Optimal Parallel Algorithm for the Knapsack Problem Without Memory Conflicts

Kenli LI *Qinghua LI* *Wang-Hui* *Shengyi JIANG*
School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, 430074,China
National High Performance Computing Center (Wuhan), 430074,China
E-mail: LKL510@263.net

Abstract- The knapsack problem is very important in cryptosystem and in number theory. This paper proposes a new parallel algorithm for the knapsack problem where the method of divide and conquer is adopted. Basing on an *EREW-SIMD* machine with shared memory, the proposed algorithm utilizes $O(2^{n/4})^{1-\varepsilon}$ processors, $0 \le \varepsilon \le 1$, and $O(2^n)$ memory to find a solution for the n-element knapsack problem in time $O(2^{n/4} (2^{n/4})^{\varepsilon})$. Thus the cost of the proposed parallel algorithm is $O(2^n)$, which is optimal, and an improved result over the past researches.

Keywords: Knapsack problem, parallel algorithm, optimal algorithm, memory conflicts

I. Introduction

Given n positive integers $W = (w_1, w_2, \ldots, w_n)$ and a positive integer M, the knapsack problem (also called the subset sum problem by some authors) is the decision problem of finding a binary n-tuple $X = (x_1, x_2, \ldots, x_n)$ that solves the equation

$$\sum_{i=1}^{n} w_i x_i = M, x_i \in \{0, 1\}, \quad (1)$$

This problem was proved to be NP-complete[1]. Solving the knapsack problem can be seen as a way to study some large problems in number theory and, because of its exponential complexity, some public-key cryptosystem are based on it[2-4]. Branch and Bound algorithms were proposed[5] and they reach good performance for particular instances of the problem, but the worst case complexity is still $O(2^n)$. A major improvement in this area was made by Horowitz and Sahni[5], who drastically reduced the time needed to solve the knapsack problem by conceiving a clear algorithm in $O(n2^{n/2})$ time and $O(2^{n/2})$ space. It is known as the *two-list* algorithm. Based on this algorithm, Schrowppel and Shamir[6] reduced the memory requirements with the *two-list four-table* algorithm which needs $O(2^{n/4})$ memory space to solve the problem in still $O(n2^{n/2})$ time. In addition, both the time and space complexity of the *two-list* algorithm can be made to be $O(2^{n/2})$ by using the method of merging[7]. Although the above algorithm is by far the most efficient algorithm to solve the knapsack problem in sequential, it means nothing for any instances where the size n is great.

With the advent of the parallelism, much effort has been done in order to reduce the computation time of problems in all research areas[7-14]. However all the existing parallel algorithms were designed for the *concurrent read exclusive write* (*CREW*) *single instruction multiple data* (*SIMD*) model of parallel computation with shared memory, and to our knowledge, there is no optimal parallel algorithm for the knapsack problem even in *CREW* model of parallel computation.

In this paper, we propose a new parallel algorithm for the knapsack problem. The design of the proposed algorithm is based on an *EREW-SIMD* model of parallel computing with shared memory. Each processor holds a constant memory space, while there are $O(2^{n/2})$ memory cells in the shared memory. Our algorithm needs $p = O((2^{n/4})^{1-\varepsilon})$ processors and $T = O(2^{n/4}(2^{n/4})^{\varepsilon})$ time where $0 \le \varepsilon \le 1$ to solve the knapsack problem. Therefore, we get a time-processor tradeoff (cost) $T \cdot p = O(2^{n/2})$, which is the first optimal algorithm and also the first without memory conflicts algorithm for the solution of the knapsack problem.

It is worth noticing that the same techniques used in this paper can be used to solve a larger class of NP-complete problems by correspondingly changing a little[6]. Exact satisfiability and exact cover are examples of such problems, known as "polynomial enumerable" NP- complete problems [6].

The rest of the paper is organized as follows. The proposed parallel algorithm is described in Section 2. Then, in Section 3, the performance analysis and comparisons follow. Finally, some concluding remarks are given in Section 4. Hereafter, let $N = 2^{n/2}$, and $e = N^{1/2} = 2^{n/4}$, while all logarithms will be to the base 2.

II. The Proposed Parallel Algorithm

1. $p = O(N^{1/2})$

To simplify the notations we will first introduce our parallel algorithm supposing that $p = O(N^{1/2}) = O(e)$ processors are available for computing. Consider the two stages of the *two-list* algorithm one by one, i.e., the generation stage and search stage of the lists A and B.

1.1 The generation stage

Consider the last step where the sorted list A is generated. Suppose the sorted list $A_{n/2-1}$ that has $N/2$ sorted subset sums in the shared memory is $A_{n/2-1} = [a_{n/2-1,1}, a_{n/2-1,2}, \ldots, a_{n/2-1,N/2}]$. Now $w_{n/2}$ is added to each element of $A_{n/2-1}$ using p processors in parallel. Since the *concurrent read* is not permitted, it is necessary to duplicate a copy of vector W for every processor at the start of the algorithm. Thus it needs

- This work has been supported by the National Natural Science Foundation of China under Grant No. 60273075 and the National High Technology Development 863 Program of China under Grant No. 863-306 ZD-11-01-06.

$O(nN^{1/2})$ assistant memory cells in the shared memory, which has a length of $O(N)$. Each processor then compute and write $N/(2p)$ subset sums of the corresponding list $A^1_{n/2\text{-}1} = [a_{n/2\text{-}1.1} + w_{n/2}, a_{n/2\text{-}1.2} + w_{n/2}, \ldots, a_{n/2\text{-}1.N/2} + w_{n/2}]$ into the different cells in the shared memory. To merge the two lists $A_{n/2\text{-}1}$ and $A^1_{n/2\text{-}1}$, we call the *optimal merging algorithm*[15] to complete it. Because the *optimal merging* is without memory conflicts, the total generation stage is also without memory conflicts. Considering the needed time for *optimal merging* is $N/p + \log p \times \log N$[15], the total parallel operation time in this period where list $A_{n/2}$ is generated from list $A_{n/2\text{-}1}$ is $N/(2p) + N/p + \log p \times \log N = 3N/(2p) + n^2/8$. As the depiction above, in the course of the sorted list $A_{n/2}$ (or A) being generated, at start there is the naturally sorted list $A_1 = [0, w_1]$, which contains two elements. Then the sorted list A_2 is generated, containing four elements, ..., the sorted list $A_{n/2\text{-}1}$ generated, containing $N/2$ elements. The needed sorted list, $A_{n/2}$ (or A) that contains N elements is generated finally. To generate lists $A_1, A_2, \ldots$, and $A_{n/4}$, only a part of processors are utilized, while to generate lists $A_{n/4+1}, \ldots, A_{n/2}$, all processors must be utilized. Therefore, the procedures for generating the nondecreasingly sorted list A can be described as following.

Parallel generation algorithm

List $A_1 = [0, w_1]$ is given
begin
  for $i = 1$ to $n/2 - 1$ do
    do in parallel
      produce all the subset sums of A^1_i by adding the value of w_{i+1} to all the subset sums of A_i.
      perform the optimal merging algorithm given in [15].
    end do
  end for
end

Obviously, the procedure to generate the list B sorted by nonincreasing order is almost as same as the above algorithm.

1.2 The search stage

Though the main conclusions in the algorithm of Lou and Chang exist error[12], their process of the search is very elegant. However, in their parallel search algorithm, there are only $O(2^{n/8})$ processors, and more importantly, their algorithm is on *CREW* model where it is not necessary to consider that all processors may concurrently read the same memory cells in the shared memory. Thus we need to reconsider the search stage.

Firstly, we divide the two sorted lists A and B into p (or e) blocks, respectively. Thus each block has $N/p = e$ sorted elements. Suppose the structure of the list A and B is respectively as $A = (\overline{A_1}, \overline{A_2}, \ldots, \overline{A_i}, \ldots, \overline{A_e})$, where $\overline{A_i} = (\overline{a_{i.1}}, \overline{a_{i.2}}, \ldots, \overline{a_{i.e}})$, $1 \le i \le p$, and $B = (\overline{B_1}, \overline{B_2}, \ldots, \overline{B_j}, \ldots, \overline{B_e})$, where $\overline{B_j} = (\overline{b_{j.1}}, \overline{b_{j.2}}, \ldots, \overline{b_{j.e}})$, $1 \le j \le p$. Secondly we formally assign the ith block of A to the processor P_i, $1 \le i \le p$. To reduce the search time of each processor, introduce the following lemma 1 and 2, which is similar to the lemma 1 and 2 in literature [12]. As the proof of the two lemmas is also similar to the proof of lemma 1 and 2 in [12], here, we omit them.

Lemma 1. For any block pair $(\overline{A_i}, \overline{B_j})$, $1 \le i, j \le p$, if $\overline{a_{i.1}} + \overline{b_{j.e}} > M$, then any element pair $(\overline{a_{i.r}}, \overline{b_{j.s}})$, where $\overline{a_{i.r}} \in \overline{A_i}$, $\overline{b_{j.s}} \in \overline{B_j}$, is not a solution of the knapsack problem.

Lemma 2. For any block pair $(\overline{A_i}, \overline{B_j})$, $1 \le i, j \le p$, if $\overline{a_{i.e}} + \overline{b_{j.1}} < M$, then any element pair $(\overline{a_{i.r}}, \overline{b_{j.s}})$, where $\overline{a_{i.r}} \in \overline{A_i}$, $\overline{b_{j.s}} \in \overline{B_j}$, is not a solution of the knapsack problem.

Based on Lemmas 1 and 2, we design the following parallel Prune algorithm, which can make each processor P_i shrink its own search space greatly.

Prune algorithm

begin
  Divide the two ordered lists, A and B, into p blocks, respectively.
  for all P_i where $1 \le i \le p$ do
    for $j = i$ to $p + i - 1$ do
    begin
      $X = \overline{a_{i.1}} + \overline{b_{j \bmod p.e}}$;
      $Y = \overline{a_{i.e}} + \overline{b_{j \bmod p.1}}$;
      if ($X = M$ or $Y = M$) then stop, a solution is found
      else if ($X < M$ and $Y > M$) then write $(\overline{A_i}, \overline{B_{j \bmod p}})$ to the shared memory;
    end
end

In the above Prune algorithm, since in one time, each memory cells of list B is read or written at mostly by one processor, hence the Prune algorithm can be executed on *EREW-SIMD* model. Because each processor prunes its individual search space, it is easy to know that the time complexity of the stated algorithm is $O(p)$, i.e., $O(2^{n/4})$. Before the Prune algorithm begins, the number of the possible block pairs must be $p \times p = N$.

Theorem 1. The Prune algorithm picks at most $2p$ block pairs.

Proof. The proof of this Theorem is similar to the Theorem 2 in literature [12].

Because the Theorem 1 is true, we can equally divide the $2p$ block pairs in the shared memory into p parts, and then formally distribute each part to a processor. Thus the number of block pairs that each processor has to search is at most for 2, i.e., a "constant" $O(1)$. Finally, each processor performs the one direction search routine so as to find the solution to the knapsack problem instances. The following theorem 2 can ensure that the search routine be performed on *EREW* only in time $O(e)$.

Theorem 2. The parallel search routine can be performed on *EREW* in time $O(e)$.

Proof: Theorem 1 shows that each processor has at most

two block pairs to search over. For any two processors P_i and P_j, where $1 \le i \ne j \le p$, suppose block pair $(\overline{A_k}, \overline{B_l})$ is belong to P_i, and $(\overline{A_s}, \overline{B_t})$ is belong to P_j. Here $\overline{A_k} = (\overline{a_{k.1}}, \overline{a_{k.2}}, \ldots, \overline{a_{k.e}})$, $\overline{B_l} = (\overline{b_{l.1}}, \overline{b_{l.2}}, \ldots, \overline{b_{l.e}})$, $\overline{A_s} = (\overline{a_{s.1}}, \overline{a_{s.2}}, \ldots, \overline{a_{s.e}})$ and $\overline{B_t} = (\overline{b_{t.1}}, \overline{b_{t.2}}, \ldots, \overline{b_{t.e}})$.

(1) If $k \ne s$ and $l \ne t$, then it is impossible for processors P_i and P_j to have any memory conflicts in the period of search, for the memory positions which P_i access are completely different with those for P_j.

(2) Else $k = s$ or $l = t$. Let us suppose the case $k < s$ and $l = t$ as shown in Fig. 1, is true. Since $\overline{a_{k.e}} + \overline{b_{l.1}} > M$, and the list A is sorted by nondecreasing order, hence $\overline{a_{s.1}} + \overline{b_{l.1}} > M$. as the parallel search routine begins, let processor P_j read $\overline{b_{l.2}}$ which is in the shared memory, thereafter it compute and check whether the value of $\overline{a_{s.1}} + \overline{b_{l.2}}$ is equal to M. While at this time processor P_i read $\overline{b_{l.1}}$ and check if $\overline{a_{k.1}} + \overline{b_{l.1}} = M$.

i) With the search routine continued, once processor P_j found that $y_0 + \overline{b_{l.c}} < M$, where $y_0 \in \overline{A_s}$, $c \in \{1, 2, \ldots, 2^{n/4}\}$, then for any value $x \in \overline{A_k}$ and any $d \ge c$, $x + \overline{b_{l.d}} < M$. thus it is not necessary for processor P_i to check if $x + \overline{b_{l.d}} = M$. therefore in this case processor i and j don't have any memory conflicts.

ii) Else in the search routine for processor P_j, it didn't found a pair which satisfied that $y_0 + \overline{b_{l.c}} < M$. i.e., for processor P_j, it only need to check if $\overline{a_{s.1}} + \overline{b_{l.z}} = M$ where $z \in \{1, 2, \ldots, 2^{n/4}\}$. Noticing that the time unit when processor P_j began search routine is earlier than that of processor P_i for *one* unit, in this case processor P_i and P_j didn't have any memory conflicts, either.

Obviously, in the case where more than two processors need to access the same block, the process procedure is similar with the above case where two processors must access the same block $\overline{B_t}$. It is clear that the time complexity of this parallel search routine is at most for $O(2(p + 2N/p))$, which is obviously bounded by $O(e)$. This completes the proof of the theorem 2.

Combining the Parallel generation algorithm and the Prune algorithm into integration, we present the whole parallel algorithm for the knapsack problem as follows.

Optimal parallel algorithm on *EREW*

1. Perform the parallel generation algorithm.
2. Perform the prune algorithm.
3. Equally assign the picked block pairs to each processor.
4. for all P_i where $1 \le i \le p$ do

   P_i performs the search routine on *EREW* of the *two-list* algorithm.

2. $p < O(N^{1/2})$

obviously, it is not difficult to extend the results stated to the case where $p < O(N^{1/2})$. Assume that the number of processors is $p = e^{1-\varepsilon}$, $0 < \varepsilon \le 1$. Based on the three main steps in the proposed parallel algorithm, the parallel algorithm in this case is almost the same.

III. Performance Analysis and Comparisons

Consider the case for $p = O(N^{1/2})$ firstly. The parallel operation time needed in parallel generation algorithm is as following:

$$O(\sum_{i=1}^{n/4-1}(2+(i+1)i)) + \sum_{i=n/4}^{n/2-1}(32^{i+1}/(2p) + n(i+1)/4)$$
$$= O(3N/p) = O(2^{n/4}) \quad (2)$$

In the search stage, the execution of the Prune algorithm needs time for $O(p)$, i.e., for $O(2^{n/4})$. The search routine takes $O(6N/p) = O(2^{n/4})$ time to find a solution in the worst case. The total time complexity in this case is therefore $O(2^{n/4})$, and the cost of the proposed parallel algorithm is $C = T \times p = O(2^{n/4}) \times O(2^{n/4}) = O(2^n)$. Secondly, if the number of processors p satisfies $1 \le p < N^{1/2}$. Through analyzing the time complexity of the three components of the proposed algorithm in this case, we can know the total time of our parallel algorithm is then as $T = O(3N/p) + O(p) + O(2p+4N/p)$. Noticing that $1 \le p < N^{1/2}$, it can result that $N/p > p$. therefore the total time T is clearly dominated by $O(N/p)$, and the cost is still as $C = O(N/p) \times O(p) = O(2^{n/2})$.

In summary, for any case of $1 \le p \le 2^{n/4}$, the cost of the proposed parallel algorithm can be kept unchanged for $O(2^n)$.

Following the previous researches, the performance comparison will be described in terms of time-processor tradeoff, i.e., the cost of the parallel algorithm. For the purpose of clarity, the comparisons of the above mentioned parallel algorithms for solving the knapsack problem are depicted in Table 1.

Table 1 shows that our algorithm is the first parallel algorithm that is both optimal and without memory conflicts. Moreover, like Ferreira's parallel algorithm in [7,14], the number of processors can be adjusted according the number of processor available. In a word, it is obvious that our parallel algorithm outtakes undoubtedly other parallel algorithms in the overall performance.

Tabel 1. Comparisons of the Parallel Algorithms for Solving the Knapsack Problem

Algorithms	Model	Processor	Time	Memory	Cost	Adaptability	Technique
Karnin[7]	CREW	$O(2^{n/6})$	$O(2^{n/2})$	$O(2^{n/6})$	$O(2^{2n/3})$	no	DG
Amirazizi[9]	CREW	$O(2^{(1-\alpha)n/2})$	$O(n2^{\alpha n})$	$O(2^{(1-\alpha)n/2})$	$O(n2^{(1+\alpha)n/2})$	yes	DG and PS
Ferreira[10]	CREW	$O(2^{\varepsilon n/2})$	$O(n2^{(1-\varepsilon/2-\alpha)n})$	$O(2^{\alpha n})$	$O(n2^{(1-\alpha)n})$	yes	DG and PS
Ferreira[14]	CREW	$O(p)$	$O(n2^{(n-n'/2)}/p)$	$O(2^{n'/4})$	$O(n2^{(n-n'/2)})$	yes	DG and PS
Ferreira[7]	CREW	$O((2^{n/2})^{1-\varepsilon})$	$O(n(2^{n/2})^{\varepsilon})$	$O(2^{n/2})$	$O(n2^{n/2})$	yes	PG and PS
Chang[11]	CREW	$O(2^{n/8})$	$O(n2^{n/2})$	$O(2^{n/4})$	$O(n2^{5n/8})$	no	DG
Lou[12]	CREW	$O(2^{n/8})$	$O(n2^{n/2})$	$O(2^{n/4})$	$O(n2^{5n/8})$	no	DG and PS
Ours	EREW	$O((2^{n/4})^{1-\varepsilon})$	$O(2^{n/4}(2^{n/4})^{\varepsilon})$	$O(2^{n/2})$	$O(2^{n/2})$	yes	PG and PS

Notations. $0 \le \varepsilon \le 1$, $0 \le \alpha \le 1/2$, $n' \le n$, $p \le 2^{n'/4}$, DG-dynamic generation, PG-parallel generation, PS-parallel search.

IV. Conclusions

Based on the *two-list* algorithm and an *EREW* and *SIMD* machine with shared memory. We proposed a new parallel algorithm for solving the knapsack problem. The proposed algorithm apply $(2^{n/4})^{1-\varepsilon}$ processors and $O(2^n)$ memory to solve the knapsack problem in time $O(2^{n/4}(2^{n/4})^{\varepsilon})$ only, $0 \le \varepsilon \le 1$. The cost of our algorithm is $O(2^{n/2})$, which is an improvement result over the past researches. Furthermore, if the open problem posed by Schroeppel and Shamir[7]: *"whether $T = O(2^{n/2})$ is a lower bound for solving the knapsack problem sequentially"* is true, then by the definition given in [14], the parallel algorithm is optimal. The theoretic importance of the proposed algorithm is that our parallel algorithm is both the first optimal and without memory conflicts algorithm for the knapsack problem.

References

[1] M. R. Garey, D.S. Johnson. *Computers and intractability: A guide to the theory of NP-Completeness.* San Francisco:W.H.Freeman and Co, 1979.

[2] A. Shamir, "A polynomial-time algorithm for breaking the basic Merkle-Hellman cryptosystem", *IEEE Trans. Inform. Theory*, Vol.30, No.5, pp.699-704, 1984.

[3] B. Chor and R.L. Rivest, "A knapsack–type public key cryptosystem based on arithmetic in finite fields", *IEEE Trans. Inform. Theory*, Vol.34, No.5, pp.901-909, 1985.

[4] C.-S. Laih, J.-Y. Lee, L. Harn and Y.-K. Su, "Linearly shift knapsack public-key cryptosystem", *IEEE J. Selected Areas Commun*, Vol.7, No.4, pp.534-539, 1989.

[5] E. Horowitz and S. Sahni, "Computing partitions with applications to the knapsack problem", *J. ACM*, Vol.21, No.2, pp.27-292, 1974.

[6] R. Schroeppel and A. Shamir, "A T = O(2n/2), S = O(2n/4) algorithm for certain NP-complete problems", *SIAM J. Comput*, Vol.10, No.3, pp.456-464, 1981.

[7] A. G. Ferreira, "A parallel time/hardware tradeoff $T \cdot H$ = O(2n/2) for the knapsack problem", *IEEE Trans. Comput*, Vol,40, No.2, pp.221-225, 1991.

[8] E. D. Karnin, "A parallel algorithm for the knapsack problem", *IEEE Trans, Comput*, Vol.33, No.5, pp.404-408, 1984.

[9] H. R. Amirazizi, M.E. Hellman, "Time memory processor tradeoffs", *IEEE Trans. Inform. Theory*, Vol.34, No.3, pp.502-512, 1988.

[10] A. G. Ferreira, "Work and memory efficient parallel algorithms for the knapsack problem", *International Journal of High Speed Computing*, Vol.4, pp.71-80, 1995.

[11] H. K.-C. Chang, J. J.-R. Chen and S.-J. Shyu, "A parallel algorithm for the knapsack problem using a generation and searching technique", *Parallel Computing*, Vol.20, No.2, pp.233-243, 1994.

[12] D. C. Lou and C. C. Chang, "A parallel two-list algorithm for the knapsack problem", *Parallel Computing*, Vol.22, pp.1985-1996, 1997

[13] C. A. Aanches, N. Y. Soma, and H. H. Yanasse, "Comments on parallel algorithms for knapsack problem", *Parallel Computing*, Vol.28, pp.1501-1505, 2002.

[14] A. G. Ferreira and J.M. Robson, "Fast and scalable parallel algorithms for knapsack-like problems", *Journal of Parallel and Distributed Computing*, Vol.39, pp.1-13, 1996.

[15] S. G. Akl, "Optimal parallel merging and sorting without memory conflicts", *IEEE Trans, Comput*, Vol.36, No.11, pp.1367-1369, 1987.