Parallel Computing 43 (2015) 27–42

Contents lists available at ScienceDirect

Parallel Computing

journal homepage: www.elsevier.com/locate/parco

A cost-optimal parallel algorithm for the 0–1 knapsack problem and its performance on multicore CPU and GPU implementations

Kenli Li [a,b], Jing Liu [a,b,*], Lanjun Wan [a,b], Shu Yin [a,b], Keqin Li [a,b,c]

[a] College of Information Science and Engineering, Hunan University, Changsha, Hunan 410082, China
[b] National Supercomputing Center in Changsha, Changsha, Hunan 410082, China
[c] Department of Computer Science, State University of New York, New Paltz, NY 12561, USA

ARTICLE INFO

Article history:
Received 6 December 2012
Received in revised form 4 January 2015
Accepted 23 January 2015
Available online 2 February 2015

Keywords:
Cost-optimal parallel algorithm
EREW PRAM
GPU
Multicore CPU
0–1 knapsack

ABSTRACT

The 0–1 knapsack problem has been extensively studied in the past years due to its immediate applications in industry and financial management, such as cargo loading, stock cutting, and budget control. Many algorithms have been proposed to solve this problem, most of which are heuristic, as the problem is well-known to be NP-hard. Only a few optimal algorithms have been designed to solve this problem but with high time complexity. This paper proposes the *cost-optimal parallel algorithm* (COPA) on an EREW PRAM model with shared memory to solve this problem. COPA is scalable and yields optimal solutions consuming less computational time. Furthermore, this paper implements COPA on two scenarios – multicore CPU based architectures using Open MP and GPU based configurations using CUDA. A series of experiments are conducted to examine the performance of COPA under two different test platforms. The experimental results show that COPA could reduce a significant amount of execution time. Our approach achieves the speedups of up to 10.26 on multicore CPU implementations and 17.53 on GPU implementations when the sequential dynamic programming algorithm for KP01 is considered as a baseline. Importantly, GPU implementations outstand themselves in the experimental results.

1. Introduction

1.1. The problem

Given a set V of n items $v_1, v_2, \ldots, v_n$ to be packed into a knapsack of capacity c, where c is a positive integer, and each item v_i has a weight $w_i \in Z^+$ and a profit $p_i \in Z^+, 1 \leqslant i \leqslant n$, the knapsack problem is to choose a subset of items that the total profit of the chosen items is maximized and the total weight does not exceed the knapsack capacity c. Let $X = (x_1, x_2, \ldots, x_n)$ be an n-tuple of nonnegative integers. Then the above problem can be mathematically formulated as follows:

$$\text{maximize} \sum_{i=1}^{n} p_i x_i, \quad \text{subject to} \sum_{i=1}^{n} w_i x_i \leqslant c. \tag{1}$$

* Corresponding author at: College of Information Science and Engineering, Hunan University, Changsha, Hunan 410082, China.
E-mail addresses: lkl@hnu.edu.cn (K. Li), idealer@126.com (J. Liu), wancanjun2008@163.com (L. Wan), shuyin@hnu.edu.cn (S. Yin), lik@newpaltz.edu (K. Li).

http://dx.doi.org/10.1016/j.parco.2015.01.004
0167-8191/© 2015 Elsevier B.V. All rights reserved.

To avoid any trivial solution, assume that $w_i \leqslant c, \forall i \in \{1, 2, \ldots, n\}$, and $\sum_{i=1}^{n} w_i > c$.

If x_i $(i = 1, 2, \ldots, n)$ is a nonnegative integer, that is, each item has unlimited availability, then Eq. (1) is called the unbounded knapsack problem (KP). If x_i is restricted to be the integer 0 or 1, that is, each item can appear at most once in an optimal solution, then Eq. (1) is called the 0–1 knapsack problem (KP01), which is a special case of KP. If $p_i = w_i$ and x_i is restricted to be the integer 0 or 1, then Eq. (1) is called the subset sum problem (SSP), which is a special case of KP01. For SSP, if it is only required to answer whether there is a solution and the vector X is not required, then the decision version of the subset sum problem (SSPd) is obtained, which is a special case of SSP. KP, KP01, SSP, and SSPd were proven to be NP-hard. In this paper, we mainly focus on KP01.

KP01 has been applied to plentiful applications in theory as well as in practice. For practical applications, KP01 is widely used in various cryptographic schemes [1] and industrial problems. Many industrial problems can be directly modeled as KP01, such as capital budgeting [2], stock cutting [3], cargo loading [4], and batch processor scheduling [5]. Furthermore, KP01 appears as a sub-problem in set-partitioning formulations for multi-dimensional stock cutting [6], crew scheduling, and generalized assignment problems [7].

1.2. Related work

Many algorithms have been proposed to solve KP. Because KP is NP-hard, the majority of these algorithms are heuristics which provide near optimal solutions. Only a few optimal algorithms have been designed to solve KP but with high time complexity. Classic optimal approaches to solving KP are largely based on dynamic programming (DP) [8–15] and the two-list algorithm [16,17], where the DP based algorithms proposed in [9,11,13,15] are designed specifically for KP01. DP based algorithms solve KP in pseudo-polynomial time; when the capacity of the knapsack and the number of items are sufficiently large, two-list based algorithms outperform DP based algorithms. In this paper, we conduct our work based on the two-list algorithm.

By splitting the item set V of n items into two disjoint parts of equal size, Horowitz and Sahni [16] first proposed a new technique which dramatically reduces the time complexity from $O(2^n)$ obtained by brute force to $O(2^{n/2})$ with space $O(2^{n/2})$ for SSP. The new technique is known as the two-list algorithm. Also in [16], Horowitz and Sahni used the dynamic programming method with splitting to solve KP01 in $O(min\{2^{n/2}, nc\})$ time and space. On the basis of the two-list algorithm, Schroeppel and Shamir [17] divided the set V into four parts of the same size, where each part contains $2^{n/4}$ possible combinations of the subset sum, and proposed the two-list four-table algorithm which solves SSP in $O(n2^{n/2})$ time and $O(2^{n/4})$ space. The two-list four-table algorithm takes less space but more time to solve KP01 compared with the two-list algorithm. Although many sequential algorithms have been designed to solve SSP in the past years, Horowitz and Sahni's algorithm still has the least known upper bound on the time complexity for this problem [18].

With the advent of parallelism, many parallel algorithms have been devised for solving problems in various areas to reduce the lengthy computation times required by sequential algorithms. The two-list algorithm is the best theoretical algorithm for solving KP in sequential models and serves as the starting point for introducing efficient parallel techniques for KP. Karnin [19] proposed a parallel algorithm for SSPd which parallelizes the generation routine of the two-list four-table algorithm [17] on a PRAM (*parallel randomly access machine*) model. The algorithm can reduce memory space to $O(2^{n/6})$ by generating four tables dynamically with $O(2^{n/6})$ processors, but it cannot degrade the computation time which is still $O(n2^{n/2})$. Amirazizi and Hellman [20] first demonstrated that parallelism could accelerate to solve larger instances of SSPd. Allowing $O(2^{(1-\alpha)n/2})$ processors to access concurrently a list of the same position in shared memory, they proposed a parallel algorithm to solve SSPd in $O(n2^{\alpha n})(0 \leqslant \alpha \leqslant 1/2)$ time. Enlightened by the parallelization of the two-list algorithm, Ferreira [21] conceived a sequential algorithm referred to as the one-list algorithm for SSPd which is easily parallelizable in contrast with the two-list algorithm and takes less memory space. The parallelized one-list algorithm can solve SSPd in time $O(n(2^{n/2})^{\varepsilon})$ $(0 \leqslant \varepsilon \leq 1)$ with $O((2^{n/2})^{1-\varepsilon})$ processors. Chang et al. [22] claimed a parallelization of the generation stage of the two-list algorithm on a CREW (*concurrent read and exclusive write*) SIMD (*single instruction stream and multiple data streams*) PRAM model with shared memory. They proposed a parallel algorithm to solve SSPd in $O(2^{n/2})$ time with $O(2^{n/4})$ memory and $O(2^{n/8})$ processors.

Furthermore, Lou and Chang [23] proposed a novel search stage for the two-list algorithm which is combined with Chang et al.'s generation stage, giving an optimal parallelization of the two-list algorithm. This parallel algorithm uses $O(2^{n/4})$ memory and $O(2^{n/8})$ processors to solve SSPd in $O(2^{3n/8})$ time.Unfortunately, Sanches et al. [24] proved that the time complexity of the generation stage of Chang et al.'s is wrong, invalidating both Chang et al.'s and Lou and Chang's results. The correct computational time is bounded by $O(n2^{n/2})$ rather than $O(2^{3n/8})$. Inspired by the work of Lou and Chang, Li et al. [25] parallelized the two-list algorithm on an EREW (*exclusive read and exclusive write*) SIMD machine with shared memory. The parallel algorithm applies $O((2^{n/4})^{1-\varepsilon})$ processors and $O(2^{n/2})$ memory space to solve SSP in time $O(2^{n/4}(2^{n/4})^{\varepsilon})), 0 \leqslant \varepsilon \leq 1$, without memory conflicts. Based on a CREW PRAM model with shared memory, Sanches et al. [18] suggested a parallel algorithm to solve SSP in time $O(2^{n/2}/k)$ using $k = 2^q$ processors, $0 \leqslant q \leqslant n/2 - 2\log n$. This parallel algorithm is an optimal and scalable parallelization of the two-list algorithm. Chedid [26,27] presented an optimal parallelization of the two-list algorithm for SSP in time $O(2^{3n/8})$ with $O(2^{n/8})$ processors based on a CREW model. The optimal time-processor cost (i.e. the product of the time complexity and the number of processors) of the proposed algorithm is $O(2^{n/2})$. Shortly, observing that the search stage of the two-list algorithm is closely related to the merge procedure for merging two sorted lists, Chedid [28] presented a second optimal and scalable parallel two-list algorithm on a CREW PRAM model. The algorithm solved SSP in time $O(2^{n/2-\alpha})$ using $O(2^{\alpha})$ processors, for $0 \leqslant \alpha \leqslant n/2 - 2\log n + 2$.

However, since the calculation of the vector X is absent, most of studies above are technically SSPd, which is the decision version of the subset sum problem. Furthermore, the parallel techniques which are designed to solve SSP or SSPd cannot be automatically transplanted to solve KP01. From discussion above, we know that two-list based parallel algorithms can bring good results for SSP. Inspired by these ideas, we design an efficient two-list based parallel algorithm for KP01.

Several programming models were designed to efficiently implement parallel algorithms, such as MPI (*message passing interface*) and OpenMP. MPI is adapted to communication between distributed memory nodes. OpenMP is used for fine-grained parallelism within shared memory nodes. Lin et al. [29] used the MPI + OpenMP hybrid programming model to implement the 0–1 knapsack algorithm by the MH method, named after Merkle and Hellman. The MH method is a public key cryptographic algorithm based on trapdoor information, and is one of several public key cryptographic algorithms which are first designed in the cryptography history.

Recently, GPU (*graphics processing unit*) computing has been recognized as a powerful way to achieve high performance for scientific applications with long execution time due to its highly parallel, multithreaded and manycore unit. Indeed, the peak computational capabilities of modern GPU exceed that of top-of-the-line CPU (*central processing unit*). NVIDIA introduced CUDA (*compute unified device architecture*) which enabled users to solve many computation intensive or complex problems on their GPU cards. CUDA technology is based on the SIMT (*single instruction and multiple threads*) programming model, similar to the SIMD model.

1.3. Our contributions

In this paper, we solve the KP01 problem with the following steps. First, we present a two-list algorithm based sequential method for KP01. Second, combining the sequential algorithm with an optimal parallel merging algorithm, we propose the *cost-optimal parallel algorithm* (COPA) for KP01 on an EREW PRAM model with shared memory. COPA is composed of five stages, i.e., a parallel generation stage, the first parallel saving max-value stage, a parallel pruning stage, the second parallel saving max-value stage, and a parallel search stage. COPA uses $O((2^{n/4})^{1-\varepsilon})$ processors to solve KP01 in $O(2^{n/4}(2^{n/4})^{\varepsilon})$ time and $O(2^{n/2})$ memory space, $0 \leqslant \varepsilon \leq 1$. There is no memory conflict for COPA when all processors access the shared memory simultaneously. Third, COPA is implemented on two scenarios– a multicore CPU architecture and a GPU configuration. The CPU architecture is implemented using the OpenMP programming model while the GPU one is via the CUDA model.

The main contributions of this paper are summarized as follows.

- We propose a sequential algorithm for KP01 which is based on the two-list algorithm. This sequential algorithm serves COPA to be presented later.
- We propose a parallel algorithm COPA for KP01 by parallelizing the sequential algorithm on an EREW PRAM model with shared memory. The time complexity of COPA can be reduced by multiple processors, hence keep the cost in terms of the product of the time complexity and the number of processors invariant.
- COPA avoids memory conflicts when all processors access the shared memory simultaneously.
- COPA is implemented on two popular environments: multicore CPU using OpenMP and GPU with CUDA. The experimental results indicate good scalability and performance of COPA. The experimental results also show that the GPU implementation outperforms the multicore CPU implementation.

The experimental results indicate that the speedup can be up to 4.35 on the multicore CPU implementation and 7.44 on the GPU implementation of COPA compared with the implementation of our two-list based sequential algorithm. However, the experimental results show that the speedup can be as great as 10.26 on multicore CPU implementation and 17.56 on GPU implementation of COPA when the sequential dynamic programming algorithm is regarded as a baseline.

The remainder of this paper is organized as follows. Section 2 presents some backgrounds to be used in this paper. Section 3 proposes a sequential algorithm which is based on the two-list algorithm for KP01. Section 4 proposes the parallel algorithm COPA for KP01. Section 5 provides the methodology of the COPA performance analysis and Section 6 presents the experimental results and evaluations of COPA. Finally, the paper is concluded in Section 7.

2. Background

Before presenting our proposed algorithms, we first introduce two algorithms. One is the optimal parallel merging algorithm on an EREW model proposed by Akl et al. [30], which will be used to merge sorted lists in Section 4. The other is the famous dynamic programming (DP) algorithm [31] for solving KP01, as one of two baselines of COPA in Section 6.

2.1. The optimal parallel merging algorithm on an EREW model

Assume that there are two sorted vectors $U = (u_1, u_2, \ldots, u_m)$ and $V = (v_1, v_2, \ldots, v_m)$, and k processors $P_1, P_2, \ldots, P_k$, where $1 \leqslant k \leqslant 2m$ is a power of 2. According to the merging algorithm, vectors U and V can be merged without memory conflicts into a new vector of length $2m$.

The merging algorithm is described as follows with a slight modification.

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

1. $|U_i| + |V_i| = 2m/k$, for $1 \leqslant i \leqslant k$;
2. the weights of all elements in U_i and V_i are smaller than those of all elements in U_{i+1} and V_{i+1}, for $1 \leqslant i \leqslant k-1$.

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

Here Step 1 can be efficiently implemented using the selection algorithm presented by Akl et al. [30]. It has been proven that this parallel merging algorithm can be executed in the EREW PRAM model of parallel computation in time $O(2m/k + \log k \times \log 2m)$. The merging algorithm is therefore optimal for $k \leqslant m/\log^2 m$, in view of the trivial $\Omega(m)$ lower bound in merging two vectors of total length $2m$.

2.2. The dynamic programming (DP) algorithm for KP01

The DP algorithm [31] for solving KP01 is described as follows. Given a knapsack of capacity c and n items $v_1, v_2, \ldots, v_n$, where each item v_i has a weight w_i and a profit p_i, the DP algorithm can generate an optimal solution by the following three steps:

1. Let $F[i, x]$ denote the maximum profit value that can be attained when the total weight is less than or equal to x using items up to v_i. For $0 \leqslant x \leqslant c, F[0, x] = 0$.
2. For $i = 1$ to n, compute $F[i, x]$ $(x = 0, 1, 2, \ldots, c)$ by:

$$F[i,x] = \begin{cases} max\{F[i-1,x], F[i-1,x-w_i] + p_i\}, & \text{if} \quad x \geqslant w_i; \\ F[i-1,x], & \text{if} \quad x < w_i. \end{cases} \tag{2}$$

3. $F[n, c]$ is the maximum profit satisfying the knapsack capacity c.

The time complexity of the DP algorithm is $O(nc)$.

Before presenting Section 3, some notations used in this paper are listed in Table 1.

3. A sequential algorithm for KP01

In this section, inspired by Horowitz and Sahni's work, a sequential algorithm based on the two-list algorithm [16] is proposed for KP01. The algorithm can be divided into two stages which depend on the activities performed, i.e., a generation

Table 1
Notations used in this paper.

Notation	Definition
V	the set of all the n items
V_1	a subset of V
V_2	a subset of V where V_1 and V_2 are disjoint with equal size and generate V
$\|V_i\|$	the number of items in set $V_i (i = 1, 2)$
n	the number of items in set V
N	the number of subsets in set $V_i (i = 1, 2)$, $N = 2^{n/2}$
A	the sorted list in nondecreasing order for V_1
B	the sorted list in nonincreasing order for V_2
a_i	a subset of V_1
b_i	a subset of V_2
c	the knapsack capacity
p	represents the profit dimension
w	represents the weight dimension
k	the number of available processors
w_{min}	represents the smallest weight among all n items
e	the number of elements in each block, equal to N/k

stage and a search stage. The generation stage is designed to generate two sorted lists. The search stage is designed to find a solution.

3.1. The generation stage

The generation stage is composed of three steps.

1. Divide V into two disjoint parts with the same size: $V_1 = (v_1, v_2, \cdots, v_{n/2})$, and $V_2 = (v_{n/2+1}, v_{n/2+2}, \cdots, v_n)$. That is, $|V_1| = |V_2|, V_1 \cap V_2 = \emptyset$, and $V_1 \cup V_2 = V$, where $|V_1|$ and $|V_2|$ represent the number of elements in sets V_1 and V_2, respectively.
2. For any subset a_i of all $N = 2^{n/2}$ subsets of V_1, calculate its weight $a_i.w$ which is the weight sum of all items in a_i and its profit $a_i.p$ which is the profit sum of all items in a_i, sort these N possible triplets $(a_i, a_i.w, a_i.p)$ in nondecreasing order of weight, and then store them as a list $A = [(a_1, a_1.w, a_1.p), (a_2, a_2.w, a_2.p), \ldots, (a_N, a_N.w, a_N.p)]$, where w represents the weight dimension and p represents the profit dimension.
3. For any subset b_i of all $N = 2^{n/2}$ subsets of V_2, calculate its weight $b_i.w$ which is the weight sum of all items in b_i and its profit $b_i.p$ which is the profit sum of all items in b_i, sort these N possible triplets $(b_i, b_i.w, b_i.p)$ in nonincreasing order of weight, and then store them as a list $B = [(b_1, b_1.w, b_1.p), (b_2, b_2.w, b_2.p), \ldots, (b_N, b_N.w, b_N.p)]$.

3.2. The search stage

The search stage is devised to find a solution for KP01, described as Algorithm 1. The basic idea of Algorithm 1 is first to compute the current maximum profit value $MaxB_i$ of all subsets from b_i to b_N, and use notation L_i to record the index of the subset from which $MaxB_i$ is obtained, $1 \leqslant i \leqslant N$. Second, the solution for KP01 is searched from all the combinations of elements from the two sorted lists A and B (see Fig. 1). The final *Bestvalue* and X show the solution, where *Bestvalue* represents the maximum profit and X is a string consisting of n 0's and 1's, revealing which items are selected when the maximum profit *Bestvalue* is obtained. If the ith position of X is 1, then the item v_i is selected; otherwise, v_i is not selected.

Algorithm 1. The search stage.

```
Input: two sorted list $A$ and $B$
Output: the final solution $Bestvalue$ and $X$
1: initialize $MaxB_N = b_N.p$, $L_N = N$, $Bestvalue = 0$, and $X1 = (0, 0)$
2: for $i = N - 1$ to 1 do
3:    if $b_i.p > MaxB_{i+1}$ then
4:       $MaxB_i = b_i.p$ and $L_i = i$
5:    else
6:       $MaxB_i = MaxB_{i+1}$ and $L_i = L_{i+1}$
7:    end if
8: end for
9: $i = 1$ and $j = 1$
10: while $i \le N$ and $j \le N$ do
11:    if $a_i.w + b_j.w > c$ then
12:       $j{+}{+}$ and continue
13:    end if
14:    if $a_i.p + MaxB_j > Bestvalue$ then
15:       $Bestvalue = a_i.p + MaxB_j$ and $X1 = (a_i, b_{L_j})$
16:    end if
17:    $i{+}{+}$
18: end while
19: convert two decimal components of $X1$ to two binary numbers, $X1 = (binary1, binary2)$
20: $X$ = strcat($binary1, binary2$) /*concatenate two binaries by strcat()*/
```

It is clear that the time complexity of Algorithm 1 is $O(N)$.

3.3. The complete sequential algorithm

In summary, the sequential algorithm mainly consists of the following three steps.

- Step 1: divide V into two disjoint parts of the same size: V_1 and V_2, and then calculate the weight and profit of all subsets of V_1 and V_2.

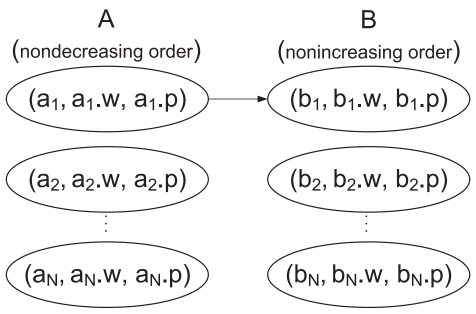

Fig. 1. The search stage.

- Step 2: store all the subsets of V_1 (V_2, respectively) together with their weights and profits in nondecreasing (nonincreasing, respectively) order of weight as a list A (B, respectively).
- Step 3: serially search the optimal solution from all the combinations of elements from the two sorted lists A and B.

Steps 1 and 2 are performed through merging, and the time complexity is $O(N)$. Let us explain how to generate the sorted list A. First, the sorted list A and a variable IC are initialized to $A_0 = [(0,0,0)]$ and $IC = 1$, respectively. Second, let $v_1 = IC$ and the triple $(v_1, v_1.w, v_1.p)$ is added to each element of list A_0, constructing a new list $A_0^1 = [(0 + v_1, 0 + v_1.w, 0 + v_1.p)] = [(IC, v_1.w, v_1.p)] = [(1, v_1.w, v_1.p)]$. Third, lists A_0 and A_0^1 are merged to generate a sorted list A_1 which has 2 elements in nondecreasing order of weight in linear time, and IC is increased by 1. Forth, let $v_2 = IC$ and the triple $(v_2, v_2.w, v_2.p)$ is added to the list A_1, resulting in a new sorted list A_1^1. Again, the merging of lists A_1 and A_1^1 is performed to produce a new sorted list A_2 with 4 elements, and IC is increased by 1. Repeat the process above until all the triples corresponding to all the items in V_1 are performed. After processing the last triple $(v_{n/2}, v_{n/2}.w, v_{n/2}.p)$, we obtain the final sorted list $A_{n/2}$, that is, the sorted list A. The generation of list B is the same as that of list A except that all the elements in B are sorted in nonincreasing order of weight. Hence, the total running time of Steps 1 and 2 is $\sum_{1\leqslant i\leqslant n/2} O(2^i) = O(N)$.

Step 3 is described as Algorithm 1 and its running time is $O(N)$. Summarizing all the times spent by these three steps, the time complexity of the sequential algorithm is $O(N)$, that is, $O(2^{n/2})$.

4. The cost-optimal parallel algorithm (COPA)

In this section, we present a novel parallel algorithm COPA for solving KP01 on an EREW PRAM model with shared memory. COPA aims at parallelizing the sequential algorithm described in Section 3 with low time complexity and high performance. Assume that k processors are available. We first discuss COPA for $k = O(N^{1/2})$. In this case, COPA contains five stages: a parallel generation stage, the first parallel saving max-value stage, a parallel pruning stage, the second parallel saving max-value stage, and a parallel search stage. These five stages are depicted in detail in the following.

4.1. The parallel generation stage

In this section, we describe the generation of two sorted lists A and B. A is sorted in nondecreasing order of weight while B is sorted in nonincreasing order of weight. Since A and B are sorted in reverse directions, their generation procedures are almost the same, and we only introduce the procedure of generating the nondecreasing list A. In the generation procedure, the optimal parallel merging algorithm presented in Section 2 is applied to merge two lists.

Let us trace the last step of generating list A (i.e., $A_{n/2}$). Suppose that the sorted list $A_{n/2-1}$ has been generated, written as $A_{n/2-1} = [(a_{n/2-1.1}, a_{n/2-1.1}.w, a_{n/2-1.1}.p), (a_{n/2-1.2}, a_{n/2-1.2}.w, a_{n/2-1.2}.p), \ldots, (a_{n/2-1.N/2}, a_{n/2-1.N/2}.w, a_{n/2-1.N/2}.p)]$. List $A_{n/2-1}$ has $N/2$ elements, each of which is a triple and stored in the shared memory. We use k processors in parallel to add the triple $(v_{n/2}, v_{n/2}.w, v_{n/2}.p)$ into each element of $A_{n/2-1}$, attaining a new sorted list $A_{n/2-1}^1 = [(a_{n/2-1.1} + v_{n/2}, a_{n/2-1.1}.w + v_{n/2}.w, a_{n/2-1.1}.p + v_{n/2}.p), (a_{n/2-1.2} + v_{n/2}, a_{n/2-1.2}.w + v_{n/2}.w, a_{n/2-1.2}.p + v_{n/2}.p), \ldots, (a_{n/2-1.N/2} + v_{n/2}, a_{n/2-1.N/2}.w + v_{n/2}.w, a_{n/2-1.N/2}.p + v_{n/2}.p)]$. The concurrent read is not permitted, so it is necessary to create a copy of $(v_{n/2}, v_{n/2}.w, v_{n/2}.p)$ for each processor before merging, which incurs $O(nN^{1/2})$ assistant memory cells in the shared memory. Then the optimal parallel merging algorithm is applied to merge lists $A_{n/2-1}$ and $A_{n/2-1}^1$, which produces the sorted list $A_{n/2}$ (i.e., A).

No memory conflicts occur during the entire process of generating A by applying the optimal parallel merging algorithm. Considering that the time for merging is $O(N/k + \log k \times \log N)$ [30], the time for generating list $A_{n/2}$ from list $A_{n/2-1}$ is $O(N/2k + N/k + \log k \times \log N) = O(3N/2k + n^2/8)$.

As shown in Section 3.2,, to generate list $A_{n/2}$, we initialize $A_0 = [(0,0,0)]$ and $IC = 1$. Next, we add the triple $(v_{i+1}, v_{i+1}.w, v_{i+1}.p)$ to all the elements in the sorted list A_i sequentially, generating a new sorted list A_i^1, where

$v_{i+1} = IC = i + 1$ and $0 \leqslant i \leqslant n/2 - 1$. Then, we merge lists A_i and A_i^1 to produce the sorted list A_{i+1} which includes 2^{i+1} elements. When generating list A_{i+1}, at most 2^i processors are required. Since there are $O(N^{1/2})$ (i.e. $O(2^{n/4})$) processors, only a fraction of them are utilized when generating lists $A_1, A_2, \ldots, A_{n/4-1}$ and $A_{n/4}$. Whereas, all the processors must be used when generating lists $A_{n/4+1}, A_{n/4+2}, \ldots, A_{n/2-1}$ and $A_{n/2}$.

The parallel generation algorithm for generating the nondecreasing sorted list A is shown as Algorithm 2. Its basic idea is analogous to that of the generation phase of [18], but with a slight modification.

Algorithm 2. The parallel generation algorithm.

```
Input: $(v_i, v_i.w, v_i.p)$, $i = 1, 2, ..., n/2$
Output: the sorted list $A_{n/2}$ (i.e. $A$)
1:     initialize $A_0 = [(0, 0, 0)]$ and $IC = 1$
2: for $i = 0$ to $n/2 - 1$ do
3:     $v_{i+1} = IC$
4: for all  $P_j$, $1 \leq j \leq k$,  in parallel do
5:     produce new list $A_i^1$ by adding the triple $(v_{i+1}, v_{i+1}.w, v_{i+1}.p)$ to all the elements in the sorted list $A_i$
6:     call the optimal parallel merging algorithm introduced in Section 2 to merge lists $A_i$ and $A_i^1$ in nondecreasing
   order of weight, producing the sorted list $A_{i+1}$
7: end for
8:     $IC++$
9: end for
```

4.2. The first parallel saving max-value stage

In this stage, both lists A and B are evenly partitioned into k blocks. For each block, the maximum profit among all elements in the block is identified.

Denote the k blocks of list A to be $\overline{A_1}, \overline{A_2}, \ldots \overline{A_k}$. Because list A contains N elements, each block contains $e = N/k$ elements. Let $\overline{A_i} = (\overline{a_{i.1}}, \overline{a_{i.2}}, \ldots, \overline{a_{i.e}}), 1 \leqslant i \leqslant k$. Each element $\overline{a_{i.r}}$ in $\overline{A_i}$ represents a triple $(\overline{a_{i.r}}, \overline{a_{i.r}}.w, \overline{a_{i.r}}.p)$, where $\overline{a_{i.r}}.w$ and $\overline{a_{i.r}}.p$ represent the weight and profit of the set $\overline{a_{i.r}}$, respectively, $1 \leqslant i \leqslant k, 1 \leqslant r \leqslant e$. All the elements in $\overline{A_i}$ are sorted in nondecreasing order of weight. The weight of any element in $\overline{A_i}$ is not larger than that of any element in $\overline{A_{i+1}}, 1 \leqslant i \leqslant n/2 - 1$. Then, list A can be represented as $A = (\overline{A_1}, \overline{A_2}, \cdots, \overline{A_i}, \ldots, \overline{A_k})$. Likewise, list B can be represented as $B = (\overline{B_1}, \overline{B_2}, \ldots, \overline{B_j}, \cdots, \overline{B_k})$, where $\overline{B_j} = (\overline{b_{j.1}}, \overline{b_{j.2}}, \ldots, \overline{b_{j.e}}), 1 \leqslant j \leqslant k$. The blocks of B and all the elements in each block are both sorted in nonincreasing order of weight. After partitioning, blocks $\overline{A_i}$ and $\overline{B_i}$ are assigned to processor $P_i, 1 \leqslant i \leqslant k$. Then, the maximum profits of blocks $\overline{A_i}$ and $\overline{B_i}$ are found, and denoted by $MaxA_i$ and $MaxB_i$, respectively. $MaxA_i$ are stored in an array $MaxA$, and $MaxB_i$ are stored in an array $MaxB$. The first parallel saving max-value algorithm shown as Algorithm 3 addresses how to obtain the maximum profit of each block.

Algorithm 3. The first parallel saving max-value algorithm.

```
Input: two sorted lists $A$ and $B$
Output: $MaxA$ and $MaxB$
1:     divide both $A$ and $B$ evenly into $k$ blocks
2: for all  $P_i$, $1 \leq i \leq k$,  in parallel do
3:     $MaxA_i = \overline{a_{i.1}}.p$, $MaxB_i = \overline{b_{i.1}}.p$
4: for  $j = 2$ to $e$  do
5: if  $\overline{a_{i.j}}.p > MaxA_i$  then
6:     $MaxA_i = \overline{a_{i.j}}.p$
7: end if
8: if  $\overline{b_{i.j}}.p > MaxB_i$  then
9:     $MaxB_i = \overline{b_{i.j}}.p$
10: end if
11: end for
12: end for
```

It is observed that the time complexity of Algorithm 3 is $O(N^{1/2})$.

4.3. The parallel pruning stage

This stage attempts to shrink the search space of KP01 in order to reduce the search time of each processor.

Before presenting the parallel pruning stage, we introduce two lemmas, which are based on the work of Lou and Chang [23]. Since the proofs of our lemmas are similar to those of Lemmas 1 and 2 in [23], we do not repeat the proof procedures in this paper. Each element of any block is simply a weight in [23], whereas each element of any block is a triple (the set, the weight, the profit) in this paper, $1 \leqslant i,j \leqslant k$.

Lemma 1. *For any block pair* $(\overline{A_i}, \overline{B_j}), 1 \leqslant i,j \leqslant k, \overline{A_i} = (\overline{a_{i.1}}, \overline{a_{i.2}}, \ldots, \overline{a_{i.e}})$ *and* $\overline{B_j} = (\overline{b_{j.1}}, \overline{b_{j.2}}, \ldots, \overline{b_{j.e}})$, *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* $(\overline{A_i}, \overline{B_j}), 1 \leqslant i,j \leqslant k, \overline{A_i} = (\overline{a_{i.1}}, \overline{a_{i.2}}, \ldots, \overline{a_{i.e}})$ *and* $\overline{B_j} = (\overline{b_{j.1}}, \overline{b_{j.2}}, \ldots, \overline{b_{j.e}})$, *if* $\overline{a_{i.e}}.w + \overline{b_{j.1}}.w \leqslant c$, *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 candidate solution for KP01.*

Based on Lemmas 1 and 2, we design the parallel pruning algorithm, which can greatly shrink the search space of each processor. For the case of Lemma 1, because any element pair $(\overline{a_{i.r}}, \overline{b_{j.s}})$ of block pair $(\overline{A_i}, \overline{B_j})$ is not a solution of KP01, we prune $(\overline{A_i}, \overline{B_j})$. For the case of Lemma 2, because any element pair $(\overline{a_{i.r}}, \overline{b_{j.s}})$ of block pair $(\overline{A_i}, \overline{B_j})$ is a candidate solution for KP01, we save the maximum profit of $(\overline{A_i}, \overline{B_j})$, namely, the sum of $MaxA_i$ and $MaxB_j$ and then prune $(\overline{A_i}, \overline{B_j})$. After pruning the blocks in these two cases, the remaining blocks are written into different cells in the shared memory. Denote $Maxvalue_i$ as the maximum profit value of block pair $(\overline{A_i}, \overline{B_{j\bmod k}})$. The parallel pruning algorithm can be described as Algorithm 4.

Algorithm 4. The parallel pruning algorithm.

```
Input: blocks $\overline{A_1}$, $\overline{A_2}$, ... $\overline{A_k}$ and $\overline{B_1}$, $\overline{B_2}$, ... $\overline{B_k}$
Output: the remaining block pairs after pruning
1: for all  $P_i$, $1 \le i \le k$,  in parallel
2:      $Maxvalue_i = 0$  do
3: for $j = i$ to $k + i - 1$
4:      $Z = \overline{a_{i.1}}.w + \overline{b_{j\bmod k.e}}.w$
5:      $Y = \overline{a_{i.e}}.w + \overline{b_{j\bmod k.1}}.w$
6: if $Y \le c$ then
7: if $MaxA_i + MaxB_{j\bmod k} > Maxvalue_i$ then
8:      $Maxvalue_i = MaxA_i + MaxB_{j\bmod k}$
9: end if
10:      prune block pair ($\overline{A_i}$, $\overline{B_{j\bmod k}}$)
11: els if $Z \le c$ and $Y > c$ then
12:      write block pair ($\overline{A_i}$, $\overline{B_{j\bmod k}}$) into different cell in the shared memory
13: els if $Z > c$ then
14:      prune block pair ($\overline{A_i}$, $\overline{B_{j\bmod k}}$)
15: end if
16: end for
17:end for
```

In Algorithm 4, each memory cell of list B is read or written by at most one processor at any time. Hence, there are no memory conflicts in the shared memory. Thus, Algorithm 4 can be executed on the EREW PRAM model. Because each processor prunes its individual search space, it is obvious that the time complexity of Algorithm 4 is $O(k)$, i.e., $O(2^{n/4})$. Before pruning, the number of block pairs is k^2. After pruning, the number of block pairs is at most $2k - 1$. Based on Theorem 2 in the work of Lou and Chang [23], the following theorem is to show that the number of the residual block pairs is at most $2k$-1.

Theorem 1. *The parallel pruning algorithm picks at most* $2k - 1$ *block pairs.*

Proof. After the parallel pruning algorithm is performed, the blocks that processor P_i picks from list B are adjacent one by one according to [23]. Assume that processor P_i picks block pairs $(\overline{A_i}, \overline{B_j}), (\overline{A_i}, \overline{B_{j+1}}), \ldots, (\overline{A_i}, \overline{B_{l-1}})$, and $(\overline{A_i}, \overline{B_l})$ as shown in Fig. 2. That is, all of the combinations of $\overline{A_i}$ and $\{\overline{B_j}, \overline{B_{j+1}}, \ldots, \overline{B_{l-1}}, \overline{B_l}\}$ must be checked later to find a solution.

Denote $BS_i = \{\overline{B_j}, \overline{B_{j+1}}, \ldots, \overline{B_{l-1}}, \overline{B_l}\}$. From Algorithm 4 and Fig. 2, we know that $\overline{a_{i.e}}.w + \overline{b_{t.1}}.w > c$, where $j \leqslant t \leqslant l$. 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, processor p_{i+1} prunes $\overline{B_{t-1}}$, and $BS_i \cap BS_{i+1} = \Phi$ or $\{\overline{B_l}\}$, $1 \leqslant i < k-1$. Thus, sets BS_i and BS_{i+1} have at most one common block.

Let $|BS_i| = \alpha_i$, where α_i $(1 \leqslant i \leqslant k)$ is a nonnegative integer and represents the number of blocks in BS_i. The following inequation is obtained:

$$\alpha_1 + (\alpha_2 - 1) + (\alpha_3 - 1) + \cdots + (\alpha_k - 1) \leqslant k. \tag{3}$$

It follows that

$$\alpha_1 + \alpha_2 + \alpha_3 + \cdots + \alpha_k \leqslant k + (k-1) = 2k - 1. \tag{4}$$

Therefore, Theorem 1 is true. □

4.4. The second parallel saving max-value stage

This section presents how to find the maximum profit value of each block on processors after pruning. After pruning, at most $2k-1$ block pairs are equally assigned to k processors. Each processor then has at most two block pairs to execute. Assume that there are s $(0 \leqslant s \leqslant 2)$ block pairs assigned to processor P_i. Denote these blocks as $(\overline{A_i}, \overline{B_{i0}}), (\overline{A_i}, \overline{B_{i1}}), \ldots, (\overline{A_i}, \overline{B_{i(s-1)}})$, where $\overline{A_i} = (\overline{a_{i.1}}, \overline{a_{i.2}}, \ldots, \overline{a_{i.e}}), \overline{B_{it}} = (\overline{b_{it.1}}, \overline{b_{it.2}}, \cdots, \overline{b_{it.e}}), 1 \leqslant it \leqslant k, 0 \leqslant t \leqslant s-1 \leqslant 1$. Let $Max_{i.j}[t]$ be the current maximum profit from $\overline{b_{it.j}}$ to $\overline{b_{it.e}}$ of block $\overline{B_{it}}$ in block pair $(\overline{A_i}, \overline{B_{it}})$. We use notations L_j^t to record the selected subset from list B when $Max_{i.j}[t]$ is obtained. The details of finding the maximum profit of each block on processors are described as Algorithm 5.

Algorithm 5. The second parallel saving max-value algorithm.

```
Input: the remaining block pairs after pruning
Output: $Max_{i.j}[t]$ and $L_j^t$
1: for all $P_i$, $1 \le i \le k$, in parallel do
2:   if $1 \le s \le 2$ then
3:     for $t = 0$ to $s - 1$ do
4:       ~~for block $\overline{B_{it}}$ in block pair $(\overline{A_i}, \overline{B_{it}})$ do~~
~~5:~~         ~~$Max_{t.e}[t] = \overline{b_{it.e}}.p$ and $L_e^t = e$~~
~~6:~~         ~~for $j = e - 1$ to 1 do~~
~~7:~~           ~~if $\overline{b_{it.j}}.p > Max_{t.(j+1)}[t]$ then~~
~~8:~~             ~~$Max_{t.j}[t] = \overline{b_{it.j}}.p$ and $L_j^t = j$~~
~~9:~~           ~~else~~
~~10:~~             ~~$Max_{t.j}[t] = Max_{t.(j+1)}[t]$ and $L_j^t = L_{j+1}^t$~~
~~11:~~           ~~end if~~
~~12:~~         ~~end for~~
~~13:~~       ~~end for~~
14:     end for
15:   end if
16: end for
```

It is clear that the time complexity of Algorithm 5 is $O(N^{1/2})$.

4.5. The parallel search stage

This stage designs the parallel search algorithm to search a solution for KP01. Because memory cells occupied by each block pair are completely different from each other, no memory conflicts happen between different processors when searching a solution. Therefore, the parallel search stage can be performed on the EREW model. Let $Maxvalue_i$ represent the maximum profit obtained from processor P_i, X_i record the solution when $Maxvalue_i$ is obtained, *Bestvalue* and X be the final solution for KP01, $(1 \leqslant i \leqslant k)$. The parallel search algorithm can be described as Algorithm 6.

Algorithm 6. The parallel search algorithm.

```
Input: the remaining block pairs after pruning
Output: the final solution $Best value$ and $X$
1: for all $P_i$, $1 \le i \le k$, in parallel do
2:     initialize $t = 0, Max value_i = 0$, and $X_i = (0,0)$
3:     while $0 \le t \le s-1$ do
4:       pick out block pair $(\overline{A_i}, \overline{B_{it}})$
5:       $x = 1$ and $y = 1$
6:       while $x \le e$ and $y \le e$ do
7:         if $\overline{a_{i.x}}.w + \overline{b_{it.y}}.w > c$ then
8:           $y++$ and continue
9:         end if
10:        if $\overline{a_{i.x}}.p + Max_{i.y}[t] > Max value_i$ then
11:          $Max value_i = \overline{a_{i.x}}.p + Max_{i.y}[t]$ and $X_i = (\overline{a_{i.x}}, \overline{b_{it.L_y^t}})$
12:        end if
13:        $x++$
14: end while
15:      $t++$
16:    end while
17: end for
18: $Best value = Max value_1$ and $X1 = X_1$
19: for $i = 2$ to $k$ do
20:    if $Best value < Max value_i$ then
21:      $Best value = Max value_i$ and $X1 = X_i$
22:    end if
23: end for
24: convert two decimal components of $X1$ to two binary numbers, $X1 = (binary1, binary2)$
25: $X$ = strcat($binary1, binary2$) /*concatenate two binaries by strcat()*/
```

It is evident that the parallel search algorithm can be performed on the EREW in time $O(N^{1/2})$.

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

In this section, we extend the results stated for the case $k = O(N^{1/2})$ to the case $k < O(N^{1/2})$. The processes of finding solutions for KP01 in these two cases are almost the same. The process of finding solutions for $k < O(N^{1/2})$ can also be divided

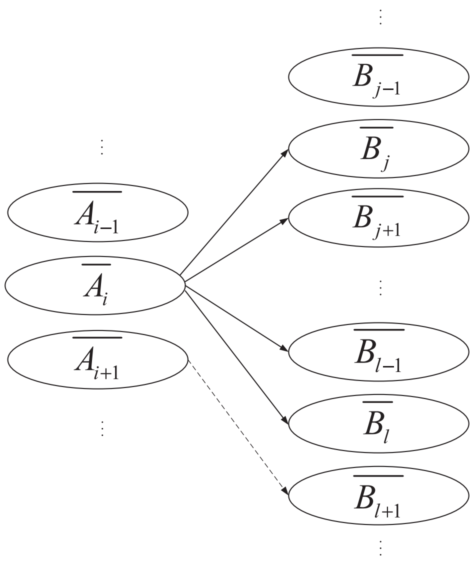

Fig. 2. The search space for processor P_i.

into five stages: the parallel generation stage, the first parallel saving max-value stage, the parallel pruning stage, the second parallel saving max-value stage, and the parallel search stage. The only difference is that the time complexities of these five stages are changed, due to reduced available processors. Therefore, COPA can solve KP01 for both cases of k with different time complexities. The time complexity of COPA under these two cases is presented in detail in the following section.

5. Theoretical performance evaluation of COPA

This section presents the theoretical performance analysis of COPA and compares it with other widely discussed parallel algorithms for solving KP01. We assume that there are k available processors and each instance of KP01 contains n items.

5.1. Performance analysis

We first consider the case of $k = O(N^{1/2})$. The running time of the parallel generation algorithm is computed as follows:

$$\sum\nolimits_{i=1}^{n/4-1}(2+(i+1)i)+\sum_{i=n/4}^{n/2-1}(2^{i+1}/(2k)+2^{i+1}/k+\log(2^{n/4})\log(2^{i+1}))$$
$$=\sum\nolimits_{i=1}^{n/4-1}(2+(i+1)i)+\sum_{i=n/4}^{n/2-1}\left(\frac{3\times 2^{i+1}}{2k}+\frac{n(i+1)}{4}\right)=3N/k. \quad (5)$$

Hence the parallel generation stage of COPA requires $O(3N/k)$ time. For the rest four stages of COPA, the first parallel saving max-value algorithm requires $O(N/k)$ time; the parallel pruning algorithm requires $O(k)$ time; the second parallel saving max-value algorithm requires $O(2N/k)$ time; the parallel search algorithm takes $O(4N/k+k)$ time to find a solution in the worst case. Due to $k = O(N^{1/2})$, the running times of these five stages are all $O(2^{n/4})$. The running time of COPA is the total time consumed by the five stages, which is $T = O(2^{n/4})$. The cost of COPA is $C = T \times k = O(2^{n/4}) \times O(2^{n/4}) = O(2^{n/2})$.

Then, let us discuss the case of $k < O(N^{1/2})$. Assuming that $k = O((N^{1/2})^{1-\varepsilon})(0 < \varepsilon \leqslant 1)$, we have $1 \leqslant k < N^{1/2}$ and $N/k > k$. Since $O(4N/k+k)$ is dominated by $O(N/k)$, each stage consumes $O(N/k) = O(2^{n/4}(2^{n/4})^{\varepsilon})$ time. The required time by COPA is then calculated as $5 \times O(N/k) = O(N/k) = O(2^{n/4}(2^{n/4})^{\varepsilon})$. In addition, the cost of COPA denoted as C, can be presented as $C = O(N/k) \times O(k) = O(2^{n/2})$. If there is a single processor, COPA leads to the sequential algorithm in Section 3 that solves KP01 in $O(2^{n/2})$ time and space. This sequential algorithm is as fast as Horowitz and Sahni's dynamic programming based algorithm [16] when $nc \geqslant 2^{n/2}$.

Briefly, the time complexity of COPA is $O(2^{n/4}(2^{n/4})^{\varepsilon})$ $(0 \leqslant \varepsilon \leqslant 1)$ when $k = O((N^{1/2})^{1-\varepsilon})$ processors are available. The cost of COPA is always $O(2^{n/2})$, independent of the knapsack capacity c. For clarity, the running times of each stage of COPA are listed in Table 2.

5.2. Performance comparison

In this section, we discuss the methodology of comparing COPA with some popular parallel algorithms for solving KP01. Six major features are considered in the comparison such as the topology of the parallel computer, the model for parallel computation, the methodology, the number of processors, the time complexity and the memory space.

Goldman and Trystram [8] presented an optimal parallel dynamic programming (DP) algorithm for solving KP01 on hypercubes. The algorithm is bounded in time by $\Theta\left(\frac{nc}{k}+\frac{c}{w_{\min}}\right)$ with $k \leqslant \frac{c}{\log w_{\min}}$, where w_{min} is the smallest weight among the weights of all items. Lee et al. [11] applied the DP method for KP01 on a hypercube topology, and proposed a parallel exact algorithm to solve KP01 in time $O(\frac{nc}{k}+c^2)$ and space $O(nc)$. They required that the knapsack capacity c be relatively small compared with the number n of items to be practical. Inspired by Lee et al. [11], Lin and Storer[12] proposed two parallel algorithms based on DP. One uses a hypercube topology and solves KP01 in $O(\frac{nc}{k}\log k)$ time and $O(nc)$ space, requiring $c = \Omega(k\log k)$. The other is based on an EREW model and solves KP01 in $O(\frac{nc}{k}\log k)$ $(k < c)$ time and $O(nc)$ space. All of these

Table 2
The running times required by each stage of COPA with $k = O((N^{1/2})^{1-\varepsilon})$ available processors, $0 \leqslant \varepsilon \leqslant 1$.

Stages	Time
The parallel generation stage	$O(3N/k)$
The first parallel saving max-value stage	$O(N/k)$
The parallel pruning stage	$O(k)$
The second parallel saving max-value stage	$O(2N/k)$
The parallel search stage	$O(4N/k+k)$

parallel algorithms are based on the DP approach, whereas our proposed parallel algorithm is based on the two-list algorithm. For clarity, comparisons of these parallel algorithms for KP01 are summarized in Table 3.

For these DP-based parallel algorithms, both their time complexities and memory requirements are related to the knapsack capacity c, and increase when c increases. $nc > 2^{n/2}$ incurs a larger cost than $O(2^{n/2})$, where the cost of an algorithm is the product of the time complexity and the number of processors for the algorithm. However, the cost of COPA can keep $O(2^{n/2})$ invariant. From discussions in [32], the cost of COPA is optimal. Hence, COPA undoubtedly outperforms these parallel algorithms, especially when $nc > 2^{n/2}$.

6. Experimental results and evaluation

In this section, we first present the experimental setting and the environment configuration. The experimental results and evaluation are then presented.

6.1. Experimental setting and environment

The COPA is implemented on two scenarios: multicore CPU and GPU. The CPU implementation employs the OpenMP programming model, and limits the number of concurrent threads to the number of available cores on the target machine. The GPU implementation adopts the CUDA programming model.

A series of experiments are conducted to test the performance of COPA. For general purposes, two different platforms are used in the experiments. One is based on an Intel platform while the other is based on an AMD platform. The details of the configurations can be presented in Table 4.

We have carried out computational tests on randomly generated instances of KP01. These instances are available in [31] with the following features:

- w_i and p_i are randomly drawn in [1, 10000000].
- $c = \frac{1}{2}\sum_{i=1}^{n} w_i$.

Six instances of KP01 with above features are generated, of sizes (i.e., the numbers of items included in these instances) 40, 42, 44, 46, 48, and 50. Due to the memory-intense nature of applications, the maximum size of the instances is constrained by the computing power. For the first three instances, the value of c is close to $2^{n/2}/8$, where n is the size of an instance. For the rest three instances, the value of c is close to $2^{n/2}/4$.

To illustrate the performance of COPA intuitively, we use two sequential algorithms for KP01 as baselines. The first is the famous dynamic programming (DP) algorithm [31] presented in Section 2, which suits the case that c is relatively small. The

Table 3
Comparisons of parallel algorithms for solving KP01

Algorithm	Topology	Model	Methodology	Num. of Processors	Time complexity	Memory space
Goldman et al. [8]	HC	–	DP	$k \leqslant \frac{c}{\log w_{\min}}$	$\Theta(\frac{nc}{k} + \frac{c}{w_{\min}})$	no mention
Lee et al. [11]	HC	–	DP	$k \leqslant n$	$O(\frac{nc}{k} + c^2)$	$O(nc)$
Lin et al. [12]	HC	–	DP	$c = \Omega(k \log k)$	$O(\frac{nc}{k} \log k)$	$O(nc)$
	–	EREW-PRAM	DP	$k \leqslant c$	$O(\frac{nc}{k})$	$O(nc)$
Ours	–	EREW-PRAM	TL	$k = O((2^{n/4})^{1-\varepsilon})$	$O(2^{n/4}(2^{n/4})^{\varepsilon})$	$O(2^{n/2})$

k: Num. of the processors. HC: Hypercube; DP: dynamic programming; TL: the two-list algorithm.

Table 4
The detailed configurations of the two test platforms.

		Test platform 1	Test platform 2
Hardware	CPU	HP single CPU four-core computer, Intel(R) Xeon(R) Processor E5504, 2.00 GHz	Dawning dual CPU sixteen-core high-performance computer, AMD Opteron(TM) Processor 6134, 2.30 GHz
	GPU	GeForce GTX 470, 448 CUDA Cores. A multiprocessor can have 1024 threads simultaneously active, or 32 warps	Tesla C2050, 448 CUDA Cores. A multiprocessor can have 1024 threads simultaneously active, or 32 warps
	Memory	4.00 GB	8.00 GB
Software		Ubuntu 10.10(linux kernel 2.6.35–22-generic) and GCC 4.4.5 CUDA Driver Version 4.2	Red Hat Enterprise Linux Server release 5.4(2.6.18–164) and GCC 4.5.0; CUDA Driver Version 4.2

second is our two-list based sequential algorithm (here we write it as TLS for brevity) proposed in Section 3 with the time complexity $O(2^{n/2})$, which is preferable to the DP algorithm when $nc \geqslant 2^{n/2}$ for KP01.

Under test platform 1 and for each instance, we first run the DP algorithm and the TLS algorithm separately on CPU with only one thread available to obtain two sequential computational times. Second, we run COPA on CPU with only two concurrent threads available, obtaining a parallel computational time. Third, we run COPA on CPU with only four concurrent threads available, obtaining a second parallel computational time. Finally, we run COPA on GPU of GeForce GTX 470 to obtain another parallel computational time. All these computational times are calculated in milliseconds (ms), each of which is the total time consumed by the five stages of COPA. After that, the speedup is computed by Eq. (6) for multicore CPU and GPU implementations as follows:

$$Speedup = \frac{sequential\ computational\ time}{parallel\ computational\ time}. \tag{6}$$

6.2. Experimental results and evaluation

The experimental results are shown in Tables 5–7.

Table 5 shows the computational times and speedups under test platform 1 for the six instances. Column “Time” under “DP” shows the sequential computational time computed by the DP algorithm. Column “Time” under “TLS” shows the sequential computational time computed by the TLS algorithm. Column “Time” under “$k = 2$” shows the parallel computational time for only two concurrent threads available. Column “Time” under “$k = 4$” shows the parallel computational time for only four concurrent threads available. Column “Time” under “GPU” shows the parallel computational time for GPU of GeForce GTX 470. Column “Sp.-DP” shows the speedup regarding the DP algorithm as a baseline. Column “Sp.-TLS” shows

Table 5
The computational times and speedups for COPA under test platform 1 for six instances of KP01.

n	DP	TLS	k = 2			k = 4			GPU		
	Time	Time	Time	Sp.-DP	Sp.-TLS	Time	Sp.-DP	Sp.-TLS	Time	Sp.-DP	Sp.-TLS
40	94.24	214.93	134.94	0.70	1.59	92.27	1.02	2.33	33.47	2.82	6.42
42	195.15	389.70	243.51	0.80	1.60	153.74	1.27	2.53	55.51	3.52	7.02
44	411.67	821.08	512.09	0.80	1.60	320.49	1.28	2.56	115.77	3.56	7.09
46	1690.09	1103.35	686.20	2.46	1.61	431.09	3.92	2.56	152.35	11.09	7.24
48	3538.57	1680.81	1037.33	3.41	1.62	657.15	5.38	2.56	231.16	15.31	7.27
50	7418.70	3230.34	1978.27	3.75	1.63	1241.69	5.97	2.60	443.36	16.73	7.29

Sp.: speedup DP: dynamic programming [31]. The unit of “Time” is milliseconds (ms).

Table 6
The computational times under test platform 2 for the six instances of KP01.

n	DP	TLS	k = 2	k = 4	k = 8	k = 16	GPU
	Time	Time	Time	Time	Time	Time	Time
40	110.54	265.33	166.20	103.14	87.25	68.23	36.48
42	230.64	484.82	300.42	180.48	152.37	116.28	66.09
44	480.16	963.25	596.85	355.38	292.87	228.68	130.04
46	2127.87	1451.48	902.86	537.64	438.75	335.04	195.57
48	4183.82	1898.28	1174.56	705.34	576.26	439.26	255.47
50	8714.37	3698.59	2301.99	1381.85	1120.44	849.47	497.16

DP: dynamic programming [31]. The unit of “Time” is milliseconds (ms).

Table 7
The speedups under test platform 2 for six instances of KP01.

n	k = 2		k = 4		k = 8		k = 16		GPU	
	Sp.-DP	Sp.-TLS	Sp.-DP	Sp.-TLS	Sp.-DP	Sp.-TLS	Sp.-DP	Sp.-TLS	Sp.-DP	Sp.-TLS
40	0.67	1.60	1.07	2.57	1.27	3.04	1.62	3.89	3.03	7.27
42	0.77	1.61	1.28	2.69	1.51	3.18	1.98	4.17	3.49	7.34
44	0.80	1.61	1.35	2.71	1.64	3.29	2.10	4.21	3.69	7.41
46	2.36	1.61	3.96	2.70	4.85	3.31	6.35	4.33	10.88	7.42
48	3.56	1.62	5.93	2.69	7.26	3.29	9.52	4.32	16.38	7.43
50	3.79	1.61	6.31	2.68	7.78	3.30	10.26	4.35	17.53	7.44

Sp.: Speedup DP: Dynamic programming [31].

the speedup considering the TLS algorithm to be a baseline. Tables 6 and 7 show the computational times and speedups, respectively, under test platform 2 for the six instances.

According to Tables 5 and 6, when the size of the instance increases, the computational time also increases. The computational time obtained by the DP algorithm increases dramatically, whereas the computational times obtained by the TLS algorithm and COPA increase at a slower rate. This difference in computational times is because the time complexity $O(nc)$ of the DP algorithm depends on the knapsack capacity c. When c increases, the computational time also increases. However, the time complexity of the TLS algorithm $O(2^{n/2})$ and the time complexity of COPA $O(2^{n/4}(2^{n/4})^{\varepsilon})$ keep unchanged when c varies under the same hardware environment, that is, the number of the available cores is the same. When the size of the instance is small, the computational time obtained by the DP algorithm is shorter than that obtained by the TLS algorithm. The latter is almost twice as much as the former. When the size of the instance is larger, the computational time obtained by the DP algorithm is longer than that obtained by the TLS algorithm. The former is several times as much as the latter. For each instance, when the number of parallel cores increases, the computational time decreases.

From Tables 5 and 7, when the size of the instance increases, the speedup increases rapidly with the DP algorithm as a baseline while the speedup increases quite slowly with the TLS algorithm as a baseline. For each instance, when the number of parallel cores increases, the speedup increases manifestly for both baselines. In Table 7, when the size of the instance is 50, the GPU implementation can achieve the maximum speedups of 17.53 and 7.44 with the DP and TLS baselines, respectively.

Through Table 5 to 7, we conclude that the number of concurrent threads available is higher and the performance of a targeted machine is better when more cores are available for a given computation. The speedup is different for the same number of concurrent threads under different test platforms, and the average speedup is also different for different GPU. One explanation might be that the speed of slower processors has an impact on the relative speedup.

Fig. 3(a) shows speedups obtained by comparing the DP algorithm with COPA for 2 and 4 concurrent threads available and GPU under test platform 1. Fig. 3(b) shows speedups obtained by comparing the TLS algorithm with COPA for 2 and 4 concurrent threads available and GPU under test platform 1. Fig. 4(a) shows speedups obtained by comparing the DP algorithm with COPA for 2, 4, 8, and 16 concurrent threads available and GPU under test platform 2. Fig. 4(b) shows speedups obtained by comparing the TLS algorithm with COPA for 2, 4, 8, and 16 concurrent threads available and GPU under test platform 2. These four figures show that the speedups for GPU and multiple concurrent threads available are better than that for only a single thread available. These figures also indicate that the GPU implementation yields higher speedup than the CPU implementation under both test platforms visually.

Fig. 5 shows the execution time of six instances for the parallel generation stage (GS), the parallel saving max-value stage (MS) and the parallel search stage (SS) of COPA on CPU and GPU implementations under test platform 2. For simplification,

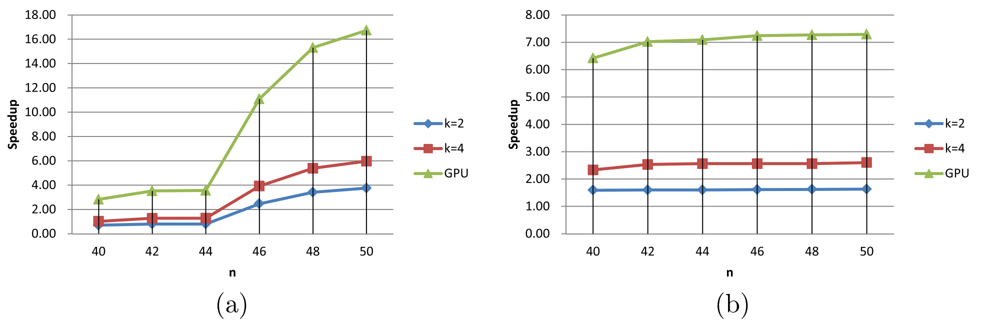

Fig. 3. The speedups of COPA on different instances under test platform 1. (a) The speedup of COPA for multi-core CPU and GPU mplementations over the DP algorithm. (b) The speedup of COPA for multi-core CPU and GPU implementations over the TLS algorithm.

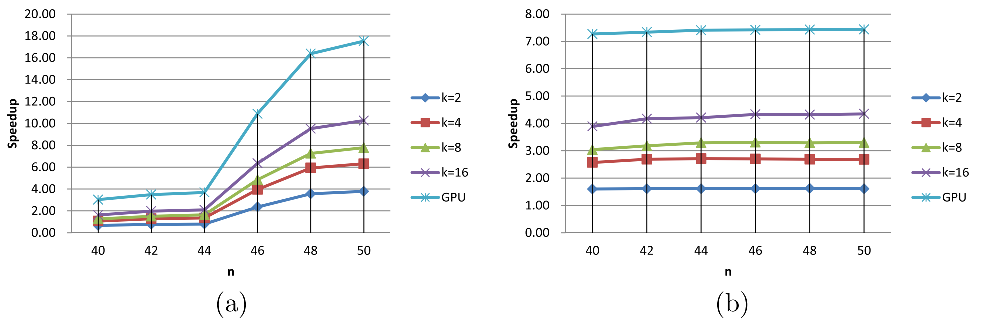

Fig. 4. The speedups of COPA on different instances under test platform 2. (a) The speedup of COPA for multi-core CPU and GPU implementations over the DP algorithm. (b) The speedup of COPA for multi-core CPU and GPU implementations over the TLS algorithm.

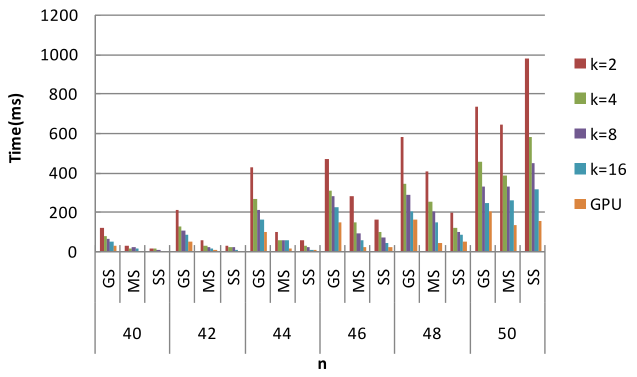

Fig. 5. Comparisons of execution times for stages GS, MS and SS of COPA on CPU and GPU implementations for all the six instances under Test Platform 2.

the two parallel saving max-value stages of COPA are incorporated into a parallel saving max-value stage. The parallel pruning stage contributes quite small amount of time without exceeding 2 ms to the total execution time. Therefore, we do not show its execution time in Fig. 5. As shown in Fig. 5, the execution time of each stage decreases with an increase in the number of concurrent cores for the same instance. During the execution of COPA, GS consumes the most time, which is sequentially followed by MS and SS.

The case of each stage for every instance under test platform 1 is similar to that in test platform 2 and is therefore omitted.

7. Conclusion

In this paper, we first present a sequential algorithm based on the two-list algorithm for KP01. Then, combining the serial algorithm and an optimal parallel merging algorithm, we design the algorithm COPA for KP01 based on an EREW PRAM model with shared memory. The proposed COPA has the advantage of better system scalability and less computational time for KP01.

The COPA is then implemented on multicore CPU architecture and GPU structure. The experimental results show that COPA is not only feasible for reducing execution time but also has improved speedup on both CPU and GPU implementations. The experimental results also indicate that the GPU implementation outperforms the multicore CPU implementation. Hence, we observe that COPA benefits more on GPU architecture.

COPA is based on a shared memory computing model, which makes it unsuitable for solving large-scale instances of KP01. In future work, COPA will be extended to deal with large-scale instance of KP01 on up-scale CPU-GPU heterogeneous parallel systems such as Tianhe supercomputer machine, which is located in National Supercomputing Center in Changsha.

Acknowledgements

The authors would like to thank the editors and anonymous reviewers for their valuable suggestions which considerably helped us to enhance this manuscript.

A preliminary version of this manuscript was presented on 19th International Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP 2011).

This work was partially funded by the Key Program of National Natural Science Foundation of China (Grant Nos. 61133005, 61432005), and the National Natural Science Foundation of China (Grant Nos. 61370095, 61472124).

References

[1] W. Diffie, M.E. Hellman, Privacy and authentication: an introduction to cryptography, Proc. IEEE 67 (1979) 397–427.
[2] H.A. Taha, Operations Research: An Introduction, vol. 8, Prentice hall Upper Saddle River, NJ, 1997.
[3] P.C. Gilmore, R.E. Gomory, A linear programming approach to the cutting-stock problem, Oper. Res. 9 (1961) 849–859.
[4] A.J. Kleywegt, J.D. Papastavrou, The dynamic and stochastic knapsack problem with random sized items, Oper. Res. 49 (2001) 26–41.
[5] J.D. Papastavrou, S. Rajagopalan, A.J. Kleywegt, The dynamic and stochastic knapsack problem with deadlines, Manage. Sci. 42 (1996) 1706–1718.
[6] P.C. Gilmore, R.E. Gomory, Multistage cutting stock problems of two and more dimensions, Oper. Res. 13 (1965) 94–120.
[7] C. Barnhart, E.L. Johnson, G.L. Nemhauser, M.W.P. Savelsbergh, P.H. Vance, Branch-and-price: column generation for solving huge integer programs, Oper. Res. 46 (1998) 316–329.
[8] A. Goldman, D. Trystram, An efficient parallel algorithm for solving the knapsack problem on hypercubes, J. Parallel Distrib. Comput. 64 (2004) 1213–1222.
[9] P.S. Gopalakrishnam, I.V. Ramakrishnam, L. Kanal, Parallel approximate algorithms for the 0–1 knapsack problem, in: 1986 International Conference on Parallel Processing, University Park, PA, 1986, pp. 444–451.
[10] G.A.P. Kindervater, J.K. Lenstra, An introduction to parallelism in combinatorial optimization, Discrete Appl. Math. 14 (1986) 135–156.
[11] J. Lee, E. Shragowitz, S. Sahni, A hypercube algorithm for the 0–1 knapsack problem, J. Parallel Distrib. Comput. 5 (1988) 438–456.

[12] J. Lin, J. Storer, Processor-efficient hypercube algorithms for the knapsack problem, J. Parallel Distrib. Comput. 13 (1991) 332–337.
[13] D. El Baz, M. Elkihel, Load balancing methods and parallel dynamic programming algorithm using dominance technique applied to the 0–1 knapsack problem, J. Parallel Distrib. Comput. 65 (2005) 74–84.
[14] S. Teng, Adaptive parallel algorithms for integral knapsack problems, J. Parallel Distrib. Comput. 8 (1990) 400–406.
[15] W. Loots, T.H.C. Smith, A parallel algorithm for the 0–1 knapsack problem, Int. J. Parallel Program. 21 (1992) 349–362.
[16] E. Horowitz, S. Sahni, Computing partitions with applications to the knapsack problem, J. ACM (JACM) 21 (1974) 277–292.
[17] R. Schroeppel, A. Shamir, A $T = O(2^{n/2}), S = O(2^{n/4})$ algorithm for certain NP-complete problems, SIAM J. Comput. 10 (1981) 456–464.
[18] C.A.A. Sanches, N.Y. Soma, H.H. Yanasse, An optimal and scalable parallelization of the two-list algorithm for the subset-sum problem, Eur. J. Oper. Res. 176 (2007) 870–879.
[19] E.D. Karnin, A parallel algorithm for the knapsack problem, IEEE Trans. Comput. 100 (1984) 404–408.
[20] H.R. Amirazizi, M.E. Hellman, Time-memory-processor trade-offs, IEEE Trans. Inf. Theory 34 (1988) 505–512.
[21] A.G. Ferreira, A parallel time/hardware tradeoff $T \cdot H = O(2^{n/2})$ for the knapsack problem, IEEE Trans. Comput. 40 (1991) 221–225.
[22] H.K.C. Chang, J.J.R. Chen, S.J. Shyu, A parallel algorithm for the knapsack problem using a generation and searching technique, Parallel Comput. 20 (1994) 233–243.
[23] D.C. Lou, C.C. Chang, A parallel two-list algorithm for the knapsack problem, Parallel Comput. 22 (1997) 1985–1996.
[24] C.A.A. Sanches, N.Y. Soma, H.H. Yanasse, Comments on parallel algorithms for the knapsack problem, Parallel Comput. 28 (2002) 1501–1505.
[25] K.L. Li, R.F. Li, Q.H. Li, Optimal parallel algorithms for the knapsack problem without memory conflicts, J. Comput. Sci. Technol. 19 (2004) 760–768.
[26] F.B. Chedid, An optimal parallelization of the two-list algorithm of cost $O(2^{n/2})$, Parallel Comput. 34 (2008) 63–65.
[27] C.A.A. Sanches, N.Y. Soma, H.H. Yanasse, Observations on optimal parallelizations of two-list algorithm, Parallel Comput. 36 (2010) 65–67.
[28] F.B. Chedid, A note on developing optimal and scalable parallel two-list algorithms, Algorithms Archit. Parallel Proces. (ICA3PP) (2012) 148–155.
[29] Q. Lin, S. Weichang, C. Jiao, G. Duan, Application of 0–1 knapsack MPI+ OpenMP hybrid programming algorithm at MH method, in: 9th International Conference on Fuzzy Systems and Knowledge Discovery (FSKD), 2012 (2012), pp. 2452–2456.
[30] S.G. Akl, N. Santoro, Optimal parallel merging and sorting without memory conflicts, IEEE Trans. Comput. 100 (1987) 1367–1369.
[31] H. Kellerer, U. Pferschy, D. Pisinger, Knapsack Problems, Springer, 2004.
[32] C.A.A. Sanches, N.Y. Soma, H.H. Yanasse, Parallel time and space upper-bounds for the subset-sum problem, Theor. Comput. Sci. 407 (2008) 342–348.