for the 2-dimensional vector packing problem," *Discrete Applied Mathematics*, vol. 111, no. 3, pp. 231–262, 2001.

[5] R. Panigrahy, K. Talwar, L. Uyeda, and U. Wieder, "Heuristics for vector bin packing," *http://research.microsoft.com*, 2011.

[6] J. Kang and S. Park, "Algorithms for the variable sized bin packing problem," *European Journal of Operational Research*, vol. 147, no. 2, pp. 365–372, 2003.

[7] M. Haouari and M. Serairi, "Heuristics for the variable sized bin-packing problem," *Computers and Operations Research*, vol. 36, no. 10, pp. 2877–2884, 2009.

[8] E. Falkenauer, "A hybrid grouping genetic algorithm for bin packing," *Journal of Heuristics*, vol. 2, no. 1, pp. 5–30, 1996.

[9] H. Iima and T. Yakawa, "A new design of genetic algorithm for bin packing," in *Evolutionary Computation, 2003. CEC '03. The Congress on*, 2003, pp. 1044–1049.

[10] D. Wilcox, A. McNabb, and K. Seppi, "Solving virtual machine packing with a reordering grouping genetic algorithm," in *Evolutionary Computation (CEC), 2011 IEEE Congress on*, 2011, pp. 362–369.