Alternating Criteria Search: A Parallel Large Neighborhood Search Algorithm for Mixed Integer Programs

Lluís-Miquel Munguía *[1], Shabbir Ahmed[2], David A. Bader[1], George L. Nemhauser[2], and Yufen Shao[3]

[1] *College of Computing*,
Georgia Institute of Technology, Atlanta, GA 30332
[2] *School of Industrial and Systems Engineering*,
Georgia Institute of Technology, Atlanta, GA 30332
[2] *ExxonMobil Upstream Research Company*, Houston, TX 77098

June 6, 2016

Abstract

We present a parallel large neighborhood search framework for finding high quality primal solutions for generic Mixed Integer Programs (MIPs). The approach simultaneously solves a large number of sub-MIPs with the dual objective of reducing infeasibility and optimizing with respect to the original objective. Both goals are achieved by solving restricted versions of two auxiliary MIPs, where subsets of the variables are fixed. In contrast to prior approaches, ours does not require a starting feasible solution. We leverage parallelism to perform multiple searches simultaneously, with the objective of increasing the effectiveness of our heuristic. Comprehensive computational experiments show the efficacy of our approach as a standalone primal heuristic and when used in conjunction with an exact algorithm.

Keywords. Discrete Optimization, Parallel algorithms, Primal Heuristics, Large Neighborhood Search

1 Introduction

We present *Parallel Alternating Criteria Search*, a parallel Large Neighborhood Search (LNS) heuristic designed for finding high quality feasible solutions to

*Corresponding Author: lluis.munguia@gatech.edu unstructured MIPs. In discrete optimization, high quality feasible solutions are valuable assets in the optimization process, and constructing them has been one of the main focuses of research for the past two decades. Berthold [5] and Fischetti et al. [12] present comprehensive literature reviews on primal heuristics and their applications to MIPs. There are two main classes of primal heuristics, depending on whether they require a starting feasible solution. The feasibility pump [10, 4] is a widely used heuristic for finding feasible solutions to MIP instances quickly. The feasibility pump consists of an iterative algorithm that combines Linear Program (LP) relaxations and roundings to enforce LP and integer feasibility until a feasible solution is found. Successive works built on the original feasibility pump to improve the solution quality [1] and its overall success rate [14, 7, 3, 20]. RENS [6] is another compelling heuristic for finding solutions. Given a fractional solution to a MIP, RENS performs a LNS to find the best possible rounding, thus increasing the chances of finding a feasible solution.

Improvement heuristics, on the other hand, require a starting feasible solution and their focus is to improve its quality with respect to the objective. A large number of improvement heuristics found in the literature rely on LNS ideas. LNS heuristics solve carefully restricted sub-MIPs, which are derived from the original problem with the objective of finding improved solutions. A MIP solver is then used within the heuristic to explore the solution neighborhood. LNS approaches differ on how the search neighborhood is defined. Successful LNS heuristics include Local Branching [11], RINS [9] or DINS [15], proximity search [13] and evolutionary algorithms [22]. The neighborhoods within these heuristics are usually defined using Branch and Bound information, such as the best available solutions or the LP relaxation at the current tree node. Because of this requirement, they must be executed as part of the node processing routine during the Branch and Bound process. Due to this input dependence, many nodes must be explored before these heuristics become effective at exploring diversified neighborhoods. Thus, high quality upper bound improvements are rarely found early in the search. In order to address this issue, the search neighborhoods used in our heuristic are defined using randomization instead of Branch and Bound dependent information. This allows us to obtain a wide range of diverse search neighborhoods from the beginning of the search, thus increasing the heuristic's effectiveness.

1.1 Parallel computing applied to Integer Optimization

Due to recent trends in processor design, parallelism has become ubiquitous in today's computers. With the advent of multi-core CPUs, there has been a necessity to rethink most conventional algorithms in order to leverage the benefits of parallel computing. In the field of discrete optimization, Bader et al. [2] and Koch et al. [19] discuss potential applications of parallelism. The most widely used strategy entails exploring the Branch & Bound (B&B) tree in parallel by solving multiple subproblems simultaneously. Due to its simplicity, most state-of-the-art MIP solvers incorporate this technique, such as CPLEX [8], GUROBI [21], and ParaSCIP [23]. However, studies have suggested that parallelizing the B&B search may not scale well to a large number of cores [19]. Thus, there is an incentive to revisit parallelizations of alternative algorithmic components, such as cut generation, primal heuristics, preprocessing and branching. To our knowledge, the works of Koc et al. [17] are the only effort to parallelize primal heuristics for the general MIP. In this work, the authors present a parallelization of the Feasibility Pump[10].

LNSs are some of the most computationally expensive heuristics, since they are based on the optimization of subMIPs. A strategy to leverage parallelism is to perform a large number of LNS simultaneously over a diversified set of neighborhoods with the objective of increasing the chances of finding better solutions. To some degree, the parallelization of the branch and bound tree already provides this diversification and improvement in performance, since the exploration of multiple nodes in parallel includes the simultaneous execution of multiple heuristics with a diverse set of inputs. Our heuristic builds upon a similar parallelization strategy, and expands it by adding an additional algorithmic step that combines and consolidates the improvements found in parallel.

Parallel Alternating Criteria Search combines parallelism and diversified large neighborhood searches in order to deal with large instances. We find our approach to be competitive or better than CPLEX at finding solutions for more than 90% of the instances in the MIPLIB2010 library. The effectiveness of our heuristic increases when we focus on the hardest instances. Additionally, we present a parallel scheme that combines the use of Alternating Criteria Search in conjunction with an exact algorithm. Results show that alternative parallelizations of the Branch and Bound process can be more efficient than traditional methods, especially when large-scale instances are considered.

We introduce our primal heuristic in section 2, where we present the individual components that define it. Section 3 gives further details on the parallel implementation. Section 4 presents computational experiments and results on standard instances from the literature. Section 5 provides some concluding remarks.

2 The Alternating Search Heuristic

Our intent is to satisfy a two-fold objective: to find a starting feasible solution and to improve it with respect to the original objective. We introduce a LNS heuristic, in which two auxiliary MIP subproblems are iteratively solved to attain both goals. The process focuses on a starting incumbent solution, which is not required to be feasible. As seen in Figure 1, this solution is improved by exploring multiple Large Neighborhood Searches, in which a subset of the variables are fixed to the values of the incumbent.

We define a Mixed-Integer Program (MIP) as:

$$\min\{c^t x | Ax = b, l \leq x \leq u, x_i \in \mathbb{Z}, \forall \in \mathcal{I}\} \tag{MIP}$$

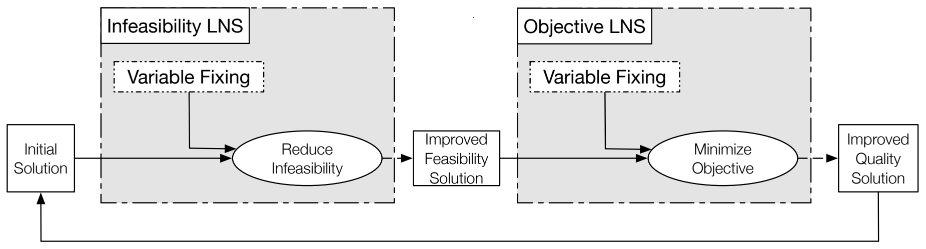

Figure 1: High level depiction of the sequential heuristic

where $c \in \mathbb{R}^n$, $A \in \mathbb{R}^{m \times n}$, $b \in \mathbb{R}^m$, and $I \subseteq \{1, \ldots, n\}$ is the subset of integer variable indices. The decision vector x is bounded by $l \in \mathbb{R}^n$ and $u \in \mathbb{R}^n$.

For linear programs, the feasibility problem is solved via the two-phase Simplex method, in which an auxiliary optimization problem is developed in order to find a feasible starting basis. We translate this approach to the case of general MIPs. In a similar fashion, the following auxiliary MIP, denoted as FMIP, poses the problem of finding a starting feasible solution as an optimization problem:

$$\begin{aligned} & \min \sum_{i=0}^{m} \Delta_i^+ + \Delta_i^- \\ & \text{s.t.} \\ & \quad Ax + I_m \Delta^+ - I_m \Delta^- = b \\ & \quad x_i = \hat{x}_i, \forall i \in \mathcal{F} \\ & \quad l \leq x \leq u \\ & \quad x_i \in \mathbb{Z}, \forall i \in \mathcal{I} \\ & \quad \Delta^+ \geq 0, \Delta^- \geq 0 \end{aligned} \tag{FMIP}$$

where I_m is an $m \times m$ identity matrix and Δ^+, Δ^- are two sets of m nonnegative continuous variables corresponding to the m constraints. A solution is feasible to a MIP if and only if it is an optimal solution of value 0 to the associated FMIP. Instead of directly solving FMIP, LNS are performed by fixing a given subset $\mathcal{F}$ of the integer variables to the values of an input vector $\hat{x}$. Due to the addition of slack variables, $\hat{x}$ is not required to be a feasible solution vector. However, it must be integer feasible and within the variable bounds in order to preserve the feasibility of the model: $l \leq \hat{x} \leq u$ and $\hat{x}_i \in \mathbb{Z}, \forall i \in \mathcal{I}$.

FMIP ensures that feasibility is preserved under any arbitrary variable fixing scheme. This represents a departure from previous approaches, where the choice of variable fixings is tied to the availability of a feasible solution. In the context of our LNS heuristic, variable fixings become a viable tool for reducing the complexity of the problem.

Using a similar approach, we introduce a second auxiliary problem aimed at improving a partially feasible solution with respect to the original objective.

OMIP is a transformation of the original MIP model, in which auxiliary slack variables are introduced in each constraint. Achieving and preserving the feasibility of the incumbent is our primary concern. In order to ensure that the optimal solution to OMIP remains at most as infeasible as the input solution, an additional constraint that limits the amount of slack is added.

$$
\begin{aligned}
\min \quad & c^t x \\
\text{s.t.} \quad & \\
& Ax + I_m \Delta^+ - I_m \Delta^- = b \\
& \sum_{i=0}^{m} \Delta_i^+ + \Delta_i^- \leq \Delta^{UB} \\
& x_i = \hat{x}_i, \forall i \in \mathcal{F} \\
& l \leq x \leq u \\
& x_i \in \mathbb{Z}, \forall i \in \mathcal{I} \\
& \Delta^+ \geq 0, \Delta^- \geq 0
\end{aligned}
\tag{OMIP}
$$

By iteratively exploring LNSs of both auxiliary MIPs, the heuristic will hopefully converge to a high quality feasible solution. By construction, infeasibility only decreases monotonically after each iteration. On the other hand, the quality may fluctuate with respect to the original objective. Figure 2 depicts the expected behavior of the algorithm.

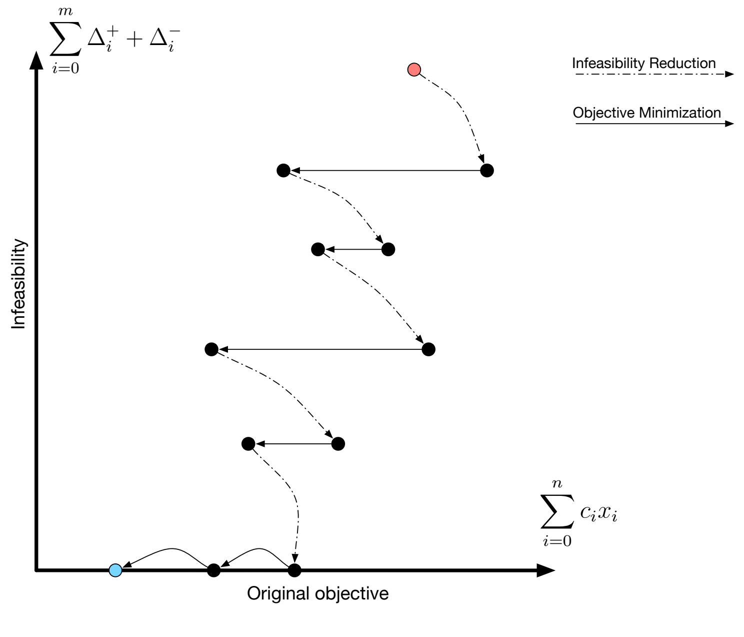

Figure 2: Transition to a high quality feasible solution

3 Parallel Implementation

In this section we present details of the parallelization of Alternating Criteria Search, as well as insights about its distributed-memory implementation and heuristic choices made throughout the algorithm.

We leverage parallelism by generating a diversified set of Large Neighborhood Searches, which are solved simultaneously. With this measure, we hope to increase the chances of finding solution improvements, hence speeding up the overall process.

Figure 3 depicts the parallel infeasibility reduction step, in which sets of LNS problems are solved. We rely on a varied population of variable fixings in order to diversify the search of the solution space, hence improving the effectiveness of parallelism. The synchronous nature of the algorithm allows the inclusion of a solution recombination step, in which the improvements found in parallel can be combined efficiently. For that purpose, an additional LNS problem is generated, in which the variables that have the same value across the different solutions are fixed. A similar approach has been previously described in the literature [22, 5]. Here, we use it in the context of a parallel algorithm. A pseudocode version is given in Algorithm 1.

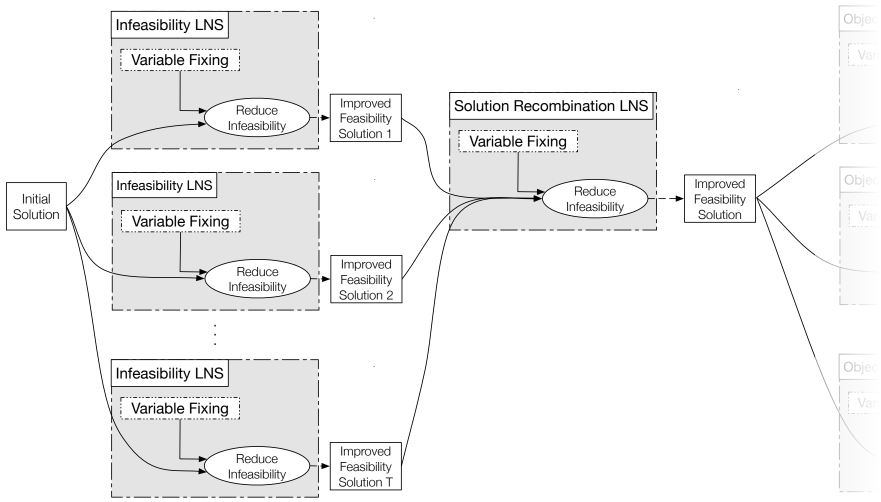

Figure 3: Diagram depicting the parallel synchronous procedure

3.1 Distributed-memory parallelism

Distributed-memory parallelism is the main paradigm for large-scale parallel computer architectures. One of the defining characteristics is the fact that

Algorithm 1 Parallel Feasibility Heuristic

```
Output: Feasible solution $\hat{x}$
  initialize $[\hat{x}, \Delta^+, \Delta^-]$ as an integer feasible solution
  $T := numThreads()$
  while time limit not reached do
    if $\sum_i \Delta_i^+ + \Delta_i^- > 0$ then
      for all threads $t_i$ in parallel do
        $\mathcal{F}_{t_i}$ := randomized variable index subset, $\mathcal{F}_{t_i} \subseteq \mathcal{I}$    ▷ Variable Fixings are diversified
        $[x^{t_i}, \Delta^{+t_i}, \Delta^{-t_i}]$ :=FMIP_LNS($\mathcal{F}_{t_i}, \hat{x}$)    ▷ $FMIP$ LNS are solved concurrently
      end for
      $\mathcal{U} := \{j \in \mathcal{I} | x_j^{t_i} = x_j^{t_k}, 0 \leq i < k < T\}$
      $[\hat{x}, \Delta^+, \Delta^-]$ :=FMIP_LNS($\mathcal{U}, x^{t_0}$)    ▷ The recombination step differs in variable fixings
    end if
    $\Delta^{UB} := \sum_i \Delta_i^+ + \Delta_i^-$
    for all threads $t_i$ in parallel do
      $\mathcal{F}_{t_i}$ := randomized variable index subset, $\mathcal{F}_{t_i} \subseteq \mathcal{I}$
      $[x^{t_i}, \Delta^{+t_i}, \Delta^{-t_i}]$ :=OMIP_LNS($\mathcal{F}_{t_i}, \hat{x}, \Delta^{UB}$)
    end for
    $\mathcal{U} := \{j \in \mathcal{I} | x_j^{t_i} = x_j^{t_k}, 0 \leq i < k < T\}$
    $[\hat{x}, \Delta^+, \Delta^-]$ :=OMIP_LNS($\mathcal{U}, x^{t_0}, \Delta^{UB}$)
  end while
  return $[\hat{x}, \Delta^+, \Delta^-]$

  function FMIP_LNS($\mathcal{F}, \hat{x}$)
    return $min\{\sum_i \Delta_i^+ + \Delta_i^- | Ax + I_m\Delta^+ - I_m\Delta^- = b, x_j = \hat{x}_j \ \forall j \in \mathcal{F}, x_j \in \mathbb{Z} \ \forall j \in \mathcal{I}\}$
  end function

  function OMIP_LNS($\mathcal{F}, \hat{x}, \hat{\Delta}$)
    return $min\{c^t x | Ax + I_m\Delta^+ - I_m\Delta^- = b, \sum_i \Delta_i^+ + \Delta_i^- \leq \hat{\Delta}, x_j = \hat{x}_j \ \forall j \in \mathcal{F}, x_j \in \mathbb{Z} \ \forall j \in \mathcal{I}\}$
  end function
```

memory is partitioned among parallel processors. As a result, processor sincronization and memory communication must be done via a message passing interface, such as MPI[16]. Processor coordination and communication becomes a problematic element in the algorithm design, inducing occasional overheads. Because our approach is synchronous, it allows for an efficient communication due to the availability of collective message passing primitives. Concretely, communication takes place after the parallel exploration and recombination. In both cases, a synchronous all-to-all broadcast is used.

3.2 Additional algorithmic components

Besides the parallelization, there are additional aspects to specify, such as the choice of starting solutions and variable fixings. We introduce general strategies for addressing them in the context of generic MIPs. However, their specification is independent of the parallel framework. Thus, it is possible to substitute them for more effective approaches when considering specific classes of problems with defined structures, such as scheduling or network flow problems.

3.2.1 Finding a starting solution

Alternating Criteria Search only requires a starting solution that is integer feasible. However, given that feasibility is one of the primary objectives of the heuristic, it is proposed to choose a starting solution that is as feasible as possible with respect to the objective function of FMIP. Many strategies can provide a starting solution for the algorithm. A common strategy solves the LP relaxation and rounds every fractional variable to the nearest integer. Since LP relaxations can potentially be very costly to solve, we propose a quick heuristic, Algorithm 2, that tries to minimize the infeasibility of a starting point. Within each iteration, subsets of variables are fixed to random integer values within bounds while the remaining ones are optimized towards feasibility. The algorithm terminates once all integer variables are fixed.

Algorithm 2 Starting solution heuristic

```
Input: Fraction of variables to fix $\theta$, $0 < \theta < 100$
Output: Starting integer-feasible solution $\hat{x}$
 1: $V$:=list of integer variables sorted by increasing bound range $u - l$
 2: $\mathcal{F}$:= top $\theta$ % of unfixed variables from $V$
 3: Bound $\hat{x}_f$ to a random integer value between $[l_f, u_f]$, $\forall f \in \mathcal{F}$
 4: while $\hat{x}$ is not integer feasible and $\mathcal{F} \neq \mathcal{I}$ do
 5:   $[x, \Delta^+, \Delta^-] := min\{\sum_i \Delta_i^+ + \Delta_i^- | Ax + I_m\Delta^+ - I_m\Delta^- = b, x_j = \hat{x}_j \ \forall j \in \mathcal{F}\}$
 6:   $\mathcal{Q}$:= index set of integer variables of $x$ with integer value
 7:   $\hat{x}_q = x_q, \forall q \in \mathcal{Q}$
 8:   $\mathcal{F} := \mathcal{F} \cup \mathcal{Q}$
 9:   $\mathcal{N}$:= top $\theta$ % of unfixed variables from $V$
10:   Bound $\hat{x}_n$ to a random integer value between $[l_n, u_n]$, $\forall n \in \mathcal{N}$
11:   $\mathcal{F} := \mathcal{F} \cup \mathcal{N}$
12: end while
13: return $\hat{x}$
```

Starting with a sorted list of variables by increasing bound range and an input parameter θ, the algorithm proceeds to fix the top θ% of variables to a random integer value within their bounds. With sorting, the goal is to drive binary variables integer first, given that binary decisions force integrality on other variables. Until all integer variables are fixed, the LP relaxation of FMIP is solved in order to optimize the unfixed variables towards feasibility. Consecutively, the variables that become integer are fixed. A minumum of θ% variables are fixed every iteration. Hence, the algorithm will require at most $\lceil \frac{100}{\theta} \rceil$ iterations to converge to a starting solution. The θ parameter controls a tradeoff of difficulty of the LP relaxations against the quality of the starting feasible solution.

3.2.2 The variable fixing scheme

Another crucial heuristic component is the variable fixing scheme, as it determines the difficulty and the efficacy of each Large Neighborhood Search. A desirable quality of the fixings is that they must not be too restrictive, as very few improvements will be found. At the same time, if not enough variables are fixed, the search space might be too large to find any improvements in a small amount of time. Neighborhood diversification is another key property, since parallel efficiency depends on it. In order to generate a large number of diverse neighborhoods early in the search, we use randomization instead of branch and bound information.

Devising a general method for fixing variables may be challenging, since there are many problem structures to consider, as illustrated in Figure 4. Problems range in structure, constraint matrix shape, number, and kinds of variables. Given these requirements, we propose a simple, yet intuitive variable fixing algorithm. It incorporates randomness, in order to satisfy the need for diversity and it allows the fixing of an adjustable number of variables. As shown in Algorithm 3, fixings are determined by selecting a random integer variable x' and fixing a consecutive set of integer variables starting from x' until a certain cap determined by an input parameter ρ is reached.

For any MIP we consider, its formulation is usually structured, and its variables are arranged consecutively depending on their type and their logical role. Consecutive sets of variables often belong to cohesive substructures within a problem. This is the case in network flow and routing problems where flow assignations for a particular entity are formulated successively. Similar properties can be found in formulations for scheduling problems. In our experience,

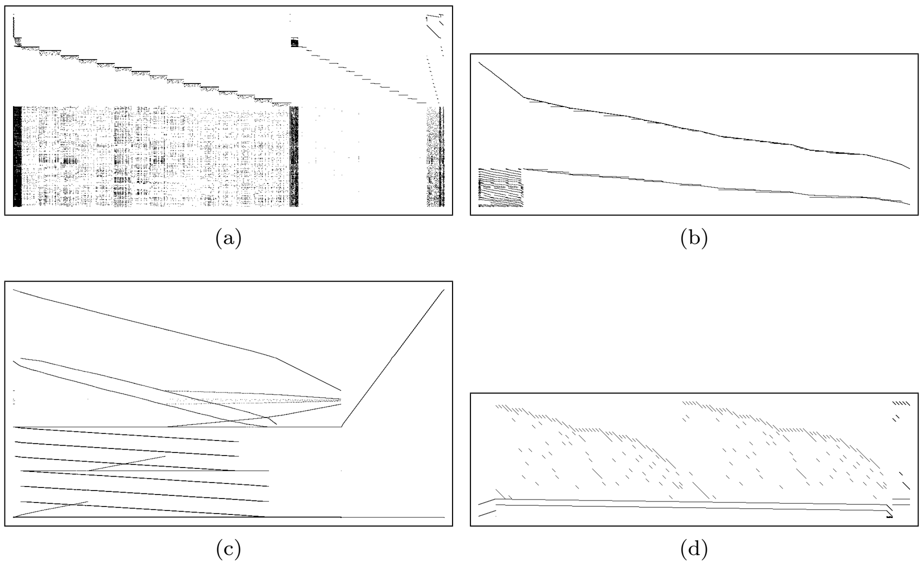

Figure 4: Constraint matrix diagrams that show the disparities in the problem structure of (a) Atlanta-ip, (b) Rail03, (c) Triptim2, and (d) N3-3. Blocks in the matrix are highlighted if the related variables (vertical axis) appear in the constraints, which are enumerated horizontally. Figures are courtesy of the MIPLIB library [18].

Algorithm 3 Variable Fixing Selection Algorithm

```
Input: Fraction of variables to fix $\rho$, $0 < \rho < 1$
Output: Set of integer indices $\mathcal{F}$
1: function RandomFixings($\rho$)
2:     $i$:= random element in $\mathcal{I}$
3:     $\mathcal{F}$:= first $\rho \cdot |\mathcal{I}|$ consecutive integer variable indices starting from $i$
4:     return $\mathcal{F}$
5: end function
```

our proposed variable selection often produces an easier subMIP, in which the fixings affect a subset of contiguous substructures and the remaining ones are left unfixed. Due to its simplicity, it is an efficient variable selection strategy. However, any permutation of rows and columns alters its effectiveness.

3.3 Framing Alternating Criteria Search within an exact algorithm

In the current parallel branch and bound paradigm, threads are focused on solving multiple subproblems in parallel, which has shown poor scalability [19]. We propose an alternative use of the parallel resources. We decouple the search for high quality solutions from the lower bound improvement by performing both tasks independently. Thus, a subset of the threads are allocated to an instance of the Branch and Bound solver focused on improving the lower bound, and our Alternating Criteria Search replaces the traditional primal heuristics. In the process, our parallel heuristic searches for solutions to the entire problem regardless of the variable fixings produced in the Branch and Bound tree, and supplies them to the solver. Both algorithms proceed to run concurrently until a time limit or optimality is reached. Communication between the parallel heuristic and the Branch and Bound is performed via MPI collective communications and new feasible solutions are added back via callbacks.

4 Experimental Results

In this section, we evaluate the performance and behavior of our algorithm in terms of solution quality, scalability, reproducibility and parallel efficiency. The framework is implemented in C++, using CPLEX 12.6.1 as a backbone solver. We compare our framework against different configurations of the state-of-the-art general purpose MIP solver CPLEX 12.6.1. Out of the 361 instances from the MIPLIB 2010 library[18], we select the 333 in which feasibility has been proven as a benchmark. MIPLIB classifies such instances by difficulty based on the time required to reach optimality. 206 easy instances are defined as the subset in which optimality has been proven in less than one hour using some MIP solver. 54 additional instances have also been solved, but not under the previous conditions (hard instances). The remaining 73 unsolved instances are classified as open. All of our computations are performed on an 8-node computing cluster, each with two Intel Xeon X5650 6-core processors and 24 GB of RAM memory.

4.1 Automating the choice of parameters

Three main parameters regulate the difficulty and the solution time of each LNS problem within the heuristic, and their appropriate selection is crucial for performance. We heuristically calibrate the settings for each instance automatically by executing multiple sample configurations and selecting the best performing one for the full run. The parameter θ regulates the amount of variables to be fixed during the initial solution generation process. Depending on the difficulty of the instance, θ ranged between 1% and 100%. In addition to θ, parameters $[\rho, t]$ determine the percentage of variables to be fixed and the time limit in each LNS. In this case, the configurator chose values between $[5\%, 5s]$ and $[95\%, 50s]$.

4.2 Evaluation of primal solution quality

The first set of results evaluate the quality of the provided solutions. Specifically, we compare the output obtained by our Parallel Alternating Criteria Search (PACS) with CPLEX. This may seem to be an unfair comparison, given the fact that CPLEX is an exact algorithm, while our heuristic focuses only on primal solutions. However, our intention is to set a benchmark against a known standard. In order to mitigate the disadvantage, CPLEX is set in its parallel distributed-memory setting in order to utilize all 96 available cores and with a focus on primal solutions (the emphasis on Hidden Feasible solutions setting). Settings are set to default otherwise. In Figure 5, we show a performance profile that illustrates the differences in solution quality output of both schemes. Let UB_{CPX} and UB_{PACS} be the objective value of the best solution found by CPLEX and our parallel heuristic respectively. We report solution differences in terms of the GAP percentage of improvement of UB_{PACS} with respect to UB_{CPX}:

$$Improvement = \frac{UB_{CPX} - UB_{PACS}}{UB_{CPX}} \cdot 100$$

Figure 5 displays the differences between the best solution found by both methods after different time limits. Specifically, each performance curve measures the cumulative percentage of instances among the total that reach or surpass certain amounts of GAP difference in favor $PACS$ (Improvement) or CPLEX (Deterioration). For example, $PACS$ produced solutions with a GAP improvement of at least 10% after 180 seconds for 17% of the problem instances. In contrast, only 3% of the instances present a deterioration of 10% or more in the same amount of time. Both schemes tie or show differences between -0.1% and 0.1% for the remaining percentage of instances. We use the space beyond the 100% GAP mark to accumulate the additional percentage of instances where only one scheme found a solution. In the case of Figure 5, after 180 seconds of execution there were a total of 7% of instances in which CPLEX failed to find a

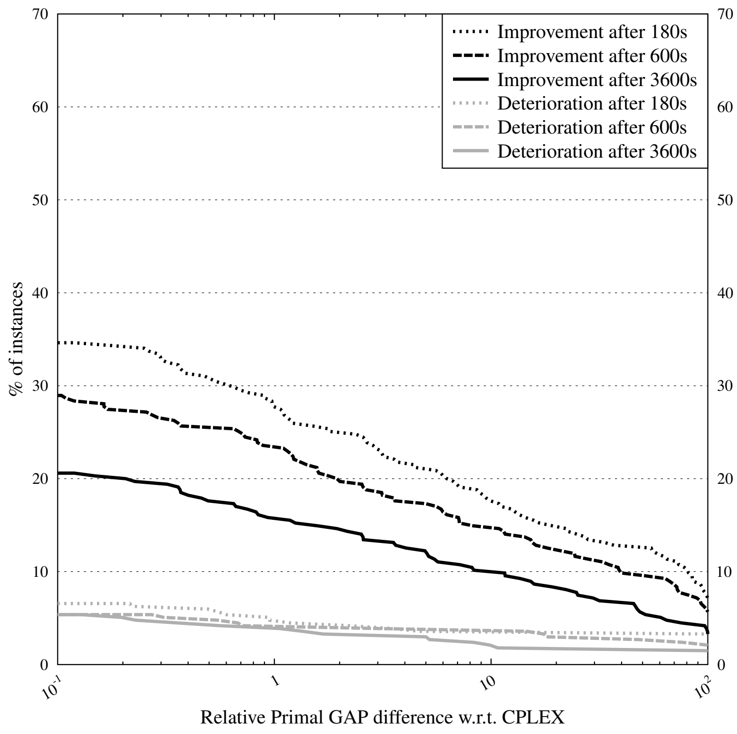

Figure 5: Improvement and Deterioration of primal solution quality w.r.t. CPLEX for all instances

solution but $PACS$ succeeded or the difference between solutions is more than a 100% of the GAP in favor of $PACS$'s solution.

Results show that our heuristic performs better for a substantial number of instances, whereas worse solutions are found in a relatively smaller subset of cases. One of our primary focuses is the ability to provide high quality solutions early in the search. After 3 minutes of execution, $PACS$ provides solution improvements for 35% of instances. At the time limit of one hour, this percentage decreases to 20% after CPLEX neutralizes the advantage for some of the instances. In contrast, worse solutions are only obtained in 7% of the cases.

In Figure 6, the comparison is restricted exclusively to hard and open instances. Results show the scalability of our heuristic in hard MIPs, as more than 60% of instances present improvements after 3 minutes of execution. The similarities between the 180 and the 600 second profiles indicate that the competitive advantage of $PACS$ is sustained throughout a large portion of the runtime.

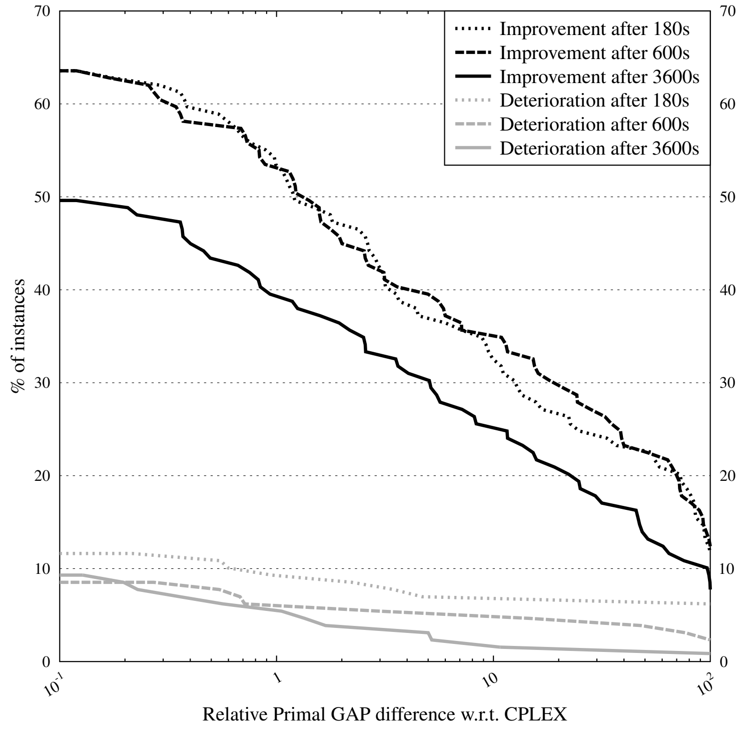

Figure 6: Improvement and Deterioration of primal solution quality in hard and open instances w.r.t. CPLEX

4.3 Framing Alternating Criteria Search within an exact algorithm

We test the performance of our heuristic when run in combination with an exact Branch & Bound algorithm. In this hybrid scheme, a fraction of the cores are dedicated to improving the primal incumbent and the rest are commited to computing the lower bound in the tree search.

The following charts demonstrate a direct performance comparison between parallel memory-distributed CPLEX using 96 threads, and our combined scheme. In the latter, 84 cores are allocated to the parallel heuristic while the remaining 12 threads are allocated to CPLEX. We report a performance comparison in terms of the difference between the optimality GAP provided by both algorithms: $GAP_{CPX} - GAP_{PACS}$. For any of the two algorithms X, its optimality GAP may be defined in terms of its best found upper and lower bound, UB_X and LB_X:

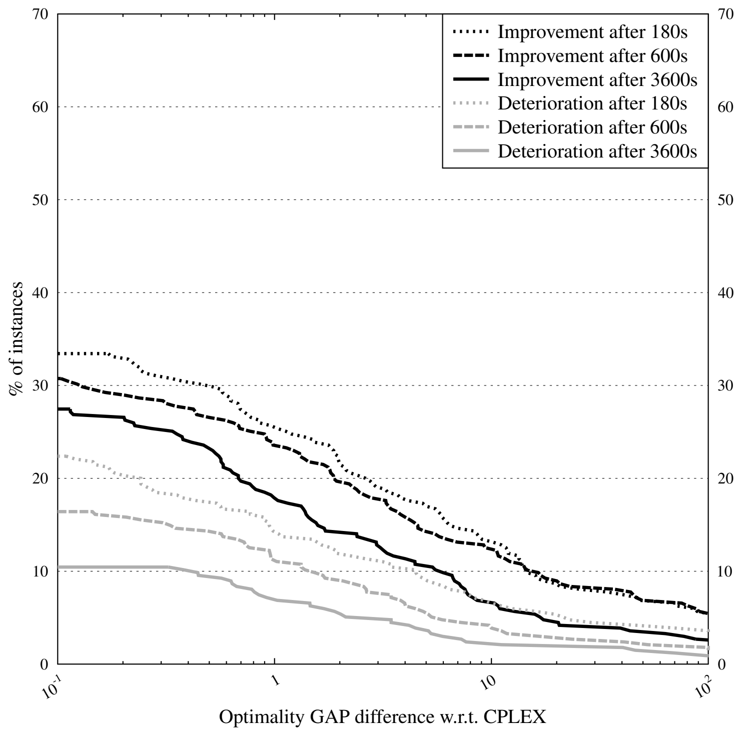

Figure 7: Improvement and Deterioration of optimality GAP w.r.t. CPLEX for all instances

$$GAP_X = \frac{UB_X - LB_X}{UB_X} \cdot 100$$

Figure 7 shows multiple performance profiles for different time cutoffs when all instances are considered. Both approaches outperform each other for different instance subsets, but our combined scheme performs better in more instances. After 10 minutes, our algorithm provides a better GAP for over 30% of the instances, while the opposite is true for 17% of the instances.

We observe a similar shift in performance when only the hard and open instances are considered, as shown in Figure 8. In this case, our ability to produce solutions of high quality early in the search allows us to achieve a smaller optimality GAP for over 60% of the instances after 10 minutes. At termination, over 15% of the instances show a competitive advantage of over 10% of the GAP. In contrast, CPLEX shows better performance for at most 4% of the instances.

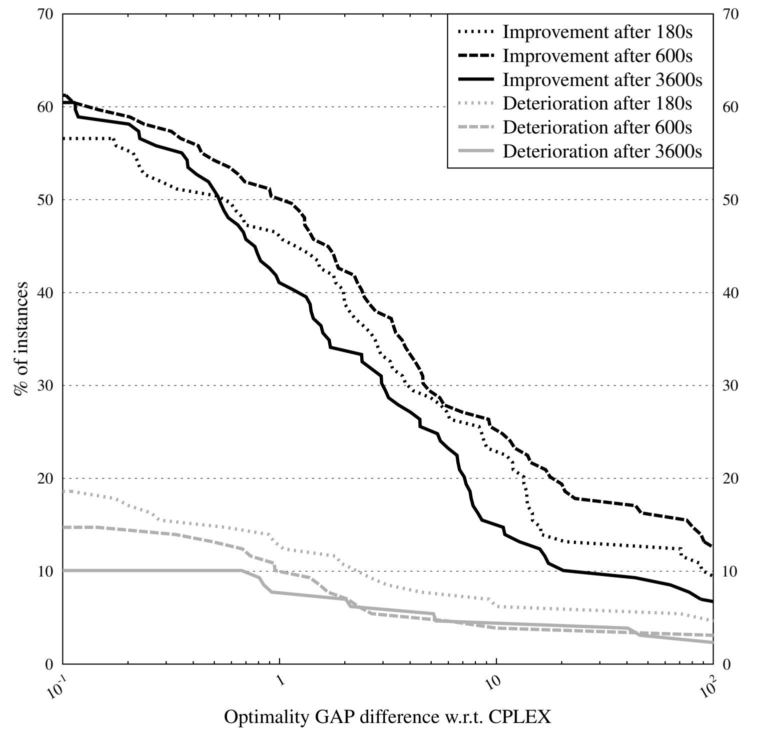

Figure 8: Improvement and Deterioration of optimality GAP in hard and open instances w.r.t. CPLEX

4.4 Additional performance tests: scalability, replicability and parallel efficiency

We examine the performance of Alternating Criteria Search from the perspective of scalability, replicability, and parallel efficiency. Due to the large amount of experiments required to evaluate this set of metrics, we reduce the test bed to a representative subset of the instances. The 15 instances shown in Table 1 are selected from all three difficulty categories.

4.4.1 Parallel scalability

Strong scalability is the ability to increase the algorithm's performance proportionally when parallel resources are incremented while the problem size remains fixed, and a particularly relevant metric is the speedup to cut off: a comparison of the time required by different processor configurations to reach certain solution quality. Let T_c^p be the time required for the algorithm to reach a cut-off

Table 1: Selected instances for Scaling experiments

Easy Instances	**Hard Instances**	**Open Instances**
a1c1s1	atm20-100	momentum3
bab5	germanrr	nsr8k
csched007	n3-3	pb-simp-nonunif
danoint	rmatr200-p5	t1717
map14	set3-20	ramos3

c when p processing nodes are used. We define the parallel speedup to cut off S_{SC} with respect to a baseline sequential configuration as $S_{SC} = \frac{T_c^1}{T_c^p}$. In Figure 9, we show the speedup demonstrated by the heuristic for several processor configurations ranging from a baseline of 12 cores to a total of 96. Among the runs for different processor configurations, the cutoff was determined to be the objective value of the best solution achieved by the worst performing run.

In terms of scaling, our heuristic shows a variable performance dependent on the characteristics of each individual instance. This behavior is expected since increasing the number of simultaneous searches does not guarantee a translation to faster improvements. In general, however, speedups are achieved more consistently as the difficulty of the problem increases. The addition of more cores sometimes exhibits a multiplicative effect. In most small instances, optimality is achieved quickly for all processor configurations, thus resulting in small speedup values.

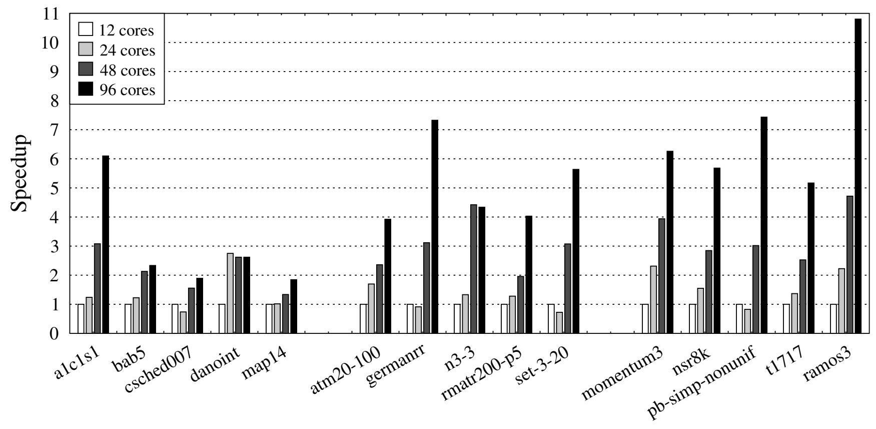

Figure 9: Parallel strong-scaling results

4.4.2 Replicability of the results

Our algorithm relies on randomness in order to diversify the set of fixings within each search. An important aspect to test is whether this diversification affects the variability of the outcome of the algorithm. We evaluate the consistency of 30 runs for each instance in the test set, each differing in the random seed used. Different time limits are set for each run, depending on the difficulty of the problem. Easy, hard and open instances are allowed 300, 1800, and 3600 seconds respectively. Table 2 summarizes the statistical results. For each instance, we report the optimality GAP calculated with the upper bound supplied by the heuristic and the best known lower bound. In general, standard deviations remain low throughout all difficulty categories. There is more variability as the problem difficulty increases.

Table 2: Replicability statistics

Problem	**Mean**	**Std Dev.**	**Min. Value**	**Median**	**Max. Value**
a1c1s1	0.01	0.00	0.01	0.01	0.01
bab5	0.01	0.00	0.01	0.01	0.01
csched007	0.00	0.00	0.00	0.00	0.00
map14	0.33	0.96	0.01	0.01	5.15
danoint	0.01	0.00	0.01	0.01	0.01
atm20-100	9.91	0.63	8.81	9.96	11.15
germanrr	0.25	0.04	0.24	0.24	0.37
n3-3	8.79	0.31	8.38	8.77	9.57
rmatr200-p5	16.16	0.12	16.11	16.11	16.46
set3-20	35.10	0.99	34.26	34.53	37.85
momentum3	84.62	0.92	82.97	84.37	86.84
nsr8k	8.34	0.68	7.23	8.34	10.06
pb-simp-nonunif	36.36	0.00	36.36	36.36	36.36
t1717	22.84	1.92	18.61	22.58	25.86
ramos3	131.27	0.38	130.42	131.23	131.93

4.4.3 Parallel load balancing

Load balancing is the property that measures the degree of uniformity of the work distribution among processors. Given the synchronous nature of our approach, an even difficulty of the subproblems is essential in order to ensure all processors optimize for an equivalent amount of time. In the case of the parallel heuristic, the size of each subproblem is regulated by the number of fixed variables and the imposed search time limit. Hence, a proper calibration of these parameters must ensure an even distribution of the workload.

The load balance of a parallel application can be evaluated as follows: Let the total execution time of a processor P_i be defined as the sum of the time spent performing useful computations (TU_{P_i}), communications (TC_{P_i}) and the synchronization time TS_{P_i} spent waiting for other processors to complete their computations. Then, we characterize the utilization of a processor U_{P_i} as the ratio of useful computation time over the total execution time:

$$U_{P_i} = \frac{TU_{P_i}}{TU_{P_i} + TS_{P_i} + TC_{P_i}}$$

We believe hard instances represent the worst-case scenario, since these are the ones that require the most computational effort and prolonged optimization times. Figure 10 shows the average core utilization for the hard instance momentum3. Performance results are displayed for different processor configurations as well as different time limit parameters. Time limit configurations are denoted as $C_{T_{LNS}-T_R}$, where T_{LNS} is the time allowed for each search and T_R is the time allowed to the recombination step. Results show that processors sustain high utilizations (above 95%) throughout the execution, even when large processor configurations are used. The setting with the shortest solution times remains the most efficient because smaller time penalties are paid due to early terminations.

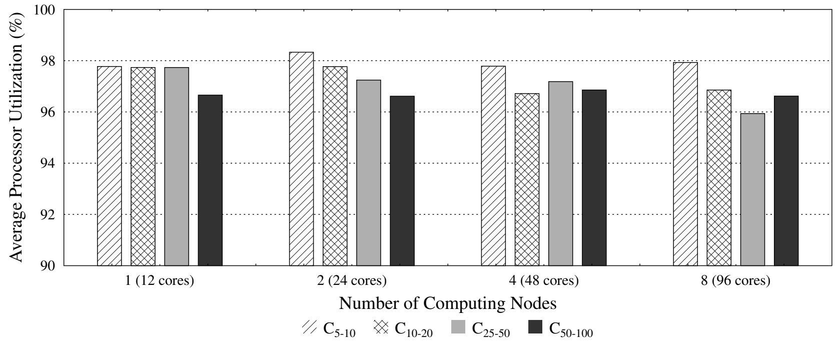

Figure 10: Parallel synchronization overhead in terms of average processor utilization for different parameter configurations

5 Conclusions

The combination of parallelism and simple Large Neighborhood Search schemes can provide a powerful tool for generating high quality solutions. The heuristic becomes especially useful in the context of large instances and time-sensitive optimization problems, where traditional branch and bound methods may not be able to provide competitive upper bounds and attaining feasibility may be challenging. Our method is a highly versatile tool, as it can be applied to any generic MIP as a standalone heuristic or in the context of an exact algorithm. Many algorithmic ideas contribute to the competitiveness of our approach, such as the introduction of two auxiliary MIP transformations and fixing diversifications as the main source of parallelism. The proposed algorithm could benefit from further improvements. For example, the variable fixing scheme could be enhanced with more application-specific variable selection methods. We hope this work sparks interest and motivation to investigate whether better parallelizations are possible in order to speed up the optimization process. In the future, we plan to evaluate other components of the branch and bound search that may profit from parallelism.

6 Acknowledgements

This research has been supported in part by ExxonMobil Upstream Research Company and the Air Force Office of Scientific Research.

References

[1] Achterberg, T., Berthold, T.: Improving the feasibility pump. Discrete Optimization **4**(1), 77–86 (2007)

[2] Bader, D.A., Hart, W.E., Phillips, C.A.: Parallel algorithm design for branch and bound. In: Tutorials on Emerging Methodologies and Applications in Operations Research, p. Chapter 5. Springer (2005)

[3] Baena, D., Castro, J.: Using the analytic center in the feasibility pump. Operations Research Letters **39**(5), 310–317 (2011)

[4] Bertacco, L., Fischetti, M., Lodi, A.: A feasibility pump heuristic for general mixed-integer problems. Discrete Optimization **4**(1), 63–76 (2007)

[5] Berthold, T.: Primal heuristics for mixed integer programs. Diploma Thesis, Technische Universitat Berlin (2006)

[6] Berthold, T.: Rens. Mathematical Programming Computation **6**(1), 33–54 (2014)

[7] Boland, N.L., Eberhard, A.C., Engineer, F.G., Fischetti, M., Savelsbergh, M.W., Tsoukalas, A.: Boosting the feasibility pump. Mathematical Programming Computation pp. 1–25 (2011)

[8] Corporation, I.B.M.: Ibm cplex optimizer (2015). `http://www-01.ibm.com/software/commerce/optimization/cplex-optimizer/`

[9] Danna, E., Rothberg, E., Le Pape, C.: Exploring relaxation induced neighborhoods to improve mip solutions. Mathematical Programming **102**(1), 71–90 (2005)

[10] Fischetti, M., Glover, F., Lodi, A.: The feasibility pump. Mathematical Programming **104**(1), 91–104 (2005)

[11] Fischetti, M., Lodi, A.: Local branching. Mathematical Programming **98**(1-3), 23–47 (2003)

[12] Fischetti, M., Lodi, A.: Heuristics in mixed integer programming. John Wiley & Sons, Inc. (2010)

[13] Fischetti, M., Monaci, M.: Proximity search for 0-1 mixed-integer convex programming. Journal of Heuristics **20**(6), 709–731 (2014)

[14] Fischetti, M., Salvagnin, D.: Feasibility pump 2.0. Mathematical Programming Computation **1**(2-3), 201–222 (2009)

[15] Ghosh, S.: Dins, a mip improvement heuristic. In: Integer Programming and Combinatorial Optimization, *Lecture Notes in Computer Science*, vol. 4513, pp. 310–323. Springer Berlin Heidelberg (2007)

[16] Gropp, W., Lusk, E., Doss, N., Skjellum, A.: A high-performance, portable implementation of the MPI message passing interface standard. Parallel computing **22**(6), 789–828 (1996)

[17] Koc, U., Mehrotra, S.: Generation of feasible integer solutions on a massively parallel computer. Article in submission

[18] 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. Mathematical Programming Computation **3**(2), 103–163 (2011). URL `http://mpc.zib.de/index.php/MPC/article/view/56/28`

[19] 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)

[20] Naoum-Sawaya, J.: Recursive central rounding for mixed integer programs. Computers & Operations Research **43**(0), 191 – 200 (2014)

[21] Optimization, G., et al.: Gurobi optimizer (2015). `http://www.gurobi.com`

[22] Rothberg, E.: An evolutionary algorithm for polishing mixed integer programming solutions. INFORMS Journal on Computing **19**(4), 534–541 (2007)

[23] Shinano, Y., Achterberg, T., Berthold, T., Heinz, S., Koch, T.: Parascip: a parallel extension of scip. In: Competence in High Performance Computing 2010, pp. 135–148. Springer (2012)