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

Brent Edmunds
Zhimin Peng
Wotao Yin
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 $A_1 x_1 + \cdots + A_m x_m = b$. 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. \( \ell_1 \) and \( \ell_2 \) 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++11 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 compilers under Linux, Mac, and Windows that support C++11. 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, multcore 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 with major improvements in multithreading support, performance, and coding simplifications.

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; Vů, 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, 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 multicore parallel dual coordinate descent method that solves $\ell_2$ regularized empirical risk minimization problems. APPROX(Fercoq and Richtárik, 2015) implements parallel coordinate descent, stochastic dual ascent, and accelerated gradient descent for block superable objective functions. AC-DC(Richtárik and Takáč, 2016) contains a suite of serial, parallel, and distributed coordinate descent algorithms for LASSO, Elastic Net, SVM, and sparse SVM. Note that the list is incomplete.

2. Case study

To illustrate the usage of TMAC, consider the $\ell_1$ regularized logistic regression problem (Ng, 2004).

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

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 $\lambda$ 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 with 1 thread, 4 threads, and 16 threads.

```bash
# ------------------- 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]
```

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

4
Computing time is: 11.01(s).

# ------------------- 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 $\lambda$ 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

$$x^{k+1} = \text{prox}_{\eta\lambda\|\cdot\|_1} \left( x^k - \eta \nabla_x \left( \sum_{i=1}^m \log(1 + e^{-b_i a_i^T x^k}) \right) \right),$$

where the gradient step of logistic loss and the proximal of $\ell_1$ 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 $\ell_1$ logistic regression.

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

**Initialization:**

- $\text{forward}(x) := x - \eta \nabla_x \sum_{i=1}^m \log(1 + e^{-b_i a_i^T x})$  // forward operator
- $\text{backward}(x) := \text{prox}_{\eta\lambda\|\cdot\|_1}(x)$  // backward operator
- $\text{fbs}(x) := \text{backward}(\text{forward}(x))$  // forward-backward splitting scheme

create $p$ computing agents

while each of the $p$ agents continuously do

selects $i \in \{1, \ldots, n\}$ 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 $\lambda$ and step size $\eta$. Line 7 defines a forward-backward splitting scheme (fbs) with the previously defined forward operator, backward

---

\(^5\) 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.
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).

```cpp
// [...] parameters are defined above
// forward operator: gradient step for logistic loss
forward_grad_for_log_loss<SpMat> forward(&A, &b, &Atx, eta);
// backward operator: proximal operator for l1 norm
prox_l1 backward(eta, lambda);
// forward-backward splitting scheme
ForwardBackwardSplitting<forward_grad_for_log_loss<SpMat>, prox_l1> fbs(&x, forward, backward);
// the multicore driver
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

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

Listing 2: modified line 5-7

solves the problem of Tikhonov regularized logistic regression:

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

Replacing line 3 and line 7 with the following two lines

```cpp
forward_grad_for_square_loss<Matrix> forward(&A, &b, &Atx, eta);
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

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

where A is a dense matrix. Therefore, `<Matrix>` is used instead of `<SpMat>`.

It is worth mentioning that the previously used operators have implementations 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\textsuperscript{7}, and BLAS\textsuperscript{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\textsuperscript{9} (e.g., gradient descent step, subgradient step) and Backward Operator (e.g., proximal mapping, projection) objects. These operators types see heavy reuse throughout optimization. For instance, Algorithm 1 along with any other gradient based method for $\ell_1$ regularized logistic regression, requires the computation of the Forward Operator $x \mapsto x - \eta \nabla_x (\sum_{i=1}^{m} \log(1 + e^{-b_i \cdot a_i^T x}))$. On a similar vein, Nonnegative Matrix Factorization and Nonnegative Least Squares share a backward 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 \texttt{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 efficiently 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 structures (Peng et al., 2016).

---

\textsuperscript{7} http://math.nist.gov/spblas/
\textsuperscript{8} http://www.netlib.org/blas/
\textsuperscript{9} “Operator object” refers to a C++ object. An “operator” refers to the mathematical object.
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.

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, $\ell_1$ regularized logistic regression. In Algorithm 1, it corresponds to Line 9. To apply the Forward-Backward Splitting Scheme to the $\ell_1$ 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 $\text{fbs}$ is specialized to $\ell_1$ regularized logistic regression by specifying its type as $\text{ForwardBackwardSplitting<forward\_grad\_for\_log\_loss<SpMat>, prox\_l1>}$.

We provide implementations 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 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:

```cpp
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:

```cpp
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 coordinate 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 modification 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 atomically constructed. We use templating heavily in our Toolbox, as most of our work flow 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 $x - \eta \nabla_x (\sum_{i=1}^{m} \log(1 + e^{-b_i \cdot a_i^T x}))$ 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.

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 Multicore Driver.

5. Numerical experiments

In this section, we illustrate the efficiency of TMAC for three applications: $\ell_1$ 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 $\ell_1$ logistic regression

In this subsection, we apply TMAC to the $\ell_1$ 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.

\(^{10}\) Only the values and locations of nonzero entries are stored.

\(^{11}\) http://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/
5.2 Portfolio optimization

Assume that we have one unit of capital and $m$ assets to invest on. The $i$th 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

$$\minimize_x \frac{1}{2} x^\top Q x,$$

subject to $x \geq 0, \sum_{i=1}^{m} x_i \leq 1, \sum_{i=1}^{m} \xi_i x_i \geq c,$

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 \( \sigma \) 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.

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

Table 1: Results for portfolio optimization.

5.3 Nonnegative matrix factorization

We consider the following Nonnegative Matrix Factorization problem

\[
\text{minimize}_{X \geq 0, Y \geq 0} \| A - X^T Y \|_2^2,
\]

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 regularizers are separable. Problems of this type have been shown (Xu and Yin, 2013, 2014; Bolte et al., 2014) to be amenable to coordinate update techniques and further amenable (Davis, 2016) 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 threshold 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.

6. Future Work

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

---

\(^{12}\) the objective function is convex with respect to each variable when others are fixed
6.1 Stochastic algorithms

Stochastic algorithms exploit the sum structure 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 development 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 interface 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 step sizes, and number of threads. In future releases we intend to provide automatic step-size 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 grant ECCS-1462398 and ONR Grants N000141410683 and N000141210838.

References


