This is the author's version of a work that was submitted/accepted for publication in the following source:

Xu, Xiaoyong & Tang, Maolin  
(2014)  
A more efficient and effective heuristic algorithm for the MapReduce placement problem in cloud computing. In  
Proceedings of the 2014 International Conference on Cloud Computing,  
IEEE, Anchorage, Alaska, USA, pp. 264-271.

This file was downloaded from: http://eprints.qut.edu.au/70354/

© Copyright 2014 Please consult the authors

Notice: Changes introduced as a result of publishing processes such as copy-editing and formatting may not be reflected in this document. For a definitive version of this work, please refer to the published source:

http://dx.doi.org/10.1109/CLOUD.2014.44
A More Efficient and Effective Heuristic Algorithm for the MapReduce Placement Problem in Cloud Computing

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

Abstract—The placement of the mappers and reducers on the machines directly affects the performance and cost of the MapReduce computation in cloud computing. From the computational point of view, the mappers/reducers placement problem is a generalization of the classical bin packing problem, which is NP-complete. Thus, in this paper we propose a new heuristic algorithm for the mappers/reducers placement problem in cloud computing and evaluate it by comparing with other several heuristics on solution quality and computation time by solving a set of test problems with various characteristics. The computational results show that our heuristic algorithm is much more efficient than the other heuristics. Also, we verify the effectiveness of our heuristic algorithm by comparing the mapper/reducer placement for a benchmark problem generated by our heuristic algorithm with a conventional mapper/reducer placement. The comparison results show that the computation using our mapper/reducer placement is much cheaper while still satisfying the computation deadline.

Keywords—MapReduce; cloud computing; mapper/reducer placement; heuristics;

I. INTRODUCTION

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

In the cloud, MapReduce is operated in a different way from that in a traditional cluster. Once an end user submits its MapReduce jobs, a dedicated cluster of virtual machines (VMs) rented from an Infrastructure-as-a-Service provider is generated, and then the jobs start running on the cluster. Once the jobs are completed, the cluster closes and the end user pay for the usage of the VMs.

In MapReduce, a job is executed by a set of mappers and reducers which are respectively used to execute the map tasks and reduce tasks in the job. Both of them are called workers. When a set of jobs are submitted concurrently, each job needs a set of workers to run. The workers for different jobs probably have heterogeneous demands for resources like CPU, memory and so on. Even during a single job execution, the workers could have different resource demands, since they have two different kinds of tasks to run, the map tasks and reduce tasks.

These workers need to be placed on VMs, such that they can acquire the resources provided by the VMs, such as CPU, memory and so on, to execute the jobs. However, an inefficient worker placement usually leads to a poor match for the resource demands of workers. For example, placing too many workers on one VM probably raises the performance degradation caused by resource competition. In contrast, placing too few workers on one VM lowers the resource utilization although the resource requirements of workers are met. Furthermore, in cloud computing, multiple kinds of VMs with different capacities and costs are usually available. Unreasonable VM selection also could raise above problems.

Then, a new problem called Mapper/Reducer Placement Problem (MRPP) is raised and needs to be addressed. The objective of MRPP is to place all the workers including mappers and reducers for MapReduce on the VMs with different capacities and costs such that the costs of the VMs are minimized while the resource demands of workers are met. MRPP can be seen as a type of Bin Packing Problems (BPPs). But compared with the classic BPP, it has several special features: 1. multiple types of VMs (bins) with different resource capacities and costs are available to load the workers (items); 2. there are multiple resource constraints on the worker placement; 3. a number of workers probably have the same resource demands, since some workers apply the same operations on the input with the similar size.

In order to run the MapReduce with the minimal cost in cloud computing while satisfying the performance requirements, most current works [1] [2] [3] have studied the optimal number of mappers/reducers to run a job in MapReduce, but few of them has studied MRPP. In these works, a fixed number of mappers and reducers are assigned to each type of VM. Such a conventional placement probably causes a poor match for the resource demands of workers, harming the job performance or raising resource waste. In
addition, MRPP is NP-hard, since it is a generalization of BPP, which has proven to be NP-complete [4]. A number of heuristics [5] [6] [7][8] have been proposed to solve BPP and its variants, but none of them are specially designed for MRPP.

Therefore, this work will study MRPP and propose a new heuristic algorithm for it. To evaluate the efficiency of our heuristic algorithm, we will compare it with a greedy algorithm and a set-covering heuristic (SCH) by solving the problem instances from a real system on the cost and the duration of the MapReduce computation. In addition, to evaluate the effectiveness of our heuristic algorithm, we will compare the placement solution obtained by it with the conventional placement solutions adopted in current works [1] [2] [3] which have not considered MRPP.

The rest of the paper is organized as follows. Section II presents the related work, Section III formulates the problem, Section IV a new heuristic algorithm for the problem, Section V presents the evaluation and Section VI concludes the study.

II. RELATED WORK

MRPP has rarely been studied. Some works [1] [2] [3] have studied the optimal number of mappers/reducers provided for MapReduce, but they all ignore MRPP. In [1], optimal numbers of mappers and reducers for running MapReduce jobs with performance goals were determined by a performance model. Then, one mapper and one reducer were assigned to one machine, such that the number of machines was determined. Actually, the number of machines determined in such way usually is not optimal, because it ignores the resource demands of workers as well as the resource capacity of machines. Similarly, the work [2] also placed a fixed number of mappers and reducers on VMs, ignoring the heterogeneity of the resource demands of workers as well as the resource capacity of VMs. The work [3] placed the workers on VMs in a conventional way. It allocated one mapper and one reducer to a small VM while two mappers and two reducers to a medium VM. All these workers ignored the influence of mapper/reducer placement on the performance and cost of the MapReduce computation.

MRPP can be seen as a generalization of BPP, which has been proven to be NP-complete [4]. There are several greedy algorithms having been proposed to solve BPP and its variants. For instance, in [5] and [7] several variants of first-fit-decreasing (FFD) algorithm were proposed to address multi-constraint BPP. In these works, several ways to calculate the surrogate weights were investigated. Kang and Park [6] presented an iterative FFD (IFFD) especially for the variable sized BPP. In addition, the works [8] [9] adopted SCH to solve the multi-constraint and variable sized BPP by transforming the BPP to a set-covering problem.

To the best of our knowledge, few works have studied MRPP, thus in this work, we will study it and propose a new heuristic algorithm to solve it.

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, just as shown in Fig. 1. Assume that, the set of workers to execute the jobs is $W = \{w_1, w_2, ..., w_n\}$, and the $i$th worker $w_i$ has the requirement $r_{ih}$ for the $h$th type of resource, which 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 types of resource, CPU ($h = 1$) and memory ($h = 2$) are considered, while other resources like I/O will be discussed in the future work. These workers are categorized into $D$ types in term of the resource demands of workers. The amount of the workers of type $d = 1, 2, ..., D$ is denoted by $q_d$ and $\sum_{d=1}^{D} q_d = n$. All the workers on the same type have the same resource demands, denoted as $r_{dh}$. Moreover, $K$ VM types are available. For the VM on the type $k = 1, 2, ..., K$, the capacity of the $h$th resource is denoted by $R_{kh}$, 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 VM has an infinite amount. Additionally there exists at least one type of VM which has enough resource capacity to load any worker in $W$. Next, we give the following definitions:

**Definition III.1.** A placement pattern is a combination of workers placed on a single VM. Let $P_{kj}$ be the $j$th placement pattern in which the type $k$ VM is used to load the set $W_{kj}$ of workers, which can be expressed as a $D$-dimensional vector:

$$P_{kj} = (p_{kj1}, p_{kj2}, ..., p_{kjD})$$

where $p_{kjd}$ ($d = 1, 2, ..., D$) is an integer indicating the number of the type $d$ workers in the placement pattern $P_{kj}$.
Definition III.2. The placement pattern \( P_{kj} \) is a feasible placement pattern if and only if
\[
\sum_{d=1}^{D} p_{kjd} \cdot r_{dh} \leq R_{kh}, h = 1, 2
\]

Then, MRPP is formulated as follows: given a set of workers \( W \), the objective of MRPP is to find a set of placement patterns which places the set of workers \( W \) on the \( K \) types of VMs:
\[
P = \{ P_{kj} : k = 1, 2, \ldots, K; j = 1, 2, \ldots, m_k \}
\]
which minimizes
\[
Z = \sum_{\forall P_{kj} \in P} c_k \cdot x_{kj}
\]
subject to
\[
\sum_{\forall P_{kj} \in P} p_{kjd} \cdot x_{kj} = q_d, d = 1, 2, \ldots, D
\]
\( P_{kj} \) is a feasible placement pattern, \( \forall P_{kj} \in P \)
\[
x_{kj}, m_k \in \mathbb{N}, k = 1, \ldots, K, j = 1, \ldots, m_k
\]

Eq. (1) is the objective function, representing the total cost of the VMs used to load the workers. Eq. (2) means that all workers need to be placed. Eq. (3) 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 the workers placed on the same VM start running concurrently, probably there is an overlap between the execution of these workers. Therefore, a VM should satisfy the total resource demands of the workers placed on this VM, even they do not start together. Eq. (4) means \( x_{kj} \) and \( m_k \) are non-negative integers. Particularly, \( m_k = 0 \) means there is no VM of type \( k \) used to load the workers.

Note that the problem formulated in this paper is just an initial placement problem in which all workers should be placed concurrently, while the dynamic placement problem in which workers are allowed to be placed in a consecutive way will be studied in the future.

IV. A NEW HEURISTIC ALGORITHM FOR MRPP

In the problem formulation Eq. (1-4), the number of variables is equal to \(|P|\). As the number of workers \( n \) or worker types \( D \) increases, \(|P|\) will increase exponentially and becomes very huge. It is hard to find the optimal solutions in a reasonable time. Thus, we propose a new heuristic algorithm to solve it, which consists of two phases, the placement pattern generation phase and the placement pattern optimization phase.

In the first phase, we generate a set of attractive feasible placement patterns \( \tilde{P} \) instead of all feasible placement patterns \( P \), and \( \tilde{P} \subset P \). Then in the second phase, we try to optimize the combinations of the placement patterns in \( \tilde{P} \), and the solution to this combination problem is taken as a solution to MRPP. The details of our heuristic algorithm are described as follows.

A. Placement Pattern Generation

In this phase, Algorithm 1 is used to generate the placement patterns. In this algorithm, to generate more attractive feasible placement, all the VM types are enumerated, and on each VM type, a subproblem needs to be solved: given a VM type \( k \) and a set \( W_k \) of workers whose resource demands will not exceed the capacity of the VMs of type \( k \), minimize the total cost of the type \( k \) VMs used to load the workers \( W_k \) without the violation of capacity constraints. A VM-centric placement procedure (Algorithm 2) is adopted to solve this subproblem. In this procedure, a VM is used to load the worker with the largest surrogate weight in current set of workers, until no more worker can be placed on this VM, then a new VM is used. After applying Algorithm 2, a set of placement patterns \( \tilde{P}_k \) are generated. After all the VM types are enumerated, \( \tilde{P}_1, \tilde{P}_2, \ldots, \tilde{P}_K \) are generated.

After the sets of placement patterns on each VM type \( k \) are generated, there probably exist some placement patterns which are not attractive. For instance, consider a placement pattern \( P_{kj} \), if there is a VM type \( k' \), \( c_{k'} < c_k \) and \( P_{kj} \) is also feasible for the VM of type \( k' \), obviously it is reasonable to replace \( P_{kj} \) by \( P_{k'j} \). Thus, to enhance the quality of the placement patterns, we will adopt a procedure called fill-to-cheapest-VM, which is executed as follow: enumerate all generated placement patterns, for a placement pattern \( P_{kj} \), if there is a VM type which is the cheapest one making \( c_{k'} < c_k \) true and \( P_{kj} \) is also feasible for the VM of type \( k' \), then replace \( P_{kj} \) by \( P_{k'j} \).

Next, since there probably exist some redundant placement patterns, we remove them in the following steps: firstly, each placement pattern is assigned with five tags, including the heaviest and lightest surrogate weights, the cumulated surrogate weight and the total number of the workers in the placement, and the variance among the surrogate weights involved in the placement; then, if two or more placement patterns have the same tags, one of them is reserved while the rest ones are removed. Consequently, all reserved placement patterns are combined into the set \( \tilde{P} \).

However, the number of placement patterns generated by Algorithm 1 is very limited. In order to generate as many attractive placement patterns probably included in the optimal solutions as possible, we repeat conducting the whole steps in Algorithm 1 several times, but replacing Algorithm 2 in it with a random version of Algorithm 2. The random version is the same as Algorithm 2, except for Step 6. In Step 6 of the random version of Algorithm 2, it randomly selects the first but different \( a \) workers in the sorted list with a probability proportional to \((1 - b)^a\) \((b \in (0, 1))\) to place, instead of selecting the first one. Note that these \( a \) workers have the distinct surrogate weights.
Algorithm 1 The placement pattern generation procedure

1: Input: \( W \);
2: Output: \( \overline{\mathcal{P}}_1, \overline{\mathcal{P}}_2, \ldots, \overline{\mathcal{P}}_K \);
3: \( \overline{\mathcal{P}}_1, \overline{\mathcal{P}}_2, \ldots, \overline{\mathcal{P}}_K \leftarrow \emptyset ; \)
4: for \( m = 1 \) to \( K \) do
5: \( k = m, W \leftarrow W \);
6: \( W_k = \{ w_i \in W : r_{ih} \leq R_{kh}, h = 1, 2 \} ; \)
7: \( \overline{\mathcal{P}}_k \leftarrow \text{Algorithm } 2(W_k, k) ; \)
8: Implement the fill-to-cheapest-VM procedure on \( \overline{\mathcal{P}}_k ; \)
9: Remove the redundant placement in \( \overline{\mathcal{P}}_k ; \)
10: \( \overline{\mathcal{P}}_k \leftarrow \overline{\mathcal{P}}_k \cup \overline{\mathcal{P}}_k ; \)
11: \( W \leftarrow W - W_k ; \)
12: if \( W = \emptyset \) then
13: \( k = k + 1, \) go to Step 6;
14: end if
15: end for

Algorithm 2 The VM-centric placement procedure

1: Input: \( W_k, k \);
2: Output: \( \overline{\mathcal{P}}_k \);
3: \( \overline{\mathcal{P}}_k \leftarrow \emptyset ; \)
4: while \( W_k \neq \emptyset \) do
5: select a new VM of type \( k \), denoted by \( v_{kj} \), initialize the placement pattern \( P_{kj} \) in which \( \forall d = 1, 2, \ldots, D, P_{kj} = 0 ; \)
6: while at least one worker in \( W_k \) is able to be placed on the selected VM do
7: sort the workers in \( W_k \) in ascending order by the surrogate weight \( L_d = \sum_{h=1}^{2} a_h \cdot r_{dh} \cdot s_{kjh} \), where \( a_h \) is the resource weight and \( a_h = \sum_{i=1}^{n} r_{ih} \cdot R_{kh} \) is the resource demand of the worker of type \( d \) for the \( h \)th type of resource and \( s_{kjh} \) is the rest space for the \( h \)th type of resource on \( v_{kj} ; \)
8: place the first worker \( w_i \) (assume that its type is \( d \) in the sorted list on \( v_{kj} ; \)
9: set \( P_{kj} = P_{kj} + 1 ; \)
10: \( W_k \leftarrow W_k - w_i ; \)
11: end while
12: \( \overline{\mathcal{P}}_k \leftarrow \overline{\mathcal{P}}_k \cup P_{kj} ; \)
13: end while

B. Placement Pattern Optimization

In this phase, we try to optimize the combinations of the placement patterns in \( \overline{\mathcal{P}} \), and the solution to this combination problem is taken as the solution to MRPP. The combination problem is formulated as follows:

\[
\text{minimize } Z = \sum_{\forall P_{kj} \in \overline{\mathcal{P}}} c_k \cdot x_{kj} \tag{5}
\]

subject to

\[
\sum_{\forall P_{kj} \in \overline{\mathcal{P}}} p_{kj} \cdot x_{kj} \geq q_d, \forall d = 1, 2, \ldots, D \tag{6}
\]

\[
P_{kj} \text{ is a feasible placement pattern } , \forall P_{kj} \in \overline{\mathcal{P}} \tag{7}
\]

\[
x_{kj}, m_k \in \mathbb{N}, k = 1, \ldots, K, j = 1, \ldots, m_k \tag{8}
\]

In this formulation, the constraint Eq. (6) relaxes Eq. (2), overcoming the drawback which leads to high degeneracy and numerical instability.

Although the input of this formulation \( |\overline{\mathcal{P}}| \) is much smaller than \( |\mathcal{P}| \), it still could be large when the number of workers or worker types is very huge, and it will take a long time to find its optimal solutions. Therefore, instead of solving this combination problem exactly, we solve it by using a standard MIP solver given a preset maximal solving time.

V. Evaluation

In this section, two experiments are conducted. In the first experiment, to evaluate the efficiency of our heuristic algorithm, we firstly construct a set of test instances, and then compare our heuristic algorithm with other several heuristic algorithms on the solution quality and computation time by solving these test instances. In the second experiment, to evaluate the effectiveness of our heuristic algorithm, we generate the mapper/reducer placement solution by using our heuristic algorithm to solve MRPP, and then compare it with the conventional placement solutions which have not considered MRPP and just place a fixed number of workers on the VMs by running a real-world MapReduce job.

All the VMs used in the experiments are deployed on three physical machines (32 Intel Xeon 2.40GHz CPUs and 320 GB memory) which are connected into a Gigabit ethernet network. The VMs are generated by Virtual Box. In addition, we run the MapReduce jobs by using Hadoop framework (1.2.1) and use Ganglia to monitor the resource consumption during the job runtime.

A. Test Instances

The process of constructing test instances is described as follows.

Firstly, eight VM types from Amazon EC2 (shown in Table I) are involved in these instances and they are categorized into three groups: the general purpose VMs (m1 small, m1 medium, m1 large and m1 xlarge), the memory optimized VMs (m2 xlarge and m2 2xlarge) and the compute optimized VMs (c1 medium and c1 xlarge). All the CPUs in these VMs are Intel Xeon processors. Note that the amounts of the \( h \)th resource a type \( k \) VM possesses, shown in Table I, is not equal to its capacity \( R_{kh} \), but equal to \( R_{kh} + R_{kh}^p \), where \( R_{kh}^p \) is a constant representing the consumption of the \( h \)th resource of an idle \( k \) type VM. It is observed that \( R_{kh}^p = 0 \) and \( R_{kh}^p = 0.3, \forall k = 1, 2, \ldots, K \). Then we record the resource demands of workers by running two
popular benchmarks, WordCount and Terasort, on a cluster of 12 VMs (4 CPUs and 8 GB memory). WordCount is an application that counts the number of occurrences of each word in a text file and Terasort is a standard MapReduce sort benchmark [10]. We run the jobs with variable input size and repeat 10 times for each job. Specifically, we configure the number of reduce tasks as 5 for all jobs. Table II presents the resource consumption of job execution. As described in Table II, for the mappers, the variation ranges in resource demands are similar since the input size of each task is independent from the total input size. Thus, the resource demands of mappers in our test instances are uniformly distributed in the demands variation range \([a, b]\). By contrast, resource demands increase as the total input size increases when the number of reduce tasks is fixed. Then, various types of reducers are generated in the way described as follows. Firstly, four linear regressions are respectively preformed between the input size \(x\) and the CPU and memory demands \((y_{mc}^{wc} \text{ and } y_{mr}^{wc})\) respectively in WordCount as well as the CPU and memory demands \((y_{mc}^{tc} \text{ and } y_{mr}^{tc})\) respectively in Terasort. Then, four corresponding linear functions are presented:

\[
\begin{align*}
y_{mc}^{wc} &= 0.0393x + 0.549 \\
y_{mc}^{tc} &= 0.0425x + 0.028 \\
y_{mr}^{tc} &= 0.066x + 1.012 \\
y_{mr}^{wc} &= 0.135x + 0.72
\end{align*}
\]

After that, give a random value \(x\) uniformly distributed in the interval \([a, b]\) and then apply Eq. (9-12), various reducer types come out.

Next two definitions are given. If the maximal number of workers of type \(d\) simultaneously running for the VM of type \(k\) is determined by CPU demands, it is defined that the type \(d\) worker is \(CPU\ bound\) for the type \(k\) VM; reversely, it is \(memory\ bound\) for the type \(k\) VM. According to the definitions, the mappers of all types are CPU bound for all VMs. But it is more complicated for the reducers. For example, in WordCount, the reducers are CPU bound for any VM with the input less than 4 GB, but memory bound for compute optimized VMs with the input more than 5.4 GB. Therefore, given the different input size, there will be different combinations of CPU and memory bound workers.

To cover all possible combinations of CPU and memory bound workers, the test instances are categorized into eight classes:

- **Class 1:** only WorkCount applications are involved; the CPU and memory demands of mapper-like worker types are uniformly distributed in \([1.7, 1.9]\) and \([0.3, 0.4]\) respectively, being CPU bound for all VMs, and \(D/2\) mapper-like worker types are generated; the input size uniformly distributed in \([0.1, 4]\), indicating the reducer-like workers are CPU bound for all VMs, and \(D/2\) reducer-like worker types are generated.
- **Class 2:** like Class 1, except that the input size is uniformly distributed in \([5.4, 156]\), indicating the reducer-like workers are CPU bound for general purpose and memory optimized VMs, but memory bound for compute optimized VMs.
- **Class 3:** only Terasort applications are involved; the CPU and memory demands of mapper-like worker types are uniformly distributed in \([1.5, 1.8]\) and \([0.1, 0.2]\) respectively, being CPU bound for all VMs, and \(D/2\) mapper-like worker types are generated; the input size uniformly distributed in \([0.1, 83]\), indicating the reducer-like workers are CPU bound for general purpose and memory optimized VMs, but memory bound for compute optimized VMs.
- **Class 4:** like Class 3, except that the input size is uniformly distributed in \([83, 156]\), indicating the reducer-like workers are CPU bound for general purpose and memory optimized VMs, but memory bound for compute optimized VMs.
- **Class 5:** a combination of Classes 1 and 3, and the numbers of the mapper-like and reducer like worker types in WordCount and Terasort are all equal to \(D/4\).
- **Class 6:** like Class 5, but combining Classes 1, 4.
- **Class 7:** like Class 5, but combining Classes 2, 3.
- **Class 8:** like Class 5, but combining Classes 2, 4.

For each instance class, the total amount \(D\) of worker types involved is in the set \(\{16, 32, 48\}\), particularly no workers on the same type have the same resource requirements. In

### Table I

<table>
<thead>
<tr>
<th>VM Type</th>
<th>CPUs (#)</th>
<th>Memory (GB)</th>
<th>Price ($/hour)</th>
</tr>
</thead>
<tbody>
<tr>
<td>m1 small</td>
<td>1</td>
<td>1.2</td>
<td>0.06</td>
</tr>
<tr>
<td>m1 medium</td>
<td>2</td>
<td>3.75</td>
<td>0.12</td>
</tr>
<tr>
<td>m1 large</td>
<td>4</td>
<td>7.5</td>
<td>0.24</td>
</tr>
<tr>
<td>m1 xlarge</td>
<td>8</td>
<td>14.7</td>
<td>0.48</td>
</tr>
<tr>
<td>m2 xlarge</td>
<td>6.5</td>
<td>17.1</td>
<td>0.41</td>
</tr>
<tr>
<td>m2 xlarge</td>
<td>13</td>
<td>34.2</td>
<td>0.82</td>
</tr>
<tr>
<td>c1 medium</td>
<td>5</td>
<td>1.7</td>
<td>0.145</td>
</tr>
<tr>
<td>c1 xlarge</td>
<td>20</td>
<td>7</td>
<td>0.58</td>
</tr>
</tbody>
</table>

### Table II

<table>
<thead>
<tr>
<th>Input (GB)</th>
<th>Mapper</th>
<th>Reducer</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CPUs (#)</td>
<td>Mem (GB)</td>
</tr>
<tr>
<td>Word Count</td>
<td>4</td>
<td>[1.7, 1.9]</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>[1.7, 1.9]</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>[1.7, 1.9]</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>[1.7, 1.9]</td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>[1.7, 1.9]</td>
</tr>
<tr>
<td>Terasort</td>
<td>2</td>
<td>[1.5, 1.8]</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>[1.5, 1.8]</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>[1.5, 1.8]</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>[1.5, 1.8]</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>[1.5, 1.8]</td>
</tr>
</tbody>
</table>
addition, the amount $q$ of workers on each type is in the set \{10, 30, 50\}.

### B. Comparison of Different Algorithms for MRPP

We respectively adopt IFFD [5] which adopts several ways to calculate the surrogate weights in [6], SCH [8] and our heuristic algorithm to solve the test instances in eight classes. Let $TPH$ denotes our heuristic as it is a Two-Phase-Heuristic. We respectively run the IFFD with different methods to calculate 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 combination problem involved in TPH 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.

Since both SCH and TPH adopt the random FFD, the result output by SCH or TPH each time is probably different. Thus we repeat solving the same test instance 20 times by using SCH and TPH respectively and then calculate the best and average results. Meanwhile, we use IFFD to solve each test instance just once since IFFD is not stochastic. With regard to the maximal time of solving the optimization problem in the placement pattern optimization phase, we follow the suggestion in [8], setting it to 30 seconds.

Tables III and IV summarize the results of all the algorithms on the test instances. It is observed that the standard deviations of the results are very small. In Tables III and IV, the columns $D$ and $q$ respectively denote the total number of worker types and the number of workers on each type in a single test instance. Best and Avg respectively denote the lowest and average prices for the VMs rented to load the workers, respectively denoting the best and average solution values found by the algorithms. The results show that, in terms of average solution values, TPH always outperforms IFFD and SCH. In detail, in Classes 2, 3, 5, 7 and 8, as the number of workers or worker types increases, the gap TPH outperforms either IFFD or SCH increases; in Classes 1, 4 and 6, as the number of workers or worker types increases, the gap TPH outperforms either IFFD or SCH changes slightly. Similarly, on the term of best solution values, TPH also outperforms IFFD and SCH. Therefore, TPH always yields better solution than IFFD or SCH as the problem size varies.

Fig. 2 presents the variations on the average computation time of the algorithms as the number of workers on each type, $q$, increases, when the number of worker types $D = 48$, and the test instances are in Class 1. Obviously, IFFD runs fastest. Although TPH is not the fastest one, the computation time of TPH increase slightly as $q$ increases. On the contrary, the computation time of SCH increases much dramatically as $q$ increases.

Fig. 3 presents the variations on the average computation time of the algorithms as $D$ increases, when $q = 50$ and the test instances are in Class 1. As seen from Fig. 3, the trends in the computation time variations are similar to those in Fig.2. Considering that the trends in the computation time variations are similar under the other configuration of $q$, $D$ and the test instance class, we will not illustrate them.

---

**Table III**

The computational results of test instances in Classes 1-4

<table>
<thead>
<tr>
<th>Class</th>
<th>$D$</th>
<th>$q$</th>
<th>IFFD</th>
<th>SCH</th>
<th>TPH</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Best</td>
<td>Avg</td>
<td>Best</td>
</tr>
<tr>
<td>1</td>
<td>16</td>
<td>10</td>
<td>5.8</td>
<td>5.8</td>
<td>5.66</td>
</tr>
<tr>
<td></td>
<td>30</td>
<td>17.4</td>
<td>16.82</td>
<td>17.16</td>
<td>16.8</td>
</tr>
<tr>
<td></td>
<td>50</td>
<td>29.58</td>
<td>28.34</td>
<td>30.02</td>
<td>27.96</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>11.6</td>
<td>11.35</td>
<td>11.54</td>
<td>11.31</td>
</tr>
<tr>
<td></td>
<td>30</td>
<td>34.7</td>
<td>36.16</td>
<td>38.26</td>
<td>33.64</td>
</tr>
<tr>
<td></td>
<td>50</td>
<td>57.83</td>
<td>60.88</td>
<td>64.44</td>
<td>56.12</td>
</tr>
<tr>
<td>48</td>
<td>10</td>
<td>17.4</td>
<td>17.11</td>
<td>17.28</td>
<td>17.08</td>
</tr>
<tr>
<td></td>
<td>30</td>
<td>51.62</td>
<td>54.38</td>
<td>56.97</td>
<td>50.72</td>
</tr>
<tr>
<td></td>
<td>50</td>
<td>86.42</td>
<td>94.66</td>
<td>97.62</td>
<td>84.82</td>
</tr>
<tr>
<td>2</td>
<td>16</td>
<td>10</td>
<td>31.02</td>
<td>20.56</td>
<td>21.47</td>
</tr>
<tr>
<td></td>
<td>30</td>
<td>88.19</td>
<td>63.69</td>
<td>66.97</td>
<td>61.2</td>
</tr>
<tr>
<td></td>
<td>50</td>
<td>155.6</td>
<td>115.08</td>
<td>117.78</td>
<td>110.02</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>57.78</td>
<td>44.3</td>
<td>47.02</td>
<td>42.28</td>
</tr>
<tr>
<td></td>
<td>30</td>
<td>178.44</td>
<td>143.52</td>
<td>148.33</td>
<td>134.85</td>
</tr>
<tr>
<td></td>
<td>50</td>
<td>290.38</td>
<td>228.78</td>
<td>236.33</td>
<td>215.9</td>
</tr>
<tr>
<td>48</td>
<td>10</td>
<td>81.89</td>
<td>66</td>
<td>68.73</td>
<td>65.02</td>
</tr>
<tr>
<td></td>
<td>30</td>
<td>264.5</td>
<td>209.2</td>
<td>213.75</td>
<td>200.5</td>
</tr>
<tr>
<td></td>
<td>50</td>
<td>453</td>
<td>371.64</td>
<td>377.78</td>
<td>359.57</td>
</tr>
<tr>
<td>3</td>
<td>16</td>
<td>10</td>
<td>32.46</td>
<td>23.76</td>
<td>24.7</td>
</tr>
<tr>
<td></td>
<td>30</td>
<td>86.3</td>
<td>61.74</td>
<td>65.86</td>
<td>59.34</td>
</tr>
<tr>
<td></td>
<td>50</td>
<td>154.26</td>
<td>109.71</td>
<td>112.25</td>
<td>104.36</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>56.05</td>
<td>42.54</td>
<td>44.78</td>
<td>41.8</td>
</tr>
<tr>
<td></td>
<td>30</td>
<td>172.68</td>
<td>139.07</td>
<td>142.1</td>
<td>133.1</td>
</tr>
<tr>
<td></td>
<td>50</td>
<td>305.06</td>
<td>224.83</td>
<td>231.32</td>
<td>211.8</td>
</tr>
<tr>
<td>48</td>
<td>10</td>
<td>86.62</td>
<td>65.1</td>
<td>66.66</td>
<td>63.99</td>
</tr>
<tr>
<td></td>
<td>30</td>
<td>236.56</td>
<td>197.28</td>
<td>199.14</td>
<td>172.92</td>
</tr>
<tr>
<td></td>
<td>50</td>
<td>457.16</td>
<td>340.18</td>
<td>344.46</td>
<td>320.83</td>
</tr>
<tr>
<td>4</td>
<td>16</td>
<td>10</td>
<td>42.58</td>
<td>32.32</td>
<td>32.42</td>
</tr>
<tr>
<td></td>
<td>30</td>
<td>127.79</td>
<td>126.48</td>
<td>127.09</td>
<td>126.36</td>
</tr>
<tr>
<td></td>
<td>50</td>
<td>212.3</td>
<td>211.4</td>
<td>217.88</td>
<td>210.85</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>85.04</td>
<td>84.05</td>
<td>84.69</td>
<td>83.93</td>
</tr>
<tr>
<td></td>
<td>30</td>
<td>254.9</td>
<td>249.9</td>
<td>253.38</td>
<td>249.11</td>
</tr>
<tr>
<td></td>
<td>50</td>
<td>424.6</td>
<td>420.03</td>
<td>424.02</td>
<td>418.12</td>
</tr>
<tr>
<td>48</td>
<td>10</td>
<td>127.38</td>
<td>126.51</td>
<td>127.03</td>
<td>126.36</td>
</tr>
<tr>
<td></td>
<td>30</td>
<td>382.14</td>
<td>379.56</td>
<td>381.38</td>
<td>378.01</td>
</tr>
<tr>
<td></td>
<td>50</td>
<td>636.32</td>
<td>633.46</td>
<td>636.47</td>
<td>629.46</td>
</tr>
</tbody>
</table>

---

**Figure 2.** The variations on the average computation time of the algorithms as the number of workers on each type increases.
ment solution provided by our heuristic algorithm and the conventional solutions adopted in [1] [2] [3] without considering MRPP. Using each mapper/reducer placement solution, we repeat running the job 20 times, since each time the execution time of the job is variable in a real system even under the same configurations. Then we compare the costs of the VMs used to run the job and the success rate which is ratio of the times of completing the job within a deadline to 20. The job input is a 8 GB text file generated by `RandomTextWriter` and the deadline of the job is 400 seconds.

The job is executed by 24 workers including 12 homogeneous mappers and 12 homogeneous reducers, which are placed on three types of VMs, m1 medium, m1 large and c1 medium. It is recorded that the CPU demands of all mappers and reducers are 1.7 CPUs and 0.82 CPUs respectively while the memory demands of all mappers and reducers are 0.3 GB and 0.4 GB respectively. Recall that the amounts of resource demands are equal to the upper bounds of the resource consumption from the past job running. The number of reduce tasks in this job is 12. Furthermore, in our test platform, network and disk I/O are not bottlenecks, thus the influence of I/O on the job runtime can be greatly weakened.

The mapper/reducer placement solutions to be compared are shown as follows. In detail, the placement solutions S_1 - S_6 are adopted in the works [1] [2] while the placement solution Sch-Mix is adopted in the work [3].

- S_H: a solution provided by our heuristic algorithm.
- S_1: 12 c1 medium VMs are used, each VM loads one mapper and one reducer.
- S_2: 12 m1 medium VMs are used, each VM loads one mapper and one reducer.
- S_3: 12 m1 large VMs are used, each VM loads two mappers and two reducers.
- S_4: 6 c1 medium VMs are used, each VM loads two mappers and two reducers.
- S_5: 6 m1 large VMs are used, each VM loads two mappers and two reducers.
- S_6: 4 c1 medium VMs are used, each VM loads three mappers and three reducers.
- S_Mix: 4 m1 large VMs and 4 c1 medium VMs are used, each m1 large VM loads two reducers while each c1 medium VM loads three mappers and one reducer.

Fig. 4 presents the costs of VMs to be used and the success rate of completing the job within the deadline (400 seconds) under different mapper/reducer placement solutions. Note that the runtime of the heuristic algorithm has not been considered in this experiment, because compared with the magnitude of the deadline which is in hundreds of seconds, it is in the magnitude of hundreds of milliseconds, which can be ignored. As shown in Fig. 4, using S_H, S_1 and S_3, the job is completed before the deadline with the

<table>
<thead>
<tr>
<th>Class</th>
<th>D</th>
<th>q</th>
<th>IFFD</th>
<th>SCH</th>
<th>TPH</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>16</td>
<td>10</td>
<td>18.1</td>
<td>13.26</td>
<td>13.67</td>
</tr>
<tr>
<td>30</td>
<td>63.7</td>
<td>43.5</td>
<td>44.85</td>
<td>32.4</td>
<td>43.46</td>
</tr>
<tr>
<td>50</td>
<td>91.22</td>
<td>70.25</td>
<td>71.14</td>
<td>65.72</td>
<td>67.76</td>
</tr>
<tr>
<td>32</td>
<td>10</td>
<td>38.97</td>
<td>28.56</td>
<td>29.83</td>
<td>27.78</td>
</tr>
<tr>
<td>30</td>
<td>180.64</td>
<td>89.26</td>
<td>91.09</td>
<td>82.33</td>
<td>85.65</td>
</tr>
<tr>
<td>50</td>
<td>190.64</td>
<td>141.69</td>
<td>142.44</td>
<td>128.54</td>
<td>130.36</td>
</tr>
<tr>
<td>48</td>
<td>10</td>
<td>55.32</td>
<td>41.5</td>
<td>41.62</td>
<td>39.72</td>
</tr>
<tr>
<td>30</td>
<td>166.28</td>
<td>130.48</td>
<td>131.9</td>
<td>1.38</td>
<td>121.8</td>
</tr>
<tr>
<td>50</td>
<td>278.17</td>
<td>217.2</td>
<td>218.35</td>
<td>200.94</td>
<td>201.45</td>
</tr>
<tr>
<td>6</td>
<td>16</td>
<td>10</td>
<td>24.25</td>
<td>23.02</td>
<td>23.5</td>
</tr>
<tr>
<td>30</td>
<td>72.58</td>
<td>70.04</td>
<td>70.4</td>
<td>69.64</td>
<td>70.08</td>
</tr>
<tr>
<td>50</td>
<td>120.36</td>
<td>114.67</td>
<td>117.63</td>
<td>113.96</td>
<td>116.87</td>
</tr>
<tr>
<td>32</td>
<td>10</td>
<td>48.26</td>
<td>46.04</td>
<td>46.92</td>
<td>45.77</td>
</tr>
<tr>
<td>30</td>
<td>144.78</td>
<td>139.82</td>
<td>140.46</td>
<td>138.84</td>
<td>139.65</td>
</tr>
<tr>
<td>50</td>
<td>240.72</td>
<td>233.5</td>
<td>237</td>
<td>232.02</td>
<td>233.67</td>
</tr>
<tr>
<td>48</td>
<td>10</td>
<td>72.24</td>
<td>69.78</td>
<td>70.19</td>
<td>69.61</td>
</tr>
<tr>
<td>30</td>
<td>216.36</td>
<td>209.8</td>
<td>213.58</td>
<td>208.47</td>
<td>210.36</td>
</tr>
<tr>
<td>50</td>
<td>360.97</td>
<td>350.27</td>
<td>354.4</td>
<td>345.26</td>
<td>348.74</td>
</tr>
</tbody>
</table>

Table IV

The computational results of test instances in Classes 5-8

C. Comparison of Different Mapper/Reducer Placement Solutions for a Real-World Problem

In this experiment, we respectively run the same CPU-intensive job, WordCount, using the mapper/reducer place-

![Figure 3](image-url)

Figure 3. The variations on the average computation time of the algorithms as the number of worker types increases

![Figure 4](image-url)

Figure 4. The variations on the average computation time of the algorithms as the number of worker types increases

![Figure 5](image-url)

Figure 4. The variations on the average computation time of the algorithms as the number of worker types increases

- S_H: a solution provided by our heuristic algorithm.
- S_1: 12 c1 medium VMs are used, each VM loads one mapper and one reducer.
- S_2: 12 m1 medium VMs are used, each VM loads one mapper and one reducer.
- S_3: 12 m1 large VMs are used, each VM loads two mappers and two reducers.
- S_4: 6 c1 medium VMs are used, each VM loads two mappers and two reducers.
- S_5: 6 m1 large VMs are used, each VM loads two mappers and two reducers.
- S_6: 4 c1 medium VMs are used, each VM loads three mappers and three reducers.
- S_Mix: 4 m1 large VMs and 4 c1 medium VMs are used, each m1 large VM loads two reducers while each c1 medium VM loads three mappers and one reducer.

Fig. 4 presents the costs of VMs to be used and the success rate of completing the job within the deadline (400 seconds) under different mapper/reducer placement solutions. Note that the runtime of the heuristic algorithm has not been considered in this experiment, because compared with the magnitude of the deadline which is in hundreds of seconds, it is in the magnitude of hundreds of milliseconds, which can be ignored. As shown in Fig. 4, using S_H, S_1 and S_3, the job is completed before the deadline with the

![Figure 5](image-url)

Figure 5. The variations on the average computation time of the algorithms as the number of worker types increases

- S_H: a solution provided by our heuristic algorithm.
- S_1: 12 c1 medium VMs are used, each VM loads one mapper and one reducer.
- S_2: 12 m1 medium VMs are used, each VM loads one mapper and one reducer.
- S_3: 12 m1 large VMs are used, each VM loads two mappers and two reducers.
- S_4: 6 c1 medium VMs are used, each VM loads two mappers and two reducers.
- S_5: 6 m1 large VMs are used, each VM loads two mappers and two reducers.
- S_6: 4 c1 medium VMs are used, each VM loads three mappers and three reducers.
- S_Mix: 4 m1 large VMs and 4 c1 medium VMs are used, each m1 large VM loads two reducers while each c1 medium VM loads three mappers and one reducer.

Fig. 4 presents the costs of VMs to be used and the success rate of completing the job within the deadline (400 seconds) under different mapper/reducer placement solutions. Note that the runtime of the heuristic algorithm has not been considered in this experiment, because compared with the magnitude of the deadline which is in hundreds of seconds, it is in the magnitude of hundreds of milliseconds, which can be ignored. As shown in Fig. 4, using S_H, S_1 and S_3, the job is completed before the deadline with the

![Figure 5](image-url)

Figure 5. The variations on the average computation time of the algorithms as the number of worker types increases

- S_H: a solution provided by our heuristic algorithm.
- S_1: 12 c1 medium VMs are used, each VM loads one mapper and one reducer.
- S_2: 12 m1 medium VMs are used, each VM loads one mapper and one reducer.
- S_3: 12 m1 large VMs are used, each VM loads two mappers and two reducers.
- S_4: 6 c1 medium VMs are used, each VM loads two mappers and two reducers.
- S_5: 6 m1 large VMs are used, each VM loads two mappers and two reducers.
- S_6: 4 c1 medium VMs are used, each VM loads three mappers and three reducers.
- S_Mix: 4 m1 large VMs and 4 c1 medium VMs are used, each m1 large VM loads two reducers while each c1 medium VM loads three mappers and one reducer.
probability of 100%, but using $S_{1}$ and $S_{3}$, the resources of VMs are not utilized so well as using $S_{H}$, thus using $S_{H}$ the costs of VMs are much cheaper. Meanwhile, using $S_{4}$ and $S_{6}$, although the costs of VMs are cheaper, but the VMs cannot provide enough resources to the workers and resource competitions are raised, thus the execution time is prolonged. Consequently, neither of them ensure the job is completed before the deadline.

Consequently, we find that, using different worker placements, the success rate and costs of job execution could be quite different. Furthermore, we observe that our heuristic algorithm is much effective, since using the placement solution found by it, the cost for running the job is much lower than that using the conventional placement solutions without considering MRPP while the job is completed within the deadline.

VI. CONCLUSION AND FUTURE WORK

This paper has studied a new problem called MRPP, whose objective is to place all the workers including mappers and reducers for MapReduce execution on the VMs in cloud computing, such that the costs of the VMs are minimized while the resource demands of workers are met. Furthermore, a new heuristic algorithm has been proposed to solve MRPP, which is a generalization of the BPP which has been proven to be NP-complete.

We have compared our heuristic algorithm with IFFD and SCH by solving the test problems from real-world and the computational results have shown that our heuristic algorithm is much more efficient than the other two algorithms on solving MRPP. Our heuristic algorithm could find a better solution than the other two algorithms in a reasonable time. In addition, we have verified the effectiveness of our heuristic algorithm by comparing the mapper/reducer placement solution found by our heuristic algorithm with the conventional placement solutions adopted in most current works. The results have indicated that using the placement solution obtained by our heuristic algorithm the cost of the MapReduce computation is much cheaper than that using the conventional placement solutions while still satisfying the computation deadline.

In the future, dynamic MapReduce placement problem will be studied. Also, the design of test problems will be enhanced, which considers the problem semantics [11].

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


