Mixed Integer Programming on Modern Parallel Architectures

Chris Misa

University of Oregon, CIS 631

Abstract. Since their initial formulation in the 1950s, Mixed Integer Linear Programming (MILP) problems have continued to pose challenges in algorithmic construction and efficient implementation. Recent efforts solve the general MILP problem using efficient parallel implementations and thousands of CPU cores. This work summarizes the development of the parallel branch-and-bound method for solving MILP problems. We also discuss recent GPU implementations of related discrete optimization problems using the branch-and-bound method in an attempt to explore the feasibility of a GPU-based solver for the general MILP problem.

1 Introduction

A wide range of scheduling, resource management, and network optimization problems can be concisely expressed by a set of variables with linear constraints and an objective function. The problem is to find values for all variables such that the constraints are satisfied and the objective function is maximized (or, equivalently, minimized). When the variables are allowed to take on any real values, this type of problem is known as Linear Programming (LP) and efficient, polynomial-time algorithms exist to find an optimal or near optimal solution [7, 35]. When some of the variables are constrained to integer values, this type of problem is known as Mixed Integer Programming (MILP). The addition of this constraint increases the difficulty of optimization, in fact, the MILP problem is known to be NP-hard [39].

Due to this difficulty, the most efficient known exact algorithms for MILP essentially search through a tree of all possible assignments to integer variables, using LP methods to optimize the non-linear variables at each step. While this strategy is accurate and sufficient for small problem sizes, many practical applications of MILP techniques generate massively complex problems. For example, Friedman [23] casts the problem of scheduling classes for a 4-year university as a MILP with ∼170K variables and a similar number of constraints. Solving this problem reportedly takes up to 40 hours using the state-of-the-art (though closed-source) Gurobi solver [2].

While some applications of MILP, such as class scheduling, are not directly sensitive to solution time, a large range of time-sensitive MILP applications have emerged in the operation of computer and other transportation networks. A particular example of such an application is the allocation of network monitoring resources to effectively detect and mitigate an attack. In this application, resources must be allocated across heterogeneous network devices to gather the data needed to locate the perpetrator. This allocation is subject to the diverse capabilities of heterogeneous network devices and the goal of minimizing the volume of reported information. A recent proposal for such a system [28] incurs latencies up to 30 minutes to compute the optimal resource allocation also using the Gurobi solver. This overhead mitigates the potential effectiveness of this system as 30 minutes is plenty of time for an attacker to penetrate, download sensitive information, and cover their tracks.

The large applicability of MILP for optimization and emergent time-sensitive optimization problems, such as the one described above, has lead to much effort in developing efficient methods to solve arbitrary MILPs on the available hardware architectures. This work surveys these efforts with a focus on recent developments including the possibility of employing GPU architectures for some or all of the work. We start in §2 with a precise definition of the MILP problem and high-level description of branch-and-bound and other proposed algorithms. In §3 we discuss work in developing parallel implementations of the branch-and-bound algorithm developed originally by Land and Doig [32]. Parallel branch-and-bound implementations have been reported to scale up to 80K CPU nodes [46]. The computation power and ready availability of GPUs has lead to several practical documented implementations of optimization problems on GPU architectures [30, 51], however, to the best of our knowledge there is no documented general-purpose MILP solver capable of leveraging GPU architectures. In §4 we highlight some recent implementations of the branch-and-bound method for particular optimization problems. We speculate that since this method

can also be adapted to solving the general case MILP problem, these efforts might lead a path to a GPU-based MILP solver.

In preparing the current work, we have leveraged several prior efforts at summarizing the state-of-the-art in optimization implementations. Of these prior efforts the work of Ralphs [40] and Schryen [42] provide excellent taxonomies of parallel implementations of the MILP problem, while Boyer [12] and Schulz [43] summarize implementation issues of specific optimization problems using GPU architectures. As demonstrated by the 286 unsolved MILP problems listed by MIPLIB2017 [3], this area presents exciting theoretic and applied research challenges.

2 Background

Mixed Interger Linear Programming (MILP) is a mathematical framework for formulating a wide variety of optimization problems. In this framework, optimization is formulated as minimizing (or maximizing) a linear combination of variables subject to a set of constraints and the condition that some variables must take only integral values. In essence, MILP is a generalization of both Linear Programming (LP), where all variables take real values, and Integer Programming (IP), where all variables take integer values. This section provides formal definitions of the LP and MILP problems and discusses several of the still relevant historic algorithms for these problems.

2.1 Linear Programming

LP has perhaps the richest history among all optimization problems, dating back to 1947 [18]. Following loosely the notation of Ralphs [40], the standard form for expressing an LP problem is as follows:

$$\min_{x \in \mathcal{F}} c^{\mathrm{T}} x \tag{1}$$

$$\text{s.t. } \mathcal{F} = \{x \in \mathbb{R}^n \mid Ax \leq b\} \tag{2}$$

where $A \in \mathbb{R}^{m \mathrm{x} n}$ and $b \in \mathbb{R}^m$ describe the constraints on the variables $x_1, \ldots, x_n$ and $c \in \mathbb{R}^n$ specifies the objective function. A setting of x which satisfies (2) is known as a *feasible* solution and a setting of x which satisfies both (1) and (2) is known as an *optimal* solution. The goal of a linear programming application is then to find the optimal solution given the objective function and constraints or to report that the problem is unbounded.

Most performant implementations of linear programming are built on algorithms derived from Dantzig's original simplex method [17]. At a high level, this method identifies that any optimal solution will fall on the edge of the n-dimensional polyhedron defined by $\mathcal{F}$. The method iteratively explores the convex hull of $\mathcal{F}$ until the optimal solution is found. The complexity of this algorithm is typically measured in the number of iterations taken for a given problem as a function of problem dimensionality.

While the worst case time complexity of the simplex method is known to be exponential, for a wide range or practical optimization problems, it executes in near-linear time [44]. Probabilistic analysis [10,47] demonstrates the average case time complexity is, in fact, linear. Several other theoretic results support the simplex method's utility when the dimension is fixed [19,36].

2.2 Mixed Integer Linear Programming

A useful restriction of the LP problem is to require some of the variables to only take integer values. For example, many scheduling problems deal with indivisible units of work. Again following the notation of Ralphs [40], we can extend the definition of $\mathcal{F}$ from above to include this constraint:

$$\mathcal{F} = \{x \in \mathbb{R}^n \mid Ax \leq b,\ x_j \in \mathbb{Z},\ \text{for all } j \in \mathcal{I}\} \tag{3}$$

where $\mathcal{I} \subseteq \{1, \ldots, n\}$ indicates which of $x_0, \ldots, x_n$ are constrained to integer values. Note that if $\mathcal{I} = \varnothing$, this is equivalent to the LP case and that if $\mathcal{I} = \{1, \ldots, n\}$, it is equivalent to the IP case. Despite the apparent similarity to LP, MILP and IP are actually much harder problems. Because $\mathcal{F}$ might now not be convex, essentially all possible assignments of the integral variables must be checked with respect to the optimization goal.

Several different approaches to the MILP problem emerged [8,27], but the most effective to this day is the branch and bound method, attributed originally to Land and Doig [32]. At a high level, this method first removes the integral constraints and solves what is known as the LP relaxation of the MILP problem. The method then searches through the set of feasible solutions to the original MILP problem while refining the maximum and minimum possible values of the objective function. This search is carried out on a tree of subproblems where subsequent layers of the tree fix more and more of the required variables to integer values. Bounds on the objective function are computed for each subproblem using the LP relaxation on a partition of the original feasible region as imposed by the integer values fixed in that sub problem. Subproblems which cannot provide a more

optimal bound on the objective function can be discarded, reducing the size of the search space. This process is repeated until integer values have been found for all variables in $\mathcal{I}$.

Much work on optimizing this method in serial implementations has been undertaken since the original algorithmic development [29,34]. This work explores the implications of different pre-processing, branching, searching, and heuristic techniques for improving performance of MILP solvers. Essentially, these techniques seek to reduce the size of the tree that must be explored, find the best order in which to process nodes, and leverage the warm-start capabilities of the simplex method typically used to solved the underlying LP relaxation.

3 Parallel Solutions

As in other fields of computing, the end of increasing clock speeds and the ready availability of parallel computing architectures has driven efforts to efficiently parallelize MILP algorithms. Since the branch-and-bound method introduced in section 2 essentially reduces to a tree search, it might seem to be an easy candidate for parallel MILP implementations. However, particular features of the tree and the operations required on each node in the search make parallel implementations of branch-and-bound quite difficult. This section summarizes the key algorithmic challenges faced and discusses several implementation strategies.

3.1 Algorithmic Challenges

While the branch-and-bound algorithm for solving MILPs is a type of tree search algorithm, several key details make it more challenging to parallelize than seemingly similar tree searches (e.g., depth-first-search). According to Ralphs [40], these details can be summarized as follows.

Children are smaller than parents. Since children are subproblems based on partitions of the original feasible region, they are easier LP relaxations to solve. Also, the overheads of many optimizations (e.g., presolving [6]) manifest in the processing of the root or parent nodes, leading to delays in the generation of enough children for parallelization to be useful.

The tree is un-balanced and dynamic. Since the branch-and-bound method eliminates child nodes which cannot achieve a better optimum value of the objective function, the tree generated by such methods are inherently unbalanced. Moreover, the shape of the tree is not known before hand [38], exacerbating attempts to determine an optimal schedule on the given compute resources.

Order determines efficiency. In theory, child nodes can be processed in any order, but in practice [34], particular ordering of child nodes impact the convergence time of the objective function. Enforcing node processing order reduces the ability of an implementation to leverage parallel resources.

Information must be shared between nodes. Nodes must at least share their updated bounds at the end of each iteration as this determines which children will get dropped in the next iteration. Other serial optimization (e.g., warm-starting the simplex method or conflict analysis [4]) leverage sharing of more detailed information from the LP relaxations between child nodes to improve performance.

The above points are direct results of the particular tree search required by the branch-and-bound algorithm and also results of the fact that efforts to optimize serial implementations often create dependencies which impeded the development of parallel implementations.

3.2 Parallelization Strategies

At a high-level, all parallel branch-and-bound implementations must determine the granularity of work to distribute as well as the method to distribute and synchronize work among the available processing units.

Here we described a simplification of the properties introduced by Ralphs [40] which build on two prior surveys [25, 33].

Work Granularity. The work of the branch-and-bound algorithm can be divided among worker threads at different levels or granularity: tree, subtree, node, and subnode. Tree-level parallelization [20] processes multiple trees with slightly different parameters in parallel, reporting the result of the first to finish or aggregating across final results. Subtree-level parallelization breaks up the global search tree and hands subtrees to worker threads. This approach facilitates sharing of information between nodes better than finer-grained solutions and is favored in modern high performance implementations [52]. Node-level parallelization distributes the operations required for individual nodes as units of work. While this approach is popular [22, 41], these implementations must carefully manage the exchange of global state and synchronization among nodes. Finally, some implementations advocate for sub-node parallelism where the operations of processing each node are split among multiple parallel

threads. For example, each node's LP relaxation can be solved using efficient LP implementations [24].

Distribution and Synchronization. The requirements for work distribution and synchronization vary for the different granularities described above. Under tree-level parallelization, little synchronization is required and each iteration of the algorithm can proceeded independently. As the granularity decreases, however, synchronization requirements become more of a bottleneck.

Static work distribution schemes are challenging to implement effectively since the shape of the tree is not know in advance. As an exception, Fischetti et al. [21] demonstrate almost linear speed up using self-splitting workloads in the related constraint programming problem. In their approach each worker thread accesses the same data, but chooses which nodes to process based on a sudo-random sequence. At the outset, each thread's random sequence is seeded with a unique value so that work is distributed uniformly among workers w.h.p., regardless of how un-balanced the search tree. As with other static work distributions, this scheme has the advantage of requiring barely any inter-thread communication.

Dynamic scheduling using variants of the master-worker paradigm are far more common with many high performance solutions. For example, CBC [22] employs a master-worker architecture with node-level or subtree-level granularity. To enforce consistency of the global state, a synchronization period is performed after each worker completes the assigned work. Another effort, BLISS [52] employs a master-hub-worker model in the CHiPPS [1] framework. This model reduces the bottleneck inherent at the master node by placing intermediate hub nodes between master and workers. BLISS also implements load balancing at the subtree-level of granularity to improve storage and communication costs.

Perhaps the largest scale deployment of parallel MILP solvers are due to the work of Shinano et al. [45] who parallelize the serial MILP solver known as SCIP [5]. They use large distributed memory environments (e.g., the HLRN II supercomputer) to run multiple SCIP solvers on subproblems (i.e., subtree-level parallelism) in a master-worker paradigm. Workers that complete their subproblems early are assigned unsolved nodes in a kind of centrally managed work-stealing approach. The massive scales possible with this implementation enable the authors to solve 10 previously unsolved MILPs on clusters with up to 80 000 cores [46].

Summary: All parallel implementations for the branch-and-bound algorithm for MILP must deal with the un-balanced search trees generated and the need to share information between nodes of the tree. These requirements motivate solutions to seek dynamic scheduling solutions often using master-worker variants and flexible partitioning of the search tree among workers.

4 GPU Implementations

Increased availability, flexible programming models, and computational power of modern GPUs has motivated efforts to implement various optimization algorithms on GPU-based systems [12,43]. While several efforts have yielded efficient implementations of the revised simplex method for the LP problem on GPU systems [9,24,48], to the best of our knowledge there is no equivalent general MILP implementation which makes substantial use of GPU systems. Nonetheless, several recent efforts implement branch-and-bound on GPU-based systems in the context of specific optimization problems (e.g., the knapsack problem [13,31]). This section discusses some of the key challenges in implementing optimization methods like branch-and-bound on GPU systems and summarizes some recent implementation efforts.

4.1 Challenges

Many of the key challenges outlined in section 3.1 are also faced by GPU implementations of the bound-and-branch method. Moreover, the irregular data structures and data sharing required by this method pose implementation challenges particular to GPU architectures. For example, solving the LP relaxations takes a large fraction (97%-99%) of node processing time [37] and there are known efficient GPU implementations of the simplex method. However, transferring data to and from the GPU may cost more than the speed up achieved by executing this computation on the GPU.

An additional challenge for GPU implementation of branch-and-bound algorithms is the prevalence of branching instructions. Despite recent hardware developments, GPU architectures are known to be highly sensitive to in-kernel branches which can lead to thread divergence. Chakroun et al. [16] develop efficient re-orderings of operations and data in the branch-and-bound algorithm to reduce the number of divergent instructions within each thread, achieving up to 7x speedup compared to a CPU-based parallel implementation.

4.2 Implementations

Prior GPU-based implementations of branch-and-bound can be roughly divided into solutions which combine CPU and GPU for execution of the algorithm and solutions which use only GPU.

GPU-CPU. Lalami, et al. [11,31] present an early implementation of branch-and-bound implementation of the knapsack problem with the branching and bounding steps implemented as CUDA kernels while the CPU executes the pruning operation. Due to the expense of copying the active nodes back and forth from GPU memory, they fall back on a CPU-only implementation when the problem size falls below a threshold. They report up to 10x speed up compared with a parallel CPU implementation using randomly generated problems.

In addition to the contributions towards reducing thread divergence mentioned previously, Chakroun et al. [15] propose an implementation of the flowshop problem using a branch-and-bound approach which also executes the node pruning operation as a GPU kernel. They also propose an implementation where the CPU and GPU work concurrently on solving subproblems drawn from a common problem pool. While standard test problems are used for evaluation, they only report speed up over a serial implementation.

Vu et al. [49, 50] also implement a branch-and-bound version of the flowshop problem, but target larger systems with tens of GPUs and hundreds of CPUs. Their key contribution is the use of dynamic work stealing to efficiently distribute the unbalanced tree across all available compute resources. Using standard test problems, they report near optimal speed up when varying the number of CPU and GPU nodes available.

Pure GPU. While the efforts described above make markable progress towards leveraging GPUs for branch-and-bound implementation, they still suffer from the bottleneck of transferring data to and from GPU memory. To this end, Gmys et al. [26] propose an implementation which executes all operations in the GPU and hence only requires data transfer at the beginning and end of execution. The key technique in their approach is the use of an Integer Vector Matrix (IVM) representation of the branch-and-bound problem tree rather than the linked-list used in prior efforts. The IVM data structure it well-adapted the GPU programming and memory models as it is constant in size and encourages better locality. Similar to Vu et al., they implement work stealing to balance load among GPU threads. They demonstrate a 3.3x speed up over prior work [14] using the same flowshop problem set.

Summary: The issues posed by un-balanced search trees and information sharing between nodes continue to challenge GPU implementations of the branch-and-bound algorithm. Recent work using the Integer Vector Matrix representation of the search tree enables pure-GPU implementations which demonstrate promising speed ups. These advances in leveraging GPUs for optimization problems in general, and the branch-and-bound algorithm in particular, expose the possibility of a GPU-based MILP solver.

5 Conclusion

In this work, we provided an updated survey of the implementation of MILP solvers on modern parallel architectures. The key challenges faced by any parallel implementation are related to the un-balanced search tree explored by the branch-and-bound algorithm. To address these challenges, most implementations use some form of dynamic scheduling such as master-worker or work stealing methods. Finally, we discussed several recent works employing GPUs for branch-and-bound-based optimization problems. These efforts shed light on the potential use of GPUs in MILP solvers. In particular, the Integer Vector Matrix representation of the search tree provides a promising path towards pure-GPU implementations of such algorithms.

References

1. COIN-OR high-performance parallel search. `https://projects.coin-or.org/CHiPPS`.
2. Gurobi - the fastest solver. `https://www.gurobi.com`.
3. MIPLIB 2017. `http://miplib.zib.de`, 2018.
4. Tobias Achterberg. Conflict analysis in mixed integer programming. *Discrete Optimization*, 4(1):4–20, 2007.
5. Tobias Achterberg. SCIP: solving constraint integer programs. *Mathematical Programming Computation*, 1(1):1–41, 2009.
6. Tobias Achterberg, Robert E Bixby, Zonghao Gu, Edward Rothberg, and Dieter Weninger. Presolve reductions in mixed integer programming. 2016.
7. Noga Alon and Nimrod Megiddo. Parallel linear programming in fixed dimension almost surely in constant time. In *Proceedings of the 31st Annual Symposium on Foundations of Computer Science*, pages 574–582. IEEE, 1990.
8. Egon Balas, Fred Glover, and Stanley Zionts. An additive algorithm for solving linear programs with zero-one variables. *Operations Research*, 13(4):517–549, 1965.

9. Jakob Bieling, Patrick Peschlow, and Peter Martini. An efficient GPU implementation of the revised simplex method. In *2010 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW)*, pages 1–8. IEEE, 2010.
10. Karl Heinz Borgwardt. *The Simplex Method: a probabilistic analysis*, volume 1. Springer Science & Business Media, 2012.
11. Abdelamine Boukedjar, Mohamed Esseghir Lalami, and Didier El-Baz. Parallel branch and bound on a CPU-GPU system. In *2012 20th Euromicro International Conference on Parallel, Distributed and Network-based Processing*, pages 392–398. IEEE, 2012.
12. Vincent Boyer and Didier El Baz. Recent advances on GPU computing in operations research. In *2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum*, pages 1778–1787. IEEE, 2013.
13. Vincent Boyer, Didier El Baz, and Moussa Elkihel. Solving knapsack problems on GPU. *Computers & Operations Research*, 39(1):42–47, 2012.
14. Imen Chakroun and Nordine Melab. Operator-level GPU-accelerated branch and bound algorithms. *Procedia Computer Science*, 18:280–289, 2013.
15. Imen Chakroun, Nordine Melab, Mohand Mezmaz, and Daniel Tuyttens. Combining multi-core and GPU computing for solving combinatorial optimization problems. *Journal of Parallel and Distributed Computing*, 73(12):1563–1577, 2013.
16. Imen Chakroun, Mohand Mezmaz, Nouredine Melab, and Ahcene Bendjoudi. Reducing thread divergence in a GPU-accelerated branch-and-bound algorithm. *Concurrency and Computation: Practice and Experience*, 25(8):1121–1136, 2013.
17. George B Dantzig, Alex Orden, Philip Wolfe, et al. The generalized simplex method for minimizing a linear form under linear inequality restraints. *Pacific Journal of Mathematics*, 5(2):183–195, 1955.
18. George Bernard Dantzig. *Linear programming and extensions*. Princeton university press, 1998.
19. Martin E Dyer and Alan M Frieze. A randomized algorithm for fixed-dimensional linear programming. *Mathematical Programming*, 44(1-3):203–212, 1989.
20. Matteo Fischetti, Andrea Lodi, Michele Monaci, Domenico Salvagnin, and Andrea Tramontani. Improving branch-and-cut performance by random sampling. *Mathematical Programming Computation*, 8(1):113–132, 2016.
21. Matteo Fischetti, Michele Monaci, and Domenico Salvagnin. Self-splitting of workload in parallel computation. In *International Conference on AI and OR Techniques in Constriant Programming for Combinatorial Optimization Problems*, pages 394–404. Springer, 2014.
22. J. Forrest. CBC MIP solver. `http://www.coin-or.org/Cbc`.
23. Joshua S Friedman. Automated timetabling for small colleges and high schools using huge integer programs. *arXiv preprint arXiv:1612.08777*, 2016.
24. Arash Raeisi Gahrouei and Mehdi Ghatee. Effective implementation of GPU-based revised simplex algorithm applying new memory management and cycle avoidance strategies. *arXiv preprint arXiv:1803.04378*, 2018.
25. Bernard Gendron and Teodor Gabriel Crainic. Parallel branch-and-bound algorithms: Survey and synthesis. *Operations research*, 42(6):1042–1066, 1994.
26. Jan Gmys, Mohand Mezmaz, Nouredine Melab, and Daniel Tuyttens. A GPU-based branch-and-bound algorithm using Integer–Vector–Matrix data structure. *Parallel Computing*, 59:119–139, 2016.
27. Ralph E Gomory et al. Outline of an algorithm for integer solutions to linear programs. *Bulletin of the American Mathematical society*, 64(5):275–278, 1958.
28. Arpit Gupta, Rob Harrison, Marco Canini, Nick Feamster, Jennifer Rexford, and Walter Willinger. Sonata: Query-driven streaming network telemetry. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 357–371. ACM, 2018.
29. Ellis L Johnson, George L Nemhauser, and Martin WP Savelsbergh. Progress in linear programming-based algorithms for integer programming: An exposition. *Informs journal on computing*, 12(1):2–23, 2000.
30. Sai Prashanth Josyula, Johanna Törnquist Krasemann, and Lars Lundberg. Exploring the potential of GPU computing in train rescheduling. In *8th International Conference on Railway Operations Modelling and Analysis*, 2019.
31. Mohamed Esseghir Lalami and Didier 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*, pages 1769–1777. IEEE, 2012.
32. A. H. Land and A. G. Doig. An automatic method of solving discrete programming problems. *Econometrica*, 28(3):497–520, 1960.
33. E. L. Lawler and D. E. Wood. Branch-and-bound methods: A survey. *Operations Research*, 14(4):699–719, 1966.
34. Jeff T Linderoth and Martin WP Savelsbergh. A computational study of search strategies for mixed integer programming. *INFORMS Journal on Computing*, 11(2):173–187, 1999.
35. Michael Luby and Noam Nisan. A parallel approximation algorithm for positive linear programming. In *STOC*, volume 93, pages 448–457, 1993.
36. Nimrod Megiddo. Linear programming in linear time when the dimension is fixed. *Journal of the ACM (JACM)*, 31(1):114–127, 1984.
37. Nouredine Melab, Imen Chakroun, and Ahcène Bendjoudi. Graphics processing unit-accelerated

bounding for branch-and-bound applied to a permutation problem using data access optimization. *Concurrency and Computation: Practice and Experience*, 26(16):2667–2683, 2014.

38. Osman Y Özaltın, Brady Hunsaker, and Andrew J Schaefer. Predicting the solution time of branch-and-bound algorithms for mixed-integer programs. *INFORMS Journal on Computing*, 23(3):392–403, 2011.
39. Christos H Papadimitriou. On the complexity of integer programming. *Journal of the ACM (JACM)*, 28(4):765–768, 1981.
40. Ted Ralphs, Yuji Shinano, Timo Berthold, and Thorsten Koch. Parallel solvers for mixed integer linear optimization. In *Handbook of parallel constraint reasoning*, pages 283–336. Springer, 2018.
41. Ted K Ralphs. Parallel branch and cut for capacitated vehicle routing. *Parallel Computing*, 29(5):607–629, 2003.
42. Guido Schryen. Parallel computational optimization in operations research: A new integrative framework, literature review and research directions. *arXiv preprint arXiv:1910.03028*, 2019.
43. Christian Schulz, Geir Hasle, André R Brodtkorb, and Trond R Hagen. GPU computing in discrete optimization. part ii: Survey focused on routing problems. *EURO journal on transportation and logistics*, 2(1-2):159–186, 2013.
44. Ron Shamir. The efficiency of the simplex method: A survey. *Management Science*, 33(3):301–334, 1987.
45. Yuji Shinano, Tobias Achterberg, Timo Berthold, Stefan Heinz, and Thorsten Koch. ParaSCIP: a parallel extension of SCIP. In *Competence in High Performance Computing 2010*, pages 135–148. Springer, 2011.
46. Yuji Shinano, Tobias Achterberg, Timo Berthold, Stefan Heinz, Thorsten Koch, and Michael Winkler. Solving open MIP instances with ParaSCIP on supercomputers using up to 80,000 cores. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 770–779. IEEE, 2016.
47. Steve Smale. On the average number of steps of the simplex method of linear programming. *Mathematical programming*, 27(3):241–262, 1983.
48. Daniele G Spampinato and Anne C Elstery. Linear optimization on modern GPUs. In *2009 IEEE International Symposium on Parallel & Distributed Processing*, pages 1–8. IEEE, 2009.
49. Trong-Tuan Vu and Bilel Derbel. Parallel branch-and-bound in multi-core multi-CPU multi-GPU heterogeneous environments. *Future Generation Computer Systems*, 56:95–109, 2016.
50. Trong-Tuan Vu, Bilel Derbel, and Nouredine Melab. Adaptive dynamic load balancing in heterogeneous multiple GPUs-CPUs distributed setting: Case study of B&B tree search. In *International Conference on Learning and Intelligent Optimization*, pages 87–103. Springer, 2013.
51. Joong-Ho Won, Yongkweon Jeon, Jarrett K Rosenberg, Sungroh Yoon, Geoffrey D Rubin, and Sandy Napel. Uncluttered single-image visualization of vascular structures using GPU and integer programming. *IEEE transactions on visualization and computer graphics*, 19(1):81–93, 2012.
52. Yan Xu, Ted K Ralphs, Laszlo Ladányi, and Matthew J Saltzman. Computational experience with a software framework for parallel integer programming. *INFORMS Journal on Computing*, 21(3):383–397, 2009.