J. Parallel Distrib. Comput. 97 (2016) 112–123

Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

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

A novel cooperative accelerated parallel two-list algorithm for solving the subset-sum problem on a hybrid CPU–GPU cluster

Lanjun Wan [a,b], Kenli Li [a,b,*], Keqin Li [a,b,c]

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

HIGHLIGHTS

- A novel cooperative accelerated parallel two-list algorithm for solving SSP is explored.
- A heterogeneous cooperative computing approach for CPU–GPU clusters is proposed.
- A communication-avoiding workload distribution scheme suitable for two-list algorithm is designed.
- An efficient heterogeneous cooperative implementation of two-list algorithm is provided.

ARTICLE INFO

Article history:
Received 16 June 2015
Received in revised form
30 May 2016
Accepted 12 July 2016
Available online 18 July 2016

Keywords:
Heterogeneous cooperative computing
Hybrid CPU–GPU cluster
Hybrid programming model
Subset-sum problem
Two-list algorithm
Workload distribution

ABSTRACT

Many parallel algorithms have recently been developed to accelerate solving the subset-sum problem on a heterogeneous CPU–GPU system. However, within each compute node, only one CPU core is used to control one GPU and all the remaining CPU cores are in idle state, which leads to a large number of CPU cores being wasted. In this paper, based on a cost-optimal parallel two-list algorithm, we propose a novel heterogeneous cooperative computing approach to solve the subset-sum problem on a hybrid CPU–GPU cluster, which can make full use of all available computational resources of a cluster. The unbalanced workload distribution and the huge communication overhead are two main obstacles for the heterogeneous cooperative computing. In order to assign the most suitable workload to each compute node and reasonably partition it between CPU and GPU within each node, and minimize the inter-node and intra-node communication costs, we design a communication-avoiding workload distribution scheme suitable for the parallel two-list algorithm. According to this scheme, we provide an efficient heterogeneous cooperative implementation of the algorithm. A series of experiments are conducted on a hybrid CPU–GPU cluster, where each node has two 6-core CPUs and one GPU. The results show that the heterogeneous cooperative computing significantly outperforms the CPU-only or GPU-only computing.

1. Introduction

1.1. The problem

Given n positive integers $W = [w_1, w_2, \ldots, w_n]$ and a positive integer M, the subset-sum problem (SSP) is the decision problem of finding a set $I \subseteq \{1, 2, \ldots, n\}$, such that $\sum w_i = M,\ i \in I$. In other words, the goal is to find a binary n-tuple solution $X = [x_1, x_2, \ldots, x_n]$ for the equation

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

SSP is well-known to be NP-complete and it is a special case of the 0/1 knapsack problem. It has many real-world applications, such as capital budgeting, cargo loading, stock cutting, job scheduling and workload allocation [13,19]. Since SSP has 2^n possible subset sums of W, an exhaustive search would take $O(2^n)$ time to find a solution in the worst case. In order to solve SSP within a reasonable computation time, Horowitz and Sahni [8] proposed the sequential two-list algorithm, which solves

* Corresponding author at: College of Computer Science and Electronic Engineering, Hunan University, Changsha, Hunan, 410082, China.
E-mail addresses: wancanjun2008@163.com (L. Wan), lkl@hnu.edu.cn (K. Li), lik@newpaltz.edu (K. Li).

http://dx.doi.org/10.1016/j.jpdc.2016.07.003
0743-7315/

SSP in time $O(n2^{n/2})$ with $O(2^{n/2})$ memory space and it is the best known sequential algorithm for solving SSP. To further reduce the computation time of solving SSP, based on the SIMD (Single Instruction Multiple Data) model with shared-memory, the parallelization of the two-list algorithm has been extensively discussed in [6,12,15,18,21]. Particularly, Li et al. [15] proposed a cost-optimal parallel two-list algorithm, which solves SSP in time $O(2^{n/4}(2^{n/4})^{\varepsilon})$ with $O((2^{n/4})^{1-\varepsilon})$ processors and $O(2^{n/2})$ memory space, where $0 \leq \varepsilon \leq 1$.

1.2. Related work

In recent years, heterogeneous CPU–GPU systems have been widely used to deal with compute-intensive applications [9], because GPU can provide massive computing power and comparatively low cost. Significant effort has been done to accelerate solving the knapsack problems on a heterogeneous system. Wan et al. [25] explored a parallelization of the two-list algorithm for the SSP on an NVIDIA GPU via CUDA. Bokhari [2] implemented a dynamic programming algorithm for the SSP on a GPU. Li et al. [16] proposed an optimal parallel two-list algorithm for the 0/1 knapsack problem and evaluated its performance on a GPU. Some works [4,14,20] have also been made to accelerate solving the 0/1 knapsack problem on a GPU. These studies effectively reduce the computation time of solving the knapsack problems by using a single GPU. However, the computational and memory resources available on a single GPU are limited and may not be sufficient for solving the large-scale knapsack problems. Therefore, it is imperative to develop new techniques capable of solving the large-scale knapsack problems within a reasonable computation time. Kang et al. [11] provided an efficient parallelization of the two-list algorithm for the SSP on a hybrid CPU–GPU cluster using MPI-CUDA programming model. In addition, parallel computing with a multi-GPU cluster has also been used for solving the 0/1 knapsack problem in recent works [3,10]. Although these existing works provide significant performance benefits over the traditional CPU-based implementation, they may fail to make full use of heterogeneous computing resources consisting of CPUs and GPUs. Specifically, within each compute node, only one CPU core is used to control one GPU and all the remaining CPU cores are in idle state, which leads to large amounts of available CPU cores being wasted.

In order to make full use of the potential computing power of both CPUs and GPUs on a heterogeneous CPU–GPU system, the CPU–GPU cooperative computing has recently attracted the attention of many researchers. Wan et al. [24] proposed an efficient CPU–GPU cooperative implementation of the parallel two-list algorithm for solving SSP on a single machine with two multi-core CPUs and one GPU. Moreover, some parallel applications have also been reported to success in performing the CPU–GPU cooperative computing, such as matrix multiplication [22], Linpack [26], QR factorization [7], LU factorization [23], molecular dynamics [27], branch-and-bound algorithm [5] and divide-and-conquer algorithm [17]. These works demonstrate that the CPU–GPU cooperative computing yields better performance than the CPU-only or GPU-only computing. However, to date, how to fully exploit both CPU and GPU computing power to cooperatively accelerate solving the large-scale SSP on a hybrid CPU–GPU cluster has not been well studied.

1.3. Our contributions

In this paper, based on Li et al.'s parallel two-list algorithm [15] and our previous works [11,24,25], we propose a novel heterogeneous cooperative computing approach to solve SSP on a hybrid CPU–GPU cluster. However, an efficient heterogeneous cooperative computing is still a difficult challenge. The main difficulties are as follows: (i) how to assign the most suitable workload to each node and then reasonably partition it between CPU and GPU within each node; and (ii) how to minimize the inter-node and intra-node communication costs. To overcome these difficulties, we design a communication-avoiding workload distribution scheme suitable for the parallel two-list algorithm. According to this scheme, an efficient heterogeneous cooperative implementation of the algorithm is provided, and a series of experiments are carried out to analyze the effectiveness of our proposed approach.

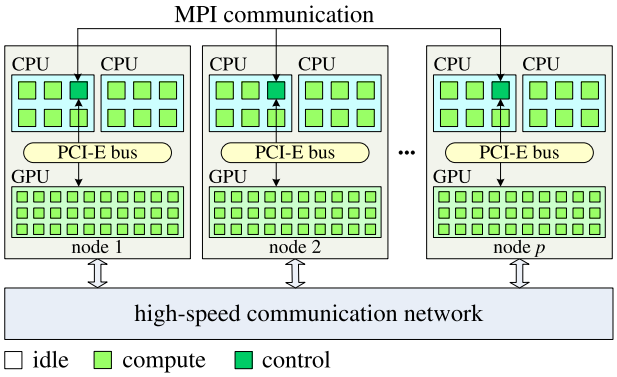


Fig. 1. A hybrid MPI-OpenMP-CUDA programming model.

The main contributions of this paper are as follows.

- A novel heterogeneous cooperative computing approach is proposed to efficiently solve SSP on a hybrid CPU–GPU cluster.
- A communication-avoiding workload distribution scheme suitable for the parallel two-list algorithm is designed to achieve inter-node and intra-node load balancing and minimize the inter-node and intra-node communication overheads.

The rest of this paper is organized as follows. Section 2 gives a heterogeneous cooperative computing approach for a hybrid CPU–GPU cluster. Section 3 presents a heterogeneous cooperative implementation of the parallel two-list algorithm for solving SSP. Section 4 describes an improved heterogeneous cooperative implementation. Section 5 gives the experimental results and performance analysis. Section 6 is conclusions and future work.

2. A heterogeneous cooperative computing approach for CPU–GPU clusters

This section describes a heterogeneous cooperative computing approach to solve SSP on a hybrid CPU–GPU cluster.

2.1. Hybrid MPI-OpenMP-CUDA programming model

Since the heterogeneous cooperative computing focuses on how to fully exploit all the available computational resources of each compute node, we introduce the commonly used hybrid MPI-OpenMP-CUDA programming model which is shown in Fig. 1. In this model, MPI is used to perform the communication operations among cluster nodes, while OpenMP and CUDA are used to drive the CPUs and GPU to perform computations, respectively.

Fig. 1 shows a hybrid CPU–GPU cluster consisting of p compute nodes, where each node has two multi-core CPUs and one GPU. Within each node, one MPI process is launched on one dedicated CPU core, and it spawns two OpenMP threads t_0 and t_1 that can run simultaneously. The thread t_0 is used to control and communicate with the GPU. Specifically, it firstly copies the required data to the GPU, then it invokes the CUDA kernel to finish the computational task assigned to the GPU, and finally it copies the results back to the CPUs. The thread t_1 spawns a suitable number of nested OpenMP threads on the CPUs so as to fully exploit all the remaining CPU cores, and these threads are used to concurrently perform the computational task assigned to the CPUs.

Table 1
Notation used in the intra-node workload distribution scheme.

Notation	Description
D	The total workload assigned to each compute node
D_{cpu}	The workload assigned to the CPUs within each compute node
D_{gpu}	The workload assigned to the GPU within each compute node
R	The ratio of the workload assigned to the CPUs over that to the GPU
V_{cpu}	The speed of the CPUs to complete the total workload D
V_{gpu}	The speed of the GPU to complete the total workload D
T_{cpu}	The time of the CPUs to complete the workload D_{cpu}
T_{gpu}	The time of the GPU to complete the workload D_{gpu}

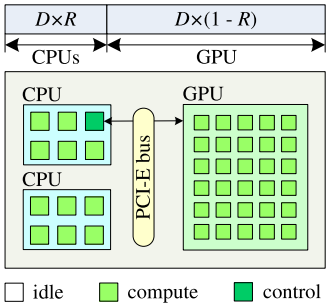


Fig. 2. The intra-node workload distribution scheme.

2.2. Workload distribution scheme

The workload distribution among compute nodes (i.e., the inter-node workload distribution) has an important impact on the performance of the heterogeneous cooperative computing. Considering that all compute nodes are homogeneous at the node level on our testing platform, namely each node has the same hardware and software configurations, the main idea of the inter-node workload distribution scheme is to evenly divide the workload into as many chunks as nodes, and one chunk is assigned to one node.

Moreover, the workload distribution between CPU and GPU within each node (i.e., the intra-node workload distribution) also can affect the overall performance. Considering that each node has two CPUs and one GPU on our testing platform, and they have different processing capabilities, memory capacities and memory bandwidths, the main idea of the intra-node workload distribution scheme is to split the workload between CPU and GPU according to an appropriate partition ratio. However, finding an optimal partition ratio is an NP-complete problem. Here we propose a simple but effective intra-node workload distribution scheme which is shown in Fig. 2, and some notation used in this scheme is listed in Table 1.

Within each node, firstly the total workload D is split into two parts D_{cpu} and D_{gpu} according to the partition ratio R; then D_{cpu} and D_{gpu} are assigned to the CPUs and the GPU, respectively; finally the CPUs and the GPU concurrently complete the workload assigned to it. The total workload is considered finished only when both the CPUs and GPU have finished their respective workload, thus the total execution time of each node is as follows:

$$\begin{cases} T_{total} = \max(T_{cpu}, T_{gpu}); \\ T_{cpu} = \dfrac{D \times R}{\frac{m-1}{m} \times V_{cpu}}; \\ T_{gpu} = \dfrac{D \times (1-R)}{V_{gpu}}. \end{cases} \tag{2}$$

In Eq. (2), m denotes the total number of CPU cores. It is easy to see that T_{total} reaches its minimum when $T_{cpu} = T_{gpu}$, namely the partition ratio can be considered optimal when both the CPUs and GPU finish their respective workload within the same amount of time. Thus, we can obtain the following equation:

$$R = \frac{1}{1 + \frac{m \times V_{gpu}}{(m-1) \times V_{cpu}}}. \tag{3}$$

From Eq. (3), we see that R is determined by V_{cpu} and V_{gpu}. To get R, programmers need to implement a given application using OpenMP and run it on the CPUs with the total workload D, and calculate V_{cpu} by dividing the total workload D by the total execution time. Similarly, programmers need to implement a given application using CUDA and run it on the GPU with the total workload D to obtain V_{gpu}. Since the optimal partition ratio is likely to change with different problem sizes and system configurations, both V_{cpu} and V_{gpu} need to be updated once the problem size or system configuration has been changed. According to R, the workload assigned to the CPUs and GPU can be calculated as follows: $D_{cpu} = \lfloor D \times R \rfloor$, and $D_{gpu} = D - D_{cpu}$.

3. A heterogeneous cooperative implementation for solving SSP

This section first introduces the parallel two-list algorithm, and then describes the heterogeneous cooperative implementation of the algorithm.

3.1. Parallel two-list algorithm

Based on Horowitz and Sahni's sequential two-list algorithm [8] and SIMD model, Li et al. [15] proposed a cost-optimal parallel two-list algorithm for solving SSP, which can be divided into three stages as follows: the parallel generation stage, the parallel pruning stage, and the parallel search stage.

The parallel generation stage, which is designed to generate two sorted lists. Firstly, we divide an n-element input vector W into two equal parts: $W_1 = [w_1, w_2, \ldots, w_{n/2}]$ and $W_2 = [w_{n/2+1}, w_{n/2+2}, \ldots, w_n]$. Secondly, we use k processors to produce $2^{n/2}$ subset sums of W_1 in parallel, and store them into the list $A = [a_1, a_2, \ldots, a_{2^{n/2}}]$ in nondecreasing order, where k is a power of 2. Thirdly, we use k processors to produce $2^{n/2}$ subset sums of W_2 in parallel, and store them into the list $B = [b_1, b_2, \ldots, b_{2^{n/2}}]$ in nonincreasing order.

The parallel pruning stage, which is used to reduce the search space. Firstly, the two sorted lists A and B are evenly divided into k blocks, respectively, where each block contains $e = 2^{n/2}/k$ elements. For clarity, let $A = [\overline{A_1}, \ldots, \overline{A_i}, \ldots, \overline{A_k}]$ and $B = [\overline{B_1}, \ldots, \overline{B_j}, \ldots, \overline{B_k}]$, where $\overline{A_i} = [\overline{a_{i.1}}, \ldots, \overline{a_{i.r}}, \ldots, \overline{a_{i.e}}]$, $\overline{B_j} = [\overline{b_{j.1}}, \ldots, \overline{b_{j.s}}, \ldots, \overline{b_{j.e}}]$ and $1 \le i, j \le k$. Each element $\overline{a_{i.r}}$ ($\overline{b_{j.s}}$) in the sublist $\overline{A_i}$ ($\overline{B_j}$) represents a subset sum of A (B), where $1 \le r, s \le e$. Secondly, the block $\overline{A_i}$ and all k blocks of the list B are assigned to the ith processor, where $1 \le i \le k$. Thirdly, the prune rule presented in [15] is used to shrink the search space. Before pruning, the number of block pairs is k^2. After pruning, the number of the picked block pairs is at most $2k - 1$. The proof of this fact is given in [15].

The parallel search stage, which is designed to find a solution from those picked block pairs. Briefly, at first those picked block pairs are evenly assigned to k processors, then each processor performs the following search subroutine to find a solution of SSP. Assuming that one block pair $(\overline{A_i}, \overline{B_j})$ has been picked and assigned to one processor, where $1 \leq i, j \leq k$. The search subroutine is described as follows.

Step 1: Initialize $r = 1$ and $s = 1$.
Step 2: **If** $\overline{a_{i.r}} + \overline{b_{j.s}} = M$ **then stop**; a solution is found.
Step 3: **If** $\overline{a_{i.r}} + \overline{b_{j.s}} < M$ **then** $r = r + 1$; **else** $s = s + 1$.
Step 4: **If** $r > e$ **or** $s > e$ **then stop**; there is no solution.
Step 5: Go to **Step 2**.

3.2. Cooperative implementation of the parallel two-list algorithm

3.2.1. Implementation of the generation stage

The generation stage begins with the initialization at each compute node. Firstly, we get an n-element input vector W and divide it into two equal parts W_1 and W_2 at the master node (node 0 for MPI). Secondly, we send W_1 and W_2 from the master node to each slave node. Thirdly, we copy them from host (CPU) to device (GPU) at each node. Finally, we initialize $A_1 = [0, w_1]$ and $B_1 = [w_{n/2+1}, 0]$ on the host side of the master node. Note that A_i represents a sublist of the list A, $A_i = [a_{i.1}, \ldots, a_{i.r}, \ldots, a_{i.2^i}]$, where $1 \leq i \leq n/2$. Each element $a_{i.r}$ in the sublist A_i represents a subset sum, where $1 \leq r \leq 2^i$.

After the initialization phase has been completed, we perform the generation procedure of the list A, which needs to execute $n/2 - 1$ iterations to complete. Each iteration consists of the following sequential activities: adding item, partitioning and merging. During the $(i-1)$th iteration, when $2 \leq i \leq \lambda - 1$, we only use the CPUs to perform the generation procedure at the master node due to the heterogeneous cooperative implementation that is not suitable for small computational task; when $\lambda \leq i \leq n/2$, we use both the CPUs and GPU to cooperatively perform the generation procedure at each compute node, where $2 \leq i \leq n/2$.

Specifically, when $\lambda \leq i \leq n/2$, during the $(i-1)$th iteration, the add item process consists of the following activities:

(1) Splitting the list A_{i-1} into p sublists of equal size, where p is the number of nodes. For clarity, let $A_{i-1} = [\overline{A_{i-1.1}}, \ldots, \overline{A_{i-1.j}}, \ldots, \overline{A_{i-1.p}}]$. The jth sublist $\overline{A_{i-1.j}}$ is copied from the master node to the $(j-1)$th slave node, where $2 \leq j \leq p$.
(2) Splitting $\overline{A_{i-1.j}}$ into two chunks according to the partition ratio R_1 at the jth node, where $1 \leq j \leq p$. For clarity, let $\overline{A_{i-1.j.cpu}}$ denote the first chunk assigned to the CPUs, and let $\overline{A_{i-1.j.gpu}}$ denote the second chunk assigned to the GPU. The sublist $\overline{A_{i-1.j.gpu}}$ is copied from host to device.
(3) Using k_{cpu} CPU threads to add the item w_i to each element of $\overline{A_{i-1.j.cpu}}$ in parallel on the host side of the jth node, generating a new sublist $\overline{A^1_{i-1.j.cpu}}$, where $k_{cpu} \leq 2^{i-1}/p \times R_1$. At the same time, we use k_{gpu} GPU threads to add w_i to each element of $\overline{A_{i-1.j.gpu}}$ in parallel on the device side of the jth node, generating a new sublist $\overline{A^1_{i-1.j.gpu}}$, where $k_{gpu} \leq 2^{i-1}/p \times (1 - R_1)$.
(4) Copying $\overline{A^1_{i-1.j.gpu}}$ from device to host, and then $\overline{A^1_{i-1.j.cpu}}$ and $\overline{A^1_{i-1.j.gpu}}$ are merged into a new sublist $\overline{A^1_{i-1.j}}$ on the host side, at the jth node.
(5) Copying $\overline{A^1_{i-1.j}}$ from the $(j-1)$th slave node to the master node, where $2 \leq j \leq p$. Then all these $p-1$ sublists and the sublist $\overline{A^1_{i-1.1}}$ are merged into a new list A^1_{i-1} at the master node.

The partition and merge processes consist of the following activities:

(1) Splitting the list A^1_{i-1} into p sublists of approximately equal size. For clarity, let $A^1_{i-1} = [\overline{A^1_{i-1.1}}, \ldots, \overline{A^1_{i-1.j}}, \ldots, \overline{A^1_{i-1.p}}]$. The jth sublist $\overline{A^1_{i-1.j}}$ is copied from the master node to the $(j-1)$th slave node, where $2 \leq j \leq p$.
(2) Splitting $\overline{A_{i-1.j}}$ into two smaller sublists $\overline{A_{i-1.j.cpu}}$ and $\overline{A_{i-1.j.gpu}}$ according to the partition ratio R_2 at the jth node, where $1 \leq j \leq p$. The sublist $\overline{A_{i-1.j.gpu}}$ is copied from host to device.
(3) Splitting $\overline{A^1_{i-1.j}}$ into two smaller sublists $\overline{A^1_{i-1.j.cpu}}$ and $\overline{A^1_{i-1.j.gpu}}$ according to R_2 at the jth node. The sublist $\overline{A^1_{i-1.j.gpu}}$ is copied from host to device.
(4) Using k_{cpu} CPU threads to merge $\overline{A_{i-1.j.cpu}}$ and $\overline{A^1_{i-1.j.cpu}}$ into a new nondecreasing list $\overline{A_{i.j.cpu}}$ in parallel by adopting the optimal parallel merging algorithm [1] on the host side of the jth node, where $k_{cpu} \leq 2^{i-1}/p \times R_2$. At the same time, we use k_{gpu} GPU threads to merge $\overline{A_{i-1.j.gpu}}$ and $\overline{A^1_{i-1.j.gpu}}$ into a new nondecreasing list $\overline{A_{i.j.gpu}}$ in parallel on the device side of the jth node, where $k_{gpu} \leq 2^{i-1}/p \times (1 - R_2)$.
(5) Copying $\overline{A_{i.j.gpu}}$ from device to host at the jth node, and then $\overline{A_{i.j.cpu}}$ and $\overline{A_{i.j.gpu}}$ are merged into a new sublist $\overline{A_{i.j}}$ on the host side, where $1 \leq j \leq p$.
(6) Copying $\overline{A_{i.j}}$ from the $(j-1)$th slave node to the master node, where $2 \leq j \leq p$. Then all these $p-1$ sublists and the sublist $\overline{A_{i.1}}$ are merged into a new list A_i at the master node.

After $n/2 - 1$ iterations have been completed, we obtain the nondecreasing list A. The generation procedure of the nonincreasing list B is almost the same as that of the nondecreasing list A, we do not discuss it for brevity.

3.2.2. Implementation of the pruning and search stages

After the two sorted lists A and B have been generated, we put the pruning and search into operations, which consist of the following activities:

(1) Splitting the list A into p equal sized sublists of $2^{n/2}/p$ elements, where p is the number of nodes. For clarity, let $A = [\overline{A_1}, \ldots, \overline{A_i}, \ldots, \overline{A_p}]$. The sublist $\overline{A_i}$ and the entire list B are copied from the master node to the $(i-1)$th slave node, where $2 \leq i \leq p$.
(2) Splitting $\overline{A_i}$ into two smaller sublists $\overline{A_{i.cpu}}$ and $\overline{A_{i.gpu}}$ according to the partition ratio R_3 at the ith node, where $1 \leq i \leq p$. The sublist $\overline{A_{i.gpu}}$ and the entire list B are copied from host to device.
(3) Splitting $\overline{A_{i.cpu}}$ and B into k_{cpu} equal sized blocks, respectively, on the host side of the ith node, where $k_{cpu} \leq 2^{n/2}/p \times R_3$ and $1 \leq i \leq p$. Similarly, we split $\overline{A_{i.gpu}}$ and B into k_{gpu} equal sized blocks, respectively, on the device side of the ith node, where $k_{gpu} \leq 2^{n/2}/p \times (1 - R_3)$.
(4) Using k_{cpu} CPU threads to perform the pruning operation in parallel on the host side of the ith node, where $1 \leq i \leq p$. At the same time, we use k_{gpu} GPU threads to perform the pruning operation in parallel on the device side of the ith node.
(5) Using k_{cpu} CPU threads to perform the search operation in parallel on the host side of the ith node, where $1 \leq i \leq p$. At the same time, we use k_{gpu} GPU threads to perform the search operation in parallel on the device side of the ith node, if a solution is found, we copy it back to the host.
(6) Copying the search results from each slave node to the master node.

It is easy to see that the above-described heterogeneous cooperative implementation needs to transfer large amounts of data between the master and slave nodes and copy data back and forth between CPU and GPU within each node. Obviously, the huge inter-node and intra-node communication overheads will become a performance bottleneck.

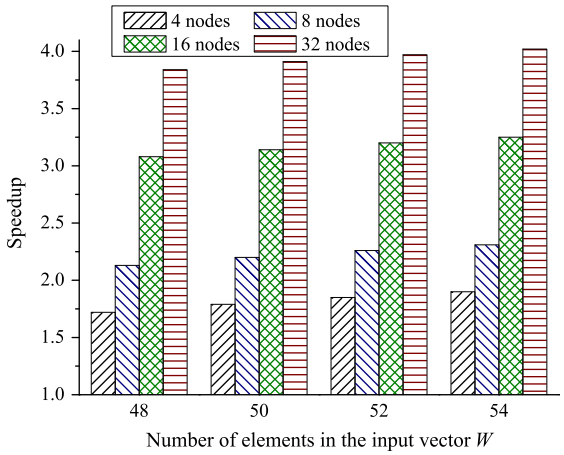


Fig. 3. The speedup of the heterogeneous cooperative implementation over the single-node CPU-only case.

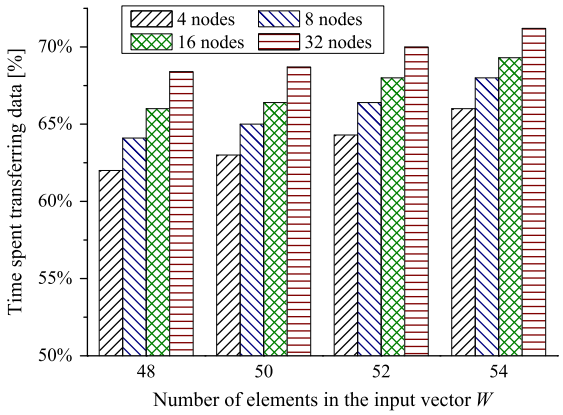


Fig. 4. The percentage of time spent on data transfer in the heterogeneous cooperative implementation.

3.3. Performance analysis of the cooperative implementation

To evaluate the performance of the heterogeneous cooperative implementation, a series of experiments are carried out on a hybrid CPU–GPU cluster. The experimental setup is described in detail in Section 5.1.

Fig. 3 shows the speedup of the heterogeneous cooperative implementation over the single-node CPU-only case for different problem sizes under the clusters with 4–32 nodes. From the figure, we see that the performance gains from using both the CPUs and GPU on multiple nodes over just using two CPUs on a single node are very modest. Specifically, compared with the single-node CPU-only case, the heterogeneous cooperative implementation only achieves an average of 0.84×, 1.25×, 2.20× and 2.96× performance improvements on 4, 8, 16 and 32 nodes, respectively.

Fig. 4 shows the percentage of time spent on data transfer in the heterogeneous cooperative implementation for different problem sizes under the clusters with 4–32 nodes. From the figure, we see that the majority of the total execution time is spent on transferring data. For example, when the problem size is fixed at 54, the data transfer dominates about 65.9%, 68.0%, 69.3% and 71.2% of the total execution time obtained on 4, 8, 16 and 32 nodes, respectively. The results demonstrate that the huge communication overhead seriously hurts the overall performance of the heterogeneous cooperative implementation. The next section will discuss how to minimize the communication overhead.

4. An improved heterogeneous cooperative implementation

This section first gives a communication-avoiding workload distribution scheme suitable for the parallel two-list algorithm, then describes an improved heterogeneous cooperative implementation of the algorithm, and finally gives a theoretic performance analysis of the improved implementation.

4.1. Communication-avoiding workload distribution scheme

Since the high communication cost greatly affects the performance of the heterogeneous cooperative implementation, a communication-avoiding workload distribution scheme suitable for the parallel two-list algorithm is designed, which not only can ensure inter-node and intra-node load balancing, but also can minimize the inter-node and intra-node communication costs.

Fig. 5 depicts the communication-avoiding workload distribution scheme for a cluster with p compute nodes, where each node has multiple CPUs and one GPU and p is a power of 2. As shown in Fig. 5, in order to achieve good load balancing, the basic idea of the inter-node workload distribution scheme is to evenly partition the workload into as many chunks as nodes, and the basic idea of the intra-node workload distribution scheme is to split the workload between CPU and GPU according to an appropriate partition ratio. Moreover, in order to minimize the inter-node and intra-node communication overheads, the scheme is designed as follows:

(1) Copying the input vector W from the master node to each slave node, and then we copy it from host to device at each node.
(2) Splitting W into four subvectors W_1, W_2, W_3 and W_4 on the host side and device side of each node, respectively. Specifically, W_1 contains the first t_1 elements of W, where $W_1 = [w_1, w_2, \ldots, w_{t_1}]$, $t_1 = \log_2 p$ and p is the number of nodes. W_2 contains the next t_2 elements of W, where $W_2 = [w_{t_1+1}, w_{t_1+2}, \ldots, w_{t_1+t_2}]$, $t_2 = \log_2 q$ and the value of q is determined by the partition ratio. W_3 contains the next t_3 elements of W, where $W_3 = [w_{t_1+t_2+1}, w_{t_1+t_2+2}, \ldots, w_{t_1+t_2+t_3}]$ and $t_3 = (n - t_1 - t_2)/2$. W_4 contains the remaining t_4 elements of W, where $W_4 = [w_{t_1+t_2+t_3+1}, w_{t_1+t_2+t_3+2}, \ldots, w_n]$ and $t_4 = n - t_1 - t_2 - t_3$.
(3) Generating the lists A, B, C and D on the host side and device side of each node, respectively. Specifically, A contains 2^{t_3} nondecreasing subset sums of W_3, where $A = [a_1, a_2, \ldots, a_{2^{t_3}}]$. B contains 2^{t_4} nonincreasing subset sums of W_4, where $B = [b_1, b_2, \ldots, b_{2^{t_4}}]$. C contains p subset sums of W_1, where $C = [c_1, c_2, \ldots, c_p]$. D contains q subset sums of W_2, where $D = [d_1, d_2, \ldots, d_q]$.
(4) Producing r and s new lists on the host side and device side of each node, respectively, based on the lists B, C and D. Specifically, we produce the list $B_{m,i}$ by adding the elements c_m (i.e., the mth element of the list C) and d_i (i.e., the ith element of the list D) to each element of the list B on the host side of the mth node, where $1 \le m \le p$ and $1 \le i \le r$. Similarly, we produce the list $B_{m,j}$ by adding the elements c_m and d_j to each element of the list B on the device side of the mth node, where $1 \le m \le p$ and $r + 1 \le j \le q$. Note that r, s and q must satisfy the following conditions: (i) $q = r + s$ and q is the power of 2; and (ii) the value of r/q is approximately equal or equal to the partition ratio R.
(5) Performing the pruning and search operations on the two sorted lists A and $B_{m,i}$ on the host side of the mth node, and meanwhile we perform the same operations on the two sorted lists A and $B_{m,j}$ on the device side of the mth node, where $1 \le m \le p$, $1 \le i \le r$ and $r + 1 \le j \le q$.
(6) Copying the search results from device to host at each node, and then we copy them from each slave node to the master node.

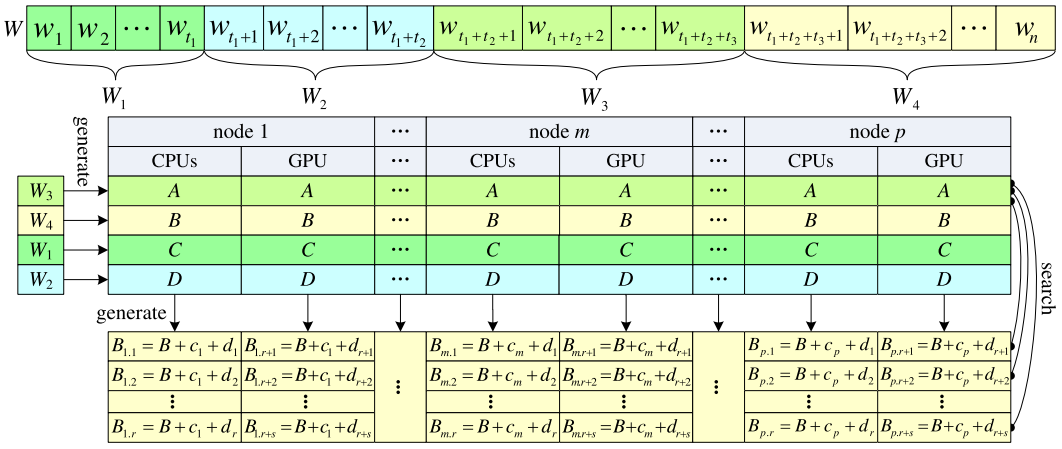


Fig. 5. The communication-avoiding workload distribution scheme for a cluster with p compute nodes.

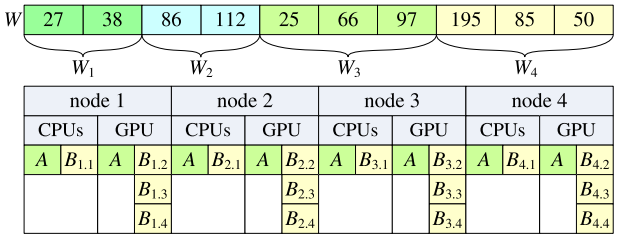


Fig. 6. An example of the workload distribution across 4 compute nodes.

To better understand the scheme, Fig. 6 shows an example of the workload distribution across 4 compute nodes with two CPUs and one GPU. For simplicity, assuming that $W = [27, 38, 86, 112, 25, 66, 97, 195, 85, 50]$ and the partition ratio $R = 23.5\%$, we split W into the following four parts: $W_1 = [27, 38]$, $W_2 = [86, 112]$, $W_3 = [25, 66, 97]$, and $W_4 = [195, 85, 50]$. After the generation stage has been completed, we obtain one nondecreasing list A and one nonincreasing list $B_{m.1}$ on the host side at the mth node, where lists A and $B_{m.1}$ contain 8 subset sums respectively and $1 \le m \le 4$. Similarly, we obtain one nondecreasing list A and three nonincreasing lists $B_{m.2}$, $B_{m.3}$ and $B_{m.4}$ on the device side at the mth node, where lists A, $B_{m.2}$, $B_{m.3}$ and $B_{m.4}$ contain 8 subset sums respectively and $1 \le m \le 4$.

As described above, it is easy to find that the communication-avoiding workload distribution scheme has very little inter-node communication cost, because only the input vector W needs to be transferred from the master node to each slave node and the search results need to be transferred from each slave node to the master node. It also has very little intra-node communication cost, because only W needs to be copied from host to device and the search results need to be copied back to the host. The effectiveness of the scheme will be discussed in detail in Section 5.2.

4.2. Inter-node parallelization with MPI

According to the above-described scheme, we propose an improved heterogeneous cooperative implementation of the parallel two-list algorithm, which is shown in Algorithm 1.

In the improved heterogeneous cooperative implementation, one MPI process is launched on a dedicated CPU core at each compute node, and it drives multiple CPUs and one GPU to carry out the following operations:

(1) Copying the input vector W from the master node to each slave node, and then we copy it from host to device at each node.
(2) Determining the values of r, s and q according to the ratio R at each node.

Algorithm 1 The improved heterogeneous cooperative implementation of the parallel two-list algorithm.

Input: A n-element input vector W, the knapsack capacity M and the partition ratio R
Output: A solution of SSP **or** NULL

1: **for all** p compute nodes **do in parallel**
2: **if** the current node is the master node **then** send W to each slave node; **end if**
3: copy W from host to device;
4: determine the values of r, s and q based on the following conditions: $q = r + s$, $r/q \cong R$, and q is the power of 2;
5: *omp_set_nested*(1);
6: #pragma omp parallel num_threads(2) {
7: $omp\_tid = omp\_get\_thread\_num()$;
8: **if** $omp\_tid = 0$ **then**
9: split W into W_1, W_2, W_3 and W_4 on the device side, which contain t_1, t_2, t_3 and t_4 elements of W, respectively, where $t_1 = \log_2 p$, $t_2 = \log_2 q$, $t_3 = (n - t_1 - t_2)/2$ and $t_4 = n - t_1 - t_2 - t_3$;
10: use k_{gpu1} GPU threads to generate the nondecreasing list A in parallel on the device side based on W_3, where $k_{gpu1} \le 2^{t_3}$;
11: use k_{gpu2} GPU threads to generate the nonincreasing list B in parallel on the device side based on W_4, where $k_{gpu2} \le 2^{t_4}$;
12: use k_{gpu3} GPU threads to generate the list C in parallel on the device side based on W_1, where $k_{gpu3} \le p$;
13: use k_{gpu4} GPU threads to generate the list D in parallel on the device side based on W_2, where $k_{gpu4} \le q$;
14: use k_{gpu5} GPU threads to generate the list $B_{m.j}$ in parallel on the device side based on the lists B, C and D, where m is the node ID of the current node, $r + 1 \le j \le q$ and $k_{gpu5} \le 2^{t_4}$;
15: use k_{gpu6} GPU threads to perform the pruning operation on the two sorted lists A and $B_{m.j}$ in parallel on the device side to reduce the search space, where $r + 1 \le j \le q$;
16: use k_{gpu6} GPU threads to perform the search operation on the two sorted lists A and $B_{m.j}$ in parallel on the device side to find a solution of SSP, where $r + 1 \le j \le q$;
17: copy the search results back to the host;
18: **else**
19: split W into W_1, W_2, W_3 and W_4 on the host side, which contain t_1, t_2, t_3 and t_4 elements of W, respectively, where $t_1 = \log_2 p, t_2 = \log_2 q, t_3 = (n - t_1 - t_2)/2$ and $t_4 = n - t_1 - t_2 - t_3$;
20: use k_{cpu1} CPU threads to generate the nondecreasing list A in parallel on the host side based on W_3, where $k_{cpu1} \le 2^{t_3}$;
21: use k_{cpu2} CPU threads to generate the nonincreasing list B in parallel on the host side based on W_4, where $k_{cpu2} \le 2^{t_4}$;
22: use k_{cpu3} CPU threads to generate the list C in parallel on the host side based on W_1, where $k_{cpu3} \le p$;
23: use k_{cpu4} CPU threads to generate the list D in parallel on the host side based on W_2, where $k_{cpu4} \le q$;
24: use k_{cpu5} CPU threads to generate the list $B_{m.i}$ in parallel on the host side based on the lists B, C and D, where m is the node ID of the current node, $1 \le i \le r$ and $k_{cpu5} \le 2^{t_4}$;
25: use k_{cpu6} CPU threads to perform the pruning operation on the two sorted lists A and $B_{m.i}$ in parallel on the host side to reduce the search space, where $1 \le i \le r$;
26: use k_{cpu6} CPU threads to perform the search operation on the two sorted lists A and $B_{m.i}$ in parallel on the host side to find a solution of SSP, where $1 \le i \le r$;
27: **end if** }
28: **if** the current node is the slave node **then** send the search results to the master node; **end if**
29: **end for**

(3) Using two concurrent CPU threads to control the cooperative execution of the three stages of the algorithm at each node. CPU thread 0 is used to control the GPU, including the following main activities: (i) splitting W into four parts W_1, W_2, W_3 and W_4 on the device side; (ii) using a suitable number of GPU threads to generate the lists A, B, C and D in parallel based on W_3, W_4, W_1 and W_2, respectively; (iii) using a suitable number of GPU threads to generate the list $B_{m.j}$ in parallel based on the lists B, C and D, where m is the node ID of the current node and $r+1 \le j \le q$; (iv) using a suitable number of GPU threads to perform the pruning and search operations on the sorted lists A and $B_{m.j}$, where $r+1 \le j \le q$; and (v) copying the search results back to the host. CPU thread 1 is used to control the CPUs, including the following main activities: (i) splitting W into four parts W_1, W_2, W_3 and W_4 on the host side; (ii) using a suitable number of CPU threads to generate the lists A, B, C and D in parallel based on W_3, W_4, W_1 and W_2, respectively; (iii) using a suitable number of CPU threads to generate the list $B_{m.i}$ in parallel based on the lists B, C and D, where $1 \le i \le r$; (iv) using a suitable number of CPU threads to perform the pruning and search operations on the sorted lists A and $B_{m.i}$.

(4) Copying the search results from each slave node to the master node and output them.

4.3. Intra-node parallelization with OpenMP and CUDA

In the improved heterogeneous cooperative implementation, within each compute node, the algorithm is implemented on the CPUs and GPU using OpenMP and CUDA, respectively. Since it is easy to implement the algorithm using OpenMP, we only describe the latter for the sake of brevity.

On the device side of each node, the implementation of the generation stage consists of the following three steps.

Step 1: Producing the sorted lists A and B. Specifically, we first initialize $A_1 = [0, w_{t_1+t_2+1}]$. Then, we execute $t_3 - 1$ iterations to obtain A. During the ith ($2 \le i \le t_3$) iteration, we perform the following operations: (i) using $k_{gpu1} = \min(g_{\max}, 2^{i-1})$ GPU threads to add the item $w_{t_1+t_2+i}$ (i.e., the ith item of the subvector W_3) to each element of the list $A_{i-1} = [a_{i-1.1}, a_{i-1.2}, \ldots, a_{i-1.2^{i-1}}]$ in parallel, generating a new list $A^1_{i-1} = [a_{i-1.1} + w_{t_1+t_2+i}, a_{i-1.2} + w_{t_1+t_2+i}, \ldots, a_{i-1.2^{i-1}} + w_{t_1+t_2+i}]$, where $g_{\max}$ is the maximum number of threads supported on the GPU; (ii) using k_{gpu1} GPU threads to concurrently merge lists A_{i-1} and A^1_{i-1} into a new nondecreasing list A_i by adopting the optimal parallel merging algorithm [1]. After $t_3 - 1$ iterations have been processed, we finally obtain the nondecreasing list A with 2^{t_3} subset sums. The procedure of generating the nonincreasing list B and that of generating the list A are similar, we omit it for clarity.

Step 2: Producing the lists C and D. Specifically, we first initialize $C_1 = [0, w_1]$. Then, we execute $t_1 - 1$ iterations to obtain C. During the ith ($2 \le i \le t_1$) iteration, we perform the following operations: (i) using $k_{gpu3} = \min(g_{\max}, 2^{i-1})$ GPU threads to add the item w_i (i.e., the ith item of the subvector W_1) to each element of the list $C_{i-1} = [c_{i-1.1}, c_{i-1.2}, \ldots, c_{i-1.2^{i-1}}]$ in parallel, generating a new list $C^1_{i-1} = [c_{i-1.1} + w_i, c_{i-1.2} + w_i, \ldots, c_{i-1.2^{i-1}} + w_i]$; (ii) using k_{gpu3} GPU threads to concurrently merge lists C_{i-1} and C^1_{i-1} into a new list C_i. After $t_1 - 1$ iterations have been processed, we finally obtain the list C with p subset sums. The procedure of generating the list D is almost the same as that of generating the list C, we omit it for brevity.

Step 3: Producing s new lists based on lists B, C and D. Specifically, we use k_{gpu5} GPU threads to add the element c_m of the list C and the element d_j of the list D into each element of the list B in parallel, generating a new list $B_{m.j} = [b_1 + c_m + d_j, b_2 + c_m + d_j, \ldots, b_{2^{t_4}} + c_m + d_j]$, where m is the node ID of the current node and $r+1 \le j \le r+s$.

Algorithm 2 The pruning operation implemented on the GPU.

Input: The knapsack capacity M, the nondecreasing list A and s different nonincreasing lists $B_{m.j}$, where $r+1 \le j \le r+s$.
Output: All the picked block pairs

```
1: for j = r + 1 to r + s do
2:   for all k_gpu6 GPU threads do in parallel
3:     if A_u and B_m.j are assigned to the uth thread, where u is the current thread
       ID and 1 ≤ u ≤ k_gpu6 then
4:       for v = 1 to k do
5:         X = a_u.1 + b_m.j.v.e_B, Y = a_u.e_A + b_m.j.v.1;
6:         if X < M and Y > M then pick the block pair (A_u, B_m.j.v);
7:         else discard the block pair (A_u, B_m.j.v); end if
8:       end for
9:     end if
10:  end for
11: end for
```

After the sorted lists A and $B_{m.j}$ have been generated on the device side of the mth node, we perform the pruning operation described in Algorithm 2 to reduce the search space, where $1 \le m \le p$ and $r+1 \le j \le r+s$. Specifically, we first split the list A into k_{gpu6} equal sized blocks of $e_A = 2^{t_3}/k_{gpu6}$ elements, and also split the list $B_{m.j}$ into k_{gpu6} equal sized blocks of $e_B = 2^{t_4}/k_{gpu6}$ elements. To facilitate our discussion, let $A = [\overline{A_1}, \ldots, \overline{A_u}, \ldots, \overline{A_{k_{gpu6}}}]$ and $B_{m.j} = [\overline{B_{m.j.1}}, \ldots, \overline{B_{m.j.v}}, \ldots, \overline{B_{m.j.k_{gpu6}}}]$, where $\overline{A_u} = [\overline{a_{u.1}}, \overline{a_{u.2}}, \ldots, \overline{a_{u.e_A}}]$ and $\overline{B_{m.j.v}} = [\overline{b_{m.j.v.1}}, \overline{b_{m.j.v.2}}, \ldots, \overline{b_{m.j.v.e_B}}]$. Secondly, we assign the block $\overline{A_u}$ and the entire list $B_{m.j}$ to the uth GPU thread, where $1 \le u \le k_{gpu6}$. Finally, based on the prune rule presented in [15], we use k_{gpu6} GPU threads to perform the pruning operation on the two sorted lists A and $B_{m.j}$ in parallel. Before pruning, the number of block pairs is $s \times k^2_{gpu6}$. After pruning, the number of the picked block pairs is at most $s \times (2k_{gpu6} - 1)$.

Algorithm 3 The search operation implemented on the GPU.

Input: The knapsack capacity M, the nondecreasing list A and s different nonincreasing lists $B_{m.j}$, where $r+1 \le j \le r+s$.
Output: The search results

```
1: for all k_gpu6 GPU threads do in parallel
2:   if one block pair (A_u, B_m.j.v) is assigned to the current thread, where r+1 ≤ j ≤
     r + s and 1 ≤ u, v ≤ k_gpu6 then
3:     x = 1, y = 1;
4:     while x ≤ e_A and y ≤ e_B do
5:       if a_u.x + b_m.j.v.y = M then stop;
6:       else if a_u.x + b_m.j.v.y < M then x = x + 1;
7:       else y = y + 1; end if
8:     end while
9:   end if
10: end for
```

After the pruning operation has been completed, we perform the search operation described in Algorithm 3 to find a solution of SSP on the device side of the mth node, where $1 \le m \le p$. Specifically, those picked block pairs are evenly assigned to k_{gpu6} GPU threads, and each GPU thread will search $2s$ block pairs at most. Let us suppose that one block pair $(\overline{A_u}, \overline{B_{m.j.v}})$ is assigned to one GPU thread, where $r+1 \le j \le r+s$ and $1 \le u, v \le k_{gpu6}$. The GPU thread finds the top elements of the block pair $(\overline{A_u}, \overline{B_{m.j.v}})$, if $\overline{a_{u.1}} + \overline{b_{m.j.v.1}} = M$, meaning that a solution is found; otherwise, it continues to search the next element of $\overline{A_u}$ or $\overline{B_{m.j.v}}$. The search process is repeated until the last element of $\overline{A_u}$ or $\overline{B_{m.j.v}}$ has been retrieved. After the search operation has been completed, the search results are copied back to the host.

Last but not least, the grid size and thread block size have a great effect on the performance of the CUDA-based implementation. The fact that the optimal thread block size can obtain the best GPU multiprocessor occupancy has been confirmed in our previous work [25]. To achieve better performance on the GPU, we determine the optimal grid size according to the workload assigned

Table 2
The performance comparison between OHCI and IHCI for different problem sizes.

n	4 nodes			8 nodes			16 nodes			32 nodes		
	OHCI	IHCI	Benefit	OHCI	IHCI	Benefit	OHCI	IHCI	Benefit	OHCI	IHCI	Benefit
48	251	121	1.07×	203	90	1.25×	140	59	1.36×	112	45	1.52×
50	448	223	1.01×	365	166	1.20×	255	109	1.34×	205	82	1.51×
52	826	423	0.95×	677	315	1.15×	477	206	1.31×	385	154	1.50×
54	1613	847	0.90×	1328	628	1.11×	943	411	1.29×	763	306	1.49×
56	3178	1701	0.87×	2627	1258	1.09×	1873	824	1.27×	1520	613	1.48×
58	–	–	–	5230	2531	1.07×	3743	1656	1.26×	3042	1231	1.47×
60	–	–	–	–	–	–	–	–	–	6103	2476	1.47×

OHCI: the original heterogeneous cooperative implementation.
IHCI: the improved heterogeneous cooperative implementation.

to the GPU, and use the CUDA Occupancy Calculator provided by NVIDIA SDK to calculate the optimal thread block size.

4.4. Theoretic performance analysis of the improved implementation

The theoretic performance analysis of the improved heterogeneous cooperative implementation is as follows. When a hybrid CPU–GPU cluster has p compute nodes with k available CPU and GPU threads each, the improved heterogeneous cooperative implementation can solve SSP in time $O(2^{n/2}/(p \times k))$, and each node needs $O(2^{n/2}/p)$ memory space, where both p and k are a power of 2. It is easy to see that the improved heterogeneous cooperative implementation can produce significantly better performance than the sequential two-list algorithm [8] with $O(n2^{n/2})$ time complexity and $O(2^{n/2})$ memory requirement.

5. Experimental evaluation

5.1. Experimental setup

Our experiments are carried out on a hybrid CPU–GPU cluster with 32 compute nodes. Each compute node has two 6-core Intel Xeon X5670 CPUs, one NVIDIA Tesla M2050 GPU, 32 GB of main memory and 3 GB of device memory. All compute nodes are interconnected through a high-speed communication network, which is constructed by high-radix Network Routing Chips (NRC) and high-speed Network Interface Chips (NIC), and both of them were designed by the National University of Defence Technology. In our multi-node experiments, each compute node is configured with one MPI process, and the version of MPI is MPICH2-1.2. In addition, the compilers used are GCC version 4.4.7 and NVIDIA NVCC version 5.0.

In order to accurately evaluate the performance of the improved heterogeneous cooperative implementation, we use the other four different methods to implement the parallel two-list algorithm: (1) single-node CPU-only case, namely we implement it on two CPUs using OpenMP; (2) single-node GPU-only case, namely we implement it on the GPU using CUDA; (3) multi-node pure-CPU case, namely we implement it on multiple nodes using MPI and OpenMP, but only the CPUs are used within each node; (4) multi-node pure-GPU case, namely we implement it on multiple nodes using MPI and CUDA, but only the GPU plus one CPU core are used within each node. Note that the multi-node pure-CPU and pure-GPU cases adopt the communication-avoiding workload distribution scheme. In addition, in order to analyze the scalability of the improved heterogeneous cooperative implementation, the best sequential CPU implementation is examined, namely we run Horowitz and Sahni's sequential two-list algorithm [8] on a single CPU core.

Since Horowitz and Sahni's sequential two-list algorithm [8] and Li et al.'s parallel two-list algorithm [15] all require $O(2^{n/2})$ memory spaces, namely the memory requirement increases exponentially with increasing problem size, implying that the problem size will be limited by the available memory. In view of this, we test seven different problem sizes which scale from 48 to 60. Specifically, the number of elements in the input vector W is specified as 48, 50, 52, 54, 56, 58 or 60.

For each problem size, we specify $M = 0.5 \sum_{i=1}^{n} w_i$ and use a random number generator to produce 100 different instances of SSP, and the average execution time of these instances is measured in milliseconds (ms). The execution time includes the computation time, data transfer time and synchronization overhead, but the data preparation time, initialization time and memory allocation and deallocation overheads are excluded.

5.2. Evaluation of the communication-avoiding workload distribution scheme

To evaluate the effectiveness of the communication-avoiding workload distribution scheme, we conduct a series of experiments to compare the performance of the original heterogeneous cooperative implementation (OHCI) described in Section 3.2 with that of the improved heterogeneous cooperative implementation (IHCI) described in Section 4.

Table 2 shows the performance comparison between OHCI and IHCI for different problem sizes under the clusters with 4–32 nodes. Due to the Tesla M2050 GPU has only 3 GB of device memory, the problem size is restricted to the available memory, e.g., the upper limit is $n = 56$ for a cluster with 4 nodes. In Table 2, the columns marked as "OHCI" and "IHCI" represent the execution time of the OHCI and that of the IHCI, respectively. Compared with the OHCI, the IHCI achieves an average of 0.96×, 1.15×, 1.31× and 1.49× performance improvements on 4, 8, 16 and 32 nodes, respectively. The results show that our proposed communication-avoiding workload distribution scheme significantly improves the performance of the heterogeneous cooperative implementation.

In order to clearly explain the reasons for the performance improvement, for each instance of SSP, we measure the computation time and communication time of the OHCI and the IHCI, respectively. Fig. 7 shows the execution time comparison between OHCI and IHCI for different cluster sizes when the problem size is fixed at 52, and Fig. 8 shows the execution time comparison between OHCI and IHCI for different problem sizes under the cluster with 32 nodes. Note that the execution time is mainly composed of computation time and communication time. In Figs. 7 and 8, the bars marked as "OHCI-comp" and "IHCI-comp" represent the computation time of the OHCI and that of the IHCI, respectively. The bars marked as "OHCI-comm" and "IHCI-comm" denote the communication time of the OHCI and that of the IHCI, respectively.

From Figs. 7 and 8, we see that the computation time of the IHCI is larger than that of the OHCI, but the communication time of the IHCI is far smaller than that of the OHCI, for different problem sizes and cluster sizes. Compared with the OHCI, although the IHCI increases the computation cost, it avoids the huge communication overhead. Due to the decrement of the communication cost being

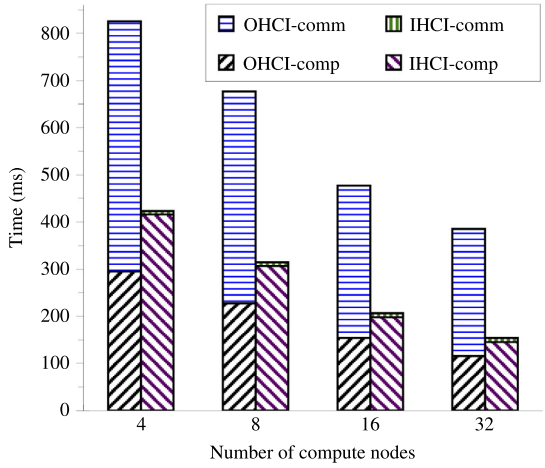


Fig. 7. The execution time comparison between OHCI and IHCI for different cluster sizes when the problem size is fixed at 52.

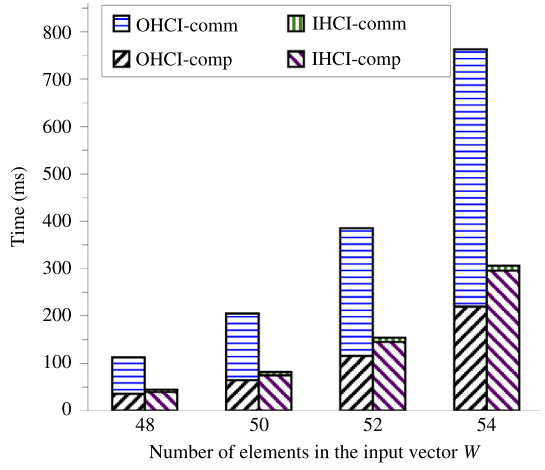


Fig. 8. The execution time comparison between OHCI and IHCI for different problem sizes under the cluster with 32 nodes.

far more than the increment of the computation cost, the total execution time has been greatly reduced. For instance, when the problem size is fixed at 52, compared with the OHCI, the computation times of the IHCI are increased by 41.01%, 34.82%, 28.56% and 25.89% on 4, 8, 16 and 32 nodes respectively, the communication times of the IHCI are reduced by 98.64%, 98.23%, 97.43% and 96.79% on 4, 8, 16 and 32 nodes respectively, and the total execution times of the IHCI are reduced by 48.73%, 53.53%, 56.78% and 60.00% on 4, 8, 16 and 32 nodes respectively. Similarly, when the cluster size is fixed at 32, compared with the OHCI, the computation times of the IHCI are increased by 10.35%, 16.09%, 25.89% and 34.60% when problem size $n = 48$, 50, 52 and 54 respectively, the communication times of the IHCI are reduced by 93.10%, 95.05%, 96.79% and 98.04% when problem size $n = 48$, 50, 52 and 54 respectively, and the total execution times of the IHCI are reduced by 60.35%, 60.23%, 60.00% and 59.85% when problem size $n = 48$, 50, 52 and 54 respectively. The results reveal that the improvement grows with the number of nodes but decreases with the problem size.

5.3. Scalability of the improved heterogeneous cooperative implementation

In this section, we conduct both weak scalability and strong scalability experiments to evaluate the scalability of the IHCI.

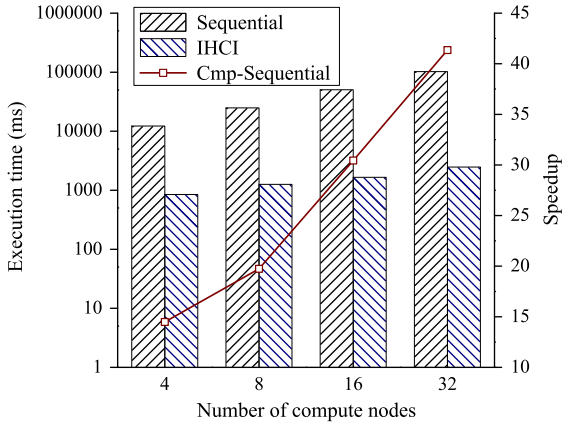


Fig. 9. The performance comparison between the IHCI and the sequential CPU implementation with increasing number of nodes and increasing problem size (weak scalability).

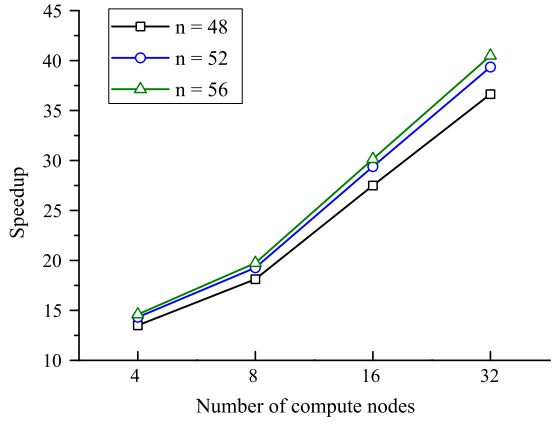


Fig. 10. The speedup of the IHCI over the sequential CPU implementation with increasing number of nodes at a fixed problem size (strong scalability).

In the weak scalability experiments, when the number of compute nodes increases from 4 to 32, the problem size increases from 54 to 60 accordingly. Fig. 9 shows the results of the weak scalability tests. In the figure, the left y-axis shows the execution time on a logarithmic scale, and the line "Cmp-Sequential" refers to the speedup of the IHCI over the sequential CPU implementation. As shown in Fig. 9, for the IHCI, the performance gained increases with increasing number of nodes. Compared with the sequential CPU implementation, the execution times of the IHCI are reduced by 93.09%, 94.93%, 96.72% and 97.58% on 4, 8, 16 and 32 nodes, respectively, and the IHCI achieves speedups from 14.48× to 41.35× with the number of nodes increasing from 4 to 32. The results show that the IHCI can solve potentially larger problems when more computing resources are available.

In the strong scalability experiments, when the number of compute nodes increases from 4 to 32, the problem size remains fixed. Fig. 10 depicts the results of the strong scalability tests. From Fig. 10, we see that the speedup of the IHCI over the sequential CPU implementation gradually increases with increasing number of nodes. For example, when the problem size is fixed at 52, the speedup increases from 14.32× to 39.35× as the number of nodes increases from 4 to 32. The results show that the IHCI scales well with the number of nodes, indicating that it can solve a given problem more rapidly if we are provided with more computing resources.

The above results demonstrate that the IHCI not only has good weak scalability, but also has good strong scalability.

Table 3
The execution time of seven different implementations for different problem sizes.

n	Sequential	Single-node parallel impl.			Multi-node parallel implementations											
		CPU-only	GPU-only	CPU + GPU	4 nodes			8 nodes			16 nodes			32 nodes		
					pure-CPU	pure-GPU	CPU + GPU	pure-CPU	pure-GPU	CPU + GPU	pure-CPU	pure-GPU	CPU + GPU	pure-CPU	pure-GPU	CPU + GPU
48	1 632	431	317	231	232	171	121	174	128	90	116	85	59	88	65	45
50	3 123	802	584	428	429	314	223	321	235	166	213	156	109	161	118	82
52	6 063	1 529	1113	818	818	595	423	611	445	315	405	295	206	306	223	154
54	12 257	3 067	2232	1645	1 637	1190	847	1223	888	628	808	587	411	608	442	306
56	24 837	6 171	–	–	3 289	2386	1701	2449	1776	1258	1620	1175	824	1219	884	613
58	50 409	12 443	–	–	6 579	–	–	4928	3570	2531	3255	2359	1656	2445	1772	1231
60	102 390	25 085	–	–	13 239	–	–	9842	–	–	6495	–	–	4919	3560	2476

CPU + GPU: the improved heterogeneous cooperative implementation.

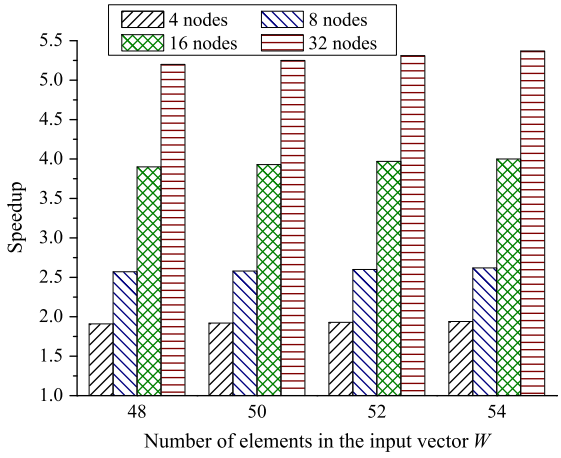


Fig. 11. The speedup of the multi-node IHCI (CPU + GPU) over the single-node IHCI (CPU + GPU) for different problem sizes under the clusters with 4–32 nodes.

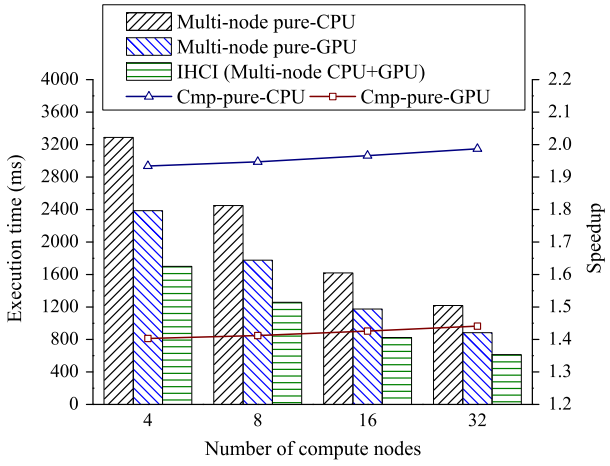


Fig. 12. The performance comparison among the IHCI, the multi-node pure-CPU and pure-GPU implementations for different cluster sizes when the problem size $n = 56$.

5.4. Comparison with single-node parallel implementation

In this section, the performance of the multi-node IHCI is compared with that of the single-node parallel implementations.

Table 3 shows the execution time of seven different implementations. From Table 3, it can be observed that the multi-node parallel implementations are clearly better than the single-node parallel implementations. For example, compared with the single-node IHCI (CPU + GPU), the execution times of the multi-node IHCI (CPU + GPU) are reduced by an average of 48.13%, 61.43%, 74.67% and 81.06% on 4, 8, 16 and 32 nodes, respectively.

Fig. 11 shows the speedup of the multi-node IHCI over the single-node IHCI for different problem sizes. As shown in Fig. 11, when the number of compute nodes increases from 4 to 32 and the problem size is fixed at 54, the speedup compared with the single-node IHCI increases from 1.94× to 5.37×. The results reveal that the more compute nodes of a cluster are available for the heterogeneous cooperative implementation, the better the performance.

5.5. Comparison with multi-node parallel implementation

Fig. 12 shows the performance comparison among the IHCI, the multi-node pure-CPU and pure-GPU implementations for different cluster sizes when the problem size is fixed at 56. From Fig. 12, we find that the IHCI clearly outperforms the multi-node pure-CPU or pure-GPU case, this is because all the available computational resources of both the CPUs and GPU have been fully utilized within each node. Fig. 12 also shows the speedup of the IHCI over the multi-node pure-CPU case and that of the IHCI over the multi-node pure-GPU case. It can be found that the speedup slowly increases with increasing number of nodes, and it will gradually reach a peak. The results reveal that the IHCI can scale well on more nodes when solving SSP, this is because there is very little inter-node and intra-node communication overheads and the workload per node becomes smaller with increasing number of nodes. The results also indicate that the performance achieved is stable in the IHCI.

Moreover, from Fig. 12, we can see that using 32 nodes is only 34.42% faster than using 16 nodes for the IHCI, this is mainly because that the workload assigned to each node within the cluster with 32 nodes is only reduced by 31.25% in comparison with the workload assigned to each node within the cluster with 16 nodes.

5.6. Performance evaluation on single-node multi-GPU platform

In order to evaluate the performance of the IHCI more effectively, we carry out a series of experiments on a single-node multi-GPU platform, which has two 6-core Intel Xeon E5-2620 CPUs and two NVIDIA Tesla M2090 GPUs. According to different hardware configurations, the following three different heterogeneous cooperative implementations are evaluated: IHCI (2GPUs), IHCI (2CPUs + 1GPU) and IHCI (2CPUs + 2GPUs). Note that for the IHCI (2GPUs), the two CPUs play a very simple role, only two CPU cores are used to control two GPUs, and all the remaining CPU cores are in idle state.

Fig. 13 shows the performance comparison among the IHCI, the CPU-only and GPU-only implementations on a single-node multi-GPU platform. In Fig. 13, the x-axis shows the problem size scales from 48 to 56, and the y-axis shows the execution time normalized to the running time of the IHCI (2CPUs + 2GPUs). The results show that the IHCI yields better performance than the CPU-only or GPU-only case. Compared with the GPU-only case, the execution times of the IHCI (2GPUs), IHCI (2CPUs + 1GPU) and IHCI (2CPUs + 2GPUs) are reduced by an average of 34.88%,

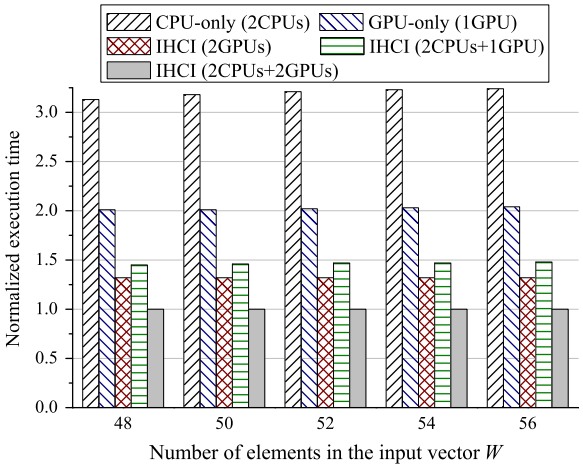


Fig. 13. The performance comparison among the IHCI, the CPU-only and GPU-only implementations on a single-node multi-GPU platform.

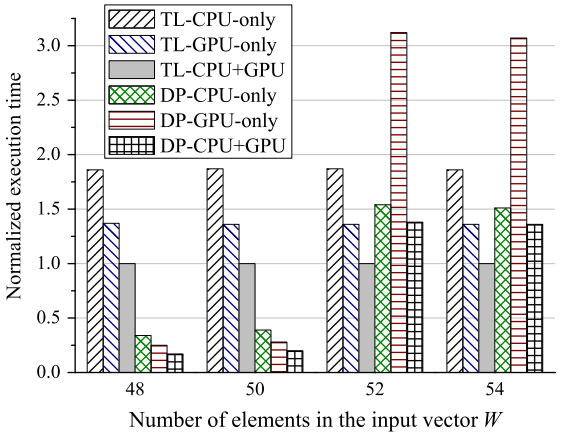


Fig. 14. The performance comparison between the parallel two-list algorithm and the parallel dynamic programming algorithm on a single cluster node.

27.47% and 50.56%, respectively. The results also show that the IHCI (2CPUs + 2GPUs) is faster than the IHCI (2GPUs) and the IHCI (2CPUs + 1GPU). Compared with the IHCI (2CPUs + 1GPU), the execution time of the IHCI (2CPUs + 2GPUs) is reduced by an average of 31.84%. This experiment demonstrates that our proposed heterogeneous cooperative computing approach can fully exploit the computing power of a heterogeneous system with different number and kinds of compute devices.

5.7. Comparison with dynamic programming algorithm

In this section, the performance of the parallel two-list algorithm is compared with that of the parallel dynamic programming (DP) algorithm. The DP algorithm [19] is one of the best known approaches to exactly solve SSP, which solves SSP in $O(nM)$ time and memory space. When solving SSP in parallel, the time required is $O(nM/k)$, where k is the number of processors. Since the performance of the DP algorithm is sensitive to the knapsack capacity M, for the sake of fairness, we choose the small knapsack capacity $M = 2^{n/2}/64$ when problem sizes $n = \{48, 50\}$ and the large knapsack capacity $M = 2^{n/2}/8$ when problem sizes $n = \{52, 54\}$ in this experiment.

Fig. 14 shows the performance comparison between the parallel two-list algorithm and the parallel DP algorithm on a single node of the hybrid CPU–GPU cluster. In Fig. 14, the x-axis shows the problem size scales from 48 to 54, and the y-axis shows the execution time normalized to the running time of the CPU–GPU cooperative implementation of the parallel two-list algorithm. The results show that the performance of the parallel DP algorithm is significantly better than that of the parallel two-list algorithm when the problem size is fixed at 48 or 50, while the parallel two-list algorithm outperforms the parallel DP algorithm when the problem size is fixed at 52 or 54. The results demonstrate that the smaller the knapsack capacity is, the better the performance achieved for the DP algorithm. Hence, the parallel DP algorithm is suitable for moderate knapsack capacity, but the parallel two-list algorithm is attractive for large knapsack capacity. In addition, the GPU-only implementation of the DP algorithm has much worse performance than the CPU-only case when the problem size is fixed at 52 or 54, this is because the large knapsack capacity leads to huge CPU–GPU communication overhead. The huge communication cost will also seriously degrade the performance of the single-node CPU–GPU cooperative implementation and the multi-node parallel implementation of the DP algorithm.

6. Conclusions and future work

In this paper, a novel cooperative accelerated parallel two-list algorithm is proposed to efficiently solve SSP on a hybrid CPU–GPU cluster. To achieve good inter-node and intra-node load balancing and minimize the inter-node and intra-node communication overheads, a communication-avoiding workload distribution scheme suitable for the parallel two-list algorithm is designed, and the experimental results confirm that the proposed scheme can effectively accelerate the algorithm. According to the scheme, an efficient heterogeneous cooperative implementation of the algorithm is provided, and the experimental results show that it not only achieves significant performance benefit over the multi-node pure-CPU or pure-GPU case by fully exploiting all available computational resources of a cluster, but also has good scalability. Although our proposed heterogeneous cooperative computing method is targeted for a cluster, where each node is configured with two CPUs and one GPU, the method can be easily employed to a cluster, where each node has multiple CPUs and multiple accelerators (such as GPUs and/or MICs).

Our experiments have proved that the proposed workload distribution scheme is effective for the given problem size and system configuration, but it is not adaptable to different problem sizes and system configurations. Thus, we will explore a dynamic workload distribution scheme which can adapt to the changes in problem size and system configuration well. Moreover, we will manage to reduce the memory requirement by redesigning the parallel two-list algorithm in future work so as to address larger-scale SSP.

Acknowledgments

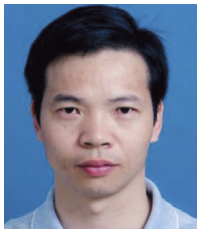
This research was partially funded by the Key Program of National Natural Science Foundation of China (Grant Nos. 61133005, 61432005), the National Natural Science Foundation of China (Grant Nos. 61370095, 61472124, 61472126, 61572175), and the International Science & Technology Cooperation Program of China (Grant No. 2015DFA11240).

References

[1] S.G. Akl, N. Santoro, Optimal parallel merging and sorting without memory conflicts, IEEE Trans. Comput. 100 (11) (1987) 1367–1369.
[2] S.S. Bokhari, Parallel solution of the subset-sum problem: an empirical study, Concurr. Comput.: Pract. Exper. 24 (18) (2012) 2241–2254.
[3] V. Boyer, D. El-Baz, M. Elkihel, Dense dynamic programming on multi GPU, in: PDP, Citeseer, 2011, pp. 545–551.
[4] V. Boyer, D. El-Baz, M. Elkihel, Solving knapsack problems on GPU, Comput. Oper. Res. 39 (1) (2012) 42–47.

[5] I. Chakroun, N. Melab, M. Mezmaz, D. Tuyttens, Combining multi-core and GPU computing for solving combinatorial optimization problems, J. Parallel Distrib. Comput. 73 (12) (2013) 1563–1577.
[6] F.B. Chedid, An optimal parallelization of the two-list algorithm of cost $O(2^{n/2})$, Parallel Comput. 34 (1) (2008) 63–65.
[7] R.-B. Chen, Y.M. Tsai, W. Wang, Adaptive block size for dense QR factorization in hybrid CPU–GPU systems via statistical modeling, Parallel Comput. 40 (5) (2014) 70–85.
[8] E. Horowitz, S. Sahni, Computing partitions with applications to the knapsack problem, J. ACM 21 (2) (1974) 277–292.
[9] Q. Huang, Z. Huang, P. Werstein, M. Purvis, GPU as a general purpose computing resource, in: Ninth International Conference on Parallel and Distributed Computing, Applications and Technologies, 2008, PDCAT 2008, IEEE, 2008, pp. 151–158.
[10] J. Jaros, Multi-GPU island-based genetic algorithm for solving the knapsack problem, in: 2012 IEEE Congress on Evolutionary Computation, (CEC), IEEE, 2012, pp. 1–8.
[11] L. Kang, L. Wan, K. Li, Efficient parallelization of a two-list algorithm for the subset-sum problem on a hybrid CPU/GPU cluster, in: 2014 Sixth International Symposium on Parallel Architectures, Algorithms and Programming, (PAAP), IEEE, 2014, pp. 93–98.
[12] E.D. Karnin, A parallel algorithm for the knapsack problem, IEEE Trans. Comput. 100 (5) (1984) 404–408.
[13] A.J. Kleywegt, J.D. Papastavrou, The dynamic and stochastic knapsack problem with random sized items, Oper. Res. 49 (1) (2001) 26–41.
[14] M.E. Lalami, D. El-Baz, GPU implementation of the Branch and Bound method for knapsack problems, in: Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2012 IEEE 26th International, IEEE, Shanghai, China, 2012, pp. 1769–1777.
[15] K. Li, R. Li, Q. Li, Optimal parallel algorithms for the knapsack problem without memory conflicts, J. Comput. Sci. Tech. 19 (6) (2004) 760–768.
[16] K. Li, J. Liu, L. Wan, S. Yin, K. Li, A cost-optimal parallel algorithm for the 0-1 knapsack problem and its performance on multicore CPU and GPU implementations, Parallel Comput. 43 (2015) 27–42.
[17] A. Lopez-Ortiz, A. Salinger, R. Suderman, Toward a generic hybrid CPU–GPU parallelization of divide-and-conquer algorithms, in: Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013 IEEE 27th International, IEEE, 2013, pp. 601–610.
[18] D.-C. Lou, C.-C. Chang, A parallel two-list algorithm for the knapsack problem, Parallel Comput. 22 (14) (1997) 1985–1996.
[19] S. Martello, P. Toth, Knapsack Problems: Algorithms and Computer Implementations, John Wiley & Sons, Inc., 1990.
[20] P. Pospíchal, J. Schwarz, J. Jaros, Parallel genetic algorithm solving 0/1 knapsack problem running on the GPU, in: 16th International Conference on Soft Computing MENDEL, Brno University of Technology, Brno, Czech Republic, 2010, pp. 64–70.
[21] 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, European J. Oper. Res. 176 (2) (2007) 870–879.
[22] F. Song, S. Tomov, J. Dongarra, Enabling and scaling matrix computations on heterogeneous multi-core and multi-GPU systems, in: Proceedings of the 26th ACM International Conference on Supercomputing, ACM, 2012, pp. 365–376.
[23] S. Tomov, J. Dongarra, M. Baboulin, Towards dense linear algebra for hybrid GPU accelerated manycore systems, Parallel Comput. 36 (5) (2010) 232–240.
[24] L. Wan, K. Li, J. Liu, K. Li, A novel CPU–GPU cooperative implementation of a parallel two-list algorithm for the subset-sum problem, in: Proceedings of Programming Models and Applications on Multicores and Manycores, ACM, 2014, pp. 70–79.
[25] L. Wan, K. Li, J. Liu, K. Li, GPU implementation of a parallel two-list algorithm for the subset-sum problem, Concurr. Comput.: Pract. Exper. 27 (1) (2015) 119–145.
[26] F. Wang, C. Yang, Y. Du, J. Chen, H. Yi, W. Xu, Optimizing Linpack benchmark on GPU-accelerated petascale supercomputer, J. Comput. Sci. Tech. 26 (5) (2011) 854–865.
[27] Q. Wu, C. Yang, T. Tang, L. Xiao, Exploiting hierarchy parallelism for molecular dynamics on a petascale heterogeneous system, J. Parallel Distrib. Comput. 73 (12) (2013) 1592–1604.

Lanjun Wan received his M.S. degree in Computer Science from Hunan University of Technology, in 2009. He is currently pursuing his Ph.D. degree in Computer Science at Hunan University. His research interests include massively parallel computing, CPU–GPU hybrid and cooperative computing, parallel programming, and parallel algorithms and implementations.



Kenli Li received his Ph.D. degree in Computer Science from Huazhong University of Science and Technology, in 2003. He was a visiting scholar at University of Illinois at Urbana–Champaign from 2004 to 2005. He is currently a full professor of Computer Science and Technology at Hunan University and deputy director of National Supercomputing Center in Changsha. His major research areas include parallel computing, high-performance computing, grid and cloud computing. He has published more than 100 research papers in international conferences and journals such as IEEE-TC, IEEE-TPDS, JPDC, ICPP, CCGrid. He is an outstanding member of CCF. He is a member of the IEEE and serves on the editorial board of *IEEE Transactions on Computers*.

Keqin Li is a SUNY Distinguished Professor of Computer Science. His current research interests include parallel computing and high-performance computing, distributed computing, energy-efficient computing and communication, heterogeneous computing systems, cloud computing, big data computing, multicore computing, storage and file systems, wireless communication networks, sensor networks, peer-to-peer file sharing systems, mobile computing, service computing. He has published over 400 journal articles, book chapters, and refereed conference papers, and has received several best paper awards. He is currently or has served on the editorial boards of *IEEE Transactions on Parallel and Distributed Systems, IEEE Transactions on Computers, IEEE Transactions on Cloud Computing, Journal of Parallel and Distributed Computing*. He is an IEEE Fellow.