

Parallel Algorithm of a Heuristic for the Multiple-Choice Multi-Dimension Knapsack Problem

by

Md. Waselul Haque Sadid

A Thesis Submitted to the Department of Computer Science and Engineering in
the Partial Fulfillment of the Requirement for the
Degree of

Master of Science in Engineering
(Computer Science and Engineering)



Department of Computer Science and Engineering
Bangladesh University of Engineering and Technology
Dhaka-1000, Bangladesh

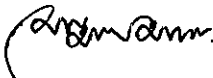



October 2005



#100941#

The thesis “**Parallel Algorithm of a Heuristic for the Multiple-Choice Multi-dimension Knapsack Problem**”, submitted by Md. Waselul Haque Sadid, Roll No. 100105042F, Registration No. 0110299, Session October 2001, to the Department of Computer Science and Engineering, Bangladesh University of Engineering and Technology, has been accepted as satisfactory for the partial fulfillment of the requirements for the degree of Master of Science and Engineering (Computer Science and Engineering) and approved as to its style and contents. The examination was held on October 2, 2005.

Board of Examiners

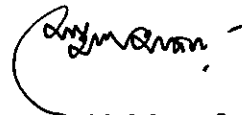
1. 
_____ **Dr. Md. Mostofa Akbar** Chairman
Assistant Professor (Supervisor)
Department of CSE
BUET, Dhaka-1000
2. 
_____ **Dr. Md. Shamsul Alam** Member
Professor & Head (Ex-officio)
Department of CSE
BUET, Dhaka-1000
3. 
_____ **Dr. Md. Abul Kashem Mia** Member
Professor
Department of CSE
BUET, Dhaka-1000
4. 
_____ **Dr. Md. Saifur Rahman** Member
Professor (External)
Department of EEE
BUET, Dhaka-1000

Certificate

This is to certify that this thesis work has been done by **Md. Waselul Haque Sadid**, Student No. 100105042, under the supervision of **Dr. Md. Mostofa Akbar**, Assistant Professor, Department of Computer Science and Engineering, Bangladesh University of Engineering and Technology, Dhaka-1000, Bangladesh. It is also declared that neither this thesis nor any part of it has been submitted or is being submitted to anywhere else for the award of any degree or diploma.



Md. Waselul Haque Sadid



Dr. Md. Mostofa Akbar
Supervisor of the Thesis

ACKNOWLEDGEMENT

First and foremost, I would like to thank my supervisor Dr. Md. Mostofa Akbar, Assistant Professor, Department of Computer Science and Engineering, Bangladesh University of Engineering and Technology. His constant supervision, constructive criticism, vast experience, invaluable advice and continual encouragement at all stages of my work have made it possible to complete this research.

I would like to express intense gratitude to Mr. Mohammad Abdul Hakim Newton, Assistant Professor (on leave), Department of Computer Science and Engineering for cordial cooperation and suggestions. I would also like to thank Mr. Abu Jafar Muhammd Shahriar, Assistant Professor, Department of Computer Science and Engineering, Ahsanullah University of Science and Technology for his good cooperation and suggestions.

I would like to acknowledge with sincere thanks to the Head of the Department and other faculty members, officers and staffs of the Department of Computer Science and Engineering for their friendly cooperation.

I would like to convey my sincere thanks to the Head of the Department and other faculty members of the Department of Computer Science and Engineering of Rajshahi University of Engineering and Technology for their valuable comments, suggestions and cordial cooperation.

Finally I would like to thank all of my friends and my family members for all their support and encouragement throughout my MSc. Engineering course.

Table of Contents

Acknowledgement	i
Table of Contents	ii
List of Figures	iv
List of Tables	vi
Abstract	1
CHAPTER ONE	2
INTRODUCTION.....	2
1.1 Definition of KP and MMKP.....	2
1.2 Applications of MMKP.....	4
1.3 Different Types of Algorithms for Solving MMKP	5
1.4 Motivation.....	5
1.5 Problem Definition.....	6
1.6 Scope and Focus	7
1.7 Outline.....	7
CHAPTER TWO	9
BACKGROUND AND PRELIMINARIES.....	9
2.1 Related Research on KP and Its Variants, MDKP and MCKP.....	9
2.1.1 Exact Solutions for KP, MDKP and MCKP.....	9
2.1.2 Approximate Solutions for KP, MDKP and MCKP.....	11
2.2 Related Research on MMKP	12
2.2.1 Algorithms for Exact Solutions to the MMKP	12
2.2.2 Algorithms for Heuristic Solutions to the MMKP.....	13
2.3 PRAM Model.....	17
2.3.1 Concurrent Versus Exclusive Memory Accesses	18
2.3.2 Optimality of PRAM Algorithms	19
2.3.3 Examples of PRAM Algorithms.....	19
2.4 Related Works on Parallel Algorithm for Knapsack Type Problems.....	25

CHAPTER THREE	29
PROPOSED ALGORITHMS FOR MMKP	29
3.1 Why M-HEU was Chosen for Parallellism.....	29
3.2 Problem of M-HEU to be Parallelized.....	29
3.3 A New Sequential Heuristic, MS-HEU	30
3.3.1 The Main Principle	30
3.3.2 The Process of Upgradation	30
3.3.3 Steps of MS-HEU	31
3.3.4 Description of MS-HEU for Upgrading n Items of an MMKP in an Arbitrary Iteration in Step 2	32
3.3.5 Description of MS-HEU	34
3.3.6 Some Arguments Regarding MS-HEU.....	37
3.3.7 Example for Demonstrating Strategy 1 of MS-HEU	38
3.4 PRAM-HEU: A Heuristic for Solving the MMKP Using PRAM Model ..	41
3.4.1 Description of the Algorithm PRAM-HEU	43
3.4.2 Complexity Analysis of Algorithm PRAM-HEU.....	47
CHAPTER FOUR	49
EXPERIMENTAL RESULTS	49
4.1 Initializing the Data.....	49
4.2 Methods of Experiment.....	50
4.3 Tets Results.....	50
4.4 Observations	60
CHAPTER FIVE	63
CONCLUSIONS	63
5.1 Major Contributions.....	63
5.2 Future Research Work	64
Appendix	66
References	87

List of Figures

1.1	The classical 0-1 knapsack problem	3
1.2	Example of an MMKP	4
2.1	Flow chart of M-HEU	16
2.2	The shared memory model	18
2.3	Summation of n elements	20
2.4	An example of prefix computation.....	21
2.5	Doubly logarithmic depth tree of 16 nodes.....	22
2.6	A binary tree T for pipelined merge-sort algorithm	24
3.1	Example of an MMKP	38
3.2	Finding actual number of upgrades	45
4.1	Performance of different strategies of MS-HEU normalized with respect to M-HEU for the MMKP data sets with $l=25$ and $m=25$	53
4.2	Performance of different strategies of MS-HEU normalized with respect to M-HEU for the MMKP data sets with $n=2500$ and $l=25$	53
4.3	Performance of different strategies of MS-HEU normalized with respect to M-HEU for the MMKP data sets with $n=2500$ and $m=25$	54
4.4	Performance of different strategies of MS-HEU and M-HEU normalized with the upper bound for the MMKP data sets with $l=10$ and $m=10$	54
4.5	Performance of different strategies of MS-HEU and M-HEU normalized with the upper bound for the MMKP data sets with $n=500$ and $l=10$	55
4.6	Performance of different strategies of MS-HEU and M-HEU normalized with the upper bound for the MMKP data sets with $n=500$ and $m=10$	55
4.7	Time required by different strategies of MS-HEU and M-HEU for the MMKP data sets with $m=25$ and $l=25$	56
4.8	Time required by different strategies of MS-HEU for the MMKP data sets with $m=25$ and $l=25$	56

4.9	Time required by different strategies of MS-HEU and M-HEU for the MMKP data sets with $n=2500$ and $l=25$	57
4.10	Time required by different strategies of MS-HEU for the MMKP data sets with $n=2500$ and $l=25$	57
4.11	Time required by different strategies of MS-HEU and M-HEU for the MMKP data sets with $n=2500$ and $m=25$	58
4.12	Time required by different strategies of MS-HEU for the MMKP data sets with $n=2500$ and $m=25$	58
4.13	Comparison of the total values of the items picked by PRAM-HEU, M-HEU and Upper-Bound for 10 uncorrelated problem sets with $n=100$, $m=5$, $l=10$	59
4.14	Comparison of the total values of the items picked by PRAM-HEU, M-HEU and Upper-Bound for 10 correlated problem sets with $n=100$, $m=5$, $l=10$	59
4.15	Example of an MMKP with available resources	60

List of Tables

2.1	The lists arising during the execution of the indicated stages of the pipelined merge-sort algorithm	25
3.1	Summary of the complexities of different steps	48
4.1	Time requirements by M-HEU and New Heuristic algorithm for solving the MMKP with correlated and uncorrelated data sets varying n	51
4.2	Time requirements by M-HEU and New Heuristic algorithm for solving the MMKP with correlated and uncorrelated data sets varying m	52
4.3	Time requirements by M-HEU and New Heuristic algorithm for solving the MMKP with correlated and uncorrelated data sets varying l	52

Abstract

Multiple-Choice Multi-Dimension Knapsack Problem (MMKP) is a variant of the classical 0-1 Knapsack Problem. It has a knapsack with a multidimensional capacity constraint and groups of items where each item having a utility value and a multidimensional weight constraint. The knapsack is to be filled by picking up exactly one item from each group. The problem is to maximize the total value of the items in the knapsack but not exceeding the knapsack capacity. MMKP is an NP-Hard problem and its exact solution is not suitable for real time decision making applications. Therefore heuristic based approximation algorithms are developed. Khan developed a heuristic, HEU, which achieves 93% of the optimal solution value. Later Akbar et al. presented M-HEU, a modification of HEU, achieving 96% of the optimal value with a time complexity of $O(mn^2l^2)$, where n is the number of groups, l is the number of items in each group and m is the number of resource constraints. But, these heuristic algorithms do not scale better for larger systems. In this thesis, a new sequential heuristic algorithm is developed by modifying M-HEU to some extent that would be parallelized. Later a parallel heuristic algorithm is introduced that is the parallel version of the new sequential heuristic algorithm. And the new sequential heuristic algorithm is used to compare the performance of the parallel heuristic algorithm. Experimental result shows the new heuristic algorithm achieves 94.5% of the optimal value. The time complexity of the parallel algorithm is $O(\log nl(\log n + \log m + \log \log l))$ with $O(n \log nl(\log n + lm))$ number of operations in Concurrent Read Concurrent Write (CRCW) PRAM model, i.e., the required number of processors is $O((n \log n + nlm)/(\log n + \log m + \log \log l))$. This also means that we have a sequential heuristic algorithm running in $O(n \log nl(\log n + lm))$ time which seems to be remarkable since M-HEU, a celebrated sequential heuristic, although achieves 96% of optimal value, takes the time complexity of $O(mn^2l^2)$.

CHAPTER-1



Introduction

Knapsack problem and its variants are widely used in many resource management problems such as resource scheduling in multimedia server, admission control and profit maximization, menu planning etc. There are several variants of Knapsack Problem (KP) such as Multiple-Choice Knapsack Problem (MCKP), Multi-Dimensional Knapsack Problem (MDKP), Multiple-Choice Multi-Dimension Knapsack Problem (MMKP) etc. In this chapter, we define the classical 0-1 KP and MMKP.

1.1 Definition of KP and MMKP

The 0-1 knapsack problem (0-1 KP) is a well-known problem in the field of computer science. In KP, there is a knapsack with finite capacity and a set of items each having a value and a weight. The knapsack is to be filled with the items, each item taken completely or excluded. The 0-1 KP is to maximize the total value of the items in the knapsack, so that the total resource required does not exceed the resource constraint of the knapsack.

The classical 0-1 Knapsack Problem (KP) can be described as follows. Suppose there are n objects, and a knapsack or bag. The value v_i denotes value (or profit) provided by Item i , weight r_i denotes resource required by Item i , and R denotes the amount of available resource. Here the problem is to allocate resource to a subset of items in order to maximize the total value such that the total allocated resource does not exceed the available resource.

Mathematically the problem is stated as follows:

$$V = \text{maximize } \sum_{i=1}^n x_i v_i$$
$$\text{such that } \sum_{i=1}^n x_i r_i \leq R,$$
$$x_i \in \{0,1\}, \quad i = 1,2,\dots,n$$

Here x_i 's for $i = 1, 2, \dots, n$ are variables. The problem is called the 0-1 knapsack problem because variable x_i can either be taken or left behind, *i.e.* a value of 0 implying Item i is not picked, or a value of 1 implying Item i is picked. Any pick of items which satisfy the constraint is called a feasible solution of the problem. The solution of the 0-1 knapsack problem is the feasible solution which maximizes the sum of the value of the picked items. Figure 1.1 illustrates a KP where maximum value that can be achieved is 18 and the weight capacity of the knapsack is 10.

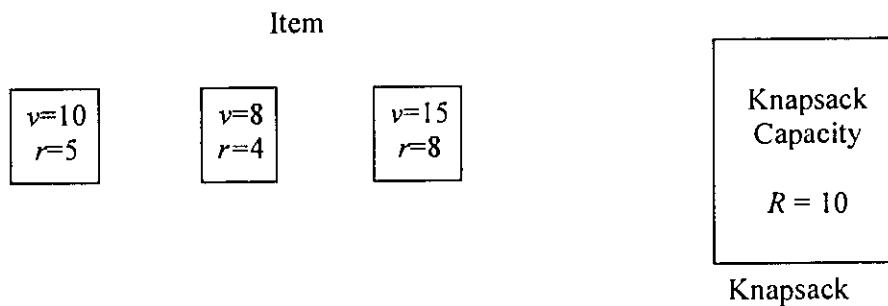


Figure 1.1: The classical 0-1 knapsack problem

The Multiple-Choice Multi-Dimension Knapsack Problem (MMKP)[1, 2, 3, 4] is a variant of the classical 0-1 KP, where there are different groups of items and exactly one item can be picked up from each group; and the constraints are multi-dimensional. Let there be n groups of items, Group i containing l_i items, Item j of Group i has a utility value v_{ij} and an m dimensional resource cost $\vec{r}_{ij} = (r_{ij1}, r_{ij2}, \dots, r_{ijm})$. The resource constraint $\vec{R} = (R_1, R_2, \dots, R_m)$ is also m dimensional. The MMKP problem is to maximize the utility value $V = \sum_{i=1}^n \sum_{j=1}^{l_i} x_{ij} v_{ij}$ subject to the resource constraint $\sum_{i=1}^n \sum_{j=1}^{l_i} x_{ij} r_{ijk} \leq R_k$, where $1 \leq k \leq m$ and $x_{ij} \in \{0,1\}$ and $\sum_{j=1}^{l_i} x_{ij} = 1$.

Figure 1.2 illustrates an MMKP with 3 groups and 2 resource requirements. Values and resource requirements of each item are shown inside the boxes representing items. Objective is to pick up exactly one item from each group to maximize the total value of picked items maintaining $\sum r_1$ of picked items ≤ 43 and $\sum r_2$ of picked items ≤ 45 .

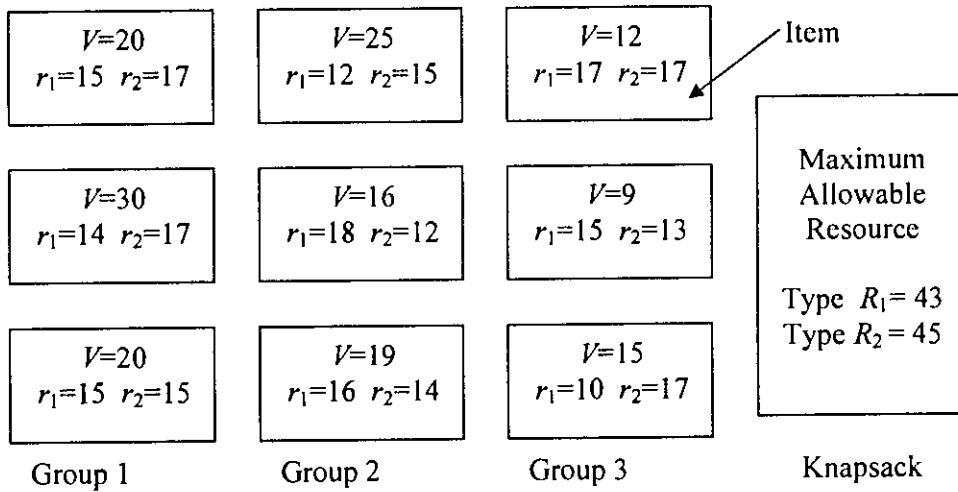


Figure 1.2: Example of an MMKP

1.2 Applications of MMKP:

MMKP has its application in many resource management problems. For example, it can be used to solve the resource management problem in the delivery of multimedia streams from an Adaptive Multimedia System (AMS) [1, 3] that has been proposed by Khan. The users place request for sessions in these systems and pay according to the Quality of Service (QoS) they are getting. As resources of servers such as CPU cycles, memory, I/O bandwidth etc. are limited decisions have to be made whether new users should be admitted or not and the QoS level of the session if it is admitted. This decision-making must consider the amount of available resources and also the maximization of total revenue that can be earned from sessions. This type of admission control in these systems is a real-time problem that requires decisions of admission or rejection within a certain amount of time. Problem of admission control can be mapped to MMKP by mapping users to groups, quality of service profiles of users to items in a group, server resources to resources and revenue earned to values. We also consider a menu selection problem [1] that can be mapped to the MMKP. A person wants to have a meal where he has to choose a beverage, an appetizer, a main dish and a dessert. For each choice, his preference is expressed as a satisfaction value. The problem is to find a meal which maximizes the total satisfaction subject to

constraints such as maximum values of calories, cholesterol and cost. Other applications such as cargo loading, capital budgeting, industrial production can also be mapped to the MMKP.

1.3 Different Types of Algorithms for Solving MMKP

There are two types of solutions for the MMKP: optimal solutions or exact solutions, and near-optimal solutions or heuristic based solutions. The worst-case computation time of the exact solutions for MMKP grows exponentially with the size of the problem and its exact solution may not be feasible in any real time problem like admission control problem in an adaptive multimedia system [1, 2, 3, 4]. Dynamic Programming or Branch and Bound Algorithm can be used to obtain an optimal solution to the MMKP. The heuristic based solutions is used to provide solutions which are close to optimal values, but require computation times which are much shorter than those of the optimal solutions. Different heuristic approaches have been developed to obtain near optimal solution to the MMKP. Khan developed a heuristic, HEU [1, 2], based on the concept of aggregate resource consumption. Later Akbar *et al.* presented a modification of HEU, M-HEU [3, 4], which finds the solution achieving better optimality than HEU. Another heuristic, C-HEU [3], also developed by Akbar finds the solution by constructing a convex hull. There are also some parallel algorithms developed for solving knapsack type problems, but so far no parallel algorithm has been reported for solving MMKP.

1.4 Motivation

Many practical problems in resource management can be mapped to the MMKP. The utility model of adaptive multimedia system (AMS) has been proposed by Akbar [3, 4]. Users place requests for sessions to AMS and pay the owners of the AMS according to the Quality of Service (QoS) they are getting. As resources of servers such as CPU cycles, memory, I/O bandwidth etc. are limited decisions have to be made whether new users should be admitted or not and if admitted which level of QoS they will enjoy. Problem of admission control in an AMS exactly fits an MMKP and can be mapped to the MMKP [4]. Users submitting their requests to a multimedia system can be represented by the groups of items. Each level of QoS of a user's

requested session is equivalent to an item. Each session is equivalent to a group of items. The values of the items are equivalent to offered prices for the session. The multimedia server is equivalent to the knapsack with limited resources, e.g. CPU cycles, I/O bandwidth and memory. This type of admission control in these systems is a real-time problem which requires decisions of admission or rejection within a limited time frame.

The exact solution of MMKP is not suitable for real time decision-making applications. So heuristic based approximation algorithms are developed. These heuristics for solving the MMKP can be used for solving real time admission control problem in Adaptive Multimedia System (AMS). But if the number of groups of the MMKP increases in a large multimedia system, it is not efficient to run M-HEU to perform admission control. So, we need a faster algorithm to achieve the real-time response in admission control. A parallel version of M-HEU can achieve better computational speed.

Parallel computation is currently an area subject to intense research activity. There has always been a need to solve large-scale computational problems [5]. These problems must be solved in a reasonable time scale, which implies fast computers to do the job. In those cases, where the application must be presented by a well-determined deadline, the parallel computation will be applied. Recent technological advances have opened up the possibility of performing massively parallel computations cost effectively and have made the solution of such problems possible.

1.5 Problem Definition

Since MMKP is an NP-Hard problem [6], algorithms for finding the exact solution for MMKP are not applicable to the real time admission control problem. To meet up any real time demand we often use heuristic solutions of such problem. There are several heuristic algorithms developed for solving MMKP. M-HEU [3, 4] is the best among these heuristics as far as percentage of achieved optimality is concerned. But quadratic complexity M-HEU does not scale better when number of groups increases in large multimedia systems. We find that the time requirement for M-HEU can be further reduced, if computations can be done in parallel. However, it is seen that it is not possible to provide a parallel algorithm from M-HEU directly. Therefore, a new

heuristic algorithm is required for solving MMKP that would be parallelized and provide a polylog time solution.

1.6 Scope and Focus

The main focus of this work is to present a parallel algorithm of a heuristic for Multiple-Choice Multidimensional Knapsack Problem (MMKP). Development of exact algorithms is beyond the scope of our work. Also PRAM machines are not available in reality. So the implementation and developing the prototype of this algorithm is beyond the scope of this thesis. As described in previous section, a new serial heuristic algorithm is developed and we implemented it. Then we can get the total value achieved from the algorithm by varying different parameters and we can compare the performance of our algorithm with respect to M-HEU. We also compute the performance of our algorithm in terms of achieved optimality. We also compare the time requirement of the new serial heuristic algorithm and that of the M-HEU by varying different parameters. But the comparison of time requirement of our proposed parallel algorithm with M-HEU is beyond the scope of the current research. We compute the time complexity and the total number of operations of the new sequential heuristic algorithm and of the proposed parallel algorithm.

1.7 Outline

This thesis is organized in five chapters. In this section we briefly describe the organization of the rest of the chapters.

In Chapter 2 a review of the literature of KP and its variants have been carried out. We describe here the algorithms related to the KP and its variants. Parallel algorithms for the KP and its variants are also described in this chapter. The parallel random access machine (PRAM) model has also been included in this chapter.

Chapter 3 presents the parallel heuristic algorithm to solve the MMKP. First two sections of this chapter describe why we chose M-HEU for this thesis and the problem of M-HEU to be parallelized. A new serial heuristic algorithm has been described in the next section, developed by modifying the M-HEU that would be parallelized. Then the parallel heuristic algorithm is described followed by the analysis of the algorithm.

In Chapter 4 we present the experimental results. A description of the experimental procedures has been given with the results presented in the graphs. An analysis of the results of the experiments has also been performed at the end of this chapter.

We conclude the thesis in Chapter 5 by describing the contributions of the current research. Some suggestions for future research work have also been included in this chapter.

CHAPTER 2

Background and Preliminaries

There are various algorithms for solving variants of Knapsack Problems. In this chapter, we briefly describe some of these algorithms.

2.1 Related Research on KP and its variants, MDKP and MCKP

The MDKP, multidimensional Knapsack Problem is one kind of KP with multiple resource constraints for the knapsack, i.e. the resources are multidimensional in this type of KP. The MCKP, Multiple Choice Knapsack Problem is another KP where the picking criterion for items is restricted. In this variant of KP, there are one or more groups of items. Exactly one item will be picked from each group subject to resource constraint.

KP is an NP-Hard problem [6]. The variants of KP are as hard as KP. So these are all NP-Hard. There are two types of algorithms for solving the KP and its variants: one is for obtaining exact solution and the other is for obtaining approximate solution.

2.1.1 Exact Solutions for KP, MDKP and MCKP

Dynamic Programming and Branch and Bound approach can be used to obtain optimal solutions [1, 3] to the classical 0-1 KP. Dynamic Programming method [7] uses sequence of decisions, regarding whether to pick an item or not, leading to an optimal solution. It is a design method that can be used when the solution to a problem can be viewed as the result of a sequence of decisions. It acts as the divide-and-conquer method, solves problems by combining the solutions of subproblems. Divide-and-conquer algorithms divide the problem into independent subproblems, solve the subproblems recursively, and then combine their solutions to solve the original problem. Dynamic programming is applicable when the subproblems are not independent, i.e., when subproblems share sub subproblems. The solution to the knapsack problem can be viewed as the result of a sequence of decisions. Let there be n items in the knapsack and x_i define the i th item. We have to decide the values of x_i for $1 \leq i \leq n$. First we make a decision on x_1 , then on x_2 , then on x_3 , and so on. An

optimal sequence of decisions maximizes the objective function satisfying the resource constraints.

On the other hand, Branch-and-bound is a general and popular method for solving combinatorial optimization problem. In this method the optimal solution is found using iterative generation of a tree, called search tree [1].

A node in the search tree represents a solution state where there may be some variables whose values are known (values are assigned) and that of some others' may be unknown (values are not assigned). This method starts with a single-node tree where the values of all the variables are unknown. A node of this tree may be expanded based on a variable whose value is unknown at the current node. For example, expanding a node based on binary variable x_i may generate two nodes: one for $x_i = 0$ and the other for $x_i = 1$. No node will be generated for $x_i = 1$ if it is not feasible (that is picking item i violates the resource constraint). A node which has been generated and whose children have not yet been generated is called a live node. The expanding node or simply the e-node is the node which has the largest upper bound among the live nodes. For exploration, upper bound of the objective function is computed from the known values at each node. The node with the largest upper bound is explored. A node producing the largest upper bound having no unknown variable is the solution node. The upper bound is computed using linear programming technique. Linear programming is a deterministic tool where all the model parameters are assumed to be known with certainty. The Simplex Method is a very powerful technique for solving linear programs. The Simplex Method requires that each of the constraints be put in a special standard form in which all the constraints are expressed as equations by augmenting slack or surplus variables as necessary. This type of conversion normally results in a set of simultaneous equations in which the number of variables exceeds the number of equations. Even though the worst case computational complexity of the Simplex Method grows exponentially with the problem size, this method is very efficient practically [1].

Kolesar [8] gave the first branch and bound algorithm for the classical 0-1 KP. This algorithm uses a greedy-like strategy where at any e-node; it branches on the not-yet-decided item which provides a highest value per unit of required resource (v_i/r_i). Shih [9] presented a branch-and-bound algorithm for the MDKP. For upper bound estimation, this algorithm treats the MDKP problem as m single dimensional KPs, and

calculates the optimal value of the objective function in each case. The minimum of these objective function values is then used as an upper-bound. Branch and bound algorithm for the MCKP was proposed by Naus [10].

2.1.2 Approximate Solutions for KP, MDKP and MCKP

Different heuristics have been developed to obtain approximate solutions to the KP and its variants. These approaches use some kind of greedy like method to generate solutions. For the classical 0-1 KP, a greedy approach to get a near optimal solution is as follows: pick the item with the largest v_i/r_i (value per unit resource), then pick the item with the second largest v_i/r_i , and so on until no more item can be picked because the available resource is not enough or no item is left. The greedy method is perhaps the most straightforward technique and it can be applied to a wide variety of problems. Most though not all, of these problems have n inputs and they require a subset that satisfies some constraints. Any subset that satisfies these constraints is called a feasible solution.

The greedy approach can be generalized for other variants of KP. Toyoda [11] proposed a simple solution to the MDKP using the concept of aggregate resource. In this algorithm the main idea was to penalize the not yet picked items depending in the current resource state. This idea penalizes the items with greater requirement of those resources that are already consumed much. So, if two items produces the same value then the item with less penalty is preferred. Suppose in an m resource MDKP instances, the current resource usage vector is given as $C=(C_1, C_2, \dots, C_m)$ and resource requirement of Item i is given by $r_i=(r_1, r_2, \dots, r_m)$. Then a_i , the aggregate resource required by Item i is computed as follows:

$$a_i = \frac{(r_1 C_1 + r_2 C_2 + \dots + r_m C_m) \cdot r_i}{|C|} = \frac{r_i \cdot C_i}{|C|}, \text{ where } |C| \text{ denotes the magnitude of vector } C_i$$

and \cdot denotes the dot product of vector. Toyoda's algorithm starts with no item as the initial solution and adds items iteratively one at a time. In each iteration, the item with the maximum v_i/a_i (value per unit of aggregate resource for item i) is picked.

Magazine [12] proposed another heuristic based on Lagrange Multipliers to solve the MDKP, for maximizing the objective function subject to constraints. All resource constraints are incorporated into the maximization goal. Initially all Lagrange

Multipliers are set to zero and this is in general not a feasible solution for the MDKP. Next all actual resource consumptions are determined and the most violated constraint is identified. The corresponding multiplier is then increased as much as necessary to update the resource consumptions. This step is repeated until the solution has become feasible.

Tabu Search [13], Simulated Annealing [14] and Genetic Algorithms [15] can be applied to solve the variants of Knapsack Problem. The Genetic algorithm has the exponential worst case complexity - it can explore all of the items. This algorithm is based on natural selection and genetics. The algorithm combines a random selection by the survival of the fittest theory. The strongest individuals in a population will have a chance to transfer their genes to the next generation. Simply it can be coded a number of different solutions of a problem as a bitstring, and evaluate their fitness in relation to each other. Every solution can be seen as an individual in a population, and the bitstring can be seen as the genes of the individual. Then the individuals are selected based on their fitness. Tabu search and simulated annealing are based on looking at the neighbours. These are costlier than the heuristics using greedy approach. The Tabu search begins by marching to local minima. To avoid retracing the steps used, the method records recent moves in one or more Tabu lists. The original intent of the list was not to prevent a previous move from being repeated, but rather to insure it was not reversed. In Tabu Search, Simulated Annealing and Genetic Algorithm approach current solution is moved to another solution by upgrading some and downgrading some. This upgrade and downgrade at the same step requires more time because we have to search all neighbouring combinations of current solution.

2.2 Related Research on MMKP

MMKP is actually the combination of MDKP and MCKP. As usual, there are two methods of finding solutions for an MMKP: one is a method for finding exact solutions and the other is heuristic solution.

2.2.1 Algorithms for Exact Solutions to the MMKP

Khan [1] presented an exact algorithm for the MMKP using the Branch and Bound Linear Programming (BBLP) technique. The Branch and Bound technique has been discussed already in the previous section.

2.2.2 Algorithms for Heuristic Solutions to the MMKP

Since MMKP is an NP-hard problem, the computation time for any optimal algorithm, such as BBLP, may grow exponentially with the size of the problem instance in the worst case. This may not be acceptable for time-critical applications such as admission control and dynamic resource allocation in a multimedia system. These applications are forced to accept a near-optimal solution if the computational time for optimal solution exceeds real-time bounds.

Moser's [16] heuristic algorithm uses the concept of graceful degradation from the most valuable items based on Lagrange Multipliers to solve the MMKP. It starts with the most valuable item of each group as the selected item and in general the resource constraints to be violated in that case. The initial choice of items is adapted to obey the resource constraints by repeatedly improving on the most violated resource constraints. But this algorithm fails to find a feasible solution when the resources are short. However in those cases, some other heuristics such as M-HEU, I-HEU [3, 4] find a feasible solution by starting from the lowest valued items and try to find a feasible solution by upgrading the solution if the current one is not feasible.

A new heuristic algorithm HMMKP is proposed for solving MMKP with the time complexity of $O(mn^2(l^2 - l))$ by Hernandez [17]. This heuristic needs to solve the Linear Programming Relaxation (LPR) of a relaxation of the MMKP. In this heuristic, an LPR of the MMKP is done, then the relaxed problem is solved and the Lagrange multipliers are obtained. Later the Lagrange multipliers are used in order to compute the pseudo-utility values needed for the MMKP solution.

The Guided Local Search (GLS) algorithm [18] is a recent approach for solving MMKP, moves out of a local Maximum/minimum by penalizing particular solution features that it considers should not occur in a near-optimal solution. The initial feasible solution is obtained here by applying a Constructive Procedure (CP). CP is a greedy procedure which generates a feasible solution by considering the Feasible State (FS) process. The Complementary CP approach, called CCP [18, 19], uses an iterative improvement of the initial feasible solution. A reactive local search based algorithm is proposed in [18], where the algorithm starts by an initial feasible solution and improved by using a fast iterative procedure. The aim of this process is to improve the complementary solution obtained by CCP. Later a different procedure,

named unblocking procedure is introduced in order to escape to local optima. Finally, a memory list is applied in order to forbid the repetition of configurations. The worst case complexity of this algorithm is $O(nlm^2)$ floating point operations.

HEU, a heuristic developed by Khan [1, 2], finds the solution of the MMKP using the concept of aggregate resource consumption. Later Akbar [3, 4] presented a modification of HEU, M-HEU, which achieves better optimality than HEU. In these heuristics, the selection of the lowest valued item in each group is defined as the initial solution of an MMKP. If this solution is not feasible then HEU terminates notifying "No solution found". However, there may be a solution using higher-valued items that requires fewer resources. Thus a new step should be added to find a feasible solution if the initial solution is not feasible. Again HEU finds a solution by only upgrading the selected items of each group. There might be some higher-valued items in the MMKP, which makes the solution infeasible, but if some of the groups are downgraded we can get a feasible solution. This method of upgrading followed by downgrading may increase the total value of the solution. Thus M-HEU modifies HEU by adding a pre-processing step to find a feasible solution and a post-processing step to improve the total value of the solution. Incremental heuristic solution, I-HEU also presented by Akbar [3, 4] with the same optimality as M-HEU. If the number of groups in the MMKP is very large then it is not efficient to run M-HEU. An incremental solution is a necessity to achieve a better computational speed. By changing the technique of finding feasible solution we can use M-HEU to solve the MMKP incrementally. C-HEU, another heuristic developed by Akbar [3] using the concept of convex-hull, provides solution to the MMKP in logarithmic worst-case time complexity. It is an incremental heuristic. It has lower order of complexity but the optimality achieved by this heuristic is much inferior to the other heuristics.

There is a number of iteration in every heuristic and each iteration is highly dependent on its previous iteration, so that it is really difficult to provide a parallel version from these heuristics directly. In this research M-HEU is modified to some extent that would be parallelized. So it is better to describe the HEU and M-HEU in detail for clear understanding of our new algorithm.

Algorithm HEU:

HEU [1, 2] achieves 93% optimal solution with a complexity of $O(mn^2l^2)$ operation.

The principles of the HEU are as follows:

- The items of each group are sorted in nondecreasing order according to the value associated with them and it selects the lowest valued items from each group as the initial solution. It then upgrades the solution gradually by choosing new items as along as the solution remains feasible.
- It uses Toyoda's concept of aggregate resource where the required resource vector of an item is converted to a scalar index using penalty factors taken from the current resource usage vector. Here the main idea is to penalize the use of resources depending on the current resource state. It applies a large penalty for a heavily used resource, and a small penalty for a lightly used resource.
- To find the next item to be picked, it chooses the one which has the highest positive change in aggregate consumed resource (one which gives the best revenue with the least aggregate resource). But if no such item is found, it chooses the one which maximizes the value gain per unit aggregate resource. It can be defined by a vector $\Delta d'_{ij}$, relative change of aggregate resource consumption and the item is chosen with the maximum value of $\Delta d'_{ij}$ which is defined as follows:

$\Delta d'_{ij} = \{v, b\}$ is a vector, where

$$\Delta d'_{ij} = \begin{cases} (\Delta v_{ij} / \Delta a_{ij}, 0) & \text{if } \Delta a_{ij} \leq 0 \\ (\Delta a_{ij}, 1) & \text{if } \Delta a_{ij} > 0 \end{cases}$$

And $\Delta d'_{ij} > \Delta d'_{kl}$, if $b(\Delta d'_{ij}) > b(\Delta d'_{kl})$ or $(b(\Delta d'_{ij}) = b(\Delta d'_{kl}) \text{ and } v(\Delta d'_{ij}) > v(\Delta d'_{kl}))$

Where, the change of aggregate resource consumption, $\Delta a_{ij} = \sum_k (r_{i\rho[i]k} - r_{ijk}) \times C_k$

and the change of revenue, $\Delta v_{ij} = v_{i\rho[i]} - v_{ij}$.

$C_k = \sum r_{i\rho[i]k}$, $\rho[i]$ is the currently selected item in group i .

Algorithm M-HEU:

A new heuristic algorithm, M-HEU [3, 4], modification of HEU is proposed by Akbar for solving MMKP with time complexity of $O(mn^2l^2)$. M-HEU finds 96% optimal solutions on average with much reduced computational complexity and performs favorably relative to other heuristic algorithms for MMKP.

The items in each group of the MMKP are sorted in non-decreasing order according to the value associated with them. So, the bottom items in each group are to be considered as *lower-valued items* than the top ones. Picking a higher-valued or lower-

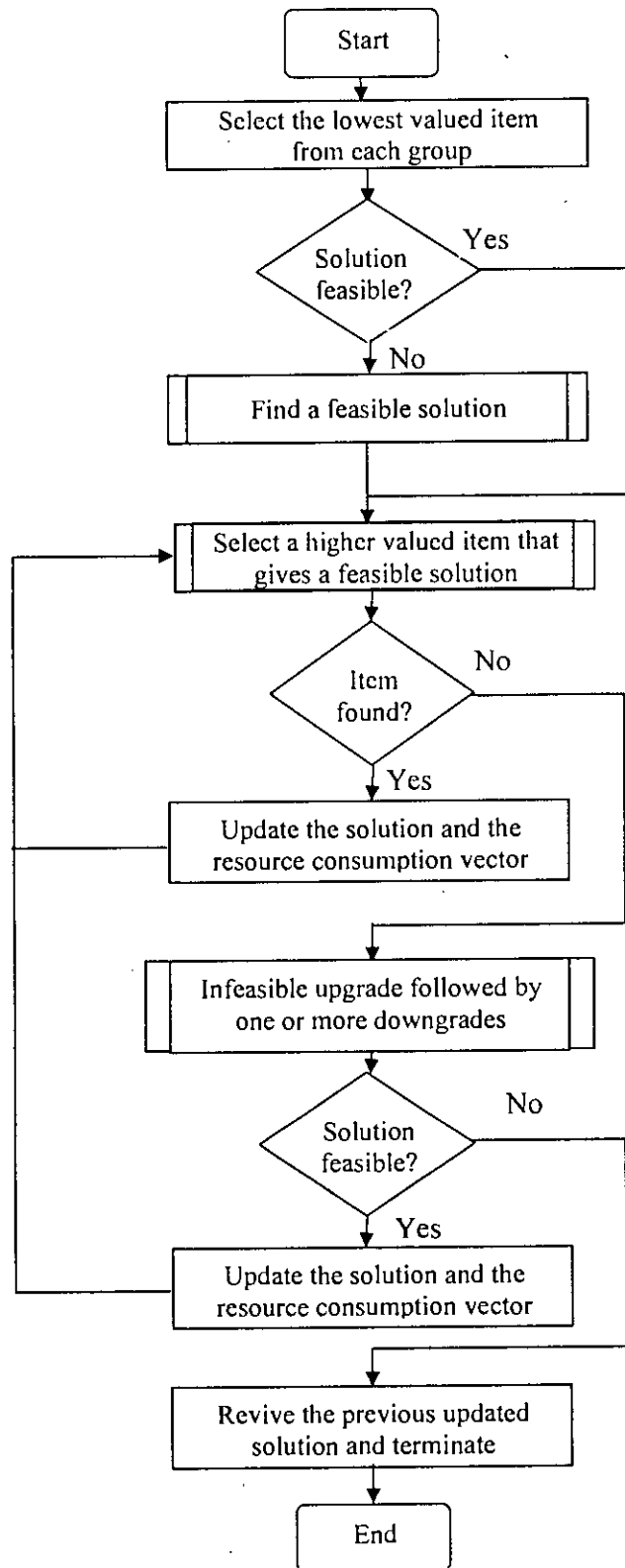


Fig 2.1: Flow chart of M-HEU

valued item than the currently selected item in a group is called an *upgrade* or a *downgrade* respectively. In the heuristic it is necessary to find an upgrade or downgrade frequently. It modifies HEU by adding a pre-processing step to find a feasible solution if the initial solution is infeasible because there may be a solution using higher valued items that requires fewer resources. It also uses a post-processing step to improve the total value of the solution with one upgrade followed by one or more downgrades. Because there might be some higher valued items in the MMKP, selection of which make the solution infeasible, but if some of the others groups are downgraded we can get a feasible solution. This method of upgrading followed by downgrading may increase the total value of the solution.

Steps in the Algorithm:

Step1: Finds a feasible solution, if initial solution is not feasible.

Step2: Feasible Upgrades in each iteration.

Step3: Infeasible upgrade followed by one or more downgrades

Fig 2.1 shows the steps of M-HEU algorithm.

2.3 PRAM Model

The Parallel Random Access Machine (PRAM) [20] model is actually the synchronous shared memory model, where all the processors operate synchronously under the control of a common clock. It consists of a number of processors, typically of the same type, each of which has its own local memory and can execute its own local program. The processors are interconnected in a certain fashion to allow the coordination of their activities and the exchange of data through a shared memory unit. Each processor is uniquely identified by an index, called a processor number or processor ID, which is available locally and hence, can be referred to in the processor's program. Figure 2.2 shows a general view of a shared memory model with p processors. These processors are indexed 1, 2, ..., p . Shared memory is also referred to as global memory.

The main purpose of parallel processing is to perform computations faster than can be done with a single processor by using a number of processors concurrently. The pursuit of this goal has had a tremendous influence on almost all the activities related to computing. There are many applications in day-to-day life that demand real time

solutions to problems. These include fluid dynamics, weather prediction, modeling and simulation of large systems, information processing and extraction, image processing, artificial intelligence and automated manufacturing. For example, weather forecasting has to be done in a timely fashion. In the case of severe hurricanes or snowstorms, evacuation has to be done in a short period of time. If an expert system is used to aid a physician in surgical procedures, decisions have to be made within seconds. And so on. Programs written for such applications have to perform an enormous amount of computation.

In the forecasting example, large-sized matrices have to be operated on. In the medical example, thousands of rules have to be tried. Even the fastest single processor machine may not be able to come up with solutions within tolerable limits. Parallel machines offer the potential of decreasing the solution time enormously.

The running time of a parallel algorithm depends on the number of processors executing the algorithm as well as the size of the problem input. Therefore we discuss both the time and the number of processors required when analyzing PRAM algorithms. Typically there is a trade-off between the number of processors used by an algorithm and its running time.

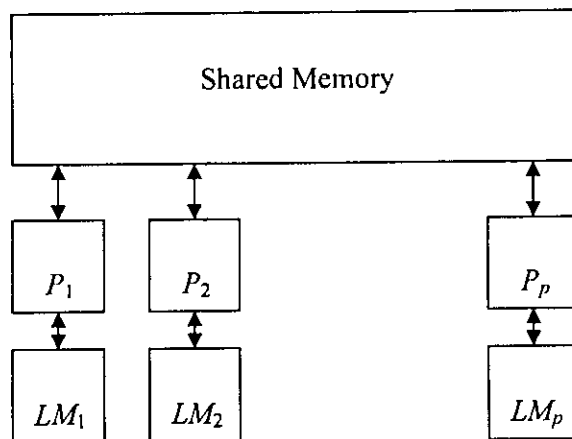


Figure 2.2: The shared memory model

2.3.1 Concurrent versus Exclusive Memory Accesses

There are several variations of the PRAM model based on the assumptions regarding the handling of the simultaneous access of several processors to the same location of the global memory [20].

A concurrent read algorithm is a PRAM algorithm during whose execution multiple processors can read from the same location of shared memory at the same time. An exclusive read algorithm is a PRAM algorithm in which no two processors ever read the same memory location at the same time. We make a similar distinction with respect to whether or not multiple processors can write into same memory location at the same time, dividing PRAM algorithms into concurrent write and exclusive write algorithms. The commonly used PRAM models are:

- 1) Exclusive Read Exclusive Write (EREW).
- 2) Concurrent Read Exclusive Write (CREW).
- 3) Concurrent Read Concurrent Write (CRCW).

The exclusive read exclusive write (EREW) PRAM does not allow any simultaneous access to a single memory location. The concurrent read exclusive write (CREW) PRAM allows simultaneous access for a read instruction only. Access to a location for a read or a write instruction is allowed in the concurrent read concurrent write (CRCW) PRAM. The three principle varieties of CRCW PRAMs are differentiated by how concurrent writes are handled. The common CRCW PRAM allows concurrent writes only when all processors are attempting to write the same value. The arbitrary CRCW PRAM allows an arbitrary processor to succeed. The priority CRCW PRAM assumes that the indices of the processors are linearly ordered and allows the one with the maximum or minimum index to succeed.

2.3.2 Optimality of PRAM Algorithms

Suppose there is a computation problem P of size n . Let the sequential time complexity of P be $T(n)$. That is there is a sequential algorithm that solves P within this time bound and in addition, it can be proved that no sequential algorithm can solve P faster. A parallel algorithm to solve P will be called optimal [20] if the work $W(n)$ required by the algorithm satisfies $W(n) = O(T(n))$. Otherwise the parallel algorithm is called non optimal.

2.3.3 Examples of PRAM Algorithms

Some PRAM algorithms such as summing, sorting, searching, prefix sum are described below:

- **Parallel Sum Algorithm:** We can determine the sum of n elements by using a balance binary tree constructed on the n input elements [20].

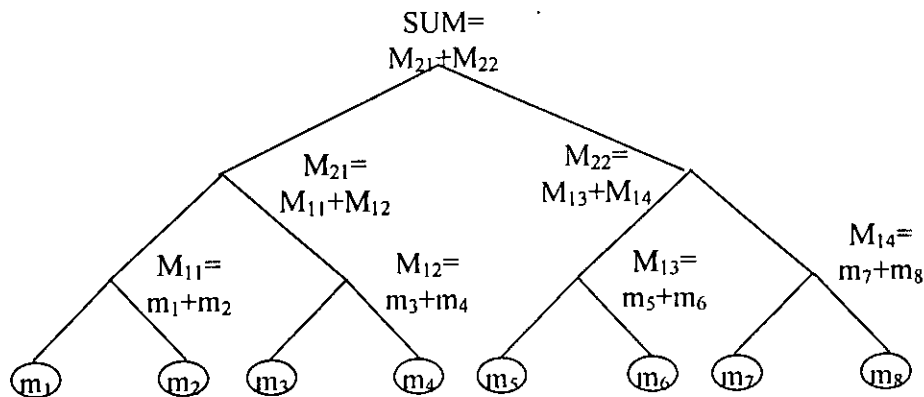


Fig2.3: Summation of n elements

The running time of the parallel algorithm is $O(\log n)$ and the total number of operations used is $O(n)$. This parallel algorithm is optimal, since the work performed matches the run time of the best known sequential algorithm of the problem. As an example, the summation of 8 numbers (m_1, m_2, \dots, m_8) is shown in Fig 2.3.

- **Prefix Algorithm:** A prefix sum algorithm presented in [7] is described as follows: Suppose there are n input elements, defined as x_1, x_2, \dots, x_n . The prefix computation is to compute the n elements as $x_1, x_1 + x_2, x_1 + x_2 + x_3, \dots, x_1 + x_2 + x_3 + \dots + x_n$. The output elements are often referred to as the prefixes. Prefix computation on an n -element input can be performed in $O(\log n)$ time using $n/\log n$ CREW PRAM processors. The work done by this algorithm is $O(n)$ and hence the algorithm has an efficiency of $O(1)$ and is work-optimal.

Suppose there are $n/\log n$ processors assigning $\log n$ elements each. Processor i ($i=1, 2, \dots, n/\log n$) computes the prefixes of its $\log n$ assigned elements in parallel. Let the results be $z_{(i-1)\log n+1}, z_{(i-1)\log n+2}, \dots, z_{i \log n}$. Then a total of $n/\log n$ processors collectively employ a non work-optimal algorithm to compute the prefixes of the $n/\log n$ elements $z_{\log n}, z_{\log 2n}, z_{\log 3n}, \dots, z_n$. Finally each processor updates the prefixes it computed in the first step except Processor 1.

Let the input to the prefix computation be 5, 12, 8, 6, 3, 9, 11, 12, 1, 5, 6, 7, 10, 4, 3, and 5. Here $n = 16$ and $\log n = 4$. So the number of processors is 4. In the first step, each processor computes prefix sums on four numbers each. In the next step,

prefix sums on the local sums is computed. And in the last step, the locally computed results are updated. In this step, Processor 1 does not update the prefixes that are computed locally. The prefixes of Processor 2 are updated by adding the 1st prefix of the global computation (Computed in the previous step) with them. In the similar way, 2nd and 3rd prefixes of the global computation are added with the locally computed prefixes of Processor 3 and 4 respectively. The prefix sums computation is shown in Fig 2.4.

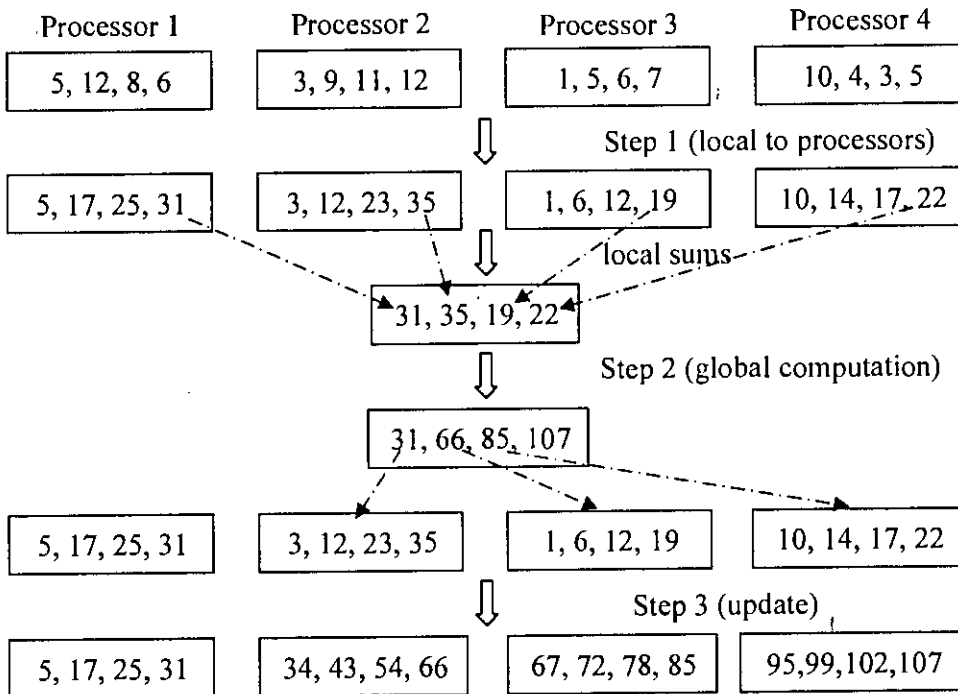


Fig 2.4: An example of prefix computation

- Parallel Maximum Finding Algorithm:** Finding the maximum of n arbitrary numbers can be presented in $O(\log \log n)$ time using $n/(\log \log n)$ common CRCW PRAM processors [20]. So the work done of the algorithm is $O(n)$ and it is work-optimal.

There is an algorithm based on the logarithmic-depth binary tree for finding the maximum in $O(\log n)$ time and it is optimal. There is another algorithm that is nonoptimal but runs in doubly logarithmic time, i.e. it requires $O(\log \log n)$ time using $O(n \log \log n)$ operations. Then these two algorithms can be combined into an optimal and a very fast algorithm. In the first step the binary tree algorithm is

applied, starting from the leaves of a binary tree and moving up to $\lceil \log \log \log n \rceil$ levels. Since the number of candidates reduces by a factor of $\frac{1}{2}$ per level as we grow up the binary tree, $n' = O(n/\log \log n)$ elements are generated at the end of the binary tree algorithm. The total number of operations used so far is $O(n)$ and the corresponding time is $O(\log \log \log n)$. Now we use the doubly logarithmic-depth tree based on the n' generated elements in the previous step. Then it requires $O(\log \log n') = O(\log \log n)$ time and uses $O(n' \log \log n') = O(n)$ operations. Therefore, the overall time is $O(\log \log n)$ and the total number of operations is $O(n)$. The logarithmic binary tree algorithm for finding the maximum is same as that of computing the sum of n elements using balanced binary tree, described earlier (in the 1st example).

The doubly logarithmic-depth tree is described here. Suppose there is a rooted tree, the level of a node u is the number of edges on the path between u and the root of the tree. Hence the level of the root is 0. Let there be n leaves in the tree. The root of the tree has \sqrt{n} children. If $n = 2^{2^k}$, then $\sqrt{n} = 2^{2^{k-1}}$, then each children of the root has $2^{2^{k-2}}$ children, and in general, each node at the i th level has $2^{2^{k-i-1}}$ children, for $0 \leq i \leq k-1$. Each node at level k will have two leaves as children.

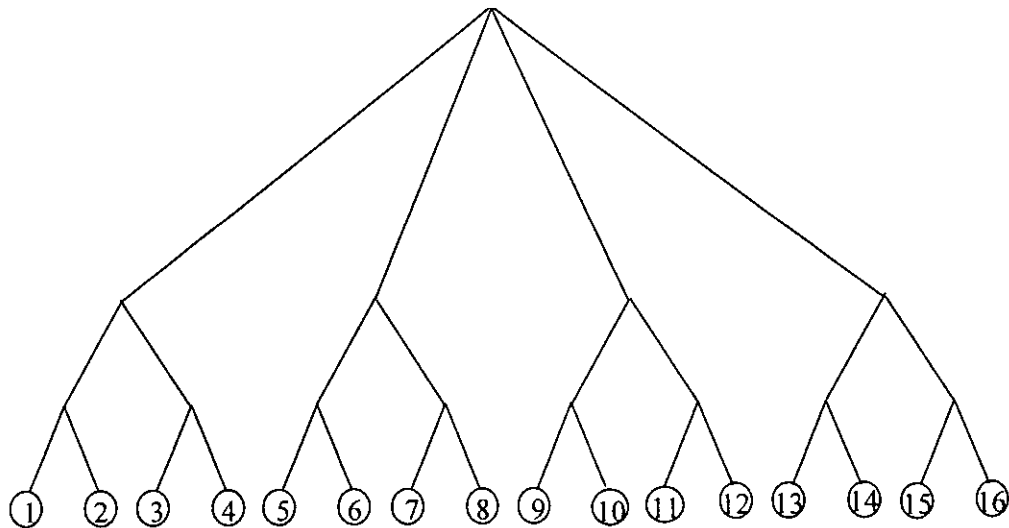


Fig 2.5: Doubly logarithmic depth tree of 16 nodes

Figure 2.5 shows the doubly logarithmic depth tree for the case when the number of items is 16. The root has four children and each of the other internal nodes has

two children. Each internal node corresponds to computing the maximum of that node's children. The number of nodes at the i th level of the doubly logarithmic-depth tree is $2^k - 2^{k-i}$, for $0 \leq i < k$. The number of nodes at the k th level is $2^{2^k-1} = n/2$. The depth of this tree is $k + 1 = \log \log n + 1$. The maxima required at any given level can be computed in $O(1)$ time using $O(p^2)$ operations for p distinct elements. Then the number of operations required at the i th level is $O\left(\left(2^{2^{k-i-1}}\right)^2\right)$ per node, for $0 \leq i \leq k$, giving a total of $O\left(\left(2^{2^{k-i-1}}\right)^2 \cdot 2^{2^k-2^{k-i}}\right) = O\left(2^{2^k}\right) = O(n)$ operations per level. Hence the total number of operations required by the overall computation is $O(n \log \log n)$.

- **Parallel Sorting Algorithm (Pipelined Merge Sort):** A pipelined merge sort algorithm presented in [20] is described as follows:

Sorting of n general numbers can be done in $O(\log n)$ time using n CREW PRAM processors by pipelined merge-sort algorithm. The work done of this algorithm is $O(n \log n)$ and it is work-optimal. It consists of determining $L[v]$ ($L[v]$ is a sorted list that contains all the numbers stored in the subtree rooted at v) over a number of stages such that, at stage s , $L_s[v]$ is an approximation of $L[v]$ that will be improved at the next stage $s+1$. At the same time, a *sample* of $L_s[v]$ is propagated upward to be used for obtaining approximations of the lists to be generated at higher heights. Let $L_0[v] = 0$ if v is an internal node; otherwise $L_0[v]$ consists of the item stored at the leaf v of a binary tree. Let the altitude of a node v be defined as $alt(v) = h(T) - level(v)$, where $h(T)$ is the height of T , and $level(v)$ is the length of the path from the root to v . The list stored at node v will be updated over the stages s satisfying $alt(v) \leq s \leq 3alt(v)$. In this algorithm v is active during stage s if $alt(v) \leq s \leq 3alt(v)$. The algorithm will update the list $L_s[v]$ such that node v will be full (i.e., $L_s[v] = L[v]$), when $s \geq 3alt(v)$.

Let u and w be the children of an internal node v and let $L'_{s+1}[u] = Sample(L_s[u])$ and $L'_{s+1}[w] = Sample(L_s[w])$, where $Sample(L_s[x])$ for an arbitrary node x is defined as:

$$Sample(L_s[x]) = \begin{cases} sample_4(L_s[x]) & \text{if } s \leq 3alt(x) \\ sample_2(L_s[x]) & \text{if } s = 3alt(x) + 1 \\ sample_1(L_s[x]) & \text{if } s = 3alt(x) + 2 \end{cases}$$

Therefore, $Sample(L_s[x])$ is the sublist consisting of every fourth element of $L_s[x]$ until it becomes full; then $Sample(L_s[x])$ is every other element in the following stage (that is, stage $3alt(x)+1$), and every element in stage $3alt(x)+2$. Then $L'_{s+1}[u]$ and $L'_{s+1}[w]$ are merged into a sorted list $L_{s+1}[v]$ and this can be done in $O(1)$ time [20].

Let T be the binary tree in Fig 2.6 where leaf nodes contain 7, 8, 6, 1, 5, 3, 4, 10, 9, 15, 2. The lists corresponding to asset of selected stages are shown in the Table 2.1.

Initially, there is no changes occur until stage $s = 3$. At the end of stage $s = 3$, all the nodes of altitude 1 become full. Consider, for example, node v_5 . Since $alt(v_5) = 1$, v_5 is active during stage 3. In this case, $L_3[v_1] = sample_1(L_2[v_1]) = (7)$ and similarly $L_3[v_2] = (8)$. Hence, $L_3[v_5] = (7, 8)$. During this stage, we also obtain $L_3[v_6] = (1, 6)$.

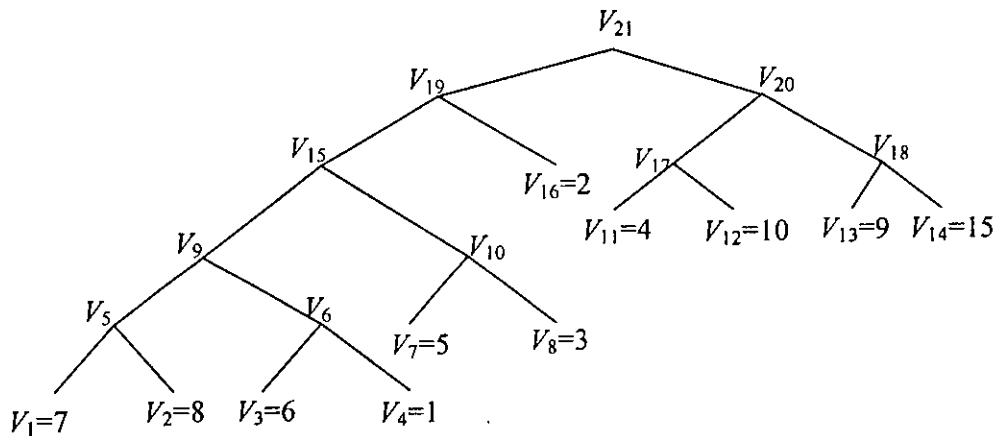


Figure 2.6: A binary tree T for pipelined merge-sort algorithm

At the end of stage $s = 6$, the nodes at altitude 2 become full, at the end of stage $s = 9$, the nodes at altitude 3 become full. The root v_{21} is active for all stages $5 \leq s \leq 15$. However $L_s[v_{21}]$ remains empty until stage $s = 13$ since, at each of the previous stages, the lists of the children nodes v_{19} and v_{20} contain less than four elements. At the end of stage $s = 12$, nodes v_{19} and v_{20} become full and each contain at least

four elements. Hence at stage $s = 13$, $L_s[v_{21}] = (5, 15)$, which results from the merging of $sample_4(L[v_{19}])$ and $sample_4(L[v_{20}])$. At the end of stage $s = 15$, $(L_{15}[v_{21}])$ consists of the sorted list of all the items stored in the tree.

Table 2.1: The lists arising during the execution of the indicated stages of the pipelined merge-sort algorithm

V	$s=0$	$s=3$	$s=5$	$s=6$	$s=8$	$s=9$	$s=11$	$s=13$	$s=15$
1	(7)	(7)	(7)	(7)	(7)	(7)	(7)	(7)	(7)
2	(8)	(8)	(8)	(8)	(8)	(8)	(8)	(8)	(8)
3	(6)	(6)	(6)	(6)	(6)	(6)	(6)	(6)	(6)
4	(1)	(1)	(1)	(1)	(1)	(1)	(1)	(1)	(1)
5	0	(7,8)	(7,8)	(7,8)	(7,8)	(7,8)	(7,8)	(7,8)	(7,8)
6	0	(1,6)	(1,6)	(1,6)	(1,6)	(1,6)	(1,6)	(1,6)	(1,6)
7	(5)	(5)	(5)	(5)	(5)	(5)	(5)	(5)	(5)
8	(3)	(3)	(3)	(3)	(3)	(3)	(3)	(3)	(3)
9	0	0	(6,8)	(1,6,7,8)	(1,6,7,8)	(1,6,7,8)	(1,6,7,8)	(1,6,7,8)	(1,6,7,8)
10	0	0	0	(3,5)	(3,5)	(3,5)	(3,5)	(3,5)	(3,5)
11	(4)	(4)	(4)	(4)	(4)	(4)	(4)	(4)	(4)
12	(10)	(10)	(10)	(10)	(10)	(10)	(10)	(10)	(10)
13	(9)	(9)	(9)	(9)	(9)	(9)	(9)	(9)	(9)
14	(15)	(15)	(15)	(15)	(15)	(15)	(15)	(15)	(15)
15	0	0	0	0	(5,6,8)	(1,3,5,6,7,8)	(1,3,5,6,7,8)	(1,3,5,6,7,8)	(1,3,5,6,7,8)
16	(2)	(2)	(2)	(2)	(2)	(2)	(2)	(2)	(2)
17	0	0	0	0	0	(4,10)	(4,10)	(4,10)	(4,10)
18	0	0	0	0	0	(9,15)	(9,15)	(9,15)	(9,15)
19	0	0	0	0	0	0	(3,6,8)	(1,2,3,5,6,7,8)	(1,2,3,5,6,7,8)
20	0	0	0	0	0	0	(10,15)	(4,9,10,15)	(4,9,10,15)
21	0	0	0	0	0	0	0	(5,15)	(1,2,3,4,5,6,7,8,9,10,15)

2.4 Related Works on Parallel Algorithm for Knapsack Type Problems

Knapsack Problems, since they are the simplest NP-hard problems have been subject to much work on the development of efficient parallel algorithms. With the advent of

parallel processors many researchers have concentrated their efforts on the development of efficient parallel algorithms for solving Knapsack Problems. For exact methods Branch and Bound (B&B) and Dynamic Programming (DP) are the most useful approaches [5]. For serial machines, it is accepted that B&B has better performance than DP for KP, but this observation has not been shown to translate to the parallel case. The B&B method usually implemented on powerful large-grain multiprocessors but involves complicated communication strategies. The communications issues are often hard to solve and an anomalous behaviour of B&B is observed. Even if these problems were solved it is shown that there exist hard knapsack problems for which the number of alternate solutions grows exponentially with problem parameters. Such growth makes the problem hard for B&B algorithms. However the use of parallel DP to solve KP in these cases is possible. In addition DP algorithms for Knapsack Type Problems (KTP) are suitable candidates offering the possibility of further hardware acceleration [5].

Most of the recent work is related to the design of dynamic programming algorithms for Unbounded Knapsack Problems (UKP) and 0/1 KP. For example, A DP algorithm for 0-1 KP which may run on any number of processors available was presented by Lin [21]. Its time complexity is $O(nc/p)$ on EREW PRAM of p processors, where c is the capacity of the knapsack. Lee [22] proposed a hypercube implementation of the DP approach is presented. The running time is $O(nc/p+c^2+c\log(p))$. They also applied the same algorithm for the two-dimensional 0-1 knapsack problem.

A divide and conquer approximation algorithm on a hypercube with a time complexity $O(\log^2(n) \log(c))$ on $O(nc\sqrt{c})$ processors is presented by Gopalakrishnan [23]. Another approximation algorithm realized on the hypercube architecture described by Mayr [24] with a time complexity $O(\log^2(n) \log(c))$ on $O(nc^2)$ processors. A pipeline-architecture containing a linear array of p processors is proposed by Chen [25]. This architecture allows one to achieve an optimal speedup of the KP algorithm with the time complexity of $O(nc/p + n)$ and an efficiency $O(1/(1+1/pc))$ which approaches $O(1)$ as capacity, c increases. Teng [26] proposed an algorithm derived by transforming it to the well solved circuit value problem in $O(\log^2(nc))$ time using $N(c)$ processors, where $N(c)$ is the number of processors needed for multiplying two c by c matrices. A parallel convolutive algorithm for the unbound KP is presented by Morales [27] in $O(c^2/p+n)$ time using p processors.

An efficient parallel algorithm for solving the knapsack problem on the hypercube proposed by Goldman [28]. He proposed a scheduling algorithm for irregular meshes on the hypercube. The efficiency of the algorithm is independent on the number of processors. A parallel tabu search algorithm for the 0-1 multidimensional knapsack problem is presented by Niar [29]. He proposed a new parallel meta-heuristic algorithm based on the tabu search for the resolution for the 0-1 multidimensional knapsack problem that reduces the execution time.

Parallel skeletons for tabu search method has been proposed by Blesa *et al.* [30, 31] for 0-1 multidimensional knapsack problem. Tabu search can be parallelized in several ways for a complete taxonomy of parallel tabu search heuristics. Two parallel implementation based on two different parallel models are presented here. The first implementation, namely, the direct parallelization is based on independent runs model with search strategies. The second implementation is based on the master slave model in which the neighborhood exploration is done in parallel by the slaves and each slave exploring a part of the neighborhood.

A multiprocessor based heuristic is proposed by Shahriar [32, 33] for multiple-choice multi-dimension knapsack problem. The work done of this heuristic is same as that of M-HEU. The time complexity of this heuristic is $O(T/p + f(p))$, where T is the time required by the algorithm using single processor, p is the number of processors and $f(p)$ is the synchronization overhead. The time requirement is roughly inversely proportional to the number of processors used for the computations. In this heuristic multiple processes can be run independently by the operating systems. If the processes are executed in a single processor machine then the operating system gives the illusion of parallelism (pseudo parallelism) by fast switching from one process to another. But if the machine has multiple processors then asynchronous parallelism can be achieved by running each process on a different processor as long as number of processes is less than the number of processors. Time requirement of an algorithm can be reduced greatly if jobs can be divided among processes that run concurrently in a multi-processor machine. If computations are divided among a number of p' processes, where p' is less than the number of processors p then turnaround time for the job is roughly divided by p' plus some overhead due to synchronization and inter-process communication. In case of M-HEU synchronization is required in each

iteration. So, in the long run this overhead may sum up to a significant amount. In this heuristic algorithm, the groups of MMKP are divided among multiple processes for computations.

There is a substantial number of sequential heuristics for MMKP problems in the literature. The parallel algorithms proposed so far are for different variants of knapsack problems other than MMKP. Unfortunately there is no parallel algorithm already proposed for MMKP. The proposed parallel algorithms for KP variants do not provide solutions in polylog time with polynomial number of processors. Finding polylog algorithms for MMKP problems is still an unexplored interesting research area. Thus our research concentrates on finding polylog algorithms by parallelizing existing heuristics of MMKP using PRAM model.

CHAPTER 3

Proposed Algorithms for MMKP

In this chapter we introduce a parallel heuristic algorithm for solving MMKP in polylog time. We find that it is not possible to find a parallel version of M-HEU in polylog time. A new sequential heuristic algorithm is proposed here by modifying M-HEU that would be parallelized. Later a parallel heuristic algorithm is introduced, which is the parallel version of the new sequential algorithm.

3.1 Why M-HEU was Chosen for Parallelism

When a parallel algorithm is developed for any problem in PRAM model, the problem can be solved in polylog time with a polynomial number of processors. As MMKP is an NP-Hard problem, its exact solution may not be feasible in any time-critical problem, since the exact solution of MMKP has an exponential time complexity. In the quest to develop efficient algorithms, no one has been able to develop a polynomial time algorithm for any NP-Hard problem. So if the exact algorithm of MMKP is going to be parallelized, the number of processors will increase exponentially with the increase in number of MMKP dataset and it is not possible to provide a parallel version of an exact algorithm for MMKP in polylog time with a polynomial number of processors. From this point of view, an exact algorithm has not been chosen for parallelism; rather a heuristic based algorithm is chosen for parallelism to solve the MMKP. There are different heuristic algorithms to obtain approximate solution to the MMKP. But M-HEU achieves the maximum optimality among these existing heuristic approaches. Again, the time complexity of M-HEU is polynomial, so that it is possible to provide a parallel algorithm of M-HEU using a polynomial number of processors. So M-HEU was chosen initially for parallelism in our research. Parallelism using M-HEU is definitely worthy as it takes non real time when the problem size gets larger.

3.2 Problem of M-HEU to Be Parallelized

In M-HEU, an item in the current solution is replaced by another item of the same group, with the highest positive Δa_{ij} (the change in aggregate resource consumption),

subject to the resource constraint. If no such item is found then an item with the highest $\Delta v_{ij}/\Delta a_{ij}$ (maximum value gain per unit aggregate resource expended) is chosen. After each feasible upgrade, the change in aggregate consumed resource, Δa_{ij} or the maximum value gain per unit aggregate resource expended, $(\Delta v_{ij})/(\Delta a_{ij})$ has been calculated for each higher-valued item for the next iteration. Actually the result of one iteration is the input to the next iteration. But in our algorithm, to achieve a polylog time complexity, at best logarithmic number of iterations are permitted and in each iteration items from different groups to be replaced in parallel with the items in their respective groups. But it is not possible to upgrade more than one item simultaneously in one iteration in M-HEU. So it is not possible to provide a polylog time algorithm from M-HEU directly. Thus to provide a polylog time parallel algorithm of M-HEU, we have to modify the M-HEU to some extent.

3.3 A New Sequential Heuristic, MS-HEU

3.3.1 The Main Principle

In this heuristic one or more new items are selected in each iteration, so that the total number of iterations is decreased significantly. That is, we have multiple upgrades in each iteration. Evaluation (i.e., the relative change of aggregate resource consumption, $\Delta a'_{ij}$) of every candidate item is done once in every iteration for multiple upgrades but the method of evaluation is exactly the same as M-HEU. M-HEU requires one evaluation of every item for each upgrade. Actually, when a single item is selected, the remaining evaluations are no more perfect. As we are ignoring this in this new heuristic, this approach will lose some revenue but provide a faster solution. We call this as heuristic using multiple selections per iteration, abbreviated as MS-HEU. Here M stands for Multiple and S stands for Selection.

3.3.2 The Process of Upgradation

Exactly one item is chosen with the maximum value of $\Delta a'_{ij}$ from each group. The items chosen from different groups are sorted in descending order according to their value of $\Delta a'_{ij}$. First few items from the sorted list will be selected for upgradation. These items will be upgraded one by one if the resources are available, i.e., without violating the resource constraints. The number of total iterations and the number of upgrades in the iterations is fixed by the following strategies:

Strategy 1: Finding solution with a fixed number of iterations and fixed number of total proposed upgrades.

Strategy 2: Finding solution with a fixed number of iterations and fixed number of proposed upgrades in each iteration.

Strategy 3: Finding solution with iterations until no further actual upgrades available.

In M-HEU, the maximum number of iterations is nl in the worst case, so that we consider $\log nl$ iterations in Strategy 1 and 2 of MS-HEU and the number of items to be upgraded in h th iteration is $nl/2^h$.

In Strategy 1, we call the number of items we want to replace as ‘scheduled upgrade’. Essentially scheduled upgrade = $nl/2^h$. If scheduled upgrade is greater than n (i.e. the number of groups), we cannot take them all and hence we have to consider only n items in such cases. We call the total number of items we tentatively calculate for possible upgrade as ‘tentative upgrade’ and the difference between tentative upgrade and scheduled upgrade is defined pending upgrade. To increase the performance, we add the pending upgrade with the scheduled upgrade and it is the minimum of this summation and n , which is ultimately considered for possible upgrade. So, in effect, in every iteration we have tentative upgrade = $\min(\text{pending upgrade} + \text{scheduled upgrade}, n)$. But in Strategy 2 and 3, the pending upgrade is not considered. In Strategy 2 we also want to upgrade our current solution by replacing $nl/2^h$ items with larger valued items in Iteration h . But when the scheduled upgrade is greater than n , only n items is considered in such cases. In Strategy 3, we want to upgrade the current solution with iterations until no further actual upgrade is available.

3.3.3 Steps of MS-HEU

Step 1: In each group, add one dummy item with value 0 and construct the initial solution with the lowest valued items.

Step 2: In Iteration h (depending on the different strategies), find the feasible upgrades.

Step 2.1: Compute the relative aggregate resource consumption, $\Delta a'_{ij}$ of each item having a higher utility value than the selected item from the same group.

Step 2.2: Find the item with the maximum value of $\Delta a'_{ij}$ from each group.

Step 2.3: Sort the items found in Step 2.2 in descending order, with respect to the value of $\Delta d'_{ij}$ of the items.

Step 2.4: One or more items found in Step 2.3 (different number of items for different strategies) have been tried to upgrade one by one, without violating the constraint.

Step 3: Deliver the solution, if there is no dummy item in the solution. If there is a dummy item in the final solution, it implies that 'no solution is found'. Also the introduction of a dummy item can be used as a special case of MMKP where the restriction of picking exactly one item from each group is relaxed and it indicates that no item will be taken from the corresponding group.

3.3.4 Description of MS-HEU for Upgrading n Items of an MMKP in an Arbitrary Iteration in Step 2

Following are the variables and the procedures to describe the steps of the algorithm.

n : The total number of groups in the MMKP.

m : The total number of resources in the MMKP.

l_i : The number of items in the i th group.

r_{ijk} : The k th resource requirement of the j th item of the i th group.

v_{ij} : The value of the j th item of the i th group.

C_k : The amount of k th resource consumed by the selected items of the groups.

R_k : The total amount of the k th resource in the MMKP.

$\rho[i]$: The index of the currently selected item of the i th group.

current_solution: The solution vector containing the indexes of the current selected items from each group.

candidate_item: The vector containing the items of groups selected by the procedure.

candidate_group: The vector containing the groups selected by the procedure.

find_candidate_item (i): finds the candidate item of Group i with the highest $\Delta d'_{ij}$ among the higher valued items than the selected item. There might be no such item if the highest valued item is already selected.

change_selection (i, j): $\rho[i] \rightarrow j$ and returns the increase of total value for this selection. Positive increase denotes upgrade.

current_resource_usage: The resource consumption of the current selected items from each group.

additional_resource (*candidate_item*, *i*): It determines additional resource requirement if *i*th item of *candidate_item* is selected instead of the currently selected item of the corresponding group.

Procedure multiple_seleccion (*num_items_to_select*)

//This procedure finds a feasible solution of the MMKP by upgrading one or more
//items selected from different groups

1. *candidate_item* = null
2. for *i*=1 to *n* do
3. *candidate_item* ← *candidate_item* + *find_candidate_item* (*i*)
4. endfor
5. *proposed_candidates* ← *sort_nondecreasing* (*candidate_item*)
//sorting the *candidate_item* in nondecreasing order with respect to the value
//of Δd_{ij} using merge sort algorithm
6. return *do_feasible_upgrade* (*proposed_candidates*, *num_items_to_select*)
7. end procedure

Procedure do_feasible_upgrade (*proposed_candidates*, *num_items_to_select*)

//This procedure does feasible upgrades as many as possible among
//*num_items_to_select* from the beginning of *proposed_candidates*. At first it tries
//whether all *num_items_to_select* can be upgraded or not. If it is not feasible to
//upgrade all, then feasibility is searched by ignoring the last item in the
//*num_items_to_select* in the subsequent iterations. If there is any feasible upgrade, it
//returns true.

1. *used_resource* = *current_resource_usage*
2. for (*i* = 1 to *num_items_to_select*) do
3. *used_resource* = *used_resource* + *additional_resource*
 (*proposed_candidates*, *i*)
 // This indicates a summation of vectors indicating *k* dimensional resource
4. endfor
5. for (*i* = *num_items_to_select* to 1) do
6. if (*used_resource* ≤ *total_resource*) then
 // All *k* resource constraints are being checked

```

7.         for (j = 1 to i) do
8.             change_selection (candidate_group, candidate_item)
                //Upgrading the selected feasible items
9.         endfor
10.        return true
11.    else
12.        used_resource = used_resource - additional_resource
                (proposed_candidates, i)
                // This indicates a subtraction of vectors indicating k dimensional resource
13.    endif
14. endfor
15.end procedure

```

Complexity analysis of an iteration in Step 2 of MS-HEU

In Step 2.1, m additions, m subtractions, m multiplications and 1 division are needed to calculate the relative change of aggregate resource consumption, $\Delta a'_{ij}$. The value of $\Delta a'_{ij}$ is calculated for every item in each group, so Step 2.1 needs $O(lmn)$ operations.. Step 2.2 directly employs the maximum finding algorithm on l items in each group. And to find the item with the highest $\Delta a'_{ij}$ from a group, the algorithm needs $O(l)$ operations. Since there are n groups in the MMKP, total number of operations is $O(nl)$. We apply the merge sort algorithm for sorting the items selected in the previous step, in descending order with respect to their value of $\Delta a'_{ij}$, requires $O(n \log n)$ operations. In Step 2.4 the maximum number of items to be upgraded is n and so the step needs $O(n)$ operations.

So the overall complexity of an iteration of Step 2 of the algorithm is $O(nlm + n \log n) = O(n(lm + \log n))$.

3.3.5 Description of MS-HEU

The variables and the procedures to describe the steps of the algorithm are same as described in the previous algorithm. Two new variables are described below.

act_upgrade: The number of items that will be upgraded satisfying the resource constraints.

tentative_upgrade: The total number of items we tentatively calculate for possible upgrade.

pending_upgrade: The difference between the number of items that we want to upgrade and the number of items of *tentative_upgrade*.

Algorithm MS-HEU (Strategy 1)

Procedure MS-HEU_Strategy 1()

//Description of Strategy 1 of MS-HEU

1. *current_solution* \leftarrow *initial_solution* ()
2. *num_items_to_select* = $n \times \max(l_i)$
3. *pending_upgrade* = 0
4. *do*
5. *if* (*pending_upgrade* + *num_items_to_select*/2) > *n*
6. *tentative_upgrade* = *n*
7. *else*
8. *tentative_upgrade* = *pending_upgrade* + *num_items_to_select*/2
9. *endif*
10. *multiple_selection*(*tentative_upgrade*)
11. *if* (*num_items_to_select* \neq 1)
12. *num_items_to_select* = *num_items_to_select*/2
13. *endif*
14. *pending_upgrade* = *pending_upgrade* + (*num_items_to_select* - *tentative_upgrade*)
15. *while* (*num_items_to_select* > 1)
16. *end procedure*

Procedure initial_solution ()

//This procedure add one dummy item in each group and construct the initial solution
//with these dummy items.

1. *for candidate_groups* *i*=1 to *n*
2. *add item with value 0 and constraint 0*
3. *endfor*
4. *end procedure*

Algorithm MS-HEU (Strategy 2)
Procedure MS-HEU_Strategy 2()

//Description of Strategy 2 of MS-HEU

1. *current_solution* ← *initial_solution* ()
2. *num_items_to_select* = $n \times \max(l_i)$
3. *do*
4. *if* (*num_items_to_select*/2) > *n*
5. *tentative_upgrade* = *n*
6. *else*
7. *tentative_upgrade* = *num_items_to_select*/2
8. *endif*
9. *multiple_selection*(*tentative_upgrade*)
10. *if* (*num_items_to_select* ≠ 1)
11. *num_items_to_select* = *num_items_to_select*/2
12. *endif*
13. *while* (*num_items_to_select* > 1)
14. *end procedure*

Algorithm MS-HEU (Strategy 3)
Procedure MS-HEU_Strategy 3()

//Description of Strategy 3 of MS-HEU

1. *current_solution* ← *initial_solution* ()
2. *num_items_to_select* = $n \times \max(l_i)$
3. *sel_success* = 1
4. *do*
5. *if* (*num_items_to_select*/2) > *n*
6. *tentative_upgrade* = *n*
7. *else*
8. *tentative_upgrade* = *num_items_to_select*/2
9. *endif*
10. *sel_success* = *multiple_selection*(*tentative_upgrade*)
11. *if* (*num_items_to_select* ≠ 1)
12. *num_items_to_select* = *num_items_to_select*/2

13. *endif*
14. *while (num_items_to_select >= 1 && sel_success == 1)*
15. *end procedure*

Complexity analysis of different strategies of MS-HEU

Step 1 needs $O(n)$ operations.

Step 2 iterates $\log nl$ times in Strategy 1. So the total number of operation in Step 2 is $O(\log nl (lmn + n \log n)) = O(n \log nl (lm + \log n))$ for Strategy 1. And the overall complexity of this strategy is $O(n \log nl (lm + \log n))$.

Step 2 iterates also $\log nl$ times in Strategy 2. So the overall complexity of this strategy is also $O(n \log nl (lm + \log n))$.

Step 2 iterates nl times in the worst case in Strategy 3. So the overall complexity of Strategy 3 is $O(nl (lm + \log n))$.

3.3.6 Some Arguments Regarding MS-HEU

Why null values are introduced in MS-HEU?

A dummy item with null values is introduced in Step 1; it gives always a feasible solution since the dummy item does not consume any resource. But if any dummy item does exist in the final solution, it implies that there is no feasible solution. However, in some practical problems these null values bear significant roles in decision making. For example, the problem of admission control can be easily mapped to the MMKP. In this case, a dummy item in the final solution indicates the rejection of a particular session in the admission control problem. Here, the session is equivalent to a group and the QoS of a user's requested session is equivalent to an item as described in Section 1.2. But in the cases where null values do not indicate any significant meaning, our proposed algorithm does not give a feasible solution though there is a feasible solution using M-HEU. In M-HEU, if the initial solution is not feasible, then a feasible solution is found by searching new items with better revenue with less resource consumptions. But in this searching technique, the solution of one iteration is dependent to the solution of previous iteration, so that it is not possible to provide a parallel version for this part of MMKP solution in polylog time

complexity. We can ignore this, because there are very good number of applications in practical problems, where null values play important roles.

Why sorting the items with respect to the value associated with them?

The lowest valued items are selected as the initial solution by sorting. Sorting is used to reduce the search space. In the proposed algorithm, we have to find the higher valued items than the selected items for upgrading the current solution. So if the items are not sorted, same computation is done for some lower valued items, which is undesirable, as it consumes extra time and space.

Why not starting from the highest valued items?

There could be another approach to start with the items with the highest value from each group, and then iteratively select lower valued items until feasibility is achieved. The chance of having feasible solution with the highest valued items is very low as it is expected in almost all cases that those items will consume more resources. The next step would be to bring the solution to feasible solution. But it is not possible to parallelize this approach in polylog iteration, because in this approach, every iteration is fully dependent to its previous iteration and it might require nl iterations in the worst case.

3.3.7 Example for Demonstrating Strategy 1 of MS-HEU

Since the pending upgrade is considered in Strategy 1, it gives better result than other strategies and a parallel algorithm is proposed in the next section using this strategy. So that Strategy 1 is demonstrating here. The other strategies differ from Strategy 1 only in the number of upgrades. The upgradation process using other strategies is similar to this. Thus it is expected that the other strategies could be understood easily from this demonstration of Strategy 1.

Figure 3.1 shows an MMKP with 4 groups. Each group has 3 items sorted according to the value associated with each item. The resource is two-dimensional.

Demonstrating Step 1:

A dummy item with zero value is added in each group and these dummy items are selected as the initial solution. Thus the solution vector can be written as (0, 0, 0, 0) where i th element of the vector is the index of the selected item of the i th group. The resource usage vector for this solution is (0, 0) where i th element denotes the i th resource consumption.

Item 3	$v=24$ $r_1=7, r_2=3$	$v=36$ $r_1=9, r_2=7$	$v=23$ $r_1=2, r_2=6$	$v=34$ $r_1=3, r_2=5$	Maximum allowable resource $R_1 : 22$ $R_2 : 20$
Item 2	$v=14$ $r_1=7, r_2=0$	$v=30$ $r_1=7, r_2=6$	$v=19$ $r_1=3, r_2=1$	$v=25$ $r_1=6, r_2=8$	
Item 1	$v=12$ $r_1=4, r_2=5$	$v=29$ $r_1=7, r_2=3$	$v=16$ $r_1=2, r_2=1$	$v=19$ $r_1=2, r_2=7$	
	Group 1	Group 2	Group 3	Group 4	Knapsack

Figure3.1: Example of an MMKP

Demonstrating Step 2 (Feasible upgrades):

Iteration 1:

Values of Δa_{ij}^r for all the feasible upgrades from the currently selected items are as follows. The currently selected items are the dummy items. The value of Δa_{ij}^r of the higher valued items can be calculated using $\sum_k (r_{i\rho[i]k} - r_{ijk})$.

$$\begin{array}{lll}
 \Delta a_{11}^r = \{1.33, 0\} & \Delta a_{12}^r = \{2, 0\} & \Delta a_{13}^r = \{2.4, 0\} \\
 \Delta a_{21}^r = \{2.9, 0\} & \Delta a_{22}^r = \{2.31, 0\} & \Delta a_{23}^r = \{2.25, 0\} \\
 \Delta a_{31}^r = \{5.33, 0\} & \Delta a_{32}^r = \{4.75, 0\} & \Delta a_{33}^r = \{2.88, 0\} \\
 \Delta a_{41}^r = \{2.11, 0\} & \Delta a_{42}^r = \{1.79, 0\} & \Delta a_{43}^r = \{4.25, 0\}
 \end{array}$$

The computation of Δa_{11}^r can be shown as follows:

$$\begin{aligned}
\Delta a_{11} &= \sum_{k=1}^2 (r_{1\rho[1]k} - r_{11k}) \\
&= (r_{101} - r_{111}) + (r_{102} - r_{112}) \\
&= (0 - 4) + (0 - 5) \\
&= -4 - 5 = -9
\end{aligned}$$

Since $\Delta a_{11} < 0$, $y_{ij} = 0$ and $\Delta v_{11}/\Delta a_{11}$ have to be calculated where

$$\begin{aligned}
\Delta v_{11} &= \Delta v_{1\rho[1]} - \Delta v_{11} \\
&= 0 - 12 \\
&= -12
\end{aligned}$$

So $\Delta v_{11}/\Delta a_{11} = -12/-9 = 1.33$ and hence $\Delta a^r_{11} = \{1.33, 0\}$.

There are four groups containing three items each. So in the first iteration, we want to upgrade min (6, 4) groups simultaneously. So the number of pending upgrades is 2 in this iteration. The items with the highest Δa^r_{ij} from each group are selected. These items are sorted in non descending order with respect to the value of Δa^r_{ij} . Here we get Item 1 from Group 2 and 3, and Item 3 from Group 1 and 4 with the highest Δa^r_{ij} from their respective groups. If four groups are upgraded, then the total resource usages are 19 and 12 respectively and this is feasible. So Group 2, 3 are upgraded to Item 1 and Group 1, 4 are upgraded to Item 3. We get current solution (3, 1, 1, 3) with resource usage (19, 12).

Iteration 2:

$$\begin{aligned}
\Delta a^r_{22} &= \{0.03, 0\} & \Delta a^r_{23} &= \{0.08, 0\} \\
\Delta a^r_{32} &= \{0.6, 0\} & \Delta a^r_{33} &= \{0.12, 0\}
\end{aligned}$$

The computation of Δa^r_{12} can be shown as follows:

$$\begin{aligned}
\Delta a_{22} &= \sum_{k=1}^2 (r_{2\rho[1]k} - r_{212k}) \times C_k \\
&= (r_{211} - r_{221}) \times C_1 + (r_{212} - r_{2122}) \times C_2 \\
&= (7 - 7) \times 19 + (3 - 6) \times 12 \\
&= 0 - 36 = -36
\end{aligned}$$

Since $\Delta a_{22} < 0$, $y_{ij} = 0$ and $\Delta v_{22}/\Delta a_{22}$ have to be calculated where

$$\begin{aligned}
\Delta v_{22} &= \Delta v_{2\rho[1]} - \Delta v_{22} \\
&= 29 - 30 \\
&= -1
\end{aligned}$$

So $\Delta v_{22}/\Delta a_{22} = 0.03$ and hence $\Delta a'_{22} = \{0.03, 0\}$.

In iteration 2, we want to upgrade min (3+2, 4) groups. But Group 1 and 4 will not be possible to upgrade in this iteration, since these groups have already upgraded to the highest valued items in the previous iteration. So it is possible to upgrade Group 2 and Group 3 from Item 1 to Item 3 and from Item 1 to Item 2 respectively. We get resource usage (22, 16), so the solution is feasible. And the current solution is (3, 3, 2, 3).

Iteration 3:

$$\Delta a'_{33} = \{0.07, 0\}$$

In this iteration there is only one item that will be upgraded: in Group 3, from Item 2 to Item 3. But if we select Item 3 in Group 3, it does not satisfy the resource constraints. Hence the final solution is (3, 3, 2, 3) and resource usage (22, 16).

3.4 PRAM-HEU: A Heuristic for Solving the MMKP Using PRAM Model

A parallel heuristic algorithm, PRAM-HEU is proposed using Strategy 1 of MS-HEU. In Strategy 1, pending upgrade is considered, i.e. an upgradation which is not feasible in one iteration, is considered in later iterations. But in Strategy 2 and 3, the pending upgrade is not considered. As a result some feasible upgrades are left unconsidered in some iterations in Strategy 2 and 3, and it is shown that Strategy 1 gives better result than that of other strategies. Also there is no polylog time complexity of Strategy 3. So that Strategy 1 of MS-HEU is considered to develop a parallel heuristic algorithm for MMKP.

Finding maximum $\Delta a'_{ij}$: Items of each group in the MMKP are sorted in non-decreasing order according to the value associated with them. First the relative change in aggregate consumed resource, $\Delta a'_{ij}$ of each item are calculated in parallel for each group. Then we find the items with the highest $\Delta a'_{ij}$ from each group in parallel. We start with the binary tree algorithm from the leaves and move until the size of the problem is reduced to a certain value. Then we apply the doubly logarithmic-depth tree based on the items generated in the first step (from the binary tree algorithm). A CRCW PRAM model is used to find the items with the highest $\Delta a'_{ij}$.

Sorting candidate items: The items found in the previous step, are sorted in non-decreasing order with respect to the value of their relative change in aggregate consumed resource using pipelined merge sort algorithm on CREW PRAM model.

Finding feasibility:

Computing prefix sum: In every iteration we have tentative upgrade = min (pending upgrade + scheduled upgrade, n) as Strategy 1 of MS-HEU, described in the previous section. The prefix sum is computed of the resource consumptions of the tentative upgrade items using logarithmic time prefix computation algorithm. The PRAM model to compute this prefix sum is CREW.

Finding feasible upgrades: The selected tentative upgrade items may not give the feasible solution due to the resource constraints. If these items do not give the feasible solution, then we find out the maximum number of items for which the solution will be feasible and these items will be upgraded. The whole procedure runs on Concurrent Read Concurrent Write Parallel Random Access Machine (CRCW PRAM).

The algorithm PRAM-HEU consists of the following steps:

Step 1: In each group add one dummy item with value 0 and constraints 0. Construct the initial solution with the lowest valued item (*i.e.* dummy item) in each group.

Step 2: In Iteration h ($0 \leq h \leq \log nl$), execute the following sub-steps.

Step 2.1: For all group, relative aggregate resource consumption, $\Delta d'_{ij}$ of each item having a higher utility value than the item selected from the same group are computed in parallel.

Step 2.2: Find the item with highest $\Delta d'_{ij}$ in each group using parallel maximum finding algorithm.

Step 2.3. Sort the items found in Step 2.2 in descending order of $\Delta d'_{ij}$ using parallel sorting algorithm.

Step 2.4: Compute prefix sums of consumed resources for the first min (pending upgrade + $nl/2^h$, n) items found in Step 2.3

Step 2.5: Using parallel maximum finding algorithm, find the maximum number of items that can be upgraded at a time in this iteration without violating the constraint and then perform the upgrade.

Step 3: Deliver the solution, if there is no dummy item in the solution. If there is a dummy item in the final solution, it implies that 'no solution is found'. Also the introduction of a dummy item can be used as a special case of MMKP where the restriction of picking exactly one item from each group is relaxed and it indicates that no item will be taken from the corresponding group.

3.4.1 Description of the Algorithm PRAM-HEU

additional_resource_using_prefix_array(i): It determines additional resource requirements of the i items, directly from the prefix sum, if these i items are selected from different groups instead of currently selected items of the corresponding groups.

find_candidate_item (i): It finds the candidate item of Group i with the highest $\Delta d'_{ij}$ among the higher valued items than the current selected item in polylog time (the algorithm is described in Chapter 2). There might be no such item if the highest valued item is already selected.

Procedure PRAM-HEU ()

//This procedure finds a feasible solution of the MMKP by upgrading the items of //different groups in parallel

1. $current_solution \leftarrow initial_solution ()$
2. $num_items_to_select = n \times max(l_i)$
3. $pending_upgrade = 0$
4. *do*
5. *for* $i=1$ to n *do*
6. $candidate_item \leftarrow candidate_item + find_candidate_item(i)$
7. *endfor*
8. $pipelined_merge_sort (candidate_item)$
//It sorts the $candidate_item$ in nondecreasing order with respect to the value
//of $\Delta d'_{ij}$ using pipelined merge sort algorithm. The algorithm is described in
//Chapter 2.
9. *if* $(pending_upgrade + num_items_to_select / 2) > n$
10. $tentative_upgrade = n$
11. *else*

```

12.           tentative_upgrade = pending_upgrade + num_items_to_select/2
13.     endif
14.     prefix_sum (tentative_upgrade)
15.     act_upgrade ← find_max_num_items_upgrade (tentative_upgrade)
16.     do_feasible_upgrade (act_upgrade)
17.     if (num_items_to_select != 1)
18.         num_items_to_select = num_items_to_select/2
19.     endif
20.     pending_upgrade = pending_upgrade + (num_items_to_select -
        tentative_upgrade)
21.   while (num_items_to_select > 1)
22. end procedure

Procedure initial_solution ( )
//This procedure adds one dummy item in each group and construct the initial solution
//with these dummy items
1.   for candidate_group i=1 to n pardo
2.       add item with value 0 and constraint 0
3.   endfor
4. end procedure

Procedure prefix_sum (x)
//This procedure calculates the prefix sum of tentative_upgrade items for each
//resource constraint. Here we find out the prefixes of 1, 2, ..., tentative_upgrade - 1,
//tentative_upgrade items for the kth resource constraint in parallel.
1.   prefix_array = {x}
2.   for resource_constraint k=1 to m pardo
3.       for processor i (i = 1 to x/logx) pardo
4.           computes the prefixes of its logx assigned items (i-1)logx+1, (i-1)
           logx+2, ....., ilogx. let the results be  $Z_{(i-1)\logx+1}$ ,  $Z_{(i-1)\logx+2}$ , ...  $Z_{ilogx}$ 
5.       endfor
6.       for processor i (i = 1 to x/logx) pardo
7.           compute the prefixes of x/logx items  $Z_{\logx}$ ,  $Z_{2\logx}$ , ...,  $Z_x$  found in the
           previous step. let  $w_{\logx}$ ,  $w_{2\logx}$ , ...,  $w_x$  be the result
8.       endfor

```

9. processor 1 outputs $z_1, z_2, \dots, z_{\log x}$
10. for processor i ($i = 2$ to $x/\log x$) pardo
11. computes and outputs $w_{(i-1)\log x + z_{(i-1)\log x + 1}}, w_{(i-1)\log x + z_{(i-1)\log x + 2}}, \dots,$
 $w_{(i-1)\log x + z_{i\log x}}$
12. endfor
13. store the outputs in `prefix_array`
14. endfor
15. return `prefix_array`
16. end procedure

Procedure find_max_num_items_upgrade (tentative_upgrade)

//This procedure finds the number of possible feasible upgrades. At first it finds
 //whether all *tentative_upgrade* items can give feasible upgrade or not. If it is not
 //feasible to upgrade all, then feasibility is searched by ignoring the last item in the
 //*tentative_upgrade* in the subsequent iterations. If there is any feasible upgrade, it
 //returns true.

1. `used_resource = current_resource_usage`
2. `index_array = {tentative_upgrade}`
3. for ($i = 1$ to *tentative_upgrade*) pardo
4. `used_resource = used_resource +`
`additional_resource_using_prefix_array(i)`
5. `if (used_resource \leq total_resource) then`
`// All k resource constraints are being checked in parallel`
6. `write the value of i in the index_array`
7. `else`
8. `write negative number in the index_array`
9. `endif`
10. endfor
11. `if there is no positive number in the index_array`
12. `return false`
13. `else`
14. `employ parallel maximum finding algorithm`
15. `return true`
16. `endif`

17. *endif*
18. *end procedure*

An example of the parallel maximum finding algorithm is given below. The parallel maximum finding algorithm is employed on the indices of the Boolean array giving “Yes” in the Figure 3.2.

Each processor, instead of writing “No”, just writes a negative number to indicate a negative verdict and the array of indices is initialized with the index of the Boolean array. Then the parallel maximum finding algorithm is employed, same as that described to find the candidate item with the highest $\Delta a'_{ij}$ in Procedure *find_candidate_item(i)*. The maximum in the array of indices will indicate the number of *act_upgrade*.

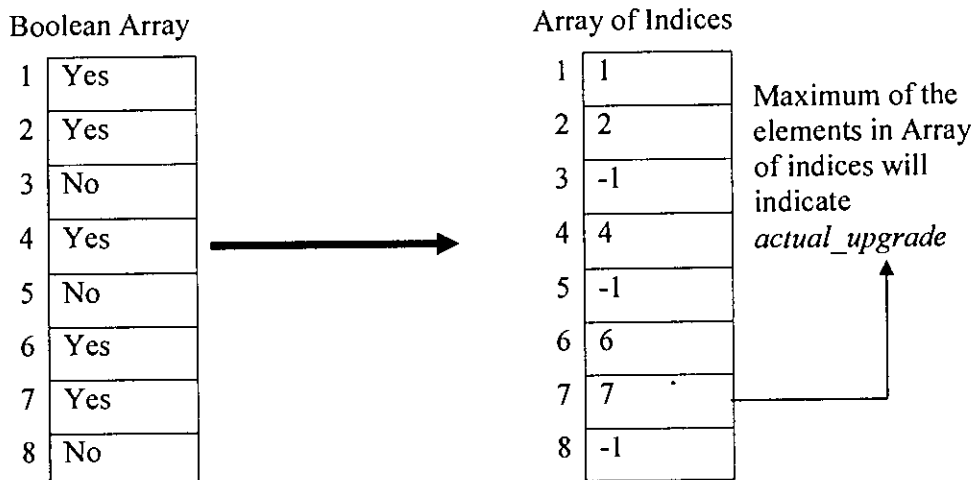


Figure 3.2: Finding actual number of upgrades

Procedure do_feasible_upgrade (act_upgrade)

//This procedure upgrades *act_upgrade* items from different groups in parallel.

1. *for (i = 1 to act_upgrade) pardo*
2. *change_selection (candidate_group, candidate_item)*
3. *endfor*
4. *return true*
5. *end procedure*

3.4.2 Complexity Analysis of Algorithm PRAM-HEU

For the convenience of the analysis we assume all the groups have the same number of items *i.e.* $l_1 = l_2 = l_3 = \dots = l_n = l$.

Step 1 and Step 3 can be performed easily in $O(1)$ parallel time, $O(n)$ operations on EREW PRAM. However, Step 2 with a number of nontrivial sub-steps needs some analysis.

In Step 2.1, m additions, m multiplications and m subtractions are needed. The additions can be done in parallel in $\log m$ time using $O(m)$ operations. For the subtraction, m processors are directly employed and it can be done in $O(1)$ time using $O(m)$ operations. m multiplications can also be done in $O(1)$ time using $O(m)$ processors, *i.e.* total number of operations is $O(m)$. Then the results of m subtractions and m multiplications should be added and it can be added in parallel. Similarly, to calculate the value of C_k , we need n additions, which can be done in parallel in $\log n$ time spending $O(n)$ operations. We have to calculate the relative change of aggregate resource consumption, Δd_{ij}^r for each item (l items) from each group (n groups). So in total Step 2.1 runs in $O(\log m + \log n)$ time using $O(lnm + nm) = O(lnm)$ operations on CREW PRAM.

Step 2.2 directly employs the parallel maximum finding algorithm on l items in each group. Since there are n groups this can be done in $O(\log \log l)$ time with $O(nl)$ operations on CRCW PRAM.

In Step 2.3 we apply the parallel sorting algorithm on n elements which can be done in $O(\log n)$ time, $O(n \log n)$ operations on CREW PRAM.

In Step 2.4 we compute prefix sums of consumed resources for *tentative_upgrade* elements. Computing prefix sum on k elements in parallel takes $O(\log k)$ time using $O(k)$ operations in EREW PRAM. Since the resources are m dimensional we can do the prefix sum separately on different dimensions in $O(\lg(\textit{tentative_upgrade}))$ time using $O(m \times \textit{tentative_upgrade})$ operations. However, to construct the Boolean array indicating whether resource constraints are met or not we have to check all the dimensions in parallel. We can do the job in $O(1)$ parallel time using $O(\textit{tentative_upgrade} \times m)$ operations as follows. We employ one processor each for each of the resource dimensions for each of the *tentative_upgrade* elements. Each processor checks the assigned dimension against the corresponding available resource

dimension. A group of m processors, corresponding to the m resource dimension of a particular element writes to particular entry of the array. The array is initialized with “Yes”.

If any check of any processor turns out to be negative, it just writes “No” in the corresponding entry. So the PRAM model needed is CRCW. Since maximum value of *tentative_upgrade* is n , Step 2.4 can be performed in $O(\log n)$ time using $O(mn)$ operations in CRCW PRAM.

In Step 2.5, we find the maximum number of items, i.e., the value of *actual_upgrade* for which the solution is feasible. In this step we can employ the parallel maximum finding algorithm directly and it will take $O(\log \log n)$ time with $O(n)$ operations on CRCW PRAM, since maximum value of *tentative_upgrade* can be in the worst case n .

Finally performing the upgrade can be done in $O(1)$ parallel time using $O(n)$ operations. The complexity analysis of the steps is summarized in Figure 3.4.

Since the Step 2 iterates for $\log nl$ time, the overall running time of Step 2 should be $O(\log nl(\log m + \log n + \log \log l))$ parallel time. The number of operations needed is $O(n \log nl(\log n + lm))$. It is easy to see that the running time and the total operations of the algorithm remain those of Step 2 and the PRAM model required is CRCW.

Table 3.1: Summary of the complexities of different steps

Step	Time	Operation	PRAM
1	$O(1)$	$O(n)$	EREW
2.1	$O(\log n + \log m)$	$O(lmn)$	CREW
2.2	$O(\log \log l)$	$O(nl)$	CRCW
2.3	$O(\log n)$	$O(n \log n)$	CREW
2.4	$O(\log n)$	$O(mn)$	CRCW
2.5	$O(\log \log n)$	$O(n)$	CRCW
3	$O(1)$	$O(n)$	EREW

CHAPTER 4

Experimental Results

In order to study the performance of PRAM-HEU, we do not simulate the PRAM algorithm actually. The corresponding serial algorithm is used to compare the performance (such as earned revenue) of PRAM-HEU and we compare the results with the value achieved by M-HEU, modified Heuristic for MMKP and the upper bound, a bound which is equal to or higher than the optimal value of the objective function of the MMKP.

We have performed experiments on an extensive set of problem sets. We used randomly generated and correlated MMKP instances for our test cases. The average of the results achieved from multiple MMKP sets are presented in tables and graphs.

4.1 Initializing the Data

We performed experiments on extensive sets of problem set. The MMKP problems were generated using pseudo-random number generators. The data generation procedure that is used here is the same as that was used for generating data for M-HEU [3, 4]. The data sets for testing the performance of the heuristic were initialized as follows:

R_c = Maximum amount of a resource consumption by an item.

P_c = Maximum value per unit resource.

R_k = Total constraint for the k th resource type = $n \times R_c \times 0.5$. Here we assume $R_c \times 0.5$ amount resource on the average for each session.

P_k = Value of the k th resource = $Random(P_c)$ = A uniform discrete random number from 0 to $(P_c - 1)$.

r_{ijk} = The k th resource of the j th item of the i th group = $Random(R_c)$.

v_{ij} = Value of the j th item of the i th group = $Random\left(m \times \frac{R_c}{10} \times \frac{P_c}{10}\right) \times \frac{j+1}{l}$, when the item values are not correlated with the resource requirement.

$v_{ij} = \sum r_{ijk} \times P_k + \text{Random}\left(m \times 3 \times \frac{R_c}{10} \times \frac{P_c}{10}\right)$, when there is a positive correlation between the resource consumption and item values.

For the experimental results reported in this chapter, we used $R_c = 10$ and $P_c = 10$. Please see <ftp://panoramix.univ-paris1.fr/pub/CERMSEM/hifi/MMKP> for some benchmark data sets on the MMKP. Although in our experiments we used larger data sets, but the data generation procedure is the same as that was used for creating benchmark datasets. The performance and the time complexity of MS-HEU have been observed for random and correlated data sets.

4.2 Methods of Experiment

It is not possible to provide a PRAM machine by using normal multiprocessors and normally the PRAM machine is not available, so that we can't get actual performance of the algorithm in the desired environment. MS-HEU gives exactly the same result of the PRAM algorithm. That is why MS-HEU is implemented to determine the total value to be earned by PRAM-HEU. MS-HEU has been implemented for different strategies using the Java programming language and ran the algorithm on a Pentium IV 1.7 GHz with 128 MB of RAM running Windows XP. We also compare the time requirement of MS-HEU using different strategies. These time requirements do not represent the time requirement of PRAM-HEU. For the same data, M-HEU has also been executed. Our solution is then benchmarked with the result of M-HEU.

We can get the exact solution by BBLP technique, but that will take exponential time complexity. In this experiment we compute an upper bound of the value using the same technique but with one iteration only, where an indefinite number of iterations finds the optimal value. The percentages of the value achieved by our algorithms with respect to this upper bound are presented, which is defined as the optimality of the solution in our algorithm.

4.3 Test Results

It is observed that PRAM-HEU achieves on an average 98% of the value of M-HEU and about 94.5% of the optimal solution. We presented the experimental results in the tables and graphs.

Table 4.1, 4.2 and 4.3 show the comparison of the time requirements among different strategies of MS-HEU and M-HEU for correlated and uncorrelated (random) data sets for varying number of groups, number of resource dimensions and number of items in each group respectively.

The graphs of Figure 4.1 to 4.3 show the performance of different strategies of MS-HEU with respect to M-HEU for different number of groups, resource dimensions and items in each group. Similarly the graphs of Figure 4.4 to 4.6 compares the optimality achieved by the M-HEU and different strategies of MS-HEU for different number of groups, resource dimensions and items in each group. The optimality of M-HEU and different strategies of MS-HEU are compared for smaller data sets, because it takes lots of time to calculate the optimality when the data sets are larger, due to the exponential complexity of upper bound finding algorithm. The graphs of Figure 4.7 to 4.12 compare the time required by M-HEU and different strategies of MS-HEU.

All the plotted data in the above mentioned graphs are the average of 10 problem sets. To verify the consistency of the results we present graphs showing Upper-Bound and total values from Strategy 1 of MS-HEU and M-HEU for 10 correlated and uncorrelated data sets in Figure 4.13 and Figure 4.14.

Table 4.1 Time requirements in milliseconds by M-HEU and different strategies of MS-HEU for solving the MMKP with correlated and uncorrelated data sets varying n

N	m	l	Time requirement (in ms) of MS-HEU						Time requirement (in ms) of M-HEU	
			Strategy 1		Strategy 2		Strategy 3		Cor	Uncor
			Cor	Uncor	Cor	Uncor	Cor	Uncor		
500	25	25	1032	451	1182	641	1482	811	67467	46297
1000	25	25	2944	1101	3136	1273	3846	1683	243850	128224
1500	25	25	4850	1803	5339	1984	6059	3465	771729	331136
2000	25	25	5708	2994	6474	3518	10014	5608	912402	563500
2500	25	25	10275	4666	11053	4810	16033	8973	1841048	930518
3000	25	25	15112	5848	16329	7437	22282	11377	2563907	1183332
3500	25	25	18070	7200	19893	8312	29763	12869	2944193	1945537
4000	25	25	24636	10225	25451	12774	42126	18747	3990298	2403366
4500	25	25	32840	12227	31886	14709	44003	23764	6825705	3067321
5000	25	25	33559	14951	34599	16034	46356	25036	9673518	3705088

Table 4.2 Time requirements in milliseconds by M-HEU and different strategies of MS-HEU for solving the MMKP with correlated and uncorrelated data sets varying m

n	m	l	Time requirement (in ms) of New Heuristic						Time requirement (in ms) of M-HEU	
			Strategy 1		Strategy 2		Strategy 3		Cor	Uncor
			Cor	Uncor	Cor	Uncor	Cor	Uncor		
2500	5	25	6870	3525	8989	4031	12263	7521	280062	251702
2500	10	25	7016	3816	9872	4390	13001	7721	667600	352117
2500	15	25	7621	3825	10301	4479	13009	7611	910629	598060
2500	20	25	11256	4276	11952	4853	13520	8021	1595173	748686
2500	25	25	11767	4427	12154	5032	13069	8020	1851331	969344
2500	30	25	13319	4847	14664	6052	14796	8703	2430012	1073493
2500	35	25	13614	5097	15432	6183	14811	9023	3247459	1319017
2500	40	25	12188	5087	15105	6392	16531	9543	3775249	1430076
2500	45	25	14235	5288	15251	7144	18136	9834	3895361	1713424
2500	50	25	14520	5708	15487	7545	19092	10024	4100316	1905040

Table 4.3 Time requirements in milliseconds by M-HEU and different strategies of MS-HEU for solving the MMKP with correlated and uncorrelated data sets varying l

n	m	l	Time requirement (in ms) of New Heuristic						Time requirement (in ms) of M-HEU	
			Strategy 1		Strategy 2		Strategy 3		Cor	Uncor
			Cor	Uncor	Cor	Uncor	Cor	Uncor		
2500	25	5	3275	2124	4376	3047	4196	4957	233626	130968
2500	25	10	4156	2834	4840	3309	8853	6079	636224	316325
2500	25	15	7460	3585	7864	4300	11687	6249	943447	546916
2500	25	20	8182	3705	9348	4641	11687	7611	1277858	724492
2500	25	25	11276	4126	11510	5042	18166	8522	1500368	960421
2500	25	30	13058	4326	13766	5123	17956	8791	1786619	1156813
2500	25	35	16964	5368	17116	5172	20630	9292	2853283	1305908
2500	25	40	15782	5538	18069	5904	21298	9914	3591443	1577088
2500	25	45	17056	5668	18447	6456	23433	9965	3895822	1850030
2500	25	50	18744	6098	20301	7034	24335	9945	4103720	2284954

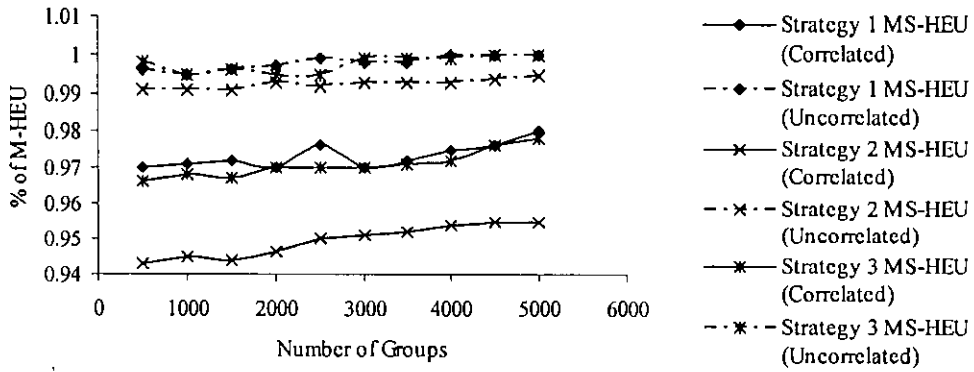


Figure 4.1: Performance of different strategies of MS-HEU normalized with respect to M-HEU for the MMKP data sets with $l=25$ and $m=25$

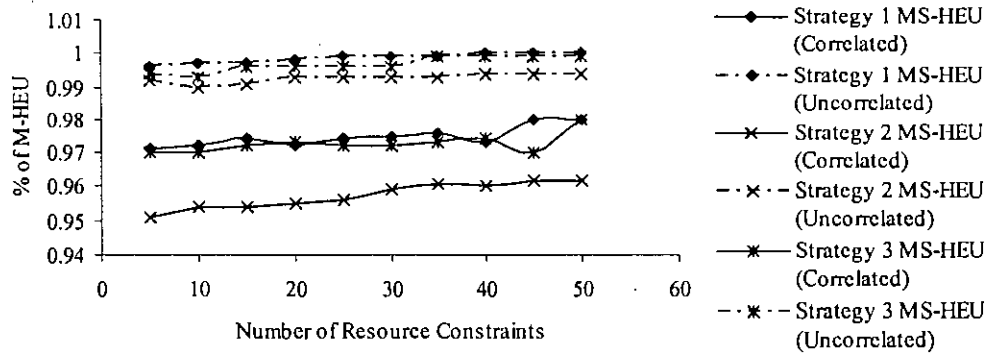


Figure 4.2: Performance of different strategies of MS-HEU normalized with respect to M-HEU for the MMKP data sets with $n=2500$ and $l=25$

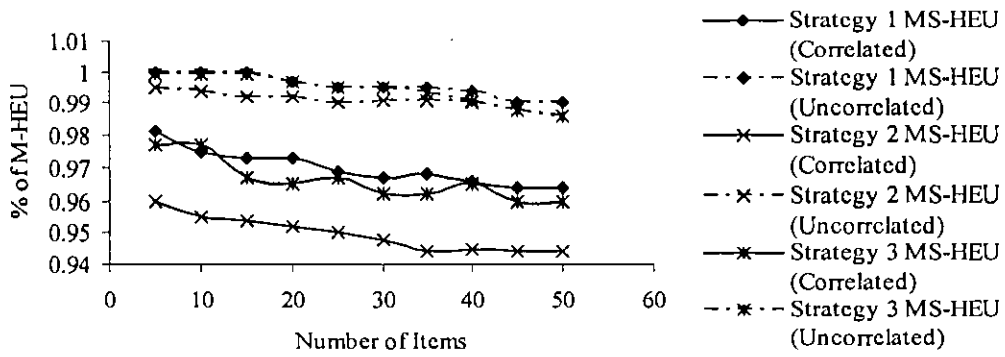


Figure 4.3: Performance of different strategies of MS-HEU normalized with respect to M-HEU for the MMKP data sets with $n=2500$ and $m=25$

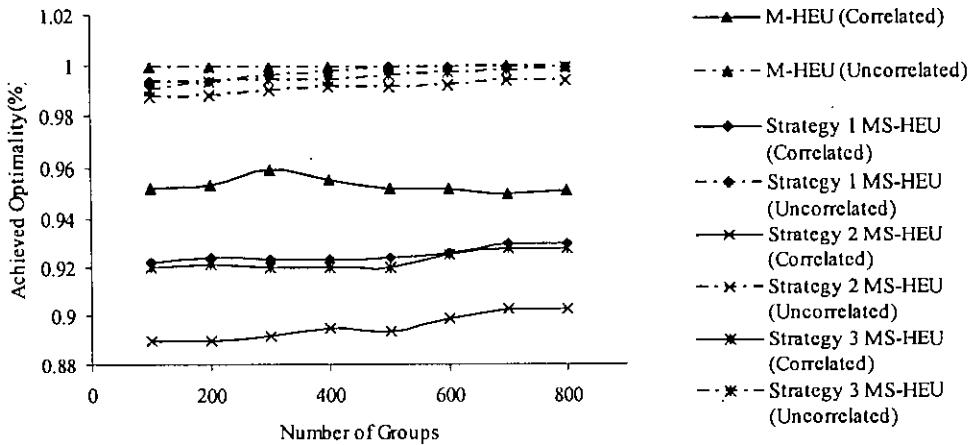


Figure 4.4: Performance of different strategies of MS-HEU and M-HEU normalized with the upper bound for the MMKP data sets with $l=10$ and $m=10$

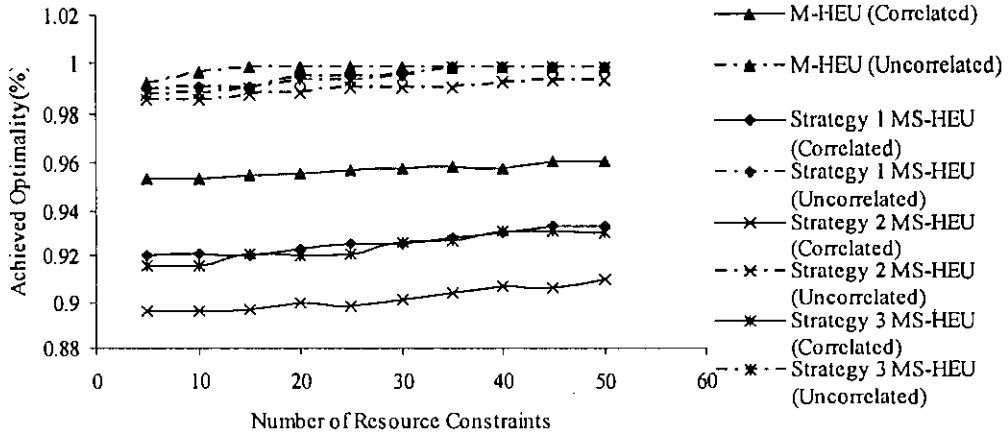


Figure 4.5: Performance of different strategies of MS-HEU and M-HEU normalized with the upper bound for the MMKP data sets with $n=500$ and $l=10$

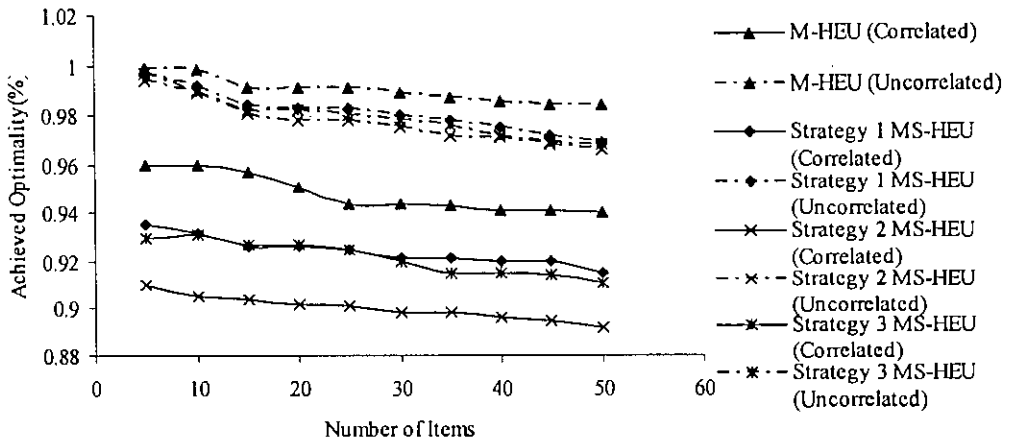


Figure 4.6: Performance of different strategies of MS-HEU and M-HEU normalized with the upper bound for the MMKP data sets with $n=500$ and $m=10$

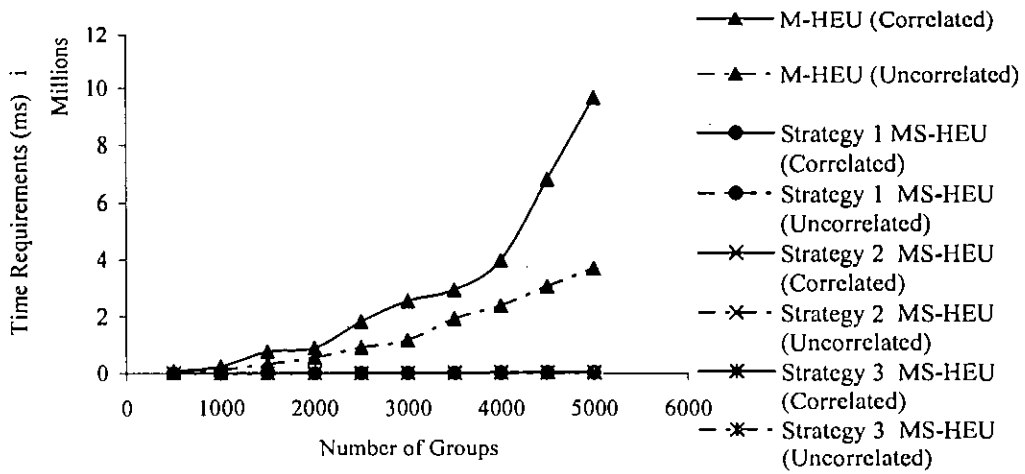


Figure 4.7: Time required by different strategies of MS-HEU and M-HEU for the MMKP data sets with $m=25$ and $l=25$

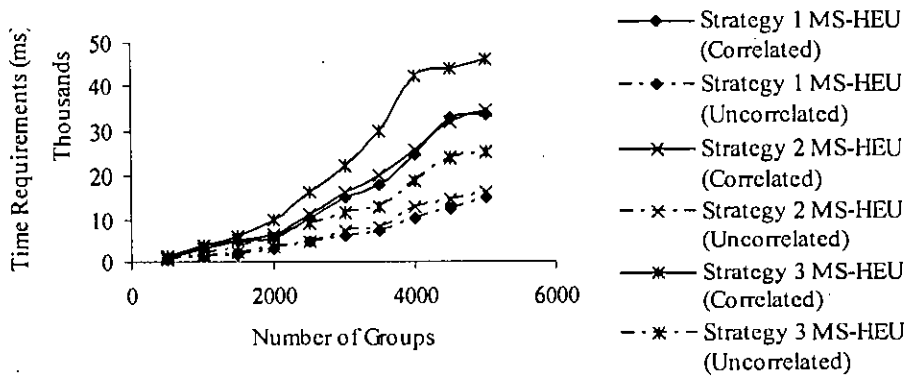


Figure 4.8: Time required by different strategies of MS-HEU for the MMKP data sets with $m=25$ and $l=25$

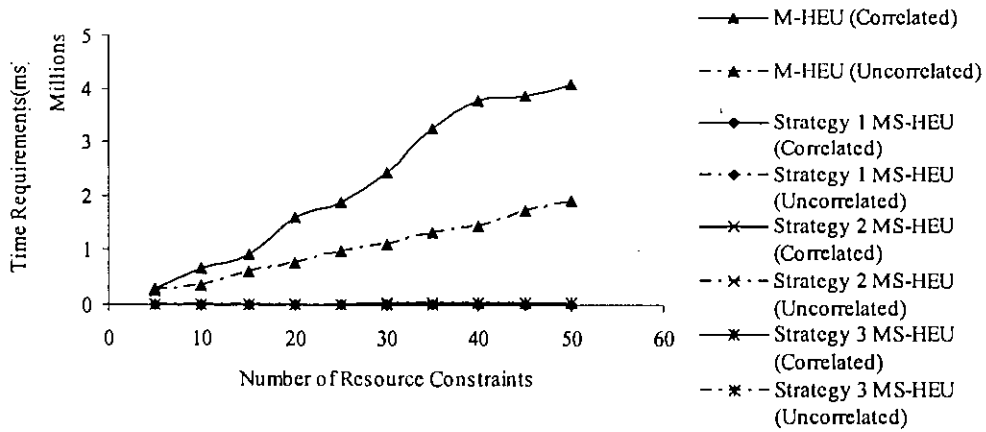


Figure 4.9: Time required by different strategies of MS-HEU and M-HEU for the MMKP data sets with $n=2500$ and $l=25$

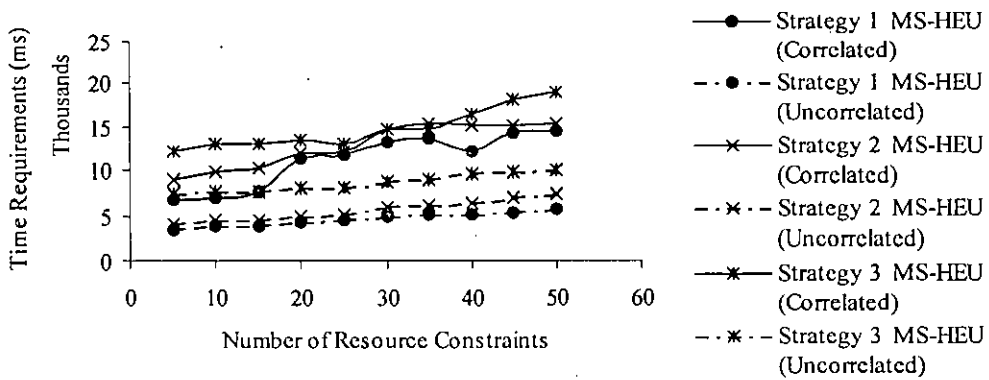


Figure 4.10: Time required by different strategies of MS-HEU for the MMKP data sets with $n=2500$ and $l=25$

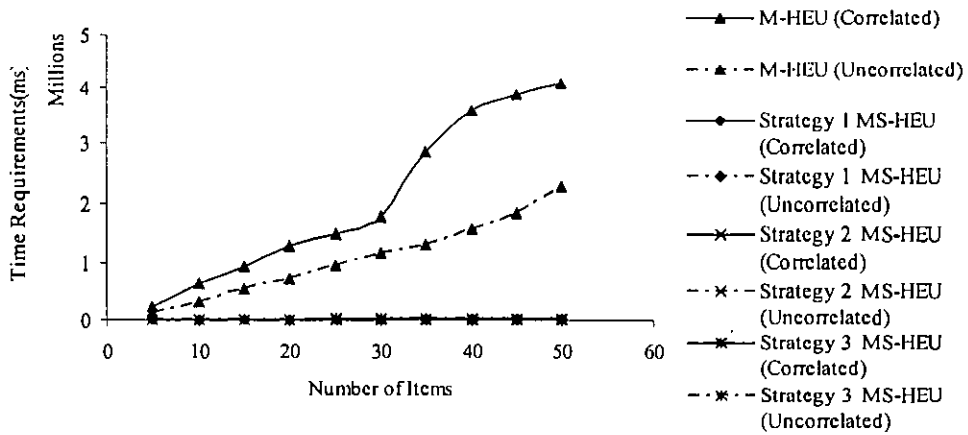


Figure 4.11: Time required by different strategies of MS-HEU and M-HEU for the MMKP data sets with $n=2500$ and $m=25$

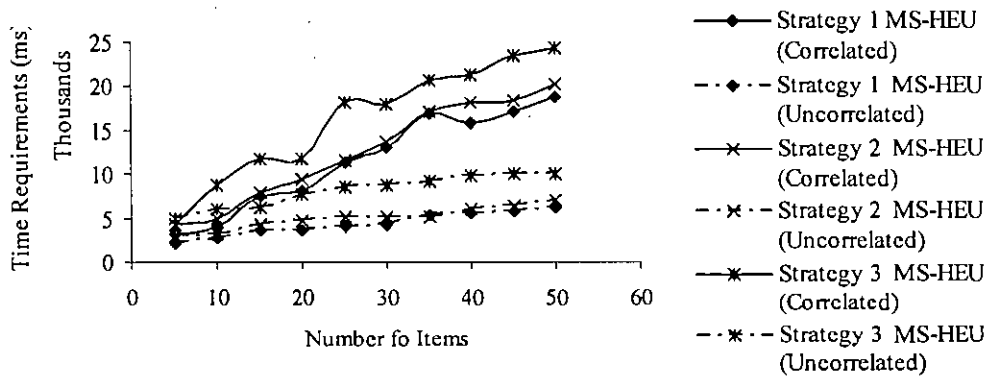


Figure 4.12: Time required by different strategies of MS-HEU for the MMKP data sets with $n=2500$ and $m=25$

0

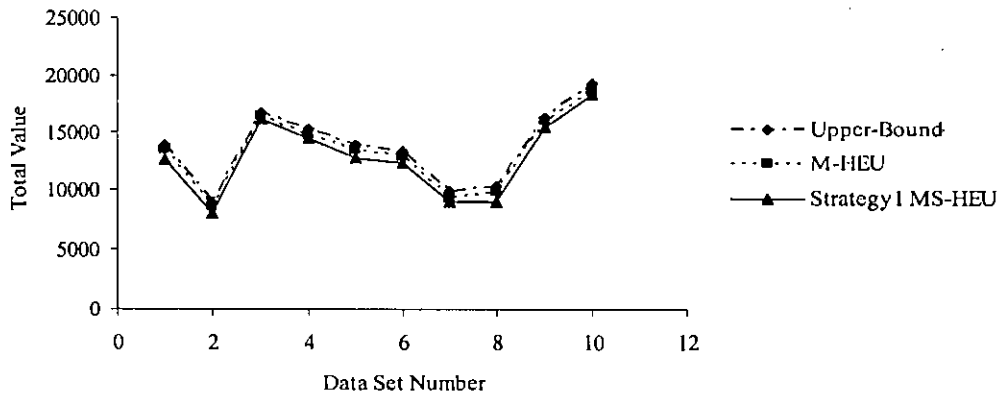


Figure 4.13: Comparison of the total values of the items picked by Strategy 1 of MS-HEU, M-HEU and Upper-Bound for 10 uncorrelated problem sets with $n=100$, $m=5$, $l=10$

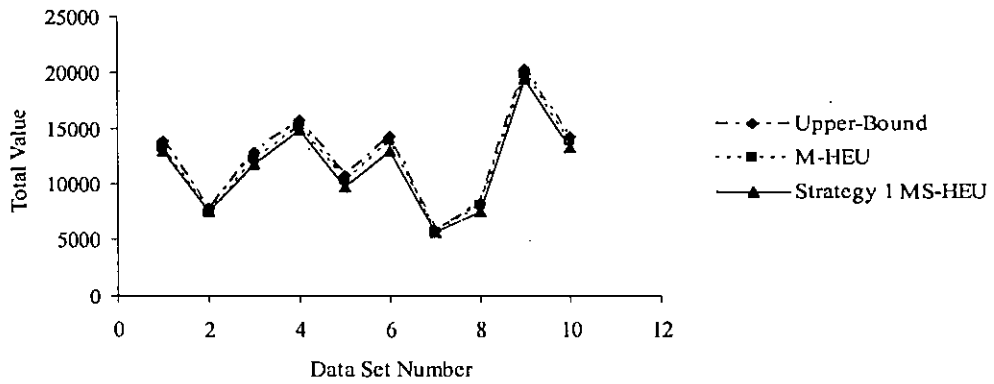


Figure 4.14: Comparison of the total values of the items picked by Strategy 1 of MS-HEU, M-HEU and Upper-Bound for 10 correlated problem sets with $n=100$, $m=5$, $l=10$

4.4 Observations

➤ M-HEU produces solutions which are close to the optimal solutions provided by the algorithm BBLP. Figure 4.1 to 4.3 show the performance of different strategies of MS-HEU with respect to the M-HEU. It is shown that Strategy 1 and Strategy 3 of MS-HEU produce better solutions than Strategy 2 of MS-HEU. The solutions achieved by Strategy 1 and 3 are close to the solutions of M-HEU and sometimes the value achieved by these strategies is about 100% to the value achieved by M-HEU. On an average Strategy 1 and 3 achieve about 98.5% and 98% of the value achieved by M-HEU, respectively.

In most of the cases, the solution achieved by Strategy 1 is better than that of Strategy 3. This is likely because, in the later iterations, one or more than one item may be upgraded in Strategy 1, but exactly one item is upgraded in Strategy 3. And it might happen that a single higher valued item may not give a feasible solution, but two or more items may give a feasible solution. Because the selection of a single item may not satisfy one or more resource constraints, if the values of those resource constraints of the selected item are high. But when more than one item is selected and if the values of the resource constraints of the next selected items are small, the resource constraints may be satisfied and give a feasible solution.

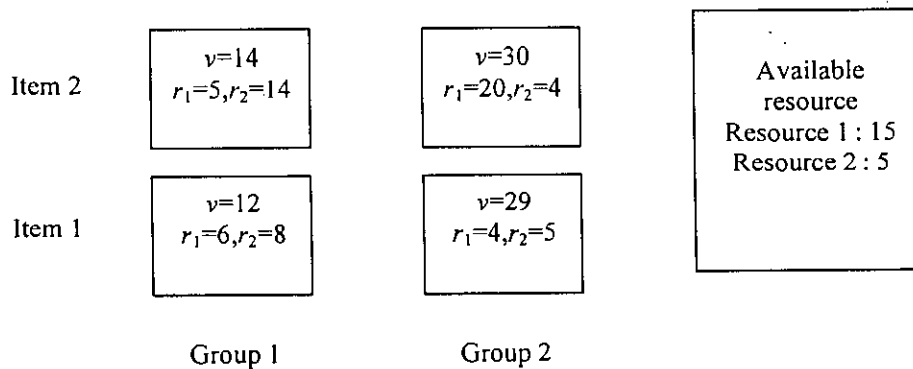


Figure 4.15: Example of an MMKP with available resources

As an example, let the resources of an MMKP is two dimensional and the available resources are 15 and 5 respectively for Resource 1 and 2 (shown in Figure 4.15). Let there are two groups where upgrade is possible and let Item 1 be the currently selected item in these groups. The possible upgrades are, from Item 1 to Item 2 in these groups. But individually none of them does satisfy the available

resources. If Group 1 is upgraded, it does not satisfy the available resources, because the required resources are -1 and 6 respectively, where the available resource is 5 for Resource 2. Similarly if Group 2 is upgraded, the required resources are 16 and -1 respectively, where the available resource is 15 for Resource 1. But when two groups are simultaneously upgraded, the required resources are 15 and 5 respectively which satisfy the available resources.

The value achieved by Strategy 2 of MS-HEU is worse than that of other strategies and it achieves about 96% of the value achieved by M-HEU. This is likely because the total number of upgradations of Strategy 2 is smaller than that of other strategies. So some feasible upgrades are left unconsidered in this strategy. As a result some resources are left unconsumed. In this strategy the upgradation that is not feasible in previous iteration is not considered in the later iterations.

- We find from Figure 4.1 to Figure 4.6 that M-HEU and different strategies of MS-HEU give better results for uncorrelated data sets than correlated data sets. We can give a plausibility argument of the behavioral differences between correlated and uncorrelated data sets of MS-HEU. When the data sets are fully correlated, the items of a group lie on a straight line. The items with high resource consumption and high values are picked first. So that higher valued items are selected quickly and resources are fulfilled. In the later iterations the lower valued items are not considered, where some of them would give a feasible solution. So that we lose some revenue here. But if the data sets are random then the picking of items will not be biased. Both high and low valued items will be picked with the same probability and we get better solutions. This is likely the reason that different strategies of MS-HEU have better optimality for uncorrelated MMKP data sets than correlated MMKP data sets
- We also find from Table 4.1 to Table 4.3 and Figure 4.7 to Figure 4.12 that for correlated data sets, these algorithms take more time than uncorrelated data sets. When a data set is correlated there is a chance that almost every combination is feasible. In an uncorrelated data set, we generally get more infeasible picking constraints than correlated data sets. We do not need to calculate the aggregate resources for those items. Therefore we can get a feasible solution with less computation for random data sets than for correlated data sets. That is why

different strategies of MS-HEU have less time requirements for uncorrelated data sets than correlated data sets.

- The optimality achieved by different strategies of MS-HEU is almost stable for larger problem sets shown in Figure 4.4. We find almost the same trend for an increase in the number of resource dimensions shown in Figure 4.5.
- Figure 4.6 shows that the achieved optimality decreases with an increase in the number of items in each group. This is likely because we ignore some items while picking items from different groups. We only consider feasible upgrade in MS-HEU. The items with higher values, which give a feasible solution, are picked first in each iteration. So that the lower valued items than the new selected items are ignored in the subsequent iterations. But if these items are considered in the later iterations, some of them may give feasible solution and the solution value may be increased. The number of ignored items is increased with the increase in number of items. Consequently the number of ignored lower valued items, which may give a feasible solution, is also increased. So that we lose more revenue with the increase in number of items. In M-HEU, when an upgradation gives an infeasible solution, some groups are downgraded for feasible solution. So that some items that are ignored in the previous iterations are considered in the later iterations in M-HEU. So this behavior is more remarkable in MS-HEU than M-HEU.
- Figure 3.7 to Figure 3.12 show that the time requirements of M-HEU and different strategies of MS-HEU. It is shown that the time requirement of different strategies of MS-HEU is much less than that of the M-HEU. From Table 4.1 to 4.3 and from Figure 3.8, 3.10, and 3.12, it is also clear that the time requirement of Strategy 3 is more than the time requirements of Strategy 1 and Strategy 2. This is likely because the number of iterations in Strategy 3 is at most nl , where there is a logarithmic number of iterations in Strategy 1 and Strategy 2.
- If we observe the difference between estimated optimal total value achieved by the Upper Bound and the total value of the items picked by Strategy 1 of MS-HEU and M-HEU of the MMKP in Figure 4.13 and Figure 4.14, for 10 different uncorrelated data and correlated data respectively, the performance of Strategy 1 of MS-HEU appears to be consistent to solve the MMKP.

Chapter 5

Conclusions

There are several heuristic algorithms for solving the MMKP. These are sequential algorithms and some of them are discussed briefly in Chapter 2. But there is no parallel algorithm for solving the MMKP. In this thesis, we have proposed a heuristic based parallel algorithm that runs on CRCW PRAM in $O(\log nl (\log n + \log m + \log \log l))$ time using $O(n \log nl (\log n + lm))$ operations exploiting $O((n \log n + lmn) / (\log n + \log m + \log \log l))$ processors. In our parallel algorithm, we have used the same candidate item evaluation criteria as used in HEU by Khan or M-HEU by Akbar. Here we summarize the major contributions from our thesis and present suggestions for the future research work.

5.1 Major Contributions

The major contributions made in this thesis are as follows:

- When the number of groups increases beyond a certain limit single processor based solutions may not be able to provide real-time response. In this thesis we proposed a parallel algorithm, PRAM-HEU in polylog time. The time complexity and the total number of operations are calculated for PRAM-HEU. The total number of processors has also been calculated.
- In chapter 3, we discussed why it is not possible to provide a parallel algorithm from M-HEU directly. Then to provide a parallel heuristic algorithm, M-HEU is modified to some extent and a new heuristic algorithm, MS-HEU is developed. Three different strategies of MS-HEU is proposed depending on the number of iterations and the number of upgrades in each iteration, from where one strategy is considered to develop the parallel heuristic algorithm. The complexities of different strategies of MS-HEU have been calculated.
- We can get the exact solution by BBLP technique, but that will take exponential time complexity. So we actually compute an upper bound of the value using the same technique but with less iteration. Then we compute the

percentage of the value computed by our algorithm as well as computed by M-HEU with respect to the upper bound. This gives us the achieved optimality.

- We have performed experiments on an extensive set of data. We present the comparison of performance and time requirements between our algorithm and M-HEU (shown in graph). We also analyze the experimental results.

5.2 Future Research Work

We suggest the following research plans on heuristics for solving the MMKP:

- *PRAM Model Simulator*: We did not design the PRAM model simulator for our algorithm; rather we implemented the serial version of PRAM-HEU to calculate the performance of PRAM-HEU. So the design of PRAM model simulator for PRAM-HEU is a good research topic from where the analysis of time requirement and performance measurement of the PRAM-HEU can easily be done with respect to other heuristics and also with respect to exact algorithms, such as BBLP.
- *Average Case Analysis*: We presented the worst-case complexities of MS-HEU and PRAM-HEU for solving the MMKP. The analysis of time requirement complexity and achieved optimality in the average case is a very interesting research topic in theoretical computer science.
- *PRAM Algorithms for Exact Solution*: There is no parallel algorithm for exact solution of MMKP. We know that, the computation time for any exact algorithm, such as BBLP, may grow exponentially with the size of the problem instance in the worst case. But if we provide a PRAM algorithm for exact solution, then the problem can be solved in reasonable time. So, this is an interesting unsolved problem and one may work on that further.
- *PRAM Models for Other Heuristics*: I-HEU having a little difference from M-HEU is used as an incremental and scalable algorithm. This will be considered in future to provide a PRAM model of I-HEU that can improve the scalability and fault tolerance of adaptive multimedia with better computation complexity. Another heuristic algorithm, C-HEU is developed for solving the MMKP using convex-hull approach. This is remarkable, because this is a

sequential algorithm with $O(n \log nl + nlm)$. So these are also good research topic to design PRAM model for other heuristics.

- *Implementation of Admission Controller:* Implementation of Admission Controller needs to be done using this system. We have not implemented an Admission Controller using the parallel heuristic. The performance of an Admission Controller using the parallel heuristic can be studied.
- *Implementation of distributed algorithms for the MMKP:* Distributed algorithm for the MMKP can be implemented using socket programming. In distributed systems, there are a collection of multimedia servers which can be located anywhere in the world. These servers may exchange information about the amount of resources available in each of them and the revenue earned by them. The algorithm is run in each of the server and a new parameter is added to the problem regarding which server to select to meet a particular request. Socket programming can be used to exchange information among servers.

Appendix

Program written using Java programming language:

```
// In this program:
// Session means group of the MMKP
// QoS level of session means Items of the MMKP

import java.util.*;
import java.lang.*;
import java.io.*;

class Node{
    int no_of_fixed_group;
    int status[];
    int next_branch_session;
    double upper_bound;
}

class can_item{
    int group_no;
    int item_no;
    double value;
    int type; //1 means less resource more revenue delr is positive
              //0 otherwise
    void assgn_null_can_item(){
        group_no=-1;
        item_no=-1;
        type=0;
        value=-1.0;
    }
    void set_can_item(int grp , int item, double delr, double
delp){
        group_no=grp;
        item_no=item;
        if (delr<0){
            type=0;
            value=delp;
        }
        else{
            type=1;
            value=delr;
        }
    }
    void print(){
        System.out.println(group_no+ " "+ item_no+ " "+type+"
"+value+" ");
    }
}

class resource{
    int no_of_resources;
    double r[];

    resource(int i){
        no_of_resources=i;
        r=new double[no_of_resources];
    }
    void add_res(double s[]){
```

```

        for (int i=0;i<no_of_resources;i++) r[i]+=s[i];
    }
    void sub_res(double s[]){
        for (int i=0;i<no_of_resources;i++) r[i]-=s[i];
    }
    int feasible(double s[]){
        for (int i=0;i<no_of_resources;i++) {
            if (r[i]>s[i]) return 0;
        }
        return 1;
    }
    void print(){
        System.out.println("");
        for (int i=0;i<no_of_resources;i++)
System.out.print(r[i]+" ");
        System.out.println("");
    }
}

public class nnMMKP{
Node solutionNode;
static final double TOL=1.0e-6;
int kp=0,icase=0,ip=0;
double ql=0.0,bmax=0.0;
int l1[],l2[],l3[];
int no_of_sessions,no_of_resources,no_of_qos;
int
no_of_variables,no_of_equations,objective_equation,no_of_live_variabl
es;
int inf_const=0;
int max_res; //Maximum consumed resource
int lhs[],rhs_var[];
double rhs_coeff[][];
double cost[][][],resource[][][],total_constraint[],used_resource[];
double cost_per_unit[]; //per unit cost of the resources
int solution[],saved_solution[],no_of_qos_levels[];
can_item candidates[];

//solution[]: for holdin the current solution
//saved_solution[]: saving a solution
//no_of_qos_levels[]: Holding the number of items in each group
Vector head;
double parallel_rev,serial_rev,bblp_rev,upper_rev;

int increased_revenue=0;

    void datainit(int no_ses,int no_res,int no_qos) {
        Random rand_var=new Random(20);
        int random=1;
        int rc=10;
        int pc=10;
        no_of_sessions=no_ses;
        no_of_resources=no_res;
        no_of_qos=no_qos;
        cost=new double[no_of_sessions][no_of_qos];
        resource=new
        double[no_of_resources][no_of_sessions][no_of_qos];
        total_constraint=new double[no_of_resources];
    }
}

```

```

cost_per_unit=new double[no_of_resources];
used_resource=new double[no_of_resources];
no_of_qos_levels=new int[no_of_sessions];
solution=new int[no_of_sessions];
saved_solution=new int[no_of_sessions];
candidates=new can_item[no_of_sessions];
for (int i=0;i<no_of_sessions;i++) candidates[i]=new
can_item();
int total_no_of_resources;
int j,i,k;
double temp;
total_no_of_resources=no_of_resources;
for (k=0;k<total_no_of_resources;k++)
total_constraint[k]=0.5*rc*no_of_sessions;
for (k=0;k<total_no_of_resources;k++)
cost_per_unit[k]=rand_var.nextInt(pc);
for (i=0;i<no_of_sessions;i++){ //Initializing
resource req of the items
no_of_qos_levels[i]=no_of_qos; //No of items
in each group
for (j=0;j<no_of_qos_levels[i];j++){
for (k=0;k<total_no_of_resources;k++) {
resource[k][i][j]=rand_var.nextInt(rc);
}
}
}

if (random==1){
// The value of item is not proportional to resource
consumption
for (i=0;i<no_of_sessions;i++){
for (j=0;j<no_of_qos_levels[i];j++){

do{
temp=0.0;
for
(k=0;k<total_no_of_resources;k++)
temp+=resource[k][i][j]*cost_per_unit[k];

cost[i][j]=rand_var.nextInt(total_no_of_resources*(rc/2)*(pc/2)
);

cost[i][j]=(cost[i][j]*(j+1))/no_of_qos;
}while (temp<cost[i][j]);

}
}
}
else{
// The value of item is proportional to resource consumption
for (i=0;i<no_of_sessions;i++){
for (j=0;j<no_of_qos_levels[i];j++){
cost[i][j]=0.0;
for (k=0;k<total_no_of_resources;k++)
cost[i][j]+=resource[k][i][j]*cost_per_unit[k];

cost[i][j]+=rand_var.nextInt(total_no_of_resources*3*(rc/10)*(p
c/10));
}
}
}
}

```

```

        }
    }

    for (i=0;i<no_of_sessions;i++) { //The items are
sorted according to the value
        sort_pile(i);
        solution[i]=0; //Initial solution
    }

}

void init_solution(){
    for (int i=0;i<no_of_sessions;i++){
        solution[i]=0;
    }
    for (int k=0;k<no_of_resources;k++){
        used_resource[k]=0.0;
        for(int i=0;i<no_of_sessions;i++)
used_resource[k]+=resource[k][i][solution[i]];
    }
}

void max_res_cons(){
    max_res=0;
    for (int i=1;i<no_of_resources;i++){
        if (used_resource[i]>used_resource[max_res]) max_res=i;
    }
}

double scaled_res_cons(int k){
    //return (used_resource[k]*used_resource[k]);
    if ((used_resource[max_res]/used_resource[k])>2.0) return
used_resource[k];
    if ((used_resource[max_res]/used_resource[k])>1.6) return
(used_resource[k]*used_resource[k]);
    if ((used_resource[max_res]/used_resource[k])>1.3) return
(used_resource[k]*used_resource[k]*used_resource[k]);
    else return
(used_resource[k]*used_resource[k]*used_resource[k]*used_resource[k])
;
}

void calculate_can_items(){

    double delr,delp;
    int j,k;
    max_res_cons();
    for(int i=0;i<no_of_sessions;i++){
        candidates[i].assgn_null_can_item();
        for(j=solution[i]+1;j<no_of_qos_levels[i];j++){
            delr=0.0;
            double delr1=0.0,delr2=0.0;
            for(k=0;k<no_of_resources;k++){
                if (Math.abs(used_resource[k])>TOL) {

delr1+=resource[k][i][solution[i]]*scaled_res_cons(k);

delr2+=resource[k][i][j]*scaled_res_cons(k);
                }
            }
        }
    }
}

```

```

        else {
            delr1+=resource[k][i][solution[i]];
            delr2+=resource[k][i][j];
        }
    }
    delr=delr1-delr2;
    // Finding the change of aggr res consumption
    delp=((double)(cost[i][solution[i]]-
cost[i][j]))/((double)delr);
    can_item t_can_item=new can_item();
    t_can_item.set_can_item(i,j,delr,delp);
    if (compare_can_item(t_can_item,candidates[i])==1)
        assgn_can_item(candidates[i],t_can_item);
    }
}

void sort_can_items(){
    for (int i=0;i<no_of_sessions;i++){
        for (int j=i+1;j<no_of_sessions;j++){
            if
(compare_can_item(candidates[i],candidates[j])==0){
                //Candidate Item i is less than j
                can_item temp=new can_item();
                assgn_can_item(temp,candidates[i]);
                assgn_can_item(candidates[i],candidates[j]);
                assgn_can_item(candidates[j],temp);
            }
            //candidates[i].print();
        }
    }
}

double[] get_resource(int group_no,int item_no) {
    double r[];
    r=new double[no_of_resources];
    for (int k=0;k<no_of_resources;k++)
        r[k]=resource[k][group_no][item_no];
    return r;
}

int select_items(int num) {
    int i;

    resource t_resource=new resource(no_of_resources);
    t_resource.add_res(used_resource);
    for (i=0;i<num;i++){
        t_resource.add_res(get_resource(candidates[i].group_no,candidat
es[i].item_no));
        t_resource.sub_res(get_resource(candidates[i].group_no,solution
[candidates[i].group_no]));
    }

    for (i=num-1;i>=0;i--){
        if (t_resource.feasible(total_constraint)==1) break;
        else{

```

```

        t_resource.sub_res(get_resource(candidates[i].group_no,candidat
es[i].item_no));

        t_resource.add_res(get_resource(candidates[i].group_no,solution
[candidates[i].group_no]));
    }
}

if (i==-1) return 0;
for (int j=0;j<=i;j++)
solution[candidates[j].group_no]=candidates[j].item_no;
for (int k=0;k<no_of_resources;k++){
    used_resource[k]=0.0;
    for(i=0;i<no_of_sessions;i++)
        used_resource[k]+=resource[k][i][solution[i]];
}

return 1;
}

int compare_can_item(can_item c1,can_item c2){
    if (c1.type<c2.type) return 0;
    if (c1.type>c2.type) return 1;
    if (c1.value<c2.value) return 0;
    else return 1;
}

void assgn_can_item(can_item c1,can_item c2){
    c1.group_no=c2.group_no;
    c1.item_no=c2.item_no;
    c1.value=c2.value;
    c1.type=c2.type;
}

// for Strategy 3 of MS_HEU
/*void do_ms_heu(){
    int num_items_to_select=no_of_sessions*no_of_qos;
    int sel_success;
    int act_upgrade=0;
    do{
        calculate_can_items();
        sort_can_items();
        int n;
        for(n=0;n<no_of_sessions;n++){
            if(candidates[n].group_no==-1)break;
        }
        if(num_items_to_select/2>(n-1))
            act_upgrade=n-1;
        else
        {
            if(num_items_to_select==1)
                act_upgrade=num_items_to_select;
            else
                act_upgrade=num_items_to_select/2;
        }
        sel_success=select_items(act_upgrade);
        if (num_items_to_select!=1) num_items_to_select/=2;
    }while (num_items_to_select>=1 && sel_success==1);
    parallel_rev=netrev();
    //System.out.println("Revenue earned by Parallel HEU"+"
"+netrev());
}

```



```

}
//for Strategy2 of MS_HEU
void do_ms_heu(){
    int num_items_to_select=no_of_sessions*no_of_qos;
    int sel_success;
    int act_upgrade=0;
    do{
        calculate_can_items();
        sort_can_items();
        int n;
        for(n=0;n<no_of_sessions;n++){
            if(candidates[n].group_no==--1)break;
        }
        if(num_items_to_select/2>(n-1))
            act_upgrade=(n-1);
        else
            act_upgrade=num_items_to_select/2;
        sel_success=select_items(act_upgrade);
        if (num_items_to_select!=1) num_items_to_select/=2;

    }while (num_items_to_select>1 && sel_success==1);
    parallel_rev=netrev();
    //System.out.println("Revenue earned by Parallel HEU"+"
    "+netrev());
}
*/
//Strategy 1 of MS_HEU
void do_ms_heu(){
    int num_items_to_select=no_of_sessions*no_of_qos;
    int pending_upgrade=0;
    int act_upgrade=0;
    int sel_success;
    do{
        calculate_can_items();
        sort_can_items();
        int n;
        for(n=0;n<no_of_sessions;n++){
            if(candidates[n].group_no==--1)break;
        }

        if((pending_upgrade+num_items_to_select/2)>no_of_sessions)
            act_upgrade=n-1;
        else
            act_upgrade=(pending_upgrade+num_items_to_select/2);
        if (num_items_to_select!=1) num_items_to_select/=2;
        sel_success=select_items(act_upgrade);
        pending_upgrade=pending_upgrade+(num_items_to_select-
        act_upgrade);
    }while(num_items_to_select>1 && sel_success==1);
    parallel_rev=netrev();
    //System.out.println("Revenue earned by Parallel HEU"+"
    "+netrev());
}

void write_to_file() {
    try{
        String S=new String("");
        S=no_of_sessions+ "\t"+ no_of_resources+ "\t"+no_of_qos+
        "\t"+parallel_rev +"\t"+ serial_rev+ "\t"+

```

```

        upper_rev+"\t"+parallel_rev/serial_rev+ "\t"+
        serial_rev/upper_rev+"\t"+ parallel_rev/upper_rev+"\n";
        RandomAccessFile p=new
        RandomAccessFile("pMMKP.txt","rw");
        p.seek(p.length());
        p.writeBytes(S);
        p.close();
    }catch ( IOException e){
        System.out.println("An Error Occurred in writing");
    }
}
void revive_solution(){
    //Revives the previously saved solution
    int i,k;
    for (i=0;i<no_of_sessions;i++){
        solution[i]=saved_solution[i];
    }
    for (k=0;k<no_of_resources;k++){
        used_resource[k]=0;
        for(i=0;i<no_of_sessions;i++){
            used_resource[k]+=resource[k][i][solution[i]];
        }
    }
}

double netrev(){
    //Calculates the total revenue : summation of the value of the
    //selected items
    int i;
    double total_rev;
    total_rev=0.0;
    for (i=0;i<no_of_sessions;i++) {
        total_rev+=cost[i][solution[i]];
        //System.out.println(solution[i]);
    }
    return total_rev;
}

void sort_pile(int i){
    //Sorts the ith group of the MMKP
    int j,k,l;
    double temp;

    for (j=0;j<no_of_qos_levels[i];j++){
        for (k=j+1;k<no_of_qos_levels[i];k++){
            if (cost[i][j]>cost[i][k]){
                temp=cost[i][j];
                cost[i][j]=cost[i][k];
                cost[i][k]=temp;
                for (l=0;l<no_of_resources;l++){
                    temp=resource[l][i][j];
                    resource[l][i][j]=resource[l][i][k];
                    resource[l][i][k]=temp;
                }
            }
        }
    }
}

void infeasible_constraint(){
    //Determines the most infeasible resource constraint for the
    //current resource consumption
    double inf,cinf;
}

```

```

int i;

inf=0.0;
cinf=0.0;
inf_const=0;
// A global variable determining the most infeasible resource
for(i=0;i<no_of_resources;i++){
    if (Math.abs(total_constraint[i])<TOL){ // if total
        constraint is 0
            if (Math.abs(used_resource[i])>TOL){ // id
                consumed resource is more than 0
                    inf_const=i;
                    return;
            }
        }else{
            cinf=used_resource[i]/total_constraint[i];
            if (cinf>inf){
                inf=cinf;
                inf_const=i;
            }
        }
    }
}

int find_feasible(){
    // Step 1 of the heuristic: Finding a feasible solution
    int i;
    do{
        infeasible_constraint();
        if
        (used_resource[inf_const]<=total_constraint[inf_const]) return 1;
        i=resource_conservation();
        // Finding an item with less resource consumption
    }while (i==1);
    return 0;
}

int downgradepossible(int session_no,int qos_no)
//Whether Group session_no can be downgraded to Item qos_no to find a
feasible solution
{
    int i;
    if (qos_no==solution[session_no]) return 0;
    for (i=0;i<no_of_resources;i++){
        if (total_constraint[i]<used_resource[i] &&
resource[inf_const][session_no][qos_no]>=resource[inf_const][session_
no][solution[session_no]])
            return 0;
        // Makes an infeasible resource more infeasible
    }else{
        if
        (resource[i][session_no][qos_no]>resource[i][session_no][solution[ses
sion_no]]){
            if ((used_resource[i]-
resource[i][session_no][solution[session_no]]+resource[i][session_no]
[qos_no])>total_constraint[i])
                return 0;
            // Makes a feasible resource infeasible
        }
    }
}
}

```

```

        return 1;
    }
}
int resource_conservation(){
    double delr,mdelr;
    int tsession,tqos,k,i,j,m;
    m=0;
    mdelr=0.0;

    tsession=-1;
    tqos=-1;
    for(i=0;i<no_of_sessions;i++){
        m=solution[i]+1;
        for(j=m;j<no_of_qos_levels[i];j++){
            if (downgradepossible(i,j)!=0){
                delr=0.0;
                for(k=0;k<no_of_resources;k++){
                    // Calculating delr: change of aggregate resource
                    if (Math.abs(used_resource[k])>TOL)
delr+=(resource[k][i][solution[i]]-
resource[k][i][j])*used_resource[k];
                    else
delr+=(resource[k][i][solution[i]]-resource[k][i][j]);
                }
                //if (absc>TOL) delr=delr/absc;
                if(delr>mdelr || tsession===-1){
                    // Finding the highest delr
                    tsession=i;
                    tqos=j;
                    mdelr=delr;
                }
            }
        }
    }
    if (tsession!=-1){
        // An item is found to find feasible solution
        for (k=0;k<no_of_resources;k++){
            used_resource[k]+=resource[k][tsession][tqos]-
            resource[k][tsession][solution[tsession]];
            solution[tsession]=tqos;
            return 1;
        }
    }
    return 0; // No item is found to find feasible solution
}

int resource_upgradation(){

    // Selecting new items by upgrading only

    double delr,mdelr;
    double delp,mdelp;

    int tsession,tqos,k,i,j;
    mdelr=0.0;
    mdelp=0.0;

    tsession=-1;
    tqos=-1;
    for(i=0;i<no_of_sessions;i++){

        for(j=solution[i]+1;j<no_of_qos_levels[i];j++){

```

```

// Only upgrading
if (resource_constraint(i,j)==1){
    //Only feasible upgrades are allowed
    delr=0.0;
    double delr1=0.0,delr2=0.0;
    for(k=0;k<no_of_resources;k++){
        if (Math.abs(used_resource[k])>TOL) {

delr1+=resource[k][i][solution[i]]*used_resource[k];
delr2+=resource[k][i][j]*used_resource[k];
        }
        else {

delr1+=resource[k][i][solution[i]];
        delr2+=resource[k][i][j];
        }
    }
    delr=delr1-delr2;
    // Finding the change of aggr res consumption
    if(delr>mdelr || tsession==-1){
        tsession=i;
        tqos=j;
        mdelr=delr;
    }
    if (mdelr<0){
// If the change of aggregate res is negative then looking for items
//with highest change of value with respect to the change of aggregate
//res consumption

        delp=((double)(cost[i][solution[i]]-
cost[i][j]))/((double)delr);

        if (delp>mdelp){
            mdelp=delp;
            tsession=i;
            tqos=j;
        }
    }
}

}

if (tsession!=-1){
    // An upgradeable item found
    for (k=0;k<no_of_resources;k++)
        used_resource[k]+=resource[k][tsession][tqos]-
        resource[k][tsession][solution[tsession]];
    solution[tsession]=tqos;
    return 1;
}
else return 0; // No item is found to upgrade the solution
}

int resource_up_de_gradation(){
    // upgrading following downgrades
    double delr;
    double target;
    double delp,mdelp,cmdelp;

    int tsession,tqos,k,i,j,tsession1,tqos1,tsession2,tqos2;

```

```

mdelp=0.0;

tsession=-1;
tqos=-1;

//Finding an infeasible upgrade
for(i=0;i<no_of_sessions;i++){
    for(j=solution[i]+1;j<no_of_qos_levels[i];j++){
        delr=0.0;
        for(k=0;k<no_of_resources;k++){
            // Determining delr(prime)
            if (Math.abs(total_constraint[k]-
used_resource[k])>TOL) delr+=(resource[k][i][solution[i]]-
resource[k][i][j])/(total_constraint[k]-used_resource[k]);
            else delr+=(resource[k][i][solution[i]]-
resource[k][i][j]);
        }
        // Determining delv/delr(prime)
        delp=((double) (cost[i][solution[i]]-
cost[i][j]))/((double) delr);
        if (delp>mdelp || tsession===-1){
            mdelp=delp;
            tsession=i;
            tqos=j;
        }
    }
}
if (tsession===-1) return 0; // No upgrade found
target=-
cost[tsession][solution[tsession]]+cost[tsession][tqos];
// Determining how much downgrade is allowed
for (k=0;k<no_of_resources;k++){
    used_resource[k]+=resource[k][tsession][tqos]-
    resource[k][tsession][solution[tsession]];
solution[tsession]=tqos;
do{
    tsession1=-1;
    tqos1=-1;
    tsession2=-1;
    tqos2=-1;
    mdelp=0.0;
    cmdelp=0.0;
    for(i=0;i<no_of_sessions;i++){
        for(j=0;j<solution[i];j++){
            if (cost_improved(i,j,target)==1 && i!=tsession){
                // The selection which downgrades less than target
                delr=0;
                for(k=0;k<no_of_resources;k++){
                    //Detarmining delr
                    if (Math.abs(used_resource[k]-
total_constraint[k])>TOL)
                    delr+=(resource[k][i][solution[i]]-
resource[k][i][j])/(used_resource[k]-
total_constraint[k]);
                    else
                    delr+=(resource[k][i][solution[i]]-
resource[k][i][j]);
                }
            }
        }
    }
}
//Detarmining delr/delv

```

```

delp=((double)delp)/((double)(cost[i][solution[i]]-
cost[i][j]));
        if (delp>mdelp || tsession1!=-1){
            mdelp=delp;
            tsession1=i;
            tqos1=j;
        }
        if (resource_constraint(i,j)==1){
            // If feasibility retained
            if (delp>cmdelp || tsession2!=-1){
                cmdelp=delp;
                tsession2=i;
                tqos2=j;
            }
        }
    }
}
if (tsession2!=-1){
    // A feasible solution found
    for (k=0;k<no_of_resources;k++){
        used_resource[k]+=resource[k][tsession2][tqos2]-
        resource[k][tsession2][solution[tsession2]];
    }
    solution[tsession2]=tqos2;
    return 1;
}else if (tsession1!=-1){
    // A downgrade found but not feasible solution
    for (k=0;k<no_of_resources;k++){
        used_resource[k]+=resource[k][tsession1][tqos1]-
        resource[k][tsession1][solution[tsession1]];
    }
    //updating the target of downgrade
    target=target+cost[tsession1][tqos1]-
    cost[tsession1][solution[tsession1]];
    solution[tsession1]=tqos1;
}
else return 0;
}while (tsession!=-1);
return 1;
}

int resource_constraint(int i, int j){
    // Determines whether the upgrade or downgrade is feasible
    int k;
    for (k=0;k<no_of_resources;k++){
        if ((used_resource[k]-
        resource[k][i][solution[i]]+resource[k][i][j])>total_constraint[k])
            return 0;
    }
    return 1;
}

void save_solution(){
    // Saves the solution to saved_solution
    int i;
    for (i=0;i<no_of_sessions;i++)
        saved_solution[i]=solution[i];
}

```

```

int cost_improved(int i, int j, double target){
    // Determines whether the selection of Item j of Group i
    //downgrades the total solution value target
    if ((cost[i][solution[i]]-cost[i][j])<target) return 1;
    else return 0;
}

int verify_solution(){
    // Verifies whether the solution is infeasible
    int k;
    for (k=0;k<no_of_resources;k++) {
        if (used_resource[k]>total_constraint[k]) {
            System.out.println("Solution invalid");
            return 0;
        }
    }
    return 1;
}

void do_heu(){
    // Main function to determine the heuristic of the MMKP
    int i,k,j;

    for (k=0;k<no_of_resources;k++){
        used_resource[k]=0;
        for (i=0;i<no_of_sessions;i++)
            used_resource[k]+=resource[k][i][solution[i]];
    }
    i=find_feasible(); // Step 1: Finding feasible solution
    if (i==0) {
        System.out.println (" Solution not Available by HEU");
        return;
    }
    do{ // Step 2: Upgrading only
        i=resource_upgradation();

    }while(i==1);
    //System.out.println("Revenue earned by HEU"+" "+ netrev());

    do{
        save_solution(); // Saving solution
        i=resource_upgradation();
        // Step 3: Infeasible upgrade followed by Downgrades
        if (i==1){ // Step 3 is successful to upgrade
            do{ // Step 2 again
                j=resource_upgradation();

            }while(j==1);
        }else revive_solution();
        // Step 3 failed, so solution revived
    }while (i==1);
    for (k=0;k<no_of_resources;k++)
        System.out.println(used_resource[k]);
    System.out.println("Revenue earned by M-HEU"+" "+netrev());
    serial_rev=netrev();
    if (verify_solution()==0) {
        System.out.println("Error in HEU");
        System.exit(0);
    }
}

```



```

}
void load_equations(Node candidate)
// Loading the MMKP to the equations of the linear programming
{
    int i,j,k,l;

    no_of_variables=0;
    no_of_live_variables=0;
    for(i=0;i<no_of_sessions;i++){
        no_of_variables+=no_of_qos_levels[i];
        if (candidate.status[i]==-1)
            no_of_live_variables+=no_of_qos_levels[i];
    }

    no_of_equations=no_of_sessions-
    candidate.no_of_fixed_group+no_of_resources;
    lhs=new int[no_of_equations+3];
    rhs_var=new int[no_of_live_variables+1];
    rhs_coeff=new
    double[no_of_equations+3][no_of_live_variables+2];
    rhs_coeff[1][1]=0.0;
    k=2; /*Objective Equation*/
    for (i=0;i<no_of_sessions;i++){
        if (candidate.status[i]==-1){
            for (j=0;j<no_of_qos_levels[i];j++) {
                rhs_coeff[1][k]=cost[i][j];
                k++;
            }
        }
    }

    for(i=0;i<no_of_resources;i++){
        rhs_coeff[i+2][1]=total_constraint[i];
        k=2;
        for(j=0;j<no_of_sessions;j++){
            if (candidate.status[j]>=0) rhs_coeff[i+2][1]-
            =resource[i][j][candidate.status[j]];
            else{
                for(l=0;l<no_of_qos_levels[j];l++){
                    rhs_coeff[i+2][k]=-resource[i][j][l];
                    k++;
                }
            }
        }
    }

    k=2;
    l=2;
    for (i=0;i<no_of_sessions;i++){
        if (candidate.status[i]==-1){
            rhs_coeff[k+no_of_resources][1]=1.0;
            for (j=0;j<no_of_qos_levels[i];j++) {
                rhs_coeff[k+no_of_resources][1+j]=-1.0;
            }
            l+=no_of_qos_levels[i];
            k++;
        }
    }
}

```

```

int testforallselection(Node candidate)
{
    int i;
    for (i=0;i<no_of_resources;i++){
        if (rhs_coeff[i+2][1]<0.0) return 0;
    }
    if (candidate.no_of_fixed_group!=no_of_sessions) return 2;
    return 1;
}

int evaluate_node(Node candidate)
// Running Linear programming to find the upper bound of a partial
solution
{

    int i;
    double tmax;

    load_equations(candidate);
    candidate.upper_bound=0.0;
    for (i=0;i<no_of_sessions;i++){
        if (candidate.status[i]>=0)
            candidate.upper_bound+=cost[i][candidate.status[i]]
        ;
    }
    i=testforallselection(candidate);
    if (i==0) {
        return 0;
    }
    else if (i==1){
        return 1;
    }
    }
    simplex(no_of_equations,no_of_live_variables,no_of_resources,0,
no_of_sessions-candidate.no_of_fixed_group);
    if (icase!=0){
        return 0;
    }

    candidate.upper_bound+=rhs_coeff[1][1];
    candidate.next_branch_session=-1;
    tmax=0.0;
    for(i=2;i<no_of_equations+2;i++){
        if (rhs_coeff[i][1]>tmax && lhs[i-
1]<=no_of_live_variables){
            candidate.next_branch_session=lhs[i-1];
            tmax=rhs_coeff[i][1];
        }
    }
    if (candidate.next_branch_session===-1) {
        return 0;
    }
    i=0;
    while(candidate.next_branch_session>0){
        if (candidate.status[i]==-1)
            candidate.next_branch_session-=no_of_qos_levels[i];
        i++;
    }
    candidate.next_branch_session=i-1;
    return 1;
}

```

```

double find_upper_bound()

// Finding the upper bound of the MMKP
{
    Node ptr;
    int i,counter=0;

    solutionNode= new Node();
    head=new Vector();
    ptr=new Node();
    ptr.no_of_fixed_group=0;
    ptr.status=new int[no_of_sessions];
    for (i=0;i<no_of_sessions;i++) ptr.status[i]=-1;
    i=evaluate_node(ptr);
    if (i==0){
        System.out.println("No Feasible Solution");
        return 0.0;
    }
    upper_rev=ptr.upper_bound;
    System.out.println("Revenue earned by Upper"+"
    "+ptr.upper_bound);
    return ptr.upper_bound;
}

void simplx(int m,int n,int m1,int m2,int m3)
// Linear programming algorithm using the simplex method
{
    int i,ir,is,k,kh,m12,n11,n12,jumpv=0;

    if (m != (m1+m2+m3)){
        System.out.println("Bad input constraint counts in
        simplx");
        System.exit(0);
    }
    l1=new int[n+2];
    l2=new int[m+1];
    l3=new int[m+1];

    n11=n;
    for (k=1;k<=n;k++) l1[k]=rhs_var[k]=k;
    n12=m;
    for (i=1;i<=m;i++) {
        if (rhs_coeff[i+1][1] < 0.0){
            System.out.println("Bad input tableau in simplx");
            System.exit(0);
        }
        l2[i]=i;
        lhs[i]=n+i;
    }
    for (i=1;i<=m2;i++) l3[i]=1;
    ir=0;
    if (m2+m3>0) {
        ir=1;
        for (k=1;k<=(n+1);k++) {
            q1=0.0;
            for (i=m1+1;i<=m;i++) q1 += rhs_coeff[i+1][k];
            rhs_coeff[m+2][k] = -q1;
        }
    }
}

```

```

do {
    jumpv=0;
    simpl(m+1,n11,0);
    if (bmax <= TOL && rhs_coeff[m+2][1] < -TOL) {
        icode = -1;
        return; }
    else if (bmax <= TOL && rhs_coeff[m+2][1] <=
    TOL) {
        m12=m1+m2+1;
        if (m12 <= m) {
            for (ip=m12;ip<=m;ip++) {
                if (lhs[ip] == (ip+n)) {
                    simpl(ip,n11,1);
                    if (bmax > 0.0){
                        jumpv=1;
                        break;
                    }
                }
            }
        }
        else
        jumpv=0;
    }
}

    if (jumpv==0){
        ir=0;
        --m12;
        if (m1+1 <= m12)
            for (i=m1+1;i<=m12;i++)
                if (l3[i-m1] == 1)
                    for (k=1;k<=n+1;k++)
                        rhs_coeff[i+1][k] = -
                        rhs_coeff[i+1][k];
        break;
    }
}

    if (jumpv==0){
simpl2(n,n12);
        if (ip == 0) {
            icode = -1;
            return;
        }
    }
    simpl3(m+1,n);
    if (lhs[ip] >= (n+m1+m2+1)) {
        for (k=1;k<=n11;k++)
            if (l1[k] == kp) break;
        --n11;
        for (is=k;is<=n11;is++) l1[is]=l1[is+1];
        ++rhs_coeff[m+2][kp+1];
        for (i=1;i<=m+2;i++) rhs_coeff[i][kp+1] = -
        rhs_coeff[i][kp+1];
    } else {
        if (lhs[ip] >= (n+m1+1)) {
            kh=lhs[ip]-m1-n;
            if (l3[kh]>0) {
                l3[kh]=0;
                ++rhs_coeff[m+2][kp+1];
                for (i=1;i<=m+2;i++)
                    rhs_coeff[i][kp+1] = -
                    rhs_coeff[i][kp+1];
            }
        }
    }
}

```

```

        }
        is=rhs_var[kp];
        rhs_var[kp]=lhs[ip];
        lhs[ip]=is;
    } while (ir!=0);
}
for (;;) {
    simp1(0,nl1,0);
    if (bmax <= 0.0) {
        icode=0;
        return;
    }
    simp2(n,nl2);
    if (ip == 0) {
        icode=1;
        return;
    }
    simp3(m,n);
    is=rhs_var[kp];
    rhs_var[kp]=lhs[ip];
    lhs[ip]=is;
}
}
void simp1(int mm,int nl1,int iabf)
{
    int k;
    double test;

    kp=l1[1];
    bmax=rhs_coeff[mm+1][kp+1];
    for (k=2;k<=nl1;k++) {
        if (iabf == 0)
            test=rhs_coeff[mm+1][l1[k]+1]-(bmax);
        else
            test=Math.abs(rhs_coeff[mm+1][l1[k]+1])-
                Math.abs(bmax);
        if (test > 0.0) {
            bmax=rhs_coeff[mm+1][l1[k]+1];
            kp=l1[k];
        }
    }
}
void simp2(int n,int nl2)
{
    int k,ii,i;
    double qp=0.0,q0=0.0,q=0.0;

    for (i=1;i<=nl2;i++) {
        if (rhs_coeff[l2[i]+1][kp+1] < -TOL) {
            q1=-rhs_coeff[l2[i]+1][1]/rhs_coeff[l2[i]+1][kp+1];
            ip=l2[i];
            for (i=i+1;i<=nl2;i++) {
                ii=l2[i];
                if (rhs_coeff[ii+1][kp+1] < -TOL) {
                    q=-rhs_coeff[ii+1][1]/
                        rhs_coeff[ii+1][kp+1];
                    if (q < q1) {

```

```

        ip=ii;
        q1=q;
    } else if (q == q1) {
        for (k=1;k<=n;k++) {
            qp = -
            rhs_coeff[ip+1][k+1]/rhs_coeff[ip+1][kp+1];
            q0 = -
            rhs_coeff[ii+1][k+1]/rhs_coeff[ii+1][kp+1];
            if (q0 != qp) break;
        }
        if (q0 < qp) ip=ii;
    }
}
}
}

void simp3(int i1,int k1)
{
    int kk,ii;
    double piv;

    piv=1.0/rhs_coeff[ip+1][kp+1];
    for (ii=1;ii<=i1+1;ii++)
        if (ii-1 != ip) {
            rhs_coeff[ii][kp+1] *= piv;
            for (kk=1;kk<=k1+1;kk++)
                if (kk-1 != kp)
                    rhs_coeff[ii][kk] -=
                    rhs_coeff[ip+1][kk]*rhs_coeff[ii][kp+1];
        }
    for (kk=1;kk<=k1+1;kk++)
        if (kk-1 != kp) rhs_coeff[ip+1][kk] *= -piv;
    rhs_coeff[ip+1][kp+1]=piv;
}

public static void main (String argv[]){
    for (int i=1;i<11;i++){
        long sec1, sec2, sec3, sec4;
        nnMMKP b=new nnMMKP();
        b.datainit(500*i,25,25);
        b.find_upper_bound();
        b.init_solution();
        sec1=new Date().getTime();
        b.do_ms_heu();
        sec2=new Date().getTime();
        System.out.println("The MS_HEU Time "+" "+(sec2-sec1));
        sec3=new Date().getTime();
        b.do_heu();
        sec4=new Date().getTime();
        System.out.println("The M_HEU Time"+" "+(sec4-sec3));
        b.write_to_file();
    }
    for (int i=1;i<11;i++){
        long sec1, sec2, sec3, sec4;
        nnMMKP b=new nnMMKP();
        b.datainit(2500,5*i,25);

```

```

b.find_upper_bound();
b.init_solution();
sec1=new Date().getTime();
b.do_ms_heu();
sec2=new Date().getTime();
System.out.println("The MS_HEU Time "+" "+(sec2-sec1));
sec3=new Date().getTime();
b.do_heu();
sec4=new Date().getTime();
System.out.println("The M_HEU Time"+" "+(sec4-sec3));
b.write_to_file();

```

```

}

```

```

for (int i=1;i<11;i++){
long sec1, sec2, sec3, sec4;
nnMMKP b=new nnMMKP();
b.datainit(2500,25,5*i);
b.find_upper_bound();
b.init_solution();
sec1=new Date().getTime();
b.do_ms_heu();
sec2=new Date().getTime();
System.out.println("The MS_HEU Time "+" "+(sec2-sec1));
sec3=new Date().getTime();
b.do_heu();
sec4=new Date().getTime();
System.out.println("The M_HEU Time"+" "+(sec4-sec3));
b.write_to_file();

```

```

}

```

```

}

```

```

}

```

- [12] M. Magazine and O. Oguz, A Heuristic Algorithm for Multidimensional Zero-One Knapsack Problem, *European Journal of Operational Research*, pp 319-326, Vol. 16(3), 1984.
- [13] F. Dammeyer and S. Voss, Dynamic Tabu List Management Using the Reverse Elimination Method, *Annals of Operations Research*, 1991.
- [14] A. Drexel. A Simulated Annealing Approach to the Multiconstraint Zero-One Knapsack Problem. *Annals of Computing*, Vol 40, pp 1-8, 1988.
- [15] S. Khuri, T. Back and J. Heitkotter. The Zero/One Multiple Knapsack Problem and Genetic Algorithms, *ACM Symposium of Applied Computation*, 1994.
- [16] R. Parra-Hernandez and N. Dimopoulos, A new Heuristic for Solving the Multi-choice Multidimensional Knapsack Problem, *IEEE Transaction on Systems, Man and Cybernetics. Part A: Systems and Humans*, 2002.
- [17] M. Moser, D. P. Jokanovic and N. Shiratori, An Algorithm for the Multidimensional Multiple-Choice Knapsack Problem, *IEICE Transactions on Fundamentals of Electronics*, pp 582-589, Vol. 80(3), 1997.
- [18] M. Hifi, M. Michrafy and A. Sbihi, Algorithms for the multiple-choice multi-dimensional knapsack problem, *Journal of the Operational Research Society*, pp. 1323-1332, Volume 55, 2004.
- [19] M. Hifi, M. Michrafy and A. Sbihi, A reactive local search-based algorithm for the multiple-choice multi-dimensional knapsack problem, *Computational Optimization and Applications*, 2003.
- [20] J. Jaja, *An Introduction to Parallel Algorithms*, Addison-Wesley, 1992.
- [21] J. Lin and J. Storer, Processor Efficient Hypercube Algorithm for the Knapsack Problem, *Journal of Parallel and Distributed Computing*, pp. 332-337, Volume 13, 1991.
- [22] J. Lee, E. Shragowitz and S. Sahni, A Hypercube Algorithm for the 0/1 Knapsack Problem, *Journal of Parallel and Distributed Computing*, pp. 438-456, Volume 5, 1988.

- [23] Gopalakrishnan, Ramakrishnan and L. N. Kanal, Parallel Approximate Algorithms for 0/1 Knapsack Problem, *International Conference of Parallel Processing*, pp. 444-451, 1986
- [24] E. Mayr, Parallel Approximation Algorithms, *International Conference on FGCS*, pp. 542-551, 1988.
- [25] H. Chen, S. Chern and H. Jang, Pipeline Architectures for Dynamic Programming Algorithms, *Parallel Computing*, pp. 111-117, Volume 13, 1990.
- [26] S. Teng, Adaptive Parallel Algorithms for Integral Knapsack Problems, *Journal of Parallel and Distributed Computing*, pp. 400-406, Volume 8, 1990.
- [27] D. Morales, L.Roda, C. Rodriguez, F. Almeida and F. Garcia, A Parallel Algorithm for the Integer Knapsack Problem, *EUROPAR 95*, Sweden, 1995.
- [28] A. Goldman and D. Trystram, An Efficient Parallel Algorithm for Solving the Knapsack Problem on the Hypercube, *An Apache Project Supported by CNRS, INRIA, INPG and UJF, CNPq 200590/95-2*, Brazil.
- [29] S. Niar and A. Frevile, A Parallel Tabu Search Algorithm for the 0-1 Multidimensional Knapsack Problem, *11th International Parallel Processing Symposium*, Geneva, Switzerland, pp. 512-516, April 01-05, 1997.
- [30] M. J. Blesa, LI. Hernandez, F. Xhafa, Parallel Skeletons for Tabu Search Method, *8th International Conference on Parallel and Distributed Systems (ICPADS'01)*, IEEE Computer Society Press, pp. 23-28, 2001.
- [31] M. J. Blesa, LI. Hernandez, F. Xhafa, Parallel Skeletons for Tabu Search Method Based on Search Strategies and Neighborhood Partition, *4th International Conference on Parallel Processing and Applied Mathematics (PPAM'01)*, Lecture Notes in Computer Science, Springer-Verlag, pp. 185-193, Volume 2328, 2002.
- [32] A. Z. M. Shahriar, M. A. H. Newton, M. M. Akbar, A Multi-processor based Heuristic for Multi-dimensional Multiple Choice Knapsack Problem, *6th International Conference on Computer & Information Technology*, Dhaka, Bangladesh, pp. 520-525, December 19-21, 2003.

- [33] A. Z. M. Shahriar, *A Multiprocessor Based Heuristic For Multidimensional Multiple-Choice Knapsack Problem*, MSc Thesis Paper, Department of Computer Science and Engineering, Bangladesh University of Engineering and Technology, March 2004.
- [34] R. Armstrong, D. Kung, P. Sinha and A. Zoltners, A Computational Study of Multiple Choice Knapsack Algorithm, *ACM Transaction on Mathematical Software*, pp 184-198, Vol. 9, 1983.
- [35] S. Chatterjee, J. Sydir and B. Sabata and T Lawrence, Modeling Applications for Adaptive QoS – based Resource Management, *2nd IEEE High Assurance Systems Engineering Workshop*, August, 1997.
- [36] P. Ç. Chu and J. E. Beasley, A genetic algorithm for the multidimensional knapsack problem, *Journal of Heuristic*, pp 63-86, Vol. 4, 1998.
- [37] K. Dudziniski and W. Walukiewicz, A Fast Algorithm for the Linear Multiple Choice Knapsack Problem, *Operation Research Letters*, pp 205-209, Vol. 3, 1984.
- [38] M. E. Dyer, J. Walker, Dominance in Multi-Dimensional Multiple-Choice Knapsack Problems, *Asia pacific Journal of Operational Research*, pp 159-168, Vol. 15, 1998.
- [39] R. P. Hernandez, N. Dimopoulos, *A New Heuristic for Solving the Multi-Choice Multidimensional Knapsack Problem*, Technical Report, Department of Electrical and Computer Engineering, University of Victoria, 2002.
- [40] M. Kearns and S. Singh. Near-optimal Reinforcement Learning in Polynomial Time, *International Conference on Machine Learning*, 1998.
- [41] M. Vasquez, Jin-Kao Hao, An Hybrid Approach for the 0-1 Multi Knapsack Problem, *4th Metaheuristics International Conference*, IJCAI-01, Seattle, Washington, pp 221-226, August 2001.

