OOESAlgorithm.jl: A Julia package for optimizing a linear function over the set of efficient solutions for bi-objective mixed integer linear programming

Alvaro Sierra-Altamiranda and Hadi Charkhgard*

Department of Industrial and Management Systems Engineering, University of South Florida, Tampa, FL, 33620 USA
E-mail: amsierra@mail.usf.edu [F. Author]; hcharkhgard@usf.edu [S. Author]

Received DD MMMM YYYY; received in revised form DD MMMM YYYY; accepted DD MMMM YYYY

Abstract
We present OOESAlgorithm.jl, a package for optimizing a linear function over the efficient set of bi-objective mixed integer linear programs. The proposed package extends our recent study (see Sierra-Altamiranda and Charkhgard (2018)) by adding two main features: (1) In addition to CPLEX, the package allows employing any single-objective solver supported by MathProgBase.jl, e.g., GLPK, CPLEX, and SCIP; (2) The package supports execution on multiple processors and is compatible with the JuMP modeling language. An extensive computational study shows the efficacy of the package and its features.

Keywords: bi-objective mixed integer linear programming; optimization over efficient set; parallelization; commercial and non-commercial solvers

1. Introduction
Many real-world optimization problems can be formulated as mixed integer linear programs. These problems often involve multiple conflicting objectives, i.e., it is impossible to find a feasible solution that simultaneously optimizes all objectives. Consequently, the focus in multi-objective optimization has been primarily on developing algorithms for generating some (if not all) efficient solutions, i.e., solutions in which it is impossible to improve the value of one objective without a deterioration in the value of at least one other objective. Although understanding the trade-offs between objectives can help decision makers select their preferred solutions, some authors (see for instance Jorge (2009)) argue that presenting too many efficient solutions may only confuse the decision makers(s). An approach that alleviates this issue is finding a preferred solution among the set of efficient solutions directly and is known as optimizing over the efficient set.

The problem of optimizing a linear function over the efficient set has a rich literature in multi-objective linear programming (see for instance Benson (1993); Yamamoto (2002)). Also, in recent years, this problem has received attention for multi-objective pure integer linear programs (see for instance Abbas and Chaabane (2006); Boland et al. (2017); Chaabane et al. (2012); Djamal and Marc (2010); Jorge (2009)). However,
no exact algorithm is known for optimizing a linear function over the set of efficient solutions of multi-objective mixed integer linear programs involving more than two objectives. In fact, in our recent study (see Sierra-Altamiranda and Charkhgard (2018)), we proposed the first algorithm (to the best of our knowledge) for cases with two objectives, i.e., Bi-Objective Mixed Integer Linear Programs (BOMILPs). In light of this observation, in this study, we extend our previous work by creating a user-friendly open-source Julia package which has the following additional desirable characteristics (compared to its original C++ implementation):

- The package is compatible with the popular JuMP modeling language (Dunning et al., 2017) and supports input in LP and MPS file formats.
- The package supports execution on multiple processors and allows users to choose different parallelization techniques by just tuning a parameter. It is worth mentioning that several studies have been conducted about parallelization for evolutionary algorithms in multi-objective optimization (see for instance Pal and Charkhgard (2018); Yu et al. (2017)). However, unfortunately, this topic has been almost untouched in the literature of exact algorithms. The recent study conducted by Özlen et al. (Pettersson and Özlen, 2017) is one of the few papers (if not the only one) in this scope.
- The package allows users to choose between different single-objective optimization solvers by just tuning a parameter. The default solvers include GLPK, CPLEX, Gurobi and SCIP, but it works for all other solvers supported by MathProgBase.jl as well.
- The package can be modified by users to return the entire nondominated frontier of a BOMILP.

To the best of our knowledge, there are currently only two documented and supported implementations of multi-objective optimization algorithms in Julia (see Gandibleux et al. (2017) and Pal and Charkhgard (2018b)). So, our package contributes to increasing the visibility of multi-objective optimization solvers in Julia. In the remaining of this paper, our package is referred to as 0OESAlgorithm.jl and its underlying algorithm as 0OESAlgorithm. The structure of the paper is organized as follows: In Section 2, the main concepts and definitions are explained. In Section 3, we provide a brief explanation of 0OESAlgorithm. In Section 4, the main characteristics of the package are detailed. In Section 5, a comprehensive computational study is conducted. Finally, in Section 6, some concluding remarks are provided.

2. Preliminaries

A Bi-Objective Mixed Integer Linear Program (BOMILP) can be stated as follows:

$$\min_{(x_I, x_C) \in \mathcal{X}} \{z_1(x_I, x_C), z_2(x_I, x_C)\}, \quad (1)$$

where \(\mathcal{X} := \{(x_I, x_C) \in \mathbb{Z}_{n_1}^2 \times \mathbb{R}_{n_2}^2 : A_1 x_I + A_2 x_C \leq b\}\) represents the feasible set in the decision space, \(\mathbb{Z}_{n_1}^2 := \{s \in \mathbb{Z}^{n_1} : s \geq 0\}\), \(\mathbb{R}_{n_2}^2 := \{s \in \mathbb{R}^{n_2} : s \geq 0\}\), \(A_1 \in \mathbb{R}^{m \times n_1}\), \(A_2 \in \mathbb{R}^{m \times n_2}\), and \(b \in \mathbb{R}^m\). It is assumed that \(\mathcal{X}\) is bounded and \(z_i(x_I, x_C) = c_i^T x_I + c_{i,C}^T x_C\) where \(c_{i,I} \in \mathbb{R}^{n_1}\) and \(c_{i,C} \in \mathbb{R}^{n_2}\) for \(i = 1, 2\) represents a linear objective function. The image \(\mathcal{Y}\) of \(\mathcal{X}\) under vector-valued function \(z := (z_1, z_2)^T\) represents the feasible set in the objective/criterion space, that is \(\mathcal{Y} := \{o \in \mathbb{R}^2 : o = z(x_I, x_C)\} \text{ for some } (x_I, x_C) \in \mathcal{X}\).

**Definition 1.** A feasible solution \((x_I, x_C) \in \mathcal{X}\) is called **efficient** if there is no other \((x'_I, x'_C) \in \mathcal{X}\) such that \(z_1(x'_I, x'_C) \leq z_1(x_I, x_C)\) and \(z_2(x'_I, x'_C) < z_2(x_I, x_C)\) or \(z_1(x'_I, x'_C) < z_1(x_I, x_C)\) and \(z_2(x'_I, x'_C) \leq z_2(x_I, x_C)\). If \((x_I, x_C)\) is efficient, then \(z(x_I, x_C)\) is called a **nondominated point**. The set of all efficient solutions is denoted by \(\mathcal{X}_E\). The set of all nondominated points \(z(x_I, x_C)\) for \((x_I, x_C) \in \mathcal{X}_E\) is denoted by \(\mathcal{Y}_N\) and referred to as the **nondominated frontier**.
Second objective value
First objective value

Fig. 1: An example of the nondominated frontier of a BOMILP

Note that if \( n_1 > 0 \) and \( n_2 > 0 \), the nondominated frontier of a BOMILP can be complicated since it may contain isolated points as well as closed, half-open, and open segments. An example of a nondominated frontier of a BOMILP is shown in Figure 1. Although we assumed that \( \mathcal{X} \) is bounded, we observe that the nondominated frontier of a BOMILP may still contain an infinite number of nondominated points (because of the existence of continuous segments) and so computing a (finite) exact representation of the nondominated frontier is quite challenging.

In light of the above, the problem of optimizing a linear function over the set of efficient solutions of a BOMILP can be stated as follows:

\[
\min_{(x_I, x_C) \in \mathcal{X}_E} f(x_I, x_C) \tag{2}
\]

where \( f(x_I, x_C) = c_{f,I}^T x_I + c_{f,C}^T x_C \) with \( c_{f,I} \in \mathbb{R}^{n_1} \) and \( c_{f,C} \in \mathbb{R}^{n_2} \) represents a linear function. To ensure that the problem cannot be solved straightforwardly, we assume that \( \mathcal{X} \neq \mathcal{X}_E \) and \( f(x) \) is not a strictly positive linear combination of \( z_1(x) \) and \( z_2(x) \). Note that if \( \mathcal{X} = \mathcal{X}_E \) or \( f(x) \) is a strictly positive linear combination of \( z_1(x) \) and \( z_2(x) \) then Problem (2) is equivalent to \( \min_{(x_I, x_C) \in \mathcal{X}} f(x_I, x_C) \). The following observation is helpful.

**Observation 1.** Let \( S \subseteq \mathcal{X} \) be any arbitrary subset of the feasible solutions in the decision space such that \( S_E \subseteq \mathcal{X}_E \) where \( S_E \) is the set of efficient solutions of \( S \). Now let \( f^l := \min_{(x_I, x_C) \in S} f(x_I, x_C) \) and \( f^u := \min_{(x_I, x_C) \in S_E} f(x_I, x_C) \). If \( S = \mathcal{X} \), then \( f^l \) is a global lower (or dual) bound and \( f^u \) is the optimal value of Problem (2). Otherwise, i.e., \( S \neq \mathcal{X} \), \( f^l \) and \( f^u \) are local lower bound and local upper bound for the optimal value of Problem (2), i.e., are valid bounds in \( S \) but not necessarily in \( \mathcal{X} \), respectively.

**3. The algorithm**

In our previous study (Sierra-Altamiranda and Charkhgard, 2018), we presented the first criterion space search algorithm, referred to as \texttt{OOES Algorithm}, for optimizing a linear function over the set of efficient solutions of BOMILPs.

\texttt{OOES Algorithm} maintains a priority queue of rectangles and right-angled triangles (defined by two nondominated points in the criterion space) that need to be explored. At the beginning, this priority queue is empty, so \texttt{OOES Algorithm} first computes the endpoints of the nondominated frontier. These two points are then used to define the first rectangle containing all the “not-yet-found” nondominated points, as shown in Figure 2a. Whenever the algorithm defines a rectangle, local lower and upper bounds are computed for the rectangle based on Observation 1. Note that computing a local upper bound is done based on the discovered
nondominated points in a rectangle, e.g., for the first rectangle, the only discovered nondominated points are the endpoints.

OOES\_Algorithm explores a rectangle by finding the (locally) extreme nondominated points within the rectangle. It can be shown that by finding these points, the rectangle can be split into a set of right-angled triangles containing all "not-yet-found" nondominated points (see Figure 2b). Consequently, any triangle can be considered as a child node of a rectangle, and so the local lower bound of a triangle will be set equal to the local lower bound of its parent rectangle. In other words, the algorithm does not use its valuable computational time to compute a better local lower bound for a triangle (based on Observation 1). Overall, OOES\_Algorithm explores a triangle by first checking whether its hypotenuse is part of the nondominated frontier. If that is the case, then a local upper bound is computed over the hypotenuse of the triangle based on Observation 1. Otherwise the triangle is split into at most two other rectangles, as shown in Figure 2c. The algorithm then adds those rectangles to the priority queue.

The priority queue maintains its elements in non-increasing order of their local lower bounds. Consequently, the local lower bound corresponding to the first element of the priority queue is always the global lower bound. Also, the algorithm keeps track of the global upper bound at any time by recording the best local upper bound found. OOES\_Algorithm reports an optimal solution if the absolute or relative gap between the global lower bound and the global upper bound is small enough at the beginning of an iteration or if the priority queue becomes empty. Note that if the priority queue becomes empty then the optimality gap is naturally zero. For a more detailed explanation of the algorithm, we refer the interested readers to Sierra-Altamiranda and Charkhgard (2018).

4. Main characteristics of the package

In this section, we detail the main additional characteristics of OOES\_Algorithm.jl compared to our previous C++ implementation of OOES\_Algorithm.

4.1. Parallelization techniques

OOES\_Algorithm.jl benefits from the recent advances in modern computers, in terms of the number of processors, by exploiting parallelization. The package explores four different parallelization techniques, one of them is based on the priority queue, and the other three are based on decomposing the criterion space.

The simplest and most natural parallelization technique explores elements of the priority queue in parallel
using different threads. Suppose that \( t \) is the number of available threads for parallelization and \( q \) is the number of elements in the priority queue. The first element of the priority queue is assigned to the first available thread to be explored, the second element to the second thread, and this procedure continues until \( t \) elements are assigned. We will later show in our numerical experiments that this technique maximizes the utilization of the available threads, i.e., improves the performance of the algorithm significantly.

The criterion space parallelization techniques are based on splitting the unexplored nondominated frontier between the endpoints by adding cuts. We consider three types of cuts to split the criterion space based on their directions including horizontal, vertical, and diagonal. An illustration of these cuts can be found in Figure 3 when \( t = 3 \) (note that the number of cuts is \( t - 1 \)).

![Fig. 3: Splitting directions on the criterion space using three threads](image)

For horizontal splitting, the height of the nondominated frontier is divided by the number of threads that are available for parallelization, i.e., the distance between consecutive cuts is \( \frac{z^T - z^B}{t} \). The following optimization problems are solved to find a nondominated point for each cut \( v \in \{1, 2, \ldots, t - 1\} \):

\[
(\tilde{x}_v^I, \tilde{x}_v^C) \in \arg \min_{(x_I, x_C) \in \mathcal{X}} \left\{ z_1(x_I, x_C) : z_2(x_I, x_C) \leq z_2^B + \frac{v(z_2^T - z_2^B)}{t} \right\},
\]

followed by,

\[
(x_v^I, x_v^C) \in \arg \min_{(x_I, x_C) \in \mathcal{X}} \left\{ z_1(x_I, x_C) + z_2(x_I, x_C) : z(x_I, x_C) \leq z(\tilde{x}_v^I, \tilde{x}_v^C) \right\}.
\]

Observe that, \( \tilde{z}^v := z(\tilde{x}_v^I, \tilde{x}_v^C) \) may not be a nondominated point, therefore, the second operation is performed to find a nondominated point \( z^v := z(x_v^I, x_v^C) \). Finally, the unexplored nondominated frontier is split into rectangles defined by every pair of consecutive nondominated points. An illustration of this technique using three threads can be found in Figure 4.

The vertical splitting technique divides the width of the nondominated frontier by the number of available threads, i.e., the distance between consecutive cuts is \( \frac{z^B - z^T}{t} \). The following optimization problems are solved to find a nondominated point for each cut \( v \in \{1, 2, \ldots, t - 1\} \):

\[
(\tilde{x}_v^I, \tilde{x}_v^C) \in \arg \min_{(x_I, x_C) \in \mathcal{X}} \left\{ z_2(x_I, x_C) : z_1(x_I, x_C) \leq z_1^T + \frac{v(z_1^B - z_1^T)}{t} \right\},
\]
followed by,

\[(x_I^v, x_C^v) \in \arg \min_{(x_I, x_C) \in \mathcal{X}} \{ z_1(x_I, x_C) + z_2(x_I, x_C) : z(x_I, x_C) \leq z(\tilde{x}_I^v, \tilde{x}_C^v) \} \].

Finally, in the diagonal splitting, the algorithm attempts to divide the criterion space by adding cuts in which their angles are from the set \(\alpha \in \{\frac{\pi}{2t}, \ldots, \frac{(t-1)\pi}{2t}\}\) and originated from the point \((z_B^1, z_T^2)\). The following optimization problems are solved to find a nondominated point for each cut \(v \in \{1, 2, \ldots, t\}\):

\[(\tilde{x}_I^v, \tilde{x}_C^v) \in \arg \min_{(x_I, x_C) \in \mathcal{X}} \{ z_2(x_I, x_C) : z_2(x_I, x_C) - z_2^T \geq \tan(\frac{v\pi}{2t})(z_1(x_I, x_C) - z_1^B) \},

followed by,

\[(x_I^v, x_C^v) \in \arg \min_{(x_I, x_C) \in \mathcal{X}} \{ z_1(x_I, x_C) + z_2(x_I, x_C) : z(x_I, x_C) \leq z(\tilde{x}_I^v, \tilde{x}_C^v) \} \].

The first optimization problem finds a feasible solution that minimize the second objective function above the line \(z_2(x_I, x_C) - z_2^T = \tan(\frac{v\pi}{2t})(z_1(x_I, x_C) - z_1^B)\). An example of this technique using three threads can be observed in Figure 5.

It is worth mentioning that splitting the criterion space results in at most \(t\) independent unexplored rectangles. However, it is possible that different cuts return the same nondominated point, i.e., there may exist \(v, w \in \{1, 2, \ldots, t-1\}\) with \(v \neq w\) such that \(z_v = z_w\). If that is the case then OOESAlgorithm.jl is only able to employ less than \(t\) processors. Overall, employing any of the criterion space parallelization techniques will lead to the exploration of at most \(t\) independent rectangles. Thus, OOESAlgorithm.jl will return at most \(t\) independent solutions for Problem (2). So, at the time of termination, OOESAlgorithm.jl chooses the one that has minimum value for \(f(x)\).
4.2. Single-objective solvers

Our package benefits from the flexibility of working with Julia and the well-known optimization library MathProgBase.jl that allows users to choose between different single-objective optimization solvers. The default solvers are GLPK, CPLEX, Gurobi, Xpress, and SCIP, but any solver supported by MathProgBase.jl can be employed. Note that, GLPK and SCIP are non-commercial solvers, which means that our package is not limited by license availability.

5. A computational study

We conduct a comprehensive computational study to evaluate the performance of 0OESAlgorithm.jl. In the remaining, 0OESAlgorithm.jl is referred to by the name of the single-objective solver (GLPK 4.61, CPLEX 12.7, Gurobi 8.1, SCIP 6.0.0, and Xpress 8.5) employed in an experiment (by the package). This implies that GLPK, CPLEX, Gurobi, SCIP, and Xpress refer to 0OESAlgorithm.jl when GLPK 4.61, CPLEX 12.7, Gurobi 8.1, SCIP 6.0.0, and Xpress 8.5 are employed as the single-objective solver, respectively. All computational experiments are carried out on a Dell PowerEdge R630 with two Intel Xeon E5-2650 2.2 GHz 12-Core Processors (30MB), 128GB RAM, and the RedHat Enterprise Linux 6.8 operating system. The user manual of our open-source Julia package can be found at http://eng.usf.edu/~amsierra/documents/Documentation_OOESAlg.pdf. We note that our package is available in METADATA.jl, so users can easily install and use it after reading the user manual.

To test the performance of the package, we use the 250 (random) instances employed in our previous study Sierra-Altamiranda and Charkhgard (2018) which can be found in https://github.com/alvsierra286/Optimization-over-efficient-set-for-BOMILP-instances. In Sierra-Altamiranda and Charkhgard (2018), in order to create the instances for the problem of optimization over the frontier, at first 25 instances of bi-objective mixed integer linear programming are generated. Afterwards, based on each of these 25 bi-objective instances, 10 different instances for the problem of optimization over the frontier are generated by simply generating 10 random functions to be used as \( f(x_I, x_C) \).

Overall, the 25 bi-objective instances are divided into 5 classes based on their size. Each class is denoted by \( C_m \) where \( m \in \{20, 40, 80, 160, 320\} \) is the number of constraints for each instance in that class. In the remaining of this section, for each bi-objective instance, its average over its 10 corresponding variants is reported/used in the figures/tables.
5.1. **OOESAlgorithm.jl vs. C++ implementation**

The goal of this section is to only provide some numerical evidences that a julia implementation can be as good as a C++ implementation in terms of the solution time. So, in this section, we show the overall performance of OOESAlgorithm.jl compared to our previous C++ implementation of OOES Algorithm (Sierra-Altamiranda and Charkhgard, 2018). Since the C++ implementation works with CPLEX, we use CPLEX (which is basically OOESAlgorithm.jl when CPLEX is employed) for comparison in this section. In Table 1, we present a comparison between CPLEX and the C++ implementation where ‘Time (sec.)’ is the solution time in seconds and ‘#MILPs’ is the number of (single-objective) mixed integer linear programs solved. The last two columns of the table show the percentage of decrease in the solution time and the number of (single-objective) MILPs solved by CPLEX compared to the C++ implementation. Numbers are averages over 10 instances and bold numbers show the instances for which CPLEX has a better performance.

Table 1: Performance of the new algorithm in comparison to C++ implementation

<table>
<thead>
<tr>
<th>Class</th>
<th>CPLEX</th>
<th>C++</th>
<th>% Decrease</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Time (sec.)</td>
<td>#MILPs</td>
<td>Time (sec.)</td>
</tr>
<tr>
<td>C20</td>
<td>0.17 85.40</td>
<td>0.21 85.50</td>
<td>26.12%</td>
</tr>
<tr>
<td></td>
<td>0.60 167.00</td>
<td>0.66 171.60</td>
<td>10.16%</td>
</tr>
<tr>
<td></td>
<td>0.17 87.80</td>
<td>0.20 87.80</td>
<td></td>
</tr>
<tr>
<td>Avg. C20</td>
<td>0.45 150.14</td>
<td>0.50 151.46</td>
<td></td>
</tr>
<tr>
<td>C40</td>
<td>6.06 724.70</td>
<td>6.64 726.00</td>
<td>9.54%</td>
</tr>
<tr>
<td></td>
<td>2.05 311.70</td>
<td>2.33 312.60</td>
<td>13.69%</td>
</tr>
<tr>
<td></td>
<td>2.64 382.90</td>
<td>2.92 383.50</td>
<td>10.63%</td>
</tr>
<tr>
<td>Avg. C40</td>
<td>2.92 418.82</td>
<td>3.24 419.46</td>
<td></td>
</tr>
<tr>
<td>C80</td>
<td>46.52 1,757.60</td>
<td>47.01 1,757.30</td>
<td>1.06%</td>
</tr>
<tr>
<td></td>
<td>36.82 1,538.60</td>
<td>38.19 1,537.40</td>
<td>3.72%</td>
</tr>
<tr>
<td></td>
<td>30.88 1,471.80</td>
<td>33.39 1,472.00</td>
<td>8.11%</td>
</tr>
<tr>
<td>Avg. C80</td>
<td>36.70 1,529.56</td>
<td>38.40 1,529.58</td>
<td></td>
</tr>
<tr>
<td>C160</td>
<td>281.70 2,457.30</td>
<td>282.50 2,459.00</td>
<td>0.28%</td>
</tr>
<tr>
<td></td>
<td>265.44 2,213.30</td>
<td>266.93 2,213.00</td>
<td>0.56%</td>
</tr>
<tr>
<td></td>
<td>298.77 2,500.30</td>
<td>299.54 2,498.10</td>
<td>0.26%</td>
</tr>
<tr>
<td>Avg. C160</td>
<td>354.34 2,721.36</td>
<td>350.37 2,721.36</td>
<td></td>
</tr>
<tr>
<td>C320</td>
<td>4,083.94 4,022.30</td>
<td>4,009.40 4,016.70</td>
<td>-1.83%</td>
</tr>
<tr>
<td></td>
<td>4,440.33 4,329.60</td>
<td>4,308.11 4,332.00</td>
<td>-2.98%</td>
</tr>
<tr>
<td></td>
<td>3,770.92 4,096.30</td>
<td>3,748.88 4,110.50</td>
<td>-0.58%</td>
</tr>
<tr>
<td>Avg. C320</td>
<td>4,855.44 4,753.78</td>
<td>4,746.06 4,756.00</td>
<td></td>
</tr>
</tbody>
</table>

Note that CPLEX solves slightly less single-objective mixed integer linear programs. This is due to some
very minor enhancements in the implementation of CPLEX (that remove some redundant calculations). Overall, we observe that CPLEX outperforms C++ implementation for classes C20 and C40 and C80. However, the solution time of these three classes are small to make any meaningful conclusion. We observe that C++ is slightly faster (no more than 3%) than CPLEX for larger classes, i.e., C160 and C320, while solving almost the same number of single-objective optimization problems. One possible explanation for this observation could be the fact that the C++ implementation uses the ILOG Concert Technology but CPLEX uses MathProgBase.jl. To solve an instance of optimization over frontier, multiple single-objective optimization problems should be generated and solved (by our algorithm). So, ILOG Concert Technology can be possibly faster in doing so because it has been in the market for a longer period of time and has been improved over time.

5.2. Comparison between different solvers

In this section, we compare the performance of O0EAlgorithm.jl under different single-objective optimization solvers. Figure 6 shows the solution time ratios of different solvers on different classes of instances. The solution time ratio is the ratio of the solution time to the maximum solution time among all settings in a given figure.

In Figure 6a, the time ratios for instances in classes C20, C40 and C80 are reported. Observe that GLPK outperforms other solvers even commercial solvers such as CPLEX, GUROBI, and XPRESS. However, as mentioned before, those are small instances in which each single-objective optimization problem takes less than 0.01 to be solved on average. Figures 6b and 6c show the time ratios for instances in classes C160 and C320, respectively. Note that GLPK is not included in these figures since the solver went out of memory when solving such large instances. Observe that SCIP performs better than XPRESS. However, overall, GUROBI seems to be the best solver for solving large instances. From figure 6c, we observe that GUROBI is around 60% faster than CPLEX for instances of C320. In other words, each single-objective optimization problem is solved around 60% faster on average by Gurobi optimizer. One reason for this observation could be the fact that we have used CPLEX 12.7 which is two years older than Gurobi 8.1.

Fig. 6: The performance of using different solvers
5.3. Parallelization

In this section, we compare the performance of GÜROBI when multiple threads are available under different parallelization techniques, i.e., *PriorityQueue*, *Horizontal*, *Vertical*, and *Diagonal*. To show the value of parallelization, in this section, only the largest instances, i.e., those in C320, are used. Also, only GÜROBI is employed in this section because it was shown to perform the best for instances of C320 in the previous section.

Figure 7 shows a comparison between solution time ratios when \( t \in \{1, 2, 3, 4, 5, 6\} \) threads are employed. Specifically, in Figure 7a, the box plots of averages of the solution time ratios over all parallelization techniques are reported. Observe that the median of the average time ratios decreases almost linearly by employing more number of threads. Also, by comparing the medians of the box plot corresponding to 1 and 6 threads, it is evident that the improvement percentage is around 40%.

\[
\begin{array}{c}
\text{Time Ratio} \\
\hline
\text{Number of available threads} \\
\end{array}
\]

![Box plot for all parallelization techniques](image1)

(a) Average of all parallelization techniques

![Box plot for only PriorityQueue](image2)

(b) For only *PriorityQueue*

Fig. 7: Performance of GÜROBI when using multiple threads

In Section 4, we mentioned that *PriorityQueue* maximizes the utilization of the available threads. This can be observed from Figure 7b in which it illustrates the box plot of the solution time ratios for different threads when only *PriorityQueue* is used. Observe that the time ratio decreases significantly when more threads are employed. In fact, by comparing the medians of the box plot corresponding to 1 and 6 threads, it is evident that the improvement percentage is around 60%. To highlight the effect of *PriorityQueue* even further, in Figure 8, we compare the solution time ratios under different parallelization techniques when \( t \in \{2, 3, 4, 5, 6\} \) threads are available. Observe that *PriorityQueue* performs the best and the percentage decrease of the median is between 20% and 50%. This is mainly because in other parallelization techniques, the solution time depends directly on the most difficult rectangle generated after splitting the nondominated frontier. So, the usage of threads can be more unbalanced compared to *PriorityQueue*.

To show that our numerical results are not limited to just GÜROBI, we conducted a set of experiments with SCIP. Figure 9 shows the time ratios (for all instances) for SCIP when multiple threads are available and *PriorityQueue* is employed. Observe that the time ratio decreases significantly when more threads are available. In fact, by comparing the medians of the box plot corresponding to 1 and 6 threads, it is evident that the improvement percentage is around 57%.

6. Conclusion

We developed *00EAlgorithm.jl*, an open-source package for optimizing a linear function over the set of efficient solutions for BOMILPs in julia. *00EAlgorithm.jl* supports execution on multiple processors and exploits different parallelization techniques. It was numerically shown that parallelization helps to im-
prove the solution time significantly. Another desirable characteristic of the package is that it allows users to
employ different commercial and non-commercial solvers for solving single-objective optimization prob-
lems arising during the course of the algorithm. The computational study showed that even non-commercial
solvers can perform quite well. Finally, it was numerically shown that a julia package can be competitive
with a C++ package.
Fig. 9: The Performance of SCIP when employing PriorityQueue

References


