Load balancing in a parallel dynamic programming multi-method applied to the 0-1 knapsack problem

Moussa Elkihel, Didier El Baz

Load balancing in a parallel dynamic programming multi-method applied to the 0-1 knapsack problem

Moussa Elkihel Didier El Baz
LAAS-CNRS
7, avenue du Colonel Roche, 31077 Toulouse CEDEX 4, France
elkihel@laas.fr elbaz@laas.fr

Abstract

The 0-1 knapsack problem is considered. A parallel dynamic programming multi-method using dominance technique and processor cooperation is proposed. Different load balancing approaches are studied. Computational experiments carried out on an Origin 3800 supercomputer are reported and analyzed.

1 Introduction

In this paper, we propose an efficient parallel dynamic programming multi-method using dominance technique and processor cooperation. This method is applied to a large class of integer programming problems: the 0-1 knapsack problem, which has many applications in operation research (see [27]). More precisely, we concentrate on an approach proposed by Ahrens and Finke (see [3]) which presents the advantage to limit the number of states using dominance techniques. The number of states, or undominated pairs, is unforeseeable. The main drawback of this approach is that it creates irregular data structures. As a consequence, the parallelization of this method is not easy and the design of an efficient load balancing strategy is very important.

In [11], a parallelization of the one list dynamic programming method using dominance technique has been proposed. Three load balancing approaches have also been proposed in [11]. In this paper, we present an original parallel dynamic programming multi-method which combines two load balancing techniques studied in [11].

We note that 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 proposed solution for arrays with up to $O(2^{\frac{n}{5}})$ 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 and presents better time complexity than the parallel algorithm studied in [8]. Reference is also made to [7] and [13], for different approaches concerning the parallelisation of the dynamic programming method.

Section II deals with the 0-1 knapsack problem and its solution via dynamic programming. A first parallel algorithm is presented in Section III. Load balancing techniques and the parallel multi-method are studied in Section IV. Computational results carried out on an Origin 3800 supercomputer are displayed and analyzed in Section V.

2 The 0-1 Knapsack Problem

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

$$\max \left\{ \sum_{j=1}^{n} p_j x_j | \sum_{j=1}^{n} w_j x_j \leq C, x_j \in \{0,1\} j = 1, \ldots, n \right\} \quad (1)$$

where C denotes the capacity of the knapsack, n the number of items, 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 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 $\mathcal{O}(\min\{2^n, nC\})$ (see [3]) and which is based on the concepts of list and dominance. We shall generate recursively lists L_k of pairs (w, p), $k = 1, 2, \ldots, n$; where w is a weight and p a profit. According to the dominance principle, all states (w, p) such that there exists a state (w', p') which satisfies: $w' \leq w$ and $p \leq p'$, are dominated. Initially, we have $L_0 = \{(0, 0)\}$. 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. The reader is referred to [11] for more details on the generation of lists in the sequential case.

3 Basic Parallel Algorithm

This Section deals with parallelization of the one list dynamic programming method using dominance technique. The parallel algorithm which is presented in detail 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. The main feature of the 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 a 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 processors to remove all dominated pairs from their sublists.

3.1 Initialization, Work Decomposition and Task Assignment

Initialization of parallel algorithm is performed by a sequential dynamic programming algorithm using dominance technique. First, 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 partitionned 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 cardinality 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 still belong to the sublists L_k^i, which will induce finally an overhead. Thus, in the beginning of each stage, it is necessary to synchronize all processors and to make them share part of the data produced at the previous stage in order to discard globally all dominated pairs.

3.2 Parallel processes

We give now some details on the parallel algorithm. We present first the construction process 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 is denoted by $(w_k^{i,0}, p_k^{i,0})$. The various sublists are defined as follows. For all $i = 0, ..., q-2$,

$$N_k^i = \{(w_{k-1}^{i,\cdot} + w_k, p_{k-1}^{i,\cdot} + p_k) \mid (w_{k-1}^{i,\cdot}, p_{k-1}^{i,\cdot}) \in L_{k-1}^i, w_{k-1}^{i,\cdot} + w_k < w_{k-1}^{i+1,0}\}, \quad (2)$$

$$N_k^{q-1} = \{(w_{k-1}^{q-1,\cdot} + w_k, p_{k-1}^{q-1,\cdot} + p_k) \mid (w_{k-1}^{q-1,\cdot}, p_{k-1}^{q-1,\cdot}) \in L_{k-1}^{q-1}, w_{k-1}^{q-1,\cdot} + w_k \leq C\}. \quad (3)$$

The sublist N_k^i corresponds to the list of new pairs created at stage k by processor E^i from the sublist L_{k-1}^i and the k-th item, which is 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 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 $i \in \{1, ..., q-2\}$,

$$C_k^i = \{(w_{k-1}^{j,\cdot} + w_k, p_{k-1}^{j,\cdot} + p_k) \mid (w_{k-1}^{j,\cdot}, p_{k-1}^{j,\cdot}) \in L_{k-1}^j, j < i, w_{k-1}^{i,0} \leq w_{k-1}^{j,\cdot} + w_k < w_{k-1}^{i+1,0} \text{ or } w_{k-1}^{j,\cdot} + w_k < w_{k-1}^{i,0}, p_{k-1}^{j,\cdot} + p_k \geq p_{k-1}^{i,0}\}, \quad (4)$$

$$C_k^0 = \emptyset, \tag{5}$$

$$\begin{aligned} C_k^{q-1} = \{&(w_{k-1}^{j,\cdot} + w_k, p_{k-1}^{j,\cdot} + p_k) \mid \\ &(w_{k-1}^{j,\cdot}, p_{k-1}^{j,\cdot}) \in L_{k-1}^j, j < q-1, \\ &w_{k-1}^{q-1,0} \leq w_{k-1}^{j,\cdot} + w_k < C \text{ or} \\ &w_{k-1}^{j,\cdot} + w_k < w_{k-1}^{q-1,0}, p_{k-1}^{j,\cdot} + p_k \geq p_{k-1}^{q-1,0}\}. \end{aligned} \tag{6}$$

The sublist C_k^i is the set of pairs that are exchanged between all processors E^m, with $m < i$ and processor E^i, either 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. 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 in particular dominated pairs, we introduce the series of sets D_k^i. For all $i = 0, ..., q-2$,

$$D_k^i = \hat{D}_k^i \cup \{(w,p) \mid w < w_{k-1}^{i+1,0} \text{ and } p \geq p_{k-1}^{i+1,0}\}, \tag{7}$$

where

$$\begin{aligned} \hat{D}_k^i = \quad &\{(w,p) \mid (w,p) \in L_{k-1}^i \cup N_k^i \cup C_k^i \text{ and} \\ &\exists (w',p') \in L_{k-1}^i \cup N_k^i \cup C_k^i, \\ &(w',p') \neq (w,p) \text{ and } w' \leq w, p \leq p'\}, \end{aligned} \tag{8}$$

$$\begin{aligned} D_k^{q-1} = \quad &\{(w,p) \mid (w,p) \in L_{k-1}^{q-1} \cup N_k^{q-1} \cup C_k^{q-1}, \\ &\exists (w',p') \in L_{k-1}^{q-1} \cup N_k^{q-1} \cup C_k^{q-1}, \\ &(w',p') \neq (w,p), w' \leq w, p \leq p'\}, \end{aligned} \tag{9}$$

We note that $\hat{D}_k^i$ is the subset of D_k^i which contains all dominated pairs of 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 belong to different processors. These copies permit processors to check dominance relationship. 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 $i = 0, ..., q-1$,

$$L_k^i = L_{k-1}^i \cup N_k^i \cup C_k^i - D_k^i. \tag{10}$$

Correctness of the parallel algorithm is studied in [11].

4 Load Balancing

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 some processors, e.g. E^{q-1}, can become overloaded.

In this Section, we propose a multi-method that combines two load balancing strategies which are designed in order to obtain a good efficiency. When using load balancing, the time complexity of the parallel algorithm presented in Section 3 is $\mathcal{O}(\min\{\frac{2^n}{q}, \frac{nC}{q}\})$, since pairs will be fairly distributed on all processors. On one processor, space complexity of the parallel algorithm is $\mathcal{O}(\frac{C}{q})$, if the solution vector is stored as a word of several bytes. In this case, the value of component x_j at the solution is given by the value of the j-th bit of the word (either 0 or 1).

4.1 Dynamic load balancing (DLB)

We present now a load balancing strategy that is called dynamic since load balancing decision is taken at each stage. This strategy relies on a load balancing test that is based upon a comparison of the work needed for performing the load balancing on the one hand and the extra 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 work resulting from load unbalancing costs more than the load balancing work, then loads are balanced, otherwise they are unbalanced. The load balancing process will assign fairly loads to processors, i.e. it tends to assign the same 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 as

$$c_u = T_p.(N_l - \frac{N_t}{q}). \tag{11}$$

The load balancing cost, denoted by c_b, is given as follows

$$c_b = N_l.(T_r + T_w), \tag{12}$$

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 loads are balanced, else the loads are not balanced. The test can be rewritten as follows

$$N_l - \frac{N_t}{q} > N_l.\frac{(T_r + T_w)}{T_p}, \tag{13}$$

$$1 - \frac{N_t}{q.N_l} > \frac{(T_r + T_w)}{T_p}, \tag{14}$$

$$1 - \frac{(T_r + T_w)}{T_p} > \frac{N_t}{q.N_l}. \quad (15)$$

In the next Section, we shall 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.

$$T_p = 2.69.10^{-7} \text{ s}, T_r = 4.2.10^{-8} \text{ s and } T_w = 3.6.10^{-8} \text{ s}. \quad (16)$$

Thus, for this machine we have

$$\frac{(T_r + T_w)}{T_p} = 0.29, \quad (17)$$

and the practical test is given as follows.

$$0.71 > \frac{N_t}{N_l.q}. \quad (18)$$

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 (13) is different from the test used in [22] (reference is also made to [9] and [16]).

4.2 Implicit load balancing (ILB)

The implicit load balancing technique has been designed with a specific version of the parallel dynamic programming list method in order to obtain fair and natural load balancing as well as a very small overhead. The principle of this technique 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 that pairs must be assigned to processors merely according to the value of their weight. For example pairs with weight between 0 and $\lfloor \frac{C}{q} \rfloor$, where q denotes the number of processors, must be assigned to processor E^0. Similarly, pairs with weight between $\lfloor \frac{C}{q} \rfloor + 1$ and $2.\lfloor \frac{C}{q} \rfloor$ must be assigned to processor E^1 and so on. We use a parallel algorithm which is slightly different from the one presented in Section III. This algorithm still relies on dynamic programming and dominance techniques. However, the parallel algorithm does not necessitate any given initialization. Another difference with the parallel algorithm presented in Section III, is that for all $k \in \{1, ..., n\}$, the weight of the smallest pair of L_k^i in both weight and profit, denoted by $w_k^{i,0}$, if it exists, is the weight w of the smallest pair $(w, p) \in L_k$ which satisfies: $i.\lfloor \frac{C}{q} \rfloor + 1 \leq w \leq (i+1).\lfloor \frac{C}{q} \rfloor$, for all $i \in \{1, ..., q-2\}$; $0 \leq w \leq \lfloor \frac{C}{q} \rfloor$, for $i = 0$ and $(q-1).\lfloor \frac{C}{q} \rfloor + 1 \leq w \leq C$, for $i = q-1$.

The main advantage of this approach is that loads tend to be balanced in a fair and implicit way among all processors. As a consequence, the overhead is very small. On the other hand, the main drawback of this technique seems to be its poor efficiency at initialization, since some processors, e.g. E^{q-1}, may be idle for a while. However, we will see in the next Section that this approach is performant even if the number of processors is large. As a matter of fact, the load of the different processors tends to be very well balanced in working regime since the number of pairs assigned to the different processors tends to be very similar and constant.

4.3 Multi-method (M)

We propose now a parallel dynamic programming multi-method which combines the advantages of dynamic and implicit load balancing. More precisely, we shall use dynamic load balancing in the beginning of the treatment since it must perform better than implicit load balancing. We recall that in the beginning, some processors may be inactive with the implicit load balancing approach while all processors are active with dynamic load balancing. Then, we shall end treatment with the implicit load balancing approach, since it presents a small overhead, contrarily to dynamic load balancing.

The major questions at this point are: at what time must we swap methods and what is the expected gain in combining dynamic and implicit load balancing as compared with both methods taken solely?

We propose an automatic procedure which swaps methods. This procedure is based on the fact that the implicit load balancing approach will be very efficient when total work will be shared by all processors; i.e. when processor E^{q-1} will start to be assigned pairs with weight between $(q-1)\lfloor \frac{C}{q} \rfloor + 1$ and C. In the sequel $(w_k^{q-1,\omega}, p_k^{q-1,\omega})$ will denote the largest pair of L_k^{q-1} in both weight and profit. The following algorithm shows how multi-method works.

$k = k(0) + 1$
WHILE $w_k^{q-1,\omega} \leq (q-1)\lfloor \frac{C}{q} \rfloor$ and $k \leq n$
DO
  BEGIN
  Run a stage of the parallel algorithm with DLB
  $k = k + 1$
  END
IF $k \leq n$
THEN
  DO
    Assign loads to processors according to ILB
    FOR $j = k$ TO n
      BEGIN
      Run a stage of the parallel algorithm using ILB
      END

We note that in some cases the multi-method may only implement dynamic load balancing. This is particularly true

if the number of processors is large; in that case $(q-1)\lfloor\frac{C}{q}\rfloor$ may be very close to C and there is no guarantee that even for a large k there exists a pair with weight $w_k^{q-1,\omega}$ greater than $(q-1)\lfloor\frac{C}{q}\rfloor$. We note also that the multi-method swaps very fast methods using dynamic and implicit load balancing with regards to the total processing time. The swaping cost corresponds merely to a reassignment of pairs in order to match the intervals defined in the beginning of subsection IV B; thus, this cost is similar to a typical load balancing cost.

5 Computational Experiments

Numerical experiments presented in this Section correspond to weakly correlated problems. These problems can be considered as difficult (see [10]). We have avoided treating noncorrelated problems which induce a large number of dominated pairs and which are easier to solve.

The various instances considered are relative to problems with 200, 400, 1000 and 5000 variables, respectively, with data range defined as follows. The weights w_j are randomly distributed in the interval $[1, 1000]$ and the non-negative profits p_j in the interval $[w_j - g, w_j + g]$, where the gap, denoted by g, is equal to 100. We have considered problems with $C = 0.5\sum_{j=1}^{n} w_j$, which correspond in general to difficult instances.

Parallel algorithms have 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 value. 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 up to 8.10^7 pairs and have noted a constant 4.10^{-8} seconds read time and write time.

Numerical experiments have been carried out with up to 32 processors. Numerical results are displayed on Table 1 for the multi-method and parallel dynamic programming algorithms using dynamic and implicit load balancing, respectively, which are denoted by M, DLB and ILB, respectively. Average running time in seconds for twenty instances are reported for sequential algorithms (they are denoted by t_s). Experimental results are given for 4, 8, 16 and 32 processors. Average efficiency of parallel algorithms is also shown. We note from Table 1 that the efficiency of a parallel algorithm is function of several parameters such as the size of the problem or the number of processors. We note also that the efficiency of a parallel algorithm tends generally to decrease when the number of processors increases. Basically, the granularity, i.e. the ratio computation time over communication time, tends to decrease when q increases, since communications play a major part in this case.

Parallel algorithms with dynamic or implicit load balancing strategies present in general a good efficiency for a coarse granularity. The performance of methods with implicit load balancing and dynamic load balancing are quite similar, although the sophisticated dynamic load balancing strategy seems a little bit more performant than implicit load balancing for large problems and large number of processors. We note also that the performance of the multi-method is in general better than the one of parallel dynamic programming algorithms using dynamic or implicit load balancing for large problems. This shows the interest of our approach. We note also that in some cases, 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 data of the problem.

q	1	4	8	16	32
Size Met.	t_s	e	e	e	e
200 M	0.116	0.52	0.31	0.12	0.04
DLB		0.45	0.3	0.12	0.03
ILB		0.49	0.35	0.21	0.09
400 M	0.775	0.69	0.54	0.32	0.13
DLB		0.60	0.48	0.32	0.11
ILB		0.60	0.52	0.4	0.22
1000 M	7.935	0.77	0.66	0.54	0.36
DLB		0.63	0.59	0.51	0.33
ILB		0.64	0.59	0.52	0.39
5000 M	463.48	1.12	0.91	0.86	0.82
DLB		0.82	0.85	0.86	0.79
ILB		0.89	0.86	0.83	0.77

Table 1. Computing time and efficiency

Acknowledgments

This study has been made possible by a grant of CINES, Montpellier, France.

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, et al. 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] D. El Baz and M. Elkihel, Load balancing methods and parallel dynamic programming algorithm using dominance technique applied to th 0-1 knapsack problem *Journal of Parallel and Distributed Computing* **65**, 74-84 (2005).

[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] L. Wolsey, *Integer Programming,* John Wiley & Sons, New York (1998).