A HYPERCUBE ALGORITHM FOR THE 0/1 KNAPSACK PROBLEM *

Jong Lee+, Eugene Shragowitz, Sartaj Sahni
Computer Science Department
University of Minnesota
Minneapolis, MN55455

Abstract

Many combinatorial optimization problems are known to be NP-complete. A common point of view is that finding fast algorithms for such problems using polynomial number of processors is unlikely. However facts of this kind usually are established for "worst" case situations and in practice many instances of NP-complete problems are successfully solved in polynomial time by such traditional combinatorial optimization techniques as dynamic programming, branch-and-bound, etc. New opportunities for effective solution of combinatorial problems emerged with the advent of parallel machines. In this paper we describe an algorithm which generates an optimal solution for the 0/1 integer Knapsack problem on the NCUBE hypercube computer. It is also demonstrated that the same algorithm can be applied for the two-dimensional 0/1 Knapsack problem. Experimental data which supports the theoretical claims are provided for large instances of the one- and two-dimensional Knapsack problems.

1. Introduction

The fact that many instances of problems, which belong to the class of NP-complete problems, were successfully solved in practice in polynomial time demonstrates some limitations of this classification, which describes the "worst" case scenario.

The 0/1 Knapsack problem is proven to be NP-complete. It is traditionally solved by the dynamic programming algorithm, which is accepted as the most practical way to solve this problem. With the advent of parallel processors, many researchers concentrated their efforts on development of approximation algorithms for NP-complete problems based on the application of parallel processors. For the 0/1 Knapsack problem, such works were reported by Peters and Rudolf [PETE84], and Gopalakrishnan et al. [GOPA86]. Another relevant branch of research was related to design of systolic arrays for dynamic programming problems. This approach was considered in works of Li et al. [LI85], Lipton et al. [LIPT86] and others. A different model for the parallel computation of the Knapsack problem with weights given by real numbers was considered by A. Yao [YAO81].

---

* This research was supported in part by the National Science Foundation under grants MCS83-05567 and DCR84-20935. + Dr Jong Lee is presently with AT&T Bell Laboratories, Allentown, PA.

Our work goes in another direction. The proposed algorithm for the optimal solution of the 0/1 integer Knapsack problem is targeted to the architecture of an existing hypercube computer and exploits parallelism discovered in the dynamic programming algorithm for this problem.

This paper is organized the following way. The second section describes the dynamic programming algorithm for the 0/1 Knapsack problem for the one-processor machine. The third section reveals parallelism existing in the problem and provides a decomposition algorithm for partitioning the original problem between processors of the hypercube. It also describes the combining operation which assembles the optimal solution from the solutions of the subproblems. In the fourth section, we describe implementation details of our algorithm on the NCUBE hypercube. The fifth section describes such aspects of the algorithm as amount of required data transmissions, storage and timing characteristics and computational complexity. In section six, we state experimental results, compare experimental time and space characteristics with theoretical estimations and provide appropriate comments.

2. Dynamic Programming Method for the Knapsack Problem on a Single-processor Machine

One-dimensional 0/1 Knapsack problem $KNAP(G,c)$ can be defined as follows: We are given a set G of m different objects and a knapsack. Object i has a weight w_i and a profit p_i, $1 \le i \le m$ and the knapsack has a capacity c. w_i, p_i and c are positive integers. We may also assume that $w_i \le c$, $1 \le i \le m$. The problem is to find a combination of objects to include in the knapsack such that

$$\sum_{i=1}^{m} p_i z_i \quad \textit{is maximized}$$

$$\textit{subject to} \quad \sum_{i=1}^{m} w_i z_i \le c$$

$$\textit{where} \quad z_i = \begin{cases} 1 & \text{if } \textit{object } i \textit{ is included} \\ 0 & \textit{otherwise.} \end{cases}$$

The vector $\mathbf{z} = (z_1, z_2, \cdots, z_m)$ is the solution vector.

Let $f_m(c)$ be the value of the optimal solution for the above problem. The principle of optimality holds for the knapsack problem, i.e.

$$f_m(c) = \max\{f_{m-1}(c),\ f_{m-1}(c - w_m) + p_m\}$$

This equation can be generalized as follows:

$$\begin{cases} f_0(x) = \begin{cases} 0 & x \geq 0 \\ -\infty & x < 0 \end{cases} \\ f_i(x) = \max\{f_{i-1}(x),\ f_{i-1}(x-w_i)+p_i\}, \ \text{for } all\ x, \\ \qquad where \quad i = 1,2,...,m \end{cases}$$

Dynamic programming solves this problem by generating $f_1, f_2, \ldots, f_m$ successively.

$f_i(x)$ is a monotonic nondecreasing integer step function. $f_i(x)$ can be represented as a list S_i of tuples from the coordinates of the step points of $f_i(x)$. The size of the list S_i, i.e. $|S_i|$, is not greater than $c+1$ and tuples are listed in increasing order of x and $f_i(x)$. The sequence of lists, $S_0, S_1, ..., S_m$, is a history of the dynamic programming and it should be traced back by Algorithm 2 to find a solution vector **z**. Algorithm 1 generates lists $S_1, S_2, \ldots, S_m$ as follows.

Algorithm 1 [forward part of dynamic programming]

$S_0 \leftarrow \{(0, 0)\}$

for $i \leftarrow 1$ **to** m **do**

begin

$S_i' \leftarrow \{(P+p_i, W+w_i) \mid (P, W) \in S_{i-1}, W+w_i \leq c\}$

$S_i \leftarrow merge\,(S_{i-1}, S_i')$

end

Here P is the value of the profit function $f_{i-1}(x)$ when the weight $x = W$. The *merge* procedure merges two lists S_{i-1} and S_i' to create list S_i. During the merging, if $S_i' \cup S_{i-1}$ contains two tuples (P_j, W_j) and (P_k, W_k) such that $P_j \leq P_k$ and $W_j \geq W_k$, i.e. (P_k, W_k) dominates (P_j, W_j), then (P_j, W_j) is discarded.

If (P, W) is the last tuple in S_m, then P is the value of the optimal solution and the solution vector **z** such that $\sum_{i=1}^{m} p_i z_i = P$ and $\sum_{i=1}^{m} w_i z_i = W$ is determined by searching through the sets $S_m, S_{m-1}, \ldots, S_0$. This task is performed by Algorithm 2.

Algorithm 2 [backtracking part of dynamic programming]

$(P, W) \leftarrow$ last tuple in S_m

for $i \leftarrow m$ **downto** 1 **do**

begin

if $(P-p_i, W-w_i) \in S_{i-1}$ **then**

$z_i \leftarrow 1; \quad P \leftarrow P-p_i; \quad W \leftarrow W-w_i$

else

$z_i \leftarrow 0$

endif

end

From Algorithm 1, it follows that the time and space required are proportional to $\sum_{i=0}^{m} |S_i|$. Since $|S_{i+1}| \le 2|S_i|$ and $|S_i| \le c+1$ for all i, in the worst case $|S_i|$ is increasing exponentially for $i \le \log_2 c$ and it remains level at $c+1$ for $\log_2 c < i \le m$ (Figure 1 (a)). The total time and space requirements can be described by the area under the function $|S_i|$.
The time in the worst case can be estimated by the formula:

$$T_{DP}(1,m,c) = O(\min\{2^m, mc\})$$

3. Multiprocessor Divide-and-Conquer Algorithm

3.1. Strategy

A natural approach to the solution of a large problem in the multiprocessor environment is to decompose the original problem into several subproblems of smaller size, to solve all subproblems in parallel and finally combine all partial solutions into a solution for the original problem. Very often, it is difficult to decompose the problem into small subproblems which can be solved in parallel. Even though some problems can be decomposed into subproblems, the gain achieved by such decomposition can easily be lost during the combining operation. We propose an algorithm for the Knapsack problem, which overcomes these difficulties. This algorithm provides substantial gain in performance as a result of implementation of the parallel processing scheme.

3.2. Decomposition of Problem

Let us consider p processors, $PR_0, PR_1, \ldots, PR_{p-1}$, where $p=2^n$ for some integer $n \ge 0$. The original Knapsack problem $KNAP(G, c)$ is partitioned into p subproblems $KNAP(G_i, c)$ for $i=0,1,\ldots,p-1$, where $G=\bigcup_{i=0}^{p-1} G_i$, $G_i \cap G_j = \phi$ if $i \ne j$ and $|G_i| = |G|/p = \sigma$ for all i. Each subproblem $KNAP(G_i, c)$ is assigned to the processor PR_i and solved independently by PR_i applying the forward part of the dynamic programming procedure described in Algorithm 1. Because Algorithm 1 computes values of the optimal solutions of $KNAP(G_i, x)$ for $0 \le x \le c$, every PR_i generates a vector $\mathbf{a}^i = (a_0^i, a_1^i, \ldots, a_c^i) = (f_\sigma^i(0), f_\sigma^i(1), \ldots, f_\sigma^i(c))$. We call the vector $\mathbf{a}^i$ an optimal profit vector for the $KNAP(G_i, c)$ and a_x^i is a value of the optimal solution of $KNAP(G_i, x)$. For simplicity, we assume that the final tuple list computed in the dynamic programming step in each processor contains close to the maximum of $c+1$ tuples. So, it is efficient to use a one dimensional array for

the profit vector during the combining procedure.

The algorithm for solving the Knapsack problem on a parallel machine may be formulated as Algorithm 3.

Algorithm 3 [parallel algorithm]

(1) [partition]: Partition $KNAP(G,c)$ into p subproblems $KNAP(G_i,c)$, $i=0,1,...,p-1$ such that $G=\bigcup_{i=0}^{p-1} G_i$, $G_i \cap G_j = \phi$ if $i \neq j$ and $|G_i| = |G|/p$ for all i.
Assign $KNAP(G_i,c)$ to PR_i, for $i=0,1,...p-1$.

(2) [forward part of dynamic programming]: Each processor solves $KNAP(G_i,c)$ applying Algorithm 1 and gets a profit vector $\mathbf{a}^i$.

(3) [forward part of the combining procedure]: the p processors combine their profit vectors $\mathbf{a}^i$, for $i=0,1,...,p-1$, to get the resulting profit vector $\mathbf{r}=(r_0,r_1,...,r_c)$ for $KNAP(G,c)$.
/* see Algorithm 4 */

(4) [backtracking part of the combining procedure]: the p processors trace back the combining history to get x_i, for $i=0,1,...,p-1$ such that

$$\sum_{i=0}^{p-1} x_i = c, \quad \sum_{i=0}^{p-1} a^i_{x_i} = r_c$$

/* see Algorithm 5 */

(5) [backtracking part of dynamic programming]: Each processor traces back its dynamic programming history applying Algorithm 2 with $(P,W)=(a^i_{x_i}, x_i)$ to get an optimal solution vector.

3.3. Combining Operation

The problem is to combine the profit vectors for the p subproblems into the resulting profit vector for $KNAP(G,c)$. Let $\mathbf{b}$ and $\mathbf{d}$ be two profit vectors for $KNAP(B,c)$ and $KNAP(D,c)$ such that $B \cap D = \phi$. We will define two operators *combine* and *history* as follows:

$\mathbf{e}=combine(\mathbf{b},\mathbf{d})$,

where $e_i = \max_{0 \leq j \leq i} \{b_j + d_{i-j}\}$, for $i=0,1,...,c$.

$\mathbf{h}=history\,(\mathbf{b},\mathbf{d})$,

where $h_i=j_0$ such that

$$b_{j_0}+d_{i-j_0}=\max_{0\le j\le i}\{b_j+d_{i-j}\},\ \text{for } i=0,1,...,c$$

For example, $e_0=b_0+d_0$, $e_1=max\{b_0+d_1,\ b_1+d_0\}$, $e_2=max\{b_0+d_2,\ b_1+d_1,\ b_2+d_0\}$, etc. The *history* operation provides information for tracing back the combining procedure later. It is easy to see that one combining operation requires c^2 basic operations. The *combine* operation is commutative and associative.

Lemma 1

Let **b** and **d** be two optimal profit vectors for $KNAP\,(B,c)$ and $KNAP\,(D,c)$ such that $B\cap D=\phi$, then $\mathbf{e}=combine\,(\mathbf{b},\mathbf{d})$ is the optimal profit vector for $KNAP\,(B\cup D,c)$.

Proof:

Let **z** be an optimal solution vector for $KNAP\,(B\cup D,x)$ for some x, where $0\le x\le c$. Let α be a value of the optimal solution and β be the sum of weights of all selected objects. Formally, this can be stated as follows:

$$\sum_{i\in B\cup D} p_i z_i=\alpha,\quad \sum_{i\in B\cup D} w_i z_i=\beta\le x$$

Each of these expressions can be broken into two expressions as follows:

$$\sum_{i\in B} p_i z_i=\alpha_B,\quad \sum_{i\in B} w_i z_i=\beta_B,$$

and

$$\sum_{i\in D} p_i z_i=\alpha_D,\quad \sum_{i\in D} w_i z_i=\beta_D.$$

Then

$$\alpha_B+\alpha_D=\alpha,\quad \beta_B+\beta_D=\beta.$$

Since b_{β_B} is a value of the optimal solution for $KNAP\,(B,\beta_B)$, then $\alpha_B\le b_{\beta_B}$. The same is true for $KNAP\,(D,\beta_D)$, thus $\alpha_D\le b_{\beta_D}$.

Thus,

$$e_x \geq e_\beta = \max_{0 \leq j \leq \beta}(b_j + d_{\beta - j}) \geq b_{\beta_B} + d_{\beta - \beta_B}$$

$$= b_{\beta_B} + d_{\beta_D} \geq \alpha_B + \alpha_D = \alpha$$

But e_x is a value of the feasible solution for $KNAP(B \cup D, x)$, i.e. $\alpha \geq e_x$, thus from $e_x \geq \alpha$, it follows that $e_x = \alpha$. So e_x is a value of the optimal solution for $KNAP(B \cup D, x)$, where $0 \leq x \leq c$, and **e** is an optimal profit vector for $KNAP(B \cup D, c)$ □

From the fact that the *combine* operation is associative and commutative, it follows that the final profit vector for $KNAP(G, c)$ can be obtained from the set $\{\mathbf{a}^0, \mathbf{a}^1, \ldots, \mathbf{a}^{p-1}\}$ of profit vectors generated by the p processors by combining them in any order.

3.4. Combining Algorithm

The combining procedure is represented as a tree in Figure 2. On level 1 of *combine* procedure, there are p profit vectors. The set of p processors is partitioned into $p/2$ groups of size 2 and each group combines two profit vectors into a new profit vector for the next level. On level 2, there are $p/2$ profit vectors from the previous level. The set of processors on level 2 is partitioned into $p/4$ groups of size 4 and each group combines two profit vectors. This procedure is repeated until the final profit vector is constructed. In our implementation, we continue the processor partitioning only until each processor group has no more than $c/2$ processors. Allowing processor groups larger than this will require the use of a different strategy to distribute the computational load than the one described later in this paper. During the combining procedure, the history of combining operations is saved.

Let us introduce some notation for the following algorithms. Let k be the current level in the combining tree and l be a level, from which the group size is not increasing. Let $\mathbf{a}^{(i,k)}$ and $\mathbf{h}^{(i,k)}$ be the profit and history vectors, respectively, generated by a group i on the level k. In the following algorithm, the group i on the level k combines two profit vectors generated by groups $2i$ and $2i+1$, respectively, on level $k-1$.

Algorithm 4 [forward part of the combining procedure]

$l \leftarrow \log_2 c - 1$

$\mathbf{a}^{(i,0)} \leftarrow \mathbf{a}^i$, for $i = 0, 1, \ldots p-1$

for $k \leftarrow 1$ **to** n **do**

begin

$g \leftarrow min(k,l)$; $r \leftarrow \frac{p}{2^g}$;

Partition the set of p processors into r groups of size 2^g ;

for each group i $(0 \le i \le r-1)$ **in parallel do**

begin

All processors in the group i compute

$\mathbf{a}^{(i,k)} \leftarrow combine(\mathbf{a}^{(2i,k-1)}, \mathbf{a}^{(2i+1,k-1)})$

$\mathbf{h}^{(i,k)} \leftarrow history(\mathbf{a}^{(2i,k-1)}, \mathbf{a}^{(2i+1,k-1)})$

end

end

After combining p profit vectors, it is necessary to trace back the combining history to get a set of components $x_0, x_1, \ldots, x_{p-1}$ of a constraint c assigned to each processor such that

$$\sum_{i=0}^{p-1} x_i = c, \quad \sum_{i=0}^{p-1} a^i_{x_i} = a^{(0,n)}_c$$

$a^{(0,n)}_c$ is a value of the optimal solution for the original problem. The trace back procedure is described as Algorithm 5.

Algorithm 5 [backtracking part of the combining procedure]

$x_0 \leftarrow h^{(0,n)}_c$, $x_1 \leftarrow c - x_0$

for $k \leftarrow n-1$ **downto** 1 **do**

begin

$g \leftarrow min(k,l)$; $r \leftarrow \frac{p}{2^g}$;

for each $\mathbf{h}^{(i,k)}$ $(0 \le i \le r-1)$ **in parallel do**

begin

$t \leftarrow x_i$ /* t is a dummy variable */

$x_{2i} \leftarrow h^{(i,k)}_t$

$x_{2i+1} \leftarrow t - h^{(i,k)}_t$

end

end

In addition, each PR_i traces back the history of the dynamic programming procedure using

Algorithm 2 with $(P,W)=(a^i_{x_i},x_i)$ to find its part of the optimal solution vector.

3.5. Distribution of Computational Load

During the combining operation all processors in each group have the same pair of input profit vectors for the *combine* operation. From the definition of the *combine* operation, it follows that each element of the resulting profit vector can be computed independently from other elements. The computation of the j'th element, $j=0,1,2,...,c$, of the resulting profit vector requires $O(j)$ operations. To equalize the computational load between all processors from the same group, the computation of the elements of the resulting profit vector is distributed between them. The resulting profit vector is partitioned into twice as many subvectors as the number of processors. These parts are assembled from one subvector with a small index and another subvector with a high index. If the constraint c is divisible by the doubled number of processors in the group, then the computational load can be divided equally among all processors. If c is not divisible, then the load is approximately equal (Figure 3). The number of computations executed by each processor is proportional to $c^2/2^g$, where g is the level of the combining operation and 2^g is the number of processors in the group as defined by Algorithm 4.

4. Implementation of the Dynamic Programming Algorithm on the NCUBE Computer

4.1. Description of the NCUBE Computer

The NCUBE computer is a hypercube of dimension n, $0 \le n \le 10$ [HAYE86]. It consists of 2^n identical nodes so that every node is a general-purpose processor with its own local memory. The word "node" is used interchangably with the word "processor" here. Every node is numbered from 0 to 2^n-1 by an n-bit binary number $(q_n q_{n-1} \cdots q_1)$. If two nodes have node numbers only different in q_j, then one node is called an opposite node of another node in the j'th direction. Every node has a direct link to the opposite node. So every node has n neighbor nodes with direct links. A set of nodes where every node has the same number except k bit positions composes hypercube of dimension k. The (Hamming) distance between two nodes is the number of different bits between two binary node numbers. It can be measured by the number of links on the shortest path between two nodes. The length of the shortest path between any two nodes in a 2^n node hypercube is at most n. We shall refer to a 2^n node hypercube as an n-cube.

4.2. Description of Combining Algorithm for a Hypercube Computer

Algorithm 6 is a version of Algorithm 4 for an n-cube. Because the computational part of

Algorithm 6 is the same as for Algorithm 4, we are going to focus on the communication part. Before level k of the combining operation, the n-cube consists of 2^{n-k+1} $(k-1)$-cubes so that every $(k-1)$-cube consists of nodes with bits $(q_n q_{n-1} \cdots q_k)$ being the same in the node numbers. All nodes in a $(k-1)$-cube have the same profit vector.

On the first step of Algorithm 6, the set of p nodes is partitioned into 2^{n-g} groups of size 2^g, where $g = min(k,l)$. Every node in the group has the same bits $(q_n q_{n-1} \cdots q_{g+1})$ in the node number, i.e. the group is a g-cube. If $k \le l$, then every group consists of two $(k-1)$-cubes so that all nodes in each $(k-1)$-cube have the same profit vector. If $k > l$, then all nodes in the group have the same profit vector.

On the second step, each node exchanges its profit vector with its opposite node in the kth direction. If $k \le l$, two $(k-1)$-cubes comprising the group exchange profit vectors. If $k > l$, group $(q_n ... q_k 0 q_{k-2} \cdots q_{l+1})$ and group $(q_n ... q_k 1 q_{k-2} \cdots q_{l+1})$ exchange profit vectors. At this moment, every node in the group has the same pair of profit vectors to be combined.

On the third step, every node in the group computes its part of the new profit vector as described in the previous section.

On the fourth step, every node broadcasts elements of the new profit vector to every other node in the group. After finishing broadcasting, every node in the group has the same whole new profit vector.

The broadcasting is performed inside the group (Figure 3). Every node exchanges its own part of the profit vector with the opposite node. The received elements are merged with the own part to become bigger elements.

Algorithm 6

$l \leftarrow \log_2 c - 1$;

/* n is a dimension of the hypercube */

for $k \leftarrow 1$ **to** n **do**

begin

1.[Group]

$g \leftarrow min(k,l)$;

$GR_i \leftarrow \{NODE_q \mid (q_n q_{n-1} \cdots q_{g+1})_2 = i\}$,

where $i = 0, 1, ..., 2^{n-g} - 1$;

2.[Exchange]

Each node exchanges its profit vector with the opposite node in the k'th direction.

3.[Compute]

Every node in the group computes its part of the resulting profit vector and saves the combining history.

4.[Gather]

Every node broadcasts its elements of the combined profit vector to every other node in the group, so that every node in the group has the same new profit vector.

end

---

5. Analysis of Time and Space for Parallel Algorithm

5.1. Analysis of Computation

In this section, we assume that the time required by each node is directly proportional to the number of basic operations executed by the algorithm. The following lemma can be stated for the dynamic programming part of Algorithm 3.

Lemma 2

$$T_{DP}(1,\frac{m}{p},c) \leq \frac{T_{DP}(1,m,c)}{p} \quad \text{for } m \geq p,$$

where T_{DP} is the worst case processing time for the dynamic programming algorithm as a function of the number of nodes, problem size and constraint size.

Proof:

It is known that for the knapsack problem

$T_{DP}(1,m,c) = \min\{mc, 2^m\}$. Hence

$$T_{DP}(1,\frac{m}{p},c) = \min\{\frac{m}{p}c, 2^{\frac{m}{p}}\} \leq \frac{1}{p}\min\{mc, 2^m\}$$

$$= \frac{T_{DP}(1,m,c)}{p} \; \square$$

Our algorithm, by construction, has the following property

$$T_{DP}(p,m,c) = T_{DP}(1,\frac{m}{p},c) \qquad (*)$$

From Lemma 2 and (*), it follows that

$$T_{DP}(p,m,c) \le \frac{T_{DP}(1,m,c)}{p}$$

Because the time for the dynamic programming procedure is directly related to $|S_i|$ for $i=1,2,...m$, the worst case space $M_{DP}(p,m,c)$ required for dynamic programming by one node is as follows:

$$M_{DP}(p,m,c) \le \frac{M_{DP}(1,m,c)}{p}.$$

The processing time T_{C1} for Algorithm 4 can be described as the sum of two components. The first component is the processing time for the combining operations with the increasing size of the group. Another component includes the processing time for the steps of the combining procedure with the size of the group equal to $\frac{c}{2}$ (upper limit for the group size).

If $n>l$, then

$$T_{C1}(p,m,c) = O\left(\sum_{k=1}^{l} \frac{c^2}{2^k} + 2c(n-l) \right)$$

$$= O\left((1-2^{-l})c^2 + 2c(n-l) \right)$$

$$= O\left(c^2 + nc \right)$$

If $n \le l$, then

$$T_{C1}(p,m,c) = O\left(\sum_{k=1}^{n} \frac{c^2}{2^k} \right)$$

$$= O\left((1-2^{-n})c^2 \right) = O\left(c^2 \right)$$

(Here as before $p=2^n$, $l=\log_2 c-1$ and n is the number of levels)

The space required by each node for Algorithm 4 is space for history vectors and profit

vectors. Because profit vectors are not needed during the backtracking procedure, each node needs to reserve space of size $2c$ only for two input profit vectors during the procedure. The history vector is partitioned between all nodes in the group. Thus, the space M_{C1} required by Algorithm 4 in each node, in case $n>l$, is as follows:

$$M_{C1}(p,m,c)=O(\,(1-2^{-l})c+2(n-l)+2c\,)$$

$$=O(\,c+n\,)$$

If $n \le l$, then

$$M_{C1}(p,m,c)=O(\,(1-2^{-n})c+2c\,)=O(\,c\,)$$

The backtracking part of the combining procedure, i.e. Algorithm 5, performs a search through the combining tree and the procedure normally requires

$$T_{C2}(p,m,c) = O(n) = O(\log_2 p)$$

$$M_{C2}(p,m,c) = O(\,1\,)$$

Thus, the total worst case processing time T_{CB} and space M_{CB} for the combining procedure are as follows:

$$T_{CB}(p,m,c)=T_{C1}(p,m,c)+T_{C2}(p,m,c)$$

$$=O(\,c^2+nc\,)$$

$$M_{CB}(p,m,c)=M_{C1}(p,m,c)+M_{C2}(p,m,c)$$

$$=O(\,c+n\,)$$

5.2. Analysis of Data Communication

The data transmission time for Algorithm 6 is composed of two parts: communication setup time β and data tranfer time (unit data transfer time is γ). The data transmission time for the [Exchange] step is $T_{D1}(k,c)=2(\beta+c\gamma)$, because each node exchanges a profit vector with its opposite node. There are g data transmissions for the [Gather] step. The size of the initial part of the profit vector is $\frac{c}{2^g}$ and it is doubled after each transmission. So total time for the data transmission for the [Gather] step on level k is

$$T_{D2}(k,c) = 2g\beta+2\frac{c}{2^g}\sum_{j=1}^{g}2^{j-1}\gamma$$

$$= 2(g\beta + c(1-2^{-g})\gamma).$$

Thus, the total data transmission time $T_{DT}(n,c)$ for Algorithm 6, in case $n>l$, is

$$T_{DT}(n,c) = \sum_{k=1}^{n}(T_{D1}(k,c)+T_{D2}(k,c))$$

$$= 2\sum_{k=1}^{n}((\beta+c\gamma)+(g\beta+c(1-2^{-g})\gamma))$$

$$= 2(n+\frac{l(l+1)}{2}+(n-l)l)\beta$$

$$+2c(2n-1+(1-n+l)2^{-l})\gamma$$

$$\leq 2(n+\frac{n(n+1)}{2})\beta+2c(2n-1+2^{-n})\gamma$$

$$=O(n^2\beta + cn\gamma).$$

If $n \leq l$ then

$$T_{DT}(n,c) = 2(n+\frac{n(n+1)}{2})\beta+2c(2n-1+2^{-n})\gamma$$

$$=O(n^2\beta + cn\gamma).$$

The data transmission time $T_{HN}(n,m)$ from the host to all nodes, not included in Algorithm 6, has to be added to the measured elapsed time. Subproblem instances are transmitted sequentially from the host to the nodes. The number of communications is equal to the number of nodes and the amount of transmitted data is determined by the size of the problem instance. Let β' and γ' be, respectively, the communication setup time and the unit data transfer time between the host and the node. Then,

$$T_{HN}(n,m)=O(2^n\beta'+m\gamma')=O(p\beta'+m\gamma')$$

5.3. Analysis of Total Time and Space

The total elapsed time T_{TT} can be presented as the sum of four components:

$$T_{TT}(p,m,c)=T_{DP}(p,m,c)+T_{CB}(p,m,c)$$

$$+T_{DT}(n,c)+T_{HN}(n,m)$$

$$=O\left(\frac{T_{DP}(1,m,c)}{p}\right)+O(c^2+cn)$$

$$+O(n^2\beta+cn\gamma)+O(2^n\beta'+m\gamma').$$

And the total space M_{TT} used can be approximated as follows:

$$M_{TT}(p,m,c)=O\left(\frac{T_{DP}(1,m,c)}{p}\right)+O(c+n)$$

6. Experimental Results

Three series of experiments with the Knapsack problem were conducted. Each series consists of a number of instances. The number of objects and size of constraint were fixed for each series but weights and profits of objects associated with them were generated by a random number generator.

Series 1 consists of 20 instances of the problem with 300 objects and the size of the knapsack equal to 30. Each instance of the problem was solved by the hypercube with the number of active nodes $p=1,2,4,8,16,32,64$. Then the average elapsed time was computed for all instances. Figure 4 presents the total elapsed time as a function of the number of nodes for 20 instances of the problem. In addition to the total time, two major components of the elapsed time: dynamic programming time (DPT) and combining time (CBT) were measured and presented on the same graph. Experimental data match with theoretical predictions very well. For example, DPT is decreasing two times, each time when the number of nodes is doubled, which exactly coincides with the theoretical result. Combining component of the elapsed time, which consists of data transmission between host and nodes and combining time itself, is growing and the increase as a function of nodes is sublinear. This result also can be expected from the theoretical derivation. The processing time for combining operations is almost independent of the number of nodes. The time for the backtracking part depends linearly on the dimension $\log_2 p$ of the hypercube and the communication time depends quadratically on the dimension of the hypercube. Thus, sublinear behavior of this component of the elapsed time also matches predictions.

The second series of experiments includes six instances of the problem with 1000 objects and knapsack size equal to 100. Results of these experiments are presented in Figure 5. The same type of behavior as for series 1 was registered for this experiment.

The third series of experiments was for the two-dimensional Knapsack problem. It includes 12 instances of the problem with 500 objects and two constraints equal to 10 and 5 respectively.

Results of this experiment are presented on Figure 6. This experiment demonstrates that the same algorithm can be successfully applied to the solution of the multidimensional Knapsack problem. The general behavior of the elapsed time and its components were the same as for the one-dimensional Knapsack problem.

The comparison of these three series of experiments allows us to formulate the following conclusions:

(1) Application of our algorithm for the Knapsack problem on a hypercube provides substantial speedup in elapsed time relative to the time spent by the minimal number of processors sufficient for the solution (sufficiency is determined by adequate total memory). The *efficiency* of this algorithm is defined as the ratio of speedup divided by the number of processors. Table 1 gives the speed up and efficiencies for the different experiments. For the series 1 experiments, these are computed relative to the single processor run time. For the series 2 expreiments, the speed up and efficiency figures are relative to the 8 processor run time. This is because the series 2 test cases could not be solved on fewer processors for lack of sufficient memory. For the same reason, the series 3 speedup and efficiency figures are relative to 2 processors. We note form Table 1 that the efficiency declines monotonically with the increase of the number of processors. We may use this observation to extrapolate the run time from 8 processors for the series 2 experiments to 1 processor and from 2 processors to 1 in the case of the series 3 experiments. When this is done, the speed up and efficiency figures of Table 2 are obtained. For this table, we used a speed up of 1.96 for each doubling of the number of processors from 1 through 16 for the series 1 experiment and a speed up of 1.88 for each doubling of processors from 1 throgh 4 for the series 3 experiment. As is evident, an order of maginitude speedup is projected for the series 2 and 3 experiments.

(2) For large p (say more than 16), our algorithm is expected to result in substantially higher speed up than indicated in Table 1 when instances with larger m are solved. This is due to the fact that the relative weight of the communication time is decreasing relative to the computation time.

(3) There is an optimal number of nodes for each problem size. This number increases with the increase of the problem size and for very large problems the full capacity of a hypercube can be effectively used.

(4) Substantial gain in the size of the solvable problem is achieved by application of this algorithm on a hypercube machine. This gain is based on faster than linear decline of requirement for the memory size of individual processors (Figures 1 and 7). As a result, when the number of processors is increasing the size of the solvable problem is growing faster than linear as a function of the number of processors.

REFERENCES

[GOPA86] P.S. Gopalakrishnan, I.V. Ramakrishnan, L.N. Kanal, "Parallel Approximate Algorithms for the 0-1 Knapsack Problem," Proceedings of the International Conference on Parallel Processing, 1986, pp 444-451.

[HAYE86] John P. Hayes, Trever N. Mudge, Quentin F. Stout, Stephen Colley, John Palmer, "Architecture of a Hypercube Supercomputer," Proceedings of the International Conference on Parallel Processing, 1986, pp. 653-660.

[LI85] Guo-jie Li, Benjamin W. Wah, "Systolic Processing for Dynamic Programming Problems," Proceedings of the International Conference on Parallel Processing, 1985, pp.434-441.

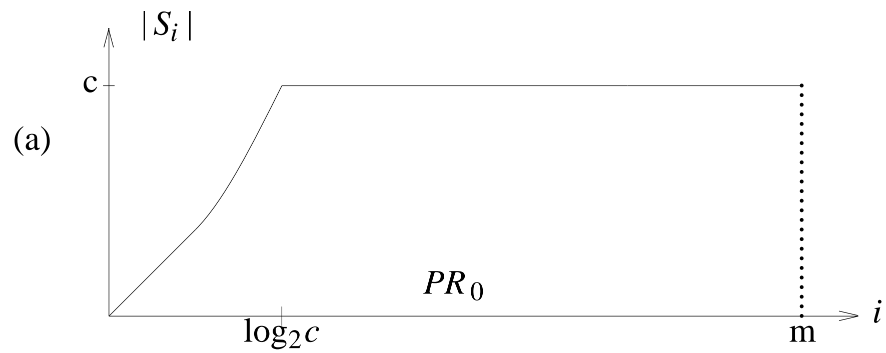
[LIPT86] Richard J. Lipton, Daniel Lopresti, "Delta Transformations to Simplify VLSI Processor Arrays for Serial Dynamic Programming," Proceedings of the International Conference on Parallel Processing, 1986, pp.917-920.

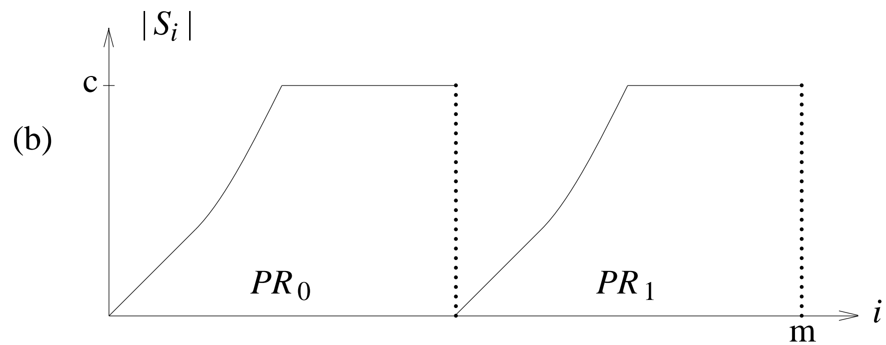
[PETE84] J. Peters, L. Rudolph, "Parallel Approximation Schemes for Subset Sum and Knapsack Problems," 22nd Annual Allerton Conference on Communication, Control and Computing, 1984, pp.671-680.

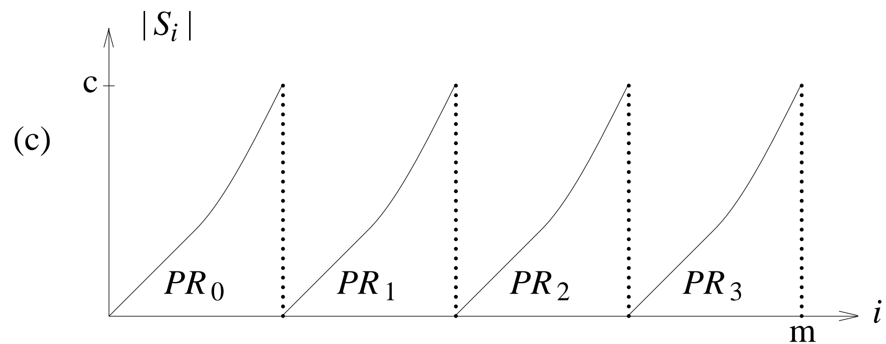
[YAO81] Andrew Chi-Chin Yao, "On the Parallel Computation for the Knapsack Problem," 13th Annual ACM Symposium on Theory of Computing, 1981, pp. 123-127.

Hypercube Computer

- There are 2^n nodes numbered from 0 to $2^n - 1$ and n is a dimension (n-cube)
- Each node of hypercube is a processor with local memory







Function $|S_i|$

- Every node has n neighbor nodes with direct links
- Two neighbor nodes have only one different bit in their binary node numbers

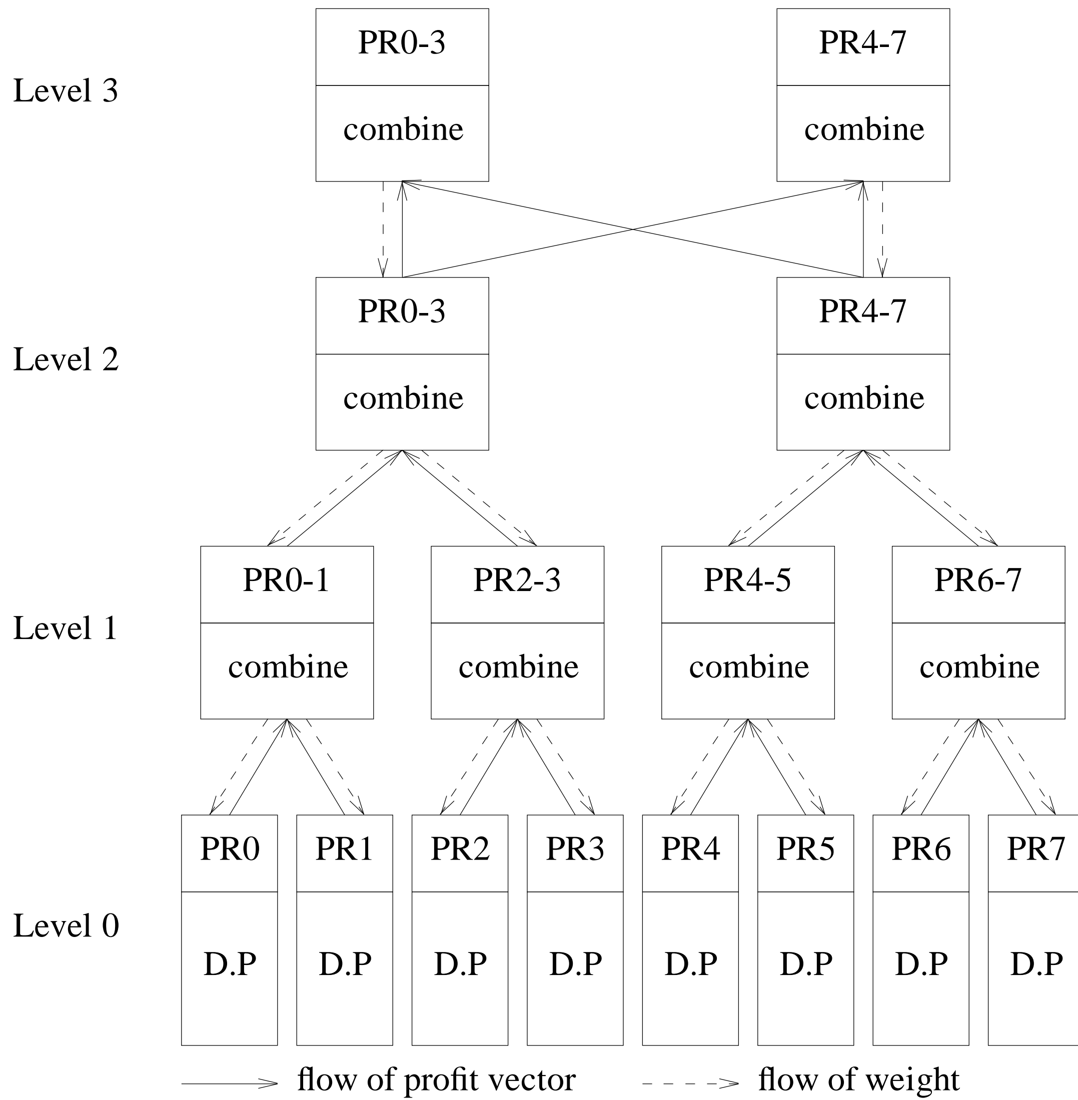
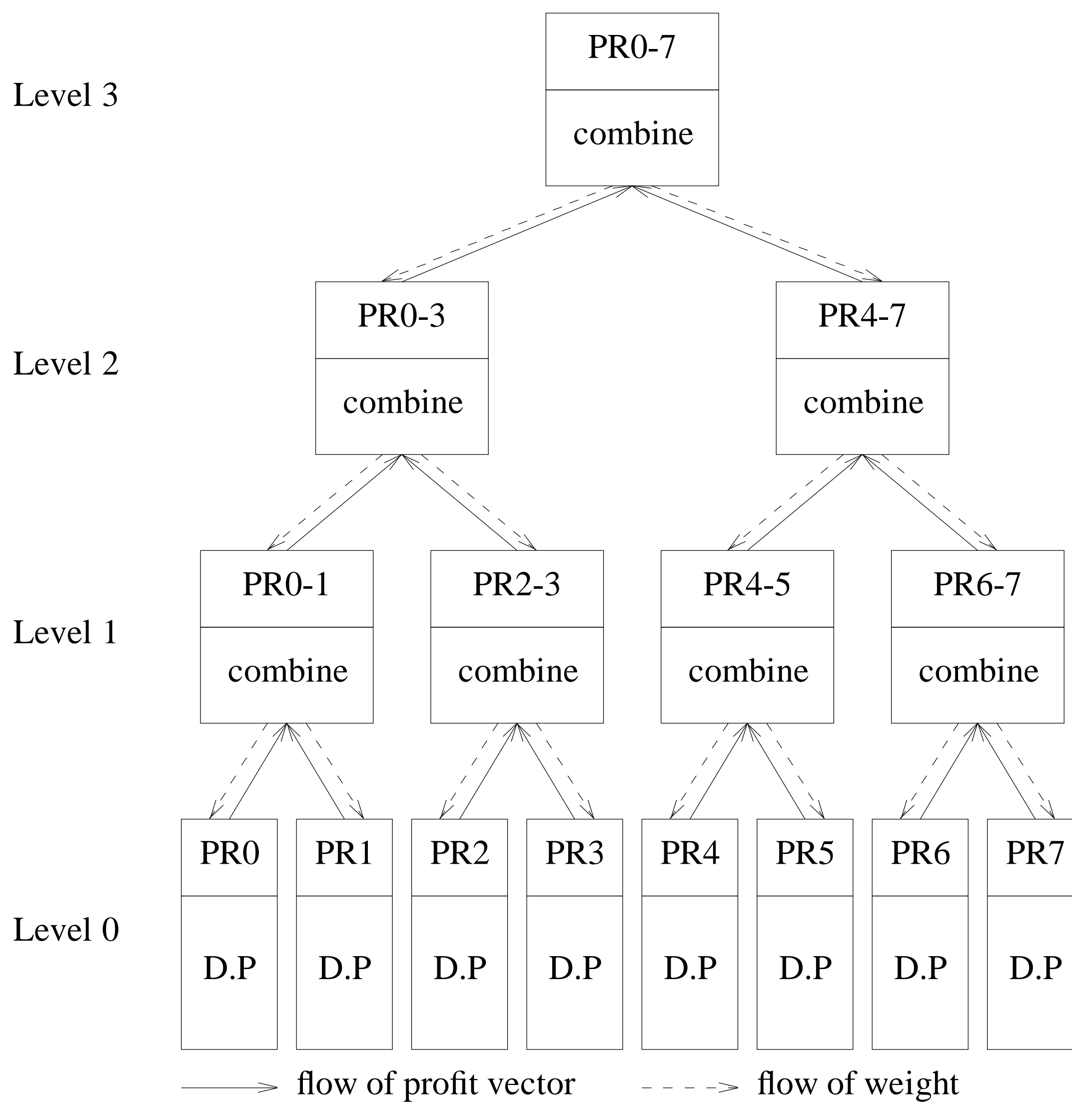
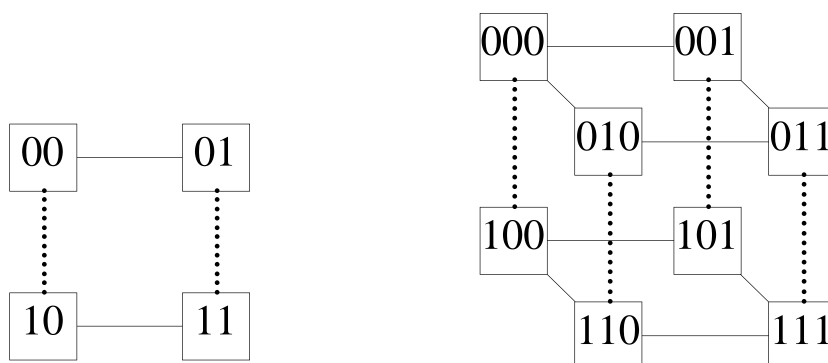


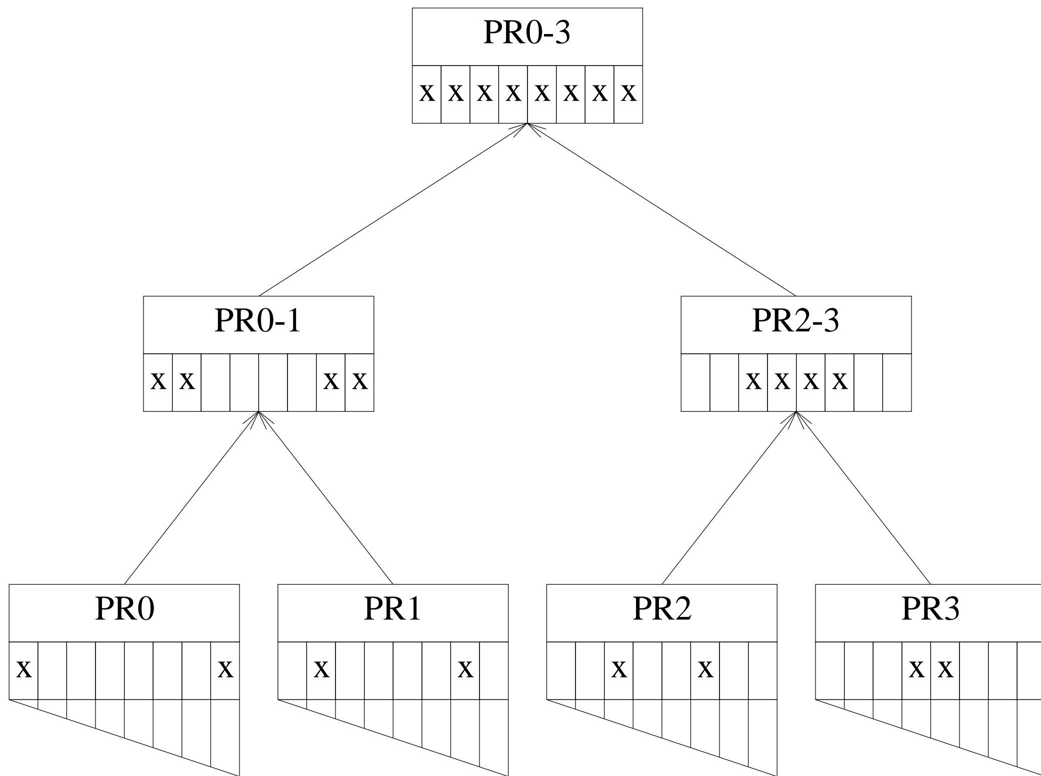
Fig. Combining operation tree

- A n−cube consists of two $(n-1)$−cubes



Combine operation tree

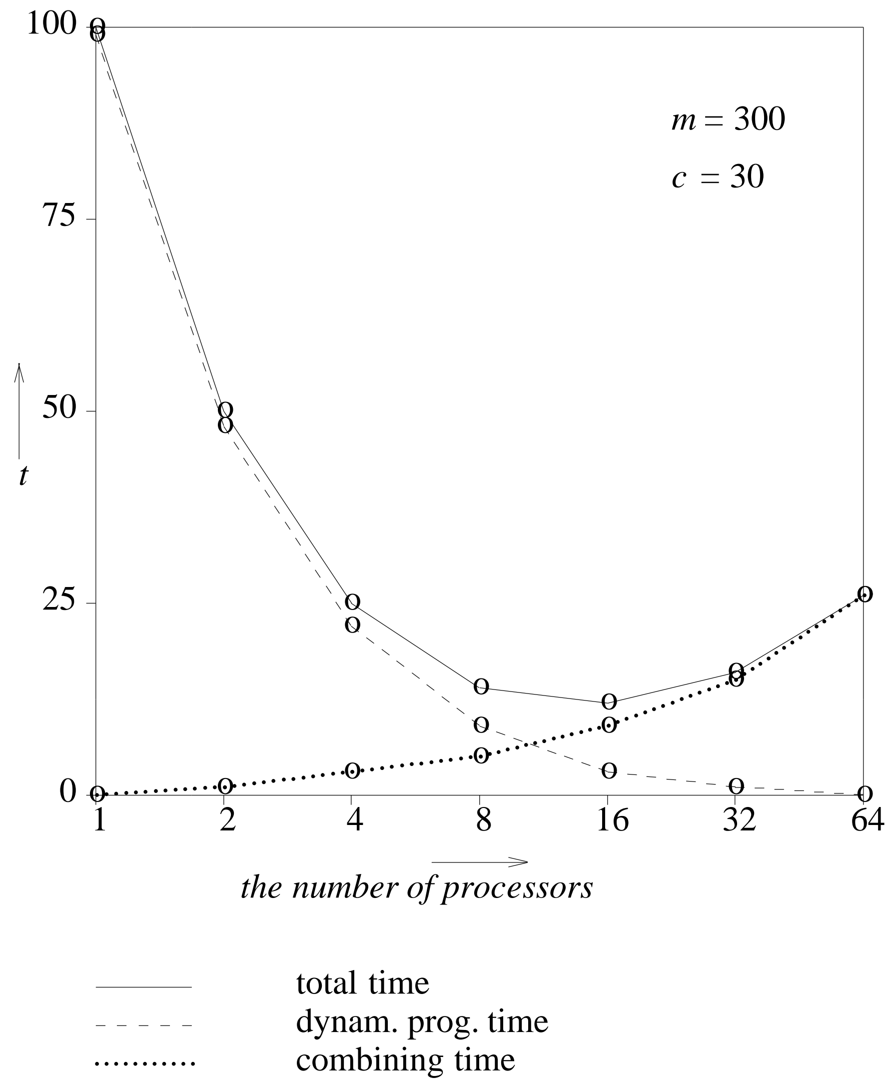




Gathering tree

Sequential Dynamic Programming (SDP)

- Principle of optimality



Components of elapsed time for series-1

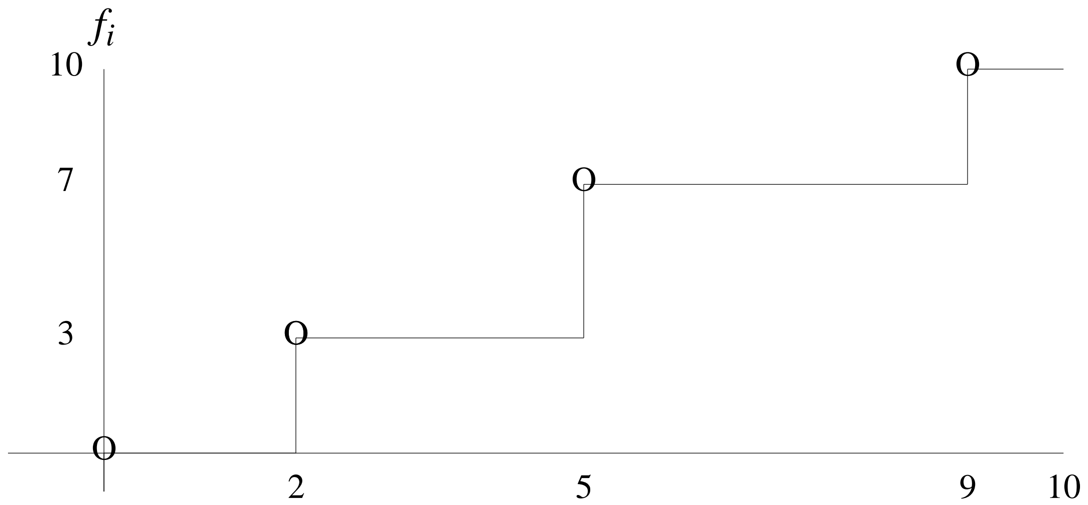
$$f_m(c) = \max\{f_{m-1}(c),\ f_{m-1}(c - w_m) + p_m\}$$

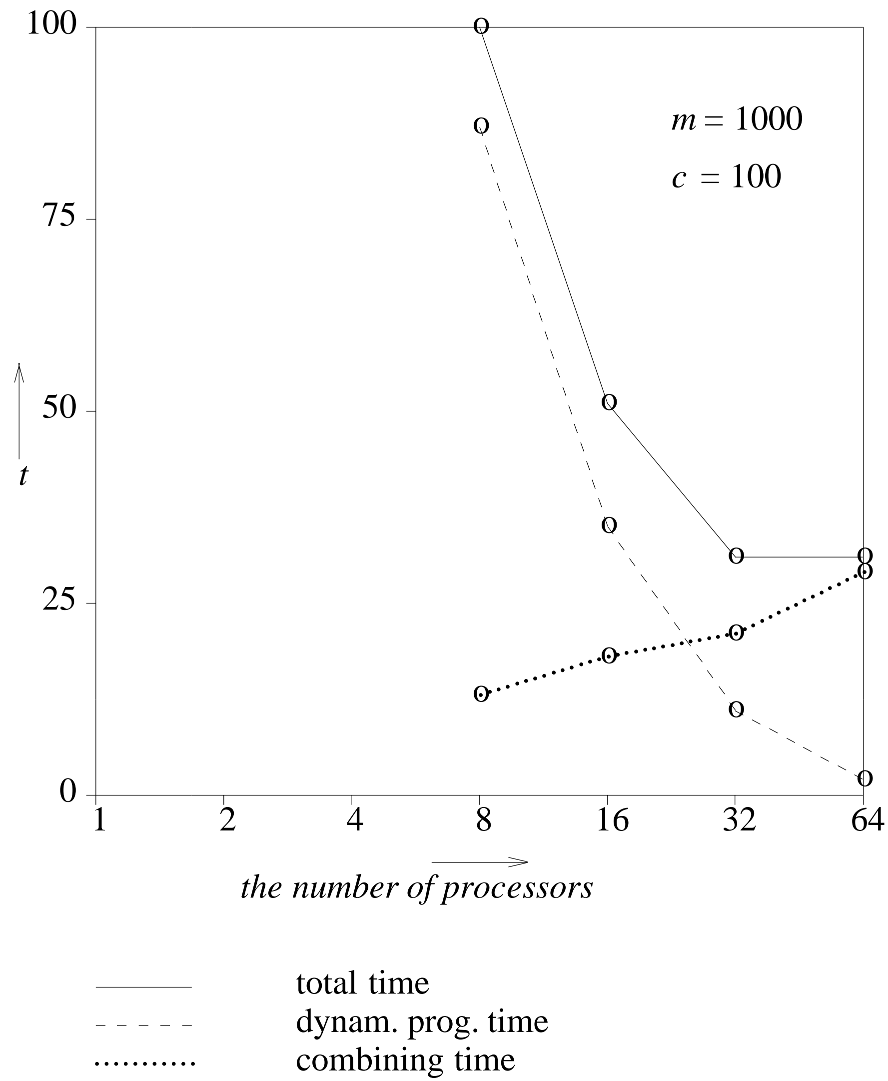
- Generalization

$$\begin{cases} f_0(x) = \begin{cases} 0 & x \geq 0 \\ -\infty & x < 0 \end{cases} \\ f_i(x) = \max\{f_{i-1}(x),\ f_{i-1}(x - w_i) + p_i\}, \quad 1 \leq i \leq m \end{cases}$$

- Data structure for function f_i is an ordered list S_i of tuples

  Example: $S_i = \{(0,0), (3,2), (7,5), (10,9)\}$





Components of elapsed time for series-2

$m = 500$

$c^1 = 10$

$c^2 = 5$

100
75
50
25
0

t

1 2 4 8 16 32 64

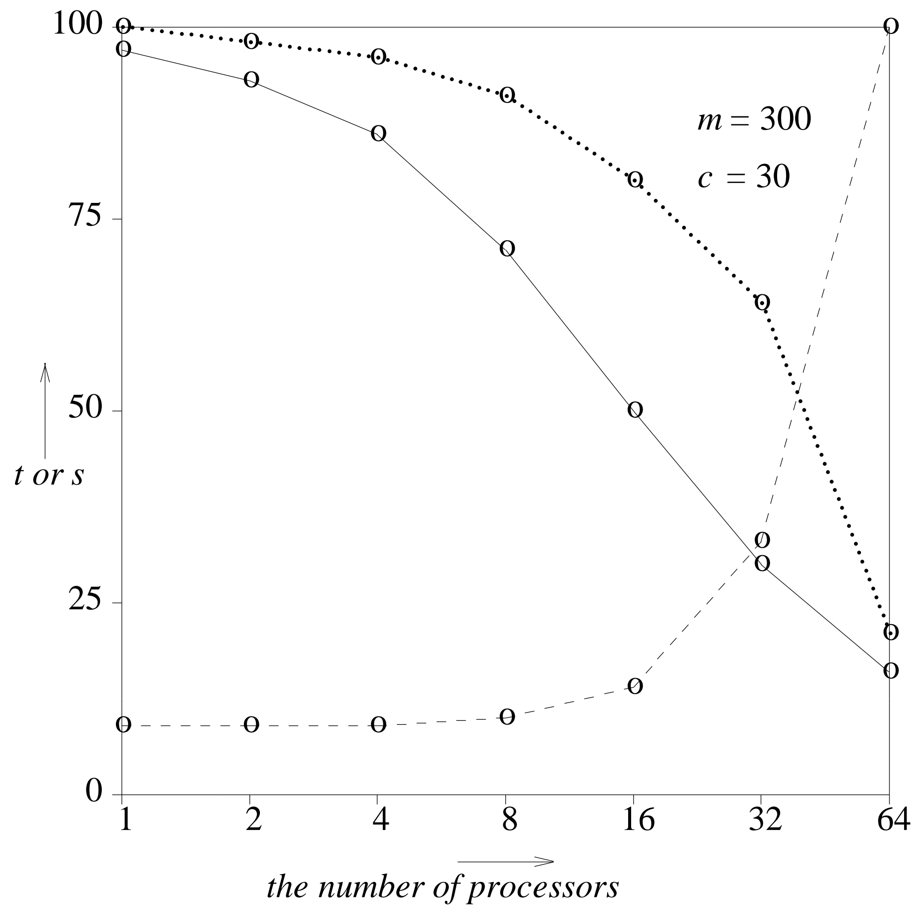
the number of processors

total time

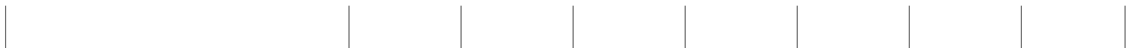
dynam. prog. time

combining time

Components of elapsed time for series-3



Total time and space for series-1



# of processors		1	2	4	8	16	32	64
series 1	speedup	1.00	2.00	3.94	6.68	8.04	6.06	3.74
	efficiency	1.00	1.00	0.99	0.84	0.50	0.19	0.06
series 2	speedup				1.00	1.96	3.13	3.19
	efficiency				1.00	0.98	0.78	0.40

series 3	speedup		1.00	1.88	3.42	5.18	5.65	4.20
	efficiency		1.00	0.94	0.85	0.65	0.35	0.13

Speedup and efficiency table

# of processors		1	2	4	8	16	32	64
series 1	speedup	1.00	2.00	3.94	6.68	8.04	6.06	3.74
	efficiency	1.00	1.00	0.99	0.84	0.50	0.19	0.06
series 2	speedup	1.00	1.96	3.84	7.53	14.76	23.57	24.02
	efficiency	1.00	0.98	0.96	0.94	0.92	0.74	0.38

series 3	speedup	1.00	1.88	3.53	6.43	9.74	10.62	7.90
	efficiency	1.00	0.99	0.88	0.80	0.61	0.33	0.12

Table 2. Projected speed up and efficiency table