TMAC: A Toolbox of Modern Async-Parallel, Coordinate, Splitting, and Stochastic Methods

Brent Edmunds BRENT.EDMUNDS@MATH.UCLA.EDU
Zhimin Peng ZHIMIN.PENG@MATH.UCLA.EDU
Wotao Yin WOTAOYIN@MATH.UCLA.EDU

Department of Mathematics
University of California, Los Angeles
Los Angeles, CA 90095, USA

Abstract

TMAC is a toolbox written in C++11 that implements algorithms based on a set of modern methods for large-scale optimization. It covers a variety of optimization problems, which can be both smooth and nonsmooth, convex and nonconvex, as well as constrained and unconstrained. The algorithms implemented in TMAC, such as the coordinate update method and operator splitting method, are scalable as they decompose a problem into simple subproblems. These algorithms can run in a multi-threaded fashion, either synchronously or asynchronously, to take advantages of all the cores available. TMAC architecture mimics how a scientist writes down an optimization algorithm. Therefore, it is easy for one to obtain a new algorithm by making simple modifications such as adding a new operator and adding a new splitting, while maintaining the multicore parallelism and other features. The package is available at `https://github.com/uclaopt/TMAC`.

Keywords: Asynchronous, Parallel, Operator Splitting, Optimization, Coordinate Update, Stochastic Methods

1. Introduction

TMAC is a toolbox for optimization that implements algorithms based on a set of modern methods for large-scale optimization. The toolbox covers a variety of optimization problems, which can be both smooth and nonsmooth, convex and nonconvex, as well as constrained and unconstrained. TMAC is designed for fast prototyping of scalable algorithms, which can be single-threaded or multi-threaded, and the multi-threaded code can run either synchronously or asynchronously.

Specifically, TMAC implements algorithms based on the following methods:

- **Operator splitting**: a collection of methods that decompose problems into simple subproblems. The original problem often takes the following forms: (i) minimizing $f_1(x) + \cdots + f_n(x)$, (ii) finding a solution x to $0 \in T_1(x) + \cdots + T_n(x)$, and (iii) minimizing $f_1(x_1) + \cdots + f_n(x_n)$ subject to linear constraints $A_1x_1 + \cdots A_mx_m = b$. In addition, any function f_i can compose with a linear operator, e.g. $f(x) = g(Ax)$.

- **Coordinate update**: a collection of methods that find a solution x by updating one, or a few, of its elements each time. The coordinate ordering can follow the random, cyclic, shuffled cyclic, and greedy rules.

- **Parallel** coordinate updates (either synchronous or asynchronous).

These methods are reviewed in Section 1.3 below.

TMAC is not a modeling language, but an algorithm development toolbox. The usage of this toolbox is demonstrated in the following examples: linear system of equations; quadratic programming; empirical risk minimization (e.g. ℓ_1 and ℓ_2 regularized regression); support vector machine; portfolio optimization; and nonnegative matrix factorization. TMAC can use multiple cores efficiently to solve these problems because it exploits their underlying structures.

1.1 Coding and design

TMAC leverages the C++11 standard[1] and object-oriented design, striking for efficiency, portability, and code readability. The package is written in C++ as Matlab does not currently support shared memory programming. The thread library, a new feature of the C++11 standard, provides multithreading that is invariant to operating system. TMAC can be compiled by C++ compilers under Linux, Mac, and Windows. We design the package so that the user can implement a sophisticated operator-splitting, coordinate-update, and sequential or parallel algorithm with little effort. Our codebase is separated into layers: executables, multicore drivers, schemes, operators, and numerical linear algebra[2] that correspond to different algorithmic components. TMAC is used by combining objects from each layer. As a result, our codes are short, clean, and thus easy to read and modify.

1.2 Download and installation

The TMAC package can be accessed from GitHub at `https://github.com/uclaopt/TMAC`. The package runs on Linux, Mac, and Windows operating systems.

1.3 Literature

Operator splitting methods:

These methods solve complicated optimization and monotone inclusion problems by simple subproblems. They started to appear in the 1950s for solving partial differential equations and feasibility problems and were rapidly developed during the 1960s–1980s. Several splitting methods such as Forward-Backward (Passty, 1979), Douglas-Rachford (Douglas and Rachford, 1956) (which is equivalent to ADMM (Gabay and Mercier, 1976; Glowinski and Marroco, 1975)), and Peaceman-Rachford (Peaceman and Rachford, 1955) were introduced.

1. C++ is standardized by ISO (The International Standards Organization) The original C++ standard was issued in 1998. A major update to the standard, C++11, was issued in 2011.
2. For the best performance, BLAS is called for numerical algebra operations (e.g., the product of a matrix and a vector plus another vector). While TMAC can parallelize coordinate updates, it is also possible to parallelize numerical algebra operations by linking TMAC with a parallel BLAS package such as ScaLAPACK(Blackford et al., 1997).

Recently, operator splitting methods such as ADMM and Split Bregman (Goldstein and Osher, 2009) (also see (Wang et al., 2008)) have found new applications in image processing, statistical and machine learning, compressive sensing, and control. New methods such as primal-dual splitting (Condat, 2013; Vũ, 2013), three-operator splitting (Davis and Yin, 2015), and other primal-dual splitting methods (Li et al., 2015; Chen et al., 2016a,b) have appeared, and they are designed to solve more complicated problems.

Coordinate update methods:

As the name suggests, these methods update the selected one, or a few, elements of the variable at each iteration. The original coordinate descent method (Hildreth, 1957; Warga, 1963; Sargent and Sebastian, 1973; Luo and Tseng, 1992) developed in 1950s and analyzed in the 1960s–1990s minimizes the original objective function with respect to the selected coordinates. Later developments such as (Grippo and Sciandrone, 2000; Tseng and Yun, 2009b,a; Xu and Yin, 2013; Bolte et al., 2014) have allowed surrogates for the objective function that are often easier or more efficient to minimize. Lately, coordinate descent has been extended so that each update no longer minimizes a function, but instead the update applies an operator, such as the coordinate projection of an operator or a coordinate-wise fixed-point to an operator (Combettes and Pesquet, 2015; Bianchi et al., 2014; Peng et al., 2015, 2016). Hence, we call it coordinate update instead of coordinate descent.

The initial coordinate selection rule is cyclic selection. It was widely used before other rules such as random (Nesterov, 2012; Richtárik and Takáč, 2014; Lu and Xiao, 2015), shuffled cyclic, greedy (Bertsekas and Bertsekas, 1999; Li and Osher, 2009; Tseng and Yun, 2009b; Peng et al., 2013; Nutini et al., 2015), and parallel (Bradley et al., 2011; Richtárik and Takáč, 2016) started to appear and gain popularity.

Asynchronous parallel methods:

In parallel algorithms, multiple agents attempt to solve a problem. The agents, to solve the problem, must exchange data. An algorithm is synchronous if all the agents must finish computing before they exchange data, and only after the exchange is completed can they start the next computing cycle. Synchronization requires every agent to wait for the slowest agent (or the one solving the most difficult subproblem) to finish computing before communicating. On shared memory architectures[3], the synchronization of communication leads to bus contention. The agents in an asynchronous parallel algorithm, however, can run continuously; they can compute with whatever information they have, even if the latest information from other agents has not arrived; they write their results to the shared memory while other agents are still computing. Async-parallel methods can be traced back to (Chazan and Miranker, 1969) for solving systems of linear equations. For function minimization, (Bertsekas and Tsitsiklis, 1989) introduced an async-parallel gradient projection method. Convergence rates are obtained in (Tseng, 1991).

For fixed-point problems, async-parallel methods date back to (Baudet, 1978). In the pre-2010 methods (Bertsekas, 1983; Bahi et al., 1997; El Baz et al., 1998; Baz et al., 2005) and the review (Frommer and Szyld, 2000), each agent updates its own subset of coordinates.

3. On multicore architectures, agents can be threads or processes. Threads automatically share memory, whereas processes require inter-process communication.

Convergence is established under the *P-contraction* condition and its variants (Bertsekas, 1983). Recently, the works (Nedić et al., 2001; Recht et al., 2011; Liu et al., 2015; Liu and Wright, 2015; Hsieh et al., 2015) introduced async-parallel stochastic methods for function minimization. For fixed-point problems, (Peng et al., 2015) introduced async-parallel stochastic methods (ARock), as well as several applications in optimization.

Software packages:

There exist several packages for solving optimization problems based on splitting and coordinate methods. TOFCS (Becker et al., 2011) is a framework for solving convex cone problems with first-order methods. SCS (O'Donoghue et al., 2016) is a convex cone program solver that applies operator splitting methods to an equivalent feasibility problem. Epsilon (Wytock et al., 2015) is a package for solving general convex programming by using fast linear and proximal operators. Though these packages solve convex optimization problems from various applications, they are sequential and do not take advantage of the multicore systems. PASSCoDe (Hsieh et al., 2015) implements the muticore parallel dual coordinate descent method that solves ℓ_2 regularized empirical risk minimization problems. APPROX(Fercoq and Richtárik, 2015) implements parallel coordinate descent, stochastic dual ascent, and accelerated gradient descent for block seperable objective functions. ACDC(Richtárik and Takáč, 2016) contains a suite of serial, parallel, and distributed coordinate descent algorithms for LASSO, Elastic Net, SVM, and sparse SVM.

2. Case study

To illustrate the usage of TMAC, consider the ℓ_1 regularized logistic regression problem (Ng, 2004) .

$$\underset{x\in\mathbb{R}^n}{\text{minimize}}\ \lambda\|x\|_1 + \sum_{i=1}^{m} \log\left(1 + e^{-b_i\cdot a_i^T x}\right), \tag{1}$$

where $\{(a_i, b_i)\}_{i=1}^m, (a_i \in \mathbb{R}^n, b \in \{1, -1\})$ is the training dataset. For demonstration purposes, we simply set the regularization parameter λ to 1, and the maximum number of epochs to 100, and use TMAC to solve (1) on a machine with 64GB of memory and two Intel Xeon E5-2690 v2 processors (20 cores in total). The following are the `tmac_fbs_l1_log` commands to solve the model on the news20 dataset[4] with 1 thread, 4 threads, and 16 threads.

```
# ------------------- running with 1 thread -----------------------#
$ tmac_fbs_l1_log -data news20.svm -epoch 100 -lambda 1 -nthread 1
[some output skipped]
Computing time  is: 29.53(s).
# ------------------- running with 4 threads ----------------------#
$ tmac_fbs_l1_log -data news20.svm -epoch 100 -lambda 1 -nthread 4
[some output skipped]
Computing time  is: 11.01(s).
```

4. This dataset is from `http://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets`. In this case, $m = 19,996$ and $n = 1,355,191$.

```
# ------------------- running with 16 threads ---------------------#
$ tmac_fbs_l1_log -data news20.svm -epoch 100 -lambda 1 -nthread 16
[some output skipped]
Computing time  is: 3.87(s).
```

The flags `-data, -epoch, -nthread, -lambda` are for the data file, maximum number of epochs, total number of threads, and regularization parameter λ respectively. We can see that the command-line tool is easy to use. Beyond the simplicity, TMAC is also efficient in the sense that the solving time is less than 30 seconds for a problem with more than 1 million variables. We can observe that using 16 threads can achieve approximately 8 times of speedup, reducing the run time to under 4 seconds. Next, we show the major components of the source codes for building `tmac_fbs_l1_log`.

We solve (1) with the forward-backward splitting scheme

$$x^{k+1} = \underbrace{\underbrace{\mathbf{prox}_{\eta\lambda\|\cdot\|_1}}_{\text{backward operator}} \left(\underbrace{x^k - \eta \nabla_x \Big(\sum_{i=1}^{m} \log(1 + e^{-b_i \cdot a_i^T x^k}) \Big)}_{\text{forward operator}} \right)}_{\text{forward-backward splitting scheme}}, \tag{2}$$

where the gradient step of logistic loss and the proximal of ℓ_1 norm correspond to the forward operator[5] and backward operator[6] respectively. Algorithm 1 shows the details of implementing (2) in an asynchronous parallel coordinate update fashion.

Algorithm 1: TMAC for ℓ_1 logistic regression.

Input : A, b and x are shared variables, p agents, $K > 0$.

Initialization:

foward$(x) := x - \eta \nabla_x \sum_{i=1}^{m} \log(1 + e^{-b_i \cdot a_i^T x})$ `// forward operator`

backward$(x) := \mathbf{prox}_{\eta\lambda\|\cdot\|_1}(x)$ `// backward operator`

fbs$(x) :=$ backward(forward(x)) `// foward-backward splitting scheme`

create p computing agents

while *each of the p agents continuously* **do**

  selects $i \in \{1, ..., n\}$ based on some index rule

  updates $x_i \leftarrow x_i - \eta\,(x_i - \text{fbs}_i(x))$

The snippet of code (extracted from `apps/tmac_fbs_l1_log.cc`) shown in Listing 1 implements Algorithm 1 with the TMAC package. Specifically, line 3 defines `forward` as an operator of type `forward_grad_for_log_loss<SpMat>` initialized by the pointers to the data `A`, which is a sparse matrix, and label `b`, which is a dense vector. Line 5 defines a `prox_l1` operator (`backward`) initialized by λ and step size η. Line 7 defines a forward-backward splitting scheme (`fbs`) with the previously defined forward operator, backward operator and the address of the unknown variable `x`. Line 9 calls the multicore driver `TMAC` on the `fbs` scheme and some user specified parameters (`params`).

5. A forward operator computes a (sub)gradient at the current point and takes a negative (sub)gradient step to obtain a new point.
6. A backward operator typically solves an optimization problem, and its optimality condition yields a (sub)gradient taken at the new point.

```
1  // [...] parameters are defined above
2  // forward operator: gradient step for logistic loss
3  forward_grad_for_log_loss<SpMat> forward(&A, &b, &Atx, eta);
4  // backward operator: proximal operator for l1 norm
5  prox_l1 backward(eta, lambda);
6  // forward-backward splitting scheme
7  ForwardBackwardSplitting<forward_grad_for_log_loss<SpMat>, prox_l1>
8    fbs(&x, forward, backward);
9  // the multicore driver
10 TMAC(fbs, params);
```

Listing 1: example code

One can easily adapt the previous code to solve other problems, for example, replacing lines 5 through 7 with the following two lines

```
1  prox_sum_square backward(eta, lambda);
2  ForwardBackwardSplitting<forward_grad_for_log_loss<SpMat>,
     prox_sum_square> fbs(&x, forward, backward);
```

Listing 2: modified line 5-7

solves the Tikhonov regularized logistic regression, i.e.,

$$\underset{x\in\mathbb{R}^n}{\text{minimize}}\, \lambda\|x\|_2^2 + \sum_{i=1}^{m} \log\left(1+\exp(-b_i\cdot a_i^T x)\right).$$

Replacing line 3 and line 7 with the following two lines

```
1  forward_grad_for_square_loss<Matrix> forward(&A, &b, &Atx, eta);
2  ForwardBackwardSplitting<forward_grad_for_square_loss<Matrix>, prox_l1>
     fbs(&x, forward, backward);
```

Listing 3: replaced line 3 and line 7

solves the Lasso problem

$$\underset{x\in\mathbb{R}^n}{\text{minimize}}\, \lambda\|x\|_1 + \frac{1}{2}\|Ax-b\|^2.$$

where `A` is a dense matrix. It is worth mentioning that the previously used operators have implementions in TMAC. Users can refer to the documentation for the complete list of implemented operators.

3. Architecture

Writing an efficient code is very different from writing an optimization algorithm. Our toolbox's architecture is designed to mimic how a scientist writes down an optimization algorithm. The toolbox achieves this by separating into the following layers: Numerical linear algebra, Operator, Scheme, Kernel, and Driver. Each layer represents a different mathematical component of an optimization algorithm.

The following is a brief description of each layer and how it interacts with the layers above and below it.

3.1 Numerical linear algebra

We use Eigen (Guennebaud et al., 2010), Sparse BLAS[7], and BLAS[8] in our Toolbox. Directly using efficient numerical packages like BLAS can be intimidating due to their complex APIs. We provide simplified APIs for common linear algebra operations like $a_i^T x$ in Algorithm 1. This layer insulates the user from the grit of raw numerical implementation. Higher layers use the Numerical Linear Algebra Layer in their implementations. If our provided functions are not sufficient, Eigen, Sparse BLAS, and BLAS are well documented.

3.2 Operator

The Operator Layer contains Forward Operator objects[9] (e.g., gradient descent step, subgradient step) and Backward Operator (e.g., proximal mapping, projection) objects. These operators types see heavy reuse throughtout optimization. For instance, Algorithm 1, along with any other gradient based method for ℓ_1 regularized logistic regression, requires the computation of the Forward Operator $x - \eta \nabla_x \left(\sum_{i=1}^m \log(1 + e^{-b_i \cdot a_i^T x})\right)$. On a similar vein, Nonnegative Matrix Factorization and Nonnegative Least Squares share a backwad operator, the projection onto the positive orthant.

Much as the Numerical Linear Algebra Layer insulates the user from the computation details of numerical linear algebra, the Operator Layer insulates the user from the computational details of operators. This is achieved by encapsulating the computation of common forward and backward operators into Operator objects. Abstracting operators into Operator objects is useful, as Operator objects provide clarity and modularity. Consider the construction of `forward` in Listing 1. At a glance, it is clear that we are using the gradient of logistic regression, the data is sparse, and we are using the Matrix `A` and Vector `b` to compute the gradient. Listing 1 and Listing 3 demonstrates how Operator objects provide modularity. Perturbations to an algorithm, such as a change of regularizer or a change of data fidelity term, can be handled by changing the corresponding operator type. The rest of the code structure is unchanged. The Operator Layer, as a result, allows users to reason and code at the level of operators. The higher layers use Operator objects as components in the creation of algorithms.

As our Toolbox is designed for coordinate update methods, each operator is implemented to compute coordinates efficiently. A coordinate or block of coordinates can be computed efficienty if the computational cost of a single coordinate or a block of coordinates of the operator is reduced by a dimensional factor compared to the evaluation of the entire operator (e.g. $(Ax)_i$ versus Ax). In some cases caching, the storing an intermediate computations, can improve the efficiency of updating a coordinate block. These ideas are formalized in the paper on coordinate friendly stuctures (Peng et al., 2016).

In the case that an operator is needed that we have not provided, a user can use (Peng et al., 2016) as a guide to identify coordinate-friendly structures and implement their own operator. In addition, there are certain rules about operator implementation that must be followed; see Section 4.1.

7. `http://math.nist.gov/spblas/`
8. `http://www.netlib.org/blas/`
9. "Operator object" refers to a C++ object. An "operator" refers to the mathematical object.

3.3 Scheme

A scheme describes how to make a single-iteration update to x. It can be written as a combination of operators. For example, (2) is the Forward-Backward Splitting Scheme (also referred as Proximal Gradient Method) for a specific problem, ℓ_1 regularized logistic regression. In Algorithm 1, it corresponds to Line 9. To apply the Forward-Backward Splitting Scheme to the ℓ_1 regularized logistic regression problem 2 , we need to specify a forward operator and a backward operator (e.g., on Line 5 of Algorithm 1). We can see this in Listing 1. The scheme object `fbs` is specialized to ℓ_1 regularized logistic regression by specifying its type as `ForwardBackwardSplitting<forward_grad_for_log_loss<SpMat>, prox_l1>`.

We provide implemenations of the following schemes: Proximal-Point Method, Gradient Descent, Forward-Backward Splitting, Backward-Forward Splitting, Peaceman-Rachford Splitting, and Douglas-Rachford Splitting.

If the provided schemes are not sufficient, the user may implement their own scheme following certain rules so that their scheme can interact with the rest of the package; see Section 4.1. The user is encouraged to use objects from the Operator Layer as building blocks, but direct calling the Numerical Linear Algebra Layer is perfectly functional.

3.4 Kernel

A Kernel function described the operations that an agent performs. As can be seen in Algorithm 1, it corresponds to the while-loop, which contains contains a coordinate choice rule and a scheme object. The agent chooses a coordinate using its rule and call the Scheme object with the chosen coordinate to update x. For each coordinate choice rule, we have implemented a corresponding Kernel function.

We provide the following coordinate choice rules: cyclic, random, and parallel Gauss-Seidel. Cyclic update rule divides coordinates into approximately equal-sized blocks, and each agent is assigned a block. Each agent chooses coordinates cyclicly from within the block. For random update rule, each agent randomly chooses a block, then chooses coordinates cyclicly from within that block. For parallel Gauss-Seidel rule, each agent updates all of the coordinates in a Gauss-Seidel fashion. If the user desires a different coordinate rule, they may implement their own Kernel function, following the specifications in Section 4.

3.5 Driver

Once the schemes and kernels have been specified, agents are responsible for carrying them out. Agents are generated as C++11 threads. A Driver creates and manages such agents. For example, if the user chooses the cyclic coordinate Kernel and ten agents, the Multicore Driver will create ten agents using that kernel. The Multicore Driver is called with a Params object that contains parameters such as kernel choice, number of iterations, and step size. An example of this is found in Listing 1, where `fbs` and `params` are passed to the `MOTAC` driver.

Optionally, a Multicore Driver can launch a controller agent to control the other agents, for example, by dynamically updating step sizes to accelerate convergence. The current

controller agent monitors convergence by periodically computing the fixed point residual, a measure that would reduce to 0 when the sequence converges to a fixed point.

Most users can treat this layer as a black box. Only when a new way to generate agents is desired, does the user need to modify this layer.

4. Implementation details

4.1 Interaction between layers

The Operator, Scheme, and Kernel Layers interact heavily with each other. To formalize interaction between each layer, we introduce Layer Interfaces. A Layer Interface describes guaranteed member functions of objects in a Layer so that other Layers can safely use these member functions to interact. The Layer Interfaces allow for specialization of objects within each layer while still maintaining a uniform means of interaction. Consider the Operator Interface:

```
class OperatorInterface {
public:
  // compute operator at index
  virtual double operator() (Vector* v, int index = 0)=0;
  // compute operator using val at index
  virtual double operator() (double val, int index = 0)=0;
  // compute full operator using v_in, storing in v_out
  virtual void operator() (Vector* v_in, Vector* v_out)=0;
  // update operator related step size
  virtual void update_step_size(double step_size_)=0;
  // update cache variable following an update at index i
  virtual void update_cache_vars(double old_x_i, double new_x_i, int i)=0;
  // update block of cache variables based upon rank of calling thread
  virtual void update_cache_vars(Vector* x, int rank, int num_threads)=0;
};
```

For an object to belong to the Operator layer, it must inherit from the Operator Interface. Code that contains an object that inherits from the Operator Interface that does not implement the functions in the Operator Interface will not compile.

The Scheme Interface:

```
class SchemeInterface {
public:
  // update scheme internal parameters
  virtual void update_params(Params* params)=0;
  // async: compute and apply coordinate update, return S_{index}
  virtual double operator() (int index)=0;
  // sync: compute and store S_{index} in S_i
  virtual void operator() (int index, double& S_i)=0;
  // sync: apply block of S stored in s to solution variable
```

```
  virtual void update(Vector& s, int range_start, int num_cords)=0;
  // sync: apply coordinate of S stored in s to solution variable
  virtual void update(double s, int idx)=0;
  // sync: update rank worth of cache_vars based on num_threads
  virtual void update_cache_vars(int rank, int num_threads)=0;
};
```

For an object to belong to the Scheme Layer, it must inherit from the Scheme Interface. Code that contains an object that inherits from the Scheme Interface that does not implement the functions in the Scheme Interface will not compile.

Schemes must be suitable for both asynchronous and synchronous computing. Synchronous computing requires all coordinate updates to be computed before applying the coordinate updates. Asynchronous computing can apply coordinates updates immediately. The Scheme Interface reflects these two computing regimes.

The Scheme Interface is very lightweight, as the iterations of optimization methods come in a variety of forms. All it requires is that a coordinate update can be produced and that parameters can be passed to the object.

4.2 Kernel and Multicore Driver interaction

A Multicore Driver creates agents using Kernel functions. The arguments to a Multicore Driver are a scheme object and a params object. A new Kernel function must be callable using information from those two objects. The Multicore Drivers must be modified to include the new kernel function as an option for creating an agent. The modfication requires adding a new case to an if-elseif chain, and knowledge of how to create a C++11 thread.

4.3 Templating

In C++, when objects have similar structures that only vary based upon an input type, templates are used to reduce code redundancy. For instance, the code for an object representing a dense matrix of *doubles* and the code for an object representing a dense matrix of *floats* is identical. Templates are not objects, but instead are blueprints for constructing an object. Based upon the arguments to the template, a corresponding type is automically constructed. We use templating heavily in our Toolbox, as most of our workflow can be genericized.

Objects in the Scheme Layer are implemented as templates. This is a natural choice, as the Scheme objects are generic iteration formulas. For example, in Listing 1, the scheme type of `fbs` is defined by the arguments to the forward backward splitting template, `forward_grad_for_log_loss<SpMat>`, and `prox_l1`.

Objects in the Object Layer are also templatized. Depending on data representation, it can be more efficient to use functionality designed for that data representation. Consider the difference between computing $x - \eta \nabla_x \left(\sum_{i=1}^{m} \log(1 + e^{-b_i \cdot a_i^T x})\right)$ when a_i is stored in the sparse format[10] instead of the dense format. Our linear algebra functions are overloaded so the compiler will deduce the proper function to use in the template.

10. Only the values and locations of nonzero entries are stored.

Kernel functions are also templatized. This allows Kernel functions to take in as input arbitrary objects from the Scheme Layer. If we did not use templating, for each coordinate rule would need a function for every possible realization of gradient descent, proximal point method, etc. Any object that satisfies the Scheme Interface can be passed to a Kernel function.

Multicore Drivers are templatized for the same reason as Kernel functions are templated. If we did not use templating, we would need a Multicore Driver for every possible realization of gradient descent, proximal point method, etc. Any object that satisfies the Scheme Interface can be passed to a Mutlicore Driver.

5. Numerical experiments

In this section, we illustrate the efficiency of TMAC for three applications: ℓ_1 regularized logistic regression, portfolio optimization, and nonnegative matrix factorization. The tests were carried out on a machine with 64GB of memory and two Intel Xeon E5-2690 v2 processors (20 cores in total).

5.1 Minimizing ℓ_1 logistic regression

In this subsection, we apply TMAC to the ℓ_1 regularized logistic regression problem (1). It implements Algorithm 1. The command-line executable is tmac_fbs_l1_log. We set $\lambda = 0.001$, and tests TMAC on two LIBSVM datasets[11]: news20, and url.

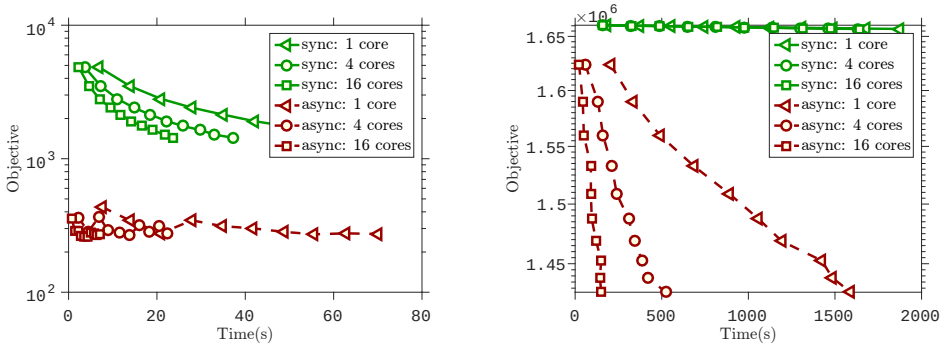
Figure 1 gives the running times of the sync-parallel and async-parallel implementations on the two datasets. Figure 2 is the speedup performance comparison of the two methods. We can observe that async-parallel achieves near-linear speedup, but sync-parallel scales poorly as we shall explain below. One can also see that async-parallel converges faster due to more relaxed forward operator step size selection.

In the sync-parallel implementation, all the running cores have to wait for the last core to finish an iteration, and therefore if a core has a large load, it slows down the iteration. Although every core is (randomly) assigned to roughly the same number of features at each iteration, their a_i's have very different numbers of nonzeros, and the core with the largest number of nonzeros is the slowest (Sparse matrix computation is used for both datasets, which are very large.) As more cores are used, despite that they altogether do more work at each iteration, the per-iteration time reduces as the slowest core tends to be relatively slower.

5.2 Portfolio optimization

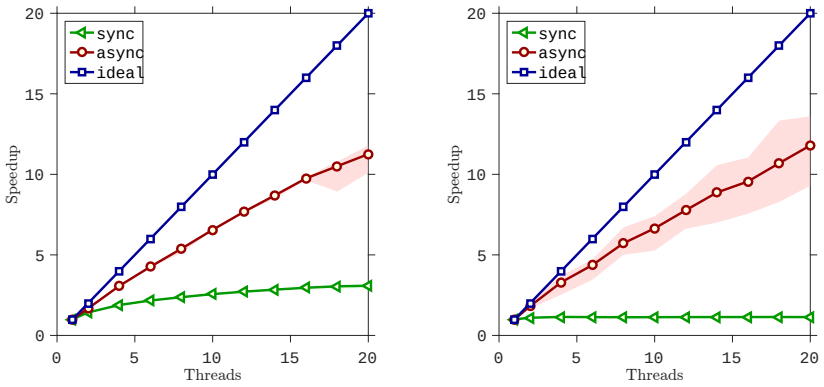
Assume that we have one unit of capital and m assets to invest on. The ith asset has an expected return rate $\xi_i \geq 0$. Our goal is to find a portfolio with the minimal risk such that the expected return is no less than c. This problem can be formulated as

11. http://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/



(a) dataset "news20" (b) dataset "url"

Figure 1: Objective vs wall clock time.



(a) dataset "news20" (b) dataset "url"

Figure 2: Speedup vs number of threads. The solid lines represent mean speedup across 10 different runs. The shaded regions represent the lower and upper bounds of speedup for the 10 runs.

$$\begin{aligned} &\underset{x}{\text{minimize}} \ \frac{1}{2} x^\top Q x, \\ &\text{subject to } x \geq 0, \sum_{i=1}^{m} x_i \leq 1, \sum_{i=1}^{m} \xi_i x_i \geq c, \end{aligned}$$

where the objective function is a measure of risk, and the last constraint imposes that the expected return is at least c. We apply the coordinate update scheme as shown in Section 5.3.1 of (Peng et al., 2016) to solve it. We tested two synthetic problem instances: one has 5,000 assets and the other instance has 30,000 assets. The vector of expected return rate was sampled from $N(0.01, 1)$ normal distribution. The matrix Q was set to $\frac{1}{2}(R + R^T) + \sigma \cdot I$, where the entries of R was sampled independently from $N(0, 0.1)$ normal distribution, and σ was chosen such that Q was positive definite. We tested both sync-parallel and async-parallel methods with 100 epochs. They reached similar final objective. We report the timing and speedup results in Table 1. One can observe that TMAC scales

well for both sync-parallel method and async-parallel method. The nice scaling performance of sync-parallel method is due to almost perfect load balancing across the threads and the homogeneous computing environment.

		async-parallel		sync-parallel	
		time (s)	speedup	time (s)	speedup
5K assets	1 thread	13.95	1.00	13.98	1.00
	4 threads	3.53	3.95	3.54	3.95
	16 threads	0.99	14.09	0.92	15.19
30K assets	1 thread	483.20	1.00	479.22	1.00
	4 threads	124.94	3.86	123.74	3.87
	16 threads	31.96	15.12	33.11	14.47

Table 1: Results for portfolio optimization.

5.3 Nonnegative matrix factorization

We consider the following Nonnegative Matrix Factorization problem

$$\underset{X\geq 0, Y\geq 0}{\text{minimize}} \ \|A - X^T Y\|_2^2,$$

where $A \in \mathbb{R}^{m\times n}$, $X \in \mathbb{R}^{k\times m}$ and $Y \in \mathbb{R}^{k\times m}$. This problem, despite being nonconvex, has a special form. The objective function is block multiconvex[12] and its reguralizers are separable. Problems of this type have been shown (Xu and Yin, 2013; Bolte et al., 2014) to be amenable to coordinate update techniques. Recent work(Davis, 2016) has shown problem of this type to amenable to the asynchronous regime. As the problem is nonconvex, convergence is given to a local minimizer, not a global minimizer.

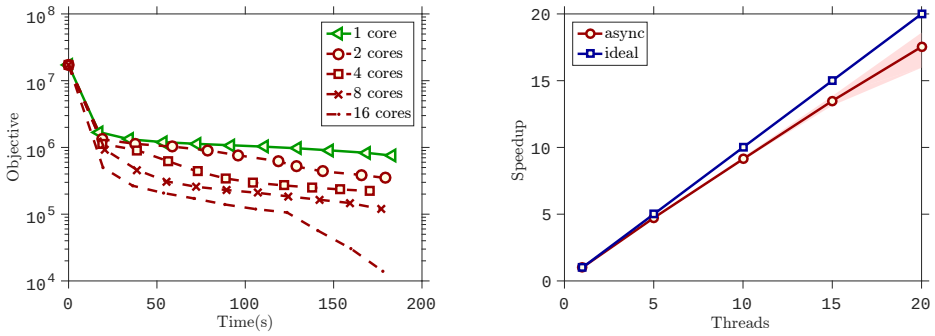
We run TMAC on a synthetic problem, $A = \hat{X}^T\hat{Y}$, with $m = 1000$ and $k = 20$. Elements of $\hat{X}$ and $\hat{Y}$ sampled independently from $N(0, 1)$ normal distribution, then thresholded positive. We ran the tests with variable number of threads and iterations. The following results are the averages resulting from 20 runs.

To show scalability we increased the dimension of k, and tested the speedup.

threads	k=10	k=20	k=100
1	1.00	1.00	1.00
2	1.97	1.98	1.98
4	3.75	3.75	3.76
8	7.12	7.33	7.35
16	13.38	14.51	14.43

Table 2: Speedup results for nonnegative matrix factorization.

12. the objective function is convex when all but a few specific variables are held fixed



(a) Objective vs walll clock time

(b) speedup

Figure 3: NMF results

6. Future Work

New features are still being added to TMAC. The following are some of the features we are exploring.

6.1 Stochastic algorithms

Stochastic algorithms exploit summative structures in problems to produce cheap updates. Our current toolbox does not currently support stochastic algorithms, but can be modified to do so. Such a modification would require stochastic operators. A developement branch will appear on our github.

6.2 Cluster computing

Currently, agents are realized as threads. This limits our toolbox to the multicore regime. Future releases intend to use MPI to bring TMAC to cluster computing. The new functionality will be provided in the Driver and the Kernel Layer.

6.3 User interface

TMAC requires the user to either use our prepackaged executables, or code moderately in C++. A graphic user interace is in development for those who wish avoid interacting directly with C++. This will limit the user to built-in functionality. In addition, interfaces to Matlab and Python will be provided for algorithms of particular interest.

6.4 Automatic parameter choice

TMAC requires the user to choose stepsizes, and number of threads. In future releases we intend to provide automatic stepsize heuristics. Optimizing thread number is a function of current processor usage and computing architecture. We intend to provide functions to survey the current architecture and suggest proper levels of parallelism and asynchrony.

6.5 Block coordinate update

Currently TMAC does not support block coordinate updates (updates consisting of a set of coordinates). Block coordinate updates present a tradeoff between iteration complexity and communication efficiency. Automatic block size deduction and block composition (the coordinates forming a block) is an open question. We plan to explore several heuristics.

6.6 New fields

Splitting methods have been used in many fields (see (Glowinski et al., 2016) for a more in depth discussion). Our current release focuses on optimization, but provides a strong foundation for branching into other fields. Fruitful fields to explore include:

- Numerical simulations;
- Large scale numerical linear algebra;
- Time varying systems such as initial value problems and partial differential equations.

7. Conclusion

We have developed TMAC, an easy-to-use open source toolbox for large scale optimization problems. The toolbox implemented both sequential and parallel algorithms based on operator splitting methods, stochastic methods, and coordinate update methods. TMAC is separated into several layers and mimics how a scientist writes down an optimization algorithm. Therefore, it is easy for one to obtain a new algorithm by modifying just one of the layers such as adding a new operator.

New features and applications will be added to TMAC based upon new research and community input. The software and user guide `https://github.com/uclaopt/TMAC` will be kept up-to-date and supported.

8. Acknowledgements

The work is supported in part by NSF grants ECCS-1462398.

References

Jacques Bahi, Jean-Claude Miellou, and Karim Rhofir. Asynchronous multisplitting methods for nonlinear fixed point problems. *Numerical Algorithms*, 15(3-4):315–345, 1997.

Gérard M Baudet. Asynchronous iterative methods for multiprocessors. *Journal of the ACM (JACM)*, 25(2):226–244, 1978.

Didier El Baz, Andreas Frommer, and Pierre Spiteri. Asynchronous iterations with flexible communication: contracting operators. *Journal of Computational and Applied Mathematics*, 176(1):91–103, 2005. ISSN 0377-0427.

Stephen R Becker, Emmanuel J Candès, and Michael C Grant. Templates for convex cone problems with applications to sparse signal recovery. *Mathematical programming computation*, 3(3):165–218, 2011.

Dimitri Bertsekas and Dimitri Bertsekas. *Nonlinear Programming*. Athena Scientific, September 1999. ISBN 978-1-886529-00-7.

Dimitri P Bertsekas. Distributed asynchronous computation of fixed points. *Mathematical Programming*, 27(1):107–120, 1983.

Dimitri P Bertsekas and John N Tsitsiklis. *Parallel and distributed computation: numerical methods*. Prentice hall Englewood Cliffs, NJ, 1989.

Pascal Bianchi, Walid Hachem, and Iutzeler Franck. A stochastic coordinate descent primal-dual algorithm and applications. In *Machine Learning for Signal Processing (MLSP), 2014 IEEE International Workshop on*, pages 1–6. IEEE, 2014.

L Susan Blackford, Jaeyoung Choi, Andy Cleary, Eduardo D'Azevedo, James Demmel, Inderjit Dhillon, Jack Dongarra, Sven Hammarling, Greg Henry, Antoine Petitet, et al. *ScaLAPACK users' guide*, volume 4. siam, 1997.

Jérôme Bolte, Shoham Sabach, and Marc Teboulle. Proximal alternating linearized minimization for nonconvex and nonsmooth problems. *Mathematical Programming*, 146(1-2): 459–494, August 2014.

Joseph K. Bradley, Aapo Kyrola, Danny Bickson, and Carlos Guestrin. Parallel Coordinate Descent for L1-Regularized Loss Minimization. *arXiv:1105.5379 [cs, math]*, May 2011.

Daniel Chazan and Willard Miranker. Chaotic relaxation. *Linear Algebra and its Applications*, 2(2):199–222, 1969.

Peijun Chen, Jianguo Huang, and Xiaoqun Zhang. A primal-dual fixed point algorithm for minimization of the sum of three convex separable functions. *Fixed Point Theory and Applications*, 2016(1), December 2016a.

Peijun Chen, Jianguo Huang, and Xiaoqun Zhang. A primal-dual fixed point algorithm for multi-block convex minimization. *arXiv:1602.00414 [math]*, February 2016b.

P. Combettes and J. Pesquet. Stochastic Quasi-Fejér Block-Coordinate Fixed Point Iterations with Random Sweeping. *SIAM Journal on Optimization*, 25(2):1221–1248, January 2015.

Laurent Condat. A Primal–Dual Splitting Method for Convex Optimization Involving Lipschitzian, Proximable and Linear Composite Terms. *Journal of Optimization Theory and Applications*, 158(2):460–479, August 2013.

D. Davis. The Asynchronous PALM Algorithm for Nonsmooth Nonconvex Problems. *ArXiv e-prints*, April 2016.

Damek Davis and Wotao Yin. A three-operator splitting scheme and its optimization applications. *UCLA CAM Report 15-13*, 2015.

Jim Douglas and H. H. Rachford. On the numerical solution of heat conduction problems in two and three space variables. *Transactions of the American Mathematical Society*, 82 (2):421–421, February 1956.

Didier El Baz, Didier Gazen, Mohamed Jarraya, Pierre Spiteri, and Jean Claude Miellou. Flexible communication for parallel asynchronous methods with application to a nonlinear optimization problem. *Advances in Parallel Computing*, 12:429–436, 1998.

Olivier Fercoq and Peter Richtárik. Accelerated, parallel and proximal coordinate descent. *SIAM Journal on Optimization*, 25(4):1997–2023, 2015.

Andreas Frommer and Daniel B Szyld. On asynchronous iterations. *Journal of computational and applied mathematics*, 123(1-2):201–216, 2000.

Daniel Gabay and Bertrand Mercier. A dual algorithm for the solution of nonlinear variational problems via finite element approximation. *Computers & Mathematics with Applications*, 2(1):17–40, 1976.

Roland Glowinski and A. Marroco. On the approximation by finite elements of order one, and resolution, penalisation-duality for a class of nonlinear Dirichlet problems. *ESAIM: Mathematical Modelling and Numerical Analysis - Mathematical Modelling and Numerical Analysis*, 9(R2):41–76, 1975.

Roland Glowinski, Tsorng-Whay Pan, and Xue-Cheng Tai. Some facts about operator-splitting and alternating direction methods. *UCLA CAM Report 16-10*, 2016.

Tom Goldstein and Stanley Osher. The split Bregman method for L1-regularized problems. *SIAM Journal on Imaging Sciences*, 2(2):323–343, January 2009.

L. Grippo and M. Sciandrone. On the convergence of the block nonlinear Gauss-Seidel method under convex constraints. *Operations Research Letters*, 26(3):127–136, 2000.

Gaël Guennebaud, Benoît Jacob, et al. Eigen v3. http://eigen.tuxfamily.org, 2010.

Clifford Hildreth. A quadratic programming procedure. *Naval Research Logistics Quarterly*, 4:79–85, 1957.

Cho-jui Hsieh, Hsiang-fu Yu, and Inderjit Dhillon. PASSCoDe: Parallel asynchronous stochastic dual co-ordinate descent. In *Proceedings of the 32nd International Conference on Machine Learning (ICML-15)*, pages 2370–2379, 2015.

Qia Li, Lixin Shen, Yuesheng Xu, and Na Zhang. Multi-step fixed-point proximity algorithms for solving a class of optimization problems arising from image processing. *Advances in Computational Mathematics*, 41(2):387–422, 2015.

Yingying Li and Stanley Osher. Coordinate descent optimization for ℓ^1 minimization with application to compressed sensing; a greedy algorithm. *Inverse Problems and Imaging*, 3 (3):487–503, 2009.

Ji Liu and Stephen J Wright. Asynchronous stochastic coordinate descent: Parallelism and convergence properties. *SIAM Journal on Optimization*, 25(1):351–376, 2015.

Ji Liu, Stephen J Wright, Christopher Ré, Victor Bittorf, and Srikrishna Sridhar. An asynchronous parallel stochastic coordinate descent algorithm. *Journal of Machine Learning Research*, 16:285–322, 2015.

Zhaosong Lu and Lin Xiao. On the complexity analysis of randomized block-coordinate descent methods. *Mathematical Programming. A Publication of the Mathematical Optimization Society*, 152(1-2, Ser. A):615–642, 2015.

Z. Q. Luo and P. Tseng. On the convergence of the coordinate descent method for convex differentiable minimization. *Journal of Optimization Theory and Applications*, 72(1): 7–35, 1992.

A Nedić, Dimitri P Bertsekas, and Vivek S Borkar. Distributed asynchronous incremental subgradient methods. *Studies in Computational Mathematics*, 8:381–407, 2001.

Yu. Nesterov. Efficiency of Coordinate Descent Methods on Huge-Scale Optimization Problems. *SIAM Journal on Optimization*, 22(2):341–362, January 2012.

Andrew Y Ng. Feature selection, l 1 vs. l 2 regularization, and rotational invariance. In *Proceedings of the twenty-first international conference on Machine learning*, page 78. ACM, 2004.

Julie Nutini, Mark Schmidt, Issam H. Laradji, Michael Friedlander, and Hoyt Koepke. Coordinate Descent Converges Faster with the Gauss-Southwell Rule Than Random Selection. *arXiv:1506.00552 [cs, math, stat]*, June 2015.

B. O'Donoghue, E. Chu, N. Parikh, and S. Boyd. Conic optimization via operator splitting and homogeneous self-dual embedding. *Journal of Optimization Theory and Applications*, 2016. URL `http://stanford.edu/~boyd/papers/scs.html`. To appear.

Gregory B. Passty. Ergodic convergence to a zero of the sum of monotone operators in Hilbert space. *Journal of Mathematical Analysis and Applications*, 72(2):383–390, 1979.

D. W. Peaceman and H. H. Rachford. The Numerical Solution of Parabolic and Elliptic Differential Equations. *Journal of the Society for Industrial and Applied Mathematics*, 3 (1):28–41, March 1955.

Zhimin Peng, Ming Yan, and Wotao Yin. Parallel and distributed sparse optimization. In *Signals, Systems and Computers, 2013 Asilomar Conference on*, pages 659–646. IEEE, 2013. ISBN 978-1-4799-2388-5.

Zhimin Peng, Yangyang Xu, Ming Yan, and Wotao Yin. ARock: An algorithmic framework for asynchronous parallel coordinate updates. *arXiv preprint arXiv:1506.02396*, June 2015.

Zhimin Peng, Tianyu Wu, Yangyang Xu, Ming Yan, and Wotao Yin. Coordinate friendly structures, algorithms and applications. *Annals of Mathematical Sciences and Applications*, 1(1):59–119, January 2016.

Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *Advances in Neural Information Processing Systems*, pages 693–701, 2011.

Peter Richtárik and Martin Takáč. Iteration complexity of randomized block-coordinate descent methods for minimizing a composite function. *Mathematical Programming. A Publication of the Mathematical Programming Society*, 144(1-2, Ser. A):1–38, 2014.

Peter Richtárik and Martin Takáč. Parallel coordinate descent methods for big data optimization. *Mathematical Programming*, 156(1):433–484, 2016.

R. W. H. Sargent and D. J. Sebastian. On the convergence of sequential minimization algorithms. *Journal of Optimization Theory and Applications*, 12:567–575, 1973.

P. Tseng and S. Yun. Block-coordinate gradient descent method for linearly constrained nonsmooth separable optimization. *Journal of Optimization Theory and Applications*, 140(3):513–535, 2009a.

Paul Tseng. On the rate of convergence of a partially asynchronous gradient projection algorithm. *SIAM Journal on Optimization*, 1(4):603–619, 1991.

Paul Tseng and Sangwoon Yun. A coordinate gradient descent method for nonsmooth separable minimization. *Mathematical Programming. A Publication of the Mathematical Programming Society*, 117(1-2, Ser. B):387–423, 2009b.

Bằng Công Vũ. A splitting algorithm for dual monotone inclusions involving cocoercive operators. *Advances in Computational Mathematics*, 38(3):667–681, 2013.

Yilun Wang, Junfeng Yang, Wotao Yin, and Yin Zhang. A new alternating minimization algorithm for total variation image reconstruction. *SIAM Journal on Imaging Sciences*, 1(3):248–272, January 2008.

J. Warga. Minimizing certain convex functions. *J. Soc. Indust. Appl. Math.*, 11:588–593, 1963.

Matt Wytock, Po-Wei Wang, and J Zico Kolter. Convex programming with fast proximal and linear operators. *arXiv preprint arXiv:1511.04815*, 2015.

Yangyang Xu and Wotao Yin. A block coordinate descent method for regularized multiconvex optimization with applications to nonnegative tensor factorization and completion. *SIAM Journal on Imaging Sciences*, 6(3):1758–1789, September 2013.