Distributed L-shaped Algorithms in Julia

Martin Biel, Mikael Johansson

***Abstract*—We present LShapedSolvers.jl, a suite of scalable stochastic programming solvers implemented in the Julia programming language. The solvers, which are based on the L-shaped algorithm, run efficiently in parallel, exploit problem structure, and operate on distributed data. The implementation introduces several flexible high-level abstractions that result in a modular design and simplify the development of algorithm variants. In addition, we demonstrate how the abstractions available in the Julia module for distributed computing are exploited to simplify the implementation of the parallel algorithms. The performance of the solvers is evaluated on large-scale problems for finding optimal orders on the Nordic day-ahead electricity market. With 16 worker cores, the fastest algorithm solves a distributed problem with 2.5 million variables and 1.5 million linear constraints about 19 times faster than Gurobi is able to solve the extended form directly.**

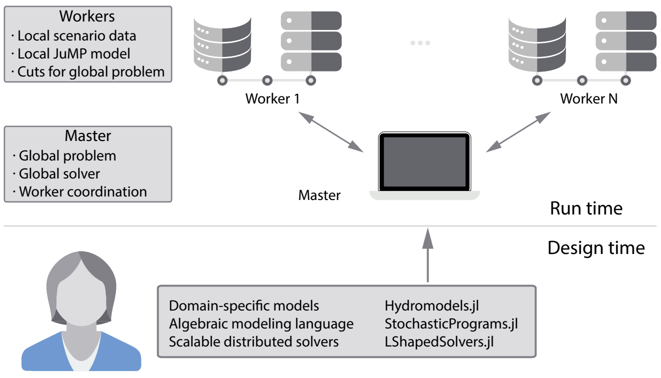

Fig. 1: Overview of software framework.

I. Introduction

Stochastic programming is a common technique for dealing with uncertain decision problems. Stochastic programs have been used to model real-world problems in diverse fields such as power systems [1], [2], [3], finance [4], [5], and transportation [6], [7]. In the simplest setting of linear two-stage stochastic programs, a first-stage decision is taken based on known information before random events occur. Once the random parameters are realised, second-stage recourse decisions, which may depend on the random outcome, can be taken to correct the initial decision. The first- and second-stage decision are modeled as optimization variables in linear programs. The accuracy of the uncertainty model is increased by including several possible future scenarios and then determining a first-stage decision that is optimal in expectation over all scenarios. The stochastic program can be formulated as a large deterministically equivalent linear program, which can be solved using standard linear programming solvers, such as open-source solvers (Clp, GLPK) or commercial solvers (Gurobi, CPLEX). However, the size of the deterministic equivalent grows with the number of scenarios. Consequently, the computation time required to solve the extended form of the stochastic program increases and standard solvers eventually become insufficient. Moreover, the memory requirement of storing the extended form eventually exceeds the capacity of a single machine. For example, the extensive form of a unit commitment problem with 16,384 scenarios amounted to 1.95 billion variables and 1.947 billion constraints, and was distributed on 8192 nodes on a Titan supercomputer [3]. Hence, solution approaches that work in parallel on distributed data are required to solve large-scale stochastic programs.

The L-shaped algorithm is a decomposition strategy that exploits the problem structure of stochastic programs. The original L-shaped algorithm [8] is an extension of Benders decomposition [9] to stochastic programs. Since the initial publication, the algorithm has been well researched [10], [11], [12], [13]. However, currently the only available general purpose implementation of L-shaped is the commerical FortSP solver [14]. The FortSP solvers are not open-source and the algorithms are implemented in serial.

This work presents `LShapedSolvers.jl` [15], a new freely available open-source solver suite implemented in the Julia [16] programming language. Julia is a dynamic language that can achieve C-like performance through type inference and just-in-time compilation. Julia includes a vast ecosystem of efficient libraries. In this work, noteworthy libraries are the Distributed module, for distributed computing, and the JuMP [17] ecosystem. JuMP includes an algebraic modeling language inside Julia, which can be used to create versatile and efficient optimization models. In addition, it implements bridges to a large number of open-source and commercial optimization solvers.

`LShapedSolvers.jl` is included in a larger software framework which supports the full process of formulating, solving and analyzing stochastic decision problems. A modeling tool, `StochasticPrograms.jl` [18], extends the JuMP modeling language with capabilities for stochastic programming models and interfaces seamlessly with the algorithms in `LShapedSolvers.jl`. The stochastic programs are automatically distributed if more processing cores are available, either locally or on remote machines. The L-shaped algorithms in `LShapedSolvers.jl` will by default run in parallel on distributed data. On a single processesor, functional serial algorithms are available as well. In addition, the framework includes a template modeling tool for domain specific models of hydropower operations, named `HydroModels.jl` [19]. This module also includes a set of predefined models, one being the day-ahead problem used to evaluate the solvers numerically.

The framework separates model design from data design. As a result, the user can formulate complex optimization models on a master node using high-level abstractions and domain level semantics. The modeling process includes specification of data dependencies, but the required data structures themselves do not have to be provided or even defined. Later, a data structure that meets the specification can be defined and loaded to create the model. This can be performed on the same local machine, or on a remote process which makes the resulting model memory-distributed. The crucial point is that domain experts can focus on formulating real-world applications into mathematical programs, and get parallel capabilities for free. This idea is illustrated in Fig. 1. In brief, the combination of these packages provides a scalable framework that can encompass large-scale stochastic programs. The focus here will primarily be on the implementation and evaluation of `LShapedSolvers.jl`.

II. Linear Two-Stage Stochastic Programs

Consider the following general form of a linear two-stage stochastic program:

$$\begin{aligned}\underset{x\in\mathbb{R}^n, y_i\in\mathbb{R}^m}{\text{minimize}} \quad & c^Tx + \sum_{i=1}^{n}\pi_i q_i^T y_i \\ \text{s.t.} \quad & Ax = b \\ & T_ix + W_iy_i = h_i, \quad i = 1,\dots,n \\ & x \geq 0, y_i \geq 0, \quad i = 1,\dots,n\end{aligned} \tag{1}$$

The idea of this formulation is to find an optimal first-stage decision $\hat{x} \in \{x \in \mathbb{R}^n \mid Ax = b,\ x \geq 0\}$ that minimizes the total expected cost over the n possible second-stage scenarios. In each second-stage scenario i, which occurs with probability π_i, a recourse decision y_i can be taken to correct any inaccuracies made in the first stage. The model can be used to approximate the more complex problem of finding an optimal decision over a complete sample space of future scenarios:

$$\begin{aligned}\underset{x\in\mathbb{R}^n}{\text{minimize}} \quad & c^Tx + \mathbb{E}_\omega[Q(x,\xi(\omega))] \\ \text{s.t.} \quad & Ax = b \\ & x \geq 0\end{aligned} \tag{2}$$

where

$$\begin{aligned}Q(x,\xi(\omega)) = \min_{y\in\mathbb{R}^m} \quad & q_\omega^T y \\ \text{s.t.} \quad & T_\omega x + Wy = h_\omega \\ & y \geq 0\end{aligned}$$

and $\omega \in \Omega$ are scenarios over some sample space and ξ is a stochastic variable that describes the optimization problem parameters for each scenario. In a sampled average approximation (SSA) approach, N scenarios $\omega_1,\dots,\omega_N$ are sampled with equal probability $\pi_i = \frac{1}{N}$ from Ω and used to formulate an instance of (1). By the law of large numbers,

$$\frac{1}{N}\sum_{i=1}^{N} q_i^T y_i$$

converges pointwise with probability 1 to

$$\mathbb{E}_\omega[Q(x,\xi(\omega))]$$

Hence, for large enough N the SSA solution will be a good approximation of the true optimal solution. Typical values of N range from 1000 to 10^6 [12], [3]. A comprehensive introduction to the field of stochastic programming is given in [20].

III. The L-shaped Algorithm

A. Nominal L-shaped Method

The implemented L-shaped solver suite comprises several multicut extensions of the original L-shaped algorithm [8]. Multicut L-shaped algorithms were first considered in [10]. In brief, the stochastic program (1) is decomposed into a master problem

$$\begin{aligned}\underset{x\in\mathbb{R}^n}{\text{minimize}} \quad & c^Tx + \sum_{i=1}^{n}\theta_i \\ \text{s.t.} \quad & Ax = b \\ & \partial Q_{i,j}x + \theta_i \geq q_{i,j}, \qquad \partial Q_{i,j}, q_{i,j} \in \mathcal{O} \\ & F_jx \geq f_j, \qquad F_j, f_j \in \mathcal{F} \\ & x \geq 0\end{aligned} \tag{3}$$

and n subproblems

$$\begin{aligned}\underset{y_i\in\mathrm{R}^m}{\text{minimize}} \quad & Q_i^k = q_i^T y_i \\ \text{s.t.} \quad & Wy_i = h_i - T_ix_k \\ & y_i \geq 0\end{aligned} \tag{4}$$

The master problem eliminates the second-stage decision variables y_i and introduces variables $\theta_i \geq Q(x,\xi(\omega_i))$. The constraints on θ_i are then replaced by increasingly accurate lower bounds on the form $\partial Q_{i,j}x + \theta_i \geq q_{i,j}$ which are referred to as optimality cuts. The optimality cuts are generated by solving the subproblems for given values of x_k corresponding to the current solution candidate of the master problem. If the current candidate is not second-stage feasible, feasibility cuts can be added to the master problem (3) through the constraint set $\mathcal{F}$. For simplicity, it is assumed from here on that stochastic programs under consideration have relatively complete recourse, so that every feasible first stage decision is also second stage feasible. This removes the need for feasibility cuts. The optimality cuts are given by

$$\pi_i\lambda_i^T T_i x + \theta_i \geq \pi_i \lambda_i^T h_i \tag{5}$$

where λ_i is the optimal dual variables for the constraints in (4). It is shown in [8] that cutting planes of this type form supports for the second stage objective $Q(x)$. Consequently,

$$\Theta_k = c^T x_k + \sum_{i=1}^{n} \theta_i^k \tag{6}$$

is a lower bound for the optimal value of (1). Morever,

$$Q_k = c^T x_k + \sum_{i=1}^{n} Q_i^k \tag{7}$$

forms an upper bound for the optimal value of (1). Thus, a termination criterion for the L-shaped procedure can be formulated as

$$|Q_k - \Theta_k| \leq \tau(\epsilon + |Q_k|) \tag{8}$$

where τ is some desired relative tolerance and ϵ is a small number to avoid division by zero.

A generic description of the L-shaped algorithms imlemented in Julia is given in Algorithm 1. Variants of

Algorithm 1 Generic L-Shaped Implementation

```
input:
                  minimize_{x∈ℝ^n, y_i∈ℝ^m}   $c^T x + \sum_{i=1}^{n} \pi_i q_i^T y_i$
    $P \leftarrow$        s.t.   $Ax = b$
                                 $T_i x + W_i y = h_i, \quad i = 1,\dots,n$
                                 $x \geq 0,\ y_i \geq 0, \quad i = 1,\dots,n$
output: $\hat{x}$, optimal to $P$, with optimal value $Q^*$.
procedure L-Shaped
    $k \leftarrow 0$
    $x_0 \leftarrow$ Crash($P$)
    $Q_0 \leftarrow \infty$
    $\Theta_0 \leftarrow -\infty$
    for $k = 0,1,2,\dots$ do
        for all scenarios $i = 1,\dots,n$ do
                          minimize_{y_i∈R^m}   $Q_i = q_i^T y_i$
            $S_i(x_k) \leftarrow$      s.t.   $W y_i = h_i - T_i x_k$
                                        $y_i \geq 0$
            $Q_i^k \leftarrow$ Solve($S_i(x_k)$)
            $\lambda_i \leftarrow$ GetConstraintDual($S_i$)
            $\mathcal{O}_k \leftarrow$ AddOptimalityCut($\mathcal{O}_{k-1}$,$\pi_i \lambda_i^T T_i x + \theta_i \geq \pi_i \lambda_i^T h_i$)
        end for
        $Q_k \leftarrow \sum_{i=1}^{n} Q_i^k$
        if $|Q_k - \Theta_{k-1}| \leq \tau(\epsilon + |Q_k|)$ then
            $\hat{x} \leftarrow x_k$
            $Q^* \leftarrow Q_k$
            return $\hat{x}, Q^*$
        end if
        $M_k \leftarrow$ FormulateMaster($x_k$,$\mathcal{O}_k$)
        $\hat{x}_k, \Theta_k \leftarrow$ Solve($M_k$)                    ▷ $\Theta_k = \sum_{i=1}^{n} \hat{\theta}_i^k$
        $x_{k+1} \leftarrow$ TakeStep($x_k$,$\hat{x}_k$,$Q_k$,$\Theta_k$)
    end for
end procedure
```

the nominal algorithm are implemented through particular implementations of the FormulateMaster(x_k,$\mathcal{O}_k$) and TakeStep(x_k,$\hat{x}_k$,Q_k,Θ_k) functions. The default multicut L-shaped algorithm corresponds to the FormulateMaster(x_k,$\mathcal{O}_k$) implementation as shown in Algorithm 2 and a TakeStep(x_k,$\hat{x}_k$,Q_k,Θ_k) implementation that just returns $\hat{x}_k$ as is. Each L-shaped algorithm is associated with a set of parameters that may require tuning for optimal performance. Also, the algorithms can make use

Algorithm 2 Nominal L-Shaped Method

```
function FormulateMaster($x$,$\mathcal{O}$)
              minimize_{x∈ℝ^n}   $c^T x + \sum_{i=1}^{n} \theta_i$
    $M \leftarrow$         s.t.   $Ax = b$
                          $\partial Q_{i,j} x + \theta_i \geq q_{i,j}, \quad \partial Q_{i,j}, q_{i,j} \in \mathcal{O}$
                          $x \geq 0$
    return $M$
end function
```

of a supplied crash procedure for finding an initial first-stage decision. The solver suite includes a set of predefined crash procedures. For example, `EVP` determines an initial decision by solving the expected value problem, obtained by replacing all scenarios with the expected scenario. A good initial guess can significantly increase the performance of the implemented solvers.

IV. Implementation

This section covers implementation details and software design of `LShapedSolvers.jl`. Concepts from the Distributed module in Julia will frequently be referred to. A quick introduction to these concepts is given in Appendix I.

A. *Distributed Stochastic Programs*

To achieve high performance and accomodate large-scale problems, it is essential that the stochastic programs to be solved are memory-distributed. To that end, `StochasticProgram.jl` has capabilities for automatically distributing stochastic programs on the available resources. In short, if several Julia processes have been made available through `addprocs`, then the second-stage subproblems are distributed on the worker nodes using channel objects. A master node maintains the first-stage problem and administers tasks and data transfers. During the L-shaped procedures, the master passes first-stage decision vectors to workers while workers compute cutting planes and return these to the master. This master-worker architecture is well suited for implementing efficient parallel L-shaped algorithms that operate on these distributed stochastic programs.

The design aims to minimize the amount of data that is being sent between computing nodes. In brief, the master initially passes model generation recipes and sampler objects to the workers. Optimization models and data are stored independently on computing nodes and workers load data in parallel. This is achieved through user-defined sampler objects that describe how to generate scenarios for a given problem. Next, the generator function created from the user defined model on the master node is passed to the worker nodes. Each worker uses the model recipe to create its subproblem in parallel. In addition, every node receives and stores a copy of the first-stage variable definition recipe. This generator is used to create an auxiliary parent model for the second stage subproblems stored on that node. Through this, a decision vector received remotely can be mapped to the correct variables and then bridged to the second stage subproblems on that node through outcome model generation. This is the basis for efficiently updating and resolving subproblems during the L-shaped procedure.

A simple example that showcases the usage of `StochasticPrograms.jl` in unison with `LShapedSolvers.jl` is given in Apppendix II.

B. Distributed L-shaped Algorithms

The parallel implementation of the L-shaped algorithms are based on an asynchronous scheme similar to the one used in [12]. In brief, the master problem (3) is solved on a master node. The resulting decision candidate is sent to workers nodes, which may or may not reside on the same machine. When a worker receives a new decision candidate, it solves the subproblem (4) for each stored scenario. The resulting cutting planes are then sent back to the master node. Each decision candidate sent to the workers is marked with a timestamp to keep track of the algorithm history. Likewise, each cutting plane generated by workers is marked with a timestamp that matches the decision vector used to generate the cut. When the master node has received enough cutting planes originating from the decision vector with the latest timestamp, then the master problem is resolved to generate a new decision with a fresh timestamp. The amount of cutting planes required is governed by an asynchronicity parameter, κ, that can be tuned. This scheme enables the master and worker tasks to be executed asynchronously, which generates efficient overlaps of useful parallel computations. If the solution of the master problem does not yield a significant change at any iterate, it might not have received enough new information due to a low value of κ. In these instances, the master process repeats the iteration at the same timestamp to allow more cuts to be generated by the workers. A simplified description of the parallel implementation is given in Algorithm 3.

In more detail, the distributed implementation revolves around four channel objects. Each worker process has a `Worker` channel that contains the second stage subproblems that are loaded on that process. The master node has a `Decisions` channel that it writes decision vectors to. In addition, each worker process has a `Work` channel that consists of integer identifiers for the decision vectors in the `Decisions` channel. After the master node has generated a new decision vector and stored it in the `Decisions` channel, it sends the timestamp of that decision to each worker process by writing to the `Work` channels. This triggers each worker to fetch the new decision candidate from the master node and resolve the subproblems. Finally, the `CutQueue` channel, again located on the master node, is written to by the workers in order to pass generated optimality cuts back to the master. Each worker also sends the current objective value of the subproblem and the timestamp worker task. This is required to properly check convergence criteria on the master node. A simplified version of the worker

Algorithm 3 Generic Parallel L-Shaped Implementation - Master Procedure

```
procedure L-Shaped
    $k \leftarrow 0$
    $t_0 \leftarrow 0$
    $x_0 \leftarrow$ Crash($P$)
    $\mathcal{D} \leftarrow$ AddDecision($x_0$)   ▷ Add initial decision to Decision channel
    $\mathcal{C} \leftarrow$ InitializeCutQueue
    $Q_0 \leftarrow \infty$
    $\Theta_0 \leftarrow -\infty$
    $\mathcal{W} \leftarrow$ InitializeWorkers($\mathcal{D}$,$\mathcal{C}$)   ▷ Each worker loads subproblems and starts the do_work task shown in Source Code 1
    QueueWork($\mathcal{W}$,0)   ▷ Workers can start working on the initial decision
    repeat
        if CutsReady then   ▷ Workers have queued new cuts
            for all $k, Q_i, C_i$ in $\mathcal{C}$ do
                $Q_{k,i} \leftarrow Q_i$
                $t_k \leftarrow t_k + 1$
                $\mathcal{O}_k \leftarrow$ AddCut($C_i$)
            end for
        end if
        if $t_k = n$ then   ▷ No more work on timestamp $k$
            $Q_k \leftarrow \sum_{i=1}^{n} Q_{k,i}$
            if $|Q_k - \Theta_{k-1}| \leq \tau(\epsilon + |Q_k|)$ then
                $\hat{x} \leftarrow x_k$
                $Q^* \leftarrow Q_k$
                return $\hat{x}, Q^*$
            end if
        end if
        if $t_k \geq \kappa n$ then   ▷ Enough work done to proceed
            $M_k \leftarrow$ FormulateMaster($x_k$,$\mathcal{O}_k$)
            $\hat{x}_k, \Theta_k \leftarrow$ Solve($M_k$)   ▷ $\Theta_k = \sum_{i=1}^{n} \hat{\theta}_i^k$
            $x_{k+1} \leftarrow$ TakeStep($x_k$,$\hat{x}_k$,$Q_k$,$\Theta_k$)
            $\mathcal{D} \leftarrow$ AddDecision($x_{k+1}$)
            QueueWork($\mathcal{W}$,$k+1$)   ▷ Send new decision vector to the workers
            $k \leftarrow k + 1$
        end if
    until done
end procedure
```

task function in `LShapedSolvers.jl` is shown in Source Code 1 The master node curates the channels for each worker and passes the appropriate ones when execution is initialized.

C. Trait-Based Method Dispatch

The software implementation of the algorithms in `LShapedSolvers.jl` is based on traits. A trait is an object-oriented concept that allows objects to adopt multiple types of behaviour without having to rely on multiple inheritance. A simple trait dispatch system for Julia is implemented in the Julia module `TraitDispatch.jl` [21], and is employed in `LShapedSolvers.jl`. Through Julia's multiple dispatch system, these traits can be efficiently matched to solver objects during just-in-time compilation. Every L-shaped algorithm is an instance of the `AbstractLShapedSolver` type. This abstraction includes most of the necessary functionality to implement the generic L-shaped algorithm shown in Algorithm 1. Consequently, every serial L-shaped variant has the same shell implementation. The variants are distinguished by the implementation of the functions FormulateMaster(x_k,$\mathcal{O}_k$) and TakeStep(x_k,$\hat{x}_k$,Q_k,Θ_k), which are specialized according to a regularization trait. This allows L-shaped algorithm variants to be implemented in a modular way. For instance, an L-shaped algorithm with an ∞-norm trust-region is implemented by adopting

```julia
function do_work!(worker::Worker,
                  work::Work,
                  cuts::CutQueue,
                  decisions::Decisions)
    subproblems::Vector{SubProblem} = fetch(worker)
    if isempty(subproblems)
       # Workers has nothing do to, return.
       return
    end
    while true
        wait(work)
        t::Int = take!(work)
        if t == -1
            # Worker finished
            return
        end
        x = fetch(decisions,t)
        @sync for subproblem in subproblems
            @async begin
                update_subproblem!(subproblem,x)
                cut = subproblem()
                Q = cut(x)
                put!(cuts,(t,Q,cut))
            end
        end
    end
end
```

Source Code 1: Simplified version of the worker tasks used in `LShapedSolvers.jl`

the `TrustRegion` trait, which specializes FormulateMaster(x_k,$\mathcal{O}_k$) and TakeStep(x_k,$\hat{x}_k$,Q_k,Θ_k) to the implementations shown in in Algorithm 6 and Algorithm 7 respectively. If no regularization trait is specified, the functionality defaults to the nominal implementations. As as example, consider the definition and implementation of the `take_step!` function shown in Source Code 2.

Distributed variants of the L-shaped algorithms are implemented through the `Parallel` trait. If an L-shaped solver is given this trait, the shell implementation becomes that of Algorithm 3 instead of the serial version.

In short, all L-shaped algorithms in `LShapedSolvers.jl` are created through combinations of the regularization traits and the parallel trait. For example, the distributed trust-region algorithm is a functor object named `DTrustRegion` whose type definition includes the lines `@implement_trait DTrustRegion TrustRegion` and `@implement_trait DTrustRegion Parallel`. As a results of the trait-based implementation, the L-shaped algorithm framework becomes modular and new algorithms can easily be designed and implemented.

V. Regularizations of the L-Shaped Method

The performance of the L-shaped algorithm can be improved through different regularization procedures. The idea of regularization is to limit the candidate search to a neighborhood of the current iterate in the master problem. This tends to lead to the generation of more effective cutting planes and faster convergence to the optimum. Moreover, regularization enables an effective use of initial decisions to warm start the L-shaped procedure. If an initial decision candidate is carefully selected, the required computation time can be significantly reduced.

```julia
# Definition of take_step!. The default behaviour
# for non-regularized algorithms is a no-operation
@define_traitfn Regularization take_step!(lshaped) = begin
    function take_step!(lshaped,!Regularization)
        nothing
    end
end

# Special implementation of take_step! for algorithms
# with the TrustRegion trait.
@implement_traitfn function take_step!(lshaped,TrustRegion)
    @unpack Q,Q̃ ,θ = lshaped.solverdata
    @unpack γ = lshaped.parameters
    need_update = false
    if Q <= Q̃  - γ*abs(Q̃ -θ)
        need_update = true
        enlarge_trustregion!(lshaped)
        lshaped.solverdata.cΔ = 0
        lshaped.ξ[:] = lshaped.x[:]
        lshaped.solverdata.Q̃  = Q
        lshaped.solverdata.major_iterations += 1
    else
        need_update = reduce_trustregion!(lshaped)
        lshaped.solverdata.minor_iterations += 1
    end
    if need_update
        set_trustregion!(lshaped)
    end
    nothing
end
```

Source Code 2: Definition of the `take_step!` trait function as well as its specialized implementation for the `TrustRegion` trait that corresponds to Algorithm 7.

1) Regularized Decomposition: The first regularization procedure is based on the regularized decomposition method by Ruszczyński [11]. Let the current candidate decision be ξ_k. The idea is to search for new candidates that are close to ξ_k in 2-norm. The modified master problem is given by Algorithm 4. Now, the solution to

Algorithm 4 Regularized Decomposition

```
function FormulateMaster($\xi$,$\mathcal{O}$)
    $M \leftarrow \begin{aligned} \underset{x\in\mathbb{R}^n}{\text{minimize}} \quad & c^Tx + \sum_{i=1}^{n}\theta_i + \frac{1}{2\sigma_k}\|x-\xi\|^2 \\ \text{s.t.} \quad & Ax = b \\ & \partial Q_{i,j}x + \theta_i \geq q_{i,j}, \quad \partial Q_{i,j}, q_{i,j} \in \mathcal{O} \\ & x \geq 0 \end{aligned}$
    return $M$
end function
```

the current master problem is only accepted as a new candidate decision if it corresponds to a decrease in the objective. If not, then σ_k is decreased to limit the search space further. If the objective is decreased, ξ_k is updated to the current master solution. Furthermore, if the current iterate yields a significant decrease, σ_k is increased to widen the search space. This update procedure is captured in the specialized TakeStep(x_k,$\hat{x}_k$,Q_k,Θ_k) function, shown in Algorithm 5. Note that these update rules deviate from those suggested in [11]. Rather, they are based on the idea of increasing or decreasing the 2-norm trust region, inspired by the update rules employed in [12]. These update rules were observed to perform better on tests problems than the original rules in the regularized decomposition method. The parameters $\bar{\sigma}$

Algorithm 5 Regularized Decomposition

```
function TakeStep($x_k$,$\hat{x}_k$,$Q_k$,$\Theta_k$)
    $\xi_k \leftarrow x_k$
    if $k = 0$ then
        $\tilde{Q}_0 \leftarrow Q_k$
    end if
    if $Q_k \leq \tilde{Q}_k - \tau(\epsilon + |Q_k|)$ then
        $\xi_k \leftarrow \hat{x}_k$
        $\tilde{Q}_k \leftarrow Q_k$
    end if
    if $Q_k \leq (1-\gamma)\tilde{Q}_k + \gamma\Theta_k$ then
        $\sigma_k \leftarrow \min\{\bar{\sigma}, 2\sigma_{k-1}\}$                    ▷ Increase trust
    else
        $\sigma_k \leftarrow \max\{\underline{\sigma}, \frac{\sigma_{k-1}}{2}\}$                    ▷ Decrease trust
    end if
    return $\xi_k$
end function
```

and $\underline{\sigma}$, that govern the search space bounds, are tuning parameters in the algorithm. The master problem is quadratic and a solver capable of solving quadratic programs is required. A linearization option is included that replaces the 2-norm term with an ∞-norm term, which allows the associated master problem to be solved as a linear program.

2) L-Shaped with Trust-Region: The second regularization procedure is based on the ∞-norm trust-region method [12]. The procedure is similar to that of regularized decomposition, but rather than penalizing deviations of x from ξ we put a hard limit Δ on how far x can be from ξ; see Algorithm 6. Note, that the trust-region

Algorithm 6 L-Shaped Method with Trust-Region

```
function FormulateMaster($\xi$,$\mathcal{O}$)
              minimize_{$x \in \mathbb{R}^n$}  $c^T x + \sum_{i=1}^{n} \theta_i$
              s.t.  $Ax = b$
    $M \leftarrow$       $F_j x \geq f_j, \quad F_j, f_j \in \mathcal{F}$
                    $\partial Q_{i,j} x + \theta_i \geq q_{i,j}, \quad \partial Q_{i,j}, q_{i,j} \in \mathcal{O}$
                    $\|x - \xi\|_\infty \leq \Delta$
                    $x \geq 0$
    return $M$
end function
```

condition is implemented as

$$-\Delta e \leq x - \xi \leq \Delta e$$

so that the master problem remains linear. The update conditions is as before to only accept a new iterate if the objective is decreased. Moreover, the search space size is increased or decreased. However, the update rules for the trust-region size are slightly more sophisticated, adopting the rules suggested in [12]. Again, the update procedure is captured in the specialized TakeStep(x_k,$\hat{x}_k$,Q_k,Θ_k) function, shown in Algorithm 7. The parameter $\bar{\Delta}$ is the maximum trust-region size. Furthermore, γ decides the acceptance tolerance of new iterates.

3) L-Shaped with Level Sets: The final regularization procedure is based on the level decomposition method [13]. This method extends the idea of regularization by including level sets. In short, the idea of only accepting iterates that yield a sufficient objective decrease is formulated as an optimization constraint. This

Algorithm 7 Trust-region

```
function TakeStep($x_k$,$\hat{x}_k$,$Q_k$,$\Theta_k$)
    $\xi_k \leftarrow x_k$
    if $k = 0$ then
        $\tilde{Q}_0 \leftarrow Q_0$
    end if
    if $Q_k \leq \tilde{Q}_k - \gamma(\tilde{Q}_k - \Theta_k)$ then                    ▷ Major iteration
        if $|Q_k - \tilde{Q}_k| \leq 0.5|\tilde{Q}_k - \Theta_k|$ and $\|\hat{x}_k - \xi_k\| = \Delta$ then
            $\Delta \leftarrow \min\{\bar{\Delta}, 2\Delta\}$                    ▷ Increase trust
        end if
        $\xi_k \leftarrow \hat{x}_k$
        $\tilde{Q}_k \leftarrow Q_k$
        counter $\leftarrow 0$
    else                    ▷ Minor iteration
        $\rho \leftarrow \min(1, \Delta)\frac{Q_k - \tilde{Q}_k}{\tilde{Q}_k - \Theta_k}$
        if $\rho > 0$ then
            counter $\leftarrow$ counter $+ 1$
        end if
        if $\rho > 3$ or ($\rho \in (1,3]$ and counter $\geq 3$) then
            $\Delta \leftarrow \frac{\Delta}{\min\{4, \rho\}}$                    ▷ Decrease trust
            counter $\leftarrow 0$
        end if
    end if
    return $\xi_k$
end function
```

is achieved by solving a projection problem when updating the current candidate decision. The master problem formulation is not altered and is therefore still given by Algorithm 2. The specialized TakeStep(x_k,$\hat{x}_k$,Q_k,Θ_k) is shown in Algorithm 8. As before, the reference to the

Algorithm 8 L-Shaped Method with Level Sets

```
function TakeStep($x_k$,$\hat{x}_k$,$Q_k$,$\Theta_k$)
    $\xi \leftarrow \hat{x}_k$
    if $k = 0$ then
        $\tilde{Q}_0 \leftarrow Q_0$
    end if
    if $Q_k \leq \tilde{Q}_k - \tau(\epsilon + |Q_k|)$ then
        $\tilde{Q}_k \leftarrow Q_k$
    end if
                  minimize_{$x \in \mathbb{R}^n$}  $\|x - \xi\|^2$
                  s.t.  $c^T x + \sum_{i=1}^{n} \theta_i \leq (1-\lambda)\Theta_k + \lambda\tilde{Q}_k$
    $M_{proj} \leftarrow$       $Ax = b$
                        $F_j x \geq f_j, \quad F_j, f_j \in \mathcal{F}$
                        $\partial Q_{i,j} x + \theta_i \geq q_{i,j}, \quad \partial Q_{i,j}, q_{i,j} \in \mathcal{O}$
                        $x \geq 0$
    $x_{k+1} \leftarrow$ Solve($M_{proj}$)
    return $x_{k+1}$
end function
```

objective value is only updated if the current candidate resulted in a sufficient decrease. The next iterate is given by any decision vector closest in 2-norm to the current candidate, that reduces the model objective to the given level

$$(1-\lambda)\Theta_k + \lambda\tilde{Q}_k$$

where λ is a tunable parameter. The linearization procedure described in Section V-.1 can be applied to the projection problem as well.

VI. Numerical Experiments

A. Day-ahead Problems

The implemented L-shaped algorithms are evaluated by solving a day-ahead planning problem. This problem involves determining optimal order strategies in a deregulated electricity market, such as the nordic market NordPool [22]. In these markets, electricity producers place orders which specify the amount of electricity that they are willing to produce at different price levels for the next day. The market price on this day-ahead market determines which of the orders that are accepted each hour and thereby the electricity volumes that each producer is responsible for producing. The market price is not known when the orders are placed and can be regarded as an uncertain parameter. Optimal orders can be generated by formulating and solving a stochastic program. The first-stage decisions are the different types of orders that can be placed each hour for the upcoming day. In each second stage scenario, the market price is known and the electricity production of the accepted electricity volumes is optimized with respect to profits and physical constraints. In addition, some recourse decisions can be taken in that any surplus or shortage of electricity can be traded on an intraday electricity market, at inferior price points. See [1] for a more detailed introduction to this type of model.

An day-ahead problem is formulated as a model recipe in `HydroModels.jl`. The problem is formualted from the perspective of a fictional hydropower producer that owns all 15 power stations in the Swedish river Skellefteälven. Physical specifications such as maximum water flow and reservoir capacity for these power stations is available in [23]. A multivariate normally distributed model of a 24 hour price curve is created from historical market prices. Specifically, the model is fit to daily price curves throughout 2017 in the nordic region, available from NordPool [24]. The model is stored in a lightweight sampler object that is passed to every worker. Using the sampler, the workers can generate subproblems corresponding to price curves using a model recipe provided by `HydroModels.jl`.

B. Serial Benchmarks

The methods are benchmarked using the Julia package `BenchmarkTools.jl`. Each solver is run once for each problem size to get a sense of the required computation time. This also allows Julia to compile all methods for maximum performance during the benchmark. The estimated times are used to run the benchmark for a long enough time to sample each run 100 times. This number of samples was deemed sufficient to get a robust estimate of the median computation time. The master problem and subproblems were solved with Gurobi in every instance. For each problem size, each L-shaped algorithm is run until a relative tolerance of 10^{-6} was reached. This gave a resulting relative tolerance of at most 10^{-5} when compared to the solution of the extended form by Gurobi. The serial benchmarks are shown in Fig. 2. The different algorithms generally return significantly different optimal solutions, although with optimal values all within a relative tolerance of 10^{-5}. This indicates that

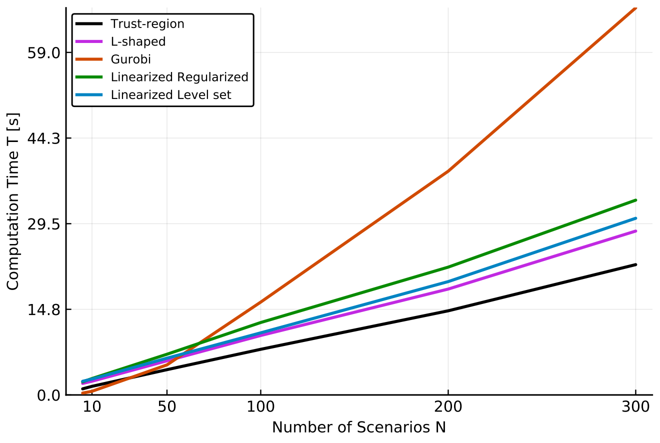

Fig. 2: Serial benchmarks of `LShapedSolvers.jl`. Median computation time required to solve a day-ahead problems as a function of the number of scenarios.

the day-ahead problem has a flat objective.

C. Distributed Benchmarks

The distributed algorithms were evaluated by solving large-scale problems of fixed size, with a varying number of worker processes. The experiments are conducted on one multi-core machine with two 3.1 GHz Intel Xeon processors (total 32 cores) and 128 GB of RAM. The generated problems have 1000 scenarios, which corresponds to an extended form of about 2.5 million variables and about 1.4 million linear constraints. For comparison, the extended form was generated once and solved with Gurobi. The total required computation time was about 351 seconds, where 139 seconds was spent generating the extended form and the remainder was spent finding an optimal solution. Next, four distributed L-shaped algorithms were benchmarked on increasing number of cores. The median computation times required for each algorithm are shown in Fig. 3. The first measurement at one core corresponds to the serial versions of the algorithms. At one core, all serial algorithm variants outperform Gurobi, which is expected from the serial benchmark trend. All distributed algorithms show signficant speedup up to four worker cores. Afterwards, the performance levels out. Strong scaling efficiencies are shown in Fig. 4. A load balance measurement was performed during one run of the nominal L-shaped method to investigate the drop in parallel efficiency. The results are shown in Fig 5. The fastest median results were obtained when κ was set to 1. To investigate why the asynchronous strategy did not improve performance, another load balance measurement was performed. The results are shown in Fig. 6.

VII. Discussion

The L-shaped algorithms are deemed effective as they solve the supplied problems to a relative tolerance of at most 10^{-5}, which is acceptable in relation to the smaller computation times. Solving the extended form

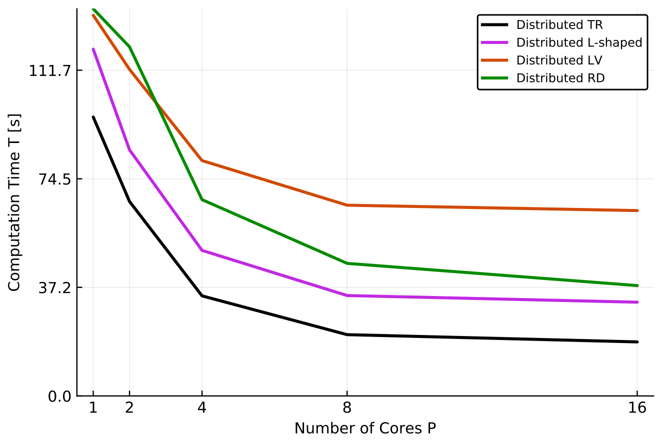

Fig. 3: Strong scaling experiment. Median computation time required to solve a day-ahead problem with 1000 scenarios as a function of processing cores. The experiment was run on four distributed variants the L-shaped algorithm.

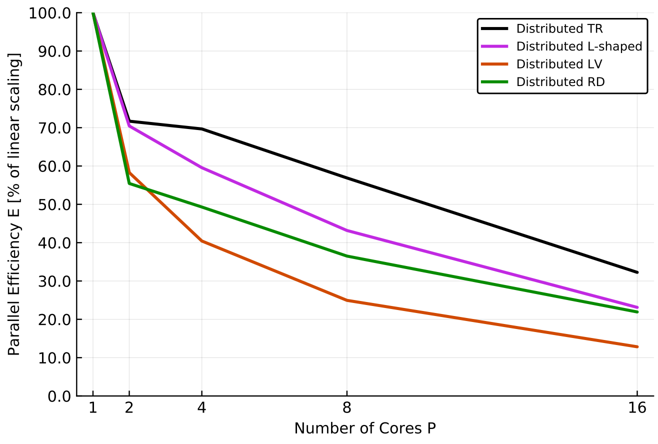

Fig. 4: Parallel efficiency as a function of number of cores, for the numerical experiment shown in Fig 3.

directly with Gurobi is more efficient at smaller problem sizes. The serial L-shaped variants start outperfoming when the number of scenarios exceed 100. Two of the regularized variants show an increase in performance compared to a standard multicut approach. The inferior performance of the regularized decomposition approach could be explained by numerical instability of the master problem. Low performance of this method has been observed before on problems with a flat objective [12]. The trust-region method is slightly more efficient than the other variants for this particular problem.

The distributed trust-region algorithm has a 30% parallel efficiency at 16 cores and is 19 times faster than solving the extended form directly with Gurobi. Almost half of the computation time required to solve the extended form directly is spent generating the deterministic equivalent. In addition, even though Gurobi can make use of all 32 cores of the machines it takes a long time to solve a linear program of this size. Solving the extended form directly is also not scalable as memory will eventually run out. Alternative approaches, such as distributed simplex methods [25] and distributed interior point methods [3], have been investigated in other works. Like our solvers, these methods perform distributed computations on distributed data. Future experiments could involve testing how these approaches compare on the same test problem. It should be noted that these tests were performed on the same machine and the latency of message passing is expected to be low. Future work includes running the experiments with a separate machine as the master node to measure the effect of message delay as well.

The drop in parallel efficiency could stem from load imbalance. The master problem generally grows by 1000 linear constraints each iteration. In contrast, the subproblems have fixed size and varying right-hand size. Consequently, solving the master problem becomes a bottleneck at larger worker counts. This is apparent in Fig 5. The effect is especially noticable in the scaling of the distributed level-set algorithm, which not only solves the master problem but also a projection problem of similar size each master iteration. The distributed trust-region method scales slightly better than the other variants. A reason for this could be that the master problem is solved relatively fast since the trust-region bounds are hit.

Fig 6 suggests one reason for the decreasing performance of asynchronous iterations. The master iterations are faster since less cuts are required to proceed. However, the generated decisions are associated with longer computation times required to solve the subproblems. The algorithm converges in 15 iterations in both cases, but not to the same optimal decision. Future work could involve benchmarks on other test problems with less flat objectives, which could give further insights.

VIII. Conclusion

A solver suite of L-shaped algorithms for stochastic programs that run in parallel on distributed data has been implemented in Julia and evaluated in numerical experiments. We have shown how distributed computation abstractions in Julia enables high-level development of complex parallel algorithms without sacrificing performance. Moreover, we used a trait based design to implement L-shaped algorithm variants with small effort. The algorithms scale well with number of scenarios and maintain high parallel efficiency in the day-ahead test problems. In particular, the distributed trust-region method achieves 30% parallel efficiency at 16 cores, and significantly outperforms solving the extended form directly. The ability to interface `LShapedSolvers.jl` with high-level modeling languages such as `HydroModels.jl` and `StochasticPrograms.jl` allows end users to rapidly formulate real-world problems as stochastic programs and solve them efficiently in a distributed setting. The framework is open-source and freely available on Github.

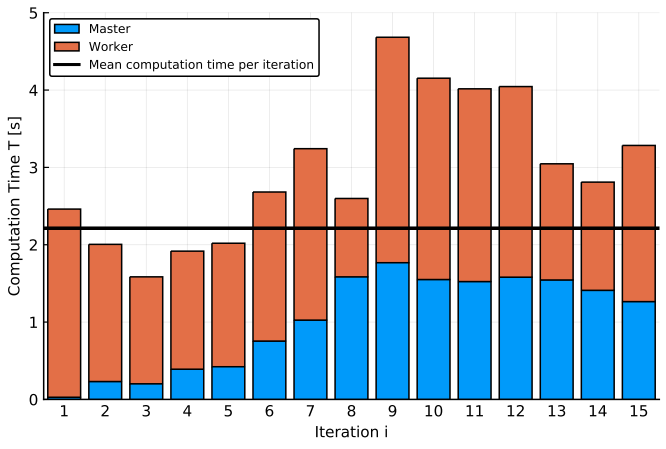

Fig. 5a: Load balance measurement of the nominal L-shaped method on 4 worker cores, when $\kappa = 1.0$

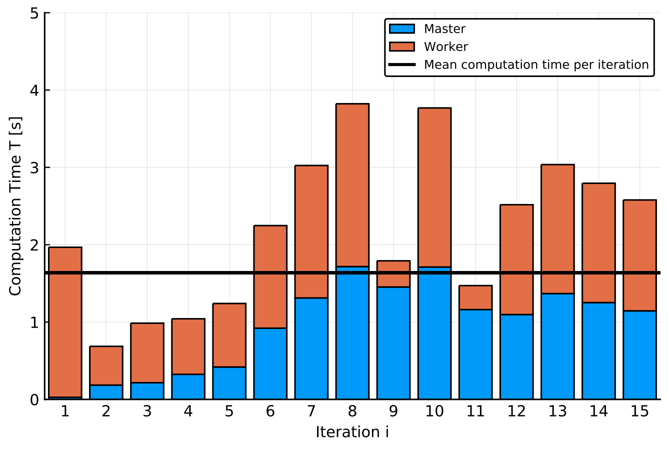

Fig. 5b: Load balance measurement of the nominal L-shaped method on 16 worker cores, when $\kappa = 1.0$

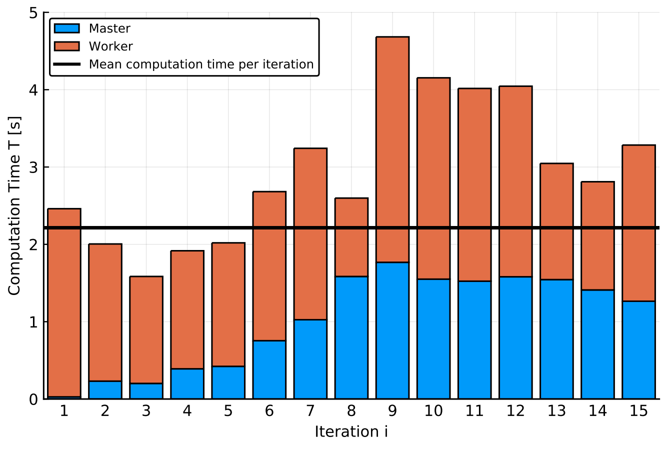

Fig. 6a: Load balance measurement of the nominal L-shaped method on 4 worker cores, when $\kappa = 1.0$.

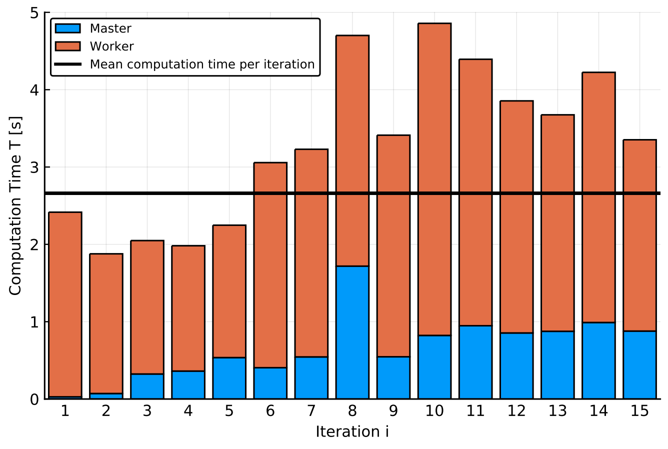

Fig. 6b: Load balance measurement of the nominal L-shaped method on 4 worker cores, when $\kappa = 0.5$.

Appendix I
Distributed Computing in Julia

The aim of this section is to introduce enough concepts from the Distributed module in Julia to make this work self contained. For further reference, consult the Julia documentation [26].

Distributed computing in Julia revolves around two primitives: remote references and remote calls. Remote references are used to administer which computing node data resides on. In addition, they provide the remaining processes access to the remote data. Remote calls are used to schedule execution tasks on the distributed data. A worker process is explicitly specified and is then given a Julia function to execute remotely on given arguments. The remote call returns a `Future`, which is a remote reference to the result of the computation. Remote calls are sent asynchronously and returns immediately. The calling process can `wait` on the future, which blocks until the computation has finished on the remote process, and `fetch` the result when it is ready. A simple example is shown in Source Code 3. These abstractions make it easy to design asynchronous algorithm schemes, as the master

```julia
# Add another Julia process
julia> addprocs(1);
# Second a vector of 5 ones to the second process
# and ask it to sum up the elements
julia> result = remotecall(sum,2,ones(5))
# Wait until the worker process has finished
julia> wait(result);
# Fetch the result
julia> fetch(result)
5.0
```

Source Code 3: Remote call example

process can do useful work while the worker tasks are running.

A `Channel` is a remote reference that administers data on one process that is readable and writable from all processes. Channels provide a flexible tool to pass around data and worker tasks between processes. A simple example of channel usage is shown in Source Code 4. Since the `work` channel resides on the second process, the `put!` calls from the master process involves data passing. In contrast, the `take!` call on the worker just fetches received data. This simple example is a good representation of

```
# Add another Julia process
julia> addprocs(1);
# The worker task function waits for incoming integers and
# sleeps for that many seconds, until -1 is received
julia> @everywhere function do_work(work::RemoteChannel)
           while true
               t = take!(work)
               if t == -1
                   # Work finished
                   println("I am done!")
                   return
               end
               # do work
               sleep(t)
               println("I finished work \$t")
           end
       end
# The work channel has a capacity of 3 integers
# that will reside on process 2
julia> work = RemoteChannel(()->Channel{Int}(3),2);
# The master process starts the worker taks on process 2
# and sends three assignments, as well -1 to finish
julia> begin
           active_worker = remotecall(do_work,2,work)
           put!(work,1)
           put!(work,2)
           put!(work,3)
           put!(work,-1)
           wait(active_worker)
       end
      From worker 2:         I finished work 1
      From worker 2:         I finished work 2
      From worker 2:         I finished work 3
      From worker 2:         I am done!
```

Source Code 4: Simple example of `Channel` usage in Julia

the data passing and work delegation that occurs in `LShapedSolvers.jl`.

Appendix II
Simple Example

Consider the following simple stochastic program:

$$\begin{aligned} \underset{x_1,x_2\in\mathbb{R}}{\text{minimize}} \quad & 100x_1 + 150x_2 + \mathbb{E}_\omega[Q(x_1,x_2,\xi)] \\ \text{s.t.} \quad & x_1 + x_2 \leq 120 \\ & x_1 \geq 40 \\ & x_2 \geq 20 \end{aligned} \tag{9}$$

where

$$\begin{aligned} Q(x_1,x_2,\xi) = \underset{y_1,y_2\in\mathbb{R}}{\text{minimize}} \quad & q_1(\xi)y_1 + q_2(\xi)y_2 \\ \text{s.t.} \quad & 6y_1 + 10y_2 \leq 60x_1 \\ & 8y_1 + 5y_2 \leq 80x_2 \\ & 0 \leq y_1 \leq d_1(\xi) \\ & 0 \leq y_2 \leq d_2(\xi) \end{aligned} \tag{10}$$

In `StochasticPrograms.jl`, the problem (9) is modeled as shown in Source Code 5. where `scenario` is a placeholder keyworder that will later be populated with scenario data. The scenarios are represented by simple Julia structures, containing the random parameters $d(\xi)$ and $q(\xi)$. An optimal first stage decision can be obtained either by solving the extended form by supplying a standard solver, or using an L-shaped algorithm. Each algorithm variant in `LShapedSolvers.jl` is represented by a Julia functor object. For convenience, there is a factory function,

```
struct SimpleScenario <: AbstractScenarioData
    π::Probability
    d::Vector{Float64}
    q::Vector{Float64}
end
s1 = SimpleScenario(0.4,[500.0,100],[-24.0,-28])
s2 = SimpleScenario(0.6,[300.0,300],[-28.0,-32])

sp = StochasticProgram([s1,s2])

@first_stage sp = begin
    @variable(model, x₁ >= 40)
    @variable(model, x₂ >= 20)
    @objective(model, Min, 100*x₁ + 150*x₂)
    @constraint(model, x₁+x₂ <= 120)
end

@second_stage sp = begin
    @decision x₁ x₂
    s = scenario
    @variable(model, 0 <= y₁ <= s.d[1])
    @variable(model, 0 <= y₂ <= s.d[2])
    @objective(model, Min, s.q[1]*y₁ + s.q[2]*y₂)
    @constraint(model, 6*y₁ + 10*y₂ <= 60*x₁)
    @constraint(model, 8*y₁ + 5*y₂ <= 80*x₂)
end
```

Source Code 5: Definition of (9) in `StochasticPrograms.jl`.

```
LShapedSolver(variant, lpsolver; kw...)
```

Source Code 6: Function signature for L-shaped solver interface

`LShapedSolver`, that interfaces as a supported solver for stochastic programs created in `StochasticPrograms.jl`. The function signature of the factory function is shown in Source Code 6. The `variant` symbol specifies which L-shaped algorithm variant should be used. Furthermore, a third-party solver, capable of solving the corresponding master problem, must be supplied as the `lpsolver` argument. Solver-specific tuning parameters can be set by supplying keyword arguments. A crash method can be supplied through the `crash` keyword. Source Code 7 exemplifies model solving.

```
julia> solve(sp,solver=GurobiSolver());

julia> optimal_value(sp)
-855.83

julia> solve(sp,solver=LShapedSolver(:ls,GurobiSolver()))
L-Shaped Gap  Time: 0:00:01 (6 iterations)
  Objective:       -855.8333333333339
  Gap:             0.0
  Number of cuts:  7

julia> optimal_value(sp)
-855.83
```

Source Code 7: Solving the simple stochastic program (9) through the extended form with Gurobi and with the nominal L-shaped Algorithm.

References

[1] S.-E. Fleten and T. K. Kristoffersen, “Stochastic programming for optimizing bidding strategies of a nordic hydropower producer,” *European Journal of Operational Research*, vol. 181, no. 2, pp. 916–928, sep 2007.

[2] N. Gröwe-Kuska and W. Römisch, “Stochastic unit commitment in hydrothermal power production planning,” in *Applications of Stochastic Programming.* Society for Industrial and Applied Mathematics, jan 2005, pp. 633–653.

[3] C. G. Petra, O. Schenk, and M. Anitescu, “Real-Time Stochastic Optimization of Complex Energy Systems on High-Performance Computers,” *Computing in Science Engineering*, vol. 16, no. 5, pp. 32–42, Sep. 2014.

[4] P. Krokhmal, S. Uryasev, and G. Zrazhevsky, “Numerical comparison of conditional value-at-risk and conditional drawdown-at-risk approaches: Application to hedge funds,” in *Applications of Stochastic Programming.* Society for Industrial and Applied Mathematics, jan 2005, pp. 609–631.

[5] S. A. Zenios, “Optimization models for structuring index funds,” in *Applications of Stochastic Programming.* Society for Industrial and Applied Mathematics, jan 2005, pp. 471–501.

[6] W. B. Powell, “An operational planning model for the dynamic vehicle allocation problem with uncertain demands,” *Transportation Research Part B: Methodological*, vol. 21, no. 3, pp. 217–232, jun 1987.

[7] W. B. Powell and H. Topaloglu, “Fleet management,” in *Applications of Stochastic Programming.* Society for Industrial and Applied Mathematics, jan 2005, pp. 185–215.

[8] R. Van Slyke and R. Wets, “L-Shaped Linear Programs with Applications to Optimal Control and Stochastic Programming,” *SIAM Journal on Applied Mathematics*, vol. 17, no. 4, pp. 638–663, Jul. 1969. [Online]. Available: http://epubs.siam.org/doi/abs/10.1137/0117061

[9] J. F. Benders, “Partitioning Procedures for Solving Mixed-variables Programming Problems,” *Numer. Math.*, vol. 4, no. 1, pp. 238–252, Dec. 1962, original publicationo of Bender’s decomposition. [Online]. Available: http://dx.doi.org/10.1007/BF01386316

[10] J. R. Birge and F. V. Louveaux, “A multicut algorithm for two-stage stochastic linear programs,” *European Journal of Operational Research*, vol. 34, no. 3, pp. 384–392, mar 1988.

[11] A. Ruszczyński, “A regularized decomposition method for minimizing a sum of polyhedral functions,” *Mathematical Programming*, vol. 35, no. 3, pp. 309–333, Jul. 1986. [Online]. Available: https://link.springer.com/article/10.1007/BF01580883

[12] J. Linderoth and S. Wright, “Decomposition Algorithms for Stochastic Programming on a Computational Grid,” *Computational Optimization and Applications*, vol. 24, no. 2-3, pp. 207–250, Feb. 2003. [Online]. Available: https://link.springer.com/article/10.1023/A:1021858008222

[13] C. I. Fábián and Z. Szőke, “Solving two-stage stochastic programming problems with level decomposition,” *Computational Management Science*, vol. 4, no. 4, pp. 313–353, aug 2006.

[14] F. Ellison, G. Mitra, C. Poojari, and V. Zverovich, “Fortsp: A stochastic programming solver,” http://www.optirisk-systems.com/manuals/FortspManual.pdf, 2009.

[15] M. Biel, “Lshapedsolvers.jl,” https://github.com/martinbiel/LShapedSolvers.jl, 2018.

[16] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah, “Julia: A fresh approach to numerical computing,” *SIAM Review*, vol. 59, no. 1, pp. 65–98, jan 2017.

[17] I. Dunning, J. Huchette, and M. Lubin, “JuMP: A modeling language for mathematical optimization,” *SIAM Review*, vol. 59, no. 2, pp. 295–320, jan 2017.

[18] M. Biel, “Stochasticprograms.jl,” https://github.com/martinbiel/StochasticPrograms.jl, 2018.

[19] ——, “Hydromodels.jl,” https://github.com/martinbiel/HydroModels.jl, 2018.

[20] J. R. Birge and F. Louveaux, *Introduction to Stochastic Programming.* Springer New York, 2011.

[21] M. Biel, “Traitdispatch.jl,” https://github.com/martinbiel/TraitDispatch.jl, 2018.

[22] NordPool, https://www.nordpoolgroup.com/, 2018.

[23] J. J. Sag, “Simulation of hydro power expansion in skellefteälven,” Master’s thesis, KTH, Optimization and Systems Theory, 2018.

[24] NordPool, “Hourly Elspot prices 2017 in EUR,” https://www.nordpoolgroup.com/globalassets/marketdata-excel-files/elspot-prices_2017_hourly_eur.xls, 2018.

[25] M. Lubin, J. A. J. Hall, C. G. Petra, and M. Anitescu, “Parallel distributed-memory simplex for large-scale stochastic LP problems,” *Computational Optimization and Applications*, vol. 55, no. 3, pp. 571–596, Jul. 2013. [Online]. Available: https://link.springer.com/article/10.1007/s10589-013-9542-y

[26] “Julia documentation: Parallel computing,” https://docs.julialang.org/en/stable/manual/parallel-computing/, 2018.

Appendix III
Artifact Description: Distributed L-shaped Algorithms in Julia

A. Abstract

The following appendix describes the dependencies and experimental setup used to generate the numerical results presented in this work. Specifically, the generation of Fig. 2, Fig. 3 and Fig. 4 is outlined to an extent where it should be possible to reproduce the results, assuming similar hardware is available. The dependencies can either be installed by following the instructions in this document, or by mounting a predefined docker file.

B. Description

1) Check-list (artifact meta information):

- **Algorithm: L-shaped**
- **Program: Julia**
- **Data set: `plantdata.csv`, `spotprices.csv`**
- **Run-time environment: Archlinux**
- **Hardware: 32-core machine**
- **Execution: Serial, Parallel**
- **Output: Serial and distributed benchmarks**
- **Publicly available?: Github**

2) How software can be obtained: All implemented code is available freely on Github:

- HydroModels.jl: https://github.com/martinbiel/HydroModels.jl
- StochasticPrograms.jl: https://github.com/martinbiel/StochasticPrograms.jl
- LShapedSolvers.jl: https://github.com/martinbiel/LShapedSolvers.jl
- TraitDispatch.jl: https://github.com/martinbiel/TraitDispatch.jl

The most convenient approach for reproducing results is to fetch the modules directly into Julia, as shown in the installation section below.

3) Hardware dependencies: The numerical experiments were performed on a server node (32 processing cores in total) with the following specifications.

- **Processor:** Two Intel Xeon E5-2687W (Eight Core, 3.10GHz Turbo, 20MB, 8.0 GT/s)
- **Memory:** 128GB (16x8GB) 1600MHz DDR3 ECC RDIMM

4) Software dependencies:

- Julia v0.6.4
- g++
- make
- Python 3.6.6 with matplotlib 2.2.2
- GMP
- Gurobi 7.0.2
- Docker (optionally)

5) Datasets: Two datasets were used. First, `plantdata.csv` contains physical specifications for the hydroplants in the river Skellefteälven. This data was first published in [23], in Table 1 and Table 2. Second, `spotprices.csv` should contain the hourly market price of electricity in the Nordic region during 2017. This data is available at NordPool [24]. Note, that line 2022 has no price data. Either remove this line or interpolate from the surrounding data. Alternatively, the `spotprices.csv` in the auxilliary repository described below contains dummy data that can be used instead.

C. Installation (Docker)

A docker image with all necessary binaries is available named `mbiel/paw-atm-reproducibility`. To use it, install docker. Next, run

```
docker run --interactive --tty mbiel/paw-atm-reproducibility
```

After fetching the binaries a Julia prompt with all necessary libraries should appear, and one can proceed to follow the instructions in `reproducibility.jl`, i.e. the experiment workflow. Note, that it is not possible to redistribute Gurobi in docker, so the premade environment uses Clp instead. See the notes at the end of the document for a discussion of the consequences of using Clp. All files in the docker image are laid out flat in one directory, so do not use `"results/"` and `"data/"` prefixes when loading files. Also, there is no GUI backend in docker, so plots will not actually be displayed. Figures can still be saved as pdfs through `savefig(plot(X),"X.pdf")`. The result benchmarks can be loaded and compared to new benchmarks directly in the Julia prompt as well.

D. Installation (Manual)

The numerical experiments were performed on Julia version v0.6.4, available at Github. For best performance, it is recommended to build Julia from source, according to the instructions at the Julia Github page. The required Julia packages are installed in Julia as follows:

```
Pkg.add("JuMP")
Pkg.add("MacroTools")
Pkg.add("Parameters")
Pkg.add("Reexport")
Pkg.add("BenchmarkTools")
Pkg.add("ProgressMeter")
Pkg.add("PlotRecipes")
Pkg.add("Plots")
# Requires matplotlib
Pkg.add("PyPlot")
# Requires g++, make
Pkg.add("Clp")
# Requires libgmp-dev
Pkg.add("GLPKMathProgInterface")
# Requires Gurobi to be installed with a valid license
Pkg.add("Gurobi")
```

The plots in this work were generated with the PyPlot backend, which requires matplotlib to be installed. A Clp binary will be installed automatically by the above command. However, all results in this work were generated using Gurobi version 7.0.2, which needs to be installed separately along with a valid license. Gurobi has free licenses available for academic users. The user made Julia modules used to generate the results in this work are installed as follows:

```
Pkg.clone(
"https://github.com/martinbiel/TraitDispatch.jl.git")
Pkg.clone(
"https://github.com/martinbiel/StochasticPrograms.jl.git")
Pkg.clone(
"https://github.com/martinbiel/HydroModels.jl.git")
Pkg.clone(
"https://github.com/martinbiel/LShapedSolvers.jl.git")
```

which fetches and install the packages from Github. Before starting, it is advised to precompile the installed packages as follows:

```
using JuMP
using BenchmarkTools
using Clp
using Gurobi
using StochasticPrograms
using HydroModels
using LShapedSolvers
```

The full installation procedure of Julia and all the required packages can be expected to take a while, up to a couple of hours on a bare setup. Finally, a public repository that contains auxilliary scripts and datasets that can be used to reproduce the results of the paper can be fetched through

```
  git clone
  https://github.com/martinbiel/paw-atm-reproducibility.git
```

E. Experiment workflow

To reproduce the results presented in Fig. 2, Fig. 3 and Fig. 4, run the code blocks in `reproducibility.jl`. In summary, the workflow is as follows:

```
include("dayahead_benchmark.jl")
```

to load auxilliary functions for benchmarking the day-ahead test problems. The functions expect a `data` folder in the same directory where `plantdata.csv` and `spotprices.csv` are located. Next,

```
serial_benchmark = prepare_sbenchmark()
warmup_benchmarks!(serial_benchmark)
```

The above prepares the serial benchmarks. This runs through all benchmarks to ensure they are compiled and to get a sense of how long time they require. Finally,

```
run_benchmarks!(serial_benchmark)
save_results!(serial_benchmark,"serial_benchmarks.json")
```

runs the benchmark and saves the results. This is a lengthy process which typically takes hours. For each problem size, the benchmark has each algorithm solving the problem about 100 times and samples the required computation time. For distributed benchmarks, Julia processes have to be added to create worker cores through `addprocs`. For example,

```
addprocs(2)
include("dayahead_benchmark.jl")
distributed_benchmark = prepare_dbenchmark()
warmup_benchmarks!(distributed_benchmark)
run_benchmarks!(distributed_benchmark)
save_results!(distributed_benchmark,"scaling_2.json")
```

runs benchmarks of the distributed algorithms with two worker cores. For small tests, individual models can be loaded and solved as follows:

```
nscen = 1000
dayahead_model = load_model(nscen)
HydroModels.plan!(dayahead_model,optimsolver=LShapedSolver(...))
```

F. Evaluation and expected result

Following the instructions in `reproducibility.jl` should produce results akin to the ones presented in Fig. 2, Fig. 3 and Fig. 4. These results are available as JSON files in the results folder of the paw-atm-reproducibility repository as well. The presented results can be regenerated as follows

```
using Plots
pyplot()
serial_benchmark = load_results(
"results/serial_benchmarks.json")
plot(TimePlot(serial_benchmark)) # Produces Fig. 2
distributed_benchmark = load_results(
"results/distributed_benchmarks.json")
plot(TimePlot(serial_benchmark)) # Produces Fig. 3
plot(ScalingPlot(serial_benchmark)) # Produces Fig. 4
```

G. Experiment customization

The serial and distributed L-shaped algorithms evaluated in the benchmarks are created in the functions `prepare_sbenchmark` and `prepare_dbenchmark`. The keyword arguments supplied to the solver calls can be changed for experiment customization. For example, removing the `linearize=true` in the `lv` creation will revert to using a 2-norm term in the level-set solver. Use `?LShapedSolver` in the Julia prompt after `using LShapedSolvers` for helper documentation that explains how to create and parametrize L-shaped solvers. For information about the the tunable parameters, and their default values, each algorithm has a separate docstring accessible through `?`. For example, run `?LShaped`. All available docstrings are listed by `?LShapedSolver`.

H. Notes

If a Gurobi license is not available, the open-source Clp solver could be used as a subsolver instead. However, convergence issues have been observed if it is used as is. Successful results were obtained by choosing a lower presolve level in Clp. The reason for this is not known. To use Clp instead, replace each `GurobiSolver(OutputFlag=0)` in `dayahead_benchmark.jl` to `ClpSolver(PresolveType=2)`. This yields a significant decrease in performance, so the presented computational results are not expected to be reproduced. Glpk can also be used, but it is even slower. The linearization option in the `:rd` and `:lv` solvers are not supported by Clp, but work with Glpk.