ISSN 1000-9825, CODEN RUXUEW
Journal of Software, Vol.18, No.6, June 2007, pp.1319−1327
DOI: 10.1360/jos181319

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel/Fax: +86-10-62562563

背包类问题的并行 $O(2^{5n/6})$时间-空间-处理机折衷*

李肯立[1+], 赵 欢[1], 李仁发[1], 李庆华[2]

[1](湖南大学 计算机与通信学院,湖南 长沙 410082)

[2](华中科技大学 计算机学院,湖北 武汉 430074)

A Parallel Time-Memory-Processor Tradeoff $O(2^{5n/6})$ for Knapsack-like NP-Complete Problems

LI Ken-Li[1+], ZHAO Huan[1], LI Ren-Fa[1], LI Qing-Hua[2]

[1](School of Computer and Communication, Hu'nan University, Changsha 410082, China)

[2](School of Computer, Huazhong University of Science and Technology, Wuhan 430074, China)

+ Corresponding author: Phn: +86-731-8821715, Fax: +86-731-8821715, E-mail: LKL510@263.net

Li KL, Zhao H, Li RF, Li QH. A parallel time-memory-processor tradeoff $O(2^{5n/6})$ for knapsack-like NP-complete problems. *Journal of Software*, 2007,18(6):1319−1327. http://www.jos.org.cn/1000-9825/18/1319.htm

Abstract: A general-purpose parallel three-list six-table algorithm that can solve a number of knapsack-like NP-complete problems is developed in this paper. This kind of problems includes knapsack problem, exact satisfiability problem, set covering problem, etc. Running on an EREW PRAM model, The proposed parallel algorithm can find a solution of these problems of size n in $O(2^{7n/16})$ time, with $O(2^{13n/48})$ space and $O(2^{n/8})$ processors, resulting in a time-space-processor tradeoff of $O(2^{5n/6})$. The performance analysis and comparisons show that it is both work and space efficient, and thus is an improved result over the past researches. Since it can break greater variables knapsack-based cryptosystems and watermark, the new algorithm has some cryptanalytic significance.

Key words: NP-complete problem; parallel algorithm; time-space-processor tradeoff; knapsack problem

摘 要: 将串行动态二表算法应用于并行三表算法的设计中,提出一种求解背包、精确的可满足性和集覆盖等背包类 NP 完全问题的并行三表六子表算法.基于 EREW-PRAM 模型,该算法可使用 $O(2^{n/8})$的处理机在 $O(2^{7n/16})$的时间和 $O(2^{13n/48})$的空间求解 n 维背包类问题,其时间-空间-处理机折衷为 $O(2^{5n/6})$.与现有文献的性能对比分析表明,该算法极大地提高了并行求解背包类问题的时间-空间-处理机折衷性能.由于该算法能够破解更高维数的背包类公钥和数字水印系统,其结论在密钥分析领域具有一定的理论和实际意义.

关键词: NP 完全问题;并行算法;时间-空间-处理机折衷;背包问题

中图法分类号: TP301 文献标识码: A

* Supported by the National Natural Science Foundation of China under Grant No.60603053 (国家自然科学基金); the Key Project of Ministry of Education of China under Grant No.105128 (国家教育部重点基金)

Received 2005-04-12; Accepted 2006-01-16

1 Introduction

Every NP-complete problem can be solved in $O(2^n)$ time by exhaustive search, but this complexity becomes prohibitive when n exceeds 60 or 70. Assuming that $NP\neq P$, we cannot hope to find algorithms whose worst-case complexity is polynomial, but it is both theoretically interesting and practically important to determine whether substantially faster algorithms exist. In this paper we describe a parallel algorithm which can solve the knapsack problem. But owing to the work done by Schoreppel and Shamir[1], our proposed algorithm actually can solve a fair number of NP-complete problems including knapsack, partition, exact satisfiability, set covering, hitting set, disjoint domination in graphs, etc. Although the proposed algorithm is a versatile algorithm that can solve the above kind of NP-complete problems, to make this algorithm more easily understood, we only take the knapsack problem as the representative. As to the details on how our proposed algorithm can be applied to solve other NP-complete problems in this kind, one can refer to Ref.[1].

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

$$\sum_{i=1}^{n} w_i x_i = M \tag{1}$$

This problem was proved to be NP-complete[2] and, unless $NP=P$, its complexity is exponential in n. 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].

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. Based on this algorithm, Schrowppel and Shamir[1] 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. They also showed their algorithm can solve the above knapsack-like NP-complete problems. Using unbalanced four tables, an adaptive algorithm is presented in Ref.[6]. Although the above algorithm is by far the most efficient algorithm to solve the knapsack-like problems 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 complexity of problems in all research areas[7−15], most of which are based on CREW (concurrent read exclusive write) PRAM (parallel random access machine). Karnin[7] proposed a parallel algorithm that parallelizes the generation routine of the *two-list four-table* algorithm. In his algorithm the knapsack problem could be solved with $O(2^{n/6})$ processors and $O(2^{n/6})$ memory cells in $O(2^{n/2})$ time. Amirazizi and Helman[8] were the first to show that parallelism could accelerate to solve larger instances of this problem. Their algorithm runs in $O(n2^{\alpha n})$ time, $0\leq\alpha\leq 1/2$, by allowing $O(2^{(1-\alpha)n/2})$ processors to concurrently access a list of this same size. Amirazizi and Helman[8] also present a more feasible *Time-Space-Processor* (*TSP*) model for evaluation of the performance of different algorithms for solution of knapsack-like NP-complete problems. In 1991, Ferreira[9] proposed a brilliant parallel algorithm that solves the knapsack problem of size n in time $T=O(n(2^{n/2})^{\varepsilon})$, $0\leq\varepsilon\leq 1$, when $P=O((2^{n/2})^{1-\varepsilon})$ processors and $S=O(2^{n/2})$ are available. Chang *et al.*[10] presented another parallel algorithm where the requirement of the sharing memory is $O(2^{n/2})$ by using $O(2^{n/8})$ processors to solve the knapsack problem still in $O(2^{n/2})$ time. Thereafter, in 1997, based on Chang *et al.*'s parallel algorithm, Lou and Chang[11] successfully parallelize the second stage of the *two-list* algorithm. Regretfully, it is independently found in Refs.[12,13] that the analysis of the complexity of the Chang *et al.*'s algorithm was wrong, which invalidate the results of Lou and Chang[11]. Except pointing out the wrong in literature[10], we also proposed a CREW-PRAM cost-optimal parallel algorithm[12], and thereafter, a cost-optimal algorithm without memory conflicts was further presented in Ref.[14].

However, because the memories required in both of these two cost-optimal parallel algorithms are still $O(2^{n/2})$, it makes the available memory cells a bottleneck when using these algorithms to break practical knapsack-based cryptosystem. When explaining the open problems existed in this kind of NP-complete problems, G. Woeginger recently concludes that the space is more important than the time[16]. Therefore, to further reduce the required memory units for the solution of knapsack-like problems is still valuable.

To reach this goal, based on Ferreira's CREW parallel *three-list* algorithm[15] and the *two-list* 2*k-table* serial algorithm[17], we propose a parallel *three-list six-table* algorithm in this paper. The novel properties of the proposed algorithm are:

(i) This algorithm can solve knapsack-like NP-complete problems in $O(2^{7n/16})$ time, $O(2^{13n/48})$ shared memory units when $O(2^{n/8})$ processors are available. The *Time-Space-Processor* tradeoff of this algorithm is only $O(2^{5n/6})$, which is considerably better than those of all algorithms published so far.

(ii) It can be performed on an EREW PRAM machine model, and thus is a totally without memory conflicts algorithm for the knapsack-like problems. Furthermore, the algorithm is completely practical in the sense that it is easy to program and its overhead is small.

The rest of this paper is organized as follows. Section 2 explains the parallel *three-list* algorithm, on which the proposed algorithm is based. The proposed parallel algorithm is described in Section 3. Then, in Section 4, the performance analysis and comparison follow. Finally, some concluding remarks and some future research directions in this field are given in Section 5.

2 The Parallel Three-List Algorithm

In 1995, Ferreira presented a parallel *three-list* algorithm, which is based on a CREW PRAM model[15]. The number of processor, time complexity, and space requirements in it are $O(2^{\beta n})$, $O(n2^{(1-\varepsilon/2-\beta)})$, $O(n2^{\varepsilon n/2})$, $0<\varepsilon<1$, $0\le\beta\le1-\varepsilon/2$, respectively. It is viewed as an important breakthrough in the research of knapsack-like problems, for it can solve the knapsack-like problems in a way of both work and space effective[15]. Because our algorithm is based on this algorithm, we introduce it.

Algorithm 1. The *Three-list* algorithm.

Generation stage

1. Divide W into three parts: $W_1=(w_1,w_2,\ldots,w_{7n/16})$, $W_2=(w_{7n/16+1},w_{7n/16+2},\ldots,w_{14n/16})$, $W_3=(w_{14n/16+1},\ldots,w_n)$.
2. Form all possible subset sums of W_1, W_2, then sort them in a nondecreasing order and store them as $A=[A_1,A_2,\ldots,A_{2^{\frac{7n}{16}}}]$ and $B=[B_1,B_2,\ldots,B_{2^{\frac{7n}{16}}}]$, respectively.
3. Form all possible subset sums of W_3, and store them as $C=[C_1,C_2,\ldots,C_{2^{\frac{n}{8}}}]$.

Search stage

1. For all C_m in C where $1\le m\le 2^{n/8}$
2. C_i execute the binary search over $A+B$.
3. If a solution is found: then stop, output the solution; else: output that there is no solution.

The time complexity of this algorithm is $O(n\times2^{9n/16})$, and the needed memory unit is $O(2^{7n/16})$[15]. Based on its serial algorithm, Ferreira's parallel algorithm is very direct. It runs on a CREW model, as shown in Fig.1[15]. The number of processors is $P=2^{n/8}$. The subset sums in lists A and B which hold $2^{7n/16}$ subset sums respectively are stored in the shared memory. And each processor P_i ($1\le i\le P$), which holds the subset sum C_i, executes a "virtual" binary search on the list $A+B$ to make sure whether $A[j]+B[l]=M-C_i$ is satisfied, $1\le j,l\le 2^{7n/16}$. The parallel *three-list* algorithm consists of the following three main steps[15]:

Algorithm 2. Parallel *three-list* algorithm.

for all P_m where $1\le m\le 2^{n/8}$ **do**

1. Generation of the three lists A, B and C
2. Sorting of the two lists
3. Binary search over $A+B$

end

The total time needed in this algorithm is bound by $O(n\times 2^{7n/16})$, and the space requirements are $O(2^{7n/16})$[15].

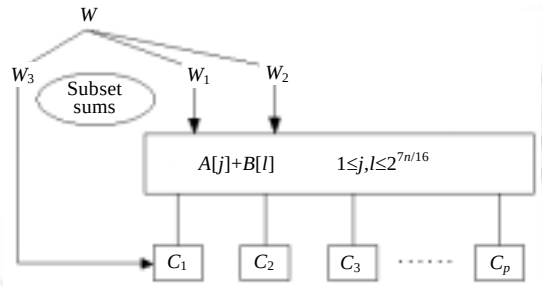

Fig.1 The parallel *three-list* algorithm ($P=2^{n/8}$)

3 The Proposed Parallel Algorithm

Although Ferreira's above algorithm is considered as a main breakthrough for the researches on the knapsack problem, it still have an obvious shortcoming, for it has a $O(n\times 2^n)$ *TSP* tradeoff, which is a little greater than that of the recent parallel algorithms[12,14] by a factor n. To overcome this shortcoming, we redesign the two main stages of the parallel *three-list* algorithm. Inspired by the idea used in serial algorithm[17], in the list generation stage, we introduce six tables to produce two ordered list A and B dynamically. By doing so, we can reduce the space complexity from $O(2^{7n/16})$ to $O(2^{13n/48})$. While in the list search stage, we replace the matrix search way in Ref.[15] with the *two-list* like search algorithm, which is more simple and able to reduce the time needed by a factor $O(n)$ in the search stage.

In our proposed algorithm, each of the two lists stored in the shared memory has a size of $O(2^{7n/16})$, whose elements will be dynamically generated one by one, by using only $O(2^{13n/48})$ shared memory units. Now consider the two stages of the algorithm. For convenience, we first introduce the algorithm used in the search stage.

3.1 The search stage

Now we use the *two-list* like search to fulfill the list search stage. Suppose the two sorted lists A and B exist before the following Algorithm 3 executes. Because each processor holds the subset sum element $C[m]$ in its local memory, $1\le m\le 2^{n/8}$. We can use the following *two-list* like search algorithm to make sure that for any $C[m]$, $1\le m\le 2^{n/8}$, whether there exist $A[i]$ and $B[j]$, $1\le i,j\le 2^{7n/16}$, such that the formula $A[i]+B[j]+C[m]=M$ are satisfied.

Algorithm 3. Parallel *two-list* like search algorithm.

The subset sums in list A are sorted in an increasing order, while the sums in list B are sorted in a decreasing order

for all processors P_m where $1\le m\le 2^{n/8}$ **do**

1. $i=1$, $j=1$.
2. **if** $A[i]+B[j]=M-C[m]$, **then stop**: A solution is found, and **write** the result into the shared memory.
3. **if** $A[i]+B[j]<M-C[m]$, **then** $i=i+1$; **else** $j=j+1$.

4. **if** $i>2^{7n/16}$ **or** $j>2^{7n/16}$ **then stop**: there is no solution.

5. **goto** step 2.

end

Lemma 1. The time needed to perform Algorithm 3 is at most $2\times2^{7n/16}$.

Proof. The condition that the loop ends shows that once the variables i or j is greater than $2^{7n/16}$, the algorithm terminates. While for each computation step, one of the above two variables must increase by 1. So it is obvious that the maximum of the needed time to perform Algorithm 3 is $2\times2^{7n/16}$. □

3.2 The three-list generation stage

Since each element in list C is stored in the local memory of each processor, and it is easy to produce it. We only discuss how to produce all elements of lists A and B stored in the shared memory. Note that in the list search Algorithm 3, each processor accesses the elements of the sorted lists A and B sequentially, and thus there is no need to store all the possible subset sums of A and B simultaneously in the shared memory—what we need is the ability to generate them quickly (on-line, upon request) in the sorted order. So if we generate the two ordered lists dynamically, the needed space will be reduced greatly. To implement this key idea, we explore the thoughts in Ref.[17] where $2k$ tables are used to dynamically produce two sorted lists in serial. Here we use *six tables* T_1, T_2, T_3, and T_4 , T_5, T_6, to dynamically produce the two sorted lists A and B, where T_1 includes all possible subset sums of knapsack entries $W_{11}=(w_1,w_2,\ldots,w_{7n/48})$,…,$T_3$ includes all subset sums of $W_{13}=(w_{14n/48+1},w_{14n/48+2},\ldots,w_{21n/48})$, and T_4 includes all sums of $W_{21}=(w_{21n/48+1},w_{21n/48+2},\ldots,w_{28n/48})$,…,$T_6$ includes all subset sums of $W_{23}=(w_{35n/48+1},w_{35n/48+2},\ldots,w_{42n/48})$. Let $e=2^{7n/48}$, and mark $T_i=(t_{i1},t_{i2},\ldots,t_{ie})$, $i=1,\ldots,6$. At first we introduce how to dynamically produce the two sorted lists A and B with these six tables in serial by only using $O(2^{7n/48})$ space.

3.2.1 Production of the two lists in sequential

We focus on the procedures to generate list A because the process to generate list B is similar. As shown in Fig.2, we first sort all sums in T_1 in an increasing order, and then use one priority queue Q_1. At beginning Q_1 stores all pairs of the first (T_1) and all elements t_{2i}. It can be updated by two operations *deletion* and *insertion*, which enables arbitrary insertions and deletions to be done in logarithmic time of the length of the queue, and makes the pair with the smallest $t_{1i}+t_{2j}$ sum accessible in constant time. The following algorithm is designed to dynamically produce all sums of T_1+T_2 in an increasing order.

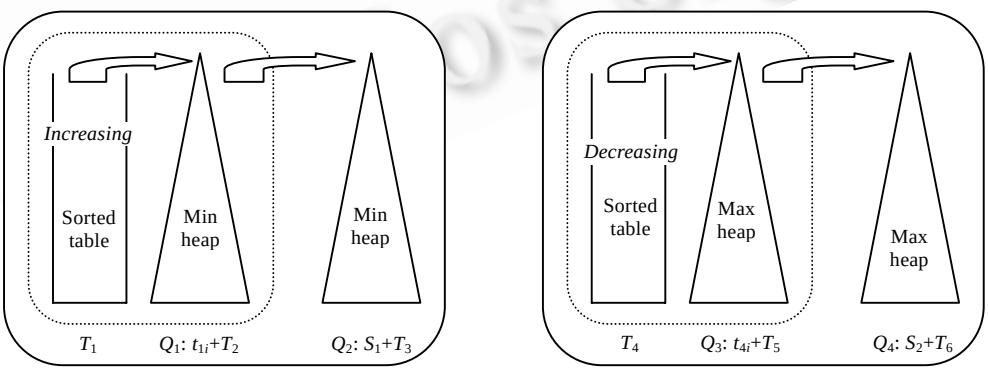

Fig.2 Structure of the six tables to produce the two lists

Algorithm 4. Algorithm for dynamically generating all sums of T_1+T_2 in an increasing order.

Tables $T_1=(t_{11},t_{12},\ldots,t_{1e})$, $T_2=(t_{21},t_{22},\ldots,t_{2e})$ are given

(1) **sort** T_1 into an increasing order;

insert into Q_1 all the pairs (first (T_1), t_{2i}) for all $t_{2i}\in T_2$;

(2) **repeat** until Q_1 becomes empty.

(t_1,t_2)←pair with the smallest t_1+t_2 sum in Q_1;

S_1←(t_1+t_2)

if S_1 is needed and used for the objectivity of computation, **delete** (t_1,t_2) from Q_1;

if the successor t_1^1 of t_1 in T_1 is defined, **insert** (t_1^1,t_2) into Q_1;

Lemma 2. If one element in T_1+T_2 is produced at any time, the required time is $O(7n/48)$; if all $2^{7n/24}$ elements are required, the time is correspondingly $O(n2^{7n/24})$.

Proof. According to the theory of heap[17], one time of deletion and insertion on the heap can be performed with logarithmic time of the size of the heap. Since the heap constructed in Algorithm 4 has a size of $2^{7n/48}$ and the combination T_1+T_2 has $2^{7n/24}$ elements, it validates the results of Lemma 2. □

Notice that in Algorithm 4, to produce all $2^{7n/24}$ sums in T_1+T_2 one by one, only $O(2^{7n/48})$ space is needed. Now we go a little further to produce all $2^{7n/16}$ sums in $T_1+T_2+T_3$ still with $O(2^{7n/48})$ space cells. The procedure to do so is exactly similar to Algorithm 4. We use another priority queue Q_2 which also has a length of $O(2^{7n/48})$. Q_2 stores all pairs of the first (T_1+T_2) and all elements t_{3i} in T_3. It can be updated by *deletion* and *insertion*, and it makes the pair with the smallest ($t_{1i}+t_{2j}$)+t_{3l} sum accessible in constant time.

Algorithm 5. Algorithm for dynamically generating all sums of (T_1+T_2)+T_3 in an increasing order.

Tables T_3=(t_{31},t_{32},…,t_{3e}) are given and all pairs in T_1+T_2 can be obtained by their increasing order dynamically.

(1) **insert** into Q_2 all the pairs (S_1,t_{3l}) for all $t_{3l}\in T_3$ where S_1 denotes the least sum in Q_1;

(2) **repeat** until Q_2 becomes empty.

(S_1,t_3)←pair with the smallest S_1+t_3 sum in Q_2;

S←(S_1+t_3);

if S is needed and used for the objectivity of computation, **delete** (S_1,t_3) from Q_2;

if the successor S_1^1 of S_1 in T_1+T_2 is defined, **insert** (S_1^1,t_3) into Q_2;

Lemma 3. The required time to produce one sum and all sums in $T_1+T_2+T_3$ in an increasing order is respectively $O(7n/48)$ and $O(n2^{7n/16})$ at the condition of the initial heap for queue Q_2 having been constructed.

Proof. Note that the number of sums in (T_1+T_2)+T_3 is $|T_1|\times|T_2|\times|T_3|=(2^{7n/48})^3=2^{7n/16}$. Following the proof of Lemma 2, the conclusions here are obviously correct. □

Therefore, by using Algorithms 4 and 5, we can dynamically obtain all sums in $T_1+T_2+T_3$ in an increasing order with only $O(2^{7n/48})$ memory units. To produce all elements dynamically in a decreasing order, the procedure is almost the same as the above procedure, except that we have to sort the elements in T_4 in a decreasing order, and use two *max* heaps for the priority queues Q_3 and Q_4.

3.2.2 Producing the two lists in parallel

Refering to Fig.1, it seems that it is possible for all processors to use the same priority queues to produce all the needed elements in $T_1+T_2+T_3$ and $T_4+T_5+T_6$, and thus $O(2^{7n/48})$ shared memory units are enough for the parallel case. However, $O(2^{7n/48})$ space cells indeed do not fit the parallel case. When Algorithm 3 starts to perform, at first all processors P_m need the sum pair $A[1]$ and $B[1]$ to make sure whether $A[1]+B[1]=M-C[m]$, $1\le m\le 2^{n/8}$. But after that time, the value $C[m]$ each processor holds may be different from each other. Therefore, to make the search algorithm perform successfully, we must prepare two queues (heaps) for each processor. As a result, in parallel case, the shared memory must have more memory units than that needed in sequential case.

By combining the discussions in 3.1 and 3.2.1 into a whole, we get the final parallel *three-list six-table* algorithm.

Algorithm 6. An EREW based parallel *three-list six-table* algorithm for knapsack-like problems.

A knapsack instance including $W=(w_1,w_2,\ldots,w_n)$ and M is given

for all processors P_m where $1\le m\le 2^{n/8}$ **do**

1. Generate list C and six tables T_1, T_2, T_3 and T_4, T_5, T_6, and sort T_1 and T_4 in parallel.
2. Construct two *min* heaps for queues Q_{1m}, Q_{2m}, and two *max* heaps for queues Q_{3m} and Q_{4m}.
3. Perform Algorithm 4.
4. Perform Algorithm 5.
5. Perform two-list like search algorithm (Algorithm 3).

end

Theorem 1. n-variable knapsack-like problems can be solved on EREW model in $O(2^{7n/16})$ time when $O(2^{n/8})$ processors and $O(2^{13n/48})$ shared memory units are available.

Proof. With $2^{n/8}$ processors, producing list C and four even tables can be finished in n and $4\times 2^{n/48}$ time respectively, while the two tables T_1 and T_4 can be generated and sorted in $4\times 2^{n/48}$ time through the parallel merging generation algorithm in Ref.[18] without any memory conflicts. It will take $4\times 2^{n/48}$ time for each processor to construct four heaps. Following Lemmas 3, to perform Algorithm 3, each processor need element pair $A[i]$ and $B[j]$. $A[i]$ comes from heap Q_2 and needs $7n/48$ time, and finding the updating elements for heaps Q_2 (from heaps Q_1) will take another $7n/48$ time. Since there are $2\times 2^{7n/16}$ elements in lists A and B, the total needed time is

$$n+4\times 2^{n/48}+4\times 2^{n/48}+2\times 2^{7n/16}\times\left(\frac{7n}{48}\right)^2=O\left(\frac{49n^2}{2304}\times 2^{7n/16}\right) \tag{2}$$

Compared with the exponential factors, the low polynomial factor has little impact on the time complexity and thus is usually omitted in the analysis of the algorithms on knapsack-like problem[7−10,15]. So the time complexity of the proposed parallel algorithm is $O(2^{7n/16})$. As for space complexity, since there are $2^{n/8}$ processors, and each of them needs $4\times 2^{7n/48}$ for the construction of heaps, the total space requirements are $O(2^{13n/48})$. To avoid memory conflicts, at first, we copy the knapsack variables for each processor. Thereafter, each processor accesses and updates its own heaps, so it is obvious that all processors have no memory conflicts. □

4 Performance Analysis and Comparison

We adopt the time-space-processor (*TSP*) tradeoff as the criterion of evaluation of relevant algorithms[8]. Karnin's parallel algorithm takes $O(n2^{n/2})$ time to solve the knapsack problem with $O(2^{n/6})$ processors and $O(2^{n/6})$ shared space, resulting in a *TSP* tradeoff of $O(2^{5n/6})$[9]. The *TSP* tradeoff of Ferreira's parallel *three-list* algorithm in Ref.[15] is $O(n2^n)$. The parallel algorithm proposed by Amirazizi and Helman[8] runs in $O(n2^{\alpha n})$ time, $0\le\alpha\le 1/2$, by allowing $O(2^{(1-\alpha)n/2})$ processors to concurrently access a list of the same size, hence the *TSP* tradeoff of this algorithm is also $O(n2^n)$. Ferreira's parallel *one-list* algorithm in Ref.[9] solves the knapsack problem in time $T=O(n(2^{n/2})^{\varepsilon})$, $0\le\varepsilon\le 1$, when $P=O((2^{n/2})^{1-\varepsilon})$ processors and $S=O(2^{n/2})$ are available. Therefore, it results in an $O(n2^n)$ *TSP* tradeoff. The *TSP* tradeoff of Chang *et al.*'s parallel algorithm[10] is $O(2^{9n/8})$, while the parallel algorithm Lou and Chang presented bears the same performance as Chang *et al.*'s algorithm[11−13]. In addition, both of the algorithms in Refs.[12,14] have $O(2^n)$ *TSP* tradeoff.

In our parallel *three-list six-table* algorithm, following Theorem 1, we can get a *TSP* tradeoff of $O(49n^2/2304\times 2^{n/8}\times 2^{13n/48}\times 2^{7n/16})=O(2^{5n/6})$. Among all algorithms that can be found in literatures, the *TSP* tradeoff of the algorithm proposed by Karnin[7] is the lowest, which is also $O(n2^{5n/6})$. However, it has obvious defects that it can't reduce the execution time even in parallel, for it must take $O(n2^{n/2})$ time to solve the knapsack-like problems. Although Ferreira's parallel *three-list* algorithm is the first algorithm that can solve the knapsack-like problem with

less than $O(2^{n/2})$ time when the available hardware is also less than $O(2^{n/2})$, it does little in reducing the overall performance tradeoff because of its $O(n2^n)$ *TSP* tradeoff. In spite of the fact of our proposed algorithm is not cost optimal, it is both work and memory efficient. Moreover, our algorithm is based on EREW-PRAM model, so it can avoid memory conflicts when different processors access the shared memory.

For the purpose of clarity, the comparisons of the mentioned parallel algorithms for solving the knapsack-like problems are depicted in Table 1. It is obvious that our parallel algorithm outtakes undoubtedly other parallel algorithms in the overall performance.

Table 1 Comparisons of the parallel algorithms for solving the knapsack-like problems

Algorithms	Model	Processor	Time	Memory	*TSP* tradeoff
1[7]	CREW	$O(2^{n/6})$	$O(2^{n/2})$	$O(2^{n/6})$	$O(2^{5n/6})$
2[8]	CREW	$O(2^{(1-\alpha)n/2})$	$O(2^{\alpha n})$	$O(2^{(1-\alpha)n/2})$	$O(2^n)$
3[15]	CREW	$O(2^{\beta n})$	$O(2^{(1-\varepsilon/2-\beta)n})$	$O(2^{\varepsilon n/2})$	$O(2^n)$
4[9]	CREW	$O(2^{(1-\varepsilon)n/2})$	$O(2^{\varepsilon n/2})$	$O(2^{n/2})$	$O(2^n)$
5[10]	CREW	$O(2^{n/8})$	$O(2^{n/2})$	$O(2^{n/2})$	$O(2^{9n/8})$
6[11]	CREW	$O(2^{n/8})$	$O(2^{n/2})$	$O(2^{n/2})$	$O(2^{9n/8})$
7[12]	CREW	$O((2^{n/4})^{1-\varepsilon})$	$O(2^{n/4}(2^{n/4})^{\varepsilon})$	$O(2^{n/2})$	$O(2^n)$
8[14]	EREW	$O((2^{n/4})^{1-\varepsilon})$	$O(2^{n/4}(2^{n/4})^{\varepsilon})$	$O(2^{n/2})$	$O(2^n)$
Ours	EREW	$O(2^{n/8})$	$O(2^{7n/16})$	$O(2^{13n/48})$	$O(2^{5n/6})$

Notations: $0\le\varepsilon\le1$, $0\le\alpha\le1/2$, $0\le\beta\le1-\varepsilon/2$.

5 Conclusions

Inspired by the ideas in parallel *three-list* algorithm[15] and serial *two-list* 2*k*-table algorithm[17], we propose a new parallel *three-list six-table* algorithm for solving the knapsack-like problems. Through dynamically producing the elements of the two lists with four priority queues and two sorted tables, we dramatically reduce the space requirements from $O(2^{7n/16})$ in parallel *three-list* algorithm[15] to $O(2^{13n/40})$. Moreover, the memory conflicts are also avoided by leaving different memory address segment for different processors, permitting the algorithm to be able to perform on EREW machine model. Performance comparison on the *TSP* criterion shows our proposed algorithm greatly outweighs the parallel algorithms presented by far, and thus it is an improved result over the past researches on parallel solution of the knapsack-like NP-complete problems. Since it can solve problems that are almost 1.6 times as big as those handled by the previous algorithms, it may have some importance in research of cryptosystem.

However, for NP-complete problems, we know that, unless NP = P, some exponential factor should appear in parallel solutions, either as the time complexity, the number of processors used or even as the memory requirements[19]. Therefore, even modern supercomputer can break 100-variable knapsack cryptosystem, but how about 120-variable or more? Perhaps, the DNA-based parallel computation may be a way to go out[20], so one of our future work is on how to combine the ideas in designing traditional algorithms and DNA methods to obtain new DNA algorithms; other possible work may be on how to design distributed algorithms on the grid sources to solve this kind of hard problems.

References:

[1] Schroeppel R, Shamir A. A $T=O(2^{n/2})$, $S=O(2^{n/4})$ algorithm for certain NP-complete problems. SIAM Journal Computing, 1981, 10(3): 456−464.

[2] Chor B, Rivest RL. A knapsack-type public key cryptosystem based on arithmetic in finite fields. IEEE Trans. on Information Theory, 1988,34(5):901−909.

[3] Laih CS, Lee JY, Harn L, Su YK. Linearly shift knapsack public-key cryptosystem. IEEE Journal Selected Areas Communication, 1989,7(4):534−539.

[4] Zhang B, Wu HJ, Feng DG, Bao F. Cyptanalysis of a knapsack based two-lock cryptosystem. In: Proc. of the ACNS 2004. LNCS 3089, 2004. 303−309.

[5] Horowitz E, Sahni S. Computing partitions with applications to the knapsack problem. Journal of the ACM, 1974,21(2):277−292.

[6] Li KL, Li QH, Dai GM. An adaptive algorithm for the knapsack problem. Journal of Computer Research and Development, 2004, 12(7):1024−1029 (in Chinese with English abstract).

[7] Karnin ED. A parallel algorithm for the knapsack problem. IEEE Trans. on Computer, 1984,33(5):404−408.

[8] Amirazizi HR, Hellman ME. Time-Memory-Processor trade-offs. IEEE Trans. on Information Theory, 1988,34(3):505−512.

[9] Ferreira AG. A parallel time/hardware tradeoff $T{\cdot}H{=}O(2^{n/2})$ for the knapsack problem. IEEE Trans. on Computer, 1991,40(2): 221−225.

[10] Chang HK, Chen JJ, Shyu SJ. A parallel algorithm for the knapsack problem using a generation and searching technique. Parallel Computing, 1994,20(2):233−243.

[11] Lou DC, Chang CC. A parallel two-list algorithm for the knapsack problem. Parallel Computing, 1997,22(14):1985−1996.

[12] Li KL, Li QH, Jiang SY. An optimal parallel algorithm for the knapsack problem. Journal of Software, 2003,14(5):891−896 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/14/891.htm

[13] Aanches CA, Soma NY, Yanasse HH. Comments on parallel algorithms for the knapsack problem. Parallel Computing, 2002, 28(10): 1501−1505.

[14] Li QH, Li KL, Li RF. Optimal parallel algorithm for the knapsack problem without memory conflicts. Journal of Computer Science and Technology, 2004,19(6):760−768.

[15] Ferreira AG. Work and memory efficient parallel algorithms for the knapsack problem. Int'l Journal of High Speed Computing, 1995,4(7):595−606.

[16] Woeginger GJ. Space and time complexity of exact algorithms: Some open problems. In: Downey R, ed. Proc. of the IWPEC 2004. LNCS 3162, Berlin, Heidelberg: Springer-Verlag, 2004. 281−290.

[17] Vyscoc J. An $O(n^{\lg k}{\cdot}2^{n/2})$ time and $O(k{\cdot}2^{n/k})$ space algorithm for certain NP-Complete problem. Theoretical Computer Science, 1987, 51(1,2):221−227.

[18] Akl SG. Optimal parallel merging and sorting without memory conflicts. IEEE Trans. on Computer, 1987,36(11):1367−1369.

[19] Cheng GL. The Design and Analysis of Parallel Algorithms. Beijing: Higher Education Press, 2002. 35−37 (in Chinese).

[20] Chang WL, Guo M, Ho M. Molecular solutions for the subset-sum problem on DNA-based supercomputing. Biosystem, 2004, 73(2):117−130.

附中文参考文献:

[6] 李肯立,李庆华,戴光明.背包问题的一种自适应算法.计算机研究与发展,2004,12(7):1024−1029.

[12] 李庆华,李肯立,蒋盛益.背包问题的最优并行算法.软件学报,2003,14(5):891−896. http://www.jos.org.cn/1000-9825/14/891.htm

[19] 陈国良.并行算法的设计与分析.北京:高等教育出版社,2002.35−37.

LI Ken-Li was born in 1971. He received his Ph.D. in Computer Science from HUST in 2003. He is now an associate professor at the Hu'nan University and a CCF senior member. His current research areas are parallel processing and DNA computer.

ZHAO Huan was born in 1967. She is a Ph.D. candidate now. Her researches areas are distributed computing and embedded computing.

LI Ren-Fa is concurrently a professor and Ph.D. supervisor at the School of Computer and Communication, Hu'nan University and a CCF senior member. His research areas are network computing and embedded computing.

LI Qing-Hua is concurrently a professor and Ph.D. supervisor at the School of Computer Science and Technology, HUST and a CCF senior member. His research areas are parallel processing, combinatorial optimization and grid computing.