High-Performance Statistical Computing in the Computing Environments of the 2020s

Seyoon Ko[*] **Hua Zhou**

Department of Biostatistics, UCLA Fielding School of Public Health, Los Angeles, CA, USA

Jin J. Zhou

Department of Medicine, UCLA David Geffen School of Medicine, Los Angeles, CA, USA
and
Department of Epidemiology and Biostatistics, College of Public Health, University of Arizona, Tucson, AZ, USA

Joong-Ho Won

Department of Statistics, Seoul National University, Seoul, Korea

Abstract. Technological advances in the past decade, hardware and software alike, have made access to high-performance computing (HPC) easier than ever. We review these advances from a statistical computing perspective. Cloud computing makes access to supercomputers affordable. Deep learning software libraries make programming statistical algorithms easy and enable users to write code once and run it anywhere — from a laptop to a workstation with multiple graphics processing units (GPUs) or a supercomputer in a cloud. Highlighting how these developments benefit statisticians, we review recent optimization algorithms that are useful for high-dimensional models and can harness the power of HPC. Code snippets are provided to demonstrate the ease of programming. We also provide an easy-to-use distributed matrix data structure suitable for HPC. Employing this data structure, we illustrate various statistical applications including large-scale positron emission tomography and ℓ_1-regularized Cox regression. Our examples easily scale up to an 8-GPU workstation and a 720-CPU-core cluster in a cloud. As a case in point, we analyze the onset of type-2 diabetes from the UK Biobank with 200,000 subjects and about 500,000 single nucleotide polymorphisms using the HPC ℓ_1-regularized Cox regression. Fitting this half-million-variate model takes less than 45 minutes and reconfirms known associations. To our knowledge, this is the first demonstration of the feasibility of penalized regression of survival outcomes at this scale.

Key words and phrases: high-performance statistical computing, graphics processing units (GPUs), cloud computing, deep learning, MM algorithms, ADMM, PDHG, Cox regression.

(e-mail: wonj@stats.snu.ac.kr)

[*] This article is partly based on the first author's doctoral dissertation (Ko, 2020).

1. INTRODUCTION

Clock speeds of the central processing units (CPUs) on the desktop and laptop computers hit the physical limit more than a decade ago, and there will likely be no major breakthrough until quantum computing becomes practical. Instead, the increase in computing power is now accomplished by using multiple cores within a processor chip. High-performance computing (HPC) means computations that are so large that their requirement on storage, main memory, and raw computational speed cannot be met by a single (desktop) computer (Hager and Wellein, 2010). Modern HPC machines are equipped with more than one CPU that can work on the same problem (Eijkhout, 2016). Often, special-purpose co-processors such as graphics processing units (GPUs) are attached to the CPU to improve the speed by orders of magnitude for certain tasks. First developed for rendering graphics on a computer screen, a GPU can be thought of a massively parallel matrix-vector multiplier and vector transformer on a data stream. With increasing needs to analyze petabyte-scale data, the success of large-scale statistical computing relies on efficiently engaging HPC in the statistical practice.

About a decade ago, the second author discussed the potential of GPUs in statistical computing: Zhou et al. (2010) predicted that "GPUs will fundamentally alter the landscape of computational statistics." Yet, it does not appear that GPU computing, or HPC in general, has completely permeated the statistical community. Part of the reason for this may be attributed to the fear that parallel and distributed code is difficult to program, especially in R (R Core Team, 2021), the *lingua franca* of statisticians.[1] On the other hand, the landscape of scientific computing in general, including so-called data science (Donoho, 2017), has indeed substantially changed. Many high-level programming languages, such as Python (van Rossum, 1995) and Julia (Bezanson et al., 2017), support parallel computing by design or through standard libraries. Accordingly, many software tools have been developed in order to ease programming in and managing HPC environments. Last but not least, cloud computing (Fox, 2011) is getting rid of the necessity for purchasing expensive supercomputers and scales computation as needed.

Concurrently, easily parallelizable algorithms for fitting statistical models with hundreds of thousand parameters have also seen significant advances. Traditional Newton-Raphson or quasi-Newton type of algorithms face two major challenges in contemporary problems: 1) explosion of dimensionality renders storage and inversion of Hessian matrices prohibitive; 2) regularization of model complexity is almost essential in high-dimensional settings, which is often realized by nondifferentiable penalties; this leads to high-dimensional, nonsmooth optimization problems. For these reasons, nonsmooth first-order methods have been extensively studied during the past decade (Beck, 2017), since Hessian matrix inversion can be completely avoided. For relatively simple, decomposable penalties (Negahban et al., 2012), the proximal gradient method (Beck and Teboulle, 2009; Combettes and Pesquet, 2011; Parikh and Boyd, 2014; Polson et al., 2015) produces a family

[1] Although there exist several R packages for high-performance computing (Eddelbuettel, 2021), their functionalities and usability appear not to match what is available in other languages. In particular, the authors were not able to come up with a simple implementation of the computational tasks presented in this paper without writing low-level C/C++ code or using an interface to Python.

of easily parallelizable algorithms. For the prominent example of the Lasso (Tibshirani, 1996), this method contrasts to the highly efficient sequential coordinate descent method of Friedman et al. (2010) and smooth approximation approaches, e.g., Hunter and Li (2005). Decomposability or separability of variables is often the key to parallel and distributed algorithms. The alternating direction method of multipliers (ADMM, Gabay and Mercier, 1976; Boyd et al., 2011) achieves this goal through variable splitting, while often resulting in nontrivial subproblems to solve. As an alternative, the primal-dual hybrid gradient (PDHG) algorithm (Zhu and Chan, 2008; Esser et al., 2010; Chambolle and Pock, 2011; Condat, 2013; Vũ, 2013) has a very low per-iteration complexity, useful for complex penalties such as the generalized lasso (Tibshirani and Taylor, 2011; Ko et al., 2019; Ko and Won, 2019). Another route toward separability is the majorization-minimization (MM) principle (Lange et al., 2000; Hunter and Lange, 2004; Lange, 2016), which has been explored in Zhou et al. (2010). In fact, the proximal gradient method can be viewed as a realization of the MM principle. Recent developments in the application of this principle include distance majorization (Chi et al., 2014) and proximal distance algorithms (Keys et al., 2019). When the matrix to be inverted to solve the optimality condition has many independent components, nonsmooth Newton methods (Kummer, 1988; Qi and Sun, 1993) can be a viable option; see Huang et al. (2021) for recent applications to sparse regression. Nonsmooth Newton methods can also be combined with first-order methods for more complex nonsmooth penalties (Chu et al., 2020; Won, 2020).

The goal of this paper is to review the advances in parallel and distributed computing environments during the past decade and demonstrate how easy it has become to write code for large-scale, high-dimensional statistical models and run it on various distributed environments. In order to make the contrast clear, we deliberately take examples from Zhou et al. (2010), namely positron emission tomography (PET), nonnegative matrix factorization (NMF), and multidimensional scaling (MDS). The difference lies in the scale of the examples: our experiments deal with data of size at least $10,000 \times 10,000$ and as large as $200,000 \times 200,000$ for dense data, and $810,000 \times 179,700$ for sparse data. This contrasts with the size of at best 4096×2016 of Zhou et al. (2010). This level of scaling is possible because the use of *multiple* GPUs in a distributed fashion has become handy, as opposed to the single GPU, C-oriented programming environment of 2010. Furthermore, using the power of cloud computing and modern deep learning software, we show that exactly the *same*, easy-to-write code can run on multiple CPU cores and/or clusters of workstations. Thus we bust the common misconception that deep learning software is dedicated to neural networks and heuristic model fitting. Wherever possible, we apply more recent algorithms in order to cope with the scale of the problems. In addition, a new example of large-scale proportional hazards regression model is investigated. We demonstrate the potential of our approach through a single multivariate Cox regression model regularized by the ℓ_1 penalty on the UK Biobank genomics data (with 200,000 subjects), featuring time-to-onset of Type 2 Diabetes (T2D) as outcome and 500,000 genomic loci harboring single nucleotide polymorphisms as covariates. To our knowledge, such a large-scale joint genome-wide association analysis has not been attempted. The reported Cox regression model retains a large proportion of *bona fide* genomic loci associated with T2D and recovers many loci near genes involved in insulin resistance and inflammation, which may have been missed in conventional univariate analysis with moderate statistical significance values.

The rest of this article is organized as follows. We review HPC systems and see how they have become easy to use in Section 2. In Section 3, we review software libraries employing the "write once, run everywhere" principle (especially deep learning software) and discuss how they can be employed for fitting high-dimensional statistical models on the HPC systems of Section 2. In Section 4, we review modern scalable optimization techniques well-suited to HPC environments. We present how to distribute a large matrix over multiple devices in Section 5, and numerical examples in Section 6. The article is concluded in Section 7.

2. ACCESSIBLE HIGH-PERFORMANCE COMPUTING SYSTEMS

2.1 Preliminaries

Since modern HPC relies on parallel computing, in this section we review several concepts from parallel computing literature at a level minimally necessary for the subsequent discussions. Further details can be found in Nakano (2012); Eijkhout (2016).

Data parallelism. While parallelism can appear at various levels such as instruction-level and task-level, what is most relevant to statistical computing is data-level parallelism or data parallelism. If data can be subdivided into several pieces that can be processed independently of each other, then we say there is data parallelism in the problem. Many operations such as scalar multiplication of a vector, matrix-vector multiplication, and summation of all elements in a vector can exploit data parallelism using parallel architectures, which will be discussed shortly.

Memory models. In any computing system, processors (CPUs or GPUs) need to access data residing in the memory. While *physical* computer memory uses complex hierarchies (L1, L2, and L3 caches; bus- and network-connected, etc.), systems employ abstraction to provide programmers an appearance of transparent memory access. Such *logical* memory models can be categorized into the shared memory model and the distributed memory model. In the shared memory model, all processors share the *address space* of the system's memory even if it is physically distributed. For example, when two processors refer to a variable x, the variable is stored in the same memory address. Hence, if one processor alters the variable, then the other processor is affected by the modified value. Modern CPUs that have several cores within a processor chip fall into this category. On the other hand, in the distributed memory model, the system has memory both physically and logically distributed. Processors have their own memory address spaces and cannot see each other's memory directly. If two processors refer to a variable x, then there are two separate memory locations, each of which belongs to each processor under the same name. Hence the memory does appear distributed to programmers, and some explicit communication mechanism is required in order for processors to exchange data with each other. The advantage at the cost of this complication is scalability — the number of processors that can work in a tightly coupled fashion is much greater in distributed memory systems (say 100,000) than shared memory systems (say four, as many recent laptops are equipped with a CPU chip with 4 cores). Hybrids of the two memory models are also possible. A typical computer *cluster* consists of multiple *nodes* interconnected in a variety of network topology. A node is a workstation that can run standalone, with its main memory shared by several processors installed on the motherboard. Hence within a node, it is a shared memory system, whereas across the nodes the cluster is a distributed memory system.

Parallel programming models. For shared-memory systems, programming models based on *threads* are most popular. A thread is a stream of machine language instructions that can be created and run in parallel during the execution of a single program. OpenMP is a widely used extension to the C and Fortran programming languages based on threads. It achieves data parallelism by letting the compiler know what part of the sequential program is parallelizable by creating multiple threads. Simply put, each processor core can run a thread operating on a different partition of the data. In distributed-memory systems, parallelism is difficult to achieve via a simple modification of sequential code. The programmer needs to coordinate communications between processors not sharing memory. A de facto standard for such processor-to-processor communication is the message passing interface (MPI). MPI routines mainly consist of *point-to-point communication calls* that send and receive data between two processors, and *collective communication calls* that all processors in a group participate in. Typical collective communication calls include

- Scatter: one processor has a data array, and each other processor receives a partition of the array;
- Gather: one processor collects data from all the processors to construct an array;
- Broadcast: one processor sends its data to all other devices;
- Reduce: gather data and produce a combined output on a root process based on an associative binary operator, such as sum or maximum of all the elements.

There are also all-gather and all-reduce, where the output is shared by all processors. At a higher abstraction level, MapReduce (Dean and Ghemawat, 2008), a functional programming model in which a “map” function transforms each datum into a key-value pair, and a “reduce” function aggregates the results, is a popular distributed data processing model. While basic implementations are provided in base R, both the map and reduce operations are easy to parallelize. Distributed implementations such as Hadoop (Apache Software Foundation, 2021) handle communications between nodes implicitly. This programming model is inherently one-pass and stateless, and iterations on Hadoop require frequent accesses to external storage (hard disks), hence slow. Apache Spark (Zaharia et al., 2010) is an implementation that substitutes external storage with memory caching, yet iterative algorithms are an order of magnitude slower than their MPI counterparts (Jha et al., 2014; Reyes-Ortiz et al., 2015; Gittens et al., 2016).

Parallel architectures. To realize the above models, a computer architecture that allows simultaneous execution of multiple machine language instructions is needed. Single instruction, multiple data (SIMD) architecture has multiple processors that execute the same instruction on different parts of the data. The GPU falls into this category of architectures, as its massive number of cores can run a large number of threads sharing memory. Multiple instruction, multiple data (MIMD), or single program, multiple data (SPMD) architecture has multiple CPUs that execute independent parts of program instructions on their own data partition. Most computer clusters fall into this category.

2.2 Multiple CPU nodes: clusters, supercomputers, and clouds

Computing on multiple nodes can be utilized in many different scales. For mid-sized data, one may build his/her own cluster with a few nodes. This requires determining the topology and purchasing all the required hardware, along with resources to maintain it. This is certainly not an expertise of virtually all statisticians. Another option may be using a well-maintained supercomputer in a nearby HPC center. A user can take advantage of the facility with up to hundreds of thousand cores. The computing jobs on these facilities are often controlled by a job scheduler, such as Sun Grid Engine (Gentzsch, 2001), Slurm (Yoo et al., 2003), and Torque (Staples, 2006). However, access to supercomputers is almost always limited. Even when the user has access to them, he/she often has to wait in a very long queue until the requested computation job is started by the scheduler.

In recent years, cloud computing, which refers to both the applications delivered as services over the Internet, and the hardware and systems software in the data centers that provide these services (Armbrust et al., 2010), has emerged as a third option. Information technology giants such as Amazon, Microsoft, and Google lend their practically infinite computing resources to users on demand by wrapping the resources as "virtual machines," which are charged per CPU hours and storage. Users basically pay utility bills for their use of computing resources. An important implication of this infrastructure to end-users is that the cost of using 1000 virtual machines for one hour is almost the same as using a single virtual machine for 1000 hours. Therefore a user can build his/her own virtual cluster "on the fly," increasing the size of the cluster as the size of the problem to solve grows. A catch here is that a cluster does not necessarily possess the power of HPC as suggested in Section 2.1: a requirement for high performance is that all the machines should run in tight lockstep when working on a problem (Fox, 2011). However, early cloud services were more focused on web applications that do not involve frequent data transmissions between computing instances, and less optimized for HPC, yielding discouraging results (Evangelinos and Hill, 2008; Walker, 2008). For instance, "serverless computing" services such as AWS Lambda, Google Cloud Functions, and Azure Functions allow users to run a function on a large amount of data, in much the same fashion as supplying it to `lapply()` in base R. These services offer reasonable scalability on a simple map-reduce-type jobs such as image featurization, word count, and sorting. Nevertheless, their restrictions on resources (e.g., single core and 300 seconds of runtime in AWS Lambda) and the statelessness of the functional programming approach result in high latency for iterative algorithms, such as consensus ADMM (Aytekin and Johansson, 2019).

Eventually, many improvements have been made at hardware and software levels to make HPC on clouds feasible. At hardware level, cloud service providers now support CPU instances such as `c4`, `c5`, and `c5n` instances of Amazon Web Services (AWS), with up to 48 physical cores of higher clock speed of up to 3.4 GHz along with support for accelerated SIMD computation. If network bandwidth is critical, the user may choose instances with faster networking (such as `c5n` instances in AWS), allowing up to 100 Gbps of network bandwidth. At the software level, these providers support tools that manage resources efficiently for scientific computing applications, such as ParallelCluster (Amazon Web Services, 2021) and ElastiCluster (University of Zurich, 2021). These tools are designed to run programs in clouds in a similar manner to proprietary clusters through a job scheduler. In contrast to a physical cluster in an HPC center, a virtual cluster on a cloud is exclusively created for the user; there is no need for waiting in a long queue. Consequently, over 10 percent of all HPC jobs are running in clouds, and over 70 percent of HPC centers run some jobs in a cloud as of June 2019; the latter is up from just 13 percent in 2011 (Hyperion Research, 2019).

In short, cloud computing is now a cost-effective option for statisticians who demand high performance, without a steep learning curve.

2.3 Multi-GPU node

In some cases, HPC is achieved by installing multiple GPUs on a single node. A key feature of GPUs is their ability to apply a mapping to a large array of floating-point numbers simultaneously. The mapping (called a *kernel*) can be programmed by the user. This feature is enabled by integrating a massive number of simple compute cores in a single processor chip, forming a SIMD architecture. While this architecture of GPUs was created for high-quality video games to generate a large number of pixels in a hard time limit, the programmability and high throughput soon gained attention from the scientific computing community. Matrix-vector multiplication and elementwise nonlinear transformation of a vector can be computed several orders of magnitude faster on GPU than on CPU. Early applications of general-purpose GPU programming include physics simulations, signal processing, and geometric computing (Owens et al., 2007). Technologically savvy statisticians demonstrated its potential in Bayesian simulation (Suchard, Holmes and West, 2010; Suchard, Wang, Chan, Frelinger, Cron and West, 2010) and high-dimensional optimization (Zhou et al., 2010; Yu et al., 2015). Over time, the number of cores has increased from 240 (Nvidia GTX 285, early 2009) to 4608 (Nvidia Titan RTX, late 2018) and more local memory — separated from CPU's main memory — has been added (from 1GB of GTX 285 to 24GB for Titan RTX). GPUs could only use single-precision for their floating-point operations, but they now support double- and half-precisions. More sophisticated operations such as tensor multiplication are also supported. High-end GPUs are now being designed specifically for scientific computing purposes, sometimes with fault-tolerance features such as error correction.

Major drawbacks of GPUs are smaller memory size, compared to CPU, and data transfer overhead between CPU and GPU. These limitations can be addressed by using multiple GPUs: recent GPUs can be installed on a single node and communicate with each other without the meddling of CPU; this effectively increases the local memory of a collection of GPUs.[2] It is relatively inexpensive to construct a node with 4–8 desktop GPUs compared to a cluster of CPU nodes with a similar computing power (if the main computing tasks are well suited for the SIMD model), and the gain is much larger than the cost. A good example would be linear algebra operations that frequently occur in high-dimensional

[2] Lee et al. (2017) explored this possibility in image-based regression.

optimization.

Programming environments for GPU computing have been notoriously hostile to programmers for a long time. The major hurdle is that a programmer needs to write two suits of code, the *host* code that runs on a CPU and *kernel* functions that run on GPU cores. Data transfer between CPU and GPU(s) also has to be taken care of. Moreover, kernel functions need to be written in special extensions of C, C++, or Fortran, e.g., the Compute Unified Device Architecture (CUDA, Kirk, 2007) or Open Computing Language (OpenCL, Munshi, 2009). Combinations of these technical barriers prevented casual programmers, especially statisticians, from writing GPU code despite its computational gains. There were efforts to sugar-coat these hostile environments with a high-level language such as R (Buckner et al., 2009) or Python (Tieleman, 2010; Klöckner et al., 2012; Lam et al., 2015), but these attempts struggled to garner large enough user base since the functionalities were often limited and inherently hard to extend.

Fortunately, GPU programming environments have been revolutionized since deep learning (LeCun et al., 2015) brought sensation to many machine learning applications. Deep learning is almost synonymous to deep neural networks, which refer to a repeated ("layered") application of an affine transformation of the input followed by identical elementwise transformations through a nonlinear link function, or "activation function." Fitting a deep learning model is almost always conducted via (approximate) minimization of the specified loss function through a clever application of the chain rule to the gradient descent method, called "backpropagation" (Rumelhart et al., 1986). These computational features fit well to the SIMD architecture of GPUs, use of which dramatically reduces the training time of this highly overparameterized family of models with a huge amount of training data (Raina et al., 2009). Consequently, many efforts have been made to ease GPU programming for deep learning, resulting in easy-to-use software libraries. Since the sizes of neural networks get ever larger, more HPC capabilities, e.g., support for multiple GPUs and CPU clusters, have been developed. As we review in the next section, programming with those libraries gets rid of many hassles with GPUs, close to the level of conventional programming.

3. EASY-TO-USE SOFTWARE LIBRARIES FOR HPC

3.1 Deep learning libraries and HPC

As of revising this article (summer 2020), the two most popular deep learning software libraries are TensorFlow (Abadi et al., 2016) and PyTorch (Paszke et al., 2019). There are two common features of these libraries. One is the computation graph that automates the evaluation of the loss function and its differentiation required for backpropagation. The other feature, more relevant to statistical computing, is an efficient and user-friendly interface to linear algebra and convolution routines that work on both CPU and GPU in a unified fashion. A typical pattern of using these libraries is to specify the model and describe how to fit the model to the training data in a high-level scripting language (mostly Python). The system on which the model is fitted can be programmed. If the target system is a CPU node, then the software can be configured to utilize the OpenBLAS (Zhang et al., 2021) or Intel Math Kernel Library (Wang et al., 2014), which are optimized implementations of the Basic Linear Algebra Library (BLAS, Blackford et al., 2002) for shared-memory systems. If the target system is a workstation with a GPU, then the same script can employ a pair of host and kernel code that may make use of cuBLAS (NVIDIA, 2021*a*), a GPU version of BLAS, and cuSPARSE (NVIDIA, 2021*b*), GPU-oriented sparse linear algebra routines. A slight change in the option for device selection — usually a line or two in the script — can control whether to run the model on a CPU or GPU. From the last paragraph of the previous section, we see that this "write once, run everywhere" feature of deep learning libraries can make GPU programming easier for statistical computing as well.

TensorFlow is a successor of Theano (Theano Development Team, 2016), one of the first libraries to support automatic differentiation based on computational graphs. Unlike Theano, which generates GPU code on the fly, TensorFlow is equipped with pre-compiled GPU code for a large class of pre-defined operations. PyTorch inherits Torch (Collobert et al., 2011), an early machine learning library written in a functional programming language called Lua, and Caffe (Jia et al., 2014), a Python-based deep learning library. PyTorch (and Torch) can also manage GPU memory efficiently. As a result, it is known to be faster than other deep learning libraries (Bahrampour et al., 2016).

Both libraries support multi-GPU and multi-node computing.[3] In TensorFlow, multi-GPU computation is supported natively on a single node. If data are distributed in multiple GPUs and one needs data from the other, the GPUs communicate with each other implicitly and the user does not need to interfere. For multi-node communication, it is recommended to use MPI through the library called Horovod (Sergeev and Del Balso, 2018) for tightly-coupled HPC environments. In PyTorch, both multi-GPU and multi-node computing are enabled by using the interface `torch.distributed`. This interface defines MPI-style (but simplified) communication primitives (see Section 2.1). Implementations include the *bona fide* MPI, Nvidia Collective Communications Library (NCCL), and Gloo (Facebook Incubator, 2021). Recent MPI implementations can map multi-GPU communication to the MPI standard as well as traditional multi-node communication. While NCCL is useful for a multi-GPU node, Gloo is useful with multiple CPU with Ethernet interconnect.

3.2 Automatic differentiation

The automatic differentiation (AD) feature of deep learning software deserves separate attention. AD refers to a collection of techniques that evaluate the derivatives of a function specified by a computer program accurately (Griewank and Walther, 2008; Baydin et al., 2017). Based on AD, complex deep models can be trained with stochastic approximation (see the next section) on huge data within a hundreds of lines of code and approximate a rich class of functions efficiently; see Schmidt-Hieber et al. (2020); Bauer et al. (2019); Imaizumi and Fukumizu (2019); Suzuki (2019); Ohn and Kim (2019) for recent theoretical developments. Most AD techniques rely on decomposition of the target function into elementary functions (primitives) whose derivatives are known, and the computational graph, either explicitly or implicitly, that describes the dependency among

[3] There are other deep learning software libraries with similar HPC supports: Apache MxNet (Chen et al., 2015) supports multi-node computation via Horovod; multi-GPU computing is also supported at the interface level. Microsoft Cognitive Toolkit (CNTK, Seide and Agarwal, 2016) supports parallel stochastic gradient algorithms through MPI.

the primitives. Fig. 1 illustrates the computational graph for the bivariate function $f(x_1, x_2) = \log(x_1 + x_2) - x_2^2$. The internal nodes represent intermediate variables corresponding to the primitives: $z_{-1} = x_1$, $z_0 = x_2$, $z_1 = z_{-1} + z_0$, $z_2 = \log z_1$, $z_3 = z_0^2$, and $z_4 = z_2 - z_3$; $y = z_4$.

There are two modes of AD, depending on the order of applying the chain rule. Forward-mode AD applies the rule from right to left (or from input to output), hence it is straightforward to implement. In Fig. 1, if we want to evaluate the partial derivative $\frac{\partial f}{\partial x_2}$ at $(3, 2)$, then by denoting $\dot{z}_i \equiv \frac{\partial z_i}{\partial x_2}$ we see that $\dot{z}_{-1} = \dot{x}_1 = 0$, $\dot{z}_0 = \dot{x}_2 = 1$, $\dot{z}_1 = \dot{z}_0 + \dot{z}_1 = 1$, $\dot{z}_2 = \dot{z}_1/z_1 = 1/5$, $\dot{z}_3 = 2z_0\dot{z}_0 = (2)(2)(1) = 4$, $\dot{z}_4 = \dot{z}_2 - \dot{z}_3 = 1/5 - 4$, and finally $\dot{y} = \dot{z}_4 = -3.8$. While this computation can be conducted in a single pass with evaluation of the original function f, computing another derivative $\frac{\partial f}{\partial x_1}$ requires a separate pass. Thus, forward mode is inefficient if the whole gradient of a function with many input variables is needed, e.g., the loss function of a high-dimensional model. Reverse-mode AD applies the chain rule in the opposite direction. In the first pass, the original function and the associated intermediate variables z_i are evaluated from input to output. In the second pass, the "adjoint" variables $\bar{z}_i \equiv \frac{\partial y}{\partial z_i}$ are initialized to zero and updated from output to input. In Fig. 1, $\bar{z}_4 \mathrel{+}= \frac{\partial y}{\partial z_4} = 1$, $\bar{z}_3 \mathrel{+}= \bar{z}_4 \frac{\partial z_4}{\partial z_3} = -1$, $\bar{z}_2 \mathrel{+}= \bar{z}_4 \frac{\partial z_4}{\partial z_2} = 1$, $\bar{z}_0 \mathrel{+}= \bar{z}_3 \frac{\partial z_3}{\partial z_0} = \bar{z}_3(2z_0) = -4$, $\bar{z}_1 \mathrel{+}= \bar{z}_2 \frac{\partial z_2}{\partial z_1} = \frac{\bar{z}_2}{z_2} = 1/5$, $\bar{z}_0 \mathrel{+}= \bar{z}_1 \frac{\partial z_1}{\partial z_0} = 1/5$, and $\bar{z}_{-1} \mathrel{+}= \bar{z}_1 \frac{\partial z_1}{\partial z_{-1}} = 1/5$. Here, the '+=' is the C-style increment operator, employed in order to observe the rule of total derivatives. (Note $\bar{z}_0$ is updated twice.) Finally, $\frac{\partial f}{\partial x_1} = \bar{x}_1 = \bar{z}_{-1} = 0.2$ and $\frac{\partial f}{\partial x_2} = \bar{x}_2 = \bar{z}_0 = -3.8$. Hence reverse-mode AD generalizes the backpropagation algorithm and computes the whole gradient ∇f in a single backward pass, at the expense of keeping intermediate variables.

Deep learning software can be categorized by the way they build computational graphs. In Theano and TensorFlow, the user needs to construct a *static* computational graph using a specialized mini-language before executing the model fitting process, and the graph cannot be modified throughout the execution. This static approach has performance advantage since there is room for optimizing the graph structure. Its disadvantage is the limited expressiveness of computational graphs and AD. On the other hand, PyTorch employs *dynamic* computational graphs, for which the user describes the model as a regular program for (forward) evaluation of the loss function. Intermediate values and computation trace are recorded in the forward pass, and the gradient is computed by parsing the recorded computation backwards. The advantage of this dynamic graph construction is the expressiveness of the model: in particular, recursion is allowed in the loss function definition. For example, recursive models such as $f(x) = f(x/2)$ if $x > 1$ and x otherwise are difficult to describe using a static graph but easy with a dynamic one. The downside is slower evaluation due to function call overheads.

3.3 Case study: PyTorch versus TensorFlow

In this section, we illustrate how simple it is to write a statistical computing code on multi-device HPC environments using modern deep learning libraries. We compare PyTorch and TensorFlow code written in Python, which computes a Monte Carlo estimate of the constant π. The emphasis is on readability and flexibility, i.e., how small a modification is needed to run the code written for a single-CPU node on a multi-GPU node and a multi-node system.

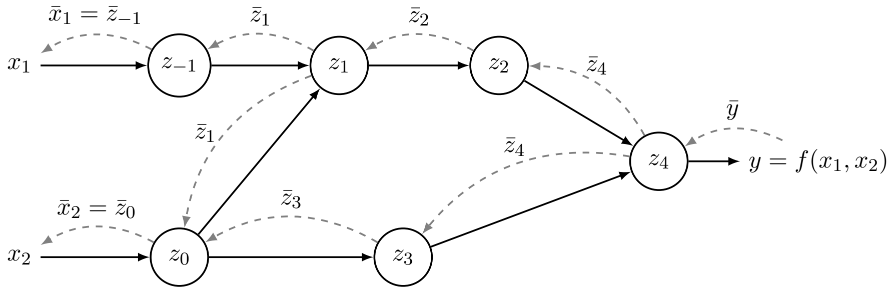

Fig 1: Computational graph for evaluating function $f(x_1, x_2) = \log(x_1 + x_2) - x_2^2$. Dashed arrows indicate the direction of backpropagation evaluating $\nabla f(x_1, x_2)$.

Listing 1 shows the code for Monte Carlo estimation of π using PyTorch. Even for those who are not familiar with Python, the code should be quite readable. The main workhorse is function `mc_pi()` (Lines 14–21), which generates a sample of size n from the uniform distribution on the unit square $[0, 1]^2$ and compute the proportion of the points that fall inside the quarter circle of unit radius centered at the origin. Listing 1 is a fully executable program. It uses `torch.distributed` interface with an MPI backend (Line 3). An instance of the program of Listing 1 is attached to a device and is executed as a "process". Each process is given its identifier (rank), which is retrieved in Line 6. The total number of processes is known to each process via Line 7. After the proportion of the points in the quarter-circle is computed in Line 22, each process gathers the sum of the means computed from all the processes in Line 25 (this is called the all-reduce operation; see Section 2.1). Line 27 divides the sum by the number of processes, yielding a Monte Carlo estimate of π based on the sample size of $n \times$ (number of processes).

We have been deliberately ambiguous about the "devices." Here, a CPU core or a GPU is referred to as a device. Listing 1 assumes the environment is a workstation with one or more GPUs, and the backend MPI is CUDA-aware. A CUDA-aware MPI, e.g., OpenMPI (Gabriel et al., 2004), allows data to be sent directly from a GPU to another GPU through the MPI protocols. Data transfer between modern GPUs does not go through CPU (Lee et al., 2017). Lines 9 –10 specify that the devices to use in the program are GPUs. For example, suppose the workstation has four GPUs, say device 0 through 3. A likely scenario for carrying out the all-reduce operation in Line 25 is to transfer the estimated π in device 1 (computed in Line 22, which is parallelized) to device 0, where the two estimates are added. At the same time, the estimate in device 3 is passed to device 2 and then added with another estimate there. After this step, the sum in device 2 is sent to device 0 to compute the final sum. This sum is broadcast to all the other devices to replace the local estimates. (The actual behavior may be slightly different from this scenario depending on the specific implementation of MPI.) If the environment is a cluster with multiple CPU nodes (or even a single node), then communication between nodes or CPU cores through high-speed interconnect replaces the inter-GPU communication. At the code level, all we need to do is change Line 10 to `device = 'cpu'`. The resulting code runs on a cluster seamlessly as long as the MPI for the cluster is properly installed.

In TensorFlow, however, a separate treatment of multi-GPU and cluster settings is almost necessary. The code for multi-GPU setting is similar to Listing 1

```
# import packages
import torch.distributed as dist
import torch
dist.init_process_group(’mpi’)  # initialize MPI

rank = dist.get_rank()          # device id
size = dist.get_world_size()    # total number of devices

# select device
device = ’cuda:{}’.format(rank) # or simply ’cpu’ for CPU computing
# select GPU based on rank.
if device.startswith(’cuda’): torch.cuda.set_device(rank)

def mc_pi(n):
    # this code is executed on each device.
    # generate n samples from Unif(0, 1) for x and y
    x = torch.rand((n), dtype=torch.float64, device=device)
    y = torch.rand((n), dtype=torch.float64, device=device)
    # compute local estimate of pi in float64.
    # type conversion is necessary, because (x ** 2 + y ** 2 < 1)
    # results in unsigned 8-bit integer.
    pi_hat = torch.mean((x**2 + y**2 <1).to(dtype=torch.float64))*4
    # sum of the estimates across processes
    #  is stored in-place in ’pi_hat’, overwriting its original value.
    dist.all_reduce(pi_hat)
    # the final estimate of pi, computed on each process
    return pi_hat / size

if __name__ == ’__main__’:
    n = 10000
    pi_hat = mc_pi(n)
    print("Pi estimate based on {} Monte Carlo samples across {}
        processes.".format(n * size, size))
    if rank == 0:
        print(pi_hat.item())
```

Listing 1: Distributed Monte Carlo estimation of π using PyTorch

and is given in Appendix C. In a cluster setting, unfortunately, it is extremely difficult to reuse the multi-GPU code. If direct access to individual compute nodes is available, that information can be used to run the code distributedly, albeit not intuitively. However, in HPC environments where computing jobs are managed by job schedulers, we often do not have direct access to the compute nodes. The National Energy Research Scientific Computing Center (NERSC), the home of the 16th most powerful supercomputers in the world (as of June 2020), advises that gRPC, the default inter-node communication method of TensorFlow, is very slow on tightly-coupled nodes, thus recommends a direct use of MPI (NERSC, 2021). Using MPI with TensorFlow requires an external library called Horovod and a substantial modification of the code, as shown in Listing 2. This is a sharp contrast to Listing 1, where essentially the same PyTorch code can be used in both multi-GPU and multi-node settings.

Due to the reasons stated in Section 3.2, we employ PyTorch in the sequel to implement the highly parallelizable algorithms of Section 4 in a multi-GPU node and a cluster on a cloud, as it allows simpler code that runs on various HPC environments with a minimal modification. (In fact, this modification can be made automatic through a command line argument.)

```
import tensorflow as tf
import horovod.tensorflow as hvd

# initialize horovod
hvd.init()
rank = hvd.rank()

# without this block, all the processes try to allocate
# all the memory from each device, causing out of memory error.
devices = tf.config.experimental.list_physical_devices("GPU")
if len(devices) > 0:
    for d in devices:
        tf.config.experimental.set_memory_growth(d, True)

# select device
tf.device("device:gpu:{}".format(rank)) # tf.device("device:cpu:0") for CPU

# function runs in parallel with (graph computation/lazy-evaluation)
# or without (eager execution) the line below
@tf.function
def mc_pi(n):
    # this code is executed on each device
    x = tf.random.uniform((n,), dtype=tf.float64)
    y = tf.random.uniform((n,), dtype=tf.float64)
    # compute local estimate for pi and save it as 'estim'.
    estim = tf.reduce_mean(tf.cast(x**2 + y ** 2 <1, tf.float64))*4
    # compute the mean of 'estim' over all the devices
    estim = hvd.allreduce(estim)
    return estim

if __name__ == '__main__':
    n = 10000
    estim = mc_pi(n)
    # print the result on rank zero
    if rank == 0:
        print(estim.numpy())
```

Listing 2: Monte Carlo estimation of π for TensorFlow on multiple nodes using Horovod

4. HIGHLY PARALLELIZABLE ALGORITHMS

In this section, we discuss some easily parallelizable optimization algorithms useful for fitting high-dimensional statistical models, assuming that data are so large that they have to be stored distributedly. These algorithms can benefit from the distributed-memory environment by using relatively straightforward operations, via distributed matrix-vector multiplication and independent update of variables.

4.1 MM algorithms

The MM principle (Lange et al., 2000; Lange, 2016), where "MM" stands for either majorization-minimization or minorization-maximization, is a useful tool for constructing parallelizable optimization algorithms. In minimizing an objective function $f(x)$ iteratively, for each iterate we consider a surrogate function $g(x|x^n)$ satisfying two conditions: the tangency condition $f(x^n) = g(x^n|x^n)$ and the domination condition $f(x) \leq g(x|x^n)$ for all x. Updating $x^{n+1} = \arg\min_x g(x|x^n)$ guarantees that $\{f(x^n)\}$ is a nonincreasing sequence:

$$f(x^{n+1}) \leq g(x^{n+1}|x^n) \leq g(x^n|x^n) = f(x^n).$$

In fact, full minimization of $g(x|x^n)$ is not necessary for the descent property to hold; merely decreasing it is sufficient. For instance, it can be shown that the EM algorithm (Dempster et al., 1977) is obtained by applying the MM principle to to the observed-data log likelihood and Jensen's inequality. (See Wu and Lange (2010) for more details about the relation between MM and EM.)

MM updates are usually designed to make a nondifferentiable objective function smooth, linearize the problem, or avoid matrix inversions by a proper choice of a surrogate function. MM is naturally well-suited for parallel computing environments, as we can choose a separable surrogate function and update variables independently. For example, when maximizing loglikelihoods, a term involving summation inside the logarithm $\log(\sum_{i=1}^p u_i)$, $u_i > 0$, often arises. By using Jensen's inequality, this term can be minorized and separated as

$$\log\left(\sum_{i=1}^p u_i\right) \geq \sum_{i=1}^p \frac{u_i^n}{\sum_{j=1}^p u_j^n} \log\left(\frac{\sum_{j=1}^p u_j^n}{u_i^n} u_i\right) = \sum_{i=1}^p \left(\frac{u_i^n}{\sum_{j=1}^p u_j^n}\right) \log u_i + c_n,$$

where u_i^n's are constants and c_n is a constant only depending on u_i^n's. Parallelization of MM algorithms on a single GPU using separable surrogate functions is extensively discussed in Zhou et al. (2010). Separable surrogate functions are especially important in distributed HPC environments, e.g. multi-GPU systems.

4.2 Proximal gradient method

The proximal gradient method extends the gradient descent method, and deals with minimization of sum of two extended real-valued convex functions, i.e.,

$$\min_x f(x) + g(x). \tag{1}$$

Function f is possibly nondifferentiable, while g is continuously differentiable.

We first define the proximity operator of f:

$$\mathbf{prox}_{\lambda f}(y) = \arg\min_x \left\{ f(x) + \tfrac{1}{2\lambda}\|x-y\|_2^2 \right\}, \quad \lambda > 0$$

For many functions their proximity operators take closed forms. We call such functions "proximable". For example, consider the $0/\infty$ indicator function $\delta_C(x)$ of a closed convex set C, i.e., $\delta_C(x) = 0$ if $x \in C$, and $+\infty$ otherwise. The corresponding proximity operator is the Euclidean projection onto C: $P_C(y) = \arg\min_{x\in C}\|y-x\|_2$. For many sets, e.g., nonnegative orthant, P_C is simple to compute. Also note that the proximity operator of the ℓ_1-norm $\lambda\|\cdot\|_1$ is the soft-thresholding operator: $[\mathcal{S}_\lambda(y)]_i := \text{sign}(y_i)(|y_i| - \lambda)_+$.

Now we proceed with the proximal gradient method for minimization of $h(x) = f(x) + g(x)$. Assume g is convex and has an L-Lipschitz gradient, i.e., $\|\nabla g(x) - \nabla g(y)\|_2 \leq L\|x - y\|_2$ for all x, y in the interior of its domain, and f is lower-semicontinuous, convex, and proximable. The L-Lipschitz gradients naturally result in the following surrogate function that majorizes h:

$$\begin{aligned} h(x) &\leq f(x) + g(x^n) + \langle \nabla g(x^n), x - x^n \rangle + \tfrac{L}{2}\|x - x^n\|_2^2 \\ &= f(x) + g(x^n) + \tfrac{L}{2}\left\|x - x^n + \tfrac{1}{L}\nabla g(x^n)\right\|_2^2 - \tfrac{1}{2L}\left\|\nabla g(x^n)\right\|_2^2 =: p(x|x^n). \end{aligned}$$

Minimizing $p(x|x^n)$ with respect to x results in the iteration:

$$x^{n+1} = \mathbf{prox}_{\gamma_n f}\left(x^n - \gamma_n \nabla g(x^n)\right), \quad \gamma_n \in (0, 1/L]\,. \tag{2}$$

If $f \equiv 0$, then iteration (2) reduces to the conventional gradient descent. This iteration guarantees a nonincreasing sequence of $h(x^n)$ by the MM principle. Proximal gradient method also has an interpretation of forward-backward operator splitting, and the step size $\gamma_n \in (0, 2/L)$ guarantees convergence (Combettes and Pesquet, 2011; Combettes, 2018). If $f(x) = \delta_C(x)$, then the corresponding algorithm is called the projected gradient method. If $f(x) = \lambda\|x\|_1$, then it is the iterative shrinkage-thresholding algorithm (ISTA, Beck and Teboulle, 2009).

For many functions f, the update (2) is simple and easily parallelized, thus the algorithm is suitable for HPC. For example, in the soft-thresholding operator above all the elements are independent. If $f(x) = -a\log x$, then

$$\mathbf{prox}_{\gamma f}(y) = \left(y + \sqrt{y^2 + 4\gamma a}\right)/2, \tag{3}$$

which is useful for the PET example in Section 6. The gradient ∇g in update (2) can also be computed in parallel. In many models the fitting problem takes the form of (1) with $g(x) = \frac{1}{m}\sum_{i=1}^m \ell(a_i^T x)$, where ℓ is a loss function and $a_i \in \mathbb{R}^p$ is the ith observation. Collect the latter into a data matrix $A \in \mathbb{R}^{m\times p}$. If $m \gg p$, then split it by the row as $A = [A_{[1]}^T, A_{[2]}^T, \cdots, A_{[d]}^T]^T$, where blocks $A_{[k]}$ are distributed over d devices. If the current iterate of the parameter x^n is known to each device, then the local gradient $\nabla g_i(x^n) = \ell'(a_i^T x)a_i$ can be computed from $A_{[k]}$ independently. The full gradient $\nabla g(x^n)$ can be computed then by averaging $\nabla g_i(x^n)$. In the MPI terminology of Section 2.1, a distributed-memory proximal gradient update consists of the following steps: 1) broadcast x^n; 2) compute the local gradient $\nabla g_i(x^n)$ in each device; 3) reduce the local gradients to compute the full gradient $\nabla g(x^n)$ in the master device; 4) update x^{n+1}. If g is not separable in observations, splitting the data matrix by column may be useful (Section 6.3).

See Parikh and Boyd (2014) for a thorough review and distributed-memory implementations, and Polson et al. (2015) for a statistically oriented review.

4.3 Primal-dual methods

Primal-dual methods introduce an additional dual variable y (where x is the primal variable) in order to deal with a larger class of problems. Consider the problems of the form $h(x) = f(Kx) + g(x)$, where $K \in \mathbb{R}^{l\times p}$. We further assume that f and g are lower semicontinuous, convex, and proper (i.e., not always ∞)

functions. Even if f is proximable, the proximity operator for $f(K\cdot)$ is not easy to compute. The conjugate of f is defined as $f^*(y) = \sup_x \langle x, y\rangle - f(x)$. It is known that $f^{**} = f$, so $f(Kx) = f^{**}(Kx) = \sup_y \langle Kx, y\rangle - f^*(y)$. Then the minimization problem $\inf_x f(Kx) + g(x)$ is equivalent to the saddle-point problem

$$\inf_x \sup_y \langle Kx, y\rangle + g(x) - f^*(y),$$

for which a solution $(\hat{x}, \hat{y})$ exists under mild conditions.

A widely known method for solving this saddle-point problem in the statistical literature is the ADMM (Xue et al., 2012; Ramdas and Tibshirani, 2016; Zhu, 2017; Lee et al., 2017; Gu et al., 2018), whose update is given by:

$$x^{n+1} = \arg\min_x g(x) + (t/2)\|Kx - \tilde{x}^n + (1/t)y^n\|_2^2 \tag{4a}$$

$$\tilde{x}^{n+1} = \mathbf{prox}_{(1/t)f}(Kx^{n+1} + (1/t)y^n) \tag{4b}$$

$$y^{n+1} = y^n + t(Kx^{n+1} - \tilde{x}^{n+1}). \tag{4c}$$

If g is separable, i.e., $g(x) = \sum_{k=1}^d g_k(x)$, then consensus optimization (Boyd et al., 2011, Chap. 7) applies ADMM to distributed copies of variables $x_k = x$ to minimize $h(x) = f(z) + \sum_{k=1}^d g_k(x_k)$ subject to $x_k = x$ and $Kx_k = z$ for each k:

$$x_k^{n+1} = \arg\min_{x_k} g_k(x_k) + \tfrac{t}{2}\|Kx_k - \tilde{x}^n + \tfrac{1}{t}y_k^n\|_2^2 + \tfrac{t}{2}\|x_k - x^n + \tfrac{1}{t}w_k^n\|_2^2 \tag{5a}$$

$$\tilde{x}^{n+1} = \mathbf{prox}_{(dt)^{-1}f}\big(\tfrac{1}{d}\textstyle\sum_{k=1}^d(Kx_k^{n+1} + \tfrac{1}{t}y_k^n)\big) \tag{5b}$$

$$y_k^{n+1} = y_k^n + t(Kx_k^{n+1} - \tilde{x}^{n+1}), \quad w_k^{n+1} = w_k^n + t(x_k^{n+1} - x^{n+1}). \tag{5c}$$

A distributed-memory implementation will iterate the following steps: 1) for each device k, solve (5a) in parallel; 2) gather local solutions x_k^n in the master device; 3) compute (5b); 4) broadcast $\tilde{x}^{n+1}$; 5) compute (5c).

Nonetheless, neither update (4a) nor (5a) results in a proximity operator, since the quadratic term is not spherical. This inner optimization problem is often nontrivial to solve. In the simplest case of linear regression, g is quadratic and (4a) involves solving a (large) linear system whose time complexity is cubic in the dimension p of the primal variable x.

PDHG avoids inner optimization via the following iteration:

$$y^{n+1} = \mathbf{prox}_{\sigma f^*}(y^n + \sigma K\bar{x}^n) \tag{6a}$$

$$x^{n+1} = \mathbf{prox}_{\tau g}(x^n - \tau K^T y^{n+1}) \tag{6b}$$

$$\bar{x}^{n+1} = 2x^{n+1} - x^n, \tag{6c}$$

where σ and τ are step sizes. If f is proximable, so is f^*, since $\mathbf{prox}_{\gamma f^*}(x) = x - \gamma\mathbf{prox}_{\gamma^{-1}f}(\gamma^{-1}x)$ by Moreau's decomposition. This method has been analyzed using monotone operator theory (Condat, 2013; Vũ, 2013; Ko et al., 2019). Convergence of iteration (6) is guaranteed if $\sigma\tau\|K\|_2^2 < 1$, where $\|M\|_2$ is the spectral norm of matrix M. If g has an L-Lipschitz gradient, then the proximal step (6b) can be replaced by a gradient step

$$x^{n+1} = x^n - \tau(\nabla g(x^n) + K^T y^{n+1}).$$

PDHG algorithms are also highly parallelizable as long as the involved proximity operators are easy to compute and separable. No inner optimization is involved in iteration (6) and only matrix-vector multiplications appear. The distributed computation of gradient in Section 4.2 can be used for the gradient step. A hybrid of PDHG and ADMM has recently been proposed (Ryu et al., 2020).

4.4 Parallel coordinate descent and stochastic approximation

Coordinate descent methods apply vector-to-scalar maps $T_i : \mathbb{R}^p \to \mathbb{R} : x = (x_1, \ldots, x_i, \ldots, x_p) \mapsto \arg\min_{x_i'} h(x_1, \ldots, x_i', \ldots, x_p)$ defined for each coordinate i successively to minimize $h(x)$. The most well-known variant is the cyclic or Gauss-Seidel version. If we denote the jth elementary unit vector in $\mathbb{R}^p$ by e_j, then the update rule is $x^{n+1} = \sum_{j \neq i} x_j^n e_j + T_i(x)e_i$ where $i = (n-1 \mod p)+1$, which possesses the descent property. The parallel or Jacobi update reads $x^{n+1} = \sum_{j=1}^p T_j(x)e_j$. Obviously, if h is separable in variables, i.e., $h(x) = \sum_{j=1}^p h_j(x_j)$, this minimization strategy will succeed. Other variants are also possible, such as randomizing the cyclic order, or updating a subset of coordinates in parallel at a time. The "argmin" map T_i can also be relaxed, e.g., by a prox-linear map $x \mapsto \arg\min_{x_i'} \langle \frac{\partial g}{\partial x_i}|_{x_i}, x_i' - x_i \rangle + \frac{1}{2\gamma_i}\|x_i' - x_i\|_2^2 + f(x)$ if h has a structure of $h = f + g$ and only g is differentiable (Tseng and Yun, 2009). See Wright (2015) for a recent review.

If p is much larger than τ, the number of devices, then choosing a subset of coordinates with size comparable to τ would reduce the complexity of an iteration. Richtárik and Takáč (2016*a*,*b*) consider sampling a random subset and study the effect of the sampling distribution on the performance of parallel prox-linear updates, deriving optimal distributions for certain cases. In particular, the gain of parallelization is roughly proportional to the degree of separability p/ω, where $\omega = \max_{J \in \mathcal{J}} |J|$ if $h(x) = \sum_{J \in \mathcal{J}} h_J(x)$ for a finite collection of nonempty subsets of $\{1, \ldots, p\}$ and h_J depends only on coordinates $i \in J$. For example, if $A = [a_1^T, \ldots, a_m^T]^T \in \mathbb{R}^{m \times p}$ is the data matrix for ordinary least squares, then ω equals to the maximum number of nonzero elements in the rows, or equivalently $\omega = \max_{i=1,\ldots,n} \|a_i\|_0$.

For gradient-descent type methods, stochastic approximation (Robbins and Monro, 1951, see Lai and Yuan (2021) for a recent review) has gained wide popularity under the name of *stochastic gradient descent* or SGD. The main idea is to replace the gradient of the expected loss by its unbiased estimator. For instance, as in the penultimate paragraph of Section 4.2, if $g(x) = \frac{1}{m}\sum_{i=1}^m \ell(a_i^T x)$, and $f \equiv 0$, then $\ell'(a_i^T x)a_i$ is an unbiased estimator of $\nabla g(x)$ under the uniform distribution on the sample indices $\{1, \ldots, m\}$. The update rule is then $x^{n+1} = x^n - \gamma_n \ell'(a_i^T x^n)a_i$ for some randomly chosen i. SGD and its variants (Defazio et al., 2014; Johnson and Zhang, 2013) are main training methods in most deep learning software, since the sample size m needs to be extremely large to properly train deep neural networks. The idea of using an unbiased estimator of the gradient has been extended to the proximal gradient (Nitanda, 2014; Xiao and Zhang, 2014; Atchadé et al., 2017; Rosasco et al., 2019) and PDHG (Chen et al., 2014; Ko et al., 2019; Ko and Won, 2019) methods. In practice, it is standard to use a minibatch or a random subset of the sample for each iteration, and the arbitrary sampling paradigm of Richtárik and Takáč (2016*a*,*b*) for parallel coordinate descent has been extended to minibatch SGD (Gower et al., 2019; Qian et al., 2019) and PDHG (Chambolle et al., 2018).

5. DISTRIBUTED MATRIX DATA STRUCTURE FOR PYTORCH

For the forthcoming examples and potential future uses in statistical computing, we propose the package `dist_stat` built on PyTorch. It consists of two submodules, `distmat` and `application`. The submodule `distmat` implements a simple distributed matrix data structure, and the submodule `application` includes the code for the examples in Section 6 using `distmat`. In the data structure `distmat`, each process, enumerated by its rank, holds a contiguous block of the full data matrix by rows or columns, which may be sparse. If multiple GPUs are involved, each process controls the GPU whose index matches the process rank. The blocks are assumed to have equal sizes. For notational simplicity, we indicate the dimension to split by a pair of square brackets: if a $[100] \times 100$ matrix is split over four processes, the rank-0 process keeps the first 25 rows of the matrix, the rank-1 process takes the next 25 rows, and so on. For the sake of simplicity, we always assume that the dimension to split is a multiple of the number of processes. The code for `dist_stat` is available at https://github.com/kose-y/dist_stat. A proper backend setup for a cloud environment is explained in Appendix B.

In `distmat`, unary elementwise operations such as exponentiation, square root, absolute value, and logarithm of matrix entries are implemented in an obvious manner. Binary elementwise operations such as addition, subtraction, multiplication, division are implemented in a similar manner to R's vector recycling: if two matrices of different dimensions are to be added together, say one is 3×4 and the other is 4×1, the latter matrix is expanded to a 3×4 matrix with the column repeated four times. Another example is adding a 1×4 matrix and a 4×1 matrix. The former is expanded to a 4×3 matrix by repeating the row four times, and the latter to a 4×3 matrix by repeating the column three times. Application of this recycling rule is in accordance with the broadcast semantics of PyTorch.

Distributed matrix multiplication requires some care. Suppose we multiply a $p \times r$ matrix A and an $r \times q$ matrix B. If matrix B is tall and split by *row* into $[B_{[1]}, \dots, B_{[T]}]^T$ and distributed among T processes, where $B_{[t]}$ is the t-th row block of B. If matrix A is split in the same manner, a natural way to compute the product AB is for each process t to *gather* (see Section 2.1) all $B_{[1]}, \dots, B_{[T]}$ to create a copy of B and compute the row block $A_{[t]}B$ of AB. On the other hand, if matrix A is wide and split by *column* into $[A^{[1]}, \dots, A^{[T]}]$, where $A^{[t]}$ is the t-th column block of A, then each process will compute the local multiplication $A^{[t]}B_{[t]}$. The product $AB = \sum_{t=1}^{T} A^{[t]}B_{[t]}$ is computed by a *reduce* or *all-reduce* operation of Section 2.1. These operations are parallelized as outlined in Section 3.3. The distribution scenarios considered in `distmat` are collected in Table 1. Each matrix can be either *broadcast* ($p \times r$ for A), row-distributed ($[p] \times r$), or column-distributed ($p \times [r]$). Since broadcasting both matrices does not require any distributed treatment in multiplication, there remain eight possible combinations of the input. For each combination, the output may involve more than one configurations. If an outer dimension (either p or q but not both) is distributed, the $p \times q$ output AB is distributed along that dimension (scenarios 4, 8, 11). If both dimensions are split, then there are two possibilities of $[p] \times q$ and $p \times [q]$ (scenarios 2, 3). Splitting of the inner dimension r does not affect the distribution of the output unless it is distributed in both A and B (scenarios

TABLE 1
Eleven distributed matrix multiplication scenarios of `distmat`*.*

	A	B	AB	Description	Communication involved (size of output)
1	$[p] \times r$	$[r] \times q$	$[p] \times q$	A wide matrix times a tall matrix.	1 all-gather ($r \times q$)
2	$[p] \times r$	$r \times [q]$	$[p] \times q$	Outer product, may require a large amount of memory.	1 all-gather ($r \times q$)
3	$[p] \times r$	$r \times [q]$	$p \times [q]$	Outer product, may require a large amount of memory.	1 all-gather ($r \times p$)
4	$[p] \times r$	$r \times q$	$[p] \times q$	A distributed matrix times a small, broadcast matrix.	None
5	$p \times [r]$	$[r] \times q$	$p \times q$	Inner product, result broadcast. Suited for inner product between two tall matrices.	1 all-reduce ($p \times q$)
6	$p \times [r]$	$[r] \times q$	$[p] \times q$	Inner product, result distributed.	T reductions ($p \times q/T$ each)
7	$p \times [r]$	$[r] \times q$	$p \times [q]$	Inner product, result distributed.	T reductions ($q \times p/T$ each)
8	$p \times [r]$	$r \times [q]$	$p \times [q]$	Multiply two column-distributed wide matrices	1 all-gather ($p \times r$)
9	$p \times [r]$	$r \times q$	$p \times q$	A distributed matrix times a tall broadcast matrix. Intended for matrix-vector multiplications.	1 all-reduce ($p \times q$)
10	$p \times r$	$[r] \times q$	$p \times q$	A tall broadcast matrix times a distributed matrix. Intended for matrix-vector multiplications.	1 all-reduce ($p \times q$)
11	$p \times r$	$r \times [q]$	$p \times [q]$	A small, broadcast matrix times a distributed matrix	None

1, 9, 10). Otherwise, we consider all the possible combinations in the output: broadcast, split by rows, and split by columns (scenarios 5, 6, 7).

The `distmat.mm()` function implements the 11 scenarios of Table 1 using the PyTorch function `torch.mm()` for within-process matrix multiplication and the collective communication directives (Section 2.1). Scenarios 3, 6, 8, 10, and 11 are implemented using the transpositions of input and output matrices for scenarios 2, 7, 1, 9, and 4, respectively. Transposition costs only a short constant time, as it only 'tags' to the original matrix that it is transposed. The data layout remains intact. A scenario is automatically selected depending on the distribution of the input matrices. The class `distmat` has an attribute for determining if the matrix is distributed by row or column. For scenarios 2, 3; 5, 6, and 7, which share the same input structure, additional keyword parameters are supplied to distinguish them and determine the shape of the output matrix. The type of collective communication operation and the involved matrix block sizes roughly determine the communication cost of the computation. For example, an all-reduce is more expensive than a reduce. The actual cost depends on the network latency, number of MPI messages sent, and sizes of the messages sent between processes, which are all system-dependent.

Listing 3 demonstrates an example usage of `distmat`. We assume that this program is run with four processes (`size` in Line 6 is `4`). Line 8 determines the device to use. If multiple GPUs are involved, the code selects one based on the rank of the process. Line 9 selects the GPU to use with PyTorch. This code runs on a system in which PyTorch is installed with a CUDA-aware MPI implementation. The number of processes to be used can be supplied by a command-line argument (see Appendix B). Line 11 selects the data type and the device used for matrices. The `TType` (for "tensor type") of `torch.cuda.FloatTensor` indicates that single-precision GPU arrays are used, while `DoubleTensor` employs

```
 1 import torch
 2 from dist_stat import distmat
 3 import torch.distributed as dist
 4 dist.init_process_group('mpi')
 5 rank = dist.get_rank()
 6 size = dist.get_world_size()
 7
 8 device = 'cuda:{}'.format(rank) # or 'cpu' for CPU computing
 9 if device.startswith('cuda'): torch.cuda.set_device(rank)
10
11 TType = torch.cuda.FloatTensor if device.startswith('cuda') else torch.
       DoubleTensor    # single precision for GPUs
12 A = distmat.distgen_uniform(4, 4, TType=TType) # create [4] x 4 matrix
13 B = distmat.distgen_uniform(4, 2, TType=TType) # create [4] x 2 matrix
14 AB = distmat.mm(A, B) # A * B, Scenario 1.
15 if rank == 0: # to print this only once
16     print("AB = ")
17 print(rank, AB.chunk) # print the rank's portion of AB.
18 C = (1 + AB).log() # elementwise logarithm
19 if rank == 0:
20     print("log(1 + AB) = ")
21 print(rank, C.chunk) # print the rank's portion of C.
```

Listing 3: An example usage of the module `distmat`.

double-precision CPU arrays. Then Line 12 creates a distributed $[4] \times 4$ matrix and initializes it to uniform $(0, 1)$ random numbers. This matrix is created once and initialized locally, and then distributed to all processes. (For large matrices, `distmat` supports another creation mode that assembles matrix blocks from distributed processes.) Line 14 multiplies the two such matrices A and B to form a distributed matrix of size $[4] \times 2$. Scenario 1 in Table 1 is chosen by `distmat` to create the output AB. Line 18 computes an elementwise logarithm of $1 + AB$, in an elementwise fashion according to the recycling rule. The local block of data residing in each process can be accessed by appending `.chunk` to the name of the distributed matrix, as in Lines 17 and 21.[4]

Although the present implementation only deals with matrices, `distmat` can be easily extended to tensor multiplication, as long as the distributed multiplication scenarios are carefully examined as in Table 1. Creating communication-efficient parallel strategies that minimize the amount of communication between computing units is an active area of research (Van De Geijn and Watts, 1997; Ballard et al., 2011; Koanantakool et al., 2016). Communication-avoiding sparse matrix multiplication has been utilized for sparse inverse covariance estimation (Koanantakool et al., 2018).

6. EXAMPLES

In this section, we compare the performance of the optimization algorithms of Section 4 on various HPC environments for the following four statistical comput-

[4] Lines 17 and 21 do not guarantee printing in order (of ranks). They are printed on a first come, first served basis.

ing examples using distmat: nonnegative matrix factorization (NMF), positron emission tomography (PET), multi-dimensional scaling (MDS), all of which were considered in Zhou et al. (2010), and ℓ_1-regularized Cox proportional hazards regression for survival analysis. For the former three examples the focus is on scaling up the size of feasible problems from those about a decade ago. For the last example, we focus on analyzing a real-world geonomic dataset of size approximately equal to $200,000 \times 500,000$.

6.1 Setup

We employed a local multi-GPU workstation and a virtual cluster consisted of multiple AWS EC2 instances for computing. Table 2 shows the setting of our HPC systems used for the experiments. For virtual cluster experiments, we utilized 1 to 20 of AWS c5.18xlarge instances with 36 physical cores with AVX-512 (512-bit advanced vector extension to the x86 instruction set) enabled in each instance through the CfnCluster resource manager. Network bandwidth of each c5.18xlarge instance was 25GB/s. A separate c5.18xlarge instance served as the "master" instance, which did not participate in computation by itself but managed the computing jobs over the 1 to 20 "worker" instances. Data and software for the experiments were stored in an Amazon Elastic Block Store (EBS) volume attached to this instance and shared among the worker instances via the network file system. Further details are given in Appendix B. For GPU experiments, we used a local machine with two CPUs (10 cores per CPU) and eight Nvidia GTX 1080 GPUs. These are desktop GPUs, not optimized for double-precision. All the experiments were conducted using PyTorch version 0.4 built on the Intel Math Kernel Library (MKL); the released code works for the versions up to 1.6.

We evaluated the objective function once per 100 iterations. For the comparison of execution time, the iteration was run for a fixed number of iterations, regardless of convergence. For comparison of different algorithms for the same problem, we iterated until $\frac{|f(\theta^n)-f(\theta^{n-100})|}{|f(\theta^n)|+1} < 10^{-5}$.

For all the experiments, single-precision computation results on GPU agreed with double-precision ones up to six significant digits, except for ℓ_1-regularized Cox regression, where the PyTorch implementation of the necessary cumulative sum operation caused numerical instability in some cases. Therefore all the experiments for Cox regression were carried out in double-precision. Extra efforts for writing a multi-device code were modest with distmat. Given around 1000 lines of code to implement basic operations for multi-device configuration in distmat, additional code for our four examples was less than 30 lines for each.

6.2 Scaling up examples in Zhou et al. (2010)

Nonnegative matrix factorization NMF is a procedure that approximates a nonnegative data matrix $X \in \mathbb{R}^{m\times p}$ by a product of two low-rank nonnegative matrices, $V \in \mathbb{R}^{m\times r}$ and $W \in \mathbb{R}^{r\times p}$. In a simple setting, NMF minimizes $f(V,W) = \|X - VW\|_{\mathrm{F}}^2$, where $\|\cdot\|_{\mathrm{F}}$ denotes the Frobenius norm. Applying the MM principle to recover the famous multiplicative algorithm due to Lee and Seung (1999, 2001) is discussed in Zhou et al. (2010, Sect. 3.1). Alternatively, the alternating projected gradient (APG) method (Lin, 2007) introduces ridge penalties to minimize $f(V,W;\epsilon) = \|X - VW\|_{\mathrm{F}}^2 + \frac{\epsilon}{2}\|V\|_{\mathrm{F}}^2 + \frac{\epsilon}{2}\|W\|_{\mathrm{F}}^2$. Then the

TABLE 2
HPC environments for experiments

	local node		AWS c5.18xlarge
	CPU	GPU	CPU
Model	Intel Xeon E5-2680 v2	Nvidia GTX 1080	Intel Xeon Platinum 8124M
# of cores	10	2560	18
Clock	2.8 GHz	1.6 GHz	3.0GHz
# of entities	2	8	2 (per instance) × 1-20 (instances)
Total memory	256 GB	64 GB	144 GB × 1–20
Total cores	20	20,480 (CUDA)	36 × 1–20

APG iteration is given by

$$\begin{aligned} V^{n+1} &= P_+\left((1-\sigma_n\epsilon)V^n - \sigma_n(V^nW^n(W^n)^T - X(W^n)^T)\right) \\ W^{n+1} &= P_+\left((1-\tau_n\epsilon)W^n - \tau_n((V^{n+1})^TV^{n+1}W^n - (V^{n+1})^TX)\right), \end{aligned}$$

where P_+ denotes the projection onto the nonnegative orthant; σ_n and τ_n are step sizes. Convergence is guaranteed if $\epsilon > 0$, $\sigma_n \le 1/(2\|W^n(W^n)^T + \epsilon I\|_{\mathrm{F}}^2)$, and $\tau_n \le 1/(2\|(V^n)^TV^n + \epsilon I\|_{\mathrm{F}}^2)$. APG has an additional advantage of avoiding creation of subnormal numbers over the multiplicative algorithm (see Appendix D). Table 3 compares the performance of APG between single-machine multi-GPU and multi-instance virtual cluster settings. Synthetic datasets of sizes [10,000] × 10,000 and [200,000] × 200,000 were created and distributed. For reference, the dimension used in Zhou et al. (2010) is 2429×361. Multi-GPU setting achieved up to 4.14x-speedup over a single CPU instance if the dataset was small, but could not run the larger dataset. The cluster in a cloud was scalable with data, running faster with more instances, yielding up to 4.10x-speedup over the two-instance cluster.

Positron emission tomography PET reconstruction is essentially a deconvolution problem of estimating the intensities of radioactive biomarkers from their line integrals, which can be posed as maximizing the Poisson loglikelihood $L(\lambda) = \sum_{i=1}^d [y_i \log(\sum_{j=1}^p e_{ij}\lambda_j) - \sum_{j=1}^p e_{ij}\lambda_j]$. Here y_i is the observed count of photons arrived coincidentally at detector pair i. Emission intensities $\lambda = (\lambda_1, \cdots, \lambda_p)$ are to be estimated, and e_{ij} is the probability that detector pair i detects an emission form pixel location j, which dependes on the geometry of the detector configuration. We consider a circular geometry for two-dimensional imaging. Adding a ridge-type penalty of $-(\mu/2)\|D\lambda\|_2^2$ to enhance spatial contrast and solving the resulting optimization problem by an MM algorithm is considered in Zhou et al. (2010, Sect. 3.2). Here D is the finite difference matrix on the pixel grid. To promote sharper contrast, we employ the anisotropy total variation (TV) penalty (Rudin et al., 1992) and minimize $-L(\lambda) + \rho\|D\lambda\|_1$. Write $E = (e_{ij})$. Then the PDHG algorithm (Sect. 4.3) can be applied. Put $K = [E^T, D^T]^T$, $f(z, w) = \sum_i(-y_i \log z_i) + \rho\|w\|_1$, and $g(\lambda) = \mathbf{1}^TE\lambda + \delta_+(\lambda)$, where $\mathbf{1}$ is the all-one vector and δ_+ is the $0/\infty$ indicator function for the nonnegative orthant. Since $f(z, w)$ is separable in z and w, applying iteration (6) using the proximity operator (3), we obtain the following iteration:

$$\begin{aligned} z_i^{n+1} &= \tfrac{1}{2}\Big[\big(z_i^n + \sigma(E\bar{\lambda}^n)_i\big) - \big[\big(z_i^n + \sigma(E\bar{\lambda}^n)_i\big)^2 + 4\sigma y_i\big]^{1/2}\Big], \quad i = 1, \ldots, d \\ w^{n+1} &= P_{[-\rho,\rho]}(w^n + \sigma D\bar{\lambda}^n) \end{aligned}$$

$$\lambda^{n+1} = P_+(\lambda^n - \tau(E^T z^{n+1} + D^T w^{n+1} + E^T \mathbf{1}))$$
$$\bar{\lambda}^{n+1} = 2\lambda^{n+1} - \lambda^n,$$

where $P_{[-\rho,\rho]}$ is elementwise projection to the interval $[-\rho,\rho]$. Convergence is guaranteed if $\sigma\tau < 1/\|[E^T \ D^T]\|_2^2$. Scalability experiments were carried out with large Roland-Varadhan-Frangakis phantoms (Roland et al., 2007) using grid sizes $p = 300\times300$, 400×400, and 900×900, with number of detector pairs $d = 179,700$. Timing per 1000 iterations is reported in Table 4. Both matrices E and D were distributed along the columns. For reference, Zhou et al. (2010) use a 64×64 grid with $d = 2016$. The total elapsed time decreases with more GPUs or nodes. The multi-GPU node could not run the $p = 810,000$ dataset, however, since the data size was too big to fit in the GPU memory. Figure 2 illustrates TV reconstructions of a $p = 128 \times 128$ extended cardiac-torso (XCAT) phantom with $d = 8128$ (Lim et al., 2018; Ryu et al., 2020). Results by a stochastic version of PDHG (Chambolle et al., 2018) are also provided. Each reconstruction was run for 20,000 iterations, which were sufficient for both algorithms to reach similar objective values. Those iterations took 20 to 35 seconds on a single GPU.

Multi-dimensional scaling The version of MDS considered in Zhou et al. (2010, Sect. 3.3) minimizes the stress function $f(\theta) = \sum_{i=1}^{q}\sum_{j\neq i} w_{ij}(y_{ij} - \|\theta_i - \theta_j\|_2)^2$ to map dissimilarity measures y_{ij} between data point pairs (i,j) to points $\theta = (\theta_1,\ldots,\theta_q)^T$ in an Euclidean space of low dimension p, where the w_{ij} are the weights. Zhou et al. (2010) derive a parallel MM iteration

$$\theta_{ik}^{n+1} = \left(\sum_{j\neq i}\left[\frac{y_{ij}}{\|\theta_i^n - \theta_j^n\|_2}(\theta_{ik}^n - \theta_{jk}^n) + (\theta_{ik}^n + \theta_{jk}^n)\right]\right) \Big/ \left(2\sum_{j\neq i}^{m} w_{ij}\right)$$

for $i = 1,\ldots,q$ and $k = 1,\ldots,p$. We generated a [10,000] × 10,000 and a [100,000] × 100,000 pairwise dissimilarity matrices from samples of the 1,000-dimensional standard normal distribution. For reference, the dimension of the dissimilarity matrix used in Zhou et al. (2010) is 401 × 401. Elapsed time is reported in Table 5. For $p = 20$, the eight-GPU setting achieved a 5.32x-speedup compared to the single 36-core CPU AWS instance and a 6.13x-speedup compared to single GPU. The larger experiment involved storing a distance matrix of size [100,000] × 100,000, which took 74.5 GB of memory. The multi-GPU node did not scale to run this experiment due to the memory limit. On the other hand, we observed a 3.78x-speedup with 20 instances (720 cores) with respect to four instances (144 cores) of CPU nodes.

Appendix D contains further details on the experiments of this subsection.

6.3 ℓ_1-regularized Cox proportional hazards regression

We apply the proximal gradient method to ℓ_1-regularized Cox proportional hazards regression (Cox, 1972). In this problem, we are given a covariate matrix $X \in \mathbb{R}^{m\times p}$, time-to-event $(t_1,\ldots,t_m)$, and right-censoring time $(c_1,\ldots,c_m)$ for individual $i = 1,\ldots,m$ as data. The "response" is defined by $y_i = \min\{t_i, c_i\}$ for each indivuali i, and whether this individual is censored is indicated by $\delta_i = I_{\{t_i \le c_i\}}$. The log partial likelihood of the Cox model is then

$$L(\beta) = \sum_{i=1}^{m} \delta_i\Big[\beta^T x_i - \log\big(\textstyle\sum_{j:y_j\ge y_i}\exp(\beta^T x_j)\big)\Big].$$

TABLE 3

Runtime (in seconds) of NMF on simulated data for different inner dimensions r. "×" denotes that the experiment could not run with a single data load to the device.

configuration	10,000 × 10,000			200,000 × 200,000		
	10,000 iterations			1000 iterations		
	$r = 20$	$r = 40$	$r = 60$	$r = 20$	$r = 40$	$r = 60$
GPUs						
1	164	168	174	×	×	×
2	97	106	113	×	×	×
4	66	78	90	×	×	×
8	57	77	92	×	×	×
AWS EC2 `c5.18xlarge` instances						
4	205	310	430	1493	1908	2232
5	230	340	481	1326	1652	2070
8	328	390	536	937	1044	1587
10	420	559	643	737	937	1179
20	391	1094	1293	693	818	1041

TABLE 4

Runtime (in seconds) comparison of 1000 iterations of TV-penalized PET. We exploited sparse structures of E and D. The number of detector pairs d was fixed at 179,700.

configuration	$p = 90,000$	$p = 160,000$	$p = 810,000$
GPUs			
1	×	×	×
2	21	35	×
4	19	31	×
8	18	28	×
AWS EC2 `c5.18xlarge` instances			
4	36	49	210
5	36	45	188
8	33	39	178
10	38	37	153
20	26	28	131

TABLE 5

Runtimes (in seconds) of 1000 iterations for MDS for different mapped dimensions q.

configuration	10,000 datapoints			100,000 datapoints		
	10,000 iterations			1000 iterations		
	$q = 20$	$q = 40$	$q = 60$	$q = 20$	$q = 40$	$q = 60$
GPUs						
1	368	376	384	×	×	×
2	185	190	195	×	×	×
4	100	103	108	×	×	×
8	60	67	73	×	×	×
AWS EC2 `c5.18xlarge` instances						
4	424	568	596	3103	3470	3296
5	364	406	547	2634	2700	2730
8	350	425	520	1580	1794	1834
10	275	414	457	1490	1454	1558
20	319	440	511	820	958	1043

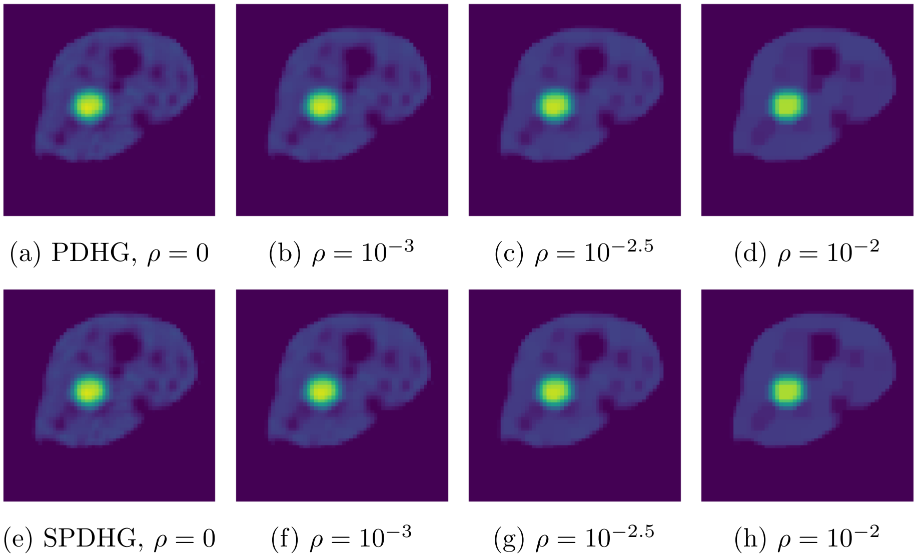

Fig 2: Reconstruction of the XCAT phantom with a TV penalty with regularization parameter ρ, using deterministic (top row) and stochastic (bottom) PDHG.

Coordinate-descent-type approaches to this model are proposed by Suchard et al. (2013) and Mittal et al. (2014).

To obtain a proximal gradient iteration, we need the gradient $\nabla L(\beta)$ and its Lipschitz constant. The gradient of the log partial likelihood is

$$\nabla L(\beta) = X^T(I - P)\delta, \quad \delta = (\delta_1, \dots, \delta_m)^T,$$

where we define the matrix $P = (\pi_{ij})$ with $\pi_{ij} = I(y_i \geq y_j)w_i/W_j$; $w_i = \exp(x_i^T\beta)$, $W_j = \sum_{i:y_i \geq y_j} w_i$. A Lipschitz constant of $\nabla L(\beta)$ can be found by finding an upper bound of the Hessian of $L(\beta)$:

$$\nabla^2 L(\beta) = X^T(P\mathrm{diag}(\delta)P^T - \mathrm{diag}(P\delta))X.$$

Note $\|P\|_2 \leq 1$, since the sum of each row of P is 1. It follows that $\|\nabla^2 L(\beta)\|_2 \leq 2\|X\|_2^2$, and $\|X\|_2$ can be quickly computed by using the power iteration (Golub and Van Loan, 2013).

We introduce an ℓ_1-penalty to the log partial likelihood in order to enforce sparsity in the regression coefficients and use the proximal gradient descent to estimate β by putting $g(\beta) = -L(\beta)$, $f(\beta) = \lambda\|\beta\|_1$. Then the iteration is:

$$\begin{aligned}
w_i^{n+1} &= \exp(x_i^T\beta); \;\; W_j^{n+1} = \textstyle\sum_{i:y_i \geq y_j} w_i^{n+1} \\
\pi_{ij}^{n+1} &= I_{\{t_i \geq t_j\}} w_i^{n+1}/W_j^{n+1} \\
\Delta^{n+1} &= X^T(I - P^{n+1})\delta, \;\; \text{where } P^{n+1} = (\pi_{ij}^{n+1}) \\
\beta^{n+1} &= \mathcal{S}_\lambda(\beta^n + \sigma\Delta^{n+1}).
\end{aligned}$$

If the data are sorted in descending order of y_i, the W_j^n can be computed by cumulative summing $(w_1, \dots, w_m)$ in the proper order. A CUDA kernel for this operation is readily available in PyTorch. The soft-thresholding operator $\mathcal{S}_\lambda(x)$ is

```
import torch
import torch.distributed as dist
from dist_stat import distmat
from dist_stat.distmat import distgen_uniform, distgen_normal
dist.init_process_group('mpi')
rank = dist.get_rank()
size = dist.get_world_size()
device = 'cuda:{}'.format(rank) # 'cpu' for CPU computing
if device.startswith('cuda'): torch.cuda.set_device(rank)
n = 10000
p = 10000
max_iter = 10000
TType = torch.cuda.FloatTensor if device.startswith('cuda') else torch.
    DoubleTensor
X = distgen_normal(p, n, TType=TType).t() # [p] x n transposed => n x [p].
delta = torch.multinomial(torch.tensor([1., 1.]), n, replacement=True).float().
    view(-1, 1).type(TType) # censoring indicator, n x 1, Bernoulli(0.5).
delta_dist = distmat.dist_data(delta, TType=TType) # distribute delta to create
     [n] x 1 data
beta = distmat.dist_data(torch.zeros((p, 1)).type(TType), TType=TType) #[p] x 1
Xt = X.t() # transpose. [p] x n
sigma = 0.00001 # step size
lambd = 0.00001 # penalty parameter
soft_threshold = torch.nn.Softshrink(lambd) # soft-thresholding

# Data points are assumed to be sorted in decreasing order of observed time.
y_local = torch.arange(n, 0, step=-1).view(-1, 1).type(TType) # local n x 1
y_dist = distmat.dist_data(y_local, TType=TType) # distributed [n] x 1
pi_ind = (y_dist - y_local.t() >= 0).type(TType)

Xbeta = distmat.mm(X, beta) # Scenario 5 (default for n x [p] times [p] x 1)

for i in range(max_iter):
    w = Xbeta.exp() # n x 1
    W = w.cumsum(0) # n x 1
    dist.barrier() # wait until the distributed computation above is finished

    w_dist = distmat.dist_data(w, TType=TType) # distribute w. [n] x 1.

    pi = (w_dist / W.t()) * pi_ind # [n] x n.
    pd = distmat.mm(pi, delta) # Scenario 4.
    dmpd = delta_dist - pd # [n] x 1.
    grad = distmat.mm(Xt, dmpd) # Scenario 1.
    beta = (beta + grad * sigma).apply(soft_threshold) # [p] x 1
    Xbeta = distmat.mm(X, beta) # Scenario 5.

    expXbeta = (Xbeta).exp() # n x 1
    obj = distmat.mm(delta.t(), (Xbeta - (expXbeta.cumsum(0)).log())) \
        - lambd * beta.abs().sum() # mm: local computation.
    print(i, obj.item())
```

Listing 4: PyTorch code for the proximal gradient method for ℓ_1-regularized Cox regression.

also implemented in PyTorch. We can write a simple proximal gradient descent routine for the Cox regression as in Listing 4, assuming no ties in y_i's.

A synthetic data matrix $X \in \mathbb{R}^{m\times[p]}$, distributed along the columns, was sampled from the standard normal distribution. The algorithm was designed to keep a copy of the estimand β in every device. All the numerical experiments were carried out with double precision even for GPUs, for the following reason. For

TABLE 6

Runtime comparison of ℓ_1-regularized Cox regression over multi-node virtual cluster on AWS EC2. Elapsed time (in seconds) after 1000 iterations.

configuration	$10,000 \times [10,000]$ 10,000 iterations	$100,000 \times [200,000]$ 1,000 iterations
GPUs		
1	386	×
2	204	×
4	123	×
8	92	×
AWS EC2 `c5.18xlarge` instances		
1	580	×
2	309	×
4	217	1507
5	170	1535
8	145	775
10	132	617
20	148	384

a very small value of λ (we used $\lambda = 10^{-5}$), when single precision was used in GPUs, the estimate quickly tended to “not a number (NaN)”s due to numerical instability of the CUDA kernel. Double-precision did not generate such a problem. Although desktop GPU models such as Nvidia GTX and Titan X are not optimized for double precision floating-point operations and is known to be 32 times slower for double precision operations than single precision operations, this does not necessarily mean that the total computation time is 32 times slower, since latency takes a significant portion of the total computation time in GPU computing.

In order to demonstrate the scalability of our approach, elapsed times for 10,000 × [10,000] and 100,000 × [200,000] simulated data are reported in Table 6. We can see 3.92x speedup from 4 nodes to 20 nodes in the virtual cluster. Even with double-precision arithmetics, eight GPUs could achieve a 6.30x-speedup over the single 36-core CPU instance. As expected, virtual clusters in a cloud exhibited better scalability.

6.4 Genome-wide survival analysis of the UK Biobank dataset

We demonstrate a real-world application of ℓ_1-regularized Cox proportional hazards regression to genome-wide survival analysis for Type 2 Diabetes (T2D). We used a UK Biobank dataset (Sudlow et al., 2015) that contains information on approximately 800,000 single nucleotide polymorphisms (SNPs) of 500,000 individual subjects recruited from the United Kingdom. After filtering SNPs for quality control and subjects for the exclusion of Type 1 Diabetes patients, 402,297 subjects including 17,994 T2D patients and 470,189 SNPs remained. We randomly sampled 200,000 subjects including 8,995 T2D patients for our analysis. Any missing genotype was imputed with the column mean. Along with the SNPs, sex and top ten principal components were included as unpenalized covariates to adjust for population-specific variations. The resulting dataset was 701 GB with double-precision.

The analysis for this large-scale genome-wide dataset was conducted as follows. Incidence of T2D was used as the event ($\delta_i = 1$) and the age of onset was used as survival time y_i. For non-T2D subjects ($\delta_i = 0$), age at the last visit was used as y_i. We chose 63 different values of the regularization parameter λ in the range $[0.7 \times 10^{-9}, 1.6 \times 10^{-8}]$, with which 0 to 111 SNPs were selected. For each value of λ, the ℓ_1-regularized Cox regression model of Section 6.3 was fitted. Every run converged after at most 2080 iterations that took less than 2800 seconds using 20 `c5.18xlarge` instances from AWS EC2.

The SNPs were ranked based on the largest value of λ with which a SNP is selected. (No variables were removed once selected within the range of λ used. The regularization path and the full list of the selected SNPs are available in Appendix E.) Among the 111 SNPs selected, three of the top four selections were located on TCF7L2, whose association with T2D is well-known (Scott et al., 2007; The Wellcome Trust Case Control Consortium, 2007). Also prominently selected were SNPs from genes SLC45A2 and HERC2, whose variants are known to be associated with skin, eye, and hair pigmentation (Cook et al., 2009). This is possibly due to the dominantly European population in the UK Biobank study. Mapped genes for 24 SNPs out of the selected 111 were also reported in Mahajan et al. (2018), a meta-analysis of 32 genome-wide association studies (GWAS) for about 898,130 individuals of European ancestry; see Tables E.1 and E.2 for details. We then conducted an unpenalized Cox regression analysis using the 111 selected SNPs. The nine SNPs with p-values less than 0.01 are listed in Table 7. The locations in Table 7 are with respect to the reference genome GRCh37 (Church et al., 2011), and mapped genes were predicted by the Ensembl Variant Effect Predictor (McLaren et al., 2016). Among these nine SNPs, three of them were directly shown to be associated with T2D (The Wellcome Trust Case Control Consortium (2007) and Dupuis et al. (2010) for rs4506565, Voight et al. (2010) for rs8042680, Ng et al. (2014) for rs343092). Three other SNPs have mapped genes reported to be associated with T2D in Mahajan et al. (2018): rs12243326 on TCF7L2, rs343092 on HMGA2, and rs231354 on KCNQ1.

Although the interpretation of the results requires additional sub-analysis, the result shows the promise of joint association analysis using multiple regression models. In GWAS it is customary to analyze the data on SNP-by-SNP basis. Among the mapped genes harboring the 111 SNPs selected by our half-million-variate regression analysis are CPLX3 and CACNA1A, associated with regulation of insulin secretion, and SEMA7A and HLA-DRA involved with inflammatory responses (based on DAVID (Huang et al., 2009*a*,*b*)). These genes might have been missed in conventional univariate analysis of T2D due to nominally moderate statistical significance. Joint GWAS may overcome such a limitation and is possible by combining the computing power of modern HPC and scalable algorithms.

7. DISCUSSION

Abstractions of highly complex computing operations have rapidly evolved over the last decade. In this article, we have explained how statisticians can benefit from this evolution. We have seen how deep learning technology is relevant to high-performance statistical computing. We have also demonstrated that many useful tools for incorporating accelerators and computing clusters have been created. Unfortunately, such developments have been mainly made in languages other than R, particularly in Python, with which statisticians may not be familiar with. Although there are libraries that deal with simple parallel computation in R, there are common issues with these libraries. First, the libraries do not

TABLE 7

SNPs with p-values of less than 0.01 on unpenalized Cox regression with variables selected by ℓ_1-penalized Cox regression

SNP ID	Chr.	Location	A1[A]	A2[B]	MAF[C]	Mapped Gene	Coefficient	p-value
rs4506565	10	114756041	A	**T**	0.238	TCF7L2	2.810e-1	<2e-16
rs12243326	10	114788815	**C**	T	0.249	TCF7L2	1.963e-1	0.003467
rs8042680	15	91521337	**A**	C	0.277	PRC1	2.667e-1	0.005052
rs343092	12	66250940	**T**	G	0.463	HMGA2	−7.204e-2	0.000400
rs7899137	10	76668462	**A**	C	0.289	KAT6B	−4.776e-2	0.002166
rs8180897	8	121699907	A	**G**	0.445	SNTB1	6.361e-2	0.000149
rs10416717	19	13521528	A	**G**	0.470	CACNA1A	5.965e-2	0.009474
rs231354	11	2706351	**C**	T	0.329	KCNQ1	4.861e-2	0.001604
rs9268644	6	32408044	**C**	A	0.282	HLA-DRA	6.589e-2	2.11e-5

[A] Minor allele, [B] Major allele, [C] Minor allele frequency. The boldface indicates the risk allele determined by the reference allele and the sign of the regression coefficient.

easily incorporate GPUs that might significantly speed up computation. Second, it is hard to write more full-fledged parallel programs without directly writing code in C or C++. This two-language problem calls for statisticians to take a second look at Python. Fortunately, this language is not hard to learn, and younger generations are quite familiar with it. A remedy from the R side may be either developing more user-friendly interfaces for the distributed-memory environment, with help from those who are engaged in computer engineering, or writing a good wrapper for the important Python libraries. A Python interface to R may be a good starting point. For example, R package `reticulate` (Ushey et al., 2021) is a basis of other interfaces packages to PyTorch (`rTorch`, Reyes, 2021) and TensorFlow (also called `tensorflow`, RStudio, 2021).

By making use of multiple CPU nodes or a multi-GPU workstation, the methods discussed in the current article can be applied efficiently even when the dataset exceeds several tens of gigabytes. The advantages of engaging multiple compute devices are two-fold. First, we can take advantage of data parallelism with more computing cores, accelerating the computation. Second, we can push the limit of the size of the dataset to analyze. As cloud providers now support virtual clusters better suited for HPC, statisticians can deal with bigger problems utilizing such services, using up to several thousand cores easily. When the data do not fit into the GPU memory (e.g., the UK Biobank example), it is still possible to carry out computation by moving partitions of the data in and out of GPUs. However, this is impractical because of slow communication between the main and GPU memories. On the other hand, virtual clusters are scalable with this size of data.

Loss of accuracy due to the default single precision of GPU arithmetic, prominent in our proportional hazards regression example, can be solved by purchasing scientifically-oriented GPUs with better double precision supports. Another option is migrating to the cloud: for example, the `P2` and `P3` instances in AWS support scientific GPUs. Nevertheless, desktop GPUs with double precision arithmetic turned on could achieve more than 10-fold speedup over CPU, even though double precision floating-point operations are 32 times slower than single precision.

Most of the highly parallelizable algorithms considered in Section 4 require no more than the first-order derivative information, and this feature contributes to their low per-iteration complexity and parallelizability. As mentioned in Section

1, some second-order methods for sparse regression (Li et al., 2018; Huang et al., 2018, 2021) maintain the set of active variables (of nonzero coefficients), and only these are involved in the Newton-Raphson step. Thus if the solution is sparse, the cost of solving the relevant linear system is moderate. With distributed matrix computation exemplified with `distmat`, residual and gradients can be computed in a distributed fashion and the linear system can be solved after gathering active variables into the master device.

A major weakness of the present approach is that its effectiveness can be degraded by the communication cost between the nodes and devices. One way to avoid this issue is by using high-speed interconnection between the nodes and devices. In multi-CPU clusters, this can be realized by a high-speed interconnection technology such as InfiniBand. Even when such an environment is not affordable, we may still use relatively high-speed connection equipped with instances from a cloud. The network bandwidth of 25Gbps supported for `c5.18xlarge` instances of AWS was quite effective in our experiments. Reducing the number of communication rounds and iterations with theoretical guarantees, for example, by one-shot averaging (Zhang et al., 2013; Duchi et al., 2014; Lee et al., 2015), by using global first-order information and local higher-order information (Wang et al., 2017; Jordan et al., 2019; Fan et al., 2019), or by quantization (Tang et al., 2019; Liu et al., 2020), is an active area of current research.

Although PyTorch has been advocated throughout this article, it is not the only path towards easy-to-use programming models in shared- and distributed-memory programming environments. A possible alternative is Julia (Bezanson et al., 2017), in which data can reside in a wide variety of environments, such as GPUs (Besard et al., 2019) and multiple CPU nodes implementing the distributed memory model (JuliaParallel Team, 2021; Janssens, 2021). While its long-term support release of version 1.0.5 in September 2019 is still fresh, Julia has the potential to be a powerful tool for statistical HPC once the platforms and user community mature.

ACKNOWLEDGEMENTS

This research was partially funded by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (2019R1A2C1007126, JHW; 2020R1A6A3A03037675, SK), the Collaboratory Fellowship program of the UCLA Institute for Quantitative & Computational Bioscience (SK), and grants from National Institutes of Health (R35GM141798, HZ; R01HG006139, HZ and JJZ; K01DK106116, JJZ; R21HL150374, JJZ) and National Science Foundation (DMS-2054253, HZ and JJZ).

REFERENCES

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y. and Zheng, X. (2016), 'TensorFlow: Large-scale machine learning on heterogeneous systems', *arXiv preprint arXiv:1603.04467* . Software available from https://tensorflow.org.

Amazon Web Services (2021), 'AWS ParallelCluster', https://aws.amazon.com/ko/hpc/parallelcluster/. Version 2.11.0. Accessed: 2021-07-03.

Apache Software Foundation (2021), 'Apache Hadoop', https://hadoop.apache.org. Version 3.3.1. Accessed: 2021-07-03.

Armbrust, M., Fox, A., Griffith, R., Joseph, A. D., Katz, R., Konwinski, A., Lee, G., Patterson, D., Rabkin, A., Stoica, I. and Zaharia, M. (2010), 'A view of cloud computing', *Commun. ACM* **53**(4), 50–58.

Atchadé, Y. F., Fort, G. and Moulines, E. (2017), 'On perturbed proximal gradient algorithms', *J. Mach. Learn. Res.* **18**(1), 310–342.

Bahrampour, S., Ramakrishnan, N., Schott, L. and Shah, M. (2016), 'Comparative study of deep learning software frameworks', *arXiv preprint arXiv:1511.06435* .

Ballard, G., Demmel, J., Holtz, O. and Schwartz, O. (2011), 'Minimizing communication in numerical linear algebra', *SIAM J. Matrix Anal. Appl.* **32**(3), 866–901.

Bauer, B., Kohler, M. et al. (2019), 'On deep learning as a remedy for the curse of dimensionality in nonparametric regression', *Ann. Statist.* **47**(4), 2261–2285.

Baydin, A. G., Pearlmutter, B. A., Radul, A. A. and Siskind, J. M. (2017), 'Automatic differentiation in machine learning: a survey', *J. Mach. Learn. Res.* **18**(1), 5595–5637.

Beck, A. (2017), *First-order methods in optimization*, SIAM.

Beck, A. and Teboulle, M. (2009), 'A fast iterative shrinkage-thresholding algorithm for linear inverse problems', *SIAM J. Imaging Sci.* **2**(1), 183–202.

Besard, T., Foket, C. and De Sutter, B. (2019), 'Effective extensible programming: Unleashing Julia on GPUs', *IEEE Trans. Parallel Distrib. Syst.* **30**(4), 827–841.

Bezanson, J., Edelman, A., Karpinski, S. and Shah, V. B. (2017), 'Julia: A fresh approach to numerical computing', *SIAM Review* **59**(1), 65–98.

Blackford, L. S., Petitet, A., Pozo, R., Remington, K., Whaley, R. C., Demmel, J., Dongarra, J., Duff, I., Hammarling, S., Henry, G. et al. (2002), 'An updated set of basic linear algebra subprograms (BLAS)', *ACM Trans. Math. Software* **28**(2), 135–151.

Boyd, S., Parikh, N., Chu, E., Peleato, B. and Eckstein, J. (2011), 'Distributed optimization and statistical learning via the alternating direction method of multipliers', *Found. Trends Mach. Learn.* **3**(1), 1–122.

Buckner, J., Wilson, J., Seligman, M., Athey, B., Watson, S. and Meng, F. (2009), 'The gputools package enables GPU computing in R', *Bioinformatics* **26**(1), 134–135.

Chambolle, A., Ehrhardt, M. J., Richtárik, P. and Schonlieb, C.-B. (2018), 'Stochastic primal-dual hybrid gradient algorithm with arbitrary sampling and imaging applications', *SIAM J. Optim.* **28**(4), 2783–2808.

Chambolle, A. and Pock, T. (2011), 'A first-order primal-dual algorithm for convex problems with applications to imaging', *J. Math. Imaging Vision* **40**(1), 120–145.

Chen, T., Li, M., Li, Y., Lin, M., Wang, N., Wang, M., Xiao, T., Xu, B., Zhang, C. and Zhang, Z. (2015), 'MXNet: A flexible and efficient machine learning library for heterogeneous distributed systems', *arXiv preprint arXiv:1512.01274* .

Chen, Y., Lan, G. and Ouyang, Y. (2014), 'Optimal primal-dual methods for a class of saddle point problems', *SIAM J. Optim.* **24**(4), 1779–1814.

Chi, E. C., Zhou, H. and Lange, K. (2014), 'Distance majorization and its applications', *Math. Program.* **146**(1-2), 409–436.

Chu, D., Zhang, C., Sun, S. and Tao, Q. (2020), Semismooth Newton algorithm for efficient projections onto $\ell_{1,\infty}$-norm ball, *in* 'ICML 2020', Vol. 119 of *Proc. Mach. Learn. Res.*, pp. 1974–1983.

Church, D. M., Schneider, V. A., Graves, T., Auger, K., Cunningham, F., Bouk, N., Chen, H.-C., Agarwala, R., McLaren, W. M., Ritchie, G. R. et al. (2011), 'Modernizing reference genome assemblies', *PLoS Biol.* **9**(7), e1001091.

Collobert, R., Kavukcuoglu, K. and Farabet, C. (2011), Torch7: A Matlab-like environment for machine learning, *in* 'BigLearn, NeurIPS Workshop'.

Combettes, P. L. (2018), 'Monotone operator theory in convex optimization', *Math. Program.* **170**, 177–206.

Combettes, P. L. and Pesquet, J.-C. (2011), Proximal splitting methods in signal processing, *in* 'Fixed-point algorithms for inverse problems in science and engineering', Springer, pp. 185–212.

Condat, L. (2013), 'A primal-dual splitting method for convex optimization involving Lipschitzian, proximable and linear composite terms', *J. Optim. Theory Appl.* **158**(2), 460–479.

Cook, A. L., Chen, W., Thurber, A. E., Smit, D. J., Smith, A. G., Bladen, T. G., Brown, D. L., Duffy, D. L., Pastorino, L., Bianchi-Scarra, G. et al. (2009), 'Analysis of cultured human melanocytes based on polymorphisms within the SLC45A2/MATP, SLC24A5/NCKX5, and OCA2/P loci', *J. Invest. Dermatol.* **129**(2), 392–405.

Cox, D. R. (1972), 'Regression models and life-tables', *J. R. Stat. Soc. Ser. B. Stat. Methodol.* **34**(2), 187–202.

Dean, J. and Ghemawat, S. (2008), 'MapReduce: simplified data processing on large clusters', *Commun. ACM* **51**(1), 107–113.

Defazio, A., Bach, F. and Lacoste-Julien, S. (2014), SAGA: A fast incremental gradient method with support for non-strongly convex composite objectives, *in* 'NeurIPS 2014', Vol. 27 of *Adv. Neural Inform. Process. Syst.*, pp. 1646–1654.

Dempster, A. P., Laird, N. M. and Rubin, D. B. (1977), 'Maximum likelihood from incomplete data via the EM algorithm', *J. R. Stat. Soc. Ser. B. Stat. Methodol.* pp. 1–38.

Donoho, D. (2017), '50 years of data science', *J. Comput. Graph. Statist.* **26**(4), 745–766.

Duchi, J. C., Jordan, M. I., Wainwright, M. J. and Zhang, Y. (2014), 'Optimality guarantees for distributed statistical estimation', *arXiv preprint arXiv:1405.0782* .

Dupuis, J., Langenberg, C., Prokopenko, I., Saxena, R., Soranzo, N., Jackson, A. U., Wheeler, E., Glazer, N. L., Bouatia-Naji, N., Gloyn, A. L. et al. (2010), 'New genetic loci implicated in fasting glucose homeostasis and their impact on type 2 diabetes risk', *Nat. Genet.* **42**(2), 105–116.

Eddelbuettel, D. (2021), 'CRAN Task View: High-performance and parallel computing with R', https://cran.r-project.org/web/views/HighPerformanceComputing.html. Version 2021-05-27.

Eijkhout, V. (2016), *Introduction to High Performance Scientific Computing*, 2nd edn, Lulu.com.

Esser, E., Zhang, X. and Chan, T. F. (2010), 'A general framework for a class of first order primal-dual algorithms for convex optimization in imaging science', *SIAM J. Imaging Sci.* **3**(4), 1015–1046.

Evangelinos, C. and Hill, C. N. (2008), Cloud computing for parallel scientific HPC applications: Feasibility of running coupled atmosphere-ocean climate models on Amazon's EC2, *in* 'CCA 2008', ACM.

Facebook Incubator (2021), 'Gloo: Collective communications library with various primitives for multi-machine training', https://github.com/facebookincubator/gloo. Accessed: 2021-07-03.

Fan, J., Guo, Y. and Wang, K. (2019), 'Communication-efficient accurate statistical estimation', *arXiv preprint arXiv:1906.04870* .

Fox, A. (2011), 'Cloud computing—What's in it for me as a scientist?', *Science* **331**(6016), 406–407.

Friedman, J., Hastie, T. and Tibshirani, R. (2010), 'Regularization paths for generalized linear models via coordinate descent', *J. Stat. Softw.* **33**(1), 1–22.

Gabay, D. and Mercier, B. (1976), 'A dual algorithm for the solution of nonlinear variational problems via finite element approximation', *Comput. Math. Appl.* **2**(1), 17–40.

Gabriel, E., Fagg, G. E., Bosilca, G., Angskun, T., Dongarra, J. J., Squyres, J. M., Sahay, V., Kambadur, P., Barrett, B., Lumsdaine, A., Castain, R. H., Daniel, D. J., Graham, R. L. and Woodall, T. S. (2004), Open MPI: Goals, concept, and design of a next generation MPI implementation, *in* 'Proceedings of the 11th European PVM/MPI Users' Group Meeting', Budapest, Hungary, pp. 97–104.

Gentzsch, W. (2001), Sun Grid Engine: Towards creating a compute power grid, *in* 'CCGRID 2001', IEEE, pp. 35–36.

Gittens, A., Devarakonda, A., Racah, E., Ringenburg, M., Gerhardt, L., Kottalam, J., Liu, J., Maschhoff, K., Canon, S., Chhugani, J. et al. (2016), Matrix factorizations at scale: A comparison of scientific data analytics in Spark and C + MPI using three case studies, *in* '2016 IEEE BigData', IEEE, pp. 204–213.

Golub, G. H. and Van Loan, C. F. (2013), *Matrix Computations*, Johns Hopkins University Press, Baltimore, MD.

Gower, R. M., Loizou, N., Qian, X., Sailanbayev, A., Shulgin, E. and Richtárik, P. (2019), SGD: General analysis and improved rates, *in* 'ICML 2019', Vol. 97 of *Proc. Mach. Learn. Res.*, pp. 5200–5209.

Griewank, A. and Walther, A. (2008), *Evaluating derivatives: principles and techniques of algorithmic differentiation*, SIAM.

Gu, Y., Fan, J., Kong, L., Ma, S. and Zou, H. (2018), 'ADMM for high-dimensional sparse penalized quantile regression', *Technometrics* **60**(3), 319–331.

Hager, G. and Wellein, G. (2010), *Introduction to High Performance Computing for Scientists and Engineers*, CRC Press.

Huang, D. W., Sherman, B. T. and Lempicki, R. A. (2009*a*), 'Bioinformatics enrichment tools: paths toward the comprehensive functional analysis of large gene lists', *Nucleic Acids Res.* **37**(1), 1–13.

Huang, D. W., Sherman, B. T. and Lempicki, R. A. (2009*b*), 'Systematic and integrative analysis of large gene lists using DAVID bioinformatics resources', *Nat. Protoc.* **4**(1), 44–57.

Huang, J., Jiao, Y., Jin, B Liu, J., , Lu, X. and Yang, C. (2021), 'A unified primal dual active set algorithm for nonconvex sparse recovery', *Statist. Sci.* **36**(2), 215–238.

Huang, J., Jiao, Y., Liu, Y. and Lu, X. (2018), 'A constructive approach to ℓ_0 penalized regression', *J. Mach. Learn. Res.* **19**(1), 403–439.

Hunter, D. and Li, R. (2005), 'Variable selection using MM algorithms', *Ann. Statist.* **33**(4), 1617–1642.

Hunter, D. R. and Lange, K. (2004), 'A tutorial on MM algorithms', *Amer. Statist.* **58**(1), 30–37.

Hyperion Research (2019), Hyperion Research HPC market update from ISC 2019, Technical report, Hyperion Research.

Imaizumi, M. and Fukumizu, K. (2019), Deep neural networks learn non-smooth functions effectively, *in* 'AISTATS 2019', Vol. 89 of *Proc. Mach. Learn. Res.*, pp. 869–878.

Janssens, B. (2021), 'MPIArrays.jl: Distributed arrays based on MPI onesided communication', https://github.com/barche/MPIArrays.jl. Accessed: 2021-07-03.

Jha, S., Qiu, J., Luckow, A., Mantha, P. and Fox, G. C. (2014), A tale of two data-intensive paradigms: Applications, abstractions, and architectures, *in* '2014 IEEE BigData', IEEE, pp. 645–652.

Jia, Y., Shelhamer, E., Donahue, J., Karayev, S., Long, J., Girshick, R., Guadarrama, S. and Darrell, T. (2014), Caffe: Convolutional architecture for fast feature embedding, *in* 'MM 2014', ACM, pp. 675–678.

Johnson, R. and Zhang, T. (2013), Accelerating stochastic gradient descent using predictive variance reduction, *in* 'NeurIPS 2013', Vol. 26 of *Adv. Neural Inform. Process. Syst.*, pp. 315–323.

Jordan, M. I., Lee, J. D. and Yang, Y. (2019), 'Communication-efficient distributed statistical inference', *J. Amer. Statist. Assoc.* **114**(526), 668–681.

JuliaParallel Team (2021), 'DistributedArrays.jl: Distributed arrays in Julia', https://github.com/JuliaParallel/DistributedArrays.jl. Accessed: 2021-07-03.

Keys, K. L., Zhou, H. and Lange, K. (2019), 'Proximal distance algorithms: Theory and practice.', *J. Mach. Learn. Res.* **20**(66), 1–38.

Kirk, D. (2007), NVIDIA CUDA software and GPU parallel computing architecture, *in* 'ISMM', Vol. 7, pp. 103–104.

Klöckner, A., Pinto, N., Lee, Y., Catanzaro, B., Ivanov, P. and Fasih, A. (2012), 'PyCUDA and PyOpenCL: A scripting-based approach to GPU run-time code generation', *Parallel Comput.* **38**(3), 157–174.

Ko, S. (2020), Easily parallelizable statistical computing methods and their applications in modern high-performance computing environments, PhD thesis, Seoul National University.

Ko, S. and Won, J.-H. (2019), Optimal minimization of the sum of three convex functions with a linear operator, *in* 'AISTATS 2019', Vol. 89 of *Proc. Mach. Learn. Res.*, pp. 1185–1194.

Ko, S., Yu, D. and Won, J.-H. (2019), 'Easily parallelizable and distributable class of algorithms for structured sparsity, with optimal acceleration', *J. Comput. Graph. Statist.* **28**(4), 821–833.

Koanantakool, P., Ali, A., Azad, A., Buluc, A., Morozov, D., Oliker, L., Yelick, K. and Oh, S.-Y. (2018), Communication-avoiding optimization methods for distributed massive-scale sparse inverse covariance estimation, *in* 'AISTATS 2018', Vol. 84 of *Proc. Mach. Learn. Res.*, pp. 1376–1386.

Koanantakool, P., Azad, A., Buluç, A., Morozov, D., Oh, S.-Y., Oliker, L. and Yelick, K. (2016), Communication-avoiding parallel sparse-dense matrix-matrix multiplication, *in* '2016 IEEE IPDPS', IEEE, pp. 842–853.

Kummer, B. (1988), Newton's method for non-differentiable functions, *in* J. Guddat, B. Bank, H. Hollatz, P. Kall, D. Karte, B. Kummer, K. Lommatzsch, L. Tammer, M. Vlach and K. Zimmerman, eds, 'Advances in Mathematical Optimization', Vol. 45, Akademie-Verlag, Berlin, pp. 114–125.

Lai, T. L. and Yuan, H. (2021), 'Stochastic approximation: from statistical origin to big-data, multidisciplinary applications', *Statist. Sci.* **36**(2), 291–302.

Lam, S. K., Pitrou, A. and Seibert, S. (2015), Numba: A LLVM-based Python JIT compiler, *in* 'LLVM 2015', number 7, ACM, pp. 1–6.

Lange, K. (2016), *MM Optimization Algorithms*, Vol. 147, SIAM.

Lange, K., Hunter, D. R. and Yang, I. (2000), 'Optimization transfer using surrogate objective functions', *J. Comput. Graph. Statist.* **9**(1), 1–20.

LeCun, Y., Bengio, Y. and Hinton, G. (2015), 'Deep learning', *Nature* **521**(7553), 436–444.

Lee, D. D. and Seung, H. S. (1999), 'Learning the parts of objects by non-negative matrix factorization', *Nature* **401**(6755), 788–791.

Lee, D. D. and Seung, H. S. (2001), Algorithms for non-negative matrix factorization, *in* 'NeurIPS 2001', Vol. 14 of *Adv. Neural Inform. Process. Syst.*, pp. 556–562.

Lee, J. D., Sun, Y., Liu, Q. and Taylor, J. E. (2015), 'Communication-efficient sparse regression: a one-shot approach', *arXiv preprint arXiv:1503.04337* .

Lee, T., Won, J.-H., Lim, J. and Yoon, S. (2017), 'Large-scale structured sparsity via parallel fused lasso on multiple GPUs', *J. Comput. Graph. Statist.* **26**(4), 851–864.

Li, X., Sun, D. and Toh, K.-C. (2018), 'A highly efficient semismooth newton augmented lagrangian method for solving lasso problems', *SIAM J. Optim.* **28**(1), 433–458.

Lim, H., Dewaraja, Y. K. and Fessler, J. A. (2018), 'A PET reconstruction formulation that enforces non-negativity in projection space for bias reduction in Y-90 imaging', *Phys. Med. Biol.* **63**(3), 035042.

Lin, C.-J. (2007), 'Projected gradient methods for nonnegative matrix factorization', *Neural Comput.* **19**(10), 2756–2779.

Liu, X., Li, Y., Tang, J. and Yan, M. (2020), A double residual compression algorithm for efficient distributed learning, *in* 'AISTATS 2020', Vol. 108 of *Proc. Mach. Learn. Res.*, pp. 133–143.

Mahajan, A., Taliun, D., Thurner, M., Robertson, N. R., Torres, J. M., Rayner, N. W., Payne, A. J., Steinthorsdottir, V., Scott, R. A., Grarup, N. et al. (2018), 'Fine-mapping type 2 diabetes loci to single-variant resolution using high-density imputation and islet-specific epigenome maps', *Nat. Genet.* **50**(11), 1505–1513.

McLaren, W., Gil, L., Hunt, S. E., Riat, H. S., Ritchie, G. R., Thormann, A., Flicek, P. and Cunningham, F. (2016), 'The Ensembl variant effect predictor', *Genome Biol.* **17**(1), 122.

Mittal, S., Madigan, D., Burd, R. S. and Suchard, M. A. (2014), 'High-dimensional, massive sample-size Cox proportional hazards regression for survival analysis', *Biostatistics* **15**(2), 207–221.

Munshi, A. (2009), The OpenCL specification, *in* '2009 IEEE HCS', IEEE, pp. 1–314.

Nakano, J. (2012), Parallel computing techniques, *in* 'Handbook of Computational Statistics', Springer, pp. 243–271.

Negahban, S. N., Ravikumar, P., Wainwright, M. J. and Yu, B. (2012), 'A unified framework for high-dimensional analysis of M-estimators with decomposable regularizers', *Statist. Sci.* **27**(4), 538–557.

NERSC (2021), 'Distributed TensorFlow', https://docs.nersc.gov/machinelearning/tensorflow/#distributed-tensorflow. Accessed: 2021-07-03.

Ng, M. C., Shriner, D., Chen, B. H., Li, J., Chen, W.-M., Guo, X., Liu, J., Bielinski, S. J., Yanek, L. R., Nalls, M. A. et al. (2014), 'Meta-analysis of genome-wide association studies in african americans provides insights into the genetic architecture of type 2 diabetes', *PLoS Genet.* **10**(8), e1004517.

Nitanda, A. (2014), Stochastic proximal gradient descent with acceleration techniques, *in* 'NeurIPS 2014', Vol. 27 of *Adv. Neural Inform. Process. Syst.*, pp. 1574–1582.

NVIDIA (2021*a*), 'Basic linear algebra subroutines (cuBLAS) library', http://docs.nvidia.com/cuda/cublas. Accessed: 2021-07-03.

NVIDIA (2021*b*), 'Sparse matrix library (cuSPARSE)', http://docs.nvidia.com/cuda/cusparse. Accessed: 2021-07-03.

Ohn, I. and Kim, Y. (2019), 'Smooth function approximation by deep neural networks with general activation functions', *Entropy* **21**(7), 627.

Owens, J. D., Luebke, D., Govindaraju, N., Harris, M., Krüger, J., Lefohn, A. E. and Purcell, T. J. (2007), A survey of general-purpose computation on graphics hardware, *in* 'Computer Graphics Forum', Vol. 26, Wiley Online Library, pp. 80–113.

Parikh, N. and Boyd, S. (2014), 'Proximal algorithms', *Found. Trends Optim.* **1**(3), 127–239.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L. et al. (2019), Pytorch: An imperative style, high-performance deep learning library, *in* 'NeurIPS 2019', Vol. 32 of *Adv. Neural Inform. Process. Syst.*, pp. 8026–

8037. Software available from https://pytorch.org.

Polson, N. G., Scott, J. G. and Willard, B. T. (2015), 'Proximal algorithms in statistics and machine learning', *Statist. Sci.* **30**(4), 559–581.

Qi, L. and Sun, J. (1993), 'A nonsmooth version of Newton's method', *Math. Program.* **58**(1-3), 353–367.

Qian, X., Qu, Z. and Richtárik, P. (2019), SAGA with arbitrary sampling, *in* 'ICML 2019', Vol. 97 of *Proc. Mach. Learn. Res.*, pp. 5190–5199.

R Core Team (2021), *R: A Language and Environment for Statistical Computing*, R Foundation for Statistical Computing, Vienna, Austria. https://www.R-project.org/. Version 4.1.0. Accessed: 2021-07-03.

Raina, R., Madhavan, A. and Ng, A. Y. (2009), Large-scale deep unsupervised learning using graphics processors, *in* 'ICML 2009', ACM, pp. 873–880.

Ramdas, A. and Tibshirani, R. J. (2016), 'Fast and flexible ADMM algorithms for trend filtering', *J. Comput. Graph. Statist.* **25**(3), 839–858.

Reyes, A. R. (2021), 'rTorch', https://f0nzie.github.io/rTorch/. Accessed: 2021-07-03.

Reyes-Ortiz, J. L., Oneto, L. and Anguita, D. (2015), Big data analytics in the cloud: Spark on Hadoop vs MPI/OpenMP on Beowulf., *in* 'INNS Conference on Big Data', Vol. 8, p. 121.

Richtárik, P. and Takáč, M. (2016*a*), 'On optimal probabilities in stochastic coordinate descent methods', *Optim. Lett.* **10**(6), 1233–1243.

Richtárik, P. and Takáč, M. (2016*b*), 'Parallel coordinate descent methods for big data optimization', *Math. Program.* **156**(1-2), 433–484.

Robbins, H. and Monro, S. (1951), 'A stochastic approximation method', *Ann. Math. Statistics* **22**, 400–407.

Roland, C., Varadhan, R. and Frangakis, C. (2007), 'Squared polynomial extrapolation methods with cycling: an application to the positron emission tomography problem', *Numer. Algorithms* **44**(2), 159–172.

Rosasco, L., Villa, S. and Vũ, B. C. (2019), 'Convergence of stochastic proximal gradient algorithm', *Appl. Math. Optim.* pp. 1–27.

RStudio (2021), 'R interface to TensorFlow', https://tensorflow.rstudio.com/. Version 2.5.0. Accessed: 2021-07-03.

Rudin, L. I., Osher, S. and Fatemi, E. (1992), 'Nonlinear total variation based noise removal algorithms', *Physica D* **60**(1), 259–268.

Rumelhart, D. E., Hinton, G. E. and Williams, R. J. (1986), 'Learning representations by back-propagating errors', *Nature* **323**, 533–536.

Ryu, E. K., Ko, S. and Won, J.-H. (2020), 'Splitting with near-circulant linear systems: applications to total variation CT and PET', *SIAM J. Sci. Comput.* **42**(1), B185–B206.

Schmidt-Hieber, J. et al. (2020), 'Nonparametric regression using deep neural networks with ReLU activation function', *Ann. Statist.* **48**(4), 1875–1897.

Scott, L. J., Mohlke, K. L., Bonnycastle, L. L., Willer, C. J., Li, Y., Duren, W. L., Erdos, M. R., Stringham, H. M., Chines, P. S., Jackson, A. U. et al. (2007), 'A genome-wide association study of type 2 diabetes in finns detects multiple susceptibility variants', *Science* **316**(5829), 1341–1345.

Seide, F. and Agarwal, A. (2016), CNTK: Microsoft's open-source deep-learning toolkit, *in* 'SIGKDD 2016', ACM, pp. 2135–2135.

Sergeev, A. and Del Balso, M. (2018), 'Horovod: fast and easy distributed deep learning in tensorflow', *arXiv preprint arXiv:1802.05799* .

Staples, G. (2006), Torque resource manager, *in* 'SC 2006', ACM, p. 8.

Suchard, M. A., Holmes, C. and West, M. (2010), 'Some of the what?, why?, how?, who? and where? of graphics processing unit computing for Bayesian analysis', *Bulletin of the International Society for Bayesian Analysis* **17**(1), 12–16.

Suchard, M. A., Simpson, S. E., Zorych, I., Ryan, P. and Madigan, D. (2013), 'Massive parallelization of serial inference algorithms for a complex generalized linear model', *ACM Trans. Model. Comput. Simul.* **23**(1), 1–23.

Suchard, M. A., Wang, Q., Chan, C., Frelinger, J., Cron, A. and West, M. (2010), 'Understanding GPU programming for statistical computation: Studies in massively parallel massive mixtures', *J. Comput. Graph. Statist.* **19**(2), 419–438.

Sudlow, C., Gallacher, J., Allen, N., Beral, V., Burton, P., Danesh, J., Downey, P., Elliott, P., Green, J., Landray, M. et al. (2015), 'UK Biobank: an open access resource for identifying the causes of a wide range of complex diseases of middle and old age', *PLoS Med.* **12**(3), e1001779.

Suzuki, T. (2019), Adaptivity of deep ReLU network for learning in Besov and mixed smooth Besov spaces: optimal rate and curse of dimensionality, *in* 'ICLR 2019'.

Tang, H., Yu, C., Lian, X., Zhang, T. and Liu, J. (2019), `DoubleSqueeze`: Parallel stochastic gradient descent with double-pass error-compensated compression, *in* 'ICML 2019', Vol. 97 of *Proc. Mach. Learn. Res.*, pp. 6155–6165.

The Wellcome Trust Case Control Consortium (2007), 'Genome-wide association study of 14,000 cases of seven common diseases and 3,000 shared controls', *Nature* **447**, 661–678.

Theano Development Team (2016), 'Theano: A Python framework for fast computation of mathematical expressions', *arXiv preprint arXiv:1605.02688* .

Tibshirani, R. (1996), 'Regression shrinkage and selection via the lasso', *J. R. Stat. Soc. Ser. B. Stat. Methodol.* **58**(1), 267–288.

Tibshirani, R. J. and Taylor, J. (2011), 'The solution path of the generalized lasso', *Ann. Statist.* **39**(3), 1335–1371.

Tieleman, T. (2010), Gnumpy: an easy way to use GPU boards in Python, Technical Report UTML TR 2010–002, Department of Computer Science, University of Toronto.

Tseng, P. and Yun, S. (2009), 'A coordinate gradient descent method for nonsmooth separable minimization', *Math. Program.* **117**(1-2), 387–423.

University of Zurich (2021), 'ElastiCluster', https://elasticluster.readthedocs.io/en/latest/. Accessed: 2021-07-03.

Ushey, K., Allaire, J. and Tang, Y. (2021), *reticulate: Interface to 'Python'*. https://CRAN.R-project.org/package=reticulate. Version 1.20. Accessed: 2021-07-03.

Van De Geijn, R. A. and Watts, J. (1997), 'SUMMA: Scalable universal matrix multiplication algorithm', *Concurrency: Practice and Experience* **9**(4), 255–274.

van Rossum, G. (1995), Python tutorial, Technical Report CS-R9526, Centrum voor Wiskunde en Informatica (CWI), Amsterdam.

Voight, B. F., Scott, L. J., Steinthorsdottir, V., Morris, A. P., Dina, C., Welch, R. P., Zeggini, E., Huth, C., Aulchenko, Y. S., Thorleifsson, G. et al. (2010), 'Twelve type 2 diabetes susceptibility loci identified through large-scale association analysis', *Nat. Genet.* **42**(7), 579.

Vũ, B. C. (2013), 'A splitting algorithm for dual monotone inclusions involving cocoercive operators', *Adv. Comput. Math.* **38**(3), 667–681.

Walker, E. (2008), 'Benchmarking Amazon EC2 for hig-performance scientific computing', *;login:: the Magazine of USENIX & SAGE* **33**(5), 18–23.

Wang, E., Zhang, Q., Shen, B., Zhang, G., Lu, X., Wu, Q. and Wang, Y. (2014), Intel Math Kernel library, *in* 'High-Performance Computing on the Intel® Xeon Phi$^{\text{TM}}$', Springer, pp. 167–188.

Wang, J., Kolar, M., Srebro, N. and Zhang, T. (2017), Efficient distributed learning with sparsity, *in* 'ICML 2017', Vol. 70 of *Proc. Mach. Learn. Res.*, pp. 3636–3645.

Won, J.-H. (2020), Proximity operator of the matrix perspective function and its applications, *in* 'NeurIPS 2020', Vol. 33 of *Adv. Neural Inform. Process. Syst.*

Wright, S. J. (2015), 'Coordinate descent algorithms', *Math. Program.* **151**(1), 3–34.

Wu, T. T. and Lange, K. (2010), 'The MM alternative to EM', *Statist. Sci.* **25**(4), 492–505.

Xiao, L. and Zhang, T. (2014), 'A proximal stochastic gradient method with progressive variance reduction', *SIAM J. Optim.* **24**(4), 2057–2075.

Xue, L., Ma, S. and Zou, H. (2012), 'Positive-definite ℓ_1-penalized estimation of large covariance matrices', *J. Amer. Statist. Assoc.* **107**(500), 1480–1491.

Yoo, A. B., Jette, M. A. and Grondona, M. (2003), Slurm: Simple linux utility for resource management, *in* 'JSSPP 2003', Springer, pp. 44–60.

Yu, D., Won, J.-H., Lee, T., Lim, J. and Yoon, S. (2015), 'High-dimensional fused lasso regression using majorization–minimization and parallel processing', *J. Comput. Graph. Statist.* **24**(1), 121–153.

Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S., Stoica, I. et al. (2010), 'Spark: Cluster computing with working sets.', *HotCloud* **10**(10-10), 95.

Zhang, X., Wang, Q. and Chothia, Z. (2021), 'OpenBLAS: An optimized BLAS library', https://www.openblas.net/. Accessed: 2021-07-03.

Zhang, Y., Duchi, J. C. and Wainwright, M. J. (2013), 'Communication-efficient algorithms for statistical optimization', *J. Mach. Learn. Res.* **14**(1), 3321–3363.

Zhou, H., Lange, K. and Suchard, M. A. (2010), 'Graphics processing units and high-dimensional optimization', *Statist. Sci.* **25**(3), 311–324.

Zhu, M. and Chan, T. (2008), An efficient primal-dual hybrid gradient algorithm for total variation image restoration, Technical Report 08-34, UCLA CAM.

Zhu, Y. (2017), ‘An augmented ADMM algorithm with application to the generalized lasso problem’, *J. Comput. Graph. Statist.* **26**(1), 195–204.