An efficient parallel algorithm for solving the knapsack problem on the hypercube*

A. Goldman† and D. Trystram
LMC - IMAG
Domaine Universitaire, B.P. 53
38041 Grenoble Cédex 09 - France
{goldman,trystram}@imag.fr

Abstract

In this paper, we present a new algorithm to solve the integral knapsack problem on the hypercube. The main idea is to use the fact that the precedence graph of the dynamic programming function of the knapsack problem is an irregular mesh.

We propose a scheduling algorithm for irregular meshes on the hypercube. The efficiency of the algorithm is independent on the number of processors. We also present some improvements for the solution of the 0/1 knapsack problem on the hypercube.

Keywords: *Hypercube, knapsack problem, irregular mesh, scheduling.*

1 Introduction

The knapsack problem is one of the most popular problems in combinatorial optimization. There are several algorithms for solving this problem, and one of the best known techniques is the use of dynamic programming.

Computing the solution of the knapsack problem in parallel using dynamic programming can be seen as a scheduling problem for irregular mesh graphs. The main result of this paper is a fast algorithm for solving the knapsack problem on a hypercube. The efficiency of the algorithm is independent on the number of processors.

We first review the more important techniques for finding exact solutions for the knapsack problem in parallel. Then, we introduce the relation between the knapsack problem and irregular mesh graphs. We then present the knapsack and 0/1 knapsack problems, with special attention to its dynamic programming precedence graphs.

In the next section we present a general lower bound for irregular mesh graphs execution, a schedule for the hypercube, and the worst case analysis. In the third section, we will see some possible improvements in the algorithm proposed by Lin and Storer [13].

1.1 State of art

In this paper we are interested in computing in parallel only the exact solutions for the knapsack problem, references to greedy parallel algorithms and approximation parallel algorithms can be found in [8]. There are several approaches to solve the knapsack problem in parallel [8], but very few are hypercube-specific.

Among the main approaches for the 0/1 knapsack resolution is the straightforward method consisting of the enumeration of all possible subsets [7]. Some methods are variations of the two-list algorithm, that is, divide the original problem, enumerate, and recombine the solutions [6], [9] and [10].

Some authors have studied the systolic model. In [5], Chen *et al.* proposed pipelined algorithms for the knapsack and 0/1 knapsack problems. The algorithm for the knapsack problem is based on a queued linear array; the algorithm for the 0/1 knapsack problem uses a linear array with several layers.

In [2], the authors proposed a systolic algorithm that achieves the best possible time to solve the knapsack problem on a linear array with limited memory. In [1], the authors used a two-dimensional systolic array to solve knapsack-like problems.

Teng [15] presented several algorithms for the knapsack problem on a PRAM, using techniques such as reduction to the Circuit Value problem and the Prefix Convolution prob-

*This research is part of the Apache project supported by CNRS, INRIA, INPG and UJF.
†Supported by CNPq (Proc. 200590/95-2). On leave from DCC-IME-USP Brazil.

lem. In all of Teng's solutions, some preprocessing computations requiring $O(\log(mc))$ time are needed, where m is the number of objects in the problem instance and c is the capacity of the knapsack. The time and the number of processors for the three algorithms in [15] are:

time	# processors
$O(\log^2(mc))$	$M(c)$
$O(\log \frac{c}{w_{\max}} \log^2 w_{\max})$	$O(\frac{M(w_{\max} c)}{w_{\max} \log \frac{c}{w_{\max}}})$
$O(\log m \log c)$	$O(\frac{mc^2}{\log m \log c})$

where M denotes the number of processors used for multiplying two n by n matrices in $O(\log n)$ time, and $w_{\max}$ is the maximum of the weights.

There are two well-known hypercube algorithms to solve the 0/1 integral knapsack [13], [11].

In [11], the idea is to solve subproblems of the original problem in parallel using dynamic programming and then to combine the results. For a problem with m objects and knapsack capacity c on p processors, the time is $O(\frac{mc}{p}+c^2)$.

Lin and Storer [13] proposed an efficient algorithm that solves the 0/1 knapsack problem on a PRAM in time $O(\frac{mc}{p})$ (where $p \leq c$). The oblivious implementation on a hypercube has time complexity $O(\frac{mc}{p}\log p)$ and efficiency $O(\frac{1}{\log p})$.

1.2 Precedence task graph

A parallel algorithm based on communicating tasks is usually represented by a directed graph where the nodes represent the tasks (or the instructions) and the arcs represent the precedence constraints between the tasks. A distributed-memory parallel computer can also be represented by a graph where the nodes represent the processors and the edges represent the communication links among the processors.

The parallelization process needs to determine a schedule for the tasks (which consists of assigning to each task an execution time and a processor). In the absence of communications and assuming an unbounded number of processors, it is easy to determine the longest path (also called *makespan*) of the graph. On the other hand, with a given number of processors, with limited memory, the problem becomes NP-hard [14]. Observe that there are some optimal scheduling algorithms for specific problems (such as the regular mesh [4] or trees [3]).

1.3 The knapsack problem

An instance $KNAP(A, c)$ of the knapsack problem consists of a set A of m objects, and an integer knapsack capacity, denoted by c. Each object $a_i \in A$ has positive weight w_i and profit p_i. For each object a_i that is placed in the knapsack, a profit p_i is earned. The objective is to fill the knapsack in such a way that the total profit is maximized.

This problem can be formalized as:

find integers $x_i \geq 0,_{(\text{for } 0 \leq i \leq m-1)}$ to
maximize $\sum_{i=0}^{m-1} p_i x_i$
subject to the constraint $\sum_{i=0}^{m-1} w_i x_i \leq c$.

Define the following function, $f(k, g)$ by

$$\max \left\{ \sum_{i=0}^{k} p_i x_i | \sum_{i=0}^{k} w_i x_i \leq g, x_i \geq 0,_{i=0,\ldots,k} \right\}.$$

Given a knapsack problem instance $KNAP(A, c)$, the profit is equal to $f(m-1, c)$. The dynamic programming iteration for $1 \leq k \leq n$ and $0 \leq g \leq c$ is defined as

$$f(k, g) = \max\{f(k-1, g), p_k + f(k, g - w_k)\}$$

$$\text{with} \begin{cases} f(-1, g) = 0 \\ f(k, 0) = 0 \\ f(k, y) = -\infty, y < 0. \end{cases}$$

Definition 1 *A 2-dimensional irregular mesh $NM(m, c)$ with jumps w_i has as vertices the set of pairs (i, j), where $0 \leq i \leq m-1$, $0 \leq j \leq c-1$. There are arcs from node (i, j) to the nodes $(i+1, j)$ and $(i, j+w_i)$ (if these nodes exist).*

An example of NM is given in figure 1. The usual grid is an irregular mesh NM where $w_i = 1$ for all i.

For the sake of simplicity, we denote the rows of NM by the sets $L_i = \{(i, j), 0 \leq j \leq c-1\}$, and the columns of NM by the sets $C_j = \{(i, j), 0 \leq i \leq m-1\}$. We also use $w_{\min} = \min_i w_i$ and $w_{\max} = \max_i w_i$.

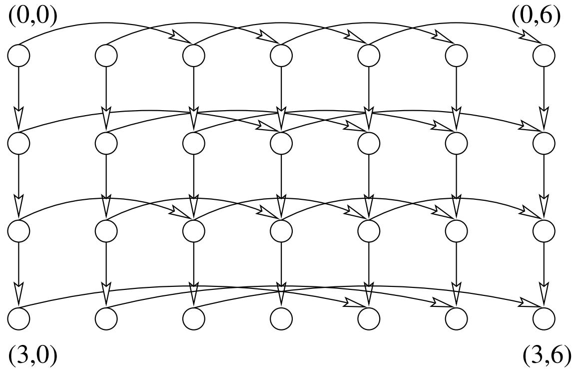

Figure 1. Irregular mesh $NM(4, 7)$ with jumps $w_0 = w_2 = 2$, $w_1 = 3$ and $w_3 = 4$.

The precedence graph to compute $f(k,g)$ is an irregular mesh $NM(m, c+1)$ with jumps $(w_i)_{i=0}^{m-1}$. Each task (i,j) of the irregular mesh receives at most two data. The task (i,j) executes the following:

If $i = 0$ and $j < w_0$
  send 0 to $(i+1, j)$, and p_0 to $(i, j+w_0)$.

If $i = 0$ and $j \geq w_0$
  receive d from $(i, j - w_0)$,
  send d to $(i+1, j)$ and $d + p_0$ to $(i, j+w_0)$.

If $i > 0$ and $j < w_i$
  receive d from $(i-1, j)$,
  send d to $(i+1, j)$ and $d + p_i$ to $(i, j+w_i)$.

If $i > 0$ and $j \geq w_i$
  receive d_1 from $(i-1, j)$ and
    d_2 from $(i, j - w_i)$,
  calculate $d = \max\{d_1, d_2\}$,
  send d to $(i+1, j)$ and $d + p_i$ to $(i, j+w_i)$.

To solve $KNAP(A,c)$, the tasks can send a bit array. The bit array is built in such a way that each position is set if $i > 0$ and $j \geq w_i$ and $\max\{d_1, d_2\} = d_2$. This array will have at most $\left\lceil \frac{c}{w_{\min}} \right\rceil$ elements. With this array it is easy to find the objects used in the maximum profit solution.

1.4 The 0/1 knapsack problem

The 0/1 knapsack problem is similar to the integral knapsack problem. The difference is that at most one object of each type can be used in the 0/1 problem ($x_i \in \{0,1\}$).

We can solve this problem using an irregular mesh. A task that receives a value from the left sends $-\infty$ to the right. If a task receives two equal values, then it chooses the value from the upper precedence task.

Given an instance 0/1 KNAP(A,c), the dynamic programming function for $1 \leq k \leq m$ and $0 \leq g \leq c$ is

$$f(k,g) = \max\{f(k-1,g), p_k + f(k-1, g-w_k)\}$$

$$\text{with } \begin{cases} f(-1,g) = 0 \\ f(k,0) = 0 \\ f(k,y) = -\infty, y < 0. \end{cases}$$

The precedence graph for computing this function has the set of the pairs (i,j) as vertices, where $0 \leq i \leq m$, $0 \leq j \leq c$. There are arcs from node (i,j) to the nodes $(i+1, j)$ and $(i+1, j+w_i)$ (an example is given in figure 2).

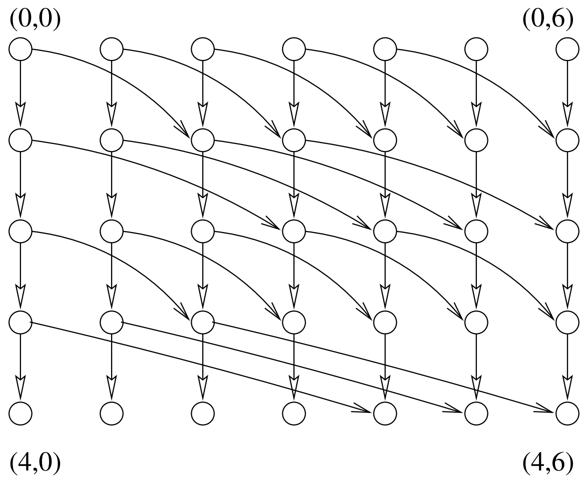

Figure 2. 0/1 knapsack problem presentation with capacity 7 and 4 objects (object weights $w_0 = w_2 = 2$, $w_1 = 3$ and $w_3 = 4$).

Each task (i,j)

If $i = 0$
  send 0 to $(i+1, j)$, and p_0 to $(i+1, j+w_0)$.

If $i > 0$ and $j < w_{i-1}$
  receive d from $(i-1, j)$,
  send d to $(i+1, j)$ and $d + p_i$ to $(i+1, j+w_i)$

If $i > 0$ and $j \geq w_{i-1}$
  receive d_1 from $(i-1, j)$ and
    d_2 from $(i-1, j - w_{i-1})$,
  calculate $d = \max\{d_1, d_2\}$,
  send d to $(i+1, j)$ and $d + p_i$ to $(i+1, j+w_i)$

One can easily verify that this graph can be executed on a PRAM with p processors ($p \leq c$) within $O(\frac{mc}{p})$ steps ($O(m)$ steps if $p > c$). The oblivious execution on a hypercube has time $O(\frac{mc}{p} \log p)$. This is another way to look at Lin and Storer's algorithm [13].

2 Irregular mesh scheduling

2.1 General lower bound

It is well-known that the minimum execution time of a precedence graph is greater than or equal to the length of its longest path. For the case of the irregular mesh, this leads to the following result:

Proposition 1 *The length of the longest path of a irregular mesh, $NM(m,c)$, with jumps $(w_i)_0^{m-1}$ is equal to* $m + \left\lceil \frac{c}{w_{\min}} \right\rceil - 1$.

Proof: Given i, $0 \leq i < m-1$, each task (i,j) of $NM(m,c)$ precedes the task $(i+1,j)$. There are m rows in $NM(m,c)$ with jumps w_i on row i. The length of the paths in each row i is $\left\lfloor \frac{c}{w_i} \right\rfloor$ or $\left\lceil \frac{c}{w_i} \right\rceil$, and the length of the longest path on the rows is $\left\lceil \frac{c}{w_{\min}} \right\rceil$. Hence, the longest path of $NM(mc)$ has $m + \left\lceil \frac{c}{w_{\min}} \right\rceil - 1$ arcs. □

2.2 Execution on hypercube

In this section we present an irregular mesh execution on the hypercube. For each task, we give the processor and the initial execution time. For the communications, we use the synchronous store and forward model, where each vertex can execute a send and/or receive operation with all of its neighbours at each step [12].

Given an irregular mesh $NM(m,c)$ with $(w_i)_{i=0}^{m-1}$ jumps, we show its schedule on a hypercube with at least $w_{\max}$ vertices.

2.2.1 Principles of the algorithm

Given a hypercube, $H(n)$, with $p = 2^n$ vertices, we will consider the Cartesian Product $H(n) = H(\lceil \log_2 w_{\max} \rceil) \times H(n')$. For the algorithm description, we will label the $2^{n'}$ hypercubes $H(\lceil \log_2 w_{\max} \rceil)$ as $H_0, \ldots, H_{2^{n'}-1}$, in such way that two consecutive hypercubes are adjacent. A detailed example is given in figure 3.

Let P be the Hamiltonian path (generated from the reflected Gray Codes [12]) of $H(\lceil \log_2 w_{\max} \rceil)$. We use the Hamiltonian path P_i associated with each H_i.

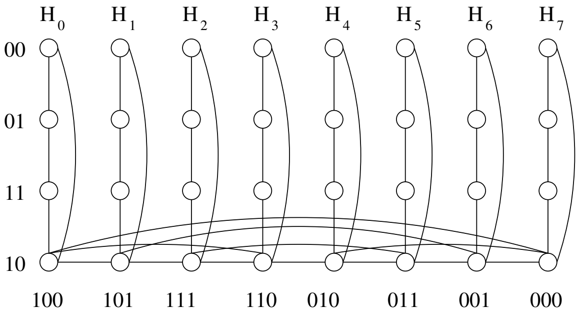

Figure 3. Example of decomposition of $H(5)$ into $H(2) \times H(3)$. Some horizontal edges have been omitted to simplify the figure.

The main idea of the algorithm is to execute each row L_i of $NM(m,c)$ in one hypercube H_i. As we know, each row i of $NM(m,c)$ has w_i chains of size less or equal than $\left\lceil \frac{c}{w_i} \right\rceil$.

We allocate the tasks from the chain j of the row i ($\{(x,y) | x = i, y \bmod w_i = j\}$) to the j-ary processor of the Hamiltonian path $P_{i \bmod 2^{n'}}$. We know that

$$2^{n'} = \frac{p}{2^{\lceil \log_2 w_{\max} \rceil}}.$$

To simplify the development we suppose that $2^{n'} = p/w_{\max}$ (i.e. $log_2 w_{\max}$ is integer). Observe that if this is not true, there will not be any changes in the complexity calculation.

2.2.2 Analysis

It is easy to verify that the algorithm satisfies the following properties:

- There are no precedence relation among tasks from different processors of the same Hamiltonian path P_i.
- The tasks in the processors of path $P_i, i > 0$ have precedence relations only with the tasks in the path $P_{(i+1) \bmod (p/w_{\max})}$.
- The maximum distance among tasks (with precedence relation) at rows P_i and $P_{(i+1) \bmod (p/w_{\max})}$ is $\log_2 w_{\max} + 1$.

2.2.3 Complexity

Now, we determine the execution times. There are two different cases to consider depending on the value of p:

- $p \geq \left\lceil \frac{c}{w_{\min}} \right\rceil \left\lceil \frac{w_{\max}}{\log_2 w_{\max} + 1} \right\rceil$.

  We suppose that once a chain is started, its tasks are executed in the subsequent steps. If a row starts its execution with w' tasks, at each of the subsequent steps w' more tasks will be executed. The tasks are always executed in an increasing column order.

  For the routing between adjacent hypercubes we use the following scheme. Whenever a hypercube has finished processing a set of tasks it sends the results to the next hypercube (according to the labeling $H_0, H_1, \ldots$). Each result has to be routed to the processor that will execute the tasks depending on the results.

  Generally at each step, a row (hypercube) $i \bmod (p/w_{\max})$ receives w' results to be routed. These results have to be routed to w_i processors. The idea for avoiding conflicts is to route by dimensions using the pipeline technique. The first w' results are first routed in the hypercube's dimension 0. In the next step, the second w' results are routed in the dimension 0, while the first w' results are routed in dimension 1.

After receiving w' results j times, the j-th set of results is routed on dimension 0, and the $(j-x)$-th set of results ($x \leq \log_2 w_{\max}$) is routed on dimension x. This will assure the routing of the results to satisfy the precedence relations within $\log_2 w_{\max} + 1$ steps.

We show in figure 4 the notation that will be used in the partial execution example (figure 5) for an irregular mesh with $w_0 = 3$ and $w_1 = w_{\max} = 8$.

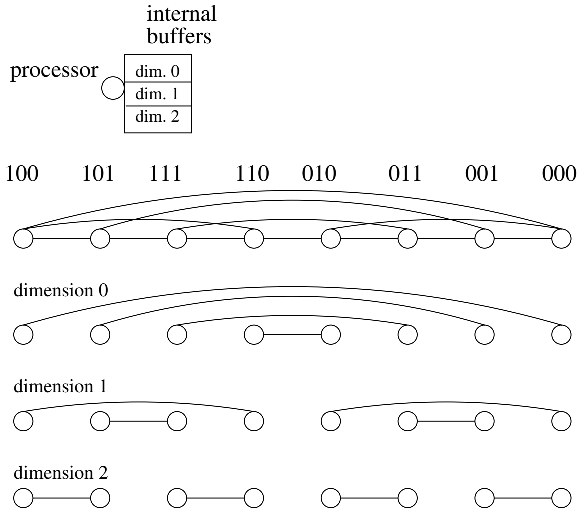

Figure 4. The hypercube and its links on each dimension.

In figures 4 and 5, on each processor (circle) we place the task being processed. On the right we have three buffers, for transmissions on dimensions 0, 1 and 2. Each of the $\log_2 w_{\max}$ buffers can store at most $\log_2 w_{\max}$ data. This means that we send at most $\log_2 w_{\max}$ data on a link at each step.

- The chains from row 0 start their execution at time 0; at time 1 w_0 tasks are ready.
- Due to the precedence relations, $\min\{w_0, w_1\}$ tasks from row 1 start their execution at time $\log_2 w_{\max} + 1$.
- At time $2\log_2 w_{\max} + 2$, $\min\{w_0, w_1, w_2\}$ tasks from row 2 will start their execution.
- In general, at time $i(\log_2 w_{\max}+1)$, $\min\{w_j | j \leq i\}$ tasks from row i start their execution. At time $p/w_{\max}(\log_2 w_{\max} + 1)$ the tasks from the first row will be finished $\left(p \geq \left\lceil \frac{c}{w_0} \right\rceil \left\lceil \frac{w_{\max}}{\log_2 w_{\max}+1} \right\rceil\right)$, so the row $p/w_{\max}$ can start its execution without delay. This is valid for all rows i, with $i \bmod p/w_{\max} = 0$.

The last row begins its execution at time $(m-1)(\log_2 w_{\max} + 1)$, and finishes its execution at time $(m-1)(\log_2 w_{\max} + 1) + \left\lceil \frac{c}{w_{\min}} \right\rceil$.

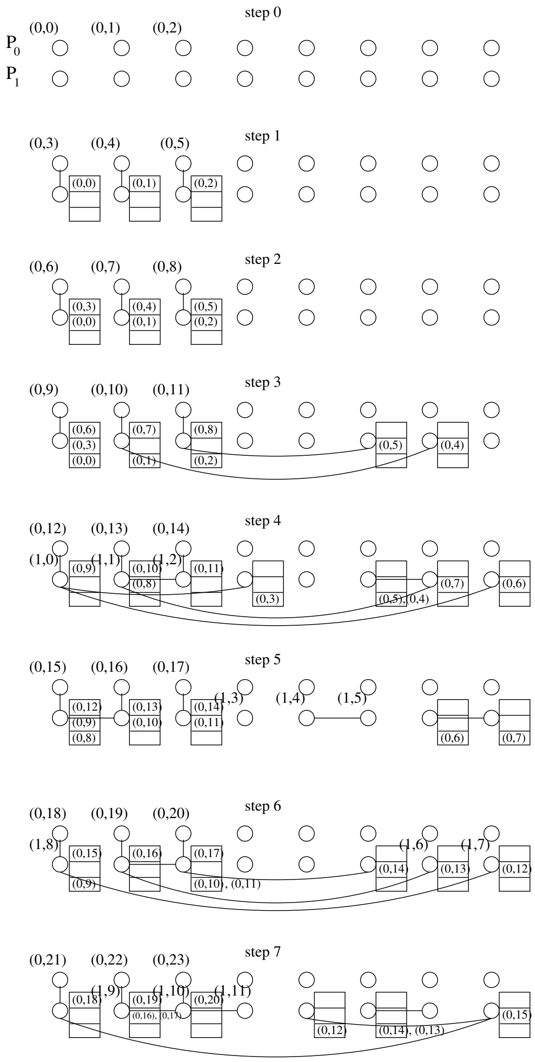

Figure 5. Partial execution example, with $w_0 = 3$ and $w_1 = w_{\max} = 8$.

- $p < \left\lceil \frac{c}{w_{\min}} \right\rceil \left\lceil \frac{w_{\max}}{\log_2 w_{\max}+1} \right\rceil$.

  For the initial rows this case is the same as above. For the row $p/w_{\max}$, we have to wait for the completion of the first row. In general we have to await the completion of row $L_i, i \bmod (p/w_{\max}) = 0$ to start row $L_{i+(p/w_{\max})}$.

The waiting time, at each allocation on path P_0, is $\left(\left\lceil \frac{c}{w_{\min}} \right\rceil - \frac{p}{w_{\max}}(\log_2 w_{\max}+1)\right)$. Before each waiting time there will be a processing time of $p/w_{\max}(\log_2 w_{\max}+1)$. Since there are $\left\lfloor \frac{m w_{\max}}{p} \right\rfloor$ such situations, the total time will be at most

$$\left\lceil \frac{c}{w_{\min}} \right\rceil + \left\lceil \frac{m w_{\max}}{p} \right\rceil \Bigg(\quad \left(\left\lceil \frac{c}{w_{\min}} \right\rceil - \right.$$

$$\left. \frac{p}{w_{\max}}(\log_2 w_{\max}+1) \right) + \frac{p}{w_{\max}}(\log_2 w_{\max}+1) \Bigg),$$

that is,

$$m(\log_2 w_{\max}+1) + \left\lceil \frac{c}{w_{\min}} \right\rceil +$$

$$\left(\left\lceil \frac{c}{w_{\min}} \right\rceil - \frac{p}{w_{\max}}(\log_2 w_{\max}+1) \right) \left\lfloor \frac{m w_{\max}}{p} \right\rfloor.$$

Now we summarize the results.

Proposition 2 *The total processing time of an irregular mesh is $O(m \log w_{\max} + \frac{c}{w_{\min}})$ using $O(\frac{c w_{\max}}{w_{\min} \log w_{\max}})$ processors.*

For the case with fewer processors, that is $p < \left\lceil \frac{c}{w_{\min}} \right\rceil \left\lceil \frac{w_{\max}}{\log_2 w_{\max}} \right\rceil$, we have to consider the waiting time.

Proposition 3 *The total processing time of an irregular mesh is $O\left(\frac{mc}{p}\frac{w_{\max}}{w_{\min}}\right)$ using p processors.*

The efficiency of the hypercube algorithm is $O\left(\frac{w_{\min}}{w_{\max}}\right)$.

2.2.4 Remarks

The efficiency of the previous algorithm is independent on the number of processors. There exist some ways to improve the execution time of the algorithm.

The first idea is to rearrange the objects in decreasing order. At each step the number of available tasks to be processed will be the greatest. This way we will realize the maximum possible parallelism

We can also execute a task as soon as its precedence results arrive at the right processor, thereby avoiding a wait of $\log_2 w_{\max}$ steps for each task.

3 Resolution of the 0/1 knapsack problem

To improve the execution of the 0/1 knapsack problem proposed in [13] we consider a special case. Given a 0/1 knapsack problem, it is difficult to assign the tasks to the processors of a hypercube with p processors in such way that adjacent tasks in the precedence graph are at a distance smaller than $\log_2 p$ in the hypercube.

Given a 0/1 knapsack problem where all object weights are power of two (and greater than one), we can solve the problem on a hypercube with $p \leq \log_2 m$ processors, as fast as on the PRAM theoretical model. First of all we build the dynamic programming precedence graph, with the objects organized in increasing order.

We work with the hypercube organized as a linear array, with the nodes ordered in increasing label order (figure 6).

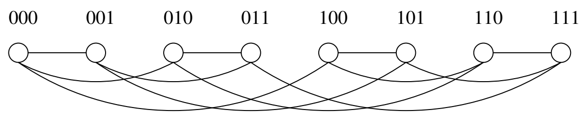

Figure 6. Hypercube with eight vertices.

In the first step the algorithm executes the first p tasks from the first row. In the second step the algorithm sends the data according to the precedence relations. In the third step the algorithm executes the first p tasks from the second row. Continuing this way, at step $2i+1$ ($i < m$) the algorithm executes the first p tasks from the row i (processing step), and at the step $2(i+1)$ sends the data according to the precedence relations (communication step) (examples in figures 7 and 8).

At the same time as the first p tasks of a row are executing the communication steps, the data according to the precedence relations of the second p tasks is sent. This is always possible because p is smaller than $\log_2 m$.

On step $2m+1$ the algorithm executes the second p tasks from the first row, and so on. This way, at step $2i+1$ the algorithm executes the $\left\lceil \frac{i}{m} \right\rceil$-th p tasks (or maybe less, if $c \bmod p \neq 0$) from the $i \bmod m$-th row. At step $2(i+1)$ the algorithm sends data according to the precedence relations.

We can see that the last tasks from row m will be processed at time $2m\left\lceil \frac{c}{p} \right\rceil$, that is $O(\frac{mc}{p})$.

There are other special cases, such as the case where all object weights have consecutive values. In this case we organize the nodes of the hypercube as a Hamiltonian cycle.

Our main interest was to show this new technique and not to present an exhaustive list of special cases.

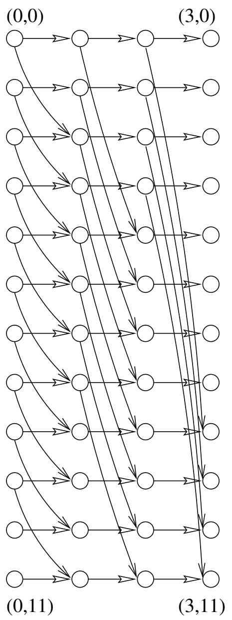

Figure 7. Task precedence graph presentation of the 0/1 knapsack problem of capacity 12 with 3 objects (with weights 2, 4 and 8).

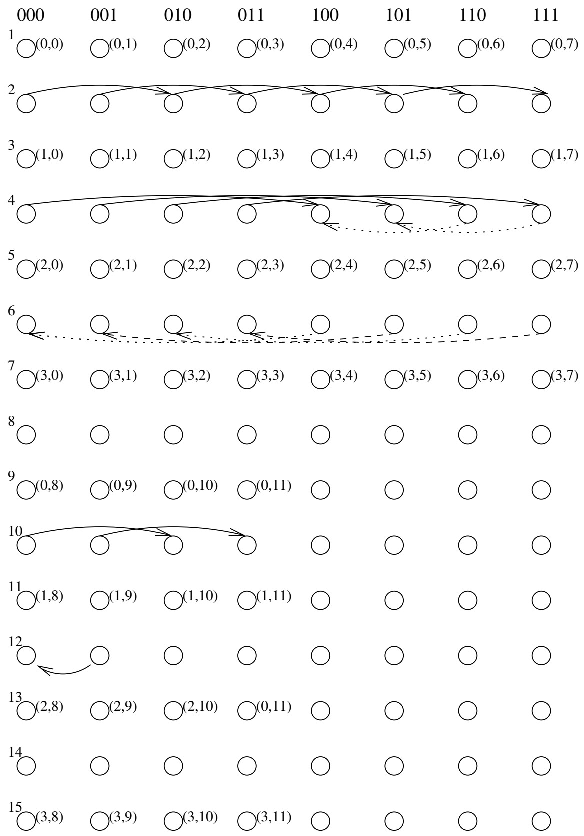

Figure 8. Execution of the 0/1 knapsack problem on a hypercube with 8 vertices.

4 Concluding remarks

We have presented in this paper an efficient algorithm for solving the knapsack problem on the hypercube.

The algorithm that we propose solves the knapsack and 0/1 knapsack problems with $O(\frac{cw_{\max}}{w_{\min}\log w_{\min}})$ processors in $O(m\log w_{\max} + \frac{c}{w_{\min}})$ steps. With fewer processors, our algorithm solves the knapsack problem within $O(\frac{mc}{p}\frac{w_{\max}}{w_{\min}})$ steps.

This new algorithm achieves a good efficiency which is independent on the number of processors. It is very efficient for problems where the sizes of the objects have the same order. If it is not the case, some derivatives can be found in order to improve the efficiency.

The challenge now is to find an optimal algorithm to solve the knapsack problem on a hypercube. We are trying to improve the algorithm to achieve an efficiency of $O(1)$.

References

[1] V. Aleksandrov and S. Fidanova. On the expected execution time for a class of non uniform recurrence equations mapped onto 1d array. *Parallel Algorithms and Applications*, 1:303 – 314, 1994.

[2] R. Andonov and P. Quinton. Efficient linear systolic array for knapsack problem. *Technical Report - IRISA - France*, 1(639), 1992.

[3] E. Bampis, F. Guinand, and D. Trystram. Some models for scheduling parallel programs with communication delay. Discrete Applied Mathematics, special issue on scheduling, to appear.

[4] E. Bampis, J.-C. König, and C. Delorme. Optimal schedule for d-d grid graphs with communication delays. Technical Report Université d'Evry, to appear in Parallel Computing.

[5] G. Chen, M. Chern, and J. Jang. Pipeline architectures for dynamic programming algorithms. *Parallel Computing*, 1(13):111 – 117, 1990.

[6] M. Cosnard, A. Ferreira, and H. Herbelin. The two list algorithm for the knapsack problem on a fps t20. *Parallel Computing*, (9):385 – 388, 1988/89.

[7] T. Gerasch and P. Wang. Implementing dynamic programming on the connection machine. Technical Report Series TR-10-89, George Mason University, 1989.

[8] T. Gerasch and P. Wang. A survey of parallel algorithm for one-dimensional integer knapsack problems. *INFOR*, 32(3):163–186, 1993.

[9] E. Horowitz and S. Sahni. Computing partitions with applications to the knapsack problem. *JACM*, (21):277 – 292, 1974.

[10] E. Karnin. A parallel algorithm for the knapsack problem. *IEEE Transactions on Computers*, 5(33):404 – 408, 1984.

[11] J. Lee, E. Shragowitz, and S. Sahni. A hypercube algorithm for the 0/1 knapsack problem. *Journal of Parallel and Distributed Computing*, (5):438–456, 1988.

[12] F. Leighton. *Parallel Algorithms and Architectures: Arrays - Trees - Hypercubes*. Morgan Kaufmann, 1992.

[13] J. Lin and J. Storer. Processor efficient hypercube algorithm for the knapsack problem. *Journal of Parallel and Distributed Computing*, 3(13):332 – 337, 1991.

[14] C. Picouleau. New complexity results on scheduling with small communication delays. *Journal of Discrete Applied Mathematics*, 1-3(60):331 – 342, 1995.

[15] S. Teng. Adaptive parallel algorithms for integral knapsack problems. *Journal of Parallel and Distributed Computing*, (8):400–406, 1990.