Zuse Institute Berlin

Takustr. 7
14195 Berlin
Germany

TED RALPHS[1], YUJI SHINANO, TIMO BERTHOLD[2], THORSTEN KOCH

Parallel Solvers for Mixed Integer Linear Programming

[1] Department of Industrial and Systems Engineering, Lehigh University, Bethlehem, PA 18015, USA, ted@lehigh.edu
[2] Fair Isaac Germany GmbH, Takustr. 7, 14195 Berlin, Germany, timoberthold@fico.com

This work has been supported by the Research Campus MODAL *Mathematical Optimization and Data Analysis Laboratories* funded by the Federal Ministry of Education and Research (BMBF Grant 05M14ZAM) and by Lehigh University. All responsibility for the content of this publication is assumed by the authors.

ZIB Report 16-74 (December 2016)

Zuse Institute Berlin
Takustr. 7
14195 Berlin
Germany

Telephone: +49 30-84185-0
Telefax: +49 30-84185-125

E-mail: bibliothek@zib.de
URL: http://www.zib.de

ZIB-Report (Print) ISSN 1438-0064
ZIB-Report (Internet) ISSN 2192-7782

Parallel Solvers for Mixed Integer Linear Programming

Ted Ralphs[‡] Yuji Shinano[§] Timo Berthold[¶] Thorsten Koch[‖]

24/Dec/2016

Abstract

In this article, we introduce parallel mixed integer linear programming (MILP) solvers. MILP solving algorithms have been improved tremendously in the last two decades. Currently, commercial MILP solvers are known as a strong optimization tool. Parallel MILP solver development has started in 1990s. However, since the improvements of solving algorithms have much impact to solve MILP problems than application of parallel computing, there were not many visible successes. With the spread of multi-core CPUs, current state-of-the-art MILP solvers have parallel implementations and researches to apply parallelism in the solving algorithm also getting popular. We summarize current existing parallel MILP solver architectures.

1 Introduction

This article addresses the solution of *mixed integer linear optimization problems* (MILPs) on parallel computing architectures. An MILP is a problem of the following general form:

$$\min_{x \in \mathcal{F}} c^\top x \tag{MILP}$$

where the set

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

is the *feasible region*, described by a given matrix $A \in \mathbb{Q}^{m \times n}$; vectors $b \in \mathbb{Q}^m$ and $c, l, u \in \mathbb{Q}^n$; and a subset $I \subseteq \{1, \ldots, n\}$ indicating which variables are required to have integer values. Members of $\mathcal{F}$ are called *solutions* and are assignments of values to the *variables*. The polyhedron

$$\mathcal{P} = \{x \in \mathbb{R}^n \mid Ax \leq b, l \leq x \leq u\}$$

is the feasible region of a *linear optimization problem* (LP)

$$\min_{x \in \mathcal{P}} c^\top x \tag{LP}$$

known as the *LP relaxation*. The class of problems that can be expressed in this form is quite broad and many optimization problems arising in practice can be modeled as MILPs, see, e.g., [65]. As a language for describing optimization problems, MILP (and mathematical optimization, more generally) has proven to be flexible, expressive, and powerful.

[‡]Lehigh University, Bethlehem, PA, USA, ted@lehigh.edu
[§]Zuse Institute Berlin, Takustraße 7, 14195 Berlin, Germany, shinano@zib.de
[¶]Fair Isaac Germany GmbH, Germany, Takustraße 7, 14195 Berlin, Germany, timoberthold@fico.com
[‖]Zuse Institute Berlin, Takustraße 7, 14195 Berlin, Germany, koch@zib.de

All state-of-the-art solvers for MILP employ one of many existing variants of the well-known branch-and-bound algorithm of [55]. This class of algorithm searches a dynamically constructed tree (known as the *search tree*), following the general scheme of the generic tree search algorithm specified in Algorithm 1. Sequential algorithms for solving MILPs vary broadly based in their

Algorithm 1: A Generic Tree Search Algorithm

```
1 Add root r to a priority queue Q.
2 while Q is not empty do
3     Remove a node i from Q.
4     Process the node i.
5     Apply pruning rules.
6     if Node i can be pruned then
7         Prune (discard) node i.
8     else
9         Create successors of node i (branch) by applying a successor function, and add the
            successors to Q.
```

methods of processing nodes (line 4); their strategies for the order of processing nodes (*search strategy*, line 3); their rules for pruning nodes (line 5); and their strategies for creating successors (*branching strategy*, line 9). We provide some details on how each of these steps is typically managed in Section 2. In the case of a parallel algorithm, some additional strategic factors come into play, such as how the search is managed (now in parallel), what information is shared as global knowledge and the specific mechanism for sharing this knowledge on a particular target computer architecture.

Tree search algorithms appear to be naturally parallelizable and soon after the advent of networked computing, researchers began to experiment with parallelizing branch and bound. In [39], Gendron and Crainic chronicle the early history, dating the first experiments to somewhere in the 1970s. It did not take long after the first large-scale systems became available to realize that good parallel performance is often difficult to achieve. In the case of MILPs, this is particularly true (and becoming more so over time) for reasons that we summarize in Section 3.

Despite the enormity of the existing literature on solving search problems in parallel, the case of MILP appears to present unique challenges. Although some progress has been made in the more than two decades in which parallel algorithms for MILP have been seriously developed, it is fair to say that many challenges remain. The difficulty comes from a few intertwining sources. For one, the most capable sequential solvers are commercial software and therefore only available to members of the research community as black boxes with which parallel frameworks can interact in limited ways. Even if this was not the case, it is also generally true that more sophisticated solvers are inherently more difficult to parallelize in a scalable fashion because of their greater exploitation of global information sharing and increasing emphasis on operations that limit the size of the search tree (and hence limit opportunities for parallelization), among other things. Despite an appearance to the contrary, state-of-the-art sequential algorithms for solving MILPs depend strongly on the order in which the nodes of the tree are processed and advanced techniques for determining this order are in part responsible for the dramatic improvements that have been observed in sequential solution algorithms [58]. It is difficult to replicate this ordering in parallel without centralized control mechanisms, which themselves introduce inefficiencies. Moreover, sequential algorithms heavily exploit the warm-starting capabilities of the underlying LP solver in that they can usually process the child of a node shoe parent has just been processed, a phenomena which is more difficult to exploit in combination with the load-balancing techniques described in

Section 3.2.5. Finally, most research has focused on parallelization of less sophisticated sequential algorithms, which both limits the sophistication of the overall algorithm (and hence limits the degree to which challenging open problems can be tackled) and inherently involves different challenges to achieving scalability.

The limits to scalability are by now well understood, but the degree to which these limits can be overcome is unclear and will unfortunately remain a moving target. Scalability involves the interplay of a changing landscape of computer architectures and evolving sequential algorithms, neither of which are being developed with the goal of making this (or any other) particular class of algorithms efficient. The days in which we could expect to see exponential improvements to sequential algorithms are gone, so there will likely be increasing emphasis on the development of algorithms that effectively exploit parallel architectures in the future.

In the remainder of this article, we survey the current landscape with respect to parallel solution of MILPs. By now, most existing solvers have been parallelized in some fashion, while in addition, there are also a few frameworks and approaches for parallelizing solvers that are only available as black boxes. We begin by briefly surveying the current state of the art in sequential solution algorithms in Section 2. In Section 3, we provide an overview of issues faced in parallelizing these algorithms and the design decisions that must be taken during development. In Section 4, we review the approaches taken by a number of existing solvers. Finally, in Section 5, we discuss the tricky issue of how to measure performance of a solver before concluding in Section 6.

2 Sequential Algorithms

2.1 Basic Components

As we have already mentioned, most modern solvers employ sophisticated variants of the well known branch-and-bound algorithm of [55]. The basic approach is straightforward, yet effective: the feasible region of the original optimization problem is partitioned to obtain smaller *subproblems*, each of which is recursively solved using the same algorithm. This recursive process can be interpreted naturally as a tree search algorithm and visualized as the exploration of a *search tree*. The goal of the search is to implicitly enumerate all potential assignments of the values of the integer variables. The power of the algorithm comes from the bounds that are calculated during the processing of nodes in the search tree and are used to truncate the recursive partitioning process in order to avoid what would eventually amount to the costly complete enumeration of all solutions.

An updated version of Algorithm 1 that reflects that specific way in which a tree search is managed according to the basic principles of branch-and-bound is shown in Algorithm 2. In employing the metaphor of this algorithm as the exploration of a search tree, we associate each subproblem with a *node* in the search tree and describe the tree itself through parent-child relationships, beginning with the root node. Thus, each subproblem has both a *parent* and zero or more *children*. Subproblems with no children are called *terminal* or *leaf* nodes. In Algorithm 2, the set Q is the set of terminal subproblems of the search tree as it exists at the end of each iteration of the algorithm. We next describe the specific elements of this algorithm in more detail.

Bounding. The processing of a subproblem in line 4 of Algorithm 2 consists of the computation of updated upper and lower bounds on the value of an optimal solution to the subproblem.

The lower bound is calculated with the help of a relaxation, which is constructed so as to be easy to solve. The lower bounding scheme of most MILP solvers is based on solution of a strengthened version of the LP relaxation (LP), which can be solved efficiently, i.e., in time polynomial in the size of the input data [50, 66]. The strengthening is done using techniques

Algorithm 2: A Generic Branch-and-Bound Algorithm

1 Add root optimization problem r to a priority queue Q. Set global upper bound $U \leftarrow \infty$ and global lower bound $L \leftarrow -\infty$
2 **while** $L < U$ **do**
3 Remove the highest priority subproblem i from Q.
4 **Bound** the subproblem i to obtain (updated) final upper bound $U(i)$ and (updated) final lower bound $L(i)$.
5 Set $U \leftarrow \min\{U(i), U\}$.
6 **if** $L(i) < U$ **then**
7 **Branch** to create child subproblems $i_1, \ldots, i_k$ of subproblem i with
 - upper bounds $U(i_1), \ldots U(i_k)$ (initialized to ∞ by default); and
 - initial lower bounds $L(i_1), \ldots, L(i_k)$ (initialized to $L(i)$ by default).
 by partitioning the feasible region of subproblem i.
8 Add $i_1, \ldots, i_k$ to Q.
9 Set $L \leftarrow \min_{i \in Q} L(i)$.

stemming from another, more involved procedure for solving MILPs known as the *cutting plane method* [40] that was developed prior to the introduction of the branch-and-bound algorithm. The basic idea of the cutting plane method is to iteratively solve and strengthen the LP relaxation of a MILP. To do so, in each iteration, one or more *valid inequalities* are added to the LP relaxation. These inequalities have to fulfill two requirements:

1. they are violated by the computed optimal solution to the LP relaxation and
2. they are satisfied by all "improving solutions" (those with objective values smaller than U) in the feasible set $\mathcal{F}$.

Since they "cut off" the observed optimal solution to LP relaxation, such inequalities are called *cutting planes* or *cuts*. For an overview on cutting plane algorithms for MILP, see, e.g., [62, 97].

An upper bound, on the other hand, results when either the solution to the relaxation is also a member of $\mathcal{F}$ or a solution feasible for the subproblem is found using an auxiliary method, such as a *primal heuristic* designed for that purpose (see Section 2.2 below). Note that when the solution to the relaxation is in $\mathcal{F}$, the pruning rule applies, since in that case, the upper and lower bounds for the subproblem are equal. For an overview on primal heuristics for MILP, see, e.g., [32, 8].

The bounds on individual subproblems are aggregated to obtain global upper and lower bounds (lines 9 and 5), which are used to avoid the complete enumeration of all (usually exponentially many) potential assignments of values to the integer variables. If a subproblem's lower bound is greater than or equal to the global upper bound (this includes the case in which the subproblem is *infeasible*, e.g., has no feasible solution), that subproblem can be pruned (line 6). The difference between the upper and lower bounds at a given point in the algorithm is referred to as the *optimality gap* and can be expressed as either an *absolute gap* (the difference itself) or as a *relative gap* (the ratio of the difference to the lower or upper bound). The progress of the algorithm is sometimes expressed in terms of how the gap decreases over time, with a final gap of zero representing the successful completion of the algorithm.

Branching. The task of *branching* in line 7 of Algorithm 2 is to successively partition the feasible region $\mathcal{F}$ into regions defining smaller *subproblems* until the individual subproblems can

either be solved explicitly or it can be proven that their feasible region cannot contain an optimal solution. This partitioning is done using logical disjunctions that must be satisfied by all feasible solutions. A solution with least cost among all those found by solving the subproblems yields the global optimum.

The *branching strategy* or *branching method* is an algorithm for determining the particular disjunction to be used for partitioning the current subproblem once the processing is done. As with the addition of cutting places, we generally require that these disjunctions be violated by the solution to the relaxation solved to obtain the lower bound. Note that part of the task of branching is to determine initial bounds for each of the created subproblems. By default, the initial bound of the child subproblem is set equal to the final bound for that of its *parent*. However, some more sophisticated branching strategies involve the calculation of more accurate initial bounds for each of the children, which can be used instead. For an overview on branching strategies for MILP, see, e.g., [3, 11].

Search. Finally, the *search strategy* is the scheme for prioritizing the subproblems and determining which to process next on line 3. The scheme typically involves a sophisticated strategy designed to accelerate the convergence of the global upper and lower bounds. This generally involves a careful balance of *diving*, in which one prioritizes nodes deep in the tree at which we are likely to be able to easily discover *some* feasible solution, and *best bound*, in which one prioritizes nodes whose feasible regions are likely to contain *high quality* solutions (though extracting those solutions may be more difficult).

Generally speaking, diving emphasizes improvement in the upper bound, while best bound emphasizes improvement in the lower bound. Once the upper and lower bounds are equal, the solution process terminates. We mentioned in Section 1 that the rate of convergence depends strongly on an effective search order. The fact that it is extremely difficult to replicate the same ordering in the parallel algorithm that would have been observed in a sequential one is one of the major impediments to achieving scalability. For an overview on search strategies for MILP, see [58] and [3].

2.2 Advanced Procedures

In addition to the fundamental procedures of *branching* and *bounding*, there are a number of auxiliary subroutines that enhance the performance of the basic algorithm. The most important such subroutines used in MILP solvers are those for preprocessing, primal heuristics, and conflict analysis, each of which we explain here briefly.

Primal Heuristics. These are algorithms that try to find feasible solutions of good quality for a given optimization problem within a reasonably short amount of time. There is typically no guarantee that they will find any solution, let alone an optimal one. General-purpose heuristics for MILP are often able to find solutions with objective values close to the global lower bound (and thus with provably high quality) for a wide range of problems; they have become a fundamental ingredient of state-of-the-art MILP solvers [9]. Primal heuristics have a significant relevance as supplementary procedures since the early knowledge of a high-quality feasible solution helps to prune the search tree by bounding and enhances the effectiveness of certain procedures that strengthen the problem formulation.

Preprocessing. These are procedures to transform the given problem instance into an equivalent instance that is (hopefully) easier to solve. The task of presolving is threefold: first, it reduces the size of the problem by removing irrelevant information, such as redundant constraints or fixed variables. Second, it strengthens the LP relaxation of the model by exploiting integrality

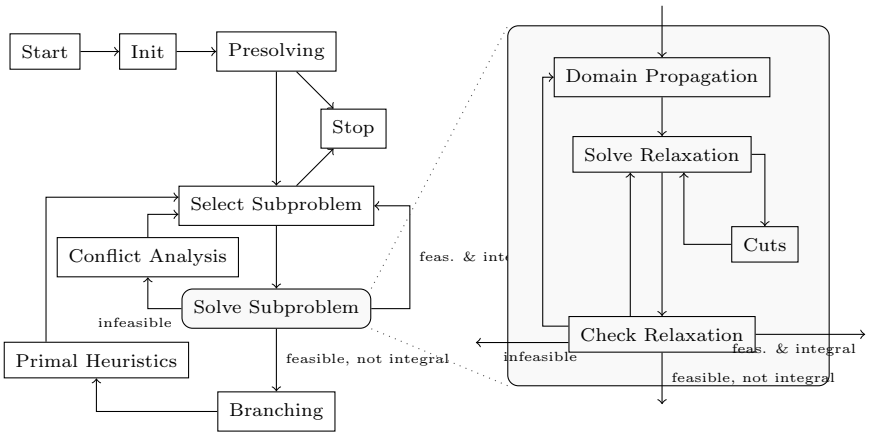

Figure 1: Flowchart of the main solving loop of a typical MILP solver

information, e.g., to tighten the bounds of the variables or to improve coefficients in the constraints. Third, it extracts information from the model that can be used later to improve the effectiveness of branching and the generation of cutting planes. For an overview on preprocessing for MILP, see, e.g., [2, 38, 60].

Conflict analysis. This analysis is performed to analyze and track the logical implications whenever a subproblem is found to be infeasible. To track these implications, a so-called *conflict graph* is constructed and gets updated whenever a subproblem is deemed to be infeasible. This graph represents the logic of how the combination of constraints enforcing the partitioning in the branching step led to the infeasibility, which makes it possible to prohibit the same combination from arising in other subporoblems. More precisely, the conflict graph is a directed acyclic graph in which the vertices represent bound changes of variables and the arcs correspond to bound changes implied by logical deductions, so-called propagation steps. In addition to the inner vertices, which represent the bound changes from domain propagation, the graph features source vertices for the bound changes that correspond to branching decisions and an artificial target vertex representing the infeasibility. Then, each cut in the graph which separates the branching decisions from the artificial infeasibility vertex gives rise to a valid conflict constraint. For an overview on conflict analysis for MILP, see, e.g., [1, 96].

Figure 1 illustrates the connections between the main algorithmic components of a MILP solver. The development of MILP solvers started long before parallel computing became a mainstream topic. Among the first commercial mathematical optimization softwares were IBM's MPS/360 and its predecessor MPSX, which were introduced in the 1960's. Interestingly, the MPS input data format, designed to work with the punch-card input system of early computers, is still the most widely supported format for state-of-the-art MILP solvers half a century later.

Today, there is a wide variety of commercial MILP solving software available, including Xpress [31], Gurobi [43], Cplex [18]. All of them providing a deterministic parallelization for shared memory systems. There are also several academic, noncommercial alternatives, such as CBC [35], Glpk [61], Lpsolve [30], SYMPHONY [74], DIP [73], and SCIP [80]. Some of these feature parallel algorithms, some do not. Every two years, Robert Fourer publishes a list of currently available codes in the field of linear and integer programming, the 2015 edition being the latest at the time of writing this book article [36].

3 Parallel Algorithms

In this section, we discuss a variety of issues that arise in designing, implementing, and measuring the performance of parallel algorithms for solving MILPs. The material here is a high-level overview and thus intentionally avoids some details that are unnecessary in describing general concepts. Parallel algorithms can be assessed with respect to both *correctness* and *effectiveness*. Generally speaking, correctness is a mathematical property of an algorithm that must be proven independent of a particular implementation, though details of the implementation and even properties of the underlying hardware may sometimes matter in proving that a particular algorithm will terminate successfully without entering a race condition or other related error caused by the parallelization itself. We don't directly address correctness here, since correctness of this class of parallel algorithms typically follows easily from correctness of an associated sequential algorithm (with some minor exceptions that we'll point out). We focus instead on how the design of a parallel algorithm informs its *effectiveness*, a general term denoting how well the algorithm performs according to some specified measure.

Naturally, it is only once an algorithm has been implemented and deployed on a particular *computational platform* that we can assess its effectiveness. The end goal of the design and implementation process is what we call a *solution platform*: the combination of an algorithm and a *computational platform* consisting of a particular architecture, communication network, storage system, OS, compiler tool chain, and other related auxiliary components. We take a rather simplistic view of the details of the computational platform, assuming that it consists of a collection of *processors* capable of executing sequential programs and an interconnection network that allows any pair of processors to communicate. The details of *how* the processors communicate and the latency involved in such communication are vitally important in practice, but beyond the scope of this overview. For our purposes, the processors can be co-located on a single *central processing unit* (CPU) and communicate through memory or be located on physically distinct computers, with communication occurring over a network. We discuss more details about architectures in Section 3.3.1, but we also recommend [6] and [44] to the interested reader for more in-depth discussion and technical definitions.

In general, a complete parallel algorithm can be viewed as a collection of procedures and control mechanisms that can be divided roughly into two groups. Some are primarily geared towards governing the parallelization strategy: these include mechanisms for moving data between processors (see Section 3.2.4) and mechanisms for determining what each processor should be doing (see Section 3.2.5). Others are primarily methods for performing one of the subtasks associated with a standard sequential algorithm listed in Section 2 (branching, bounding, generating valid inequalities, etc.). It is the control mechanism governing the parallelization strategy that we generally refer to as the *parallel algorithm*. The collection of remaining methods form the *underlying sequential algorithm*, since this latter collection of methods is generally sufficient to specify a complete sequential algorithm. Because most state-of-the-art sequential algorithms are based on some variant of tree search, a particular parallelization strategy can usually be used in tandem with any number of underlying sequential algorithms. We explore this idea in more detail in Section 3.2.1.

3.1 Scalability and Performance

By its very nature, tree search appears to be highly parallelizable—each time the successor function is applied, new tasks are generated that can be executed in parallel. By splitting the nodes in the set Q among multiple processors, it would seem that the search can be accomplished much faster. Unfortunately, there are a number of reasons why a naive implementation of this

idea has proven not to work well, especially in the case of MILPs. At a high level, this is largely due to a few important factors.

- Most modern solvers spend a disproportionate amount of time processing the shallowest nodes in the tree (particularly the root node) because this generally results in a smaller overall tree, which generally leads to smaller running times in the case of a sequential algorithm. However, it is generally not until these shallowest nodes are processed that all computing resources can be effectively utilized.
- The tree being searched is constructed dynamically and its shape cannot be easily predicted [69, 17]. On top of that, state-of-art MILP solvers generate a highly unbalanced tree in general. This makes it difficult to divide the node queue into subsets of nodes whose processing (including the time to explore the entire resulting subtrees) will require approximately the same amount of effort.
- The order in which the nodes are considered can make a dramatic difference in the efficiency of the overall algorithm and enforcing the same ordering in a parallel algorithm as that which would be observed in a highly-tuned sequential algorithm without a loss of efficiency is extremely difficult.
- Most state-of-the-art solvers generate a wide range of information while searching one part of the tree that, if available, could inform the search in another part of the tree.
- The combination of the above two points means that it's generally desirable/necessary to move large amounts of data dynamically during the algorithm, but this is costly, depending on the architecture on which the algorithm is being executed.

It is not difficult to see that there are several important trade-offs at play. On the one hand, we would like to effectively share data in order to improve algorithmic efficiency. On the other hand, we must also limit communication to some extent or the cost of communication will overwhelm the cost of the algorithm itself. There is also a trade-off between time spent exploring the shallowest part of tree and how quickly the algorithm is able to fully utilize all available processing power.

3.1.1 Scalability

Generally speaking, *scalability* is the degree to which a solver platform is able to take advantage of the availability of increased computing resources. In this article, we focus mainly on the ability to take advantage of more processors, but the increased resources could also take the form of additional memory, among other things. Scalability is a property of an entire solution platform, since scalability is inherently affected by properties of the underlying computational platform. Most parallel algorithms are designed to exhibit good performance on a particular such platform, but we do not attempt to survey the properties of platforms affecting scalability in this article and rather focus mainly on the properties of the algorithms themselves. For a good general introduction to computational platforms, see [44] and [6] (also see Section 3.3.1 for a brief summary). In the remainder of this section, we briefly survey issues related to scalability of algorithms.

Phases. The concept of algorithm phases is an important one in parallelizing tree search algorithms. Conceptually, a parallel algorithm consists of three phases. The *ramp-up phase* is the period during which work is initially partitioned and allocated to the available processors. In our current setting, this phase can be defined loosely as lasting until all processors have been assigned at least one task. The second phase is the *primary phase*, during which the algorithm operates in

steady state. This is followed by the *ramp-down* phase, during which termination procedures are executed and final results are tabulated and reported. Defining when the ramp-down phase begins is slightly problematic, but we define it here as the earliest time at which one of the processors becomes permanently idle.

In tree search, a naive parallelization considers a "task" to be the processing of a single subproblem in the search tree. The division of the algorithm into phases is to highlight the fact that when parallelized naively, a tree search algorithm cannot take full advantage of available resources during the ramp-up and ramp-down portions of the algorithm without changing the granularity of the tasks (see Section 3.2.2). For some variants of the branch-and-bound algorithm and for instances with large size or particular structure, the processing of the root subproblems and immediate descendants can be very time-consuming relative to the time to search deeper parts of the tree, which results in lengthy ramp-up and ramp-down periods. This can make good scalability difficult to achieve.

Sequential Algorithm. Although the underlying sequential algorithm is often viewed essentially as a black box from the viewpoint of the parallel algorithm, it is not possible in general to parallelize a sequential algorithm without inherently affecting how it operates. Most obviously, the parallel algorithm generally re-orders the execution of tasks and thus changes the search strategy. However, other aspects of the parallel algorithm's approach can affect the operation of the sequential parts in important ways. The parallel algorithm may even seek to control the sequential algorithm to make the combination more scalable. The degree to which the parallel algorithm can control the sequential algorithm and the degree to which it has access to information inside the "black box" has important implications.

Overhead. Scalability of parallel MILP solvers is often assessed by measuring the amount of *parallel overhead* introduced by the parallelization scheme. Roughly speaking, parallel overhead is the work done in parallel that would not have been performed in the sequential algorithm. It can be broken down into the following general categories (see [52]).

- *Communication overhead*: Computation time spent sending and receiving information, including time spent inserting information into the send buffer and reading it from the receive buffer. This is to be differentiated from time spent waiting for access to information or for data to be transferred from a remote location.

- *Idle time (ramp-up/ramp-down)*: Time spent waiting for initial tasks to be allocated or waiting for termination at the end of the algorithm. The ramp-up phase includes inherently sequential parts of the algorithm, such as time spent reading in the problem, processing the root node, etc., but also the time until enough B&B nodes are created to utilize all available processors. The ramp-up and ramp-down time is highly influenced by the shape of the search tree. If the tree is "well balanced" and "wide" (versus deep), then both ramp-up and ramp-down time will be minimized.

- *Idle time (latency/contention/starvation)*: Time spent waiting for data to be moved from where it is currently stored to where it is needed. This can include time waiting to access local memory due to contention with other threads, time spent waiting for a response from a remote thread either due to inherent latency or because the remote thread is performing other tasks and cannot respond, and even time spent waiting for memory to be allocated to allow for the storage of locally generated data.

- *Performance of redundant work*: Time spent performing work (other than communication overhead) that would not have been performed in the sequential algorithm. This includes

the evaluation of nodes that would not have been evaluated with fewer threads. The primary reason for the occurrence of redundant work is differences in the order in which the search tree nodes are explored. In general, one can expect that the performance of redundant work will increase when parallelizing the computation, since information that would have been used to avoid the enumeration of certain parts of the search space may not yet have been available (locally) at the time the enumeration took place in the parallel algorithm. However, it is entirely possible that the parallel algorithm will explore fewer nodes in some cases.

Effective parallelization is about controlling overhead, but how best to do this can vary depending on a wide range of factors and there is no "one-size-fits-all" solution. Properties of the search algorithm itself, properties of the instances to be solved, and properties of the architecture on which the algorithm will be deployed all play a role in determining what approach should be taken. This is one of the reasons why we see such a wide variety of approaches when we consider solvers in the wild. Evidence of this is presented in Section 4.

3.1.2 Performance

It is important to point out that scalability, also called *parallel performance*, is *not* the same as overall "performance." As we have already noted, parallelization of more sophisticated sequential algorithms is inherently more difficult in terms of achieving scalability than parallelization of a less sophisticated algorithm. Moreover, it could be the case that a solver platform exploiting a more sophisticated underlying sequential algorithm, although not as scalable when parallelized, would nevertheless outperform a more scalable but less sophisticated solver platform on the usual metrics used to assess overall effectiveness and solution power, such as wall clock solution time or ability to solve difficult instances. The fact that there is a performance trade-off between scalability and overall performance is one of the fundamental difficulties in determining how to measure performance and in assessing progress in this challenging field of research. We discuss measures of performance in much greater detail in Section 5.

3.2 Properties

In this section, we provide some means by which to classify the existing algorithms. This is certainly not the first attempt to do such classification. Already in 1994, [39] provided a very thorough classification for parallel branch-and-bound algorithms. An updated survey was later published in [19]. By now, much has changed and it is no longer possible to provide a classification of existing solvers in the traditional sense of a partitioning into subsets taking similar approaches. Modern solvers vary along many different axes in a complex design space. No partition of algorithms into subsets based on fundamental properties will likely be satisfactory or if so, the subsets will have cardinality one. We therefore simply list basic properties of existing algorithms that we refer to in Section 4 when describing existing algorithms and software.

3.2.1 Abstraction and Integration

The separation between the parallel and sequential parts of the algorithm discussed in the previous part need not be as clean as we have indicated in our discussion in Section 3.1.1. In some cases, there is a tight integration of parallel and sequential parts into a single monolithic whole. In other cases, the parallel algorithm is completely separated from the underlying sequential algorithm or even encapsulated as a *framework*, whose overall approach does not depend on the details of the sequential algorithm. To make the clean separation of the parallel algorithm and the underlying

sequential algorithm possible, the parallel algorithm must have a high level of *abstraction* and a low level of *integration* with the sequential solver. The concept of abstraction is similar to that which forms the basic philosophy underlying object-oriented programming languages, but we distinguish here between two different types of abstraction: *algorithmic abstraction* and *interface abstraction*.

To exemplify what we mean by *algorithmic abstraction*, consider Algorithm 2 once again. This tree search algorithm is stated at a high level of *algorithmic abstraction* because the internal details of how the bounding, branching, and prioritization methods work are not specified and the algorithm does not depend on these details. Naturally, any concrete implementation of this algorithm must specify these elements, but the algorithm as stated constitutes an *abstract framework* for branch and bound that would be agnostic with respect to the internal implementation of these algorithmic elements.

Of course, there is some interaction between the unspecified elements of the underlying elements of the branch-and-bound algorithm and the abstract framework. These constitute the algorithmic *interface*, by which the framework interacts with the components of the underlying algorithm through the production of outputs in a specified form. For example, the bounding method on line 4 in Algorithm 2 is required to produce a lower bound that the framework may then use later on line 3 to determine the search order or to prune nodes on line 6. The framework needs not know *how* the bound is produced, it only requires the underlying sequential methods produce it *somehow*. This may be done, for example, through a function call to the underlying sequential algorithm that encapsulates the processing of a single node in the search tree.

The flexibility and level of access to internal parts of the sequential algorithm that are present in a particular sequential solver's API determines the degree to which the parallel solver is able to control various aspects of the underlying sequential algorithm. Lack of ability to execute certain internal procedures independent of the sequential algorithm may limit the options for parallelization. In some case, the only access to the sequential solver's functionality is by limited execution of the sequential solution algorithm, as a "black box", for solution of subproblems associated with nodes in the search tree. In other cases, fine-grained access to individual procedures, such as those for bounding and branching of individual nodes in the search tree is available.

A related concept is that of *implementational abstraction*, which is a more practical measure of the degree to which the *implementation* (in the form of source code) of a parallel search algorithm is customized to work in tandem with a particular *implementation* of an underlying sequential solver. This has primarily to do with whether or not the interface between the parallel and sequential parts is well-defined and "generic." In general, frameworks intended to work with multiple underlying sequential solvers would need to have high degrees of both implementational and algorithmic abstraction. When there is a tight integration between the parallel and sequential parts (especially when they are produced as a monolithic whole), the level of implementational abstraction is often low, but this does not mean that the level of algorithmic abstraction is low.

Implementations with a low degree of abstraction depend on internal interfaces that may either not be well-defined or depend on the passing of data that only a solver with a particular internal implementation could produce. Implementations with a high degree of abstraction interact with the underlying sequential solver only through well-defined interfaces that only require the passing of data that almost any sequential algorithm should be able to produce. How those interfaces work and at what points in the sequential algorithm the solver is required to pass information depends on the level of granularity of the solver.

With an abstract framework, such as the one specified in Algorithm 2, parallelization becomes much easier, since we can conceptually parallelize the abstraction without dependence on the unspecified internals of the various algorithmic elements. A simple way of parallelizing Algorithm 2

is just to maintain set Q and global bounds L and U as global data, but allow multiple subproblems to be simultaneously bounded (this is one variant of the Master-Worker coordination scheme described in Algorithms 5 and 6). In other words, each processor independently executes the same processing loop, but all update the same global bounds, access the same common global queue (Q) of subproblems at the top of the loop, and insert into the same global queue any newly produced subproblems. As a practical matter, this requires the avoidance of conflicts in writing and reading global data, which is one primary source of overhead, but as long as this can be achieved, the convergence of the algorithm in parallel follows from the same principles that guarantee convergence of the sequential algorithm.

3.2.2 Granularity

One way in which we can differentiate algorithmic approaches is by their *granularity*. Roughly speaking, the granularity refers to what is considered to be the atomic unit of work in the algorithm. We consider four primary levels of granularity, but of course, there are many gradations and most algorithms either fall somewhere between these levels or may even take different approaches in different phases of the algorithm. Ranked from coarse to fine, they are *tree parallelism*, *subtree parallelism*, *node parallelism* and *subnode parallelism*.

Tree parallelism. Several trees can be searched in parallel using different strategies and the knowledge generated when building one tree can be used for the construction of other trees. In other words, several tree searches are looking for solutions in the same state space at the same time. Each tree search algorithm can take a different approach, e.g., different successor function, search strategy, etc. For example, Pekny [70] developed a parallel tree search algorithm for solving the Traveling Salesman Problem in which the trees being built differed only in the successor functions (branching mechanism). A brute-force strategy for parallelization is to concurrently start single-threaded solves of slightly perturbed (but mathematically equivalent) copies of the same MILP problem. One implementation of this is the racing scheme described in Algorithm 3

Subtree parallelism. Multiple subtrees of the same overall search tree may be explored simultaneously, but independently (without sharing information). This could be accomplished, for example, by passing a single subproblem to a sequential solver and executing the sequential algorithm for some fixed amount of time. Most of the more advanced algorithms described later in this article use some form of subtree parallelism to avoid the overhead associated with more fine-grained approaches.

Node parallelism. A single search tree can be searched in parallel by executing an algorithm similar to the sequential one, but processing multiple nodes simultaneously, with a centralized control mechanism of some sort and more information sharing. A straightforward implementation would utilize a master process to coordinate the search, distribute nodes from the queue and collect results. This is the most widely used type of parallelism for tree search algorithms and is reflected in the Master-Worker coordination scheme described in Algorithms 5 and 6.

Subnode parallelism. The parallelism may be introduced by performing the processing of a single node in parallel. This can be done effectively with decomposition-based algorithms, such as column-generation, for example.

- **Strong branching**: A natural candidate for parallelization is the so-called strong branching approach to selecting a branching disjunction in which some pre-computations are done to evaluate the potential effectiveness of each candidate. Similarly, probing or domain propagation techniques, which also involve pre-computations to simplify a given subproblem improve the efficiency of later bounding computations, might also be parallelizable [41].

- **Solution of LPs**: All solution algorithms for solving MILPs rely fundamentally on the ability to quickly solve sequences of LPs, usually in the context of deriving a bound for a given subproblem. The barrier algorithm for solving the initial LP relaxation in the root node, always one of the first steps in any algorithm for solving an MILP, can be parallelized effectively. Recently, there has also been progress in the development of a parallel implementation of the dual simplex algorithm, the most commonly used algorithm for solving the LP relaxations in non-root nodes, due to its superior warm-starting capabilities (see [46]).

- **Heuristics**: Primal heuristics can often be executed independently of the search itself. Hence, they can be used to employ processor idle time to best effect. Furthermore, multiple expensive heuristics, such as large neighborhood search algorithms, can be run concurrently or can themselves be parallelized.

- **Cutting**: During bounding of individual search tree nodes, the relaxations are strengthened by executing additional sub-procedures, called *separation algorithms*. Multiple such algorithms can be executed simultaneously to achieve faster strengthening.

- **Decomposition**: If a problem naturally decomposes due to block structure in the matrix A, the problem can be solved as a sequence of completely independent subproblems. Even when the matrix A does not have perfect block structure (it only decomposes after removing some rows or columns), decomposition methods, such as Dantzig-Wolfe [21], can be employed, which enable parallelization of the bounding procedure.

3.2.3 Adaptivity

Another general property of algorithms is the degree to which they are *adaptive*. All algorithms must be adaptive to some extent. For example, most algorithms switch strategies from phase to phase, e.g., initially pursuing tree level parallelism for exploring shallow parts of multiple trees during the ramp-up phase, then switching to subtree or node parallelism during the primary phase. In general, an algorithm for solution of MILPs could be changed adaptively as the solution process evolves based on many different possible aspects of the underlying algorithm, such as the global upper and lower bounds, the distribution of subproblems across processors, the overall "quality" of the remaining work (see Section 3.2.5) and others. When, how and which part of algorithms are adapted at run-time is a crucial aspect of performance.

3.2.4 Knowledge Sharing

Knowledge is the data generated during the course of a search, such as solutions, valid inequalities, conflict information, bounds, and other "globally valid" data generated as a by-product of the search in one part of the tree that may inform the search in another. Such knowledge can be used to guide the search, e.g., by determining the order in which to process available nodes. *Global knowledge* is knowledge that is valid for all subproblems in the search tree. *Local knowledge* is knowledge that is only valid with respect to certain subproblems and their descendants. The general categories of knowledge to be shared in solving MILPs includes the following.

- **Bounds**: Global upper bounds, as well as the bounds of individual nodes in the search tree can be shared.

- **Nodes**: Descriptions of the search tree nodes themselves are data that can (and usually must) be shared.

- **Solutions**: Feasible solutions from one part of the tree may help with computations in another part.
- **Pseudocost estimates**: So-called *pseudocost estimates* are used to predict the effectiveness of branching disjunctions based on historical statistics.
- **Valid inequalities**: Valid inequalities used to strengthen relaxations in one part of the tree may be useful in strengthening relaxations in another part of the tree.

With knowledge about the progress of the search, the processes participating in the search are able to make better decisions. When knowledge about global bounds, for example, is shared among processes, it is easier to avoid the performance of redundant work and the parallel search can be executed in a fashion more similar to its serial counterpart. If all processes had complete knowledge of all global information, then no redundant work should be performed at all, in principle.

Despite the obvious benefits of knowledge sharing, there are several challenges associated with it. Most obviously, increased sharing of knowledge can have significant impact on parallel overhead. Communication overhead and a possible increase in idle time due to the creation of communication bottlenecks are among the inherent costs of sharing knowledge. Increases in redundant work, on the other hand, is the main cost of *not* sharing knowledge. This highlights a fundamental trade-off: sharing of knowledge reduces the performance of redundant work and can improve the effectiveness of the underlying sequential algorithms, but these benefits come at a price. The goal is to strike the proper balance in this trade-off. Trienekens and Bruin [90] give a detailed description about of the issues involved in knowledge generation and sharing.

3.2.5 Load Balancing

Load balancing is the method by which information describing the units of work to be done is moved from place to place as needed to keep all processors busy with useful work. It is characterized by how often it is done, what information must be moved (mainly a product of what are considered to be the atomic tasks), and whether it is done statically (just once at the beginning) or dynamically (throughout the algorithm).

Static load balancing. The first task of any parallel algorithm is to create enough units of work to ensure all processors can be employed with useful work as quickly as possible. The goal is to make the initial work distribution as even as possible in order to avoid having to either re-balance later or absorb the overhead resulting from the idling of processors whose work is finished quickly.

In a pure static load balancing scheme, each processor works independently on its assigned tasks following the initial distribution, reporting final results at the end. If initial work distribution is uneven, there is no recourse. Such a scheme requires very little communication, but can result in the performance of a vast amount of redundant work, as well as a large amount of idle time due to the fact that the tasks assigned to one processor may be much less time-consuming than the tasks assigned to another processor. Because it is extremely challenging to predict the difficulty of any given subproblem, static load balancing on its own is usually not very effective, though a well-designed static load balancing scheme has proven effective [78] in some cases.

In an early paper, Henrich [45] summarized four categories of initialization methods and describes the advantages and disadvantages of each. We repeat these here to give a flavor for these methods.

- **Root initialization**: When applying this method, one process processes the root of the search tree and creates children according to a branching scheme that may or may not be

the same as the one used during the primary phase. These children of the root are then distributed to other processors and the process is continued iteratively until all processors are busy. Root initialization is the most common approach due to the fact that it is easy to implement. For example, one can use the same branching scheme and mechanisms for distributing nodes to idle processors that are use during the primary phase of the algorithm. The method is most effective when the time to process a node is short and the number of children created during branching is large, so that all processors are engaged in work as early as possible. A major shortcoming of root initialization is that when the processing times are not short or when the number of available processors is large, many of the processes are *idle* while waiting to receive their allocation of nodes.

- **Enumerative initialization**: This method broadcasts the root node to all processes, which then performs an initial tree search according to the sequential algorithm. When the number of leaf nodes on each processor is at least the number of processes, processes can stop expanding. The i^{th} process then keeps the i^{th} node and deletes the rest. In this method, all processes are working from the very beginning and no communication is required. On the other hand, there is redundant work because each process is initially doing an identical task. Note also that it is crucial that all processors generate identical search trees (this requires determinism of the sequential algorithm, see Section 3.2.7). This method has been successfully implemented in PEBBL [29].

- **Selective initialization**: This method starts with the broadcasting of the root node to each process. Each process then generates one single path from the root (without generating any others). The method requires little communication, but requires a sophisticated scheme to ensure processes work on distinct paths. It also requires determinism as mentioned above.

- **Direct initialization**: This method does not build up the search tree explicitly. Instead, each process directly creates a node from a certain depth of the search tree. The number of nodes at this depth should be no less than the number of processes. This method require little computation and communication, but it works only if the structure of the search tree is known in advance.

More sophisticated schemes have been implemented recently that build on these basic ideas, such as the *racing ramp-up* of the UG framework [86, 92] (described in Section 3.3.3) or the *spiral* and *two-level root initialization* schemes of the CHiPPS framework [103, 98].

Dynamic Load Balancing. Although static load balancing is effective in certain cases, the uneven processor utilization, processor speed, memory size, and the change of tree shape due to the pruning of subtrees can make the workloads on processes gradually become unbalanced, especially in distributed computing environments. This necessitates the need for dynamic load balancing, which involves reallocating workload among the processes during the execution of a parallel program.

Dynamic load balancing has been studied in many computational contexts and the literature abounds with surveys of various techniques that have been implemented ([23] and [95] are but two thorough and relevant such surveys). Load balancing in general parallel tree search generally addresses the need to distribute the quantity of work evenly among the processors (or at least to keep all processors busy). The challenge of tree search for solving optimization problems is that not all work in the current queue has the same priority. When work of higher priority is produced (new nodes are generated through branching), this work should preempt the currently existing lower-priority work. Unfortunately, the distribution of high-priority work can (and is very likely to) become uneven through the natural course of the algorithm. Even when all processors

are busy, some of the work being done may not be useful (it will turn out to be redundant in hindsight). For this reason, load balancing needs to consider both *quality* and *quantity* of work when redistributing workload. Following are the definitions of quality and quantify of work:

Definition 1. *The* quality of work *is a numeric value to measure the possibility that the region to be explored (represented by a node or a set of nodes) contains* high quality *solutions.*

Definition 2. *The* quantity of work *is a numeric value to measure the* amount *of work. For instance, this could be the estimated number of nodes to be processed in a subtree.*

The relative quality (and also quantity) of existing work may change as global information is received. Furthermore, because it is far more efficient to process a node in the search tree directly following the processing of its parent on the same processor due to the ability to warm start the computation, one must be careful not to be overly aggressive in balacing the load.

For these reasons, load balancing in optimization applications can be more challenging than in many general tree search applications. This is certainly not a new observation in the context of parallel optimization and was discovered early during the development of parallel algorithms for branch and bound [26, 27].

In general tree search, a vast number of methods have been proposed to dynamically balance workloads, many of which have been focused on tree-based computations [53, 68, 77, 79, 88]. Despite the wide array of work on this topic, most schemes, even when described as very general, have been targeted at particular problem classes or particular architectures and are best-suited for these use cases. Few truly general-purpose schemes have been proposed for the simple reason that requirements vary dramatically for different applications and algorithmic approaches.

Broadly, load balancing strategies can be categorized based on the degree of centralization of the mechanism and based on whether the balancing is initiated by a separate manager process, by the sender, or by the receiver. Centralized schemes involving one or more processors that act as managers, tracking workload and directing transfers are generally referred to as *work-sharing* schemes, whereas as schemes in which transfers are initiated by individual processors with either surplus or deficit workload are referred to as *work-stealing*.

Kumar, *et al.* [53] studied a number of early dynamic load balancing schemes, such as *asynchronous round robin*, *nearest neighbor*, and *random polling*, and performed extensive scalability testing. We briefly describe these schemes here so as to provide a flavor of how such schemes work.

- In an asynchronous round robin scheme, each process maintains an independent variable *target*, which is the identification of the process to ask for work, i.e., whenever a process needs more work, it sends a request to the process identified by the value of the target. The value of the target is incremented each time the process requests work. Assuming P is the number of processes, the value of target on process i is initially set to $(i+1)$ modulo P. A process can request work independently of other processes. However, it is possible that many requests are sent to processes that do not have enough work to share.

- The nearest neighbor scheme assigns each process a set of neighbors. Once a process needs more work, it sends a request to its immediate neighbors. This scheme ensures locality of communication for requests and work transfer. A disadvantage is that the workload may take a long time to be balanced globally.

- In the random polling scheme, a process sends request to a randomly selected process when it needs work. The possibility of selection of any process is the same. Although it is very simple, random polling is quite effective in some applications [53].

There are many important tradeoffs encapsulated in the load balancing scheme. The work units themselves can be considered to be "knowledge" of a certain type that can be shared either more or less aggressively. More aggressive balancing will result in less idling and less redundant work, but also increased overhead, much as in the case of sharing of other types of knowledge. Modern solvers generally employ more sophisticated, adaptive schemes and these will be described in Section 4.1.

3.2.6 Synchronization and Coordination

Synchronization is a requirement that at some step in a parallel algorithm, a set of processes needs to proceed simultaneously from a known state for reasons of either correctness or efficiency. One common purpose of synchronization is to enforce *determinism* (see Section 3.2.7), but the need for synchronization arises in many applications for a variety of reason. It is a natural requirement when parallelizing an algorithm that was originally designed to be sequential and sequential algorithms often aggregate results obtained in a distributed fashion at intermediate points.

Synchronization can be achieved either by using a *barrier* or by some kind of counting scheme [94]. A barrier is a mechanism that prevents processes from continuing past a specified point in a parallel program until certain other processes reach this point. The substantial downside of introducing synchronization is that some processes might reach the synchronization point much more quickly than others and will waste time in waiting for other processes to reach the same state. From the standpoint of parallel scalability, synchronization introduces overhead and overhead leads to a loss of performance. Therefore, synchronization is to be avoided if possible.

The most obvious need for synchronization arises from the need to compute accurate global bounds. It is evident that if Algorithm 2 were to be parallelized straightforwardly in an asynchronous fashion, computing accurate global bounds would not be possible. This is because the global lower bound requires minimizing over the terminal nodes (those without children) in the current search tree, but the set Q doesn't always contain all terminal nodes. In the sequential algorithm, this problem doesn't arise because the bound is only calculated at a time when we *know* that set Q *does* contain all terminal nodes. In parallel, however, some form of synchronization would be required.

Fortunately, the maintenance of accurate global lower and upper bounds is not a strict requirement in parallel branch-and-bound. An upper bound is only needed for applying pruning rules and certain other advanced functionality, but any provably valid upper bound will suffice. It is not necessary (though it is generally desired) for a global upper bound to be accurately computed and known to all processes. Likewise, knowledge of the global lower bound is not necessary either except possibly as a measure of progress. Application of pruning rules only requires the lower bound computed for an individual subproblem to be accurate.

A related, though different, phenomena is the introduction of *coordination* that may introduce communication bottlenecks due to the requirement that an algorithm *coordinate* actions. Coordination does not always require strict synchronization, but it may introduce points at which one process must wait for a reply from another one. Load balancing is an obvious example of coordination, but there are many other possibilities. In general, coordination leads to overhead. At the same time, coordination can also improve performance of the underlying sequential algorithm, so this introduces a fundamental trade-off and a balance that must be struck. We discuss coordination schemes used in modern solvers in Section 3.3.3.

The appropriate amount of synchronization and coordination has to do with the relative cost of communication on the computational platform on which the algorithm is to be deployed. An asynchronous execution mode is appropriate for algorithms running on networks of workstations and computational grids, where synchronization is hard to achieve. A major issue with asyn-

chronous algorithms is that there is typically no process that has accurate information about the overall state of the search. This can cause difficulties in effectively balancing workload and detecting termination. Load balancing and termination detecting schemes should have the ability to deal with such inaccurate information issue.

3.2.7 Determinism

A deterministic solver platform is one that is guaranteed to perform the same operations and to produce the same end result when provided with the same input. Although all valid parallel algorithms should produce a valid result in the end, it is often the case that more than one valid result is possible (due to alternative optimal solutions) and even if not, the intermediate computations done to achieve the result can vary substantially. Small variations in the timing of the discovery of intermediate feasible solutions may lead to small difference in the global upper bounds that get applied in the pruning step of an algorithm, leading to a different search tree being produced. Even tiny differences in the execution path in the beginning stages of a parallel algorithm can lead to very large differences in the overall execution and can even cause changes in running time of an order of magnitude or more. Non-determinism is even easily possible in the case of a sequential algorithm. For example, some algorithms depend on the generation of random numbers. It's also possible for tie-breaking to occur differently on different computational platforms, depending on how memory is allocated.

It should be clear then that ensuring an algorithm executes deterministically, even in the sequential case, requires careful attention to detail. In the parallel case, this is not only difficult, but inevitably requires some kind of synchronization and control of when inter-process communication happens. Thus, determinism does also lead to a degradation in performance.

Further, it is not entirely clear what determinism *means* in the parallel case. In the sequential case, determinism means performing the exact same atomic operations in the exact same sequence (putting aside possible reordering of operations by hyper-threading and such). In the parallel case, we generally cannot hope to ensure that individual atomic operation are performed in precisely the same order(*strong deterministic parallelism*), but rather must make a similar requirement that allows for different ordering at a low level with order preserved at a higher level (*weak deterministic parallelism*) [67]. The precise definition is implementation-dependent, but one might require for example, that the exact same search tree be explored with the exact same set of subproblems and the exact same relaxations solved at each step to produce the exact same sequence of bounds and thus the exact same end result. Accomplishing this requires not only synchronization, but also awareness that computations may vary on different processing elements on a single computational platform.

3.3 Implementation

In this section, we discuss some of the algorithmic issues that arise mainly in the implementation phase when the conceptual algorithm is translated into a computer program, compiled on a given computational platform, and deployed.

3.3.1 Platform

Although it is possible for algorithms to be conceived independent of the computational platform on which they are to be run, design must, to some extent, be informed by one more target platforms. We have so far discussed a number of important tradeoffs, such as the fact that the sharing of knowledge may both *reduce* the total amount of computation time spent executing the parts of the algorithm associated with the underlying sequential algorithm while *increasing*

the parallel overhead. We have the same sort of tradeoff with regards to the frequency of load balancing. Determining the appropriate compromise between the two sides of such tradeoffs is only possible in light of a particular platform. It is only with specific knowledge of the platform that particular hardware parameters, such as the communication latency, can be known.

Naturally, the development of a solver is time-consuming and one would therefore like to ensure effectiveness across as wide a variety of platforms as possible. To a certain extent, this can be done through parameterization. We specify certain parameters, such as the frequency of load balancing, and then tune them at run-time, based on collected statistics. This works fine for relatively small variations occurring between platforms, but for larger variations, design decision must be made that inherently limit the algorithm to certain classes of platform.

Communication Network. Perhaps the most important property of the platform that needs to be taken into account is the nature of the underlying communication network. A full treatment of the properties of communication networks is well beyond the scope of this article. Roughly speaking, we can characterize the network by its *latency* and its *topology*. The former is a measure of how long it takes data to travel between specific pairs of processors and the latter concerns the general physical layout of the network. These two are connected in that the expected latency between a pair of processors has largely to do with their relative physical locations.

We generally divide platform into two broad categories: *shared memory* and *distributed memory*, though the distinction is not as fine on modern architectures as it was historically. In a shared memory architecture, all processors have access to a common memory and data does not need to be physically moved in order for different cores to make use of it. In such an architecture, latency is, in principle, negligible. In distributed memory architectures, on the other hand, processors have physically memories and one must account for the non-negligible time it takes to move data from the memory of one processor to the memory of another.

In modern practice, one finds that these distinctions are rather blurred at best. With respect to shared memory architectures, modern CPUs have multiple computing *cores*, which, although they share a physical address space, may or may not share an actual physical memory. Further, a single computing device may have multiple multi-core CPUs that communicate over a bus. There may be significant differences in the latency for passing data between pairs of cores on the same CPU versus on different CPUs, although from the point of view of the communication abstraction at the level of the programming interface, these things appear identical. To make matters worse, even access to different data in the same memory from the same individual cores may have different latencies due to the existence of a complex memory hierarchy in which the kernel attempts to cache data that is predicted to be needed in the near future. This exacerbates the imbalance in the time to access the same data from different cores. All of this may be more or less invisible to the programmer except in as much as there are ways to indirectly influence the communication patterns. This topic is also beyond the scope of the present article. For more information, see [44].

Shared memory architectures also present the very real issue of *memory contention* in which multiple processors attempt to access the same physical memory simultaneously or to use the same physical channel for retrieving such data. Contention results in longer memory access times than would otherwise be expected and may also necessitate the use of *locks*. A *lock* restricts the access of certain memory to only a single physical processor until that lock is released. The use of locks is necessary in some situations to prevent run-time error conditions, but also contributes to contention and leads to effective reductions in the time required to access data.

All modern CPUs have multiple cores and most modern computers have multiple such CPUs. A distributed memory architecture generally consists of multiple computers connected by a communication network. Because each of these computers is a miniature parallel computer in

its own right, "pure" distributed architectures have essentially ceased to exist. For efficiency, all algorithms that target distributed memory architectures must thus account for the massive differences in the time to communicate data between cores on the same physical computer versus cores on different physical computers.

Taking account of the major differences in performance between the different possible types of computers discussed above can lead to major differences in the high-level design of parallel algorithms, as we discuss below in Section 3.3.3.

Communication Protocol. Aside from the many variations in architecture that we have just discussed, algorithm design may also be informed by the chosen *communication protocol*, the mechanism by which data is transferred from one processor to another. Although protocols are usually closely associated with some underlying physical transfer mechanism. What we refer to as the "communication protocol" here is the *interface* used by the programmer to cause the data transfer programmatically.

Very broadly, there are two categories of communications protocols: *threads* and *message passing*. The threads model is generally associated with shared memory architectures and involves communication that occurs by passing data through local memory. Message passing is generally associated with distributed memory architectures and and involves communication that occurs by passing data across a network. These two categories of protocol are in turn associated with the two basic ways of achieving parallelism programmatically: multi-threaded computation versus multi-process computation. A *process* presents the execution of a single computer program with its own private memory address space. A process can spawn multiple *threads*, all of which may execute sequential procedures simultaneously with access to the same memory.

The importance of both the communication protocol and the associated programming model and architecture is both that it can influence the efficiency of data transfer and that, merely by restrictions in the interface itself, limit the algorithmic options available. The most prevalent communication protocols at the time of this writing are:

- Threads
  - OpenMP: an interface standard for providing instructions in source code that allow for the semi-automatic parallelization of *multi-threaded* applications by OpenMP-aware compilers using communication through shared memory.
  - pthreads: an interface standard for allowing communication between individual threads of a single process running on a shared-memory computer.
- Message Passing
  - MPI: an interface standard and a set of associated libraries for allowing separate processes running either on the same computer or on remotely located computers to communicate with each other either through shared memory or over an associated communication network (the precise conduit depends on the details of the implementation of the MPI library used).
  - PVM: A much older and no longer commonly used message passing protocol similar to MPI.

Programming Language. As with communication protocols, the choice of programming language can also heavily influence both the options for parallelization and the design of a given algorithm. Many newer high-level languages (e.g., Go) includes parallel constructs directly in the language. Older, low-level languages, such as C, do not include direct support for parallelization.

On the other hand, low-level languages tend to be the choice for implementation of numerical algorithms for well-known efficiency reasons.

3.3.2 Frameworks and Solvers

In Section 3.2.1, we described the level of abstraction of an algorithm as a fundamental property. At the level of an actual implementation, an abstract parallel algorithm, in which the parallel algorithm does not depend on the details of the associated sequential algorithm except through well-defined interfaces, can be implemented completely independent of the sequential solver and can even be made to work with multiple sequential solvers. The implementation of a parallel algorithm in such a way as to enable any sequential algorithm (with possible slight modification to conform to the interface) to work within the scheme specified by the parallel algorithm is called a *framework*. The combination of a parallel framework and (the implementation of) a particular sequential algorithm make a *solver* (which when deployed on a particular platform becomes a *solver platform*). Naturally, it is not necessary for a solver to utilize a framework in order to execute in parallel. In some cases, the solver includes its own parallel algorithm in a tightly integrated package. We provide examples of both solvers and frameworks in Section 4 below.

3.3.3 Coordination Mechanisms

In this section, we review what we generally refer to as *coordination mechanisms*. This is a general term for the overall way in which the parallel algorithm controls execution, including both static and dynamic load balancing and the interaction with the underlying sequential solver. We review here the most common existing mechanisms employed by the solvers and frameworks discussed in Section 4.1.

It should be highlighted that in all the approaches reviewed below that involve dynamic load balancing, the adaptive tuning of granularity is a centrally important concept. Generally speaking, the atomic unit of work that we consider is the exploration of a subtree according using the strategy of the underlying sequential algorithm, but with a work limit imposed. This work limit is generally imposed in the form of a limit on the execution time or a limit on the number of nodes enumerated. Depending on the specific details of how the parallel and sequential solvers interact (through a restrictive API or by direct internal function calls with access to the solver's internals), the work limit may be imposed in different ways.

Parallel Racing. Despite the decades of effort that have resulted in increasingly sophisticated sequential solution platforms, current state-of-the-art MILP solvers still have a high *performance variability*, which means that the impact on performance of seemingly performance neutral changes in the input or in minor implementation details of the solver can sometimes result in large variations in solution time and other performance measures. As the most striking example of this kind of variation, simply permuting the rows or columns of the constraint matrix, which yields an identical instance from a mathematical standpoint, can cause even the most sophisticated solvers to vary wildly [51]. It is impossible to predict, for a particular instance, what perturbations to the model or to the solver's run-time parameters will minimize running time.

Given this situation, one of the most straightforward ways to parallelize an existing sequential algorithm is simply to exploit this performance variability by executing the same sequential algorithms either with different parameter settings or with different permuted instances of the same MILP (or both) in parallel. We call this approach *parallel racing*, since it executes a simple race among the solver instances with no communication. The computation is terminated when the first solver finishes and a winner is declared. The idea dates back to the early days of the

development of parallel branch-and-bound algorithms [70, 63, 47] and has been recently shown to be surprisingly effective in some cases [33].

This approach has some obvious advantages. It is simple to implement and can be used easily with any sequential solver or combination of different sequential solvers and thus can capitalize on all available sequential solution technology. It requires no coordination of the solver instances and thus reduces parallel overhead to nearly zero. The cost of this simplicity, of course, is that the algorithm may perform a potentially vast amount of redundant work. By passing *some* small amount of data between the solvers, the basic procedure can be easily improved. Algorithm 3 shows a simplified racing-type algorithm in which global upper bounds are communicated during computation. Naturally, it would be possible to communicate other information as well, but there is a trade-off between the communication overhead and the improvement that comes from communicating such data. See Section 4.2 for a description of the UG framework that provides an implementation of this approach. Commercial solvers CPLEX and Gurobi can also execute in this manner (see Section 4.1). There is a study specialized for this paradigm [15].

Algorithm 3: Basic Racing Algorithm

Input : Set of N different MILP solvers, N processors indexed by $S = \{1, \ldots, N\}$, and an MILP instance to be solved.

Output : An optimal solution

```
1 terminated ← false
2 Spawn N parallel processes with solver i solving the MILP instance o processor i, ∀i ∈ S
3 while terminated = false do
4 |   (i, tag) ← Wait for message from solver processes // Returns source SOLVER
  |       identifier and message tag
5 |   if tag = incubentValue then
6 |   |   ∀j ∈ S \ {i} : Send the incumbent value to solver j
7 |   else
8 |   |   terminated ← true  // tag = optimalSolution
9 Output optimal solution
```

Pure Static Load Balancing. Another simple approach to parallelization is to use a static load balancing scheme (see Section 3.2.5) to generate and distribute a set of subproblems that can then be solved independently in parallel. This approach has advantages similar to those of the parallel racing approach—no coordination is needed after the initial subproblem generation and distribution phase and thus parallel overhead is near zero. As with parallel racing, it can be used to parallelize almost any sequential solver. It can therefore take advantage of the state-of-the-art performance of sequential solvers. Finally, the amount of redundant work is minimized.

The scheme is, however, vitally dependent on the ability to predict a priori the difficulty of the subproblems being generated in order to balance the load. This problem of predicting difficulty is notoriously difficult except for problems with certain special structure and lies at the core of the difficulty of parallelizing. If the predictions made are not accurate, then some solvers will end up solving their assigned subproblem(s) well before others and this will result in a potentially large amount of idle time for the assigned processors, introducing large overhead.

As with racing strategies, this idea dates back to the early days of the development of parallel branch-and-bound algorithms (see, e.g., [56]). A very recent implementation of it, the so-called SelfSplit approach, is described in [34]. As in the parallel racing case, the basic scheme can be improved by allowing *some* communication between the solvers, at the cost of increased complexity

and a small amount of overhead. Algorithm 4 provides the basic outline of an algorithm similar to that of [34].

Algorithm 4: Static Load Balancing Algorithm

Input : Single MILP solver, set of N processors $i \in S = \{1, \ldots, N\}$, and a MILP instance to be solved

Output : An optimal solution

```
1  Spawn N identical processes solving the MILP instance on processors 1 to N with a fixed
     limit of L nodes.
2  forall i ∈ S IN PARALLEL do
3      forall leaf nodes j in the partial search tree do
4          compute a score for the difficulty of node j
5      Sort the nodes by decreasing scores
6      Assign a color c between 1 and N to all nodes, in round robin
7      Discard all nodes which color c ≠ i from the branch-and-bound tree
8      Enumerate the remaining parts of the search tree.
9      Output solution x*_i for solver i.
10 Output x* = argmin_{i∈S} c^⊤ x*_i
```

Master-Worker. The *Master-Worker paradigm* is a well-known and widely used paradigm for many parallel workloads. The basic scheme involves a single *Master* process that coordinates the efforts of a set of *Workers*. In most cases, the role of the Master is to balance the load as effectively as possible, though it may possibly play other roles as well. In its straightforward implementation, the Master may become a communication bottleneck as the amount of communication with Workers may increase linearly with the number of Workers, eventually becoming a bottleneck. Naturally, there are approaches to combat this, such as having the Master request Workers to send information (such as the description of a workload) directly to each other rather than via the Master. The granularity of the workload can also be dynamically changed so that Workers do more work independently and intervals between communication with the Master is increased, decreasing the communication load on the Master.

This scheme, while not quite as simple as the previously described ones, is still rather simplified—all coordination decision are made in a single sequential process. The potential advantage of this scheme is that the Master maintains a complete (though inevitably somewhat outdated) picture of the state of the entire procedure. As long as the Master's global view remains accurate, this allows the search order in the parallel algorithm to replicate, to a large extent, the search order that would be observed in sequential mode. The down side, of course, is that in order for this fine-grained global view to be maintained accurately requires a high communication frequency with Workers. There must inevitably be a point at which the Master becomes a communication bottleneck. Here again, there is an obvious tradeoff at play. Less data being shared will result in less effectiveness coordination decision in the long-run, most likely resulting in redundant work being done. More data being shared results in higher overhead.

Despite the simplicity and the potential downsides, this approach has been used successfully to solve difficult open instances (`a1c1s1, roll3000, timtab2`) from MIPLIB 2003 in a large computational grid. A MILP instance was decomposed carefully using CPLEX, and the generated subproblems were distributed across a computational grid. Solution of the subproblems continued until predefined termination criteria, such as a time limit, were met. Subproblems not yet solved were decomposed using CPLEX again, and the newly generated subproblems were again

distributed on a computational grid [14]. FATCOP [16] is a solver developed to perform this process automatically.

Algorithms 5, 6 show a simplified Master-Worker based parallel algorithm for solution of MILPs. After sending a subproblem to a Worker, there is no communication between the Master and the Worker. If a Worker solves a received subproblem, an optimal solution for the subproblem is returned; otherwise, the terminal nodes of the search tree for the subproblem (a new collection of subproblems) are returned after the Worker performs. The Master coordinates distribution of the global collection of subproblem to Workers. In order to keep the number of subproblems as small as possible, the search strategy is the so-called *depth-first* strategy which processes the deepest node in the search tree first and minimizes the generation of new terminal nodes. All subproblems managed by Master are treated independently, which makes fault tolerance easy to handle when computing resources are being added and removed at random, as on a computational grid. In the algorithm, N could be changed dynamically.

Algorithm 5: Master (Master-Worker)

Input : Single MILP solver, set of N processors $i \in S = \{1, \ldots, N\}$ and an MILP instance to be solved

Output : An optimal solution

```
1  Spawn N Workers with the MILP solver on processors 1 to N
2  x* ← NULL
3  I ← S   // Idle processors
4  A ← ∅   // Busy processors
5  Q ← {0} // Queue of indices of subproblems for processing, 0 is the index
       of the root problem
6  R ← ∅   // Subproblems currently being processed
7  while Q ≠ ∅ and R ≠ ∅ do
8      while I ≠ ∅ and Q ≠ ∅ do
9          i ∈ I, I ← I \ {i}, A ← A ∪ {i}
10         j ∈ Q, Q ← Q \ {j}, R ← R ∪ {(i, j)}
11         Send subproblem j and best solution to processor i
12     (i, tag) ← Wait for message // Returns processor identifier and message tag
13     if tag = optimalSolutionFound or tag = solutionFound then
14         Receive solution x̂ from processor i
15         if x* = NULL or c^⊤ x̂ < c^⊤ x* then
16             x* ← x̂
17     Receive list of candidate subproblems generated by processor i and add them to Q
18     R ← R \ {(i, j)}
19     A ← A \ {i}, I ← I ∪ {i}
20 Output x*
```

Supervisor-Worker. In contrast to the Master-Worker paradigm, the idea of Supervisor-Worker is that the Supervisor functions only to coordinate workload, but does not actually store the data associated with the search tree. The terminal nodes of search tree in the Workers are collected on demand and a set of subproblems in the Supervisor works as a buffer to ensure subproblems are available to idle Workers as needed. This coordination scheme has seen success in solving open instances from both MIPLIB2003 and MIPLIB2010 by using the Ubiquity

Algorithm 6: Worker (Master-Worker)

Input : A subproblem and an incumbent solution
Output : A termination code, improved solution (if found) and a list of candidate subproblems.

```
1  Set initial global upper bound based on the incumbent solution.
2  Set termination criteria and the other parameters, such as search strategy etc.
3  Execute sequential solution algorithm until termination criteria are reached.
4  if algorithm solves subproblem to optimality then
5  |  tag ← optimalSolutionFound
6  else
7  |  if algorithm found feasible solution then
8  |  |  tag ← solutionFound
9  |  else
10 |  |  tag ← noSolutionFound
11 Send candidate subproblems, any solution found and tag to Master.
```

Generator(UG) framework described in Section 4.2 and the underlying sequential solver SCIP on a large supercomputer. This coordination scheme is also used in the CPLEX distributed MILP solver mentioned in Section 4.1. Algorithms 7 and 8 show a parallel algorithm with a simplified Supervisor-Worker coordination scheme similar to the one used in UG.

In the Supervisor-Worker approach in UG, the load balancing is accomplished mainly by toggling the collection mode flag in the Worker. Turning collecting mode on results in additional "high quality" subproblems being sent to the Supervisor, which can then be distributed to Workers. Naturally, the method of selecting which Worker to collect from is crucial to effectiveness of the approach. Some additional keys to avoiding having the Supervisor becomes a communication bottleneck are:

- Frequently of status updates can be controlled depending on the number of Workers.
- The maximum number of Workers in collection mode is capped and the Workers are carefully chosen in a dynamic fashion.

Naturally, there is a tradeoff between the frequency of communication and the number of Workers in collection mode and the degree to which the parallel search order replicates the sequential one. As the number of processors is scaled up, this tradeoff must be carefully navigated.

Multiple-Master-Worker and Master-Hub-Worker. An alternative approach to ensuring that the Master doesn't become a bottleneck is to either create additional Master processes (Multiple-Master-Worker) or to even create a layer of "middle management" (Master-Hub-Worker). In both schemes, Workers are grouped into collectives called Hubs, each of which has its own Hub Master. In creating this management hierarchy, the hope is that the Hub Masters can effectively balance the workload within the collective for some time before having to coordinate with other Hubs through the Master to do higher-level global balancing. Naturally, more levels can be added and experiment with a scheme such as having a dynamic number of levels have appeared in the literature [104].

The CHiPPS framework described in Section 4.2 uses the Master-Hub-Worker paradigm, whereas the PEBBL framework, described in 4.2 uses a multiple master approach. This scheme can be extended to allow for even more layers in a hierarchical load-balancing scheme [48]. A

Algorithm 7: Supervisor (Supervisor-Worker)

Input : Single MILP solver, set of N processors $i \in S = \{1, \ldots, N\}$ and an MILP instance to be solved

Output : An optimal solution

Spawn N Workers with the MILP solver on processors 1 to N
collectMode $\leftarrow$ **false**
$x^* \leftarrow$ NULL
$I \leftarrow N \setminus \{1\}$
$A \leftarrow \{1\}$
$Q \leftarrow \emptyset$
$R \leftarrow \{(1, 0)\}$ `// Subproblems currently being processed, 0 is the index of the root problem`
Send the root problem to processor 1
while $Q \neq \emptyset$ *and* $R \neq \emptyset$ **do**
    $(i, \text{tag}) \leftarrow$ Wait for message `// Returns processor identifier and message tag`
    if tag = solutionFound **then**
        Receive solution $\hat{x}$ from processor i **if** $x^* =$ NULL **or** $c^\top \hat{x} < c^\top x^*$ **then**
            $x^* \leftarrow \hat{x}$
    else
        if tag = subproblem **then**
            Receive a subproblem indexed by k from processor i
            $Q \leftarrow Q \cup \{k\}$
        else
            if tag = terminated **then**
                $R \leftarrow R \setminus \{(i, j)\}$ `//` j `is the index of the terminated subproblem`
                $A \leftarrow A \setminus \{i\}$, $I \leftarrow I \cup \{i\}$
            else
                if tag = status **then**
                    if collectMode = **true then**
                        if *there are enough heavy subproblems in* Q **then**
                            `//` *heavy subproblem* `is a subproblem which is expected to generate a large subtree`
                            Send message with tag = stopCollecting to processors in collecting mode.
                            collectMode $\leftarrow$ **false**
                    else
                        `//` collectMode = **false**
                        if *there are not enough heavy subproblems in* Q **then**
                            Select processors which have heavy subproblems
                            Send message with tag = startCollecting to the selected processors
                            collectMode $\leftarrow$ **true**
    while $I \neq \emptyset$ **and** $Q \neq \emptyset$ **do**
        $i \in I$, $I \leftarrow I \setminus \{i\}$, $A \leftarrow A \cup \{i\}$
        subproblem $j \in Q$, $Q \leftarrow Q \setminus \{j\}$, $R \leftarrow R \cup \{(i, j)\}$
        Send subproblem j and x^* to processor i
$\forall i \in S$: Send message with tag = termination to processor i
Output x^*

Algorithm 8: Worker (Supervisor-Worker)

Input : An MILP solver and an original MILP instance to be solved

```
1  collectMode ← false
2  terminate ← false
3  while terminate = false do
4      (i, tag) ← Wait for message from Supervisor // Returns Supervisor identifier 0
           and message tag
5      if tag = subproblem then
6          Receive subproblem and solution from Supervisor
7          Solve the subproblem, periodically communicating with supervisor as follows
               - Send message with tag solutionFound anytime a new solution is discovered.
               - Periodically send message with tag status to report current lower bound for this
                   subproblem.
               - When messages with tag startCollecting or stopCollecting are received, toggle collectMode.
               - When collectMode = true, periodically send message with tag subproblem containing best
                   candidate subproblem.
8          Send a message with tag = terminated
9      else
10         if tag = termination then
11             terminate ← true
```

basic scheme similar to the one in CHiPPS is described in Algorithms 9–11. The keys to ensuring effectiveness of this framework are:

- The number of clusters (Hub Masters) and thereby the cluster size can be dynamically controlled.
- The frequency of status updates between workers and hubs, as well as hubs and masters can be fixed or automatically adjusted adaptively (the default)
- The frequency of inter-cluster and intra-cluster load balancing can be fixed or automatically adjusted adaptively (the default)
- The granularity of the work unit can also be fixed or automatically adjusted adaptively (default).

As with Supervisor-Worker, there is a clear tradeoff in adjusting these parameters between communication frequency and the ability to replicate the sequential search order.

Self Coordination. Recently, a completely decentralized approach to parallel branch-and-bound was introduced and implemented in PIPS-SBB [64], a distributed-memory parallel solver for Stochastic Mixed Integer Programs (SMIPs). Parallel PIPS-SBB features a lightweight mechanism for redistributing the most promising nodes among all the parallel processors without the need for a centralized load coordinator. This alternative scheme seeks to keep the load in balance without formally introducing any notion of a separate process to coordinate the load. In order to

Algorithm 9: Master (Master-Hub-Worker)

Input : Single MILP solver, set of N processors $i \in S = \{1, \ldots, N\}$, number of hubs H, and an MILP instance to be solved

Output : An optimal solution

```
Spawn N Processes with the MILP solver on processors 1 to N
// Process 1 is the master, processes 1 to H are hubs (master is also hub),
   all processes also function as workers.
x* ← NULL
L_i ← −∞, W_i ← 0, 2 ≤ i ≤ H // Best bound and workload of cluster i
Do initial static load balancing // Either 2-level root initialization or spiral
while ∃i, W_i > 0 do
    while timeSinceLastBalanceCheck < masterBalancePeriod do
        (i, tag) ← Check for messages // Returns processor identifier and message
            tag or NULL
        if tag = solutionFound then
            Receive newSolution from processor i
            if x* = NULL or c^⊤ x̂ < c^⊤ x* then
                x* ← x̂
        else
            if tag = hubStatusUpdate then
                Update W_i, L_i
            else
                Process message as hub or worker (see Algorithms 10 and 11)
        Do a unit of work // As worker
        Update timeSinceLastBalanceCheck
    if W_i < workloadThreshold or L_i > boundThreshold then
        Balance cluster loads
∀i ∈ S : Send message with tag = termination to processor i
Output x*
```

accomplish this, a synchronization point must be added, potentially introducing an alternative source of overhead. This scheme is untested with regard to solving generic MILPs, so it is unclear how to assess this tradeoff. Nevertheless, we introduce the basic scheme here in Algorithm 12. Instead of point-to-point communications, parallel processors exchange subproblems via all-to-all collective MPI asynchronous communications, allowing to rebalance the computational load using a single communication step. Parallel processors proceed to solve subproblems until the problem has been solved to optimality.

4 Software

In this section, we summarize software architectures of existing software for solving MILPs in parallel. The development of parallel software for solving MILP has a long history by now and many solvers and frameworks have preceded the one listed here. In addition to the ones listed below, previous efforts include ABACUS [49], PPBB-LIB [91], FATCOP [16], PARINO [57],

Algorithm 10: Hub Master (Master-Hub-Worker)

Input : Process index k, set S_k of workers assigned to cluster.

```
1  L_i ← −∞, W_i ← 0, i ∈ S_k // Best bound and workload of worker i
2  terminate ← false
3  Participate in initial static load balancing
4  while terminate = false do
5      while timeSinceLastBalanceCheck < hubBalancePeriod and timeSinceLastHubReport <
        hubReportPeriod do
6          (i, tag) ← Check for message // Returns processor identifier and message
               tag or NULL
7          if tag = masterRequestsBalance or tag = workerRequestsBalance then
8              Identify donors and notify them of need to donate
9          else
10             if tag = workerStatusUpdate then
11                 Update W_i, L_i
12             else
13                 if tag = terminate then
14                     terminate ← true
15                 else
16                     Process message as worker (see Algorithm 11)
17         Do a unit of work and request balance if necessary // As worker
18         Incorporate worker status into hub status
19         Increment timeSinceLastBalanceCheck,timeSinceLastHubReport
20     if ∃i, W_i < workloadThreshold or L_i > boundThreshold then
21         Balance load of workers
22     if timeSinceLastHubReport ≥ hubReportPeriod then
23         Send hub status to master
```

MW [42], BoB [7], Bob++ [24], PUBB [87], PUBB2 [85], ParaLEX [83], ZRAM [12], and MallBa [5].

We divide this section into two subsections. In the first, we describe solvers that have embedded, generally tightly integrated parallelization schemes. In the second, we describe frameworks that can be used in tandem with multiple underlying sequential solvers.

4.1 Solvers

SYPHONY. SYMPHONY [74, 76, 72, 22] was originally developed in the early 1990s as a framework that was intended to be customized by the addition of user-defined subroutines for generation of valid inequalities and other functionality (known today as *callbacks*). It did not initially have an execution mode as a generic MILP solver, but this capability was added later by leveraging libraries for I/O and generation of valid inequalities provided by the COIN-OR project [59].

SYMPHONY was originally designed to run on distributed memory platforms and was later modified to run on shared-memory platforms. It is implemented mainly in pure C and in its dis-

Algorithm 11: Worker (Master-Hub-Worker)

Input : Process index k

```
1  L_k ← −∞, W_k ← 0 // Best bound and workload of this worker
2  terminate ← false
3  Participate in initial static load balancing
4  while terminate = false do
5      while timeSinceLastWorkerReport < workerReportPeriod do
6          (i, tag) ← Check for message // Returns processor identifier and message
               tag or NULL
7          if tag = hubRequestsDonation then
8              Identify subtree to donate or split current tree
9          else
10             if tag = subTree then
11                 Receive donated subtree
12             else
13                 if tag = terminate then
14                     terminate ← true
15         Do a unit of work on best locally available subtree, send improved solution (if
             found), and request more work if necessary
16         Increment timeSinceLastWorkerReport
17     if timeSinceLastHubReport ≥ hubReportPeriod then
18         Send worker status to hub
```

tributed execution mode, it uses the message-passing protocol PVM for communication. Generally speaking, it employs a Master-Worker coordination mechanism with node-level parallelism (the unit of work is a single node), though it has a variety of execution modes, some of which enable sub-node parallelism and parallelize some auxiliary processes that involve knowledge sharing.

SYMPHONY's functionality is divided into five modules that are designed to execute independently in parallel or in various bundled combinations.

- *Master*: This module contains functions that perform problem initialization and I/O. The primary reason for a separate master module is fault-tolerance, as this module is not heavily tasked once the computation has begun.
- *Tree Manager* (TM): The TM controls the execution of the algorithm by maintaining a complete description of the search tree and deciding which candidate node should be chosen as the next to be processed at each step of the algorithm.
- *Node Processor* (NP): The NP modules perform basic node processing to calculate bounds and also perform the branching operation.
- *Cut Generator*: (CG): The CG modules generate valid inequalities used to strengthen the relaxations solved by the NP modules. Multiple CG modules can be executed in parallel in tandem with NP modules.
- *Cut Pool* (CP): The CP modules store previously generated inequalities and act as auxiliary cut generators. It is possible to have multiple cut pools for different parts of the tree and even to store locally valid inequalities in them.

Algorithm 12: Self Coordination Algorithm

Input : Single MILP solver, set of N processors $i \in S = \{1, \ldots, N\}$ and an MILP instance to be solved

Output : An optimal solution

```
1  Spawn N identical processes solving the MILP instance on processors 1 to N
2  forall i ∈ S IN PARALLEL do
3      Add the root problem to priority queue Q_1.
4      Set upper bound U^i ← ∞ and lower bound L^i ← −∞ on processor i.
5      while min_{i∈S}{L^i} < min_{i∈S}{U^i} do
6          if Load imbalance exists or synchronization point is reached then
7              Exchange best solutions, set U^i ← min_{1≤i≤N}{U^i}.
8              Determine the top M candidate subproblems from ∪_{i∈S} Q_i and redistribute
                them among all processors in a round robin fashion.
9              if termination conditions are met then
10                 return
11         Remove subproblem s from Q_i
12         Process subproblem s, update U^i, L^i
13         if s not fathomed then
14             Branch to create children of s and add them to Q^i.
```

It is possible to combine the modules in various ways, such as either

- combining the NP module with the CG module to obtain one single sequential module that performs both functions or
- combining the CP, TM, and Master modules into a single module maintaining all global information.

After processing each node and making a branching decision, the NP module queries the TM module as the what to do next: retain one of the child nodes just generated and continue "diving" or wait for a new node to be sent. This approach minimizes redundant computation by ensuring that all NP modules are processing high-quality nodes, but increases communication overhead substantially. Scalability is limited by the TM's ability to handle incoming requests from the NP modules.

SYMPHONY's data structures are designed to ensure that all data that needs to be stored and communicated is represented as compactly as possible. All data in the tree is stored using a differencing scheme in which only the differences between a child node and parent node are stored. Descriptions of valid inequalities are only stored once and referred to elsewhere by index. In this way, parallel overhead is reduced as much as possible.

SYMPHONY also has a share memory parallel mode implemented using the OpenMP protocol to create a multi-threaded program that functions in roughly the same fashion as the distributed parallel version but with all communication through memory rather than over the network. The scalability issues with the shared memory version are similar to that of the distributed version.

SYMPHONY has been used to develop a number of custom solvers for combinatorial problems, such as the vehicle routing problem [71].

CBC. Cbc [35] is an open source solver originally developed by IBM. It employs a simple thread-based Master-Worker scheme, which is a straightforward parallelization of its sequential algorithm.

Nodes are handed off by the master thread to idle workers one at a time and the results collected, with all global data stored centrally. The sequential algorithm is itself quite sophisticated and this simple approach to parallelization is quite effective at a small scale, since it mirrors the algorithmic approach taken by the underlying sequential solver. Cbc has a deterministic execution mode in which the parallelization is at the subtree level. In this mode, each thread works on an entire subtree, with the amount of work fixed deterministically. After all threads complete their unit of work, there is a synchronization point, after which computation continues from this deterministic state.

BCP. BCP [54, 75] is a re-implementation and generalization of SYMPHONY in the C++ language using the more modern MPI message-passing protocol. Its modular design is similar to that of SYMPHONY. It was originally conceived as a framework for implementing customized column-generation algorithms. Its abilities as a generic MILP solver are limited, as it was not intended for use in this mode. It employs the Master-Worker coordination scheme with node parallelism in a fashion similar to SYMPHONY, with a complete description of the tree maintained centrally and all decisions about search order made centrally. Its limitations from a scalability standpoint are also similar to SYMPHONY's.

BLIS and DisCO. BLIS [101, 103] is an open source parallel MILP solver that is part of the CHiPPS hierarchy to be described in section 4.2 below. DisCO [13] is a recent re-implementation and generalization of BLIS that supports the solution of mixed integer second-order conic optimization problems.

DIP. DIP (Decomposition in Integer Programming) [73, 37] is a decomposition-based solver that takes a different approach to parallelism than any of the others listed so far. DIP was built on the ALPS tree search framework [102], although it does not currently take advantage of the built-in ability of ALPS to parallelize the tree search at the subtree level. Rather, it parallelizes the bounding process of individual search tree nodes (subnode parallelism). This can be done in a number of different ways. First, it can utilize an interior point-based LP solver to solve the LPs that arise. More importantly, however, since it can recursively use an MILP solver for solving the *column generation subproblem* that must be solved during the bounding process, this step can itself be parallelized by using one of the other solvers listed in this section. Furthermore, when there is block structure present (see [93]), the solution of the subproblem can itself be decomposed into independent subproblems that can then also be solved in parallel. These two strategies may even be hybridized.

FICO Xpress-Optimizer. The internal parallelization of the FICO Xpress-Optimizer is based on a general task scheduler which is independent of the concrete MILP solving application. It can handle the execution of interdependent tasks in a deterministic fashion. A core aspect of its design is the capability to handle *asymmetric tasks* that might have different levels of complexity. It is not only be possible to have, e.g., cutting, heuristics, and exploration of the branch-and-bound tree parallelized individually, but to run tasks of each type at the same time.

The parallel design of Xpress avoids fixed synchronization points. At the time when a task is created, it gets a *deterministic stamp*. The task may only use information which is itself tagged with a smaller stamp. In this way, the task uses only a subset of information that could be available if a synchronization had been triggered when the task was created. The idea is that the potential performance loss from using slightly "outdated" information will be easily made up for by the performance gain from dropping the need for regular complete synchronization. When information is collected, all data that is transferred back to global data receives a deterministic stamp. All tasks that have a stamp which is greater than this, will be allowed to use that information.

Concerning synchronization, the incumbent solution is always shared globally and as soon as possible. Apart from solutions, Xpress shares branching statistics and selected cuts. For both information, each task has a local variant that contains more and potentially newer information which is combined with the global information. For the ramp-up, a root-initialization-like scheme is used.

In principle, tasks could be anything; in practice, they either refer to subtrees, more particularly individual dives, or to the execution of expensive heuristics. In that sense, Xpress employs a node parallelization scheme, with a slight flavor of subtree parallelization since the nodes explored within local search heuristics are not necessarily distinct from the nodes in the main tree search or within other heuristics. For the main tree search, however, it holds that no node is explored twice. Subnode parallelization is exclusively used at the root node, mainly for solving the global LP relaxation by parallel barrier and/or parallel dual simplex, subordinately for parallel heuristics and cutting.

A problem with deterministic parallelization approaches that use fixed synchronization points is that they do not scale well on large numbers of cores. One consequence taken in Xpress is to break with the one-to-one association between threads and tasks to be performed. As a consequence, more tasks, typically by a factor between two and four, are maintained than there are threads available. By doing so, Xpress aims at immediately having new tasks available for a thread when it completes a previous task, without needing to wait for other threads to synchronize. As a consequence, the load balancer might need to dynamically put certain tasks on hold when the require information from a task which is currently not being executed and exchange them with the task lacking behind. For more details on the parallelization of Xpress, see [10].

As an important consequence, by breaking the link between threads and tasks, it is possible to make the solution path independent of the number of threads used – it only depends on the maximum number of tasks that may exist at the same time. Moreover, the parallelization of Xpress is not only deterministic, but Xpress as a whole is also platform-independent, meaning that the solver takes the exact same solution path independent of whether the underlying machine is a Mac, Windows or Linux system and what brand of CPUs is used.

Other Commercial Sovers. CPLEX [18] and Gurobi [43] are commercial solvers that also have parallel execution modes. However, not much public information is available on the approach to parallelization that these solvers take.

4.2 Frameworks

CHiPPS. CHiPPS [99, 100, 101, 103, 102] is a generic framework for performing parallel tree search, but with particular support for branch-and-bound based algorithms for optimization. CHiPPS is implemented in C++ and uses MPI as a communications protocol. The coordination mechanism is a Master-Hub-Worker scheme with subtree parallelism. The unit of work performed by a Worker is the exploration of an entire subtree until some specific criteria are met (time limit, node limit, etc.). These criteria can be dynamically adjusted to limit overhead.

The base layer of CHiPPS is ALPS, which is an abstract implementation of parallel tree search. To develop an algorithm using ALPS, the user must provide implementations of the node processing method and the branching method of the tree search algorithm to be implemented, as well as providing classes for storing descriptions of the problem data and the data required to describe a node in the search tree.

ALPS is optimized for "data-intensive" tree search algorithms in which the amount of data required to describe a single subproblem in the search tree may be large and in which additional types of knowledge also might be shared. ALPS has an extensible mechanism for defining new

types of knowledge and a general mechanism for storing such knowledge in auxiliary pools and sharing it between processors. Each processor has a *knowledge broker* responsible for routing all communication. All that's required to convert a sequential algorithm to a parallel one is to replace the serial knowledge broker with the parallel one. No other part of the implementation depends on the communication protocol or even whether the algorithm is to executed in parallel.

The BiCePs layer, built on top of ALPS, provides support for implementing relaxation-based branch-and-bound algorithms for solving optimization problems. It provides an abstract notion of modeling "objects," collections of which can be used to describe subproblems. Subtrees, in turn, are described using a compact differencing scheme in which nodes are described in terms of differences between parent and child.

CHiPPS employs a unique load balancing scheme in which entire subtrees are shifted directly from one worker to another to balance the load instead of individual nodes. A subtree can be seen as a collection of related nodes, which can be stored more efficiently if kept together as a single unit. By load balancing in this way, we hope to minimize both communication overhead and storage overhead. The overall mechanism is a hierarchical coordination scheme with several static balancing options; sender-initiated balancing (if necessary); and periodic intra- and inter-cluster dynamic load balancing, as described earlier in Algorithms 9–11.

CHiPPS has been used to develop three MILP solvers to date: DIP, BLIS (the third layer of the CHiPPS hierarchy), and DisCO (a generalization of BLIS to support solution of mixed integer conic optimization problems). It has also been to develop MibS, a solver for mixed integer bilevel optimization problems.

UG. The core idea behind UG was to make it possible to utilize a powerful state-of-the-art MILP solver as the underlying sequential solver while still achieving good parallel performance. Development was started in 2001 using a general parallel branch-and-bound software framework PUBB2 [84]. After recognizing how difficult it is to use a powerful black-box solver with a general parallel branch-and-bound framework in order to improve overall solver performance, development was begun on ParaLEX [83], which was specialized for the CPLEX solver on a distributed memory computing environment. ParaLEX was redesigned in 2008 [82], after which the idea to have a general software framework to exploit state-of-the-art MILP solvers was conceived.

Ubiquity Generator(UG) framework [86, 92] is a generic framework to parallelize an existing state-of-the-art branch-and-bound based solver, which is referred to as the *base solver*, from "outside." UG is composed of a collection of base C++ classes, which define interfaces that can be customized for any base solvers (MILP/MINLP solvers) and allow descriptions of subproblems and solutions to translated into a solver independent form. Additionally, there are base classes that define interfaces for different message-passing protocols. Implementations of ramp-up, dynamic load balancing, and check-pointing and restarting mechanisms are available as a generic functionality. The branch-and-bound tree is maintained as a collection of subtrees by the base solvers, while UG only extracts and manages a small number of subproblems (typically represented by variable bound changes) from the base solvers for load balancing.

The basic concept of UG is thus to abstract from a base solver and parallelization library and to provide a framework that can be used, in principle, to parallelize any powerful state-of-the-art base solver on any computational environment (shared or distributed memory, multithreading or massively parallel). For a particular base solver, only the interface to UG in form of specializations of base classes, as provided by UG, needs to be implemented. Similarly, for a particular parallelization library (e.g., MPI), a specialization of an abstract UG class is necessary.

The message-passing functions used in UG are limited as much as possible and are wrapped within the base class. Therefore, adding support for an additional parallelization library should be easy. The most used libraries for implementing distributed parallel programs are MPI (Message

Passing Interface) implementations. The virtual functions in the base class provided by UG can be mapped straightforward onto corresponding MPI functions. Pthreads is a popular library that is used to make multi-threaded programs and the UG specialization for Pthreads uses a simple message queue implementation, which has been developed as a part of UG code.

From the UG framework point of view, a particular instantiated parallel solver is referred to as

ug [a specific solver name, a specific parallelization library name].

Here, the specific parallelization library is used to realize the message-passing based communications. Solvers have been developed for the non-commercial SCIP solver (ParaSCIP (= ug [SCIP, MPI]), FiberSCIP (= ug [SCIP, Pthreads]) and the commercial Xpress solver (ParaXpress (= ug [Xpress, MPI]), FiberXpress (= ug [Xpress, Pthreads]). UG has also been used to parallelize the PIP-SBB solver for two-stage stochastic programming problem (ug [PIPS-SBB,MPI]).

UG employs a Supervisor-Worker coordination mechanism with subtree-level parallelism (the unit of work is a subtree). One of the most important characteristics of UG is that it makes algorithmic changes to that of the base solver, such as multiple presolving and performs very adaptive algorithms, such as racing ramp-up. These features make an instantiated parallel solver difficult to measure its performance. However, from solvability point of view, instantiated parallel solvers are one of the most successful ones. ParaSCIP successfully solved 14 previously unsolved instances from MIPLIB2003 and MIPLIB2010 as of writing this document.

UG has been developed mainly in concert with SCIP. Therefore, ug [SCIP,*] is the most mature and ug [SCIP,*] has user-customizable libraries. By using these libraries with the plug-in architecture of SCIP, a customized parallel solver can be developed with minimal effort. One of the successful results of using this development mechanism is the SCIP-Jack solver for Steiner Tree Problems and its variants. ug [SCIP-Jack, MPI] solved three open instances from the SteinLib[89] benchmark set. The largest-scale computation conducted with ParaSCIP is up to 80,000 cores on TITAN at Oak Ridge National Laboratory[81].

PEBBL. The development of the Parallel Enumeration and Branch-and-Bound Library (PEBBL) [28] was sponsored by Sandia National Laboratories and has been on-going for close to two decades. Its purpose was to support the solution of optimization problems arising in applications of interest to that laboratory. The PEBBL project itself resulted from the splitting of the parallel MILP solver PICO into an abstract framework for implementing parallel branch-and-bound algorithms and the parts of PICO specific to the solution of MILPs. PICO is now an application layer built on top of the base layer PEBBL. PEBBL uses a multiple-master-worker coordination mechanism with a sophisticated load-balancing scheme described in detail in [28] to achieving scalability.

5 Performance Measurement

Performance measurement presents exceedingly difficult challenges when it comes to parallel MILP solvers. As we mentioned in Section 3.1.1, performance of parallel MILP solvers is often assessed by measuring the amount of *parallel overhead* introduced by the parallelization scheme. The direct measurement of such overhead is problematic, so parallel overhead is often measured indirectly. The most common way of doing this involves measuring the *efficiency*, which is an intuitive and simple measure that focuses on the effect of using more cores, assumed to be the bottleneck resource, to perform a fixed computational task (e.g., solve a given optimization problem). The efficiency of a parallel program running on N processors is

$$E_N := (T_0/T_N)/N,$$

with T_0 being the sequential running time and T_N being the parallel running time with N threads. Generally speaking, the efficiency attempts to measure the fraction of work done by the parallel algorithm that could be considered "useful." An algorithm that scales perfectly on a given system would have an efficiency of $E_N = 1$ for all N. A related measure is the *speed-up*, which is simply

$$S_N := NE_N.$$

Although this way of measuring performance seems reasonable, one faces many problems with it in practice. We outline these problems in the sections below before discussion alternatives.

5.1 Performance Variability

Sequential Algorithms. Modern MILP solvers employ complex algorithms. Many algorithmic decisions are made heuristically, such as, e.g., when and how often primal heuristics are called, what disjunction to select for branching, or how many cutting planes should be generated/added. Furthermore, the results of certain operations are not necessarily unique. For example, the root LP might have several different optimal solutions and which one is selected will be influenced by the breaking of ties during algorithmic decision-making. How these ties are broken may end up being determined by any number of factors, including details of the hardware on which the algorithm is run. Small differences in algorithm parameters, how ties are broken, and other details, can result in enormous variations in the course of the algorithm. The tree generated by the algorithm, even when the algorithm itself is deterministic for a given input, might vary greatly in both structure and size, depending on such things as the order of constraints or variables in the model or slight numerical differences introduced by different CPU types.

Some instances are more vulnerable to variation than others. One way to test whether a particular instances is prone to performance variability in regard to a particular solver, is to solve instances multiple times, each with a different permutation of the rows and columns of the constraint matrix (and associated other input data). Such permutations create a problem that is mathematically equivalent to the original one, but for which the running time of a given algorithm might vary dramatically (see, e.g., [20, 51]).

The computer architecture might have an influence also. As mentioned different CPUs might introduce slight numerical differences. NUMA architectures where the assignment of processes to cores is done by the operating system can easily have performance variations of 10%. In general, when doing benchmarking it is useful to bind processes to cores if possible to limit such variation.

To complicate matters, modern CPUs may change their clock frequency depending on the number of cores currently employed. A CPU might run much faster single-threaded and otherwise empty then it does when all cores are fully loaded. Switching this behavior off decreases variability, but on the other hand, real world performance might be quite different.

As result of all this performance variability, the number of instances that one would need to include in a test set and the number of experiments one would need to do in order to ensure that an observed, e.g., 5%, performance increase is actually statistically significant is rather high [4].

Parallel Algorithms. On top of the issues noted above, when an algorithm is executed in parallel, additional sources of variability are introduced, including possible non-determinism of the algorithm itself, due to the unpredictability of the order in which operations occur. As we mentioned in Section 3.2.7, it is usually possible to implement parallel algorithms in a deterministic mode. However, this will inevitably worsen performance, due to the required introduction of synchronization points, and if the goal is to assess the non-deterministic variant of the algorithm, then enforcing determinism does not make sense. Due to timing issues and the reasons mentioned

above, the number of branch-and-bound nodes generated to solve a particular instance might vary strongly depending on the number of cores used (see [52] for more details).

Measuring Running Time. Finally, we briefly mention that the measurement of running time is itself problematic in the case of a parallel algorithm. In the sequential case, one typically measures the running time of an algorithm not using wall-clock time (the actual real time elapsed), but rather the amount of CPU time taken by the process. The CPU time and the wall-clock time can differ if other processes are running on the computer, which they often are, especially if testing is done on a platform shared with others. In the parallel case, however, one needs to rely on wall-clock time measurement because the running time must include idle time, which may not be properly measured if one only includes CPU time. It is the total elapsed running time of the parallel algorithm that matters, but as we have already noted, wall clock time measurements are inherently more variable due to the possible influence of external processes and other extraneous factors.

5.2 Comparisons

It is typical in the literature to compare alternative algorithms for solving the same problem on the basis of some objective measures of performance and this is a primary reason for measuring such performance. We have so far motivated why performance measurement is problematic due to inherent variability. This is a general phenomenon that affects all parallel algorithms. In the case of comparing parallel MILP solvers, however, there are additional problematic factors. In general, the main goal of algorithmic research in MILP solvers is to reduce the number of branch-and-bound nodes required to solve an instance. At present, it is not unusual for an instance to be solved in the root node or within less than a few hundred nodes. Unfortunately, as we mentioned earlier, reductions in the size of the search tree have a negative impact on scalability. This means that differences in the underlying sequential algorithm can impact our assessment of scalability of the parallel algorithms, although these differences may be tangential to the differences we are actually attempting to observe (differences due to the approach to parallelization). In the extreme, one could imagine comparing a parallel algorithm employing an underlying sequential solver capable of solving most instances in a given benchmark set by enumerating only a handful of nodes against one that requires thousands of nodes. The former parallel algorithm will likely be more effective (faster) overall, while the latter is more likely to be scalable. Separating the effects of the underlying sequential solver from the approach taken by parallel algorithm itself is difficult at best. This is further highlighted by difficulties encountered in selecting a proper test set.

5.3 Instance Selection

Due to well-established data formats collections of widely used data-sets, e.g. the MIPLIB2010[51], are available and are generally recommended to be used for comparisons. However, it is important to realize that properties of individual instances can limit the scalability that it will be possible to achieve, independent of a given solver's approach to parallelization. Instances that can be solved by most solvers in a small number of nodes or for which the LP relaxation in the root node is extremely difficult to solve will not scale with any current solver. It is therefore important to select instances that are suitable for parallel testing.

- Instances should produce a tree suitably large and broad enough that parallelization is both necessary and effective. Unfortunately, this property depends very much on the effectiveness of the underlying sequential solver. An instance may be suitable in this regard with respect

to one solver and not with respect to another. In addition, the size of the tree is not fixed and may vary based on random factors, as noted above.

- If one wants to use efficiency as a performance measure, it is important that instances be solvable with one processor (or at least a small number of processors), since this is used as a baseline for assessing the amount of parallel overhead. Unfortunately, instances that can be solved in a reasonable amount of time on a single processor may not be difficult enough with a large number of processors to be interesting and may not be suitable with respect to the first criteria.

This makes standard benchmark sets only partly useful to test parallel performance, at least as far as we limit ourselves to parallelizing the tree search itself. Naturally, subnode parallelism could be employed in the case of small trees, but this approach has so far not been pursued very vigorously.

It should be noted that the performance of a solver on any single instances has very little meaning. After the release of MIPLIB 2010 the overall geometric mean performance of CPLEX, Gurobi, and XPRESS was nearly equal, while the performance on individual instances varied by a factor of up to 1,500.

5.4 Alternative Performance Measures

Although efficiency is the most commonly employed measure of performance, we have motivated above why it might be slightly problematic in the case of measuring the performance of parallel MILP solvers. [52] suggest alternative measures based on separation of the overall running time into two factors: the number of search tree nodes required to be processed (the size of the search tree) and the throughput rate (the number of search tree nodes processed per second per core). Variation in the former can be mainly attributed to the performance of redundant work due to differences in the search order and a possible lack of global knowledge of the upper bound. Variation in the latter is mainly due to other sources of overhead, such as idle time. Whereas the former measure is subject to the effects of algorithmic variabilities described earlier, the latter is not. If both the size of the tree and the throughput rate remain constant as the number of cores is increased, then the result will be an efficiency of one (ideal). Otherwise, either the size of the tree must have increased (i.e., redundant work is being performed) or the throughput rate has dropped due to increased overhead. By considering these two kinds of statistics together, along with any other fine-grained measurements we can obtain (direct measurement of various sources of overhead, such as idle time from blocking), we obtain a more nuanced picture of performance.

Overall it can be said that benchmarking parallel MILP solvers is a very difficult topic. To get meaningful results, one needs well-defined settings, a large number of suitable instances, the ability to execute a large number of experiments, and a clear understanding regarding the factors influencing the results.

5.5 Summary Measures

Finally, we mention that a few guidelines have been established regarding how to summarize performance over an entire benchmark set. If the results over several instances are to be combined, it has been observed that it is better done using the geometric mean or the shifted geometric mean, as opposed to the arithmetic mean. Experience has shown that the latter is often dominated by a few instances in a given test set. When comparing two or more solvers special care has to be given to the question of how to deal with instances that can be solved by only a subset of the solvers. Choosing the time limit will have an substantial impact on the overall result. The same

applies to the difficult question how deal with wrong results in a useful way. On the other hand, only selecting those instances for comparison which can be solved by all solvers is certainly a disadvantage to those solvers that are superior in this regard. Alternatives to single summary statistics, such as performance profiles [25], should also be considered.

6 Concluding Remarks

In this article, we have provided an overview of the main challenges involved in parallelizing solution methods for MILP solvers. We have also surveyed the current state-of-the-art in terms of available software implementations. Although tremendous effort has been directed towards the development of scalable parallel algorithms, the tension between scalability and overall effectiveness is ever-present and strategies for parallelization must constantly evolve in order to effectively exploit improvements in sequential solvers. Replicating the algorithmic schemes of sequential solvers in parallel continues to pose significant challenges. Nevertheless, substantial progress has been observed and more is expected as technology continues to evolve. New frontiers, such as the exploitation of GPUs, will continue to pose interesting research questions for years to come.

References

[1] Achterberg, T.: Conflict analysis in mixed integer programming. Discrete Optimization **4**(1), 4–20 (2007). Special issue: Mixed Integer Programming

[2] Achterberg, T., Bixby, R.E., Gu, Z., Rothberg, E., Weninger, D.: Presolve reductions in mixed integer programming. ZIB-Report 16-44, Zuse Institute Berlin, Takustr.7, 14195 Berlin (2016)

[3] Achterberg, T., Koch, T., Martin, A.: MIPLIB 2003. ORL **34**(4), 1–12 (2006)

[4] Achterberg, T., Wunderling, R.: Mixed integer programming: Analyzing 12 years of progress. In: M. Jünger, G. Reinelt (eds.) Facets of Combinatorial Optimization: Festschrift for Martin Grötschel, pp. 449–481. Springer Berlin Heidelberg (2013)

[5] Alba, E., Almeida, F., Blesa, M., Cabeza, J., Cotta, C., Díaz, M., Dorta, I., Gabarró, J., León, C., Luna, J., Moreno, L., Pablos, C., Petit, J., Rojas, A., Xhafa, F.: Mallba: A library of skeletons for combinatorial optimisation. In: B. Monien, R. Feldmann (eds.) Euro-Par 2002 Parallel Processing: 8th International Euro-Par Conference Paderborn, Germany, August 27–30, 2002 Proceedings, pp. 927–932. Springer Berlin Heidelberg (2002). DOI 10.1007/3-540-45706-2_132

[6] Barney, B.: Introduction to Parallel Computing. `https://computing.llnl.gov/tutorials/parallel\_comp/`

[7] Bénichou, M., Cung, V.D., Dowaji, S., Cun, B.L., Mautor, T., Roucairol, C.: Building a parallel branch and bound library. In: Solving Combinatorial Optimization Problems in Parallel, Lecture Notes in Computer Science **1054**, pp. 201–231. Springer, Berlin (1996)

[8] Berthold, T.: Primal heuristics for mixed integer programs. Diploma thesis, Technische Universität Berlin (2006)

[9] Berthold, T.: Heuristic algorithms in global MINLP solvers. Ph.D. thesis, Technische Universität Berlin (2014)

[10] Berthold, T., Farmer, J., Heinz, S., Perregaard, M.: Parallelization of the FICO Xpress-Optimizer. In: G.M. Greuel, T. Koch, P. Paule, A. Sommese (eds.) Mathematical Software – ICMS 2016, pp. 251–258. Springer International Publishing (2016). DOI 10.1007/978-3-319-42432-3_31

[11] Berthold, T., Salvagnin, D.: Cloud branching. In: C. Gomes, M. Sellmann (eds.) Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, *Lecture Notes in Computer Science*, vol. 7874, pp. 28–43. Springer Berlin Heidelberg (2013)

[12] Brüngger, A., Marzetta, A., Fukuda, K., Nievergelt, J.: The parallel search bench ZRAM and its applications. Annals of Operations Research **90**(0), 45–63 (1999). DOI 10.1023/A:1018972901171

[13] Bulut, A., Ralphs, T.K.: Disco version 0.95 (2017). DOI 10.5281/zenodo.237107

[14] Bussieck, M.R., Ferris, M.C., Meeraus, A.: Grid-enabled optimization with GAMS. IJoC **21**(3), 349–362 (2009). DOI 10.1287/ijoc.1090.0340

[15] Carvajal, R., Ahmed, S., Nemhauser, G., Furman, K., Goel, V., Shao, Y.: Using diversification, communication and parallelism to solve mixed-integer linear programs. Operations Research Letters **42**(2), 186–189 (2014). DOI 10.1016/j.orl.2013.12.012

[16] Chen, Q., Ferris, M.C., Linderoth, J.: Fatcop 2.0: Advanced features in an opportunistic mixed integer programming solver. Annals of Operations Research **103**(1), 17–32 (2001). DOI 10.1023/A:1012982400848. URL `http://dx.doi.org/10.1023/A:1012982400848`

[17] Cornuéjols, G., Karamanov, M., Li, Y.: Early estimates of the size of branch-and-bound trees. INFORMS J. on Computing **18**(1), 86–96 (2006). DOI 10.1287/ijoc.1040.0107

[18] IBM ILOG CPLEX Optimizer. `http://www-01.ibm.com/software/integration/optimization/cplex-optimizer/`

[19] Crainic, T., Le Cun, B., Roucairol, C.: Parallel branch-and-bound algorithms. In: E. Talbi (ed.) Parallel Combinatorial Optimization, pp. 1–28. Wiley, New York (2006)

[20] Danna, E.: Performance variability in mixed integer programming (2008). Presentation, Workshop on Mixed Integer Programming (MIP 2008), Columbia University, New York. `http://coral.ie.lehigh.edu/~jeff/mip-2008/talks/danna.pdf`

[21] Dantzig, G.B., Wolfe, P.: Decomposition principle for linear programs. Operations Research **8**(1), 101–111 (1960)

[22] DeNegre, S., Ralphs, T.K.: A branch-and-cut algorithm for bilevel integer programming. In: Proceedings of the Eleventh INFORMS Computing Society Meeting, pp. 65–78 (2009). DOI 10.1007/978-0-387-88843-9_4. URL `http://coral.ie.lehigh.edu/~ted/files/papers/BILEVEL08.pdf`

[23] Dinan, J., Olivier, S., Sabin, G., Prins, J., Sadayappan, P., Tseng, C.W.: Dynamic load balancing of unbalanced computations using message passing. In: 2007 IEEE International Parallel and Distributed Processing Symposium, pp. 1–8 (2007). DOI 10.1109/IPDPS.2007.370581

[24] Djerrah, A., Cun, B.L., Cung, V.D., Roucairol, C.: Bob++: Framework for solving optimization problems with branch-and-bound methods. In: 2006 15th IEEE International Conference on High Performance Distributed Computing, pp. 369–370 (2006). DOI 10.1109/HPDC.2006.1652188

[25] Dolan, E.D., Moré, J.J.: Benchmarking optimization software with performance profiles. Mathematical Programming **91**(2), 201–213 (2002). DOI 10.1007/s101070100263. URL http://dx.doi.org/10.1007/s101070100263

[26] Eckstein, J.: Control strategies for parallel mixed integer branch and bound. In: Proceedings of the 1994 conference on Supercomputing, pp. 41–48. IEEE Computer Society Press (1994)

[27] Eckstein, J.: Distributed versus centralized storage and control for parallel branch and bound: Mixed integer programming on the CM-5. Comput. Optim. Appl. **7**(2), 199–220 (1997). URL http://dx.doi.org/10.1023/A:1008699010646

[28] Eckstein, J., Hart, W.E., Phillips, C.A.: Pebbl: an object-oriented framework for scalable parallel branch and bound. Mathematical Programming Computation **7**(4), 429–469 (2015). DOI 10.1007/s12532-015-0087-1. URL http://dx.doi.org/10.1007/s12532-015-0087-1

[29] Eckstein, J., Phillips, C.A., Hart, W.E.: PEBBL 1.0 user guide (2007)

[30] Eikland, K., Notebaert, P.: lp_solve 5.5.2. http://lpsolve.sourceforge.net

[31] FICO Xpress-Optimizer. http://www.fico.com/en/Products/DMTools/xpress-overview/Pages/Xpress-Optimizer.aspx

[32] Fischetti, M., Lodi, A.: Heuristics in mixed integer programming. In: J.J. Cochran, L.A. Cox, P. Keskinocak, J.P. Kharoufeh, J.C. Smith (eds.) Wiley Encyclopedia of Operations Research and Management Science. John Wiley & Sons, Inc. (2010). Online publication

[33] Fischetti, M., Lodi, A., Monaci, M., Salvagnin, D., Tramontani, A.: Improving branch-and-cut performance by random sampling. Mathematical Programming Computation **8**(1), 113–132 (2016)

[34] Fischetti, M., Monaci, M., Salvagnin, D.: Self-splitting of workload in parallel computation. In: H. Simonis (ed.) Integration of AI and OR Techniques in Constraint Programming: 11th International Conference, CPAIOR 2014. Proceedings, pp. 394–404. Springer International Publishing (2014). DOI 10.1007/978-3-319-07046-9_28

[35] Forrest, J.: CBC MIP solver. http://www.coin-or.org/Cbc

[36] Fourer, R.: Linear programming: Software survey. OR/MS Today **42**(3) (2015)

[37] Galati, M.V., Ralphs, T.K., Wang, J.: Computational experience with generic decomposition using the DIP framework. In: Proceedings of RAMP 2012 (2012). URL http://coral.ie.lehigh.edu/~ted/files/papers/RAMP12.pdf

[38] Gamrath, G., Koch, T., Martin, A., Miltenberger, M., Weninger, D.: Progress in presolving for mixed integer programming. Mathematical Programming Computation **7**(4), 367–398 (2015)

[39] Gendron, B., Crainic, T.G.: Parallel branch-and-branch algorithms: Survey and synthesis. Operations Research **42**(6), 1042–1066 (1994). DOI 10.1287/opre.42.6.1042. URL `http://dx.doi.org/10.1287/opre.42.6.1042`

[40] Gomory, R.E.: Outline of an algorithm for integer solutions to linear programs. Bulletin of the American Mathematical Society **64**(5), 275–278 (1958)

[41] Gottwald, R.L., Maher, S.J., Shinano, Y.: Distributed domain propagation. ZIB-Report 16-71, Zuse Institute Berlin, Takustr. 7, 14195 Berlin (2016)

[42] Goux, J.P., Kulkarni, S., Linderoth, J., Yoder, M.: An enabling framework for master-worker applications on the computational grid. In: Proceedings the Ninth International Symposium on High-Performance Distributed Computing, pp. 43–50 (2000). DOI 10.1109/HPDC.2000.868633

[43] Gurobi Optimizer. `http://www.gurobi.com/`

[44] Hager, G., Wellein, G.: Introduction to High Performance Computing for Scientists and Engineers. CRC Press, Inc., Boca Raton, FL, USA (2010)

[45] Henrich, D.: Initialization of parallel branch-and-bound algorithms. In: Second International Workshop on Parallel Processing for Artificial Intelligence(PPAI-93) (1993)

[46] Huangfu, Q., Hall, J.: Parallelizing the dual revised simplex method. Tech. rep., arXiv preprint arXiv:1503.01889 (2015)

[47] Janakiram, V.K., Gehringer, E.F., Agrawal, D.P., Mehrotra, R.: A randomized parallel branch-and-bound algorithm. International Journal of Parallel Programming **17**(3), 277–301 (1988). DOI 10.1007/BF02427853

[48] Jeannot, E., Mercier, G., Tessier, F.: Topology and affinity aware hierarchical and distributed load-balancing in Charm++. In: Proceedings of the First Workshop on Optimization of Communication in HPC, COM-HPC '16, pp. 63–72. IEEE Press, Piscataway, NJ, USA (2016). DOI 10.1109/COM-HPC.2016.12

[49] Jünger, M., Thienel, S.: Introduction to ABACUS—a branch-and-cut system. Operations Research Letters **22**, 83–95 (1998)

[50] Khachiyan, L.G.: A polynomial algorithm in linear programming. Doklady Akademii Nauk SSSR **244**(5), 1093–1096 (1979). English translation in Soviet Math. Dokl. 20(1):191–194, 1979

[51] Koch, T., Achterberg, T., Andersen, E., Bastert, O., Berthold, T., Bixby, R.E., Danna, E., Gamrath, G., Gleixner, A.M., Heinz, S., Lodi, A., Mittelmann, H., Ralphs, T., Salvagnin, D., Steffy, D.E., Wolter, K.: MIPLIB 2010. Math. Prog. Comp. **3**, 103–163 (2011)

[52] Koch, T., Ralphs, T., Shinano, Y.: Could we use a million cores to solve an integer program? Mathematical Methods of Operations Research **76**(1), 67–93 (2012). DOI 10.1007/s00186-012-0390-9. URL `http://dx.doi.org/10.1007/s00186-012-0390-9`

[53] Kumar, V., Grama, A.Y., Vempaty, N.R.: Scalable load balancing techniques for parallel computers. Journal of Parallel and Distributed Computing **22**(1), 60–79 (1994)

[54] Ladányi, L.: BCP: Branch-cut-price framework (2000). URL `https://projects.coin-or.org/Bcp`

[55] Land, A.H., Doig, A.G.: An automatic method of solving discrete programming problems. Econometrica **28**(3), 497–520 (1960)

[56] Laursen, P.S.: Can parallel branch and bound without communication be effective? SIAM Journal on Optimization **4**, 288–296 (1994)

[57] Linderoth, J.: Topics in parallel integer optimization. Ph.D. thesis, School of Industrial and Systems Engineering, Georgia Institute of Technology, Atlanta, GA (1998)

[58] Linderoth, J.T., Savelsbergh, M.: A computational study of search strategies for mixed integer programming. INFORMS Journal on Computing **11**, 173–187 (1998)

[59] Lougee-Heimer, R.: The common optimization interface for operations research. IBM Journal of Research and Development **47**(1), 57–66 (2003)

[60] Mahajan, A.: Presolving mixed-integer linear programs. In: J.J. Cochran, L.A. Cox, P. Keskinocak, J.P. Kharoufeh, J.C. Smith (eds.) Wiley Encyclopedia of Operations Research and Management Science. John Wiley & Sons, Inc. (2010). DOI 10.1002/9780470400531.eorms0437. Online publication

[61] Makhorin, A.: the GNU linear programming kit. `http://www.gnu.org/software/glpk`

[62] Marchand, H., Martin, A., Weismantel, R., Wolsey, L.: Cutting planes in integer and mixed integer programming. Discrete Applied Mathematics **123**(1), 397–446 (2002)

[63] Miller, D., Pekny, J.: Results from a parallel branch and bound algorithm for the asymmetric traveling salesman problem. Operations Research Letters **8**(3), 129–135 (1989). DOI http://dx.doi.org/10.1016/0167-6377(89)90038-2

[64] Mungua, L.M., Oxberry, G., Rajan, D.: Pips-sbb: A parallel distributed-memory branch-and-bound algorithm for stochastic mixed-integer programs. In: 2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), pp. 730–739 (2016). DOI 10.1109/IPDPSW.2016.159

[65] Nemhauser, G.L., Wolsey, L.A.: Integer and combinatorial optimization. Wiley (1988)

[66] Nesterov, Y., Nemirovski, A.: Interior-Point Polynomial Algorithms in Convex Programming. Studies in Applied and Numerical Mathematics. Society for Industrial and Applied Mathematics (1994)

[67] Olszewski, M., Ansel, J., Amarasinghe, S.: Kendo: efficient deterministic multithreading in software. ACM SIGPLAN Notices **44**(3), 97–108 (2009). DOI 10.1145/1508284.1508256

[68] Osman, A., Ammar, H.: Dynamic load balancing strategies for parallel computers. URL `http://citeseer.nj.nec.com/osman02dynamic.html`

[69] Ozaltin, O.Y., Hunsaker, B., Schaefer, A.J.: Predicting the solution time of branch-and-bound algorithms for mixed-integer programs. INFORMS J. on Computing **23**(3), 392–403 (2011). DOI 10.1287/ijoc.1100.0405

[70] Pekny, J.F.: Exact parallel algorithms for some members of the traveling salesman problem family. Ph.D. thesis, Carnegie-Mellon University, Pittsburgh, PA, USA (1989)

[71] Ralphs, T.K.: Parallel branch and cut for capacitated vehicle routing. Parallel Computing **29**, 607–629 (2003). DOI 10.1016/S0167-8191(03)00045-0. URL `http://coral.ie.lehigh.edu/~ted/files/papers/PVRP.pdf`

[72] Ralphs, T.K.: Parallel branch and cut. In: E. Talbi (ed.) Parallel Combinatorial Optimization, pp. 53–101. Wiley, New York (2006). URL http://coral.ie.lehigh.edu/~ted/files/papers/PBandC.pdf

[73] Ralphs, T.K., Galati, M.V., Wang, J.: Dip version 0.92 (2017). DOI 10.5281/zenodo.246087

[74] Ralphs, T.K., Guzelsoy, M., Mahajan, A.: Symphony version 5.6 (2017). DOI 10.5281/zenodo.237456

[75] Ralphs, T.K., Ladányi, L.: Coin/bcp user's manual. Tech. rep., COR@L Laboratory, Lehigh University (2001). URL http://coral.ie.lehigh.edu/~ted/files/papers/BCP-Manual.pdf

[76] Ralphs, T.K., Ladányi, L., Saltzman, M.J.: Parallel branch, cut, and price for large-scale discrete optimization. Mathematical Programming **98**, 253–280 (2003). DOI 10.1007/s10107-003-0404-8. URL http://coral.ie.lehigh.edu/~ted/files/papers/PBCP.pdf

[77] Sanders, P.: A detailed analysis of random polling dynamic load balancing. In: International Symposium on Parallel Architectures Algorithms and Networks, pp. 382–389 (1994)

[78] Sanders, P.: Randomized static load balancing for tree-shaped computations. In: Workshop on Parallel Processing, pp. 58–69 (1994)

[79] Sanders, P.: Tree shaped computations as a model for parallel applications. In: ALV'98 Workshop on application based load balancing, pp. 123–132 (1998)

[80] SCIP: Solving Constraint Integer Programs. http://scip.zib.de/

[81] Shinano, Y., Achterberg, T., Berthold, T., Heinz, S., Koch, T., Winkler, M.: Solving open MIP instances with ParaSCIP on supercomputers using up to 80,000 cores. In: 2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS), pp. 770–779. IEEE Computer Society, Los Alamitos, CA, USA (2016)

[82] Shinano, Y., Achterberg, T., Fujie, T.: A dynamic load balancing mechanism for new ParaLEX. In: Proceedings of ICPADS 2008, pp. 455–462 (2008)

[83] Shinano, Y., Fujie, T.: ParaLEX: A parallel extension for the CPLEX mixed integer optimizer. In: F. Cappello, T. Herault, J. Dongarra (eds.) Recent Advances in Parallel Virtual Machine and Message Passing Interface. Proceedings, pp. 97–106. Springer Berlin Heidelberg (2007). DOI 10.1007/978-3-540-75416-9_19

[84] Shinano, Y., Fujie, T., Kounoike, Y.: Effectiveness of parallelizing the ILOG-CPLEX mixed integer optimizer in the PUBB2 framework. In: H. Kosch, L. Böszörményi, H. Hellwagner (eds.) Euro-Par 2003 Parallel Processing: Proceedings, pp. 451–460. Springer Berlin Heidelberg (2003). DOI 10.1007/978-3-540-45209-6_67

[85] Shinano, Y., Fujie, T., Kounoike, Y.: Pubb2: A redesigned object-oriented software tool for implementing parallel and distributed branch-and-bound algorithms. In: Proceedings of ISTEAD International Conference: Parallel and Distributed Computing and Systems, pp. 639–647 (2003)

[86] Shinano, Y., Heinz, S., Vigerske, S., Winkler, M.: FiberSCIP – a shared memory parallelization of SCIP. ZIB-Report 13-55, Zuse Institute Berlin (2013)

[87] Shinano, Y., Higaki, M., Hirabayashi, R.: A generalized utility for parallel branch and bound algorithms. In: Proceedings.Seventh IEEE Symposium on Parallel and Distributed Processing, pp. 392–401 (1995). DOI 10.1109/SPDP.1995.530710

[88] Sinha, A., Kalé, L.V.: A load balancing strategy for prioritized execution of tasks. In: Seventh International Parallel Processing Symposium, pp. 230–237. Newport Beach, CA. (1993)

[89] SteinLib Testdata Library. http://steinlib.zib.de/steinlib.php

[90] Trienekens, H.W.J.M., de Bruin, A.: Towards a taxonomy of parallel branch and bound algorithms. Tech. Rep. EUR-CS-92-01, Department of Computer Science, Erasmus University (1992)

[91] Tschoke, S., Polzer, T.: Portable parallel branch and bound library (2008). http://www.cs.uni-paderborn.de/cs/ag-monien/SOFTWARE/PPBB/ppbblib.html

[92] UG: Ubiquity Generator framework. http://ug.zib.de/

[93] Wang, J., Ralphs, T.K.: Computational experience with hypergraph-based methods for automatic decomposition in discrete optimization. In: Proceedings of the Conference on Constraint Programming, Artificial Intelligence, and Operations Research, pp. 394–402 (2013). DOI 10.1007/978-3-642-38171-3

[94] Wilkinson, B., Allen, M.: Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers. Prentice-Hall, Inc, New Jersey, USA (1999)

[95] Willebeek-LeMair, M.H., Reeves, A.P.: Strategies for dynamic load balancing on highly parallel computers. IEEE Transactions on Parallel and Distributed Systems **4**, 979–993 (1993). DOI 10.1109/71.243526

[96] Witzig, J., Berthold, T., Heinz, S.: Experiments with conflict analysis in mixed integer programming. ZIB-Report 16-63, Zuse Institute Berlin, Takustr. 7, 14195 Berlin (2016)

[97] Wolter, K.: Implementation of Cutting Plane Separators for Mixed Integer Programs. Master's thesis, Technische Universität Berlin (2006)

[98] Xu, Y.: Scalable algorithms for parallel tree search. Ph.D. thesis, Department of Industrial and Systems Engineering, Lehigh University, Bethlehem, PA, USA (2007)

[99] Xu, Y., Ralphs, T.K., Ladányi, L., Saltzman, M.: Alps version 1.5 (2016). DOI 10.5281/zenodo.245971

[100] Xu, Y., Ralphs, T.K., Ladányi, L., Saltzman, M.: Biceps version 0.94 (2017). DOI 10.5281/zenodo.245652

[101] Xu, Y., Ralphs, T.K., Ladányi, L., Saltzman, M.: Blis version 0.94 (2017). DOI 10.5281/zenodo.246079

[102] Xu, Y., Ralphs, T.K., Ladányi, L., Saltzman, M.J.: Alps: A framework for implementing parallel search algorithms. In: The Proceedings of the Ninth INFORMS Computing Society Conference, pp. 319–334 (2005). DOI 10.1007/0-387-23529-9_21. URL http://coral.ie.lehigh.edu/~ted/files/papers/ALPS04.pdf

[103] Xu, Y., Ralphs, T.K., Ladányi, L., Saltzman, M.J.: Computational experience with a software framework for parallel integer programming. The INFORMS Journal on Computing **21**, 383–397 (2009). DOI 10.1287/ijoc.1090.0347. URL `http://coral.ie.lehigh.edu/~ted/files/papers/CHiPPS-Rev.pdf`

[104] Zheng, G., Bhatelé, A., Meneses, E., Kalé, L.V.: Periodic hierarchical load balancing for large supercomputers. Int. J. High Perform. Comput. Appl. **25**(4), 371–385 (2011). DOI 10.1177/1094342010394383