A Multi-processor Based Heuristic

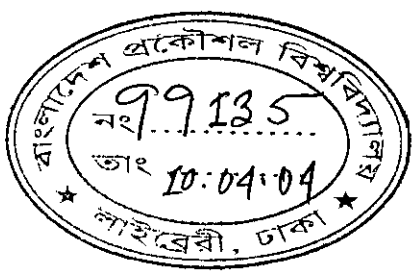
For

Multidimensional Multiple-Choice Knapsack Problem

by

Abu Zafar Mohammad Shahriar

March, 2004



Submitted to

Department of Computer Science and Engineering

Bangladesh University of Engineering and Technology

Dhaka-1000, Bangladesh

in partial fulfillment of the requirement for the degree of

M. Sc. Engineering (Computer Science and Engineering)

A Multi-processor Based Heuristic

For

Multidimensional Multiple-Choice Knapsack Problem

A thesis submitted by

ABU ZAFAR MOHAMMAD SHAHRIAR
Student No. 100105031P
for the partial fulfillment of the degree of
M. Sc. Engineering (Computer Science and Engineering).
Examination held on March 13, 2004.

Approved as to style and contents by:

Dr. Md. Mostofa Akbar Assistant Professor Department of CSE, BUET, Dhaka-1000, Bangladesh.	Chairman
Dr. Md. Shamsul Alam Professor & Head Department of CSE, BUET, Dhaka-1000, Bangladesh.	Ex-officio Member
Dr. M. Kaykobad Professor Department of CSE, BUET, Dhaka-1000, Bangladesh.	Member
Dr. Md. Monirul Islam Assistant Professor Department of CSE, BUET, Dhaka-1000, Bangladesh.	Member
Dr. Md. Zafar Iqbal Professor Department of CSE, Shahjalal University of Science & Technology, Sylhet, Bangladesh.	Member (External)

Certificate

This is to certify that this thesis work has been done by ***Abu Zafar Mohammad Shahriar***, *Student No. 100105031p*, 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.

Abu Zafar Mohammad Shahriar

Dr. Md. Mostofa Akbar
Supervisor of the Thesis

Acknowledgement

First of all, 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. With his profound knowledge, vast experience, invaluable advice and continual encouragement, he guided me in the proper way that helped me to achieve a smooth completion of this thesis work.

I would like to thank the members of the graduate committee, Dr. Md. Shamsul Alam, Professor and Head, Dr. M. Kaykobad, Professor, Dr. Md. Monirul Islam, Assistant Professor, Department of Computer Science and Engineering, Bangladesh University of Engineering and Technology, and Dr. Md. Zafar Iqbal, Professor, Department of Computer Science and Engineering, Shahjalal University of Science and Technology for their valuable suggestions.

I would like to mention the name of Mr. Abdul Hakim Newton, Assistant Professor, Department of Computer Science and Engineering, Bangladesh University of Engineering and Technology, for his all out help to the completion of this thesis.

I used a server of New Media Innovation Centre (NewMIC), Vancouver, B.C., Canada for performing the experiments. I must thank everyone in NewMIC for this valuable help.

I would also like to acknowledge the co-operation and services rendered by the faculty members and staffs of the CSE Department of Bangladesh University of Engineering and Technology and Ahsanullah University of Science and Technology.

Contents

List of Figures

Abstract

In this thesis we have devised a multi-processor based heuristic algorithm for the Multi-dimensional Multiple Choice Knapsack Problem (MMKP). MMKP is a variation of the classical 0-1 Knapsack Problem where there are multiple groups of items and more than one resource is required by each item. Admission control problem in an Adaptive Multimedia Server System can be mapped to MMKP. The problem is NP-hard which is not feasible for the real time admission control problem. So, heuristic solutions have been developed to solve the MMKP. Among all the heuristics, M-HEU developed by Akbar solves the MMKP achieving a reasonable percentage of optimality. This thesis presents an algorithm based on M-HEU using multiple processors. This algorithm runs in time $O(T/p + f(p))$ where T is the time required by the algorithm using single processor, p is the number of processors and $f(p)$, a function of p, is the synchronization overhead. The worst case analysis of the algorithm and the computation of the optimal number of processors are also done.

Chapter 1

Introduction

Knapsack problem and its variants are widely used in formulating different practical problems such as menu planning, resource scheduling, admission control and profit maximization. So, researchers rendered their effort to develop various algorithms for solving those problems. Unfortunately these problems are all NP Hard problems [9] and may be impossible to solve in real time for practical problems. For this reason, faster solution of the knapsack type problems is being studied extensively by the researchers to make it applicable for real time applications. 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 its variant MMKP. MMKP has its application in an Adaptive Multimedia System (AMS) [1, 2, 3]. AMS provides multimedia services to the users. Due to the resource limitation of the servers, it may not be possible to provide services to all the users. So, some form of admission control is required by the system. Quality adaptation is also done in this system. Solutions to the MMKP can be used by the admission controllers in multimedia server systems for admission control and quality adaptation. Several heuristics have been developed to solve the MMKP for the AMS. But, faster algorithms developed for solving the MMKP do not scale better for larger systems. In this thesis we develop a faster solution by using existing multi-processor systems. In a multi-processor system multiple processors can run concurrently. We reduce the time requirement of the existing faster algorithms by dividing computations among

multiple processors that work concurrently. Following show the picked items techniques we used to solve the KP in approximately real time response.

- Applying the heuristic to solve the problem to achieve the suboptimal solution
- Applying parallel algorithms to solve the Knapsack type problems.

1.1 Definition of KP and MMKP

The classical 0-1 KP is a well known problem in the field of computer science. In this problem, items have to be picked up for a knapsack for maximum total value, so that the total weight of the picked items does not exceed the weight capacity of the knapsack. Let there be n items, item i has weight w_i and value v_i. The knapsack has a maximum capacity M. Mathematically, the objective of the KP is to maximize value, $V = \sum_{i=1}^{n} x_i v_i$, where $\sum_{i=1}^{n} x_i w_i \leq M$ and $x_i \in \{0,1\}$ are the picking variables. Figure 1.1 illustrates a KP where maximum value that can be achieved is 23 where the weight capacity of the knapsack is 20.

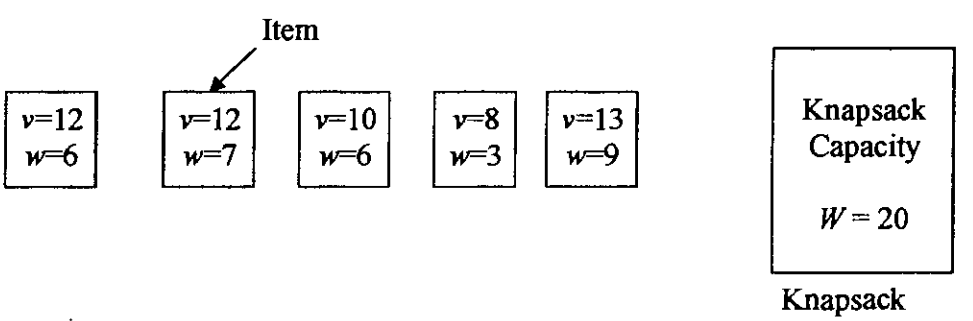


Figure 1.1 An instance of KP

MMKP is a variant of classical 0-1 KP where there are more than one groups of items and resource constraints. Objective of the MMKP is to pick up exactly one item from

each group to maximize some total value while not exceeding the resource constraints. Let there be n groups, Group i containing l_i items, Item j of Group i has a value v_{ij} and m resource requirements denoted by the resource vector, $\vec{r}_{ij} = (r_{ij1}, r_{ij2}, \cdots, r_{ijm})$ and $\vec{R} = (R_1, R_2, \cdots, R_m)$ be the resource constraints. In mathematical notation, the problem is to find value, $V = \text{maximize} \sum_{i=1}^{n} \sum_{j=1}^{l_i} x_{ij} v_{ij}$ subject to $\sum_{i=1}^{n} \sum_{j=1}^{l_i} x_{ij} r_{ijk} \le R_k$, where $k = 1,2,\cdots,m$, $x_{ij} \in \{0,1\}$ and $\sum_{j=1}^{l_i} x_{ij} = 1$. Figure 1.2 illustrates an MMKP with 3 groups and 2 resource requirements. Value 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 while not over consuming any resource i.e. $\sum r_1$ of picked items ≤ 15 and $\sum r_2$ of picked items ≤ 19.

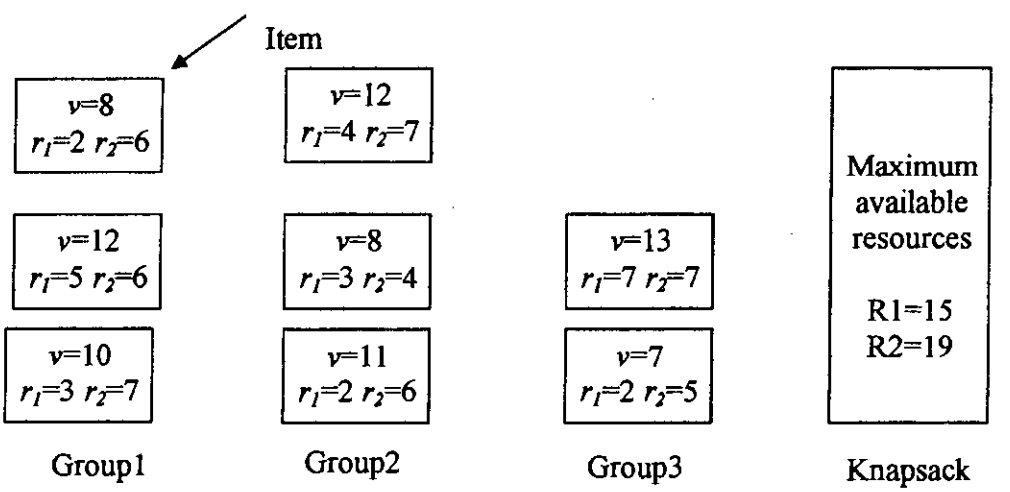


Figure 1.2 An instance of the MMKP

1.2 Motivation

Providing multimedia services is becoming more and more important in the current world. Delivery of multimedia streams from an AMS has been proposed by Akbar [1, 2, 3]. 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. Faced with requests for sessions, admission controller has to consider the following to maximize the revenue that can be earned from the sessions under the resource constraints of the server:

- Whether to admit or reject the new users
- The QoS levels of the newly admitted users
- If QoS levels of sessions already in progress should be upgraded or downgraded to release some of the resources for the new sessions

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. Problem of admission control in an AMS exactly fits an MMKP and can be mapped to the MMKP by mapping users to groups, accepted quality of service profiles of users to the items in a group, server resources (CPU, IO and Network BW, memory) to resources and revenue earned to values [8]. Figure 1.3 shows the mapping of admission control problem to an MMKP with two resources (say, r_1 = CPU time and r_2 = main memory).

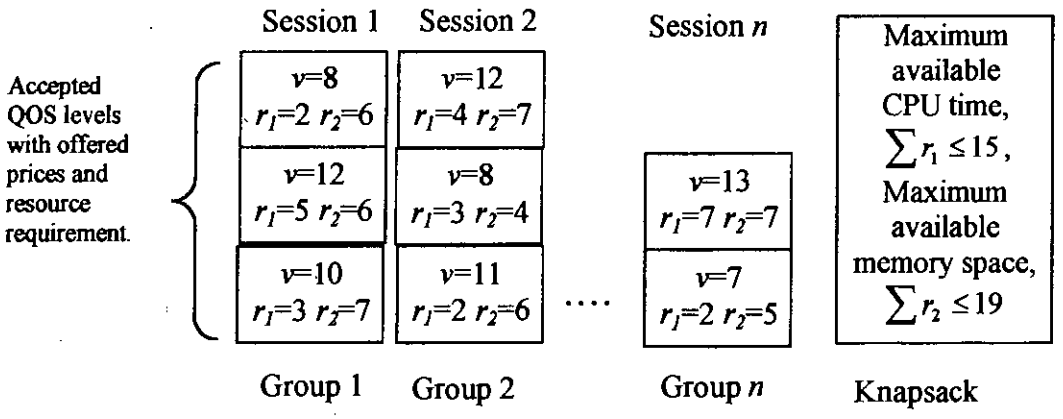


Figure 1.3 Mapping of the admission control problem to the MMKP

In recent times systems with more than one processor is becoming available by virtue of the improvement in VLSI technology and reduced hardware cost. Multi-processing operating systems can run processes concurrently in these systems. An algorithm can be made faster if we can divide computations among processes in these systems. Since admission control is a real-time problem, it requires a solution as fast as possible with satisfactory results. A heuristic solution, M-HEU, to solve the MMKP achieves a satisfactory solution in reasonable amount of time [1, 3]. We find that the time requirement of M-HEU can be further reduced by utilizing multiple processors in a multi-processor system.

1.3 Problem Definition and Previous Work

Admission controller in an AMS needs the solution of the MMKP, since the admission control problem in an AMS can be mapped to the MMKP. Finding an exact solution to MMKP is an NP-hard problem [9] which is not applicable to the real-time admission control problem. That is why heuristic solutions have been developed for

solving the MMKP. Khan developed a heuristic, HEU, based on the concept of aggregate resource consumption which achieves 93% of the optimal solution value [8, 14]. Later Akbar *et al.* in [2] presented a modification of HEU, M-HEU, which finds the solution achieving 96% optimality with the worst-case time complexity $O(mn^2l^2)$, where n = number of groups, l = number of items in each group and m = resource dimension. Another heuristic, C-HEU, also developed by Akbar finds the solution by constructing a convex hull and achieves 90% optimality with worst-case time complexity $O(nl \log nl + nlm)$ [1, 3]. So M-HEU is the best 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. So, we need a faster algorithm to achieve the real-time response in admission control. We find that the time requirement for M-HEU can be further reduced if computations are shared by multiple processes in a multiprocessor machine. But using multiple processes for M-HEU requires the following:

- Efficient synchronization of the processes
- Efficient inter-process communication

In this thesis we present a multi-processor based heuristic by implementing the steps of M-HEU using multiple processes. This algorithm does computations faster than M-HEU in a multi-processor system but achieves the same optimality.

1.4 Scope and Focus

The main focus of this thesis is to present a faster version of M-HEU that exploits the multi-processor based systems. Computations required by M-HEU can be shared among multiple processes in a multi-processor based system that works in

asynchronous mode. Development of algorithms for multi-processor based systems, where unlimited number of processors work in synchronous mode (PRAM model [6]), is beyond the scope of this thesis. To evaluate the performance of the algorithm we use a quad-processor machine running UNIX operating system. We observe the performance of the algorithm and the overhead of using multiple processes by varying different parameters

1.5 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 has been done. We describe here the algorithms related to the KP and its variants. Parallel algorithms for the KP and its variants are also descried in this chapter. We concluded this chapter by giving a detail description of the heuristic, M-HEU, for solving the MMKP.

Chapter 3 presents the multi-processor based heuristic to solve the MMKP. First two sections of this chapter describe the use of multiple processes in the heuristic and the problem of using parallel processes. Then the algorithm is described in the subsequent sections 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 done at the end of this chapter.

We conclude the thesis in Chapter 5 by describing the major contributions of this research. Some future research works have also been suggested in this chapter.

Chapter 2

Literature Review of Knapsack Algorithms

In this chapter we briefly describe some of the previous algorithms that were designed to solve different variants of the KP. We also discuss the details of the heuristic algorithm, M-HEU, for the MMKP.

2.1 Algorithms of KP, MDKP and MCKP

There exist two types of algorithms to solve the KP and its variants: one type is for obtaining exact solution and the other is for obtaining approximate solution.

2.1.1 Exact algorithms for KP, MDKP and MCKP

Dynamic Programming or Branch and Bound approach can be used to obtain optimal solutions to the classical 0-1 KP [5]. Dynamic Programming method uses sequence of decisions, regarding whether to pick an item or not, leading to an optimal solution. On the other hand, branch and bound method searches the state space by generating a search tree iteratively to find the solution. A node in the search tree represents a solution state where there may be some variables whose values are known and that of some others' may be unknown. This method starts with a single-node tree where the values of all the variables are unknown. A node of this tree may be explored based on a variable whose value is unknown at the current node. 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 largest upper bound

having no unknown variable is the solution node. But these approaches take exponential time to find an exact solution.

MDKP is a variant of KP where the resource is multidimensional. Shih [12] presented an algorithm for exact solution using Branch and Bound Linear Programming technique. For upper bound estimation, this algorithm treats MDKP 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.

MCKP is another variant of KP where there are multiple groups of items and exactly one have to be picked from each group. Branch and Bound Linear Programming Technique for MCKP were proposed by Nauss [10].

2.1.2 Approximate solutions to the variants of KP

Different heuristics have also 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 left.

The greedy like approach can be generalized for other variants of KP. Toyoda [14] proposed a simple solution to the MDKP using the concept of aggregate resource. In his algorithm the main idea was to penalize the free items depending in the current resource usage. 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. This 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.

2.2 Algorithms for the MMKP

MMKP is actually the combination of the two other variants of KP: the MDKP and the MCKP. As usual there are two ways of finding solutions for an MMKP: one is a method for finding exact solutions and the other is heuristic that achieves approximate solutions. In the following subsections we summarize both types of algorithms for solving the MMKP.

2.2.1 Algorithm for exact solution

Finding exact solutions to the MMKP is an NP hard problem [8, 9]. Khan [7] presented an exact algorithm for the MMKP using the Branch and Bound Linear Programming (BBLP) technique. In this method a search tree is generated iteratively. A node of the tree is expanded by selecting an item of a particular group, called branching group. At a node, if the items of a group are not selected is called free group. Initially there is only one node and all the groups are free. An estimate of optimum total value is determined by applying linear programming technique on the free groups of a node. Linear programming technique is also used to determine the branching group of a node. In each iteration the node with the highest upper bound is explored. The nodes which do not give any solution value using linear programming are considered as infeasible. These nodes are deleted from the tree.

A solution is found when a node without any free group has the maximum estimated total value. This method of finding exact solution to MMKP is not applicable to real-

time admission control problem and non real-time large problem sets due to its exponential time requirement

2.2.2 Heuristic solutions to the MMKP

Since method of finding exact solution to the MMKP is not applicable to the real-time admission control problem, heuristic solutions have been developed. HEU, a heuristic developed by Khan, finds the solution of the MMKP using the concept of aggregate resource consumption [8]. Later Akbar presented a modification of HEU, M-HEU, which achieves a better of optimality than HEU [1, 3]. C-HEU, another heuristic developed by Akbar, provides solution to the MMKP in logarithmic worst-case time complexity [1]. But the optimality achieved by this heuristic is much inferior to the other two heuristics. So, we do not describe C-HEU in this literature. Table 2.1 summarizes the performance and worst case time complexity of different single processor based heuristics.

Heuristics	Average percentage of optimality achieved from typical data sets(%)	Worst case time complexity
C-HEU	90	$O(nl\log nl+nlm)$
HEU	93	$O(mn^2l^2)$
M-HEU	96	$O(mn^2l^2)$

Table 2.1: Different heuristic solutions for the MMKP

2.3 Parallel Algorithms for KP

Parallel algorithms for KP have also been developed by several researchers [4]. With the advent of parallel processors many researchers concentrated their effort to the development of efficient parallel algorithms for KP. In the previous sections, we described two different approaches for solving the KP optimally. Dynamic Programming (DP) is one of them. A DP algorithm for 0-1 KP may run in $O(nc\log(p)/p + n\log p)$ for p processors, where c is the capacity of the knapsack [4]. A pipeline-architecture containing a linear array of p processors has a time complexity $O(nc/p + m)$ [4]. A divide and conquer approximation algorithm with a time complexity $O(\log^2(n)\log(c))$ on $(nc\sqrt{c})$ processors is presented in [4]. Another approximation algorithm was realized on the hyperbolic architecture with a time complexity $O(\log^2(n)\log(c))$ on $O(nc^2)$ processors [4]. But no parallel or multi-processor based algorithm has been reported in the literature for the approximate solution to the MMKP. The steps of M-HEU (to be described in the next section) must be executed in sequence since the results of a step is a prerequisite for starting the next step. So, the steps of the M-HEU can not be executed in parallel although the computations within each step can be done in parallel. This restricts the development of a poly-log parallel algorithm for the M-HEU. Newton *et al.* [11] developed a poly-log parallel heuristic for the MMKP whose percentage of achieved optimality is less than that of M-HEU. So we propose a multi-processor version of M-HEU in Chapter 3 that gives a solution faster than M-HEU but with an upper bound of number of processors.

2.4 Details of M-HEU

M-HEU is a modified version of HEU. So we will start by describing HEU first. HEU starts by sorting the items of each group in non-decreasing order of the values associated with each item. This is a preprocessing step for HEU. So the items with higher values are on the top in each group. Picking a higher-valued or a lower-valued item than the currently selected one is called upgrading or downgrading, respectively. If a particular pick of items does not satisfy the resource constraints it is defined as infeasible solution. A feasible solution is one that satisfies the resource constraints. HEU then selects the lowest valued item of each group as the initial solution. If this solution is not feasible HEU terminates notifying "No Solution Found". Otherwise it tries to upgrade the solution using Step 2 described below.

M-HEU 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 found by HEU with one upgrade followed by one or more downgrade. This is because there might be some higher-valued items in the MMKP, selection of which makes the solution infeasible, but if some of the other groups are downgraded a feasible solution may be obtained. M-HEU starts by selecting the lowest valued item of each group as the initial solution and iterates through the following steps:

Step 1: If the initial solution is feasible it goes to Step 2. If the initial solution is infeasible then it tries to find a feasible solution by upgrading. For upgrading it calculates the change in aggregate resource consumption of the jth item of Group i,

$\Delta a_{ij} = \sum_k \left(r_{i\rho[i]k} - r_{ijk}\right) \times C_k$, where C_k is the kth resource consumption, $\rho[i]$ is the index of the currently selected item of the ith group. This step tries to find a higher valued item of a group than the currently selected item. This is done by selecting the item which releases some of the infeasible resources and does not make any feasible resource infeasible. There might be more than one item like this. The item which gives the highest Δa_{ij} is selected. If no such item is found then this step notifies "No solution found".

Step 2: This step is analogous to the hill climbing approach in classical AI problems. After a feasible solution is found in Step 1 this step tries to upgrade the solution iteratively by selecting a higher valued item of a group than the currently selected one while retaining feasibility. As the selection criteria it calculates the relative change in aggregate resource consumption, $\Delta a_{ij}^r = \{a, b\}$, where

$$a = \frac{\Delta v_{ij}}{\Delta a_{ij}} \quad and \quad b = 0 \quad if \quad \Delta a_{ij} \le 0$$
$$a = \Delta a_{ij} \quad and \quad b = 1 \quad if \quad \Delta a_{ij} > 0$$
$$\Delta v_{ij} = v_{i\rho[i]} - v_{ij}$$

and

$$\Delta a_{ij}^r > \Delta a_{kl}^r \quad if \quad b\left(\Delta a_{ij}^r\right) > b\left(\Delta a_{kl}^r\right) \quad or \quad \left(b\left(\Delta a_{ij}^r\right) = b\left(\Delta a_{kl}^r\right) \quad and \quad a\left(\Delta a_{ij}^r\right) > a\left(\Delta a_{kl}^r\right)\right).$$

This step tries to find a higher valued item of a group than the currently selected one that retains feasibility and gives the highest Δa_{ij}^r. It iterates as long as this kind of upgrading is possible.

Step 3: This step is analogous to the escaping technique to get rid of the local maxima found in Step 2. It tries for iterative improvement of the solution using one infeasible upgrade followed by one or more downgrade to make the solution feasible. The upgrading selects a higher valued item of a group than the currently selected one. As the selection criteria it calculates $\Delta v_{ij}/\Delta a'_{ij}$, where $\Delta a'_{ij} = \sum_k \frac{r_{i\rho[i]k} - r_{ijk}}{R_k - C_k}$, the scaled change of resource consumption with respect to available resource. In this step an item of a group is selected that produces the highest $\Delta v_{ij}/\Delta a'_{ij}$. If no such item is found for upgrading then the last solution found in Step 2 is the final solution and the algorithm terminates. After the successful upgrading this step goes for one or more downgrading. For downgrading it selects a lower valued item of a group than the currently selected one. It calculates $\Delta a''_{ij}/\Delta v_{ij}$ for selection, where $\Delta a''_{ij} = \sum_k \frac{r_{i\rho[i]k} - r_{ijk}}{C_k - R_k}$, the scaled change of resource consumption with respect to over consumed resource. Then the item of a group is selected that gives the highest $\Delta a''_{ij}/\Delta v_{ij}$ and keeps the gain (improvement achieved in this step by the infeasible upgrade) positive. If this selection makes the solution feasible then the algorithm goes back to Step 2 again. If the solution is still infeasible after the downgrade this step goes for another downgrade. If no such downgrade is possible and the solution is still infeasible then the algorithm terminates and the solution found in Step2 is the final solution.

The algorithm based on the heuristic achieves a solution value which is on average 96% of the exact solution value (See [1, 3] for the details experimental results). The worst case time complexity of this algorithm is $O(mn^2l^2)$. So M-HEU is the best as

far as percentage of optimality is considered while the worst case time complexity is quadratic. So our interest is confined to modifying the heuristic that reduces time requirements even further.

Chapter 3

Multi-processor based heuristic for the MMKP

In this chapter we introduce a new heuristic by dividing the computations of the steps of M-HEU among several processors. The use of multiple processors for efficient concurrent processing and the problems of using multiple processors for asynchronous concurrent processing are discussed as well. Calculation of the optimal number of processors is followed by the analysis of the algorithm.

3.1 Use of multiple processes in the heuristic

In a multi-processing environment 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 we divide computations among a number of p' processes, where p' is less than the number of processors p then turnaround time for our 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 of Step 1, 2 and 3. So, in the long run this

overhead may sum up to a significant amount. In our algorithm we have divided the groups of MMKP among multiple processes for computations.

In our algorithm we require the processes ready for operations before it gets into the details of the processing. Let us call these processes as child processes. At the time of creation of the processes we assign each process a unique number called *id*. The groups are equally divided among the processes. The range of groups on which a process is working is determined according to the *id*s given to them. Since the child processes work asynchronously a separate process called the master process co-ordinates all the activities of all the processes including the creation of the child processes and assigning *id*s to them. Figure 3.1 shows the division of n groups among p processes and also the interaction of processes. When everything is set the master process signals the child processes to start their processing. Each of these child processes selects a candidate item from the items of their assigned range of groups. Let us call this candidate item local candidate. When a child process is done with

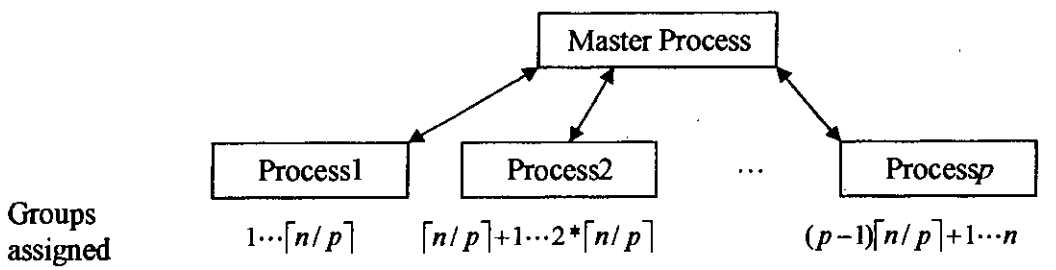


Figure 3.1 Use of processes in the multi-processor based heuristic

its job it signals the master process along with the results of the processing and holds back being ready for the next iteration. When all the child processes are done, the master process starts its selection of a global candidate item from the local candidates

selected by the child processes. After it is finished with its job it may signal the child processes again to start processing for the next iteration or terminates the child processes. Figure 3.2 shows the selection of a candidate item for upgrading or downgrading in the multi-processor based heuristic.

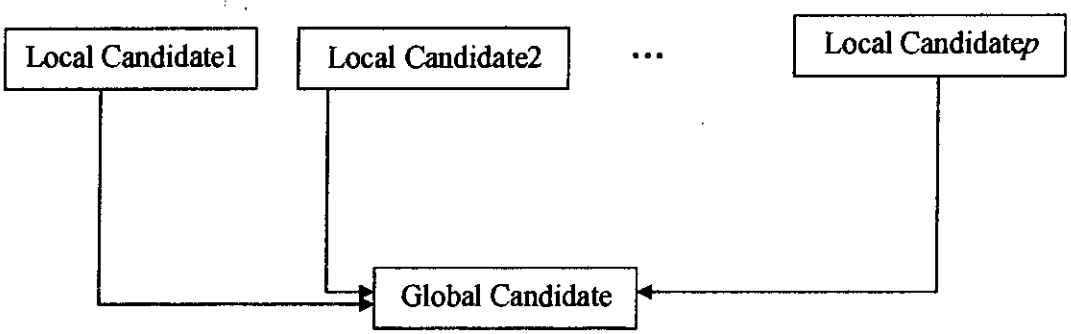


Figure 3.2 Selection of a candidate item in the multi-processor based heuristic

3.2 Problems of Using Parallel Processes

Our algorithm uses multiple processes that work in asynchronous mode. So, one of the processes synchronizes the others. Moreover, the processes also need to communicate with the synchronizer process. So synchronization and communication among processes are required. These cause overhead. This type of synchronization and communication is required repeatedly in our algorithm. So while using multiple processes to reduce the time requirement of the heuristic we need the following:

- Efficient synchronization of the processes
- Very fast inter process communication

In our experiment we used the most efficient method of synchronization and inter process communication. For synchronization purpose we used *semop*() and *semctl*()

functions. These functions were used to lock or unlock the semaphores for synchronization purpose. For inter-process communication *shmget*() and *shmat*() function were used to create and manage shared memory.

Also while processes are working some of the processes may stay idle because they have done their job earlier than the other processes in a particular iteration. Figure 3.3 shows a possible scenario of the processors' active/idle states in iterations of the algorithm. In the worst case this may cause ill-utilization of the processors. But on the average this is a very unlikely situation when a processor is idle for long. Balancing the load of the processors is beyond the scope of this thesis. This will introduce a new overhead measure. Moreover, if the processors are already busy with some other tasks then the performance of the heuristic may degrade. In this case number of processes used has to be reduced.

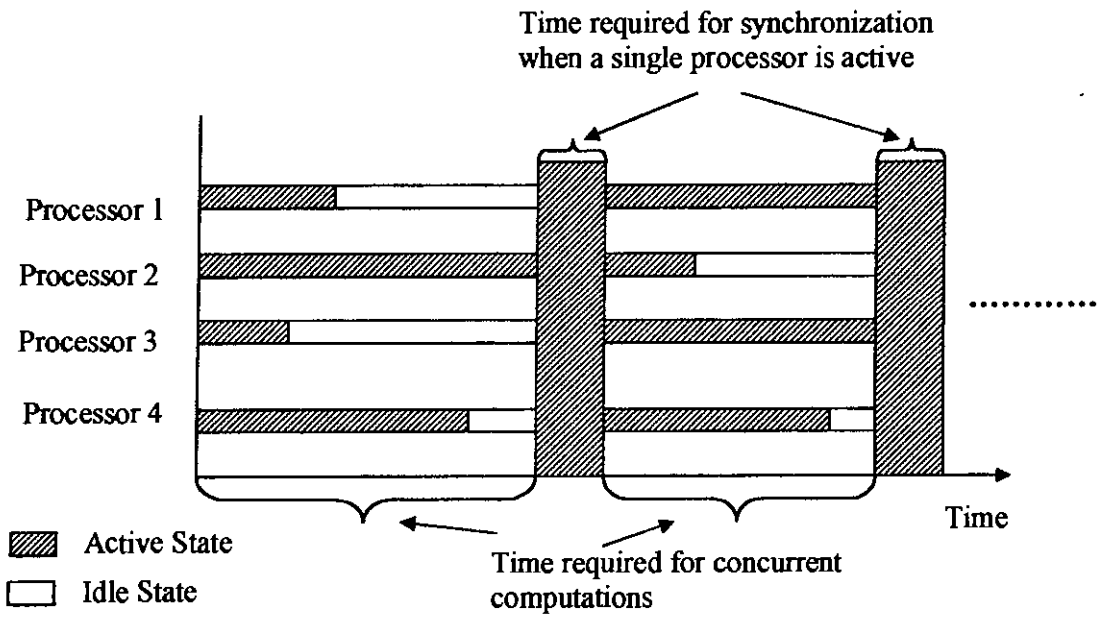


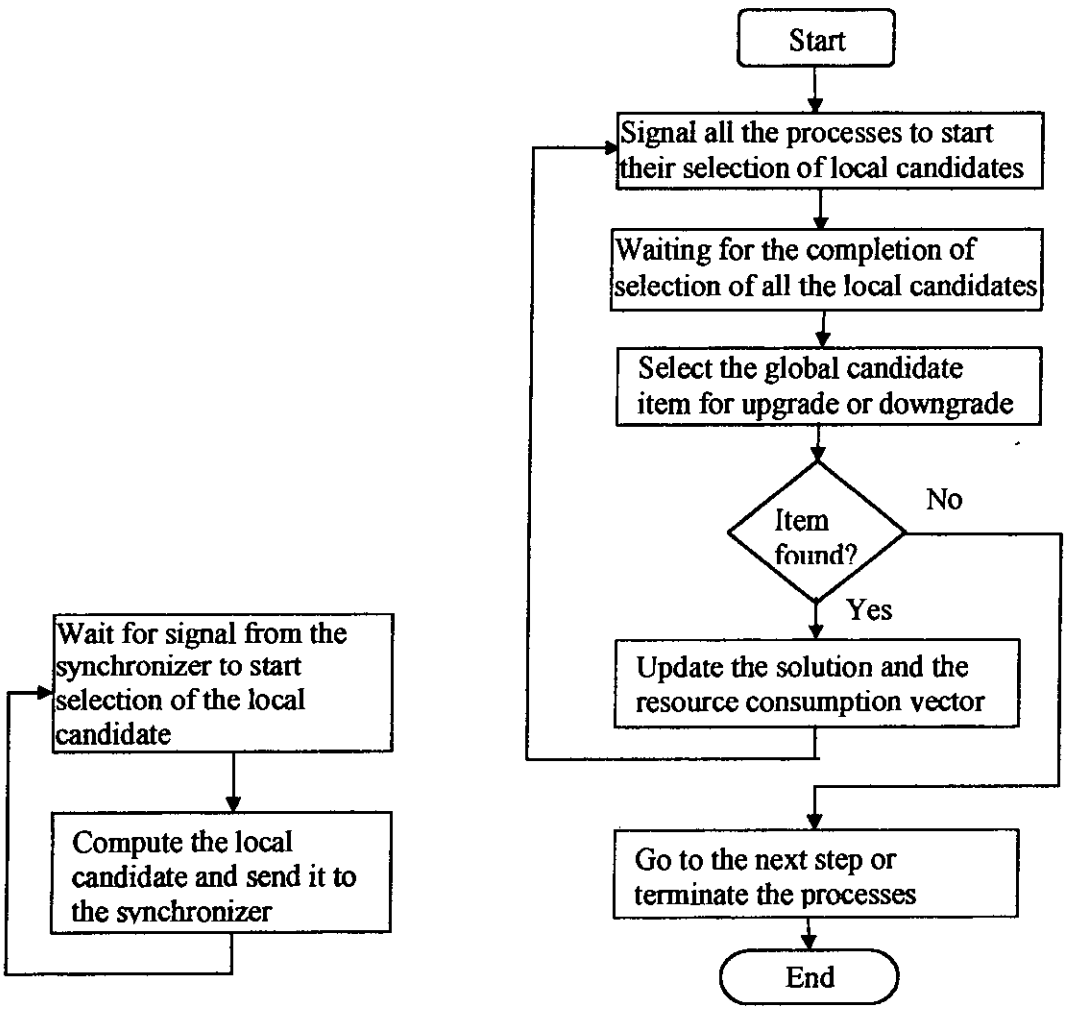
Figure 3.3 A scenario of processors' possible active/idle states in iterations of the algorithm.

3.3 Multi-processor Based Heuristic

In this section we describe the steps of the multi-processor based heuristic followed by an example that demonstrates the steps of the algorithm.

3.3.1 Description of the multi-processor based heuristic

The flowchart of the multi-processor based heuristic is given in Figure 3.4 followed by the description of the steps shown in the flowchart. We divide the flowchart into two parts.



(a) A process computing a local candidate

(b) Synchronizer process

Figure 3.4 Flowchart of the multi-processor based algorithm

Figure 3.4(a) shows the flowchart for the child processes and the major steps are as follows:

Step 1) The child processes enters this state where they wait for a signal from the master process. When the signal is received from the master process it goes to step 2.

Step 2) A local candidate item for upgrading is selected and the results are sent to the master process. A range of groups are assigned to a particular process for computations. An item of a group is selected locally for upgrading from those groups. Selection criteria for finding an item for upgrading are described in Section 2.4. The group number, the item number and the Δa_{ij}^{r} values are sent to the master process.

Figure 3.4(b) shows the flowchart for the Master processes and the major steps are as follows:

Step 1) Signal all the child processes to start their computation.

Step 2) Wait until all the child processes complete their computations. Each of the child processes signals the master process when it completes its job. When the master process receives signal from all the child processes it goes to Step 3.

Step 3) Compute the global candidate item for upgrading from the results received in Step 2. This involves finding the group number and the item number of the item that gives the highest Δa_{ij}^{r}.

Step 4) If an item is found for upgrading then go to Step 5. If no item is found for upgrading then go to Step 6.

Step 5) Make necessary changes to the solution vector using the group number and the item number of the item found in Step 3. Also update the resource

consumption vector by adding the resource consumption of the selected item and subtracting the resource consumption of the previously selected item. Then go to Step 1 again for the next iteration.

Step 6) Transfer control to another step of the M-HEU (e.g. from feasible upgrade step to infeasible upgrade step) or terminate the child processes to terminate the algorithm.

Following are the variables and procedures to describe the algorithm in detail.

no_of_process: The total number of processes used to solve the MMKP

n: The total number of groups in the MMKP

m: The total number of resources in the MMKP

i: A variable indicating the group number

l_i: The number of items in Group i

j: A variable indicating the item number

k: A variable indicating the resource dimension number

r_{ijk}: The kth resource requirement of the jth item of the ith group

v_{ij}: The value of the jth item of the ith group

C_k: The amount of kth resource consumed by the selected items of the groups

R_k: The total amount of kth resource in the MMKP

current_sol: The solution vector containing the indices of the currently selected items of each group

saved_sol: The solution vector containing the indices of the selected items of each group before going to Step 3

start_group: The variable containing the starting group number of the range of groups assigned to a process

end_group: The variable containing the ending group number of the range of groups assigned to a process

local_candidate_item: The vector containing the item numbers of groups selected by the processes

local_candidate_group: The vector containing the group numbers selected by the processes

local_candidate_mdelp: The vector containing the Δa^r_{ij} s computed by the processes

local_candidate_mdelr: The vector containing the values of $\Delta v_{ij}/\Delta a_{ij}$, $\Delta v_{ij}/\Delta a'_{ij}$, $\Delta a''_{ij}/\Delta v_{ij}$ computed by the processes

calculation_type: The variable indicating the type of calculation to be performed by the processes. It can only take values which are from the set *{FEASIBLE_UPGRADE, INFEASIBLE_UPGRADE, FEASIBLE_DOWNGRADE}*

gain: The increase of total value for an infeasible upgrade

UTILITY: A function returning the total value of the selected items

sema: The vector containing the semaphore values for each process

change_selction(i, j): Assign value *j* to the *i*th position of the vector *current_sol* and returns the increase in total value for this selection

feasible_change(i, j): Returns whether *change_selection(i, j)* satisfies all resource constraints

feasible(): Returns whether the current solution satisfies all resource constraints

unlock_process(id: integer): Start the process numbered id to select an item of a group from its assigned range of groups

lock_process(id: integer): Pause the process numbered *id*

wait_for_all(): Make a process to wait till all processes' completion

find_global_upgrade_feasible_item(): Selects an item of a group from the items of the groups selected by the processes locally that gives the highest Δa^r_{ij}

find_global_upgrade_infeasible_item(): Selects an item of a group from the items of the groups selected by the processes locally that gives the highest $\Delta v_{ij}/\Delta a'_{ij}$

find_global_downgrade_item(): Selects an item of a group from the items of the groups selected by the processes locally that gives the highest $\Delta a''_{ij} / \Delta v_{ij}$

find_upgrade_feasible_item(start_group: integer, end_group: integer): Selects an item of a group from the items of the groups ranging from *start_group* to *end_*group that gives the highest $\Delta a'_{ij}$

find_upgrade_infeasible_item(start_group: integer, end_group: integer): Selects an item of a group from the items of the groups ranging from *start_group* to *end_*group that gives the highest $\Delta v_{ij} / \Delta a'_{ij}$

find_downgrade_item(start_group: integer, end_group: integer): Selects an item of a group from the items of the groups ranging from *start_group* to *end_*group that gives the highest $\Delta a''_{ij} / \Delta v_{ij}$

procedure M-HEU()

//This is the main procedure to solve the heuristic

1. *current_sol* ← lowest valued items from each group
2. *while true do*
3. *feasible_upgrades()*
4. *save_solution()*
5. *if infeasible_upgrade() = false then*
6. Solution found
7. *Return*
8. *end if*
9. *do*
10. *status* ← *feasible_downgrade()*
11. *while status=INFEASIBLE_DOWN*

```
12.     if status= NO_DOWN then
13.         revive_solution()
14.         Solution found
15.         return
16.     end if
17. end while
18. end procedure
```

procedure feasible_upgrades()

// This procedure upgrades the solution keeping the solution feasible. It uses the processes to

// select the item of a group.

```
1.  calculation_type ← FEASIBLE_UPGRADE
2.  do
3.      for p= 1 to no_of_process do
4.          unlock_process(p)          // resumes the process with id p that was paused
                                          earlier
5.      end for
6.      wait_for_all()                 // Wait at this point for all the child processes to
                                          complete their assigned job
7.      (candidate_group, candidate_item) ← find_global_upgrade_feasible_item()
8.      if candidate_group ≠ null then
9.          change_selection(candidate_group, candidate_item)
10.     end if
11. while candidate_group ≠ null
12. end procedure
```

procedure infeasible_upgrade() returns boolean

/* This procedure finds an infeasible upgrade of the solution of the MMKP by upgrading one item of one group. If no such item is found it returns false otherwise it returns true */

1. *calculation_type* ← *INFEASIBLE_UPGRADE*
2. *for p= 1 to no_of_process do*
3. *unlock_process(p)* // resumes the process with id *p* that was paused earlier
4. *end for*
5. *wait_for_all()* // Wait at this point for all the child processes to complete their assigned job
6. *(candidate_group, candidate_item)* ← *find_global_upgrade_infeasible_item()*
7. *if candidate_group* ≠ *null then*
8. *return false*
9. *else*
10. *gain* ← *change_selection(candidate_group, candidate_item)*
11. *end if*
12. *return true*
13. *end procedure*

procedure feasible_downgrade() returns downgrade_type

/* This procedure tries to find a feasible solution of the MMKP by downgrading an item of a group no more than *gain* from an infeasible solution. It returns *downgrade_type* which is an enumerated type with the following values:

FEASIBLE_DOWN: A downgrade found that makes the solution feasible with a positive *gain*

INFEASIBLE_DOWN: A downgrade found with a positive *gain* and the solution is still infeasible

NO_DOWN: No downgrade found keeping the *gain* positive*/

```
1.  calculation_type ← FEASIBLE_DOWNGRADE
2.  for p= 1 to no_of_process do
3.      unlock_process(p)          // resumes the process with id p that was paused
                                   earlier
4.  end for
5.  wait_for_all()                 // Wait at this point for all the child processes to
                                   complete their assigned job
6.  (candidate_group, candidate_item) ← find_global_downgrade_item()
7.  if candidate_group ≠ null then
8.      gain ← gain + change_selection(candidate_group, candidate_item))
9.      if feasible() then
10.         return FEASIBLE_DOWN
11.     else
12.         return INFEASIBLE_DOWN
13.     end if
14. end if
15. return NO_DOWN
16. end procedure
```

procedure do_computations(id: integer)

/* This procedure selects an item of a group locally from the items of the assigned range of groups. All processes running in parallel use this procedure for local selections. It does the selection for feasible upgrade, infeasible upgrade and feasible downgrade. */

```
1.  start_group ← ⌈n/no_of_process⌉ × (id − 1) + 1
2.  if id = no_of_process then
3.      end_group ← n
4.  else
```

5. $end\_group \leftarrow \left\lceil n/no\_of\_process \right\rceil \times id$

6. *end if*

7. *while true do*

8. *lock_process(id)* // the child process pauses itself at this point until resumed by the synchronizer

9. *case*

10. *calculation_type is FEASIBLE_UPGRADE*

11. *find_upgrade_feasible_item(start_group, end_group)*

12. *calculation_type is INFEASIBLE_UPGRADE*

13. *find_upgrade_infeasible_item(start_group, end_group)*

14. *calculation_type is FEASIBLE_DOWNGRADE*

15. *find_downgrade_item(start_group, end_group)*

16. *end case*

17. *end while*

18. *end procedure*

procedure lock_process(id: integer)

/* This procedure pauses the activity of the process numbered *id* */

1. *sema[id]* ← *sema[id]* - 1 //normally the semaphore values are set to 1 which indicates that the semaphores are free. By setting the values to 0 semaphores are locked.

2. *end procedure*

procedure unlock_process(id: integer)

/* This procedure resumes the activity of the process numbered *id* */

1. *sema[id]* ← *sema[id]* + 1 //When the value of the semaphore is 0, it is locked. By setting the value of the semaphore to 1 it is

unlocked.

2. *end procedure*

procedure wait_for_all(id: integer)

/* This procedure pauses the activity of the process numbered *id* until all other processes are paused*/

1. *while not sema[id] = 0* //The value of the semaphore (*sema*[*id*]) is set to $-p$ before entering this procedure. Each of the processes increases the value of this semaphore by unlocking it. When all the p processes unlock the same semaphore its value becomes 0 and the conditional value in line 1 is false.

2. *end while*

3. *end procedure*

3.3.2 An example demonstrating the Step 2 of the algorithm

Consider the following problem instance of the MMKP for our example:

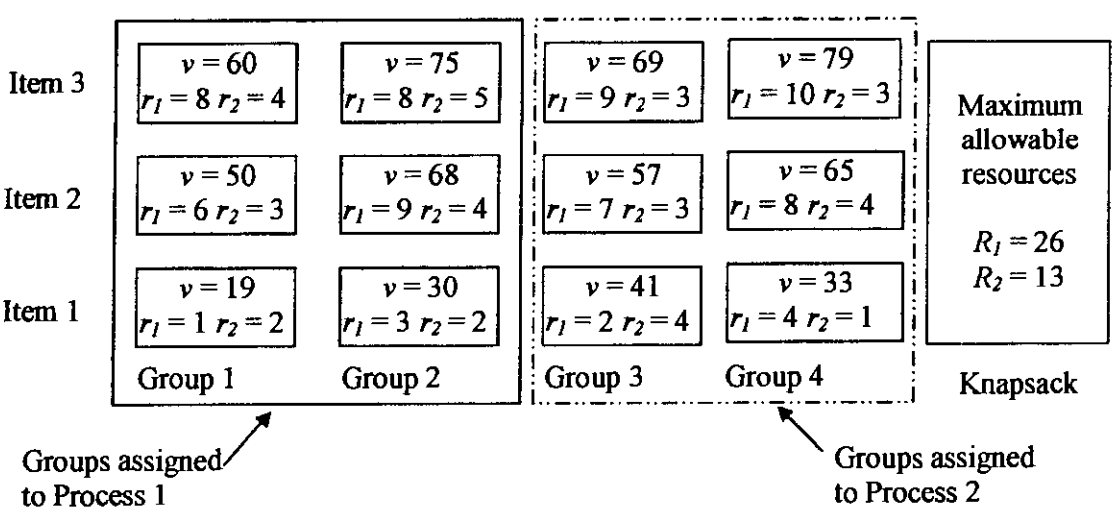


Figure 3.5 An instance of the MMKP

Figure 3.5 is an MMKP with 4 groups. Each group has 3 items sorted according to their values. The resource is two dimensional. Item 1 of each group is selected as the initial solution. Thus the solution vector can be written as (1, 1, 1, 1) 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 (10, 9) where *i*th element denotes the *i*th resource consumption. Number of processes used for this example is 2. Process 1 is assigned to work with Group 1 and 2 where as Process 2 is assigned to work with Group 3 and 4. Assignment of the groups is also shown in Figure 3.3 by bordering the groups separately using different line styles. Groups assigned to Process 1 are bordered using solid line and the groups assigned to Process 2 are bordered using dashed line.

Feasible Upgrades:

Values of Δa^r_{ij} for all the feasible upgrades from the currently selected items calculated by Process 1 are as follows.

$\Delta a^r_{12} = \{0.53,0\}$ $\Delta a^r_{13} = \{0.47,0\}$

$\Delta a^r_{22} = \{0.49,0\}$ s $\Delta a^r_{23} = \{0.58,0\}$

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

$$\Delta a_{12} = \sum_{k=1}^{2} \left(r_{1\rho[1]k} - r_{12k}\right) \times C_k$$
$$= \left(r_{111} - r_{121}\right) \times C_1 + \left(r_{112} - r_{122}\right) \times C_2$$
$$= (1-6) \times 10 + (2-3) \times 9$$
$$= -59$$

Since $\Delta a_{12} < 0$, $b = 0$ and $\Delta v_{12} / \Delta a_{12}$ have to be calculated where

$$\Delta v_{12} = \Delta v_{1\rho[1]} - \Delta v_{12}$$
$$= 19 - 50$$
$$= -29$$

So $\Delta v_{12}/\Delta a_{12} = 0.53$ and hence $\Delta a^r{}_{12} = \{0.53,0\}$.

Now among the values of $\Delta a^r{}_{ij}$ s calculated by Process 1, $\Delta a^r{}_{23}$ is the highest. So Process 1 selects the Item 3 of Group 2 locally. These values are conveyed to the master process for global selection.

Values of $\Delta a^r{}_{ij}$ s for all the feasible upgrades from the currently selected items calculated by Process 2 are as follows.

$\Delta a^r{}_{32} = \{0.39,0\}$ $\Delta a^r{}_{33} = \{0.46,0\}$

$\Delta a^r{}_{42} = \{0.48,0\}$ $\Delta a^r{}_{43} = \{0.59,0\}$

Now Process 2 selects Item 3 of Group 4 as the candidate for upgrade as it gives the highest $\Delta a^r{}_{ij}$ locally. These values are also passed to the master process by Process 2 for global selection.

Since $\Delta a^r{}_{43}$ is the highest among the two $\Delta a^r{}_{ij}$ s selected locally master process selects the Item 3 of Group 4 for upgrading. It then upgrades the solution to (1, 1, 1, 3) and the resource usage is (16, 11).

Since a feasible upgrade is found is this iteration, Step 2 goes for another iteration to find another feasible upgrade.

In the next iteration for finding feasible upgrade, Process 1 calculates the following $\Delta a^r{}_{ij}$ s:

$\Delta a^r{}_{12} = \{0.34,0\}$ $\Delta a^r{}_{13} = \{0.31,0\}$

$\Delta a^r{}_{22} = \{0.32,0\}$

So, Item 2 of Group 1 is selected by Process 1 locally.

Process 2 calculates the following $\Delta a^r{}_{ij}$ s:

$\Delta a^r{}_{32} = \{0.23,0\}$ $\Delta a^r{}_{33} = \{0.28,0\}$

So, Item 3 of Group 3 is selected by Process 2.

Among the two local selections $\Delta a'_{12}$ is the highest. So Group 1 is upgraded to Item 2. Hence the new solution vector is (2, 1, 1, 3) and the resource consumption vector is (21, 12).

Step 2 goes for yet another iteration as a feasible upgrade was found in the last iteration. In this iteration only candidate item is $\Delta a'_{13} = \{0.19,0\}$, computed by Process 1. So, this item is selected globally for upgrading and the new solution vector becomes (3, 1, 1, 3) with the new resource consumption vector (23, 13).

In the next iteration of Step 2 no feasible upgrade is possible. So the solution is saved and Step 3 is entered.

3.4 Analysis of the multi-processor based algorithm

We only present the worst case analysis of Step 2, since this step does the most of the computations and thus dominates in the time complexity of the algorithm [4]. As the available resource is very small in Step 3 this step requires less iteration. Empirical results show that Step 3 doesn't have significant impact on the time required by the algorithm. We also calculate the optimal number of processors required to minimize the time requirement. For the convenience of the analysis we assume all the groups have the same number of items *i.e.* $l_1 = l_2 = l_3 = \cdots\cdots\cdots = l_n = l$. We also assume that n is an integer multiple of the total number of processes, p.

3.4.1 Synchronization overhead

Worst case scenario of the feasible upgrade step (Step 2) occurs when this step starts from the lowest valued item of each group as the initial solution, upgrading one level

of one group in each iteration and finally returns the solution consists of the highest valued item of each group. This requires $n(l-1)$ iterations. Activities of the processes have to be synchronized in each of these iterations of Step 2. Selection of a global candidate from the local candidates is also performed along with synchronization. Let the combined computational time for these operations be proportional to c floating point operations per process. So, the total time required by the synchronization activities for p processes is proportional to the following number of operations:

$$\sum_{l=1}^{n(l-1)} pc \quad = n(l-1)pc \quad \cdots\cdots\cdots\cdots\cdots\cdots (3.1)$$

3.4.2 Worst case computational complexity of Step 2

For the selection of the local candidate item in the first iteration, the number of Δa_{ij}^r s required to be computed by each concurrently running process is $O(n(l-1)/p)$. After upgrading a particular group, number of Δa_{ij}^r s computations in the next iteration reduces by one for the process to which that group belongs and the others have to do the same number of computations as was done in the previous iteration. The situation is the worst when no upgrading takes place in the range of groups pertaining to a particular process until all other groups are upgraded to their highest valued items. It may end up with the situation that no other process except one process has to do any computation for the last $n(l-1)/p$ iteration of this step. In this case for each of the first $n(l-1)(p-1)/p$ iterations, $O(n(l-1)/p)$ computations are required by one of the process that does the highest number of computations. For the last $n(l-1)/p$ iterations number of computations will reduce by one in each iteration for that process while all the other process have no computations to do. In each iteration of this step

computation of Δa_{ij}^{r}, feasibility testing for the selection of an item and selection of an item requires $(6m+4)$ floating point operations.

When all the local candidate items selected by the processes are available master process has to select the global candidate item for upgrading. Time required for this has been included in the synchronization overhead calculation for convenience. After selecting an item for upgrading $2m$ floating point operations are required to find the new resource consumption vector which is required for the computation of the Δa_{ij}. Hence the total time required by this step is proportional to the following number of floating point operations:

$$\sum_{j=1}^{n(l-1)}\left\{\frac{n(l-1)}{p}\times(6m+4)+2m\right\}-\sum_{j=1}^{n(l-1)/p-1}j(6m+4)+Synchronization\ Overhead$$
$$=\left\{\frac{n^2(l-1)^2(2p-1)}{2p^2}+\frac{n(l-1)}{2p}\right\}\times(6m+4)+2mn(l-1)+n(l-1)pc \quad \cdots\cdots\cdots\cdots\cdots\cdots(3.2)$$

So the worst case time complexity of this step is $O(mn^2l^2/p)$.

In the preprocessing step sorting the items of each group requires $O(l^2)$ operations. So if n groups are divided equally among p concurrently running processes then the time required for sorting $O(nl^2/p)$. So the overall time complexity of the algorithm is $O(mn^2l^2/p)$.

3.4.3 Additional computation in multi-processor based algorithm

It may be the case that it is nearly but not truly the worst case for Step 2 when a single processor is used. But for the same problem instance it may be the worst case when multiple processors are used. This is because in each iteration of Step 2 of single-

processor based algorithm, computations of $\Delta v_{ij}/\Delta a_{ij}$ s are not required for the remaining items once Δa_{ij} become non-negative. Let on the average k ($0 < k \leq 1$) is the fraction of items for which computation of $\Delta v_{ij}/\Delta a_{ij}$ is needed. So the total floating point operations required for finding $\Delta v_{ij}/\Delta a_{ij}$ is $\sum_{i=1}^{n(l-1)} 3n(l-1)k$ because for each item, 3 floating point operations are required – one subtraction for computing Δv_{ij}, one division for $\Delta v_{ij}/\Delta a_{ij}$ and one comparison for maximum. But in the multi-processor based algorithm calculation of $\Delta v_{ij}/\Delta a_{ij}$ may not be exempted and one of the processor might require to calculate $\Delta v_{ij}/\Delta a_{ij}$ for all the assigned items. So floating point operations in the worst case for finding $\Delta v_{ij}/\Delta a_{ij}$ will be $\sum_{i=1}^{n(l-1)} \frac{3n(l-1)}{p}$.

Therefore the total time required by these additional computations for p processors is proportional to

$$\sum_{i=1}^{n(l-1)} \left\{ \frac{3n(l-1)}{p} - \frac{3n(l-1)k}{p} \right\} = \frac{3n^2(l-1)^2}{p}(1-k) \quad \text{.....(3.3)}.$$

3.4.4 Optimal number of processors

Number of operations required for the synchronization activities increases linearly with the number of processors. We call it synchronization overhead. If the number of processors is limited, as is in multi-processor machines available in practice, this overhead is negligible. If the number of processors increases this overhead also increases though the overall time requirement may decrease due to the division of computations among processors. But at some point increasing the number of processors won't decrease the time requirement any further as the overhead begins to dominate. We find an expression based on Step 2 that gives us an idea about the

number of processors beyond which no further decrease in the time requirement is possible. For the simplicity of the mathematical model we simplify the expression found in section 3.4.2 assuming $\frac{2p-1}{2p^2} \approx \frac{1}{p}$ and the optimal number of processor becomes,

$$p = \sqrt{\frac{(6m+4)}{c}\left\{n(l-1)+\frac{1}{2}\right\}} \quad \text{.....(3.4)}$$

Value of p in equation (3.4) minimizes the expression (3.2) found in section 3.4.2. As c is a constant for a particular system and l, m are not expected to change much, this expression can be simplified and optimal number of processors will actually be proportional to the square root of the number of groups. The graphs of Figure 3.6 and 3.7 show the plot of optimal number of processors vs. number of groups and the plot of optimal number of processors vs. number of items in each group respectively. We do not show here the plot of optimal number of processors vs. number of resource dimensions because the number of resources in a server is small and not going to change much in practice.

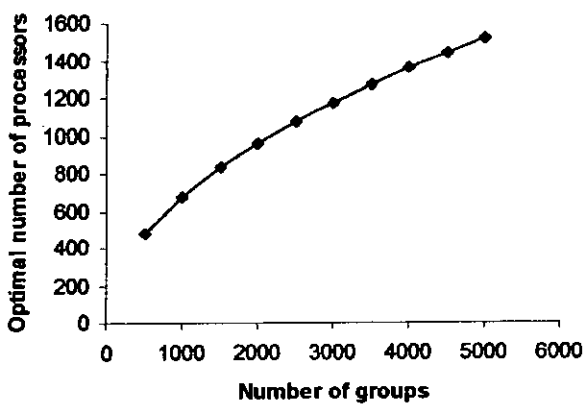


Figure 3.6: Plot of optimal number of processors vs. Number of groups for the MMKP with $l = 25$, $m = 25$ and $c = 8$ using equation (3.4)

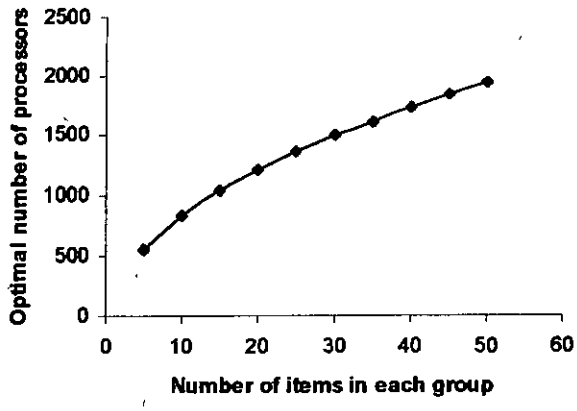


Figure 3.7: Plot of optimal number of processors vs. Number of items in each group for the MMKP with $n = 4000$, $m = 25$ and $c = 8$ using equation (3.4)

3.4.5 Average case scenario

Though the total number of iterations required by Step 2 is $O(nl)$ in the worst case, it is far less in the average case. In this section we make an effort to give an explanation for this. At the initial stage when there are sufficient amount of resources available, generally a higher valued item in a group is selected in each iteration because $\Delta v_{ij}/\Delta a_{ij}$ is higher for higher valued items as C_k is very small resulting into negative low valued Δa_{ij}. Therefore within a few number of iteration the selected items are almost the topmost items in each group and it does not depend on the number of items in a group. That is why number of iterations is $O(n)$ instead of $O(nl)$. Also resources are consumed due to the upgradation in iterations. So, the number of feasible items for which $\Delta a^r{}_{ij}$ s have to be calculated is reduced as available resources are reduced with the upgradations in iterations. So the overall time complexity on the average is $O(n^2lm)$ instead of $O(n^2l^2m)$. This is also evidenced by the experimental results that we present in Chapter 4.

Chapter 4

Experiments

In this chapter we show the results of the experiments done to evaluate the run-time performance of the multiprocessor based heuristic. Results are expressed here using graphs. Analysis of the results are also made and summarized at the end of this chapter

4.1 Experimental Setup

To evaluate the performance we implemented the multi-processor based algorithm using the *C* programming language and ran the algorithm in a quad-processor machine running UNIX SysV. We used semaphores and shared memory for synchronization and inter-process communication respectively because they are the fastest way for these purposes [13]. 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 that was used for generating data for M-HEU [1, 3, 8]. 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 cost per unit resource

R_i = Total amount of the ith resource = $n \times R_c \times 0.5$. Here $R_c \times 0.5$ amount resource is assumed for each session.

P_i = Cost of the ith resource = $Random(P_c)$ = A uniform discrete random number from 0 to (P_c - 1).

r_{ijk} = The kth resource of the jth item of the ith group = $Random(R_c)$. As the total resource for each session is $R_c \times 0.5$, we can say approximately 50% solutions are infeasible because there is a chance that items with resource consumption between $R_c \times 0.5$ to R_c may be infeasible.

v_{ij} = Value of the jth item of the ith 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.

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

For the experimental results reported in this article, we used the following values: Rc = 10 and Pc = 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. Two categories of data sets were used for M-HEU. In one kind of data sets item values are not correlated with item's resource consumption. In the other kind of data sets item values are correlated with the resource requirement. Issues related to the performance of M-HEU using those two categories of data sets were discussed in [1, 3]. We are only concerned with the reduction in time requirement with increasing number of processors. We observed the performance of the multi-processor based heuristic using both kinds of data sets. But,

correlated data sets are more practical in real world problems. So, in this thesis we present only the experimental results using correlated data sets.

4.2 Experimental Results

We presented the experimental results in the graphs. The graphs of Figure 4.1, 4.3 and 4.5 show the time requirement of the algorithm for different number of processors with the increase in the number of groups, number of items in each group and number of resource dimensions. The graphs of Figure 4.2, 4.4 and 4.6 show the time requirement of the algorithm for different number of groups, number of items in each group and number of resource dimensions with the increase in the number of processors. The graphs of Figure 4.7, 4.8 and 4.9 shows the overhead required for different number of processors with the increase in the number of groups, number of items in each group and number of resource dimensions. It is to be mentioned here that for the overhead calculation we used the following expression:

$$\text{Overhead} = \text{Time required using } p \text{ processor} - \frac{\text{Time required using 1 processor}}{p}.$$

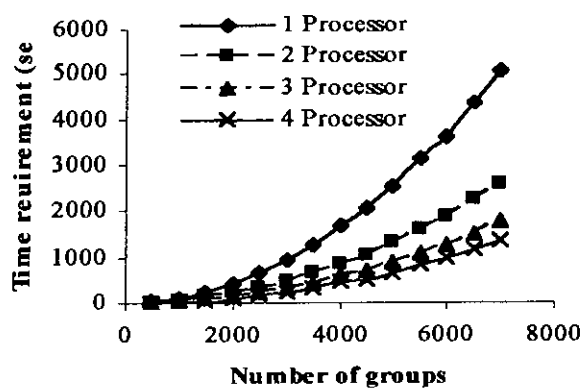


Figure 4.1 Time required by the multi-processor based heuristic for different number of processors for the MMKP data sets with l=25 and m=25

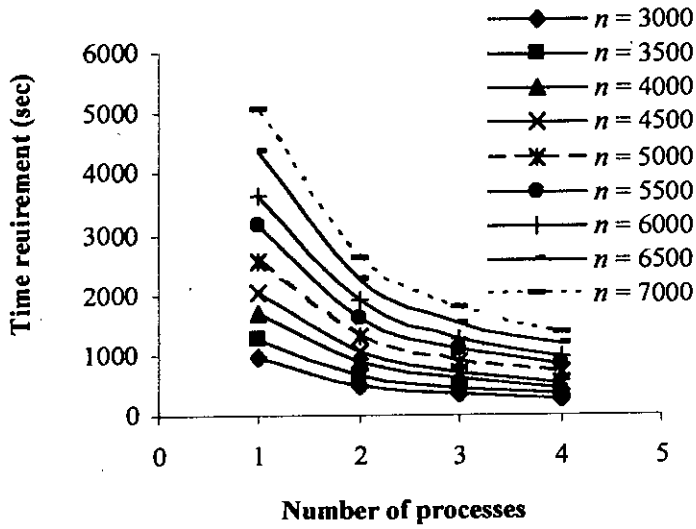


Figure 4.2 Time required by the multi-processor based heuristic for different number of groups for the MMKP data sets with l=25 and m=25

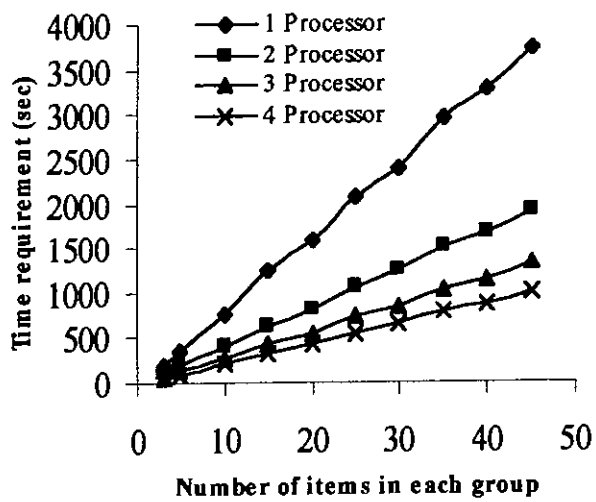


Figure 4.3 Time required by the multi-processor based heuristic for different number of processors for the MMKP data sets with n=4000 and m=30

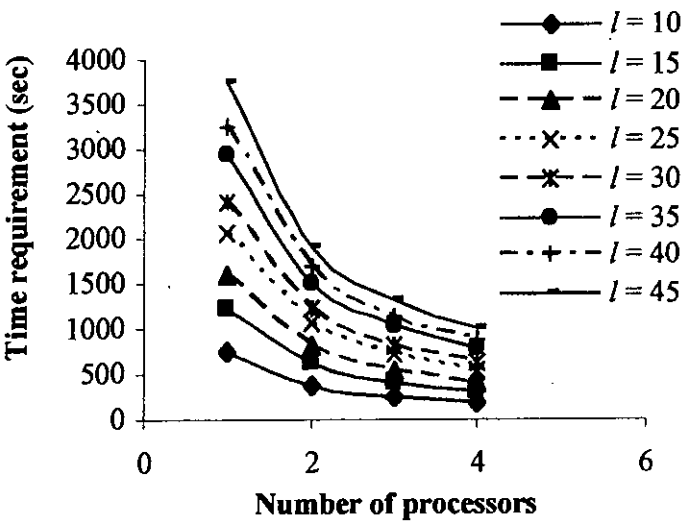


Figure 4.4 Time required by the multi-processor based heuristic for different number of items in each group for the MMKP data sets with n=4000 and m=30

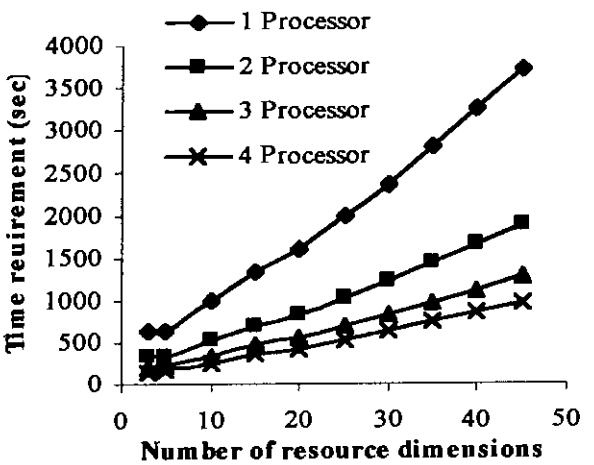


Figure 4.5 Time required by the multi-processor based heuristic for different number of processors for the MMKP data sets with n=4000 and l=30

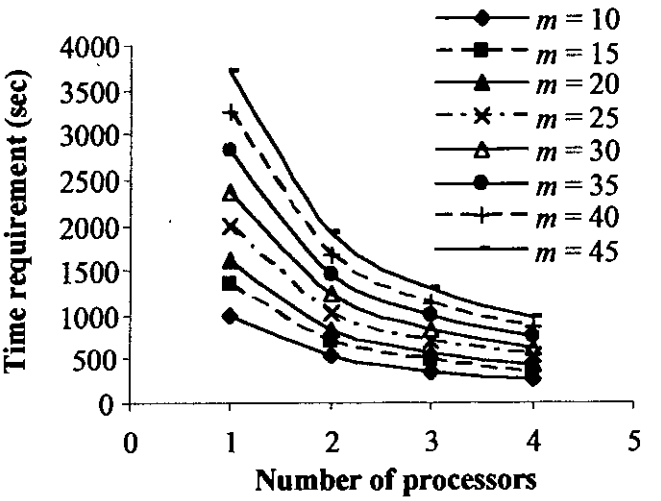


Figure 4.6 Time required by the multi-processor based heuristic for different number of resource dimensions for the MMKP data sets with n=4000 and l=30

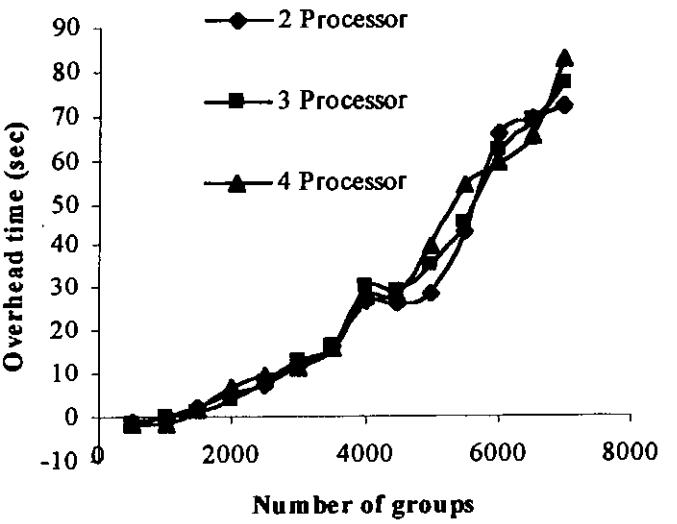


Figure 4.7 Overhead for the multi-processor based heuristic using different number processors for the MMKP data sets with l=25 and m=25

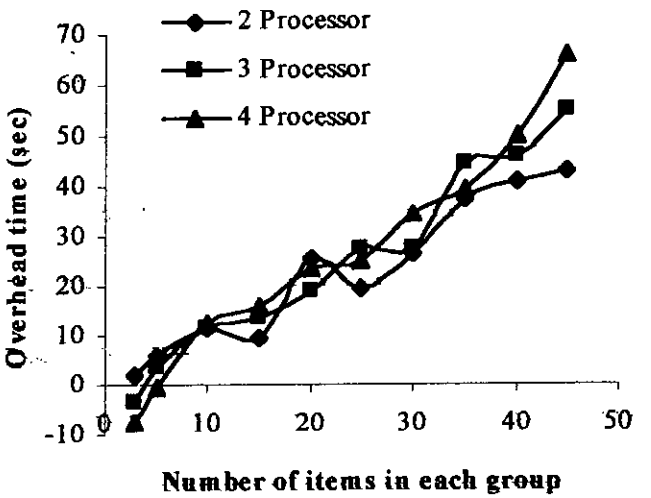


Figure 4.8 Overhead for the multi-processor based heuristic using different number processors for the MMKP data sets with n=4000 and m=30

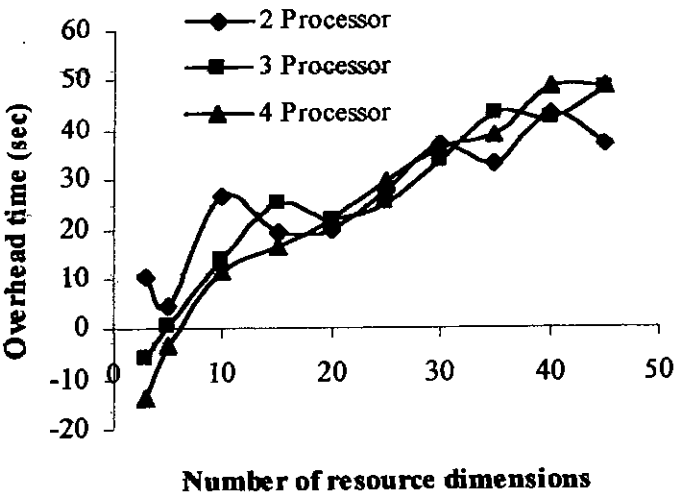


Figure 4.9 Overhead for the multi-processor based heuristic using different number processors for the MMKP data sets with n=4000 and l=30

4.3 Analysis of the Results

Following observations can be made from the graphs presented in the previous section:

- Figure 4.1, 4.3 and 4.5 show that time requirement of the algorithm decreases with the increase in the number of processors and these are expected results. Figure 4.2, 4.4 and 4.6 also show that time requirement of the algorithm is roughly inversely proportional to the number of processors. But as equation 3.1 in section 3.4.4 shows, increasing the number of processors will also increase the synchronization overhead and increasing the number of processors beyond a certain limit will cause this overhead to dominate over the reduction achieved by adding the extra processors. In the graphs presented here, this point is not shown. Here we could only show a part of the left side of the curves due to the limited number of processors. Figure 4.10 shows a sample curve, drawn using the equation derived in Section 3.4.2, that shows this feature.

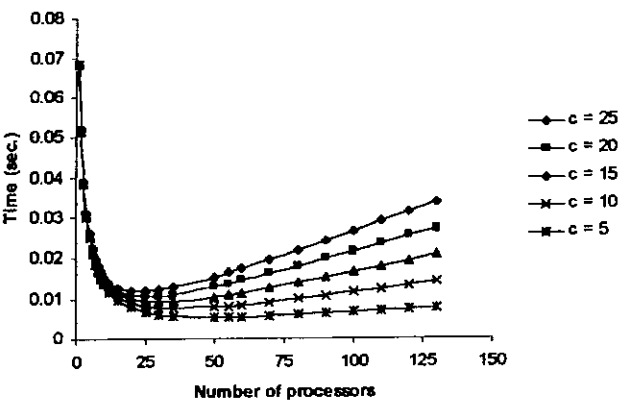


Figure 4.10 Sample curves showing theoretical results for different values of synchronization overhead per process for the multi-processor based heuristic with $n = 100,\ m = 5, l = 5$.

- Figure 4.7 and 4.8 shows that synchronization overhead is increasing with the increase of number of groups and number of items in each group. Although the overhead should remain constant with the variation of number of resource dimensions, the graph of Figure 4.9 shows it increases with the number of resource dimensions. This is because our overhead calculation method, as shown below, introduces an element that is directly proportional to the number of resource dimensions. The calculation of the overhead from the experimental data is as follows:

  Substituting $p = 1$ in equation 3.2, we get time required for 1 processor,

$$T_1 = \left\{\frac{n^2(l-1)^2}{2} + \frac{n(l-1)}{2}\right\} \times (6m+4) + 2mn(l-1) + n(l-1)c \quad \cdots\cdots(4.1)$$

  From equation 3.2 we get time required for p processor,

$$T_p = \left\{\frac{n^2(l-1)^2(2p-1)}{2p^2} + \frac{n(l-1)}{2p}\right\} \times (6m+4) + 2mn(l-1) + n(l-1)pc \quad \cdots\cdots(4.2)$$

  So, from equation 4.1 and 4.2 we find the overhead for p processors,

$$T_p - \frac{T_1}{p} = \frac{n^2(l-1)^2(p-1)}{2p^2} \times (6m+4) + 2mn(l-1)\left(1-\frac{1}{p}\right) + n(l-1)c\left(p-\frac{1}{p}\right) \quad \cdots\cdots(4.3)$$

  So the term $2mn(l-1)\left(1-\frac{1}{p}\right)$ causes the overhead to increase linearly with the increase of resource dimensions.

- Figure 4.7, 4.8 and 4.9 show that overhead is negative for some smaller values of n, l and m. We can give some argument to support these results. There are two types of memory used for runtime data storage – one is the *main* memory (shared by all processors and relatively slower) and the other is the *cache* memory (each processor has its own cache memory and relatively faster).

When a processor processes data it loads data from main memory to the cache memory. Since amount of cache memory is much smaller than that of main memory, all the required data from the main memory can not be loaded into the cache memory at the same time. So, data has to be transferred back and forth in batches repeatedly. This type of memory management activity takes time. When the values of n, l and m are smaller, although memory management activity causes overhead if single processor is used, but due to the division of data among the processors no overhead is required for memory management when multiple processors are used. This is because each processor needs to process a smaller part of data that can be loaded in its cache memory entirely and no transferring of data is required from main memory to cache memory repeatedly. So the use of single processor requires some extra overhead that dominates the synchronization overhead and makes the overall overhead negative. As the values of n, l and m are increased further, memory management overhead also starts to take effect in multi-processor version and at this point synchronization overhead also starts to dominate too due to increased values of n, l and m. So the overhead becomes positive.

- Although theoretically the overhead increases linearly with the n, Figure 4.7 shows a little quadratic characteristic because it actually plots the synchronization overhead plus the time required by the additional computations mentioned in Section 3.4.2 (equation 3.3).

- In Figure 4.4, 4.5 and 4.6 overhead-curves for different number of processors overlap. This is because they plot the synchronization overhead plus the time

required by the additional computations mentioned in Section 3.4.2 (equation 3.3). And the expression of the additional computations includes a parameter k $(0 < k \leq 1)$ which completely depends on the distribution of data and is explained more detailed in Section 3.4.2.

- In Section 3.4.4 we made an effort to explain the average case scenario which reveals that on the average time required is linearly proportional to the number of items in each group rather than quadratic. This is also evidenced by Figure 4.3.

Optimal number of processors is yet to be calculated empirically since it requires a machine with large number of processors. But the equation given in Section 3.4.3 can be used to find the optimal number of processors for the worst case situation. The number obtained from this equation is fairly high even for a smaller problem instance. This is because the synchronization overhead is very low compared to the time required for computations and thus the curves obtained from this equation is roughly a hyperbola. We calculated the time requirement for higher number of processors using the equation 3.2 and plotted the data demonstrated in Figure 4.10. The curves obtained from the experimental data are relatively flatter than the curves obtained from the equation. The reason for this is that the number of operations required in the average case is far less than that of the worst case. Though the optimal number of processors required to minimize the time requirement is fairly large, after the use of a certain number of processors no significant reduction in time requirement is achieved. This is shown in the figure by drawing a vertical line. On the right side of the vertical line the curves are almost horizontal. The value of number of processors beyond this

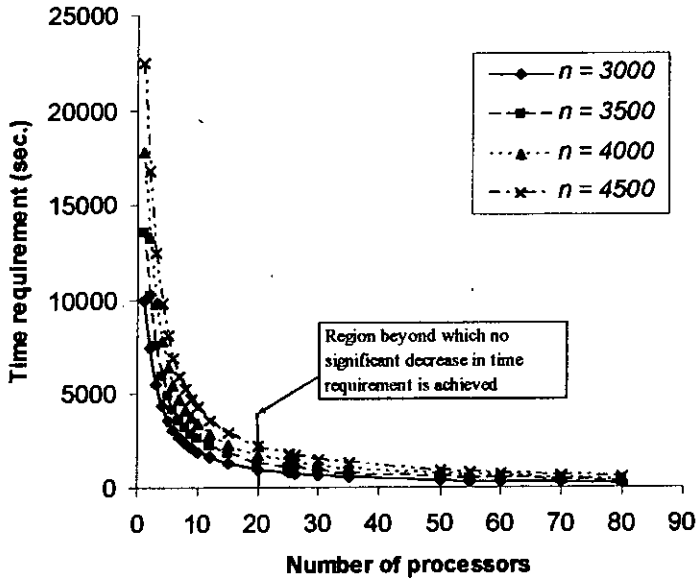


Figure 4.11 Estimated time required by the algorithm for different number of groups for the MMKP data sets with l=25 and m=25

vertical line is not of practical interest as increasing the number of processors at this point does not produce useful amount of reduction in computation time.

Chapter 5

Conclusion

In this chapter we summarize the major contributions from this thesis and present suggestions for the future research work.

5.1 Major contributions

The major contributions made in this thesis are as follows:

- A more scalable algorithm has been produced to solve the MMKP that makes the best use of available processors. When the number of groups increases beyond a certain limit single processor based solutions may not be able to provide real-time response. By using more than one processor this limit can be extended. We have made an effort to devise a more scalable algorithm that uses the available processors as per the time requirement and the problem size.
- The experiments reveal the nature of the synchronization overhead and some advantages that is gained by dividing data among processors.
- We also provide an analytical solution for the calculation of the optimal number of processors.

5.2 Future Research Work

We suggest the following research works on the algorithm to solve the MMKP:

- *Empirical value for the optimal number of processors*: The optimal number of processors is yet to be calculated empirically that requires a system with large

number of processors. In this thesis we have provided an analytical solution to obtain the value of optimal number of processors for the worst case situation. But we could not provide any practical value for the optimal number of processors, since we could only use a machine with four processors. Experimental results show that number of computations required on the average is far less than that is required in the worst case. So, the value of optimal number of processors should be smaller than that can be found using Equation 3.4. An expression to find the value of the optimal number of processors for the average case can be obtained by running the algorithm in a machine with large number of processors. Thorough analysis of the results will be required for obtaining the expression. Equation 3.4 can provide a guideline for obtaining the equation to find optimal number of processors in practice. It may be the case that no significant decrease in time requirement can be achieved after using a certain number of processors. This number of processors may be smaller than the value of the optimal number of processors. Research issues should also include this point of significance.

- *Parallel algorithms to solve the MMKP*: Parallel algorithms for PRAM model can be developed. We developed an algorithm that uses limited number of processors working asynchronously. In PRAM model unlimited number of processors works in synchronous mode. In each major steps of M-HEU, upgrading or downgrading is done in one group iteratively. These major steps of M-HEU can not be executed in parallel. So, it is not possible to design a poly-log parallel algorithm for the M-HEU directly. But the upgrading or downgrading can be done in more than one group simultaneously. This may

degrade the average percentage of optimality achieved and lead to ill utilization of the resources. While developing poly-log parallel algorithms these issues have to be considered.

- *Experiments on other systems*: In this thesis we have done experiments on a particular system with limited number of processors. By virtue of memory management techniques, we got some unexpected reduction in time requirement. The implementation of semaphores in different systems may affect the synchronization overhead. Methods of inter-process communication will also regulate the overhead. Experiments need to be done on other systems to observe the nature of synchronization overhead and the other issues related to the division of data among processors.

- *Implementation of Admission Controller*: Implementation of Admission Controller needs to be done using this system. We have not implemented an Admission Controller using the multi-processor based heuristics. The performance of an Admission Controller using the multi-processor base heuristic can be studied.

- *Average case analysis*: We have presented the worst case analysis of the algorithm. Average case analysis needs a thorough statistical analysis. We have provided a discussion on the average case scenario though it does not present any quantitative analysis. An equation for finding the optimal number of processors in the average case can be derived using the Equation 3.4.

- *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. But it will be slower as socket programming is a slower inter-process communication method.

Appendix

Program written using *C* programming language for the multi-processor based heuristics:

The header file (keys.h) used in the program is as follows:

```
// This header file contains constants values and reuired
// procedures to allocate semaphores and shared memory

#include </usr/include/sys/sem.h>
#include </usr/include/sys/shm.h>
#include </usr/include/sys/ipc.h>

#define C  5

// Constants used to lock or unlock semaphores
#define LOCK   -1
#define UNLOCK  1

// Constants required to allocate semaphores and shared memory
#define PERMS                0666
#define SEM_KEY     7890
#define S_KEY       7891
#define URES_KEY    7892
#define TSESSION_KEY 7893
#define TQOS_KEY    7894
#define MDELR_KEY   7895
#define MDELP_KEY   7896

#define F_UPGRADE 0
#define I_UPGRADE 1
#define DOWNGRADE 2

#if defined(_GNU_LIBRARY_) && !defined(_SEM_SEMUN_UNDEFINED)
/**********************/
#else

// Structures reuired to operate on semaphores
union semun{
  int val;
  struct semid_ds *buf;
  unsigned short int *array;
  struct seminfo *_buf;
};
#endif
```

```
// This function is used to lock or unlock a semaphore
void csemop(int semid,int semnum, int op)
{
 struct sembuf sb;

 sb.sem_num = semnum;
 sb.sem_op = op;
 sb.sem_flg = SEM_UNDO;

 semop(semid,&sb,1);
}

// This function waits until the value of the semaphore specified by
// semnum becomes zero
void waitzero(int semid,int semnum)
{
 struct sembuf sb;

 sb.sem_num = semnum;
 sb.sem_op = 0;
 sb.sem_flg = 0;

 semop(semid,&sb,1);
}

// This function sets the value of the semaphore specified by semnum
void csetval(int semid,int semnum,int val)
{
 union semun sunion;

 sunion.val = val;
 semctl(semid,semnum,SETVAL,sunion);
}

// This function gets the value of the semaphore specified by semnum
int cgetval(int semid,int semnum)
{
 union semun sunion;

 return semctl(semid,semnum,GETVAL,sunion);
}
```

The main file containing the main functions is as follows:

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

// The reasonm for this:
// Admission control of Multimedia session maps to the MMKP exactly


#include </usr/include/stdio.h>
#include </usr/include/math.h>
#include </usr/include/stdlib.h>
#include </usr/include/sys/time.h>
#include </usr/include/signal.h>
#include </usr/include/unistd.h>
#include </usr/include/sys/wait.h>
#include </usr/include/values.h>
#include "keys.h"

void datainit();
void revive_solution();
double netrev();
void sort_pile(int);
void infeasible_constraint();
int find_feasible();
int downgradepossible(int session_no,int qos_no);
int resource_conservation();
int presource_upgradation();
int presource_up_de_gradation();
int resource_constraint(int,int);
void save_solution();
int cost_improved(int,int,double);
int verify_solution();
void pdo_heu();
void do_calculation(int);
void get_shared_memory();


char ch;
int dsemid,dshmidts,dshmidtq,dshmidmdr,dshmidmdp,dshmidur,dshmids;
int semid,shmid;
int *ltsession,*ltqos;
```

```
double *lmdelr,*lmdelp;
double *used_resource;
int pl, no_of_process,cal_type,target,s_boundary,t_boundary;

const double TOL=1.0e-6;
int Rc=10,Pc=10;
int no_of_sessions,no_of_resources,no_of_qos;
double **cost,***resource,*total_constraint;
double *cost_per_unit;          //per unit cost of the resources
int *solution,*saved_solution,*no_of_qos_levels;
//solution[]: for holdin the current solution
//saved_solution[]: saving a solution
//no_of_qos_levels[]: Holding the number of items in each group


        void datainit(){

                int l1,l2,a;
                int rand=0;
                int rc=10;
                int pc=10;
                int total_no_of_resources;
                int j,i,k,seed;
                double temp;
                FILE *in;


                in=fopen("./input.txt","rt");
                fscanf(in,"%d                                %d                                %d
%d",&no_of_sessions,&no_of_qos,&no_of_resources,&seed);
                fclose(in);
                srandom(seed);

                cal_type=no_of_resources;
                target=no_of_resources+1;

                if(ch=='P'||ch=='p')
                {
                printf("How many process to use?");
                scanf("%d",&no_of_process);
                s_boundary=no_of_process*2;
        //second array boundary
                t_boundary=no_of_process*3;                                //third
array boundary
                pl=ceil((double)no_of_sessions/no_of_process);
                }
```

```
get_shared_memory();

cost=(double **) malloc(sizeof(double*)*no_of_sessions);
for(l1=0;l1<no_of_sessions;l1++)
        cost[l1]=(double *) malloc(sizeof(double)*no_of_qos);

resource=(double***) malloc(sizeof(double**)*no_of_resources);
for(l1=0;l1<no_of_resources;l1++)
        resource[l1]=(double**)malloc(sizeof(double*)*no_of_sessions);
for(l1=0;l1<no_of_resources;l1++)
        for(l2=0;l2<no_of_sessions;l2++)
                resource[l1][l2]=(double*)
malloc(sizeof(double)*no_of_qos);

total_constraint=(double*)malloc(sizeof(double)*no_of_resources);

cost_per_unit=(double*)malloc(sizeof(double)*no_of_resources);

no_of_qos_levels=(int*)malloc(sizeof(int)*no_of_sessions);

saved_solution=(int*)malloc(sizeof(int)*no_of_sessions);

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]=(int)random()%pc;


for (i=0;i<no_of_sessions;i++)
        for (k=0;k<total_no_of_resources;k++)
                resource[k][i][0]=0;


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=1;j<no_of_qos_levels[i];j++){
                for (k=0;k<total_no_of_resources;k++) {
                        resource[k][i][j]=(int)random()%rc;
                }
        }
}

if (rand==1){ // The value of item is not proportional to resource
consumption
```

```
                    for (i=0;i<no_of_sessions;i++)
                      cost[i][0]=0;
                    for (i=0;i<no_of_sessions;i++){
                          for (j=1;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]=(int)random()%(total_no_of_resources*(rc/2)*(pc/2));//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++)
                      cost[i][0]=0;
                    for (i=0;i<no_of_sessions;i++){
                          for (j=1;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]+=(int)random()%(total_no_of_resources*3*(rc/10)*(pc/10));//rand_var.
nextInt(total_no_of_resources*3*(rc/10)*(pc/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
              }

        }


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]];
        }
        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 do_calculation(int id)
{
 int sid = id+1,id1=no_of_process+id,id2=s_boundary+id;
 int starts = id*pl,ends;
 int shmsemnum=no_of_process+1;
//id1 used for second portion of shared arrays
//id2 used for third portion of shared arrays

        double delr,mdelr;
        double delp,mdelp,cmdelp;
        double delr1=0.0,delr2=0.0;

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


 if(sid == no_of_process)
   ends = no_of_sessions;
 else
```

```
  ends = sid*pl;
 printf("Child entered...id start end %d %d %d\n",sid,starts,ends);

 while(1)
  {
        csemop(semid,sid,LOCK);     //locking itself
   //calculations

switch((int)used_resource[cal_type])
{
  case F_UPGRADE:
        mdelr=0.0;
        mdelp=0.0;

        tsession=-1;
        tqos=-1;
        for(i=starts;i<ends;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
                          flag=0;
                          delr=0.0;
                          delr1=0.0;
                          delr2=0.0;
                          for(k=0;k<no_of_resources;k++){
                                if (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){
                                mdelr=delr;
                                flag=1;
                          }
                          if (mdelr<0){
                                // If the change of agrregate res is negative then
looking for
```

```
                                        // items with higest 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;
                                        }
                                }
                                else{
                                        if(flag){
                                                tsession=i;
                                                tqos=j;
                                        }
                                }
                        }
                }
        }

   // csemop(semid,shmsemnum,LOCK);
   ltsession[id]=tsession;
   ltqos[id]=tqos;
   lmdelr[id]=mdelr;
   lmdelp[id]=mdelp;
   //  csemop(semid,shmsemnum,UNLOCK);
   break;

  case I_UPGRADE:
        mdelp=0.0;
        tsession=-1;
        tqos=-1;
        //Finding an infeasible upgrade
        for(i=starts;i<ends;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          (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;
                    }
                }
            }
        // csemop(semid,shmsemnum,LOCK);
        ltsession[id]=tsession;
        ltqos[id]=tqos;
        lmdelp[id]=mdelp;
        // csemop(semid,shmsemnum,UNLOCK);
        break;

    case DOWNGRADE:
            tsession1=-1;
            tqos1=-1;
            tsession2=-1;
            tqos2=-1;
            mdelp=0.0;
            cmdelp=0.0;
            for(i=starts;i<ends;i++){
                for(j=0;j<solution[i];j++){
                    if      (cost_improved(i,j,used_resource[target])==1        &&
i!=ltsession[0]){
                        // The selection which downgrades less than target
                        delr=0;
                        for(k=0;k<no_of_resources;k++){
                            //Detaermining delr (dprinme)
                            if  (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]);
                        }
                        //Detaermining delr(dprinme)/delv
                        delp=((double)delr)/((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;
                                        }
                                }
                        }
                }
        }
   // csemop(semid,shmsemnum,LOCK);
   ltsession[id1]=tsession1;
   ltqos[id1]=tqos1;
   lmdelp[id1]=mdelp;
   ltsession[id2]=tsession2;
   ltqos[id2]=tqos2;
   lmdelp[id2]=cmdelp;
   //  csemop(semid,shmsemnum,UNLOCK);

   break;
}                                       //end switch
   csemop(semid,0,LOCK);      //Actully decrease the value of controller's sema by 1
                                        //and signals the controller to continue
 }                                      //end while
}

int presource_upgradation(){
        // Selecting new items by upgrading only
      int i,t,k,lmi;
        double mdelr,mdelp;

        /**no use**********************/
        double delr;////////////////
        double delp;////////////////
        double delr1=0.0,delr2=0.0;/////////////

        int tsession,tqos,j,flag;//////////////////
        /*****************************/

        used_resource[cal_type]=F_UPGRADE;
        csetval(semid,0,no_of_process);
        for(i=1;i<=no_of_process;i++)  //unlocking subprocesses
          csemop(semid,i,UNLOCK);
        waitzero(semid,0);                      //locking itself
//      printf("Controller is working...\n");
```

```
        lmi = -1;
        for(i=0;i<no_of_process;i++)
         if(ltsession[i]!= -1)
           {
                mdelr=lmdelr[i];
                mdelp=lmdelp[i];
                lmi=i;
                break;
           }
        if(lmi!=-1)
        {
        t=lmi;
        for(++i;i<no_of_process;i++)
         if(ltsession[i]!=-1 && lmdelr[i]>mdelr)
                {
                        lmi=i;
                        mdelr=lmdelr[i];
                }
        if(mdelr<0)
                {
                        lmi=t;
                        for(i=lmi+1;i<no_of_process;i++)
                                if(lmdelp[i]>mdelp)
                                        {
                                                lmi=i;
                                                mdelp=lmdelp[i];
                                        }
                }
                // An upgradeable item found
        for (k=0;k<no_of_resources;k++)
                used_resource[k]+=resource[k][ltsession[lmi]][ltqos[lmi]]-
resource[k][ltsession[lmi]][solution[ltsession[lmi]]];
        solution[ltsession[lmi]]=ltqos[lmi];
        return 1;
         }
        else
        return 0;
// No item is found to upgrade the solution
}


int presource_up_de_gradation(){
        // Selecting new items by upgrading only
   int i,t,k,lmi,tsession,tqos,tsession2,tqos2,tsession1,tqos1;
        double mdelp,cmdelp;
```

```
/**********no use******/
        double delr;
        double targett;
        double delp;
/**************/

        mdelp=0.0;

        //setting calculation type
        used_resource[cal_type]=I_UPGRADE;
        csetval(semid,0,no_of_process);
        for(i=1;i<=no_of_process;i++)  //unlocking subprocesses
          csemop(semid,i,UNLOCK);
        waitzero(semid,0);                              //locking itself
//      printf("Controller is working...\n");

        lmi = -1;
        mdelp=-MAXINT;
        for(i=0;i<no_of_process;i++)
          if(ltsession[i]!=-1 && lmdelp[i]>mdelp)
          {
                mdelp=lmdelp[i];
                lmi=i;
          }
        if(lmi==-1) return 0;

        tsession=ltsession[lmi];
        ltsession[0]=tsession;
        tqos=ltqos[lmi];
        //determining target
        used_resource[target]=-cost[tsession][solution[tsession]]+cost[tsession][tqos];
        for(k=0;k<no_of_resources;k++)
          used_resource[k]+=resource[k][tsession][tqos]-
resource[k][tsession][solution[tsession]];
        solution[tsession]=tqos;

        used_resource[cal_type]=DOWNGRADE;
        do{
          csetval(semid,0,no_of_process);
          for(i=1;i<=no_of_process;i++)  //unlocking subprocesses
            csemop(semid,i,UNLOCK);
          waitzero(semid,0);                            //locking itself
//        printf("Controller is working...\n");
          lmi=-1;
          cmdelp=-MAXINT;
          for(i=s_boundary;i<t_boundary;i++)
```

```
                if(ltsession[i]!=-1 && lmdelp[i]>cmdelp)
                 {
                        cmdelp=lmdelp[i];
                        lmi=i;
                 }
          if(lmi!=-1)
          {
                tsession2=ltsession[lmi];
                tqos2=ltqos[lmi];
                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
          {
                lmi=-1;
                mdelp=-MAXINT;
                for(i=no_of_process;i<s_boundary;i++)
                 if(ltsession[i]!=-1 && lmdelp[i]>mdelp)
                        {
                                mdelp=lmdelp[i];
                                lmi=i;
                        }
                if(lmi!=-1)
                 {
                        tsession1=ltsession[lmi];
                        tqos1=ltqos[lmi];
                        for(k=0;k<no_of_resources;k++)
                                used_resource[k]+=resource[k][tsession1][tqos1]-
resource[k][tsession1][solution[tsession1]];

        used_resource[target]=used_resource[target]+cost[tsession1][tqos1]-
cost[tsession1][solution[tsession1]];
                        solution[tsession1]=tqos1;
                 }
                else
                 return 0;
          }
        }while(tsession!=-1);

        return 1;
}

void pdo_heu(){
```

```
        // Main function to fetermine the heuristic of the MMKP
        int i,k,j;


        do{     // Step 2: Upgrading only
                i=presource_upgradation();
        }while(i==1);

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

                        }while(j==1);
                }else revive_solution(); // Step 3 failed, so solution revived
        }while (i==1);

        printf("Revenue earned by HEU %f\n",netrev());
        if (verify_solution()==0) {
        printf("Error in HEU\n");
        exit(0);
        }
}


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]) {
                        printf("Solution invalid\n");
                        return 0;
                }
        }
        return 1;
}


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]];
        }
}



void get_shared_memory()
{

   shmid = shmget(S_KEY,sizeof(int)*no_of_sessions,IPC_CREAT|PERMS);
   if(shmid <0 )
    {
     printf("Error getting memory..1\n");
     exit(1);
    }
   dshmids = shmid;
   solution = (int *) shmat(shmid,(char *)0,0);
```

```
shmid=shmget(URES_KEY,sizeof(double)*(no_of_resources+2),IPC_CREAT|PERMS);
   if(shmid <0 )
    {
     printf("Error getting memory..1\n");
     exit(1);
    }
   dshmidur = shmid;
   used_resource = (double *) shmat(shmid,(char *)0,0);

if(ch=='P'||ch=='p')
{
   semid = semget(SEM_KEY,no_of_process+1,PERMS|IPC_CREAT);
   if(semid <0 )
   {
    printf("Error getting semaphore...\n");
    exit(1);
   }
   dsemid = semid;
   csetval(semid,no_of_process+1,1);


   shmid = hmget(TSESSION_KEY,sizeof(int)*no_of_process*3,IPC_CREAT|PERMS);
   if(shmid <0 )
    {
     printf("Error getting memory..1\n");
     exit(1);
    }
   dshmidts = shmid;
   ltsession = (int *) shmat(shmid,(char *)0,0);

   shmid = shmget(TQOS_KEY,sizeof(int)*no_of_process*3,IPC_CREAT|PERMS);

   if(shmid <0 )
    {
     printf("Error getting memory..2\n");
     exit(1);
    }
   dshmidtq = shmid;
   ltqos = (int *) shmat(shmid,(char *)0,0);

   shmid = shmget(MDELR_KEY,sizeof(double)*no_of_process,IPC_CREAT|PERMS);
   if(shmid <0 )
    {
     printf("Error getting memory..3\n");
     exit(1);
```

```
    }
    dshmidmdr = shmid;
    lmdelr = (double *) shmat(shmid,(char *)0,0);

    shmid
shmget(MDELP_KEY,sizeof(double)*no_of_process*3,IPC_CREAT|PERMS);
    if(shmid <0 )
     {
      printf("Error getting memory..4\n");
      exit(1);
     }
    dshmidmdp = shmid;
    lmdelp = (double *) shmat(shmid,(char *)0,0);
}
}

void delsem(void)
 {
  if(dshmids>=0)
   {
    printf("Deleting shm.\n");
    shmctl(dshmids,IPC_RMID,NULL);
   }

  if(dshmidur>=0)
   {
    printf("Deleting shm.\n");
    shmctl(dshmidur,IPC_RMID,NULL);
   }

if(ch=='P'||ch=='p')
{
  if(dsemid >=0)
   {
    printf("Deleting sema.\n");
    semctl(dsemid,0,IPC_RMID,0);
   }
  if(dshmidts >=0)
   {
    printf("Deleting shm.\n");
    shmctl(dshmidts,IPC_RMID,NULL);
   }
  if(dshmidtq >=0)
   {
    printf("Deleting shm.\n");
    shmctl(dshmidtq,IPC_RMID,NULL);
```

```
  }
 if(dshmidmdr >=0)
  {
   printf("Deleting shm.\n");
   shmctl(dshmidmdr,IPC_RMID,NULL);
  }
 if(dshmidmdp >=0)
  {
   printf("Deleting shm.\n");
   shmctl(dshmidmdp,IPC_RMID,NULL);
  }
}
 }

void sigdelete(int signum)
 {
 exit(0);
 }



int main ()
{
  struct timeval start,end;
  struct timezone tz;
  int r,i,k;
  pid_t cpid[10];
  long usec;
  FILE *out;


  datainit();
// use fork here
   for(i=0;i<no_of_process;i++)
   {
    csetval(semid,i+1,0);
    cpid[i] = fork();
    if(cpid[i]==0)
         {
         do_calculation(i);
         exit(0);
         }
   }
   atexit(&delsem);
   signal(SIGINT,&sigdelete);
```

```
        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]];
        }

   gettimeofday(&start,&tz);
   pdo_heu();
   gettimeofday(&end,&tz);
   for(i=0;i<no_of_process;i++)
        kill(cpid[i],SIGKILL);
}

        out=fopen("./pout.txt","a+t");
        usec=(end.tv_sec-start.tv_sec)*1000000+((end.tv_usec-start.tv_usec));
        fprintf(out,"%d\t%d\t%d\t%d\t%lf\n",no_of_process,no_of_sessions,no_of_qos,n
o_of_resources,(double)usec/1000);
        fclose(out);
        return 0;

}
```

References

[1] M. M. Akbar. *The Distributed Utility Model Applied to Optimal Admission Control & QoS Adaptation in Multimedia Systems & Enterprise Networks.* PhD Dissertation, Department of Computer Science, University of Victoria, 2002.

[2] M. M. Akbar, E. G. Manning, G. C. Shoja. Admission Control and QoS adaptation in Distributed Multimedia Server System. *ITCom 2001.* pp 246-257, August 2001, Denver, USA.

[3] M. M. Akbar, E. G. Manning, G. C. Shoja, S. Khan, Heuristic Solutions for the Multidimensional Multiple-Choice Knapsack Problem, *International Conference on Computational Science.* pp 659-668, Part II, May 2001, San Francisco, U.S.A.,

[4] V. N. Alexandrov, G. M. Megson, *Parallel Algorithms for Knapsack type problems*, World Scientific Publishing Co. Pte. Ltd., 1999.

[5] Ellis Horwitz and Sartaj Shahni. *Fundamentals of Computer Algorithms.* Galgotia Publications, 1995.

[6] Joseph JáJá. *An Introduction to Parallel Algorithms.* Addison-Wesley Publishing Company, 1992.

[7] S. Khan and K. F. Li and E. G. Manning. The Utility Model for Adaptive Multimedia Systems. *International Conference on Multimedia Modelling*, Singapore, pp 111-126, Nov 17-20, 1997.

[8] S. Khan, K. F. Li, E. G. Manning, M. M. Akbar. Solving the Knapsack Problem for Adaptive Multimedia System. *Studio Informica.* pp 161-182, 2002.

[9] S. Martello and P. Toth. Algorithms for Knapsack Problems. *Annals of Discrete Mathematics*, pp 70-79, Volume 31, 1987.

[10] R. Nauss. The 0-1 Knapsack Problem with Multiple Choice Constraints. *European Journal of Operation Research*, pp 125-131, Volume 2, 1978.

[11] M. A. H. Newton, M. W. H. Sadid, M. M. Akbar. A Parallel Heuristic Algorithm for Multiple-Choice Multidimensional Knapsack Problem. *International Conference on Computer and Information Technology*, pp 181-184, Part I, Dec 19-21, 2003, Dhaka, Bangladesh.

[12] W. Shih. A branch and Bound Method for Multiconstraint Knapsack Problem. *Journal of the Operational Research Society*, pp 369-378, Volume 30, 1979.

[13] W. R. Stevens. *Unix Network Programming*. PHI, 1999.

[14] Y. Toyoda. A Simplified Algorithm for Obtaining Approximate Solution to Zero-one Programming Problems. *Management Science*, pp 1417-1427, Volume 21, 1975.