2011 19th International Euromicro Conference on Parallel, Distributed and Network-Based Processing

Adaptive and Cost-Optimal Parallel Algorithm for the 0-1 Knapsack Problem

Kenli Li [1], Lingxiao Li [1], Teklay Tesfazghi [1,2], Edwin Hsing-Mean Sha [1,3]

[1]*College of Computer and Communication, Hunan University*
Changsha 410082, China
lkl510@263.net, jacqueli@163.com

[2]*University of Asmara, P.O. Box 1220*
Asmara, Eritrea
tekliya@yahoo.com

[3]*Department of Computer Science, University of Texas at Dallas*
Dallas, U.S.A
edsha@utdallas.edu

***Abstract*—The 0-1 knapsack problem is well known to be NP-complete problem. In the past two decades, much effort has been done in order to find techniques that could lead to algorithms with a reasonable running time. This paper proposes a new parallel algorithm for the 0-1 knapsack problem where the optimal merging algorithm is adopted. Based on an EREW PRAM machine with shared memory, the proposed algorithm utilizes $O((2^{n/4})^{1-\varepsilon})$ processors, $0 \leq \varepsilon \leq 1$, and $O(2^{n/2})$ memory to find a solution for the n-element 0-1 knapsack problem in time $O(2^{n/4}(2^{n/4})^{\varepsilon})$. Thus the cost of the proposed parallel algorithm is $O(2^{n/2})$, which is both the lowest upper-bound time and without memory conflicts if only quantity of objects is considered in the complexity analysis for the 0-1 knapsack problem. Thus it is an improvement result over the past researches.**

***Keywords*-parallel computing; combinatorial optimization; 0-1 knapsack problem; divide and conquer; EREW PRAM**

I. Introduction

Given a set of n objects v_1, v_2, $\cdots$, v_n, each one with a weight $w_i \in Z^+$ and a profit $p_i \in Z^+$, $w_i = v_i.w$ and $p_i = v_i.p$, $1 \leq i \leq n$, the 0-1 knapsack problem named KP01, is to find the most profitable subset amongst objects from the n objects without exceeding the knapsack capacity of $c \in Z^+$. Alternatively, determine a binary n-tuple X = (x_1, x_2, $\cdots$, x_n) which maximizes $\sum_{i=1}^{n} p_i x_i$, subject to $\sum_{i=1}^{n} w_i x_i \leq c$ [1]. Another problem which is relevant to KP01 is subset sum problem (SSP). In SSP, it is requested to determine a binary n-tuple X = (x_1, x_2, $\cdots$, x_n), such that $\sum_{i=1}^{n} w_i x_i = c$. KP01 is proven to be NP-complete problem, and unless NP = P, it is unlikely to be solved in polynomial time [2]. Because of its importance in combinatorial optimization and its varied practical applications in cargo loading problem, capital budgeting, cutting stock problem, and computer cryptosystem, it is heavily researched since the past three decades.

The algorithms for solving KP01 can be classified into two methods: the dynamic programming and the method of divide and conquer. By using dynamic programming, starting from fifties of the last century, KP01 has been sequentially solved with in $O(nc)$ time and space complexity using the well-known Bellman's dynamic programming paradigm [3]. Based on it, many parallel algorithms for the KP01 appeared in the literature [4-11]. Kindervater and Lenstra [6] and latter on Lin and Storer [8] suggested a direct parallelization for the Bellman's approach. In an EREW PRAM (exclusive read and exclusive write) with k processors, this algorithm demands $O(nc/k)$ time and $O(nc)$ space. Goldman and Trystram [4] introduced an algorithm for the hypercube to the unbounded knapsack problem where the variables are non-negative integers instead of 0 or 1, which is bounded in time by $O(n \log w_{\max} + \frac{c}{w_{\min}})$ with $O(\frac{cw_{\max}}{w_{\min} \log w_{\max}})$ processors. Although these algorithms can be efficiently implemented, they do not improve the time and space upper-bounds for the KP01. Recently, Sanches et al [12] proposed a parallel time and space upper-bounds, $O(\frac{n}{k}(c - w_{\min}))$ and $O(n + c)$ for the KP01. It is better than the direct parallelization of Bellman's algorithm and is the most efficient known result.

The second method of solving KP01, divide and conquer, is introduced by Horowitz and Sahni [13], who drastically reduced the time needed in the brute force from $O(2^n)$ to $O(n2^{n/2})$. Even though in that paper [13] only the SSP is solved, it pointed out that the method of divide and conquer can be applied to the similar "polynomial enumerable" NP complete problems including KP01, exact satisfiability and exact cover problem, etc. This result is still the lowest upper-bound sequential time if only quantity of objects is considered in the complexity analysis. On the parallel methods for SSP, there are a few parallel algorithms for it on PRAM. Karnin [14] proposed a parallel algorithm that parallelizes the generation routine of the two-list four-table algorithm. Amirazizi and Helman [15] were the first to show that parallelism could accelerate to solve larger instances of SSP. Ferreira [16] proposed a brilliant parallel algorithm that solves the SSP of size n in time $T = O(n(2^{n/2})^{\varepsilon})$,

1066-6192/11 $26.00 © 2011 IEEE
DOI 10.1109/PDP.2011.11

$0 \leq \varepsilon \leq 1$, when $k = O((2^{n/2})^{1-\varepsilon})$ processors are available. Recently, optimal PRAM parallel algorithms are proposed by F. B. Chedid, and C.A. Sanches et al in [18,19] respectively.

However for KP01, the parallel method of divide and conquer has not been applied as successfully as in case of the above mentioned problems. Here on this paper, we are proposing a sequential algorithm for KP01 on the basis of the $O(n2^{n/2})$ algorithm of Horowitz and Sahni [13], and thereafter we are going to present an optimal parallel algorithm in which to solve KP01 with $O(2^{n/4}(2^{n/4})^{\varepsilon})$ time complexity if $O((2^{n/4})^{1-\varepsilon})$ processors are available, $0 \leq \varepsilon \leq 1$, on the EREW PRAM model.

A PRAM (Parallel Random Access Machine) is usually considered just a theoretical model due to infeasibility of efficient and practical implementations. However, the results obtained in this model do fully express the relations that can be processed simultaneously and therefore they have a wide and longstanding impact. A series of recent new algorithms [20-24] do confirm the interest in finding new time or space upper-bounds for a great variety of problems to this model. our parallel algorithm base on an EREW PRAM machine.

The rest of this paper is organized as follows. Section 2 explains the serial algorithm and optimal merging without memory conflicts, on which the proposed parallel algorithm is based. The proposed parallel algorithm is described in Section 3. Then, in Section 4, the performance analysis and comparisons follows. Finally, some concluding remarks are given in Section 5 as well as some future research directions in this field.

For simplicity reasons of the variables used, let $N = 2^{n/2}$, and $e = N^{1/2} = 2^{n/4}$ in the rest of this paper.

II. Two basic algorithms

Before introducing the main results of this paper, since our proposed solution uses the serial algorithm for KP01 and the optimal merging algorithm on EREW, we have discussed them here:

At first to define the problem of KP01, we introduce a structure a_i and b_i , $1 \leq i \leq n$, each having two member variables named w and p, w represent weight of the object, p represent profit of the object, and $a_i.w$ represent sums of the weight w of objects which belongs to a_i , $a_i.p$ represent the sums of value p of objects which belongs to a_i. The two basic algorithms that we discuss here are: The serial algorithm for KP01 and optimal merging algorithm.

A. The serial algorithm for KP01

The serial algorithm can be divided into two stages based on the activities performed: generation and search stages. Given a set of n objects v_1, v_2,$\cdots$, v_n named V, The former is designed for generating two sorted lists for V, such that $A = [a_1, a_2, \cdots, a_N]$ and $B = [b_1, b_2, \cdots, b_N]$ and the latter for each b of B, first save $MaxB_i$ which represent the current max value of p from b_i to b_N, then search the final solutions from the combinations of the two sorted lists as it is shown in Figure 1.

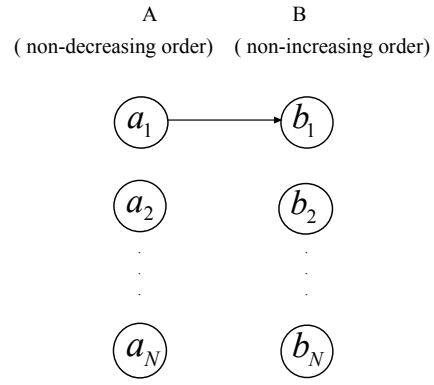

Figure 1. search stage

1) Generation stage: Generation stage have three phases:

1. Divide V into two equal parts: $V_1 = (v_1, v_2, \cdots, v_{n/2})$, and $V_2 = (v_{n/2+1}, v_{n/2+2}, \cdots, v_n)$.

2. Form all N possible subset sums of $V_1 = (v_1, v_2, \cdots, v_{n/2})$, and then according to the value of member variable w, sort them in a non-decreasing order and store them as the list $A = [a_1, a_2, \cdots, a_N]$.

3. Form all N possible subset sums of $V_2 = (v_{n/2+1}, v_{n/2+2}, \cdots, v_n)$, and then according to the value of member variable w, sort them in a non-increasing order and store them as the list $B = [b_1, b_2, \cdots, b_N]$.

2) Search stage: The search stage have two phases.

The first phase:

For each b of B, we save $MaxB_i$ which represent the current max value of p from b_i to b_N .

Algorithm 1 The first step of search algorithm

Require: int $MaxB_N = b_N.p$;

```
for i = N − 1 to 1 do
    if b_i.p > MaxB_{i+1} then
        MaxB_i = b_i.p ;
    else
        MaxB_i = MaxB_{i+1} ;
    end if
end for
return MaxB ;
```

The second phase:

The serial search finds the final solutions from the combinations of the two sorted lists. The return value $Bestvalue$ of this algorithm represents the final solutions of KP01. The second step of search algorithm can be described as the algorithm 2.

In fact, we can divide the serial algorithm into the following three main steps, among which step a and b belong

Algorithm 2 The second step of search algorithm

Require: int $Bestvalue = 0$;
1: int j=1 ; int i =1 ;
2: **if** $a_i.w + b_j.w > c$ **then**
3: $j = j + 1$ and Go to Step 9 ;
4: **end if**
5: **if** $a_i.p + MaxB_j > Bestvalue$ **then**
6: $Bestvalue = a_i.p + MaxB_j$;
7: **end if**
8: i = i +1 ;
9: **if** $i > N$ **or** $j > N$ **then**
10: stop and return $Bestvalue$;
11: **end if**
12: Go to Step 2 ;

to the generation stage, step c belongs to search stage.

a. generation all subset sums of A and B.

b. according to the value of member variable w, sorting of A and B.

c. serial search on lists A and B.

In this algorithm, if step a and b are done all at once through merging, then the time complexity can be reduced to $O(N)$. Beginning with the naturally sorted list $A_1 = [a_0, a_1]$, such that $a_0.w = 0, a_0.p = 0, a_1 = v_1$, we construct another one A_1^1 by adding $v_2.p$ and $v_2.w$ to each element of the former list A_1, getting the list $A_1^1 = [a_0 + v_2, a_1 + v_2]$. Then the sorted list A_2 can be obtained by merging these two lists A_1 and A_1^1 in linear time. Next we add $v_3.p$ and $v_3.w$ to the resulting list A_2, getting the list A_2^1, then merge them both, and repeat until all the subset sums of V_1 have been generated and the sorted list $A_{n/2}$ (i.e. the list A) has been obtained. Hence the total time of step a and b is $\sum_{0 \le i \le n/2-1} O(2^i) = O(N)$.

It is obvious that step c can be completed in time $O(N)$. Therefore, the algorithm's worst case complexity can be reduced from $O(nN)$ to $O(N)$.

B. Optimal merging on EREW

The algorithm in [17] was the first optimal parallel merging algorithm without memory conflicts. Here, for the sake of consistency in this paper, the description of this algorithm may have a little difference on the subscript parameter with the one in [17]. If there are two sorted vectors $U = (u_1, u_2, \cdots, u_m)$ and $V = (v_1, v_2, \cdots, v_m)$ and k processors $P_1, P_2, \cdots, P_k$ where $1 \le k \le 2m$ is a power of 2. According to the following steps, vector U and V can be merged without memory conflicts into a new vector which has a length of $2m$.

The optimal merging algorithm

Step 1: Use the k processors to partition U and V, in parallel and without memory conflicts, each into k (possibly empty) subvectors $(U_1, U_2, \cdots, U_k)$ and $(V_1, V_2, \cdots, V_k)$ such that

1 $|U_i| + |V_i| = 2m/k$, for $1 \le i \le k$.

2 Member variable w of all elements in U_i and V_i are smaller than that of all elements in U_{i+1} and V_{i+1}, for $1 \le i \le k-1$.

Step 2: Use processor P_i to merge U_i and V_i, for $1 \le i \le k$.

Here step 1 can be efficiently implemented using the selection algorithm presented in [17]. It has proven that this parallel algorithm can be demonstrated in the EREW PRAM model of parallel computation in time $O(2m/k + logk \times log2m)$. The algorithm is therefore optimal for $k \le m/\log^2 m$, in view of the trivial $\Omega(m)$ lower bound in merging two vectors of total length $2m$ [17].

III. The proposed parallel algorithm

To simplify the notations we will first introduce our parallel algorithm supposing that $k = O(N^{1/2}) = O(e)$ processors are available for computing. In the subsection F, we will extend it to the case where $k < O(N^{1/2})$. Using the two-list algorithm as a base, we design a new algorithm to solve KP01. The algorithm has five-steps, and we will describe it in detail one by one.

A. The generation stage

Since the procedures to generate sorted lists in nondecreasing or nonincreasing order is similar, the proposed algorithm only describes the procedures for generation of a nondecreasing list through merging.

Let us 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 $A_{n/2-1} = [a_{n/2-1.1}, a_{n/2-1.2}, \cdots, a_{n/2-1.N/2}]$. Now the object $v_{n/2}$ is added to each element of $A_{n/2-1}$ using k processors in parallel. Since the concurrent read is not permitted, it is necessary to duplicate a copy of vector $v_{n/2}.w$ and $v_{n/2}.p$ for every processor before the algorithm begins. Thus it needs $O(nN^{1/2})$ assistant memory cells in the shared memory, which has a length of $O(N)$. Each processor then compute N/k subset sums of the corresponding list $A_{n/2-1}^1$ = [$a_{n/2-1.1}.w + v_{n/2}.w$, $a_{n/2-1.1}.p + v_{n/2}.p$, $a_{n/2-1.2}.w + v_{n/2}.w$, $a_{n/2-1.2}.p + v_{n/2}.p$, $\cdots$, $a_{n/2-1.N/2}.w + v_{n/2}.w$, $a_{n/2-1.N/2}.p + v_{n/2}.p$], and write them into the different cells in the shared memory. According to the value of w merge the two lists $A_{n/2-1}$ and $A_{n/2-1}^1$, we call the optimal merging algorithm [17] which is given in section $II.B$ 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 $N/k + \log k \times \log N$ [17], the total parallel operation time in this period where list $A_{n/2}$ is generated from list $A_{n/2-1}$ is $N/2k + N/k + \log k \times \log N = 3N/2k + n^2/8$.

As it is shown above, in the course of the sorted list $A_{n/2}$(or A) being generated, at start there is the naturally sorted list $A_1 = [a_0, a_1]$, which contains two elements, such

that $a_0.w = 0$, $a_0.p = 0$ and $a_1 = v_1$. Then the sorted list A_2 is generated, containing four elements, $\cdots$, and the sorted list $A_{n/2-1}$ is generated, containing $N/2$ elements. The needed sorted list, $A_{n/2}$ (or A) that contains N elements is generated finally. To generate lists $A_1, A_2, \cdots, and\ A_{n/4}$, only a part of the processors are utilized, while to generate lists $A_{n/4+1}, \cdots, A_{n/2}$, all processors must be utilized. Therefore, the procedures for generating the nondecreasingly sorted list A can be described as the algorithm 3.

Algorithm 3 Parallel generation algorithm

Require: $A_1 = [a_0, a_1]$;
1: **for** $i = 1$ to $n/2 - 1$ **do**
2: **for all** P_i such that $1 \le i \le k$ **in parallel do**
3: produce all the subset sums of A_i^1 by adding the value of $v_{i+1}.w$ to member variable *w* of all the subset sums of A_i
4: and by adding the value of $v_{i+1}.p$ to member variable *p* of all the subset sums of A_i ;
5: for A_i and A_i^1 , according to the value of member variable *w* nondecreasingly order, perform the optimal merging algorithm given in section $II.B$, then produce all the subset sums of A_{i+1} ;
6: **end for**
7: **end for**
8: **return** $A_{n/2}$;

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

B. The first parallel save maxvalue stage

At first, given the sorted lists A and B, we evenly divide the two sorted lists A and B into k blocks, respectively. Thus each block has $N/k = e$ sorted elements. As Figure 2 shows, the two lists A and B are: $A = (\overline{A_1}, \overline{A_2}, \cdots, \overline{A_i}, \cdots, \overline{A_k})$, where $\overline{A_i} = (\overline{a_{i.1}}, \overline{a_{i.2}}, \cdots, \overline{a_{i.e}})$, $1 \le i \le k$, and $B = (\overline{B_1}, \overline{B_2}, \cdots, \overline{B_j}, \cdots, \overline{B_k})$, where $\overline{B_j} = (\overline{b_{j.1}}, \overline{b_{j.2}}, \cdots, \overline{b_{j.e}})$, $1 \le j \le k$.

Secondly, we formally assign the i-th block of A and B to the processor P_i, $1 \le i \le k$, then execute the first parallel save maxvalue algorithm which can be described as the algorithm 4.

Let $MaxA_i$ represent the max value of p from $\overline{a_{i.1}}$ to $\overline{a_{i.e}}$, and $MaxB_i$ represent the max value of p from $\overline{b_{i.1}}$ to $\overline{b_{i.e}}$.

Obviously, the total parallel operation time in this period is $O(N^{1/2})$.

C. The prune stage

We formally assign the i-th block of A to the processor P_i, $1 \le i \le k$. To reduce the search time of each processor, introduce the following lemma 1 and 2. As the proof of the two lemmas is also similar to the proof of lemma 1 and 2 in [12], here, we omit them.

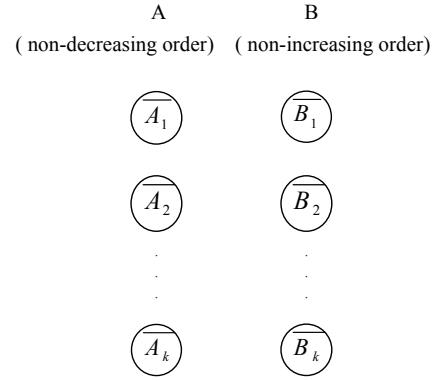

Figure 2. The block structure of the list A and B

Algorithm 4 The first parallel save maxvalue algorithm

Require: Divide the two ordered lists A and B into k blocks, respectively ;
1: **for all** P_i such that $1 \le i \le k$ **do in parallel do**
2: $int\ MaxA_i = \overline{a_{i.1}}.p$; $int\ MaxB_i = \overline{b_{i.1}}.p$;
3: **for** $j = 2$ to e **do**
4: **if** $\overline{a_{i.j}} > MaxA_i$ **then**
5: $MaxA_i = \overline{a_{i.j}}.p$;
6: **end if**
7: **if** $\overline{b_{i.j}} > MaxB_i$ **then**
8: $MaxB_i = \overline{b_{i.j}}.p$;
9: **end if**
10: **end for**
11: **end for**
12: **return** $MaxA$ **and** $MaxB$;

Lemma 1. For any block pair ($\overline{A_i}$, $\overline{B_j}$), $1 \le i, j \le k$, $\overline{A_i} = (\overline{a_{i.1}}, \overline{a_{i.2}}, \cdots, \overline{a_{i.e}})$ and $\overline{B_j} = (\overline{b_{j.1}}, \overline{b_{j.2}}, \cdots, \overline{b_{j.e}})$, if $\overline{a_{i.1}}.w + \overline{b_{j.e}}.w > c$, 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 KP01.

Lemma 2. For any block pair ($\overline{A_i}$, $\overline{B_j}$), $1 \le i, j \le k$, $\overline{A_i} = (\overline{a_{i.1}}, \overline{a_{i.2}}, \cdots, \overline{a_{i.e}})$ and $\overline{B_j} = (\overline{b_{j.1}}, \overline{b_{j.2}}, \cdots, \overline{b_{j.e}})$, if $\overline{a_{i.e}}.w + \overline{b_{j.1}}.w \le c$, 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 a possible solution of KP01.

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.The prune algorithm can be described as the algorithm 5.

In the above prune algorithm, since in any time, each memory cell of list B is read or written at most by one processor, hence the prune algorithm can be executed on EREW PRAM model. Because each processor prunes its individual search space, it is easy to see that the time complexity of the stated algorithm is $O(k)$, i.e., $O(2^{n/4})$. As it is shown in the prune algorithm, once the possible block pair ($\overline{A_i}$, $\overline{B_j}$) has been picked, it is written without memory conflicts into the shared memory. Before the prune algorithm begins, the number of all possible block pairs must

Algorithm 5 Prune algorithm

```
1: for all P_i such that 1 ≤ i ≤ k in parallel do
2:   int Maxvalue_i = 0 ;
3:   for j = i to k + i − 1 do
4:     X = a_{i.1}.w + b_{j mod k.e}.w ;
5:     Y = a_{i.e}.w + b_{j mod k.1}.w ;
6:     if Y ≤ c then
7:       if MaxA_i + MaxB_{j mod k} > Maxvalue_i then
8:         Maxvalue_i = MaxA_i + MaxB_{j mod k} ;
9:       end if
10:      drop out the block pair ;
11:    else if X ≤ c and Y > c then
12:      write block pair (A_i, B_{j mod k}) into the different
         cell in the shared memory ;
13:    else if X > c then
14:      drop out the block pair ;
15:    end if
16:  end for
17: end for
```

be $k \times k = N$. but after it has been executed, the following theorem shows that the number of the residual block pairs will be at most $2k$ - 1.

Theorem 1. The prune algorithm picks at most $2k$ - 1 block pairs.

Proof. Let the block pairs $(\overline{A_i}, \overline{B_{y_1}})$, $(\overline{A_i}, \overline{B_{y_2}})$, $\cdots$, $(\overline{A_i}, \overline{B_{y_m}})$ be picked by the processor P_i after the prune algorithm is performed, where $y_1 < y_2 < \cdots < y_m$, and $m \geq 2$. Then all the blocks $\overline{B_{y_1}}, \overline{B_{y_2}}, \cdots, \overline{B_{y_m}}$ are adjacent one by one, i.e., $y_g = y_{g-1} + 1$ for $2 \leq g \leq m$.

Assuming that after the prune algorithm is performed, the processor P_i has picked block pairs $(\overline{A_i}, \overline{B_j})$, $(\overline{A_i}, \overline{B_{j+1}})$, $\cdots$, $(\overline{A_i}, \overline{B_{l-1}})$, $(\overline{A_i}, \overline{B_l})$ as shown in Figure 3. that is, all of the combinations of $\overline{A_i}$ and $\{\overline{B_j}, \overline{B_{j+1}}, \cdots, \overline{B_{l-1}}, \overline{B_l}\}$ must be checked later in order to find a solution.

Let $BS_i = \{\overline{B_j}, \overline{B_{j+1}}, \cdots, \overline{B_{l-1}}, \overline{B_l}\}$ denote the set of blocks, where each block is picked by the processor P_i from the list B. Based on the two-list algorithm and Figure 3, we know that $\overline{a_{i.e}}.w + \overline{b_{t.1}}.w > c$, where $j < t \leq l$; otherwise, the block should not be picked by the processor P_i. Since $\overline{a_{i.e}}.w < \overline{a_{i+1.1}}.w$ and $\overline{b_{t.1}}.w < \overline{b_{t-1.e}}.w$, then $\overline{a_{i+1.1}}.w + \overline{b_{t-1.e}}.w > c$,Therefore, the processor P_{i+1} prunes $\overline{B_{t-1}}$. Thus, $BS_i \cap BS_{i+1} = \Phi$ or $\{\overline{B_l}\}$, and $|BS_i \cap BS_{i+1}| \leq 1$, $1 \leq i < k$.

Let $|BS_i| = \alpha_i$, then from the above description, it results to:

$$\alpha_1 + (\alpha_2 - 1) + (\alpha_3 - 1) + \cdots + (\alpha_k - 1) \leq k \quad (1)$$

It follows that

$$\alpha_1 + \alpha_2 + \alpha_3 + \cdots + \alpha_k \leq k + (k-1) = 2k - 1 \quad (2)$$

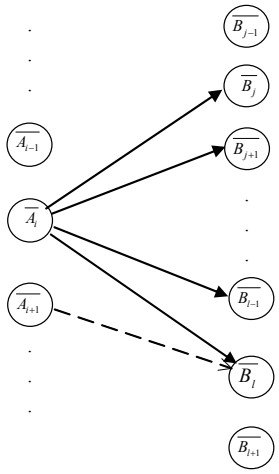

Figure 3. the search space for processor P_i

Therefore the conclusion of Theorem 1 is valid.

D. The second parallel save maxvalue stage

Because the number of the residual block pairs will be at most $2k$ - 1, So we can divide it into k processors, respectively, then each processor have at most two block pairs which are the pair (A_i, B_i). Now we can use the second parallel save maxvalue algorithm to save the current max value of p of each block $\overline{B_i}$ from $\overline{b_{i.e}}$ to $\overline{b_{i.j}}$ in the $Max_{i.j}[t]$, $\overline{B_i} = (\overline{b_{i.1}}, \overline{b_{i.2}}, \cdots, \overline{b_{i.e}})$. The second parallel save maxvalue algorithm can be described as the algorithm 6.

Algorithm 6 The second parallel save maxvalue algorithm

```
1: for all P_i such that 1 ≤ i ≤ k in parallel do
2:   for each one block and t = 0 to 1 do
3:     Max_{i.e}[t] = b_{i.e}.p ;
4:     for j = e − 1 to 1 do
5:       if b_{i.j}.p > Max_{i.(j+1)}[t] then
6:         Max_{i.j}[t] = b_{i.j}.p ;
7:       else
8:         Max_{i.j}[t] = Max_{i.(j+1)}[t] ;
9:       end if
10:    end for
11:  end for
12: end for
13: return Max ;
```

Obviously, the total parallel operation time in this period is $O(N^{1/2})$.

E. The parallel search stage

Now each processor has at most two pair of blocks to search in. Since the memory positions occupied by each one block pairs is completely different with each other, processors P_i and P_j can't have any memory conflicts while

searching. So in this step, the algorithm can perform on EREW.

For any processors P_i, $1 \leq i \leq k$, suppose block pair ($\overline{A_i}, \overline{B_i}$) belongs to P_i , where $\overline{A_i} = (\overline{a_{i.1}}, \overline{a_{i.2}}, \cdots, \overline{a_{i.e}})$, $\overline{B_i} = (\overline{b_{i.1}}, \overline{b_{i.2}}, \cdots, \overline{b_{i.e}})$. The parallel search algorithm can be described as the algorithm 7 and 8. In the algorithm $Bestvalue$ represents the final solutions of KP01.

Algorithm 7 The first step of parallel search algorithm

Require: int t=0 ; int x=0 ; int y=0 ;
1: **for all** P_i such that $1 \leq i \leq k$ **in parallel do**
2: **if** $t \leq 1$ **then**
3: begin the next block pair ;
4: **else**
5: stop ;
6: **end if**
7: x= 1 ; y = 1 ;
8: **if** $\overline{a_{i.x}}.m + \overline{b_{i.y}}.m > c$ **then**
9: $y = y + 1$ **and** Go to Step 15 ;
10: **end if**
11: **if** $\overline{a_{i.x}}.p + Max_{i.y}[t] > Maxvalue_i$ **then**
12: $Maxvalue_i = a_{i.x}.p + Max_{i.y}[t]$;
13: **end if**
14: $x = x + 1$;
15: **if** $x > N$ **or** $y > N$ **then**
16: $t = t + 1$ **and** Go to Step 2 ;
17: **else**
18: Go to Step 8 ;
19: **end if**
20: **end for**
21: **return** $Maxvalue$;

Algorithm 8 The second step of parallel search algorithm

Require: int $Bestvalue = Maxvalue_1$;
1: **for** $i = 2$ to k **do**
2: **if** $Bestvalue < Maxvalue_i$ **then**
3: $Bestvalue = Maxvalue_i$;
4: **end if**
5: **end for**
6: **return** $Bestvalue$;

Obviously, the first phase operation time in this period is $O(N^{1/2})$ and the second phase operation time in this period is $O(k)$. So The parallel search routine can be performed on EREW in time $O(N^{1/2})$.

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

In this subsection, we extend the results stated to the case where $k < O(N^{1/2})$. Assume that the number of processors is $k = e^{1-\varepsilon}$, $0 < \varepsilon \leq 1$. Based on the five main steps in the proposed parallel algorithm, the parallel algorithm in this case is almost the same.

In the generation stage, at first all processors add w_{i+1} and p_{i+1} in parallel to the list $A_i.w$ and $A_i.p$, getting the list A_i^1, $1 \leq i \leq n/2 - 1$. Secondly, for list pairs A_i and A_i^1, perform the parallel merging algorithm, and obtain the sorted list A_{i+1}. In the first parallel save maxvalue stage, we divide both lists A and B into k blocks evenly, then all processors in parallel execute the first save maxvalue algorithm to compute the largest value of member variable p in each block. In the prune stage, all processors perform prune algorithm (the prune algorithm must be modified a little bit according to the available number of processors). After this step, most of all possible k^2 pairs will be deleted, leaving at most $2k$ - 1 block pairs. In the second parallel save maxvalue stage, all processors parallel execute the second save maxvalue algorithm to save the current max value of p of each block $\overline{B_i}$ in the $Max_{i.j}[t]$. At last, in the parallel search stage, the left block pairs are distributed onto the k processors evenly. Finally, all processors perform parallel search algorithm to search for the solution.

IV. Performance analysis and comparisons

A. Performance analysis

The performance of the proposed algorithm in terms of hardware and computation time is readily obtained.

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

$$\sum_{i=1}^{n/4-1} (2 + (i+1)i) + \sum_{i=n/4}^{n/2-1} (2^{i+1}/(2k) + 2^{i+1}/k + \log(2^{n/4}) \log(2^{i+1})) \quad (3)$$

It follows that:

$$\sum_{i=1}^{n/4-1} (2 + (i+1)i) + \sum_{i=n/4}^{n/2-1} (\frac{3 \times 2^{i+1}}{2k} + \frac{n(i+1)}{4}) \quad (4)$$

= $3N/k = 2^{n/4}$

So the execution of the parallel generation algorithm needs $O(3N/k)$ time, that is $O(2^{n/4})$. In the first parallel save maxvalue stage, the execution of this stage needs $O(N/k)$ time, which is $O(2^{n/4})$. In the prune stage, the execution of the prune algorithm needs $O(k)$ time, that is $O(2^{n/4})$. In the second parallel save maxvalue stage, the execution of this stage needs $O(2N/k)$ time that is equal to $O(2^{n/4})$. The parallel search stage takes $O(4N/k + k) = O(2^{n/4})$ time to find a solution in the worst case. At last, the total time complexity in this case is therefore $O(2^{n/4})$, and the cost of the proposed parallel algorithm is $C = T \times k$ = $O(2^{n/4}) \times O(2^{n/4})$ = $O(2^{n/2})$.

Table I
COMPARISONS OF THE PARALLEL ALGORITHMS IN SOLVING THE KP01

Algorithms	Model	Technique type	Processor	Time	Memory
Goldman[4]	hypercube	dynamic programming	O($\frac{cw_{\max}}{w_{\min}\log w_{\max}}$)	O($n\log w_{\max}+\frac{c}{w_{\min}}$)	not mentioned
Lin[8][1]	hypercube	dynamic programming	$k\le c$	O($\frac{nc}{k}\log k$)	O(nc)
Lin[8][2]	EREW PRAM	dynamic programming	$k\le c$	O($\frac{nc}{k}$)	O(nc)
Sanches[12][1]	CREW PRAM	dynamic programming	$k\le w_{min}$	O($\frac{n}{k}(c-w_{\min})$)	O($n+c$)
Sanches[12][2]	EREW PRAM	dynamic programming	$\log n\le k\le n$	O($\frac{n}{k}(c-2w_{\min})$)	O($n+c$)
Ours	EREW PRAM	divide and conquer	O($(2^{n/4})^{1-\varepsilon}$)	O($2^{n/4}(2^{n/4})^{\varepsilon}$)	O($2^{n/2}$)

In the case where the number of processors k satisfies $1\le k<N^{1/2}$, through analyzing the time complexity of the five components of the proposed algorithm, we can know that the total time of our parallel algorithm is $T=O(3N/k)+O(N/k)+O(k)+O(2N/k)+O(4N/k+k)$. Noticing that $1\le k<N^{1/2}$, it can result that $N/k>k$, therefore the total time T is clearly dominated by $O(N/k)$, and the cost is still the same as the previous case which is $C=O(N/k)\times O(k)=O(2^{n/2})$.

B. Performance comparisons

Already many parallel algorithms have been designed for parallel computation for the KP01. Goldman and Trystram [4] introduced an algorithm for the hypercube to the unbounded knapsack problem where the variables are non-negative integers instead of 0 or 1, which is bounded in time by $O(n\log w_{\max}+\frac{c}{w_{\min}})$ with $O(\frac{cw_{\max}}{w_{\min}\log w_{\max}})$ processors. Lin and Storer [8] suggested a direct parallelization for the Bellman's approach. On an EREW PRAM model with $k\le c$ processors, this algorithm requires $O(\frac{nc}{k})$ time and $O(nc)$ space complexity. This algorithm can be simulated by a hypercube with $k\le c$ processors in time $O(\frac{nc}{k}\log k)$ and space $O(nc)$. Sanches et al's parallel algorithm takes $O(\frac{n}{k}(c-w_{\min}))$ time and $O(n+c)$ space to solve the KP01 with $k\le w_{min}$ processors on the CREW PRAM model, and takes $O(\frac{n}{k}(c-2w_{\min}))$ time and $O(n+c)$ space to solve the KP01 with $\log n\le k\le n$ processors on the EREW PRAM model [12]. But in these algorithm in addition to the quantity, the capacity of the objects is also considered in the complexity analysis. For the purpose of clarity, the comparisons of the mentioned parallel algorithms for solving the KP01 are depicted in Table I. Notations. $0\le\varepsilon\le 1$, w_{max} is the maximum of the weights and w_{min} is the minimum of the weights, c is the knapsack capacity. As it is shown in the Table I, dynamic programming has been successfully applied to the KP01 problem, but to our knowledge the parallel method of divide and conquer has not been applied successfully before.

In our parallel algorithm, for any case of $1\le k\le 2^{n/4}$, the cost of the proposed parallel algorithm can be kept unchanged for $O(2^{n/2})$, which is by far both the lowest upper-bound time and without memory conflicts if only quantity of objects is considered in the complexity analysis for the KP01.

V. CONCLUSION

In this paper, based on the two-list algorithm and an EREW PRAM machine with shared memory, we have presented a highly effective parallel algorithm for solving KP01. The proposed algorithm uses $(2^{n/4})^{1-\varepsilon}$ processors and $O(2^{n/2})$ memory to solve KP01 in time $O(2^{n/4}(2^{n/4})^{\varepsilon})$, $0\le\varepsilon\le 1$. The cost of our algorithm is $O(2^{n/2})$, which is an improvement result over the past researches. The theoretic importance of the proposed algorithm is that our parallel algorithm is both the lowest upper-bound time and without memory conflicts if only quantity of objects is considered in the complexity analysis for the 0-1 knapsack problem.

Another important thing that needs to be mentioned is, although the proposed algorithm need shared memory with a large number of computers, with the advent of the supercomputer and parallel programming language, we can rely on this kind of algorithms to solve the large scale of knapsack problems. But still there are some questions like how to program the proposed algorithm and run it on the supercomputer, and to find out that under what circumstances in practice the parallel method of divide and conquer have better running time than the parallel method of dynammic programming, and the answers to these questions are still a worthwhile work for the future.

REFERENCES

[1] H. Kellerer, U. Pferschy, D. Pisinger, Knapsack Problems, Springer, 2004.

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

[3] R.E. Bellman, Dynamic Programming, Princeton University Press, Princeton, 1957.

[4] A. Goldman, D. Trystram, An efficient parallel algorithm for solving the knapsack problem on hypercubes, Journal of Parallel and Distributed Computing, 2004, 64(11): 1213-1222.

[5] P.S. Gopalakrishnam, I.V. Ramakrishnam, L.N. Kanal, Parallel approximate algorithms for the 0-1 knapsack problem, in: Proceedings of International Conference on Parallel Processing, 1986, pp. 444-451.

[6] G.A.P. Kindervater, J.K. Lenstra, An introduction to parallelism in combinatorial optimization, Discrete Applied Mathematics, 1986, 14(2): 135-156.

[7] J. Lee, E. Shragowitz, S. Sahni, A hypercube algorithm for the 0/1 knapsack problem, Journal of Parallel and Distributed Computing, 1988, 5(4): 438-456.

[8] J. Lin, J. Storer, Processor efficient hypercube algorithm for the knapsack problem, Journal of Parallel and Distributed Computing, 1991, 13(3): 332-337.

[9] D. El baz, M. Elkihel, Load balancing methods and parallel dynamic programming algorithm using dominance techniques applied to the 0-1 knapsack problem, Journal of Parallel and Distributed Computing, 2005, 65: 74-84.

[10] S. Teng, Adaptive parallel algorithms for integral knapsack problems, Journal of Parallel and Distributed Computing, 1990, 8(4): 400-406.

[11] W. Loots t and T. H. C. Smith, A Parallel Algorithm for the 0-1 Knapsack Problem, International Journal of Parallel Programming, 1992. 21(5): 349-350.

[12] C.A. Sanches, N.Y. Soma, H. Yanasse, Parallel time and space upper-bounds for the subset-sum problem, Theoretical Computer Science, 2008, 407(1-3): 342-348.

[13] E. Horowitz, S. Sahni, Computing partitions with applications to the knapsack problem, Journal of ACM, 1972, 21(2): 277-292.

[14] Karnin E D, A parallel algorithm for the knapsack problem, IEEE Transactions on Computers, 1984, 33(5): 404-408.

[15] Amirazizi H R, Hellman M E, Time-memory-processor trade-offs, IEEE Transactions on Information Theory, 1988, 34(3): 502-512.

[16] Ferreira A G, A parallel time/hardware tradeoff $T \cdot H = O(2^{n/2})$ for the knapsack problem, IEEE Transactions on Computers, 1991, 40(2): 221-225.

[17] Akl S G, Optimal parallel merging and sorting without memory conflicts, IEEE Transactions on Computers, 1987, 36(11): 1367-1369.

[18] F B Chedid, An optimal parallelization of the two-list algorithm of cost O($2^{n/2}$), Parallel Computing, 2008, 34(1): 63-65.

[19] C A Sanches, N Y Soma, H H Yanasse, An optimal and scalable parallelization of the two-list algorithm for the subset-sum problem, European Journal of Operational Research, 2007, 176(2): 870-879.

[20] S. Rajasekaran, Efficient parallel hierarchical clustering algorithms, IEEE Transactions on Parallel and Distributed Systems, 2005, 16(6): 497-502.

[21] Y.R. Wang, S.J. Horng, An O(1) time algorithm for the 3D Euclidean distance transform on the CRCW PRAM model, IEEE Transactions on Parallel and Distributed Systems, 2003, 14(10): 973-982.

[22] A.K. Datta, R.K. Sen, O($log4n$) time parallel maximal matching algorithm using linear number of processors, Parallel Algorithms and Applications, 2004, 19(1): 19-32.

[23] E.W. Mayr, Parallel approximation algorithms, in: Proceedings of International Conference on Fifth Generation Computer Systems, 1988, pp. 542-551.

[24] B F Wang, S C Ku, K H Shil, Cost-optimal parallel algorithms for tree bisector and related problems, IEEE Transactions on Parallel and Distributed Systems, 2001, 12(9): 888-898.