Systems & Control Transactions

Original Research Article
Peer Reviewed Conference Proceeding
ESCAPE 36 - European Symposium on Computer Aided Process Engineering
Sheffield, UK. 21-24 June 2026

PSE PRESS

libDIPS: An Open-Source Platform for Global Optimization of Hierarchical Optimization Problems

Adrian W. Lipow[b], Daniel Jungen[b], Aron Zingler[b], Hatim Djelassi[b] and Alexander Mitsos[abc]*

[a] JARA-CSD, 52062 Aachen, Germany
[b] Process Systems Engineering (AVT.SVT), RWTH Aachen University, 52074 Aachen, Germany
[c] Institute of Climate and Energy Systems: Energy Systems Engineering (ICE-1), Forschungszentrum Jülich GmbH, 52425 Jülich, Germany
* Corresponding Author: amitsos@alum.mit.edu.

ABSTRACT

Hierarchical optimization problems such as (generalized) semi-infinite optimization problems and bilevel problems appear in various disciplines of process systems engineering, such as flexibility analysis or parameter estimation. Adaptive discretization-based algorithms are a family of methods to solve these problems. In these methods, the original problem is decomposed into subproblems, which are solved with a standard optimization solver and then refined iteratively. Several related algorithms have been published. Until recently, computational studies have typically been performed using publication-specific implementations and benchmark problems. We recently published a software package – libDIPS – comprising existing adaptive discretization-based algorithms and a library of test problems for comparison. Several of the algorithms implemented in libDIPS exhibit strong parallelization potential in their algorithmic steps: In the algorithms of Mitsos [Optimization 60:1291-1308 (2011)] and Mitsos and Tsoukalas [J Glob Optim 61:1-17 (2015)], the upper- and lower-bounding procedures are independent. In the algorithm of Tsoukalas and Rustem [Optim Lett 5:705-716 (2011)], several objective function values can be probed independently for feasibility. However, algorithm statements in the literature typically only consider serial execution of the algorithm steps. While parallelization of the used subsolver is already possible in libDIPS, leveraging parallelization in the algorithmic steps has not been investigated until now. In this contribution, we present an overview of libDIPS and introduce our parallelized implementations of several existing algorithms. We investigate the effect of parallelization at the algorithm-level compared to parallelization within the subsolver. Furthermore, we propose two improvements to speed up the solution process of the algorithms compared to their original publications. Results are presented for the existing library of test problems collected from literature.

Keywords: Optimization, Parallelization, Numerical Methods, Semi-Infinite Programming

INTRODUCTION

Hierarchical optimization problems occur in various applications in process engineering, such as flexibility analysis [12] or thermodynamics [11]. Broadly speaking, hierarchical optimization problems include (generalized) semi-infinite optimization problems ((G)SIPs), existence-constrained semi-infinite optimization problems (ESIPs), bilevel problems and minmax problems. In hierarchical optimization problems, an upper-level problem is constrained by the solution of an embedded lower-level problem. Solving these optimization problems to global optimality is challenging especially when their lower-level problems are nonconvex, since even establishing feasibility of a point requires global optimization of the lower-level problem.

Several approaches for global optimization of hierarchical problems have been developed over the past decades, among them adaptive discretization-based methods, which decompose the original problem into several subproblems and iteratively refine them. A major advantage of adaptive discretization methods is that standard optimization solvers can be used to solve the subproblems. However, until recently, computational

studies to evaluate different methods' performance were typically done using publication-specific prototype implementations and benchmark problems, thereby limiting the possibility for fair comparisons between different methods. In previous work, we published libDIPS [7], a software of unified implementations of existing adaptive discretization-based algorithms [1-6, 8-10] and a large library of test problems collected from literature, enabling benchmarking and comparisons of the different algorithms.

In this contribution, we present several recent additions to libDIPS aimed at improving the computational performance of several solvers. We focus on solvers for SIPs and GSIPs and exploit parallelization potentials at the algorithmic level (as opposed to the subproblem level, which is already possible with appropriate subproblem solvers). Furthermore, we present two heuristics that improve some of the solvers' performance compared to their original publications.

We consider SIPs (1) and GSIPs (2):

$$\min_{\boldsymbol{x}\in X} f(\boldsymbol{x}) \tag{1a}$$
$$\text{s.t. } g(\boldsymbol{x},\boldsymbol{y}) \leq 0 \;\forall \boldsymbol{y} \in Y \tag{1b}$$

$$\min_{\boldsymbol{x}\in X} f(\boldsymbol{x}) \tag{2a}$$
$$\text{s.t. } g(\boldsymbol{x},\boldsymbol{y}) \leq 0 \;\forall \boldsymbol{y} \in Y(\boldsymbol{x}) \tag{2b}$$

We call $\boldsymbol{x}$ the upper-level variables and $\boldsymbol{y}$ the lower-level variables. We assume continuity of f on X and g on $X \times Y$. Further, we assume that the sets X, Y, and $Y(\boldsymbol{x})$ are compact and defined by continuous functions. These assumptions are made by the algorithms considered in the following [8, 9, 13] and are also typically required by the subsolvers. We only consider a single semi-infinite constraint ((1b) and (2b), respectively) for ease of notation, although the algorithms and our implementations can handle multiple semi-infinite constraints.

A SOFTWARE FRAMEWORK FOR SOLVING HIERARCHICAL OPTIMIZATION PROBLEMS

We briefly describe libDIPS from a user's perspective, specifically focusing on aspects of modeling and solving hierarchical optimization problems. For a detailed description of all features and implementation details, we refer to [7] and the libDIPS repository (see Digital Supplementary Material).

To model a problem in libDIPS, users must specify the subproblems as text files using the domain-specific language provided by libALE [18], which allows writing optimization problems in an intuitive fashion. There are two options for modeling the subproblems: users can formulate them in an opinionated way or use the provided template files to automatically derive the subproblems from the overall problem description. While the latter approach is arguably simpler to follow, the former allows users to exploit problem characteristics to potentially construct subproblem formulations tailored for the used subsolver. We illustrate this using an example workflow below.

As mentioned above, one advantage of adaptive discretization algorithms is that their subproblems can be solved by state-of-the-art MINLP optimization solvers. libDIPS provides interfaces to several commercial and open-source solvers, among them CPLEX [15], Gurobi [16], and our in-house solver MAiNGO [17]. For a continuously updated list of supported solvers, we refer to the libDIPS repository.

Example workflow

At a high level, the workflow for solving problems with libDIPS consists of writing the subproblems into text files, parsing these and passing the parsed problem objects to one of the solvers implemented in libDIPS, which then solves the problem. Output is written to the console and subsolver logs are written to files, enabling the user to track the overall solution progress.

Modeling in libDIPS

Assume we want to solve the following SIP using the *B&F* algorithm [1] (to be described):

$$\min_{\boldsymbol{x}\in[-2,2]^2} x_1^2 + 2x_2^3 \tag{3a}$$
$$\text{s.t. } \min\{x_1 - y_1, x_2 - y_2\} \leq 0, \forall \boldsymbol{y} \in [-1,1]^2 \tag{3b}$$

The required input files are given in Listings 1-4. Note that we show two formulations of the lower-level problem: Listing 2 uses the provided template and Listing 3 formulates the problem in an opinionated way. Besides defining the subproblems `lbp` and `llp`, a definitions file is required to mark special symbols in the subproblems, such as those related to the discretization or algorithm-specific ones.

Solving problems using libDIPS

As the name suggests, libDIPS is a library of solvers, which users can utilize in their own software tools. Furthermore, we provide standalone demos for all solvers, which demonstrate how to call the solvers from a C++ workflow. The demo executables can also be used to solve any problem defined in the format specified above. The default subsolver for the subproblems is MAiNGO; users can easily change this to their preferred solver in the build configuration file.

IMPROVING THE PERFORMANCE OF (G)SIP ALGORITHMS

The original publications of all algorithms

```
definitions:
real[2] x in [-2,2];
set{index} lbp_k_disc := {};
real[0,2] lbp_y_disc := 0;
real f(real[2] x) := x[1]^2+2*x[2]^3;
real g(real[2] x, real[2] y) := min(x[1]-y[1], x[2]-y[2]);
objective:
f(x);
constraints:
forall k in lbp_k_disc:
    g(x, lbp_y_disc[k]) <= 0;
```

Listing 1: The file `lbp.txt` for the example problem (3). Function symbols `f` and `g` are used to enable templated derivation of the subproblems.

```
definitions:
real[2] y in [-1,1];
objective:
-g(x,y);
```

Listing 2: The file `llp.txt` for the lower-level problem of the example problem (3), using the provided template. For this problem, this results in a nonlinear optimization problem due to the min-operator. Note that subproblems are always written as minimization problems.

```
definitions:
real[2] y in [-1,1];
real t in [-3,3];
objective:
-t
constraints:
x[1]-y[1] >= t;
x[2]-y[2] >= t;
```

Listing 3: The file `llp.txt` for the lower-level problem of the example problem (3), using an opinionated formulation. For this problem, this results in a linear optimization problem.

```
programdefinitions("lbp"):
set_disc(1) = lbp_k_disc;
set_disc_parameter_src(1, lbp_y_disc) = y;
```

Listing 4: The file def.`txt` for the example problem (3), defining `lbp_k_disc` as the set of discretization indices and `lbp_y_disc` as the discretized set Y^{LBP}.

implemented in libDIPS only consider serial execution of the algorithmic steps. Multi-core architectures can still be exploited by parallelism in the subsolver called to solve the subproblems. However, several of the algorithms offer potential for parallelization on their own. Here, we consider the *RRHS* algorithm [8], the *GSIP-RRHS* algorithm [9] and the *Oracle* algorithm [13]. Furthermore, we propose changes to the algorithmic steps to improve the algorithms' performance.

Parallel bounding in *RRHS* and *GSIP-RRHS*

RRHS [8] and *GSIP-RRHS* [9] combine the lower-bounding procedure from the *B&F* algorithm [1] with an upper-bounding procedure based on restricting the

right-hand-side of the semi-infinite constraint. We briefly state the main ideas of the algorithm here and refer to the original publications for full details. The two subproblems are given for an SIP in the following:

$$\min_{\boldsymbol{x}\in X} f(\boldsymbol{x}) \tag{4a}$$
$$\text{s. t. } g(\boldsymbol{x}, \boldsymbol{y}^k) \leq 0 \; \forall \boldsymbol{y}^k \in Y^{\text{LBD}} \tag{4b}$$

$$\min_{\boldsymbol{x}\in X} f(\boldsymbol{x}) \tag{5a}$$
$$\text{s. t. } g(\boldsymbol{x}, \boldsymbol{y}^k) \leq -\varepsilon \; \forall \boldsymbol{y}^k \in Y^{\text{UBD}}, \tag{5b}$$

Where $\epsilon > 0$, and $Y^{\text{UBD}}, Y^{\text{LBD}}$ constitute sets of finite cardinality, i.e., discrete subsets of the infinite set Y. Problem (4) is the lower-bounding problem. Problem (5) is the upper-bounding problem, which is generally neither a relaxation nor a restriction and may not yield an upper bound in every iteration. Note that the two problems use separate discretizations, Y^{LBD} and Y^{UBD} (a shared discretization is possible, but this leads to larger subproblems and may offset any performance gains seen by reducing the number of iterations). Thus, while the original publications alternate between solving the upper- and lower-bounding subproblems, these can be solved in parallel, as subsequent iterations are not dependent on information from the respective other procedure.

Candidate points $\bar{\boldsymbol{x}}$ computed from problems (4) and (5) are checked for feasibility by solving the lower-level problem:

$$g^* = \max_{\boldsymbol{y}\in Y} g(\bar{\boldsymbol{x}}, \boldsymbol{y}) \tag{6}$$

If g^* is non-positive, the candidate point $\bar{\boldsymbol{x}}$ is SIP-feasible, otherwise, it is infeasible. In the latter case, the associated value $\boldsymbol{y}^*$ is used to refine the discretization of the corresponding upper-level problem (i.e., if $\bar{\boldsymbol{x}}$ was generated by (4), Y^{LBD} would be updated to $Y^{\text{LBD}} \cup \{\boldsymbol{y}^*\}$). Note that problems (4) and (6) are the subproblems used in the *B&F* algorithm, which only computes lower bounds.

Besides the obvious benefit of potentially being able to solve more subproblems in a fixed amount of time, decoupling the upper- and lower-bounding procedures has another advantage: It enables the algorithm to progress faster when one procedure has already found a "good" bound. The sequential implementation of *RRHS* and *GSIP-RRHS* alternates between solving problems (4) and (5). If one of the procedures quickly finds a "good" bound and stops progressing, subsequent iterations of that procedure are still performed but do not contribute to convergence of the algorithm. By decoupling the two procedures, the optimality gap may be closed faster because neither procedure needs to wait for an iteration of the other to complete before continuing. We test this claim using the library of test problems in libDIPS.

Early refining of the upper-bounding discretization in *RRHS* and *GSIP-RRHS*

For *RRHS*, we already presented a guard heuristic in [7], in which the upper-bounding procedure is skipped until the lower-bounding procedure produces ε^a-SIP-feasible points with $\varepsilon^a > 0$. Whenever the upper-bounding procedure is skipped, the upper-bounding discretization is populated with the discretization point obtained from the lower-bounding procedure. The aim of the guard is to quickly refine the upper-bounding discretization in early iterations, when the upper-bounding procedure is unlikely to find a feasible point. For *GSIP-RRHS*, this approach is analogously applicable. However, the guard heuristic only makes sense when the algorithms are executed in serial fashion. In the parallel implementations, such a guard cannot be directly applied, as the upper- and lower-bounding procedures are run in parallel, so skipping the upper-bounding procedure does not make sense.

In the parallel versions of *RRHS* and *GSIP-RRHS*, we propose to achieve this aim in a similar fashion. Whenever the lower-bounding procedure furnishes a new discretization point, we check if a valid upper bound has been found. If this is not the case, we add the discretization point to the upper-bounding discretization.

Parallel objective probing in *Oracle*

The *Oracle* algorithm [13] adapts the algorithm from [1] to obtain both lower bounds and feasible points. It does so by solving the oracle problem (7) to determine whether an objective function value f^{target} is attainable.

$$\min_{\boldsymbol{x}\in X} \max\{f(\boldsymbol{x}) - f^{\text{target}}, \max_{\boldsymbol{y}^k \in Y^{\text{ORA}}} g(\boldsymbol{x}, \boldsymbol{y}^k)\}, \tag{7}$$

Where Y^{ORA} is a set of finite cardinality If f^{target} is not attainable with a given discretization Y^{ORA}, it is a lower bound to the optimal objective value. On the other hand, if the target is attainable in the discretized subproblem and the lower-level problem proves the candidate point SIP-feasible, an upper bound is obtained. In the latter case, the lower-level problem may also declare the candidate point SIP-infeasible, in which case the discretization is refined, and the oracle problem is re-solved for the same target value. After successfully finding a new upper or lower bound, the target value is updated (in the original publication by bisection) and the oracle problem is solved again, starting with the discretization of the last iteration.

The *Oracle* algorithm offers great potential for parallelization similarly to runahead computing [14], where additional threads operate a few steps ahead of the main thread to speculate on the future state. In *Oracle*, this means that multiple objective value targets can be probed in parallel. For example, rather than just probing the midpoint between the current lower and upper bound, two additional threads can be used to probe the

upper and lower quarter of the interval as well, thereby effectively computing the next iteration of the algorithm in parallel. This idea can be generalized to arbitrarily many targets, assuming sufficient computing capabilities. Whenever a thread computes a new bound, any threads probing targets outside the remaining interval are aborted and assigned new valid targets.

Using multiple threads to solve multiple instances of problem (7) for different values of f^{target} raises the question of how to deal with the discretization Y^{ORA}. On one hand, Y^{ORA} should not grow unnecessarily large to avoid generating overly large subproblems. On the other hand, discretization points generated by one thread for a particular value of f^{target} may also be relevant for other target values. Thus, we present three strategies for handling the discretization. First, it is possible for each thread to have its own discretization, which is never synchronized with other threads (referred to as strategy "Off" in our numerical experiments later). This results in discretizations of minimal size in the sense that every thread only includes those points in its discretization which have been proven to be necessary in at least one iteration. However, this approach may lead to multiple threads computing the same discretization points, potentially increasing the total number of subproblems solved.

Second, it is possible for the threads to share a single common discretization (strategy "All"). Whenever the solution of a lower-level problem in a thread adds a discretization point, all other threads also use that point in subsequent iterations. While this leads to the subproblems becoming tighter and potentially requiring fewer iterations, it may also yield unnecessarily fine discretizations, increasing the difficulty of solving individual subproblems.

A third option is to combine the previous two approaches. Each thread uses its own discretization, but whenever a thread produces an update to the lower or upper bound, all other discretizations are updated to include the discretization points of the successful thread (strategy "Opt"). This approach attempts to combine the best aspects of the other two approaches: sharing discretization points that have proved useful in computing tighter bounds without producing unnecessarily fine discretizations. We test all three approaches using the library of test problems in libDIPS to confirm our claims.

A secant-based heuristic for updating the target value of *Oracle*

In the original publication of *Oracle* [13], new objective function target values are selected through bisection of the objective space. While this approach is simple to implement, it may lead to situations where a very good upper or lower bound is found quickly, but many iterations are required to prove (approximate) optimality or find a "good" feasible point.

To improve the convergence of the algorithm, we attempt to improve on the simple bisection of the objective space when selecting a new target objective value. In particular, we want to exploit iterates that lie near the boundary of the semi-infinite constraint. Once the algorithm has furnished an upper or lower bound with a sufficiently small absolute value of the semi-infinite constraint, we propose to probe the objective space in the vicinity of this bound as follows:

$$f^{\text{target}} = LBD + \frac{-g^{\text{LBD}}}{g^{\text{UBD}} - g^{\text{LBD}}}(UBD - LBD) \quad , \tag{8}$$

Where $g^{\text{LBD}} > 0$ and $g^{\text{UBD}} < 0$ are the violations of the semi-infinite constraint at the solutions generating the lower and upper bound, respectively, and LBD and UBD refer to the current lower and upper bounds on the objective value. The intuition behind (8) is to estimate a root of the semi-infinite constraint using the secant method, and that this approximation provides a valid proxy for the optimal objective function value, i.e., we expect the ratio $\frac{-g^{\text{UBD}}}{g^{\text{LBD}}}$ to be a good approximation of $\frac{UBD - f^*}{f^* - LBD}$, where f^* denotes the optimal objective value. In a neighborhood around the optimum, this is typically a valid assumption and can lead to significant reductions in the remaining objective space compared to bisection. We only apply the heuristic once a threshold g^{t} for the absolute value of the semi-infinite constraint is reached, i.e., $\min\{g^{\text{LBD}}, -g^{\text{UBD}}\} < g^{\text{t}}$, as we do not necessarily expect the linear approximation from the constraint space to the objective space to be sufficiently accurate far away from the minimum due to non-convexity. We present numerical experiments to determine this threshold below.

NUMERICAL RESULTS

We performed numerical experiments using the library of test problems collected in the libDIPS repository, which contains 336 SIPs and 86 GSIPs. Further details on the test set can be found in [7].

All computations were performed on the RWTH High-Performance Computing cluster CLAIX-2023. All subproblems were solved with MAiNGO v0.9.1, configured to use CPLEX v22.1.2 to solve (mixed-integer) linear or quadratic optimization problems. Additionally, multistarting in MAiNGO was performed using SLSQP, implemented as part of NLopt [19]. In some of the following experiments, we configured MAiNGO to use multiple threads. In those runs, MAiNGO set the number of threads used by CPLEX correspondingly. All other settings were left unchanged compared to those used for benchmarks in [7]. As in [7], we rounded all runtimes below 1 second (s) to 1 s, as we compare the solver configurations relative to each other and cannot fairly differentiate times below 1 s. Due to the increased memory requirement of the multithreaded implementation, 18 large

SIP instances were not tractable (despite being tractable but unsolved in [7]) and are omitted in the results below. This omission increases the relative percentage of problems solved compared to [7], however, the absolute numbers of our serial implementation are consistent with the results reported there.

Parallelization of *RRHS* and *GSIP-RRHS*

We first investigated the performance of our parallelized implementations of *RRHS* and *GSIP-RRHS* using several parameter configurations. The results in [7] already showed that the guard heuristic performs favorably for a serial implementation of the algorithm. We now compare this to our parallel implementation, both with and without the heuristic enabled. For the parallel implementation, we ran the upper- and lower-bounding procedures using a single thread each. We also re-ran the serial implementation using two threads for the subsolver to enable a fair comparison and demonstrate the difference between parallelizing the algorithmic steps and parallelizing in the subsolver.

Figure 1 shows the performance profile for all configurations of *RRHS* on the SIP problem set. The multithreaded implementation without the early refinement of the upper-bounding problem's discretization performs best, solving problems faster than the other configurations and solving a larger fraction of problems overall. Surprisingly, the early refinement does not carry its advantage from the serial case to the parallel case. We suspect that this is due to the extra discretization points not being as useful in the parallel case, as the upper-bounding problem is solved from the start. Thus, the upper-bounding problem's discretization may become unnecessarily dense in this case, leading to harder subproblems.

For unsolved instances, we define the remaining optimality gap as $\min\{UBD - LBD, \frac{UBD-LBD}{UBD}\}$. In Figure 2, we plot the percentage of unsolved problems over the remaining optimality gap for each configuration. The multithreaded implementation achieves better remaining gaps than the serial implementation, indicating that parallelizing the algorithmic steps of *RRHS* is indeed more efficient than using all available threads for the subsolver.

We show the performance profile for GSIP-*RRHS* applied to the GSIP problem set in Figure 3. Again, parallelism at the algorithmic level pays off, as the multithreaded implementations are faster and solve a slightly higher percentage of the problem set. In contrast to the SIP case, extra refinement of the upper-bounding procedure does not seem to impact the performance of the multithreaded implementation but harms the serial one.

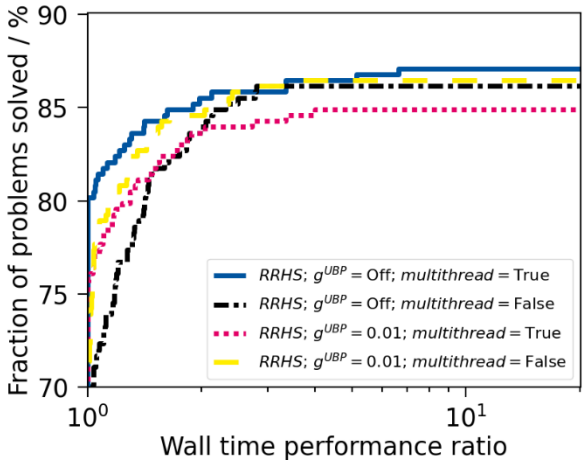

Figure 1. Dolan-Moré plot of wall time performance ratio for *RRHS* on the SIP problem test set.

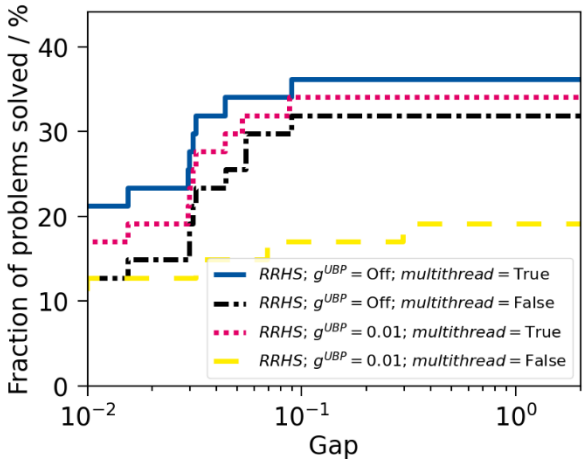

Figure 2. Fraction of SIP instances that are not solved by one of the *RRHS* configurations but would have been considered solved with a larger optimality tolerance.

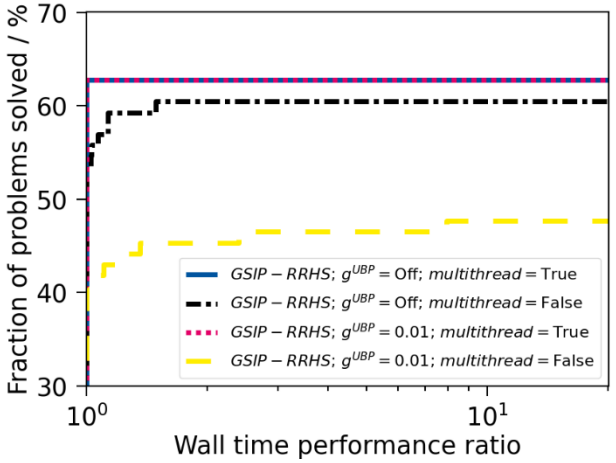

Figure 3. Dolan-Moré plot of wall time performance ratio for GSIP-*RRHS* on the GSIP problem test set.

Secant-based heuristic in *Oracle*

To determine a good value for the threshold hyperparameter g^t, we solved the benchmark set of SIPs using the serial implementation of *Oracle* while varying $g^t \in \{0, 0.01, 0.1, 0.5, 1, 1000\}$. Note that setting $g^t = 0$ disables the heuristic and setting $g^t = 1000$ effectively enables it as soon as a feasible upper bound is found. We evaluated each setting based on (rounded) runtime (RT) and the total number of subproblems (NSP) solved. We present the results in Table 1. Note that in many cases, multiple settings yielded equal numbers of subproblems or (rounded) runtimes. The number of unique winners is lower for both metrics.

The results indicate that always using the secant heuristic for target selection is superior in terms of the number of subproblems solved in many cases, while there are some problems in the test set that benefit from setting the threshold g^t to a small value or completely disabling the heuristic. In terms of runtime, all settings performed similarly due to most of the problems being solvable in under 1 s. Nevertheless, the number of unique winners exhibits qualitatively similar behavior for runtime as for the number of subproblems.

We found that enabling the heuristic resulted in up to six problems (depending on the value of g^t) of the test set not being solved that were solved without the heuristic. There were two reasons for this: in some cases, the heuristic was exact and proposed a target value equal to the minimum, in which case finite termination is not guaranteed [13]. In other cases, the initially computed upper bound was feasible and triggered the heuristic, but the ratio $\frac{-g^{\mathrm{UBD}}}{g^{\mathrm{LBD}}}$ was significantly smaller than $\frac{UBD - f^*}{f^* - LBD}$, leading to target updates close to the upper bound. In combination with a very large upper bound, this resulted in the algorithm failing to converge within the iteration limit. On the other hand, there were two instances that were solved with some settings of the heuristic but were unsolved without it.

Table 1. Statistics for the secant heuristic over the entire benchmark set. The number of unique winners is given in parentheses.

Threshold g^t	# lowest NSP	# lowest RT
0	99 (11)	217 (11)
0.01	125 (15)	228 (19)
0.1	143 (19)	220 (12)
0.5	132 (0)	215 (6)
1	139 (3)	224 (23)
1000	214 (95)	228 (28)

Parallelization of *Oracle*

We investigated multiple configurations of *Oracle* to determine the benefit of parallelism. Based on the results of the previous section, we always enabled the secant heuristic as it usually required fewer subproblems to be solved. As a baseline, we used the serial implementation, giving 16 threads to the subsolver. We then compared this to two levels of parallelism: probing four targets in parallel with four threads per subsolver and probing 16 targets in parallel with one thread per subsolver. For both configurations involving parallelism at the algorithmic level, we investigated all three strategies for the discretization as mentioned earlier.

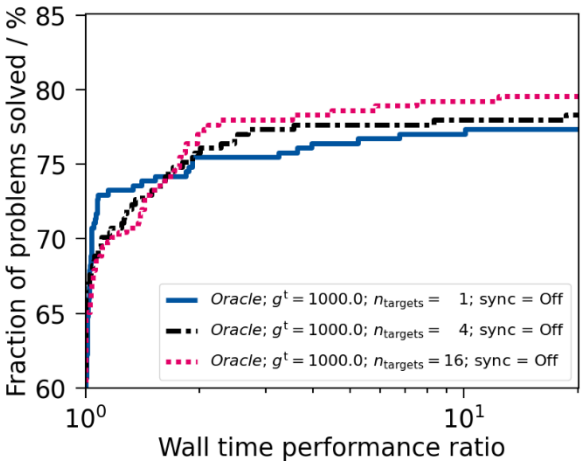

Figure 4. Dolan-Moré plot of wall time performance ratio for different parallelization strategies in *Oracle*.

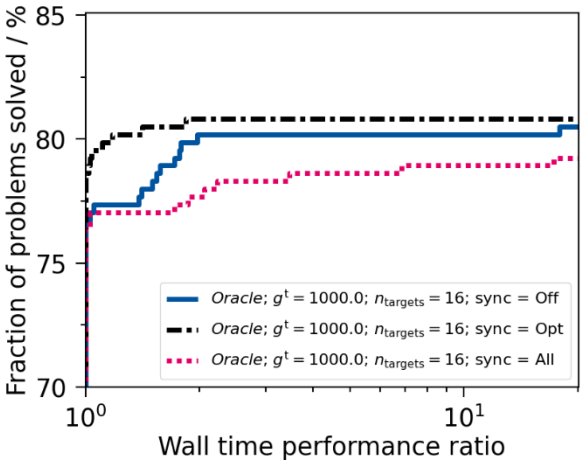

Figure 5. Wall time performance ratio for different discretization synchronization strategies in *Oracle*.

We plot the wall time performance ratio for all three parallelization strategies with no synchronization of discretizations in Figure 4. As for *RRHS*, parallelization at the algorithmic level leads to a larger percentage of the test set being solved. At small time ratios, the plot shows that the serial strategy outperforms the parallel strategies. We attribute this to the lack of a proper method to abort the optimization within MAiNGO. Thus, the parallel implementations may achieve optimality faster than their reported runtimes, as they must wait for all running

MAiNGO instances to complete before terminating. In some rare instances, this led to an estimated performance degradation by a factor of 10 or more. We aim to add an abort-functionality to MAiNGO in a future release and test this hypothesis.

In Figure 5, we plot the performance ratios of different discretization synchronization strategies when probing 16 targets in parallel. Strategy "Opt" performs best, resulting in faster solutions and solving the largest fraction of the problem set. Synchronizing every discretization point (strategy "All") performs worst, indicating that the extra information gained from more discretization points is not worth the extra computational effort involved with solving larger subproblems. Letting each thread use its own discretization with no synchronization (strategy "Off") performs in between the other two strategies, achieving similar performance to strategy "All" for fast problems (likely due to computing the same discretization points across multiple threads), but solving nearly the same fraction of the problem set overall as strategy "Opt" (likely due to the better performance of later subproblems with smaller discretizations).

CONCLUSION

We presented improved implementations of some of the algorithms for hierarchical optimization problems in libDIPS. We exploited parallelism in the algorithmic steps and demonstrated that this is a more efficient use of available threads than parallelizing only at the subsolver level, enabling faster solutions and increasing the percentage of solved problems. This observation holds for all investigated algorithms. However, parallelism increases memory usage, which led to some instances becoming intractable. Additionally, we presented a promising strategy for synchronizing the discretization used in the multithreaded implementation of the *Oracle* algorithm. The open-source nature of libDIPS enables others to implement and benchmark their algorithms.

We plan to extend parallelization in libDIPS in future releases. This includes investigating parallelization in the steps of other implemented algorithms as well as improving the existing parallelization, such as also enabling the solution of multiple lower-level problems in parallel.

DIGITAL SUPPLEMENTARY MATERIAL

libDIPS is freely available at https://git.rwth-aachen.de/avt-svt/public/libdips.

ACKNOWLEDGEMENTS

Computations were performed with computing resources granted by RWTH Aachen University. This research is funded by Réseau de transport d'electricité (RTE, France) through the project "Hierarchical Optimization for Worst-Case Analysis of Power Grids."

AUTHOR IDENTIFIERS

Author ORCIDs:
Lipow AW: 0000-0002-5860-5323
Jungen D: 0000-0002-1802-1884
Zingler A: 0000-0003-1135-0236
Djelassi H: N/A
Mitsos A: 0000-0003-0335-6566

REFERENCES

1. Blankenship JW, Falk JE. Infinitely constrained optimization problems. J Optim Theory Appl 19:261-281 (1976). https://doi.org/10.1007/bf00934096
2. Djelassi H, Mitsos A. A hybrid discretization algorithm with guaranteed feasibility for the global solution of semi-infinite programs. J Glob Optim 68:227-253 (2016). https://doi.org/10.1007/s10898-016-0476-7
3. Djelassi H, Mitsos A. Global solution of semi-infinite programs with existence constraints. J Optim Theory Appl 188:863-881 (2021). https://doi.org/10.1007/s10957-021-01813-2
4. Djelassi H, Glass M, Mitsos A. Discretization-based algorithms for generalized semi-infinite and bilevel programs with coupling equality constraints. J Glob Optim 75:341-392 (2019). https://doi.org/10.1007/s10898-019-00764-3
5. Falk JE, Hoffman K. A nonconvex max?min problem. Naval Research Logistics 24:441-450 (2006). https://doi.org/10.1002/nav.3800240307
6. Jungen D, Mitsos A. An improved oracle adaption for bilevel programs. In: Computer Aided Chemical Engineering. Ed: Manenti F, Reklaitis GV. Elsevier (2024).
7. Jungen D, Zingler A, Djelassi H, Mitsos A. libDIPS – discretization-based semi-infinite and bilevel programming solvers. *Math Prog Comp* (in press).
8. Mitsos A. Global optimization of semi-infinite programs via restriction of the right-hand side. Optimization 60:1291-1308 (2011). https://doi.org/10.1080/02331934.2010.527970
9. Mitsos A, Tsoukalas A. Global optimization of generalized semi-infinite programs via restriction of the right hand side. J Glob Optim 61:1-17 (2014). https://doi.org/10.1007/s10898-014-0146-6
10. Mitsos A, Lemonidis P, Barton PI. Global solution of bilevel programs with a nonconvex inner program. J Glob Optim 42:475-513 (2007). https://doi.org/10.1007/s10898-007-9260-z

11. Mitsos A, Bollas GM, Barton PI. Bilevel optimization formulation for parameter estimation in liquid–liquid phase equilibrium problems. Chemical Engineering Science 64:548-559 (2009). https://doi.org/10.1016/j.ces.2008.09.034
12. Swaney RE, Grossmann IE. An index for operational flexibility in chemical process design. part I: formulation and theory. AIChE Journal 31:621-630 (2004). https://doi.org/10.1002/aic.690310412
13. Tsoukalas A, Rustem B. A feasible point adaptation of the blankenship and falk algorithm for semi-infinite programming. Optim Lett 5:705-716 (2010). https://doi.org/10.1007/s11590-010-0236-4
14. Bakshalipour M, Sarbazi-Azad H. Parallelizing bisection root-finding: a case for accelerating serial algorithms in multicore substrates. https://arxiv.org/abs/1805.07269
15. International Business Machines Corporation. IBM ILOG CPLEX v 22.1.2.
16. Gurobi Optimization, LLC. Gurobi Optimizer Reference Manual. https://www.gurobi.com
17. Bongartz D, Najman J, Sass S, Mitsos A. MAiNGO – McCormick-based Algorithm for mixed-integer Nonlinear Global Optimization. http://permalink.avt.rwth-aachen.de/?id=729717
18. Zingler A, Jungen D, Djelassi H, Mitsos A. libALE – a library for algebraic-logical expression trees. https://git.rwth-aachen.de/avt-svt/public/libale
19. Johnson SG. The NLopt nonlinear-optimization package. https://github.com/stevengj/nlopt