INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Dynamic Programming Parallel Implementations for the Knapsack Problem

Rumen Andonov , Frédéric Raimbault and Patrice Quinton

N° 2037

Septembre 1993

PROGRAMME 1

Architectures parallèles,
bases de données,
réseaux et systèmes distribués

Rapport de recherche

1993

Dynamic Programming Parallel Implementations for the Knapsack Problem

Rumen Andonov *, Frédéric Raimbault and Patrice Quinton **

Programme 1 — Architectures parallèles, bases de données, réseaux et systèmes distribués
Projet API

Rapport de recherche n° 2037 — Septembre 1993 — 15 pages

Abstract: A systolic algorithm for the dynamic programming approach to the knapsack problem is presented. The algorithm can run on any number of processors and has optimal time speedup and processor efficiency. The running time of the algorithm is $\Theta(mc/q + m)$ on a ring of q processors, where c is the knapsack size and m is the number of object types. A new procedure for the backtracking phase of the algorithm with a time complexity $\Theta(m)$ is also proposed which is an improvement on the usual strategies used for backtracking with a time complexity $\Theta(m + c)$. Given a practical implementation, our analysis shows which of two backtracking algorithms (the classical or the modified) is more efficient with respect to the total running time. Experiments have been performed on an iWARP machine for randomly generated instances. They support the theoretical results and show the proposed algorithm performs well for a wide range of problem size.

Key-words: integer programming, dynamic programming, partitionning, recurrences, parallelism, knapsack problem, systolic algorithm, iWarp machine

(Résumé : tsvp)

Soumis à la revue *Journal of Parallel and Distributed Computers*

*One leave from Center of Computer Science and Technology, Acad. G. Bonchev st., bl. 25-a, 1113 Sofia, Bulgaria
**[andonov,raimbaul,quinton]@irisa.fr

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex (France)
Téléphone : (33) 99 84 71 00 – Télécopie : (33) 99 38 38 32

Implémentations parallèles d'un algorithme de programmation dynamique pour le problème du sac à dos

Résumé : Cet article présente un algorithme systolique pour la programmation dynamique du problème du sac à dos. Cet algorithme est exécutable sur un nombre quelconque de processeurs et possède des propriétés d'accélération et d'efficacité optimales. Le temps d'exécution est en $\Theta(mc/q + m)$ sur un anneau de q processeurs, où c représente la taille du sac à dos et m le nombre d'objets. Une nouvelle méthode de calcul de complexité en temps de $\Theta(m)$ est également proposée pour la phase de "backtraking". Ce résultat constitue théoriquement une amélioration par rapport aux solutions antérieures dont la complexité atteignait $\Theta(mc)$. Nos analyses montrent dans quels cas notre algorithme de "backtraking" est plus efficace que le précédent quand la totalité du temps d'exécution est pris en compte. Nos expériences sont réalisées sur la machine iWarp avec des jeux de données générés aléatoirement. Ces expériences vérifient nos résultats théoriques et prouvent l'efficacité de notre algorithme pour des problèmes de taille quelconque.

Mots-clé : programmation linéaire, programmation dynamique, récurrences, partitionnement, problème du sac à dos, parall'elisme, algorithme systolique, machine iWarp

1 Introduction

Suppose that m types of objects are being considered for inclusion in a knapsack of capacity c. For $i = 1, 2, \ldots, m$, let p_i be the unit value and w_i the unit weight of the i-th type of object. The values w_i, p_i, $i = 1, 2, \ldots, m$, and c are all positive integers. The problem is to find the maximum total profit without exceeding the capacity constraint, i.e.

$$\max \{\sum_{i=1}^{m} p_i z_i : \sum_{i=1}^{m} w_i z_i \leq c, z_i \geq 0 \text{ integer}, \ i = 1, 2, \ldots, m\}, \tag{1}$$

where z_i is the number of i-th type objects included in the knapsack. Sometimes (1) is referred to as the integral knapsack problem (Teng [1]), sometimes as the unbounded knapsack problem (Martello and Toth [2]). We shall call it simply the knapsack problem. If additional constrains $z_i \in \{0, 1\}, i = 1, 2, \ldots, m$ are added to (1), then the restricted problem is called the 0/1 knapsack problem.

This classical combinatorial optimization problem has a wide range of application (see e.g. Garfinkel and Nemhauser [3], Hu [4], Martello and Toth [2]). Moreover this is the most elementary integer programming problem, in the sense that there is only one constraint. In principle, any integer programming problem can be transformed into this problem, see [3]. The difficulties which arise when solving problem (1) are typical for the whole area called integer programming. Effective algorithms for the knapsack problem are not only interesting for the problem itself but for this domain of research in general.

Problem (1) belongs to the class of NP-*complete* problems (see e.g. Garey and Johnson [5]). However it is known that this problem can be solved sequentially in $O(mc)$ time. This time bound is not polynomial in the size of the input since $\log_2 c$ bits are required to encode the input c. Such a time bound is called *pseudo-polynomial* time [5].

Two basic approaches are currently popular for finding the exact solution of the knapsack problem: *dynamic programming* (DP) *and branch-and-bound* (B&B).

Serial machine implementations: The first knapsack algorithm based on dynamic programming approach was developed by Gilmore and Gomory [6]. It takes $\Theta(mc)$ operations to solve the problem and is totally insensitive to the parameters $p_i, w_i, i = 1, 2, \ldots, m$. This is typical of dynamic programming serial implementations which can be good for poorly-behaved problems and bad for well-behaved ones, as noted in [7].

The dominant opinion today is that algorithms for large-size problems (1) based on B&B approach are more efficient on the average for a serial machine implementation. This is shown by the encouraging theoretical and experimental results obtained by numerous researchers (see the book of Martello and Toth [2] on related topic). However, one can raise questions about the randomly generated test problems used in such computational experiments. In general, when the parameters p_i and w_i are independently generated the problems tend to be easy. When these coefficients are correlated the problems seem to be more difficult. Hard knapsack problems have been constructed by researchers including Jeroslaw [8], Chvatal [9] and Chung, Hung and Rom [7]. The latter authors study a class of problems with constant difference between p_i and w_i which requires a branch and boundalgorithm to run exponentially in the problem size m. They prove also that the difficulty of the problem does not depend on the bounding mechanism used in B&B algorithm. Computational tests indicate that these problems are truly difficult for even a very small problem size. In such problems DP algorithms behave better than B&B. The difficult question is is in reality how often are the hard knapsack problems encountered. But in any case the available knowledge for problem (1) implies that both approaches - DP and B&B - are worthy of study for parallel implementations.

Parallel machine implementations: With the advent of parallel processors, many researchers concentrated their efforts on the development of efficient parallel algorithms for solving the knapsack and the 0/1 knapsack problems. As in the sequential case, dynamic programming [10, 11, 12, 1] and branch-and-bound [13, 14] are the most popular combinatorial optimization techniques for finding the exact solution to these problems. The results obtained show that either of these two approaches is suitable for parallelizing, but each in its own manner. Each one requires a different type of parallel machine in order to ensure a good trade-off between communication and computation, resulting in a good performance for the parallel algorithm.

The inherent parallelism in B&B method is suitable to implementation either on MIMD shared memory multiprocessor machines [1] see e.g. [15, 16, 17, 18, 19] or on MIMD distributed-memory coarse-grained multiprocessors [2] (see the paper of Dehne, Ferreira and Rau-Chaplin [20] for more references). Two major problems arise in this approach: (i) *Load balancing:* how to manage the sizes of the problems assigned to individual processors; the site may vary significantly, and this unbalanced distribution of work load may result in performance degradation; (ii) *Global information:* how to distribute the global information in order to avoid the additional computational overhead or the unnecessary search. It seems these two problems are rather difficult in the case of parallel B&B implementation, as shown by the numerous publications on this subject [19, 21, 13, 22].

Recently in [20], parallel B&B was studied on fine-grained hypercube multiprocessors (the Connection Machine). To the knowledge of the authors this approach has not yet been applied to problem (1).

As in the sequential case, the best time complexity parallel algorithm for (1) is based on the DP technique, as proposed by Teng in [1]. The number of processors required is exponential in the size of the input and this algorithm has a very low processor efficiency. More precisely it requires $M(c)$ processors to solve the problem in $O(log^2(mc))$ time. The function $M(n)$ above denotes the number of processors needed for multiplying two n by n matrices in $O(log\ n)$ parallel time. It is known that $n^2 \leq M(n) \leq n^3$. Therefore the knapsack capacity c is a factor in the processor complexity of the algorithm, and $1/c$ is a factor in its efficiency which approaches zero as c increases.

In this paper we concentrate on the dynamic programming approach with a fixed number of processing elements. A brief overview of the obtained results show that the research in this area has primarily been restricted to the 0/1 case [12, 23, 24, 10, 25]. As we show in section 4 the dependence graphs for the general and the 0/1 cases belong to two different families of graphs. This implies different data partitioning schemes.

To the knowledge of the authors, the proposed algorithm is the first implementation for the general case of problem (1) with asymptotically (with respect to c) linear speedup $\Theta(q)$. Moreover, this approach can be easily applied for the 0/1 case, preserving its efficiency. For the 0/1 case, the implementation proposed here is more efficient than the parallel implementation of Lin and Storer [12] which has time speedup of $\Theta(q/log\ q)$, on q processors on the Connection Machine.

We follow the direction of research of [26] where the main lines of our parallel approach have been briefly presented. Here we develop in details these ideas and study the constants appearing in the different parallel implementations. Furthermore in this research note we analyze the application of the dependence mapping approach to all types of knapsack problem recurrences. The problems of the implementation and performance of the algorithm on a real parallel machines are analysed and a large number of computational experiments are performed.

Our approach is close to that of Chen, Chern and Jang in [11], who propose a pipeline architecture containing a linear array of q processors, and queue and memory modules of size α for solving the knapsack problem. Their algorithm has a time complexity $\Theta(mc/q + m)$ and their architecture allows one to asymptotically achieve a linear speedup.

Using the same architecture and new data partitioning we propose an algorithm with $\alpha \geq w_{max}$ requirement for the size α of any memory module, where $w_{max} = \max\ _{i=1}^{m}\{w_i\}$. It is well known that in many practical applications w_{max} is much less than the knapsack capacity c, i.e., $w_{max} \ll c$. Therefore this is an improvement of the memory requirement of the algorithm in [11] which needs $\alpha \geq c$ memory for any processors. Another difference is that our approach is systolically oriented in contrast to [11] where a transputer implementation is given.

A new procedure for the backtracking phase of the algorithm with a time complexity $\Theta(m)$ is also proposed. It is an improvement on the usual strategies used for backtracking (see Hu [4] and Garfinkel and

[1] as Cray X_MP or Cray 2

[2] multiprocessors with a relatively small number of relatively powerful processors each having a considerable amount of memory such as the FPS hypercube, NCUBE, or Intel iPSC

Nemhauser [3]) which have a time complexity $\Theta(m + c)$. Thus this phase of the dynamic programming approach, which is sequential, becomes independent of the parameter c.

This paper is organized in the following way. Section 2 describes both phases of the dynamic programming approach for problem (1) on a serial machine. Section 3 presents our new algorithm for backtracking and analyses its complexity. In section 4 we discuss our parallel implementation on a linear array containing an unbounded number of processing elements (PE). Section 5 is devoted to the analysis of the constants in the classical and the modified backtracking algorithm. In section 6 we consider the case of a fixed number of PEs. In section 7 the computational experiments are presented.

2 Dynamic programming approach for the knapsack problem

In this section we present the dynamic programming approach for the knapsack problem on a serial machine. This approach is based on the *principle of optimality* of Bellman [27] and usually contains two phases. In the first (forward) phase, the maximum value of the objective function is computed, i.e. the value $f_m(c)$ such that

$$f_m(c) = \max \{\sum_{i=1}^{m} p_i z_i : \sum_{i=1}^{m} w_i z_i \leq c,\ z_i \geq 0 \text{ integer}, i = 1, 2, \ldots, m\}$$

.

In the second (backtracking) phase the integers $z_i^*, i = 1, 2, \ldots, m$, such that

$$\sum_{i=1}^{m} p_i z_i^* = f_m(c)$$

are found.

2.1 Forward phase

Let $f_k(j)$ be the maximum value that can be achieved in (1) from a knapsack of size j, $0 \leq j \leq c$, using only the first k types of objects, $1 \leq k \leq m$. That is

$$f_k(j) = \max \{\sum_{i=1}^{k} p_i z_i : \sum_{i=1}^{k} w_i z_i \leq j, z_i \geq 0 \text{ integer}, i = 1, 2, \ldots, k\}.$$

The principle of optimality [27, 3] states that for $\forall k, 1 \leq k \leq m$ and $\forall j, 0 \leq j \leq c$ we have :

$$f_k(j) = \max \{f_{k-1}(j), f_k(j - w_k) + p_k\}. \tag{2}$$

For the 0/1 knapsack problem, equation (2) can be rewritten as

$$f_k(j) = \max \{f_{k-1}(j), f_{k-1}(j - w_k) + p_k\}. \tag{3}$$

The optimal value of $f_m(c)$ (1), can be found in m stages by generating successively the functions $f_1, f_2, \ldots, f_m$ using equation (2) (or (3)) assuming the initial conditions $f_0(j) = 0, f_k(0) = 0$ and $f_k(i) = -\infty$ for $k = 1, 2, \ldots m, c = 1, 2, \ldots c$ and $i < 0$. By stage k we shall denote the computation of all the values of the function f_k.

Any serial algorithm which solves problem (1) using equation (2) requires $\Theta(mc)$ time. Currently this is the serial algorithm which has the best behavior when solving the worst case instances for the knapsack problem. We shall use its running time to determine the *speedup* and the *efficiency* of our parallel algorithm.

The communication required for the execution of equation (2) or (3) can be described by means of a directed graph, called the *dependence graph (DG)*. Let $\mathbf{N}$ denote the set of natural numbers, i.e. $\mathbf{N}=\{0, 1, 2, \ldots\}$. Let $\mathcal{G} = (\mathcal{D}, \mathcal{A})$ be the DG for equation (2) or (3), where $\mathcal{D} = \{(j, k) \in \mathbf{N}^2 : 0 \leq j \leq c, 1 \leq k \leq m, \}$ is

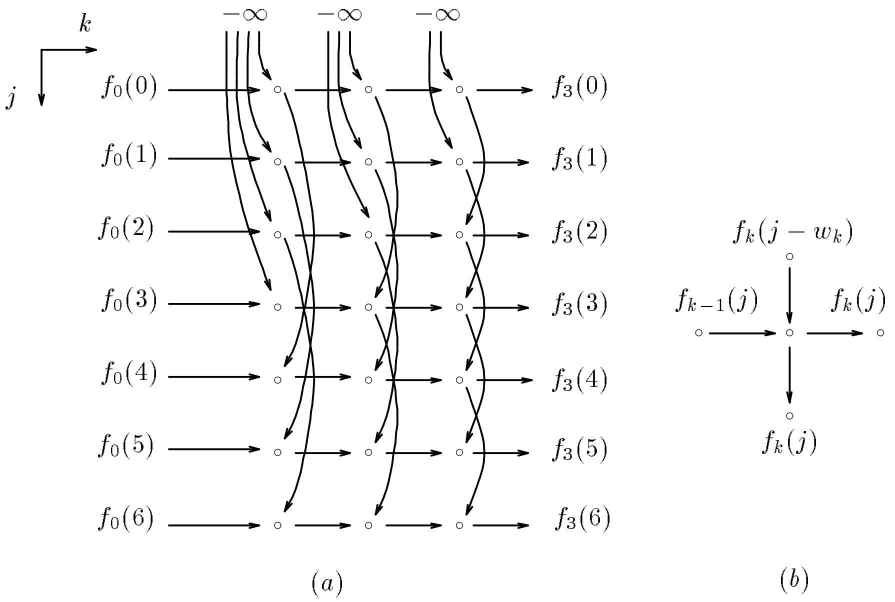

Figure 1: Dependence graph for equation (2)

the set of nodes and $\mathcal{A}$ is the set of directed arcs. Each node $(j,k) \in \mathcal{D}$ of the DG represents an operation performed by the algorithm and the arcs are used to represent data dependencies. For example figure 1 (a) depicts the DG for equation (2), with $m=3, c=6, w_1=4, w_2=3, w_3=2$. Each node (j,k) represents one calculation, detailed in figure 1 (b). Figure 2 depicts the corresponding DG for equation (3) with the same data.

The peculiarity of the graphs in figures 1 and 2 is that in any column the dependence vectors depend on the weights $w_i, i=1,2,\ldots,m$. Such a dependency is *non-affine* dependency. This peculiarity makes the knapsack recurrence equations difficult to transform into a systolic array using the well-known dependence mapping approach (see Quinton and Robert, [28] or S. Rajopadhye [29]).

2.2 The classical backtracking algorithm

An approach to find the solution vector of problem (1), i.e. a vector $z^* \in \mathbf{N}^m$ such that $\sum_{i=1}^{m} p_i z_i^* = f_m(c)$, is discussed in this section. It is based on the work of Hu [4].

In the course of the forward phase a pointer $u_k(j)$ is associated with each value $f_k(j), (j,k) \in \mathcal{D}$ in such a way that $u_k(j)$ is the index of the last type of object used in $f_k(j)$. In other words, $u_k(j)=r$ means that $z_r \geq 1$, or the r-th object is used in $f_k(j)$ and $z_l = 0$ for all $l > r$. The value $u_k(j)$ is used to keep a history of the first dynamic programming phase.

The boundary conditions for $u_k(j)$ are

$$\forall j : 0 \leq j \leq c : u_1(j) = \begin{cases} 0 & \text{if } f_1(j) = 0 \\ 1 & \text{if } f_1(j) \neq 0. \end{cases}$$

In general we set

$$u_k(j) = \begin{cases} k & \text{if } f_k(j-w_k) + p_k > f_{k-1}(j) \\ u_{k-1}(j) & \text{otherwise} \end{cases} \tag{4}$$

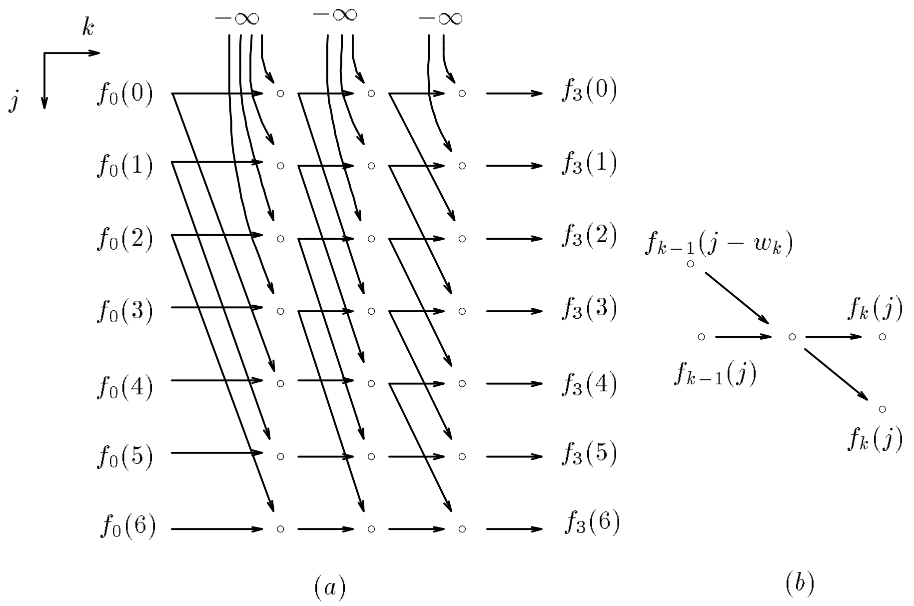

Figure 2: Dependence graph for equation (3)

$\forall k : 1 < k \leq m$ and $\forall j : 0 \leq j \leq c$.

As shown in [4], definition (4) allows the solution vector $z^* \in \mathbf{N}^m$ of the knapsack problem to be found from the values of the function u_m by the following algorithm:

Hu's backtracking algorithm

```
j := c;
for k = m downto 1 do begin
    z*_k = 0;
    while u_m(j) = k do begin
        z*_k := z*_k + 1;
        j := j - w_k;
    end{while}
end{for}
```

Corollary 1 *The previous algorithm has time and space complexity* $T = \Theta(m + c)$ *and* $S = \Theta(c + m)$ *respectively.*

Remark: In the 0/1 case the algorithm runs in $\Theta(m)$ time.

3 A modified backtracking algorithm

Let us associate a value p_k to any vertical arc $((i, k), (i + w_k, k))$ in column k of the DG for problem (1). Then $f_m(c)$ can be regarded as the value of the shortest path from point $(0, 1)$ to point (c, m) (see figure 3). Let us denote by S_k, the subpath of the optimal path in column k. The elements of S_k are of the form $((i, k), (i + w_k, k), \ldots, (i + w_k z_k^*, k))$, for some $0 \leq i \leq c$. Obviously

$$z_k^* = \mid S_k \mid -1, \tag{5}$$

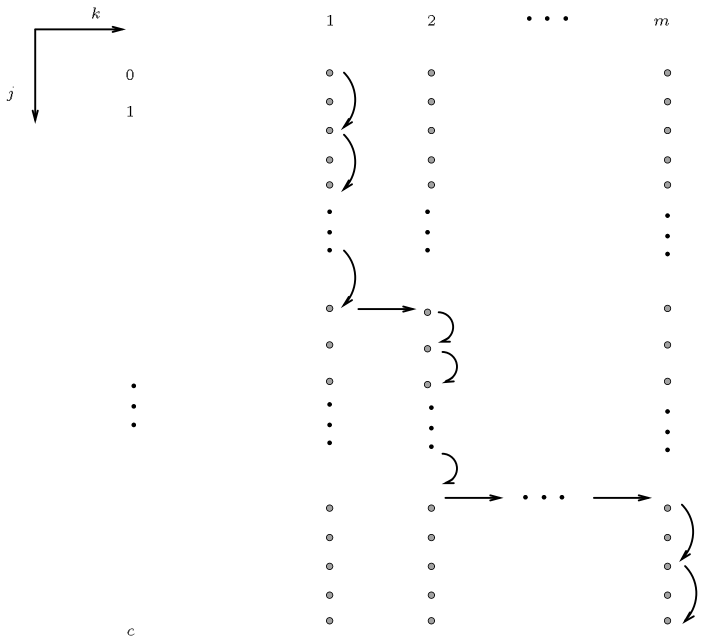

Figure 3: The shortest path from point (0,1) to point (c,m)

where $\mid S_k \mid$ denotes the cardinality of the set S_k. In fact (5) determines how many units of object k are used in the optimal solution. It is easily seen that Hu's backtracking algorithm takes $\mid S_k \mid$ steps in column k. Thus this algorithm requires $\sum_{k=1}^{m} \mid S_k \mid = \Theta(m+c)$ steps. The improvement is to compute for any k the cardinality $\mid S_k \mid$ in constant time and to thus decrease the total time for the backtracking phase to $\Theta(m)$.

Let I_k denote the sequence of the first indices of the elements of the optimal subpath S_k in column k, i.e.

$$I_k = \{i : (i,k) \in S_k\}.$$

Let i^k_{min} denote the minimum index in I_k, i.e.

$$i^k_{min} = \min\{i : i \in I_k\}.$$

By the definition of the function u_k we have

$$u_k(i^k_{min}) = u_{k-1}(i^k_{min}) < k$$

and

$$\forall i \in I_k : i \neq i^k_{min} \Rightarrow u_k(i) = k.$$

In the modified backtracking algorithm, for any $(j,k) \in \mathcal{D}$ we keep a record of the candidate for i^k_{min}. In order to generate these values during the forward phase a new function v_k is introduced:

$$v_k(j) = \begin{cases} v_{k-1}(j) & \text{if } u_k(j) < k \\ j & \text{if } u_k(j) = k \quad \text{and } u_k(j-w_k) < k \\ v_k(j-w_k) & \text{if } u_k(j) = k \quad \text{and } u_k(j-w_k) = k \end{cases} \tag{6}$$

for all k, and all j:$1 \leq k \leq m, 0 \leq j \leq c$ and $v_k(j) = 0$ for $k = 0, 0 \leq j \leq c$.

When the values of v_k is found, we can easily trace the value z_k that yields $f_k(j)$ by the formula

$$z_k = \begin{cases} 0 & \text{if } u_k(j) < k \\ (j - v_k(j))/w_k + 1 & \text{if } u_k(j) = k. \end{cases} \quad (7)$$

Once the value z_k has been computed, the total weight limitation j is reduced to $j_{new} = v_k(j) - w_k$, for which the inequality $u_k(j_{new}) < k$ holds and therefore $u_k(j_{new}) = u_{k-1}(j_{new})$. To find the value z_{k-1} we examine the values $u_{k-1}(j_{new})$ and $v_{k-1}(j_{new})$ in the same way. It is easily seen that formula (7) gives the same results as (5) when the backtracking process starts from point (c, m).

The functions $v_k, k = 1, 2, \ldots, m$ are similar to the functions u_k by their properties. For all j such that $u_k(j) < k$ the function v_k keeps the values of the functions v_{k-1}. This property allows the values $z_k^*, k = 1, 2, \ldots, m$ to be found from the values of the function u_m and v_m only. The following algorithm is used.

Modified backtracking algorithm

```
j := c;
for k = m downto 1 do
    if u_m(j) < k then z*_k = 0;
      else begin
          z*_k := (j − v_m(j))/w_k + 1;
          j := v_m(j) − w_k;
      end;{if}
```

In this way the values of u_m are used to move back along the k axis of the DG. The values of v_m are used to move back along the j axis. Since the computation of z_k^* for any $k, 1 \leq k \leq m$ requires $\Theta(1)$ operations we obtain the following property:

Corollary 2 *The modified backtracking algorithm has time and space complexity $T = \Theta(m)$ and $S = \Theta(c + m)$ respectively.*

The computation of the functions $v_k, k = 1, 2, \ldots, m$ can be done simultaneously with the computation of the functions f_k. Comparing (6) and (4) we see that the computation of the functions v_k does not influence the time complexity of the forward phase (i.e. the total time of the algorithm is still $\Theta(mc)$). The advantage of the modification is to make the backtracking phase independent of the parameter c. However this modification also modifies the constant in the forward time. How large is the new constant? When the modification proposed results in better total time? The answers to these questions and more details concerning the implementation of the proposed modification are discussed in the next two sections.

4 Application of the projection method to the knapsack problem recurrence equations

In this section we consider the implementation of the knapsack problem recurrence equations on linear array composed of q identical processing elements $C_k, k = 1, 2, \ldots, q$, where $q \geq m$. Each PE C_k has two addressable memories F_k and V_k each of size α. The purpose of F_k and V_k is to save the values of the functions f_k and v_k respectively.

Forward phase: The operation of the systolic array is best explained using the dependence projection method (see [28]), which corresponds to scheduling the dependence graph and projecting it along a conveniently chosen direction. As was noted in section 2 the similarity of the graphs corresponding to equations (2) and (3) is that they are both data dependence graphs. In any column k the dependence vectors depend on the associated weight w_k which is input data for the problem. That is, this is a run-time dependency. This peculiarity makes these dependence graphs rather irregulars. The most convenient direction of projection has to be chosen in order to hide this irregularity in the local memory of the processing element. Note that the DGs considered are well determined by the initial data at their borders (upper and left). This allows the computations to be performed successively in a pipeline fashion.

The computations in any column of the DP in figure 2 depend only on the computations of the previous column. Similar graphs have been studied in the paper of Li and Wa [23], where they are called multistage graphs. Usually such graphs are projected along the k axis, which yields a linear array of c processing elements. This projection is used by Lin and Storer in [12]. Their algorithm has optimal linear speedup of $O(q)$ on an EREW PRAM of q processors. But the implementation on the Connection Machine achieves speedup of $O(q/log\ q)$. This is due to the dependencies along the axis j which are not local and whose implementation on a hypercube takes up to $log\ q$ steps.

The dependence graph associated with equation (2) offers less opportunity for parallelization. The unique reasonable direction of projection is the j axis because of the dependencies in the columns themselves. In this case a straightforward projection along the j axis yields a linear unidirectional array of m processing elements. This mapping was studied in [11, 25, 30]. Compared to these papers, the partitioning we propose provides the best memory size requirement for the algorithm while preserving its processor efficiency. The approach is applicable to the DG resulting from equation (2) as well as to DG corresponding to equation (3).

A *timing function* is a mapping $t : \mathcal{D} \rightarrow N$, such that if the computations on vertex $v \in \mathcal{D}$ depend on the computations on vertex $w \in \mathcal{D}$, then $t(v) > t(w)$. An *allocation function* is a mapping $a : \mathcal{D} \rightarrow [1, q]$, such that a($v$) is the number of the processor that executes the calculations attached to vertex $v \in \mathcal{D}$. The mapping a must be chosen in such a way that a processor has no more than one computation to perform at a given instant. In addition to the previous well known functions we need the so-called *address function*. It is as a mapping $addr : \mathcal{D} \rightarrow [1, \alpha]$ such that addr(v) is the number of the memory location in processor a(v) where data $v \in \mathcal{D}$ is stored.

Obviously, an allocation function $a(j, k) = k$ corresponds to the chosen direction of projection of the dependence graph along axis j. Since for any k, the values $f_k(j), j = 0, 1, \ldots, c$ are computed sequentially and $f_k(j)$ depends on $f_k(j - w_k)$, the value $f_k(j)$ can be stored in the same memory location in C_k as the value $f_k(j-w_k)$. This implies the memory size α must meet the requirement $\alpha \geq w_k$, for any $1 \leq k \leq m$. The address of any data element $(j, k) \in \mathcal{D}$ in F_k is given by addr$(j, k) = j \bmod w_k$. The function $t(j, k) = j + k$ is a timing function. For the total time of the forward phase we obtain:

$$t(c, m) = c + m. \tag{8}$$

The program of the processing element is the following:

PE algorithm

```
repeat {forever}
        addr := j_in mod w;
        {compute f_k(j) in the variable f_max}
        if j_in < w then  f_max := f_in;
                  else  f_max  := max{f_in, F(addr) + p};
        {compute u_k(j) }
        if f_max ≤ f_in then  u_out := u_in;
                      else  u_out := obj;
        {compute v_k(j) and store in V}
        if f_max ≤ f_in then V(addr) := −1;
                      else  if V(addr) = −1 then V(addr) := j_in;
        if f_max ≤ f_in then  v_out := v_in;
                      else  v_out := V(addr);
        j_out  := j_in;
        f_out  := f_max;
end_repeat
```

To realize this algorithm each cell C_k keeps the values p_k, w_k and k in registers p, w, obj respectively. The data input into the leftmost cell during the first $c+1$ instants t, $t = 0, 1, \ldots, c$ are $f_{in} = 0, j_{in} = t, u_{in} = 0, v_{in} = 0$. The solution of the problem is $f_m(c)$. The values $u_m(j)$ and $v_m(j), j = 0, 1, \ldots, c$ are used in the backtracking phase.

Remark: Obviously the approach presented above can be applied to any recurrence similar to equation (2) or (3). For example *the subset-sum problem* exibits DP recurrence as in equation (3), while the *the change-making problem* exibits DP recurrence as in equation (2) (for the definitions of these problems see the book of Martello and Toth [2]).

Backtracking phase: The values $u_k(j)$ and $v_k(j)$ are computed and propagated through the considered array simultaneously with the value $f_k(j)$. As with the values of the function f_k, the values $v_k(j), j = 0, 1, \ldots, c$ are computed sequentially and $v_k(j)$ depends on $v_k(j - w_k)$, i.e. the value $v_k(j)$ can be stored in the same memory location in V_k as is the value $v_k(j - w_k)$. In other words, a memory V_k of size w_k in cell C_k is enough for the computation of the function v_k. The computation of the function u_k does not require a supplementary local memory. The values of the functions u_m and v_m leave the last cell C_m and are stored in a supplementary memory of size $2c$. Then the modified backtracking algorithm can be executed by the host computer. More details concerning the program implementation of recurrence (6) can be found in [26].

5 Discussion of the backtracking phase

A serial implementation of the backtracking phase is far from being crucial for efficiently solving the considered problem. Indeed the time for this phase ($\Theta(m+c)$ in the classical algorithm or $\Theta(m)$ in the modification proposed here) is negligable as compared with the total time $\Theta(mc)$. But the backtracking algorithm is sequential and its time cannot be improved on a parallel machine. In contrast to this, as we see from (8), given a multiprocessor with enough number of processing elements, the forward phase can be performed in $\Theta(m + c)$ time, which is comparable to the time for the backtracking. Therefore this phase is also worth studied when an implementation on a massively parallel architecture is considered.

Let us examine under which conditions the application of the modified backtracking algorithm can give better total time than the algorithm of Hu. The comparison is not obvious since the time constant in the forward phase is not the same when these two variants of the backtracking are executed. Denote by $T_{forward-Hu}$, $T_{forward-modi}$, $T_{backtrack-Hu}$, $T_{backtrack-modi}$ the time associated to the corresponding phases of the two variants of the dynamic programming implementation considered here. Assume the number of processors is equal to m and suppose we consider a software or firmware implementation. Then we have $T_{forward-Hu} = C_1(m + c)$ and $T_{backtrack-Hu} = C_2(m + c)$, where C_1 is the time taken to implement statements (2) and (4) and C_2 -the time constant from corollary 1. Analogously for the modified forward phase we require C_1 for (2), (4), but we incur an extra penalty C_3 to implement (6). This gives $T_{forward-modi} = (C_1 + C_3)(m + c)$. Finally corollary 2 implies $T_{backtrack-modi} = C_4 m$, where C_4 is the corresponding time constant. Then

$$T_{forward-Hu} + T_{backtrack-Hu} = C_1(m + c) + C_2(m + c)$$
$$> T_{forward-modi} + T_{backtrack-modi} = (C_1 + C_3)(m + c) + C_4 m$$

$$\Leftrightarrow C_2(m + c) > C_3(m + c) + C_4 m. \tag{9}$$

Theoretically C_3 and C_2 are much closer, and in practice the inequality (9) is hardly satisfied [3].

Consider now a custom VLSI implementation [4]. Denote by C_5 the time constant in the classical forward phase (i.e. recurrences (2) and (4)). Therefore we have $T_{forward-Hu} = C_5(m + c)$. However in the modified

[3] an illustration of this fact is our implementation on iWARP(see section 7)

[4] this point is reasonable since the results of [26, 31] demonstrate that efficient VLSI implementations for recurrences 2 and 3 exist

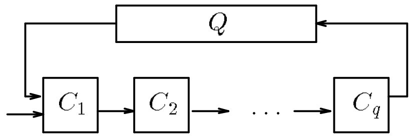

Figure 4: A linear processor array of length q and a queue

forward phase, in this case, the supplementary statement (6) is implemented in parallel with (2) and (4). Therefore we have the same time constant C_5 resulting in

$$T_{forward-modi} = T_{forward-Hu} = C_5(m+c).$$

Hence

$$T_{forward-Hu} + T_{backtrack-Hu} = C_5(m+c) + C_2(m+c)$$

and

$$T_{forward-modi} + T_{backtrack-modi} = C_5(m+c) + C_4 m.$$

Obviously the inequality

$$T_{forward-Hu} + T_{backtrack-Hu} > T_{forward-modi} + T_{backtrack-modi} \tag{10}$$

is valid since C_2 and C_4 are closer. For large values of c, (10) is satisfied even when the number of processor is not equal to m.

6 A Ring

The algorithm in the previous section needs m processing elements to solve the knapsack problem. In this section we compute the time of the algorithm on a ring composed of a linear array of q processing elements, each one with storage capacity at least w_{max} and a queue, which are connected as illustrated in figure 4. The queue memory receives the data from processing element C_q, stores it and sends it to C_1 when necessary. This memory saves the vectors u_m and v_m at the end of the forward phase. The number q is supposed fixed and $q \leq m$. Therefore, the linear array in the ring has to be used $\lceil m/q \rceil$ times to find $f_m(c)$. For this purpose, the set $f_k(j), k = 1, 2, \ldots, m, j = 0, 1, \ldots, c$ is partitioned into $\lceil m/q \rceil$ bands and each one of these bands can be evaluated in turn by the linear processor array. Let B_i denote the ith band. The partitioning is given by the relation $f_k(j) \in B_i \Leftrightarrow \lceil k/q \rceil = i$.

Since c values are input in any cell and a data needs to pass q time through C_1 to C_q, a new band can be input to the cell C_1 every $\max\{c, q\} = \Theta(c+q)$ time. Hence, we obtain for the total time complexity of our algorithm on the ring

$$T_q = \Theta(mc/q + m). \tag{11}$$

The speedup and the efficiency of the algorithm both approach their optimal values respectively $\Theta(q)$ and $\Theta(1)$ as c increases.

7 Computational experiments

The algorithm discussed in this paper is a fine-grained parallel algorithm which requires only local communication on a linear array. The partitioning scheme is suitable for systolic implementation on a multiprocessor machine. Our experiments have been conducted on an iWARP [32]. This machine supports the register-to-register communication model that is required to efficiently execute systolic programs [33]. The machine we use has 8 processors connected in a ring. To support efficient systolic experiments, we use the parallel language ReLaCS[34] which embodies both the computation and the communications aspects of systolic algorithms in a terse programming model.

The systolic network is viewed as a programmable accelerator connected to a host workstation. The programming model assumes a linear array of identical processors running in an SIMD mode. The input/output data management of the systolic network, as well as the computational tasks, are handled using a scalar processor connected between the two end-processors and the host.

The compiler produces three files of C source language from the program written in ReLaCS: the first is responsible for the interface between the host and the systolic accelerator (it is loaded on the host); the second is in charge of correctly feeding the data to the network and storing the results on the scalar processor; the third performs the actual parallel computation of the systolic network. Parallelism is explicitly expressed by data types and communication operators. Synchronization, and target-dependent communications and control mechanisms, are generated by the compiler.

To test the performance of the algorithm we performed several experiments. In each instance the number of objects and the size of the knapsack were specified, and the profits and weights were randomly generated.

The first experiment was designed to test the effect of the parameter c on the running time. In order to eliminate the influence of the ring, the number of objects m was fixed at 8 while the capacity c varied from 10 to 150000. Note that we are able to solve problems with really very large knapsack capacity because in our implementation this parameter does not influence the PE's memory size (see section 4). We observe that the running time (plotted in figure 5) is strictly linear in c. We compare as well the time of the classical DP implementation from section 2 with the running time of the modification proposed in this article section 3. The better behavior of the classical algorithm was theoretically expected because of the reasons explained in section 5 where we argue why the effect of the modified algorithm is difficult to observe on a software implementation. For these reasons in the next experiments we investigate only the behavior of the classical DP algorithm from section 2.

The effect of the parameter m on the running time of the algorithm is observed in the second experiment. The knapsack capacity and the number of PE is fixed respectively at 150000 and 8. The number of objects varies from 10 to 1000. This forced the data to circulate through the ring several times each time loading the new coefficients according to the partitioning in section 6. As we see from figure 6 this does not result in a slowdown and the theoretical linear behavior of the algorithm from (11) is observed.

The third experiment was designed to observe the speedup of the parallel algorithm (i.e. the ratio T_1/T_q where T_q is the running time of the algorithm on q processors and T_1 is the time required by a program written on the C language (not on our parallel language ReLaCS) and executed on one processor of iWARP. This experiment proves that the parallel algorithm is faster than the sequential one when more than 3 processors were used. The linear behavior of the speedup is also confirmed (see figure 7).

8 Concluding remarks

Dynamic programming is a based approach used to solve all types of knapsack problems. The following features make this approach worthy of study for parallel implementation:

- it efficiently solves instances of the knapsack problem which pose difficulties for the more popular B&B approach,
- it can be easily parallelized in a pipelined way using fine-grained parallelism,
- this requires only local communication on a linear array,

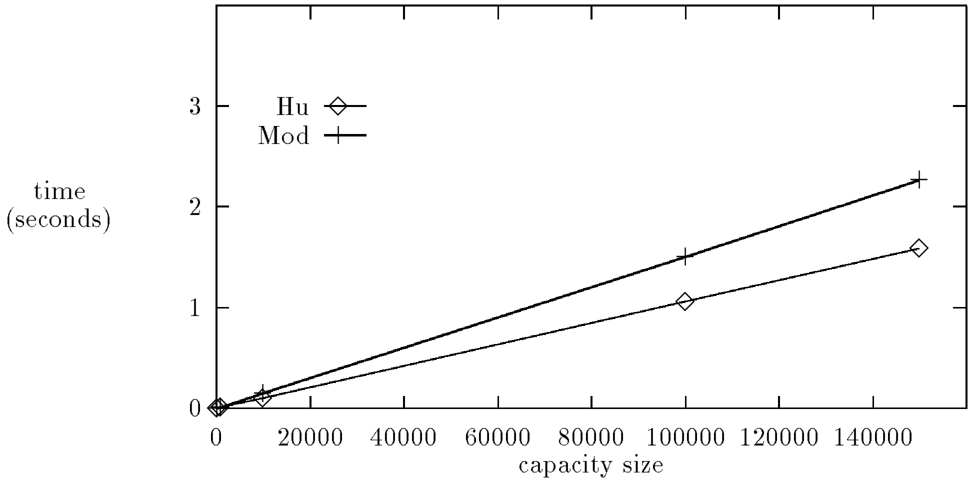

Figure 5: Problem size effect (q= 8, m= 8)

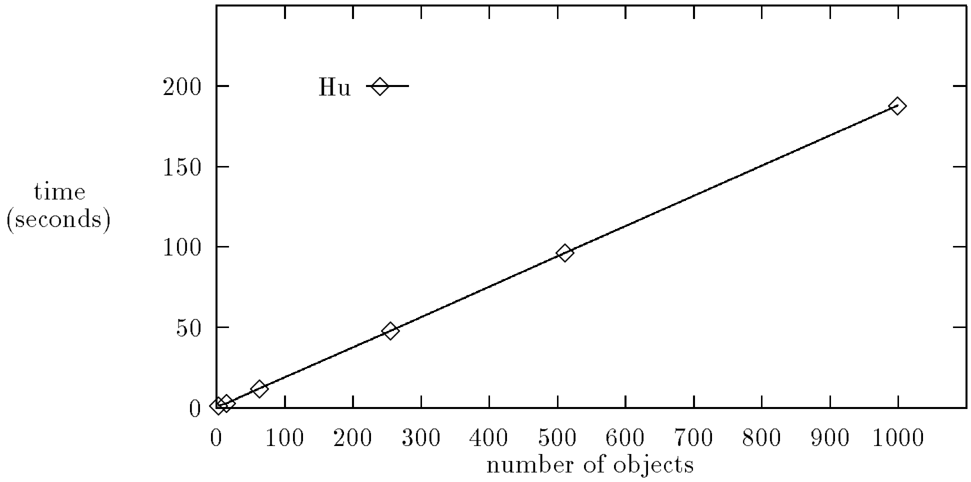

Figure 6: Ring effect (q= 8, c= 150000)

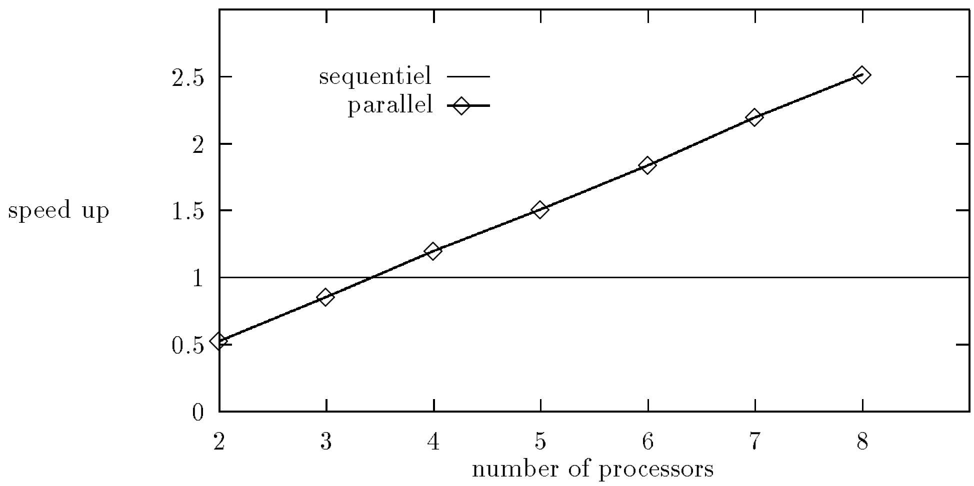

Figure 7: Speed up (c= 150000, m= 1000)

- the speedup of the obtained algorithm is theoretically optimal; when m processors are available the parallel implementation requires $m + c$ time units versus mc time units in a serial machine implementation,
- the second (backtracking) phase of the DP method is sequential; this phase requires some computational work during the first (forward) phase; these computations can be efficiently implemented when the first phase is parallelized,
- problems with very large knapsack capacity c can be considered because this parameter does not influence the memory needed to solve the problem; only the size of the object's weights determines the PE memory size necessary,
- the data partitioning scheme has the flexibility to run on a varied number of processors independent of the problem sizes.

Our computational experiments on the iWARP machine completely confirm the theoretical results and show the proposed algorithm performs well for a wide range of problem size.

Acknowledgments

The authors would like to thank Sanjay Rajopadhye and Nicola Yanev for many helpful discussions.

References

[1] S. Teng, "Adaptive parallel algorithm for integral knapsack problems," *J. of Parallel and Distributed Computing*, vol. 8, pp. 400–406, 1990.

[2] S. Martello and P. Toth, *Knapsack Problems: Algorithms and Computer Implementation.* John Wiley and Sons, 1990.

[3] R. Garfinkel and G. Nemhauser, *Integer Programming.* John Wiley and Sons, 1972.

[4] T. C. Hu, *Combinatorial Algorithms.* Addison-Wesley Publishing Company, 1982.

[5] M. Garey and D. Johnson, *Computers and Intractability: a Guide to the Theory of NP-Completeness.* Freeman, San Francisco, 1979.

[6] P. C. Gilmore and R. E. Gomory, "The theory and computation of knapsack functions," *Operations Research*, vol. 14, pp. 1045–1074, 1966.

[7] C. Chung, M. S. Hung, and W. O. Rom, "A hard knapsack problem," *Naval Research Logistics*, vol. 35, pp. 85–98, 1988.

[8] R. Jeroslow, "Trivial integer programs unsolvable by branch and bound," *Mathematical Programmnig*, vol. 6, pp. 105–109, 1974.

[9] V. Chvatal, "Hard knapsack problems," *Operations Research*, vol. 28, pp. 1402–1411, 1980.

[10] J. Lee, E. Shragowitz, and S. Sahni, "A hypercube algorithm for the 0/1 knapsack problems," *J. of Parallel and Distributed Computing*, vol. 5, pp. 438–456, 1988.

[11] G. Chen, M. Chern, and J. Jang, "Pipeline architectures for dynamic programming algorithms," *Parallel Computing*, vol. 13, pp. 111–117, 1990.

[12] J. Lin and J. A. Storer, "Processor-efficient hypercube algorithm for the knapsack problem," *J. of Parallel and Distributed Computing*, vol. 13, pp. 332–337, 1991.

[13] T. Lai and S. Sahni, "Anomalies in parallel branch-and-bound algorithms," *CACM*, vol. 27, no. 6, 1984.

[14] J. Jansen and F.M.Sijstermans, "Parallel branch-and-bound algorithms," *Future Generation Computer Systems*, vol. 4, pp. 271–279, 1988.

[15] R.L.Boehning, R. Butler, and B. Gillett, "A parallel integer linear programming algorithm," *European Journal of Operational Research*, vol. 34, pp. 393–398, 1988.

[16] G. Plateau and C. Roucairol, "A supercomputer algorithm for the 0-1 multiknapsack problem," in *Impact of recent computers advances on Operations Research*, pp. 144–157, 1989.

[17] C. Roucairol, "Parallel branch and bound algorithms - an overview," in *Parallel and Distributed Algorithms,* M. Cosnard et al. (editors), pp. 153–163, Elsevier Sc. Publishers, North Holland, 1989.

[18] G. Plateau, C. Roucairol, and I. Valabregue, "Un algorihme parallele PR^2 du multiknapsack," Tech. Rep. 811, INRIA, Domaine de Voluceau, Rocquencourt, B.P. 105, 78153 Le Chesney Cedex, France, 1988.

[19] A. P. Sprague, "Wild anomalies in parallel branch and bound," in *International Conference on Parallel Processing*, pp. III–308,III–309, 1991.

[20] F. Dehne, A. Ferreira, and A. Rau-Chaplin, "Parallel branch and bound on fine-grained hypercube multiprocessors," *Parallel Computing*, no. 15, pp. 201–209, 1990.

[21] G. Li and B. Wah, "Coping with anomalies in parallel branch-and-bound algorithms," *IEEE Trans. Computers*, vol. 35, pp. 568–573, 1986.

[22] T. Lai and A. Sprague, "Performance of parallel branch-and-bound algorithms," *IEEE Trans. Computers*, vol. 34, pp. 962–964, 1985.

[23] G. Li and B. W. Wah, "Systolic processing for dynamic programming problems," in *Proc. International Conference on Parallel Processing*, pp. 434–441, 1985.

[24] R. J. Lipton and D. Lopresti, "Delta transformation to symplify VLSI processor arrays for serial dynamic programming," in *Proc. International Conference on Parallel Processing*, pp. 917–920, 1986.

[25] G. Chen and J. Jang, "An improved parallel algorithm for 0/1 knapsack problem," *Parallel Computing*, vol. 18, pp. 811–821, 1992.

[26] R. Andonov and P. Quinton, "Efficient linear systolic array for the knapsack problem," in *CONPAR'92, Lecture Notes in Computer Science 634,* (Lyon, France), September 1992.

[27] R. Bellman, *Dynamic Programming.* Princeton University Press, Princeton, NJ, 1957.

[28] P. Quinton and Y. Robert, *Algorithmes et architectures systoliques.* Masson, 1989. English translation by Prentice Hall, *Systolic Algorithms and Architectures*, Sept. 1991.

[29] S. V. Rajopadhye and R. M. Fujimoto, "Synthesizing systolic arrays from recurrence equations," *Parallel Computing*, vol. 14, pp. 163–189, June 1990.

[30] R. Andonov, V. Aleksandrov, and A. Benaini, "A linear systolic array for the knapsack problem," Tech. Rep., Center of Computer Science and Technology, Acad. G. Bonchev st., bl. 25a, Sofia 1113, Bulgaria, 1991.

[31] W. P. Marnane and R. Andonov, "Algorithm engineering and its impact on VLSI design," Tech. Rep., IRISA, Campus de Beaulieu, 35042 Rennes Cedex, France, 1993. to appear.

[32] S. Borkar, R. Cohn, G. Cox, S. Gleason, T. Gross, H. Kung, M. Lam, B. Moore, C. Peterson, J. Pieper, L. Rankin, P. Tseng, J. Sutton, J. Urbanski, and J. Webb, "iWarp :An integrated Solution to High-Speed Parallel Computing," in *ICS*, 1988.

[33] H. Kung, "Systolic Communication," in *ISCA*, pp. 695–703, may 1988.

[34] D. Lavenier and F. Raimbault, "ReLaCS for Systolic Programming," Rapport interne 726, IRISA, may 1993. submitted to ASAP'93.

Unité de recherche INRIA Lorraine, Technôpole de Nancy-Brabois, Campus scientifique,
615 rue de Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 46 avenue Félix Viallet, 38031 GRENOBLE Cedex 1
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
ISSN 0249-6399