POLO: a POLicy-based Optimization library

Arda Aytekin*[1], Martin Biel†[1], and Mikael Johansson‡[1]

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

October 10, 2018

Abstract

We present POLO — a C++ library for large-scale parallel optimization research that emphasizes ease-of-use, flexibility and efficiency in algorithm design. It uses multiple inheritance and template programming to decompose algorithms into essential policies and facilitate code reuse. With its clear separation between algorithm and execution policies, it provides researchers with a simple and powerful platform for prototyping ideas, evaluating them on different parallel computing architectures and hardware platforms, and generating compact and efficient production code. A C-API is included for customization and data loading in high-level languages. POLO enables users to move seamlessly from serial to multi-threaded shared-memory and multi-node distributed-memory executors. We demonstrate how POLO allows users to implement state-of-the-art asynchronous parallel optimization algorithms in just a few lines of code and report experiment results from shared and distributed-memory computing architectures. We provide both POLO and POLO.jl, a wrapper around POLO written in the Julia language, at https://github.com/pologrp under the permissive MIT license.

1 Introduction

Wide adoption of Internet of Things (IoT) enabled devices, as well as developments in communication and data storage technologies, have made the collection, transmission, and storage of bulks of data more accessible than ever. Commercial cloud-computing providers, which traditionally offer their available computing resources at data centers to customers, have also started supporting low-power IoT-enabled devices available at the customers' site in their cloud ecosystem [1]–[3]. As a result, we are experiencing an increase in not only the problem dimensions of data-driven machine-learning applications but also the variety of computing architectures on which these applications are deployed.

There is no silver bullet for solving machine-learning problems. Because the problems evolve continuously, one needs to design new algorithms that are

*Email: aytekin@kth.se, corresponding author.
†Email: mbiel@kth.se.
‡Email: mikaelj@kth.se.

tailored for the specific problem structure, and exploit all available computing resources. However, algorithm design is a non-trivial process. It consists in prototyping new ideas, benchmarking their performance against that of the state-of-the-art, and finally deploying the successful ones in production. Many ideas today are prototyped in high-level languages such as MATLAB, Python and Julia, and it is very rare that researchers implement their algorithms in lower-level languages such as C and C++. Even so, these prototypes are often incomplete, have different abstractions and coding style, and are hand-crafted for specific problem-platform combinations. Ultimately, this results in either performance degradations of high-level implementations in production, or high engineering costs attached to rewriting the low-level implementations from scratch for each different problem-platform combination.

Currently, there exist numerous machine-learning libraries, each targeting a different need. Libraries such as `PyTorch/Caffe2` [4], `Theano` [5] and `TensorFlow` [6] primarily target deep-learning applications, and support different powerful computing architectures. High-level neural-network frameworks such as `Keras` [7] provide user-friendly interfaces to backends such as `Theano` and `TensorFlow`. Even though these libraries implement many mature algorithms for solving optimization problems resulting from deep-learning applications, they fall short when one needs to prototype and benchmark novel solvers [8], [9]. Algorithm designers need to either use the provided communication primitives explicitly [4] or interfere with the computation graph facilities [5], [6] to write the algorithms from scratch. Recent libraries and domain-specific languages such as `Ray` [10] and `Tensor Comprehensions` [9] aim at extending the capabilities of computation graph facilities of aforementioned libraries, and targeting users' custom needs regarding network architectures and data shapes, respectively. More lightweight options such as `mlpack` [8] and `Jensen` [11], on the other hand, aim at providing more generic optimization frameworks in C++. Unfortunately, these options are not centered around algorithm design, either. Although both options provide a good selection of predefined algorithms, these algorithms are implemented for serial [11] or, to a limited extent, single-machine parallel [8] computations. Designers still need to rewrite these implementations for, say, multi-machine parallel computations. In short, we believe that there is a need for a generic optimization platform that facilitates prototyping, benchmarking and deployment of algorithms on different platforms without much performance penalty.

In this paper, we present `POLO` — an open-source, header-only C++ library that uses the policy-based design technique [12]. It consists of two primary layers (Figure 1). The utilities layer builds on standard libraries and lightweight third-party libraries, and offers a set of well-defined primitives for atomic floating-point operations, matrix algebra, communication and serialization, logging and dataset reading. Throughout the development, we have put special emphasis on making the utilities layer portable and efficient. The algorithms layer, on the other hand, builds on the utilities layer and implements different families of algorithms. The library abides by modern C++ design principles [12], and follows ideas from recent parallel algorithms library proposals [13], [14] to

- decouple optimization algorithm building blocks from system primitives,
- facilitate code reuse, and,

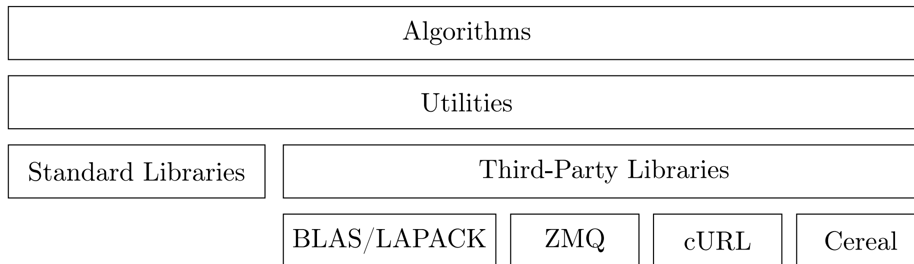

Figure 1: Structure of `POLO`. Utilities layer provides thin wrappers for the essential functionalities provided by the standard and third-party libraries, and implements custom ones, which are then used in the algorithms layer.

- generate tight and efficient code.

In the rest of the paper, we introduce `POLO` in more detail. In Section 2, we motivate the design of `POLO` based on our observations from a family of optimization algorithms. In Section 3, we provide detailed information about the design of the library and the supported computing architectures. In Section 4, we show how some of the state-of-the-art algorithms can be quickly prototyped in `POLO` together with their performance comparisons against each other. Finally, in Section 6, we conclude the paper with further discussions.

2 Motivation

To demonstrate the power of policy-based design for distributed optimization library development, we consider *regularized optimization problems* on the form:

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

Here, $F(\cdot)$ is a differentiable function of x and $h(\cdot)$ is a possibly non-differentiable function. In machine-learning applications, the smooth part of the loss typically represents the empirical data loss, and is a finite sum of loss functions

$$F(x) = \sum_{n=1}^{N} f_n(x) \, ,$$

while the non-smooth part is a regularizer 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 the proximal gradient iteration is

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

where γ_k is a step-size parameter. Thus, the next iterate, x_{k+1}, is selected to be the minimizer 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 [15]. After

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 [29], respectively.

	g			$h(x)$	
Algorithm	`boosting`	`smoothing`	`step`	`prox`	`execution`
SGD	×	×	γ, γ_k	×	`s, cr, ps`
IAG [16]	`aggregated`	×	γ	×	`s, cr, ps`
PIAG [17]	`aggregated`	×	γ	✓	`s, cr, ps`
SAGA [18]	`saga`	×	γ	✓	`s, cr, ps`
Momentum [19]	`classical`	×	γ	×	`s`
Nesterov [20]	`nesterov`	×	γ	×	`s`
AdaGrad [21]	×	`adagrad`	γ	×	`s`
AdaDelta [22]	×	`adadelta`	γ	×	`s`
Adam [23]	`classical`	`rmsprop`	γ	×	`s`
Nadam [24]	`nesterov`	`rmsprop`	γ	×	`s`
AdaDelay [25]	×	×	γ_k	✓	`s, cr, ps`
HOGWILD! [26]	×	×	γ	×	`ir`
ASAGA [27]	`saga`	×	γ	×	`ir`
ProxASAGA [28]	`saga`	×	γ	✓	`ir`

some algebraic manipulations, one can rewrite (2) in terms of the proximal operator [15]

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

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, in the sense that it requires solving an optimization problem at each iteration, the prox-mapping can actually be evaluated very efficiently for several important functions $h(\cdot)$. Together with its strong theoretical convergence guarantees, this makes the proximal gradient method a favorable option in many applications. However, the gradient calculation step in the vanilla 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 more clever query points than the simple gradient step in (2) [16]–[28]. As a result, the proximal gradient family encompasses a large variety of algorithms. We have listed some of the more influential variants in Table 1.

2.1 A Look at Proximal Gradient Methods

A careful review of the serial algorithms in the proximal gradient family reveals that they differ from each other 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) which step-size policy they use; and (5) the type of projection they do in the prox step. For instance, stochastic gradient descent (SGD) algorithms use partial gradient information coming from functions (N) or decision vector coordinates (d) as the gradient surrogate at each iteration, whereas their aggregated versions [16]–[18] accumulate previous partial gradient information to boost the descent direction. Similarly, different momentum-based methods such as the classical [19] and Nesterov's [20] momentum accumulate the full gradient information over iterations. Algorithms such as AdaGrad [21] and AdaDelta [22], 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 [23] and Nadam [24], 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.

Algorithm 1: Serial implementation of proximal gradient methods.

Data: Differentiable functions, $f_n(\cdot)$; regularizer, $h(\cdot)$.
Input: Initial decision vector, x_0; step size, γ_k.
Output: Final decision vector, x_k.

1. $k \leftarrow 0$;
2. $g \leftarrow 0$;
3. **while** `not_done(`k, g, x_k, $\phi(x_k)$`)` **do**
4. $\quad g \leftarrow$ `gradient_surrogate(`x_k`);` `// partial or full gradient`
5. $\quad g \leftarrow$ `boosting(`k, g`);` `// optional`
6. $\quad g \leftarrow$ `smoothing(`k, g, x_k`);` `// optional`
7. $\quad \gamma_k \leftarrow$ `step(`k, g, x_k, $\phi(x_k)$`);`
8. $\quad x_{k+1} \leftarrow \text{prox}_{\gamma_k h}(x_k - \gamma_k g)$; `// prox step`
9. $\quad k \leftarrow k + 1$;
10. **end**
11. **return** x_k;

Most of the serial algorithms in this family either have direct counterparts in the parallel setting or can be extended to have the parallel *execution* support. However, adding parallel execution support brings yet another layer of complexity to algorithm prototyping. First, 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. Second, some of these environments, such as the shared-memory architectures, offer different choices in how to manage race conditions. For example, some algorithms [16]–[18], [25] choose to use mutexes to *consistently* read from and write to the shared decision vector from different processes, whereas others [26]–[28] prefer atomic operations to allow for *inconsistent* reads and writes. Finally, the choice in a specific computing architecture might constrain choices in other algorithm primitives. For instance, if the algorithm is to run on a distributed-memory architecture

such as the Parameter Server [29], where only parts of the decision vector and data are stored in individual nodes, then only updates and projections that embrace data locality can be used.

2.2 Algorithmic perspective

Out of the large number of proximal gradient methods proposed in the literature, only very few have open source low-level implementations, and the existing codes are difficult to extend. This means that algorithm designers need to reimplement others' work to benchmark their ideas even when the ideas differ from each other only in a few aspects. Still, our observations in the previous subsection reveal that algorithm design can be seen as a careful selection of a limited number of compatible policies, see Figure 2.

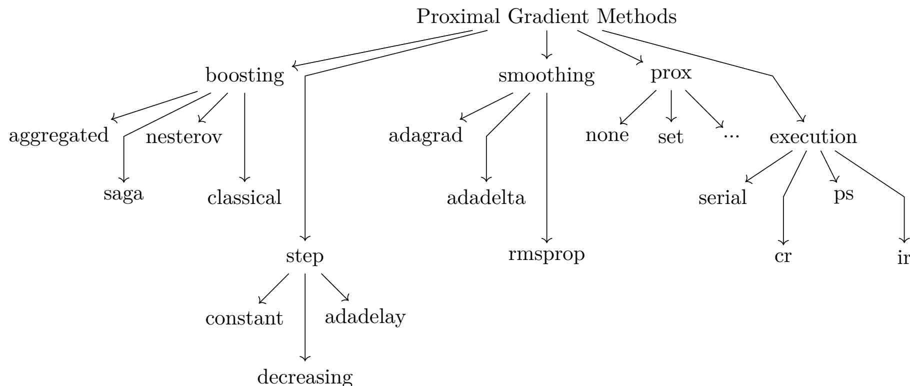

Figure 2: Tree representation of different choices in proximal gradient methods. Both Adam [23] and Nadam [24] use `rmsprop` smoothing, `constant` step and `none` prox policies on `serial` executors. They differ from each other in their respective choices of `classical` and `nesterov` boosting policies. Recent research [30] suggests that using a different smoothing policy, i.e., `adamax`, results in better performance compared to Adam-like algorithms.

`POLO` aims at being an optimization library centered around algorithm design, that facilitates rapid experimentation with new ideas without sacrificing performance. It has a clear separation between algorithm and system primitives to allow for (1) prototyping of new ideas with as few lines of code as possible, and (2) benchmarking resulting algorithms against each other on different platforms as simple as possible. Ultimately, this will lead to faster development of new algorithms as well as easily reproducible results.

2.3 Implementation perspective

Algorithm development is an iterative process of designing and analyzing an algorithm, prototyping and benchmarking it on small-sized test problems, and making modifications until it performs well. Successful designs are then used to solve larger problems. This workflow typically involves implementing and testing the algorithm on different computing environments such as, e.g., multi-core

personal computers, dedicated servers, on-demand cloud-computing nodes, and even a cluster of low-power IoT-enabled devices.

From an implementer's perspective, we would like to have a flexible library that allows us to implement growing number of algorithm primitives on a diversity of computing architectures. For this reason, we aim for building a low-level library with small code-print, high performance and as few dependencies as possible that would also support high-level language integrations and cloud-based solutions.

3 Design

Our observations on different platforms and algorithms in the previous sections reveal a trade-off in algorithm design. Decomposing algorithms into their small, essential policies facilitates behavior isolation and code reuse at the expense of combinatorially increasing design choices (cf. Figure 2). To facilitate code reuse while handling the combinatorial explosion of choices, `POLO` relies on *policy-based design* [12], which combines *multiple inheritance* and *template programming* in a clever way.

To visualize how policy-based design can be used to implement optimization algorithms, let us first assume that we would like to compute functions $y = \bar{f}(\bar{x})$ of the average of a streaming sequence of vectors $\{x_k\}$. In languages that support both multiple inheritance and template programming such as C++, one can define this functionality in terms of its primitives $\bar{f}(\cdot)$ and $ave(\cdot)$ as given in Listing 1.

Listing 1: Simple example that visualizes how policy-based design can be used to support functions operating on the average of a streaming vector sequence. Common methods for construction, initialization and destruction are omitted for brevity.

```
1  /* Template class for f */
2  template <class value_t, template <class> class fbar,
3            template <class> class ave>
4  struct f : fbar<value_t>, ave<value_t> {
5    value_t operator()(const vector<value_t> &x) {
6      vector<value_t> temp(x.size());
7      ave<value_t>::operator()(begin(x), end(x), begin(temp));
8      return fbar<value_t>::operator()(begin(temp), end(temp));
9    }
10 };
11
12 /* Exponential Moving Average */
13 template <class value_t> struct ema {
14   template <class InputIt, class OutputIt>
15   OutputIt operator()(InputIt xb, InputIt xe, OutputIt tb) {
16     size_t idx{0};
17     while (xb != xe) {
18       average[idx] = (1 - alpha) * average[idx] + alpha * *xb++;
19       *tb++ = average[idx++];
20     }
21     return tb;
22   }
23
24   void set_parameters(const value_t alpha) { this->alpha = alpha; }
25 private:
```

```
26    value_t alpha;
27    vector<value_t> average;
28  };
29
30  /* Sum of squared elements of a vector */
31  template <class value_t> struct sumsquared {
32    template <class InputIt>
33    value_t operator()(InputIt xb, InputIt xe) {
34      value_t result{0};
35      while (xb != xe) {
36        result += *xb * *xb;
37        xb++;
38      }
39      return result;
40    }
41  };
```

Above, the library code first defines a template class for the function `f`, which publicly inherits from its generic template parameters `fbar` and `ave`, both of which are templated against the value type (`value_t`) they are working on. The library implements one concrete policy for each function primitive. As can be observed, `f` can work on different precision value-types such as `float`, `double` and `fixed`, and its policy classes can have internal states (`average`) and parameters (`alpha`) that need be persistent and modifiable. Note how `f` is merely a shell that integrates its policy classes and determines *how* they should be applied to achieve the desired transformation while not knowing their internal details.

A user of the library can then mix and match the provided policy classes to achieve the desired functionality. The compiler assembles only the functionality which has been requested and optimizes the code for the specified value types. If the user wants to have a different averaging function such as the cumulative moving average or they want to get the maximum absolute value of the averaged x, which are not originally provided by the library, they can simply implement these policies and use them together with the rest (see Listing 2).

Listing 2: Simple example that visualizes how policy-based design can help users pick and implement different policies in a library.

```
1   /* Use library-provided policies */
2   f<float, sumsquared, ema> f1;
3   f1.set_parameters(0.5);
4   f1.initialize({0,0});
5   f1({3,4}); /* returns 6.25 */
6   f1({6,8}); /* returns 39.0625 */
7
8   /* Implement cumulative moving average */
9   template <class value_t> struct cma {
10    template <class InputIt, class OutputIt>
11    OutputIt operator()(InputIt xb, InputIt xe, OutputIt tb) {
12      size_t idx{0};
13      while (xb != xe) {
14        average[idx] = (N * average[idx] + *xb++) / (N + 1);
15        *tb++ = average[idx++];
16      }
17      N++;
18      return tb;
19    }
20
```

```
21  private:
22    size_t N{0};
23    vector<value_t> average;
24  };
25
26  /* Implement maximum absolute value */
27  template <class value_t> struct maxabs {
28    template <class InputIt>
29    value_t operator()(InputIt xb, InputIt xe) {
30      value_t result{0};
31      while (xb != xe) {
32        value_t current = abs(*xb++);
33        if (current > result)
34          result = current;
35      }
36      return result;
37    }
38  };
39
40  /* Use user-defined policies */
41  f<double, maxabs, cma> f2;
42  f2.initialize({0,0});
43  f2({3,4}); /* cumulative average becomes (3,4), f2 returns 4 */
44  f2({6,8}); /* cumulative average becomes (4.5,6), f2 returns 6 */
```

Based on these simple examples, we immediately realize the strengths of the policy-based design technique. First, the library designer and the user only need to write code for the distinct functionalities, and can then combine them into exponentially many design combinations. Second, the library can support user-defined functionalities, which are not known in advance, and the compiler can optimize the executable code for the selected data types thanks to template programming. Last, the library can also support some enriched functionality that the library designer has not foreseen. For example, the library designer provides a shell `f` that only enforces the call function (`operator()`) on its policies. However, thanks to the inheritance mechanism, the user can use some added functionality (`set_parameters`) when the policy classes provide them. In short, policy-based design helps handle combinatorially increasing design choices with linearly many policies while also allowing for extensions and enriched functionalities in ways library designers might not have predicted.

Optimization algorithms have a lot in common with the examples provided in Listings 1 and 2. Any family of algorithms defines, at each iteration, how a sequence of different transformations should be applied to some shared variable until a stopping condition is met. However, optimization algorithms are much more involved than the examples above. They have state loggers, different samplers and termination conditions, as well as different ways of parallelizing various parts of the code. Moreover, not all the policies of optimization algorithms are compatible with each other. As a result, a similar approach is needed to represent these algorithms in a flexible way while also preventing incompatible policies. To this end, we follow the standard C++ parallel algorithms library proposals [13], [14] in `POLO` to add *execution* support to optimization algorithms, and use *type traits* to disable incompatible policies at compile-time.

In the rest of the section, we will revisit proximal gradient methods to show our design choice, and list the executors provided in `POLO`.

3.1 Revisiting Proximal Gradient Methods

In POLO, we provide the template class proxgradient to represent the family of proximal gradient methods. It consists of boosting, step, smoothing, prox and execution policies, all of which are templated against the real value-type, such as float or double, and the integral index-type, such as int32_t or uint64_t (see Listing 3 for an excerpt).

Listing 3: proxgradient algorithm template implemented in POLO. Its policy templates default to implement the vanilla gradient-descent algorithm running serially without any proximal step.

```
/* namespace polo::algorithm */
template <class value_t = double, class index_t = int,
      template <class, class> class boosting = boosting::none,
      template <class, class> class step = step::constant,
      template <class, class> class smoothing = smoothing::none,
      template <class, class> class prox = prox::none,
      template <class, class> class execution = execution::serial>
struct proxgradient : public boosting<value_t, index_t>,
                      public step<value_t, index_t>,
                      public smoothing<value_t, index_t>,
                      public prox<value_t, index_t>,
                      public execution<value_t, index_t> {
  /* constructors and initializers */

  template <class Loss, class Terminator, class Logger>
  void solve(Loss &&loss, Terminator &&terminator, Logger &&logger)
  {
    execution<value_t, index_t>::solve(this,
      std::forward<Loss>(loss),
      std::forward<Terminator>(terminator),
      std::forward<Logger>(logger)
    );
  }

  /* other member functions */
};
```

As can be observed, proxgradient is simply a shell that glues together its essential policies by using policy-based design, and delegates how to solve the optimization problem to its execution policy. Based on this excerpt only, we can see that any policy class that implements the corresponding solve member function can be an executor for the proxgradient family. For example, one can write a simple serial executor as in Listing 4.

Listing 4: serial execution policy class implemented for the proxgradient family in POLO. Due to space considerations, only the relevant lines are shown.

```
/* namespace polo::execution */
template <class value_t, class index_t> struct serial {
  /* defaulted constructors and assignments */

protected:
  /* initializers and other functions */

  template <class Algorithm, class Loss, class Terminator,
            class Logger>
  void solve(Algorithm *alg, Loss &&loss, Terminator &&terminate,
             Logger &&logger) {
```

```
12      while (!std::forward<Terminator>(terminate)(k, fval, xb_c,
13                                                  xe_c, gb_c)) {
14        fval = std::forward<Loss>(loss)(xb_c, gb);
15        iterate(alg, std::forward<Logger>(logger));
16      }
17    }
18
19  private:
20    template <class Algorithm, class Logger>
21    void iterate(Algorithm *alg, Logger &&logger) {
22      alg->boost(index_t(0), k, k, gb_c, ge_c, gb);
23      alg->smooth(k, k, xb_c, xe_c, gb_c, gb);
24      const value_t step = alg->step(k, k, fval, xb_c, xe_c, gb_c);
25      alg->prox(step, xb_c, xe_c, gb_c, xb);
26      std::forward<Logger>(logger)(k, fval, xb_c, xe_c, gb_c);
27      k++;
28    }
29
30    index_t k{1};
31    value_t fval{0};
32    std::vector<value_t> x, g;
33    value_t *xb; /* points to the first element of x */
34    const value_t *xb_c; /* points to the first element of x */
35    const value_t *xe_c; /* points to past the last element of x */
36    value_t *gb; /* points to the first element of g */
37    const value_t *gb_c; /* points to the first element of g */
38    const value_t *ge_c; /* points to past the last element of g */
39  };
```

The `serial` execution policy implements, between lines 22–25 in Listing 4, the pseudocode given between lines 5–8 in Algorithm 1. In fact, it only keeps track of the shared variables `k`, `fval`, `x` and `g`, and determines the order of executions, while delegating the actual task to the policy classes. In other words, it calls, sequentially in the given order, the `boost`, `smooth`, `step` and `prox` functions, which are inherited in `proxgradient` from `boosting`, `smoothing`, `step` and `prox` policies, respectively, without knowing what they actually do to manipulate x and g. This orthogonal decomposition of algorithms into policies makes it easier for both library writers and users to implement these primitives independently of each other.

Next, let us see how a user of `POLO` can build their algorithms by selecting from predefined policies in Listing 5.

Listing 5: Example code that uses `POLO`'s policy classes to assemble different proximal gradient methods.

```
1  /* include libraries */
2  #include "polo/polo.hpp"
3  using namespace polo;
4  using namespace polo::algorithm;
5
6  /* Assemble Heavyball */
7  proxgradient<double, int, boosting::momentum,
8               step::constant, smoothing::none> heavyball;
9
10 /* Assemble AdaGrad */
11 proxgradient<double, int, boosting::none,
12              step::constant, smoothing::adagrad> adagrad;
13
14 /* Assemble Adam */
15 proxgradient<double, int, boosting::momentum,
```

```
16              step::constant, smoothing::rmsprop> adam;
17
18 /* Assemble Nadam */
19 proxgradient<double, int, boosting::nesterov,
20              step::constant, smoothing::rmsprop> nadam;
```

Here, the user can easily assemble their favorite proximal gradient methods, such as Heavyball [19], AdaGrad [21], Adam [23] and Nadam [24], by choosing among the provided policy classes. Similarly, if the user would like to try a different `smoothing` policy instead of `adagrad` and `rmsprop`, they can define their `custom` smoother (lines 2–41 in Listing 6), and later, use it with other policy classes (lines 44–45). In Listing 6, the `custom` smoother is indeed the `amsgrad` smoother, which improves the convergence properties of Adam-like algorithms [30].

Listing 6: Example code that defines a custom policy class to be used together with provided functionality.

```
1  /* Define a custom smoother */
2  template <class value_t, class index_t> struct custom {
3    custom(const value_t beta = 0.99, const value_t epsilon = 1E-6)
4        : beta{beta}, epsilon{epsilon} {}
5
6    /* defaulted copy/move operations */
7
8    template <class InputIt1, class InputIt2, class OutputIt>
9    OutputIt smooth(const index_t, const index_t, InputIt1 xbegin,
10                   InputIt1 xend, InputIt2 gprev, OutputIt gcurr) {
11     value_t g_val{0};
12     index_t idx{0};
13     while (xbegin != xend) {
14       xbegin++;
15       g_val = *gprev++;
16       nu[idx] = beta * nu[idx] + (1 - beta) * g_val * g_val;
17       nu_hat[idx] = std::max(nu_hat[idx], nu[idx]);
18       *gcurr++ = g_val / (std::sqrt(nu_hat[idx]) + epsilon);
19       idx++;
20     }
21     return gcurr;
22   }
23
24 protected:
25   void parameters(const value_t beta, const value_t epsilon) {
26     this->beta = beta;
27     this->epsilon = epsilon;
28   }
29
30   template <class InputIt> void initialize(InputIt xbegin,
31                                            InputIt xend) {
32     nu = std::vector<value_t>(std::distance(xbegin, xend));
33     nu_hat = std::vector<value_t>(nu);
34   }
35
36   ~custom() = default;
37
38 private:
39   value_t beta{0.99}, epsilon{1E-6};
40   std::vector<value_t> nu, nu_hat;
41 };
42
43 /* Assemble an algorithm using the custom smoother */
44 proxgradient<double, int, boosting::momentum,
```

```
45                  step::constant, custom> myalg;
```

Finally, it is also worth mentioning that, in Listing 5, the unused `prox` and `execution` policies default to `none` and `serial`, respectively, which means that the proximal operator is identity and the algorithm runs sequentially on one CPU. By mixing in different choices of these policies, the user can extend the respective algorithms' capabilities to handle different proximal terms on different, supported executors.

3.2 Provided Executors

`POLO` provides 4 different `execution` policy classes for the `proxgradient` family to support 3 major computing platforms (see Figure 3):

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 [29] similar to that discussed in [31], for distributed-memory parallelism.

When the problem data can fit in a single memory space, users of `POLO` can choose between the provided `serial`, `consistent` and `inconsistent` execution policy classes. As we have partly seen in Listing 4, the `serial` execution simply implements the pseudocode in Algorithm 1, and runs on a single CPU. When multiple CPUs are available to speed-up the computations, users can choose between `consistent` and `inconsistent` execution policies. In both executors, updates to the decision vector are all well-defined. The name `inconsistent` comes from the fact that it uses custom floating-point types that support atomic operations, until these operations are supported officially with C++20, inside the decision vector. Accessing and modifying individual coordinates of the decision vector simultaneously from different worker processes without locking them usually results in faster convergence at the expense of suboptimal results, which can be controlled in certain algorithm-problem pairs, especially when updates are relatively sparse.

`POLO` also provides `paramserver` execution policy to support computations which involve problem data scattered among different computing nodes. Such solutions are needed when the data is either too large to fit on a single node or it has to be kept at a certain place due to storing and accessing requirements. The `paramserver` executor is not a full-fledged Parameter Server [29] implementation but rather a lightweight distributed memory executor similar to that mentioned in DSCOVR [31][1]. It uses `cURL` [32] and `ZMQ` [33] libraries for message passing, and `cereal` [34] for serialization of the transmitted data in a portable way. The `paramserver` executor has 3 main agents (see Figure 4). The `scheduler` is the

[1] At the time of writing this text, the Parameter Server website was not functioning and there was no available code for either the Parameter Server or DSCOVR.

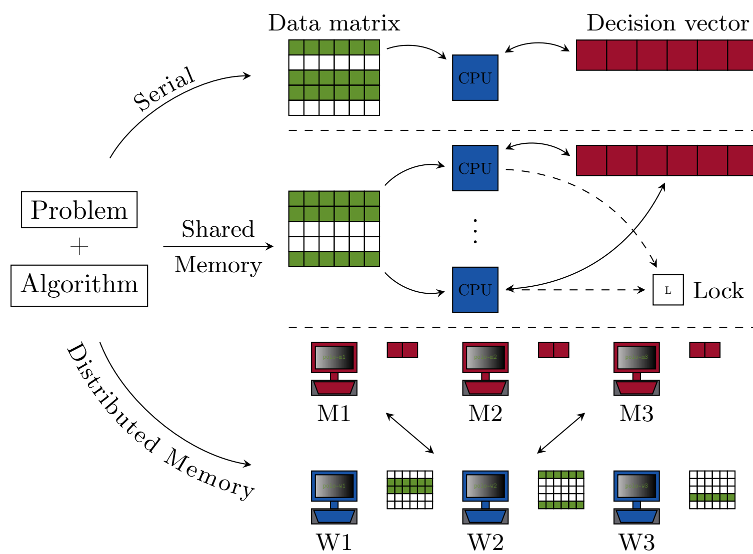

Figure 3: Supported platforms in `POLO` for the `proxgradient` family. When the problem data can fit on a single computer, the user can select among `serial`, `consistent` and `inconsistent` executions. The `consistent` and `inconsistent` executions differ from each other in that the former uses mutexes to lock (L) the whole decision vector when accessing the coordinates whereas the latter uses atomic operations on individual elements in shared-memory platforms. The library also supports distributed-memory executions when the problem data and decision vector are shared among different worker and master machines.

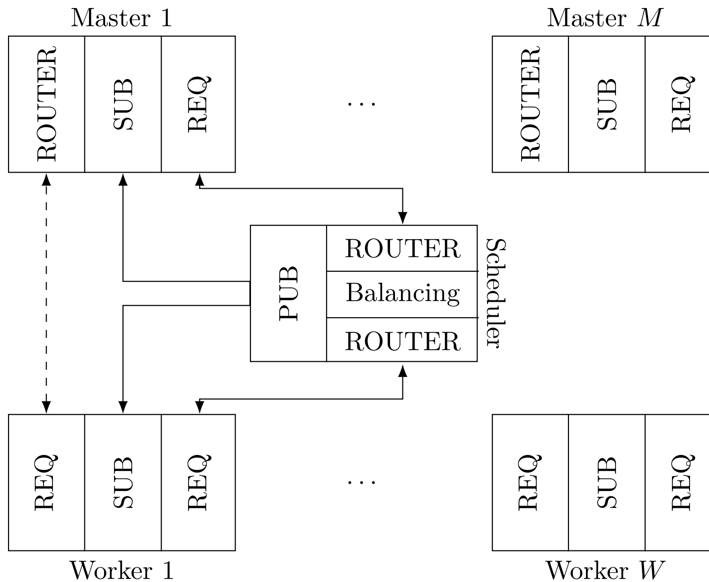

Figure 4: Implementation of Parameter Server in POLO. Arrow directions represent the flow of information. Solid lines represent static connections that have to be established at all times, whereas dashed lines represent dynamic connections that are established when needed.

only static node in the network. It is responsible for bookkeeping of connected `master` nodes, publishing (PUB) global messages to the subscribed (SUB) `master` and `worker` nodes, and directing `worker` nodes to the corresponding `master` nodes when `worker` nodes need to operate on specific parts of the decision vector. The `master` nodes share the full decision vector in a linearly load-balanced fashion. They are responsible for receiving gradient updates from `worker` nodes, taking the `prox` step and sending the updated decision vector to the `worker` nodes when they request (REQ). Finally, `worker` nodes share the smooth loss-function data, and they can join the network dynamically. Based on the sampling they use, `worker` nodes request a list of `master` nodes that keep the corresponding part of the decision vector, and they establish connections with them to communicate decision vectors and (partial) gradient updates.

4 Examples

In this section, we will demonstrate how POLO helps us implement state-of-the-art optimization algorithms easily on different executors To this end, we will first solve a randomly generated [35] unconstrained quadratic problem using different serial algorithms from the proximal gradient family. Then, we will solve a regularized logistic regression problem on the `rcv1` [36] dataset using different algorithms in various ways of parallelism.

We provide a sample code listing (Listing 8) in the appendix for the examples we cover in this section, and state which parts of the listing that need to be changed for each example.

4.1 Unconstrained Quadratic Programming

In POLO, we provide a QP generator that follows the approach described in [35] to create randomly generated QP problems on the form:

$$\begin{aligned} \underset{x \in \mathbb{R}^d}{\text{minimize}} \quad & \left\| \tilde{Q}x - \tilde{q} \right\|_2^2 \\ \text{subject to} \quad & Ax \leq b \,, \end{aligned}$$

where A is an $m \times d$ matrix, b is an m-vector, $\tilde{Q}$ is an $n \times d$ matrix, and $\tilde{q}$ is an n-vector. We use the generator to create an unconstrained QP problem

$$\underset{x \in \mathbb{R}^d}{\text{minimize}} \quad \frac{1}{2} x^\top Q x + q^\top x$$

with $d = 10000$, $\mu = \lambda_{\min}(Q) = 1/20$ and $L = \lambda_{\max}(Q) = 20$. We use vanilla gradient descent, Nesterov momentum and Adam to solve the problem. For this, one needs to change lines 23–25 of Listing 8 to have the following:

```
1  #ifdef GD
2  algorithm::proxgradient<value_t, index_t> alg;
3  alg.step_parameters(2 / (mu + L));
4  alg.initialize(x0);
5  #elif defined NESTEROV
6  algorithm::proxgradient<value_t, index_t, boosting::nesterov> alg;
7  alg.boosting_parameters(0.9, 1 / L);
8  alg.initialize(x0);
9  #else
10 algorithm::proxgradient<value_t, index_t, boosting::momentum,
11                         step::constant, smoothing::rmsprop> alg;
12 alg.step_parameters(0.08);
13 alg.boosting_parameters(0.9, 0.1);
14 alg.smoothing_parameters(0.999, 1E-8);
15 alg.initialize(x0);
16 #endif
```

In the end, we compile the code with appropriate definitions, and run the code until the termination criterion is satisfied. We post-process the iterates generated by the algorithms, and present the optimality gap and iterate convergence in Figure 5.

4.2 Regularized Logistic Regression

In POLO, we also provide various convenience functions such as dataset readers, loss abstractions, samplers and basic matrix algebra functionality that uses the installed BLAS/LAPACK implementations. It is worth noting that POLO is not designed as a replacement for mature linear algebra packages such as Eigen [37] and Armadillo [38], and it can be used in combination with these libraries when working with matrices.

In this subsection, we use the convenience functions to define a regularized logistic loss

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

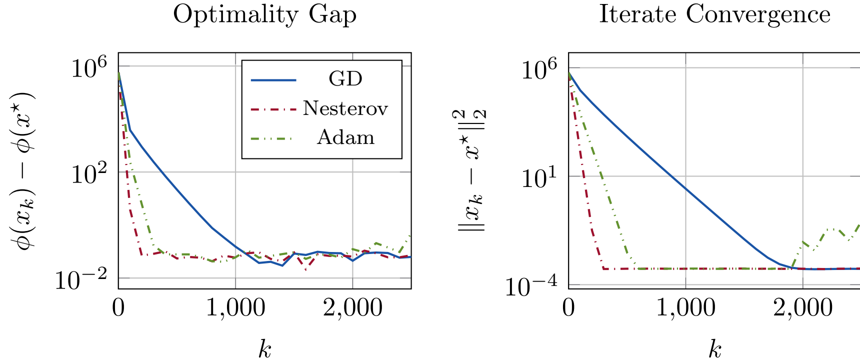

Figure 5: Performance comparisons of different algorithms on the randomly generated QP problem. Adding `boosting` to full gradient iterations increases the rate of convergence, whereas adding `smoothing` results in a slightly slower convergence rate than that of `boosting`-only methods.

where the pair $\{a_i, b_i\}$ is the feature vector and the corresponding label, respectively, of each sample in a given dataset, while λ_1 is the regularization parameter. For this example, we will read the `rcv1` [36] dataset in LIBSVM format. The dataset is sparse with $N = 697641$ samples and $d = 47236$ features.

We first assemble a different serial algorithm, AMSGrad [30], and use mini-batch gradients as the gradient surrogate, where mini-batches are created by sampling the component functions uniformly at random. Necessary changes to the sample code in Listing 8 are as follows:

```
/* Remove auxiliary variables between lines 7-10 in Listing 8 */
/* Replace the loss definition on line 13 in Listing 8 */
const string dsfile = "rcv1"; /* name of the dataset file */
auto dataset = utility::reader<value_t, index_t>::svm({dsfile});
loss::logistic<value_t, index_t> logloss(dataset);

const index_t K = 20000;
const index_t M = 1000; /* mini-batch size */
const index_t N = dataset.nsamples();
const index_t d = dataset.nfeatures();
const value_t L = 0.25 * M; /* L_i = 0.25; rcv1 is normalized */
const index_t B = N / M; /* number of mini-batches */
const value_t lambda1 = 1e-4;
/* Replace the algorithm between lines 23-25 in Listing 8 */
algorithm::proxgradient<value_t, index_t, boosting::momentum,
                        step::constant, smoothing::amsgrad,
                        prox::l1norm> alg;

alg.step_parameters(1. / B); /* tuned for reasonable performance */
alg.boosting_parameters(0.9, 0.1); /* default suggested */
alg.smoothing_parameters(0.999, 1E-8); /* default suggested */
alg.prox_parameters(lambda1);
auto loss = [&](const value_t *x, value_t *g,
                const index_t *ibegin,
                const index_t *iend) -> value_t {
  return logloss.incremental(x, g, ibegin, iend);
};
/* Finally, change the solve method on line 34 in Listing 8 */
utility::sampler::uniform<index_t> sampler;
```

```
30 sampler.parameters(0, N - 1);
31 alg.solve(loss, utility::sampler::component, sampler, M,
32           utility::terminator::maxiter<value_t, index_t>(K),
33           logger);
```

As before, we compile and run the algorithm, this time with different mini-batch sizes M, and then we reconstruct the total function loss values from the traces saved in the `logger`. We report the results in Figure 6.

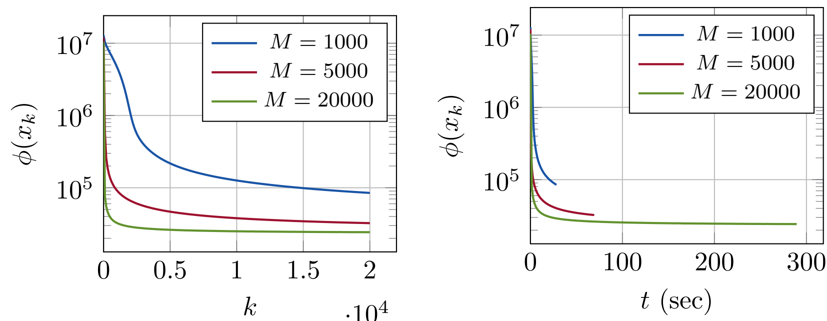

Figure 6: Total function losses for AMSGrad with respect to the iteration count k (left) and wall-clock time t (right) on the `rcv1` dataset. As the number of mini-batches M increases, the convergence rate (in terms of iteration count) increases and the error decreases, as expected. Moreover, the total computation time increases with the increasing number of mini-batches.

Next, we use the distributed memory functionality of `POLO` to solve logistic regression on a parameter server architecture. To demonstrate how `POLO` can be used equally well on embedded systems as on high-performance computers, we first form a cluster of low-power devices. In the cluster, we assign 5 Raspberry PI2 devices as the worker nodes, each of which has parts of the samples in the `rcv1` dataset, and an NVIDIA Jetson TK1 as the single master, which only keeps track of the decision vector and does the ℓ_1-norm ball projection. We assign a laptop as the scheduler, which orchestrates the communication between the static master nodes and the dynamically joining worker nodes in the network. For this example, we choose to use the proximal incremental aggregated gradient (PIAG) [17] method to solve the problem. Necessary modifications are given in the next listing.

```
1  /* Each worker node has their own local dataset */
2  /* Change the line 13 in Listing 8 */
3  #ifdef WORKER
4  const string dsfile = "rcv1";
5  auto dataset = utility::reader<value_t, index_t>::svm({dsfile});
6  loss::logistic<value_t, index_t> logloss(dataset);
7
8  const index_t N = dataset.nsamples();
9  const index_t d = dataset.nfeatures();
10 const value_t L = 0.25 * N;
11 #else
12 /* scheduler and master(s) need not know the loss */
13 auto loss = nullptr;
14 #endif
15 /* Change the lines 23-25 in Listing 8 */
16 const value_t lambda1 = 1e-4;
```

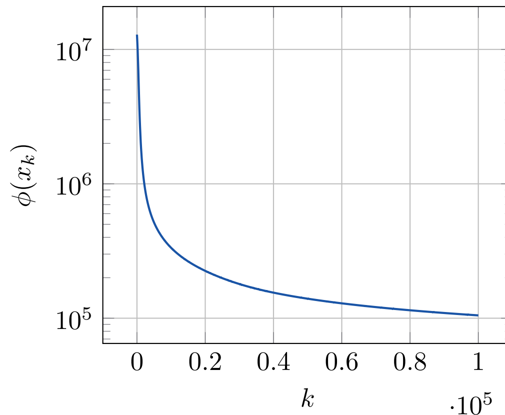

Figure 7: Total function loss on the `rcv1` dataset for PIAG executed on the Parameter Server executor provided in `POLO`.

```
17  const index_t K = 100000;
18
19  algorithm::proxgradient<value_t, index_t, boosting::aggregated,
20                          step::constant, smoothing::none,
21                          prox::l1norm,
22                          execution::paramserver::executor> alg;
23
24  alg.step_parameters(1 / L);
25  alg.prox_parameters(lambda1);
26
27  execution::paramserver::options psopts;
28  /* workers timeout after 10 seconds of inactivity */
29  psopts.worker_timeout(10000);
30  /* master's own IP address; used by the worker */
31  psopts.master(maddress, 50000);
32  /* scheduler's address & ports; needed globally */
33  psopts.scheduler(saddress, 40000, 40001, 40002);
34
35  alg.execution_parameters(psopts);
```

We compile the code three times, one for each of the worker, master and scheduler agents in the distributed executor. We run the algorithm and post-process the iterates logged by the master node. We report the result in Figure 7.

We finally recompiled the same code and ran it on virtual machines in a cloud computing environment at Ericsson Research Data Center. No changes, apart from reconfiguration of IP addresses, were needed to the code. Although the wall-clock time performance improved significantly, the per-iterate convergence is similar to the one for the PI-cluster and therefore not reported here.

5 High-Level Language Integration

Even though `POLO` helps us prototype ideas and implement state-of-the-art optimization algorithms easily in C++, it is often even more convenient to use a high-level language such as MATLAB, Python and Julia to access data from different sources, define and automatically differentiate loss functions, and

visualize results. To be able to support integration with the high-level languages, we also provide a C-API that implements the influential algorithms listed in Table 1 on the executors covered in Section 3.2. Using the provided C header file and the compiled library, users can access these algorithms from their favorite high-level language.

Central to the C-API is 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.

Listing 7: Loss abstraction used in the C-API. `data` is an opaque pointer used to support callback functions from high-level languages in the C library.

```
template <class value_t>
using loss_t = value_t (*)(const value_t *x, value_t *g, void *data);
```

Any given loss should read x, write the gradient of F at x into g, and return $F(x)$. 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 loss and gradient at x. Next, the C-API defines algorithm calls for a set of algorithm types and executors as follows.

```
using value_t = double; /* double-precision floating points */
void run_serial_alg(const value_t *xbegin, const value_t *xend,
                    loss_t<value_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 varied to specific algorithm types 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<double, int, boosting::none, step::constant,
               smoothing::none, prox::none, execution::serial>;
```

High-level languages can then make use of these algorithms by supplying loss functions that adhere to the abstraction in Listing 7.

The C-API also defines a set of custom policies within the `proxgradient` algorithm family. These policies have empty implementations that can be filled in by high-level languages. This feature is implemented using abstractions with opaque pointers, similar to the loss abstraction in Listing 7 and gives the user extended functionality outside the range of precompiled algorithms in the C-API. To illustrate this versatility, we have implemented `POLO.jl` in the Julia language. `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 termination criteria and advanced logging. As an example, the following implements Adam in `POLO.jl`

```
Adam(execution::ExecutionPolicy) =
```

```
2    ProxGradient(execution, Boosting::Momentum(), Step.Constant(),
3                  Smoothing.RMSprop(), Prox.None())
```

In essence, POLO.jl provides a dynamic setting for algorithm design while also enabling the use of the powerful executors implemented in POLO. Due to current limitations in the Julia language, the multi-threaded executors could not yet be applied.

6 Conclusion and Future Work

In this paper, we have presented POLO, an open-source, header-only, C++ template library. Compared to other low-level generic optimization libraries, POLO is focused on algorithm development. With its proper algorithm abstractions, policy-based design approach and convenient utilities, POLO not only offers optimization and machine-learning researchers an environment to prototype their ideas flexibly without losing much from performance but also allows them to test their ideas and other state-of-the-art optimization algorithms on different executors easily. POLO is still a work in progress, and we are working on adding other families of algorithms and different executors to the library as well as wrapping the C-API as convenient packages in high-level languages such as Python and Julia.

Acknowledgment

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.

A Sample Code Listing

Listing 8: Sample code listing to reproduce experiments in Section 4. Only POLO related functionality is shown for brevity.

```
1  /* Include necessary libraries */
2  using index_t = int32_t;
3  using value_t = float;
4
5  int main(int argc, char *argv[]) {
6    /* Define auxiliary variables */
7    const index_t d = 10000;
8    const index_t K = 2500;
9    const value_t L = 20;
10   const value_t mu = 1 / L;
11
12   /* Define a loss function */
13   problem::qp<value_t, index_t> qp(d, L);
14
15   /* Randomly initialize the starting point */
16   mt19937 gen(random_device{}());
```

```
17    normal_distribution<value_t> normal(5, 3);
18    vector<value_t> x0(d);
19    for (auto &val : x0)
20      val = normal(gen);
21
22    /* Select, configure and initialize an algorithm */
23    algorithm::proxgradient<value_t, index_t> alg;
24    alg.step_parameters(2 / (mu + L));
25    alg.initialize(x0);
26
27    /* Use a decision logger */
28    utility::logger::decision<value_t, index_t> logger;
29
30    /* Terminate after K iterations */
31    utility::terminator::maxiter<value_t, index_t> maxiter(K);
32
33    /* Solve the problem */
34    alg.solve(qp, maxiter, logger);
35
36    /* Post-process the logged states, i.e., function values,
37       decision vector, etc.
38    */
39
40    return 0;
41  }
```

References

[1] Amazon Web Services. (Aug. 2018). AWS Greengrass, [Online]. Available: `https://aws.amazon.com/greengrass/`.

[2] Google Cloud. (Aug. 2018). Cloud IoT Edge - Extending Google Cloud's AI & ML, [Online]. Available: `https://cloud.google.com/iot-edge/`.

[3] Microsoft Azure. (Aug. 2018). IoT Hub, [Online]. Available: `https://azure.microsoft.com/en-us/services/iot-hub/`.

[4] PyTorch core team. (Aug. 2018). PyTorch, [Online]. Available: `https://pytorch.org/`.

[5] T. T. D. Team, R. Al-Rfou, G. Alain, *et al.*, "Theano: A python framework for fast computation of mathematical expressions", May 9, 2016. arXiv: `http://arxiv.org/abs/1605.02688v1 [cs.SC]`.

[6] M. Abadi, P. Barham, J. Chen, *et al.*, "Tensorflow: A system for large-scale machine learning", May 27, 2016. arXiv: `http://arxiv.org/abs/1605.08695v2 [cs.DC]`.

[7] F. Chollet *et al.*, *Keras*, `https://keras.io`, 2015.

[8] R. R. Curtin, S. Bhardwaj, M. Edel, *et al.*, "A generic and fast c++ optimization framework", Nov. 17, 2017. arXiv: `http://arxiv.org/abs/1711.06581v1 [cs.MS]`.

[9] N. Vasilache, O. Zinenko, T. Theodoridis, *et al.*, "Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions", Feb. 13, 2018. arXiv: `1802.04730v1 [cs.PL]`.

[10] P. Moritz, R. Nishihara, S. Wang, *et al.*, "Ray: A distributed framework for emerging ai applications", Dec. 16, 2017. arXiv: `1712.05889v1 [cs.DC]`.

[11] R. Iyer, J. T. Halloran, and K. Wei, "Jensen: An easily-extensible c++ toolkit for production-level machine learning and convex optimization", Jul. 17, 2018. arXiv: http://arxiv.org/abs/1807.06574v1 [cs.LG].

[12] A. Alexandrescu, *Modern C++ Design*. Addison Wesley, Mar. 23, 2005, 352 pp., ISBN: 0201704315. [Online]. Available: http://www.ebook.de/de/product/3247239/andrei_alexandrescu_modern_c_design.html.

[13] J. Hoberock, J. Marathe, M. Garland, *et al.*, "A parallel algorithms library | n3724", Open Standards, Tech. Rep., Aug. 30, 2013. [Online]. Available: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3724.pdf.

[14] J. Hoberock, M. Garland, and O. Girioux, "Parallel algorithms need executors | n4406", Open Standards, Tech. Rep., Apr. 10, 2015. [Online]. Available: http://open-std.org/JTC1/SC22/WG21/docs/papers/2015/n4406.pdf.

[15] A. Beck, *First-Order Methods in Optimization (MOS-SIAM Series on Optimization)*. Society for Industrial and Applied Mathematics (SIAM), 2017, ISBN: 978-1-611974-98-0. [Online]. Available: https://www.amazon.com/First-Order-Methods-Optimization-MOS-SIAM/dp/1611974984?SubscriptionId=0JYN1NVW651KCA56C102&tag=techkie-20&linkCode=xm2&camp=2025&creative=165953&creativeASIN=1611974984.

[16] D. Blatt, A. O. Hero, and H. Gauchman, "A convergent incremental gradient method with a constant step size", *SIAM Journal on Optimization*, vol. 18, no. 1, pp. 29–51, Jan. 2007. DOI: 10.1137/040615961.

[17] A. Aytekin, H. R. Feyzmahdavian, and M. Johansson, "Analysis and implementation of an asynchronous optimization algorithm for the parameter server", Oct. 18, 2016. arXiv: 1610.05507v1 [math.OC].

[18] A. Defazio, F. Bach, and S. Lacoste-Julien, "SAGA: A fast incremental gradient method with support for non-strongly convex composite objectives", in *Advances in Neural Information Processing Systems 27 (NIPS)*, Curran Associates, Inc., 2014, pp. 1646–1654. [Online]. Available: http://papers.nips.cc/paper/5258-saga-a-fast-incremental-gradient-method-with-support-for-non-strongly-convex-composite-objectives.pdf.

[19] B. T. Polyak, "Some methods of speeding up the convergence of iteration methods", *USSR Computational Mathematics and Mathematical Physics*, vol. 4, no. 5, pp. 1–17, Jan. 1964. DOI: 10.1016/0041-5553(64)90137-5.

[20] Y. E. Nesterov, "A method for solving the convex programming problem with convergence rate $O(1/k^2)$", *Soviet Mathematics Doklady*, vol. 27, no. 2, pp. 372–376, 1983.

[21] J. C. Duchi, E. Hazan, and Y. Singer, "Adaptive subgradient methods for online learning and stochastic optimization", *Journal of Machine Learning Research (JMLR)*, vol. 12, pp. 2121–2159, 2011, ISSN: 1533-7928. [Online]. Available: http://www.jmlr.org/papers/volume12/duchi11a/duchi11a.pdf.

[22] M. D. Zeiler, "Adadelta: An adaptive learning rate method", Dec. 22, 2012. arXiv: 1212.5701v1 [cs.LG].

[23] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization", Dec. 22, 2014. arXiv: `1412.6980v9 [cs.LG]`.

[24] T. Dozat, "Incorporating nesterov momentum into adam", in *ICLR Workshop*, 2016.

[25] S. Sra, A. W. Yu, M. Li, *et al.*, "Adadelay: Delay adaptive distributed stochastic convex optimization", Aug. 20, 2015. arXiv: `1508.05003v1 [stat.ML]`.

[26] B. Recht, C. Ré, S. J. Wright, *et al.*, "HOGWILD!: A lock-free approach to parallelizing stochastic gradient descent", in *Advances in Neural Information Processing Systems 24 (NIPS)*, Curran Associates, Inc., 2011, pp. 693–701. [Online]. Available: `http://papers.nips.cc/paper/4390-hogwild-a-lock-free-approach-to-parallelizing-stochastic-gradient-descent.pdf`.

[27] R. Leblond, F. Pedregosa, and S. Lacoste-Julien, "Asaga: Asynchronous parallel saga", Jun. 15, 2016. arXiv: `1606.04809v1 [math.OC]`.

[28] F. Pedregosa, R. Leblond, and S. Lacoste-Julien, "Breaking the nonsmooth barrier: A scalable parallel method for composite optimization", *Advances in Neural Information Processing Systems 30 (NIPS 2017)*, Jul. 20, 2017. arXiv: `http://arxiv.org/abs/1707.06468v3 [math.OC]`.

[29] M. Li, L. Zhou, Z. Yang, *et al.*, "Parameter Server for distributed machine learning", in *Big Learning Workshop, Advances in Neural Information Processing Systems 26 (NIPS)*, 2013. [Online]. Available: `http://web.archive.org/web/20160304101521/http://www.biglearn.org/2013/files/papers/biglearning2013_submission_2.pdf`.

[30] S. J. Reddi, S. Kale, and S. Kumar, "On the convergence of adam and beyond", in *International Conference on Learning Representations*, 2018. [Online]. Available: `https://openreview.net/forum?id=ryQu7f-RZ`.

[31] L. Xiao, A. W. Yu, Q. Lin, *et al.*, "Dscovr: Randomized primal-dual block coordinate algorithms for asynchronous distributed optimization", Oct. 13, 2017. arXiv: `1710.05080v1 [math.OC]`.

[32] (Aug. 2018). curl, [Online]. Available: `https://curl.haxx.se/`.

[33] ∅MQ. (2017). Distributed Messaging, [Online]. Available: `http://zeromq.org/`.

[34] USCiLab. (Aug. 2018). cereal, [Online]. Available: `https://uscilab.github.io/cereal/`.

[35] M. L. Lenard and M. Minkoff, "Randomly generated test problems for positive definite quadratic programming", *ACM Transactions on Mathematical Software*, vol. 10, no. 1, pp. 86–96, Jan. 1984. DOI: `10.1145/356068.356075`.

[36] D. D. Lewis, Y. Yang, T. G. Rose, *et al.*, "RCV1: A new benchmark collection for text categorization research", *Journal of Machine Learning Research (JMLR)*, vol. 5, pp. 361–397, 2004, ISSN: 1532-4435. [Online]. Available: `http://www.jmlr.org/papers/volume5/lewis04a/lewis04a.pdf`.

[37] G. Guennebaud, B. Jacob, *et al.*, *Eigen v3*, http://eigen.tuxfamily.org, 2010.

[38] C. Sanderson and R. Curtin, "Armadillo: A template-based c++ library for linear algebra", *The Journal of Open Source Software*, vol. 1, no. 2, p. 26, Jun. 2016. DOI: 10.21105/joss.00026.