AN EFFICIENT DYNAMIC PROGRAMMING PARALLEL ALGORITHM FOR THE 0-1 KNAPSACK PROBLEM

M. ELKIHEL AND D. EL BAZ
LAAS du CNRS, 7, avenue du Colonel Roche, Toulouse 31077, France
E-mail: elkihel@laas.fr elbaz@laas.fr

An efficient parallel algorithm for the 0-1 knapsack problem is presented. The algorithm is based on dynamic programming in conjunction with dominance of states techniques. An original load balancing strategy which is designed in order to achieve good performance of the parallel algorithm is proposed. To the best of our knowledge, this is the first time for which a load balancing strategy is considered for this type of algorithm. Finally, computational results obtained with an Origin 2000 supercomputer are displayed and analyzed.

1 Introduction

In this paper, we consider the parallel solution of the 0-1 knapsack problem. The 0-1 knapsack problem is a combinatorial optimization problem which is used to model many industrial situations such as, for example: cargo loading, assignation problems, capital budgeting, or cutting stocks. Moreover, this problem occurs as a subproblem to be solved when more complex optimization problems are solved. This problem is well known to be NP-hard.

The 0-1 knapsack problem has been intensively studied in the past (see Horowitz and Sahni [7], Ahrens and Finke [1], Fayard and Plateau [5], Martello and Toth [9] and Plateau and Elkihel [11]). More recently, several authors have proposed parallel algorithms for the solution of this combinatorial problem.

Cosnard and Ferreira [4] have presented a parallel implementation of the two lists dynamic programming algorithm on a MIMD architecture for the solution of the exact subset sum knapsack problem. We note that data exchanges are not needed for this type of application.

Other interesting parallel approaches for the solution of dense problems which are based on dynamic programming have been proposed. We can quote for example Chen, Chern and Jang [3] for a method carried out on pipeline architectures and Bouzoufi, Boulenouar and Andonov [2] for an efficient parallel algorithm using tiling techniques on MIMD architectures.

The reader is referred to Gerash and Wang [6] for a survey of parallel algorithms for the 0-1 knapsack problem.

In this paper, we propose an efficient parallel algorithm for the 0-1 knapsack problem. The algorithm is based on dynamic programming in conjunction with dominance of states techniques. We present also an original load balancing strategy which is designed in order to achieve good performance of the parallel algorithm studied. To the best of our knowledge, this is the first time for which a load balancing strategy is proposed for this type of algorithm. Finally, computational results obtained with an Origin 2000 supercomputer are displayed and analyzed.

Section 2 deals with the 0-1 knapsack problem and its solution via dynamic programming. An original parallel algorithm and associated load balancing strategy are presented in Section 3. Finally, computational results obtained with an Origin 2000 supercomputer are displayed and analyzed in Section 4.

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 total 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 moreover 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 methods based on dynamic programming studied by Horowitz and Sahni [7], Ahrens and Finke [1] and Toth [12], branch and bound methods proposed by Fayard and Plateau [5], Lauriere [8], Martello and Toth [9] and finally mixed algorithms combining dynamic programming and branch and bound methods studied by Martello and Toth [10] and Plateau and Elkihel [11].

In this paper, we concentrate on a dynamic programming procedure (see Ahrens and Finke [1]) which is based on the concepts of list and dominance. For each positive integer k, we define the dynamic programming recursive list L_k as the following set of monotone increasing ordered pairs :

$$L_k = \{(w,p) \mid w \leq C \text{ and } w = \sum_{j=1}^{k} w_j x_j, \; p = \sum_{j=1}^{k} p_j x_j\}. \tag{2}$$

According to the dominance principle, all states (w, p) such that there exists a state (w', p') which satisfies: $w' < w$ and $p < p'$, must be removed from the list L_k. In this case, we usually say that the state (w', p') dominates the state (w, p). As a consequence, we note that by using the dominance concept, any two pairs $(w, p), (w', p')$ of the list L_k must satisfy: $p < p'$ if $w < w'$. The following sequential algorithm permits one to build recursively the list L_n from which we can derive the optimal solution of problem (1).

Sequential Algorithm

$$\begin{array}{lll} L_0 & = & \{(0,0)\} \\ \text{FOR} & \multicolumn{2}{l}{k = 1 \text{ TO } n \text{ DO}} \\ L_k & = & \{(w + w_k x_k, p + p_k x_k) \mid (w,p) \in L_{k-1} \text{ and } w + w_k x_k \leq C, \\ & & \text{with } x_k \in \{0,1\}\} - D_k \end{array} \tag{3}$$

where D_k denotes the set of all dominated pairs at stage k.

3 Parallel Algorithm

In this Section, we present an original parallel algorithm based on the dynamic programming method. The main feature of this parallel algorithm is that all processors cooperate via data exchange to the construction of the list. In particular, all dominated pairs are removed from the list at each stage of the parallel dynamic programming method. More precisely, each processor creates its own part of the global list L_k at each stage k; so that the total work is shared by the different processors and data exchange permits each processor to remove all dominated pairs.

Several issues must be addressed when considering the design of a parallel algorithm: the initialization of the parallel algorithm, the work decomposition and tasks assignation and the load balancing strategy.

3.1 Initialization, Work Decomposition and Task Assignation

The initialization of the parallel algorithm is sequential. 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 be the ordered list which results from the sequential initialization. The list L is partitionned as follows: $L = L_1 \cup ... \cup L_q$, with $\mid L_i \mid = l$, for $i = 1, ..., q-1$ and $\mid L_q \mid \geq l$, where $\mid L_i \mid$ denotes the number of elements of the sublist L_i. For simplicity of presentation, L_i will denote in the sequel the sublist assigned to processor E_i.

We note that if the different processors E_i work independently on their list L_i without sharing any data, then at the global level some dominated pairs may not be removed from the local sublists; which induces an overhead. Thus, it is necessary to synchronize the processors at each iteration and share part of the data produced at the previous iteration in order to eliminate all the dominated pairs from a global point of view. These operations may reduce the size of the sublists L_i and the size of the union of the sublists will be the same as if the list was generated by a sequential algorithm.

We detail now the parallel algorithm. In the case where the iteration number k is odd, data exchange occurs only between any processor E_i and processors $E_{i+1}, ..., E_q$. In the case where k is even data exchange occurs between any processor E_i and processors $E_1, ..., E_{i-1}$, for natural load balancing.

We note that for simplicity of presentation, we consider here only the case where the iteration number k is odd. In the following parallel algorithm we present computation and communication at iteration k. We shall denote by L_i the sublist associated with processor E_i and the smallest pair of L_i will be denoted by $(w_{i,1}, p_{i,1})$.

Parallel Algorithm

FOR $i = 1$ TO q DO
  BEGIN
    IF $i \neq 1$
    THEN
      Processor E_i copies $(w_{i,1}, p_{i,1})$ in the shared memory;
    IF $i \neq q$
    THEN
      CASE
        The pair $(w_{i+1,1}, p_{i+1,1})$ is available
          Processor E_i copies the following pairs in the shared memory:
          $\{(w_{i,j}, p_{i,j}) \mid w_{i,j} + w_k \geq w_{i+1,1}\}$
          Processor E_i merges the list L_i with all pairs $(w_{j,.} + w_k, p_{j,.} + p_k)$
          which are such that $w_{i,1} - w_k \leq w_{j,.} < w_{i+1,1} - w_k$ and $j \leq i$
          All dominated pairs are removed;
    IF $i = q$
    THEN
      Processor E_i merges the list L_i with all pairs $(w_{j,.} + w_k, p_{j,.} + p_k)$
      which are such that $w_{i,1} - w_k \leq w_{j,.} \leq C$ and $j \leq i$
      All dominated pairs are removed;
  END

We note that the copy of $(w_{i,1}, p_{i,1})$ stored in the shared memory by processor E_i will be used by processors $E_1, ..., E_{i-1}$, in order to know what part of its sublist must be copied in the memory for the purpose of removing dominated pairs.

We have chosen to exchange data between any processor E_i and processors $E_{i+1}, ..., E_q$ every odd iteration and between any processor E_i and processors $E_1, ..., E_{i-1}$ every even iteration. This strategy is designed in order to realize a kind of simple and natural load balancing. With this type of data exchange, two processors, say E_1 and E_q, play a particular part since, generally, they tend to accumulate more pairs than other processors. In this sense, processors E_1 and E_q contain valuable information that will permit one to design an efficient load balancing strategy as we shall see in the next subsection.

3.2 Load Balancing Strategy

In order to obtain a good performance of the parallel algorithm, 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 from the merging process described in the Parallel Algorithm of the previous Section that processors E_1 and E_q become often overloaded.

We propose a load balancing strategy which is designed in order to obtain a good efficiency while presenting a small overhead. As a consequence, the strategy considered does not balance loads at each iteration; it is rather an adaptive strategy which takes into account several measures which are presented in the sequel.

The strategy chosen is such that a load balancing is made if a typical processor which is overloaded can take benefit of it. The main test is made systematically on processor E_1 since we have chosen to take a decision every two iterations and processor E_1 can become overloaded every odd iterations according to the Parallel Algorithm presented in Section 2. The load balancing process will assign an equal load i.e. an equal number of pairs to all processors.

In the sequel, O_k, O^e_{k+2}, respectively, will denote a measure of the load balancing overhead at iteration k, and an estimation of the load balancing overhead at iteration $k+2$, respectively. O_k, is function of the total number of pairs in the global list and O^e_{k+2} is estimated in function of the augmentation of pairs ratios per processor. Similarly, T_k, T^e_{k+1}, T^e_{k+2}, respectively, denote the computation time of processor E_1 at iteration k, and estimations of the computation time of processor E_1 at iterations $k+1$ and $k+2$, respectively. These estimations are computed according to:

$$T^e_{k+1} = rT_k \text{ and } T^e_{k+2} = r^2 T_k, \tag{4}$$

where the ratio r satisfies:

$$r = \frac{T_k}{T_{k-1}}. \tag{5}$$

More generally, we shall use the following notations:

$$T_p(k) = \sum_{i=0}^{p-1} T_{k-i} + O_k, \tag{6}$$

$$T_{p,2}(k) = \sum_{i=0}^{p-1} T_{k-i} + T^e_{k+1} + T^e_{k+2} + O^e_{k+2}. \tag{7}$$

We note that iteration $k-p$ is relative to the last load balancing phase. The condition for load balancing will be:

$$\frac{T_p(k)}{p} < \frac{T_{p,2}(k)}{p+2}. \tag{8}$$

If this condition is satisfied, then all the benefit of the load balancing will be for processor E_1, and as a consequence for the parallel machine (load balancing is then made at the end of iteration k); else, there is no load balancing. So, load balancing is made according to the following algorithm:

Load Balancing Algorithm

$p = 0$

FOR $k = k(0) + 2$ TO $n - 2$, 2 DO

  BEGIN

    compute $\frac{T_p(k)}{p}$ and $\frac{T_{p,2}(k)}{p+2}$

    IF $\frac{T_p(k)}{p} < \frac{T_{p,2}(k)}{p+2}$

    THEN

      perform load balancing

      $p = 0$

    ELSE

      $p = p + 2$

  END

Table 1. Computing time in seconds and efficiency for a gap 100 and range 10000.

	1 processor	*2 processors*		*4 processors*		*8 processors*	
size	time	time	efficiency	time	efficiency	time	efficiency
200	4.42	2.58	86%	2.04	64%	1.25	43%
400	33.36	18.46	90%	11.88	70%	9.10	46%
600	107.90	56.89	95%	34.19	79%	23.02	59%

Table 2. Computing time in seconds and efficiency for a gap 10 and range 10000.

	1 processor	*2 processors*		*4 processors*		*8 processors*	
size	time	time	efficiency	time	efficiency	time	efficiency
200	19.92	10.18	98%	6.91	72%	5.11	49%
400	114.68	60.69	94%	34.84	82%	21.01	68%
600	300.29	164.54	91%	89.36	84%	58.70	64%

4 Numerical Results

The numerical experiments presented here are relative to difficult 0-1 knapsack problems. It is well known that the number of variables is not the fundamental criterion for difficulty. We have studied several series of strongly coupled problems which are considered as difficult problems. The numerous instances considered are relative to problem sizes which are equal to 200, 400, 600, respectively, with data range defined as follows. The weights w_j are randomly distributed in the segment $[1, 10000]$ and the profits p_j in $[w_j - g, w_j + g]$, where the gap, denoted by g, is equal to 100 for the first set of data and 10 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 on a NUMA shared memory supercomputer Origin 2000 by using the Open MP environment. Parallel numerical experiments have been carried out with up to 8 processors. Numerical results are displayed on Tables 1 and 2, where the average running time in seconds and efficiency are given at each time for over 20 instances randomly generated of knapsack problems.

From Tables 1 and 2, we note that the efficiency of parallel algorithms generally increases with the size of the problem. We note also that the efficiency increases with the difficulty of the combinatorial problem: parallel algorithms are generally more efficient for problems with a smaller gap. We see that the efficiency decreases when the number of processors increases. However, the efficiency remains good; which shows that the load balancing strategy is efficient.

Acknowledgments

Part of this study has been made possible by a support of CALMIP, Toulouse and CINES, Montpellier.

References

1. J. H. Ahrens and G. Finke, *Journal of the Association for Computing Machinery* **23**, 1099 (1975).
2. H. Bouzoufi and S. Boulenouar and R. Andonov *Actes du Colloque RenPar'10* (1998).
3. G. H. Chen and M. S. Chern and J. H. Jang, *Parallel Computing* **13**, 111 (1990).
4. M. Cosnard and A. G. Ferreira, *Parallel Computing* **9**, 385 (1989).
5. D. Fayard and G. Plateau, *Mathematical Programming* **8**, 272 (1975). GerWan94
6. T. E. Gerash and P. Y. Wang, *INFOR* **32**, 163 (1994).
7. E. Horowitz and S. Sahni, *Journal of the Association for Computing Machinery* **21**, 275 (1974).
8. M. Lauriere, *Mathematical Programming* **14**, 1 (1978).
9. S. Martello and P. Toth, *European Journal of Operational Research* **1**, 169 (1977).
10. S. Martello and P. Toth, *Computing* **21**, 81 (1978).
11. G. Plateau and M. Elkihel, *Methods of Operations Research* **49**, 277 (1985).
12. P. Toth, *Computing* **25**, 29 (1980).