ParaILP: A Parallel Local Search Framework for Integer Linear Programming with Cooperative Evolution Mechanism

Peng Lin[1,2], **Mengchuan Zou**[1], **Zhihan Chen**[1,2], **Shaowei Cai**[1,2*]
[1] Key Laboratory of System Software (Chinese Academy of Sciences) and State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing, China
[2] School of Computer Science and Technology, University of Chinese Academy of Sciences, Beijing, China
{linpeng, zoumc, chenzh, caisw}@ios.ac.cn,

Abstract

The integer linear programming (ILP) problem is a fundamental research topic in operations research, and the local search method is an important class of algorithms for quickly solving many combinatorial optimization problems. With rapidly increasing computing power, parallelism turns out to be a promising approach to enhancing the efficiency of problem-solving. However, rare studies investigate parallel local search algorithms for solving the general ILP problem. We propose the first parallel local search framework (ParaILP) for solving the general ILP problem, based on two novel ideas: a new initialization method named polarity initialization to construct different initial solutions for local search threads and a cooperative evolution mechanism for managing and generating high-quality solutions using information shared by different threads. Extensive experiments demonstrate that ParaILP is significantly better than the state-of-the-art academic parallel solvers FiberSCIP and HiGHS, and is competitive with the state-of-the-art commercial parallel solver Gurobi. Experiments are also conducted to analyze the parallelization scalability and the effectiveness of our techniques.

1 Introduction

The integer linear programming (ILP) describes the optimization problem of a linear objective function over variables that are restricted to being integers and constrained by linear constraints. The ILP problem is known to be NP-hard [Kannan and Monma, 1978], and due to its strong expressive ability, many NP-hard problems, such as FJSP [Roshanaei *et al.*, 2013] and TSP [Dantzig *et al.*, 1954], can be formalized in the form of ILP and solved efficiently by ILP solvers. In addition, ILPs are widely used in real industry and have close ties to operation research applications [Wolsey, 2020].

Previous studies have explored both complete and heuristic algorithms for ILP. As for complete solving, the branch-and-bound framework [Land and Doig, 2010; Lawler and Wood, 1966] is commonly admitted and is the primary framework used in modern commercial complete solvers. However, the worst-case exponential running time of complete solving algorithms makes complete solving suffer from long running time as the problem size increases, limiting the use of complete algorithms for large-scale instances [Genova and Guliashki, 2011].

Local search is a typical heuristic algorithm for many combinatorial optimization problems and is suitable for scenarios requiring obtaining a good solution for large-scale instances quickly [Jacobs and Brusco, 1995; Vaessens *et al.*, 1996; Merz and Freisleben, 1997; Dorne and Hao, 1998; Stützle, 2006]. However, there are few local search works for general ILP problem-solving. To the best of our knowledge, there are only [Verachi and Prestwich, 2008] and [Lin *et al.*, 2023] that studied local search for general ILP.

Meanwhile, with the increasing computing power of multi-core computer structures, algorithm design adapted to parallel environments brings excellent benefits to problem-solving in modern computing architectures. For parallel ILP solving, the most common approach is divide-and-conquer, where the original problem is decomposed into sub-problems, and each solver solves some sub-problems [Bixby *et al.*, 1995]. The node-level and sub-tree parallelizations of the branch-and-bound framework belong to this type and were adopted by most commercial solvers, such as Gurobi and CPLEX. Due to the large size of the search tree, these complete parallel algorithms are still incompetent to solve large instances within short time limits [Xu *et al.*, 2009]. However, to the best of our knowledge, local search algorithms, which are known to solve many hard problems efficiently, have not been investigated for solving general ILP in parallel.

In this work, we propose the first parallel local search framework for ILP, dubbed ParaILP. It is based on a novel initialization method named polarity initialization, and a cooperative evolution mechanism that manages and generates high-quality solutions throughout the search process. ParaILP uses the state-of-the-art sequential local search algorithm LocalILP [Lin *et al.*, 2023] as a subroutine, but any other sequential local search algorithm could be plugged in as well.

Experiments conducted on the MIPLIB dataset show the effectiveness of ParaILP in finding a high-quality feasible solution quickly for large-scale hard ILP problems. We compare ParaILP with the state-of-the-art parallel ILP solvers, including academic and commercial solvers. The experimen-

*Corresponding author

tal results demonstrate the excellent performance of ParaILP, which significantly outperforms academic solvers and is competitive with commercial solvers. We also compare ParaILP with the state-of-the-art local search algorithm Local-ILP, indicating a significant improvement. Experiments are also conducted to analyze the effectiveness of each component in ParaILP and the scalability of parallel solvers when adopting different numbers of threads.

The main contributions of this work are as follows:

1. We propose the first parallel local search framework and develop an efficient parallel solver for solving general ILP (ParaILP). Our framework could be easily extended by plugging other sequential local search algorithms as a subroutine to create new solvers.
2. We propose a new initialization method, named polarity initialization, that makes use of the influence of variables on the objective function and constraints, to decide whether to assign larger or smaller values to variables.
3. We propose a cooperative evolution mechanism based on population maintenance policy and solution evolution process to manage and generate high-quality solutions.
4. Experiments show that ParaILP significantly outperforms the state-of-the-art academic parallel solvers FiberSCIP and HiGHS, and is competitive with the state-of-the-art commercial parallel solver Gurobi.

2 Preliminaries

2.1 Integer Linear Programming

Let $m, n \in \mathbb{N}^+$, $\boldsymbol{A} \in \mathbb{R}^{m \times n}$, $\boldsymbol{b} \in \mathbb{R}^m$, $\boldsymbol{c} \in \mathbb{R}^n$, and $\boldsymbol{l}, \boldsymbol{u} \in \mathbb{R}^n$. The optimization problem described by:

$$\begin{aligned} \min \quad & \boldsymbol{c}^\top \boldsymbol{x} \\ \text{subject to:} \quad & \boldsymbol{A}\boldsymbol{x} \leq \boldsymbol{b} \\ & \boldsymbol{l} \leq \boldsymbol{x} \leq \boldsymbol{u} \\ & \boldsymbol{x} \in \mathbb{Z}^n \end{aligned} \tag{1}$$

is an instance of general integer linear programming (ILP).

We call $\boldsymbol{c}^\top \boldsymbol{x}$ the objective function. $\boldsymbol{l}$ and $\boldsymbol{u}$ are called the lower and upper bounds of the variables $\boldsymbol{x}$. We call $l_j \leq x_j \leq u_j$ the global bounds, $\boldsymbol{A}\boldsymbol{x} \leq \boldsymbol{b}$ the linear constraints, and $\boldsymbol{x} \in \mathbb{Z}^n$ the integrality constraints of the ILP. A row $\boldsymbol{A_i}$ of the matrix $\boldsymbol{A}$ is identified with the linear constraint $\boldsymbol{A_i}\boldsymbol{x} \leq b_i$. The coefficient of x_j in $\boldsymbol{A_i}\boldsymbol{x} \leq b_i$ is A_{ij}, and $\boldsymbol{A_i}\boldsymbol{x} \leq b_i$ contains x_j if $A_{ij} \neq 0$. Integer linear programming aims to minimize the objective function while satisfying all constraints. A maximization problem can easily be transformed into this minimization formulation.

A complete solution (solution for short) $\boldsymbol{s}$ of an ILP instance is a vector of values assigned for each variable, and s_j denotes the value of x_j. $\boldsymbol{s}$ satisfies $\boldsymbol{A_i}\boldsymbol{x} \leq b_i$ if $\boldsymbol{A_i}\boldsymbol{s} \leq b_i$. $\boldsymbol{s}$ is a feasible solution if and only if it satisfies all constraints.

2.2 Local Search and Evolutionary Algorithm

The local search algorithm and the evolutionary algorithm both play vital roles in solving combinatorial optimization problems. Local search starts with a given solution; and then, iteratively implements local modifications to explore the neighborhood of the current solution and move towards a better one, to find a high-quality feasible solution.

In contrast to the local search focusing on the current solution, the evolutionary algorithm maintains a set of solutions called a population, and generates offspring through operators that resemble natural evolution, such as mutation and crossover actions. A fitness function is designed to evaluate the quality of solutions and decide whether they could stay in the population.

These two methods are integrated into our framework: we utilize local search to find new solutions, and use evolution mechanism to manage and generate high-quality solutions.

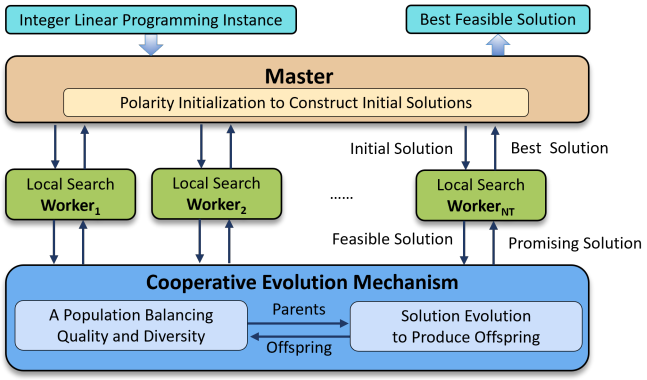

Figure 1: Our parallel local search framework for solving general integer linear programming with cooperative evolution mechanism.

3 Parallel Framework for Solving ILP

As shown in Figure 1, we propose an efficient parallel framework for solving integer linear programming problems, which takes an ILP instance as an input and outputs the highest-quality feasible solution found through the search process. The framework includes a master thread, *NT* worker threads, and a collaboration thread. Each type of thread has its own functionalities, which are explained in the following.

Master thread. The master thread reads the problem instance and constructs shared data structures, including basic information about variables and constraints. The master thread also assigns parameters and initial solutions for each local search worker thread. Here, diversity strategies are adopted to initialize workers with different settings to explore a broader search space and strengthen the framework's robustness. We proposed a new method to generate initial solutions named **polarity initialization** to construct *NT* different initial solutions, which are distributed to each worker thread. When the solving time limit is reached, the master thread collects the best feasible solutions found by each thread and outputs the feasible solution with the highest-quality objective function value.

Worker threads. Each worker thread performs a local search algorithm to explore the search space. It starts with the initial solution received from the master thread and iteratively modifies the current solution to find high-quality solutions. Whenever a worker finds a new feasible solution, it passes the solution to the collaboration thread, trying to put it

into the population. Whenever a worker thread is trapped in a local optimum and cannot find a better solution for enough steps, the worker will get a solution shared from the collaboration thread as a starting point and restart the local search process. At the end of the time limit, each worker thread synchronizes its history-best feasible solution with the master.

Collaboration thread. The collaboration thread is responsible for performing the **cooperative evolution mechanism**. It collects feasible solutions from worker threads, and manages a population balancing quality and diversity. In addition, it attempts to produce new promising solutions through the solution evolution process by exploiting information from high-quality solutions in the population. When a worker struggles to make progress, it also shares a high-quality solution to help the worker escape from the local optimum.

In the following two sections, we present the details of polarity initialization and cooperative evolution mechanism.

4 New Initialization Method

As for solving ILP in parallel, we propose a novel approach called polarity initialization, to construct different initial solutions for each local search worker thread. We try to generate high-quality solutions by considering the polarity of variables, which refers to their negative or positive influence to the objective function and constraints. The provided information is utilized to compute the initial values of the variables.

4.1 Influence Factors and Polarity

We try to capture the influence of every variable to the objective function and constraints. For the objective function $\boldsymbol{c}^\top \boldsymbol{x}$ or a constraint $\boldsymbol{A_i x} \leq b_i$, we regard the influence of a variable x_j as the product of the difference between its upper and lower global bounds (i.e., $u_j - l_j$), multiplied by its coefficient (i.e., c_j or A_{ij}). This quantity reflects the range of difference that the change of this variable may bring to $\boldsymbol{c}^\top \boldsymbol{x}$ or $\boldsymbol{A_i x}$. To evaluate the relative importance of a variable among all variables, we take the ratio of a variable's influence over all variables in the objective function or a constraint, deriving the concept of the influence factor:

Definition 1. *The **influence factor** of variable x_j in the objective function $\boldsymbol{c}^\top \boldsymbol{x}$, denoted as IO_j, is defined as*

$$IO_j = \frac{(u_j - l_j) \cdot c_j}{\sum_{k=1}^{n} (u_k - l_k) \cdot |c_k|} \tag{2}$$

Similarly, the influence factor of variable x_j in constraint $\boldsymbol{A_i x} \leq b_i$, denoted as IC_{ij}, is defined as

$$IC_{ij} = \frac{(u_j - l_j) \cdot A_{ij}}{\sum_{k=1}^{n} (u_k - l_k) \cdot |A_{ik}|} \tag{3}$$

For the objective function $\boldsymbol{c}^\top \boldsymbol{x}$ or a constraint $\boldsymbol{A_i x} \leq b_i$, the influence factor of the variables possesses the following properties: (1) If the sign of the influence factor of x_j is positive (i.e., $IO_j > 0$ or $IC_{ij} > 0$), a smaller value of x_j is more favorable for improving the quality of the objective function or satisfying the constraint, and vice versa. (2) x_j with a larger absolute value of the influence factor causes a larger change in $\boldsymbol{c}^\top \boldsymbol{x}$ or $\boldsymbol{A_i x}$, indicating a more significant impact of x_j on the objective function or the constraint, and vice versa.

When constructing initial solutions, to decide whether to assign a larger or smaller value to a variable, we consider both its influence to the objective function and also the constraints; thus, we propose the idea of the polarity of variables to summarize the influences:

Definition 2. *Let m_j denote the number of constraints containing x_j, the **polarity** of variable x_j, denoted as P_j, is defined as*

$$P_j = \frac{\sum_{i=0}^{m} IC_{ij}}{m_j} + IO_j \tag{4}$$

The polarity P_j synthesizes the influences of x_j on relevant constraints and the objective function. If the polarity of x_j is negative, taking the maximization of x_j brings overall advantageous influences to relevant constraints and the objective function, and vice versa.

Algorithm 1: Polarity Initialization

Input: Integer linear programming instance Q
Output: the set of NT initial solutions $\boldsymbol{S}$ for Q

1 **for** $j \leftarrow 1$ **to** n **do**
2 Calculate the polarity P_j according to **Eq. 4**
3 **foreach** $k \leftarrow 1$ **to** NT **do**
4 Let $\boldsymbol{s}$ be the k-th solution in $\boldsymbol{S}$
5 **if** $P_j < 0$ *and* $u_j \neq +\infty$ **then**
6 $s_j \leftarrow u_j$
7 **else if** $P_j > 0$ *and* $l_j \neq -\infty$ **then**
8 $s_j \leftarrow l_j$
9 **else**
10 $s_j \leftarrow$ the integer closest to 0 within $[l_j, u_j]$
11 **if** $0 \in [l_j, u_j]$ *and* $random(0,1) < dp$ **then**
12 $s_j \leftarrow 0$
13 **return** *the resulting set of NT initial solutions $\boldsymbol{S}$*

4.2 Polarity Initialization

Based on the influence factors and polarity, the master performs the polarity initialization to construct NT different initial solutions and distributes them to each local search worker.

Algorithm 1 details polarity initialization in pseudo-code. For each variable, polarity initialization assigns it to one of its global bounds based on its polarity (lines 3-8). Note that if the polarity of a variable equals 0 or the preferred assignment is unbounded, it is assigned to the integer closest to 0 within the global bound, as 0 is the simplest value (lines 9-10). To diversify the search space of each local search worker thread, we introduce random perturbations to generate different initial solutions: if 0 is within its global bounds, with probability dp, it is randomly assigned to 0, where dp is a parameter that controls the degree of perturbation (lines 11-12).

5 Cooperative Evolution Mechanism

We now present our cooperative evolution mechanism based on population maintenance and solution evolution. It uses a

population to store the feasible high-quality solutions found by each worker, and tries to produce better solutions by evolutionary operations on solutions in the population. The cooperative evolution mechanism comprises the information exchanging scheme, the solution evolution process, and the population maintenance policy. We then present each component.

5.1 Information Exchanging Scheme

The collaboration thread is responsible for performing the cooperative evolution mechanism. Here we introduce the information exchanging scheme between the collaboration thread and the local search worker threads, including the receiving and sharing process.

For receiving, whenever a local search worker finds a new feasible solution, it transfers the solution to the collaboration thread. Based on the population maintenance policy, the cooperative evolution mechanism determines whether to accept it as a new individual into the population (see Section 5.3).

For sharing, whenever a worker thread fails to improve its solution for L iterations, the collaboration thread shares a solution with that worker thread as the restarting point for its local search, where L is a parameter. To balance feasibility and diversity, we design two sharing schemes:

Elite sharing: The collaboration thread shares a randomly selected solution from the population with the worker. Given that the population maintenance policy ensures that all solutions in the population are of high quality and feasible, the worker can explore new solutions based on feasibility.

Offspring sharing: The collaboration thread performs the evolution operations (see Section 5.2) to produce an offspring solution and shares it with the worker. The offspring may not be a feasible solution, but it is derived from crossover and mutation, thus reflecting the exploitation of high-quality information and the exploration of new search areas.

The collaboration thread selects one of the sharing schemes with equal probability and shares the solution with the trapped local search worker thread.

5.2 Solution Evolution Process

Now, we present the solution evolution process of the cooperative evolution mechanism that generates new solutions from high-quality solutions in the population. Our strategy consists of three steps: random parent selection, uniform crossover, and bound-aware mutation.

Random parent selection. Since all solutions in the population are highly competitive, to enhance the diversity of the system, we randomly select two solutions $\boldsymbol{s}^1$ and $\boldsymbol{s}^2$ as parents instead of favoring those with higher quality.

Uniform crossover. To leverage the information from two selected parents, we employ uniform crossover to produce an offspring $\boldsymbol{s}^o$. Specifically, for variable x_j, if $s_j^1 = s_j^2$, the assignment from the parents is retained, i.e., $s_j^0 = s_j^1 = s_j^2$; otherwise, s_j^0 is randomly assigned as either s_j^1 or s_j^2. Through crossover, $\boldsymbol{s}^0$ carries information from the parents.

Bound-aware mutation. The mutation is applied to $\boldsymbol{s}^0$ to explore new promising search space. To prevent excessive mutation that deviates too far from the feasible regions, we require that the mutation satisfy the basic global bounds of variables, i.e., $\boldsymbol{l} \leq \boldsymbol{x} \leq \boldsymbol{u}$. For variable x_j, with probability mp, $\boldsymbol{s}^0$ is randomly assigned an integer value within $[l_j, u_j]$, where mp is a parameter that controls the degree of mutation. The detailed pseudo-code is shown in Algorithm 2.

Algorithm 2: Solution Evolution Process

```
Input: Population P
Output: An offspring solution s^o
Randomly Select 2 solution s^1 and s^2 from P
for j = {1, ..., n} do
    if s_j^1 = s_j^2 then
        Inherit the same value: s_j^o ← s_j^1
    else
        Uniform crossover: randomly assign s_j^o to
         either s_j^1 or s_j^2
    if random(0, 1) < mp then
        Bound-aware mutation: assign s_j^o to an
         random integer value within [l_j, u_j]
```

5.3 Population Maintenance Policy

The population maintenance policy controls the admission of a solution into the population. It is activated in two situations: when a solution is received from a local search worker thread and when an offspring solution is produced by evolution process. To determine whether to accept a new individual joining the population, we consider three factors to evaluate a solution: the objective value, the informative degree, and the differences from other solutions.

We use $Q(\boldsymbol{s})$ to measure a solution's objective quality and informative degree. To capture the character of the quality of the solution, $Q(\boldsymbol{s})$ is initially set to $-\text{obj}(\boldsymbol{s})$. Then we use a penalty mechanism to make $Q(\boldsymbol{s})$ also consider the informative degree: whenever a solution's information is utilized (i.e., shared as a feasible solution in elite sharing with worker threads, or selected as a parent in offspring sharing for producing offspring), $Q(\boldsymbol{s})$ is penalized. Specifically, if $Q(\boldsymbol{s}) > 0$, $Q(\boldsymbol{s})$ is updated as $Q(\boldsymbol{s}) \times (1 - \beta)$; if $Q(\boldsymbol{s}) < 0$, $Q(\boldsymbol{s})$ is updated as $Q(\boldsymbol{s}) \times (1 + \beta)$; if $Q(\boldsymbol{s}) = 0$, $Q(\boldsymbol{s})$ is unchanged, where β is a given parameter to adjust the degree of punishment. We see that the more times a solution is used, the more its $Q(\boldsymbol{s})$ is reduced.

We now analyze the influence and the proposed way to update $Q(\boldsymbol{s})$: Assuming there are few population updates but many solutions shared with workers, this may result in premature convergence. Our penalty mechanism penalizes solutions participating in elite sharing and offspring sharing, decreasing their $Q(\boldsymbol{s})$ values. As a result, the threshold for population updating gets lower, thus increasing the frequency of population updating and balancing the exploitation of elite solutions and the exploration of new search areas.

We use $D(\boldsymbol{s})$ to measure the difference between a solution $\boldsymbol{s}$ and other solutions in the population:

$$D(\boldsymbol{s}) = \sum_{s' \in \text{Population}, s' \neq s} \sum_{j=1}^{n} |s_j - s'_j| \tag{5}$$

Let $R_Q(\boldsymbol{s})$ be the rank of $Q(\boldsymbol{s})$ value when comparing $\boldsymbol{s}$ with all solutions in the population, and $R_D(\boldsymbol{s})$ the rank of $D(\boldsymbol{s})$, respectively. We define the fitness function to evaluate the overall quality of a solution as follows:

$$R(\boldsymbol{s}) = R_Q(\boldsymbol{s}) \cdot p + R_D(\boldsymbol{s}) \cdot (1-p) \tag{6}$$

where p is a parameter to adjust the weight of two factors.

We now introduce the population maintenance policy based on the fitness function $R(\boldsymbol{s})$. As shown in Algorithm 3, initially, the population is empty. When a new feasible solution is received from a local search worker or is produced during the solution evolution, it can be added to the population until the population size reaches ps, where ps is a parameter to control population size. Afterward, the population is kept at the size of ps. Specifically, when a new solution attempts to join the population, it is evaluated by the fitness function $R(\boldsymbol{s})$ to decide whether to accept it as a new member of the population or not. If its fitness is better than the worst solution in the current population, it will replace that solution.

Algorithm 3: Population Maintenance Policy

Input: Population P and a new feasible Solution $\boldsymbol{s}^0$
Output: Updated population P

Tentatively add $\boldsymbol{s}^0$ to population P: $P \leftarrow P \cup \boldsymbol{s}^0$;
if $|P| <= ps$ **then return** ;
for *each solution* $\boldsymbol{s}^i$ *in population* P **do**
  Calculate the fitness value $R(\boldsymbol{s}^i)$ according to **Eq. 6**
Identify the worst $\boldsymbol{s}^w$ with the least fitness value: $\boldsymbol{s}^w = arg\ min\{R(\boldsymbol{s}) | \boldsymbol{s} \in P\}$;
if $\boldsymbol{s}^w \neq \boldsymbol{s}^0$ **then** Replace $\boldsymbol{s}^w$ with $\boldsymbol{s}^0$: $P \leftarrow P \setminus \boldsymbol{s}^w$;
else
  Expel $\boldsymbol{s}^0$ which failed to update P: $P \leftarrow P \setminus \boldsymbol{s}^0$;

6 Experiments

To evaluate the performance of the proposed framework ParaILP, we compare ParaILP with the state-of-the-art parallel ILP solvers, including academic and commercial solvers. The experimental results demonstrate the excellent performance of our proposed algorithm for the ability to find a high-quality feasible solution quickly, which significantly outperforms academic solvers and is competitive with commercial solvers. We also compare ParaILP with the state-of-the-art sequential local search algorithm Local-ILP by adopting the same number of threads, indicating a significant improvement. The results also show the effectiveness of each component in ParaILP, including the polarity initialization, and the cooperative evolution mechanism. Additionally, experiments are conducted to analyze the scalability of parallel solvers when adopting different threads.

6.1 Experimental Setup

ParaILP is implemented in C++ and compiled in g++. We use the sequential local search algorithm Local-ILP[1] [Lin *et al.*, 2023] as a subroutine to implement the worker threads. There are six parameters in ParaILP: mp for evolution mutation, L for worker thread restart, β for fitness punishment, p for fitness weight, dp for perturbation degree, and ps for population size. We tuned parameters with the automatic configuration tool irace [López-Ibáñez *et al.*, 2016] on 20% randomly sampled instances, with parameter domains as follows: [0.1,1] for mp, [1000, 10000] for L, [1e-6,1] for β, [0.1,1] for p, [0.1,1] for dp, and [5, 20] for ps. Finally, the default settings of these parameters are as follows: mp = 0.25, L = 2000, β = 1e-4, p = 0.2, dp = 0.5, and ps = 7 for all benchmarks. Detailed results and the sourced code are reported in github[2].

We tested the algorithms on MIPLIB, the standard benchmark for ILP, selecting the ILP instances marked hard and open on a union dataset of MIPLIB 2003[3] [Achterberg *et al.*, 2006], MIPLIB 2010[4] [Koch *et al.*, 2011], and MIPLIB 2017[5] [Gleixner *et al.*, 2021]. As ParaILP is not a complete solver, we excluded the infeasible instances, resulting in a benchmark of 121 instances. Each instance is calculated by each algorithm with time limits of 10, 60, and 300 seconds, which is consistent with the settings in [Lin *et al.*, 2023]. By default, the number of physical threads is set to 32, the same as the setting of the latest parallel track of SAT competition[6].

We use 3 widely used metrics in comparison to evaluate the ability to find a high-quality feasible solution quickly:

#Feas: the number of instances where a solver can find a feasible solution within the given time limit.

#Win: the number of instances in which the solver yields the best solution among all the solvers.

P(T): the primal integral *P(T)* [Berthold, 2013] is a well-established measure to evaluate the performance of ILP solvers, which depends on the quality of solutions found during the solving process as well as on the time points when they are found. It can be interpreted as the average solution quality during a time limit. Smaller values indicate that high-quality solutions were found earlier.

All experiments are carried out on a server using two AMD EPYC 7763 CPUs @ 2.45GHz with 128 physical cores and 1TB RAM, running the Ubuntu 20.04 Linux operation system. The best results in the tables appear in **bold**.

6.2 Comparisons with Parallel ILP Solvers

We compare ParaILP with three state-of-the-art parallel ILP solvers according to the latest result of Hans Mittelmann's Benchmark[7], which is a famous professional solver rating website. There are 2 academic solvers (i.e., HiGHS[8] and FiberSCIP[9]) and a commercial solver (i.e., Gurobi[10]):

HiGHS [Huangfu and Hall, 2018]: a high-performance parallel solver for large-scale sparse ILP (version 1.5.3).

[1]https://github.com/shaowei-cai-group/Local-ILP/

[2]https://github.com/shaowei-cai-group/ParaILP

[3]https://miplib2010.zib.de/miplib2003/index.php

[4]https://miplib2010.zib.de

[5]https://miplib.zib.de

[6]https://satcompetition.github.io/2023/tracks.html

[7]https://mattmilten.github.io/mittelmann-plots/

[8]https://github.com/ERGO-Code/HiGHS

[9]https://ug.zib.de

[10]https://www.gurobi.com/solutions/gurobi-optimizer/

Benchmark Domain	#Ins	*#Feas*									*#Win*									*P(T)*								
		HiGHS			FiberSCIP			ParaILP			HiGHS			FiberSCIP			ParaILP			HiGHS			FiberSCIP			ParaILP		
		10s	60s	300s	10s	60s	300s	10s	60s	300s	10s	60s	300s	10s	60s	300s	10s	60s	300s	10s	60s	300s	10s	60s	300s	10s	60s	300s
Singleton	2	**2**	**2**	**2**	**2**	**2**	**2**	**2**	**2**	**2**	**1**	0	0	0	0	**1**	**1**	**2**	**1**	0.788	0.473	0.324	0.485	0.371	0.188	**0.109**	**0.088**	**0.081**
Aggregations	2	0	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	0	0	**1**	**1**	**1**	0	0	0	0	1.000	0.615	0.523	**0.672**	**0.531**	**0.508**	0.761	0.633	0.555
Bin Packing	2	0	**2**	**2**	0	1	1	**1**	**2**	**2**	0	**2**	0	0	0	**1**	**1**	0	**1**	1.000	**0.905**	**0.715**	1.000	1.000	0.993	**0.979**	0.923	0.773
Equation Knapsack	3	0	0	0	0	0	0	0	0	**1**	0	0	0	0	0	0	0	0	**1**	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
Knapsack	4	3	3	3	3	3	3	**4**	**4**	**4**	1	1	**2**	1	1	0	**2**	**2**	**2**	0.529	0.388	0.304	0.412	0.316	0.282	**0.111**	**0.071**	**0.048**
Set Packing	5	2	**4**	**4**	3	**4**	**4**	**4**	**4**	**4**	1	1	1	0	0	1	**3**	**3**	**2**	1.000	0.802	0.609	0.759	0.549	0.323	**0.369**	**0.260**	**0.219**
Cardinality	6	1	1	2	1	1	2	**2**	**2**	**3**	0	0	**1**	0	0	**1**	**2**	**2**	**1**	0.958	0.820	0.746	0.866	0.830	0.753	**0.628**	**0.571**	**0.522**
Hybrid	7	3	3	4	2	4	4	**4**	**5**	**5**	0	0	1	0	0	0	**4**	**5**	**5**	1.000	1.000	1.000	1.000	1.000	0.699	**0.689**	**0.548**	**0.524**
Mixed Binary	8	0	0	0	2	2	2	**4**	**5**	**6**	0	0	0	1	2	2	**4**	**4**	**5**	1.000	1.000	1.000	1.000	0.980	0.976	**0.890**	**0.861**	**0.762**
Set Partitioning	9	1	2	4	3	4	6	**4**	**6**	**7**	0	0	0	0	2	3	**4**	**4**	**4**	1.000	0.999	0.927	0.930	0.884	0.805	**0.887**	**0.827**	**0.765**
Set Covering	11	5	6	8	5	8	7	**9**	**10**	**10**	2	0	2	0	**5**	**6**	**7**	**5**	3	0.861	0.780	0.656	0.780	**0.648**	**0.523**	**0.733**	0.650	0.610
Precedence	13	1	3	4	10	11	**12**	**12**	**12**	**12**	0	0	0	1	1	2	**11**	**11**	**10**	0.996	0.974	0.860	0.889	0.813	0.653	**0.576**	**0.454**	**0.345**
General Linear	15	3	4	6	7	7	9	**10**	**10**	**10**	2	2	4	2	1	3	**6**	**7**	**7**	0.780	0.700	0.591	0.712	0.575	0.425	**0.355**	**0.317**	**0.301**
Variable Bound	16	7	9	9	11	13	13	**14**	**15**	**15**	0	0	1	0	0	2	**14**	**15**	**14**	0.895	0.784	0.653	0.851	0.688	0.472	**0.381**	**0.278**	**0.228**
Invariant Knapsack	18	4	7	11	10	11	**13**	**12**	**13**	**13**	2	1	3	2	3	2	**12**	**12**	**11**	0.924	0.847	0.736	0.778	0.706	0.664	**0.549**	**0.475**	**0.421**
Total	121	32	47	60	60	72	79	**83**	**91**	**95**	9	7	16	8	16	24	**71**	**72**	**67**	0.911	0.831	0.730	0.818	0.724	0.610	**0.582**	**0.508**	**0.452**

Table 1: Performance evaluation between SOTA academic solvers HiGHS, FiberSCIP and ParaILP.

Benchmark Domain	#Ins	*#Feas*									*#Win*									*P(T)*								
		$\text{Gurobi}_{\text{comp}}$			$\text{Gurobi}_{\text{heur}}$			ParaILP			$\text{Gurobi}_{\text{comp}}$			$\text{Gurobi}_{\text{heur}}$			ParaILP			$\text{Gurobi}_{\text{comp}}$			$\text{Gurobi}_{\text{heur}}$			ParaILP		
		10s	60s	300s	10s	60s	300s	10s	60s	300s	10s	60s	300s	10s	60s	300s	10s	60s	300s	10s	60s	300s	10s	60s	300s	10s	60s	300s
Singleton	2	**2**	**2**	**2**	**2**	**2**	**2**	**2**	**2**	**2**	0	**2**	**1**	**1**	0	**1**	**1**	1	0	0.237	0.113	**0.042**	0.240	0.115	0.045	**0.109**	**0.088**	0.081
Aggregations	2	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	0	**1**	**1**	0	0	0	**0.514**	**0.502**	**0.500**	0.519	0.503	0.501	0.761	0.633	0.555
Bin Packing	2	**2**	**2**	**2**	**2**	**2**	**2**	1	**2**	**2**	**2**	**2**	0	**2**	**2**	**1**	0	0	**1**	**0.963**	**0.748**	**0.303**	0.963	0.757	0.305	0.979	0.923	0.773
Equation Knapsack	3	0	0	**1**	0	0	0	0	0	**1**	0	0	**1**	0	0	0	0	0	**1**	1.000	1.000	**0.952**	1.000	1.000	1.000	1.000	1.000	1.000
Knapsack	4	3	3	**4**	3	3	**4**	**4**	**4**	**4**	1	**2**	2	1	1	**4**	**2**	**2**	1	0.317	0.271	0.100	0.315	0.274	0.093	**0.111**	**0.071**	**0.048**
Set Packing	5	**4**	**4**	**4**	**4**	**4**	**4**	**4**	**4**	**4**	2	1	**2**	**3**	**2**	**2**	1	**2**	**2**	0.408	0.271	0.231	0.407	0.271	0.230	**0.369**	**0.260**	**0.219**
Cardinality	6	**2**	**3**	**3**	**2**	**3**	**3**	**2**	2	**3**	**1**	**2**	1	0	1	**3**	**1**	1	0	0.741	**0.543**	0.364	0.741	0.543	**0.359**	**0.628**	0.571	0.522
Hybrid	7	**4**	**5**	**5**	**4**	**5**	**5**	**4**	**5**	**5**	1	1	2	1	1	**3**	**3**	**5**	2	0.777	0.660	0.548	0.784	0.664	0.546	**0.689**	**0.548**	**0.524**
Mixed Binary	8	2	3	3	2	3	3	**4**	**5**	**6**	1	0	1	1	1	1	**4**	**4**	**5**	0.986	0.975	0.960	0.986	0.970	0.958	**0.890**	**0.861**	**0.762**
Set Partitioning	9	**7**	**7**	**7**	**7**	**7**	**7**	4	6	**7**	4	**4**	3	**5**	**4**	**4**	2	2	2	**0.815**	0.748	0.625	0.815	**0.738**	**0.617**	0.887	0.827	0.765
Set Covering	11	**9**	9	9	**9**	9	9	**9**	**10**	**10**	**5**	5	4	2	**6**	**7**	4	2	2	0.616	0.384	0.291	**0.613**	**0.371**	**0.264**	0.733	0.650	0.610
Precedence	13	**12**	**12**	**12**	**12**	**12**	**12**	**12**	**12**	**12**	4	3	4	1	4	**9**	**8**	**6**	2	0.684	0.529	0.350	0.688	0.534	0.361	**0.576**	**0.454**	**0.345**
General Linear	15	9	**11**	**11**	8	**11**	**11**	**10**	10	10	**6**	5	6	3	**8**	**8**	4	3	3	0.421	**0.286**	**0.251**	0.460	0.295	0.254	**0.355**	0.317	0.301
Variable Bound	16	**14**	14	14	13	14	14	**14**	**15**	**15**	2	4	7	2	5	5	**12**	**10**	**8**	0.612	0.430	0.297	0.595	0.399	0.291	**0.381**	**0.278**	**0.228**
Invariant Knapsack	18	**12**	12	**15**	**12**	12	**15**	**12**	**13**	13	6	6	7	4	6	7	**9**	**10**	**9**	0.699	0.602	0.367	0.700	0.604	**0.367**	**0.549**	**0.475**	0.421
Total	121	**83**	88	93	81	88	92	**83**	**91**	**95**	36	38	42	26	42	**56**	**51**	**48**	38	0.653	0.526	0.398	0.656	0.522	**0.396**	**0.582**	**0.508**	0.452

Table 2: Performance evaluation between SOTA commercial solver Gurobi (both the exact and heuristic version) and ParaILP.

FiberSCIP [Shinano *et al.*, 2018]: a parallel branch-and-bound ILP solver (version 1.0.0, using SCIP 8.0.3 and Soplex 6.0.3 as its internal ILP and LP solver, respectively.).

Gurobi [Gurobi Optimization, 2022]: the most powerful commercial ILP solvers (version 10.0.2). We use both its complete and heuristic versions, denoted by $\text{Gurobi}_{\text{comp}}$ and $\text{Gurobi}_{\text{heur}}$, respectively.

The binaries of all competitors are downloaded from their websites, and default configurations are used. In MIPLIB dataset, each instance may contain various types of constraints[11], including knapsack, set covering, and others. We classified all instances based on the type of the dominant constraint class (i.e., the constraint class with the largest number of constraints), resulting in 15 benchmark domains. Note that an instance containing multiple dominant constraint classes is labeled as hybrid.

Comparisons with Academic Solvers. We first compare ParaILP with the state-of-the-art academic solvers, i.e., HiGHS and FiberSCIP, and the results are shown in Table 1. For all time limits, ParaILP performs best for 14 domains for *#Feas*, 11 domains for *#Win* and *P(T)*. For the total instances, ParaILP performs best in all 3 metrics for each time limit. The results indicate that the solving power of ParaILP is significantly better than that of HiGHS and FiberSCIP, at finding a high-quality feasible solution quickly.

Comparison with Commercial Solver. Further, we compare ParaILP with Gurobi, the state-of-the-art commercial ILP solver, and the result is shown in Table 2. In terms of *#Feas*, ParaILP performs best with 12 domains for the 10s time limit, 11 domains for 60s, and 13 domains for 300s. In terms of *#Win*, ParaILP performs best with 8 domains for 10s, 7 domains for 60s, and 6 domains for 300s. In terms of *P(T)*, ParaILP performs best with 10 domains for 10s, 8 domains for 60s, and 6 domains for 300s. For the total instances, ParaILP performs best in *#Feas* for all time limits, and in *#Win* and *P(T)* for both 10s and 60s time limits. In general, ParaILP outperforms Gurobi in some settings for 3 metrics, indicating that ParaILP is competitive with the state-of-the-art commercial solver.

6.3 Comparisons with the Basic Parallelization of Local-ILP

We compare ParaILP with the state-of-the-art sequential local search solver Local-ILP, which is also a subroutine in our

[11]https://miplib.zib.de/statistics.html

Time Limit	Local-ILP$_{32_seeds}$			ParaILP		
	#Feas	*#Win*	*P(T)*	*#Feas*	*#Win*	*P(T)*
10s	80	19	0.648	**83**	**77**	**0.582**
60s	89	19	0.570	**91**	**80**	**0.508**
300s	90	16	0.510	**95**	**88**	**0.452**

Table 3: Comparison of Local-ILP$_{32_seeds}$ and ParaILP on the benchmark for 10s, 60s, and 300s time limits.

framework. Since it is a sequential algorithm, we construct a basic parallel version of Local-ILP by adopting the diversity strategy on random seeds, executing it in 32 threads using 32 different random seeds, and selecting the best solution as the final output, which is denoted as Local-ILP$_{32_seeds}$.

The comparative results are depicted in Table 3, which substantiates that ParaILP outperforms Local-ILP when employing the same number of threads. Particularly, the *#Win* instances of Local-ILP are 3.1-4.5 times greater than those of Local-ILP, indicating a significant improvement and confirming the effectiveness of our entire parallel framework.

6.4 Ablation Study

We perform ablation studies to validate the efficacy of our ideas. We first compare ParaILP with two variants to evaluate the proposed polarity initialization. The first one is ParaILP$_{\text{init_1}}$, a variant of ParaILP using random initialization, which randomly assigns each variable an integer within its global bound. The second one is ParaILP$_{\text{init_2}}$, a variant of ParaILP that replaces the polarity initialization with the initialization method used in Local-ILP, in which variables are assigned to their bounds or 0 according to the signs of their bounds. As shown in Table 4, our proposed method significantly outperforms the other two initialization methods, confirming the effectiveness of polarity initialization.

We further compare ParaILP with two other variants to evaluate the cooperative evolution mechanism. The first one is ParaILP$_{\text{pop}}$, a variant of ParaILP that sets the population size ps to 0, which actually removes the cooperative evolution mechanism from the framework and makes each local search worker thread solve separately. The second one is ParaILP$_{\text{evo}}$, a variant of ParaILP that removes the solution evolution process from the cooperative evolution mechanism and makes the population only collect the solutions from local search worker threads. As shown in Table 5, ParaILP outperforms other variations, indicating that the cooperative evolution mechanism is effective and necessary.

Time Limit	ParaILP$_{\text{init_1}}$			ParaILP$_{\text{init_2}}$			ParaILP		
	#Feas	*#Win*	*P(T)*	*#Feas*	*#Win*	*P(T)*	*#Feas*	*#Win*	*P(T)*
10s	65	20	0.651	79	20	0.610	**83**	**62**	**0.582**
60s	82	25	0.571	90	28	0.521	**91**	**62**	**0.508**
300s	89	30	0.494	93	33	0.454	**95**	**67**	**0.452**

Table 4: Comparison with ParaILP$_{\text{init_1}}$ and ParaILP$_{\text{init_2}}$.

6.5 Scalability of ParaILP

Robust scalability is the ability to increase the algorithm's performance proportionally when parallel resources are incremented while the problem size remains fixed. As the primal integral *P(T)* is a well-established and independent measure to evaluate the performance of ILP, we apply the ratio of *P(T)* between different processor configurations and 1 processor configuration to evaluate the scalability of the parallel solver. Let $P(T)_N$ denote the primal integral *P(T)* obtained by using N processors, and $P(T)_{base}$ denote the baseline configuration by using 1 processor. We test each solver with the number processors in $\{2, 4, 8, 16, 32\}$, and use the ratio $\frac{P(T)_N}{P(T)_{base}}$ to observe the improvement between different processors and 1 processor. We compare the ratio $\frac{P(T)_N}{P(T)_{base}}$ obtained by each parallel solver on the benchmark for the time limit of 300s. The results are shown in Figure 2, indicating that the scalability of ParaILP is better than the academic solvers HiGHS and FiberSCIP, but not as good as the commercial solver Gurobi.

Time Limit	ParaILP$_{\text{pop}}$			ParaILP$_{\text{evo}}$			ParaILP		
	#Feas	*#Win*	*P(T)*	*#Feas*	*#Win*	*P(T)*	*#Feas*	*#Win*	*P(T)*
10s	**83**	20	0.616	**83**	19	0.594	**83**	**77**	**0.582**
60s	**91**	24	0.543	**91**	33	0.517	**91**	**78**	**0.508**
300s	**95**	22	0.501	**95**	29	0.471	**95**	**76**	**0.452**

Table 5: Comparison with ParaILP$_{\text{pop}}$ and ParaILP$_{\text{evo}}$.

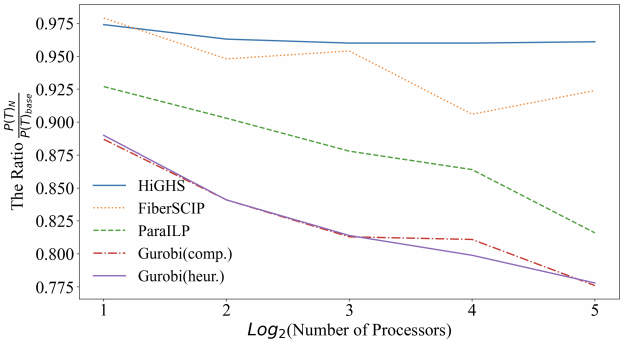

Figure 2: Comparison of HiGHS, FiberSCIP, Gurobi and ParaILP on the ratio $\frac{P(T)_N}{P(T)_{base}}$ when adopting different processors.

7 Conclusions

This paper proposed a parallel local search framework called ParaILP for solving general integer linear programming (ILP) problems. We proposed the polarity initialization to construct different initial solutions for local search threads. Moreover, we proposed a cooperative evolution mechanism that manages and generates high-quality solutions found throughout the search process. To the best of our knowledge, this is the first parallel local search framework for the general ILP problem. We performed extensive experiments to measure the ability to find a high-quality feasible solution quickly. Experiments show that ParaILP is significantly better than the state-of-the-art academic solvers FiberSCIP and HiGHS, and is competitive with the state-of-the-art commercial solver Gurobi. Additionally, ParaILP significantly improves the state-of-the-art sequential local search algorithm Local-ILP.

Acknowledgments

This work is s supported by National Key R&D Program of China (2023YFA1009500).

References

[Achterberg *et al.*, 2006] Tobias Achterberg, Thorsten Koch, and Alexander Martin. MIPLIB 2003. *Oper. Res. Lett.*, 34(4):361–372, 2006.

[Berthold, 2013] Timo Berthold. Measuring the impact of primal heuristics. *Oper. Res. Lett.*, 41(6):611–614, 2013.

[Bixby *et al.*, 1995] Robert E Bixby, William Cook, A Cox, and Eva K Lee. Parallel mixed integer programming. *Rice University Center for Research on Parallel Computation Research Monograph CRPC-TR95554*, 1995.

[Dantzig *et al.*, 1954] George B. Dantzig, D. Ray Fulkerson, and Selmer M. Johnson. Solution of a large-scale traveling-salesman problem. *Oper. Res.*, 2(4):393–410, 1954.

[Dorne and Hao, 1998] Raphaël Dorne and Jin-Kao Hao. A new genetic local search algorithm for graph coloring. In A. E. Eiben, Thomas Bäck, Marc Schoenauer, and Hans-Paul Schwefel, editors, *Parallel Problem Solving from Nature - PPSN V, 5th International Conference, Amsterdam, The Netherlands, September 27-30, 1998, Proceedings*, volume 1498 of *Lecture Notes in Computer Science*, pages 745–754. Springer, 1998.

[Genova and Guliashki, 2011] Krasimira Genova and Vassil Guliashki. Linear integer programming methods and approaches–a survey. *Journal of Cybernetics and Information Technologies*, 11(1), 2011.

[Gleixner *et al.*, 2021] Ambros M. Gleixner, Gregor Hendel, Gerald Gamrath, Tobias Achterberg, Michael Bastubbe, Timo Berthold, Philipp Christophel, Kati Jarck, Thorsten Koch, Jeff T. Linderoth, Marco E. Lübbecke, Hans D. Mittelmann, Derya B. Özyurt, Ted K. Ralphs, Domenico Salvagnin, and Yuji Shinano. MIPLIB 2017: data-driven compilation of the 6th mixed-integer programming library. *Math. Program. Comput.*, 13(3):443–490, 2021.

[Gurobi Optimization, 2022] LLC Gurobi Optimization. Gurobi optimizer ref. manual. https://www.gurobi.com, 2022. Accessed: 2023-10-28.

[Huangfu and Hall, 2018] Qi Huangfu and J. A. J. Hall. Parallelizing the dual revised simplex method. *Math. Program. Comput.*, 10(1):119–142, 2018.

[Jacobs and Brusco, 1995] Larry W Jacobs and Michael J Brusco. Note: A local-search heuristic for large set-covering problems. *Naval Research Logistics (NRL)*, 42(7):1129–1140, 1995.

[Kannan and Monma, 1978] Ravindran Kannan and Clyde L Monma. On the computational complexity of integer programming problems. In *Optimization and Operations Research: Proceedings of a Workshop Held at the University of Bonn, October 2–8, 1977*, pages 161–172. Springer, 1978.

[Koch *et al.*, 2011] Thorsten Koch, Tobias Achterberg, Erling Andersen, Oliver Bastert, Timo Berthold, Robert E. Bixby, Emilie Danna, Gerald Gamrath, Ambros M. Gleixner, Stefan Heinz, Andrea Lodi, Hans D. Mittelmann, Ted K. Ralphs, Domenico Salvagnin, Daniel E. Steffy, and Kati Wolter. MIPLIB 2010. *Math. Program. Comput.*, 3(2):103–163, 2011.

[Land and Doig, 2010] Ailsa H. Land and Alison G. Doig. An automatic method for solving discrete programming problems. In Michael Jünger, Thomas M. Liebling, Denis Naddef, George L. Nemhauser, William R. Pulleyblank, Gerhard Reinelt, Giovanni Rinaldi, and Laurence A. Wolsey, editors, *50 Years of Integer Programming 1958-2008 - From the Early Years to the State-of-the-Art*, pages 105–132. Springer, 2010.

[Lawler and Wood, 1966] Eugene L. Lawler and D. E. Wood. Branch-and-bound methods: A survey. *Oper. Res.*, 14(4):699–719, 1966.

[Lin *et al.*, 2023] Peng Lin, Shaowei Cai, Mengchuan Zou, and Jinkun Lin. New characterizations and efficient local search for general integer linear programming. *arXiv preprint arXiv:2305.00188*, 2023.

[López-Ibáñez *et al.*, 2016] Manuel López-Ibáñez, Jérémie Dubois-Lacoste, Leslie Pérez Cáceres, Mauro Birattari, and Thomas Stützle. The irace package: Iterated racing for automatic algorithm configuration. *Operations Research Perspectives*, 3:43–58, 2016.

[Merz and Freisleben, 1997] Peter Merz and Bernd Freisleben. Genetic local search for the tsp: New results. In *Proceedings of 1997 Ieee International Conference on Evolutionary Computation (Icec'97)*, pages 159–164. IEEE, 1997.

[Roshanaei *et al.*, 2013] V Roshanaei, Ahmed Azab, and H ElMaraghy. Mathematical modelling and a meta-heuristic for flexible job shop scheduling. *International Journal of Production Research*, 51(20):6247–6274, 2013.

[Shinano *et al.*, 2018] Yuji Shinano, Stefan Heinz, Stefan Vigerske, and Michael Winkler. Fiberscip - A shared memory parallelization of SCIP. *INFORMS J. Comput.*, 30(1):11–30, 2018.

[Stützle, 2006] Thomas Stützle. Iterated local search for the quadratic assignment problem. *Eur. J. Oper. Res.*, 174(3):1519–1539, 2006.

[Vaessens *et al.*, 1996] Rob J. M. Vaessens, Emile H. L. Aarts, and Jan Karel Lenstra. Job shop scheduling by local search. *INFORMS J. Comput.*, 8(3):302–317, 1996.

[Verachi and Prestwich, 2008] Stefania Verachi and Steven Prestwich. Constructive vs perturbative local search for general integer linear programming. In *Proceedings of the Fifth International Workshop on Local Search Techniques in Constraint Satisfaction (LSCS)*, 2008.

[Wolsey, 2020] Laurence A Wolsey. *Integer programming*. John Wiley & Sons, 2020.

[Xu *et al.*, 2009] Y. Xu, Ted K. Ralphs, Laszlo Ladányi, and Matthew J. Saltzman. Computational experience with a

software framework for parallel integer programming. *INFORMS J. Comput.*, 21(3):383–397, 2009.