Available online at www.sciencedirect.com
SCIENCE DIRECT®
J. Parallel Distrib. Comput. 64 (2004) 1213–1222
Journal of Parallel and Distributed Computing
www.elsevier.com/locate/jpdc

An efficient parallel algorithm for solving the Knapsack problem on hypercubes

A. Goldman[1], D. Trystram*

ID - IMAG, 51 Rue Jean Kuntzman, 38330 Montbonnot St. Martin, France

Received 21 September 1999; received in revised form 12 April 2001
Available online 18 Septemeber 2004

Abstract

We present in this paper an efficient algorithm for solving the integral Knapsack problem on hypercube. The main idea is to represent the computations of the dynamic programming formulation as a precedence graph (which has the structure of an irregular mesh). Then, we propose a time optimal scheduling algorithm for computing the irregular meshes on hypercube.
© 2004 Elsevier Inc. All rights reserved.

Keywords: Hypercube; Knapsack problem; Irregular mesh; Scheduling

1. Introduction

The Knapsack problem is one of the most popular problems in combinatorial optimization, this problem is known to be *NP*-hard [13]. Using dynamic programming the problem can be solved serially in pseudo-polynomial time.

Computing the solution of the Knapsack problem in parallel using dynamic programming can be seen as solving a scheduling problem for irregular mesh graphs. The main contribution of this paper is the design of an efficient schedule.

After presenting a mathematical formulation of the Knapsack problem using dynamic programming, we detail the relation between this problem and irregular mesh graphs. Similarly, we analyze the $0/1$ knapsack problem with special attention to its dynamic programming precedence graph.

Then, we review the more important techniques for finding exact Knapsack problem solutions in parallel. We compare the existing solutions with our solution and we present a general lower bound for irregular mesh graphs execution. Then, we extend our previous work [16], an algorithm for solving the Knapsack problem on Hypercube, to obtain a better algorithm, whose efficiency is asymptotically equal to 1 for large Knapsack capacities.

2. Solving the Knapsack problem with dynamic programming

2.1. Mathematical formulation

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 of A has a positive weight w_i. For each object of type a_i that is placed in the knapsack, a positive profit p_i is earned. From now on the objects placed on the knapsack will be called items. The objective is to fill the knapsack in such a way that the total profit is maximized.

* Corresponding author. Fax: +33-4-76-61-20-99.
E-mail addresses: gold@ime.usp.br (A. Goldman), trystram@imag.fr (D. Trystram).
[1] Supported by CNPq (200590/95-2); On leave from University of São Paulo, Brazil.

0743-7315/$ - see front matter © 2004 Elsevier Inc. All rights reserved.
doi:10.1016/j.jpdc.2002.10.001

This problem can be formalized as

find integers $x_i \geqslant 0$, (for $0 \leqslant i \leqslant m-1$) to

maximize $\sum_{i=0}^{m-1} p_i x_i$

subject to the constraint $\sum_{i=0}^{m-1} w_i x_i \leqslant c$.

Define the following function:

$$f(k,g) = \max\left\{\sum_{i=0}^{k} p_i x_i \,|\, \sum_{i=0}^{k} w_i x_i \leqslant g,\ x_i \text{positive integers},\ i = 0,\ldots,k\right\}.$$

Given a Knapsack problem instance KNAP(A, c), the maximal profit is equal to $f(m-1, c)$. The dynamic programming iteration for integers $0 \leqslant k < m$ and $0 \leqslant g \leqslant c$ is defined as follows:

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

2.2. Representation

A parallel algorithm based on communicating tasks is usually represented by a directed acyclic graph where the nodes represent the tasks (or the instructions) and the arcs represent the precedence constraints between the tasks [10]. The parallelization process needs to determine a schedule for the tasks (which consists of assigning to each task an execution time and a processor location).

In the absence of communications and assuming an unbounded number of processors, it is easy to determine the longest path of the graph, that corresponds to the scheduling time (also called *makespan*). On the other hand, with constrained resources (such as limited memory or bounded number of processors), the problem becomes *NP*-hard [11] even for the simplest communication model [25]. Observe that there are some optimal scheduling algorithms for specific problems (such as the regular mesh [7]).

Definition 1. A two-dimensional irregular mesh $NM(m,c)$ with jumps $(w_i)_{i=0}^{m-1}$ is the directed graph whose nodes are the set of pairs (i,j) (for $0 \leqslant i \leqslant m-1$ and $0 \leqslant j \leqslant c$). Given a node (i,j), if $0 \leqslant i < m-1$ there is an arc from (i,j) to $(i+1,j)$, if $0 \leqslant j \leqslant c - w_i$ there is an arc from (i,j) to $(i, j+w_i)$.

An example of *NM* with 4 rows and 7 columns is given in Fig. 1. This irregular mesh has jumps $(2,3,4,5)$, and corresponds to the precedence graph of KNAP$(A, 6)$, with

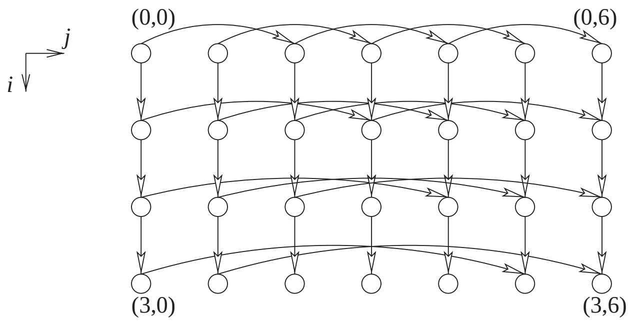

Fig. 1. Irregular mesh $NM(4,7)$ with jumps $(w_i)_{i=0}^{3} = (2,3,4,5)$.

objects of weight 2, 3, 4 and 5. For the sake of simplicity, we denote the rows by the sets $L_i = \{(i,j), 0 \leqslant j \leqslant c\}$. We also use $w_{\min} = \min_i\{w_i\}$ and $w_{\max} = \max_i\{w_i\}$.

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

```
If $i = 0$ and $j < w_0$      if $(m \geqslant 2)$ send 0 to $(i+1, j)$,
                              if $(j \leqslant c - w_0)$ send $p_0$ to
                                $(i, j + w_0)$.
If $i = 0$ and $j \geqslant w_0$   receive $d$ from $(i, j - w_0)$,
                              if $(m \geqslant 2)$ send $d$ to $(i+1, j)$,
                              if $(j \leqslant c - w_0)$ send $d + p_0$ to
                                $(i, j + w_0)$.
If $i > 0$ and $j < w_i$      receive $d$ from $(i-1, j)$,
                              if $(i < m-1)$ send $d$ to
                                $(i+1, j)$,
                              if $(j \leqslant c - w_i)$ send $d + p_i$ to
                                $(i, j + w_i)$.
If $i > 0$ and $j \geqslant w_i$   receive $d_1$ from $(i-1, j)$ and
                                $d_2$ from $(i, j - w_i)$,
                              calculate $d = \max\{d_1, d_2\}$,
                              if $(i < m-1)$ send $d$ to
                                $(i+1, j)$,
                              if $(j \leqslant c - w_i)$ send $d + p_i$ to
                                $(i, j + w_i)$.
```

In addition to the value of the final profit, we have to obtain the full description of the knapsack. For that purpose, we propose three solutions. Two solutions consist in sending a data array along with each transmitted data, and the third one uses backtracking. In order to make the solutions description uniform, we suppose that a task that has only one predecessor calculates the maximum between $-\infty$ and the received data.

bit array In this solution, task (i,j) sends, with the value to be transmitted, a bit array and an index. Tasks $(0,j)$, $j < w_0$, send index 0 with their data. If task (i,j) receives its maximum from its upper task $(i-1,j)$, it sets to 0 the position index of the array, otherwise it sets it to 1. Then, the task increments the index, and sends it along with the bit array to the next tasks.

This array of bits has at most $\left\lfloor \frac{c}{w_{\min}} \right\rfloor + m$ elements, and in this array there are $m-1$ 0's. We can easily construct the knapsack instance by visiting this array from the beginning, counting the number of 1's before each 0, this number corresponds to the number of items a_i, where i corresponds to the number of 0's already visited.

integer array Each task updates an array of m elements containing a partial solution. Tasks $(0, j)$, $j < w_0$ send an array of 0's with their data. If a task (i, j) chooses the maximum from the left task $(i, j - w_i)$, it updates the number of items of type a_i.

backtracking In [18], Hu suggested a backtracking technique with $\Theta(c)$ memory. The idea is to compute the value $F(k, g)$ which corresponds to the maximum index of the object used to obtain $f(k, g)$. Basically, the boundary conditions are: $F(1, i) = 0$, if $f(1, i) = 0$ and $F(1, i) = 1$, if $f(1, i) \neq 0$. In general,

$$F(k, g) = \begin{cases} F(k-1, g) & \text{if } f(k-1, g) \\ & \quad > f(k, g - w_k) + p_k, \\ k & \text{otherwise.} \end{cases}$$

Using this function $F(k, g)$, we can easily reconstruct the instance that gives the final profit by visiting the nodes which have computed the tasks $F(m, g), 0 \leqslant g \leqslant c$. Starting at $F(m, c) = i$, which means that object a_i is on the solution, continue on $F(m, c - w_i)$, and so on. Thus only $O(c)$ nodes have to be visited to construct an optimal solution instance.

2.3. 0/1 *Knapsack problem*

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

We can also 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 integers $0 \leqslant k < m$ and $0 \leqslant g \leqslant c$ is

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

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

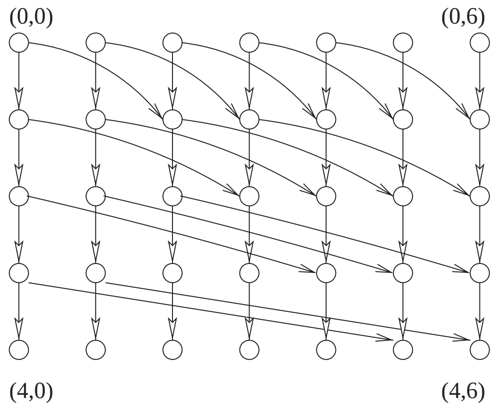

Fig. 2. Precedence graph of 0/1 KNAP(A,6) with $w_{i=0}^{3} = (2, 3, 4, 5)$.

The description of task (i, j) is the following:

If $i = 0$ and $m \geqslant 1$ send 0 to $(i+1, j)$,
if $(j \leqslant c - w_0)$ send p_0 to $(i+1, j+w_0)$.
If $i > 0$ and $j < w_{i-1}$ receive d from $(i-1, j)$,
if $(i < m-1)$ send d to $(i+1, j)$,
if $(j \leqslant c - w_i)$ send $d + p_i$ to $(i+1, j+w_i)$.
If $i > 0$ and $j \geqslant w_{i-1}$ receive d_1 from $(i-1, j)$,
receive d_2 from $(i-1, j - w_{i-1})$,
calculate $d = \max\{d_1, d_2\}$,
if $(i < m-1)$ send d to $(i+1, j)$
if $(j \leqslant c - w_i)$ send $d + p_i$ to $(i+1, j+w_i)$.

3. Related work

In this paper, we are interested in computing only the exact solutions for the Knapsack problem on a parallel machine with p processors, but we also introduce some of the recent work done in sequential algorithms. In this section, we first introduce some fundamental properties of the Knapsack problem, then we present the main algorithms proposed to solve the problem in parallel, and finally we describe the parallel algorithms to solve the Knapsack problem on hypercubes. As we present solutions for both Knapsack and 0/1 Knapsack problems, among with each work we cite the problem solved.

There are two main properties that are very useful to reduce the search space when solving the Knapsack problem: dominance and periodicity [18]. For both these properties we have to focus on the ratio p_i/w_i for each object. For the dominance the main idea is to find objects that may never occur on the optimal solutions. For example, if there are three objects a_i, a_j and a_k such that $w_i + w_j = w_k$ and $p_i + p_j \geqslant p_k$, there exists an optimal solution without objects a_k. There are several, including some very recent, work on elaborated dominance relations [4,28]. The periodicity states that beyond a certain capacity only the object a_j with the bigger ratio ($p_j/w_j = \max_i p_i/w_i$) contributes to the optimal solution. That is, for g sufficiently large [18], for the function $f(k, g)$ (defined in Section 2.1) we have $f(m, g) = f(m, g - w_j) + p_j$. These two properties can reduce significantly the solution search space. However, there still exist hard to solve instances like SSP [23] and the realistic random sets [4].

Concerning the 0/1 Knapsack problem there are two main approaches: Branch and Bound [23] and dynamic programming [26]. As pointed out before, the irregular mesh precedence graph can be used to solve this problem, however properties as dominance and periodicity can not be used.

3.1. *Solving the Knapsack problem in parallel*

There are several approaches for solving the Knapsack problems in parallel [15], but only a few have been specifically designed for hypercubes. Among the main approaches for the 0/1 Knapsack resolution is the straight-forward method consisting of the enumeration of all possible subsets [14]. Some other methods are based on variations of the two-list algorithm. They correspond to dividing the original problem, enumerating, and recombining the solutions [12,17,19].

The other solutions are more dedicated to some specific computational models or topologies. We present below the most important approaches:

Systolic: Some authors have studied the systolic model. In [5], 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 [2], the authors used a one-dimensional systolic array to solve Knapsack-like problems. In [1,3] the authors solved the Knapsack problem on a two-dimensional orthogonal systolic array. In [6] there is a more recent state of the art. As the hypercube is a Hamiltonian graph, such results on linear arrays can be easily adapted. The main results are summarized in Table 1.[2] In this table α refers to the memory size of the processing elements.

Fine Granularity: In [8], Chen et al. propose 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 [9] the previous work is extended using a technique called delayed dominance. Table 2 summarizes these results.

[2] In this table and in the following ones, the conditions column refers to the conditions to obtain cost optimality.

Table 1
Knapsack problems on systolic arrays

	Topology	Knapsack	Conditions
Andonov and Quinton [5]	Ring	$O\left(\frac{mc}{p} + m\right)$	$p \leqslant m$
Andonov and Rajopadhye [6]	Ring	$O\left(\frac{mc}{p}\left\lceil\frac{w_{\max}+w_{\min}}{\alpha}\right\rceil\right)$	$p \leqslant m$
Andonov and Gruau [3]	Torus $(q \times p)$	$O\left(\frac{mc}{pq}\right)$	$p \leqslant m$, $q \leqslant \frac{w_{\max}}{w_{\max}-w_{\min}+1}$

Table 2
Knapsack problems on fine grain architectures

	Topology	0/1 KP	Knapsack	Conditions
Chen et al. [8]	Ring	$O\left(\frac{mc}{p} + m\right)$	$O\left(\frac{mc}{p} + m\right)$	$p \leqslant m$
Chen and Jang [9]	Ring	$O\left(\frac{mc}{p} + m\right)$		$p \leqslant m$

Pipeline parallelism: In [24], Morales et al. proposed general parallel dynamic programming algorithms for pipeline and ring networks. For the Knapsack problem the basic idea is to partition the irregular mesh columns among the processors. On one side this kind of partition fits for linear and ring networks. However, on the other side this algorithms can be easily adapted to coarse grain computers.

PRAM: Teng [27] presents several algorithms for the Knapsack problem on a PRAM, using techniques such as reduction to some well-known problems (circuit value problem and prefix convolution problem). In all of Teng's solutions, some preprocessing computations requiring $O(\log(mc))$ units of time are needed, where m is the number of object types in the problem instance and c is the knapsack capacity. All the solutions proposed by Teng require at least c processors.

3.2. *Solving the Knapsack problem on hypercubes*

There exist two well-known hypercube algorithms to solve the 0/1 integral Knapsack problem.

In [20], 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 types of objects and a knapsack capacity c on p processors, the time is $O(\frac{mc}{p} + c^2)$. This algorithms was designed specifically for hypercubes.

Lin and Storer [22] propose an efficient algorithm that solves the 0/1 Knapsack problem on a PRAM within time $O(\frac{mc}{p})$ (where $p \leqslant c$). The oblivious implementation on hypercube has time complexity $O(\frac{mc}{p} \log p)$ and efficiency $O(\frac{1}{\log p})$.

One can easily verify that the Lin and Storer's algorithm can be seen as the execution of the precedence graph of the 0/1 Knapsack problem (cf. Fig. 2) on a PRAM.

Table 3
Complexity table for solving the Knapsack problems on hypercubes

	0/1 KP	Knapsack	Conditions
Lee et al. [20]	$O\left(\frac{mc}{p}+c^2\right)$		$p \leqslant m$
Lin and Storer [22]	$O\left(\frac{mc}{p}\log p\right)$		$c=\Omega(p\log p)$
Goldman and Trystram [16]	$O\left(\frac{mc}{p}\frac{w_{\max}}{w_{\min}}\right)$	$O\left(\frac{mc}{p}\frac{w_{\max}}{w_{\min}}\right)$	$p<\frac{c}{\log w_{\max}}$
This work	$O\left(\frac{mc}{p}+\frac{c}{w_{\min}}\right)$	$O\left(\frac{mc}{p}+\frac{c}{w_{\min}}\right)$	$p<\frac{c}{\log w_{\min}}$

Given $p(\leqslant c)$ processors, the graph is partitioned into sets of $\left\lceil \frac{c}{p} \right\rceil$ columns, each one being allocated to a different processor.

3.3. Our contribution

In this paper, we solve the integral Knapsack problem by giving an irregular mesh scheduling on the hypercube. As pointed out before, this schedule can also be used to solve the 0/1 Knapsack problem. Using the dynamic programming approach to solve the 0/1 Knapsack problem, the sequential algorithm requires a time in $\Theta(mc)$. So, the first hypercube solution has an additive factor of c^2 [20], the second one has a multiplicative factor of $\log p$ [22].

On [16] a specific algorithm for solving the Knapsack problem on the hypercube topology is proposed, it has a constant multiplicative factor of $\frac{w_{\max}}{w_{\min}}$. The algorithm that we present in this paper solves the Knapsack problem with a given number of processors (denoted by p) within $\Theta(\frac{mc}{p}+\frac{c}{w_{\min}})$ $(p<c/\log_2 w_{\min})$. We summarize all these results in Table 3.

3.4. Model

Like in a PRAM or systolic computation, we consider an algorithm as a succession of elementary steps. In each step the processors can perform some elementary mathematical operations. At the end of each step, there are communications among adjacent processors. As each processor has more than one direct neighbor, it can communicate simultaneously with all of them.[3] This model is very close to the behavior of SIMD computers. The adopted model is similar to the model used on related previous works [22]. On this work each hypercube processor may communicate small messages with all their neighbors in one step. On [20] the authors computed in separate, the computation and communication time, in our work for each communication step there is a computation step so this separation does not make sense.

In our analysis, as the tasks of the irregular mesh have the same computation time, we assume a UET (unit execution time) model.

3.5. Lower bound

It is also important to determine the lower bound with an unbounded number of processors. 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 longest path length of $NM(m,c)$ with jumps $(w_i)_0^{m-1}$ is at most $m+\left\lfloor \frac{c}{w_{\min}} \right\rfloor -1$.*

Proof. There are m rows in $NM(m,c)$ with jumps w_i on row i $(0\leqslant i<m)$. The length of the disjoint paths in each row i is $\left\lfloor \frac{c}{w_i} \right\rfloor$ or $\left\lfloor \frac{c}{w_i} \right\rfloor -1$, and the length of the longest path among all the rows is $\left\lfloor \frac{c}{w_{\min}} \right\rfloor$. Hence, the longest path of $NM(m,c)$ has $m+\left\lfloor \frac{c}{w_{\min}} \right\rfloor -1$ arcs. □

Given an irregular mesh $NM(m,c)$ (or a precedence graph for one 0/1 Knapsack problem) at most $c+1$ tasks can be executed in parallel, this is due to the precedence relations inside each column. This implies that it is not possible to improve the performance with more than $c+1$ processors.

4. Algorithm

In this section we first present an irregular mesh schedule, then we illustrate this schedule by an example and we prove its correctness. At the end we analyze the algorithm complexity.

4.1. Principle of the algorithm

On our previous work [16] an algorithm for scheduling irregular meshes on hypercube was presented. The allocation scheme used smaller hypercubes with $2^{\lceil \lg w_{\max} \rceil}$ vertices.[4] Each irregular mesh row was then scheduled on these smaller hypercubes. As on the irregular mesh row L_i there are at most w_i independent tasks, it is clear that there are some idle processors, which appear during the execution of all rows with less than $w_{\max}$ chains. It is easy to remark that the efficiency will be better using fewer idle processors.

The idea to improve the efficiency is to consider the hypercube as a Cartesian product, namely $H(n)=H(\lceil \lg w_{\min} \rceil)\,\square H(n')$. For the algorithm description, we label the $2^{n'}$ hypercubes $H(\lceil \lg w_{\min} \rceil)$ as $H_0,\dots,H_{2^{n'}-1}$, in such a way that two consecutive indices are adjacent hypercubes.

[3] Actually, the proposed scheduling works on a more restricted model where at each step each processor can send/receive $O(1)$ unitary messages, but it can route $O(\log p)$ incoming unitary messages.

[4] From now on we use lg to denote $\log_2$.

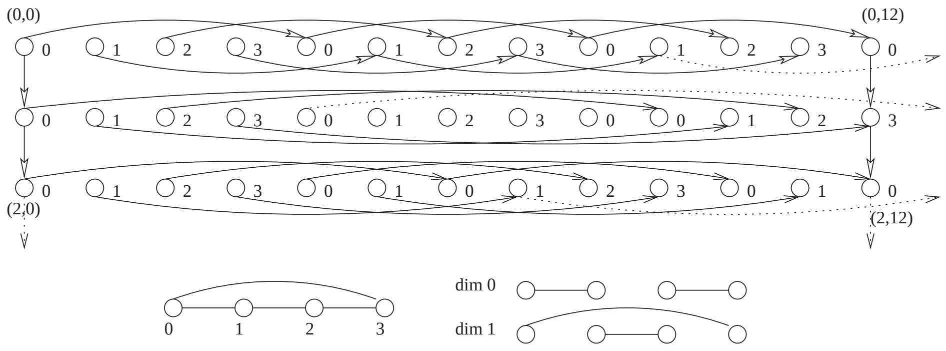

Fig. 3. Irregular mesh with weights $4, 9, 6, \ldots$ and $c \geqslant 13$ and its allocation on path P_k $(H(2)\Box H(0))$.

Let P be one of the $2^{\lceil \lg w_{\min} \rceil}$ Hamiltonian paths (generated from the reflected Gray Codes [21]) of $H(\lceil \lg w_{\min} \rceil)$. We use the Hamiltonian path P_k associated with each H_k $(0 \leqslant k < 2^{n'})$. In order to describe the allocation of the tasks of the irregular mesh, we assign labels from 0 to $w_{\min} - 1$ to the nodes of path P_k. Each row L_i $(0 \leqslant i < m)$ will be executed on path $P_{i \bmod 2^{n'}}$. As there is no change on the complexity calculation, in order to simplify the presentation it is supposed that $\lg w_{\min}$ is integer, that is $2^{n'} = p/w_{\min}$.

The allocation is such that the tasks from chain j of row L_i will be executed on the $(j \bmod w_{\min})$-processor of the Hamiltonian path $P_{i \bmod 2^{n'}}$. This allocation is depicted in Fig. 3 where beside each irregular mesh task we show its allocation.

4.2. The schedule

To determine the schedule we have to provide both the allocation and the execution time for each task of the irregular mesh. However, to obtain the right schedule we have to sort the knapsack objects in a special order. We define the following function for this purpose.

Definition 2. Given two weights $w_i, w_{\min}$, and $y_i = w_i \bmod w_{\min}$, the average idle time is defined by

$$I(w_i) = \begin{cases} 0 & \text{if } y_i = 0, \\ \dfrac{w_{\min} - y_i}{\left\lceil \frac{w_i}{w_{\min}} \right\rceil} & \text{if } y_i > 0. \end{cases}$$

We use the irregular mesh obtained by sorting the knapsack objects in nondecreasing order of average idle time. To simplify the presentation, let us introduce the value

$$M = \max_i \left\lfloor \frac{c}{w_i} \right\rfloor \left\lceil \frac{w_i}{w_{\min}} \right\rceil \tag{1}$$

and the number of paths $N = 2^{n'} = p/w_{\min}$. We will see shortly that M is the largest time among the rows execution time.

Without this special order, we could assume that the tasks should be executed as soon as all the necessary data are available. However, with this commonly used assumption for systolic models, the execution time for each task would depend on the object weights. We choose to process the objects in a special order to provide a simpler explicit timing function.

The allocation and execution time for the irregular mesh tasks are:

Allocation: Task (i, j) of the irregular mesh is allocated on path $P_{i \bmod N}$, on processor $P_{i \bmod N}^{(j \bmod w_i) \bmod w_{\min}}$.

Execution time: Task (i, j) is executed at time

$$i(\lg w_{\min} + 2) + \left\lfloor \frac{j}{w_i} \right\rfloor \left\lceil \frac{w_i}{w_{\min}} \right\rceil + \left\lfloor \frac{j \bmod w_i}{w_{\min}} \right\rfloor + \left\lfloor \frac{i}{N} \right\rfloor t_w,$$

where the value of t_w depends on the number of processors, and is given by

$$t_w = \begin{cases} 0 & \text{if } p \geqslant M \left\lceil \frac{w_{\min}}{\lg w_{\min} + 2} \right\rceil \\ M - (\lg w_{\min} + 2)N & \text{otherwise.} \end{cases}$$

We will also see shortly that t_w corresponds to the waiting time in order to avoid the execution of two different tasks simultaneously on the same processor.

4.3. Example

We detail in Fig. 4 the Gantt chart of the execution of the example introduced in Fig. 3.

4.4. Correctness of the algorithm

To verify the correctness of this schedule, we have to verify the following properties:

(1) Each task is executed by at least one processor.
(2) When processing a task, the necessary data from its predecessors is available.
(3) Each processor executes at most one task on each time step.

The first property is trivially respected. To verify the second property we present a routing scheme that guarantees the delivery of the data. For the routing between adjacent

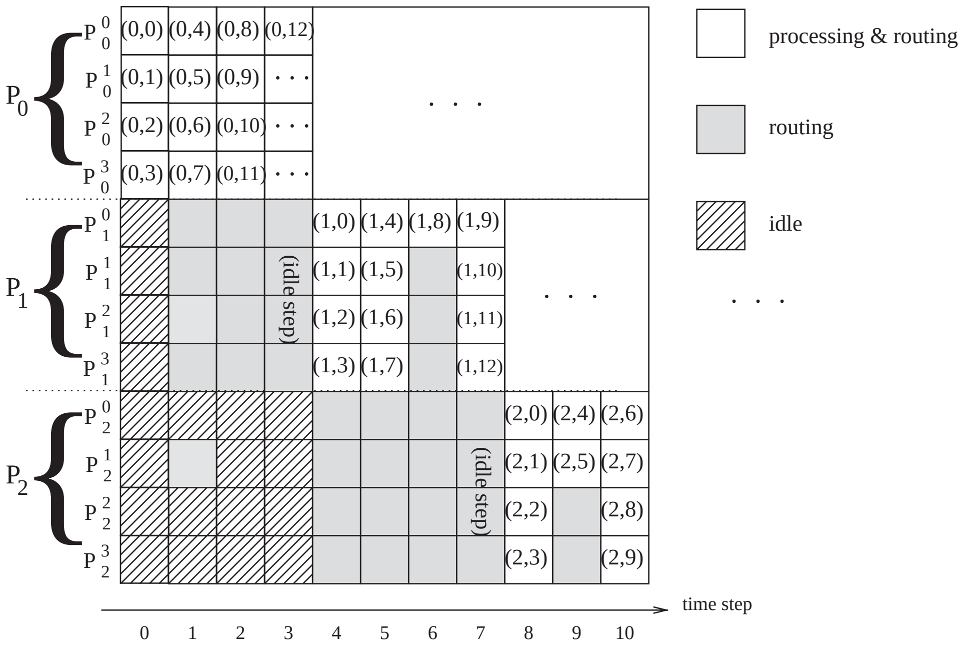

Fig. 4. Execution diagram of an irregular mesh with weights $4, 9, 6, \ldots$ and $c \geqslant 12$.

Hamiltonian paths we use the following principle: whenever a path has finished to process a set of tasks (from independent chains), it sends the results to the next path (according to the labeling $P_0, \ldots, P_N$). Each result has to be transmitted to the processor that will execute the task depending on this result. For instance, on the diagram of Fig. 4, task $(1, 9)$ needs the results from tasks $(1, 0)$ and $(0, 9)$. The first one is executed on the same processor P_1^0. The second one has to be routed from P_0^1 to P_1^0.

At each step, a path P_k receives up to $w_{\min}$ results to be routed, each result has to be delivered to one of its $w_{\min}$ processors. The idea for avoiding conflicts is to route by dimensions using a pipeline technique. The first set of results are first routed in dimension 0 of the hypercube corresponding to the path. In the next step, the second set of results are routed in dimension 0, while the first set of results continue to be routed in dimension 1, and so on.

After receiving results for y steps, the yth set of results is routed on dimension 0 and the $(y - x)$-th set of results (if $x \leqslant \lg w_{\min}$) is routed on dimension x. This ensures that results will reach their destination processor within at most $\lg w_{\min} + 1$ steps.

However, we have to consider also the flow of results. So, we will focus the attention on the execution inside a path P_k. For this purpose we suppose that all the results computed in the previous path are ready when needed.

Row L_i starts its execution at time t_i on path P_k. Let $x_i = \left\lfloor \frac{w_i}{w_{\min}} \right\rfloor$ and $y_i = w_i \bmod w_{\min}$. By the given schedule, at the beginning (time t_i), $w_{\min}$ tasks are executed. In the $x_i - 1$ subsequent steps $w_{\min}$ new tasks are executed. Then, if $y_i > 0$, only y_i tasks will be executed at time $t_i + x_i$. This scheduling is executed until the completion of all the tasks

Table 4
Execution scheme

time	t_i	...	$t_i + x_i - 1$	$t_i + x_i$	$t_i + x_i + 1$	...
# tasks	$w_{\min}$	...	$w_{\min}$	y_i	$w_{\min}$	...
	$\longleftarrow$		w_i tasks	$\longrightarrow$	$\longleftarrow w_i$ tasks $\longrightarrow$	

(see Table 4). Hence, the total time for executing all the tasks of row L_i is $\left\lfloor \frac{c}{w_i} \right\rfloor \left\lceil \frac{w_i}{w_{\min}} \right\rceil$ which is bounded by M.

When $y_i > 0$, the flow of $w_{\min}$ results on each step is periodically interrupted with the routing of only y_i results. Considering these interruptions it can happen that the precedence results will not be available just after the routing. We have to consider also the number of results routed. For instance on the diagram of Fig. 4, the third execution step of row L_2 (with execution flow $4, 2, 4, 2, \ldots$) can not start just $\log_2 w_{\min} + 1$ steps after the third execution step of row L_1 (with execution flow $4, 4, 1, 4, 4, 1, \ldots$), the result of $(1, 9)$ is available only one step later. Just after the third execution step of row L_1 there are 9 processed tasks, and for the same execution step of row L_2 there will be 10 executed tasks.

We focus now in the relationship between two adjacent paths. Using the fact that the objects are in a nondecreasing order in the average idle time, and given two adjacent rows (L_i, L_{i+1}) of an irregular mesh, we can state the following property:

Property 1. *The execution of row L_i may cause an idle step, due to a data flow interruption, on the execution of row L_{i+1} only due to the execution of $y_i > 0$ tasks.*

Proof. If $y_i = 0$, row L_i does not cause an idle step on the execution of row L_{i+1}. Otherwise, on the execution of row L_i, the flow of $w_{\min}$ tasks by step (see Table 4) is only interrupted after the execution of $y_i < w_{\min}$ tasks. $\square$

We can also state the following proposition:

Proposition 2. *Given two adjacent rows* (L_i, L_{i+1}) *of an irregular mesh there is at most one idle step on the execution of row* L_{i+1} *due to the data precedence.*

Proof. By property 1, the existence of an idle step implies that $y_i > 0$. When executing row L_i, at each time that y_i tasks are executed, there are $w_{\min} - y_i$ idle processors. Suppose that the first idle step, due to the data flow, occurs at time step x, so we have the following formula:

$$\left\lfloor \frac{x}{\left\lceil \frac{w_i}{w_{\min}} \right\rceil} \right\rfloor (w_{\min} - y_i) > \left\lfloor \frac{x}{\left\lceil \frac{w_{i+1}}{w_{\min}} \right\rceil} \right\rfloor (w_{\min} - y_{i+1}),$$

where the left side counts the number idle processors while processing row L_i, and similarly on the right side for row L_{i+1}. That is, there were more idle processors while processing L_1 for the x-time rather than on processing L_2 for the x-time.

But, by property 1, the idle steps may occur only when x is a multiple of $\left\lceil \frac{w_i}{w_{\min}} \right\rceil$, so we can restrict the analysis to the case

$$xI(w_i) > \left\lfloor \frac{x}{\left\lceil \frac{w_{i+1}}{w_{\min}} \right\rceil} \right\rfloor (w_{\min} - y_{i+1}). \tag{2}$$

Let us suppose that there are more than one idle step, due to the execution of row L_i. If we consider the first idle step on the execution of row L_{i+1} on Eq. (2), which means that its execution starts one step after the routing, we have to find x that satisfies:

$$xI(w_i) > \left\lfloor \frac{x}{\left\lceil \frac{w_{i+1}}{w_{\min}} \right\rceil} \right\rfloor (w_{\min} - y_{i+1}) + w_{\min},$$

which implies

$$xI(w_i) > xI(w_{i+1}) + y_{i+1},$$

but as $y_{i+1} \geqslant 0$ and $I(w_i) < I(w_{i+1})$, such x must be negative. So there is at most one idle step on the execution of row L_{i+1}. $\square$

This last proposition ensures that all the results from predecessor tasks will be available within at most $\lg w_{\min} + 2$ steps.

The third property is easily verified observing the value of t_w which is the difference between M, an upper bound on the rows processing time, and the period within the first path P_0 is used by different irregular mesh rows. If this difference is smaller than or equal to zero, the first path is always available to the rows allocated to it. The same is valid for all other paths as their allocation period $((\log_2 w_{\min} + 2)N)$ is the same. On the other hand, when the difference is greater than zero, the first path may be still processing a row L_i when the row L_{i+N} is ready to be executed, so the delay t_w is introduced. As this delay is introduced for P_0 it is not necessary to introduce this delay for the other paths. This way the third property is verified. So, the previous schedule is correct.

4.5. Complexity analysis

The algorithm analysis is divided in two cases, on the first we present the makespan when there is no waiting time ($t_w = 0$). Then, we will compute the makespan with waiting time.

- When $p \geqslant M \left\lceil \frac{w_{\min}}{\lg w_{\min}+2} \right\rceil$, each row L_i starts its execution at time $i(\lg w_{\min} + 2)$. That is, whenever a row is ready to execute, it can start immediately. Let us observe that the last irregular mesh row starts its execution at time step $(m-1)(\lg w_{\min} + 2)$, and its execution takes time $\left\lfloor \frac{c}{w_{m-1}} \right\rfloor \left\lceil \frac{w_{m-1}}{w_{\min}} \right\rceil$. So, the makespan is at most
$$(m-1)(\lg w_{\min} + 2) + M. \tag{3}$$
- When $p < M \left\lceil \frac{w_{\min}}{\lg w_{\min}+1} \right\rceil$, there exists a waiting time before starting the execution of each row $L_i, i \mod N = 0, i > 0$. The total time is at most the previous one (Eq. (3)) plus $\left\lfloor \frac{m}{N} \right\rfloor$ times the waiting time $t_w = M - (\lg w_{\min} + 2)N$, that is at most:
$$M\left(\frac{mw_{\min}}{p} + 1\right). \tag{4}$$

Observe that M (Eq. (1)) belongs to $\Theta\left(\frac{c}{w_{\min}}\right)$. When there are processors available to execute each row without waiting time, we have the following proposition:

Proposition 3. *The total processing time of an irregular mesh on a hypercube is given by Eq.* (3), *that is* $\Theta\left(m \log w_{\min} + \frac{c}{w_{\min}}\right)$ *with at least* $\frac{c}{\log w_{\min}}$ *processors.*

With fewer processors, there is waiting time before starting the execution of row $L_i, i \mod N = 0, i > 0$, and the total time is given by Eq. (4). In this case:

Proposition 4. *The total processing time of an irregular mesh on hypercube is* $\Theta\left(\frac{mc}{p} + \frac{c}{w_{\min}}\right)$ *using p processors, p between* $2^{\lceil \lg w_{\min} \rceil}$ *and* $\frac{c}{\log w_{\min}}$.

As a corollary we can state a more precise result: If enough processors are available ($p > mw_{\min}$) then the complexity is dominated by $O\left(\frac{c}{w_{\min}}\right)$ which corresponds to the lower bound of Proposition 1 (for large c). With less processors the complexity is dominated by $O\left(\frac{mc}{p}\right)$ (which corresponds to the sequential time divided by the number of processors).

5. Concluding remarks

We have presented in this paper an efficient algorithm for solving the Knapsack problem on the hypercube. The complexity analysis were done within a synchronous fine grain computational model. The proposed algorithm solves the Knapsack and $0/1$ Knapsack problems with $\frac{c}{\log w_{\min}}$ processors in $\Theta\left(m \log w_{\min} + \frac{c}{w_{\min}}\right)$ steps. Which corresponds to the introduction of a multiplicative factor of $\log w_{\min}$ on the smallest processing time ($O\left(m + \frac{c}{w_{\min}}\right)$). With fewer processors, the proposed algorithm solves these problems within $\Theta\left(\frac{mc}{p} + \frac{c}{w_{\min}}\right)$ steps.

Acknowledgments

We would like to thank the anonymous reviewers for the multiple and always fruitful exchanges we had during the evaluation process, and the managing editor for his patience.

References

[1] V. Aleksandrov, S. Fidanova, Non-uniform recurrence equations on 2d regular arrays, in: Advance in Numerical Methods and Applications, in Proceedings of NMA94, August 1994, pp. 217–225.
[2] V. Aleksandrov, S. Fidanova, On the expected execution time for a class of non uniform recurrence equations mapped onto 1d array, Parallel Algorithms Appl. 1 (1994) 303–314.
[3] R. Andonov, F. Gruau, A 2D modular toroidal systolic array for the Knapsack problem, in: ASAP'91, 1991, pp. 458–472.
[4] R. Andonov, V. Poirriez, S. Rajopadhye, Unbounded Knapsack problem: dynamic programming revisited, European J. Oper. Res. 123 (2000) 394–407.
[5] R. Andonov, P. Quinton, Efficient linear systolic array for Knapsack problem, CONPAR'92, Lecture Notes in Computer Science, vol. 634, Springer, Berlin, 1992, pp. 247–258.
[6] R. Andonov, S. Rajopadhye, Knapsack on VLSI: from algorithm to optimal circuits, IEEE Trans. Parallel Distributed Systems 8 (6) (1997) 545–562.
[7] E. Bampis, J-C. König, C. Delorme, Optimal schedule for d-d grid graphs with communication delays, Parallel Computing 24 (11) (1998) 1653–1664.
[8] G-H. Chen, M-S. Chern, J-H. Jang, Pipeline architectures for dynamic programming algorithms, Parallel Comput. 13 (1) (1990) 111–117.
[9] G-H. Chen, J-H. Jang, An improved parallel algorithm for $0/1$ knapsack problem, Parallel Comput. 18 (7) (1992) 811–821.
[10] E.G. Coffman, P.J. Denning, Operating Systems Theory, Prentice-Hall Series in Automatic Computation, 1973.
[11] E.G. Coffman, M.R. Garey, D.S. Johnson, A.S. Lapaugh, Scheduling file transfers, SIAM J. Comput. 14 (3) (August 1985) 744–780.
[12] M. Cosnard, A. Ferreira, H. Herbelin, The two list algorithm for the Knapsack problem on a FPS T20, Parallel Comput. 3 (9) (1988/1989) 385–388.
[13] M.R. Garey, D.S. Johnson, Computers and Intractability: A Guide to the Theory of NP-completeness, W.H. Freeman and Company, San Francisco, 1979.
[14] T. Gerasch, P. Wang, Implementing dynamic programming on the connection machine, Technical Report Series TR-10-89, George Mason University, 1989.
[15] T. Gerasch, P. Wang, A survey of parallel algorithm for one-dimensional integer Knapsack problems, Infor 32 (3) (1993) 163–186.
[16] A. Goldman, D. Trystram, An efficient parallel algorithm for solving the Knapsack problem on hypercube, in: 11th International Parallel Processing Symposium, April 1997, pp. 608–615.
[17] E. Horowitz, S. Sahni, Computing partitions with applications to the Knapsack problem, J. ACM 21 (1974) 277–292.
[18] T.C. Hu, Combinatorial algorithms, Addison-Wesley, Reading, MA, 1982.
[19] E. Karnin, A parallel algorithm for the Knapsack problem, IEEE Trans. Comput. 33 (5) (1984) 404–408.
[20] J. Lee, E. Shragowitz, S. Sahni, A hypercube algorithm for the $0/1$ Knapsack problem, J. Parallel Distributed Comput. 5 (4) (1988) 438–456.
[21] F. Leighton, Parallel Algorithms and Architectures: Arrays—Trees—Hypercubes, Morgan Kaufmann, Los Altos, CA, 1992.
[22] J. Lin, J. Storer, Processor efficient hypercube algorithm for the Knapsack problem, J. Parallel Distributed Comput. 13 (3) (1991) 332–337.
[23] S. Martello, P. Toth, Knapsack Problems: Algorithms and Computer Implementations, Wiley, New York, London, Sydney, 1990.
[24] D.G. Morales, F. Almeida, C. Rodríguez, J.L. Roda, I. Coloma, A. Delgado, Parallel dynamic programming and automata theory, Parallel Comput. 26 (2000) 113–134.
[25] C. Picouleau, New complexity results on scheduling with small communication delays, J. Discrete Appl. Math. 60 (1995) 331–342.
[26] D. Pisinger, A minimal algorithm for the 0-1 Knapsack problem, Oper. Res. 45 (5) (September–October 1997) 758–767.
[27] S. Teng, Adaptive parallel algorithms for integral Knapsack problems, J. Parallel Distributed Comput. 8 (1990) 400–406.
[28] N. Zhu, K. Broughan, On dominated terms in the general Knapsack problem, Oper. Res. Lett. 21 (1997) 31–37.

Alfredo Goldman received his B.Sc. in applied mathematics and his M.Sc. in computer science from University of São Paulo, Brazil. He received his doctorate in computer science from the Institut National Polytechnique de Grenoble, France. He is an assistant professor in the Department of Computer Science at University of São Paulo. His research interests include parallel and distributed computing, mobile computing, and grid computing.

Denis Trystram obtained his M.Sc. in computer science in Grenoble in 1982. He received a Ph.D. in applied mathematics in 1984 and a Ph.D. in computer science in 1988, both in Grenoble at INPG. He has been a professor since 1991. In 1999, he moved to ID-IMAG where he is leading the group "02" (Scheduling and Combinatorial Optimization). He is currently Regional Editor for Europe for the *Parallel Computing Journal*. Professor Trystram was co-organizer of the two major European conferences in the field of parallel and distributed computing (CONPAR and PARCO), and he participates regularly in the program committees of major conferences (PARCO, EuroPar, IPDPS, SPAA, etc.). The research activities of professor Trystram concern all aspects of the study of the impact of parallelism on the conception of efficient algorithms. His main interest is on scheduling and the design of efficient approximation algorithms with a special focus on clusters and heterogeneous networks (grid and global computing). Professor Trystram has published five books and edited four books, including the "Handbook of Parallel and Distributed Processing" published by Springer Verlag in 2000. He has published more than 60 articles in international journals and as many international conferences.