Object-Parallel Infrastructure for Implementing First-Order Methods, with an Example Application to LASSO

Jonathan Eckstein[*] György Mátyásfalvi[†]

April 3, 2015

Abstract

We describe the design of a C++ vector-manipulation substrate that allows first-order optimization algorithms to be expressed in a concise and readable manner, yet still achieve high performance in parallel computing environments. We use standard object-oriented techniques of encapsulation and operator overloading, combined with a novel "symbolic temporaries" delayed-evaluation system that greatly reduces the overhead induced by compiler temporaries and economizes on memory references. We also provide infrastructure to support line-search methods by caching function values and gradients at previously-visited points in a transparent manner that does not "clutter" the principal implementation. We demonstrate the usefulness of our vector-substrate tools by employing them to efficiently solve large-scale LASSO problems using hundreds of processor cores. We reformulate the LASSO problem as a bound-constrained quadratic optimization, and then solve it using the Spectral Projected Gradient (SPG) method implemented through our vector-manipulation substrate.

Acknowledgements. This research was supported in part by National Science Foundation grant CCF-1115638.

The authors acknowledge the Texas Advanced Computing Center (TACC) at The University of Texas at Austin for providing HPC resources that have contributed to the research results reported within this paper. See `http://www.tacc.utexas.edu`.

1 Introduction and Motivation

Efficient implementation of numerical algorithms in parallel computing environments can be challenging. Prevalent parallel implementation tools such as MPI [19], CUDA [17], and OpenMP [5] are linked to particular classes of hardware, are organized around relatively low-level operations, and require significant amounts of code "clutter". The same underlying

[*]Department of Management Science and Information Systems (MSIS) and RUTCOR, Rutgers University, 100 Rockafeller Road, Piscataway, NJ 08854

[†]Doctoral program in Operations Research, Rutgers University, 100 Rockafeller Road, Piscataway, NJ 08854

algorithm may have to be re-implemented multiple times to adapt to different hardware environments or applications, and the resulting code may be difficult to read. There is at present no expressive, portable high-level language that allows implementers to exploit most of the performance of the available hardware, something that would do for parallel computing what the FORTRAN and C languages accomplished for serial computing in the 1960's through 1980's.

In the absence of such a language breakthrough, a natural route to more elegant and portable implementation of parallel algorithms is to use established object-oriented programming concepts. This paper describes an effort to use standard C++ techniques to create an environment for efficient parallel implementation of a particular class of numerical algorithms. Specifically, we target first-order algorithms for continuous optimization, although our basic infrastructure could be used for related mathematical problem settings such as complementarity or systems of equations. By first-order algorithms, we mean methods whose basic operations consist of function evaluations, gradient evaluations, vector scaling, vector addition, and inner products. For example, the classical conjugate gradient method falls into this category. We have created a C++ environment in which such algorithms can be expressed about as concisely and readably as in prototyping/scripting environments such as MATLAB or NumPy [22], and acts as a kind of template (although not literally a C++ template) that can be applied to varying vector representations on varying hardware, serial or parallel. Fundamentally, this effort is an application of the standard techniques of encapsulation and operator overloading, but making its performance comparable to what can be achieved with less layered coding methods required careful attention to the creation of compiler temporaries. Thus, we developed an apparently novel "symbolic temporaries" delayed-evaluation infrastructure to minimize both the time and memory-reference impact of operator overloading, as described in Section 2.2 below.

So far, we have used our substrate to implement two algorithms, the CG-Descent unconstrained optimization method of Hager and Zhang [9, 10] and the spectral projected gradient (SPG) method for problems with simple constraints [2, 3]. To support the line search required by the former method, which resembles that of many other unconstrained and box-constrained optimization algorithms, we built infrastructure that automates the caching of the function and gradient values at already-visited points, avoiding multiple evaluations at the same point without cluttering the basic expression of the algorithm.

Our long-term goal is to expand the implementation of Hager and Zhang's algorithm to the "ASA" version of CG-Descent [11], which is able to handle box constraints, and then use this algorithm as the subproblem solver in an implementation of the approximate augmented Lagrangian method of [6]. Implemented using our vector substrate, the result would be a general-purpose nonlinear optimization solver framework for parallel environments, capable of handling general nonlinear constraints. However, to demonstrate the utility of our underlying tools before completing this substantial development effort, we here show the utility of our underlying tools by using them to solve some large-scale problems having only box constraints. In particular we solve some large-scale LASSO problems — see for example [21] — by reformulating them as quadratic optimization problems over the nonnegative orthant, and then solve them using our SPG method, obtaining good scalability results.

The remainder of this paper is organized as follows: Section 2 describes the basic organization of our abstract vector manipulation substrate, with Section 2.1 covering the funda-

mentals and Section 2.2 describing the symbolic-temporaries technique that we use to obtain acceptable efficiency. Section 3 describes the elements of our infrastructure that are specific to optimization problems, including the `AbstractProblem` class for encapsulating problem instances and their data representations (covered in Section 3.1) and our tools for supporting efficient line search (in Section 3.2). Section 4 reproduces the actual code of our SPG implementation, to illustrate its relative simplicity and clarity. Finally, Section 5 describes the application of our tools to large-scale LASSO problems.

2 Abstract Vector Substrate for First-Order Methods

2.1 Abstract Vector Types and Encapsulation

Our basic goal was to use the operator overloading capabilities of the C++ language [20] to be able to express simple algorithmic manipulations of vectors in a concise, readable manner, portable without recoding between different applications, data representations, and hardware platforms. To this end, we created a class called `AbstractVector`. This class creates a general interface that abstracts both the representation of vectors and the methods used to perform simple mathematical manipulations such as addition and scaling. Examples of the classes we have derived from `AbstractVector` include

`SerialVector`: a simple serial implementation representing vectors using the templated container `vector<double>` from the C++ Standard Template Library (STL) [18].

`BareSerialVector`: a simple serial implementation using C++ `double` arrays. This class lacks some convenient features, such as automatic resizing, that `SerialVector` acquires from the STL `vector` container, but runs more efficiently.

`BlasSerialVector`: similar to `BareSerialVector`, but coded to use BLAS routines for numerical operations.

`EpetraVector`: encapsulates vectors represented using the Epetra [12] parallel linear algebra system. Vectors may be distributed among multiple processors, and the Epetra BLAS is used for numerical operations.

Other classes may be derived from `AbstractVector` or from its derived classes. For example, one could derive a class from `EpetraVector` to represent vectors with specialized kinds of parallel data layouts.

To avoid cluttering algebraic expression code with excessive pointer dereferencing, we defined a second class, called `VectorObject`, which essentially encapsulates a pointer to an `AbstractVector`. The `AbstractVector` class contains abstract methods for common operations such as inner products and assignment, and `VectorObject` contains corresponding "pass-through" methods to access them. For example, if `x` and `y` are `VectorObjects` encapsulating pointers to the same `AbstractVector`-derived class, one may write the C++ expression `x.inner(y)` to denote $\langle x, y \rangle$. Evaluating this simple expression will end up invoking the `inner` method of the underlying `AbstractVector`-derived class.

Another class of `AbstractVector` methods are "cloning" routines that construct new vector objects that have the same representation as a given object, and whose contents may be selected to be copied, zeroed, or uninitialized. These routines are also typically accessed through the `VectorObject` wrappers, and allow an algorithm to properly allocate all the vector storage it needs cloning by single "template" vector object passed to it.

2.2 Symbolic Temporaries: Efficient Operator Overloading through Delayed Evaluation

The key goal or our abstract vector substrate is to be able to write a limited range of vector-related expressions with the same simplicity as a prototyping environment like MATLAB, and yet still retain most of the performance and control available through C++. For example, suppose an algorithm contains the calculation $z \leftarrow w + \alpha x - \beta y$, where w, x, y, and z are vectors and α and β are scalars. For simplicity and clarity, we would like to be able to translate this assignment into the C++ statement `z = w + alpha*x - beta*y`, where `w`, `x`, `y`, and `z` are all `VectorObject` encapsulating pointers to the same `AbstractVector`-derived type, and `alpha` and `beta` are of type `double`. If were to define overloaded operators in the conventional C++ manner, however, the resulting compiler temporary objects would cause excessive memory allocation and access operations. In the case of the expression above, for example, the standard C++ approach to operator overloading would result in the following operations:

1. An `operator*(double&,VectorObject&)` method would allocate new memory for a similar `VectorObject` and fill it with a the values of `x`, scaled by `alpha`.

2. A similar operation would create a temporary object containing `beta*y`.

3. An `operator-(VectorObject&,VectorObject&)` method would be invoked to calculate `alpha*x - beta*y` from the preceding two temporary objects and place the results in a third temporary `VectorObject`.

4. An `operator+(VectorObject&,VectorObject&)` method would be invoked to calculate the sum of `w` and the third temporary object, creating a fourth temporary `VectorObject`.

5. Finally, an `operator=(VectorObject&,VectorObject&)` method would be invoked to copy the contents of the last temporary into `w`.

For relatively compact objects — for example, representations of complex numbers consisting of two `doubles` — the overhead resulting from such operations might not be significant or could be reduced or perhaps eliminated by "downstream" code optimization steps within the C++ compiler. In situations in which each vector consists of many megabytes of data, possible spread over different processor memory spaces, there will be a significant speed and memory penalty for executing the calculation in the manner shown above. Specifically, the calculation will allocate four unnecessary temporary objects, each of which will be written and read one time. If one were implementing this assignment calculation using BLAS routines, it could be coded with just two DAXPY calls and no temporary objects, although

at the cost of some opaqueness in the code. In modern computer architectures, memory access operations and cache misses can be much more time-consuming than arithmetic calculations, so it is imperative to economize on memory operations if one is attempting to optimize performance. Furthermore, if the vectors `w`, `x`, `y`, and `z` are extremely large, allocating temporary objects of the same size might strain or exceed available system memory. For these reasons, the standard approach to operator overloading is not a viable option.

Instead, we have developed an alternative, "delayed evaluation" approach to operator overloading, involving an auxiliary class called `LinearExpression`, which is essentially an STL vector of pairs of the form (α_i, p_i), where α_i is a `double` and p_i is a pointer to an `AbstractVector`. In this approach, all the overloaded operators involving `VectorObject`s, except those performing assignment, create `LinearExpression` objects. The result of evaluating the entire right-hand side of an assignment is a `LinearExpression` of the form $\big((\alpha_1, p_1), \ldots, (\alpha_\ell, p_\ell)\big)$. Then, an overloaded assignment operator, most commonly of the form `operator=(VectorObject&,LinearExpression&)`, has the job of calculating the vector $\sum_{i=1}^{\ell} \alpha_i(*p_i)$ (here, the "$*$" denotes pointer dereferencing). Consider the same assignment expression `z = w + alpha*x - beta*y` discussed above: with our overloaded `VectorObject` and `LinearExpression` methods, the actual evaluation of this expression is as follows:

1. The `operator*(double&,VectorObject&)` method creates a temporary object of type `LinearExpression`, containing `(alpha,x.vp)`, that is, a pair containing the value of `alpha` and the address of the `AbstractVector`-derived vector representation encapsulated by `x`.

2. Another application of `operator*(double&,VectorObject&)` creates another temporary `LinearExpression`, containing `(beta,y.vp)`.

3. The method `operator-(LinearExpression&,LinearExpression&)` combines these two linear expressions and creates a third temporary `LinearExpression`, containing $\big($`(alpha,x.vp)`,`(-beta,y.vp)`$\big)$.

4. The method `operator+(double&,LinearExpression&)` appends an additional pair to this `LinearExpression`, producing a `LinearExpression` of the form

   $\big($`(1,w.vp)`,`(alpha,x.vp)`,`(-beta,y.vp)`$\big)$.

5. The method `operator=(VectorObject&,LinearExpression&)` computes the value of this last `LinearExpression` and places it in the memory dedicated to `z`. This task is accomplished by calling wrapper methods within the `VectorObject z`, which in turn call virtual methods of the `AbstractVector` class. Thus, how the assignment calculation is implemented depends on exactly what kind of `AbstractVector`-derived representations the `VectorObject`s are encapsulating. If they are of type `BlasVector`, for example, the evaluation would indeed reduce to two DAXPY operations.

Our procedure generates exactly the same number of temporary objects as the conventional approach, but they are extremely compact objects, basically small arrays of pairs each consisting of 16 bytes (assuming a system with 64-bit addresses). We call these objects *symbolic temporaries* because each symbolically encodes the linear combinations of vectors that needs

to be calculated. These objects are so small that they are likely to fit in the fastest level of processor cache, and the overhead involved in manipulating them is therefore likely to be negligible for the applications we have in mind, in which the vectors themselves will typically be extremely large. We may also call the technique *delayed evaluation* because it only performs arithmetic when it processes the assignment operation. At this point, the entire expression to be calculated is known and its evaluation can be optimized. However, this optimization occurs at run time, not at compile time.

In addition to the `=` operator, we also overload C++'s `+=` and `-=` operators in similar ways, allowing them to be used efficiently on expressions involving `VectorObject`s. We also provide versions of the `inner` method for all four possible combinations of `VectorObject` and `LinearExpression` arguments. For example, a mathematical expression of the form $\langle x, y + \alpha z \rangle$ would be rendered in our C++ approach as `x.inner(y + alpha*z)`. At run time, this expression would end up invoking an inner-product evaluation method in the `AbstractVector`-derived class encapsulated by `x` on a `LinearExpression` containing the pairs $\big($`(1,&y)`,`(alpha,&z)`$\big)$. Depending on how the encapsulated `AbstractVector`-derived class is implemented, this calculation might be done very efficiently, without allocating significant temporary memory. Such inner-product calculations are the only circumstances in which we might evaluate the result of a `LinearExpression` before encountering of an assignment operator.

It is certainly possible to extend our symbolic-temporary techniques to more complicated expressions than simple linear combinations of vectors. This effort might be worthwhile, although it could conceivably add significant complexity. The basic functionality we have implemented should be sufficient to create readable yet efficient first-order algorithms that we have in mind.

Sufficiently powerful parallel-environment-targeted compilers could in principle produce more efficient code than our techniques, because the analysis of arithmetic expressions could be performed completely and be optimized just once at compile time, rather than having some aspects of expression analysis extensively repeated at run time, as can effectively happen with our approach. However, such compilers do not presently exist, nor even does an active, accepted language standard such compilers could implement. The HPF standard [14] appears to be the closest the high-performance computing community has come to such a standard, but HPF compilers are not commonly available, and the language has some limitations due to its FORTRAN heritage. Because our run-time temporary objects are very small, the time required to manipulate them is likely to be negligible in the kind of large-scale applications that realistically require large-scale parallel computing. Finally, while the effort required to develop our symbolic temporary classes was considerable, it was minuscule compared to the effort that would be required to develop a flexible, efficient, and portable parallel language and compiler.

3 Optimization-Specific Infrastructure

3.1 The `AbstractProblem` Class: Encapsulating Problem Instances and Data Representations

We also developed and `AbstractProblem` class, which constitutes a unified interface through which our first-order optimization algorithms interact with problem instances. One defines a class of problems by deriving a class from `AbstractProblem` (although perhaps indirectly). `AbstractProblem`-derived classes hold problem instance data and contain methods for objective function and gradient evaluation. The problem class also stores upper and lower bounds on variables, when present. `AbstractProblem`-derived classes will contain methods to represent general constraints, once we extend our framework to handle such constraints. `AbstractProblem` also contains a `VectorObject` that represents the solution algorithm starting point and abstract methods to generate objects for caching function and gradient values.

Through the `VectorObject` representing the algorithm starting point, the `AbstractProblem`-derived class determines the vector class that the solution algorithm uses to execute the linear algebra computations. The algorithm classes initialize their `VectorObject` members by executing "cloning" methods on the initial point provided by the problem class. For example, the `AbstractProblem`-derived class we created called `LassoProblemEpetra` implements the objective function and gradient evaluations for LASSO problems (see Section 5 below), representing the solution through `EpetraVector` objects. In parallel settings, it is the `AbstractProblem`-derived class, through the algorithm starting point vector, that determines how decision variables are distributed and possibly replicated among processors. This class is also responsible for storing problem instance data, including distribution between processors in parallel settings. When they are running, the solution algorithms interact with problem instances mainly by calling the `AbstractProblem` virtual methods `objVal` and `objGrad` to evaluate the objective function value and gradient, respectively.

3.2 Line-Search Support: Transparent Function and Gradient Caching

First-order algorithms often employ line-search procedures. Essentially, after determining a step direction d^k, they perform some kind of backtracking procedure to determine the stepsize α_k in the step calculation $x^{k+1} = x^k + \alpha_k d^k$. To promote efficient implementation of such procedures, our framework provides a base class called `LineSearchBasedMethod`. This class also provides built-in members for scalar and vector quantities typically maintained by line search algorithms, including the objective values, the current iterate, the objective function gradient, and the search direction.

Crucially, the `LineSearchBasedMethod` class also holds an array of `PointMemory` objects. `PointMemory` is a class designed to cache function values, gradient values, and, through derived classes, related application-specific information. `LineSearchBasedMethod` provides methods `Phi` and `GradPhi` for computing the value and gradient of the line-search function $\phi_k(\alpha) = f(x^k + \alpha d^k)$, where f is the problem objective function. In some line-search procedures, $\phi_k(\alpha)$ or $\nabla\phi_k(\alpha)$ may be evaluated more than once at the same value of α; for example, this phenomenon can occur quite often in the conjugate gradient algorithm of [11].

The caching mechanism built into the `Phi` and `GradPhi` methods prevents potentially time-consuming recomputation of the function value or gradient in such cases, while keeping the solution algorithm code free of clutter from caching-related details. Another possible situation is that the line-search algorithm may compute $\phi_k(\alpha)$ and subsequently compute $\nabla\phi_k(\alpha)$ for the same value of α. For many problem classes, these two computations share significant common underlying computations, whose results it may be more efficient to cache than to recompute. Application-specific classes derived from `PointMemory` may be used for this purpose. Section 5.3 gives an example of such a derived class.

4 Actual Code for the SPG Method

As we already mentioned in the introduction, we chose the SPG method [2] as a relatively simpole test the usefulness of our classes. In our C++ environment, the SPG algorithm takes the following form:

```
VectorObject& SPG::minimize(AbstractTermination& termination) {

   while(termination.check() && iter < maxIter) {

       iter++;

       xk = x - stepSize*g;
       pr->projectOnBounds(xk, proj);
       d = proj - x; //d is the spectral projected gradient

       double gtd = g.inner(d);

       //Nonmonotone line search with safeguarded quadratic interpolation
       objValMax = *(std::max_element(objValArray, objValArray + M));
       alpha = 1.0;

       phia = phi(alpha);
       int counter = 0;

       while(phia > objValMax + gamma*alpha*gtd &&
             counter < maxLineIter) {                  //Armijo
           counter++;

           if (alpha <= 0.1) {
               alpha /= 2.0;
           }
           else {
               alphaTemp = (-gtd*alpha*alpha )
                           / (2.0*( phia - objVal - alpha*gtd ));

               if (alphaTemp < sigmaOne || alphaTemp > sigmaTwo*alpha)
                 alphaTemp = alpha/2.0;
```

```
                alpha = alphaTemp;
            }
            phia = phi(alpha);
        }
        //End of line search

        if(counter > maxCount)
          maxCount = counter;

        xk = x + alpha*d;
        pr->objGrad(xk, gk);

        xMinus = xk -x;
        double a = xMinus.norm2sq();
        gMinus = gk - g;
        double b = xMinus.inner(gMinus);

        if(b <= 0) {
            stepSize = stepSizeMax;
        }
        else {
            stepSize = std::max(stepSizeMin, std::min(stepSizeMax, a/b));
        }

        x = xk;
        g = gk;

        objVal = phia;
        objValArray[iter%M] = objVal;

        xk = x - g;
        pr->projectOnBounds(xk, proj);
        pg = proj - x;

        if(objVal < objValBest) {
            objValBest = objVal;
            xBest = x;
        }

        reset();

    }

    return xBest;
}
```

5 Application to LASSO Problems

As an initial test case to demonstrate the viability of our approach, we decided to solve some large bound-constrained problems using the tools we have developed. We selected the LASSO problem class as a test case for this effort; see for example [21]. This problem and its generalizations have gained great popularity due to large-scale data-analysis tools in recent years. In its original form, the LASSO problem may be written

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

where A is an $m \times n$ real matrix, $b \in \mathbb{R}^m$, and $\nu > 0$ is a given parameter. Essentially, the problem is a form of linear regression augmented with an ℓ_1 penalty for moving the regression coefficients x away from zero. Often we have $n \gg m$, that is, A is a "wide" matrix with many more columns than rows. Our interest here will be in problem instance in which A is an extremely large matrix, either sparse or dense.

5.1 LASSO as Orthant-Constrained Smooth Optimization

In its original form (1), the LASSO problem is an unconstrained convex minimization problem, but nonsmooth due the presence of the term $\nu \|x\|_1$. However, it is easily converted to a bound-constrained smooth optimization problem as follows: we let $x = x^+ - x^-$, where x^+ and x^- are two vectors in $\mathbb{R}^n$, both constrained to be nonnegative. Letting $\mathbf{1}$ denote the vector of all 1's in $\mathbb{R}^n$, we may then write the term $\nu \|x\|_1$ as $\nu \mathbf{1}^\top (x^+ + x^-)$. Therefore, the problem (1) may be re-expressed as

$$\begin{array}{ll} \displaystyle\min_{x^+, x^- \in \mathbb{R}^n} & \tfrac{1}{2} \|A(x^+ - x^-) - b\|^2 + \nu \mathbf{1}^\top (x^+ + x^-) \\ \text{ST} & x^+, x^- \geq 0. \end{array} \tag{2}$$

The problem is thus converted to minimization of a positive semidefinite quadratic (and hence smooth) function of (x^+, x^-) subject to the constraint that (x^+, x^-) must lie in the nonnegative orthant. Note that vectors (x^+, x^-) in the feasible region of this problem may possess an index i for which $x_i^+ > 0$ and $x_i^- > 0$, but this situation cannot occur in an optimal solution because setting $x_i^+ \leftarrow \max\{x_i^+ - x_i^-, 0\}$ and $x_i^- \leftarrow \max\{x_i^- - x_i^+, 0\}$ immediately yields a lower value of the objective function without violating the constraints.

5.2 Data Distribution and Parallel Implementation

The problem formulation (2) is well-suited to solution by the SPG method. Therefore, we use the problem class to demonstrate how the implementation of SPG described in Section 4 can be used within an efficient, scalable parallel algorithm, despite its MATLAB-like simplicity, focusing on the fairly common case that $n \gg m$, meaning that x, x^+, and x^- are very high-dimensional vectors in comparison with b.

We adopt the straightforward approach of simply partitioning responsibility for the elements of x^+ and x^-, along with the corresponding columns of A, among the available processor cores. Suppose we have P processors numbered $1, \ldots, P$, and let $\{J_1, \ldots, J_P\}$ be

a partition of $\{1, \ldots, n\}$, with J_i denoting the set of indices assigned to processor i. We momentarily defer discussing how we determine the partition $\{J_1, \ldots, J_P\}$. We may now depict the memory layout of the problem as follows:

1		i		P
$A_{(1)}$	$\cdots$	$A_{(i)}$	$\cdots$	$A_{(P)}$
$x^+_{(1)}$ $x^-_{(1)}$	$\cdots$ $\cdots$	$x^+_{(i)}$ $x^-_{(i)}$	$\cdots$ $\cdots$	$x^+_{(P)}$ $x^-_{(P)}$

.

Here, $x^+_{(i)}$ and $x^-_{(i)}$ denote the respective subvectors of x^+ and x^- consisting of the coefficients with indices assigned to processor i, that is, $x^+_{(i)}$ denotes the vector consisting of x^+_j for $j \in J_i$, and similarly for $x^-_{(i)}$. Similarly, $A_{(i)}$ denotes the matrix consisting of the columns of A_j of A for which $j \in J_i$. We replicate the vector b in the memory of all processors. However, b is a much shorter vector than x, so the impact of this replication on memory usage is limited. Throughout this discussion, we treat each processor as having its own memory; this does not prevent the method from being implemented on a system in which physical memory is shared.

To give the SPG implementation of Section 4 the ability to operate on our problem formulation and memory layout, we derived two classes from `AbstractProblem`, one using a dense representation of A and the other using a sparse representation. In both cases, we represent the decision variable vector $(x^+_{(i)}, x^-_{(i)})$ through a class derived from `EpetraVector`, using a map descriptor that gives each processor i ownership of $x^+_{(i)}$ and $x^-_{(i)}$. Note that with this setup, each vector `x` in the code for the SPG algorithm represents a pair of vectors (x^+, x^-) from the problem formulation (2), with processor i "owning" the subvector $(x^+_{(i)}, x^-_{(i)})$.

Let $h(x^+, x^-)$ denote the objective function of (2). We calculate

$$\nabla h(x^+, x^-) = \begin{bmatrix} A^\top\big(A(x^+ - x^-) - b\big) + \nu \mathbf{1} \\ -A^\top\big(A(x^+ - x^-) - b\big) + \nu \mathbf{1} \end{bmatrix}. \tag{3}$$

The dominant numerical work in calculating $h(x^+, x^-)$ and $\nabla h(x^+, x^-)$ consists of the matrix multiplication $A(x^+ - x^-)$, needed by both the function and gradient calculations, and the subsequent multiplication by $A^\top$ needed to calculate the gradient. Considering the first multiplication, we note that

$$A(x^+ - x^-) = \sum_{i=1}^{P} A_{(i)}(x^+_{(i)} - x^-_{(i)}).$$

Hence, we may calculate $A(x^+ - x^-)$ as follows:

1. Each processor i locally calculates $d_{(i)} = x^+_{(i)} - x^-_{(i)}$
2. Each processor i locally calculates $p_{(i)} = A_{(i)} d_{(i)} = A(x^+_{(i)} - x^-_{(i)})$

3. We sum the vectors $p_{(i)} \in \mathbb{R}^m$ over all processors using an MPI `ReduceAll` operation. Every processor thus receives the vector $p = A(x^+ - x^-)$.

The additional steps needed to calculate the objective function $h(x^+, x^-)$ are now as follows:

1. Each processor i locally computes $\omega_i = \sum_{j \in J_i}(x_j^+ + x_j^-)$
2. We compute the sum $\omega = \sum_{i=1}^P \omega_i = \mathbf{1}^\top(x^+ + x^-)$ by an MPI `ReduceAll` operation. For efficiency on systems with high per-message communication overhead, this operation could conceivably be combined with the `ReduceAll` operation needed to compute p (step 3 immediately above).
3. Each processor locally computes $r = p - b = A(x^+ - x^-) - b$ and then $h(x^+, x^-) = \frac{1}{2}\|r\|^2 + \nu\omega$.

At points (x^+, x^-) where the gradient $(y^+, y^-) = \nabla h(x^+, x^-)$ is also required, we proceed as follows:

1. Each processor locally computes $u_{(i)} = A_{(i)}^\top r = A_{(i)}^\top \big(A(x^+ - x^-) - b\big)$
2. Each processor locally computes $y_{(i)}^+ = u_{(i)} + (\nu, \ldots, \nu)$ and $y_{(i)}^- = -u_{(i)} + (\nu, \ldots, \nu)$.

This procedure leaves the system with a representation of the gradient vector (y^+, y^-) that is distributed across processors in exactly the same manner as the decision variable vector (x^+, x^-). This is precisely what is needed for our parallel implementation of the SPG method (or any other first-order algorithm implementation). In summary, the function value and gradient may be calculated using one or two global reduction operations, local vector operations, and two local matrix multiplications (or just one if only the function value is needed).

The key to obtaining parallel efficiency in this situation is balancing the workload involved in the local multiplications by $A_{(i)}$ and $A_{(i)}^\top$, which are the dominant workload in terms of numerical operations. To a lesser extent, it is also desirable to balance the number of indices j assigned to each processor, so that vector addition and scaling operations are efficient in parallel. When the matrix A is dense, these goals are compatible and easily attained: we determine the partition members J_i by simply dividing the indices $\{1 \ldots, n\}$ into P contiguous groups whose size varies by at most one element: formally, we may take the first $n \bmod P$ groups to have size $\lceil n/P \rceil$ and the remaining groups to have size $\lfloor n/P \rfloor$. This simple procedure yields near-perfect balance between the number of numerical calculations performed by the various processors.

When the matrix A is sparse, the situation is more complicated. Fundamentally, the primary goal is to approximately balance the number of nonzero entries in the matrices $A_{(i)}$, as these nonzero counts essentially determine the amount of work required for the local matrix multiplications, the dominant portion of the workload. This problem is equivalent to the $\mathcal{NP}$-hard minimum makespan scheduling problem; see for example [23]. Given a list of integers $a_1, \ldots, a_n$, this is the problem of partitioning $\{1, \ldots, n\}$ into P sets $J_1, \ldots, J_P$ so as to minimize $\max_{i=1,\ldots,P}\{\sum_{j \in J_i} a_j\}$. However, rather than solving or approximating this problem at run time for each possible value of P, we elected to heuristically solve the problem

Data: Matrix $A \in \mathbb{R}^{m \times n}$, integer m, integer n, even integer $\overline{P}$ (maximum number of cores)
Result: A partition $\{J_1, \ldots, J_{\overline{P}}\}$ such that the number of nonzero elements in the submatrices $A_{(i)}$ are approximately balanced
Sort the columns by number of nonzero entries, yielding a permutation π such that column $\pi(j)$ is the jth densest column
for $j = 0, \ldots, n-1$ **do**
  if $j \mod 2 = 0$ **then**
    $k = \frac{j}{2} \mod \frac{\overline{P}}{2}$ /* $k \in \left[0, \frac{\overline{P}}{2} - 1\right]$ */
  else
    $k = \left(\overline{P} - 1\right) - \left(\frac{j-1}{2} \mod \frac{\overline{P}}{2}\right)$ /* $k \in \left[\frac{\overline{P}}{2}, \overline{P} - 1\right]$ */
  end
  $J_{k+1} \leftarrow J_{k+1} \cup \{\pi(j+1)\}$ /* Convert between 0- and 1-based indexing */
end

Figure 1: Heuristic partitioning scheme for columns of the LASSO data matrix.

just once for a large value of P, which we call $\overline{P}$. For smaller values of P that divide $\overline{P}$, we simply aggregate adjacent groups from the $\overline{P}$-partition. Furthermore, all groups of columns are as similar in size as possible, that is, all groups in the $\overline{P}$-partition are of size $\lceil n/\overline{P} \rceil$ or $\lfloor n/\overline{P} \rfloor$, and a similar property holds for the aggregated partitions. The heuristic we use for this purpose is shown in Figure 1. Essentially, we sort the columns from most nonzero elements to fewest nonzero elements, and then assign them to partition groups "from the outside in", placing columns in the first group, then the last, then the second group, then the second-to-last group, and so forth. When running on a system configuration with whose number of processors P divides $\overline{P}$, we construct our partition groups by simply aggregating adjacent sets of $\overline{P}/P$ groups from the $\overline{P}$-partition. This heuristic procedure achieves an acceptable balance if the problem instance lacks any highly dense columns, or the number of such dense columns is a multiple of the number of partitions and they have similar numbers of nonzero elements.

5.3 Sharing computational effort between objective function value and gradient computations for the LASSO problem class

As already mentioned in Section 3.2, the line-search methods often used by first-order constrained or box-constrained optimization algorithms compute values and gradients of the line search function ϕ_k at various values of its argument α, which in turn means evaluating the objective function h and its gradient at various points. If the gradient is computed at a point α, there is a high likelihood that the corresponding gradient will also be needed, and *vice versa*. Referring to (1) and (3), it is evident that the bulk of the numerical calculations required for both the function value and gradient calculations for h are embodied in calculating $r = A(x^+ - x^-) - b$. Therefore, if we cache the vector r while calculating the gradient, the objective value computation at the same point reduces to evaluating the square of the Euclidean norm of r and the inner product $\nu \mathbf{1}^T(x^+ + x^-)$. Conversely, if we save r while

calculating the objective value, the work required to evaluate (3) is greatly reduced. Therefore, we derived a specialized version of the `PointMemory` class that stores the value of r whenever the objective value or gradient is computed, to prevent unnecessary matrix-vector multiplications. The LASSO problem classes provide objects of this type to the first-order solution methods.

5.4 Parallel I/O

In our implementation, we used the Lustre file system to "stripe" our test input data, storing it across multiple disks. We then used MPI's parallel I/O functions, such as `MPI_File_seek` and `MPI_File_read`, to read in the data. In a large binary file, each core seeks to the position where its data starts, and then reads in an appropriate portion of the data. We thus prevented data read-in from being a bottleneck operation detrimental to scalability.

5.5 Computational Results

We now present scaling results for our implementation of SPG for the LASSO problem. We ran the tests on the TACC Stampede supercomputer. Each node of this system consists of two 2.7 GHz 8-core Xeon processors with 32GB of RAM, and the nodes are connected by an InfiniBand network with a fat-tree-class topology. This system also has "Xeon Phi" accelerator cards, but for portability reasons we did not attempt to use them.

We give scaling results for six different problem instances, four of which are derived from real-world datasets, and two of which we randomly generated. The real-world datasets are in fact from classification problems, which we translate to related LASSO problems using a standard procedure. Three of these problems were obtained from the UCI repository [1] and one from the LIBSVM data sets [7, 4] website. Table 1 gives the run-time results for the real-world problems, and Table 2 shows the results for the randomly generated problems. In these tables, "efficiency" denotes empirical execution-time efficiency relative to the the run with the smallest number of cores (usually one core). Figures 2-6 graphically display the same information in a series of charts. The upper right area of each chart contains the following information about the data matrices A:

S or D	Indicates sparse or dense
$m \times n$	m rows and n columns
Im	Load Imbalance (for sparse problems only)

Here, "Load Imbalance" denotes the difference in the number of nonzero entries between the densest and the sparsest column. The time axis uses a $\log_{10}$ scale, whereas the processor count axis has a $\log_2$ scale. The solid line represents perfect scaling, while the red dashed line the performance of our SPG implementation. The points that fall below the line denoting perfect scaling (indicating superlinear speedup) are due to the numerical sensitivity of the SPG algorithm: as one varies the number of processor cores, there tend to be variations in the number of iterations SPG requires to find the optimal solution. When one changes the number of processor cores, the terms in each inner product operation are effectively summed in a different order, causing small variations in the results and affecting the subsequent path taken by the method. As we further develop our software, we plan

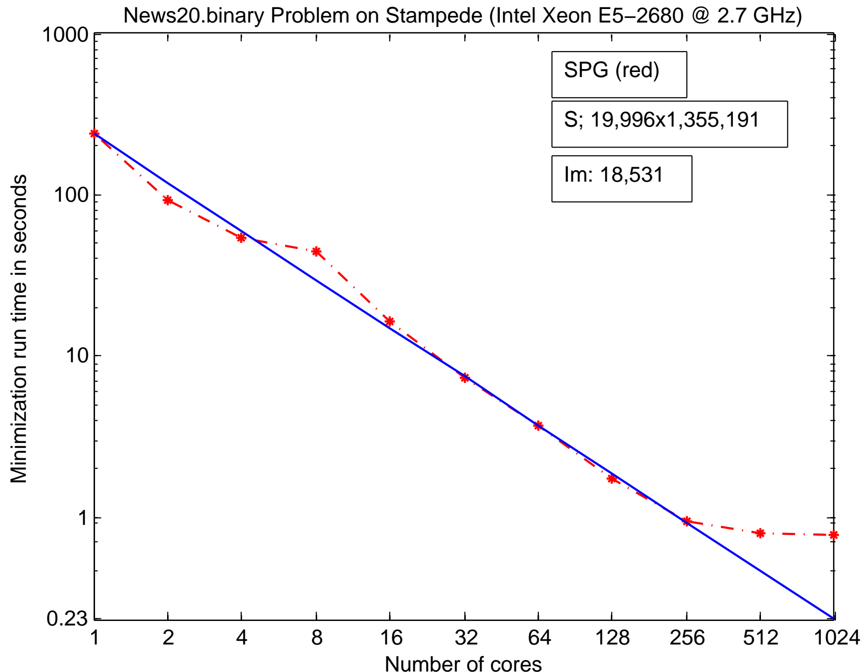

Figure 2: SPG speedup performance on text classification.

to solve augmented Lagrangian subproblems using the ASA CG-Descent algorithm [11] in addition to the SPG method. This method is known to be less numerically sensitive than SPG, so such numerically-induced speedup anomalies should become less pronounced; we have already implemented the unconstrained version of CG-Descent [9, 10], but still need to implement the version that can process variable bounds.

We do not know the exact cause of the "spikes" in speedup behavior between 4 and 8 processor cores which may be observed in all the charts below. However, we have been able to reproduce the phenomenon outside our SPG algorithm and it seems to be caused by an interaction of the Intel Xeon architecture with certain MPI and BLAS functions.

5.5.1 Text classification

This sparse data set [13, 16] is a two-class variant of the UCI "Twenty Newsgroups" data set. The problem involves classifying messages as positive or negative in tone. The dataset is sparse, with A having about 1.35 million columns and 20,000 rows. As shown in Figure 2, our algorithm exhibits nearly linear scaling behavior through 256 processor cores, after which little further speedup is obtained.

5.5.2 Drug discovery

The sparse data set "Dorothea" [8], which is part of the UCI Machine Learning Repository, was prepared for the NIPS 2003 variable and feature selection benchmark by Isabelle Gyuon. The problem involves classifying chemical compounds represented by structural molecular features as active (binding to thrombin) or inactive.

Problem Name	Cores	SPG Iterations	Run Time	Efficiency
News 20 Binary (sparse)	1	647	237.71 sec	100%
	2	494	92.16 sec	128%
	4	458	54.24 sec	109%
	8	475	44.18 sec	67%
	16	475	16.35 sec	90%
	32	489	7.30 sec	101%
	64	515	3.71 sec	99%
	128	460	1.74 sec	106%
	256	460	0.94 sec	97%
	512	486	0.79 sec	58%
	1024	466	0.76 sec	30%
Dorothea (sparse)	1	140	1.51 sec	100%
	2	140	0.77 sec	98%
	4	146	0.43 sec	87%
	8	145	0.27 sec	69%
	16	145	0.10 sec	94%
	32	145	0.11 sec	42%
	64	145	0.62 sec	4%
	128	145	0.74 sec	2%
	256	145	0.55 sec	1%
	512	145	0.10 sec	3%
	1024	145	0.26 sec	1%
PEMS-SF (dense)	1	686	82.15 sec	100%
	2	761	44.99 sec	91%
	4	720	24.50 sec	83%
	8	755	20.16 sec	50%
	16	797	10.69 sec	48%
	32	712	4.89 sec	52%
	64	622	2.22 sec	57%
	128	678	0.84 sec	76%
	256	711	0.35 sec	91%
	512	599	0.26 sec	60%
	1024	649	0.30 sec	26%
Acetone (dense)	1	14274	511.71 sec	100%
	2	12659	236.30 sec	108%
	4	12615	133.60 sec	95%
	8	14273	112.72 sec	56%
	16	11868	45.17 sec	70%
	32	13283	12.77 sec	125%
	64	12421	5.27 sec	151%
	128	12531	3.95 sec	101%
	256	14564	4.25 sec	47%
	512	14049	4.44 sec	22%
	1024	14146	5.45 sec	9%

Table 1: Computational results for real data, with efficiencies based on run times.

Problem Name	Cores	SPG Iterations	Run Time	Efficiency
Dense Random	128	842	889.16 sec	100%
	256	787	417.34 sec	106%
	512	686	185.99 sec	119%
	1024	834	112.99 sec	98%
	2048	681	47.59 sec	116%
	4096	714	28.54 sec	97%
Sparse Random	1	704	1490.18 sec	100%
	2	597	634.98 sec	117%
	4	749	444.66 sec	83%
	8	929	382.72 sec	48%
	16	761	151.82 sec	61%
	32	682	60.10 sec	77%
	64	657	27.22 sec	85%
	128	680	13.94 sec	83%
	256	717	7.92 sec	73%
	512	711	4.46 sec	65%
	1024	721	2.71 sec	53%
	2048	662	1.52 sec	47%

Table 2: Computational results for randomly generated data, with efficiencies based on run times.

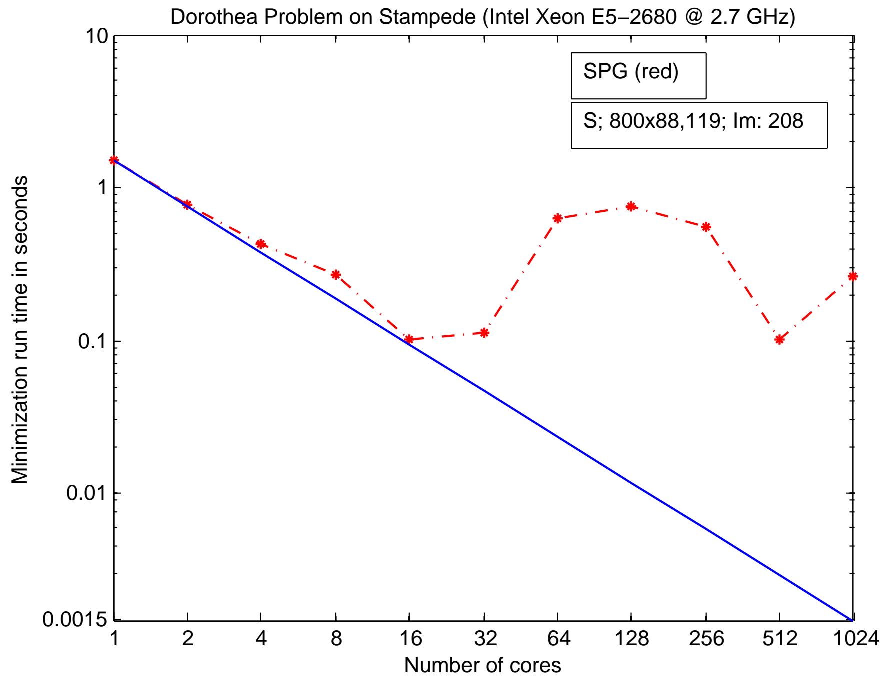

Figure 3: SPG speedup performance on Dorothea drug discovery dataset.

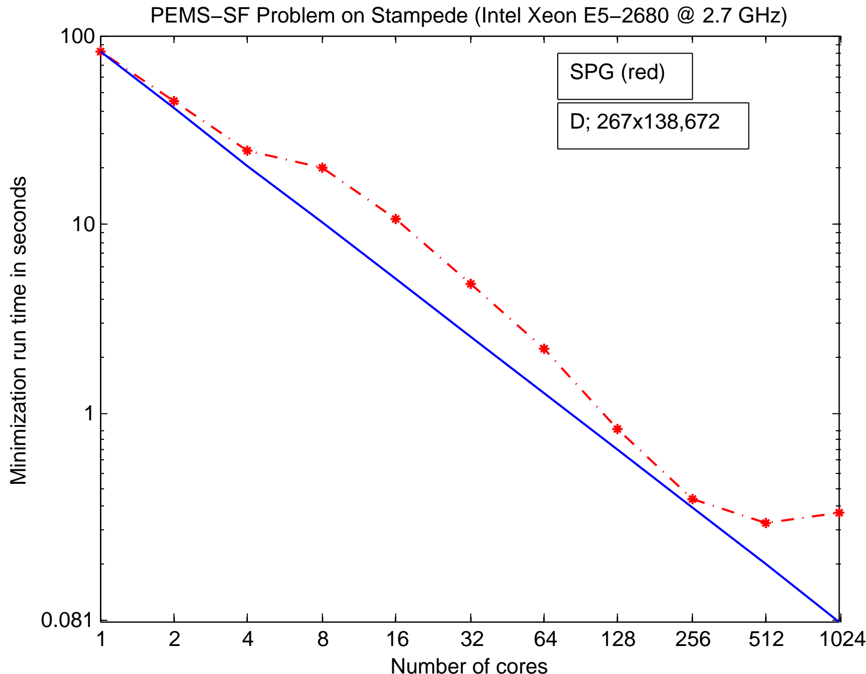

Figure 4: SPG speedup performance on traffic dataset.

Dorothea is a relatively small data set, with only 800 rows and about 88,000 columns. The size of the binary file holding the non-zero values of matrix A is only 5.8MB. As shown in Figure 3, we obtain very good scaling through 16 cores, after which performance degrades. The computation time for a single core is only 1.52 seconds, in which SPG performs 447 matrix-vector multiplications. That we obtain near-linear speedup through 16 cores in such a case suggests that the overhead associated with our code design is minimal.

5.5.3 Classification with traffic data

"PEMS-SF" is a dense data set from the UCI Machine Learning Repository [1], consisting of 15 months' worth of daily data describing the occupancy rate, across time, of various travel lanes in the San Francisco area freeway system. It has 267 rows and about 139,000 columns. The associated task proposed in the repository is to classify each observed day as the correct day of the week, from Monday to Sunday. To produce reasonable related LASSO instances, we first transformed this seven-class classification problem into a two-class variant by simply attempting to classify each observed day as either a weekend day or a weekday. As shown in Figure 4, our SPG obtains good speedups through 256 cores on this instance, but little or no additional speedup for higher core counts.

5.5.4 Classifying gas flow

This dense data set [24] from the UCI Machine Learning Repository was created by Andrey Ziyatdinov and Jordi Fonollosa. It contains 58 time series acquired from 16 chemical sensors exposed to gas flow. The sensors measured gaseous mixtures that contained different levels of acetone and ethanol. We set the goal of the classification problem to be separating gaseous

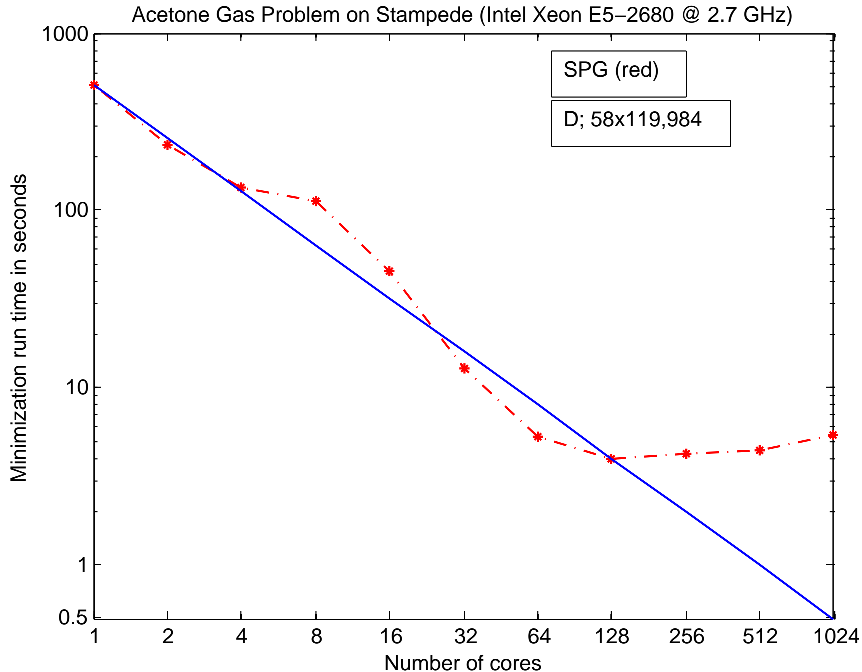

Figure 5: SPG speedup performance on acetone gas-flow dataset.

mixtures that contain acetone from those that did not, hence the problem is identified as "Acetone" in Table 1. Results are also shown in Figure 5. This data set is less than 1/5 the size of the traffic dataset, and it appears that linear scaling tails off between 64 and 128 cores. Comparing the vertical scales of the above charts, it is apparent that SPG spends proportionally more time on this problem than on others. This is due to relatively slow convergence of the SPG method, which requires SPG to perform around 13,000 iterations and 48,000 matrix-vector multiplies, whereas the traffic data only requires about 750 iterations and 2,500 matrix-vector multiplies.

5.5.5 Randomly generated instances

In this subsection, we give performance results for two large-scale instances that we randomly generated, one dense and one sparse, both having 10,000 rows and 2.5 million columns. The sparse dataset has a density of 1%, with nonzeroes located randomly. In both cases, the nonzero matrix coefficients were generated from a unit normal distribution. These examples serve to illustrate that when the processor/data ratio is sufficiently large, good scaling can be achieved for large numbers of cores. The dense data set consumes 200GB in binary format whereas the sparse dataset requires only 2GB. Results are shown in Figures 6 and 7.

We solves the 200GB dense instance in 28.5 seconds on 4096 cores, with the algorithm performing 2448 matrix-vector multiplications. Speedups still appear to be nearly linear at this number of cores. Note that, due to its size, we did not attempt to solve this instance below 128 processor cores.

The sparse dataset can be solved on a single core, and speedups appear to be near-linear through 2048 cores, with some gradual degradation becoming evident at 1024 cores. We

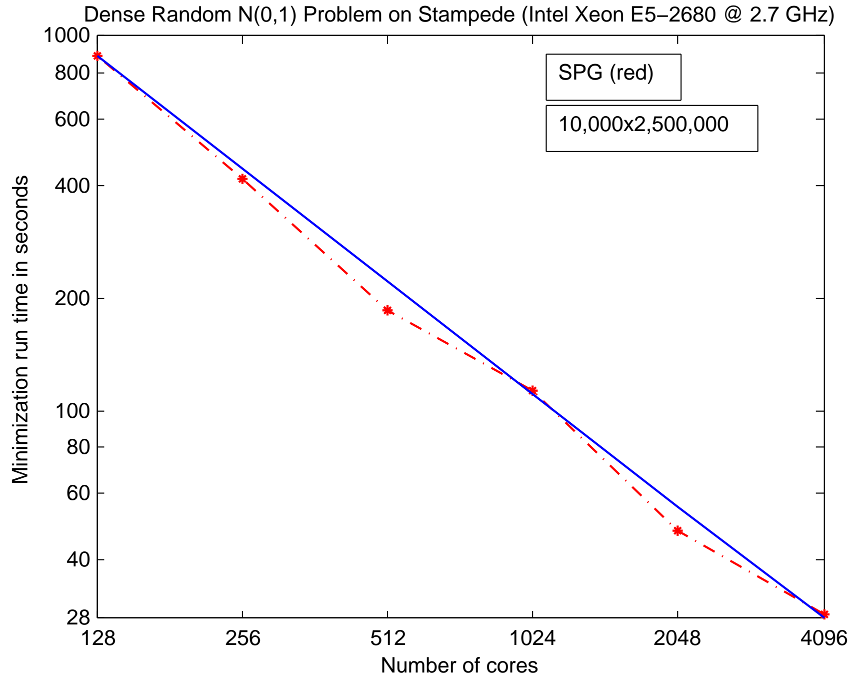

Figure 6: SPG speedup performance on random dense data.

hypothesize that degradation is more gradual than with the real sparse datasets due to the instance's more balanced numbers of nonzeroes in each column.

6 Conclusions and Planned Future Work

The performance results of the smaller problems indicate that our class structure and symbolic temporaries scheme do not create significant overhead, the results on the large datasets show that our code design allows for good scalability. To reinforce the latter conclusion, it would be helpful to use a less numerically sensitive algorithm than SPG, an issue we are currently addressing.

It may be observed that while we used object-oriented techniques and operator overloading to obtain an elegant expression of the SPG algorithm, we did not do the same for our lower-level code expressing the LASSO problem. It would certainly be possible to make our application-specific code more elegant, but it is not necessary for our approach. Our philosophy is that if the application developers can focus on efficiently performing just a few operations, namely function evaluation and gradient evaluation, in whatever way they see fit, then our object-oriented framework can use those low-level operations to construct an efficient parallel solver. The benefits of the operator overloading techniques are to make the solver-level code easier to understand and maintain, and to facilitate relatively easy development of new solver algorithms as necessary.

There are some LASSO instances for which our current application will not exhibit good speedups. We have observed that some of the datasets in the UCI repository have a small number of extremely dense columns that contain a substantial fraction of the nonzero ele-

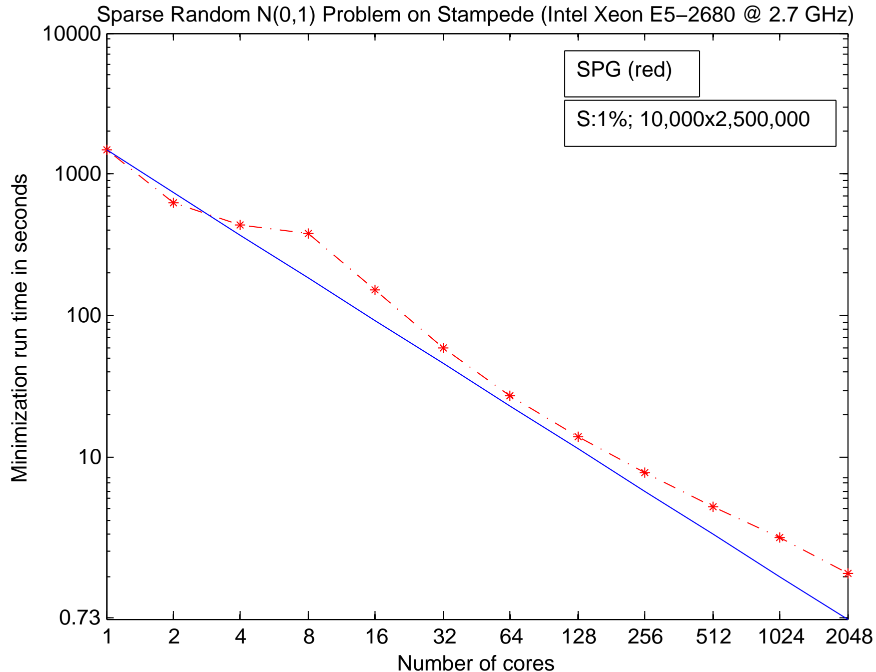

Figure 7: SPG speedup performance on random sparse data.

ments in the data matrix. One such example is the URL Reputation dataset [15]. We are currently revising our implementation to use a more sophisticated sparse matrix multiplication algorithm and data representation scheme, which we believe will be much less sensitive to imbalances in the number of nonzeroes between columns. In this approach, each processor will store nearly the same number of nonzeroes from the matrix A, but different processors will be responsible for different numbers of decision variables in a manner facilitating efficient matrix multiplication. There will in general be some variable overlap, meaning that some decision variables will be stored on multiple processors. This arrangement of replicated and distributed responsibility for variables is a simple example of what we call a *distribution scheme*, that is, some plan for distributing and (perhaps partially) replicating variables of an optimization problem so that a parallel first-order method may be applied efficiently. A key advantage of our object-oriented coding approach is that exactly the same SPG algorithm code in Section 4 could still be used in this situation, although the arrangement of decision variables across processors is different.

A longer-term goal is to extend the concept of a distribution scheme to constrained problems by specifying the layout of constraints and associated dual variables as well as the layout of decision variables. We plan to use this concept with a general augmented Lagrangian optimization solver based on the algorithm in [6], with a bound-constrained first-order algorithm, implemented using our `AbstractVector` substrate, approximately solving the subproblems.

References

[1] K. Bache and M. Lichman. UCI machine learning repository, 2014.

[2] E. G. Birgin, J. M. Martínez, and M. Raydan. Nonmonotone spectral projected gradient methods on convex sets. *SIAM J. Optim.*, 10(4):1196–1211, 2000.

[3] E. G. Birgin, J. M. Martínez, and M. Raydan. Inexact spectral projected gradient methods on convex sets. *IMA J. Numer. Anal.*, 23(4):539–559, 2003.

[4] C.-C. Chang and C.-J. Lin. LIBSVM: A library for support vector machines. *ACM Trans. Intell. Syst. Technol.*, 2(3):27:1–27:27, May 2011.

[5] L. Dagum and R. Menon. OpenMP: An industry-standard API for shared-memory programming. *IEEE Comput. Sci. Eng.*, 5(1):46–55, Jan. 1998.

[6] J. Eckstein and P. J. S. Silva. A practical relative error criterion for augmented Lagrangians. *Math. Program.*, 141(1-2):319–348, 2013.

[7] R.-E. Fan and C.-J. Lin. A library for support vector machines data sets. Technical report, National Taiwan University Department of Computer Science and Information Engineering, 2014.

[8] I. Guyon, S. Gunn, A. Ben-Hur, and G. Dror. Result analysis of the NIPS 2003 feature selection challenge. In L. Saul, Y. Weiss, and L. Bottou, editors, *Advances in Neural Information Processing Systems 17*, pages 545–552. MIT Press, 2005.

[9] W. W. Hager and H. Zhang. A new conjugate gradient method with guaranteed descent and an efficient line search. *SIAM J. Optim.*, 16(1):170–192, 2005.

[10] W. W. Hager and H. Zhang. Algorithm 851: CG_DESCENT, a conjugate gradient method with guaranteed descent. *ACM Trans. Math. Softw.*, 32(1):113–137, 2006.

[11] W. W. Hager and H. Zhang. A new active set algorithm for box constrained optimization. *SIAM J. Optim.*, 17(2):526–557, 2006.

[12] M. A. Heroux, R. A. Bartlett, V. E. Howle, R. J. Hoekstra, J. J. Hu, T. G. Kolda, R. B. Lehoucq, K. R. Long, R. P. Pawlowski, E. T. Phipps, A. G. Salinger, H. K. Thornquist, R. S. Tuminaro, J. M. Willenbring, A. Williams, and K. S. Stanley. An overview of the trilinos project. *ACM Trans. Math. Softw.*, 31(3):397–423, 2005.

[13] S. S. Keerthi and D. DeCoste. A modified finite Newton method for fast solution of large scale linear SVMs. *Journal of Machine Learning Research*, 6:341–361, 2005.

[14] K. Kennedy, C. Koelbel, and H. Zima. The rise and fall of high performance Fortran: An historical object lesson. In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*, HOPL III, pages 7-1–7-22, New York, NY, USA, 2007.

[15] J. Ma, L. K. Saul, S. Savage, and G. M. Voelker. Identifying suspicious URLs: An application of large-scale online learning. In *Proceedings of the International Conference on Machine Learning (ICML)*, pages 681–688, 2009.

[16] T. Mitchell. *Machine Learning*. McGraw Hill, 1997.

[17] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable parallel programming with CUDA. *Queue*, 6(2):40–53, 2008.

[18] P. Plauger, M. Lee, D. Musser, and A. A. Stepanov. *C++ Standard Template Library*. Prentice Hall, Upper Saddle River, NJ, USA, 1st edition, 2000.

[19] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI: The Complete Reference*. MIT Press, 1994.

[20] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, Boston, MA, USA, 3rd edition, 2000.

[21] R. Tibshirani. Regression shrinkage and selection via the lasso: a retrospective. *J. R. Stat. Soc.*, 73(3):273–282, 2011.

[22] S. van der Walt, S. Colbert, and G. Varoquaux. The NumPy array: A structure for efficient numerical computation. *Comput. Sci. Eng.*, 13(2):22–30, March 2011.

[23] V. V. Vazirani. *Approximation Algorithms*. Springer-Verlag, New York, NY, USA, 2001, 2003.

[24] A. Ziyatdinov, J. Fonollosa, L. Fernandez, A. Gutierrez-Galvez, S. Marco, and A. Perera. Bioinspired early detection through gas flow modulation in chemo-sensory systems. *Sensors and Actuators B: Chemical*, 206:538–547, 2014.