Parallel Algorithm for Combinatorial Optimization Problem

Narameth Nananukul
Sirindhorn International Institute of Technology, Thammasat University, Pathum Thani, Thailand, 12121.
narameth@siit.tu.ac.th

***Abstract*— Combinatorial optimization problems (COP) are difficult to solve by nature. One of the reasons is because the amount of neighborhood search required to generate high quality solutions based on sequential methods is intractable. In this paper, parallel algorithm for COP such as Knapsack Problem is presented. Knapsack problem arises in different types of resource allocation problems and has many applications in real-world problems. The proposed algorithm is based on MapReduce framework where the workload for neighborhood search is distributed across available computing nodes in the cluster. The design of Map and Reduce phases is proposed based on consecutive runs of MapReduce jobs. The computational results that shows the effect of degree of parallelism on the solution quality are provided.**

***Index Terms*— MapReduce; Combinatorial Optimization Problem; Knapsack Problem; Metaheuristics.**

I. INTRODUCTION

Knapsack Problem (KP) often arises in different types of resource allocation problems and has many applications in real-world problems. In general, KP is classified as NP-hard problem. Although it is possible to use a very large and powerful machine to solve the problem, the approach tends to be impractical due to the relevant costs involved in acquiring hardware. Thus, it is much more economical and practical to use a cluster of computers system that charges the service based on the amount of usage such as cloud system. MapReduce (MR) framework can be used to help implementing parallel algorithms. MR is based on a simple programming model and is commonly implemented on top of Hadoop [1], an open source MapReduce framework. Not only it can assist in distributing workload to different nodes in the cluster, it also provides necessary services such as load balancing, fault tolerance, etc.

MR was first introduced by Google. It was originally designed to process larger datasets that normally are located at different locations. In this paper, the MR based algorithms are proposed for COP such as KP. The computational results that shows the effect of degree of parallelism of the proposed algorithm on the solution quality are provided. The paper is organized as follows. Section II gives details of KP and MR framework. The design of parallel algorithms based on MR framework is presented in Section III. The computational results are presented and summarized in Section IV. Section V provides conclusion and future research directions.

II. BACKGROUND

In this section, basic information of Knapsack Problem and MR framework are provided. First, the general description of KP is provided, then the framework of MR is presented.

A. Knapsack Problem

KP has many applications in different fields and typically arises in resource allocation problem. For example, it can be used to determine the set of data files chosen to store with a given available bytes of storage. The formal description of KP can be described as follows:

Given two sets of numbers, $\{v_1, v_2,\ldots, v_n\}$(values) and $\{w_1, w_2,\ldots, w_n\}$ (weights) and W (capacity) > 0, the objective is to determine the subset S of $\{1,2,\ldots,n\}$(set N) such that $\sum_{i\in S} v_i$ is maximized, subjected to $\sum_{i\in S} w_i \le W$. In general, the KP is considered NP-hard and there is no known polynomial algorithm which can solve the problem. There exists many algorithms for KP problem, branch and bound, heuristics and dynamic programming approach but most of them are intractable as the problem size increases. The dynamic programming algorithm for KP [1] is shown in Figure 1. In the algorithm, an array $V[i,w]$ is used to store the value of each combination of i and w. The size of the array is nW which increases significantly as the parameter W or the capacity of the knapsack increases. As a result, the algorithm becomes intractable.

1. An array $V[0..n, 0..W]$ for $1 \le i \le n$ and $0 \le w \le W$ is constructed and set to 0 initially.
2. Recursively, update V[i,w]

```
For (i = 1 to n)
  For (w = 0 to W)
  If (w[i] ≤ W)
    V[i,w] = Max{V[i − 1,w], v[i] + V[i − 1,w − w[i]]}
  Else
    V[i,w] = V[i − 1,w]
Return V[n,W]
```

Figure 1: Dynamic programming algorithm for KP

In this paper, a parallel heuristic algorithm (PHKP) for KP is proposed to demonstrate how it can take advantage of parallel and distributed functions from MR framework.

B. MapReduce Framework

In general, MR framework requires input data in the form of a list of records in the form of <key, value> pairs. The data can be stored by using different distributing file systems; i.e, Google's MR uses GFS while Hadoop uses HDFS [1]. In Hadoop, one of the nodes is defined as the Master while others are defined as slave nodes. The Master node distributes MR jobs to slave nodes based on the predefined MAP and REDUCE functions. Based on an input data in the form of <key, value> pairs, the MAP function will generate a set of intermediate records, also in the form of <key2, value2>. Intermediate records having the same key2 are grouped

together and processed by a REDUCE function which will generate a number of output records. Hadoop manages the scheduling of intermediate records to available slave nodes while considering overall load balancing.

A simple MR program consists only a MAP and a REDUCE functions. In a complex MR program, it is possible to have more than one MR job run in sequence where the output of a MR job becomes the input of the next MR job. The MR framework is shown in Figure 2.

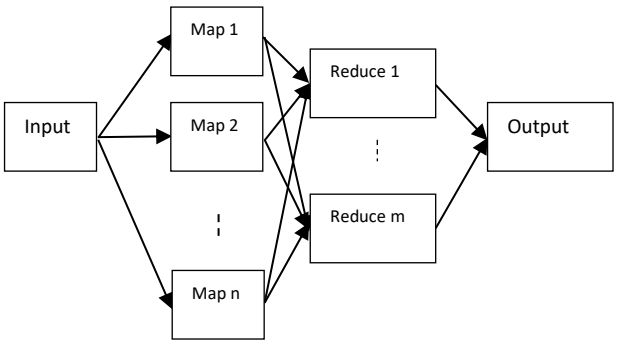

Figure 2: MapReduce framework

III. MAPREDUCE ALGORITHM FOR KP

Although there has been much work in developing parallelizing heuristics for COP such as traveling salesman problem (TSP) [3][4][5], none of them has taken advantage of existing cluster computing architectures such as MR for solving the problem. The contribution of this paper is to develop an algorithm based on neighborhood search [6] and demonstrate how it can be implemented on MR framework.

A. Solution Representation

The set of candidate solutions for a KP with i items is represented by a set Si. Each solution s in Si needs to satisfy the constraint imposed by the KP, the capacity of the knapsack. The knapsack value for any solution Si is defined as $\sum_{s \in S_i} v_s$.

B. Initial Solution

The initial solution is based on a greedy heuristic algorithm where the items are selected based on the ratios of value and weight (vw ratio) in decreasing order. The algorithm is summarized in Figure 3.

1. The ratio of profit and weight (pw ratio) for each item is calculated and ordered in decreasing order.

$$\frac{v_{i_1}}{w_{i_1}} \leq \frac{v_{i_2}}{w_{i_2}} \leq \cdots \leq \frac{v_{i_n}}{w_{i_n}}$$

2. The items are included to the solution based on the order set by step 1 until the capacity constraint is violated.

Figure 3: Greedy heuristic algorithm

C. Neighborhood Search

All solutions that can be reached from a current solution (incumbent solution) by using one or more moves represent a neighborhood [7]. In general, the moves for COP take the form of insertions, exchanges or replacements. For the PHKP, the neighborhood is defined as all feasible points that can be reached by two types of moves. The first is called a swap and involves an exchange of assignment of two items $item_1$ and $item_2 \in \{1,\ldots,n\}$ when either ($item_1 \in S_i$ and $item_2 \notin S_i$) or ($item_1 \notin S_i$ and $item_2 \in S_i$) is true.

The swap moves trade an item with high vw ratio with one or more items with lower vw ratios. The second move is called an insert and selects an item not already included in the solution to be inserted to the solution.

Example of moves. Figure 4 depicts a swap between items 3 and 4. The items are represented by the circles where the numbers correspond to items' ids. In the example, there are five items considered, the parameters for the items are as follows. The values for items 1 to 5 are $v_1 = 50$, $v_2 = 40$, $v_3 = 15$, $v_4 = 80$ and $v_5 = 20$. The weights for items 1 to 5 are $w_1 = 2$, $w_2 = 9$, $w_3 = 3$, $w_4 = 8$ and $w_5 = 5$. The capacity is limited to 20. Before the swap, items 1, 3 and 5 are in the solution. After the swap, the assignment of items 3 and 4 are exchanged; item 3 is removed from the solution while item 4 is included to the solution. The capacity of the solution increases from 10 to 15 and the value of the solution increases from 85 to 150.

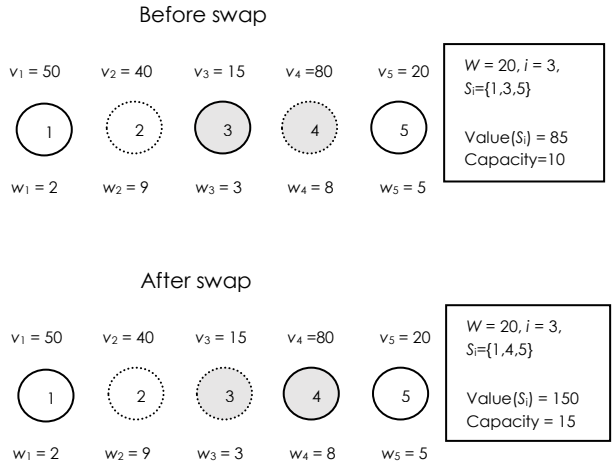

Figure 4: A swap between items 3 and 4

Using the same data and starting with the solution in the bottom portion of Figure 4. Figure 5 gives an example of an insert move. Before the insert, items 1, 4 and 5 are in the solution. The move inserts item 3 to the solution. The capacity of the solution increases from 15 to 18 and the value of the solution increases from 150 to 165.

D. MapReduce Jobs

The proposed MR for KP consists of sequential MR jobs where the solution is adjusted based on neighborhood search in each MR job. Once the job is completed, the improved solutions are retrieved and used as the input for the next iteration. The algorithm terminates when no more improved solution can be found. In each iteration of the algorithm, improved solutions can be found by using a neighborhood search based on the previously generated solutions at each node. The pseudo code for the main program of the algorithm is shown in Figure 6.

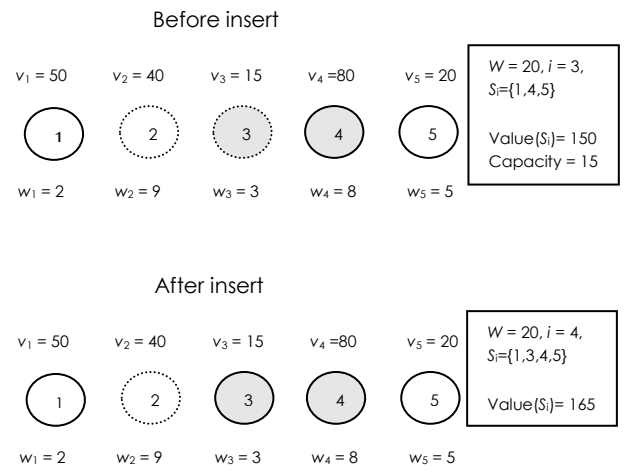

Figure 5: An insert of item 3

1. Iteration = 0
2. Do
   a. Create a new MR job
   b. Set the MR's MAP and REDUCE classes
   c. Set input and output paths
   d. Submit the job and wait until the job terminates
   e. Retrieve improved solutions
   f. Iteration = Iteration + 1

While (number of improved solutions > 0)

Figure 6: The pseudo code for MR main program

Proposition 1

If the current solution is not optimal, when the initial solution is generated by using a greedy heuristic algorithm, the solution can be improved by either applying a swap move or applying a swap and an insert.

Proof

Consider a knapsack problem with n items. Let S_i be the solution generated by following the greedy heuristic algorithm and the list of items included in the solution in increasing order of pw ratios is $item_1$, $item_2$,...,$item_i$. Without loss of generality, a single pair of items, $item_m$ and $item_n$ ($m < n < i$), is considered in the swap.

Case1: $item_m \notin S_i$ and $item_n \in S_i$

Let S_i^{new} be the solution after swapping $item_m$ and $item_n$. The capacity and the value of S_i^{new1} are Capacity(S_i^{new1}) = Capacity(S_i) - w_{item_n} + w_{item_m} and Value(S_i^{new1}) = Value(S_i) - v_{item_n} + v_{item_m}, respectively. Based on the assumption, $\frac{v_{item_m}}{w_{item_m}} \leq \frac{v_{item_n}}{w_{item_n}}$, there are 2 possible cases.

Case 1.1:

Capacity (S_i^{new1}) $\leq W$ and Value (S_i^{new1}) > Value (S_i)

In this case, the solution is improved by applying only a swap.

Case 1.2:

The solutions that follow case1.1 are ignored in this case. Consider applying an insert after a swap. By contradiction, there must exist $item_o$, o < n, such that Capacity (S_i^{new2}) = Capacity (S_i^{new1}) + $w_{item_o} \leq W$ and Value(S_i^{new2}) = Value(S_i^{new1}) + v_{item_o}, otherwise S_i^{new1} is optimal.

Case 2: $item_m \in S_i$ and $item_n \notin S_i$

If $\exists$ $item_n$ such that $\frac{v_{item_m}}{w_{item_m}} \leq \frac{v_{item_n}}{w_{item_n}}$ and Capacity (S_i^{new3}) = Capacity(S_i) - w_{item_m} + $w_{item_n} \leq W$, then by contradiction, S_i was not generated by greedy heuristic algorithm.

Based on Proposition 1, in each MR job, a swap or an insert is applied to the solutions alternately in the MAP function. The pseudo code of the MAP function is listed in Figure 7.

Function *MAP*(*key*, (l, N_l, (Sol_1,..,Sol_{N_l}))
1. Initialize *HashMap* = ∅
2. For each $j \in N_l$ do
   If(l is odd)
      $Sol_j^{new} = SWAP(Sol_j)$
   Else
      $Sol_j^{new} = INSERT(Sol_j)$
   if($HASH\_VALUE(Sol_j^{new}) \notin HashMap$)
      $HashMap = HashMap \cup HASH\_VALUE(Sol_j^{new})$
   $seed = CAL\_SEED(Sol_j^{new})$
   EMIT($seed$, Sol_j^{new})

: l is iteration id
: N_l is the number of solutions at iteration l
: Sol_1,..,Sol_{N_l} is the list of solutions at iteration l
: *HashMap* is a hash map that stores solutions based on hash value

Figure 7: The pseudo code for MR main program

The input of MAP function consists of iteration id (l), number of solutions (N_l) and the list of solutions for iteration l (Sol_1,..,Sol_{N_l}). The algorithm iterates through each solution and applies a swap when l is odd and an insert when l is even. Examples of a swap and an insert are shown in Figures 4 and 5, respectively. Each generated solution is checked against the HashMap to make sure that no duplicate solution is stored in the HashMap. This helps reduce the number of solutions that needs to be processed in the following iterations. At the end of *MAP* function, a seed is assigned to each solution and used as a *key* that will be passed to a reducer.

Example of grouping solutions based on seed:

In order to take advantage of parallel computing function from MR framework, the solutions are partitioned and assigned to different reducers based on the special key called "seed". For any solution S_i, the seed is defined as an item with the maximum ratio of profit and weight, item $j^* = argmax_{j \in S_i}\left(\frac{v_j}{w_j}\right)$. Figure 8 illustrates how the seeds are assigned to solutions. Note that whenever there are more than one solution set based on i. Each member of S_i can be referenced by using notation S_i^k,where $k \in \{1,..,|S_i|\}$. In the example, the number of items is 5 and i = 3.

It is possible to have multiple solutions assigned to a reducer. The solutions' values are compared with the incumbent value of the best known solution. Solutions with values worse than the incumbent solution' value are excluded from the next iteration, otherwise the incumbent solution is updated. Figure 9 lists the pseudo code of the REDUCE function.

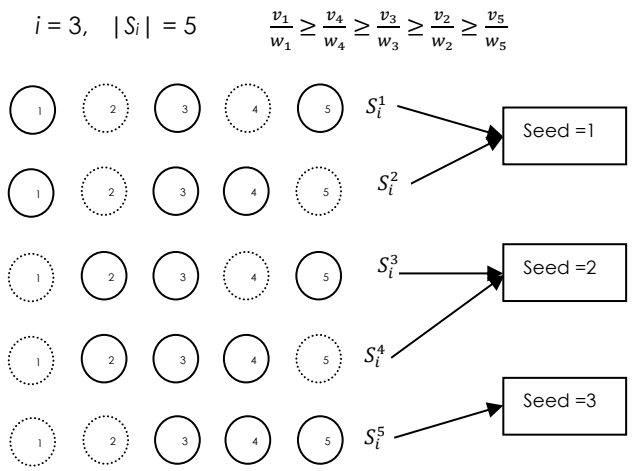

Figure 8: Example of seeds assignment

Function *REDUCE*(*seed*, (Sol_1,..,$Sol_{N_{seed}}$))
1. initialize val_{inc} and sol_{inc}
2. for each ($j \in N_{seed}$) do
   if($val(S_j) < val_{inc}$)
   update val_{inc} and sol_{inc}
   EMIT(seed, sol_{inc})

: *seed* is the *key* generated from *MAP* function
: N_{seed} is the number of solutions with the same seed
: Sol_1,..,$Sol_{N_{seed}}$ is the list of solutions with the same seed
: sol_{inc} is an incumbent solution
: val_{inc} is the value of the incumbent solution

Figure 9: The pseudo code of REDUCE function

The flow of MR jobs is summarized in Figure 10. In the first iteration the input is retrieved from an input file, the format of input file for MAP and REDUCE phases are described in Section III. In the following iterations the inputs are the outputs generated from all the reducers used in the previous iteration. Note that the solutions generated by all Maps are grouped and passed to Reducers based on the assigned seeds. The flow of MR jobs is terminated if no improved solution can be determined in the Reduce phase of the last iteration.

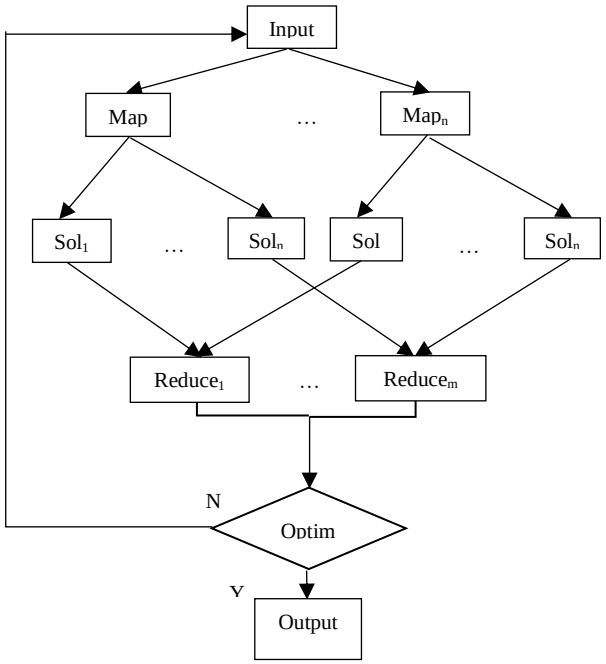

Figure 10: The pseudo code of REDUCE function

IV. Computational Results

The proposed algorithm was implemented on Amazon Elastic MapReduce (Amazon EMR) with various provided configurations as shown in Table 1. The Apache Hadoop Version 1.0.4 was chosen for compilation of MapReduce code.

Table 1
Number of default Mappers and Reducers from Amazon EMR

Amazon EC2 Instance Name	Mappers	Reducers
m1.small	2	1
m1.medium	2	1
m1.large	4	2
m1.xlarge	8	4
c1.medium	4	2
c1.xlarge	8	4
m2.xlarge	4	2
m2.2xlarge	8	4
m2.4xlarge	16	8

The data sets for the KP are from CMU artificial intelligence repository which were used as test cases in many research work [8][9][10]. The data sets contain instances with number of items ranging from 20 to 100 items. The data sets were solved with different Amazon EC2 instance to show the effect of degree of parallelism on the solutions. The results are summarized in Figure 11.

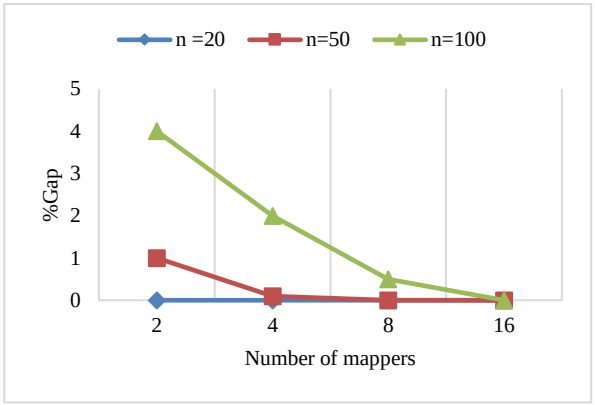

Figure 11: Effect of number of mappers on solution quality

Figure 11 shows how well the proposed algorithm scales. The percent gap was used as a measurement of solution quality. Three types of test cases (n = 20, 50, 100) were used in the experiment. For small test cases (20 items), the quality of solution did not depend on the number of mappers. For medium and large test cases (50 and 100 items), increasing the number of mappers improved the solutions and the gap became zero when number of mappers reached 16.

To determine the effect of number of mappers on the number of iterations of the algorithm (number of MR jobs) the %gap was set to 0.01% and the algorithm was executed until the required %gap was achieved. The results are shown in Figure 12. For all test cases, the number of required iterations decreases as the number of mappers increases. However, the rate of decrease for larger test cases is less than the rate of decrease for smaller test cases. This is because larger test cases require searching through larger neighborhood in order to achieve the required %gap.

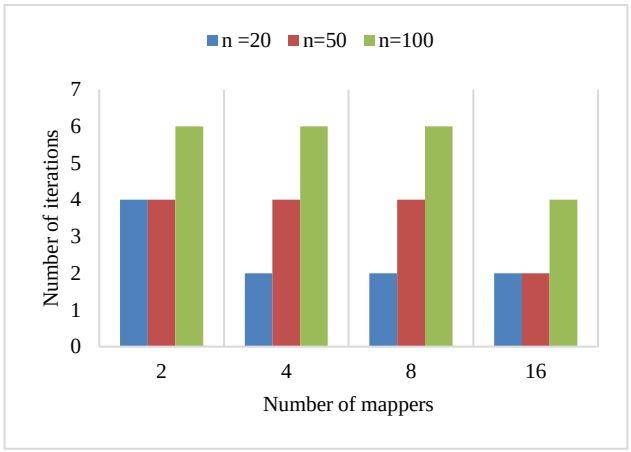

Figure 12: Effect of number of mappers on iterations

V. CONCLUSION

In this paper, a parallel heuristic algorithm for a KP is proposed. The description of MAP and REDUCE phases as well as the flow of MR jobs are provided. The efficiency of the algorithm was evaluated on Amazon EMR. Algorithm for COB such as knapsack problem can be developed on MR framework. The parallelization feature of the algorithm improved the overall efficiency of the algorithm, especially the solution quality (%gap). Similar concept can be applied to solve other COBs that are difficult to solve such as vehicle routing problem and travelling salesman problem.

REFERENCES

[1] Martello, S., Pisinger, P. and Toth, P. 1999. Dynamic programming and strong bounds for the 0-1 knapsack problem. *Manag. Sci*., 45:414-424.
[2] Lam, C. 2010. *Hadoop in Action*. Manning Publications Co.
[3] Fiechter, C.N. 1994. A parallel tabu search algorithm for large traveling salesman problems. *Discrete Applied Mathematics*, 51(3): 243-267.
[4] Cesari, G. 1996. Divide and conquer strategies for parallel TSP heuristics. *Computers and Operations Research*, 23(7): 681-694.
[5] Tsutsui, S. and Fujimoto, N. 2010. Parallel Ant Colony Optimization algorithm on a Multi-core Processor. *Lecture Notes in Computer Science*, 6234(2010): 488-495.
[6] Hoos, H. H. and T. Stutzle. 2005. *Stochastic Local Search: Foundations and Applications*. Morgan Kaufmann.
[7] Glover, F. and M. Laguna. 1997. Tabu search. Kluwer.
[8] Drexel, A. 1988. A Simulated Annealing Approach to the Multiconstraint Zero-One Knapsack Problem. *Computing*, 40:1-8.
[9] Freville, A. and Plateau, G. 1990. Hard 0-1 multiknapsack testproblems for size reduction methods. *Investigation Operativa*, 1:251-270.
[10] Shi, W. 1979. A branch and bound method for the multiconstraint zero one knapsack problem. *J. Opl. Res. Soc.*, 30:369-378.
[11] Lin, J. and C. Dyer. 2010. Data-Intensive Text Processing with MapReduce. Morgan and Claypool Publishers.
[12] Amazon Elastic MapReduce Developer Guide, *API Version 2009-03-31*.