Advances in Engineering Software 136 (2019) 102695

Contents lists available at ScienceDirect

Advances in Engineering Software

journal homepage: www.elsevier.com/locate/advengsoft

Research paper

POLO.Jl: Policy-based optimization algorithms in Julia

Martin Biel[*,a], Arda Aytekin[a], Mikael Johansson[a]

KTH Royal Institute of Technology, School of Electrical Engineering and Computer Science, Stockholm SE-100 44, Sweden

ABSTRACT

We present `POLO.jl` — a Julia package that helps algorithm developers and machine-learning practitioners design and use state-of-the-art parallel optimization algorithms in a flexible and efficient way. `POLO.jl` extends our C+ + library `POLO`, which has been designed and implemented with the same intentions. `POLO.jl` not only wraps selected algorithms in `POLO` and provides an easy mechanism to use data manipulation facilities and loss function definitions in Julia together with the underlying compiled C+ + library, but it also uses the policy-based design technique in a Julian way to help users prototype optimization algorithms from their own building blocks. In our experiments, we observe that there is little overhead when using the compiled C+ + code directly within Julia. We also notice that the performance of algorithms implemented in pure Julia is comparable with that of their C+ + counterparts. Both libraries are hosted on GitHub[1]under the free MIT license, and can be used easily by pulling the pre-built 64-bit architecture Docker images.[2]

1. Introduction

Many key problems in statistics, image processing and machine-learning can be reduced to solving large-scale optimization problems. Examples include sparse regression, image classification and neural network training, to name a few. Improved optimization algorithms allow these applications to run faster, produce more accurate results, and deal with more data. The recent surge in machine-learning research has spurred an intense effort in developing scalable optimization algorithms with low memory and iteration complexity. For a given problem class, there are often several dozens of competing algorithms (see Section 2), and new ones continue to appear at a high pace. In such a dynamic environment, it is essential to be able to quickly prototype new algorithms, benchmark them against existing proposals, and swiftly put improved state-of-the art algorithms to practice.

This paper introduces `POLO.jl`, a Julia package that implements state-of-the-art *gradient*-based algorithms for solving *continuous* optimization problems on different executors, and provides necessary building blocks for the users to fine-tune and extend both algorithms and executors in many different ways. It is worth noting that there exist other search- and sampling-based methods such as [1–3], which do not require the evaluation of gradients and have theoretical support, as well as other libraries such as SYMPHONY [4], which are aimed primarily for *discrete* optimization problems (e.g., mixed-integer programs), which are all orthogonal to the functionalities that `POLO.jl` offers. The idea of policy-based design is central to the implementation of our library. On one hand, `POLO.jl` can be used as an interface to our C+ + library, `POLO` [5], to benefit from and extend the many compiled algorithms and executors in C+ + without much overhead. Serial and distributed-memory parallel algorithms provided by `POLO` can be easily used within Julia through this interface, and `POLO.jl` interfaces seamlessly with many other packages in the Julia ecosystem. On the other hand, `POLO.jl` attempts to follow the same design approach in `POLO` to decompose algorithms into their building blocks, and implements shared-memory parallel algorithms using Julia's experimental multi-threading support.

The paper is organized as follows. In the next section, we motivate the need for `POLO.jl` based on our observations from a class of problems and algorithms. In Section 3, we give an overview of our library. In the subsequent sections, we show how the underlying C+ + library can be used and extended from Julia (Section 4), how different building blocks of algorithms can be prototyped in Julia (Section 5) and which parallel computation settings are covered by the library (Section 6) with specific examples. In Section 7, we conclude the paper with final remarks and give possible directions for further improvements.

2. Motivation

A large number of data science applications rely on solving *regularized optimization problems* on the form

$$\underset{x\in\mathbb{R}^d}{\text{minimize}}\ \phi(x) := F(x) + h(x). \tag{1}$$

Here, $F(\cdot)$ is a sum of N differentiable component functions of x, i.e., $F(x) = \sum_{n=1}^{N} f_n(x)$, and $h(\cdot)$ is a possibly non-differentiable function.

* Corresponding author.
E-mail addresses: mbiel@kth.se (M. Biel), aytekin@kth.se (A. Aytekin), mikaelj@kth.se (M. Johansson).

[1] https://github.com/pologrp

[2] https://hub.docker.com/r/pologrp/polo-julia/

https://doi.org/10.1016/j.advengsoft.2019.102695
Received 21 October 2018; Received in revised form 11 June 2019; Accepted 20 July 2019
Available online 02 August 2019

Table 1
Some members of the proximal gradient methods. `s`, `cr`, `ir` and `ps` under the `execution` column stand for serial, consistent read/write, inconsistent read/write and Parameter Server [20], respectively.

	g			$h(x)$	
Algorithm	`boosting`	`smoothing`	`step`	`prox`	`execution`
SGD	×	×	γ, γ_k	×	`s, cr, ps`
IAG [7]	`aggregated`	×	γ	×	`s, cr, ps`
PIAG [8]	`aggregated`	×	γ	✓	`s, cr, ps`
SAGA [9]	`saga`	×	γ	✓	`s, cr, ps`
Momentum [10]	`classical`	×	γ	×	`s`
Nesterov [11]	`nesterov`	×	γ	×	`s`
AdaGrad [12]	×	`adagrad`	γ	×	`s`
AdaDelta [13]	×	`adadelta`	γ	×	`s`
Adam [14]	`classical`	`rmsprop`	γ	×	`s`
Nadam [15]	`nesterov`	`rmsprop`	γ	×	`s`
AdaDelay [16]	×	×	γ_k	✓	`s, cr, ps`
HOGWILD! [17]	×	×	γ	×	`ir`
ASAGA [18]	`saga`	×	γ	×	`ir`
ProxASAGA [19]	`saga`	×	γ	✓	`ir`

The smooth function typically represents the empirical data loss. For example, it could quantify the cost of errors between predicted and true measurements in a regression problem, or quantify the error rate in a classification problem. The non-smooth part of the objective, on the other hand, is typically a regulariser that encourages certain properties, such as sparsity, of the optimal solution.

One approach for solving problems on the form (1) is to use *proximal gradient methods*. The basic form of the proximal gradient iteration is

$$x_{k+1} = \underset{x \in \mathbb{R}^d}{\arg\min} \{F(x_k) + \langle \nabla F(x_k), x - x_k \rangle + h(x) + \frac{1}{2\gamma_k} \|x - x_k\|_2^2\}, \tag{2}$$

where γ_k is the step size parameter. Thus, the next iterate, x_{k+1}, is selected to be the minimiser of the sum of the first-order approximation of the differentiable loss function around the current iterate, x_k, the non-differentiable loss function, and a quadratic penalty on the deviation from the current iterate [6]. After some algebraic manipulations, one can rewrite (2) in terms of the proximal operator [6]

$$\begin{aligned} x_{k+1} &= \underset{x \in \mathbb{R}^d}{\arg\min} \{\gamma_k h(x_k) + \frac{1}{2} \|x - (x_k - \gamma_k \nabla F(x_k))\|_2^2\} \\ &:= \operatorname{prox}_{\gamma_k h}(x_k - \gamma_k \nabla F(x_k)). \end{aligned}$$

As a result, the method can be interpreted as a two-step procedure: first, a query point is computed by modifying the current iterate in the direction of the negative gradient, and then the prox operator is applied to this query point.

Even though the proximal gradient method described in (2) looks involved, the prox-mapping is available in closed form for several regularization functions $h(\cdot)$. The gradient calculation step in the original proximal gradient method can be prohibitively expensive when the number of data points (N) or the dimension of the decision vector (d) is large enough. To improve the scalability of the proximal gradient method, researchers have long tried to come up with ways of parallelizing the proximal gradient computations and also use more clever query points [7–19]. As a result, there exist numerous variations of the proximal gradient algorithm, some of which are listed in Table 1.

As the table indicates, the family of proximal gradient algorithms differ in their choices of five distinctive algorithm primitives: (1) which gradient surrogate they use; (2) how they combine multiple gradient surrogates to form a search direction, a step we refer to as *boosting*; (3) how this search direction is filtered or scaled, which we call *smoothing*; (4) how the step size is chosen; and (5) the type of projection they do in the proximal step. The main idea of policy-based optimization revolves around exploiting this natural decomposition of the proximal gradient algorithm. For instance, stochastic gradient descent (SGD) algorithms use partial gradient information coming from component functions (N) or decision vector coordinates (d) as the gradient surrogate at each iteration, whereas their aggregated versions [7–9] accumulate previous partial gradient information to boost the descent direction. Similarly, different momentum-based methods such as the classical [10] and Nesterov's [11] momentum accumulate the full gradient information over iterations. Algorithms such as AdaGrad [12] and AdaDelta [13], on the other hand, use the second-moment information from the gradient and the decision vector updates to adaptively scale, i.e., smooth, the gradient surrogates. Popular algorithms such as Adam [14] and Nadam [15], available in most machine-learning libraries, incorporate both boosting and smoothing to get better update directions. Algorithms in the serial setting can also use different step-size policies and projections independently of the choices above, which results in the pseudocode representation of these algorithms given in Algorithm 1.

Finally, we recognize that most algorithm variants can be extended to a parallel setting. This is captured in a separate, more involved, execution policy. There are a variety of parallel computing architectures to consider, from shared-memory and distributed-memory environments with multiple CPUs to distributed-memory heterogeneous computing environments which involve both CPUs and GPUs. Some of these environments, such as the shared-memory architectures, offer different choices in how to manage race conditions. For example, some algorithms [7–9,16] choose to use mutexes to *consistently* read from and write to the shared decision vector from different processes, whereas others [17–19] prefer atomic operations to allow for *inconsistent* reads and writes. In addition, algorithms can run on distributed-memory architectures such as the Parameter Server [20], where only parts of the decision vector and data are stored in individual nodes.

The main motivation for implementing `POLO` and `POLO.jl` is to provide researchers with an environment where

1. new proximal gradient methods can be designed and implemented through the powerful policy-based framework,
2. new algorithms can be tested against efficient and standardized implementations of existing algorithms, such as the ones listed in Table 1, and,
3. implemented algorithms can be run on a variety of parallel executors.

3. POLO.jl

`POLO.jl` is a Julia package that aims at providing users with a flexible and easy-to-use environment to prototype and use different algorithms for solving constrained optimization problems.

First, `POLO.jl` wraps the C-API of the well-tested and efficient C++ implementations of many state-of-the-art algorithms and executors covered in `POLO` [5]. `POLO.jl` provides an easy mechanism to fine-tune these algorithms in many different ways, and allows to extend the executors available in `POLO`. It does so by leveraging Julia's efficient C-calling capabilities, passing user-defined loss functions and custom policies in Julia as callback functions to the underlying compiled library. This way, users of `POLO.jl` can use, e.g., the distributed-memory executor without dealing with the system primitives such as handling TCP communications, dynamic joining of workers nodes or managing date serialization, and just focus on the algorithm primitives directly from Julia.

Second, `POLO.jl` attempts to emulate the policy-based design technique [21], which combines multiple-inheritance and template-

```julia
module OptimizationLibrary
# Useful abstract types
abstract type AbstractAlg end
# Useful methods to determine behaviour
function terminator() end
function terminate!() end
function booster() end
function boost!() end
function init!() end
function loss!() end
# What an abstract algorithm does when solving an optimization problem
function solve!(alg::AbstractAlg, loss, x)
  T = eltype(x)
  k, f, g = 1, zero(T), zeros(T, length(x))
  term, boost = terminator(alg), booster(alg)
  init!(term, x)
  init!(boost, x)
  while !terminate!(term, k, f, x, g) # delegate termination to ‘term‘
    f += loss!(loss, x, g) # obtain loss value, save gradient in ‘g‘
    boost!(boost, k, g) # delegate boosting to ‘boost‘
    x .-= g # in practice, the step size is defined by another policy
    k += 1
  end
  return f, x, g
end
# A concrete algorithm
struct GradientDescent <: AbstractAlg end
# A concrete terminator
struct MaxIter{T<:Integer}
  K::T
end
init!(::MaxIter, x) = nothing
terminate!(m::MaxIter, k, f, x, g) = k >= m.K
# A concrete booster
struct NoBoosting end
init!(::NoBoosting, x) = nothing
boost!(::NoBoosting, k, g) = g
# Some defaults for an Algorithm
terminator(::AbstractAlg) = MaxIter(1000)
booster(::AbstractAlg) = NoBoosting()
export GradientDescent, solve!
end
```

Listing 1. Library code that defines an algorithm abstraction. The algorithm abstraction is for a serial implementation of the proximal gradient family of algorithms. For brevity, only the part that boosts the gradient is presented. In general, this family of algorithms have other policies such as the smoothing, step size, proximal and execution policies (Algorithm 1).

programming in C++ to facilitate code-reuse and obtain tight code, using Julia's language capabilities. POLO.jl currently supports prototyping serial and shared-memory parallel versions of the state-of-the-art optimization algorithms in pure Julia[3] by leveraging on the multiple-dispatch mechanism and the experimental multi-threading features.

Next we briefly mention about POLO, our C++ library that uses the policy-based design technique to decompose a family of algorithms into their building blocks, and show how we can follow this design principle using Julia's type system and multiple-dispatch mechanism.

3.1. POLO: a POLicy-based Optimization library

POLO is an open-source, header-only C++ library that builds on standard C++ libraries and lightweight third-party libraries. It provides well-defined primitives for atomic floating-point operations, matrix algebra, logging and dataset reading. It uses ∅MQ and cereal for handling TCP communications and data serialization, which are essential for distributed-memory parallel algorithm implementations on different architectures. More importantly, POLO uses the policy-based design technique and follows recently-proposed parallel-algorithms library design principles to (1) decouple algorithm primitives from system primitives, (2) facilitate code reuse, and (3) generate tight, efficient code.

The policy-based design technique uses template programming and multiple-inheritance to create lightweight shell classes that glue together different policy classes. Template programming allows for writing generic code without knowing the actual types in advance. The specialized code is later generated by the compiler once the user specifies the actual types. Multiple-inheritance, on the other hand, allows for the shell classes to inherit both structure and behaviour from their

[3] As of Julia v1.0.1, we can not conveniently let multiple threads from a C++ library access Julia's memory simultaneously.

```julia
using OptimizationLibrary: AbstractAlg
import OptimizationLibrary: booster, boost!, init!
# Momentum booster
mutable struct Momentum{T<:AbstractFloat}
  mu::T
  eps::T
  nu::Vector{T}
  function (::Type{Momentum})(mu::Real, eps::Real)
    T = float(promote_type(typeof(mu), typeof(eps)))
    new{T}(mu, eps, [])
  end
end
init!(m::Momentum{T}, x) where T = m.nu = zeros(T, length(x))
function boost!(m::Momentum, k, g)
  m.nu .= m.mu .* m.nu .+ m.eps .* g
  g .= m.nu
end
# Custom algorithm --- just change the default booster
struct CustomAlg{T} <: AbstractAlg
  booster::Momentum{T}
end
booster(c::CustomAlg) = c.booster
# loss = ...; x = ...
alg1 = GradientDescent()
result1 = solve!(alg1, loss, x)
alg2 = CustomAlg(Momentum(0.9, 1E-3))
result2 = solve!(alg2, loss, x)
```

Listing 2. User code that uses the example library code and extends its capabilities by (re-)defining only the needed policies.

```c
typedef double value_t; /* double-precision floating points */
typedef int index_t; /* machine-dependent integer numbers */
typedef value_t (*loss_t)(const value_t *x, value_t *g, void *data);
typedef value_t (*inc_loss_t)(const value_t *x, value_t *g, const index_t *ib,
                              const index_t *ie, void *data);
```

Listing 3. C-API loss abstraction; here, `data` is an opaque pointer used to support callback functions from high-level languages in the C library.

```julia
# Full loss
function loss!(loss::AbstractLoss, x, g) end
# Dimension of the decision vector
function nfeatures(loss::AbstractLoss) end
# Incremental loss; optional
function loss!(loss:AbstractLoss, x, g, indices) end
# Number of component functions; needed for incremental losses
function nsamples(loss:AbstractLoss) end
```

Listing 4. Loss abstraction in `POLO.jl`.

policy classes. In short, shell classes determine *what* algorithm logic to execute in a generic way, and delegate *how* to execute the logic to their policies.

For brevity, we skip the details of the policy-based design technique and how it can be applied in optimization algorithm abstractions (readers can refer to [5,21], respectively, for thorough discussions). Instead, we focus on how to emulate this design approach in the Julia language.

3.2. Policy-based design the Julian way

Julia's multiple-dispatch mechanism natively supports template programming. For instance, when the user writes a generic code, `f(x, y) = x + y`, without annotating the types of `x` and `y`, Julia's just-in-time compiler will generate efficient and specialized codes for any given pair of `x` and `y` whose addition is well defined. When doing so, in contrast to traditional object-oriented languages, Julia checks both the number and actual types of *all* the function arguments to determine the most specialized candidate, i.e., the dispatch, `f` to apply.

On the contrary, Julia does not have a traditional inheritance mechanism that supports both structural and behavioural inheritance. Abstract types in Julia are nodes in the type graph that represent sets of different concrete types with similar behaviour, but do not impose any structural similarity. Common behaviour of the super-type can be defined by a function, and this behaviour can later be modified by adding different methods to this function.

Using Julia's type system and multiple-dispatch mechanism, one can effectively emulate the policy-based design technique to facilitate code reuse and generate tight code. To illustrate this, we provide a small example in Listings 1 and 2.

Listing 1 shows an example library that defines an abstract type and a set of predefined, empty functions that are later used in the `solve!` function. The function gets the two policies, i.e., the termination and the boosting policies, and initializes the state of the algorithm in a

```julia
struct FluxLoss{M,D1,D2} <: AbstractLoss
  model::M
  loss::Function
  X::D1
  Y::D2
  function (::Type{FluxLoss})(m, loss, X, Y)
    return new{typeof(m),typeof(X),typeof(Y)}(m, loss, X, Y)
  end
end
function loss!(loss::FluxLoss, x::AbstractVector, g::AbstractVector)
  let curr = 1
    for param in params(loss.model)
      curr = update!(param, x, curr)
    end
  end
  l = loss.loss(loss.X, loss.Y)
  back!(l)
  let curr = 1
    for param in params(loss.model)
      curr = update_gradient!(param, g, curr)
    end
  end
  return l.data
end
```

Listing 5. `Flux.jl` usage in `POLO.jl`.

```julia
imgs = MNIST.images() # load the image dataset
X = hcat(float.(reshape.(imgs, :))...) # stack images into a large batch
labels = MNIST.labels() # get the labels
Y = onehotbatch(labels, 0:9) # one-hot-encode the labels
m = Chain(Dense(28^2, 32, relu), Dense(32, 10), softmax) # build model
loss = FluxLoss(m, (x, y)->crossentropy(m(x), y), X, Y) # build loss
gradient = GradientDescent() # gradient descent algorithm from POLO.jl
gradient(randn(Loss.nfeatures(loss), loss, Utility.MaxIteration(200)))
```

Listing 6. Training a multilayer perceptron for 200 epochs on the MNIST dataset with `POLO.jl` and `Flux.jl`.

```julia
Adam(execution::ExecutionPolicy) =
  ProxGradient(execution, Boosting.Momentum(), Step.Constant(),
                    Smoothing.RMSprop(), Prox.None())
```

Listing 7. Definition of Adam in `POLO.jl`.

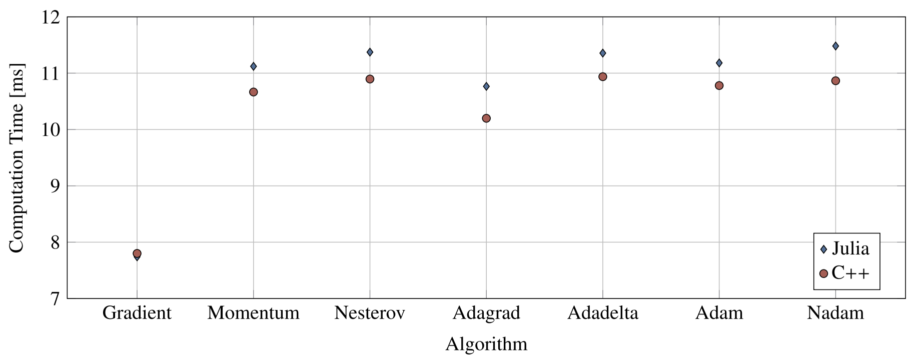

Fig. 1. Comparison benchmark of several gradient algorithms. The difference in computation time required to take 100 algorithms iterations between the C++ implementations and the custom policies in Julia is shown. The experiments were performed on a logistic loss (4) of size 1000 × 100.

generic way. The exact behaviour of the algorithm, such as when the algorithm should terminate or how (if at all) the gradient should be boosted, is delegated to these policies. Last, the library defines a set of sane defaults for the family of the algorithms. If the user chooses to use the exported `GradientDescent` algorithm, which is illustrated in Listing 2, the solve method will run for 1000 iterations using pure gradient values. In Listing 2, the user of the example `OptimizationLibrary` wants to implement a custom algorithm that uses the classical momentum as the boosting policy. To this end, they implement the `Momentum` policy as a class that encapsulates the booster's states, overload the `init!` and `boost!` methods that are used by the library's `solve!` method, and finally create a custom algorithm in the same

```
@with_kw mutable struct BBParameters{T<:AbstractFloat} <: AbstractPolicyParameters
  n0::T = 1.
end

mutable struct BB <: AbstractStep
  params::BBParameters{Float64}
  xprev::Vector{Float64}
  gprev::Vector{Float64}

  function (::Type{BB})(; kwargs...)
    return new(BBParameters(; kwargs...))
  end
end

function initialize!(bb::BB, x0)
  bb.xprev = zeros(length(x0))
  bb.gprev = zeros(length(x0))
end

function stepsize(bb::BB, klocal, kglobal, fval, x, g)
  if kglobal == 1
    bb.xprev .= x
    bb.gprev .= g
    return bb.params.n0
  end
  s = x - bb.xprev
  y = g - bb.gprev
  n = norm(s,2)^2 / cdot(s,y)
  bb.xprev .= x
  bb.gprev .= g
  return n
end
```

Listing 8. Implementing the BB step size in Julia.

```
BB(ex::ExecutionPolicy; n0 = 1.) =
  ProxGradient(ex, Boosting.None(), BB(; n0 = n0), Smoothing.None(), Prox.None())
```

Listing 9. Gradient descent with BB step size. By default, the execution is serial.

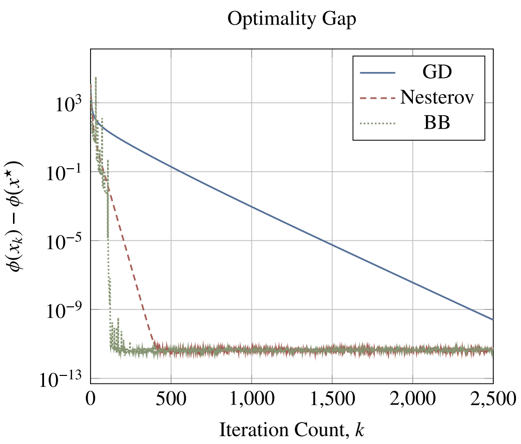

Fig. 2. Performance comparisons of different algorithms on a randomly generated QP problem (3) of size 10000×10000 and with condition number κ: $=L/\mu = 400$.

`AbstractAlg` family whose `booster` points to the `Momentum` booster. When the `solve!` method is called on `alg1` with some `loss` and an initial decision vector x, the sane defaults provided by the library for both the termination check and the boosting policy will be used. However, when the user calls `solve!` on their custom algorithm, the specialized code will be generated by Julia for the custom boosting policy while keeping the default maximum iteration termination check in place.

In fact, this approach is followed in `POLO.jl` to effectively apply the policy-based design technique when prototyping different members of the proximal gradient family of algorithms on different executors. It helps us obtain specialized and tight code while facilitating code reuse.

3.3. Supported executors in POLO.jl

`POLO` provides 4 different `execution` policy classes for proximal gradient algorithms to support 3 major computing platforms:

1. `serial` executor for sequential runs,
2. `consistent` executor, which uses mutexes to lock the whole decision vector for consistent reads and writes, for shared-memory parallelism,
3. `inconsistent` executor, which uses atomic operations to allow for inconsistent reads and writes to individual coordinates of the decision vector, for shared-memory parallelism, and,
4. `paramserver` executor, which is a lightweight implementation of the Parameter Server [20] similar to that discussed in [22], for distributed-memory parallelism.

With the exception of the shared-memory variants, all executors are

```julia
abstract type MultiThread <: ExecutionPolicy end
mutable struct Consistent <: MultiThread
  mutex::Threads.Mutex  # "Inconsistent" executor does not have it
  fval::Float64         # "Inconsistent" executor uses Atomic{Float64}
  x::Vector{Float64}    # "Inconsistent" executor uses Atomic{Float64}
  # other variables
end
function (p::ProxGradient{<:MultiThread})(x0, loss, term, logger)
  initialize!(p, x0)
  Threads.@threads for i = 1:Threads.nthreads()
    kernel!(p, Threads.threadid(), loss, term, logger)
  end
end
function kernel!(p::ProxGradient{<:MultiThread}, wid, loss, term, logger)
  # initialization of local variables for each worker
  while true
    klocal::Int = getk(p.execution) # get the global iteration count
    read!(p, xlocal) # "Consistent" locks the mutex, inside
    flocal = loss!(loss, xlocal, glocal) # evaluate local loss
    if !iterate!(p, wid, klocal, flocal, glocal, term, logger)
      return nothing # worker has terminated
    end
  end
end
function iterate!(p::ProxGradient{<:Consistent}, wid, klocal, flocal,
                  glocal, term, logger)
  exec = p.execution
  lock(exec.mutex)
  res = _iterate!(p, wid, klocal, flocal, glocal, term, logger)
  log!(logger, klocal, flocal, readx(exec), readg(exec))
  unlock(exec.mutex)
  return res
end
function _iterate!(p::ProxGradient{<:MultiThread}, wid, klocal, flocal,
                  glocal, term, logger)
  # obtain global information
  exec = p.execution
  if terminate!(term, k, flocal, readx(exec), glocal)
    return false
  end
  boost!(boosting(p), wid, klocal, k, glocal, g)
  smooth!(smoothing(p), klocal, k, readx(exec), readg(exec), g)
  gamma = step!(step(p), klocal, k, flocal, readx(exec), readg(exec))
  prox!(prox(p), gamma, readx(exec), readg(exec), x)
  increment!(exec)
  return true
end
```

Listing 10. A glimpse of the `consistent` executor.

available in Julia as well through the C-API. For example, `nesterov_s` would execute gradient descent with nesterov boosting on a serial executor. In addition, the executors can also be used in algorithms with custom policies implemented in Julia. Currently, the Julia environment does not support calling back into Julia from another thread on the same machine, which hinders the use of `POLO`'s shared-memory executors. However, Julia has experimental support for shared-memory parallelism in the language itself. We used these constructs to re-implement the consistent and inconsistent shared-memory executors in pure Julia.

4. Julia interface

`POLO` provides a C-API that includes implementations of the algorithms listed in Table 1 on a number of serial, shared-memory and distributed-memory executors. Using the provided C header file and the compiled library, these implemented algorithms are wrapped and available in Julia through `POLO.jl`. Consequently, high-level features of the Julia language can be used to read data and formulate problems that can be solved by the `POLO` algorithms.

The C-API defines a loss abstraction that allows for arbitrary loss function implementations in any high-level language. The function signature of the loss abstraction is given below.

Any given loss function should read *x*, write the (incremental) gradient into *g*, and return the (incremental) loss. The signature also includes an opaque pointer to implementation-specific data. For example, it could point to some loss function object implemented in a high-level language, which calculates the (incremental) loss and

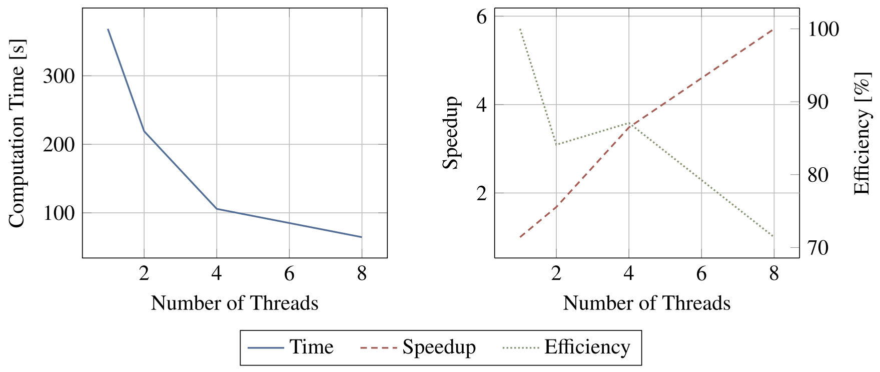

Fig. 3. Benchmark of the consistent multithreaded executor. Computation time required to take 2500 gradient descent iterations (left), and relative speedup and efficiency values (right), as a function of number of worker threads. The experiments were performed on a logistic loss on the `rcv1` [26] dataset.

gradient at x.

Next, the C-API defines algorithm calls for a set of algorithm types and executors as follows.

```
void run_serial_alg(const value_t *xbegin, const value_t *xend,
 loss_t loss_fcn, void *loss_data) {
 serial_alg_t alg;
 alg.initialize(xbegin, xend);
 alg.solve([=](const value_t *xbegin, value_t *gbegin) {
 return loss_fcn(xbegin, gbegin, loss_data);
 });
}
```

The `serial_alg_t` is instantiated to specific algorithms through different policy combinations. For example, the following defines regular gradient descent with a serial executor.

```
using namespace polo;
using namespace polo::algorithm;
using serial_alg_t =
 proxgradient<value_t, index_t, boosting::none, step::constant,
 smoothing::none, prox::none, execution::serial>;
```

High-level languages can then run these algorithms by supplying loss functions that adhere to the abstraction in Listing 3.

`POLO.jl` provides a simple loss abstraction along with a few predefined loss functions. Every loss is a subtype of `AbstractLoss` and should implement the methods given in Listing 4. In brief, the `loss!` method for an `AbstractLoss` takes a decision vector x, writes the gradient vector to g, and returns the associated loss value. Furthermore, an `AbstractLoss` must define the `nfeatures` method, which returns the dimension (d) of the decision vector. To be able to support incremental loss and gradient computations, an `AbstractLoss` must also define the `loss!` method that takes an `indices` vector, which contains the indices of the sampled component functions, as well as the `nsamples` method, which returns the total number (N) of component functions available in the loss object. As long as one adheres to this loss abstraction, any problem of type (1) can be defined and solved in `POLO.jl`. There are two notable predefined loss functions. The first one is the convex quadratic loss, which results in the following unconstrained quadratic program:

$$\underset{x \in \mathbb{R}^d}{\text{minimize}} \ \frac{1}{2} x^\top Q x + q^\top x \, . \qquad (3)$$

`POLO.jl` also supplies a random problem generator for quadratic loss functions, which follows the approach in [23] to ensure that the Q matrix has a desired condition number. The second one is the logistic loss, resulting in the following:

$$\underset{x \in \mathbb{R}^d}{\text{minimize}} \ \sum_{n=1}^{N} \log(1 + \exp(-b_i \langle a_i, x \rangle)) \, , \qquad (4)$$

where the pair $\{a_i, b_i\}$ is the feature vector and the corresponding label, respectively, of each sample in a given dataset. `POLO.jl` includes a datafile reader for the LIBSVM format, which can be used to generate a corresponding logistic loss object directly from one of the many data sets curated by LIBSVM. In addition, there are a vast number of machine learning modules available in the Julia ecosystem that can be used to create loss functions compatible with `POLO.jl`. For example, `Flux.jl` [24] implements powerful abstractions for training neural networks. A `Flux.jl` neural net can be trained using `POLO.jl` by defining the following loss object:

As an example, we can train a multilayer perceptron on the MNIST dataset as given in Listing 6. This example is a small showcase of how easy it is to integrate `POLO.jl` with other packages in the Julia ecosystem.

5. Custom policies in Julia

The C-API also allows to define custom policies within Julia and combine these with the existing policies compiled in `POLO`. This feature is implemented using abstractions with opaque pointers, similar to the loss abstraction in Listing 3 and gives the user extended functionality outside the range of precompiled algorithms in the C-API. `POLO.jl` includes Julia implementations of all `proxgradient` policies available in `POLO`. Through the custom policy API, any algorithm type can be instantiated and tested interactively. In addition, it allows us to implement other high-level abstractions in Julia such as custom

```
# Worker code
using POLO
widx = parse(Int, ARGS[1])
function runme(widx)
  w = PSWorker(timeout = 10000, scheduler = "scheduler")
  A, b = Utility.svmreader("data/rcv1-$(widx)")
  loss = Loss.LogLoss(A, b)
  solve!(w, 3*randn(nfeatures(loss)) .+ 5, loss)
end
runme(widx)
# Master code on "scheduler"
using POLO
using POLO: Bossting.IAG, Step.Constant, Smoothing.None, Prox.L1Ball
N = 697641
L = 0.25 * N
K = parse(Int, get(ARGS, 1, "100"))
lambda = parse(Float64, get(ARGS, 2, "1E-4"))
function runme(K, L, lambda)
  m = PSMaster(timeout = -1, address = "scheduler", scheduler = "scheduler")
  boost = IAG()
  step = Constant(1 / L)
  smooth = None()
  prox = L1Ball(lambda)
  xtrace = Vector{Vector{Float64}}()
  ktrace = Vector{Int}()
  idx = 0
  solve!(m, boost, step, smooth, prox) do k, fval, x, g
    idx += 1
    if idx == 1 || mod(idx, K) == 0
      println("Logging at k = $(k)...")
      push!(ktrace, k)
      push!(xtrace, copy(x))
    end
  end
  return ktrace, xtrace
end
ktrace, xtrace = runme(K, L, lambda)
# Do post-processing, plotting, etc. with x and k history
# Scheduler code on "scheduler"
using POLO
K = parse(Int, ARGS[1])
function runme(K)
  s = PSScheduler(timeout = -1)
  solve!(s, 3*randn(47236) .+ 5, K)
end
runme(K)
```

Listing 11. User code needed to run a distributed-memory parallel optimization algorithm.

termination criteria and advanced logging. As an example, the code exerpt in Listing 7 implements Adam in `POLO.jl`.

To quantify if there is any loss of performance from using the Julia policies, we benchmarked[4] some of the common gradient algorithms. We performed the benchmarks with `BenchmarkTools.jl` using 1000 samples. Each sample consisted of running 100 algorithm iterations of the C++ and Julia implementations on a randomized 1000×100 logistic minimization problem. Median values[5] over the samples are shown in Fig. 1.

[4] All the experiments in the paper were run on a server having 2 Intel Xeon E5-2687W CPUs, 128GB of RAM, and Arch Linux operating system.

[5] We chose to report median values, as opposed to the means, to avoid outliers, which result from (uncontrolled) garbage collection and (just-in-time) compilation times in Julia.

As shown in Fig. 1, there is a slight performance regression in Julia on the more complicated algorithms. Common to these algorithms is that they have memory, which could be the origin of the efficiency gap. However, the performance gap is on the order of one microsecond, and the gain in interactivity and quick prototyping is a strong trade-off. In essence, `POLO.jl` provides a dynamic setting for algorithm design while also enabling the use of the powerful executors implemented in `POLO`.

To demonstrate how new policies can be quickly implemented and tested in `POLO.jl`, we consider the Barzilai-Borwein (BB) step size [25]. This is an adaptive step size of the form

$$\eta_k = \frac{\|s_k\|_2^2}{s_k^\top y_k}, \quad s_k = x_k - x_{k-1}, \quad y_k = \nabla F(x_k) - \nabla F(x_{k-1})$$

which aims to replicate a quasi-Newton method without the burden of

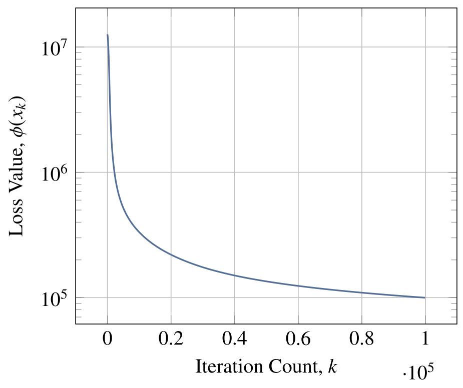

Fig. 4. Loss function value convergence for the ℓ_1-regularized logistic regression problem on the `rcv1` dataset using 5 worker, 1 master and 1 scheduler nodes in the distributed-memory executor of `POLO.jl`. The loss is reduced two orders of magnitude in 10, 0000 iterations, which corresponds to 20, 000 full passes on the dataset.

solving a dense linear system. To implement BB step size in `POLO.jl`, we introduce a new policy of type `AbstractStep` as shown in Listing 8. Since Julia is interpreted, the new policy can be tested immediately without recompiling `POLO`. A gradient descent algorithm that makes use of the BB step size can be defined in Julia as shown in Listing 9.

Next, we can investigate how the performance of the new gradient algorithm compares to existing `POLO` algorithms. We define a custom logger that records the function value each iteration. We solve a randomly generated quadratic program (3) of size 10000 × 10000, with known largest and smallest eigenvalues, $L{:}=\lambda_{\max}(Q) = 20$ and $\mu{:}=\lambda_{\min}(Q) = 1/20$, respectively, using classical gradient descent (GD), gradient descent with Nesterov acceleration, and gradient descent with the BB step size. The constant step size is chosen as $\gamma = 2/(\mu + L)$, which is the best known constant step size for this problem. The results are shown in Fig. 2. The adaptive BB step size converges faster than the constant step-size algorithms, although it does not converge monotonically. The strenght of `POLO.jl` is that the BB step size policy can now be retuned, or even reimplemented, without having to recompile the C+ + library.

6. Parallel execution

6.1. Shared-memory parallelism

The two shared-memory executors provided by `POLO` are not compatible with the Julia runtime. Instead, we used the experimental shared-memory features in Julia to implement similar execution policies. Both mutexes and atomic variables are available. Hence, in spirit of the rest of `POLO.jl`, we mimicked the design of the `consistent` and `inconsistent` executors of `POLO` in a Julian way. A code excerpt that shows a glimpse of the `consistent` executor is given in Listing 10.

There is a slight loss in interactiveness with this implementation as the number of hardware threads available to Julia, governed by the environmental variable `JULIA_NUM_THREADS`, can currently not be changed dynamically after the Julia process has started. Moreover, the threading support is as stated experimental and can be changed or removed in future Julia versions. We benchmarked the `consistent` executor by running 2500 iterations of classical gradient descent on the `rcv1` [26] dataset with a logistic loss, on a varying number of Julia threads. The results are shown in Fig. 3.

Although the threading constructs in Julia, mutex and atomics, are not necessarily as efficient as in C+ +, the `consistent` executor still achieves speedup on this sparse data set. The strong scaling efficiency is above 80% up to four threads, and drops to 70% on eight threads.

This also showcases that `POLO.jl` can be extended to include new executors with little effort. The main motivation behind implementing the shared-memory executors was to compensate for the loss of functionality from `POLO`. However, one could also design other interesting Julia based execution policies. For instance, heterogeneous distributed-memory executors could be designed to use all the available CPU and GPU computation resources of a given computer. Similarly, Julia's `Distributed` module could be used to implement a Julian distributed-memory executor that runs on multiple computers. Nevertheless, obtaining a flexible distributed-memory executor that supports dynamic joining and leaving of the participating nodes as well as serializing data in a portable way needs considerable efforts. As a result, we choose to wrap this functionality provided by `POLO`.

6.2. Distributed-memory parallelism

To demonstrate how easy it is to run an optimization algorithm on a dataset distributed among different worker nodes, we consider solving an ℓ_1-regularized logistic regression problem

$$\underset{x\in\mathbb{R}^d}{\text{minimize}} \sum_{n=1}^{N} log(1 + exp(-b\langle a_i, x\rangle)) + \lambda_1 \|x\|_1^1$$

on the `rcv1` [26] dataset. To this end, we split the dataset into 5 equal chunks and allocate 2 computers for the experiment. One of them has 5 worker Julia processes, each of which loads the corresponding chunk into their local memory, whereas the other one spawns one scheduler process and one master process. Worker nodes have the smooth part of the loss function and calculate the gradient based on their own batches, while the master nodes uses the proximal, incremental aggregated gradient descent algorithm [8] to boost the partial gradients and project the candidate decision vector to the ℓ_1-norm ball.

In Listing 11 we provide the user code necessary to run the

```
Data: Differentiable functions, $f_n(\cdot)$; regulariser, $h(\cdot)$.
Input: Initial decision vector, $x_0$; step size, $\gamma_k$.
Output: Final decision vector, $x_k$.
1 Initialize: $k \leftarrow 0$. while not_done($k, g, x_k, \phi(x_k)$) do
2     $g \leftarrow$ gradient_surrogate($x_k$);                 // partial or full gradient
3     $g \leftarrow$ boosting($k, g$);                           // optional
4     $g \leftarrow$ smoothing($k, g, x_k$);                     // optional
5     $\gamma_k \leftarrow$ step($k, g, x_k, \phi(x_k)$);
6     $x_{k+1} \leftarrow \text{prox}_{\gamma_k h}(x_k - \gamma_k g)$;   // prox step
7     $k \leftarrow k + 1$;
8 end
9 return $x_k$;
```

Algorithm 1. Serial implementation of proximal gradient methods.

experiment. As can be observed, the user can easily assemble an algorithm and run it in a distributed-memory parallel fashion in Julia. Post-processing the traces of the decision vector logged by the master node gives the convergence plot given in Fig. 4.

7. Conclusion and future work

In this paper, we have presented `POLO.jl`, a Julia package that implements state-of-the-art optimization algorithms on different executors and provides necessary building blocks for the users to fine-tune and extend both algorithms and executors in many different ways. On one hand, `POLO.jl` can be used as an interface to our C++ library, `POLO`, to benefit from and extend the compiled algorithms and executors in C++ without much overhead. Serial and distributed-memory parallel algorithms provided by `POLO` can be easily used within Julia through this interface. On the other hand, `POLO.jl` attempts to follow the same design approach in `POLO` to decompose algorithms into their building blocks, and implements shared-memory parallel algorithms using Julia's experimental multi-threading support. In our experiments, we have observed a near-linear speedup, i.e., above 70% efficiency, when running shared-memory parallel algorithms up to 8 worker threads on a set of optimization problems. Both `POLO` and `POLO.jl` are work in progress, and we are currently improving these libraries in several different directions. In particular, we are adding support for more families of algorithms, working on integrating `POLO.jl` tightly with Julia's machine-learning eco-system (e.g., `Flux.jl`) to provide algorithm and parallel executor support to these packages, and building Docker images to support different architectures such as the ARM devices.

Acknowledgements

This research is sponsored in part by the Swedish Research Council under project grant "Scalable optimization: dynamics, architectures and data dependence," and by the Swedish Foundation for Strategic Research under project grant "Societal-scale cyber-physical transport systems." We also thank Ericsson Research for their generosity in letting us use the computing resources at Ericsson Research Data Center for our experiments.

References

[1] Geyer CJ. Markov chain Monte Carlo maximum likelihood. Proceedings of the 23rd symposium on the interface. Computing Science and Statistics; 1991. p. 156–63.

[2] Martino L, Elvira V, Luengo D, Corander J, Louzada F. Orthogonal parallel MCMC methods for sampling and optimization. Digit Signal Process 2016;58:64–84. https://doi.org/10.1016/j.dsp.2016.07.013.

[3] Strobl MAR, Barker D. On simulated annealing phase transitions in phylogeny reconstruction. Mol Phylogenet Evol 2016;101:46–55. https://doi.org/10.1016/j.ympev.2016.05.001.

[4] Ralphs T., Mahajan A., Vigerske S., Mgalati13, LouHafer, Jpfasano, et al. coin-or/symphony: Version 5.6.17. 2019. 10.5281/zenodo.2576603.

[5] Aytekin A., Biel M., Johansson M. POLO: a POLicy-based Optimization library; 2018. 1810.03417v1.

[6] Beck A. First-Order methods in optimization. Society for Industrial and Applied Mathematics (SIAM)978-1-611974-98-0; 2017.

[7] Blatt D, Hero AO, Gauchman H. A convergent incremental gradient method with a constant step size. SIAM J Optim 2007;18(1):29–51. https://doi.org/10.1137/040615961.

[8] Aytekin A., Feyzmahdavian H.R., Johansson M. Analysis and implementation of an asynchronous optimization algorithm for the parameter server; 2016. arXiv: 1610.05507v1.

[9] Defazio A, Bach F, Lacoste-Julien S. SAGA: a fast incremental gradient method with support for non-strongly convex composite objectives. Advances in neural information processing systems 27 (NIPS). Curran Associates, Inc.; 2014. p. 1646–54.

[10] Polyak BT. Some methods of speeding up the convergence of iteration methods. USSR Comput Math MathPhys 1964;4(5):1–17. https://doi.org/10.1016/0041-5553(64)90137-5.

[11] Nesterov YE. A method for solving the convex programming problem with convergence rate $O(1/k^2)$. Soviet Math Doklady 1983;27(2):372–6.

[12] Duchi JC, Hazan E, Singer Y. Adaptive subgradient methods for online learning and stochastic optimization. J Mach Learn Res) 2011;12:2121–59.

[13] Zeiler M.D. ADADELTA: an adaptive learning rate method; 2012. arXiv: 1212.5701v1.

[14] Kingma D.P., Ba J. Adam: a method for stochastic optimization; 2014. arXiv: 1412.6980v9.

[15] Dozat T. Incorporating Nesterov momentum into Adam. 2016. ICLR Workshop

[16] Sra S., Yu A.W., Li M., Smola A.J. AdaDelay: delay adaptive distributed stochastic convex optimization; 2015. arXiv: 1508.05003v1.

[17] Recht B, Ré C, Wright SJ, Niu F. HOGWILD!: a lock-free approach to parallelizing stochastic gradient descent. Advances in Neural Information Processing Systems 24 (NIPS). Curran Associates, Inc.; 2011. p. 693–701.

[18] Leblond R., Pedregosa F., Lacoste-Julien S. Asaga: Asynchronous parallel saga; 2016. arXiv: 1606.04809v1.

[19] Pedregosa F, Leblond R, Lacoste-Julien S. Breaking the nonsmooth barrier: a scalable parallel method for composite optimization. Advances in neural information processing systems 30 (NIPS). Curran Associates, Inc.; 2017. p. 56–65.

[20] Li M, Zhou L, Yang Z, Li A, Xia F, Andersen D, et al. Parameter server for distributed machine learning. Big Learning Workshop, Advances in Neural Information Processing Systems 26 (NIPS). 2013.

[21] Alexandrescu A. Modern C++ design. Addison Wesley0201704315; 2005.

[22] Xiao L., Yu A.W., Lin Q., Chen W. DSCOVR: randomized primal-dual block coordinate algorithms for asynchronous distributed optimization; 2017. arXiv: 1710.05080v1.

[23] Lenard ML, Minkoff M. Randomly generated test problems for positive definite quadratic programming. ACM Trans Math Softw 1984;10(1):86–96. https://doi.org/10.1145/356068.356075.

[24] Innes M. Flux: elegant machine learning with julia. J Open Source Softw 2018. https://doi.org/10.21105/joss.00602.

[25] Tan C, Ma S, Dai Y-H, Qian Y. Barzilai-Borwein step size for stochastic gradient descent. Advances in neural information processing systems 29 (NIPS). Curran Associates, Inc.; 2016. p. 685–93.

[26] Lewis DD, Yang Y, Rose TG, Li F. RCV1: a new benchmark collection for text categorization research. J Mach Learn Res 2004;5:361–97.