RAIRO-Oper. Res. 58 (2024) 3347–3367
https://doi.org/10.1051/ro/2024125

RAIRO Operations Research
www.rairo-ro.org

PARALLEL IMPLEMENTATION OF AN EXACT TWO-PHASE METHOD FOR THE BIOBJECTIVE KNAPSACK PROBLEM

Khadidja Chaabane[1,*], Sadek Bouroubi[1] and Younes Djellouli[2]

Abstract. This paper introduces a parallel implementation of an exact two-phase method for solving the bi-objective knapsack problem on a CPU-GPU system. We utilize the Branch-and-Bound procedure in both phases, along with a highly efficient reduction technique to generate all efficient solutions. However, in the first phase, we focus on identifying all supported extreme efficient solutions, followed by reducing the dimension of the problem using an object efficiency reduction algorithm. The second phase is responsible for generating all unsupported efficient solutions. We develop a combined algorithm incorporating both phases, which is implemented in the CUDA language. Our study investigates the impact of parallel computing performance on various numerical instances compared to other exact methods in the literature. Additionally, we confirm the effectiveness of our proposed parallel-solving method by testing uncorrelated instances.

Mathematics Subject Classification. 90C05, 90C25, 90C29, 65K05.

Received April 8, 2023. Accepted June 8, 2024.

1. Introduction

Over the past few decades, there has been significant interest in multi-objective integer linear programming problems (MOILP), and a variety of techniques have been developed to address challenges in real-world applications. One specific subset of these problems is multi-objective combinatorial optimization (MOCO), which involves a discrete feasible set with multiple integer-valued objectives.

This paper focuses on the bi-objective knapsack problem (BOKP), a challenging multi-objective combinatorial optimization problem known to be NP-hard [9,10]. Various exact and approximate methods have been proposed to obtain high-quality solutions within reasonable time frames (*e.g.*, [2, 6, 7, 12, 13, 18, 27, 29]). However, these approaches have limitations, such as exponential growth in the solution space with increasing instance sizes and constraints related to limited execution time and memory capacity.

In the literature, both exact and approximate methods have been proposed to address the multi-objective knapsack problem. Notably, three exact methods have been developed, including a two-phase Branch and Bound (B&B) algorithm for the bi-objective case [26], which adopts a single-criterion approach similar to

Keywords. Multi-objective optimization, efficient solution, combinatorial optimization, GPU computing, CUDA.

[1] LIFORCE Laboratory, DGRSDT, USTHB, Faculty of Mathematics, Department of Operations Research, B.P. 32 El-alia, Bab Ezzouar 16111, Algiers, Algeria.

[2] AMCD-RO Laboratory, USTHB, Faculty of Mathematics, Department of Operations Research, B.P. 32 El-alia, 16111 Bab Ezzouar, Algiers, Algeria.

*Corresponding author: khadidjachaabane03@gmail.com

Martello and Toth's [22], evaluated on small and medium-sized instances. Visée *et al.* [29] presented a two-phase method, determining supported efficient solutions in the first phase and employing two versions of Branch and Bound in the second phase ("breadth-first" and "depth-first"). Extensive numerical experiments, involving up to 500 objects, compared the performance of both approaches. However, these algorithms are primarily effective for bi-objective problems and were outperformed by Bazgan *et al.* [3], who introduced an efficient dynamic programming algorithm for problems with up to three objectives. Nonetheless, similar challenges to its single-objective counterpart were observed, such as significant memory usage and limitations with larger instances. In 2010, Jorge *et al.* [18] presented novel techniques for solving the unidimensional multi-criteria knapsack problem with binary variables. Their approaches were specifically designed to handle both multi-objective cases and generalizations from single to multi-objective scenarios. Moreover, they introduced a preprocessing technique to reduce the feasible region before initiating the resolution process. Subsequently, Daoud-Chaabane [8] further enhanced the efficiency of the reduction algorithm and empirically demonstrated its superiority. In this paper, we provide a comprehensive theoretical proof of this superiority (see Proposition 2.1).

In the domain of approximate methods, E.L. Ulungu *et al.* [28] introduced multi-objective simulated annealing method (MOSA), an approach that approximates efficient solutions for multi-objective combinatorial optimization problems. Their research showcased different implementation options and included a comparison of results with exact methods on bi-objective knapsack problems. Furthermore, various other publications present a range of approximate techniques for solving multi-objective knapsack problems (*e.g.*, [13–15]).

To overcome the challenges posed by these methods, researchers are increasingly turning to high-performance computing (HPC) and utilizing parallel architectures such as multicore processors and graphics processing units (GPU).

General-purpose graphics processors (GPGPUs) have become a popular choice for programming, and many of the world's fastest supercomputers, such as PizDant, Titan, and Tianhe1A, incorporate NVIDIA GPUs. CUDA, introduced by NVIDIA in November 2006, offers a parallel computing platform and programming method capable of handling complex computing problems faster than CPUs. In light of this technological advancement, our focus is on proposing a solution using exact methods in parallel programming mode. While there are a few articles on single-objective exact methods with parallel linear programming (referenced in [5,11,19,20,25]). There is a lack of research in the field of multiple objective problems solved with exact methods of parallel linear programming, which we propose to address in this article. Recently, William Pettersson and Melih Ozlen have developed three new exact algorithms to solve integer linear programming problems with at least three objectives, as indicated in reference [24].

This article presents two main contributions. Firstly, we provide a theoretical proof that our data preprocessing algorithm, as outlined in [8], consistently yields superior results compared to those obtained by Jorge, J. in [18]. Secondly, we introduce a parallel implementation of the two-phase method, utilizing the Branch and Bound algorithm and the preprocessing procedure from [8], to solve the Bi-Objective Knapsack Problem. Our experiments, conducted on a CPU-GPU system, include a comprehensive comparison of our results with those of other exact methods in sequential implementation. The performance of the proposed algorithm is clearly illustrated through the graphs shown in Figure 9.

The paper is organized as follows: Section 2 provides the BOKP formulation and defines the notations used in this paper. It also describes the two-phase method with detailed reduction approaches. In Section 3, we present the parallel implementation of the two-phase approach using CUDA on a CPU-GPU system. Section 4 includes a discussion of our experimental results. Finally, we conclude the paper with some interesting remarks.

2. Preliminaries

This section presents fundamental concepts and outcomes in multi-objective combinatorial optimization, with a specific emphasis on the bi-objective knapsack problem.

2.1. Multi-Objective Combinatorial Optimization Problem

A multiple objective combinatorial optimization problem can be represented in the following way:

$$(\text{MOCO}) \begin{cases} \text{“max”} & z(x) = (z_1(x), \ldots, z_p(x)) \\ \text{s.t.} & x \in X \end{cases} \tag{1}$$

where X is a set of feasible solutions defined implicitly by a set of constraints. The set of feasible points in the objective space is denoted by $Z = z(X)$.

An admissible solution $x^\star$ to problem (1) is efficient (or Pareto optimal) if there is no other feasible solution x that satisfies $z_k(x) \geq z_k(x^\star)$ for all $k = 1, \ldots, p$ with at least one strict inequality. The point $z(x^\star)$ corresponding to this solution is known as a non-dominated point. The set of efficient solutions and the set of non-dominated points are denoted by $X_E \subseteq X$ and $Z_{ND} \subseteq Z$, respectively.

A feasible solution $x^* \in X$ is said to be weakly efficient if there is no other feasible solution $x \in X$ such that $z_k(x) > z_k(x^*)$ for all $k = 1, \ldots, p$. In this case, $z_k(x)$ is said to be weakly nondominated.

2.2. The 0–1 Bi-Objective Knapsack Problem

The formulation of the Bi-Objective Knapsack Problem (BOKP) is as follows: Given n items, a selection must be made while respecting a capacity constraint W. Each item j has a positive integer weight w_j and two non-negative integer profits c_j^k $(k = 1, 2)$. A feasible solution is represented by a binary decision vector $X = (x_1, \ldots, x_n)$, where $x_j = 1$ if item j is included in the knapsack, and 0 otherwise. The solution must satisfy the weight constraint $\sum_{j=1}^{n} w_j x_j \leq W$. The goal is to find the set of non-dominated criterion vectors. Formally, the BOKP is defined as follows:

$$(\text{BOKP}) \begin{cases} \text{“max”} z_k = \sum_{j=1}^{n} c_j^k \; x_j; \; k = 1, 2 \\ s.t. \quad \sum_{j=1}^{n} w_j x_j \leq W \\ \qquad x_j \in \{0, 1\}; \quad \forall j = 1, \ldots, n \end{cases} \tag{2}$$

Given that the feasible region is $\left\{ (x_i)_{i \in \{1,2,\ldots,n\}} \in \{0,1\}^n \mid \sum_{j=1}^{n} w_j x_j \leq W \right\}$, which is a non-convex set, it becomes necessary to distinguish two types of efficient solutions, namely [4]:

– The set of supported efficient solutions X_{SE} consists of optimal solutions of a weighted sum single-objective problem
(P_λ):

$$(P_\lambda) \begin{cases} \max Z_\lambda = \lambda z_1(x) + (1 - \lambda) z_2(x) \\ s.t. \quad \sum_{j=1}^{n} w_j x_j \leq W \\ \qquad x_j \in \{0, 1\}; \quad \forall j = 1, \ldots, n \\ \qquad 0 < \lambda < 1 \end{cases} \tag{3}$$

The image of this set is called the set of **non-dominated supported points**, denoted Z_{NDS}. We denote by $conv(Z)$ the **convex hull** of Z. A non-dominated point that is located on a vertex of $conv(Z)$ is called a **non-dominated extreme point**. Otherwise, it is a **non-dominated non-extreme point**.

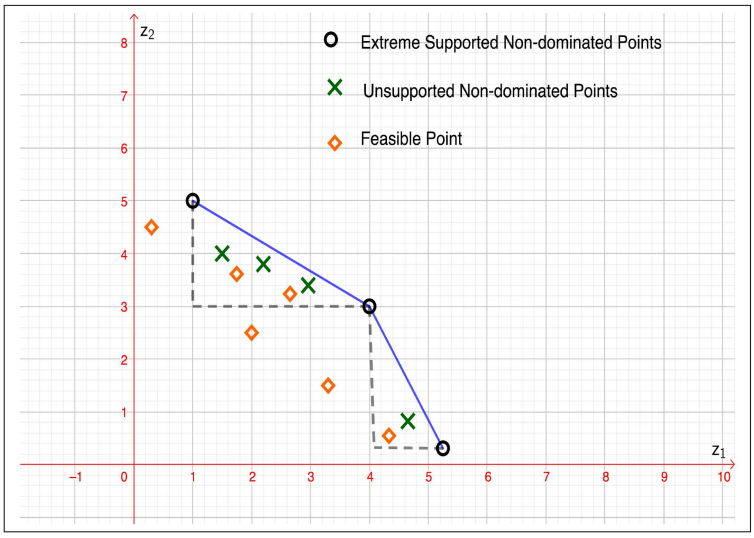

FIGURE 1. Supported and Unsupported non-dominated points.

– The set of **unsupported efficient solutions**, $X_{UE} = X_E \setminus X_{SE}$, which cannot be found by optimization of problem (P_λ) and located in the triangles formed by the non dominated supported solutions $\Delta Z_i Z_{i+1}$ are called potential areas for efficient solutions.
We denote by X_{SEe} and X_{SEne} the set of extreme supported efficient solutions and the set of non-extreme supported efficient solutions, respectively. Their images in the objective space are denoted by Z_{NDSe} and Z_{NDSne}, respectively.

2.2.1. Pretreatment of Data

The process of data preparation plays a pivotal role in refining the scope of problems, leading to significant reductions in the time required for their resolution. Previous research, as highlighted in [18] and [8], underscores the critical importance of data preprocessing in this context. Furthermore, we provide theoretical validation supporting the superiority of the approach proposed by M. Daoud and D. Chaabane over that of J. Jorge, a distinction previously evidenced empirically. This subsection introduces essential notations, revisits key definitions, and outlines pertinent findings to establish the foundation for our analysis.

Glover [17] introduced lower bound (LB) and upper bound (UB) constraints for the one-dimensional Knapsack Problem, defining feasible solution sizes. These bounds were further elaborated by Xavier Gandibleux and Arnaud Fréville to include multi-objective considerations [15], revealing a nuanced relationship between these constraints as illustrated below (6):

$$LB = \max\left\{ s : \sum_{i=1}^{s} w_i \leq W \right\}, \quad \text{given } w_i \geq w_{i+1}, \forall i \in \{1, \ldots, n-1\}, \tag{4}$$

$$UB = \max\left\{ s : \sum_{i=1}^{s} w_i \leq W \right\}, \quad \text{given } w_i \leq w_{i+1}, \forall i \in \{1, \ldots, n-1\}, \tag{5}$$

$$\text{If } x \in X_E, \text{ then } LB \leq \sum_{i=1}^{n} x_i \leq UB. \tag{6}$$

This framework is instrumental in tackling multi-objective challenges, setting a precedent for evaluating solution efficiency within defined problem spaces.

Drawing upon the notations from [18], let X_E denote the exhaustive set of efficient solutions for the BOKP. The vectors defining the problem space are outlined as follows:

$$V = \left\{ v^i \in \mathbb{N}^p \times \mathbb{Z} | v^i = (c_i^1, \ldots, c_i^p, -w_i), i \in \{1, \ldots, n\} \right\}. \tag{7}$$

The dominant (preferred) set, $P(v^i)$, consists of indices j for vectors v^j that are preferred over v^i.

Similarly, indices for vectors v^j that are dominated by v^i compile the dominated set. The indices categorizing regular variables are as follows:

$$C_0 = \{i \in \{1, \ldots, n\} | x_i = 0, \forall x \in X_E\}, \tag{8}$$

$$C_1 = \{i \in \{1, \ldots, n\} | x_i = 1, \forall x \in X_E\}. \tag{9}$$

Incorporating notations from [8], the efficiency vectors are defined as:

$$E = \left\{ E^i \in \mathbb{R}^p \,|\, E^i = (e_i^1, e_i^2, \ldots, e_i^p)', e_i^k = \frac{c_i^k}{w_i} \right\}, \tag{10}$$

where $i \in \{1, 2, \ldots, n\}$ and $k \in \{1, 2, \ldots, p\}$. The sets D_0 and D_1 represent indices for regular variables.

Proposition 2.1. *The regular variables identified by the algorithm stated in [18] are a subset of those identified by the algorithm proposed in [8].*

Proof. The conditions required for a variable to be regular are provided in [18] and [8]. We demonstrate that $C_0 \subset D_0$ and $C_1 \subset D_1$.

Let i be in C_0 then by construction, one of the following properties is met :

$\omega_i + \sum_{j \in P(v^i)} \omega_j > W$ or $\left|P(v^i)\right| \geq UB$. In both cases $i \in D_0$.

Given that $\omega_i + \sum_{j \in P(v^i)} \omega_j > W$, it follows that x_i must be equal to 0. Therefore, we can conclude that i belongs to the set D_0.

On the other hand, if $\left|P(v^i)\right| \geq UB$, we demonstrate that $P(e^i)$ contains $P(v^i)$.

In fact, when j belongs to $P(v^i)$, we have $c_k^j \geq c_k^i$ and $-\omega_j \geq -\omega_i$. This implies that for any $k \in \{1, \ldots, n\}$, $\frac{c_k^j}{\omega_j} \geq \frac{c_k^i}{\omega_i}$, which in turn implies $e^j \geq e^i$, *i.e.*, $j \in P(e^i)$. Thus, we can conclude that $P(v^i) \subset P(e^i)$. Furthermore, since $UB \leq \left|P(v^i)\right| \leq \left|P(e^i)\right|$, it follows that i belongs to D_0, and therefore D_0 contains C_0. Similarly, we can demonstrate that D_1 contains C_1.

If i belongs to C_1, then either $\sum_{j \notin D(v^i)} \omega_j \leq W$ or $\left|D(v^i)\right| \geq n - LB$.

If $\sum_{j \notin D(v^i)} \omega_j \leq W$ then $x_i = 1, \forall x \in X_E$. Consequently, $i \in D_1$.

Alternatively, if $\left|D(v^i)\right| \geq n - LB$ and if $j \in D(v^i)$ then $\forall k \in \{1, \ldots, n\}$, $c_k^i \geq c_k^j$ and $-\omega_i \geq -\omega_j$. As a result, $\frac{c_k^i}{\omega_i} \geq \frac{c_k^j}{\omega_j}$, which means $e^i \geq e^j$, so $j \in D(e^i)$. We conclude then $D(v^i) \subset D(e^i)$.

We have $\left|D(v^i)\right| \leq \left|D(e^i)\right|$, $n - LB \leq \left|D(v^i)\right| \leq \left|D(e^i)\right|$ which means $n - LB \leq \left|D(e^i)\right|$, then $i \in D_1$. Hence, C_1 is contained in D_1.

□

To our knowledge, there exist three exact methods for addressing the 0–1 multi-objective knapsack problem: the two-phase approach, the dynamic programming method, and the Branch-and-Bound procedure. In our study, we construct Z_N by utilizing a two-phase technique that relies on the Branch-and-Bound algorithm.

2.3. The two-phase method

The two-phase methodology, introduced by [27], provides a framework for addressing bi-objective MOCO (Multi-Objective Combinatorial Optimization) problems. This approach, as the name suggests, aims to find efficient solutions in two stages: during the first stage, it employs the dichotomic method outlined in 2.3.1 of [1] to generate both the sets X_{SEe} and X_{SEne}, and in the second stage, we construct the set of unsupported efficient solutions, denoted as X_{UE}, using the information obtained from these sets in the previous phase.

2.3.1. Phase 1: Determination of Z_{NDSe}

At this phase, we commence the identification of the set of efficient solutions by obtaining two optimal solutions denoted as x^1 and x^2. These solutions are derived through the resolution of the lexicographic optimization problems $\underset{x\in X}{\text{lexmax}}\{z_1(x), z_2(x)\}$ and $\underset{x\in X}{\text{lexmax}}\{z_2(x), z_1(x)\}$, respectively. The resolution of these lexicographic optimization problems involves the search for the most favorable solutions that conform to the lexicographic order of the objective functions.

For $\underset{x\in X}{\text{lexmax}}\{z_1(x), z_2(x)\}$, the main goal is to identify the set of solutions where $z_1(x)$ is maximized while simultaneously keeping $z_2(x)$ as high as possible. Essentially, we give priority to optimizing $z_1(x)$ first, and among the solutions with the same maximum value of $z_1(x)$, we select the one with the highest value of $z_2(x)$.

Similarly, for $\underset{x\in X}{\text{lexmax}}\{z_2(x), z_1(x)\}$, the objective is to find the set of solutions where $z_2(x)$ is maximized while maintaining $z_1(x)$ at its highest possible level. In this case, we prioritize the optimization of $z_2(x)$ first, and among the solutions with the same maximum value of $z_2(x)$, we choose the one with the highest value of $z_1(x)$.

Through solving these lexicographic optimization problems, we obtain the optimal solutions x^1 and x^2, which serve as the foundation for identifying the set of efficient solutions in the objective space.

These two solutions x^1 and x^2 define the exploration region in which we aim to identify a set of non-dominated points within the objective space. Consider two adjacent non-dominated points, $z^r = z(x^r) = (z_1^r, z_2^r)$, and $z^s = z(x^s) = (z_1^s, z_2^s)$, belonging to Z_{NDSe}, where $z_1^r < z_1^s$ and $z_2^r > z_2^s$. We introduce weights $\lambda_1 = z_1^s - z_1^r$ and $\lambda_2 = z_2^r - z_2^s$ and address the optimization problem (P_λ). Let $\{x^\ell, \ell \in L\}$ be the optimal solutions of (P_λ) and $\{z(x^\ell), \ell \in L\}$ their respective images in the objective space. We denote the line segment joining the points z^r and z^s by $[z^r, z^s]$. Two scenarios are considered:

- If the set $z(x^\ell), \ell \in L$ has no intersection with the line segment $[z^r, z^s]$, then the optimal solutions $x^\ell, \ell \in L$ are considered as new supported efficient extreme solutions and their images are added to Z_{NDSe}, *i.e.*, $Z_{NDSe} \leftarrow Z_{NDSe} \cup z(x^\ell), \ell \in L$.
- If $z(x^\ell), \ell \in L$ is a subset of the line segment $[z^r, z^s]$, then the optimal solutions $\{x^\ell, \ell \in L\}\backslash\{x^r, x^s\}$ are non-extreme supported efficient solutions and are added to X_{SEne}. Their corresponding images in the objective space are also appended to Z_{NDSne}.

The process described above is applied to every pair of neighboring points in Z_{SEe} until either both lexicographic optimization problems, $\underset{x\in X}{lexmax}\{z_1(x), z_2(x\}$ and $\underset{x\in X}{lexmax}\{z_2(x), z_1(x)\}$, are infeasible or their solutions coincide. The technical description of this phase is provided by Algorithm 1.

2.3.2. Phase 2: Determination of X_{UE}

The objective of the second phase is to identify the set of additional efficient solutions X_{UE} not determined in phase I. To do this, a search region for those solutions is determined using the supported points already recorded in the first phase. In our case, the search area is given by the set of triangles, $\Delta(z^r, z^s)$, whose hypotenuses are defined by adjacent supported points with $z_1^r < z_1^s$ and their corresponding local nadir point (z_1^s, z_2^r) (see Fig. 1). Each triangle in the second phase has a single-objective problem (P_λ) associated with it, where $(\lambda_1, \lambda_2) = (z_2^r - z_2^s, z_1^s - z_1^r)$ denotes a search direction orthogonal to the hypotenuse. Then, for the single-objective Knapsack problem, these issues are resolved using Branch and Bound (B&B) of S. Martello and P. Toth (1988) [21], with the evaluation component modified to investigate each triangle. The different strategic stages of a branch-and-bound technique are detailed as follows:

Algorithm 1: PHASE I: Determination of the supported efficient solutions (Dichotomic method)

Input Data: A bi-objective combinatorial optimization problem instance;
Output: The sets X_{SEe}, X_{SEne} of this instance;
begin
  $X_{SEe} \leftarrow \emptyset$,$X_{SEne} \leftarrow \emptyset$;
  $L \leftarrow \emptyset$;
  $x^1 \leftarrow \text{lexmax}_{x\in\mathcal{X}}\{z_1(x), z_2(x)\}$;
  $x^2 \leftarrow \text{lexmax}_{x\in\mathcal{X}}\{z_2(x), z_1(x)\}$;
  if $z(x^1) = z(x^2)$ **then**
    // Stop (Only one non-dominated point is found);
    $X_{SEe} \leftarrow \{x^1\}$;
  else
    $addToL(z(x^1), z(x^2))$;
    $X_{SEe} \leftarrow \{x^1, x^2\}$;
    while $L \neq \emptyset$ **do**
      $(z^r, z^s) \leftarrow popFromL()$;
      $\lambda_1 = z_2^r - z_2^s$;
      $\lambda_2 = z_1^s - z_1^r$;
      $x^* \leftarrow solveOptimalP(\downarrow \lambda_1, \downarrow \lambda_2)$ and value $z^* = (z_1(x^*),\ z_2(x^*))$;
      if $\frac{z_2^r - z_2^\lambda(x^*)}{z_1^r - z_1^\lambda(x^*)} \neq \frac{z_2^s - z_2^\lambda(x^*)}{z_1^s - z_1^\lambda(x^*)}$ **then**
        $X_{SEe} \leftarrow X_{SEe} \cup \{x^*\}$;
        $addToL(\downarrow z^r, \downarrow z(x^*))$;
        $addToL(\downarrow z(x^*), \downarrow z^s)$;
      else
        $X_{SEne} \leftarrow X_{SEne} \cup \{x^*\}$;
  return $X_{SE} = X_{SEe} \cup X_{SEne}$

Separation procedure

In this procedure, the problem is divided into a set of sub-problems such that the union of their feasible spaces forms the feasible space of the original problem. This technique is inspired by the branching method proposed by [29]. The number of sub-problems generated corresponds to the number of variables set to 1 in the solution obtained after evaluating the sub-problems. Denote q as the total number of sub-problems generated, $S_1, S_2, \ldots, S_q$, and let $B = \{j \mid x_j = 1\} = \{p_1, p_2, \ldots, p_q\}$ represent the set of variable indices fixed at 1. The formation of sub-problems is detailed as follows: The first sub-problem, S_1, involves setting $x_{p_1} = 0$, keeping other variables free. Each subsequent sub-problem fixes one additional variable to 0, while setting previously considered variables to 1, as illustrated:

$$\begin{cases} S_1 : x_{p_1} = 0 \\ S_2 : x_{p_2} = 0\ ,\ x_{p_1} = 1 \\ \quad\vdots \quad\vdots \qquad\quad \vdots \\ S_q : x_{p_q} = 0\ ,\ x_{p_1} = \cdots = x_{p_{q-1}} = 1 \end{cases} \tag{11}$$

Lower bound set

In the bi-objective context, the region to be explored in a triangle is delimited by the three bounds ($\underline{z_1}$, $\underline{z_2}$, $\underline{z_\lambda}$), as described in Figure 2 below.

If z_r and z_s are the two points describing the triangle, so $\underline{z_1} = z_1^r$ is a good lower bound with respect to the first objective for non supported solutions whose images are included in the triangle. The same applies to $\underline{z_2} = z_2^s$ about the second objective. Finally, a lower bound to the problem (P_λ) is given by $\underline{z_\lambda} = \lambda.z_n$, where z_n is the local nadir point defined by $z_n = (z_1^r, z_2^s)$. This last bound is updated as new potentially efficient solutions

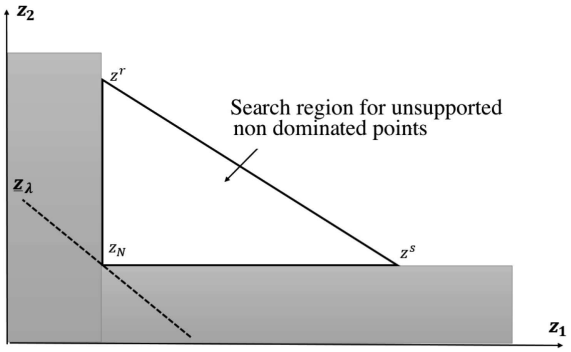

FIGURE 2. Bounds when exploring a triangle.

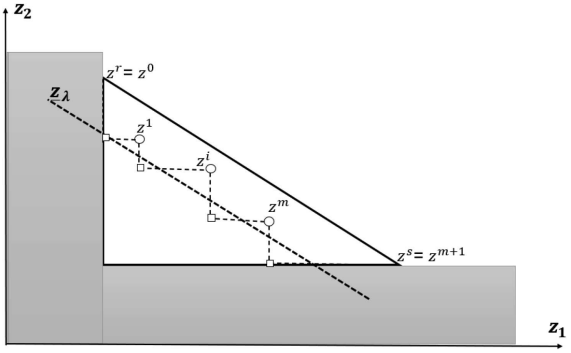

FIGURE 3. The new lower bound $\underline{z_\lambda}$.

are found in the triangle (Fig. 3). Let $z^1, \ldots, z^m$ be the known potentially non-dominated feasible points in the triangle, such as $z_1^i < z_1^{i+1}$, $\forall i \in \{1, \ldots, m-1\}$. By renaming $z^0 = z^r$, $z^{m+1} = z^s$, the value of this lower bound is

$$\underline{z_\lambda} = min\Big(\min_{i\in\{0,\ldots,m+1\}} \lambda.z^i, \min_{i\in\{0,\ldots,m\}} \lambda_1.(z_1^i + 1) + \lambda_2.(z_2^{i+1} + 1)\Big). \tag{12}$$

This value, represented by the dashed line in Figure 3, is the lowest of the values that the hypothetical efficient solutions remaining to be discovered in the triangle can have.

Evaluation of a node

The evaluation of nodes during the Branch and Bound (B&B) process involves calculating an upper bound for each of the objectives (z_1, z_2, P_λ). Let $\overline{z}_1$, $\overline{z}_2$, and $\overline{z}_\lambda$ be these bounds, for example, the bounds proposed by S. Martello and P. Toth (see [22]).

If one of these upper bounds is less than or equal to its corresponding lower bound (*i.e.*, $\overline{z}_1 \leq \underline{z}_1$, $\overline{z}_2 \leq \underline{z}_2$, or $\overline{z}_\lambda \leq \underline{z}_\lambda$), then the node is considered fathomed and is not further explored.

A typical implementation of Phase II in the B&B algorithm is detailed in Algorithm 2. It contains the following procedures:

During the process of branch and bound (B & B), nodes are evaluated by computing upper bounds for each objective (z_1, z_2, P_λ), denoted as $\overline{z}_1$, $\overline{z}_2$, and $\overline{z}_\lambda$, respectively. The bounds used can vary, but examples include the Martello and Toth bounds. If any of the computed upper bounds are less than or equal to the corresponding lower bound ($\underline{z}_1$, $\underline{z}_2$, $\underline{z}_\lambda$), the node is pruned (fathomed) as it cannot lead to a better solution. Algorithm 2

outlines a typical implementation of phase II in the branch and bound process, which includes procedures for branching and updating the lower bounds, as well as other relevant steps for the algorithm.

- CreateS0(x^r, x^s): Builds root node S_0 (*i.e.*, the global problem to solve);
- *AddToL*($\downarrow S0$): adds the node S_0 to the set of nodes L;
- *UpperBounds*($\downarrow S$): calculates the upper bounds of the objectives z_1, z_2, P_λ associated to the node S by using the Martello and Toth bounds;
- *solveOptimalP*($\downarrow S$): returns an optimal solution of the node S.

Algorithm 2: PHASE II: Branch and Bound in triangle

Input Data: A bi-objective combinatorial optimization problem instance;
$\{x^r, x^s\}$: Two adjacent supported efficient extreme solutions;
$\{z^r, z^s\}$: Two adjacent supported non-dominated extreme points;
Output: $\bar{\mathcal{X}}_E$: Set of efficient solutions in triangle $\Delta(z^r, z^s)$;
Begin
  $\bar{\mathcal{X}}_E \leftarrow \{x^r, x^s\}$, $L \leftarrow \emptyset$;
  $\lambda_1 = z_2^r - z_2^s$, $\lambda_2 = z_1^s - z_1^r$;
  Calculus of the lower bounds
  $\underline{z}_1 \leftarrow z_1^r$, $\underline{z}_2 \leftarrow z_2^s$, $\underline{z}_\lambda \leftarrow \lambda_1 z_1^r + \lambda_2 z_2^s$
  Create the root S0
  CreateS0(x^r, x^s);
  AddToL($\downarrow S0$);

  while *L is not empty* **do**
    $S \leftarrow PopFromL()$;
    Calculus of the upper bounds
    $(\overline{z}_1, \overline{z}_2, \overline{z}_\lambda) \leftarrow UpperBounds(\downarrow S)$;
    if $\overline{z}_1 \leq \underline{z}_1$ *or* $\overline{z}_2 \leq \underline{z}_2$ *or* $\overline{z}_\lambda \leq \underline{z}_\lambda$ **then**
      Fathom the node S
    else
      $x^* \leftarrow solveOptimalP(\downarrow S)$
      Update $\bar{\mathcal{X}}_E$
      $\bar{\mathcal{X}}_E \leftarrow \bar{\mathcal{X}}_E \cup \{x^*\}$;
      Update $\underline{z}_\lambda$ by solving the equation (12)
      application of the separation procedure;
      create q subnodes $S_1, \cdots, S_q$ which are characterized by one to q fixed variables in the manner illustrated in subsection;
      for *i=1 to q* **do**
        AddToL(S_i);
  return $\bar{\mathcal{X}}_E$

The execution of a method in two phases could lead to memory consumption and difficulties when dealing with larger instances, especially in the second phase. As such a solution, implementing the method *via* parallel programming on multicore machines, grids, or GPU architecture is considered to be a practical option. The next section details our second contribution, the methodology employed for the parallel implementation of the method on a GPU.

3. Implementation on CPU-GPU system

In this section, we present our latest contribution which involves implementing the two-phase method on a CPU-GPU system. Our paper proposes several enhancements, which are discussed in Section 4, resulting in a

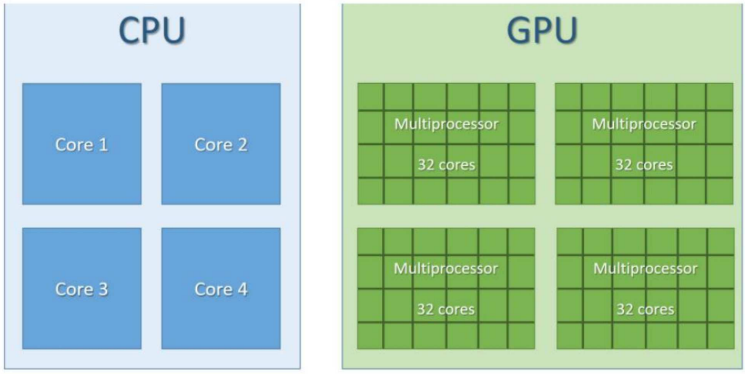

FIGURE 4. CPU and GPU (NVIDIA) architectures.

significant increase in speed compared to vOptSolver's results [16]. Firstly, we offer a brief introduction to GPU architecture.

3.1. GPU architecture and CUDA

Scientific computing widely recognizes General-Purpose Graphics Processing Units (GPGPUs) as a well-established and appealing option. Nowadays, GPGPUs are a common feature in numerous top-performing supercomputers worldwide. In contrast to CPUs, GPUs are built with a focus on highly parallel computing, dedicating more transistors to computation at the expense of data flow management and storage.

The GPU architecture typically comprises multithreaded streaming multiprocessors (SM), each containing many streaming processor (SP) cores. These SP cores are highly multithreaded and can manage a large number of concurrent threads and their state in hardware. In addition to SP cores, each SM includes special function units (SFUs), instruction and constant caches, a multithreaded instruction unit, and shared memory. NVIDIA has released several architectures, such as Fermi, Maxwell, Tesla, Pascal, and Ampere, since 2007, each with varying hardware specifications. Figure 4 illustrates the difference between the architectures of the central processor and graphics accelerator.

CUDA (Compute Unified Device Architecture) is the programming framework most commonly used for NVIDIA GPUs, where the CPU is known as the Host and the GPU as the Device. In Figure 5, warps and blocks represent the first two layers of the thread group. A thread denotes a single computational element that runs on a scalar processor as part of a streaming multiprocessor. Each SM operates in a Single Instruction Multiple Thread (SIMT) manner, issuing one set of instructions to a warp of threads to perform operations on different data elements. The programmer can set the number of threads per block, subject to specific hardware constraints. To perform calculations, a "kernel" function is created, which is shared across all multiprocessors and threads that contain the program code. When invoking this function, the parameters of the grid must be set, which consist of flow blocks and flows inside the block. Graphic cards have several types of memory:

- **Global memory:** has the highest volume, the lowest access speed, "lifetim" and program execution;
- **Local memory:** is characterized by a small volume, low access speed, "lifetime" and the execution of 1 thread;
- **Shared memory:** is small, high speed access growth, "scope" - execution of 1 block of threads;
- **Constant memory:** has a small volume, very high access speed, "scope" - program execution;
- **Register memory:** occupies a very small amount, has the highest access rate, "visibility" is executing 1 thread, and is used by the compiler;
- **Texture memory:** is used when working with graphic information.

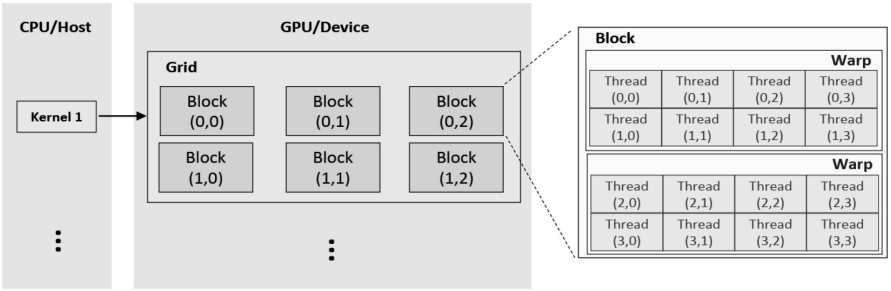

FIGURE 5. CUDA programming model.

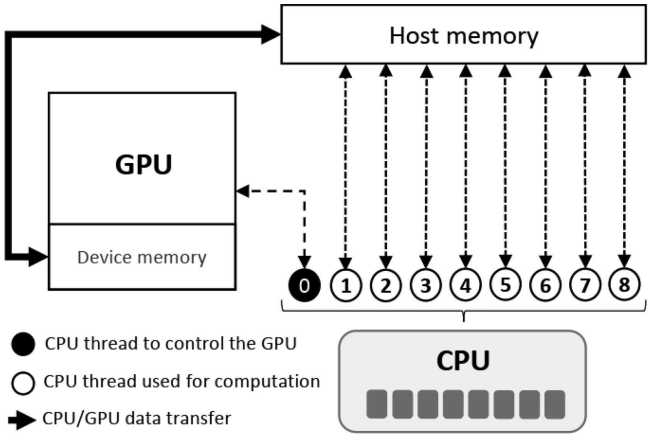

FIGURE 6. The CPU-GPU heterogeneous parallel computing model.

For more details on GPU architecture and strategies for optimizing your code on GPUs, see [23].

3.2. Parallel programming of the two phase method

In order to efficiently program CPU-GPU heterogeneous platforms, various programming models like OpenCL, OpenACC, and OpenMP 5.2 have been proposed. However, our work utilizes a different approach by employing the Sparse Matrix Vector Multiplication (SpMV). (SpMV) kernel with different sparse formats, using the open source CUDA based CUSP library for GPU, and Intel MKL library for CPU. In this model, the CPU controls the GPU and transfers data from host memory to GPU global memory before launching the computation process on the GPU by calling the kernel function. A simple heterogeneous programming model can be achieved by combining OpenMP and CUDA for CPU-GPU heterogeneous platforms. This involves dedicating one or more OpenMP threads to control the GPUs while the rest of the threads handle the computation workload on the CPU. However, it's important to note that when using this model, the data must be partitioned on different processing units before executing in the OpenMP parallel section. Algorithm 3 illustrates the programming of a computing system with one CPU and one GPU.

3.2.1. Parallel Phase I

As previously mentioned, our initial contribution in this context involved proposing a GPU-accelerated approach for the first phase.This phase is focused on calculating the set of supported efficient solutions, denoted as X_{SE}, through a dichotomy-based method that involves solving a series of single-objective Knapsack Problems

Algorithm 3: Example of CPU-GPU heterogeneous programming using OpenMP

```
1- Find optimal data partitioning();
2- Data transfer to device memory();
3- #pragma omp parallel sections
   {
   #pragma omp section
      {                    //This section is executed by one CPU thread
       lunch CUDA kernel();
  }
#pragma omp section
      {
        //This section executed by all the other CPU threads
                          lunch CUDA kernel();
       }
   }
```

(KP). However, to parallelize this phase effectively, it is necessary to have an efficient parallel algorithm that can run on a CPU-GPU system. We note that Lalami *et al.* previously presented a relevant parallel algorithm related to this topic, as referenced in [19].

The method proposed by the authors involves utilizing the Branch and Bound algorithm on the GPU to address the Knapsack Problem (KP). The approach initially solves the problem on the CPU until the number of nodes exceeds a specified threshold. However, once the number of generated nodes exceeds a certain threshold, the GPU is then employed. At this point, the list of nodes is transferred to the global memory of the GPU. In case the number of nodes generated is below the threshold, the problem is addressed in a sequential manner on the CPU. Compared to the total execution time of the Branch and Bound algorithm, the time taken for communication between the CPU and GPU is minimal. Following the communication, the separation step is executed in parallel using a Kernel that includes threads equal to the number of nodes to be separated. This creates q new child nodes. Subsequently, the bound calculation step is launched in parallel *via* a Kernel to evaluate the upper and lower bounds of the q new nodes. The algorithm that summarizes the implemented method is outlined in Pseudo-code Algorithm 4.

Algorithm 5 presents the initial implementation of the two-phase method in the CUDA environment, which involves two levels of parallelism.

In the first level (lines 11 to 18), we concurrently determine the two lexicographic optimal solutions, x_1 and x_2, corresponding to the permutations of the objective functions (1,2) and (2,1), respectively, by utilizing the *#pragma omp parallel section* directive in OpenMP. This directive enables the first processor to run the algorithm on a Streaming Multiprocessor (SM) and find x_1, while the second processor finds x_2 on another SM. To address the dependencies between supported nondominated points, the second level involves using a single processor to determine the new solution $z(x^*)$ between each pair of points (z^r, z^s) using Algorithm 4. Algorithm 5 provides technical details for the parallel Phase I method.

3.2.2. Parallel_Reduction

Our reduction algorithm parallel framework consists of three primary parallel sections named P1, P2, and P3, which operate on multiple CPUs. In P1, each CPU (i) executes the operation $\frac{c_i^j}{w_i}$. In P2, we simultaneously perform the lower bound (LB) and upper bound (UB) procedures while exchanging data. Meanwhile, we initiate the parallel section P3, where each CPU executes the indices of the regular variable sets D_0 and D_1. Figure 7 provides a visual representation of this framework.

Algorithm 4: CPU-GPU accelerated Branch and bound algorithm

```
Create the initial problem;
Insert the initial problem into the tree (List L);
Calculate the best lower bound (S_best) of the KP;
begin
    while L is not empty do
        if |L| ≥ threshold then
            Transfert the nodes of L to device memory;
            Branching kernel <<>>;
            Bounding kernel <<>>;
            Purring kernel <<>>;
            Update (S_best);
            Insert promising nodes to L;
        else
            S = next(L);
            S_1, ..., S_n = Branching(S);
            for i = 1 to n do
                Bounding(S_i);
                Update (S_best);
                Insert promising nodes to L;
    return S_best;
```

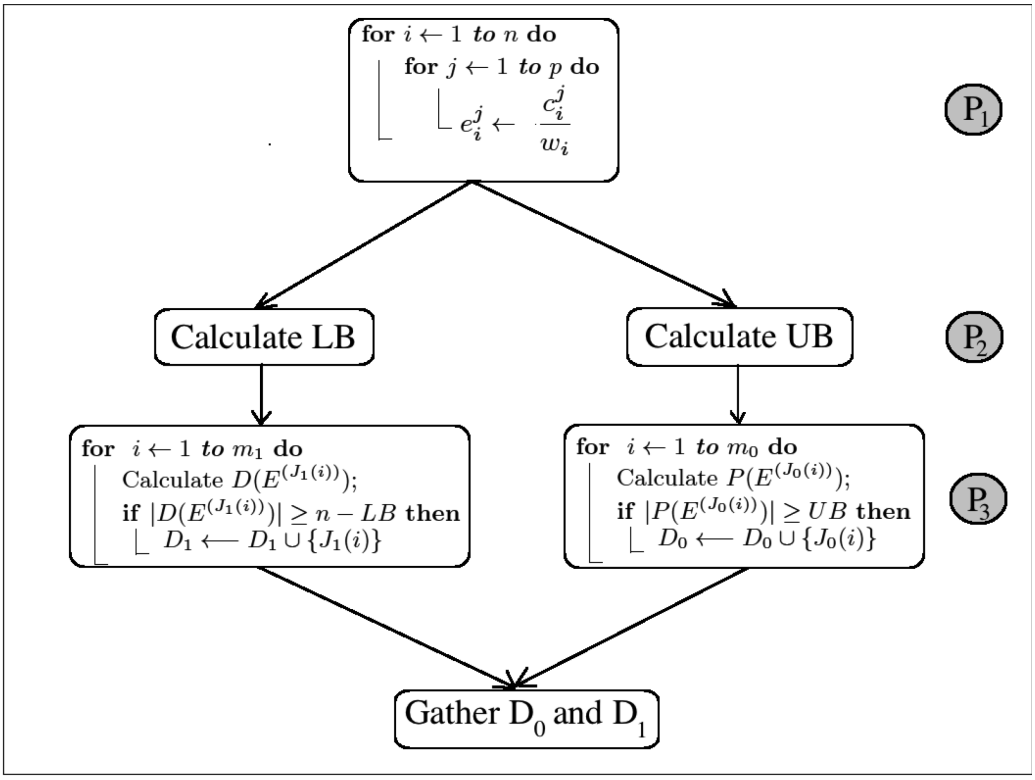

FIGURE 7. Framework of parallel Reduction algorithm.

Algorithm 5: PHASE I on CPU-GPU system

```
Input Data:
A bi-objective combinatorial optimization problem instance;
Output:
The X_SEe, X_SEne of this instance;
begin
    X_SEe ← ∅, X_SEne ← ∅;
    L ← ∅;
     #pragma omp parallel sections
        {
            #pragma omp section
            {
              x^1 ← lexmax_{x∈X}{z_1(x), z_2(x)}(Algorithm 4);
            }
            #pragma omp section
            {
              x^2 ← lexmax_{x∈X}{z_2(x), z_1(x)}(Algorithm 4);
             }
       }

    if z(x^1) = z(x^2) then
        // Stop (Only one non-dominated point is found);
        X_SEe ← {x^1};
    else
        addToL(z(x^1), z(x^2));
        X_SEe ← {x^1, x^2};
        while L ≠ ∅ do
            (z^r, z^s) ← popFromL();
            λ = (z_2^r − z_2^s, z_1^s − z_1^r);
            x* ← solveOptimalP(↓ λ_1, ↓ λ_2)(Algorithm 4);
            z(x*) = (z_1(x*), z_2(x*));
            if λz(x*) > λz(x^r) then
                X_SEe ← X_SEe ∪ {x*};
                addToL(↓ z^r, ↓ z(x*));
                addToL(↓ z(x*), ↓ z^s);
            else
                X_SEne ← X_SEne ∪ {x*};
    return X_SE = X_SEe ∪ X_SEne;
```

3.2.3. Parallel phase II

This subsection introduces a parallel implementation of algorithm 2 on a system that incorporates both CPUs and GPUs. To develop a parallel algorithm for phase two, it is necessary to leverage all parallelization criteria, including researching non-dominated solutions, distributing workload among CPUs and GPUs, and optimizing memory management. These stages form the different components of the algorithm. To achieve this, we need to enhance the performance of the sequential algorithm by intervening at all levels. The primary objective is to break down the challenging main problem into smaller, independent sub-problems that can be simultaneously solved. The parallel algorithm we propose is designed with two levels. The initial stage involves parallel exploration of the triangles formed during Phase I. The first triangle will be executed by the first CPU, the second triangle will be explored by the second CPU, and the remaining triangles will be handled by the remaining CPUs. The second stage (as depicted in Fig. 8) involves implementing the Branch and Bound method on the GPU. The reduced problem of each triangle is processed sequentially on CPU as long as the number of nodes created is not large. The "threshold" number of nodes, which the problem on the GPU, is set to 256. This number represents the minimum number of nodes at which the GPU used (NVIDIA GeForce RTX 3060) is more efficient than the CPU in terms of computing time. Note that if the number of nodes subsequently falls below 256, the resolution will again be done sequentially. Once this number is reached, the

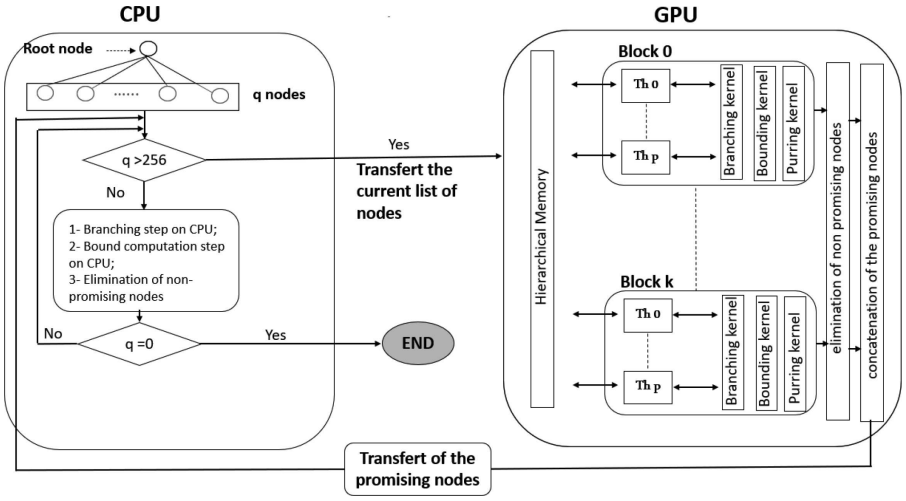

FIGURE 8. Phase II on a CPU-GPU system.

list of nodes is transferred to the GPU's global memory (see Fig. 8).
The communication time between the CPU and the GPU is negligible compared to the overall execution time of the Branch and Bound algorithm. Once the communication is done, the separation step is launched in parallel *via* a Kernel involving as many threads as nodes to be separated; this allows to create q new child nodes. The bound calculation step is then launched, in parallel *via* a Kernel, to evaluate the upper and lower bounds of the q new nodes. We then derive the best lower bound which will be used as a comparison criterion for the pruning step. The main tasks of our parallel algorithm are presented below (see also Fig. 8):

Tasks performed on the CPU:

- Execute the Branch and Bound algorithm on the CPU for small lists;
- Transfer the node list to the GPU once the threshold is surpassed;
- Initiate the separation and bounds computation phases on the GPU;
- Replace non-promising nodes with promising ones to consolidate the node list.

Operations carried out by the GPU:

- Launch the branching Kernel;
- Perform bounds calculations;
- Determine optimal upper and lower bounds;
- Consolidate the node list by removing unpromising nodes.

4. Computational results

To assess the effectiveness of our parallel implementation, we conducted tests on a CPU-GPU system comprising an Intel i7-11800H CPU with 6 cores and 12 threads, leveraging hyper-threading technology, and running at 3.4 GHz. Additionally, we utilized an NVIDIA GeForce RTX 3060 graphics card, which is based on the Ampere architecture, the latest CUDA architecture at the time of our study. Our parallel code was developed using CUDA version 10.1, whereas the sequential code was implemented using the gcc compiler. All algorithms were written in the C++ programming language.

TABLE 1. Sizes and numbers of the different instances.

Group	Number	Size
A	5	50 at 500
1B/A	10	50 at 450
1B/B	10	50 at 450
1B/C	10	50 at 450
C/UNCOR	20	50
C/WEAK	15	50 at 450
C/STRONG	10	50 at 450

4.1. The instances of vOptLib

We validated our proposed parallel algorithm by utilizing instances obtained from https://github.com/vOptSolver/vOptLib. These instances were categorized into three distinct groups based on certain features, which are described below. Table 1 presents a summary of the number and sizes of the instances in each group. Furthermore, the knapsack's capacity is uniformly established at $W = \lfloor 0.5 \times \sum_{j=1}^{n} w_j \rfloor$ for all the instances.

There are five data files in Group A, each containing a set of 0–1 bi-objective knapsack problems. The profit vectors and item weights are generated uniformly, and the instance sizes vary from 50 to 500 items. The ratios of the knapsack's capacity to the total weight of the items ($r = W/\sum_{j=1}^{n} w_j$) in this group range from 0.11 to 0.92, indicating that the knapsack's capacity varies across different instances in this group.

Group B is divided into three subgroups, each containing instances with fixed ratios of 0.5. We examine a total of nine instances in this group, which are labeled as 1B/A, 1B/B, and 1B/C. The first two variants have gains and weights generated from a uniform distribution ranging from 1 to 100. Class 1B/A instances have uncorrelated coefficients, whereas Class 1B/B instances are derived by reversing the second profit vector from Class 1B/A. In contrast, Class 1B/C instances have profit vector components generated uniformly in intervals, with the interval length being less than or equal to 10% of the problem size.

The instances of the last group, C, contain 45 data files representing three sets of 0–1 bi-objective knapsack problems: UNCOR for uncorrelated, WEAK for weakly correlated, and STRONG for strongly correlated. Furthermore, the ratio r is set at 0.5. The three sets are described as follows:

- **UNCOR:** Profits and weights are uniformly generated across all instances in the dataset. Specifically, in 10 of these instances, profits and weights are within the ranges of $[1, 300]$ and $[1, 1000]$, respectively. The remaining instances feature different ranges.
- **WEAK:** This set contains 15 instances with sizes ranging from 50 to 450. Weights are uniformly generated within the range $[1, 1000]$. The second profit vector, denoted by c^2, features values between $[111, 1000]$. Meanwhile, the values for the first profit vector are randomly selected within the range $[c^2 - 100, c^2 + 100]$.
- **STRONG:** This set contains ten highly correlated cases with sizes ranging from 50 to 450. The weights are uniformly generated in the range $[1, 1000]$. The value range for the second profit vector is $[1, 1000]$, while the first profit vector is set to $w_j + 100$.

4.1.1. Results from instances vOptLib

The presented data in Figure 9 gives an outline of the average time taken by our parallel method compared to the sequential algorithm and vOptSolver. The graphs depict the clear superiority of the parallel method over the sequential method and vOptSolver for various problems. These experiments highlight the advantage of using a CPU-GPU system for performance enhancement. However, it is important to note that all three versions of the algorithm exhibit similar behavior in general. As the instance size increases, the resolution time increases significantly, and the manner in which the instance is generated has a direct impact on these times.

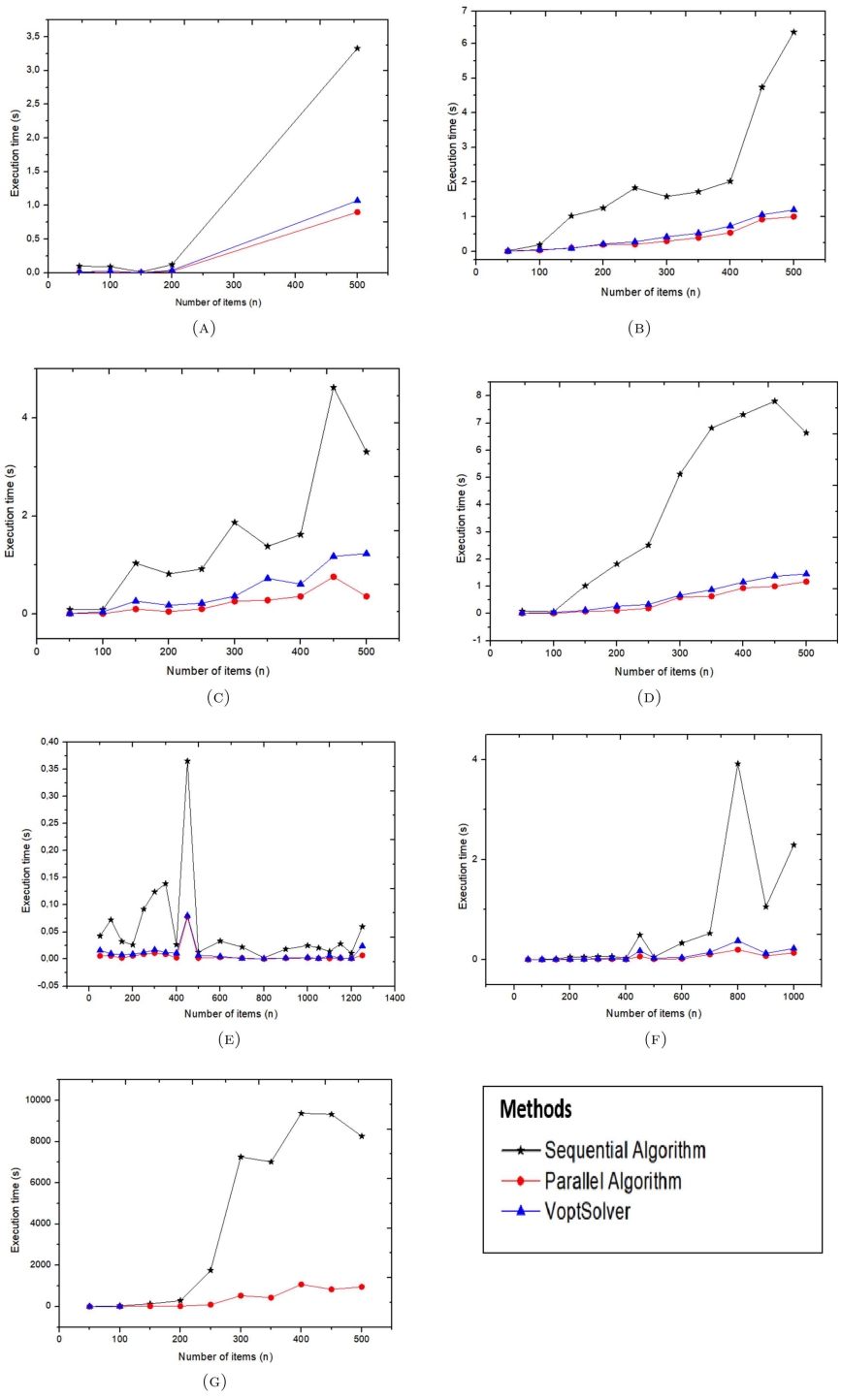

FIGURE 9. Comparative graphs of Sequential, Parallel, and VoptSlover execution times. (a) Inst. 1A, (b) Inst. 1B/A, (c) Inst. 1B/B, (d) Inst. 1B/C, (e) Inst. C/UNCOR, (f) Inst. C/WEAK, (g) Inst. C/STRONG.

Specifically, instances from the **C/Strong** category with objectives that are strongly correlated with the weight vector cannot be solved beyond 150 variables using the two sequential algorithms and vOptSolver (see Fig. 9g). Hence, a parallel algorithm is necessary for such situations.

The graphs presented in Figure 9 demonstrate the speed of the two-phase algorithm on a CPU and a CPU-GPU system. It is clear from the graphs that utilizing an NVIDIA GeForce RTX 3060 GPU leads to a significant improvement in computation time. The proposed parallel two-phase algorithm reduces computation time substantially by taking advantage of the available compute cores on the GPU, enabling more threads to execute in parallel and, thus, improving overall performance. Moreover, as the size of the problem increases, the speedups obtained also increase significantly, with an acceleration of up to 43, indicating a substantial improvement in performance.

4.2. Random Instances

We present the numerical results obtained using both a CPU and a CPU/GPU system, for randomly generated strongly correlated knapsack problems. It is well-known that the single-objective Knapsack problem becomes more difficult to solve when the ratio $\frac{W}{\sum_{j=1}^{n} w_j}$ approaches 0.5. In addition to these problems, we also tested weakly correlated and uncorrelated knapsack problems which are relatively easy to solve and do not generate sufficient nodes to require the use of a GPU. For these types of problems, all computations are carried out on the CPU, as pruning plays a critical role in their solution. However, strongly correlated problems generate a large number of nodes, which can even saturate the memory of the GPU. These problems exhibit the following characteristics:

- Each value of w_j, where j is an integer between 1 and n, is randomly selected from the interval $[1, 100]$.
- The first profit vector is set to $c^1 = w_j + 10$, $j \in \{1, \ldots, n\}$;
- The second profit is created uniformly and randomly in the range $[1, 1000]$;
- $W = \frac{1}{2} \sum_{j=1}^{n} w_j$.

Table 2 displays the average results obtained from ten instances of each problem type presenting the following metrics for each instance:
The CPU execution time of the parallel two-phase algorithm denoted by T_{Parallel} (in seconds) includes its mean, minimum, and maximum values. In contrast, $T_{\text{Sequential}}$ (in seconds) refers to the average CPU execution time for the sequential two-phase algorithm, along with its lower and upper bounds.
$T_{\text{vOptSolver}}$ (in seconds) represents the CPU execution time of vOptSolver [16] including its average, minimum, and maximum values.
T_{WR} (in seconds) indicates the mean, minimum, and maximum CPU execution time for the two-phase sequential algorithm without employing the reduction algorithm. The "-" symbol signifies instances where the method failed to yield results.
The variable $|\text{sol}|_{\text{Parallel}}$ indicates the total count of efficient solutions identified by the end of the parallel algorithm.
speedup: is determined by the ratio between the sequential and parallel runtimes, $T_{\text{Sequential}}$ and T_{Parallel}, respectively.

$$speedup = T_{\text{Sequential}}/T_{\text{Parallel}}. \tag{13}$$

Within Table 3, our detailed analysis targets bi-objective $0, 1$ knapsack problems, meticulously tailored to cover even those instances that are not listed in the compilation by Bazgan *et al.* [3]. Our evaluation transcends the variations inherent in these problem instances by adopting CPU time, quantified in seconds, as the primary measure for evaluating performance across the diverse range of Type A problem instances.

Employing this metric as a cornerstone for our analysis ensures that our study is both comprehensive and comparative, effectively showcasing the enhanced efficiency of our parallel two-phase method. This approach

TABLE 2. Computational results - **Strong instances** -.

	T_{Parallel} (Sec.)			$T_{\text{Sequential}}$ (Sec.)			$T_{\text{vOptSolver}}$ (Sec.)			T_{WR} (Sec.)				
n	Min	Avg	Max	Min	Avg	Max	Min	Avg	Max	Min	Avg	Max	$\lvert sol \rvert_{\text{Parallel}}$	speedup
100	0.052	0.228	0.521	0.832	1.25	2.915	0.621	1.16	2.148	1.021	1.38	3.875	184.2	5.48
200	6.521	12.634	21.826	425.32	528.501	729.204	1308.021	1713.3	2101.207	489.279	724.231	796.294	458.25	43.28
300	57.475	83.892	108.239	1034.87	1538.627	1999.375	4172.520	6782.01	8382.927	1147.024	1682.571	2217.004	1334.28	30.35
400	243.28	307.97	362.44	1308.21	1618.3	2101.30	–	–	–	1248.054	1784.282	3418.025	1521.82	25.81
500	672.38	888.32	1193.12	2031.01	2537.03	2994.23	–	–	–	2147.451	2741.238	3348.173	2666.14	23.15
600	1832.04	2251.41	3116.27	2795.29	3593.04	4746.08	–	–	–	2541.235	3748.279	4872.237	4714.21	21.14
700	2046.45	3774.92	5250.53	3768.28	4814.8	5934.05	–	–	–	3708.947	4899.259	6821.089	5935.24	12.83
800	3192.19	4624.517	5180.86	4800.03	5966.9	6114.07	–	–	–	–	–	–	5323.14	18.92
900	3189.24	4897.264	5897.23	4304.9	5542.3	6752.31	–	–	–	–	–	–	5668.24	21.81
1000	4176.53	5651.37	7027.11	6311.74	6921.08	7291.4	–	–	–	–	–	–	6199.23	11.08
2000	5069.29	6390.786	8670.50	6631.8	7631.83	8087.32	–	–	–	–	–	–	7456.81	16.36
3000	7974.528	8145.922	8927.7	–	–	–	–	–	–	–	–	–	7709.15	–
4000	8792.04	9383.541	9983.71	–	–	–	–	–	–	–	–	–	7954.02	–

TABLE 3. Comparison between Bazgan *et al.* approach [3] and our parallel Two-Phase procedure.

	T_{parallel} (Sec.)			Bazgan (Sec.)		
n	Min	Avg	Max	Min	Avg	Max
100	0.052	0.228	0.521	0.152	0.328	0.600
200	6.521	12.634	21.826	6.768	12.065	21.025
300	57.475	83.892	108.239	57.475	84.001	101.354
400	243.28	307.97	362.44	243.215	307.093	369.999
500	672.38	888.32	1193.12	677.398	889.347	1198.190
600	1832.04	2251.41	3116.27	1833.080	2253.421	3116.670
700	2046.45	3774.92	5250.53	4046.450	5447.921	7250.530

not only meets the high standards expected of comparative analysis but also underscores the versatility and capability of our methodology to tackle a broad spectrum of complex optimization challenges.

Moving forward to Table 4, we offer a detailed comparison of our algorithm against the various alternatives proposed by Figueira *et al.*, as chronicled in their study [12]. This analysis is firmly rooted in the benchmarks established by Bazgan [3]. Our findings reveal a significant decrease in execution time with our method in comparison to the multiple versions introduced by Figueira *et al.*, marking a clear testament to the efficacy of parallelization in addressing knapsack problems of considerable size and complexity.

This meticulous comparative assessment not only demonstrates the superior performance of our approach but also highlights the critical role of parallelization strategies in enhancing the resolution of knapsack problems. Through this analysis, we affirm the significant benefits of adopting parallel computational techniques in optimization, particularly for problems characterized by large dimensions and complex solution spaces.

Table 2 displays the performance of the two-phase algorithm on both CPU and CPU-GPU systems. Our findings demonstrate that utilizing the NVIDIA GeForce RTX 3060 GPU results in substantial speedups, with the parallel two-phase algorithm achieving significant reductions in computation time. The availability of compute cores plays a critical role in parallelization performance and the overall speedup, which tends to increase with the problem size n. We observe accelerations reaching up to 43 times faster than the baseline. Additionally, our experimental results indicate that the sequential method, when employing the reduction algorithm, outperforms the two-phase method without the reduction strategy, highlighting the reduction strategy's significant impact on reducing overall execution time.

TABLE 4. Comparaison between different variants of Figueira *et al.* [12] approach and our Parallel Two-Phase procedure.

n	T_{parallel}	B-DP	B-DP1	B-DP2	B-DP3a.95	B-DP3a 90	B-DP3a.85	B-DP3b.5	B-DP3b.7	B-DP3b.9
100	0.228	0.500	0.400	0.400	0.400	0.400	0.500	0.400	0.400	0.400
200	12.634	24.300	17.200	18.200	17.600	17.700	19.900	21.900	17.700	17.500
300	83.892	193.400	149.000	148.000	146.300	149.800	157.800	267.0	156.400	153.000
400	307.97	782.100	585.700	585.400	583.400	587.600	598.700	888.000	590.400	591.200
500	888.32	2528.500	1913,500	1978.200	1899.400	1906.900	2002.900	3681.500	1887.600	1876.800
600	2251.41	6553.400	5022.3 50	5067.700	5031.300	5028.300	5227.000	13563.700	5275.300	5022.900
700	3774.92	13641.3	11988.7 00	12162.300	11905.8	11860.5	12215.300	25448.000	12037.300	12078.200

5. Conclusion and remarks

This work presents two key findings. Firstly, we establish the superiority of the reduction method [8] over the method proposed in [18] through theoretical analysis. Secondly, we implement a parallel version of the two-phase method for solving the 0–1 bi-objective knapsack problem using CUDA on a CPU-GPU system that incorporates the NVIDIA GeForce RTX 3060 GPU.

Furthermore, we have demonstrated how the parallelism offered by the new massively parallel architecture can be utilized to accelerate the different steps of the two-phase method. We identified the tasks within the algorithm that can be efficiently parallelized, with a particular focus on minimizing the negative effects of diverging paths and thread synchronization within the same warp. Our experimental results reveal that challenging problems can be solved with significantly reduced computation times, resulting in acceleration rates of up to 43. Our study highlights the potential of GPUs for tackling complex combinatorial optimization problems.

It is widely recognized that serial algorithms have played a significant role in solving real-world industrial problems. However, utilizing a parallel architecture for these algorithms has the potential to introduce novel and valuable solutions. Despite this potential, there seems to be a gap in the literature regarding this approach, especially in relation to binary decision variables.Thus, it is crucial for future research to investigate these possibilities, particularly in the context of the multiple objective knapsack problem, multi-objective set covering problem, and multi-objective traveling salesman problem. Further research should also explore the impact of parallel programming on exact and approximate multi-criteria resolution methods, with the potential for cooperative approaches.

Acknowledgements

We would like to express our deepest gratitude to everyone who contributed to the completion of this research.

References

[1] Y.P. Aneja and K.P.K. Nair, Bicriteria transportation problem. *Manag. Sci.* **25** (1979) 73–78.
[2] C. Bazgan, H. Hugot and D. Vanderpooten, Implementing an efficient fptas for the 0–1 multi-objective knapsack problem. *Eur. J. Oper. Res.* **198** (2009) 47–56.
[3] C. Bazgan, H. Hugot and D. Vanderpooten, Solving efficiently the 0–1 multi-objective knapsack problem. *Comput. Oper. Res.* **36** (2009) 260–279.
[4] V.J. Bowman Jr., On the relationship of the tchebycheff norm and the efficient frontier of multiple-criteria objectives. In: *Multiple Criteria Decision Making: Proceedings of a Conference Jouy-en-Josas, France May 21–23, 1975*. Springer (1976) 76–86.
[5] V. Boyer, D. El Baz and M. Elkihel, Solving knapsack problems on gpu. *Comput. Oper. Res.* **39** (2012) 42–47.
[6] M.E. Captivo, J. Climaco, J. Figueira, E. Martins and J.L. Santos, Solving bicriteria 0–1 knapsack problems using a labeling algorithm. *Comput. Oper. Res.* **30** (2003) 1865–1886.
[7] A. Cerqueus, A. Przybylski and X. Gandibleux, Surrogate upper bound sets for bi-objective bi-dimensional binary knapsack problems. *Eur. J. Oper. Res.* **244** (2015) 417–433.
[8] M. Daoud and D. Chaabane, New reduction strategy in the biobjective knapsack problem. *Int. Trans. Oper. Res.* **25** (2018) 1739–1762.
[9] M. Ehrgott and X. Gandibleux, A survey and annotated bibliography of multiobjective combinatorial optimization. *OR-Spektrum* **22** (2000) 425–460.

[10] M. Ehrgott and X. Gandibleux, Multiobjective Combinatorial Optimization–Theory, Methodology, and Applications. Springer (2002) 369–444.
[11] D. El Baz and M. Elkihel, Load balancing methods and parallel dynamic programming algorithm using dominance technique applied to the 0–1 knapsack problem. *J. Parallel Distr. Comput.* **65** (2005) 74–84.
[12] J. Figueira, L. Paquete, M. Simões and D. Vanderpooten, Algorithmic improvements on dynamic programming for the bi-objective {0, 1} knapsack problem. *Comput. Optim. Appl.* **56** (2013) 97–111.
[13] K. Florios, G. Mavrotas and D. Diakoulaki, Solving multiobjective, multiconstraint knapsack problems using mathematical programming and evolutionary algorithms. *Eur. J. Oper. Res.* **203** (2010) 14–21.
[14] X. Gandibleux and A. Freville, Tabu search based procedure for solving the 0–1 multiobjective knapsack problem: The two objectives case. *J. Heuristics* **6** (2000) 361–383.
[15] X. Gandibleux, N. Mezdaoui and A. Fréville, A tabu search procedure to solve multiobjective combinatorial optimization problems. In: *Advances in Multiple Objective and Goal Programming: Proceedings of the Second International Conference on Multi-Objective Programming and Goal Programming, Torremolinos, Spain, May 16–18, 1996*. Springer (1997) 291–300.
[16] X. Gandibleux, G. Soleilhac, A. Przybylski, F. Lucas, S. Ruzika and P. Halffmann, voptsolver, a get and run solver of multiobjective linear optimization problems built on julia and jump. In Vol. 88 *MCDM2017: 24th International Conference on Multiple Criteria Decision Making* (2017).
[17] F. Glover, A multiphase-dual algorithm for the zero-one integer programming problem. *Oper. Res.* **13** (1965) 879–919.
[18] J. Jorge, *Nouvelles propositions pour la résolution exacte du sac-à-dos multi-objectif unidimensionnel en variables binaires*, Ph.D. thesis, Université de Nantes (2010).
[19] M.E. Lalami and D. El-Baz, Gpu implementation of the branch and bound method for knapsack problems. In: *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*. IEEE (2012) 1769–1777.
[20] D.-C. Lou and C.-C. Chang, A parallel two-list algorithm for the knapsack problem. *Parallel Comput.* **22** (1997) 1985–1996.
[21] S. Martello and P. Toth, A new algorithm for the 0–1 knapsack problem. *Manag. Sci.* **34** (1988) 633–644.
[22] S. Martello and P. Toth, Knapsack Problems: Algorithms and Computer Implementations. John Wiley & Sons, Inc. (1990).
[23] Nvidia, Cuda C++ Programming Guide: 11.7. 0 (2022).
[24] W. Pettersson and M. Ozlen, Multiobjective integer programming: Synergistic parallel approaches. *INFORMS J. Comput.* **32** (2020) 461–472.
[25] J. Shen, K. Shigeoka, F. Ino and K. Hagihara, Gpu-based branch-and-bound method to solve large 0–1 knapsack problems with data-centric strategies. *Concurr. Comput. Pract. Exp.* **31** (2019) e4954.
[26] E.L. Ulungu, *Optimisation Combinatoire multicritère: Détermination de l'ensemble des solutions efficaces et méthodes interactives*, Ph.D. thesis, Faculté des Sciences, Université de Mons-Hainaut, Mons, Belgique (1993).
[27] E.L. Ulungu and J. Teghem, The two phases method: An efficient procedure to solve bi-objective combinatorial optimization problems. *Found. Comput. Decis. Sci.* **20** (1995) 149–165.
[28] E.L. Ulungu, J. Teghem, Ph. Fortemps and D. Tuyttens, MOSA method: a tool for solving multiobjective combinatorial optimization problems. *J. Multi-Criteria Decis. Anal.* **8** (1999) 221–236.
[29] M. Visée, J. Teghem, M. Pirlot and E.L. Ulungu, Two-phases method and branch and bound procedures to solve the bi–objective knapsack problem. *J. Glob. Optim.* **12** (1998) 139–155.