# Load balancing methods and parallel dynamic programming algorithm using dominance technique applied to the 0-1 knapsack problem

Didier El Baz, Moussa Elkihel

# Load balancing methods and parallel dynamic programming algorithm using dominance technique applied to the 0-1 knapsack problem

D. El Baz M. Elkihel

*LAAS-CNRS, 7, avenue du Colonel Roche, 31077 Toulouse CEDEX 4, France*
*E-mail: elbaz@laas.fr elkihel@laas.fr*

**Abstract**

The parallelization on a supercomputer of a one list dynamic programming algorithm using dominance technique and processor cooperation for the 0-1 knapsack problem is presented. Such a technique generates irregular data structure, moreover the number of undominated states is unforeseeable. Original and efficient load balancing strategies are proposed. Finally, computational results obtained with an Origin 3800 supercomputer are displayed and analyzed. To the best of our knowledge, this is the first time for which computational experiments on a supercomputer are presented for a parallel dynamic programming algorithm using dominance technique.

*Key words:* parallel computing, load balancing, 0-1 knapsack problem, dynamic programming, dominance technique,

## 1 Introduction

The 0-1 knapsack problem has been intensively studied in the literature (see for example [14], [3], [12], [19], [26], [24], [25], [21] and [23]). The objective of this paper is to concentrate on the parallelization on a supercomputer of a one list dynamic programming method using dominance technique and processor cooperation and to propose efficient load balancing strategies in order to achieve good performance.
Many authors have considered dense dynamic programming, i.e. an approach for which one takes into account all possible states ( see [1], [2], [4], [5], [7] and [15]). In this case, the number of states is equal to the capacity of the knapsack. In this paper, we study a different approach proposed by Ahrens and Finke (see [3]) which permits one to limit the number of states by using dominance technique. In this

case, the number of states or undominated pairs generated is unforeseeable. If we compare the two approaches, then we note that the former will generate a regular data stucture, from which one can easily deduce the total amount of work needed in order to treat the list. This approach leads generally to an easy parallelization; its main drawback is that it produces large lists in the case of problems with large capacity. The later approach presents the advantage to generate lists which are smaller; its main drawback is the creation of an irregular data structure. As a consequence, the parallelization of the later approach is not easy and the design of an efficient load balancing strategy is very important.
In [11] we have presented an original parallelization of the one list dynamic programming method using dominance technique. The cooperation via data exchange of processors of the architecture is the the main feature of the proposed parallel algorithm. A first load balancing strategy was also proposed in [11]. In this paper, we develop the parallel algorithm, specially on a theoretical point of view and propose several original load balancing strategies.
Our contribution is different from the other works in the literature devoted to parallel dynamic programming for 0-1 knapsack problems. In particular, it is different from [6] and [18], where the authors have considered approaches based on massive parallelism. More precisely, in the above quoted papers, the authors have proposed solution for arrays with up to $O(2^{\frac{n}{8}})$ processors, where $n$ is the number of variables in the knapsack problem. Our work is also different from [8], where the authors have studied the parallel implementation of a two lists algorithm on a MIMD architecture for the solution of a particular class of 0-1 knapsack problems: the exact subset sum problem where profits are equal to weights. In this later approach, total work is decomposed initially and processors do not cooperate. Note that our parallel algorithm is designed for a broad class of 0-1 knapsack problems including subset sum problems. Moreover, our parallel algorithm presents better time complexity than the parallel algorithm studied in [8], as we shall see in the sequel. Reference is also made to [7] and [13] for different approaches concerning the parallelisation of the dynamic programming method.
Section 2 deals with the 0-1 knapsack problem and its solution via dynamic programming. Parallel algorithm is studied in Section 3. Original load balancing strategies are proposed in Section 4. Finally, computational results obtained with an Origin 3800 supercomputer are displayed and analyzed in Section 5.

## 2 The 0-1 Knapsack Problem

The 0-1 unidimensional knapsack problem is defined as follows:

$$\max\left\{\sum_{j=1}^{n} p_j x_j | \sum_{j=1}^{n} w_j x_j \leq C \; ; x_j \in \{0,1\}, j = 1,2,\ldots,n\right\}, \tag{1}$$

where $C$ denotes the capacity of the knapsack, $n$ the number of items considered, $p_j$ and $w_j$, respectively, the profit and weight, respectively, associated with the $j$-th

item. Without loss of generality, we assume that all the data are positive integers. In order to avoid trivial solutions, we assume that we have: $\sum_{j=1}^{n} w_j > C$ and $w_j < C$ for all $j \in \{1, ..., n\}$. Several methods have been proposed in order to solve problem (1). We can quote for example: branch and bound methods proposed by Fayard and Plateau (see [12]), Lauriere (see [17]) and Martello and Toth (see [19]), methods based on dynamic programming studied by Horowitz and Sahni (see [14]), Ahrens and Finke (see [3]) and Toth (see [27]) and finally mixed algorithms combining dynamic programming and branch and bound methods presented by Martello and Toth (see [20]) and Plateau and Elkihel (see [26]).
In this paper, we concentrate on a dynamic programming method proposed by Ahrens and Finke whose time and space complexity are $O(\min\{2^n, nC\})$ (see [3]) and which is based on the concepts of list and dominance. We shall generate recursively as follows lists $L_k$ of pairs $(w, p)$, $k = 1, 2, \ldots, n$; where $w$ is a weight and $p$ a profit. Initially, we have $L_0 = \{(0, 0)\}$.
Let us define the set $N_k$ of new pairs generated at stage $k$; where new pairs results from the fact that a new item, i.e. the $k$-th item, is taken into account.

$$N_k = \{(w + w_k, p + p_k) \mid (w, p) \in L_{k-1},\ w + w_k \leq C\}. \tag{2}$$

According to the dominance principle, which is a consequence of Bellman's optimality principle, all pairs $(w, p) \in L_{k-1} \cup N_k$ obtained by construction such that there exists a pair $(w', p') \in L_{k-1} \cup N_k$, $(w', p') \neq (w, p)$, which satisfies: $w' \leq w$ and $p \leq p'$, must not belong to a list $L_k$. In this case, we usually say that the pair $(w', p')$ dominates the pair $(w, p)$. As a consequence, any two pairs $(w', p'), (w'', p'')$ of the list $L_k$ must satisfy: $p' < p''$ if $w' < w''$. Thus, we can define the set $D_k$ of dominated pairs at stage $k$ as follows.

$$\begin{aligned} D_k = \{(w, p) \mid (w, p) \in L_{k-1} \cup N_k, \exists (w', p') \in L_{k-1} \cup N_k \text{ with} \\ w' \leq w, p \leq p', (w', p') \neq (w, p)\}. \end{aligned} \tag{3}$$

As a consequence, for all positive integers $k$, the dynamic programming recursive list $L_k$ is defined as follows

$$L_k = L_{k-1} \cup N_k - D_k. \tag{4}$$

Note that the lists $L_k$ are organized as sets of monotonically increasing ordered pairs in both weight and profit. As a consequence, the largest pair of the list $L_n$, corresponds to the optimal value of the knapsack problem. We illustrate the dynamic programming procedure presented in this Section on a simple example displayed as follows.

$$n = 6 \text{ and } C = 16, \tag{5}$$

$$(w_1, ..., w_n) = (5, 3, 2, 1, 5, 9), \tag{6}$$

$$(p_1, ..., p_n) = (20, 8, 5, 4, 14, 27). \tag{7}$$

Table 1 shows the contents of the lists $N_k$, $D_k$ and $L_k$, for $k = 1$ to 6.

We note that the optimal pair is $(16, 52) \in N_6$, the optimal solution corresponds to $(x_1, ..., x_n) = (1, 0, 1, 0, 0, 1)$. We see that infeasible pairs with total weight greater than $C = 16$, such as for example $(8 + 9, 29 + 27) = (17, 56)$ and $(9 + 9, 32 + 27) = (18, 59)$ do not belong to the list $N_6$. Finally, we note that the cardinality of $D_k$ can become relatively large when $k$ increases.

| $k$ | lists | content |
|---|---|---|
| 0 | $L_0$ | (0, 0) |
| 1 | $N_1$ | (5, 20) |
| | $D_1$ | |
| | $L_1$ | (0, 0), (5, 20) |
| 2 | $N_2$ | (3, 8), (8, 28) |
| | $D_2$ | |
| | $L_2$ | (0, 0), (3, 8), (5, 20), (8, 28) |
| 3 | $N_3$ | (2, 5), (5, 13), (7, 25), (10, 33) |
| | $D_3$ | (5, 13) |
| | $L_3$ | (0, 0), (2, 5), (3, 8), (5, 20), (7, 25), (8, 28), (10, 33) |
| 4 | $N_4$ | (1, 4), (3, 9), (4, 12), (6, 24), (8, 29), (9, 32), (11, 37) |
| | $D_4$ | (3, 8), (8, 28) |
| | $L_4$ | (0, 0), (1, 4), (2, 5), (3, 9), (4, 12), (5, 20), (6, 24), (7, 25), (8, 29), (9, 32), (10, 33), (11, 37) |
| 5 | $N_5$ | (5, 14), (6, 18), (7, 19), (8, 23), (9, 26), (10, 34), (11, 38), (12, 39), (13, 43), (14, 46), (15, 47), (16, 51) |
| | $D_5$ | (5, 14), (6, 18), (7, 19), (8, 23), (9, 26), (10, 33), (11, 37) |
| | $L_5$ | (0, 0), (1, 4), (2, 5), (3, 9), (4, 12), (5, 20), (6, 24), (7, 25), (8, 29), (9, 32), (10, 34), (11, 38), (12, 39), (13, 43), (14, 46), (15, 47), (16, 51) |
| 6 | $N_6$ | (9, 27), (10, 31), (11, 32), (12, 36), (13, 39), (14, 47), (15, 51), (16, 52) |
| | $D_6$ | (9, 27), (10, 31), (11, 32), (12, 36), (13, 39), (14, 46), (15, 47), (16, 51) |
| | $L_6$ | (0, 0), (1, 4), (2, 5), (3, 9), (4, 12), (5, 20), (6, 24), (7, 25), (8, 29), (9, 32), (10, 34), (11, 38), (12, 39), (13, 43), (14, 47), (15, 51), (16, 52) |

Table 1. Example in the sequential case

## 3 Parallel Algorithm

In this Section, we detail the parallelization of the one list dynamic programming method using dominance technique. The parallel algorithm which was briefly presented in [11] is designed according to the SPMD model for a parallel architecture that can be viewed as a shared memory machine on a logical point of view. As we shall see in detail in Section 5 experiments have been carried out on a NUMA (non uniform memory access) supercomputer Origin 3800 by using the Open MP environment.
The main feature of this parallel algorithm is that all processors cooperate via data exchange to the construction of the global list. The global list is partitioned into sublists. Sublists are organized as sets of monotonically increasing ordered pairs in both weight and profit. Each sublist is generated by one processor of the parallel architecture. In particular, all dominated pairs are removed at each stage of the parallel dynamic programming method. More precisely, at stage $k$, each processor $E^i$ generates a sublist of the global list $L_k$, which is denoted by $L_k^i$. The total work is shared by the different processors and data exchange permits each processor to remove all dominated pairs from its sublist based on global information. It is important to keep in mind that parallel algorithms whereby dominated pairs are not removed at each stage do not correspond to a parallel implementation of the dynamic programming using dominance technique algorithm, except in the special case of the subset sum problem where profits are equal to weight. Moreover, in the general case, the approach whereby all dominated pairs are not removed from the sublists may not be efficient, since the number of dominated pairs may be huge. The benefit of parallelism can then be lost since parallel algorithms will have to deal with a greater number of pairs than the sequential dynamic programming algorithm using dominance technique.
Several issues must be addressed when considering the design of a parallel algorithm: initialization of the parallel algorithm, work decomposition, tasks assignation, data exchange and load balancing strategy.

### *3.1 Initialization, Work Decomposition and Task Assignment*

The initialization of the parallel algorithm is performed by a sequential dynamic programming algorithm using dominance technique. First of all, a sequential process performs $k(0)$ stages of the dynamic programming algorithm. This process generates a list which contains at least $lq$ pairs, where $q$ denotes the total number of processors and $l$ the minimal number of pairs per processor. Let $L_{k(0)}$ be the ordered list which results from the sequential initialization. The list $L_{k(0)}$ is partitioned as follows: $L_{k(0)} = \cup_{i=0}^{q-1} L_{k(0)}^i$, with $\mid L_{k(0)}^i \mid = l$, for $i = 1, ..., q-1$ and $\mid L_{k(0)}^0 \mid \geq l$, where $\mid L_{k(0)}^i \mid$ denotes the number of pairs of the sublist $L_{k(0)}^i$.
If all processors $E^i$ generate independently their sublist $L_k^i$ without sharing data

with any other processor, then, on a global point of view, some dominated pairs may belong to the sublists $L_k^i$, which induces finally an overhead. Thus, in the beginning of each stage, it is necessary to synchronize all the processors which must then share part of the data produced at the previous stage in order to discard all dominated pairs in a global way. The use of global dominance technique can reduce the cardinality of the sublists $L_k^i$. At each stage, the contents of the union of the generated sublists is the same as the contents of the list generated by a sequential dynamic programming algorithm using dominance technique as we shal see in detail in the next subsection.

### *3.2 Parallel processes*

We detail now the parallel algorithm. We present first the process of construction of the sublists $L_k^i$ generated at each stage. We introduce the various sublists that are created by the different processors $E^i$, $i = 0, ..., q-1$, at each stage $k$ in order to generate the sublists $L_k^i$. For all $i \in \{0, ..., q-1\}$, the smallest pair of $L_k^i$ in both weight and profit will be denoted by $(w_k^{i,0}, p_k^{i,0})$. The various sublists are defined as follows. For all $i = 0, ..., q-2$,

$$N_k^i = \{(w_{k-1}^{i,\cdot} + w_k, p_{k-1}^{i,\cdot} + p_k) \mid (w_{k-1}^{i,\cdot}, p_{k-1}^{i,\cdot}) \in L_{k-1}^i, w_{k-1}^{i,\cdot} + w_k < w_{k-1}^{i+1,0}\}, \quad (8)$$

and

$$N_k^{q-1} = \{(w_{k-1}^{q-1,\cdot} + w_k, p_{k-1}^{q-1,\cdot} + p_k) \mid (w_{k-1}^{q-1,\cdot}, p_{k-1}^{q-1,\cdot}) \in L_{k-1}^{q-1}, w_{k-1}^{q-1,\cdot} + w_k \leq C\}. \quad (9)$$

The sublist $N_k^i$ corresponds to the new list of pairs created at stage $k$ by processor $E^i$ from its sublist $L_k^{i-1}$ and the $k$-th item and which are assigned to processor $E^i$. Some pairs clearly do not belong to the sublist $N_k^i$, i.e. the pairs for which the weight $w_{k-1}^{i,\cdot} + w_k$ is greater than or equal to the weight $w_{k-1}^{i+1,0}$ of the smallest pair of the list $L_{k-1}^{i+1}$ generated by processor $E^{i+1}$. Those discarded pairs which are stored as shared variables are used by processors $E^j$ with $i < j$, in order to generate their sublists $L_k^j$ as we shall see in the sequel. It is important to note at this point that for all $i \in \{0, ..., q-1\}$, data exchange can then occur only between processor $E^i$ and processors $E^j$ with $i < j$. For this purpose, consider now the series of sets $C_k^i$ defined as follows. For all $i \in \{1, ..., q-2\}$,

$$C_k^i = \{(w_{k-1}^{j,\cdot} + w_k, p_{k-1}^{j,\cdot} + p_k) \mid (w_{k-1}^{j,\cdot}, p_{k-1}^{j,\cdot}) \in L_{k-1}^j, j < i,$$
$$w_{k-1}^{i,0} \leq w_{k-1}^{j,\cdot} + w_k < w_{k-1}^{i+1,0} \text{ or } w_{k-1}^{j,\cdot} + w_k < w_{k-1}^{i,0}, p_{k-1}^{j,\cdot} + p_k \geq p_{k-1}^{i,0}\}, \quad (10)$$

$$C_k^0 = \emptyset, \tag{11}$$

and

$$C_k^{q-1} = \{(w_{k-1}^{j,.} + w_k, p_{k-1}^{j,.} + p_k) \mid (w_{k-1}^{j,.}, p_{k-1}^{j,.}) \in L_{k-1}^j, j < q-1,$$
$$w_{k-1}^{q-1,0} \leq w_{k-1}^{j,.} + w_k < C \text{ or } w_{k-1}^{j,.} + w_k < w_{k-1}^{q-1,0}, p_{k-1}^{j,.} + p_k \geq p_{k-1}^{q-1,0}\}. \tag{12}$$

The sublist $C_k^i$ is in fact the set of pairs that are exchanged between all processors $E^m$, with $m < i$ and processor $E^i$, either in order to complete the sublist $L_k^i$ that will be produced by processor $E^i$ at stage $k$ or to permit processor $E^i$ to discard from its sublist some dominated pairs, at stage $k$. This last decision being made on a global point of view. In particular, it is important to note that all processors $E^j$, with $j > i$, must share with processor $E^i$ all the pairs created by $E^i$ that will permit $E^j$ to eliminate dominated pairs. In order to discard all the pairs which must not belong to the sublist $L_k^i$ and particularly dominated pairs, we introduce the series of sets $D_k^i$. For all $i = 0, ..., q-2$,

$$D_k^i = \hat{D}_k^i \cup \{(w,p) \mid w < w_{k-1}^{i+1,0} \text{ and } p \geq p_{k-1}^{i+1,0}\}, \tag{13}$$

with

$$\hat{D}_k^i = \{(w,p) \mid (w,p) \in L_{k-1}^i \cup N_k^i \cup C_k^i \text{ and } \exists (w',p') \in L_{k-1}^i \cup N_k^i \cup C_k^i,$$
$$(w',p') \neq (w,p) \text{ and } w' \leq w, p \leq p'\}, \tag{14}$$

and

$$D_k^{q-1} = \{(w,p) \mid (w,p) \in L_{k-1}^{q-1} \cup N_k^{q-1} \cup C_k^{q-1}, \exists (w',p') \in L_{k-1}^{q-1} \cup N_k^{q-1} \cup C_k^{q-1},$$
$$(w',p') \neq (w,p), w' \leq w, p \leq p'\}, \tag{15}$$

We note that $\hat{D}_k^i$ is the subset of $D_k^i$ which contains all dominated pairs in processor $E^i$ at stage $k$. A similar remark can be made for the set $D_k^{q-1}$. We note also that copies of the same pair may be present in in different processors. These copies permit processors to check dominance relation. All copies must be removed at each stage but one, i.e. the last one. As a consequence, the dynamic programming recursive sublists $L_k^i$ are defined as the following sets of monotonically increasing ordered pairs in both weight and profit. For all positive integer $k$ and all $i = 0, ..., q-1$,

$$L_k^i = L_{k-1}^i \cup N_k^i \cup C_k^i - D_k^i. \tag{16}$$

Initially, note that we have $L_{k(0)} = \cup_{i=0}^{q-1} L_{k(0)}^i$, as it was stated in the beginning of subsection 3.1. Assume now that for a given $k$, we have $L_{k-1} = \cup_{i=0}^{q-1} L_{k-1}^i$, then it follows clearly from the definition of $N_k$, $N_k^i$ and $C_k^i$ that $\cup_{i=0}^{q-1}(N_k^i \cup C_k^i) = N_k$, i.e.

no pair is lost in the construction and data exchange processes. Moreover it follows from the definitions of $\hat{D}_k^i$ and $D_k^{q-1}$ that $\cup_{i=0}^{q-2}\hat{D}_k^i \cup D_k^{q-1} = D_k$, i.e. no dominated pair remains in the sublists $L_k^i$. We note that the elimination of all dominated pairs in each processor is not local. The elimination is rather based on global information. Thus, it follows from (4) and (16) that $\cup_{i=0}^{q-1} L_k^i = L_k$.
We present now the parallel dynamic programming algorithm designed according to the Single Program Multiple Data (SPMD) model. The parallel algorithm was carried out on a nonuniform memory access (NUMA) shared memory Origin 3800 supercomputer by using the Open MP environment. More details about the machine can be found in Section 5. All the variables are local, otherwise it is said in the algorithm.

**Parallel Algorithm**

FOR $k = k(0) + 1$ TO $n$
DO
 FOR $i = 0$ TO $q - 1$
 DO IN PARALLEL
  BEGIN
   IF $i \neq q - 1$
   THEN
    IF the pair $(w_{k-1}^{i+1,0}, p_{k-1}^{i+1,0})$, is available
    THEN
     BEGIN
      $E^i$ generates $N_k^i$;
      $E^i$ stores as shared variables all pairs $(w_{k-1}^{i,.} + w_k, p_{k-1}^{i,.} + p_k)$ |
      $(w_{k-1}^{i,.}, p_{k-1}^{i,.}) \in L_{k-1}^i$, $w_{k-1}^{i+1,0} \leq w_{k-1}^{i,.} + w_k \leq C$;
      $E^i$ stores as shared variables all pairs $(w_{k-1}^{i,.} + w_k, p_{k-1}^{i,.} + p_k)$ |
      $(w_{k-1}^{i,.}, p_{k-1}^{i,.}) \in L_{k-1}^i$, $w_{k-1}^{i,.} + w_k < w_{k-1}^{i+1,0}$, $p_{k-1}^{i,.} + p_k \geq p_{k-1}^{i+1,0}$;
     END
   ELSE
    $E^{q-1}$ generates $N_k^{q-1}$;
   BARRIER OF SYNCHRONIZATION;
   $E^i$ generates $C_k^i$;
   $E^i$ generates $D_k^i$;
   $E^i$ generates $L_k^i$;
   IF $i \neq 0$
   THEN
    $E^i$ stores $(w_k^{i,0}, p_k^{i,0})$ as a shared variable;
  END

We note that all processors $E^j$, $j = 1, ..., q-1$, store the smallest pair $(w_{k-1}^{j,0}, p_{k-1}^{j,0})$ of their sublist $L_{k-1}^j$ as a shared variable since this value is used by processors $E^i$ with $i < j$ to determine what pairs must be stored as a shared variable in order to be exchanged with $E^j$. We note also that pair exchange between processors occurs always between a processor $E^i$ and a processor $E^j$, with $j > i$. Thus, processor $E^{q-1}$

plays a particular part with this type of data exchange since, generally, it tends to accumulate more pairs than any other processor.

The particular data structure chosen in order to store exchanged pairs as shared variables is a table with three entries. Typically, processor $E^i$ will read data exchanged in $i$ lists which are generated, respectively by processors $E^0$ to $E^{i-1}$.

In order to illustrate the parallel algorithm proposed in this Section, we consider the same simple example as in Section 2, for which the data are given in equations (5) to (7) and the number of processors $q = 3$.

Table 2 displays the content of the sublists $L_k^i$, $N_k^i$, $C_k^i$ and $D_k^i$, $i = 0, 1, 2$, in function of the stage $k$. In order to maximize the parallel part of the program, we have chosen a value of $k(0)$ which as small as possible, while permitting processors to get non void sublists $L_k^i$, i.e. $k(0) = 2$. Thus, $\mid L_{k(0)}^i \mid = l = 1$, for $i = 1, 2$ and $\mid L_{k(0)}^0 \mid = 2$. We note that processor $E^0$ stores the pair $(3 + 2, 8 + 5) = (5, 13)$ as a shared variable in order to be used by processor $E^1$, at stage 3, since $w_2^{1,0} = 5$ and $5 \leq w_2^{2,0} = 8$. As a result, the pair $(5, 13)$ will belong to the set $C_3^1$. However, the pair $(5, 13)$ is dominated by the pair $(5, 20)$. Thus, the pair $(5, 13)$ will not belong to the sublist $L_3^1$. Similarly, processor $E^0$ stores the pair $(0+9, 0+27) = (9, 27)$ as a shared variable in order to be used by processor $E^2$, at stage 6, since $9 \geq w_5^{2,0} = 8$. The pair $(9, 27)$ will belong to the set $C_6^2$. However, once again the pair $(9, 27)$ is dominated by the pair $(8, 29)$. Thus, the pair $(9, 27)$ will not belong to the sublist $L_6^2$. We note also that some load unbalancing can appear, such as for example at stage 5 and can increase during the following stages.

## 4 Load Balancing Strategies

In order to obtain good performance, it is necessary to design an efficient load balancing strategy. As a matter of fact, if no load balancing technique is implemented, then it results in particular from the data exchange process described in the previous Section that processor $E_q$ can become overloaded.

In this Section, we propose and compare several load balancing strategies which are designed in order to obtain a good efficiency while presenting a small overhead. With these load balancing strategies the time complexity of the parallel algorithm presented in Section 3 is $O(\min\{\frac{2^n}{q}, \frac{nC}{q}\})$, since the number of pairs will be fairly distributed on the different processors. These strategies are different from the one considered in [11], which is an adaptive strategy whereby a load balancing is made if any processor which is overloaded can take benefit of it and a decision is taken every two stages according to measures and estimations of the load. For more details concerning this strategy, the reader is referred to [11].

| | i | 0 | 1 | 2 |
|---|---|---|---|---|
| $k$ | lists | | | |
| 2 | $L_2^i$ | (0, 0), (3, 8) | (5, 20) | (8, 28) |
| 3 | $N_3^i$ | (2, 5) | (7, 25) | (10, 33) |
| | $C_3^i$ | | (5, 13) | |
| | $D_3^i$ | | (5, 13) | |
| | $L_3^i$ | (0, 0), (2, 5), (3, 8) | (5, 20), (7, 25) | (8, 28), (10, 33) |
| 4 | $N_4^i$ | (1, 4), (3, 9), (4, 12) | (6, 24) | (9, 32), (11, 37) |
| | $C_4^i$ | | | (8, 29) |
| | $D_4^i$ | (3, 8) | | (8, 28) |
| | $L_4^i$ | (0, 0), (1, 4), (2, 5), (3, 9), (4, 12) | (5, 20), (6, 24), (7, 25) | (8, 29), (9, 32), (10, 33), (11, 37) |
| 5 | $N_5^i$ | | | (13, 43), (14, 46), (15, 47), (16, 51) |
| | $C_5^i$ | | (5, 14), (6, 18), (7, 19) | (8, 23), (9, 26), (10, 34), (11, 38), (12, 39) |
| | $D_5^i$ | | (5, 14), (6, 18), (7, 19) | (8, 23), (9, 26), (10, 33), (11, 37) |
| | $L_5^i$ | (0, 0), (1, 4), (2, 5), (3, 9), (4, 12) | (5, 20), (6, 24), (7, 25) | (8, 29), (9, 32), (10, 34), (11, 38), (12, 39), (13, 43), (14, 46), (15, 47), (16, 51) |
| 6 | $N_6^i$ | | | |
| | $C_6^i$ | | | (9, 27), (10, 31), (11, 32), (12, 36), (13, 39), (14, 47), (15, 51), (16, 52) |
| | $D_6^i$ | | | (9, 27), (10, 31), (11, 32), (12, 36), (13, 39), (14, 46), (15, 47), (16, 51) |
| | $L_6^i$ | (0, 0), (1, 4), (2, 5), (3, 9), (4, 12) | (5, 20), (6, 24), (7, 25) | (8, 29), (9, 32),(10, 34), (11, 38), (12, 39), (13, 43), (14, 47), (15, 51), (16, 52) |

Table 2. Example in the parallel case without load balancing

### *4.1 A dynamic load balancing strategy*

The first strategy considered in this paper does not necessarily balance loads at each stage; this strategy is based upon a test which is made at each stage. The load balancing test is based upon a comparison of the work needed for performing the load balancing on the one hand and the work resulting from the load unbalancing on the other hand. The later work is related to the difference of number of pairs between the largest sublist and the other lists. If the load balancing work is more expensive than the work needed for processing pairs, then the loads are not balanced, otherwise they are balanced. The load balancing process will assign fairly loads to processors, i.e. it will give approximatively an equal number of pairs to all processors as we shall see in what follows.

In the sequel, $T_p$, $T_w$ and $T_r$, respectively, denote the processing time, the writing time and the reading time relative to one pair, respectively. At any given stage $k$, the number of pairs of the largest sublist is denoted by $N_l$ and the total number of pairs assigned to all processors is denoted by $N_t$. The load unbalancing cost which is denoted by $c_u$ is given by

$$c_u = T_p.(N_l - \frac{N_t}{q}). \tag{17}$$

The load balancing cost which is denoted by $c_b$ is given by

$$c_b = N_l.(T_r + T_w), \tag{18}$$

since read and write are made in parallel in each processor. Thus, the test will be basically as follows. If $c_u > c_b$, then we balance loads, else the loads are not balanced. The test can be rewritten as follows

$$N_l - \frac{N_t}{q} > N_l.\frac{(T_r + T_w)}{T_p}, \tag{19}$$

which can also be rewritten

$$1 - \frac{N_t}{q.N_l} > \frac{(T_r + T_w)}{T_p}, \tag{20}$$

or

$$1 - \frac{(T_r + T_w)}{T_p} > \frac{N_t}{q.N_l}. \tag{21}$$

In the next Section, we will present computational experiments carried out on the Origin 3800 parallel supercomputer. We have obtained the following measurements on the Origin 3800 for $T_p$, $T_r$ and $T_w$.

$$T_p = 2.69.10^{-7} \text{ s}, T_r = 4.2.10^{-8} \text{ s and } T_w = 3.6.10^{-8} \text{ s}. \tag{22}$$

Thus, for this machine we have

$$\frac{(T_r + T_w)}{T_p} = 0.29, \tag{23}$$

and the practical test is given as follows.

$$0.71 > \frac{N_t}{N_l.q}. \tag{24}$$

The reader is referred to [22] for dynamic load balancing approaches which present some similarities with our dynamic strategy and which are applied to adaptive grid-calculations for unsteady three-dimensional problems. However, we note that our test (19) is different from the test used in [22] (reference is also made to [9] and [16]). In order to illustrate the load balancing strategy proposed in this subsection, we condider the same simple example as in the previous Section, for which the data are given in equations (5) to (7).

Table 3 displays the content of the sublists $L_k^i$, $N_k^i$, $C_k^i$ and $B_k^i$, $i = 0, 1, 2$, in function of $k$, where $B_k^i$ denotes the sublist assigned to processor $E^i$ after a load balancing performed at stage $k$. The sublists $D_k^i$, $i = 0, 1, 2$, do not appear in Table 3 for simplicity of presentation; however, dominance techniques are applied and as a consequence, the resulting sublists $L_k^i$ do not contain dominated pairs.

We note that the load balancing condition is not satisfied before stage 5. At stages 3 and 4, respectively, $\frac{N_t}{N_l.q}$ is equal to 0.78 and 0.8, respectively. Thus, there is no load balancing when $k$ is equal to 3 or 4. At stage 5, we have $N_t = 17$ and $N_l = 9$. Thus, $0.71 > \frac{N_t}{N_l.q} = 0.63$. The load balancing phase assigns fairly loads to processors, i.e. almost the same number of pairs are assigned to the different processors. We have $\mid B_5^1 \mid = \mid B_5^2 \mid = \lfloor \frac{N_t}{3} \rfloor = 5$ and $\mid B_5^0 \mid = \mid N_t \mid -2.\lfloor \frac{N_t}{3} \rfloor = 7$, where $\lfloor y \rfloor$ denotes the entire part of $y$.

The new sublists $B_5^i$ which are assigned to processors $E^i$, $i = 0, 1, 2$, after load balancing are used at stage 6, by processor $E^i$ together with the six-th and last item, i.e. $(9, 27)$, in order to generate the sublists $N_6^i$, $i = 0, 1, 2$. We note that the sets $N_6^i$, $i = 0, 1, 2$, are empty since the generated pairs at stage 6 are such that their weights $w_5^{i,\cdot} + 9$ are greater than or equal to the weight $w_5^{i+1,0}$ of the smallest pair of the list $B_5^{i+1}$ relevant to processor $E^{i+1}$ or greater than the capacity of the knapsack $C$. Those discarded pairs which are stored as shared variables are used by processors $E^j$ with $j > i$, in order to generate their sublists $C_6^j$. Finally, new pairs resulting from the last object taken into account and data exchange between processors permit one to build the sublists $L_6^i$, $i = 0, 1, 2$.

| | i | 0 | 1 | 2 |
|---|---|---|---|---|
| $k$ | lists | | | |
| 2 | $L_2^i$ | (0, 0), (3, 8) | (5, 20) | (8, 28) |
| 3 | $N_3^i$ | (2, 5) | (7, 25) | (10, 33) |
| | $C_3^i$ | | (5, 13) | |
| | $L_3^i$ | (0, 0), (2, 5), (3, 8) | (5, 20), (7, 25) | (8, 28), (10, 33) |
| 4 | $N_4^i$ | (1, 4), (3, 9), (4, 12) | (6, 24) | (9, 32), (11, 37) |
| | $C_4^i$ | | | (8, 29) |
| | $L_4^i$ | (0, 0), (1, 4), (2, 5), (3, 9), (4, 12) | (5, 20), (6, 24), (7, 25) | (8, 29), (9, 32), (10, 33), (11, 37) |
| 5 | $N_5^i$ | | | (13, 43), (14, 46), (15, 47), (16, 51) |
| | $C_5^i$ | | (5, 14), (6, 18), (7, 19) | (8, 23), (9, 26), (10, 34), (11, 38), (12, 39) |
| | $L_5^i$ | (0, 0), (1, 4), (2, 5), (3, 9), (4, 12) | (5, 20), (6, 24), (7, 25) | (8, 29), (9, 32), (10, 34), (11, 38), (12, 39), (13, 43), (14, 46), (15, 47), (16, 51) |
| | $B_5^i$ | (0, 0), (1, 4), (2, 5), (3, 9), (4, 12), (5, 20), (6, 24) | (7, 25), (8, 29), (9, 32), (10, 34), (11, 38) | (12, 39), (13, 43), (14, 46), (15, 47), (16, 51) |
| 6 | $N_6^i$ | | | |
| | $C_6^i$ | | (9, 27), (10, 31), (11, 32) | (12, 36), (13, 39), (14, 47), (15, 51), (16, 52) |
| | $L_6^i$ | (0, 0), (1, 4), (2, 5), (3, 9), (4, 12), (5, 20), (6, 24) | (7, 25), (8, 29), (9, 32), (10, 34), (11, 38) | (12, 39), (13, 43), (14, 47), (15, 51), (16, 52) |

Table 3. Example in the parallel case with load balancing

In the next subsection, we present a different load balancing strategy which is simple and performant.

### *4.2 Implicit load balancing*

Implicit load balancing has been designed in order to decrease overhead while performing a fair load balancing. The principle of implicit load balancing is very simple. Since the capacity $C$ of the knapsack is given and the size of the lists $L_k$ is at most equal to $C$, the idea is to assign to processor $E^0$ the pairs with weight between 0 and $\lfloor \frac{C}{q} \rfloor$ where $q$ denotes the number of processors, similarly, processor $E^1$ will be assigned the pairs with weight between $\lfloor \frac{C}{q} \rfloor + 1$ and $2.\lfloor \frac{C}{q} \rfloor$ and so on.

The main advantage of this strategy is that pairs are directly assigned to a given processor according to their weight. As a consequence, there is no overhead like in the case of the previous strategy. On the other hand, the main drawback of this strategy is its inefficiency at initialization, since idle times of some processors such as for example $E^{q-1}$, $E^{q-2}$, ... may be nonnegligeable. However, we will see in the next Section that this load balancing technique is performant even when the number of processors is large. As a matter of fact, with these technique, the work of the different processors tends to be well distributed, since according to the strategy, the number of pairs assigned to all processors tends to be constant in working regime.

In the case of implicit load balancing, it is straightforward to deduce the contents of the lists; thus, the presentation is skipped. Finally, we conclude this Section by presenting a last strategy the so-called cascade load balancing.

### *4.3 Cascade load balancing*

We have seen in subsection 4.1 that one of the major drawbacks of dynamic load balancing was that every processor must write the pairs of its sublist as shared variables during load balancing phases. This mechanism clearly generates a non-negligeable overhead. Moreover, the sublists assigned to the different processors differ only by a limited number of pairs before and after a load balancing. The cascade load balancing concept tends to suppress this drawback. The principle of cascade load balancing consists in the transmission of pairs at each stage from a given overloaded processor say $E^i$ to the processors next to it: say $E^{i-1}$ or $E^{i+1}$. This mechanism tends to limit the number of communicated pairs at each stage.

In order to implement this load balancing strategy, one needs to compute and store in a table the number of pairs per processor and the desired fair distribution of pairs. By using the data of this table, it is then possible to compute the number of pairs that must be communicated between each processor $E^i$ and processors $E^{i-1}$ or $E^{i+1}$, starting from processor $E^0$, data exchange operations being considered in sequence i.e. one at a time. A positive number $m$ implies that $m$ pairs must be sent from $E^i$ to $E^{i+1}$. A negative number $m$ implies that $\mid m \mid$ pairs must be sent from $E^{i+1}$ to $E^i$. As an example, if at a given stage $k$, the number of pairs of processors $E^0$, $E^1$ and $E^2$, respectively, are 18, 8 and 4, repectively, then at the first step of the cascade load balancing, $E^0$ will communicate 8 pairs to $E^1$. Thus, at the end of

the first step the respective number of pairs assigned to processors $E^0$, $E^1$ and $E^2$, will be 10, 16 and 4, respectively. At the second step of the cascade load balancing, $E^1$ will communicate 6 pairs to $E^2$. So, each processor will have the same number of pairs, i.e. 10 pairs, at the end of step 2. As we shall see in the sequel, very few pairs are communicated at each stage. The ratio communicated pairs per total number of pairs was usually one per one thousand in the numerical experiments we have carried out. In working regime, there are relatively few data exchanges between processors and the phenomenom of cascade communication of pairs generally occurs only at the beginning of computations.

Finally, we note that we have implemented many more approaches for load balancing then the ones quoted in this Section. In particular, the approach whereby the processor which tends to be overloaded i.e. the last processor get only few pairs at each load balancing has not proven to be very efficient, mainly because pair generation is expensive. We have also tested an approach whereby exchanges occur in the sense of the augmenting indexes at each odd stage and in the reverse sense at each even stage. But this last approach (see [11]) is less performant as well.

## 5 Numerical Results

The numerical experiments presented here correspond to difficult 0-1 knapsack problems, i.e problems included in a range from weakly correlated problems to very strongly correlated problems (see [10]). We have avoided treating noncorrelated problems which induce a large number of dominated pairs and which are thus more easy to solve.

The various instances considered are relative to problem sizes which are equal to 200, 400, 1000, 5000 and 10000, respectively, with data range defined as follows. The weights $w_j$ are randomly distributed in the segment $[1, 1000]$ and the nonnegative profits $p_j$ in $[w_j - g, w_j + g]$, where the gap, denoted by $g$, is equal to 10 for the first set of data and 100 for the second set. We have considered problems such that $C = 0.5 \sum_{j=1}^{n} w_j$.

The parallel algorithm has been implemented in C on a NUMA (non uniform memory access) shared memory supercomputer Origin 3800 using the Open MP environment. The architecture of the machine is hybrid, i.e. there is some shared and distributed memory. However, total memory can be viewed as shared on a logical point of view. More precisely, the parallel architecture is an hypercube constituted by 512 processors MIPS R14000 with 500 MHz clock frequency and 500 Mo of memory per processor. The operating system is IRIX 6.5 IP35. The total bandwith of the communication network is 716 Go/s. We note that generally, all processors do not have the same read or write time for every variable. With this architecture, the read or write time in a remote part of the memory may be 2 or 3 times greater than the read or write time in the local memory of a processor. However, we have made computational tests with $2.10^6$ up to $8.10^7$ pairs; in this range, the read time, as well as the write time were always the same: $4.10^{-8}$ seconds.

Parallel numerical experiments have been carried out with up to 32 processors. Numerical results are displayed on Tables 4 to 10.

For all tables, except Table 10, we give the running time in seconds of sequential algorithms, denoted by $t_s$, and parallel algorithms, denoted by $t_p$, which corresponds to an average time taken over 25 instances; we also give the efficiency of parallel algorithms which is equal to $\frac{t_s}{q.t_p}$. Table 10 corresponds to a single instance without load balancing strategy.

In the case of parallel algorithms with load balancing, we note from Tables 4 to 7 that the efficiency of the parallel algorithm is function of several parameters such as the size of the problem, the number of processors or the type of correlation of the data.

If the size of the problem is small, i.e. 200, 400 or 1000, then we do not need a large number of processors, since the running time is of the order of just few seconds. In that case, the efficiency of the parallel dynamic programming algorithm using dynamic or implicit load balancing tends generally to decrease when the number of processors increases. Basically, for these problems the granularity, i.e. the ratio computation time over communication time, which is initially small, decreases when $q$ increases since in this case, communications play a prominent part.

If the size of the problem is nonnegligible or great, i.e. 5000 or 10000, then the efficiency of the parallel dynamic programming algorithm using dynamic or implicit load balancing increases first because when $q$ is small the last processor tends naturally to accumulate more pairs and the problem is more unbalanced. Thus, the efficiencies are smaller for small values of $q$ since the effect of load unbalancing is relatively costly. Finally, the efficiency decreases when $q$ becomes large since the load balancing overhead in the case of dynamic load balancing and the granularity effects in both cases become prevalent.

A surprising result is that the performance of implicit load balancing is quite similar to the performance of dynamic load balancing. However, the sophisticated dynamic load balancing strategy seems to perform better than the simple implicit load balancing strategy for large size problems and large number of processors; this case being of course the most interesting.

We note that cascade load balancing strategy is generally very efficient for a small number of processors i.e. when $q$ is smaller or equal to 8. We note also that in some cases, the efficiency may be greater than one. Experiments were carried out on a non uniform memory architecture. Thus, several processors can use more efficiently their fast local memory, where local data are stored, then a single processor which needs to access sometimes remote part of the memory in order to use all the data of the problem. We recall that the access to remote part of the memory costs more than the access to local memory on a NUMA architecture. If the number of processors is large, then the performance of this strategy is very poor. It seems that although there are generally few pairs communicated as compared with the total number of pairs (we have measured an average of 1 per 1000), the sequentiality of data exchanges, i.e. one processor communicate at a time with another, seems particularly costly if the number of processors is large; this phenomenon eventualy induces an important communication overhead.

Finally, we note that load balancing is really important for parallel dynamic programming algorithms. Parallel algorithms without load balancing strategy are gen-

erally totally inefficient as shown in Table 10. Parallel algorithms with dynamic or implicit load balancing strategy whose performance is displayed in Tables 4 to 7 present in general a good efficiency for a coarse granularity. This shows that the first two load balancing strategies that we have designed are efficient. The performance of these load balancing strategies is also better than the one of the strategy presented in one of our previous papers (see [11]).

| q | 1 | 2 | | 4 | | 8 | | 16 | | 32 | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| size | $t_s$ | $t_p$ | e | $t_p$ | e | $t_p$ | e | $t_p$ | e | $t_p$ | e |
| 200 | 0.406 | 0.326 | 62% | 0.177 | 57% | 0.111 | 46% | 0.09 | 28% | 0.13 | 10% |
| 400 | 2.046 | 1.638 | 62% | 0.828 | 62% | 0.447 | 57% | 0.285 | 45% | 0.277 | 23% |
| 1000 | 15.099 | 11.571 | 65% | 5.907 | 64% | 3.034 | 62% | 1.65 | 57% | 1.111 | 42% |
| 5000 | 528.195 | 362.882 | 73% | 163.258 | 81% | 80.645 | 82% | 40.163 | 82% | 21.571 | 77 % |
| 10000 | 2125.38 | 1595.35 | 67% | 699.192 | 76% | 324.488 | 82% | 160.465 | 83% | 83.066 | 80% |

Table 4. Computing time in seconds and efficiency
for a gap 10 and range 1000 with dynamic load balancing

| q | 1 | 2 | | 4 | | 8 | | 16 | | 32 | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| size | $t_s$ | $t_p$ | e | $t_p$ | e | $t_p$ | e | $t_p$ | e | $t_p$ | e |
| 200 | 0.116 | 0.1 | 58% | 0.063 | 46% | 0.046 | 32% | 0.053 | 14% | 0.1 | 4% |
| 400 | 0.775 | 0.606 | 64% | 0.325 | 60% | 0.19 | 51% | 0.148 | 33% | 0.191 | 13% |
| 1000 | 7.935 | 6.252 | 63% | 3.137 | 63% | 1.66 | 60% | 0.96 | 52% | 0.768 | 32% |
| 5000 | 463.48 | 299.352 | 77% | 137.928 | 84% | 67.866 | 85% | 34.122 | 85% | 18.014 | 80% |
| 10000 | 2109.3 | 1655.84 | 64% | 644.888 | 82% | 302.5 | 87% | 144.158 | 91% | 76.518 | 86% |

Table 5. Computing time in seconds and efficiency
for a gap 100 and range 1000 with dynamic load balancing

| q | 1 | 2 | | 4 | | 8 | | 16 | | 32 | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| size | $t_s$ | $t_p$ | e | $t_p$ | e | $t_p$ | e | $t_p$ | e | $t_p$ | e |
| 200 | 0.406 | 0.30 | 68% | 0.17 | 58% | 0.11 | 48% | 0.07 | 35% | 0.07 | 18% |
| 400 | 2.046 | 1.45 | 70% | 0.81 | 64% | 0.45 | 57% | 0.26 | 48% | 0.20 | 32% |
| 1000 | 15.099 | 10.59 | 71% | 5.77 | 65% | 3.08 | 61% | 1.68 | 56% | 1.02 | 46% |
| 5000 | 528.195 | 349.57 | 76% | 161.62 | 82% | 83.88 | 79% | 43.94 | 75% | 23.33 | 71% |
| 10000 | 2125.38 | 1597.39 | 67% | 749.33 | 71% | 347.56 | 76% | 177.71 | 75% | 92.50 | 72% |

Table 6. Computing time in seconds and efficiency
for a gap 10 and range 1000 with implicit load balancing

| q | 1 | 2 | | 4 | | 8 | | 16 | | 32 | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| size | $t_s$ | $t_p$ | e | $t_p$ | e | $t_p$ | e | $t_p$ | e | $t_p$ | e |
| 200 | 0.116 | 0.09 | 62% | 0.06 | 49% | 0.04 | 35% | 0.04 | 21% | 0.04 | 9% |
| 400 | 0.775 | 0.56 | 69% | 0.32 | 60% | 0.19 | 52% | 0.12 | 40% | 0.11 | 22% |
| 1000 | 7.935 | 5.67 | 70% | 3.09 | 64% | 1.68 | 59% | 0.95 | 52% | 0.64 | 39% |
| 5000 | 463.48 | 273.42 | 85% | 130.27 | 89% | 67.18 | 86% | 34.80 | 83% | 18.78 | 77% |
| 10000 | 2109.3 | 1954.24 | 54% | 773.64 | 68% | 303.8 | 87% | 154.49 | 85% | 80.12 | 82% |

Table 7. Computing time in seconds and efficiency
for a gap 100 and range 1000 with implicit load balancing

| q | 1 | 2 | | 4 | | 8 | | 16 | | 32 | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| size | $t_s$ | $t_p$ | e | $t_p$ | e | $t_p$ | e | $t_p$ | e | $t_p$ | e |
| 200 | 0.406 | 0.23 | 88% | 0.18 | 56% | 0.3 | 17% | 0.81 | 3% | 2.09 | 0.6% |
| 400 | 2.046 | 1.09 | 94% | 0.7 | 73% | 0.88 | 29% | 2.2 | 6% | 5.4 | 1% |
| 1000 | 15.099 | 7.6 | 99% | 4.38 | 86% | 3.63 | 52% | 6.86 | 14% | 16.29 | 3% |
| 5000 | 528.195 | 256.84 | 103% | 109.82 | 120% | 63.24 | 104% | 61.32 | 53% | 106.82 | 15% |
| 10000 | 2125.38 | 1854.95 | 57% | 682.26 | 78% | 254.75 | 104% | 199.14 | 67% | 257.25 | 26% |

Table 8. Computing time in seconds and efficiency
for a gap 10 and range 1000 with cascade load balancing

| q | 1 | 2 | | 4 | | 8 | | 16 | | 32 | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| size | $t_s$ | $t_p$ | e | $t_p$ | e | $t_p$ | e | $t_p$ | e | $t_p$ | e |
| 200 | 0.116 | 0.08 | 73% | 0.07 | 41% | 0.12 | 12% | 0.35 | 2% | 0.8 | .5% |
| 400 | 0.775 | 0.47 | 82% | 0.32 | 61% | 0.4 | 24% | 1.03 | 5% | 2.53 | 1% |
| 1000 | 7.935 | 4.34 | 91% | 2.52 | 79% | 2.17 | 46% | 4.25 | 12% | 9.51 | 3% |
| 5000 | 463.48 | 198.63 | 117% | 90.83 | 128% | 51.79 | 112% | 49.19 | 59% | 81.19 | 18% |
| 10000 | 2109.3 | 1093.25 | 96% | 437.26 | 121% | 211.56 | 125% | 161.84 | 81% | 183.89 | 36% |

Table 9. Computing time in seconds and efficiency
for a gap 100 and range 1000 with cascade load balancing

| q | 1 | 2 | | 4 | | 8 | | 16 | | 32 | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| size | $t_s$ | $t_p$ | e | $t_p$ | e | $t_p$ | e | $t_p$ | e | $t_p$ | e |
| 200 | 0.43 | 0.45 | 48% | 0.48 | 22% | 0.58 | 9% | 1.27 | 2% | 1.72 | 1% |
| 1000 | 15.32 | 16.46 | 47% | 16.43 | 23% | 16.16 | 12% | 16.88 | 6% | 17.32 | 3% |
| 5000 | 527.69 | 655.30 | 40% | 672.91 | 20% | 666.82 | 10% | 792.03 | 4% | 844.34 | 2% |
| 10000 | 2163.48 | 2589.30 | 42% | 2716.85 | 20% | 3445.72 | 8% | 2660.50 | 5% | 3698.69 | 2% |

Table 10. A case without load balancing,
computing time and efficiency for a gap 10 and range 1000

## Acknowledgments

Part of this study has been made possible by a grant of CINES, Montpellier, France. The authors wish to thank also the referees for their helpful comments.

## References

[1] R. Andonov and F. Gruau, A linear systolic array for the knapsack problem, *Proceedings of the International Conference on Application Specific Array Processors* **23**, 458-472 (1991).

[2] R. Andonov and P. Quinton, Efficient linear systolic array for the knapsack problem, *in L. Bougé, M. Cosnard, Y Robert and D. Trystram editors Parallel Processing: CONPAR 92 - VAPP V, Lecture Notes in Computer Science, Springer Verlag* **634**, 247-258 (1992).

[3] J. H. Ahrens and G. Finke, Merging and Sorting Applied to the Zero-One Knapsack Problem, *Operations Research* **23**, 1099-1109 (1975).

[4] H. Bouzoufi and S. Boulenouar and R. Andonov, Pavage optimal pour des dépendances de type sac à dos, *Actes du Colloque RenPar'10* (1998).

[5] J. Casti and M. Richardson and R. Larson, Dynamic programming and Parallel Computers, *Journal of Optimization Theory and Applications* **12**, 423-438 (1973).

[6] H. K. C. Chang and J. J. R. Chen and S. J. Shyu, A parallel algorithm for the knapsack problem using a generation and searching technique, *Parallel Computing*, **20** 233-243 (1994).

[7] G. H. Chen and M. S. Chern and J. H. Jang, Pipeline Architectures for Dynamic Programming Algorithms, *Parallel Computing* **13**, 111-117 (1990).

[8] M. Cosnard and A. G. Ferreira and H. Herbelin, The two lists algorithm for the knapsack problem, *Parallel Computing* **9**, 385-388 (1989).

[9] G. Cybenko, Dynamic load balancing for distributed-memory multiprocessors *Journal of Parallel an Distributed Computing* **7**, 279-301 (1989).

[10] M. Elkihel, Programmation dynamique et rotation de contraintes pour les problèmes d'optimisation entière, *Thèse de Doctorat, Université des Sciences et Techniques de Lille*, (1984).

[11] M. Elkihel and D. El Baz, An efficient dynamic programming parallel algorithm for the 0-1 knapsack problem, in Joubert et al. (eds) *Parallel Computing Advances and Current Issues* Imperial College Press, London, 298-305 (2002).

[12] D. Fayard and G. Plateau, Resolution of the 0-1 Knapsack Problem: Comparison of methods, *Mathematical Programming* **8**, 272-307 (1975).

[13] T. E. Gerash and P. Y. Wang, A survey of parallel algorithms for one-dimensional integer knapsack problems, *INFOR* **32**, 163-186 (1994).

[14] E. Horowitz and S. Sahni, Computing partitions with application to the knapsack problems, *Journal of the Association for Computing Machinery* **21**, 275-292 (1974).

[15] G. Kindervater and J.K. Lenstra, An introduction to parallelism in combinatorial optimization, *Discrete and Applied Mathematics*, **14** 135-156 (1986).

[16] G. Kohring, Dynamic load balancing for parallelized particle simulations on MIMD computers *Parallel Computing*, **21** 638-693 (1995).

[17] M. Lauriere, An Algorithm for the 0-1 knapsack problem, *Mathematical Programming* **14**, 1-10 (1978).

[18] D. C. Lou and C. C. Chang, A parallel two list algorithm for the knapsack problem, *Parallel Computing*, **22**, 1985-1996 (1997).

[19] S. Martello and P. Toth, An Upper bound for the Zero One Knapsack Problem and a Branch and Bound Algorithm, *European Journal of Operational Research* **1**, 169-175 (1977).

[20] S. Martello and P. Toth, Algorithm for the Solution of the 0-1 single Knapsack Problem, *Computing* **21**, 81-86 (1978).

[21] S. Martello and D. Pisinger and P. Toth, Dynamic programming and strong bounds for the 0-1 knapsack problem, *Management Science* 45, 414-424 (1999).

[22] L. Oliker and R. Biswas, PLUM: Parallel load balancing for adaptive unstructured meshes, *Journal of Parallel and Distributed Computing* 52, 150-157 (1998).

[23] U. Pferschy, Dynamic programming revisited: Improving knapsack algorithms, *Computing* 63, 419-430 (1999).

[24] D. Pisinger, A minimal algorithm for the 0-1 knapsack problem, *Operations Research* **45**, 758-767 (1997).

[25] D. Pisinger and P. Toth, Knapsack problems, in D. Z. Du and P. Pardalos (eds) *Handbook of Combinatorial Optimization*, Kluwer, 1-89 (1998).

[26] G. Plateau and M. Elkihel, A Hybrid Method for the 0-1 Knapsack Problem, *Methods of Operations Research* **49**, 277-293 (1985).

[27] P. Toth, Dynamic Programming Algorithms for the Zero-One Knapsack Problem, *Computing* **25**, 29-45 (1980).