이학박사 학위논문

Easily parallelizable statistical computing methods and their applications in modern high-performance computing environments

병렬화 용이한 통계계산 방법론과 현대 고성능 컴퓨팅 환경에의 적용

2020년 8월

서울대학교 대학원

통계학과

고 세 윤

이학박사 학위논문

Easily parallelizable statistical computing methods and their applications in modern high-performance computing environments

병렬화 용이한 통계계산 방법론과 현대 고성능 컴퓨팅 환경에의 적용

2020년 8월

서울대학교 대학원

통계학과

고 세 윤

Easily parallelizable statistical computing methods and their applications in modern high-performance computing environments

병렬화 용이한 통계계산 방법론과 현대 고성능 컴퓨팅 환경에의 적용

지도교수 원 중 호

이 논문을 이학박사 학위논문으로 제출함

2020 년 6 월

서울대학교 대학원

통계학과

고 세 윤

고세윤의 이학박사 학위논문을 인준함

2020 년 6 월

위 원 장	박 태 성	(인)
부위원장	원 중 호	(인)
위 원	임 요 한	(인)
위 원	김 용 대	(인)
위 원	유 동 현	(인)

Abstract

Seyoon Ko
Department of Statistics
The Graduate School
Seoul National University

Technological advances in the past decade, hardware and software alike, have made access to high-performance computing (HPC) easier than ever. In this dissertation, easily-parallelizable, inversion-free, and variable-separated algorithms and their implementation in statistical computing are discussed. The first part considers statistical estimation problems under structured sparsity posed as minimization of a sum of two or three convex functions, one of which is a composition of non-smooth and linear functions. Examples include graph-guided sparse fused lasso and overlapping group lasso. Two classes of inversion-free primal-dual algorithms are considered and unified from a perspective of monotone operator theory. From this unification, a continuum of preconditioned forward-backward operator splitting algorithms amenable to parallel and distributed computing is proposed. The unification is further exploited to introduce a continuum of accelerated algorithms on which the theoretically optimal asymptotic rate of convergence is obtained. For the second part, easy-to-use distributed matrix data structures in PyTorch and Julia are presented. They 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. With these data structures, various parallelizable statistical applications, including nonnegative matrix factorization, positron emission tomography, multidimensional scaling, and ℓ_1-regularized Cox regression, are demonstrated. The

examples scale up to an 8-GPU workstation and a 720-CPU-core cluster in a cloud. As a case in point, the onset of type-2 diabetes from the UK Biobank with 400,000 subjects and about 500,000 single nucleotide polymorphisms is analyzed using the HPC ℓ_1-regularized Cox regression. Fitting a half-million-variate model took about 50 minutes, reconfirming known associations. To my knowledge, the feasibility of a joint genome-wide association analysis of survival outcomes at this scale is first demonstrated.

Keywords: monotone operator theory, primal-dual algorithms, high-performance computing, multi-GPU, distributed computing, cloud computing

Student Number: 2014-30997

Contents

List of Figures

List of Tables

Chapter 1

Prologue

1.1 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 it is likely that there will be no major breakthrough until quantum computing becomes practical. Now the increase in computing power is 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 graphical processing units (GPUs) are attached to the CPU for orders of magnitude of acceleration for some tasks. A GPU can be thought of a massively parallel matrix-vector multiplier and vector transformer on a data stream. With the needs of analyzing terabyte- or even

petabyte-scale data common, the success of large-scale statistical computing heavily relies on how to engage HPC in the statistical practice.

About a decade ago, Zhou et al. (2010) discussed the potential of GPUs in statistical computing. In this landmark paper, the authors predicted that "GPUs will fundamentally alter the landscape of computational statistics." Yet, it appears that GPU computing, or HPC in general, has not completely smeared into the statistical community. Part of the reasons for this may be attributed to the fear that parallel and distributed code is difficult to program, especially in R (R Core Team, 2018), "the" programming language of statisticians. 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, e.g., 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 storing 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). 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 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 the smooth approximation approaches, e.g., Hunter and Li (2005). Decomposability or separability of variables is often the key to parallel and distributed algorithms. The popular alternating direction method of multipliers (ADMM, Gabay and Mercier, 1976; Glowinski and Marroco, 1975; Boyd et al., 2010) 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). Another route toward separability is through the 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).

This dissertation reviews the advances in parallel and distributed computing environments during the last decade, and develops easily parallelizable algorithms for statistical computing. In particular, two classes of easily parallelizable optimization algorithms suitable to statistical estimation of structually sparse models are unified, and accelerated to the asymptotic optimum (Chapter 2). In addition, software packages are developed to make programming for large-scale, high-dimensional statistical models easy for statisticians. These packages scale up to about $400,000 \times 500,000$ multivariate analysis for Cox regression

model regularized by the ℓ_1 penalty on the UK Biobank genomics data, featuring time-to-onset of Type 2 Diabetes (T2D) as outcome and genomic loci harboring single nucleotide polymorphisms as covariates (Chapter 3). To my knowledge, such a large-scale joint genome-wide association analysis with the Cox model has not been attempted. The dissertation is concluded in Chapter 4.

The rest of this chapter reviews HPC systems and how they have become easy to use (Section 1.2), and modern scalable optimization techniques that suit well to the HPC environment (Section 1.3).

1.2 Accessible High-Performance Computing Systems

1.2.1 Preliminaries

Since modern HPC relies on parallel computing, in this section several concepts from parallel computing literature are reviewed 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 split into several chunks 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 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 with 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, if two processors refer to a variable x, that means the variable is stored in the same memory address; if a processor alters the variable, then the other processor is affected by the changed 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 the only way processors can exchange information with each other is by passing data through some explicit communication mechanism. 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). 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 the 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 of 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 like by using OpenMP. 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 data as an array, and each other processor receives a partition of the array;

- Gather: one processor collects data from all the other processors to construct an array;

- Broadcast: one processor sends its data to all the other devices;

- Reduce: one processor gathers data and produces a combined output based on an associative binary operator, such as sum or maximum of all the elements.

Parallel architectures. To realize the above models, a computer architecture that allows simultaneous execution of multiple machine language instructions is required. A 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 that share memory. A 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.

1.2.2 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 to determine the topology and to purchase all the required hardware, along with resources to maintain it. This is certainly not familiar to 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 thousands of 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), Torque (Staples, 2006), etc. However, access to supercomputers is almost always limited. (Can you name a "nearby" HPC center from your work? If so, how can you submit your job request? What is the cost?) 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 has emerged as a third option. It refers to both the applications delivered as services over the Internet and the hardware

and systems software in the data centers that provide those services (Armbrust et al., 2010). Big information technology companies 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 using 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 that of 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 1.2.1: 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 did not involve frequent data transmissions between computing instances, and were less optimized for HPC, yielding discouraging results (Evangelinos and Hill, 2008; Walker, 2008).

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, 2019) and ElastiCluster (University of Zurich, 2019). These tools are designed to run programs in clouds in a similar manner to proprietary clus-

ters 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. Accordingly, 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 are in demand for high performance, not with such a steep learning curve.

1.2.3 Multi-GPU node

In some cases, HPC is achieved by installing multiple GPUs on a single node. Over the past two decades, GPUs have gained a sizable amount of popularity among scientists. GPUs were originally designed to aid CPUs in rendering graphics for video games quickly. 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, realizing the SIMD architecture. While this architecture of GPUs was created in need of generating a large number of pixels in a limited time due to the frame rate constraint of high-quality video games, 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 et al., 2010a,b) and high-dimensional optimiza-

tion (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 operations are also supported. High-end GPUs are now being designed specifically for scientific computing purposes, sometimes with fault-tolerance features such as error correction.

A major drawback of GPUs for statistical computing is that GPUs have a smaller memory compared to CPU, and it is slow to transfer data between them. Using multiple GPUs can be a cure: 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. (Lee et al. (2017) explored this possibility in image-based regression.) 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 for the cost. Linear algebra operations that frequently occur in high-dimensional optimization are good examples.

Programming environments for GPU computing have been notoriously hostile to programmers for a long time. The major sophistication 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(s). 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., CUDA (Nvidia, 2007) or OpenCL (Munshi, 2009). Combinations of these technical barriers made ca-

sual programmers, e.g., statisticians, keep away 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 user base big enough to maintain the community in general. 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 in 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., 1988). These computational features fit well to the SIMD architecture of GPUs, whose use 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 had 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 reviewed in the next section, programming with those libraries gets rid of many hassles with GPUs, close to the level of conventional programming.

Readers might ask: why should statisticians care about deep learning software? As Cheng and Titterington (1994) pointed out 25 years ago, "neural networks provide a representational framework for familiar statistical constructs," and "statistical techniques are sometimes implementable using neural-network

technology." For example, linear regression is just a simple neural network with a single layer and linear activation functions. Many more sophisticated statistical frameworks can be mapped to that of neural networks and can benefit from those easy-to-use deep learning libraries for computational performance boosting.

1.3 Highly Parallelizable Algorithms

In this section, some easily parallelizable optimization algorithms useful for fiting high-dimensional statistical models are discussed, 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.

1.3.1 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. The EM algorithm (Dempster et al.,

1977) is an instance of the MM principle. In order to maximize the marginal loglikelihood

$$\ell(\theta) = \log \int p_\theta(o, z) dz,$$

where o is the observed data, z is unobserved missing data, and θ is the parameter to estimate, we maximize the surrogate function

$$Q(\theta|\theta^n) = \mathbb{E}_{Z|X,\theta^n} \left[\log p_\theta(o, z)\right] = \int \log \left[p_\theta(o, z)\right] p_{\theta^n}(z|o) dz,$$

since

$$\begin{aligned}
\ell(\theta) &= \log \int p_\theta(o, z) dz = \log \int \frac{p_\theta(o) p_\theta(z|o)}{p_{\theta^n}(z|o)} p_{\theta^n}(z|o) dz \\
&\geq \int \log \left[\frac{p_\theta(o) p_\theta(z|o)}{p_{\theta^n}(z|o)}\right] p_{\theta^n}(z|o) dz \\
&= Q(\theta|\theta^n) - \int \log \left[p_{\theta^n}(z|o)\right] p_{\theta^n}(z|o) dz
\end{aligned}$$

by Jensen's inequality, and the second term in the last inequality is irrelavent to θ. (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 the 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 $\log(\sum_{i=1}^p u_i)$ often arises. By Jensen's inequlity, this term can be minorized and separated as

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

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 environments, e.g. multi-GPU systems.

1.3.2 Proximal gradient descent

The proximal gradient descent method is an extension of the gradient descent method, which deals with minimization of sum of two convex functions, i.e.,

$$\min_x f(x) + g(x).$$

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

We first define the proximity operator of g:

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

For many functions their proximity operators take closed forms. We say such functions "proximable". For example, consider the $0/\infty$ indicator function of a closed convex set C

$$\delta_C(x) = \begin{cases} 0, & x \in C \\ +\infty, & x \notin C \end{cases}.$$

The corresponding proximity operator is the Euclidean projection onto C: $P_C(y) = \arg\min_{x \in C} \|y - x\|_2$. The proximity operator of the ℓ_1-norm $\lambda\|\cdot\|_1$ is the soft-thresholding operator:

$$[\mathcal{S}_\lambda(y)]_i := \text{sign}(y_i)(|y_i| - \lambda)_+$$

For many sets, e.g., nonnegative orthant, P_C is simple to compute.

Now we proceed with the proximal gradient descent for minimization of $\mathcal{F}(x) = f(x) + g(x)$. Assume f is convex and has L-Lipschitz gradients, i.e., $\|\nabla f(x) - \nabla f(y)\|_2 \leq L\|x - y\|_2$ 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 following surrogate function that majorizes h:

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

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

$$x^{n+1} = \mathbf{prox}_{\gamma_n g}\left(x^n - \gamma_n \nabla f(x^n)\right), \quad \gamma_n \in \left(0, \frac{1}{L}\right]. \tag{1.1}$$

This update guarantees a nonincreasing sequence of $\mathcal{F}(x^n)$ by the MM principle. Proximal gradient method also has an interpretation of forward-backward operator splitting, and the step size $\gamma_n \in \left(0, \frac{2}{L}\right)$ guarantees convergence (Combettes and Pesquet, 2011; Bauschke and Combettes, 2011; Combettes, 2018). If $g(x) = \delta_C(x)$, then the corresponding algorithm is called the projected gradient method. If $g(x) = \lambda\|x\|_1$, then the corresponding algorithm is the iterative shrinkage-thresholding algorithm (ISTA, Beck and Teboulle, 2009). For many functions g, the update (1.1) is simple and easily parallelized, thus the algorithm is suitable for HPC computing. For example, the soft-thresholding operator is elementwise hence the updates are independent. In addition, if $g(x) = -a\log x$, then

$$\mathbf{prox}_{\gamma g}(y) = \frac{y + \sqrt{y^2 + 4\gamma a}}{2}. \tag{1.2}$$

This proximity operator is useful for the example in Section 3.5.2. See Parikh and Boyd (2014) for a thorough review and distributed-memory implementations, and Polson et al. (2015) for a statistics-oriented review.

1.3.3 Proximal distance algorithm

Proximal distance algorithm (Keys et al., 2019) is a recent addition to the class of MM algorithms that deserved a separate treatment. This algorithm is an interplay of the penalty method for constrained minimization and distance majorization (Chi et al., 2014). Consider the minimization problem for a convex, closed, and proper[1] function f in a constraint set $C = \cap_{i=1}^{k} C_i$, where $C_1, \ldots, C_k$ are closed. Either convexity of C_i or differentiability of f is required. A choice for the penalty function would be $q(x) = \frac{1}{2k} \sum_{i=1}^{k} \operatorname{dist}(x, C)^2$, where $\operatorname{dist}(x, C) = \inf_{y \in C} \|x - y\|_2$ so that a minimizer x_ρ of the unconstrained problem $\min_x f(x) + \rho q(x)$ is found. If ρ is sent to infinity, x_ρ would tend to the solution for the original constrained optimization. Distance majorization is achieved by $\|x - P_{C_i}(x^n)\|^2 \geq \operatorname{dist}^2(x, C_i)$, hence the surrogate function g_ρ that majorizes $f(x) + \rho q(x)$ is defined by

$$\begin{aligned} g_\rho(x|x^n) &= f(x) + \frac{\rho}{2k} \sum_{i=1}^{k} \|x - P_{C_i}(x^n)\|^2 \\ &= f(x) + \frac{\rho}{2} \left\| x - \sum_{i=1}^{k} P_{C_i}(x^n) \right\|^2 + \text{const.} \end{aligned}$$

By definition of proximity operator, the minimum of $g_\rho(x|x^n)$ occurs at $x^{n+1} = \operatorname{prox}_{\rho^{-1} f}\left[\frac{1}{k} \sum_{i=1}^{k} P_{C_i}(x^n)\right]$. When C is convex, P_C is single-valued and the following holds:

$$\nabla \frac{1}{2} \operatorname{dist}(x, C)^2 = x - P_C(x).$$

[1]For a convex function $f : X \to \mathbb{R} \cup \{\pm\infty\}$, f is proper if $f(x) < \infty$ for some x and $f(x) > -\infty$ for any $x \in X$.

Since the proximity operator is nonexpansive, the gradient of the function q is 1-Lipschitz. This results in the proximal distance update $x^{n+1} = \text{prox}_{\rho^{-1}f}[x^n - \nabla q(x^n)]$, showing that the proximal distance method is a case of the proximal gradient method for convex C_i's. Parallel computing can be applied if the proximity operators involved and projection onto the convex set C_i's are easily parallelized, e.g., ℓ_2 and ℓ_∞ norm balls or nonnegative orthants.

1.3.4 Primal-dual methods

The algorithms discussed so far are primal methods. Primal-dual methods introduce additional dual variables but can deal with a larger class of problems. Consider the problems of minimizing

$$\mathcal{F}(x) = f(x) + h(Kx), \tag{1.3}$$

where K is a linear map. We further assume that f and h are lower semicontinuous, convex, and proper functions. Even if h is proximable, the proximity operator for $h(K\cdot)$ is not easy to compute. Define the convex conjugate of h as $h^*(y) = \sup_x \langle x, y\rangle - h(x)$. It is known that $h^{**} = h$ since h is lower semicontinuous and convex, so $h(Kx) = h^{**}(Kx) = \sup_y \langle Kx, y\rangle - h^*(y)$. Then the minimization problem $\inf_x f(x) + h(Kx)$ is equivalent to the saddle-point problem

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

Under mild conditions (Theorem 19.1 and Proposition 19.18, Bauschke and Combettes, 2011), strong duality

$$\inf_x \sup_y \langle Kx, y\rangle + f(x) - h^*(y) = \sup_y \inf_x \langle x, K^T y\rangle + f(x) - h^*(y)$$

holds and the saddle point $(\hat{x}, \hat{y})$ satisfies the optimality conditions

$$K\hat{x} - \partial h^*(\hat{y}) \ni 0 \text{ and } K^T\hat{y} + \partial f(\hat{x}) \ni 0,$$

where $\partial\phi$ denotes the subdifferential of a convex function ϕ. The vector y is the dual variable and the maxmin problem

$$\sup_y \inf_x \langle x, K^T y\rangle + f(x) - h^*(y) = \sup_y -h^*(y) - f^*(-K^T y)$$

is called the dual of the original (primal) minimization problem.

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

$$x^{n+1} = \arg\min_x f(x) + (t/2)\|Kx - \tilde{x}^n + (1/t)y^n\|_2^2 \tag{1.4a}$$

$$\tilde{x}^{n+1} = \mathbf{prox}_{(1/t)h}(Kx^{n+1} + (1/t)y^n) \tag{1.4b}$$

$$y^{n+1} = y^n + t(Kx^{n+1} - \tilde{x}^{n+1}). \tag{1.4c}$$

The update (1.4a) is *not* a proximity operator, as the quadratic term is not spherical. It defines an inner optimization problem that is often nontrivial. In the simplest case of f being linear or quadratic (which arises in linear regression), (1.4a) involves solving a linear system. While it is plausible to obtain the inverse of the involved matrix once and reuse it for future iterations, inverting a matrix even once quickly becomes intractable in the high-dimensional setting, as its time complexity is cubic in the number of variables.

The primal-dual hybrid gradient method (PDHG, Zhu and Chan, 2008; Esser et al., 2010; Chambolle and Pock, 2011) avoids such inversion via the following iteration:

$$y^{n+1} = \mathbf{prox}_{\sigma h^*}(y^n + \sigma K\bar{x}^n) \tag{1.5a}$$

$$x^{n+1} = \mathbf{prox}_{\tau f}(x^n - \tau K^T y^{n+1}) \tag{1.5b}$$

$$\bar{x}^{n+1} = 2x^{n+1} - x^n, \tag{1.5c}$$

where (1.5a) and (1.5b) are dual ascent and primal descent steps, respectively; σ and τ are step sizes. The last step (1.5c) corresponds to the extrapolation. If h is proximable, so is h^*, since $\mathbf{prox}_{\gamma h^*}(x) = x - \gamma \mathbf{prox}_{\gamma^{-1} h}(\gamma^{-1} x)$ by Moreau's decomposition. This method has been studied using monotone operator theory (Condat, 2013; Vũ, 2013), introduced in Appendix A. Convergence of iteration (1.5) is guaranteed if $\sigma\tau\|K\|_2^2 < 1$, where $\|M\|_2$ is the spectral norm of matrix M. If f has L-Lipschitz gradients, then the proximal step (1.5b) can be replaced by a gradient step

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

The PDHG algorithms are also highly parallelizable as long as the involved proximity operators are easy to compute and separable; no matrix inversion is involved in iteration (1.5) and only matrix-vector multiplications appear. In Chapter 2, we consider a three-function variant of this problem for application on statistical estimation problems with structured sparsity.

Chapter 2

Easily Parallelizable and Distributable Class of Algorithms for Structured Sparsity, with Optimal Acceleration

2.1 Introduction

As discussed in Chapter 1, many statistical learning problems can be formulated as an optimization problem of the three-function variant of (1.3) discussed in Section 1.3.4:

$$\min_{x \in \mathbb{R}^p} \; f(x) + g(x) + h(Kx), \tag{2.1}$$

where $K \in \mathbb{R}^{l \times p}$, and f , g, and h are closed, proper, and convex. In this chapter, it is assumed that f is differentiable and its gradient ∇f is Lipschitz continuous with modulus L_f; g and h are not necessarily smooth. We further assume that $\|K\|_2 \leq L_K$. As discussed in Section 1.3.4 in Chapter 1, (2.1) has a solution under a mild condition. If $(x^\star, y^\star)$ is a solution, then it is a saddle

point for the saddle point formulation of (2.1):

$$\min_{x\in\mathbb{R}^p}\max_{y\in\mathbb{R}^l}\mathcal{L}(x,y) \tag{2.2}$$

where $\mathcal{L}(x,y)=f(x)+g(x)+\langle Kx,y\rangle-h^*(y)$ is the saddle function. Also the strong duality holds: $x^\star$ is a primal solution to (2.1), and $y^\star$ is a solution to the associated dual (Condat, 2013):

$$\max_{y\in\mathcal{Y}}\left(-(f+g)^*(-K^Ty)-h^*(y)\right). \tag{2.3}$$

In the sequel, we assume that (2.2) has a solution, and seek an algorithm that finds it. It is shown how to solve (2.1) in a fashion that is easy to parallelize or distribute on modern high-performance computing environment such as workstations equipped with multiple graphics processing units (GPUs).

A pinnacle instance of (2.1) is high-dimensional penalized regression with structured sparsity penalty:

$$\min_{x\in\mathbb{R}^p}\quad\sum_{i=1}^n l_i(a_i^Tx,b_i)+\lambda_1\|x\|_1+H(Dx), \tag{2.4}$$

with direct identification $f(x)=\sum_{i=1}^n l_i(a_i^Tx;b_i)$, $g(y)=\lambda_1\|x\|_1$, $h(u)=H(u)$, and $K=D$, where the set $\{(a_i,b_i):a_i\in\mathbb{R}^p,b_i\in\mathbb{R},i=1,\dots,n\}$ constitutes a training sample, $l_i:\mathbb{R}^2\to\mathbb{R}$ is the loss function that may depend on the sample index, $D\in\mathbb{R}^{l\times p}$ is the structure-inducing matrix, and H is the penalty function, which is typically non-smooth. Loss functions with Lipschitz gradients arise in many important problems: in linear regression we have $f(x)=(1/2)\|\mathsf{A}x-b\|_2^2$ and the gradient $\nabla f(x)=\mathsf{A}^T(\mathsf{A}x-b)$ is $\|\mathsf{A}^T\mathsf{A}\|_2$-Lipschitz, where $\mathsf{A}=[a_1,\dots,a_n]^T$ denotes the data matrix; in logistic regression $f(x)=-\sum_{i=1}^n\left(b_i(a_i^Tx)+\log(1+e^{a_i^Tx})\right)$ has $(1/4)\|\mathsf{A}^T\mathsf{A}\|_2$-Lipschitz gradients. Choosing the ℓ_1-penalty $H(z)=\lambda\|z\|_1$ for some $\lambda>0$ yields the generalized

lasso (Tibshirani and Taylor, 2011) with sparsity of variables, which includes a sparse version of fused lasso (Tibshirani et al., 2005) as a special case. For the group lasso (Yuan and Lin, 2006) with $\mathcal{G}$ possibly *overlapping* groups, we can choose $H(y) = \lambda_1\|y_{[1]}\|_q + \cdots + \lambda_{\mathcal{G}}\|y_{[\mathcal{G}]}\|_q$ for $y = (y_{[1]}^T, \ldots, y_{[\mathcal{G}]}^T)^T$, where $[g] \subset \{1, 2, \ldots, p\}$ is a given set of group indexes and $y_{[g]} \in \mathbb{R}^{|[g]|}$ for each $g = 1, 2, \ldots, \mathcal{G}$; $\|\cdot\|_q$ denotes the ℓ_q norm with $q > 1$. Now set D as a $(|[1]| + \cdots + |[\mathcal{G}]|) \times p$ binary matrix with a single one (*1*) in each row; the 1 corresponds to the group membership. Then, $H(Dx) = \lambda_1\|x_{[1]}\|_q + \cdots + \lambda_{\mathcal{G}}\|x_{[\mathcal{G}]}\|_q$ as desired; D has a column with more than a single nonzero entry if and only if there is an overlapping group. Judicious choices of f, g, h, and K in (2.1) allow more flexibility in solving (2.4). In particular, non-smooth loss functions, such as the hinge loss, can also be handled. More complex penalty functions such as the latent group lasso (Jacob et al., 2009) are also allowed in (2.4), as shown below.

More than one penalty. When (2.4) involves more than one penalty with different linear operators, the problem can be formulated as (2.1) by augmenting the dual variable. Suppose we solve the following penalized regression problem

$$\min_{x \in \mathbb{R}^p} \quad \sum_{i=1}^{n} l_i(a_i^T x, b_i) + \lambda_1\|x\|_1 + H_1(D_1 x) + H_2(D_2 x).$$

Then we can set

$$f(x) = \sum_{i=1}^{n} l_i(a_i^T x, b_i), \quad g(x) = \lambda_1\|x\|_1, \quad h(y_1, y_2) = H_1(y_1) + H_2(y_2),$$

$$K = \begin{bmatrix} D_1 \\ D_2 \end{bmatrix}, \quad y = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix}.$$

It is easy to verify that $\mathbf{prox}_h(v_1, v_2) = (\mathbf{prox}_{H_1}(v_1), \mathbf{prox}_{H_2}(v_2))^T$ due to

separability of h. For example, consider the latent group lasso problem (Jacob et al., 2009). The latent group lasso selects groups less conservatively than the original group lasso (Yuan and Lin, 2006), and allows overlaps. The penalty is defined as

$$H(x) = \inf_{v_{[g]} \in \mathbb{R}^{|[g]|}, D^T v = x} \sum_{g=1}^{\mathcal{G}} \lambda_g \|v_{[g]}\|_q,$$

where $[g]$ and D are the group index set and the membership matrix as discussed in Section 2.1 for the original group lasso. Thus the latent group lasso problem can be written as

$$\min_{x,v} f(x) + g(x) + h(v) + \delta_{\{0\}}(x - D^T v),$$

where $h(v) = \sum_{i=1}^{\mathcal{G}} \lambda_g \|v_{[g]}\|_q$ and δ_S is the indicator function for set S so that $\delta_S(u) = 0$ if $u \in S$ and $\delta_S(u) = +\infty$ otherwise. Let $z = (x^T, v^T)^T$, $\tilde{f}(z) = f(\begin{bmatrix} I & 0 \end{bmatrix} z)$, $\tilde{g}(z) = g(\begin{bmatrix} I & 0 \end{bmatrix} z)$, $\tilde{h}(y_1, y_2) = h(y_1) + \delta_{\{0\}}(y_2)$, and $K = \begin{bmatrix} 0 & I \\ I & -D^T \end{bmatrix}$. We have an equivalent formulation

$$\min_z \tilde{f}(z) + \tilde{g}(z) + \tilde{h}(Kz).$$

It has the form of (2.1). Note that g, h and $\delta_{\{0\}}$ are all proximable.

Elastic net penalties. The elastic net (Zou and Hastie, 2005) regression uses a linear combination of ℓ_1 and ℓ_2 penalties in order to promote both sparsity of solution and the grouping effect that highly correlated variables are selected or unselected together. The relevant optimization problem is

$$\min_{x \in \mathbb{R}^p} \frac{\lambda_2}{2} \|x\|_2^2 + \lambda_1 \|x\|_1 + l(\mathsf{A}x, b), \tag{2.5}$$

where the data matrix A is the same as in the sparse generalized lasso, and

$b = (b_1, \ldots, b_n)^T$. This formulation admits nonsmooth loss function l, e.g., $l(\mathsf{A}x, b) = \|\mathsf{A}x - b\|_2$ (Belloni et al., 2011).

Nonsmooth losses: split dual formulation. In fact, more general formulation with nonsmooth loss is possible. When the loss function l_i in (2.4) does not have Lipschitz gradients yet is closed, proper, and convex, a split-dual formulation (Nesterov, 2005) can be utilized. This includes the case where the loss is not differentiable (e.g. hinge loss). To cope with this, we exploit the saddle-point representation (2.2) of (2.1), and *dualize* the loss function in addition to the penalty. That is, express $\sum_{i=1}^n l_i(a_i^T x; b_i) = \sup_{w \in \mathbb{R}^n} \langle \mathsf{A}x, w \rangle - \sum_{i=1}^n l_i^*(w_i; b_i)$, yielding

$$\min_x \max_{y,w} \ \langle Dx, y \rangle + \langle \mathsf{A}x, w \rangle - \left(\sum_{i=1}^n l_i^*(w_i; b_i) + H^*(y) \right). \tag{2.6}$$

In terms of (2.2), $f(x) \equiv 0$, $K = [D^T, \mathsf{A}^T]^T$, $h^*(y, w) = H^*(y) + \sum_{i=1}^n l_i^*(w_i; b_i)$. Because h^* is separable in y and w, we have

$$\mathbf{prox}_{\sigma h^*}(u, v_1, \ldots, v_n) = (\mathbf{prox}_{\sigma H^*}(u), \mathbf{prox}_{\sigma l_1^*(\cdot; b_1)}(v_1), \ldots, \mathbf{prox}_{\sigma l_n^*(\cdot; b_n)}(v_n)).$$

The cost is that the number of dual variables increases by n. For example, in the linear support vector machine, the proximity operator for the hinge loss $l_i(\cdot; b_i) = \max(0, 1 - b_i \cdot)$ is given by $\mathbf{prox}_{\sigma l_i^*}(v_i) = \max(\min(v_i - \sigma b_i, 0), -b_i)$. Thus computation of $\mathbf{prox}_{\sigma h^*}$ can be conducted in parallel for each element of $v = (v_1, \ldots, v_n)$. Note that this formulation is not limited to the separable losses in (2.4). For example, in the square-root lasso (Belloni et al., 2011), we

solve

$$\min_x \|\mathsf{A}x - b\|_2 + H(Dx) = \min_x \max_{y,w:\|w\|_2\leq 1} \langle Dx, y\rangle + \langle \mathsf{A}x, w\rangle - \left(\langle b, w\rangle + H^*(y)\right), \tag{2.7}$$

yielding $f(x) \equiv 0$, $K = [D^T, \mathsf{A}^T]^T$, $\mathbf{prox}_{\sigma h^*}(u, v) = \big(\mathbf{prox}_{\sigma H^*}(u), P_{\mathcal{B}_2}(v - 2\sigma b)\big)$, where $P_{\mathcal{B}_2}(\cdot)$ denotes the projection to the unit ℓ_2-ball.

PET image reconstruction. In positron emission tomography (PET), photon emissions from a radioactive tracer inside the brain are counted and the location-dependent emission rates are estimated. In this task, the Radon transform (Jain, 1989) is often discretized as matrix A. See Secton 3.5.2 for more details. This results in a regularized nonnegative least squares problem, which can be written as

$$\min_{x\in\mathbb{R}^p} \frac{1}{2}\|\mathsf{A}x - b\|_2^2 + \delta_+(x) + \lambda\|Dx\|_1, \tag{2.8}$$

where x is the unknown emission map (image), b is the vector of counts, and δ_+ is the indicator function of the nonnegative orthant defined by $\delta_+(x) = 0$ if $x_1, \ldots, x_p \geq 0$ and $\delta_+(x) = +\infty$ otherwise. The D is a discrete gradient operator encoding penalty on total variation.

Therefore, ability to solve (2.1) efficiently provides a versatile tool for many important statistical learning problems. In spite of its importance, solving (2.1) is challenging because the non-separability of the non-smooth part hampers use of efficient methods. If $h \equiv 0$, then the proximal gradient method reviewed in Section 1.3.2 is arguably the method of choice, which provides a simple gradient-descent-like iteration

$$x^{k+1} = \arg\min_x f(x^k) + \langle \nabla f(x^k), x - x^k\rangle + \frac{1}{2t}\|x - x^k\|_2^2 + g(x)$$

$$= \mathbf{prox}_{tg}(x^k - t\nabla f(x^k))$$

for $0 < t < 2/L_f$, where $\langle u, v\rangle$ denotes the standard inner product $u^T v$. However, nontrivial h, e.g., group lasso, proximal gradient involves evaluating $\mathbf{prox}_{th\circ K}(\cdot)$, which is nontrivial even for tractable cases (Friedman et al., 2007; Liu et al., 2010b; Xin et al., 2014; Yu et al., 2015). While approximating h by a smooth function has been considered (Nesterov, 2005; Chen et al., 2012), this approach introduces an additional smoothing parameter that is difficult to choose in practice. As reviewed in Section 1.3.4, the popular ADMM can be applied to solve (2.1) with $g \equiv 0$ as well, however, *inner minimization subproblem* (1.4a) is potentially expensive to compute. For example, if f is a loss function for a generalized linear model, then the corresponding update involves solving a linear equation of the form $(\mathsf{A}^T W \mathsf{A} + tK^T K)x = r$, W diagonal, *iteratively*. While K is structured and known *a priori*, the data matrix A is hardly structured. A similar problem arises in medical imaging reconstruction problems, such as undersampled multi-coil MRI reconstruction (Ramani and Fessler, 2011) or sparse-view CT reconstruction (Sidky et al., 2012) using the total variation penalty (Rudin et al., 1992; Goldstein and Osher, 2009). In this case the "measurement matrix" A is large and unstructured. Hence avoiding inner minimization subproblem is crucial in both statistical learning and imaging problems where the problem dimensions are ever increasing. The PDHG and linearized alternating directions method (LADM; Lin et al., 2011) add an additional regularization term to (1.4a) in order to avoid the costly inner minimization subproblem. However, these methods often involve evaluating $\mathbf{prox}_f(\cdot)$, which may lead to another inner minimization subproblem in the presence of A.

The goal of this chapter is to introduce a class of algorithms that requires

neither smoothing nor quadratic minimization. This class of algorithms only involve evaluation of the gradient $\nabla f(x)$, matrix-vector multiplications and simple proximity operators. Thus it is simple to implement and attractive for parallel and distributed computation. We begin with introducing two known algorithms for $g \equiv 0$. One is due to Loris and Verhoeven (2011), later studied by Chen et al. (2013), and Drori et al. (2015):

$$\begin{aligned} \tilde{x}^{k+1} &= x^k - \tau \left(\nabla f(x^k) + K^T y^k \right) \\ y^{k+1} &= (1-\rho_k) y^k + \rho_k \, \mathbf{prox}_{\sigma h^*}(y^k + \sigma K \tilde{x}^{k+1}) \\ x^{k+1} &= (1-\rho_k) x^k + \rho_k (\tilde{x}^{k+1} - \tau K^T (y^{k+1} - y^k)), \end{aligned} \tag{Algorithm LV}$$

and the other is due to Condat (2013) and Vũ (2013):

$$\begin{aligned} \bar{x}^{k+1} &= x^k - \tau(\nabla f(x^k) + K^T y^k) \\ \tilde{x}^{k+1} &= 2x^{k+1} - \bar{x}^{k+1} \\ x^{k+1} &= (1-\rho_k) x^k + \rho_k \bar{x}^{k+1} \\ y^{k+1} &= (1-\rho_k) y^k + \rho_k \, \mathbf{prox}_{\sigma h^*}(y^k + \sigma K \tilde{x}^{k+1}) \end{aligned} \tag{Algorithm CV}$$

In particular, Algorithm CV is a relaxed version of the PDHG algorithm given in iteration (1.5). Choices of the sequence $\{\rho_k\}$ and the step size parameters (σ, τ) for convergence of these algorithms are discussed in Section 2.2. As can be seen, the proximity operator employed by both algorithms depends only on h^* but not K. Thus they are simple to implement and attractive for parallel and distributed computation as long as either $\mathbf{prox}_{h^*}(\cdot)$ or $\mathbf{prox}_h(\cdot)$ is proximable. Table 2.1 illustrates the proximity operators for popular choices of h. Once the conditions for convergence is understood, the rate of convergence and acceleration of the algorithm are the next interest.

In cases of $g \not\equiv 0$, many variants of Algorithm LV and CV have been studied. Algorithm CV in this case falls into the forward-backward operator splitting

scheme (Bauschke and Combettes, 2011), achieving the usual $O(1/N)$-rate. The primal-dual fixed-point algorithm (PDFP, Chen et al., 2016) subsumes Algorithm LV for this more general case. Other operator splitting approaches for $g \not\equiv 0$ include the Davis-Yin three-operator splitting (Davis and Yin, 2017, for $K = I$), asymmetric forward-backward-adjoint splitting (AFBA, Latafat and Patrinos, 2017) and primal-dual 3-operator splitting (PD3O, Yan, 2018). The latter two include the above forward-backward splitting methods for $g \equiv 0$ as special cases, and allow general K. Acceleration by using variable step sizes and inertia has been studied (Combettes and Vũ, 2014; Lorenz and Pock, 2015; Boţ et al., 2015; Goldstein et al., 2015; Chambolle and Pock, 2016). Despite the reduction of the constant, they all remain in the $O(1/N)$ regime or require strong convexity.

On the other hand, interests in stochastic first-order methods for the primal-dual formulation in general settings appear to be rather recent. When $h \equiv 0$, stochastic versions of the proximal gradient method were considered (Hu et al., 2009; Lin et al., 2014; Nitanda, 2014; Rosasco et al., 2014; Atchadé et al., 2017). For the two-function problem ($K \neq I$ but $g \equiv 0$), mirror-prox algorithms have been considered (Nemirovski et al., 2009; Juditsky et al., 2011; Lan, 2012). Ouyang and Gray (2012) developed a near-optimal algorithm under a strong convexity assumption on f and smoothing of g. Zhong and Kwok (2014) achieved a similar rate to $O\left(\frac{L_f}{N^2} + \frac{L_K}{N} + \frac{\chi}{\sqrt{N}}\right)$ under strong convexity. Without additional assumptions on f or g but assuming $K = I$, Yurtsever et al. (2016) introduced a stochastic variant of the Davis-Yin three-operator splitting. For general K, the stochastic primal-dual algorithm for three-composite convex minimization method (SPDTCM, Zhao and Cevher, 2018) is proposed. This method can be seen as a stochastic version of Chambolle and Pock (2016), and has the rate of $O(L_f/N + L_K/N + \chi/\sqrt{N})$, which is not optimal.

In this regard, the contributions of this chapter, presented in Ko et al. (2019) and Ko and Won (2019) are as follows. First, we connect Algorithms LV and CV from a perspective of monotone operator theory to show that they are essentially the *same* preconditioned forward-backward splitting algorithm (see, e.g., Combettes and Wajs, 2005) sharing a common preconditioner. Second, from this connection a new, broader family of preconditioners that generates an entire *continuum* of forward-backward algorithms is proposed. Third, by a unified analysis, it is shown that this continuum of algorithms enjoys common ergodic and non-ergodic rates of convergence over the entire region of convergence. Prior to the connection the rates of the above two algorithms have been available under much more stringent conditions than those for convergence; we close this gap. Fourth, we proceed further to *accelerate the whole continuum of algorithms to achieve the theoretically optimal rate of convergence, and generalize it further to the case of* $g \not\equiv 0$. Only an optimal acceleration of Algorithm CV has been known (Chen et al., 2014), and acceleration of LV has remained an open problem. Finally, the scalability of the studied algorithms is demonstrated by implementing them on a distributed computing environment in case that data do not fit in the memory of a single device.

The rest of this Chapter is organized as follows. In Section 2.2, the relation between Algorithms LV and CV is shown and they are unified to propose a broader class of algorithms. The rates of convergence of this class of algorithms is also analyzed. In Section 2.3, an accelerated variant of the new class of algorithms achieving the optimal rate is developed. Its stochastic counterpart, also possessing the optimal rate, is discussed in Section 2.4. Section 2.5 demonstrates the convergence behavior and scalability of the new algorithms through their multi-GPU implementations. Discussion and conclusion follow thereafter in Section 2.6. All the proofs of our results can be found in Appendix B.

Notation. That a symmetric matrix M is positive (semi)definite is denoted by $M \succ 0$ ($M \succeq 0$); $L \succ M$ refers to $L - M \succ 0$, etc. For $M \succ 0$, we define its associated inner product and norm by $\langle x, x' \rangle_M = \langle Mx, x' \rangle$ and $\|x\|_M = \sqrt{\langle x, x \rangle_M}$, respectively. For a symmetric matrix M, $\lambda_{\max}(M)$ and $\lambda_{\min}(M)$ respectively denote the maximum and minimum eigenvalues.

2.2 Unification of Algorithms LV and CV ($g \equiv 0$)

In this section, a unified treatment to Algorithms LV and CV from the perspective of monotone operator theory is provided. For a brief summary of monotone operator theory, see Appendix A. To develop relationship between the two algorithms more straightforwardly, we only consider the case $g \equiv 0$ for this section.

2.2.1 Relation between Algorithms LV and CV

It can be shown that both Algorithms LV and CV are instances of preconditioned forward-backward splitting. To be specific, note the first-order optimality condition for (2.1) is given by

$$0 = \nabla f(x^\star) + K^T y^\star, \tag{2.9a}$$

$$y^\star \in \partial h(Kx^\star). \tag{2.9b}$$

where $\partial h(y) = \{w \in \mathbb{R}^l : h(y') \geq h(y) + \langle w, y' - y \rangle,\ \forall y' \in \mathbb{R}^l\}$ is the subdifferential of the convex function h at y, which is a set-valued operator. Since h is closed and proper, condition (2.9b) is equivalent to $Kx^\star \in (\partial h)^{-1}(y^\star) = \partial h^*(y^\star)$ (Bertsekas, 2009), thus (2.9) can be equivalently written as an inclusion problem

$$\begin{bmatrix} 0 \\ 0 \end{bmatrix} \in \begin{bmatrix} \nabla f & K^T \\ -K & \partial h^* \end{bmatrix} \begin{bmatrix} x^\star \\ y^\star \end{bmatrix} =: T(z^\star), \quad z^\star = (x^\star, y^\star). \tag{2.10}$$

Table 2.1: Convex conjugates and proximity operators for selected choices of h. Function δ_S denotes the indicator function for set S so that $\delta_S(u) = 0$ if $u \in S$ and $\delta_S(u) = +\infty$ otherwise; P_S denotes the projection onto set S, which is unique if S is closed and convex; $\sigma_j(M)$ denotes the jth largest singular value of matrix M. All min, max operations are elementwise. In $\ell_{1,q}$-norm, $1/q + 1/s = 1$.

Name	$h(y)$	$h^*(z)$	$\mathbf{prox}_{h^*}(z)$
ℓ_1-norm	$\lambda\|y\|_1$	$\delta_{\mathcal{B}_\infty}(z)$, $\mathcal{B}_\infty = \{z : \|z\|_\infty \leq \lambda\}$	$\min\{\max\{z, -\lambda\}, \lambda\}$
ℓ_2-norm	$\lambda\|y\|_2$	$\delta_{\mathcal{B}_2}(z)$, $\mathcal{B}_2 = \{z : \|z\|_2 \leq \lambda\}$	$P_{\mathcal{B}_2}(z)$
ℓ_∞-norm	$\lambda\|y\|_\infty$	$\delta_{\mathcal{B}_1}(z)$, $\mathcal{B}_1 = \{z : \|z\|_1 \leq \lambda\}$	$P_{\mathcal{B}_1}(z)$
$\ell_{1,q}$-norm	$\sum_{g=1}^{\mathcal{G}} \lambda_g \|y_{[g]}\|_q$	$\delta_{\mathcal{B}_s^1 \times \cdots \times \mathcal{B}_s^{\mathcal{G}}}(z)$, $\mathcal{B}_s^g = \{z : \|z_{[g]}\|_s \leq \lambda_g\}$	$\left(P_{\mathcal{B}_s^1}(z_{[1]}), \ldots, P_{\mathcal{B}_s^{\mathcal{G}}}(z_{[\mathcal{G}]})\right)$
nuclear norm	$\lambda \sum_{i=1}^{\text{rank}(Y)} \sigma_i(Y)$	$\delta_{\mathcal{B}_*}(Z)$, $\mathcal{B}_* = \{Z : \|Z\|_2 \leq \lambda\}$	$U \min\{\Sigma, \lambda I\} V^T$, $Z = U\Sigma V^T$
hinge loss	$\sum_{i=1}^{l} \max\{1 - y_i, 0\}$	$\sum_{i=1}^{l} \left(z_i - \delta_{[0,1]}(-z_i)\right)$	$\min\{z + 1, \max\{z, 1\}\}$

The set-valued operator T is split into $T = F + G$, where

$$F = \begin{bmatrix} 0 & K^T \\ -K & \partial h^* \end{bmatrix} \quad \text{and} \quad G = \begin{bmatrix} \nabla f & 0 \\ 0 & 0 \end{bmatrix}. \tag{2.11}$$

The operator F is maximally monotone and G is $1/L_f$-cocoercive (Bauschke and Combettes, 2011). A preconditioned forward-backward splitting for solving (2.10) is

$$\begin{aligned} \tilde{z}^k &= (I + M^{-1}F)^{-1}(I - M^{-1}G)(z^k) \\ z^{k+1} &= (1 - \rho_k)z^k + \rho_k \tilde{z}^k, \end{aligned} \tag{2.12}$$

for $z^k = (x^k, y^k)$, $\tilde{z}^k = (\tilde{x}^k, \tilde{y}^k)$, and $M \succ 0$. If the modulus of cocoercivity of $M^{-1}G$ is denoted by γ (cocoercivity of G is preserved; see Davis, 2015), then (2.12) converges if $\gamma > 1/2$ for a sequence $\{\rho_k\} \subset [0, \delta]$ such that $\sum_{k=0}^{\infty} \rho_k(\delta - \rho_k) = \infty$ with $\delta = 2 - 1/(2\gamma)$. Note $\rho_k \equiv 1$ is allowed, which yields a simple iteration $z^{k+1} = (I + M^{-1}F)^{-1}(I - M^{-1}G)z^k$. The inverse operator $(I + M^{-1}F)^{-1}$ is single-valued due to maximal monotonicity of $M^{-1}F$ (Bauschke and Combettes, 2011, Theorems 25.8 and 24.5). (For instance, $(I + \partial\phi)^{-1}(z) = \arg\min_{z' \in \mathbb{R}^n} \phi(z') + \frac{1}{2}\|z' - z\|_2^2 = \mathbf{prox}_\phi(z)$.) In particular, the preconditioners for Algorithms LV and CV are respectively given by Combettes et al. (2014); Condat (2013); Vũ (2013):

$$M = M_{\mathsf{LV}} := \begin{bmatrix} \frac{1}{\tau} I & \\ & \frac{1}{\sigma} I - \tau K K^T \end{bmatrix} \quad \text{and} \quad M = M_{\mathsf{CV}} := \begin{bmatrix} \frac{1}{\tau} I & -K^T \\ -K & \frac{1}{\sigma} I \end{bmatrix}.$$

Now we are ready to see that Algorithms LV and CV are essentially the same algorithm. The "LDL" decomposition of M_{CV} reveals that

$$M_{\mathsf{CV}} = \begin{bmatrix} I & \\ -\tau K & I \end{bmatrix} \begin{bmatrix} \frac{1}{\tau} I & \\ & \frac{1}{\sigma} I - \tau K K^T \end{bmatrix} \begin{bmatrix} I & -\tau K^T \\ & I \end{bmatrix} = L M_{\mathsf{LV}} L^T. \tag{2.13}$$

It is clear that both M_{LV} and M_{CV} are positive definite if and only if $1/(\tau\sigma) > \|K\|_2^2$. Also it is easy to see that Algorithm CV, i.e., (2.12) with $M = M_{\mathsf{CV}}$, is equivalent to

$$L^T z^{k+1} = (1-\rho_k) L^T z^k + \rho_k (I + M_{\mathsf{LV}}^{-1}\tilde{F})^{-1}(I - M_{\mathsf{LV}}^{-1}\tilde{G})(L^T z^k), \qquad (2.14)$$

where $\tilde{F} = L^{-1}FL^{-T}$ and $\tilde{G} = L^{-1}GL^{-T}$. Letting $w = L^T z$, *we see that Algorithm CV is in fact Algorithm LV applied to the linearly transformed variable* w *by splitting the similarly transformed operator* $L^{-1}TL^{-T}$ *into* $\tilde{F}$ *and* $\tilde{G}$. The cocoercivity constant of $M_{\mathsf{LV}}^{-1}\tilde{G}$ is found by the following proposition.

Proposition 1. *$M_{\mathsf{LV}}^{-1}\tilde{G}$ is $(1/\tau - \sigma\|K\|_2^2)/L_f$-cocoercive with respect to $\|\cdot\|_{M_{\mathsf{LV}}}$.*

Thus from the discussion below (2.12) we have $\gamma = (1/\tau - \sigma\|K\|_2^2)/L_f$ and $\delta = 2 - \frac{L_f}{2} \cdot \frac{1}{1/\tau - \sigma\|K\|_2^2}$. Then Algorithm CV converges if

$$\frac{1}{\tau} > \frac{L_f}{2} \quad \text{and} \quad \left(\frac{1}{\tau} - \frac{L_f}{2}\right)\frac{1}{\sigma} > \|K\|_2^2 \qquad (2.15)$$

With respect to the untransformed sequence $\{z^k\}$, observe that $M_{\mathsf{CV}}^{-1}G$ is also $(1/\tau - \sigma\|K\|_2^2)/L_f$-cocoercive (with respect to $\|\cdot\|_{M_{\mathsf{CV}}}$). In light of (2.14), it is natural to measure convergence using the metric $\|L^T \cdot\|_{M_{\mathsf{LV}}}$, and this metric coincides with $\|\cdot\|_{M_{\mathsf{CV}}}$. On the other hand, it is easy to see $M_{\mathsf{LV}}^{-1}G$ is $1/(\tau L_f)$-cocoercive with respect to $\|\cdot\|_{M_{\mathsf{LV}}}$, hence Algorithm LV has $\gamma = 1/(\tau L_f)$ and $\delta = 2 - \tau L_f/2$. It converges if

$$1/\tau > L_f/2 \quad \text{and} \quad 1/(\tau\sigma) > \|K\|_2^2. \qquad (2.16)$$

Both (2.15) and (2.16) recover the known convergence regions in the literature (Condat, 2013; Chen et al., 2013).

2.2.2 Unified algorithm class

The relation between the two algorithms suggests a more general family of preconditioners, namely

$$M = \tilde{L} M_{\mathsf{LV}} \tilde{L}^T = \begin{bmatrix} \frac{1}{\tau} I & C^T \\ C & \frac{1}{\sigma} I + \tau (C C^T - K K^T) \end{bmatrix}, \tag{2.17}$$

where $\tilde{L}$ replaces $(2,1)$ block of L in (2.13) by τC. In particular, if $CK^T = KC^T$, then (2.12) yields the following iteration (for simplicity we set $\rho_k \equiv 1$):

$$\begin{aligned} y^{k+1} &= \mathbf{prox}_{\sigma h^*}(\sigma K x^k + \sigma\tau (C - K) \nabla f(x^k) + (I + \sigma\tau K (C - K)^T) y^k) \\ x^{k+1} &= x^k - \tau (\nabla f(x^k) - C^T y^k + (C + K)^T y^{k+1}). \end{aligned} \tag{2.18}$$

Condition $CK^T = KC^T$ is satisfied if and only if $C = US\Sigma^{-1}V^T + N\bar{V}^T$, where U, V, and Σ are from the reduced singular value decomposition of $K = U\Sigma V^T$ so that Σ is an $r \times r$ positive diagonal matrix where $r = \operatorname{rank}(K)$; $\bar{V}$ is such that $\tilde{V} = [V, \bar{V}]$ is orthogonal; S is symmetric, and N is arbitrary. A simple choice is $S = \kappa\Sigma^2$ for some $\kappa \in \mathbb{R}$ and $N = 0$, yielding $C = \kappa K$. Choosing $\kappa = 0$ and -1 respectively recovers Algorithms LV and CV; for $\kappa = 1$, we have

$$\begin{aligned} y^{k+1} &= \mathbf{prox}_{\sigma h^*}(\sigma K x^k + y^k) \\ x^{k+1} &= x^k - \tau \nabla f(x^k) - \tau K^T (2y^{k+1} - y^k), \end{aligned}$$

which is the dual version of Algorithm CV (Condat, 2013, Algorithm 3.2). Another choice is to set $S = \pm\Sigma^2$ and N so that NN^T is diagonal. In this case $CC^T - KK^T$ reduces to a diagonal matrix, $C = [\bar{K}, N]\tilde{V}$ where $\bar{K}$ is the first r columns of $K\tilde{V}$. If the eigenspace of $K^T K$ is well-known and multiplication with $\bar{V}$ can be computed fast, e.g., the discrete cosine transform matrix for the

fused lasso on a regular grid (Lee et al., 2017), this choice can be useful.

2.2.3 Convergence analysis

Region of convergence

A condition for (2.12) with general M to converge is

$$M \succ \begin{bmatrix} \frac{L_f}{2} I & \\ & 0 \end{bmatrix}, \tag{2.19}$$

which follows from Theorem 2 and Proposition 3 later in this section. Thus with M in (2.17) the following region of convergence is obtained.

Proposition 2. *Algorithm* (2.18) *converges for* (σ, τ) *such that*

$$\frac{1}{\tau} > \frac{L_f}{2} \quad \textit{and} \quad \left(\frac{1}{\tau} - \frac{L_f}{2}\right)\left(\frac{1}{\sigma} - \tau\|K\|_2^2\right) > \frac{\tau L_f}{2}\|C\|_2^2. \tag{2.20}$$

Note that (2.20) reduces to (2.16) for Algorithm LV and to (2.15) for CV. In general for $C = \kappa K$, $\kappa \in [-1, 1]$, the region of convergence shrinks gradually from $|\kappa| = 0$ (LV) to 1 (CV); see Figure 2.1. This extends the observation made in Section 2.2.1 regarding convergence conditions (2.16) and (2.15) to a continuum of algorithms between LV and CV.

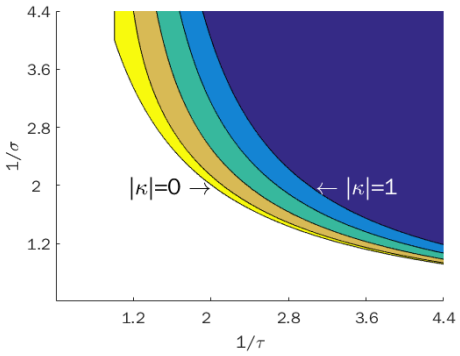


Figure 2.1: Region of convergence in $(1/\sigma, 1/\tau)$. Boundaries correspond to $|\kappa|$ = 0, 0.25, 0.5, 0.75, 1.

Remark 1. *Condat (2013) also considers the case $g \not\equiv 0$. In this case, the second term of the first line of Algorithm CV is replaced by* $\mathbf{prox}_{\tau g}(x^k - \tau(\nabla f(x^k) + K^T y^k))$. *This algorithm is still a preconditioned forward-backward splitting one with preconditioner M_{CV}, where the zero in the (1,1) block of operator F is replaced by ∂g, and converges under* (2.15). *For this extended F,* (2.18) *generates a feasible algorithm only when $C = \pm K$.*

Rates of convergence

We now analyze the rates of convergence of the preconditioned forward-backward splitting algorithm (2.12) for the preconditioner matrices M of (2.17). A pre-duality gap function $\mathcal{G}(\tilde{z}, z) := \mathcal{L}(\tilde{x}, y) - \mathcal{L}(x, \tilde{y})$, where $z = (x, y)$ and $\tilde{z} = (\tilde{x}, \tilde{y})$, is used to measure the convergence of the objective value, because the duality gap $\mathcal{G}^\star(\tilde{z}) := \sup_{z \in Z} \mathcal{G}(\tilde{z}, z)$, $Z \subset \mathbb{R}^p \times \mathbb{R}^l$, guarantees that the pair $\tilde{z} = (\tilde{x}, \tilde{y})$ is a primal-dual solution to (2.2) if $\mathcal{G}^\star(\tilde{z}) \leq 0$. The rate of convergence of a gap function is typically analyzed in terms of an averaged solution sequence $\bar{z}^N = \sum_{k=0}^N \alpha_k z^k / \sum_{k=0}^N \alpha_k$ for some positive sequence $\{\alpha_k\}$, yielding an *ergodic rate*. Ergodic rates are widely studied in the literature (Loris and Verhoeven, 2011; Chen et al., 2013; Boţ and Csetnek, 2015; Chambolle and Pock, 2011, 2016), partly due to ease of analysis. Sometimes the unaveraged (last) solution sequence $\{z_k\}$ or $\{\tilde{z}_k\}$ is preferred as it tends to preserve the desired structural properties better than the ergodic counterpart. Analysis based on the unaveraged sequence yields the *non-ergodic rate* (Davis, 2015).

First we establish an $O(1/N)$ ergodic convergence rate of the pre-duality gap evaluated for an average of the first N terms of the sequence $\{(\tilde{x}^k, \tilde{y}^k)\}$:

Theorem 1. *In iteration* (2.12), *let μ be a constant such that $\|(x, 0)\|^2_{M^{-1}} \leq (1/\mu)\|x\|_2^2$, for all $x \in \mathbb{R}^p$. Let $\alpha = (2\mu)/(4\mu - L_f)$ and denote $z^k = (x^k, y^k)$, $\tilde{z}^k = (\tilde{x}^k, \tilde{y}^k)$. Define $\bar{z}^N = (\bar{x}^N, \bar{y}^N)$ with $\bar{x}^N = \sum_{k=0}^N \rho_k \tilde{x}^k / \sum_{k=0}^N \rho_k$ and $\bar{y}^N = \sum_{k=0}^N \rho_k \tilde{y}^k / \sum_{k=0}^N \rho_k$. Also let $\bar{\rho} = \sup_{k \geq 0} \rho_k$. If $\mu > L_f/2$ and $\{\rho_k\}$ is chosen so that $0 < \rho_k < 1/\alpha$ for all k, then the following holds for all*

$z = (x, y) \in \mathbb{R}^p \times \mathbb{R}^l$:

$$\mathcal{G}(\bar{z}^N, z) \leq \frac{1}{2\sum_{k=0}^N \rho_k} \left(\|z^0 - z\|_M^2 + \frac{\alpha L_f}{(1-\alpha\bar{\rho})\lambda_{\min}(M)} \|z^0 - z^\star\|_M^2 \right),$$

where $z^\star = (x^\star, y^\star)$ *is a solution to* (2.2).

The key observation in proving Theorem 1 is the following lemma, also used in the proof of Theorem 2.

Lemma 1. *For* $\rho \in (0, 2)$, *consider a relation* $z^+ = (I + M^{-1}F)^{-1}(I - M^{-1}G)z^-$, $z_\rho = (1-\rho)z^- + \rho z^+$. *Write* $z_\rho = (x_\rho, y_\rho)$, $z^+ = (x^+, y^+)$, $z^- = (x^-, y^-)$, *all in* $\mathbb{R}^p \times \mathbb{R}^l$. *Then,*

$$\begin{aligned} 2\rho\, \mathcal{G}(z^+, z) \leq \|z^- - z\|_M^2 - \|z_\rho - z\|_M^2 \\ + (1 - 2/\rho)\|z^- - z_\rho\|_M^2 + (L_f/\rho)\|x^- - x_\rho\|_2^2, \quad \forall z = (x, y). \end{aligned}$$

Now let $\mathcal{F}(x) = f(x) + h(Kx)$ be the primal objective function and $\mathcal{F}^\star$ be the primal optimal value. For an important class of penalty functions h including those for the generalized and group lasso, the following rate for primal suboptimality holds.

Corollary 1. *Assume the conditions for Theorem 1. If* $\mathbf{dom}(h) = \mathbb{R}^l$, *i.e., h does not take the value* $+\infty$, *then there exists a constant* C_1 *independent of N such that for all N,*

$$0 \leq \mathcal{F}(\bar{x}^N) - \mathcal{F}^\star \leq C_1/(\textstyle\sum_{k=0}^N \rho_k).$$

Thus if $\{\rho_k\}$ is chosen so that $\inf_{k\geq 0} \rho_k > 0$, we obtain $O(1/N)$ convergence of the primal suboptimality.

The following theorem establishes the non-ergodic counterpart of Theorem 1.

Theorem 2. *For some* $\nu > L_f/2$ *and* $\epsilon > 0$, *suppose M in iteration* (2.12) *satisfies*

$$M \succeq \begin{bmatrix} \nu I & \\ & \epsilon I \end{bmatrix} \tag{2.21}$$

Let $\alpha = 2\nu/(4\nu - L_f)$ and write $z^k = (x^k, y^k)$, $\tilde{z}^k = (\tilde{x}^k, \tilde{y}^k)$. If $\{\rho_k\}$ is chosen so that $0 < \rho_k < 1/\alpha$ for all k and $\tau = \inf_{k\geq 0} \rho_k(1-\alpha\rho_k) > 0$, then the following holds:

$$\mathcal{G}(\tilde{z}^k, z) \leq \ \|z^0 - z^\star\|_M(\|z^0 - z^\star\|_M + \|z^\star - z\|_M)/(\sqrt{\tau(k+1)}), \qquad \forall z = (x, y) \in \mathbb{R}^p \times \mathbb{R}^l,$$

and additionally, $\mathcal{G}(\tilde{z}^k, z) = o(1/\sqrt{k+1})$. Furthermore, if $\mathbf{dom}(h) = \mathbb{R}^l$, then there exists a constant C_2 independent of k such that $0 \leq \mathcal{F}(\tilde{x}^k) - \mathcal{F}^\star \leq C_2/\sqrt{k+1}$ for all k and $\mathcal{F}(\tilde{x}^k) - \mathcal{F}^\star = o(1/\sqrt{k+1})$.

Remark 2. *The little-o result suggests that the non-asymptotic upper bound of the gap function may be conservative and the gap may diminish faster than the $1/\sqrt{k+1}$ rate. The outcomes of the numerical experiments in Section 2.5 also suggest that the bound is not tight.*

Closing the gap

Here, how the results close the gap in the literature between the conditions for convergence and those for the rate is described. The following fact helps understanding the conditions for Theorems 1 and 2:

Proposition 3. *For $M \succ 0$ and a given $L_f > 0$, the following are equivalent.*

1. *For all $x \in \mathbb{R}^p$, there exists $\mu > L_f/2$ such that $\|(x, 0)\|^2_{M^{-1}} \leq (1/\mu)\|x\|_2^2$.*
2. *The condition* (2.19) *holds.*
3. *There exist $\nu > L_f/2$ and $\epsilon > 0$ such that $M \succeq \begin{bmatrix} \nu I & \\ & \epsilon I \end{bmatrix}$.*

That is, the conditions for Theorems 1 and 2 are both equivalent to (2.19). This implies that the rates of convergence results in this section hold for M in (2.17) satisfying (2.20). *Thus, for the entire range of (σ, τ) for which* (2.18) *converges, an $O(1/N)$ ergodic and an $o(1/\sqrt{k+1})$ non-ergodic convergence rates for the objective values are established.*

For Algorithm LV ($M = M_{\mathsf{LV}}$), Loris and Verhoeven (2011) obtain an $O(1/N)$ ergodic convergence rate for $f(x) = \frac{1}{2}\|\mathsf{A}x - b\|_2^2$. For general f, Chen

et al. (2013) show that Algorithm LV converges under (2.16), but the rate is given only for strongly convex f and full row rank K. This special case is not very interesting in statistical learning applications in which f is almost always not strongly convex. To the best of my knowledge, the result for the rates of convergence for Algorithm LV and its variants (including the optimal accelerated one in the next subsection) without this impractical assumption is novel. For Algorithm CV ($M = M_{\mathsf{CV}}$), the result extends the region of parameters for which ergodic converge rate is known from $(1/\tau - \sigma\|K\|^2)/L_f \geq 1$ (Chambolle and Pock, 2016, Theorems 1 and 2) to the full range $(1/\tau - \sigma\|K\|^2)/L_f \geq 1/2$ of (2.15). Therefore the gap between the conditions for convergence and those for the rate is closed.

Remark 3. *An inspection of the proof of Lemma 1 asserts that the results of this section also holds for the extended F (see Remark 1). Thus the gap for the three-function extension of Algorithm CV is closed as well.*

Remark 4. *Davis (2015, Proposition 5.3) analyzes both ergodic and non-ergodic rates for general F and G, under the condition $M \succeq \underline{\lambda} I$ for some $\underline{\lambda} > 0$. When applied to* (2.12)*, this analysis results in a convergence region smaller than that is allowed by* (2.19)*. Here the special structure of G in* (2.11) *is exploited.*

2.3 Optimal acceleration

It is well known that first-order methods can be accelerated by introducing some "inertia" (Nesterov, 2004; Beck and Teboulle, 2009; Chen et al., 2012). For the saddle-point problem of the form (2.2), i.e., $g \equiv 0$, the optimal rate of convergence is known to be $O(L_f/N^2 + L_K/N)$ in terms of the duality gap $\mathcal{G}^\star$, where N is the total number of iterations (Nesterov, 2005; Chen et al., 2014).

A natural question arises regarding whether the same optimal rate can be attained for the entire continuum (2.18) of algorithms, for the case of $g \not\equiv 0$. It turns out that this rate is also optimal for this more general case, in the

following sense.

1. The optimal rate of solving $\min_{x\in\mathcal{X}}(f(x)+g(x))$ by using any first-order method is $O(L_f/N^2)$ (Nesterov, 2004), e.g., by using FISTA (Beck and Teboulle, 2009).

2. For sufficiently large p, there exist $b \in \mathcal{Y} \subset \mathbb{R}^l$ and $K \in \mathbb{R}^{l\times p}$ such that $h^*(y) = \langle b, y\rangle$ and the rate of convergence for solving

$$\min_{x\in\mathcal{X}} \max_{y\in\mathcal{Y}}(\langle Kx, y\rangle - h^*(y)) = \min_{x\in\mathcal{X}} \max_{y\in\mathcal{Y}}\langle Kx - b, y\rangle$$

is $\Omega(L_K/N)$ (Nemirovsky, 1992; Nemirovski, 2004).

2.3.1 Algorithms

Chen et al. (2014) devise an accelerated variant of Algorithm CV (for $g \equiv 0$) that achieves the theoretically optimal rate of convergence $O(L_f/N^2 + L_K/N)$, where N is the total number of iterations:

$$\bar{x}^k = \tilde{x}^k + \theta_k(\tilde{x}^k - \tilde{x}^{k-1}) \tag{2.22a}$$

$$x^k_{md} = (1-\rho_k)x^k + \rho_k\tilde{x}^k \tag{2.22b}$$

$$\tilde{y}^{k+1} = \mathbf{prox}_{\sigma_k h^*}(\tilde{y} + \sigma_k K\bar{x}^k) \tag{2.22c}$$

$$\tilde{x}^{k+1} = \tilde{x}^k - \tau_k(\nabla f(x^k_{md}) + K^T\tilde{y}^{k+1}) \tag{2.22d}$$

$$x^{k+1} = (1-\rho_k)x^k + \rho_k\tilde{x}^{k+1} \tag{2.22e}$$

$$y^{k+1} = (1-\rho_k)y^k + \rho_k\tilde{y}^{k+1}. \tag{2.22f}$$

Note an extrapolation step (2.22a) with a parameter θ_k, and a "middle" relaxation step (2.22b) are introduced. For (2.18), i.e., $g \not\equiv 0$, we consider the following generalization:

$$\bar{u}^k = K\tilde{x}^k - \theta_k A(\tilde{x}^k - \tilde{x}^{k-1}) \tag{2.23a}$$

$$\bar{v}^k = K^T\tilde{y}^k + \theta_k\left(\tau_k^{-1}\tau_{k-1}(K+B)^T - B^T\right)(\tilde{y}^k - \tilde{y}^{k-1}) \tag{2.23b}$$

$$x_{md}^k = (1-\rho_k)x^k + \rho_k\tilde{x}^k \tag{2.23c}$$

$$\tilde{u}^{k+1} = \bar{u}^k - \tau_k(K+A)(\nabla f(x_{md}^k) + \bar{v}^k) \tag{2.23d}$$

$$\tilde{y}^{k+1} = \mathbf{prox}_{\sigma_k h^*}(\tilde{y}^k + \sigma_k\tilde{u}^{k+1}) \tag{2.23e}$$

$$\tilde{v}^{k+1} = K^T\tilde{y}^{k+1} + B^T(\tilde{y}^{k+1} - \tilde{y}^k) - \theta_k B^T(\tilde{y}^k - \tilde{y}^{k-1}) \tag{2.23f}$$

$$\tilde{x}^{k+1} = \mathbf{prox}_{\tau_k g}\left(\tilde{x}^k - \tau_k(\nabla f(x_{md}^k) + \tilde{v}^{k+1})\right) \tag{2.23g}$$

$$x^{k+1} = (1-\rho_k)x^k + \rho_k\tilde{x}^{k+1} \tag{2.23h}$$

$$y^{k+1} = (1-\rho_k)y^k + \rho_k\tilde{y}^{k+1}. \tag{2.23i}$$

Step sizes (σ_k, τ_k) are allowed to depend on the iteration count k. This algorithm reduces to (2.18) (hence to Algorithms LV, CV, and in between) if $g \equiv 0$, $A = -C$, $B = C$, $\rho_k \equiv 1$, $\theta_k \equiv 0$, $\sigma_k \equiv \sigma$, and $\tau_k \equiv \tau$, and to Chen et al. (2014) for $g \equiv 0$, $A = -K$, and $B = 0$. The optimal rate of convergence of (2.23) is established in Section 2.3.2. In particular, the optimal acceleration of Algorithm LV and cases $g \not\equiv 0$ is novel.

2.3.2 Convergence analysis

We first consider the case in which the bounds for $\{x^k\}$, $\{y^k\}$ is known *a priori*. In this case we can assume that the search space is $Z = X \times Y$, where $X \subset \mathbb{R}^p$, $Y \subset \mathbb{R}^l$ are both closed and bounded. Under this assumption, we have the following bound for the duality gap:

Theorem 3. *Let $\{z^k\} = \{(x^k, y^k)\}$ be the sequence generated by (2.23). Fix $A = -K$ if $g \not\equiv 0$. Assume for some Ω_X, $\Omega_Y > 0$,*

$$\sup_{x,x'\in X}\|x - x'\|_2^2 \le 2\Omega_X^2, \;\; \sup_{y,y'\in Y}\|y - y'\|_2^2 \le 2\Omega_Y^2, \tag{2.24}$$

and the parameter sequences $\{\rho_k\}$, $\{\theta_k\}$, $\{\tau_k\}$, and $\{\sigma_k\}$ satisfy $\rho_1 = 1$ and

$$\rho_{k+1}^{-1} - 1 = \rho_k^{-1}\theta_{k+1}, \tag{2.25a}$$

$$\frac{1-q}{\tau_k} - L_f\rho_k - \frac{1}{r}\|A\|_2^2\sigma_k \geq 0, \tag{2.25b}$$

$$\frac{1-r}{\sigma_k} - \tau_k\left(2\|K+A\|_2\|K+B\|_2 + \frac{1}{q}\|B\|_2^2\right) \geq 0 \tag{2.25c}$$

for some $q \in (0,1)$, $r \in (0,1)$. *Further suppose that*

$$0 < \theta_k \leq \min(\tau_{k-1}/\tau_k, \sigma_{k-1}/\sigma_k), \max(\tau_{k-1}/\tau_k, \sigma_{k-1}/\sigma_k) \leq 1. \tag{2.26}$$

Then for all $k \geq 1$,

$$\mathcal{G}^\star(z^{k+1}) \leq \tfrac{\rho_k}{\tau_k}\Omega_X^2 + \tfrac{\rho_k}{\sigma_k}\Omega_Y^2. \tag{2.27}$$

The duality gap $\mathcal{G}^\star(z)$ is defined in Section 2.2.3. For the following choice of the algorithm parameters, we obtain the claimed optimal convergence rate.

Corollary 2. *Assume* $\|B\|_2 \leq bL_K$ *and* $\|K+B\|_2 \leq dL_K$ *for some positive* b *and* d. *If* $g \equiv 0$, *further assume* $\|A\|_2 \leq aL_K$ *and* $\|K+A\|_2 \leq cL_K$ *for some positive* a *and* c. *Otherwise, put* $A = -K$, $a = 1$, *and* $c = 0$. *When the parameters are set to*

$$\rho_k = \tfrac{2}{k+1}, \quad \theta_k = \tfrac{k-1}{k}, \quad \tau_k = \tfrac{k}{2P_1L_f + kQL_K\Omega_Y/\Omega_X}, \quad \sigma_k = \tfrac{\Omega_Y}{L_K\Omega_X}, \text{ where} \tag{2.28}$$

$$P_1 = \tfrac{1}{1-q} \quad \text{and} \quad P_2 = \max\left\{\tfrac{1}{(1-q)r}a^2, \tfrac{1}{1-r}(2cd + b^2/q)\right\}, \tag{2.29}$$

then

$$\mathcal{G}^\star(z^k) \leq \tfrac{4P_1\Omega_X^2}{k(k-1)}L_f + \tfrac{2\Omega_X\Omega_Y(P_2+1)}{k}L_K, \quad \forall k \geq 2. \tag{2.30}$$

Remark 5. *For* $g \equiv 0$, $A = -K$, $B = 0$, (2.25) *recovers the condition for Chen et al. (2014, Theorem 2.1) by putting* $r \to 1$ *and* $q \to 0$. *For* $A = -\kappa K = -B$, *we obtain* $(1 - |\kappa|q)/\tau_k \geq L_f\rho_k + |\kappa|L_K^2\sigma_k/r$ *and* $(1-|\kappa|r)/\sigma_k \geq L_K^2\tau_k\left(2(1-\kappa^2) + |\kappa|/q\right)$. *In particular for Algorithm LV* ($\kappa = 0$), *we have* $1/\tau_k \geq L_f\rho_k$ *and* $1/(\tau_k\sigma_k) \geq 2L_K^2$ *regardless of* q *and* r; *this condition resembles* (2.16).

Now suppose the bounds for $\{x^k\}$, $\{y^k\}$ are unavailable. In this case the duality gap $\sup_{z\in Z}\mathcal{G}(\tilde{z}, z)$, $Z = \mathbb{R}^p \times \mathbb{R}^l$, may be unbounded above. Instead, we

define a perturbed gap function:

$$\tilde{\mathcal{G}}(\tilde{z}, v) := \sup_{z \in Z} \mathcal{G}(\tilde{z}, z) - \langle v, \tilde{z} - z \rangle. \tag{2.31}$$

There always exists a perturbation vector v such that (2.31) is finite (Monteiro and Svaiter, 2011). Thus we want to find a sequence of perturbation vectors $\{v^k\}$ that make $\tilde{\mathcal{G}}(\tilde{z}^k, v^k)$ small.

Theorem 4. *Suppose that $\{z^k\} = \{(x^k, y^k)\}$ are generated by Algorithm (2.23). Fix $A = -K$ if $g \not\equiv 0$. If the parameter sequences $\{\rho_k\}$, $\{\theta_k\}$, $\{\tau_k\}$, and $\{\sigma_k\}$ satisfy (2.25) and*

$$\theta_k = \tau_{k-1}/\tau_k = \sigma_{k-1}/\sigma_k \leq 1 \tag{2.32}$$

for some $0 < q < 1$, $0 < r < 1/2$. Then there exists a vector v^{k+1} such that for any $k \geq 1$,

$$\tilde{\mathcal{G}}(z^{k+1}, v^{k+1}) \leq \tfrac{\rho_k}{\tau_k} \left(2 + \tfrac{q}{1-q} + \tfrac{2r+1}{1-2r}\right) R^2 =: \epsilon_{k+1}, \text{ and} \tag{2.33}$$

$$\begin{aligned} \|v^{k+1}\|_2 \leq \Big(&\tfrac{\rho_k}{\tau_k}\|\hat{x} - \tilde{x}^1\|_2 + \tfrac{\rho_k}{\sigma_k}\|\hat{y} - \tilde{y}^1\|_2\Big) \\ &+ \big(\tfrac{\rho_k}{\tau_k}(\mu + \tfrac{\tau_1}{\sigma_1}\nu) + 2\rho_k(\mu\|A\|_2 + \nu\|B\|_2) \\ &\qquad + 2\tau_k\rho_k\nu\|K + A\|_2\|K + B\|_2\big)R, \end{aligned} \tag{2.34}$$

where $(\hat{x}, \hat{y})$ is a pair of solutions to problem (2.2), and

$$R = \sqrt{\|\hat{x} - \tilde{x}^1\|_2^2 + \tfrac{\tau_1}{\sigma_1}\|\hat{y} - \tilde{y}^1\|_2^2}, \quad \mu = \sqrt{\tfrac{1}{1-q}}, \quad \nu = \sqrt{\tfrac{2\sigma_1}{\tau_1(1-2r)}}. \tag{2.35}$$

For the following choice of the algorithm parameters, we obtain the claimed optimal convergence rate.

Corollary 3. *Assume $\|B\|_2 \leq bL_K$ and $\|K + B\|_2 \leq dL_K$ for some positive b and d. If $g \equiv 0$, further assume $\|A\|_2 \leq aL_K$ and $\|K + A\|_2 \leq cL_K$ for some positive a and c. Otherwise, put $A = -K$, $a = 1$, and $c = 0$. N is fixed, and the parameters are set to*

$$\rho_k = \tfrac{2}{k+1}, \quad \theta_k = \tfrac{k-1}{k}, \quad \tau_k = \tfrac{k}{2P_1L_f + P_2NL_K}, \quad \sigma_k = \tfrac{k}{NL_K}, \text{ where} \tag{2.36}$$

$$P_1 = \tfrac{1}{1-q}, \quad P_2 = \max\left\{\tfrac{a^2}{(1-q)r}, \tfrac{2cd+b^2/q}{1-r}, 1\right\}, \tag{2.37}$$

then

$$\epsilon_{N+1} \leq \left(\tfrac{4P_1L_f}{N^2} + \tfrac{2P_2L_K}{N}\right)\left[2 + \tfrac{q}{1-q} + \tfrac{r+1/2}{1/2-r}\right]R^2, \text{ and} \tag{2.38}$$

$$\|v^{N+1}\|_2 \leq \tfrac{4P_1L_f}{N^2}\left[\left(\|\hat{x}-\tilde{x}^1\|_2 + \|\hat{y}-\tilde{y}^1\|_2\right) + R\left(\mu + \tfrac{\tau_1}{\sigma_1}\nu\right)\right] \tag{2.39}$$

$$+ \tfrac{L_K}{N}\left[2P_2\left(\left(\|\hat{x}-\tilde{x}^1\|_2 + \|\hat{y}-\tilde{y}^1\|_2\right) + R\left(\mu + \tfrac{\tau_1}{\sigma_1}\nu\right)\right) + 4R(a\mu + b\nu) + \tfrac{4Rcd\nu}{P_2}\right].$$

This result can be interpreted as follows. Theorem 4 and Corollary 3 state that for every pair of positive scalars (ρ, ε), Algorithm (2.23) generates (v^N, ϵ_N) such that $\|v^N\| \leq \rho$ and $\epsilon_N \leq \varepsilon$ (see (2.33), (2.34), (2.38), and (2.39)) for a sufficiently large N. The associated pair (x^N, y^N) is called a (ρ, ε)-saddle point of the unperturbed saddle point problem (2.2) (Monteiro and Svaiter, 2011, Definition 3.10). With this notion, the following proposition can be stated.

Proposition 4. *Under the assumptions of Theorem 4 and Corollary 3, there exists a vector* $w^N = (w_x^N, w_y^N)$ *such that* $w^N \in T_{\epsilon^N}(x^N, y^N)$ *and* $\|w^N\| \leq \rho + \sqrt{4L\varepsilon}$ *for some constant* $L > 0$*, where*

$$T_\varepsilon = \begin{bmatrix} \nabla f & K^T \\ -K & \partial_\varepsilon h^* \end{bmatrix}.$$

Here, $\partial_\varepsilon h^*$ *is the* ε*-subgradient of* h^* *defined as* $\partial_\varepsilon h^*(y) = \{g : h^*(y') \geq h^*(y) + \langle y' - y, g\rangle - \varepsilon, \forall y' \in \mathbb{R}^l\}, \quad \forall y \in \mathbb{R}^l$.

Proof. The result follows directly from Proposition 3.13, Definition 3.4, Proposition 3.5, and Proposition 3.6 of Monteiro and Svaiter (2011). □

The condition $w^N \in T_{\epsilon_N}(x^N, y^N)$ in Proposition 4 can be written as the following two inequalities

$$0 \geq -\langle \nabla f(x^N) + K^T y^N, x - x^N\rangle + \langle w_x^N, x - x^N\rangle - \epsilon_N, \quad \forall x, \tag{2.40a}$$

$$h^*(y) \geq h^*(y^N) + \langle Kx^N, y - y^N\rangle + \langle w_y^N, y - y^N\rangle - \epsilon_N, \quad \forall y. \tag{2.40b}$$

Comparing with the optimality conditions (2.9) for the unperturbed saddle

point problem (2.2):

$$0 \geq -\langle \nabla f(x^\star) + K^T y^\star, x - x^\star \rangle, \quad \forall x,$$
$$h^*(y) \geq h^*(y^\star) + \langle K x^\star, y - y^\star \rangle, \quad \forall y,$$

we see that the sum of the last two terms in each right-hand side of (2.40a) and (2.40b) is the error of the approximate solution (x^N, y^N). Indeed, in the unit ball centered at (x^N, y^N), each error is bounded by $\rho + \sqrt{4L\varepsilon} + \varepsilon$, which can be made arbitrarily small since the choice of (ρ, ε) is free. In this sense, for large N, (x^N, y^N) is a "nearly optimal" primal-dual solution.

2.4 Stochastic optimal acceleration

2.4.1 Algorithm

In large-scale ("big data") applications, it is often the case that even the first-order information on the objective of (2.1) or (2.2) cannot be obtained exactly. Such settings can be modeled by a stochastic oracle, which provides unbiased estimators of the first-order information. To be precise, at the k-th iteration suppose the oracle returns the stochastic gradient $(\hat{\mathcal{F}}(\tilde{x}^k), \hat{\mathcal{K}}_x(\tilde{x}^k), \hat{\mathcal{K}}_y(\tilde{y}^k))$ independently from the previous iteration, such that

$$\mathbb{E}[\hat{\mathcal{F}}(\tilde{x}^k)] = \nabla f(\tilde{x}^k), \;\; \mathbb{E}\left[\begin{pmatrix} -\hat{\mathcal{K}}_x(\tilde{x}^k) \\ \hat{\mathcal{K}}_y(\tilde{y}^k) \end{pmatrix}\right] = \begin{pmatrix} -K\tilde{x}^k \\ K^T\tilde{y}^k \end{pmatrix}, \tag{2.41}$$
$$\mathbb{E}[\hat{\mathcal{A}}(\tilde{x}^k)] = A\tilde{x}^k, \;\; \text{and} \;\; \mathbb{E}[\hat{\mathcal{B}}(\tilde{y}^k)] = B^T\tilde{y}^k.$$

We further assume that the variance of these estimators is uniformly bounded, i.e.,

$$\begin{aligned}
\mathbb{E}[\|\hat{\mathcal{F}}(\tilde{x}^k) - \nabla f(\tilde{x}^k)\|^2] &\leq \chi_{x,f}^2, \\
\mathbb{E}[\|\hat{\mathcal{K}}_x(\tilde{x}^k) - K\tilde{x}^k\|^2] &\leq \chi_y^2, \\
\mathbb{E}[\|\hat{\mathcal{K}}_y(\tilde{y}^k) - K^T\tilde{y}^k\|^2] &\leq \chi_{x,K}^2, \\
\mathbb{E}[\|\hat{\mathcal{A}}(\tilde{x}^k) - A\tilde{x}^k\|^2] &\leq \chi_A^2, \\
\mathbb{E}[\|\hat{\mathcal{B}}(\tilde{y}^k) - B^T\tilde{y}^k\|^2] &\leq \chi_B^2.
\end{aligned} \tag{2.42}$$

For notational convenience, we define $\chi_x := \sqrt{\chi_{x,f}^2 + \chi_{x,K}^2}$.

We consider the following stochastic variant of (2.23):

$$\begin{aligned}
\overline{u}_k &= \hat{\mathcal{K}}_x(\tilde{x}^k) - \theta_k\hat{\mathcal{A}}(\tilde{x}^k - \tilde{x}^{k-1}) \\
\overline{v}_k &= \hat{\mathcal{K}}_y(\tilde{y}^k + \frac{\theta_k\tau_{k-1}}{\tau_k}) + \hat{\mathcal{B}}\left(\left(\frac{\tau_{k-1}}{\tau_k} - 1\right)(\tilde{y}^k - \tilde{y}^{k-1})\right) \\
\tilde{x}_{md}^k &= (1-\rho_k)x^k + \rho_k\tilde{x}^k \\
\tilde{u}^{k+1} &= \overline{u}_k - \tau_k(\hat{\mathcal{K}_x} + \hat{\mathcal{A}})(\hat{\mathcal{F}}(\tilde{x}_{md}^k) + \overline{v}_k) \\
\tilde{y}^{k+1} &= \mathbf{prox}_{\sigma_k h^*}(\tilde{y}^k + \sigma_k\tilde{u}^{k+1}) \\
\tilde{v}^{k+1} &= \hat{\mathcal{K}}_y(\tilde{y}^{k+1}) + \hat{\mathcal{B}}(\tilde{y}^{k+1} - \tilde{y}^k - \theta_k(\tilde{y}^k - \tilde{y}^{k-1})) \\
\tilde{x}^{k+1} &= \mathbf{prox}_{\tau_k g}\,\tilde{x}^k - \tau_k(\hat{\mathcal{F}}(\tilde{x}_{md}^k) + \tilde{v}^{k+1}) \\
x^{k+1} &= (1-\rho_k)x^k + \rho_k\tilde{x}^{k+1} \\
y^{k+1} &= (1-\rho_k)y^k + \rho_k\tilde{y}^{k+1},
\end{aligned} \tag{2.43}$$

which can be considered a generalization of the stochastic variant of (2.22) by Chen et al. (2014). The optimal rate of convergence of solving (2.2) stochastically is known to be $O\left(\frac{L_f}{N^2} + \frac{L_K}{N} + \frac{\chi_x+\chi_y}{\sqrt{N}}\right)$ in terms of the expected duality gap $\mathbb{E}[\mathcal{G}^\star(\cdot)]$ (Chen et al., 2014). In the sequel, we show that Algorithm (2.43) achieves this rate.

2.4.2 Convergence analysis

We obtain the following results for Algorithm (2.43) when Z is bounded. Note part 2.47 of Theorem 5 is strengthened under the tail assumption

$$\begin{aligned}\mathbb{E}\left[\exp(\|\nabla f(x)-\hat{\mathcal{F}}(x)\|_2^2/\chi_{x,f}^2)\right] &\le \exp(1)\\ \mathbb{E}\left[\exp(\|Kx-\hat{\mathcal{K}}_x(x)\|_2^2/\chi_y^2)\right] &\le \exp(1)\\ \mathbb{E}\left[\exp(\|K^Ty-\hat{\mathcal{K}}_y(y)\|_2^2/\chi_{x,K}^2)\right] &\le \exp(1).\end{aligned} \tag{2.44}$$

Observe that (2.44) implies (2.42) by Jensen's inequality.

Theorem 5. *Fix $A=-K$ if $g\not\equiv 0$. Assume that (2.24) holds, for some Ω_X, $\Omega_Y>0$. Also suppose that for all $k\ge 1$, the parameters ρ_k, θ_k, τ_k, and σ_k in (2.43) satisfy (2.25a), (2.26),*

$$\frac{s-q}{\tau_k}-L_f\rho_k-\frac{\|A\|_2^2\sigma_k}{r}\ge 0, \tag{2.45a}$$

$$\frac{t-r}{\sigma_k}-\tau_k\left(2\|K+A\|_2\|K+B\|_2+\frac{\|B\|_2^2}{q}\right)\ge 0 \tag{2.45b}$$

for some q, r, s, $t\in(0,1)$. Then the following holds.

(i) Under (2.42), we have $\mathbb{E}[\mathcal{G}^\star(z^{k+1})]\le\mathcal{Q}_0(k)$ for all $k\ge 1$, where

$$\begin{aligned}\mathcal{Q}_0(k):=\tfrac{\rho_k}{\gamma_k}\left(\tfrac{2\gamma_k}{\tau_k}\Omega_X^2+\tfrac{2\gamma_k}{\sigma_k}\Omega_Y^2\right)+\tfrac{\rho_k}{2\gamma_k}\textstyle\sum_{i=1}^k\Big(\tfrac{(2-s)\tau_i\gamma_i}{1-s}(\chi_x^2+\chi_B^2)\\ +\frac{(2-t)\sigma_i\gamma_i}{1-t}(\chi_y^2+\chi_A^2+\tau_k^2\|K+A\|_2^2(\chi_x^2+\chi_B^2))\Big)\end{aligned} \tag{2.46}$$

(ii) Suppose $A=-K$ and $B=bK$, then under the assumption (2.44), we have

$$\mathbf{Pr}(\mathcal{G}^\star(z^{k+1})>\mathcal{Q}_0'(k)+\lambda\mathcal{Q}_1(k))\le 3\exp(-\lambda^2/3)+3\exp(-\lambda), \tag{2.47}$$

for all $\lambda>0$ and $t\ge 1$, where

$$\mathcal{Q}_0'(k):=\tfrac{\rho_k}{\gamma_k}\left(\tfrac{2\gamma_k}{\tau_k}\Omega_X^2+\tfrac{2\gamma_k}{\sigma_k}\Omega_Y^2\right)+\tfrac{\rho_k}{2\gamma_k}\textstyle\sum_{i=1}^k\left(\tfrac{(2-s)\tau_i\gamma_i}{1-s}\chi_x^2+\tfrac{(2-t)\sigma_i\gamma_i}{1-t}\chi_y^2\right), \tag{2.48}$$

$$\mathcal{Q}_1(k):=\tfrac{\rho_k}{\gamma_k}\left(\sqrt{2}\chi_x\Omega_X+\chi_y\Omega_Y\right)\sqrt{2\textstyle\sum_{i=1}^k\gamma_i^2} \tag{2.49}$$

$$+ \frac{\rho_k}{2\gamma_k} \sum_{i=1}^{k} \left(\frac{(2-s)\tau_i\gamma_i}{1-s}\chi_x^2 + \frac{(2-t)\sigma_i\gamma_i}{1-t}\chi_y^2 \right).$$

Corollary 4. *Assume condition* (2.24) *holds. In Algorithm* (2.43)*, if* $N \geq 1$ *is given,* $A = -K$*,* $\|B\|_2 \leq b\|K\|_2$*, and the parameters are set to*

$$\rho_k = \frac{2}{k+1}, \quad \theta_k = \frac{k-1}{k}, \quad \tau_k = \frac{\Omega_X k}{2P_1 L_f \Omega_X + P_2 L_k \Omega_Y k + P_3 \chi_x k^{3/2}}, \tag{2.50}$$

$$\sigma_k = \frac{\Omega_Y}{L_K \Omega_X + P_3 \chi_y \sqrt{k}} \tag{2.51}$$

where P_1 *and* P_2 *satisfies*

$$P_1 = \frac{1}{s-q}, \quad P_2 \geq \max\left\{ \frac{1}{r(s-q)}, \frac{b^2/q}{t-r} \right\}, \quad P_3 > 0 \tag{2.52}$$

the following holds.

(i) Under assumption (2.42)*, we have* $\mathbb{E}[\mathcal{G}^\star(z^N)] \leq \mathcal{C}_0(N)$*, where*

$$\begin{aligned} \mathcal{C}_0(k) = {} & \frac{8P_1 L_f \Omega_X^2}{k(k+1)} + \frac{4L_K \Omega_X \Omega_Y (P_2+1)}{k} + \frac{4P_3(\chi_x \Omega_X + 4\chi_y \Omega_Y)}{\sqrt{k}} \\ & + \frac{\sqrt{2}(2-s)\Omega_X \chi_x}{3(1-s)\sqrt{k}} + \frac{\sqrt{2}(2-t)\Omega_Y \chi_y}{3(1-t)\sqrt{k}}. \end{aligned} \tag{2.53}$$

(ii) Under assumption (2.44)*, then we have*

$$P(\mathcal{G}^\star(z^N) > \mathcal{C}_0(N) + \lambda \mathcal{C}_1(N)) \leq 3\exp(-\lambda^2/3) + 3\exp(-\lambda), \tag{2.54}$$

for all $\lambda > 0$*, where*

$$\mathcal{C}_1(k) = \left(4 + \frac{\sqrt{2}(2-s)}{3(1-s)}\right) \frac{\Omega_X \chi_x}{\sqrt{k}} + \left(2\sqrt{2} + \frac{\sqrt{2}(2-s)}{3(1-s)}\right) \frac{\Omega_Y \chi_y}{\sqrt{k}}. \tag{2.55}$$

Remark 6. *Zhao and Cevher (2018, Remark 3), who achieve the rate* $O(L_f/N + L_K/N + \chi/\sqrt{N})$*, suggest that the rate for the smooth part* f *may be improved to* $O(L_f/N^2)$*. We have shown that this is indeed possible and the resulting rate is optimal.*

When Z is unbounded, we have the following theorem.

Theorem 6. *Assume that* $\{z^k\} = \{(x^k, y^k)\}$ *is the sequence generated by* (2.43)*. Further assume that the parameters* ρ_k*,* θ_k*,* τ_k*, and* σ_k *in* (2.43) *satisfy* (2.25a)*,* (2.32)*, and* (2.45)*. for all* $k \geq 1$ *and some* q*,* s*,* $t \in (0,1)$*,* $r \in (0, 1/2)$*.*

If $A = -K$ or $g \equiv 0$, then there is a perturbation vector v^{k+1} satisfying

$$\mathbb{E}[\tilde{\mathcal{G}}(z^{k+1}, v^{k+1})] \leq \frac{\rho_k}{\tau_k}\left[\left(6 + \frac{4q}{1-q} + \frac{4(r+1/2)}{1/2-r}\right)R^2 + \left(\frac{5}{2} + \frac{2q}{1-q} + \frac{2(r+1/2)}{1/2-r}\right)S^2\right] \tag{2.56}$$

for all $k \geq 1$. Furthermore,

$$\mathbb{E}[\|v^{k+1}\|_2] \leq \frac{2\rho_k\|\hat{x}-x^1\|_2}{\tau_k} + \frac{2\rho_k\|\hat{y}-y^1\|_2}{\sigma_k} + \sqrt{2R^2+S^2}\left[\frac{\rho_k(1+\mu)}{\tau_k} + \left(\nu + \sqrt{\frac{\sigma_1}{\tau_1}}\right)\frac{\rho_k}{\sigma_k}\right. \tag{2.57}$$

$$\left. + 2\rho_k(\|A\|_2\mu + \|B\|_2\nu) + 2\tau_k\rho_k\|K+A\|_2\|K+B\|_2\nu\right] =: \epsilon_{k+1} \tag{2.58}$$

where $(\hat{x}, \hat{y})$ is a pair of solutions for (2.2), R, μ, and ν are as defined in (2.35), and

$$S := \sqrt{\sum_{i=1}^k \frac{(2-s)\tau_i^2(\chi_x^2+\chi_B^2)}{1-s} + \sum_{i=1}^k \frac{(2-t)\tau_i\sigma_i(\chi_y^2+\chi_A^2+\tau_k^2\|K+A\|_2^2(\chi_x^2+\chi_B^2))}{1-t}}. \tag{2.59}$$

Corollary 5. *In Algorithm (2.43), if N is given, $A = -K$, $B = bK$, and the parameters are set to*

$$\rho_k = \frac{2}{k+1}, \quad \theta_k = \frac{k-1}{k}, \quad \tau_k = \sigma_k = \frac{k}{2P_1L_f + P_2L_K(N-1) + P_3N\sqrt{N-1}\chi'}, \tag{2.60}$$

for some $\tilde{R} > 0$, where χ' is defined by $\chi' = \sqrt{\frac{2-s}{1-s}\chi_x^2 + \frac{2-t}{1-t}\chi_y^2}$. Then for P_1, P_2, and P_3 satisfying

$$\begin{aligned} &P_1 = \frac{1}{s-q}, \quad P_2 \geq \max\left\{\frac{1}{r(s-q)}, \frac{b^2}{q(t-r)}, 1\right\} \\ &P_3 = 1/\tilde{R}, \end{aligned} \tag{2.61}$$

for some $\tilde{R} > 0$, q, s, and $t \in (0,1)$, $r \in (0, 1/2)$, $q < s$, and $r < t$. Then we have

$$\begin{aligned} \epsilon_N \leq &\left(\frac{4P_1L_f}{N(N-1)} + \frac{2P_2L_K}{N} + \frac{2\chi'/\tilde{R}}{\sqrt{N-1}}\right) \\ &\times \left(\left(6 + \frac{4q}{1-q} + \frac{4(r+1/2)}{1/2-r}\right)R^2 + \left(\frac{5}{2} + \frac{2q}{1-q} + \frac{2(r+1/2)}{1/2-r}\right)\tilde{R}^2/3\right), \end{aligned}$$

and

$$\mathbb{E}[\|v^N\|_2] \leq \left(\frac{4P_1L_f}{N(N-1)} + \frac{2P_2L_K}{N} + \frac{2\chi'/\tilde{R}}{\sqrt{N-1}}\right)$$

$$\times \left(2R\left(1+\sqrt{\tfrac{\sigma_1}{\tau_1}}\right)+\left(\sqrt{2}R+\tfrac{\tilde{R}}{\sqrt{3}}\right)\left(1+\mu+\left(\sqrt{\tfrac{\sigma_1}{\tau_1}}+\nu\right)\right)\right)$$
$$+\tfrac{4L_K}{N}(\sqrt{2}R+\tilde{R}/\sqrt{3})(\mu+b\nu).$$

Therefore we obtain the desired order for both ϵ_N and $\mathbb{E}[\|v_N\|_2]$.

2.5 Numerical experiments

In this section, the actual convergence behavior of the algorithms generated by (2.18) and their accelerated variant (2.23) is illustrated. In addition, the scalability of these algorithms by implementing a distributed version of (2.18) is demonstrated. The experiment was conducted on a system with two Intel Xeon CPUs (E5-2680 v2 @2.80GHz) with eight Nvidia GTX 1080 GPUs with 8 GB of RAM each.

2.5.1 Model problems

Overlapping group elastic net. We consider an overlapping group elastic net problem with a quadratic loss

$$\min_x \frac{1}{2}\|b-Ax\|_2^2+\frac{\lambda_1}{2}\|x\|_2^2+\lambda_2\sum_{j=1}^{R}\sqrt{|G_j|}\|x_{G_j}\|_2.$$

where $\mathsf{A}=[a_1,\cdots,a_n]^T$ is the data matrix, and $b=(b_1,\cdots,b_n)$ is the response vector. A test dataset was generated based on the methods in Chen et al. (2012). For the group designation, R groups of S adjacent variables were defined, with 10 overlaps of adjacent groups. i.e., $g_j=\{90(j-1)+1,\ldots,90j+10\}$, thus $p=R(S-10)+10$. The true value of x_j was set by $x_j=(-1)^j\exp(-(j-1)/100)$ for $j=1,\ldots,p$. Each element of A was sampled from the standard normal distribution, and added Gaussian noise $\epsilon\sim\mathcal{N}(0,1)$ to $\mathsf{A}x$ to generate $b=\mathsf{A}x+\epsilon$. For the convergence experiments, $R=100$ and $S=100$ were chosen, so that the dimension is given by $p=9010$. For the scalability experiment,

$S = 130$ and $R = 1000, 5000, 8000, 10000$ were selected so that the dimensions are $p = 120010, 600010, 960010, 1200010$. For all experiments, the number of data points was chosen as $n = 5000$.

Graph-guided sparse fused lasso. The graph-guided fused lasso problem we consider is given by

$$\min_x \frac{1}{2}\|Ax - b\|_2^2 + \lambda_1\|x\|_1 + \lambda_2\|Dx\|_1,$$

where D is the difference matrix imposed by the network structure. The dataset for the graph-guided fused lasso experiments was generated following the transcription factor (TF) model of Zhu (2017). This is a simple gene network model with J fully connected subnetworks of size T, where each subgroup has one TF with $T-1$ regulatory target genes. Variables corresponding to TFs are sampled independently from $\mathcal{N}(0, 1)$. Variables for target genes are sampled so that each target gene and the corresponding TF has a bivariate normal distribution with correlation 0.7, and these variables are conditionally independent given the TF. For j-th subnetwork, the true values of x_i were chosen by

$$x_i = \begin{cases} (-1)^{j+1}\left\lfloor \frac{j+1}{2} \right\rfloor & \text{if } j = 1, \ldots, J_a \\ 0 & \text{otherwise} \end{cases}, \quad i = (j-1)r + 1, \ldots, jr,$$

where J_a is the number of active groups. Response b_i is sampled so that $b_i = \mathsf{A}x + \epsilon_i$, with $\epsilon_i \overset{\text{i.i.d.}}{\sim} \mathcal{N}(0, 100^2)$. In addition to the edges comprised of fully-connected subnetworks, random edges were added between the active variables and the inactive variables. For each active variable, edges connecting this variable and $J - 1$ distinct inactive variables were added. For the convergence experiments, $T = 10$, $J_a = 20$, $J = 1000$ were used so that the dimension p is 10000. For the scalability experiment, $T = 12$, and $J_a = 20$. We se-

lected $J = 10000, 50000, 80000, 100000$ were selected to generate the dataset with $p = 120000, 600000, 960000, 1200000$, respectively. For all experiments, the number of samples was chosen by $n = 5000$.

2.5.2 Convergence behavior

Two-function case ($g \equiv 0$)

First, the algorithms are applied to the two function cases ($g \equiv 0$) without acceleration: overlapping group lasso (group elastic net with $\lambda_1 = 0$, $\lambda_2 = R/100$), graph-guided fused lasso (graph-guided sparse fused lasso with $\lambda_1 = 0$, $\lambda_2 = 1$), and latent group lasso (as discussed in Section 2.1 with a quadratic loss, $\lambda_1 = 0$, $\lambda_g = R\sqrt{|G_j|}/100$). For the forward-backward (FB) splitting (2.18), $C = \kappa K$, and $|\kappa| \leq 1$ were used, with $\rho_k = 0.9\left(2 - \frac{\tau L_f}{2}\frac{1-(1-\kappa^2)\tau\sigma\|K\|_2^2}{1-\tau\sigma\|K\|_2^2}\right)$. Step sizes were chosen as $\tau = 0.9\frac{2}{L_f}$ and $\sigma = 0.9\frac{1}{\tau}\frac{1-\tau L_f/2}{1-(1-\kappa^2)\tau L_f/2}$, so that (2.20) is satisfied. For the acceleration (2.23), four cases were tested: Algorithm LV ($A = B = 0$), CV ($A = -K$, $B = K$), their "midpoint" ($A = -0.5K$, $B = 0.5K$), and Chen et al. (2014) ($A = -K$, $B = 0$). The number of iterations N is set to 10000. For bounded (Corollary 2) and unbounded (Corollary 3) cases, (q, r) are chosen so that they minimize $\frac{4P_1\Omega_X^2}{k(k-1)}L_f + \frac{2\Omega_X\Omega_Y(P_2+1)}{N}\|K\|_2$ in (2.30) and $\left(\frac{4P_1L_f}{N^2} + \frac{2P_2L_K}{N}\right)\left(2 + \frac{q}{1-q} + \frac{r+1/2}{1/2-r}\right)$ in (2.38), respectively. Those minimizers were found using sequential least squares programming. As a benchmark, an inertial version of the forward-backward-forward (FBF) algorithm (Combettes and Pesquet, 2012) was applied, as described in Boţ and Csetnek (2016):

$$\begin{aligned}
\tilde{x}^{k+1} &= x^k - \tau\left(\nabla f(x^k) + K^T y^k\right) + \alpha_1(x^k - x^{k-1}) \\
\tilde{y}^{k+1} &= \mathbf{prox}_{\tau h^*}(y^k + \tau K x^k + \alpha_1(y^k - y^{k-1}) \\
y^{k+1} &= \tilde{y}^{k+1} + \tau K(\tilde{x}^{k+1} - x^k) + \alpha_2(y^k - y^{k-1}) \\
x^{k+1} &= \tilde{x}^{k+1} - \tau K^T(\tilde{y}^{k+1} - y^k) + \alpha_2(x^k - x^{k-1}).
\end{aligned} \tag{2.62}$$

With $\alpha_1 = \alpha_2 = 0$, (2.62) resembles Algorithm LV, but requires one more step per iteration; its convergence rate has not been established.

Figures 2.2(a-b), 2.3(a-b), and 2.4(a-b) show the convergence of the FB (2.18) with respect to the averaged sequence $\{(\bar{x}^N, \bar{y}^N)\}$, and the convergence of the accelerated FB algorithms (2.23) with respect to $\{(x^N, y^N)\}$. The gap between the primal objective value at x^k and the "optimal" objective value versus iteration count k is plotted. Following Loris and Verhoeven (2011), the reference "optimal" value was computed by running the accelerated LV algorithm with bounded parameters for 100000 iterations; this obtained the minimal value up to the point that the machine precision allows. Figures 2.2(a), 2.3(a), and 2.4(a) used parameters given by (2.28), which assumes x^k and y^k are bounded. This is true as long as $\|x^k\|_2 < \Omega_X/\sqrt{2}$ and $\|y^k\|_2 < \Omega_Y/\sqrt{2}$; $\Omega_X = 12$ and $\Omega_Y = 15$ were used for group lasso problems, and $\Omega_X = 141.4$ and $\Omega_Y = 305.9$ were used for graph-guided fused lasso. The resulting iterates respected these bounds. Figures 2.2(b), 2.3(b), and 2.4(b) used parameters given by (2.36), which does not require Ω_X and Ω_Y. Since the reference optimal value was an order of 10^4, the values in the oscillating region correspond to the 7th or 8th significant decimal digit of the objective value.

We observe that Theorems 1 and 4 faithfully describe the convergence behavior. The convergence rates of the accelerated ones were close to $O(1/N^2)$, because in this experiment $L_f \gg \|K\|_2$. On the other hand, the base FB algorithms appear very close to the $O(1/N)$ line. All of the optimal acceleration settings exhibit very similar convergence behaviors, which suggest that we have a good degree of freedom in choosing an optimal primal-dual algorithm.

Figures 2.2(c), 2.3(c), 2.4(c) compare the non-ergodic convergence with respect to $\{(\tilde{x}^k, \tilde{y}^k)\}$ of the FB and FBF. The FB algorithms behave like $O(1/k)$ initially, and then converge faster than $O(1/k^2)$. This behavior is much faster

than what is predicted by Theorem 2. On the contrary, the FBF algorithm stalls after a few hundred iterations.

Now actual convergence behaviors of the optimal stochastic algorithm (2.43) for the group lasso and graph-guided fused lasso model problems are illustrated. The estimate $\hat{\mathcal{F}}(x^k)$ is computed by $\nabla f(\mathcal{M}x^k)$, where $\mathcal{M}$ is a diagonal matrix where each diagonal entry is independently chosen as $1/p$ with probability π, and 0 with probability $1-\pi$. This strategy meets the assumption (2.41).

The convergence behavior of the stochastic algorithm is illustrated in Figure 2.5. Figures 2.5a and 2.5c show the result of (2.43) with parameters (2.50) for the group lasso and graph-guided fused lasso problems, respectively. Figures 2.5b and 2.5d show those with parameters given by (2.60). Note that for the assumption (2.42) to hold, both cases need estimates of Ω_X and Ω_Y. The experiment was conducted using $\pi = 0.2$. For the simplicity of illustration, $\chi = 3 \times 10^5$ was used for the overlapping group lasso, and $\chi = 10^7$ was used for the graph-guided fused lasso. In (2.60), $\tilde{R}$ was set to 10 for overlapping group lasso and 100 for graph-guided fused lasso. The horizon N was set to 10000 for all cases. In (2.50) and (2.60), q, r, s, and t were chosen to minimize the error bounds $\mathcal{C}_0(N)$ in Corollary 4 and $\frac{4P_1L_f}{N(N-1)} + \frac{2P_2\|K\|_2}{N} + \frac{2\chi/\tilde{R}}{\sqrt{N-1}}$ in Corollary 5, respectively, in a similar fashion to the deterministic counterparts. For a comparison, cases with parameters chosen for the deterministic setting (2.28) and (2.36) but with stochastic estimation of gradients were included. In Figure 2.5, the convergence of the stochastic algorithms is slow initially because the step sizes τ_k and σ_k are very small for small k due to the presence of an $N^{3/2}$ term in their denominators, but they eventually converge faster than the $O(1/k)$ rate for both bounded and unbounded parameter selections. (Also note the log-log scale of the plots.) While Corollaries 4 and 5 guarantee the optimal rate for $A = -K$ (corresponding to CV if $B = K$ and Chen et al. (2014) if $B = 0$),

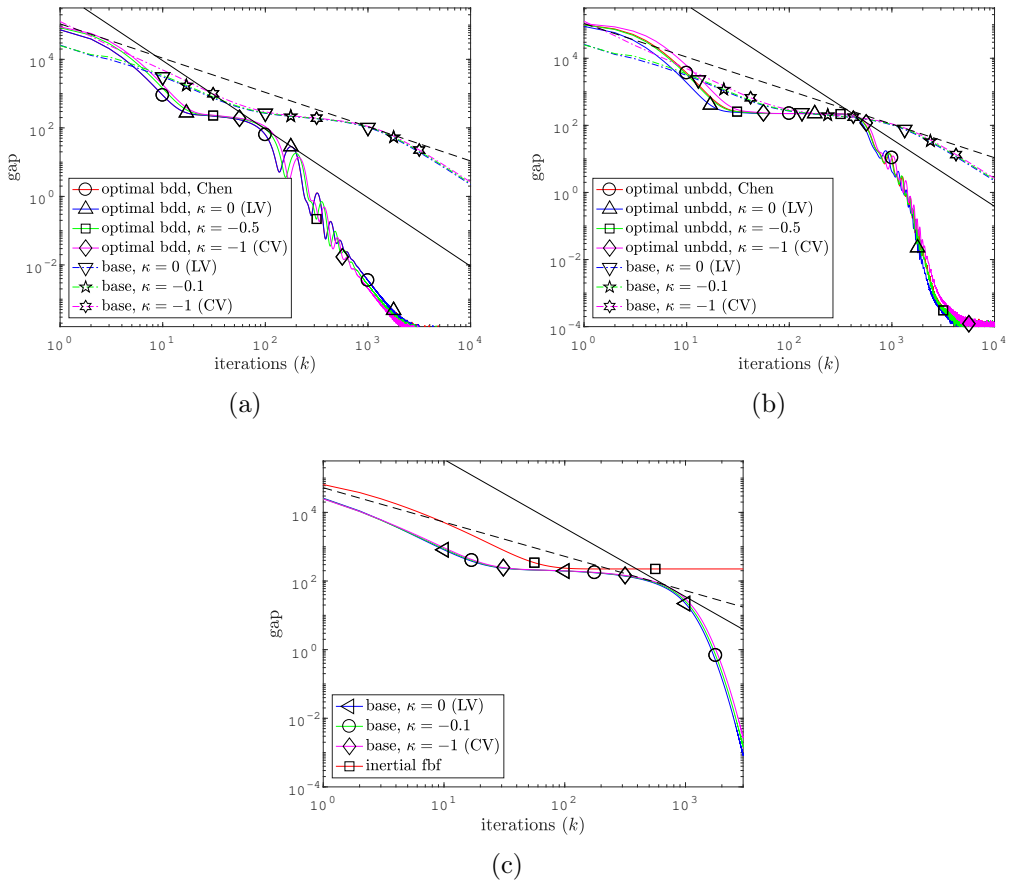


Figure 2.2: Convergence of the forward-backward (FB) algorithms generated by (2.18) and their accelerated variants (2.23) for a overlapping group lasso model. (a) optimal acceleration with bounded parameter setting ("optimal") with ergodic convergence of the FB algorithm ("base"). (b) optimal acceleration with unbounded parameter setting ("optimal") with ergodic convergence of the FB algorithm ("base"). (c) non-ergodic convergence of the FB ("base") and inertial FBF ("inertial fbf") algorithms. Solid black lines represent $O(1/k^2)$ convergence, and dashed black lines represent $O(1/k)$ convergence.

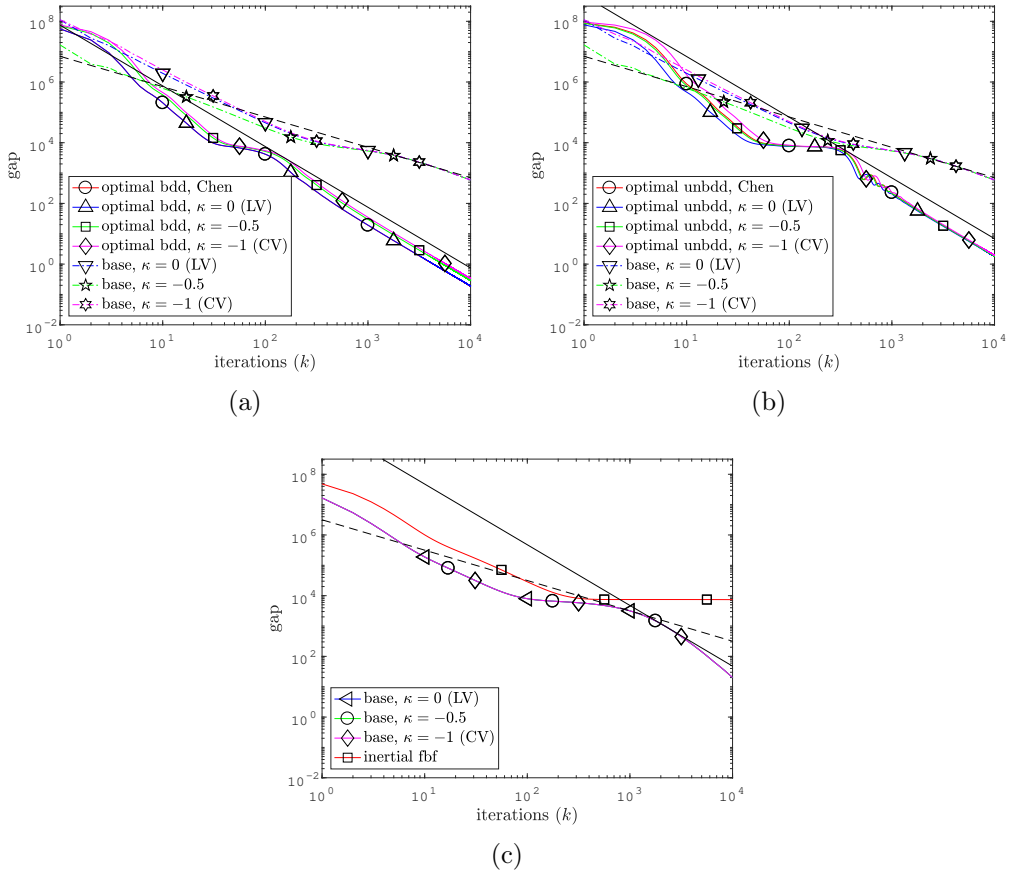


Figure 2.3: Convergence of the forward-backward (FB) algorithms generated by (2.18) and their accelerated variants (2.23) for a graph-guided fused lasso model. (a) optimal acceleration with bounded parameter setting ("optimal") with ergodic convergence of the FB algorithm ("base"). (b) optimal acceleration with unbounded parameter setting ("optimal") with ergodic convergence of the FB algorithm ("base"). (c) non-ergodic convergence of the FB ("base") and inertial FBF ("inertial fbf") algorithms. Solid black lines represent $O(1/k^2)$ convergence, and dashed black lines represent $O(1/k)$ convergence.

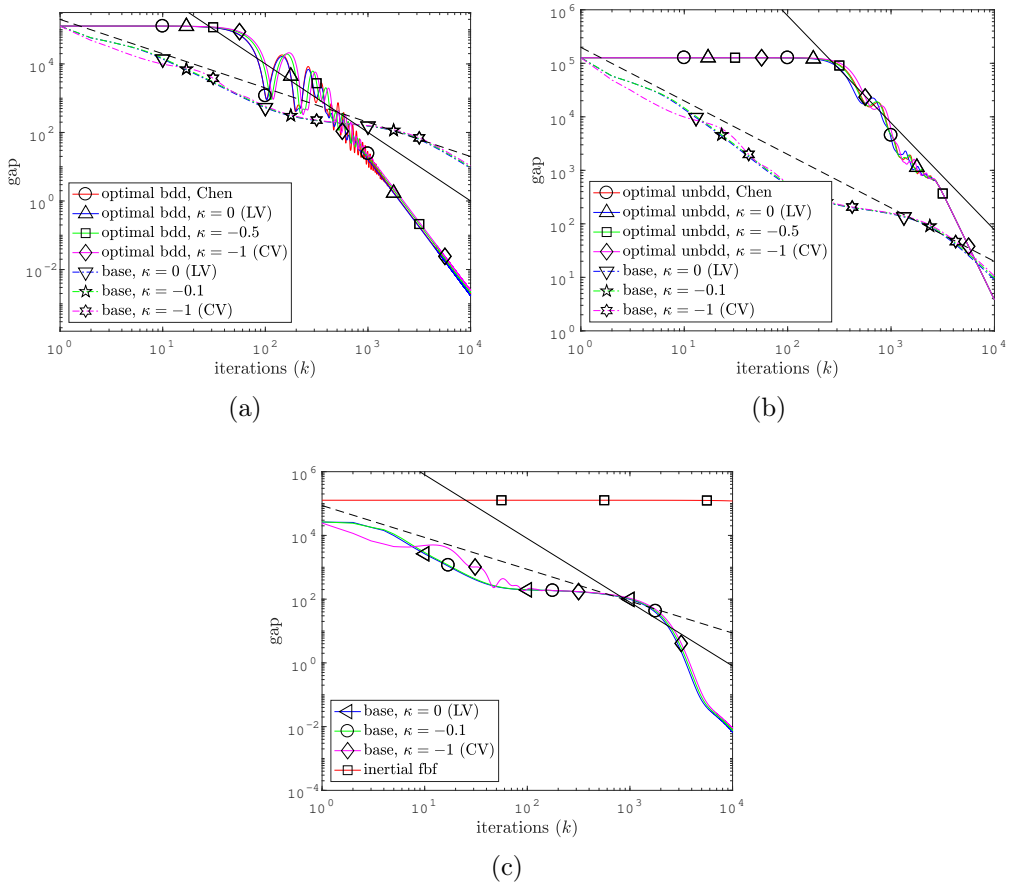


Figure 2.4: Convergence of the forward-backward (FB) algorithms generated by (2.18) and their accelerated variants (2.23) for a latent group lasso model. (a) optimal acceleration with bounded parameter setting ("optimal") with ergodic convergence of the FB algorithm ("base"). (b) optimal acceleration with unbounded parameter setting ("optimal") with ergodic convergence of the FB algorithm ("base"). (c) non-ergodic convergence of the FB ("base") and inertial FBF ("inertial fbf") algorithms. Solid black lines represent $O(1/k^2)$ convergence, and dashed black lines represent $O(1/k)$ convergence.

the choice $A = -\kappa K$, $B = \kappa K$ with $0 \leq \kappa < 1$ (corresponding to LV and "in-between") also exhibited a similar convergence behavior. On the contrary, for the "deterministic" choice of the parameters the algorithm diverged.

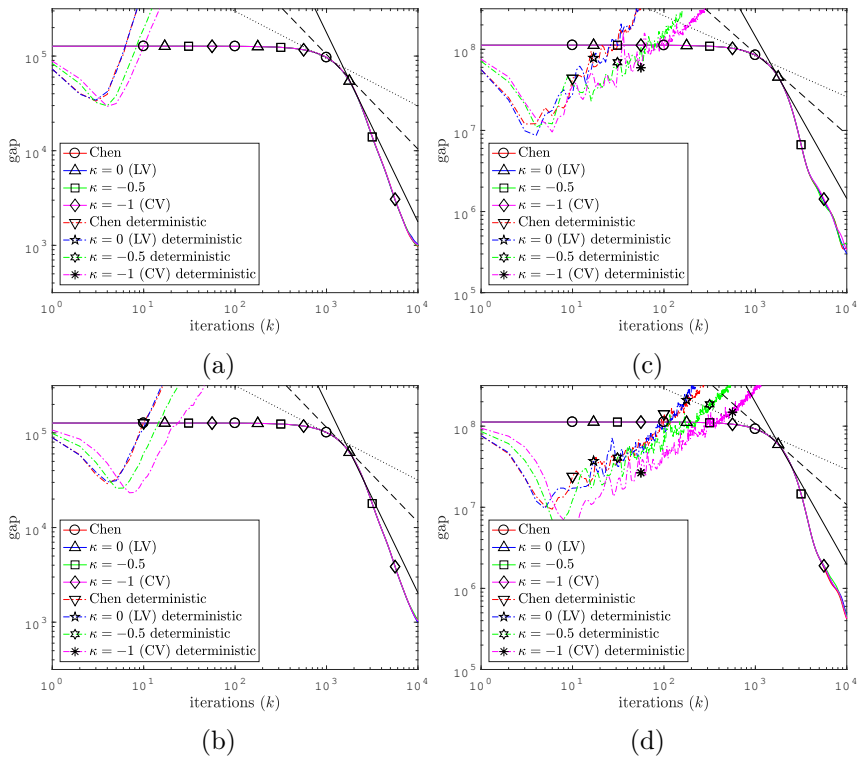


Figure 2.5: Convergence of optimal rate stochastic algorithm for a group lasso model (a-b) and a graph-guided fused lasso model (c-d). (a), (c), optimal rate stochastic algorithm assuming bounded domain (2.50) ("optimal") compared to ergodic convergence of the FB algorithm. (b), (d), optimal rate stochastic algorithm with parameters in (2.60). The cases labeled "deterministic" in the legend denote the deterministic-case parameters given by (2.28) for bounded case and (2.36) for unbounded case. Solid black lines, dashed black lines, and dotted black lines represent $O(1/k^2)$, $O(1/k)$, and $O(1/\sqrt{k})$ convergence, respectively.

Three-function optimal acceleration ($g \not\equiv 0$)

Now, we compare the practical performance of optimal three-function sum acceleration ($g \not\equiv 0$ in (2.23) and (2.43)) with the benchmark methods. For the

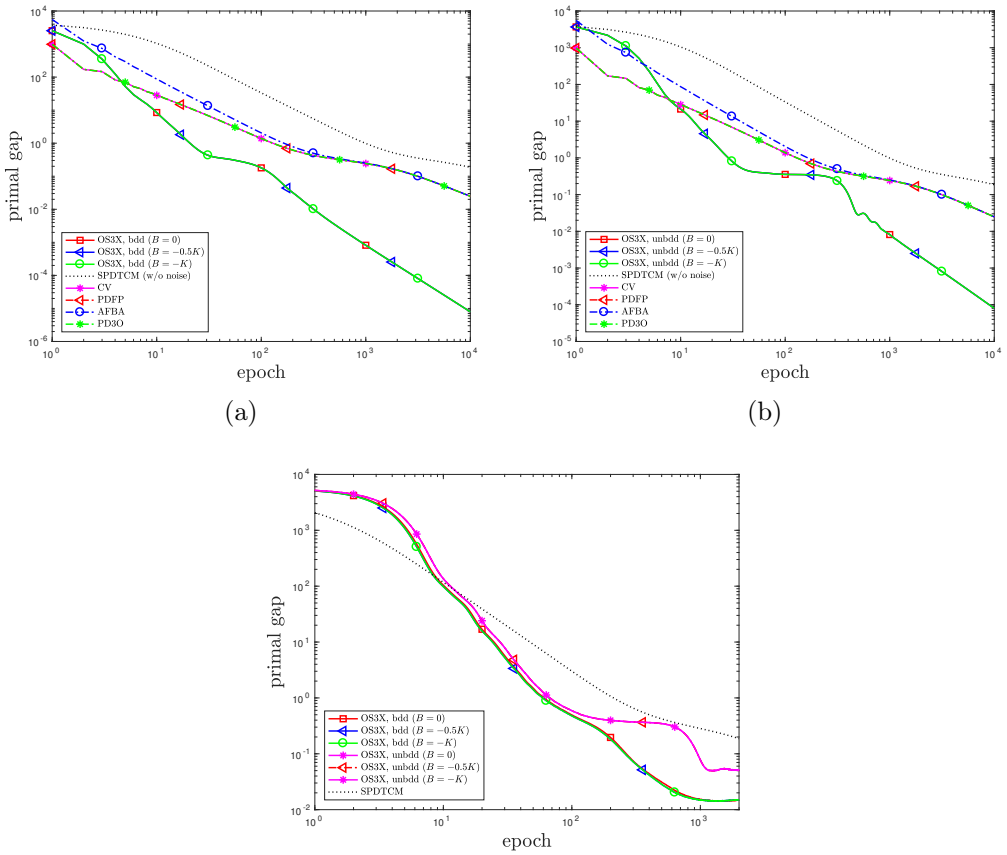


Figure 2.6: Convergence of deterministic and stochastic OS3X under various parameter settings and other methods for a sparse graph-guided fused lasso model. (a) (2.23) (labeled "OS3X") with bounded parameter settings with SPDTCM with deterministic updates, CV, PDFP, AFBA, and PD3O. (b) (2.23) (labeled "OS3X") with unbounded parameter settings with SPDTCM with deterministic updates, CV, PDFP, AFBA, and PD3O. (c) (2.43) (labeled "OS3X") with bounded and unbounded parameter settings with SPDTCM.

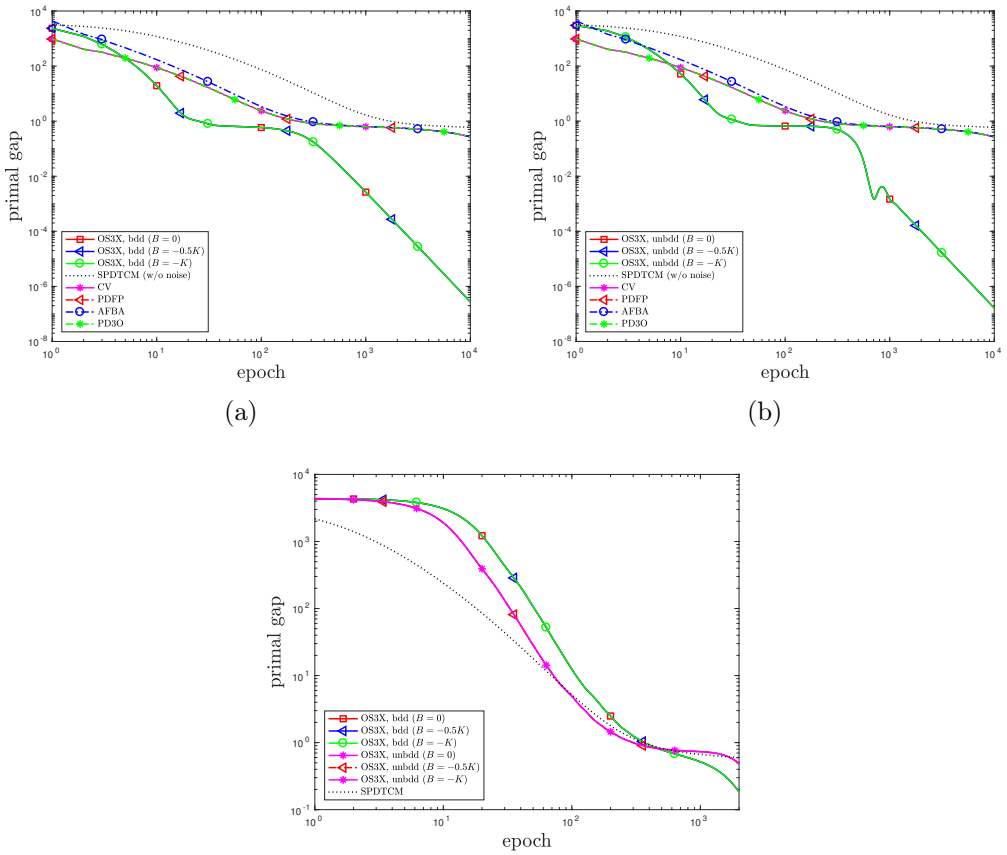


Figure 2.7: Convergence of deterministic and stochastic OS3X under various parameter settings and other methods for a overlapping group elastic net model. (a) (2.23) (labeled "OS3X") with bounded parameter settings with SPDTCM with deterministic updates, CV, PDFP, AFBA, and PD3O. (b) (2.23) (labeled "OS3X") with unbounded parameter settings with SPDTCM with deterministic updates, CV, PDFP, AFBA, and PD3O. (c) (2.43) (labeled "OS3X") with bounded and unbounded parameter settings with SPDTCM.

deterministic setting, we consider Condat-Vũ (CV), PDFP, AFBA, PD3O, and SPDTCM without noisy gradients. For the stochastic setting, we compare the accelerated method with SPDTCM with noise. The algorithms were tested with graph-guided fused lasso and overlapping group elastic net. For all stochastic experiments, 10 separate runs were averaged. For each experiment, primal gap versus the number of epochs is shown. An epoch was defined as (cumulative number of data points used in the estimation of $\hat{\mathcal{F}}$)/(number of data points in the dataset). The primal gap is the difference between the objective value at the epoch and the optimal objective value, approximated by the objective value after 100000 epochs of deterministic method (2.23). Three instances of (2.23) and (2.43) were tested: $B = 0$, $B = -0.5K$, and $B = -K$, with $A = -K$ fixed.

In the deterministic setting, from Corollary 3 and Corollary 4, $q = 0.3$, $r = 0.7$, and $P_1 = 0.9$ were chosen. For stochastic setting, (q, r, s, t) from Corollary 5 and Corollary 6 were chosen as $(0.3, 0.3, 0.7, 0.7)$. The variance χ was set to 1000. For CV, PDFP, AFBA, and PD3O, $\tau = 1.9/L_f$ and $\sigma = 1/(4\tau)$ were used. Finally, for SPDTCM, the constant parameter recipe as provided by Zhao and Cevher (2018) was utilized.

At iteration k, the stochastic gradient $\hat{\mathcal{F}}(x^k)$ was obtained from a random subsample of A. For a random permutation π, we define a subsample $\tilde{A} := \mathsf{A}_{\pi(1):\pi(n_s),:}$ (in Matlab notation), where $n_s = \lfloor 0.2n \rfloor$. Thus for the quadratic loss, we have $\hat{\{}(x^k) = (n/n_s)\tilde{A}^T(\tilde{A}x - b)$. $\hat{\mathcal{A}}$, $\hat{\mathcal{B}}$, $\hat{\mathcal{K}}_x$, and $\hat{\mathcal{K}}_y$ are estimated without artificial noise.

For graph-guided sparse fused lasso, $\lambda_1 = \lambda_2 = 1$ were used, with domain boundaries estimated as $\Omega_X = 200$, $\Omega_Y = 450$. All the iterates remained within these boundaries. For stochastic unbounded parameter settings, we chose $\tilde{R} = 100$. The results are shown in Figure 2.6. The convergence speed gap between (2.23) and the other methods is clear (note the log-log scale). Using the

parameters with known bounds is faster than using the parameters that do not involve bound assumption, but we still achieve faster convergence compared to other methods without the bound assumption. There was no noticeable difference between the choices of B.

For the overlapping group elastic net, $\lambda_1 = 0.1$ and $\lambda_2 = 0.3$ were used with $\Omega_X = 20$, $\Omega_Y = 45$. For stochastic case with unbounded parameter setting, we chose $\tilde{R} = 50$. The results are shown in Figure 2.7. All the instances of deterministic acceleration (2.23) converged faster than SPDTCM. Stochastic (2.43) starts slowly, but it surpasses SPDTCM eventually.

2.5.3 Scalability

To test the scalability of the studied algorithms, we consider the scenario that the number of features p is so large that, for each sample, the features do not fit into the memory. In other words, the data matrix $\mathsf{A} = [\mathsf{A}^{[1]}, \dots, \mathsf{A}^{[M]}]$, where $\mathsf{A}^{[i]} \in \mathbb{R}^{n \times p_i}$, $\sum_{i=1}^{m} p_i = p$, is stored distributedly in M devices. In this case, it is desirable to also split the vectors $x \in \mathbb{R}^p$ conformally and store distributedly, i.e., $x = [x_{[1]}^T, \dots, x_{[M]}^T]^T$, $x_{[i]} \in \mathbb{R}^{p_i}$. For many instances of (2.1) including the generalized lasso and group lasso, $l \gtrsim p$, so it is desirable to partition and store the dual variable $y \in \mathbb{R}^l$ likewise. i.e., $y = [y_{[1]}^T, \dots, y_{[M]}^T]^T$, $y_{[i]} \in \mathbb{R}^{l_i}$, $\sum_{i=1}^{m} l_i = l$. To compute $K^T y$ and Kx efficiently, it is desirable to also distribute rows and columns of K across the devices, i.e., $K^T = [K_{[1]}^T, \dots, K_{[M]}^T]$ and $K = [K^{[1]}, \dots, K^{[M]}]$, where $K_{[i]} \in \mathbb{R}^{l_i \times p}$ and $K^{[i]} \in \mathbb{R}^{l \times p_i}$. Duplicating K does not incur too much cost, as K is typically sparse. Then, we can carry out computation in a distributed fashion as follows.

Suppose that device i stores $\mathsf{A}_{[i]}$, $K_{[i]}$, $K^{[i]}$, $x_{[i]}$, and $y_{[i]}$. To compute $\mathsf{A}x$, we compute $\mathsf{A}^{[k]} x_{[k]}$ within each device, and aggregate the result in a master device. The communication cost required is $O(n)$. Computing Kx is more com-

plicated. Denote the submatrix made of the row $1 + \sum_{i'=1}^{i-1} l_{i'}$ through $\sum_{i'=1}^{i} l_{i'}$ and the column $1 + \sum_{j'=1}^{j-1} p_{j'}$ through $\sum_{j'=1}^{j} p_{j'}$ of K by $K_{[i]}^{[j]}$. First, we compute $K_{[i]}^{[j]} x_{[j]} =: [Kx]_{ij}$. Then we transfer nonzero values in each $[Kx]_{ij}$ to device i. Finally, within device i, we aggregate $[Kx]_{ij}$ over j. When the number of nonzero elements in K is $O(p)$, which is the case for both overlapping group lasso and graph-guided fused lasso, the communication cost is $O(Mp)$ in the worst case. This type of distribution is especially suitable for multi-GPU platforms. We solved the model problems using TensorFlow (Abadi et al., 2015) v1.2, which deals with inter-GPU communications automatically. The code is available at `https://github.com/kose-y/dist-primal-dual`.

Each experiment was conducted for 1100 iterations with time recorded every 100 iterations. This is repeated three times. The result for the first 100 iterations was discarded, as this figure includes the time elapsed to build computation graphs. Average time per 100 iterations and their standard deviations were computed. Table 2.2 shows that the distributed implementation is highly scalable across multiple GPUs. The algorithm runs faster with more GPUs in general; for the data that do not fit in the memory, it only requires more GPUs.

2.6 Discussion

In this chapter, a unified view to Algorithms CV and LV, two classes of primal-dual algorithms for a convex composite minimization problem based on monotone operator theory has been provided. This unification suggests a continuum of forward-backward operator splitting algorithms for this important optimization problem having many applications in statistics. It is also this unified understanding that enables us to establish the $O(1/N)$ and $o(1/\sqrt{k})$ convergence rates for the full regions of convergence of Algorithms CV and LV (and those in between) as well as the $O(L_f/N^2 + L_K/N)$ optimal asymptotic accelerations of

these algorithms and their three-function extensions. A practical implication of this understanding is that we bring these algorithms to the same arena: as they share the same convergence rate, other factors such as the ability of choosing wider step sizes can be fairly compared in empirical settings. Thus practitioners now possess more degrees of freedom in choosing from a suite of algorithms with theoretical guarantees.

The simplicity of the algorithms proposed and analyzed here also enables us to implement their distributed multi-GPU version almost painlessly using existing packages. This contrasts to the previous works (Yu et al., 2015; Lee et al., 2017), which resort to exploiting the structure of the matrix K in (2.1).

There remain several avenues of future research. First, in this chapter a minimal assumption on the convexity of the functions is maintained since the interest is in the worst-case rates. How the bounds of the algorithm class can be improved with additional assumptions, e.g., the strong convexity of g (Ghadimi and Lan, 2012), would be of interest. Second, in the unbounded settings we assume the horizon N is fixed in advance. Using step sizes that depend on N at least dates back to Nesterov (2005); achieving optimal rates without this information is a challenging task (Zhao and Cevher (2018) report a factor of $\log N$ slowdown in the asymptotic rate). However, in many scenarios (e.g., early stopping) the knowledge of N is unavailable, thus horizon-independent convergence analysis is warranted. Third, techniques for estimating the problem parameters L_f and L_K, and combining them with algorithm parameter selection will have an important practical impact.

Table 2.2: Scalability of the distributed version of (2.18) for graph-guided fused lasso and group lasso models. Time was measured in seconds per 100 iterations. Standard deviations are listed in parentheses. Any cell with missing values indicates that the experiment failed to run due to lack of memory.

Graph-guided fused lasso

	#GPUs	1	2	3	4	5	6	7	8
#groups	p								
10000	120000	4.895	3.801	3.274	2.468	2.081	1.739	1.584	1.518
		(0.019)	(0.048)	(0.027)	(0.021)	(0.029)	(0.025)	(0.023)	(0.014)
50000	600000		20.631	13.779	11.962	10.124	8.568	7.699	6.520
			(0.253)	(0.309)	(0.126)	(0.031)	(0.058)	(0.053)	(0.050)
80000	960000			22.695	16.957	13.712	11.559	10.343	10.828
				(0.288)	(0.302)	(0.140)	(0.124)	(0.133)	(0.056)
100000	1200000				20.517	16.190	15.590	11.704	12.498
					(0.166)	(0.227)	(0.170)	(0.148)	(0.145)

Overlapping group lasso

	#GPUs	1	2	3	4	5	6	7	8
#groups	p								
1000	120010	4.828	4.156	2.973	2.465	2.102	1.853	1.591	1.538
		(0.015)	(0.057)	(0.034)	(0.014)	(0.015)	(0.012)	(0.014)	(0.015)
5000	600010		19.312	13.670	10.164	8.374	7.369	6.727	5.960
			(0.075)	(0.059)	(0.055)	(0.029)	(0.040)	(0.029)	(0.038)
8000	960010			22.792	17.044	14.722	12.671	10.866	10.103
				(0.228)	(0.101)	(0.107)	(0.157)	(0.110)	(0.080)
10000	1200010				22.210	16.658	15.386	14.088	11.689
					(0.273)	(0.049)	(0.098)	(0.104)	(0.105)

Latent group lasso

	#GPUs	1	2	3	4	5	6	7	8
#groups	p								
1000	120010	4.754	3.359	2.524	2.166	1.894	1.649	1.598	1.602
		(0.003)	(0.024)	(0.090)	(0.068)	(0.017)	(0.020)	(0.053)	(0.050)
5000	600010		19.133	14.378	10.888	9.299	7.883	7.386	7.251
			(0.142)	(0.083)	(0.344)	(0.451)	(0.042)	(0.025)	(0.074)
8000	960010			22.023	17.825	14.236	12.141	10.964	10.133
				(0.132)	(0.180)	(0.150)	(0.145)	(0.077)	(0.057)
10000	1200010				22.271	17.647	15.045	13.320	12.194
					(0.439)	(0.476)	(0.165)	(0.067)	(0.070)

Chapter 3

Towards Unified Programming for High-Performance Statistical Computing Environments

3.1 Introduction

As introduced earlier, increasing computing power is often achieved by using more cores or machines. Supercomputers and local clusters utilize multiple cores of CPUs over multiple machines with fast communication between the machines. Also, GPUs are now widely used for accelerating many computing tasks involving linear algebra and convolution operations. In addition, with maturing of cloud computing, users can now access virtual clusters through cloud service providers without need of purchasing and maintaining the machines physically. With the demand for analysis of terabyte- or petabyte-scale data in diverse disciplines, the crucial factor for the success of large-scale data analysis is how well we utilize high-performance computing hardware in the statistical computing task.

However, statistical community appears yet to fully embrace the power of high-performance computing. This is partly because of difficulty of programming in multiple nodes and on GPUs in R, the most popular programming environment among statisticians. Furthermore, researchers often face the burden of writing separate code for different hardware environments. While there are a number of packages in high-level languages including R, Python, or Julia that simplifies GPU programming and multi-node programming separately (reviewed in Section 3.2), the package choices that enable simplification for both multi-GPU programming and multi-node programming with the same code base is limited. This leads to necessity of a tool that merges programming for multiple hardware environments.

In this chapter, a Python package `dist_stat` and a Julia (Bezanson et al., 2017) package `DistStat.jl` are introduced, which define an easy-to-use distributed array data structure over distributed CPU and GPU environments in a Python package `PyTorch` and the Julia programming language, respectively. Users can decide underlying array implementation to work on CPU cores or GPUs only with minor configuration changes.

To make the contrast clear, examples from the landmark paper for GPU in statistical computing (Zhou et al., 2010), nonnegative matrix factorization (NMF), positron emission tomography (PET), and multidimensional scaling (MDS) are deliberately chosen and implemented with `dist_stat` and `DistStat.jl`. 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, CUDA C implementation of 2010. Furthermore,

using the power of cloud computing and modern deep learning software and programming language, it is shown that exactly the *same* code can run on multiple CPU cores and/or clusters of workstations.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 400,000 subjects), featuring time-to-onset of Type 2 Diabetes (T2D) as outcome and 500,000 genomic loci harboring single nucleotide polymorphisms as covariates. To my 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 chapter is organized as follows. In Section 3.3, we review software libraries employing the "write once, run everywhere" principle, and discuss how they can be employed for fitting high-dimensional statistical models on the HPC systems of Section 1.2. How to distribute a large matrix over multiple devices is presented in Section 3.4. Numerical examples of NMF, PET, MDS, and ℓ_1-regularized Cox regression are given in Section 3.5. Finally, we conclude the chapter in Section 3.6. The code is available at `https://github.com/kose-y/dist_stat` and `https://github.com/kose-y/DistStat.jl`, and is released under the MIT License.

3.2 Related Software

3.2.1 Message-passing interface and distributed array interfaces

The de facto standard for inter-node communication in distributed computing environments is the message passing interface (`MPI`). The latter defines several ways to communicating between two processes (point-to-point communication) or among a group of processes (collective communication). Although `MPI` is originally defined in C and Fortran, many other high-level languagues have interfaces to it in the form of a wrapper, for example, `Rmpi` (Yu, 2009) for R, `mpi4py` (Dalcin et al., 2011) for Python, and `MPI.jl` (JuliaParallel Contributors, 2020) for Julia.

There have been several attempts to incorporate array and linear algebra operations through the basic syntax of the base programming language. MATLAB has a distributed array implementation that uses `MPI` as a backend in the `Parallel Computing Toolbox`. In Julia, `MPIArrays.jl` (Janssens, 2018) defines a matrix-vector multiplication routine that uses `MPI` as its backend. `DistributedArrays.jl` (JuliaParallel Contributors, 2019) is a more general attempt to create a distributed array, allowing various communication modes, including Transmission Control Protocol/Internet Protocol (TCP/IP) and Secure Shell (SSH), and MPI. In R, a package called `ddR` (Ma et al., 2016) supports distributed array operations.

3.2.2 Unified array interfaces for CPU and GPU

For GPU programming, `CUDA` C for Nvidia GPUs and `OpenCL` for general GPUs are by far the most widely used. R package `gputools` (Buckner et al., 2010) is one of the earliest efforts to incorporate GPU in R. `PyCUDA` and `PyOpenCL` (Klöckner et al., 2012) for Python and `CUDAnative.jl` and `CUDAdrv.jl` (Besard et al., 2018) for Julia allow users to access low-level

features of the respective interfaces.

For better productivity, an interface to array and linear algebra operations that works transparently on both CPU and GPU is desirable. In MATLAB, the `Parallel Computing Toolbox` includes the data structure `gpuArray`. Simply wrapping an ordinary array with the function `gpuArray()` allows the users to use predefined functions to exploit the single instruction, multiple data (SIMD) architecture of Nvidia GPUs. In Python, the recent deep learning (LeCun et al., 2015) boom accelerated development of easy-to-use array interfaces and linear algebra operations along with automatic differentiation for both CPU and GPU. The most popular among them are `TensorFlow` (Abadi et al., 2015) and `PyTorch` (Paszke et al., 2017). It is worth noting that the `Distributions` subpackage of `TensorFlow` (Dillon et al., 2017) allows convenient development of GPU computation in Bayesian setting, for example, stochastic gradient Monte Carlo Markov chain (Baker et al., 2018). In Julia, a central package named `CUDA.jl` (Besard et al., 2019; JuliaGPU Contributors, 2020) defines many array operations and simple linear algebra routines using the same syntax as the base CPU arrays.

3.3 Easy-to-use Software Libraries for HPC

3.3.1 Deep learning libraries and HPC

As of writing this dissertation (Summer 2020), the two most popular deep learning software libraries are TensorFlow (Abadi et al., 2015) and PyTorch (Paszke et al., 2017). 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). To fit a model, the software selects a backend optimized for the system in which the model runs. If the target system is a CPU node, then the software can be configured to utilize the OpenBLAS (Xianyi et al., 2014) or the 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, 2013), a GPU version of BLAS, and cuSPARSE (NVIDIA, 2018), GPU-oriented sparse linear algebra routines. Whether to run the model on a CPU or GPU can be controlled by a slight change in the option for device selection, which is usually a line or two of the script. 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 (Bergstra et al., 2011), one of the first libraries to support symbolic differentiation based on computational graphs. Unlike Theano that generates GPU code on the fly, TensorFlow is equipped with pre-compiled GPU code for a large class of pre-defined operations. The computational graph of TensorFlow is static so that a user has to pre-define all the operations prior to execution. Unfortunately, such a design does not go along well with the philosophy of scripting languages that the library should work with, and makes debugging difficult. To cope with this issue, an "eager execution" mode, which executes commands without building a computational graph, is supported.

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. Unlike TensorFlow, PyTorch uses dynamic computation graphs, so it does not require computational graphs to be pre-defined. Thanks to this dynamic execution model, the library is more intuitive and flexible to the user than most of its competitors. 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., 2015).

Both libraries support multi-GPU and multi-node computing. In Tensorflow, multi-GPU computation is supported natively. If data are distributed in multiple GPUs and one needs data from the other, the GPUs communicate implicitly and the user does not need to care. Multi-node communication is more subtle: while remote procedure call is supported natively in the same manner as multi-GPU communications, it is recommended to use MPI through the library called Horovod (Sergeev and Del Balso, 2018) for tightly-coupled HPC environments (more information is given in Section 3.3.2). 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 the *parallel programming models* paragraph in Section 1.2.1), whose specific implementation is called a backend. Possible communication backends include the MPI, Nvidia Collective Communications Library (NCCL), and Gloo (Solo.io, 2019). NCCL is useful for a multi-GPU node; (CUDA-aware) MPI maps multi-GPU communications to the MPI standard as well as traditional multi-node communications; Gloo is useful in cloud environments.

This feature of unified interfaces for various HPC environments is supported through operator overloading or polymorphism in modern programming languages, but achieving this seamlessly with a single library, along with multi-

device support, is remarkable. This is partially because of injection of capital in pursuit of commercial promises of deep learning (TensorFlow is being developed by Google, and PyTorch by Facebook). 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.

3.3.2 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 a 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.

Listing 3.1 shows the Monte Carlo π estimation code for 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 $[0, 1]^2$ and computes the proportion of the points that fall inside the quarter circle of unit radius centered at the origin. Listing 3.1 is a fully executable program. It uses `torch.distributed` interface with an MPI backend (Line 3). An instance of the program of Listing 3.1 is attached to a device and is executed as a "process". Each process is given its identifier (rank), which is retrieved in Line 5. The total number of processes is known to each process via Line 6. After the proportion of the points in the quarter-circle is computed in Line 17, each process gathers the sum of the means computed from all the processes in Line 18 (this is called the

all-reduce operation; see Section 1.2.1). Line 19 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 3.1 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. Lines 9–10 specify that the devices to be used in the program are GPUs. If the environment is a cluster with multiple CPU nodes (or even a single node), then all we need to do is changing Line 9 to `device = 'cpu'`. The resulting code runs on a cluster seamlessly.

In TensorFlow, however, a separate treatment to multi-GPU and cluster settings is almost necessary. The code for multi-GPU setting is similar to Listing 3.1 hence given in Appendix 3.2. 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 being much intuitive. 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 13th most powerful supercomputers in the world (as of November 2019), 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, 2019). Using MPI with TensorFlow requires an external library called Horovod and a substantial modification of the code, as shown in Listing 3.3. This is a sharp contrast to Listing 3.1, where essentially the same PyTorch code can be used in both multi-GPU

Listing 3.1 Distributed Monte Carlo estimation of π for PyTorch

```
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
if device.startswith('cuda'): torch.cuda.set_device(rank)

def mc_pi(n):
    # this code is executed on each device.
    x = torch.rand((n), dtype=torch.float64, device=
            device)
    y = torch.rand((n), dtype=torch.float64, device=
            device)
    # compute local estimate of pi
    r = 4 * torch.mean((x**2 + y**2 <1).to(dtype=
                    torch.float64))
    # sum of 'r's in each device is stored in 'r'
    dist.all_reduce(r)
    return r / size

if __name__ == '__main__':
    n = 10000
    r = mc_pi(n)
    if rank == 0:
        print(r.item())
```

and multi-node settings.

Therefore we employ PyTorch in the sequel to implement the highly parallelizable algorithms of Section 1.3 in a multi-GPU node and a cluster on a cloud,

Listing 3.2 Monte Carlo estimation of π for TensorFlow on a workstation with multiple GPUs

```
import tensorflow as tf

# Enforce graph computation. With eager execution,
# the code runs sequentially w.r.t. GPUs. e.g.,
# computation for '/gpu:1' would not
# start until the computation for '/gpu:0' finishes.
@tf.function
def mc_pi(n, devices):
    estim = []
    for d in devices:
        # use device d in this block
        with tf.device(d):
            x = tf.random.uniform((n,), dtype=tf.float64)
            y = tf.random.uniform((n,), dtype=tf.float64)
            # compute local estimate of pi
            # and save it as an element of 'estim'.
            estim.append(tf.reduce_mean(tf.cast(x ** 2 +
                y ** 2 < 1, tf.float64)) * 4)
    return tf.add_n(estim)/len(devices)

if __name__ == '__main__':
    n = 10000
    devices = ['/gpu:0', '/gpu:1', '/gpu:2', '/gpu:3']
    r = mc_pi(n, devices)
    print(r.numpy())
```

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.)

3.3.3 A brief introduction to PyTorch

In this section, we introduce simple operations on PyTorch. Note that Python uses 0-based, row-major ordering, like C and C++ (R is 1-based, column-major

ordering). First we import the PyTorch library. This is equvalent to `library()` in R.

```
import torch
```

Tensor creation

The following is equivalent to `set.seed()` in R.

```
torch.manual_seed(100)
```

One may create an uninitialized tensor. This creates a 3×4 tensor (matrix).

```
torch.empty(3, 4) # uninitialized tensor
```

```
tensor([[-3934621601449902 08.0000,       0.0000,
         -3934621601449902 08.0000,       0.0000],
        [                    0.0000,       0.0000,
                             0.0000,       0.0000],
        [                    0.0000,       0.0000,
                             0.0000,       0.0000]])
```

This generates a tensor initialized with random values from $(0, 1)$.

```
y = torch.rand(3, 4) # from Unif(0, 1)
```

```
tensor([[0.1117, 0.8158, 0.2626, 0.4839],
        [0.6765, 0.7539, 0.2627, 0.0428],
        [0.2080, 0.1180, 0.1217, 0.7356]])
```

We can also generate a tensor filled with zeros or ones.

```
z = torch.ones(3, 4) # torch.zeros(3, 4)
```

```
tensor([[1., 1., 1., 1.],
        [1., 1., 1., 1.],
        [1., 1., 1., 1.]])
```

A tensor can be created from standard Python data.

```
w = torch.tensor([3, 4, 5, 6])
```

```
tensor([3, 4, 5, 6])
```

Indexing

The following are standard method of indexing tensors.

```
y[2, 3] # indexing: zero-based,
# returns a 0-dimensional tensor
```

```
tensor(0.7356)
```

The indexing always returns a (sub)tensor, even for scalars (treated as zero-dimensional tensors). A standard Python number can be returned by using `.item()`.

```
y[2, 3].item() # A standard Python floating-point number
```

```
0.7355988621711731
```

To get a column from a tensor, we use the indexing as below. The syntax is similar but slightly different from R.

```
y[:, 3] # 3rd column. The leftmost column is 0th.
# cf. y[, 4] in R
```

```
tensor([0.4839, 0.0428, 0.7356])
```

The following is for taking a row.

```
y[2, :] # 2nd row. The top row is 0th. cf. y[3, ] in R
```

```
tensor([0.2080, 0.1180, 0.1217, 0.7356])
```

Simple operations

Here we provide an example of simple operations on PyTorch. Addition using the operator '+' acts just like anyone can expect:

```
x = y + z # a simple addition.
```

```
tensor([[1.1117, 1.8158, 1.2626, 1.4839],
        [1.6765, 1.7539, 1.2627, 1.0428],
        [1.2080, 1.1180, 1.1217, 1.7356]])
```

Here is another form of addition.

```
x = torch.add(y, z) # another syntax for addition
```

The operators ending with an underscore (_) change the value of the tensor in-place.

```
y.add_(z) # in-place addition
```

```
tensor([[1.1117, 1.8158, 1.2626, 1.4839],
        [1.6765, 1.7539, 1.2627, 1.0428],
        [1.2080, 1.1180, 1.1217, 1.7356]])
```

Concatenation

We can concatenate the tensors using the function `cat()`, which resembles `c()`, `cbind()`, and `rbind()` in R. The second argument indicates the dimension that the tesors are concatenated along: zero means by concatenation rows, and one means by columns.

```
torch.cat((y, z), 0) # along the rows
```

```
tensor([[1.1117, 1.8158, 1.2626, 1.4839],
        [1.6765, 1.7539, 1.2627, 1.0428],
        [1.2080, 1.1180, 1.1217, 1.7356],
        [1.0000, 1.0000, 1.0000, 1.0000],
        [1.0000, 1.0000, 1.0000, 1.0000],
        [1.0000, 1.0000, 1.0000, 1.0000]])
```

```
torch.cat((y, z), 1) # along the columns
```

```
tensor([[1.1117, 1.8158, 1.2626, 1.4839,
            1.0000, 1.0000, 1.0000, 1.0000],
        [1.6765, 1.7539, 1.2627, 1.0428,
            1.0000, 1.0000, 1.0000, 1.0000],
        [1.2080, 1.1180, 1.1217, 1.7356,
            1.0000, 1.0000, 1.0000, 1.0000]])
```

Reshaping

One can reshape a tensor, like changing the attribute `dim` in R.

```
y.view(12) # 1-dimensional array
```

```
tensor([1.1117, 1.8158, 1.2626, 1.4839,
        1.6765, 1.7539, 1.2627, 1.0428,
        1.2080, 1.1180, 1.1217, 1.7356])
```

Up to one of the arguments of `view()` can be -1. The size of the reshaped tensor is inferred from the other dimensions.

```
# reshape into (6)-by-2 tensor;
# (6) is inferred from the other dimension
y.view(-1, 2)
```

```
tensor([[1.1117, 1.8158],
        [1.2626, 1.4839],
        [1.6765, 1.7539],
        [1.2627, 1.0428],
        [1.2080, 1.1180],
        [1.1217, 1.7356]])
```

3.3.4 A brief introduction to Julia

Julia is a high-level programming language that has a flavor of scripting language such as R and Python, but compiles for efficient execution via LLVM (Lattner and Adve, 2004). Its syntax is similar to those of MATLAB and R, leading to easy-to-read code that can run on various hardware with only minor changes, including CPUs and GPUs. In this section, we review the basic syntax of Julia. Our description regarding Julia is based on the version 1.4. For more details, see the official documentation (Julia Contributors, 2020).

3.3.5 Methods and multiple dispatch

In Julia, a function is "an object that maps a tuple of argument values to a return value." A function can have different specific implementations, depending on the

types of input arguments. This feature is called **multiple dispatch**, and each specific implementation is called a **method**. Many core functions in Julia have several methods attached to each of them. A user can also define additional methods to existing functions. For example, a method for function `f` can be defined as follows:

```
julia> f(x, y) = "foo"
f (generic function with 1 method)
```

Each argument can be constrained to certain type, for example:

```
julia> f(x::Float64, y::Float64) = x * y
f (generic function with 2 methods)

julia> f(x::String, y::String) = x * y
f (generic function with 3 methods)
```

`Float64` is the data type for a double-precision (64-bit) floating point number. An asterisk (`*`) between two `String` objects means string concatenation in Julia. At runtime, the most specific method is used for the given combination of input arguments.

```
julia> f("Candy", 3.0)
"foo"

julia> f("test", "me")
"testme"

julia> f(2.0, 3.0)
6.0
```

Methods and types may have parameters, enclosed by curly braces (`{}`). A parametric method is defined as follows:

```
julia> g(x::T, y::T) where {T <: Real} = x * y
g (generic function with 1 method)
```

The function `g()` performs multiplication of the two arguments if the two arguments are the same subtype of `Real` (a type for real numbers, for example, `Float64`, `Int32` (32-bit integer), etc.) and they are of the same type.

```
julia> g(2.0, 3.0)
6.0

julia> g(2, 3)
6

julia> g(2.0, 3)
ERROR: MethodError: no method matching g(::Float64,
        ::Int64)
Closest candidates are:
  g(::T<:Real, ::T<:Real) where T<:Real at REPL[17]:1
Stacktrace:
 [1] top-level scope at REPL[28]:1
```

The third command throws an error, because the two arguments have different types. Such an error can be avoided by defining a more general method:

```
julia> g(x::Real, y::Real) = x * y
g (generic function with 2 methods)
```

Here, the exact type of `x` and `y` may be different. An example of parametric types, `AbstractArray`, is discussed in Section 3.3.6.

3.3.6 Multidimensional arrays

An array in Julia is defined as "a collection of objects stored in a multi-dimensional grid". Each object should be of a specific type for optimized performance, such as `Float64`, `Int32`, or `String`.

The top-level abstract type for a multidimensional array is `AbstractArray{T,N}`, where parameter `T` is the type of element (such as `Float64`, `Int32`), and `N` is the number of dimensions. `AbstractVector{T}` and `AbstractMatrix{T}` are aliases for `AbstractArray{T, 1}` and

`AbstractArray{T, 2}`, respectively. Operations for `AbstractArrays` are provided as fallback methods which would generally work correctly in many cases, but are often slow.

The type `DenseArray` is a subtype of `AbstractArray` representing an array stored in contiguous CPU memory. The most frequently used instance of `DenseArray` is `Array`, a type for basic CPU array with a grid structure. `Vector{T}` and `Matrix{T}` are aliases for `Array{T, 1}` and `Array{T, 2}`, respectively. Another subtype of `DenseArray` is `CuArray`, defined in `CUDA.jl`, a contiguous array data type on a `CUDA` GPU. Many of array operations for `CuArray` are provided using the same syntax as `Arrays`.

A `Matrix` (or an instance of `Array{T, 2}`) is easily created using a MATLAB-like syntax such as:

```
julia> A = [1 2; 3 4]
2×2 Array{Int64,2}:
 1  2
 3  4
```

An `Array` can be allocated with undefined values using:

```
julia> B = Array{Float64}(undef, 2, 3)
2×3 Array{Float64,2}:
 6.90922e-310  6.90922e-310  6.90922e-310
 6.90922e-310  6.90922e-310  6.90921e-310
```

There are predefined basic functions for array operations, such as `size(A)` that returns a tuple of dimensions of `A`, `eltype(A)` that returns the type of elements in `A`, and `ndims(A)`, that shows the number of dimensions of `A`.

3.3.7 Matrix multiplication

Linear algebra operations in Julia are defined in the basic package LinearAlgebra. The functions in LinearAlgebra can be loaded to the workspace with the keyword `using`:

```
julia> using LinearAlgebra
```

Matrix multiplication in Julia is defined in the function `LinearAlgebra.mul!(C, A, B)`.[1] This function computes the postmultiplication of matrix `B` to matrix `A`, and stores the result in matrix `C`. The most general definition of `LinearAlgebra.mul!()` is:

```
LinearAlgebra.mul!(C::AbstractMatrix, A::AbstractVecOrMat,
    B::AbstractVecOrMat)
```

which implements a naive algorithm for matrix multiplication. For a `Matrix` stored in the CPU memory, the call to `LinearAlgebra.mul!()` with arguments

```
LinearAlgebra.mul!(C::Matrix, A::Matrix, B::Matrix)
```

invokes the `gemm` (general matrix multiplication) routine of the `BLAS`, or the basic linear algebra subprograms (Blackford et al., 2002), e.g.,

```
julia> A= [1. 2.; 3. 4.]
2×2 Array{Float64,2}:
 1.0  2.0
 3.0  4.0

julia> B = [3. 4.; 5. 6.]
2×2 Array{Float64,2}:
 3.0  4.0
 5.0  6.0

julia> C = Array{Float64, 2}(undef, 2, 2)
2×2 Array{Float64,2}:
 6.90922e-310  6.90921e-310
 6.90922e-310  6.90922e-310

julia> mul!(C, A, B)
2×2 Array{Float64,2}:
```

[1] It is a convention in Julia to end the name of a function that changes the value of its arguments with an exclamation mark (`!`).

```
 13.0  16.0
 29.0  36.0

julia> C
2×2 Array{Float64,2}:
 13.0  16.0
 29.0  36.0
```

On the other hand, for matrices on GPU, a call to

```
LinearAlgebra.mul!(C::CuMatrix, A::CuMatrix, B::CuMatrix)
```

results in operations using `cuBLAS` (NVIDIA, 2013), a high-level linear algebra subroutines for `CUDA`, e.g.,

```
julia> using CUDA

julia> A_d = cu(A)
2×2 CuArray{Float32,2,Nothing}:
 1.0  2.0
 3.0  4.0

julia> B_d = cu(B)
2×2 CuArray{Float32,2,Nothing}:
 3.0  4.0
 5.0  6.0

julia> C_d = cu(C)
2×2 CuArray{Float32,2,Nothing}:
 0.0  0.0
 0.0  0.0

julia> mul!(C_d, A_d, B_d)
2×2 CuArray{Float32,2,Nothing}:
 13.0  16.0
 29.0  36.0
```

The function `cu()` transforms an `Array{T, N}` into a `CuArray{Float32, N}`.

3.3.8 Dot syntax for vectorization

Julia has a special "dot" syntax for vectorization. The dot syntax is invoked by prepending a dot to an operator (e.g., `.+`) or postpending a dot to a function name (e.g., `soft_threshold.()`). Unlike many other programming languages, vectorization in Julia can be applied to any function without a need to deliberately tailor the corresponding method. Julia's JIT compiler automatically matches singleton dimensions of array arguments to the dimensions of other array arguments. For example,

```
julia> a = [1, 2]
2-element Array{Int64,1}:
 1
 2

julia> b = [3 4; 5 6]
2×2 Array{Int64,2}:
 3  4
 5  6

julia> a .+ b
2×2 Array{Int64,2}:
 4  5
 7  8
```

Note that `a` is a column vector and `b` is a matrix.

The dot syntax can be extended by defining the method `broadcast()` for each array interface, allowing its generalization to any underlying hardware architecture. In addition, multiple dots on the same line of code fuse into one call to the function `broadcast()`, resulting in a single vectorized loop (for CPU) or a single generated kernel (for GPU) for that line.

While broadcasting is one of the simplest way to represent generalized elementwise operations, it may not be the fastest option. Broadcasting often allocates excessive memory, thus well-optimized compiled loops without memory

allocation may be faster in many cases.

3.4 Distributed matrix data structure

For the forthcoming examples and potential future uses in statistical computing, simple distributed data structures are proposed. For `dist_stat`, the structure is named `distmat`. For `DistStat.jl`, it is named `MPIArray`. In these structures, each process, enumerated by its rank, holds a contiguous block of the full data matrix by rows or columns. The data may be a sparse matrix in `distmat`. If GPUs are involved, each process controls a GPU whose index matches the process rank. For notational simplicity, the dimension to split is denoted by a pair of square brackets. If a $[100] \times 100$ matrix is split over four processes, the process with rank 0 keeps the first 25 rows of the matrix, and the rank 3 process takes the last 25 rows. For `dist_stat`, it is assumed that the size along the split dimension is divided by the number of processes. Such constraint is lifted for `DistStat.jl`. The code along with the examples in Section 3.5 is available at `https://github.com/kose-y/dist_stat` and `https://github.com/kose-y/DistStat.jl`.

3.4.1 Distributed matrices in PyTorch: `distmat`

In `distmat`, unary elementwise operations such as exponentiation, square root, absolute value, and logarithm of matrix entries were implemented in an obvious way. Binary elementwise operations such as addition, subtraction, multiplication, division were implemented in a similar manner to R's vector recycling. For example, if two matrices of different dimensions are to be added together, say one is three-by-four and the other is three-by-one, the latter matrix is expanded to a three-by-four matrix with the column repeated four times. Another example is adding a one-by-three matrix and a four-by-one matrix. The former

matrix is expanded to a four-by-three matrix by repeating the row four times, and the latter to a four-by-three matrix by repeating the column three times. Application of this concept is natural using the broadcast semantics of PyTorch. Reduction operations, such as row-wise (column-wise, and matrix-wise) summation, (maximum, and minimum) were also implemented in a similar fashion.

Matrix multiplications are more subtle. Six different scenarios of matrix-matrix multiplications, each representing a different configuration of the split dimension of two input matrices and the output matrix, were considered and implemented. These scenarios are listed in Table 3.1. Note that "broadcasting" and "reduction" in this subsection and the upcoming subsection are defined over a matrix dimension (rows or columns), unlike in the other parts of this dissertation where they are defined over multiple processes or ranks. The implementation of each case is carried out using the collective communication directives introduced in Section 1.2.1. Matrix multiplication scenarios are automatically selected based on the shapes of the input matrices A and B, except for the Scenarios 1 and 3 sharing the same input structure. Those two are further distinguished by the shape of output, AB. The nonnegative matrix factorization example of Section 3.5.1, which utilizes `distmat` most heavily among others, involves Scenarios 1 to 5. Scenario 6 is for matrix-vector multiplications, where broadcasting small vectors is almost always efficient.

In Listing 3.4, we demonstrate an example usage of `distmat`. We assume that this program is run with 4 processes (`size` in Line 5 is `4`). Line 11 generates a $[4] \times 4$ double-precision matrix on CPU sampled from the uniform distribution. The function `distgen_uniform` has an optional argument `TType` that allows users to choose the data type and location of the matrix: Line 10 specifies the matrix to be a double-precision matrix on CPU. The user may change it to `torch.cuda.FloatTensor` to create this matrix on a GPU with single-

Table 3.1: Six distributed matrix multiplication configurations. Matrix with no distributed dimension is duplicated in all the processes. Sizes p and q are assumed to be much larger than r and s.

	A	B	AB	Description	Usage in Section 3.5
1	$r \times [p]$	$[p] \times q$	$r \times [q]$	Inner product, result distributed.	$V^T X$
2	$[p] \times q$	$[q] \times r$	$[p] \times r$	Fat matrix multiplied by a thin and tall matrix.	XW^T
3	$r \times [p]$	$[p] \times s$	$r \times s$	Inner product, result broadcasted. Suited for inner product between two thin matrices.	$V^T V$, WW^T
4	$[p] \times r$	$r \times [q]$	$[p] \times q$	Outer product, may require large amount of memory. Often used to compute objective function.	VW
5	$[p] \times r$	$r \times s$	$[p] \times s$	A distributed matrix multiplied by a small, broadcasted matrix.	VC where $C = WW^T$; CW where $C = V^T V$ (transposed)
6	$r \times [p]$	$p \times s$	$r \times s$	A distributed matrix multiplied by a thin and tall broadcasted matrix. Intended for matrix-vector multiplications.	Ax

precision. Line 13 multiplies the two matrices `A` and `B` to form a distributed matrix of size $[4] \times 2$. The matrix multiplication routine internally chooses to utilize Scenario 2 in Table 3.1. In order to compute $\log(1 + AB)$ elementwise, all that is needed to do is to write `(1 + AB).log()` as in Line 17. Here, $1 + AB$ is computed elementwise first, then its logarithms are computed. The local block of data can be accessed by appending `.chunk` to the name of the distributed matrix, as in Lines 16 and 20.

3.4.2 Distributed arrays in Julia: `MPIArray`

`DistStat.jl` implements a distributed `MPI`-based array data structure `MPIArray` as the core data structure for implementations of `AbstractArrays`. It uses `MPI.jl` as a backend. It has been tested for basic `Arrays` and `CuArrays`. The standard vectorized "dot" operations can be used for convenient element-by-element operations as well as broadcasting operations on `MPIArrays`. Furthermore, simple distributed matrix operations for `MPIMatrix`, or two-dimensional `MPIArrays`, are also implemented. Reduction and accumulation operations are supported for `MPIArrays` of any dimension. The package can be loaded by:

```
using DistStat
```

If GPUs are available, one that is to be used is automatically selected in a round-robin fashion upon loading the package. The rank, or the "ID" of a process, and the size, or the total number of the processes, can be accessed by:

```
DistStat.Rank()
DistStat.Size()
```

Ranks are indexed 0-based, following the `MPI` standard.

In `DistStat.jl`, a distributed array data type `MPIArray{T,N,AT}` is defined. Here, parameter `T` is the type of each element of an array, e.g., `Float64` or `Float32`. Parameter `N` is the dimension of the array, `1` for vector and `2` for

matrix, etc. Parameter `AT` is the implementation of `AbstractArray` used for base operations: `Array` for the basic CPU array, and `CuArray` for the arrays on Nvidia GPUs (requires `CUDA.jl`). If there are multiple CUDA devices, a device is assigned to a process automatically by the rank of the process modulo the size. This assignment scheme extends to the setting in which there are multiple GPU devices in multiple CPU nodes. The type `MPIArray{T,N,AT}` is a subtype of `AbstractArray{T, N}`. In `MPIArray{T,N,AT}`, each rank holds a contiguous block of the full data in `AT{T,N}` split by the `N`-th dimension, or the last dimension of an `MPIArray`.

In the special case of a two-dimensional array, aliased by `MPIMatrix{T,AT}`, the data are column-major ordered and column-split. The transpose of this matrix has type of

```
Transpose{T,MPIMatrix{T,AT}}
```

which is row-major ordered and row-splitted. There also is an alias for one-dimensional array `MPIArray{T,1,A}`, which is `MPIVector{T,A}`.

Creation

The syntax `MPIArray{T,N,A}(undef, m, ...)` creates an uninitialized `MPIArray`. For example,

```
a = MPIArray{Float64, 2, Array}(undef, 3, 4)
```

creates an uninitialized 3×4 distributed array based on local `Array`s of double precision floating-point numbers. The size of this array, the type of each element, and the number of dimensions can be accessed using the usual functions in Julia: `size(a)`, `eltype(a)`, and `ndims(a)`. Local data held by each process can be accessed by appending `.localarray` to the name of the array, e.g.,

```
a.localarray
```

Matrices are split as evenly as possible. For example, if the number of processes is 4 and the `size(a) == (3, 7)`, processes of ranks 0 through 2 hold the local data of size (3, 2) and the rank-3 process holds the local data of size (3, 1).

An `MPIArray` can also be created by distributing an array residing in a single process. For example, in the following code:

```
if DistStat.Rank() == 0
    dat = [1, 2, 3, 4]
else
    dat = Array{Int64}(undef, 0)
end
d = distribute(dat)
```

the data are defined in the rank-0 process, and each other process has an empty instance of `Array{Int64}`. Using the function `distribute`, the `MPIArray{Int64, 1, Array}` of the data `[1, 2, 3, 4]`, equally distributed over four processes, is created.

Filling an array

An `MPIArray` `a` can be filled with a number `x` using the usual syntax of the function `fill!(a, x)`. For example, `a` can be filled with zero:

```
fill!(a, 0)
```

Random number generation

An array can also be filled with random values, extending `Random.rand!()` for the standard uniform distribution and `Random.randn!()` for the standard normal distribution. The following code fills `a` with uniform(0, 1) random numbers:

```
using Random
rand!(a)
```

In cases such as unit testing, generating identical data for any configuration is important. For this purpose, the following interface is defined:

```
function rand!(a::MPIArray{T,N,A}; seed=nothing,
    common_init=false, root=0) where {T,N,A}
```

If the keyword argument `common_init=true` is set, the data are generated from the process with rank `root`. The `seed` can also be configured. If `common_init == false` and `seed == k`, the seed for each process is set to `k` plus the rank.

The "dot" broadcasting feature of `DistStat.jl` follows the standard Julia syntax. This syntax provides a convenient way to operate on both multi-node clusters and multi-GPU workstations with the same code. For example, the soft-thresholding operator, which commonly appears in sparse regression can be defined in the element level:

```
function soft_threshold(x::T, lambda::T)::T where {T
        <: AbstractFloat}
    x > lambda && return (x - lambda)
    x < -lambda && return (x + lambda)
    return zero(T)
end
```

This function can be applied to each element of an `MPIArray` using the dot broadcasting, as follows. When the dot operation is used for an `MPIArray{T,N,AT}`, it is naturally passed to inner array implementation `AT`. Consider the following arrays filled with random numbers from the standard normal distribution:

```
a = MPIArray{Float64, 2, Array}(undef, 2, 4) |> randn!
b = MPIArray{Float64, 2, Array}(undef, 2, 4) |> randn!
```

The function `soft_threshold()` is applied elementwisely as the following:

```
a .= soft_threshold.(a .+ 2 .* b, 0.5)
```

The three dot operations, `.=`, `.+`, and `.*`, are fused into a single loop (in CPU) or a single kernel (in GPU) internally.

A singleton non-last dimension is treated as if the array is repeated along that dimension, just like `Array` operations. For example,

```
c = MPIArray{Float64, 2, Array}(undef, 1, 4) |> rand!
a .= soft_threshold.(a .+ 2 .* c, 0.5)
```

works as if `c` were a 2×4 array, with its content repeated twice. It is a little bit subtle with the last dimension, as the `MPIArray{T,N,AT}`s are split along that dimension. It works if the broadcast array has the type `AT` and holds the same data across the processes. For example,

```
d = Array{Float64}(undef, 2, 1); fill!(d, -0.1)
a .= soft_threshold.(a .+ 2 .* d, 0.5)
```

As with any dot operation in Julia, the dot operations for `DistStat.jl` are convenient but usually not the fastest option. Its implementations can be further optimized by specializing in specific array types. An example of this is given in Section 3.5.4.

Reduction operations, such as `sum()`, `prod()`, `maximum()`, `minimum()`, and accumulation operations, such as `cumsum()`, `cumsum!()`, `cumprod()`, `cumprod!()`, are implemented just like their base counterparts, computing cumulative sums and products. Example usages of `sum()` and `sum!()` are:

```
sum(a)
sum(abs2, a)
sum(a, dims=1)
sum(a, dims=2)
sum(a, dims=(1,2))
sum!(c, a)
sum!(d, a)
```

The first line computes the elementwise sum of `a`. The second line computes the sum of squared absolute values (`abs2()` is the method that computes the

squared absolute values). The third and fourth lines compute the column sums and row sums, respectively. Similar to the dot operations, the third line reduces along the distributed dimensions, and returns a broadcast local `Array`. The fifth line returns the sum of all elements, but the data type is a 1×1 `MPIArray`. The syntax `sum!(p, q)` selects which dimension to reduce based on the shape of `p`, the first argument. The sixth line computes the columnwise sum and saves it to `c`, because `c` is a 1×4 `MPIArray`. The seventh line computes rowwise sum, because `d` is a 2×1 local `Array`.

Given below are examples for `cumsum()` and `cumsum!()`:

```
cumsum(a; dims=1)
cumsum(a; dims=2)
cumsum!(b, a; dims=1)
cumsum!(b, a; dims=2)
```

The first line computes the columnwise cumulative sum, and the second line computes the rowwise cumulative sum. So do the third and fourth lines, but save the results in `b`, which has the same size as `a`.

Distributed linear algebra operations are implemented as follows.

Dot product

The method `LinearAlgebra.dot()` for `MPIArrays` is defined just like the base `LinearAlgebra.dot()`, which sums all the elements after an element-wise multiplication of the two argument arrays:

```
using LinearAlgebra
dot(a, b)
```

Operations on the diagonal

The "getter" method for the diagonal, `diag!(d, a)`, and the "setter" method for the diagonal, `fill_diag!()`, are also available. The former obtains the

main diagonal of the `MPIMatrix` `a` and is stored in `d`. If `d` is an `MPIMatrix` with a single row, the result is obtained in a distributed form. On the other hand, if `d` is a local `AbstractArray`, all elements of the main diagonal is copied to all processes as a broadcast `AbstractArray`:

```
M = MPIMatrix{Float64, Array}(undef, 4, 4) |> rand!
v_dist = MPIMatrix{Float64, Array}(undef, 1, 4)
v = Array{Float64}(undef, 4)
diag!(v_dist, M)
diag!(v, M)
```

Matrix multiplication

The method `LinearAlgebra.mul!(C, A, B)` is implemented for `MPIMatrixes`, in which the multiplication of `A` and `B` is stored in `C`. Matrix multiplications for 17 different combinations of types for `A`, `B`, and `C`, including matrix-vector multiplications, are included in the package. It is worth noting that transpose of an `MPIMatrix` is a row-major ordered, row-split matrix. While the base syntax of `mul!(C, A, B)` is always available, any temporary memory to save intermediate results can also be provided as a keyword argument in order to avoid repetitive allocations in iterative algorithms, as in `mul!(C, A, B; tmp=Array(undef, 3, 4))`. The user should determine which shape of `C` minimizes communication and suits better for their application. `MPIColVector{T, AT}` is defined as `Union{MPIVector{T,AT}, Transpose{T, MPIMatrix{T,AT}}}` to include transposed `MPIMatrix` with a single row. The 17 possible combinations of arguments available are listed in Table 3.2.

Operator norms

The method `opnorm()` either evaluates (ℓ_1 and ℓ_∞) or approximates (ℓ_2) matrix operator norms, defined for a matrix $A \in \mathbb{R}^{m\times n}$ as $\|A\| = \sup\{\|Ax\| : x \in$

	C ($p \times r$)	A ($p \times q$)	B ($q \times r$)	tmp (if defined)	dimension of tmp
1	MPIMatrix	MPIMatrix	Transposed MPIMatrix	AbstractMatrix	$p \times r$
2	Transposed MPIMatrix	MPIMatrix	Transposed MPIMatrix	AbstractMatrix	$r \times p$
3	MPIMatrix	MPIMatrix	MPIMatrix	AbstractMatrix	$p \times q$
4	Transposed MPIMatrix	Transposed MPIMatrix	Transposed MPIMatrix	AbstractMatrix	$r \times q$
5	AbstractMatrix	MPIMatrix	Transposed MPIMatrix		
6	MPIMatrix	Transposed MPIMatrix	MPIMatrix	AbstractMatrix	$q \times p$
7	Transposed MPIMatrix	Transposed MPIMatrix	MPIMatrix	AbstractMatrix	$q \times r$
8	MPIMatrix	AbstractMatrix	MPIMatrix		
9	MPIMatrix	Transposed AbstractMatrix	MPIMatrix		
10	Transposed MPIMatrix	Transposed MPIMatrix	Transposed AbstractMatrix		
11	Transposed MPIMatrix	Transposed MPIMatrix	AbstractMatrix		
12	AbstractVector	MPIMatrix	AbstractVector		
13	AbstractVector	Transposed MPIMatrix	AbstractVector		
14	AbstractVector	MPIMatrix	MPIColVector		
15	MPIColVector	Transposed AbstractMatrix	AbstractVector		
16	MPIColVector	MPIMatrix	MPIColVector	AbstractVector	p
17	MPIColVector	Transposed MPIMatrix	MPIColVector	AbstractVector	q

Table 3.2: List of implementations for `LinearAlgebra.mul!(C, A, B)` with an optional keyword argument `tmp` for temporary storage.

$\mathbb{R}^n$ with $\|x\| = 1\}$ for each respective vector norm.

```
opnorm(a, 1)
opnorm(a, 2)
opnorm(a, Inf)
```

The ℓ_2-norm is estimated via the power iteration (Golub and Van Loan, 2013), and can be further configured for convergence and the number of iterations. There also is an implementation based on the inequality $\|A\|_2 \leq \|A\|_1 \|A\|_\infty$ (`method="quick"`), which overestimates the ℓ_2-norm.

```
opnorm(a, 2; method="power", tol=1e-6, maxiter=1000,
        seed=95376)
opnorm(a, 2; method="quick")
```

3.5 Examples

In this section, we compare the performance of the optimization algorithms on four statistical computing examples: nonnegative matrix factorization (NMF), positron emission tomography (PET), multidimensional scaling (MDS), and ℓ_1-regularized Cox model for survival analysis. Single-device codes are provided to show the simplicity of the programming and distribute it over a cluster composed of multiple AWS EC2 instances or a local multi-GPU workstation. For NMF and PET, we compare two algorithms, one more classical, and the other based on recent development. We evaluate the objective function once per 100 iterations. For the comparison of execution time, the iteration is run for a fixed number of iterations, regardless of convergence. For comparison of different algorithms regarding the same problem, we iterate until $\frac{|f(\theta^n) - f(\theta^{n-100})|}{(|f(\theta^n)|+1)} < 10^{-5}$. Table 3.3 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 CfnCluster.

Network bandwidth of each `c5.18xlarge` instance was 25GB/s. A separate `c5.18xlarge` instance served as the "master" instance. This instance does not participate in computation by itself but manages 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 C. For GPU experiments, a local machine with two CPUs (10 cores per CPU) and eight Nvidia GTX 1080 GPUs was used. These are desktop GPUs, not optimized for double-precision. All the experiments were conducted using PyTorch version 0.4 built on the MKL; the released code works for the versions up to 1.4, the most recent stable version as of June 2020.

For all of the experiments, the single-precision computation results on GPU were almost the same as the double-precision results up to six significant digits, except for ℓ_1-regularized Cox regression, the necessary cumulative sum operation implemented in PyTorch caused by numerical instability in some cases with small penalties. Therefore the computations for Cox regression with `dist_stat` were performed in double-precision. Extra efforts for writing a multi-device code were modest using `dist_stat` and `DistStat.jl`, less than 100 lines for each application.

As can be verified in the sequel, computing on GPUs was effective on mid-sized (around 10,000 $\times$ 10,000) datasets, but stalled on larger (around 100,000 $\times$ 100,000) datasets due to memory limitation. In contrast, the virtual clusters were not very effective on mid-sized data, and may even slow down due to communication burden. They were effective and scaled well on larger (around 100,000 $\times$ 100,000) datasets.

In general, multi-GPU implementation results of `DistStat.jl` are largely comparable to those of `dist_stat` with more GPUs. In large-scale AWS

Table 3.3: Configuration of 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 × 4–20
Total cores	20	20,480 (CUDA)	36 × 4–20

EC2 experiments, `DistStat.jl` achieved faster computation thanks to increased flexibility in process configuration. When communication is heavy, we can use the configuration with less jobs, with each job using more threads. When communication is a little bit of a problem, we can use the configuration with more jobs, with each job using a single thread. This is nearly impossible with `dist_stat`, because due to the limitation of `torch.distributed` subpackage of `PyTorch`, each process has to hold the same size of data. In addition, the `MPI` wrappers in PyTorch forces copy of data before and after the data communication, while `MPI.jl` does not.

3.5.1 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 $W \in \mathbb{R}^{r \times p}$. It is widely used in image processing, bioinformatics, and recommender systems (Wang and Zhang, 2013) where the data have only nonnegative values. One of the first effective algorithms was the multiplicative algorithm introduced by Lee and Seung (1999, 2001). In a simple setting, NMF minimizes

$$f(V, W) = \|X - VW\|_{\mathrm{F}}^2,$$

where $\|\cdot\|_{\mathrm{F}}$ denotes the Frobenius norm.

The multiplicative algorithm written using PyTorch for a single device is given as:

```
# initialize X, W, V in a single device: a CPU or a GPU.
for i in range(max_iter):
    # Update V
    XWt =  torch.mm(X, W.t()) # compute XW^T
    WWt =  torch.mm(W, W.t()) # compute WW^T
    VWWt = torch.mm(V, WWt) # compute VWW^T
    # V = V * XW^T / VWW^T elementwise in-place.
    V = V.mul_(XWt).div_(VWWt + eps)
    # Update W
    VtX  = torch.mm(V.t(), X)
    VtV  = torch.mm(V.t(), V)
    VtVW = torch.mm(VtV, W)
    W = W.mul_(VtX).div_(VtVW + eps)
```

This algorithm can be interpreted as a case of MM algorithm with a surrogate function of f based on Jensen's inequality:

$$g(V,W|V^n,W^n) \quad = \quad \sum_{i,j,k} \frac{v_{ik}^n w_{kj}^n}{\sum_{k'} v_{ik'}^n w_{k'j}^n} \left(x_{ij} - \frac{\sum_{k'} v_{ik'}^n w_{k'j}^n}{v_{ik}^n w_{kj}^n} v_{ik} w_{kj} \right)^2 .$$

The update rule is:

$$V^{n+1} = V^n \odot [X(W^n)^T] \oslash [V^n W^n (W^n)^T]$$
$$W^{n+1} = W^n \odot [(V^{n+1})^T X] \oslash [(V^{n+1})^T V^{n+1} W^n],$$

where $\odot$ and $\oslash$ denote elementwise multiplication and division, respectively.

The simple-looking code can fully utilize the shared-memory parallelism: if the matrices are stored on the CPU memory, it runs parallelly, fully utilizing OpenMP and MKL/OpenBLAS (depending on installation). If the data are stored on a single GPU, the code runs parallely utilizing GPU cores through the CUDA libraries. Distributing this algorithm on a large scale machine is

straightforward (Liu et al., 2010a). An implementation of the multiplicative algorithm of NMF in native Julia is given by:

```
for i in 1:iter
    mul!(WXt, W, transpose(X))
    mul!(WWt, W, transpose(W))
    mul!(WWtVt, WWt, Vt)
    Vt .= Vt .* WXt ./ (WWtVt .+ eps)

    mul!(VtX, Vt, X)
    mul!(VtV, Vt, transpose(Vt))
    mul!(VtVW, VtV, W)
    W .= W .* VtX ./ (VtVW .+ eps)
end
```

A very small number (`eps`) is added to the denominator for numerical stability. Exactly the same code can run on various HPC environments including multiple CPU nodes and multi-GPU workstations in a distributed fashion. In the numerical experiments, `X`, `W`, `Vt`, `WXt`, `WWtVt`, `VtX`, and `VtVW` were defined as column-distributed `MPIMatrixs` using `DistStat.jl`, and further optimization for memory efficiency was conducted.

Figure 3.1 shows an example of NMF on a publicly available hyperspectral image. It was acquired by the reflective optics system imaging spectrometer sensor in a flight campaign over Pavia University in Italy. The image is essentially a 610 (height) $\times$ 340 (width) $\times$ 103 (bands) hyperspectral cube. It is interpreted as a $207,400$ (pixels) $\times$ 103 (bands) matrix and then analyzed using NMF. The rank r was set to 20. In the resulting $207,400 \times 20$ matrix V, each column can be interpreted as a composite channel from the original 103 bands. Three of these channels showing distinct features chosen by hand are shown in Figure 3.1.

A problem with the multiplicative algorithm is the potential to generate subnormal numbers, significantly slowing down the algorithm. A subnormal

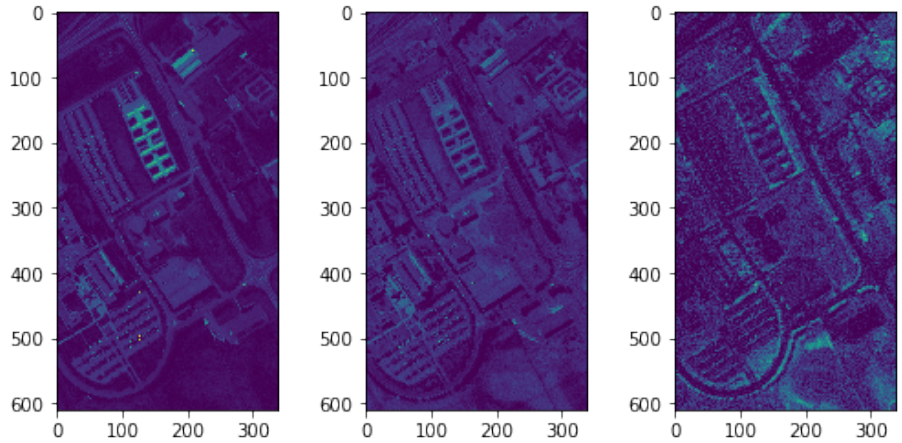


Figure 3.1: Three selected bands from the NMF of the Pavia University hyperspectral image with $r = 20$

number or denormal number is a number smaller (in magnitude) than the smallest positive number that can be represented by the floating-point number system. Subnormal numbers are generated by the multiplicative algorithm if values smaller than 1 are multiplied repeatedly. Indeed, when the NMF code was run on a CPU with a small synthetic data of size 100×100, a significant slowdown was observed. The IEEE floating-point standard is to deal with subnormal numbers properly with a special hardware or software implementation (IEEE Standards Committee, 2008). In many CPUs, the treatment of subnormal numbers relies on software and hence is very slow. Forcing such value to zero is potentially dangerous depending on applications because it becomes prone to division-by-zero error. In our experiments, division-by-zero error did not occur when flushing the subnormal numbers to zero. In contrast, Nvidia GPUs support subnormal numbers at a hardware level since the Fermi architecture, and simple arithmetic operations do not slow down by subnormal numbers (Whitehead and Fit-Florea, 2011).

Subnormal numbers can be completely avoided (especially in CPUs) by using a different algorithm. The alternating projected gradient (APG) method (Lin, 2007) is such an algorithm, and it is also easy to introduce regularization terms.With ridge penalties the objective function

$$f(V, W; \epsilon) = \|X - VW\|_{\mathrm{F}}^2 + \frac{\epsilon}{2}\|V\|_{\mathrm{F}}^2 + \frac{\epsilon}{2}\|W\|_{\mathrm{F}}^2$$

is minimized. The corresponding APG update is given by

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

where P_+ denotes the projection onto the nonnegative orthant; σ_n and τ_n are the step sizes. This update rule can be interpreted as an MM algorithm, due to the nature of projected gradient. Convergence of APG is guaranteed if $\epsilon > 0$, $\sigma_n \leq 1/(2\|W^n(W^n)^T + \epsilon I\|_{\mathrm{F}}^2)$, and $\tau_n \leq 1/(2\|(V^n)^T V^n + \epsilon I\|_{\mathrm{F}}^2)$.

For the distributed implementation, X is assumed to be an $[m] \times p$ matrix. The resulting matrix V is distributed as an $[m] \times r$ matrix, and W is distributed as an $r \times [p]$ matrix. The distributed code is equivalent to replacing `torch.mm` with `distmat.mm` in the `dist_stat` code provided, with an additional optional argument `out_sizes=W.sizes` on the tenth line. As discussed in Section 3.4, distributed matrix multiplication algorithms are automatically selected from Table 3.1 based on the arguments.

Table 3.4 compares the performance of the two NMF algorithms on the multi-GPU setting in Table 3.3 with $10,000 \times 10,000$ data for 10,000 iterations. The data are row-distributed in `dist_stat` and column-distributed in `DistStat.jl`. It can be seen that the performances are comparable between the two algorithms, with APG being slightly slower with fixed number of iter-

Table 3.4: Runtime (in seconds) of NMF algorithms on $10,000 \times 10,000$ simulated data on GPUs

	`dist_stat`			`DistStat.jl`		
GPUs	$r = 20$	$r = 40$	$r = 60$	$r = 20$	$r = 40$	$r = 60$
Multiplicative						
1	62	71	75	62	72	83
2	43	55	63	42	60	72
3				38	57	71
4	37	51	63	34	54	68
5	39	54	66	38	56	75
6				36	56	80
7				37	58	81
8	40	60	75	37	59	83
APG						
1	68	76	82	61	80	85
2	49	61	69	43	60	79
3				38	59	74
4	44	58	70	36	54	72
5	46	60	73	37	59	78
6				37	56	75
7				38	61	88
8	47	68	83	39	59	82

ations. This is because APG has slightly more operations involved. With more than 4 GPUs, the communication burden outweighs the speedup from using more GPU cores, and the algorithm becomes slower. The execution time between the `dist_stat` and `DistStat.jl` implementations are also largely comparable, with `DistStat.jl` version being faster in $r = 20$ cases. Experiments with 3, 6, or 7 GPUs were impossible with $10,000 \times 10,000$ data with `dist_stat`, because the size of dataset was not divisible by 3, 6, and 7.

Additional experiments were conducted to see how the value of ϵ affects the convergence. The results are shown in Table 3.8. Convergence was faster for higher values of ϵ. The number of iterations to convergence in the multiplicative algorithm was higher than the APG with $\epsilon = 10$ for higher-rank decompositions ($r = 40$ and 60) due to heavier communication burden.

Table 3.5: Runtime (in seconds) of NMF algorithms on $200,000 \times 200,000$ simulated data on multiple AWS EC2 instances

	`dist_stat`			`DistStat.jl`		
Instances	$r = 20$	$r = 40$	$r = 60$	$r = 20$	$r = 40$	$r = 60$
Multiplicative						
4	1419	1748	2276	1392	1576	2057
5	1076	1455	1698	1187	1383	1847
8	859	966	1347	851	936	1430
10	651	881	1115	708	856	1065
16	549	700	959	553	694	907
20	501	686	869	554	672	832
APG						
4	1333	1756	2082	1412	1711	2023
5	1088	1467	1720	1215	1372	1775
8	766	994	1396	849	916	1388
10	677	870	1165	673	799	1014
16	539	733	936	547	684	867
20	506	730	919	538	727	836

Table 3.5 compares the algorithms and implementations using $200,000 \times 200,000$ data on multiple AWS EC2 instances for 1000 iterations. For `DistStat.jl` implementation, two processes per instance were used to avoid the communication burden. Once again, elapsed time was largely similar between the two algorithms. APG is faster than the multiplicative algorithms in more cases compared to GPU, because the multiplicative algorithm on CPU often suffers from the slowdown due to creation of denormal numbers. The cluster in a cloud was scalable on larger datasets, running faster with more instances, up to 2.83x-speedup on 20-instance cluster over a four-instance cluster. Between the two implementations, the `DistStat.jl` implementation was faster in 24 out of 30 cases.

Table 3.6: Runtime (in seconds) comparisons for NMF on the simulated $[10,000] \times 10,000$ data

		10,000 iterations				
method	r	CPU	1 GPU	2 GPUs	4 GPUs	8 GPUs
Multiplicative	20	655	160	93	62	50
	40	978	165	102	73	72
	60	1355	168	109	85	86
APG	20	504	164	97	66	57
($\epsilon = 0$)	40	783	168	106	78	77
	60	1062	174	113	90	92

Table 3.7: Comparison of objective function values for simulated $[10,000] \times 10,000$ data after 10,000 iterations and 100,000 iterations

method	r	10,000 iterations	100,000 iterations
Multiplicative	20	8.270667E+06	8.270009E+06
	40	8.210266E+06	8.208682E+06
	60	8.155084E+06	8.152358E+06
APG	20	8.271248E+06	8.270005E+06
($\epsilon = 0$)	40	8.210835E+06	8.208452E+06
	60	8.155841E+06	8.151794E+06

Table 3.8: Convergence time comparisons for different values of ϵ in APG and the multiplicative method

	$r = 20$, 8 GPUs			$r = 40$, 8 GPUs			$r = 60$, 4 GPUs		
Method	iterations	time (s)	function	iterations	time (s)	function	iterations	time (s)	function
Multiplicative	21200	110	8.270530E+06	36600	269	8.209031E+06	50000	446	8.152769E+06
APG $\epsilon = 0$	31500	198	8.270202E+06	37400	310	8.208875E+06	55500	536	8.152228E+06
APG $\epsilon = 0.1$	30700	191	8.274285E+06	36700	302	8.210324E+06	55500	537	8.153890E+06
APG $\epsilon = 1$	30500	190	8.282346E+06	37300	307	8.223108E+06	47800	460	8.168503E+06
APG $\epsilon = 10$	28000	178	8.389818E+06	31000	257	8.347859E+06	46400	448	8.308998E+06

3.5.2 Positron emission tomography

Positron emission tomography (PET) is one of the earliest applications of the EM algorihtm in computed tomography (Lange and Carson, 1984; Vardi et al., 1985). In this scenario, we consider a two-dimensional imaging consisting of p pixels obtained from the circular geometry of q photon detectors. We estimate Poisson emission intensities $\lambda = (\lambda_1, \cdots, \lambda_p)$, which is proportional to the concentration of radioactively labeled isotopes injected to biomolecules. Such an isotope emits a positron, which collides with a nearby electron, forming two gamma-ray photons flying in almost opposite directions. These two photons are detected by a pair of photon detectors corresponding to the line of flight. The coincidence counts $(y_1, \ldots, y_d)$ are observed. Detector pairs are enumerated by 1, 2, $\ldots$, $d = q(q-1)/2$. The likelihood of detection for a detector pair i is modeled by Poisson distribution with mean $\sum_{j=1}^{p} e_{ij}\lambda_j$, where e_{ij} is the probability that a pair of photons is detected by the detector pair i given that a positron is emitted in the pixel location j. The matrix $E = (e_{ij}) \in \mathbb{R}^{d \times p}$ can be precomputed based on the geometry of the detectors. The corresponding loglikelihood to maximize is given by

$$L(\lambda) = \sum_{i=1}^{d} \left[y_i \log \left(\sum_{j=1}^{p} e_{ij}\lambda_j \right) - \sum_{j=1}^{p} e_{ij}\lambda_j \right].$$

Without a spatial regularization term, the reconstructed intensity map is grainy. One remedy is adding a ridge-type penalty of $-(\mu/2)\|D\lambda\|_2^2$, where D is the finite difference matrix on the pixel grid; each row of D has one $+1$ and one -1. The MM iteration based on separation of the penalty function by the minorization

$$(\lambda_j - \lambda_k)^2 \geq -\frac{1}{2}(2\lambda_j - \lambda_j^n - \lambda_k^n)^2 - \frac{1}{2}(2\lambda_k - \lambda_j^n - \lambda_k^n)^2$$

is:

$$z_{ij}^{n+1} = e_{ij} y_i \lambda_j^n \Bigg/ \left(\sum_k e_{ik} \lambda_k^n \right)$$

$$b_j^{n+1} = \mu \left(n_j \lambda_j^n + \sum_k g_{jk} \lambda_k^n \right) - 1$$

$$\lambda_j^{n+1} = \left(-b_j^{n+1} - \sqrt{(b_j^{n+1})^2 - 4 a_j \sum_{i=1}^{d} z_{ij}^{n+1}} \right) \Bigg/ (2a_j),$$

where $n_j = \sum_k g_{jk}$ and $a_j = -2\mu n_j$ are precomputed. Matrix $G = (g_{jk})$ is the adjacency matrix corresponding to the grid. See Section 3.2 of Zhou et al. (2010) for the detailed derivation. The sparse structure of G and E is exploited for software implementation in `dist_stat`. Implementation with `DistStat.jl` is omitted, since the package does not support sparse matrices yet. By using matrix notations and broadcasting semantics, the PyTorch code can be succinctly written as:

```
# G: adjacency matrix, sparse p-by-p
# mu: roughness penalty parameter
# E: detection probability matrix, d-by-p
# lambd: poisson intensity, p-by-1, randomly initialized
# y: observed data, d-by-1
# eps: a small positive number for numerical stability
N = torch.mm(G, torch.ones(G.shape[1], 1))
a = -2 * mu * N
for i in range(max_iter):
    el = torch.mm(E, lambd)
    gl = torch.mm(G, lambd)
    z  = E * y * lambd.t() / (el + eps)
    b = mu * (N * lambd + gl) -1
    c = z.sum(dim=0).t()
    # update lambda
    if mu != 0:
        lambd = (-b - (b**2-4*a*c).sqrt())/(2*a+eps)
    else:
        lambd = -c/(b+self.eps)
```

Figure 3.2 shows the results with a $p = 64 \times 64$ Roland-Varadhan-Frangakis (RVF) phantom (Roland et al., 2007) with $d = 2016$ with various values of μ, and Figure 3.4 shows the results with a 128×128 extended cardiac-torso (XCAT) phantom (Lim et al., 2018; Ryu et al., 2020) with $d = 8128$. Images get smooth as the value of μ increases, but the edges are blurry.

To promote sharp contrast, the total variation (TV) penalty (Rudin et al., 1992) can be employed. Adding an anisotropic TV penlty yields minimizing

$$-L(\lambda) + \rho\|D\lambda\|_1 = \sum_{i=1}^{d} [(E\lambda)_i - y_i \log((E\lambda)_i)] + \rho\|D\lambda\|_1,$$

which is equivalent to the formulation in Section 2.1. We can use the PDHG algorithm discussed in Section 1.3.4. Put $K = [E^T, D^T]^T$, $f(z, w) = \sum_i (-y_i \log z_i) + \rho\|w\|_1$, and $g(\lambda) = \mathbf{1}^T E\lambda + \delta_+(\lambda)$, where $\mathbf{1}$ is the all-one vector of conforming shape and δ_+ is the $0/\infty$ indicator function for the nonnegative orthant. Since $f(z, w)$ is separable in z and w, applying iteration (1.5) using the proximity operator (1.2), we obtain the following update rule:

$$\begin{aligned}
\lambda^{n+1} &= P_+(\lambda^n - \tau(E^T z + D^T w + E^T \mathbf{1})) \\
\tilde{\lambda}^{n+1} &= 2\lambda^{n+1} - \lambda^n \\
z^{n+1} &= \frac{1}{2}\left((z^n + \sigma E\tilde{\lambda}^{n+1}) - \sqrt{(z^n + \sigma E\tilde{\lambda}^{n+1})^2 + 4\sigma y}\right) \\
w^{n+1} &= P_{[-\rho,\rho]}(w^n + \sigma D\tilde{\lambda}^{n+1}),
\end{aligned}$$

where $P_{[-\rho,\rho]}$ is elementwise projection to the interval $[-\rho, \rho]$. Convergence is guaranteed if $\sigma\tau < 1/\|[ED]\|_2^2$. An implementation is given by:

```
# tau, sig: predetermined
# E: d-by-p
# D: l-by-p
# y: d-by-1, observed count
# rho: penalty parameter
```

Table 3.9: Convergence time comparisons for TV-penalized PET with different values of ρ. Problem dimension is $p = 10,000$ and $d = 16,110$. Eight GPUs were used.

ρ	iterations	time (s)	function
0	6400	20.6	-2.417200E+05
0.01	4900	15.8	-2.412787E+05
0.1	5000	16.1	-2.390336E+05
1	2800	9.5	-2.212579E+05

```
# lambd: p-by-1, randomly initialized
# z: d-by-1, initialized to -1
# w: l-by-1, initiailzed to 0

Et1 = torch.mm(E.t(), torch.ones(E.shape[0], 1))
for i in range(max_iter):
    lambd_prev = lambd
    Etz = torch.mm(E.t(), z)
    Dtw = torch.mm(D.t(), w)

    lambd = torch.clamp(lambd - tau * (Etz + Dtw + Et1),
                min=0.0)
    lambd_tilde = 2 * lambd - lambd_prev

    el = torch.mm(E, lambd_tilde)
    z_step = z + sig * el
    z  = 0.5 * (z_step - torch.sqrt(z_step ** 2 +
        4 * sig * y))

    dl = torch.mm(D, lambd_tilde)
    w_step = w + sig * dl
    w  = torch.clamp(w, max=rho, min=-rho)
```

Figures 3.3 and 3.5 are the TV-reconstructed versions of Figures 3.2 and 3.4, respectively. Compare the edge contrast.

Table 3.9 shows the convergence with different values of penalty parameters. Observe that the algorithm converges faster for large values of ρ. Scalability

experiments were carried out with large RVF-like phantoms using grid sizes $p = 300 \times 300$, 400×400, and 900×900, with the number of detectors $q = 600$ ($d = 179,700$). The matrix E is distributed as a $d \times [p]$ matrix, and the matrix D is distributed as an $l \times [p]$ matrix. The symmetric adjacency matrix G is distributed as a $[p] \times p$ matrix. The sparse structure of these matrices is exploited using the sparse tensor data structure of PyTorch. Timing per 1000 iterations is reported in Table 3.10. For reference, the data used in Zhou et al. (2010) were for 64×64 grid with $q = 64$, or $d = 2016$. Time per iterations of the PDHG method for the TV penalty is noticeably shorter as each iteration is much simpler than the MM counterpart for the ridge penalty, with no intermediate matrix created. The total elapsed time gets shorter with more GPUs. Although the speedup when adding more devices is somewhat mitigated in this case due to using sparse structure, resulting in 1.25x-speedup for 8 GPUs over 2 GPUs with $p = 160,000$, we can still take advantage of the scalability of memory with more devices.

3.5.3 Multidimensional scaling

Multidimensional scaling is one of the earliest applications of the MM principle (de Leeuw, 1977; de Leeuw and Heiser, 1977). In this example, we reduce the dimensionality of m data points by mapping them into $\theta = (\theta_1, \ldots, \theta_m)^T \in \mathbb{R}^{[m] \times q}$ in q-dimensional Euclidean space in a way that keeps the dissimilarity measure y_{ij} between the data points x_i and x_j as close as possible to that in the original manifold. In other words, we minimize the stress function

$$\begin{aligned} f(\theta) &= \sum_{i=1}^{q} \sum_{j \neq i} w_{ij} (y_{ij} - \|\theta_i - \theta_j\|_2)^2 \\ &= \sum_{i=1}^{q} \sum_{j \neq i} \left[-2 w_{ij} y_{ij} \|\theta_i - \theta_j\|_2 + w_{ij} \|\theta_i - \theta_j\|_2^2 \right] + \text{const.}, \end{aligned}$$

Table 3.10: Runtime (in seconds) comparison of 1,000 iterations of absolute-value penalized PET. Sparse structures of E and D were exploited. 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			
1	63	108	530
2	46	84	381
4	36	49	210
5	36	45	188
8	33	39	178
10	38	37	153
20	26	28	131

where the w_{ij} are the weights. We adopt the following surrogate function that majorizes f:

$$g(\theta|\theta^n) = 2\sum_{i=1}^{q}\sum_{j\neq i}\left[w_{ij}\left\|\theta_i - \frac{1}{2}(\theta_i^n + \theta_j^n)\right\|_2^2 - \frac{w_{ij}y_{ij}(\theta_i)^T(\theta_i^n - \theta_j^n)}{\|\theta_i^n - \theta_j^n\|_2}\right].$$

The corresponding update equation obtained from setting the gradient of $g(\theta|\theta^n)$ to zero is

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

for $i = 1, \ldots, n$ and $k = 1, \ldots, q$. See Zhou et al. (2010) for the detailed derivation. In PyTorch syntax, this can be parallely computed by the code:

```
# initialize theta from Unif(-1, 1)
for i in range(max_iter):
    # compute Z_{ij} = y_{ij} /
    #    \|\theta^i - \theta^j\|_2^2
```

```
    d   = torch.mm(self.theta, self.theta.t())
    # to broadcast the below
    TtT_diag = torch.diag(d).view(-1, 1)

    d   = d.mul_(-2.0)
    d.add_(TtT_diag)
    d.add_(TtT_diag.t())
    # directly modify the diagonal
    d_diag = d.view(-1)[::(self.q+1)]
    d_diag.fill_(inf)
    Z      = torch.div(self.y, d)
    # the below is length-q vector
    Z_sums = Z.sum(dim=1, keepdim=True)

    # Compute \theta^T (W - Z_n), where
    # W = 1 - diag(1,1,...1)
    weight_minus_Z = 1.0 - Z
    weight_minus_Z_diag = WmZ.view(-1)[::(self.q+1)]
    weight_minus_Z_diag.fill_(0)
    # # directly modify the diagonal
    # where the weight is zero
    TWmZ = torch.mm(self.theta.t(), weight_minus_Z)

    theta = (self.theta * (self.w_sums + Z_sums) +
                TWmZ.t())/(self.w_sums * 2.0)
```

The code below is a simple implementation of MDS in `DistStat.jl`.

```
W_sums = sum(W; dims=2)
for i in 1:iter
    mul!(theta_distances, transpose(theta), theta)
    diag!(d_dist, theta_distances)
    diag!(d_local, theta_distances)
    theta_distances .= -2theta_distances .+ d_dist .+
            d_local
    fill_diag!(theta_distances, Inf)
    Z .= y ./ theta_distances
    Z_sums .= sum(Z; dims=1) # Z sums, length m.
    WmZ .= W .- Z
    mul!(theta_WmZ, theta, WmZ)
    theta .= (theta .* (Z_sums .+ W_sums) .+ theta_WmZ)./
```

Table 3.11: Runtime (in seconds) of MDS on $10,000 \times 10,000$ simulated data on multiple GPUs

	dist_stat			DistStat.jl		
GPUs	$q = 20$	$q = 40$	$q = 60$	$q = 20$	$q = 40$	$q = 60$
1	292	301	307	402	415	423
2	146	151	154	267	275	279
3				210	212	216
4	81	84	88	89	93	97
5	74	78	80	77	83	85
6				64	70	72
7				58	64	69
8	52	58	64	53	60	65

```
            2W_sums
end
```

This code can also run for local array only with minor modifications involving the matrix diagonals.

For numerical experiments, a [10,000] $\times$ 10,000 and a [100,000] $\times$ 1,000 dataset was sampled from the standard normal distribution. For reference, the dataset used in Zhou et al. (2010) was 401×401. The pairwise Euclidean distances between data points were computed distributedly (Li et al., 2010): in each stage, data on one of the processors are broadcast and each processor computes pairwise distances between the data residing on its memory and the broadcast data. This is repeated until all the processors broadcast its data.

Table 3.11 compares the performance of `DistStat.jl` and `dist_stat` on multiple GPUs with $10,000 \times 10,000$ dataset. While `dist_stat` is faster with fewer GPUs employed, the gap between the two implementations vanishes dramatically as more GPUs are used.

For the AWS experiments, 36 processes per instance were used for `DistStat.jl`, because the step that mainly causes inter-instance communi-

Table 3.12: Runtime (in seconds) of MDS on $100,000 \times 1000$ simulated data on multiple AWS EC2 instances

	`dist_stat`			`DistStat.jl`		
Instances	$q = 20$	$q = 40$	$q = 60$	$q = 20$	$q = 40$	$q = 60$
4	2875	3097	3089	2093	2007	2188
5	2315	2378	2526	1625	1704	1746
8	1531	1580	1719	1073	1105	1215
10	1250	1344	1479	909	980	1022
16	821	914	1031	630	714	736
20	701	823	903	531	663	701

cation is the matrix-vector multiplication, and its communication cost is much less than NMF. Note that this setting is impossible with the `dist_stat` implementation, because 36 does not divide 100,000. For `dist_stat`, the job was run with two processes with 18 threads each per instance. Table 3.12 shows the runtime of each experiment for 1000 iterations on $100,000 \times 1000$ dataset. It can be seen that `DistStat.jl` implementation is significantly faster.

3.5.4 ℓ_1-regularized Cox regression

Finally, we apply the proximal gradient descent to ℓ_1-regularized Cox regression (Cox, 1972). In this problem, we are given a covariate matrix $X \in \mathbb{R}^{m \times p}$, and a possibly right-censored survival time $y = (y_1, \ldots, y_m)$ as data. Each element of y is defined by $y_i = \min\{t_i, c_i\}$, where t_i is time to event and c_i is right-censoring time for that sample. $\delta_i = I_{\{t_i \leq c_i\}}$ indicates if the sample i is censored or not. We put $\delta = (\delta_1, \ldots, \delta_m)^T$. The log partial likelihood of the Cox model is then

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

Coordinate descent-type approaches for this type of analyses are proposed by Suchard et al. (2013) and Mittal et al. (2014).

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

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

where we define $w_i = \exp(x_i^T\beta)$, $W_j = \sum_{i:y_i \geq y_j} w_i$, and the matrix $P = (\pi_{ij})$ whose elements are

$$\pi_{ij} = I(y_i \geq y_j)w_i/W_j.$$

Each row of P is normalized to sum to one. A Lipschitz constant of $\nabla L(\beta)$ can be found by finding an upper bound of $\|\nabla^2 L(\beta)\|_2$, where $\nabla^2 L(\beta)$ is 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 update rule is:

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

If the data are sorted in the nonincreasing order of y_i, W_j^n can be computed using the cumulative sum function. While this is not so obvious to implement

in a parallel environment, a CUDA device kernel function for this operation is readily provided with PyTorch. We can write a simple proximal gradient descent update for the Cox regression as:

```
# X: data matrix, m-by-p
# delta: censoring indicator, m-by-1
# y: right-censored survival time
# X is assumed to be sorted in decreasing order of y_i
# lambd: penalty parameter

soft_threshold = torch.nn.Softshrink(lambd)
L = 2 * power(X) ** 2
# power(X): power iteration to compute
# the spectral norm of X
sigma = 1/L

# mask: pi_ind[i, j] = (y[i] >= y[j])
pi_ind = (y - y.t() >= 0).to(dtype=tf.float64)

for i in range(max_iter):
    Xbeta = torch.mm(X, beta)
    w = torch.exp(Xbeta)
    W = w.cumsum(0)
    pi = (w / W.t()) * pi_ind
    grad = torch.mm(X.t(), delta - torch.mm(pi, delta))
    beta = soft_threshold(beta + grad * sigma)
```

assuming no ties in y_i's for simplicity. The soft-thresholding operator $\mathcal{S}_\lambda(x)$ is also implemented in PyTorch. We compute the full w_i/W_j first with `w / W.t()` then multiply it to the indicator $I(y_i \geq y_j)$ precomputed. A simple implementation of this algorithm in Julia, assuming no ties in y_i can be written by:

```
y_dist = distribute(reshape(y, 1, :))
fill!(pi_ind, one(T))
pi_ind .= ((pi_ind .* y_dist) .- y) .<= 0
for i in 1:iter
    mul!(Xbeta, X, beta)
```

```
    w .= exp.(Xbeta)
    cumsum!(W, w)
    W_dist .= distribute(reshape(W, 1, :))
    pi .= pi_ind .* w ./ W_dist
    mul!(pi_delta, pi, delta)
    dmpd .= delta .- pi_delta
    mul!(gradient, transpose(X), dmpd)
    beta .= soft_threshold.(beta .+ sigma .* gradient,
            lambda)
end
```

For simulation, the data matrix $X \in \mathbb{R}^{m \times [p]}$, distributed along the columns, is sampled from the standard normal distribution. The algorithm is designed to keep a copy of the estimand β in every device.

For performance optimization, note that in addition to the memory for X, an intermediate storage for two $m \times m$ matrices are needed. This can be avoided by environment-specific implementation. For example, the CPU function to compute $P_{(n+1)}\delta$ can be written using `LoopVectorization.jl` (Elrod, 2020) for efficient single instruction, multiple data parallelization using the Advanced Vector Extensions (AVX). These environment-specific implementations not only use less memory, but also result in some speedup. On the local node used, the device-specific CPU implementation with four processes with each process using a single core took almost half the time compared to the dot broadcasting-based implementation. The GPU implementation with four GPUs was 5-10% faster. Code for accelerating computation of $P_{(n+1)}\delta$ is avialable in Appendix D.

Table 3.13 demonstrates the scalability of the proximal gradient algorithm for ℓ_1-regularized Cox regression on multiple GPUs. While the `dist_stat` was faster with double precision arithmetics in many cases, the `DistStat.jl` implementation was faster in some cases. Unfortunately, the underlying algorithm for the `cumsum()` method in `PyTorch` is known to be numerically unstable, and it could not be used for very small values of λ. On the other hand, the

Table 3.13: Runtime (in seconds) of ℓ_1-regularized Cox regression on $10,000 \times [10,000]$ simulated data on multiple GPUs with $\lambda = 10^{-8}$.

GPUs	`dist_stat` (`Float64`)	`DistStat.jl` (`Float64`)	`DistStat.jl` (`Float32`)
1	382	447	292
2	205	196	113
3		160	91
4	115	136	80
5	98	121	75
6		113	71
7		106	69
8	124	86	67

Table 3.14: Runtime (in seconds) of ℓ_1-regularized Cox regression on $100,000 \times [200,000]$ simulated data on multiple AWS EC2 instances with $\lambda = 10^{-8}$.

Nodes	`dist_stat`	`DistStat.jl`
4	1455	918
5	1169	819
8	809	558
10	618	447
16	389	290
20	318	245

`cumsum()` function from `CuArrays.jl` is numerically stable for small values of λ. Using single-precision, the users can get the results more quickly.

For the AWS experiments on `DistStat.jl`, 36 processes per instance were used once again. Table 3.14 shows the runtime of the algorithm for 1000 iterations with a simulated $100,000 \times [200,000]$ dataset. Thanks to the flexibility of the Julia implementation, the speedup of `DistStat.jl` over `dist_stat` is obvious.

3.5.5 Genome-wide survival analysis of the UK Biobank dataset

Now, let us see real-world application of ℓ_1-regularized Cox regression to genome-wide survival analysis for Type 2 Diabetes (T2D). The UK Biobank dataset (Sudlow et al., 2015) was used, which 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. For the analysis with `dist_stat`, the information of 200,000 randomly sampled subjects including 8,995 T2D patients were used. 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. Breslow's method (Breslow, 1972) was applied for any tie in y_i. 63 different values of the regularization parameter λ in the range $[0.7 \times 10^{-9}, 1.6 \times 10^{-8}]$ were used, with which 0 to 111 SNPs were selected. For each value of λ, the ℓ_1-regularized Cox regression model of Section 3.5.4 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 are ranked based on the largest value of λ for which each 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 are located on TCF7L2, whose association with T2D is well-known (Scott et al., 2007; The Wellcome Trust Case Control Consortium, 2007). Also prominently selected are 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

Table 3.15: 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.

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. Then, an unpenalized Cox regression analysis using the 111 selected SNPs was conducted. The nine SNPs with the p-values less than 0.01 are listed in Table 3.15. The locations in Table 3.15 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.

With `DistStat.jl`, the entire dataset for this experiment was used, thanks to memory efficiency. 43 different values of λ in range $[6.0\times10^{-9}, 1.5\times10^{-8}]$ were used, where 0 to 320 SNPs were selected. For the analysis, 20 `c5.18xlarge`

instances were used. It took less than 2050 iteration until convergence, where convergence is determined by testing if $\frac{|f(\beta_{(n)})-f(\beta_{(n-10)})|}{|f(\beta_{(n)})+1|} < 10^{-5}$. For each λ, the experiment took 3180 to 3720 seconds.

The SNPs were ranked based on the largest of λ for which each SNP has nonzero coefficient, then breaking any tie based on the absolute values of the coefficients. The set of top nine selections is identical to that of the analysis with 200,000 subjects with `dist_stat` with slightly different order, as listed in Table 3.16. As before, significance test using unpenalized Cox regression with only selected SNPs, gender, and top 10 principal components is carried out. SNPs with p-values less than 0.01/333 were selected using Bonferroni correction to control family-wise error rate less than 0.01. Table 3.17 lists the 9 selected SNPs.

Table 3.16: Top nine SNPs selected by ℓ_1-penalized Cox regression

Rank	SNP ID	Chr	Location	A1[A]	A2[B]	MAF[C]	Mapped Gene	Sign
1	rs4506565	10	114756041	A	**T**	0.314	TCF7L2	+
2	rs16891982	5	33951693	G	**C**	0.073	SLC45A2	−
3	rs12243326	10	114788815	T	**C**	0.281	TCF7L2	+
4	rs12255372	10	1148088902	G	**T**	0.285	TCF7L2	+
5	rs28777	5	33958959	A	**C**	0.062	SLC45A2	−
6	rs35397	5	33951116	T	**G**	0.096	SLC45A2	−
7	rs1129038	15	28356859	T	**C**	0.261	HERC2	−
8	rs12913832	15	28365618	G	**A**	0.259	HERC2	−
9	rs10787472	10	114781297	A	**C**	0.470	TCF7L2	+

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

Six of the SNPs, including the SNPs with five lowest p-values are previously reported to have direct association with T2D (rs1801212 from WFS1 (Fawcett et al., 2010), rs4506565 from TCF7L2 (The Wellcome Trust Case Control Consortium, 2007; Dupuis et al., 2010), rs2943640 from IRS1 (Langenberg et al., 2014), rs10830962 from MTNR1B (Klimentidis et al., 2014; Salman et al., 2015),

rs343092 from HMGA2 (Ng et al., 2014), and rs231362 from KCNQ1 (Riobello et al., 2016)). In addition, rs1351394 is from HMGA2, known to be associated with T2D. This seems to be an improvement over the `dist_stat` result in which three of the top nine selections were found to be directly associated with T2D and three others were on the known T2D-associated genes.

Table 3.17: SNPs with significant coefficients with significance level 0.01 after Bonferroni correction

SNP ID	Chr	Location	A1[A]	A2[B]	MAF[C]	Mapped Gene	Coefficient	p-value
rs1801212	4	6302519	**A**	G	0.270	WFS1	0.1123	<2E-16
rs4506565	10	114756041	A	**T**	0.314	TCF7L2	0.2665	<2E-16
rs2943640	2	227093585	**C**	A	0.336	IRS1	0.0891	1.57E-14
rs10830962	11	92698427	C	**G**	0.402	MTNR1B	0.0731	1.46E-11
rs343092	12	66250940	G	**T**	0.166	HMGA2	-0.0746	2.26E-07
rs1351394	12	66351826	**C**	T	0.478	HMGA2	0.0518	1.70E-06
rs2540917	2	60608759	**T**	C	0.389	RNU1-32P	-0.0476	2.18E-05
rs1254207	1	236368227	C	**T**	0.395	GPR137B	0.0458	2.84E-05
rs231362	11	2691471	**G**	A	0.461	KCNQ1	0.0607	2.87E-05

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

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. The mapped genes harboring the SNPs selected by the half-million-variate regression analysis include CPLX3 and CACNA1A associated with regulation of insulin secretion, and SEMA7A and HLA-DRA involved with inflammatory responses (based on DAVID (Huang et al., 2009a,b)). These genes might have been missed in conventional univariate analysis of T2D due to moderate statistical significance values. Joint GWAS may overcome such a limitation, and be possible by combining the computing power of modern HPC and scalable algorithms.

3.6 Discussion

Packages `dist_stat` and `DistStat.jl` provide first steps to provide a unified development environment for multiple nodes with multiple GPUs. The packages supply distributed array data structure based on any type of underlying array. In particular, `DistStat.jl` can be used with any array type on any hardware provided that the array interface is implemented in Julia with `MPI` support.

Statistical applications including NMF, MDS, PET, and ℓ_1-regularized Cox regularization are considered, and scalability is shown on a 8-GPU workstation and a virtual cluster on AWS cloud with up to 20 instances. Performance of `DistStat.jl` was equivalent to or better than its `dist_stat` counterpart. With the newly-developed packages, the biological dataset of size $400,000 \times 500,000$ could be analyzed.

Listing 3.3 Monte Carlo estimation of π for TensorFlow on multiple nodes using Horovod

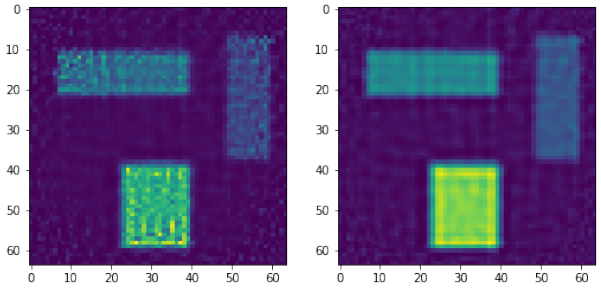
```
import tensorflow as tf
import horovod.tensorflow as hvd
# initialize horovod
hvd.init()
rank = hvd.rank()
# without this block, all the processes try to allocate
# all the memory from each device, causing out of memory
# error.
devices = tf.config.experimental.list_physical_devices(
    "GPU")
if 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 3.4 An example usage of the module `distmat`.

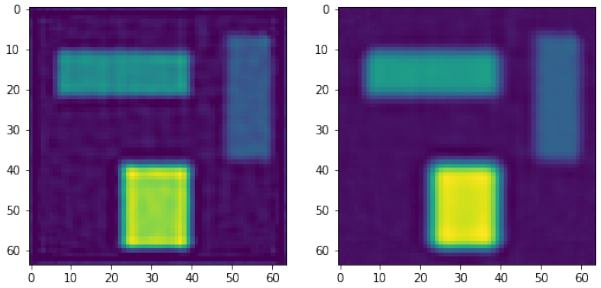
```
import torch, distmat
import torch.distributed as dist
dist.init_process_group('mpi')
rank = dist.get_rank()
size = dist.get_world_size()

device = 'cuda:{}'.format(rank)
# or simply 'cpu' for CPU computing
if device.startswith('cuda'): torch.cuda.set_device(rank)

tensortype = torch.DoubleTensor
# torch.cuda.FloatTensor for
# a single-precision matrix on a GPU
A = distmat.distgen_uniform(4, 4, TType=tensortype)
B = distmat.distgen_uniform(4, 2, TType=tensortype)
AB = distmat.mm(A, B) # A * B
if rank == 0: # to print this only once
    print("AB = ")
print(rank, AB.chunk) # print the rank's protion of AB.
C = (1 + AB).log() # elementwise logarithm
if rank == 0:
    print("log(1 + AB) = ")
print(rank, C.chunk) # print the rank's portition of C.
```

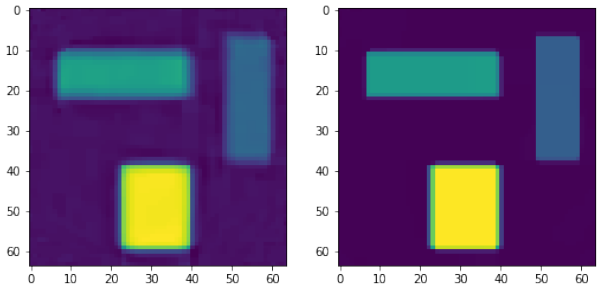


(a) $\mu = 0$ (b) $\mu = 10^{-7}$

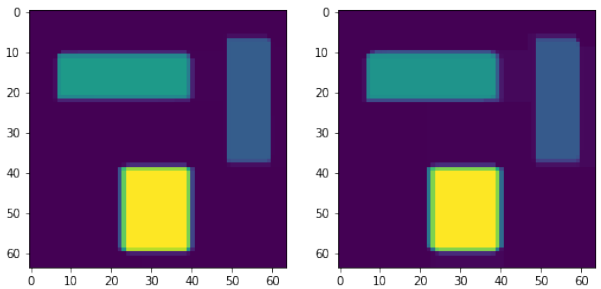


(c) $\mu = 10^{-6}$ (d) $\mu = 10^{-5}$

Figure 3.2: Reconstructed images of the RVF phantom with a ridge penalty.

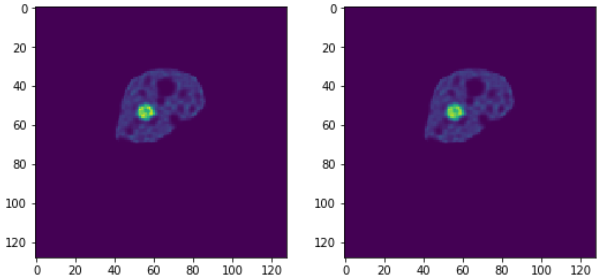


(a) $\rho = 2^{-10}$ (b) $\rho = 2^{-8}$

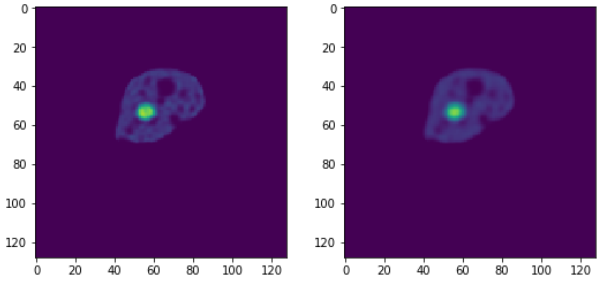


(c) $\rho = 2^{-6}$ (d) $\rho = 2^{-4}$

Figure 3.3: Reconstructed images of the RVF phantom with a TV penalty.

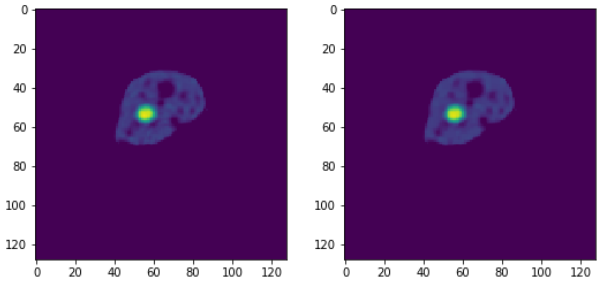


(a) $\mu = 0$ (b) $\mu = 10^{-6}$

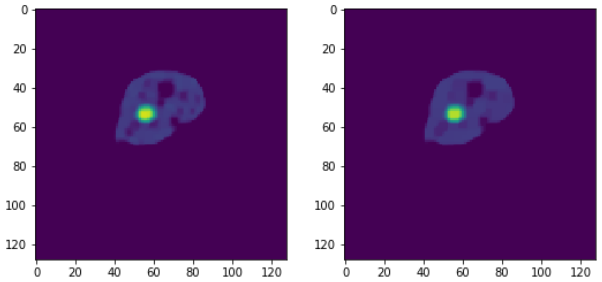


(c) $\mu = 10^{-5}$ (d) $\mu = 10^{-4}$

Figure 3.4: Reconstructed images of the XCAT phantom with a ridge penalty.



(a) $\rho = 0$ (b) $\rho = 10^{-3}$



(c) $\rho = 10^{-2.5}$ (d) $\rho = 10^{-2}$

Figure 3.5: Reconstructed images of the XCAT phantom with a TV penalty.

Chapter 4

Conclusion

In this dissertation, highly parallelizable algorithms for statistical computing are reviewed, and a class among them, namely, a variant of primal-dual hybrid gradient (PDHG) for three-function sum is accelerated to its asymptotic optimum. Then, easy-to-use software packages to implement various statistical algorithms, including the former, are developed.

Abstraction of highly complex computing operations have rapidly evolved over the last decade. In this dissertation, how statisticians can benefit from this evolution is explored. It is also shown that many useful tools to incorporate computing clusters and accelerators have been created outside of the statistical community. Unfortunately, such developments have been mainly made in languages other than R, particularly in Python and Julia, with which statisticians might not be familiar. Although there are libraries that deal with simple parallel computation in R, common issues with these libraries are that it is difficult for them to incorporate GPUs which might significantly speed up the computation and that 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 and Julia. Fortunately, these languages are not hard to learn. 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 R community writing a good wrapper for the important Python and libraries. A good starting point may be a Python or Julia interface to R. The R package `reticulate` (Ushey et al., 2019) and `JuliaCall` (Li, 2019) might be good candidates. For example, there is an interface to `TensorFlow` based on `reticulate` (RStudio, 2019).

The methods discussed in this dissertation can be applied efficiently even when the dataset is larger than several gigabytes by using multiple CPU machines or using multiple GPUs. The advantages of using multiple CPU machines and multiple GPUs are two-fold. First, we can take advantage of data parallelism with more computing cores, accelerating the computation. Second, we can push the upper 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.

A major weakness of the current 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. With multi-CPU machines, it can be covered by a high-speed interconnection technology such as InfiniBand. Even when such kind of environment is not affordable, we may still use relatively high-speed connection equipped with instances from a cloud. The network bandwidth of 25 Gbps supported for `c5.18xlarge` instances of AWS was quite effective in our experiments. Another way to alleviate the communication issue is employing

communication-avoiding algorithms (Van De Geijn and Watts, 1997; Ballard et al., 2011; Koanantakool et al., 2016) to minimize the amount of communication between computing units. This approach has been utilized for statistical inference (Jordan et al., 2019) and sparse inverse covariance estimation (Koanantakool et al., 2018).

Loss of accuracy due to the single-precision of the GPU, prominent in our Cox regression example, can be solved by purchasing scientifically-oriented GPUs with better double-precision supports, which costs money. Another option is to go to clouds: for example, `P2` and `P3` instances in AWS support scientific GPUs. Nevertheless, even with that double-precision floating-point operation speed is 1/32 compared to single-precision, desktop GPUs with double-precision could achieve more than 10-fold speedup over CPU.

Appendix A

Monotone Operator Theory

Here we briefly state necessary results from monotone operator theory for the proofs in the subsequent section. For more details, see Bauschke and Combettes (2011).

Set-valued operators. A set-valued operator $T : \mathbb{R}^n \to 2^{\mathbb{R}^n}$ maps a vector $z \in \mathbb{R}^n$ to a set $T(z) \subset \mathbb{R}^n$. The graph of T is denoted by $\mathbf{gra}\, T = \{(z, w) \in \mathbb{R}^n \times \mathbb{R}^n : w \in T(z)\}$. When $T(z)$ is single-valued, i.e., $T(z) = \{w\}$, T is a function, and we write simply as $T(z) = w$. We use I to denote the identity operator, i.e, $I(z) = z$. When no confusion incurs, we also use Tz to mean $T(z)$. In particular, when T is a single-valued linear operator, Tz is identified with a multiplication of the corresponding matrix $T \in \mathbb{R}^{n\times n}$ by a vector z. The set of zeros of T is defined as $\mathbf{zer}\, T = \{z \in \mathbb{R}^n : 0 \in Tz\}$. The inverse of T is $T^{-1} : \mathbb{R}^n \to 2^{\mathbb{R}^n}$ such that $T^{-1}(w) = \{z \in \mathbb{R}^n : w \in Tz\}$, hence $\mathbf{gra}\, T^{-1} = \{(w, z) \in \mathbb{R}^n \times \mathbb{R}^n : w \in Tz\}$. The resolvent of T is $R_T = (I + T)^{-1}$. Scaling of an operator T by $t \in \mathbb{R}$ is defined by $(tT)(z) = tT(z)$. Composition

of two set-valued operators $T_1 : \mathbb{R}^n \to 2^{\mathbb{R}^n}$ and $T_2 : \mathbb{R}^n \to 2^{\mathbb{R}^n}$ is defined by $T_2T_1z = \bigcup_{w \in T_1 z} T_2 w$.

Fixed points. An operator $T : \mathbb{R}^n \to 2^{\mathbb{R}^n}$ is called nonexpansive if $\|u - u'\|_2 \leq \|z - z'\|_2$ for all $u \in T(z), u' \in T(z') \in \mathbb{R}^n$; it is called contractive if the inequality is strict. Any nonexpansive operator is single-valued. The set of fixed points of a single-valued operator T is denoted by $\mathbf{Fix}\, T$, i.e., $\mathbf{Fix}\, T = \{z : z = Tz\}$. For a contractive operator T, the fixed point iteration $z^{k+1} = Tz^k$ converges to a point in $\mathbf{Fix}\, T$, if $\mathbf{Fix}\, T \neq \emptyset$.

Averaged operators. An operator T is called α-averaged, $0 < \alpha < 1$, if $T = (1 - \alpha)I + \alpha R$ for some nonexpansive operator R. Usually R is defined implicitly. Note that T itself is nonexpansive, and $\mathbf{Fix}\, T = \mathbf{Fix}\, R$. If T_1 is α_1-averaged and T_2 is α_2-averaged, then T_1T_2 is α-averaged where $\alpha = (\alpha_1 + \alpha_2 - 2\alpha_1\alpha_2)/(1 - \alpha_1\alpha_2)$. An α-averaged operator T is nonexpansive but not necessarily contractive, hence the fixed point iteration $z^{k+1} = Tz^k$ above may not converge to a fixed point even if $\mathbf{Fix}\, T \neq \emptyset$. In this case, the Krasnosel'skiĭ-Mann (KM) iteration $z^{k+1} = z^k + \rho_k(Tz^k - z^k)$ with a sequence $\{\rho_k\} \subset (0, 1/\alpha]$ such that $\sum_{k=0}^{\infty} \rho_k(1 - \alpha\rho_k) = \infty$ ensures convergence.

Monotone operators. An operator T is called monotone if $\langle z - z', w - w' \rangle \geq 0$ for all $z, z' \in \mathbb{R}^n$ and for all $w \in Tz$, $w' \in Tz'$, and maximally monotone if it is monotone and there is no monotone operator T' such that $T \neq T'$ and $\mathbf{gra}\, T \subset \mathbf{gra}\, T'$. The resolvent of a maximally monotone operator is single-valued; it is 1/2-averaged.

Cocoercive operators. A single-valued operator T is called γ-cocoercive if for some $\gamma > 0$, $\langle z - z', Tz - Tz' \rangle \geq \gamma \|Tz - Tz'\|_2^2$. A cocoercive operator

is maximally monotone. If an operator T is γ-cocoercive with $\gamma > 1/2$, then $I - tT$ ($t > 0$) is $t/(2\gamma)$-averaged. A convex, closed, and proper function ϕ has L-Lipschitz continous gradient $\nabla\phi$ if and only if $\nabla\phi$ is $1/L$-cocoercive.

Subdifferential. An important example of a maximally monotone operator is the subdifferential of a convex closed proper function. A vector $g \in \mathbb{R}^n$ is a subgradient of a convex function ϕ at z if $\phi(z') \geq \phi(z) + \langle g, z' - z\rangle,\ \forall z' \in \mathbb{R}^n$. The subdifferential of ϕ at z is the set of subgradients at z: $\partial\phi(z) = \{g \in \mathbb{R}^n : \phi(z') \geq \phi(z) + \langle g, z' - z\rangle,\ \forall z' \in \mathbb{R}^n\}$. When ϕ is differentiable, $\partial\phi(z) = \{\nabla\phi(z)\}$. If ϕ is in addition closed and proper, $(\partial\phi)^{-1} = \partial\phi^*$ holds, where ϕ^* is convex conjugate defined by $\phi^*(w) = \sup_{z\in\mathbb{R}^n}\{\langle z, w\rangle - \phi(z)\}$. The resolvent of a maximally monotone subdifferential operator is the proximity operator: $R_{\partial\phi} = (I + \partial\phi)^{-1}(z) = \mathbf{prox}_\phi(z) = \arg\min_{z'\in\mathbb{R}^n} \phi(z') + \frac{1}{2}\|z' - z\|_2^2$.

Skew-symmetric operators. Another example of a maximally monotone operator is a skew-symmetric matrix. The sum of a maximally monotone operator and a skew-symmetric matrix is also maximally monotone.

Change of metric. Note that the notion of nonexpansiveness, averagedness, cocoercivity, and monotonicity of an operator requires the inner product $\langle\cdot,\cdot\rangle$ and its associated norm $\|\cdot\|_2$. We can appropriately define these concepts with respect to another inner product and its associated norm as well, say $\langle\cdot,\cdot\rangle_M$ and $\|\cdot\|_M$, for M a symmetric, positive definite matrix. In particular, averagedness of composition, convergence of the KM iteration, and averagedness of $I - tT$ for cocoercive T hold by substituting the inner products and norms by $\langle\cdot,\cdot\rangle_M$ and $\|\cdot\|_M$, respectively.

Forward-backward splitting. Some optimization problems can be translated to finding an element of $\mathbf{zer}\, T$ for an appropriate choice of maximally monotone operator T. Often T can be split into a sum of two maximally monotone operators F and G. If G is γ-cocoercive (hence single-valued), then we see

$$\begin{aligned} 0 \in T(z) &\iff (I + tF)(z) \ni (I - tG)(z) \\ &\iff z = R_{tF}(I - tG)(z), \end{aligned} \tag{A.1}$$

for $t > 0$. Equivalence (A.1) shows that $\mathbf{zer}\,(F + G) = \mathbf{Fix}(R_{tF}(I - tG))$, thus we may solve the problem of finding a zero of T by the following fixed-point iteration

$$z^{k+1} = (1 - \rho_k)z^k + \rho_k R_{tF}(I - tG)(z^k). \tag{A.2}$$

This iteration is a KM iteration because $R_{tF}(I - tG)$ is a $1/\delta$-averaged operator, where $\delta = 2 - t/(2\gamma)$. Thus (A.2) converges for $t \in (0, 2\gamma)$ if $\mathbf{zer}(F + G) \neq \emptyset$ and under the aforementioned condition for $\{\rho_k\}$. Furthermore, the following hold (Bauschke and Combettes, 2011, proof of Theorems 25.8):

$$\|z^{k+1} - z\|_2^2 \leq \|z^k - z\|_2^2, \quad \forall z \in \mathbf{zer}(F + G); \tag{A.3a}$$

$$\textstyle\sum_{k=0}^{\infty} \frac{\delta - \rho_k}{\rho_k} \|z^{k+1} - z^k\|_2^2 \leq \|z^0 - z\|_2^2, \quad \forall z \in \mathbf{zer}(F + G); \tag{A.3b}$$

$$\|z^{k+1} - z^k\|_2 \to 0. \tag{A.3c}$$

Preconditioning. In the forward-backward splitting above, observe that the identity matrices in the first line can be replaced by an invertible matrix M, yielding a *preconditioned* forward-backward splitting algorithm

$$z^{k+1} = (1 - \rho_k)z^k + \rho_k R_{tM^{-1}F}(I - tM^{-1}G)(z^k). \tag{A.4}$$

Preconditioning is useful when evaluating the resolvent $R_{tM^{-1}F}$ is easier than R_{tF}. It can be shown that if M is symmetric positive definite, $M^{-1}F$ is maximally monotone with respect to $\langle \cdot, \cdot \rangle_M$ (Combettes and Vũ, 2014), and $M^{-1}G$ is $\gamma\lambda_{\min}(M)$-cocoercive with respect to $\| \cdot \|_M$ (Davis, 2015). Therefore we can replace $\| \cdot \|_2$ by $\| \cdot \|_M$, and γ by $\gamma\lambda_{\min}(M)$ in (A.3).

Appendix B

Proofs for Chapter II

B.1 Preconditioned forward-backward splitting

Proof of Lemma 1. Observe that

$$\begin{aligned}\|z^- - z\|_M^2 &= \|z^- - z_\rho + z_\rho - z\|_M^2 \\ &= \|z^- - z_\rho\|_M^2 - 2\langle z^- - z_\rho, z - z_\rho\rangle_M + \|z_\rho - z\|_M^2, \end{aligned} \tag{B.1}$$

and, from (A.1),

$$\begin{aligned} & z^+ + M^{-1}Fz^+ \ni z^- - M^{-1}Gz^- \\ & \iff z^+ + M^{-1}\begin{bmatrix} 0 & K^T \\ -K & \partial h^* \end{bmatrix}\begin{bmatrix} x^+ \\ y^+ \end{bmatrix} \ni z^- - M^{-1}\begin{bmatrix} \nabla f & \\ & 0 \end{bmatrix}\begin{bmatrix} x^- \\ y^- \end{bmatrix} \\ & \iff (1/\rho)(z^- - z_\rho) = z^- - z^+ \in M^{-1}\begin{bmatrix} \nabla f(x^-) + K^T y^+ \\ -Kx^+ + \partial h^*(y^+) \end{bmatrix} \end{aligned} \tag{B.2}$$

Then,

$$\begin{aligned}\langle z^- - z_\rho, z - z_\rho\rangle_M &= \langle \rho(z^- - z^+), z - z_\rho\rangle_M \\ &= \rho\left\langle \begin{bmatrix} \nabla f(x^-) + K^T y^+ \\ -Kx^+ + \partial h^*(y^+) \end{bmatrix}, \begin{bmatrix} x - x_\rho \\ y - y_\rho \end{bmatrix}\right\rangle \\ &= \rho\langle \nabla f(x^-), x - x_\rho\rangle + \rho\langle K^T y^+, x - x_\rho\rangle \\ &\quad + \rho\langle -Kx^+, y - y_\rho\rangle + \rho\langle \partial h^*(y^+), y - y_\rho\rangle \end{aligned} \tag{B.3}$$

$$
\begin{aligned}
&= \rho\langle \nabla f(x^-), x^- - x_\rho\rangle + \rho\langle \nabla f(x^-), x - x^-\rangle && \text{(B.4)}\\
&\quad + \rho\langle K^T y^+, x - x_\rho\rangle + \rho\langle -Kx^+, y - y_\rho\rangle \\
&\quad + \rho\langle \partial h^*(y^+), y^+ - y_\rho\rangle + \rho\langle \partial h^*(y^+), y - y^+\rangle \\
&\leq \rho\langle \nabla f(x^-), x^- - x_\rho\rangle + \rho(f(x) - f(x^-)) \\
&\quad + \rho\langle K^T y^+, x - x_\rho\rangle + \rho\langle -Kx^+, y - y_\rho\rangle \\
&\quad + \rho\langle \partial h^*(y^+), y^+ - y_\rho\rangle + \rho(h^*(y) - h^*(y^+)), && \text{(B.5)}
\end{aligned}
$$

understanding that "$\partial h^*(\cdot)$" represents a subgradient in the corresponding subdifferential. The first and second equalities follow from (B.2); the last inequality is due to the definition of subgradient. By plugging the inequality (B.3) in (B.1) and rearranging terms, we obtain

$$
\begin{aligned}
&2\rho(\mathcal{L}(x^+, y) - \mathcal{L}(x, y^+)) - \|z^- - z\|_M^2 + \|z_\rho - z\|_M^2 \\
&\qquad \leq -\|z^- - z_\rho\|_M^2 + 2\rho\langle \nabla f(x^-), x^- - x_\rho\rangle + 2\rho\langle \partial h^*(y^+), y^+ - y_\rho\rangle \\
&\qquad\quad - 2\rho\langle K^T y^+, x_\rho\rangle + 2\rho\langle Kx^+, y_\rho\rangle + 2\rho(f(x^+) - f(x^-)) && \text{(B.6)}
\end{aligned}
$$

Now it suffices to show that the right-hand side of (B.6) is less than or equal to $(1 - 2/\rho)\|z^- - z_\rho\|_M^2 + (L_f/\rho)\|x^- - x_\rho\|_2^2$. To see this,

$$
\begin{aligned}
\text{(RHS)} &= -\|z^- - z_\rho\|_M^2 + 2\rho\langle \nabla f(x^-), x^- - x_\rho\rangle + 2\rho\langle \partial h^*(y^+), y^+ - y_\rho\rangle \\
&\quad - 2\rho\langle K^T y^+, x_\rho - x^+\rangle - 2\rho\langle K^T y^+, x^+\rangle + 2\rho\langle Kx^+, y_\rho - y^+\rangle \\
&\quad + 2\rho\langle Kx^+, y^+\rangle + 2\rho(f(x^+) - f(x^-)) \\
&= -\|z^- - z_\rho\|_M^2 + 2\rho(f(x^+) - f(x^-) - \langle \nabla f(x^-), x^+ - x^-\rangle) \\
&\quad + 2\rho\langle \nabla f(x^-) + K^T y^+, x^+ - x_\rho\rangle + 2\rho\langle -Kx^+ + \partial h^*(y^+), y^+ - y_\rho\rangle \\
&= -\|z^- - z_\rho\|_M^2 + 2\rho(f(x^+) - f(x^-) - \langle \nabla f(x^-), x^+ - x^-\rangle) \\
&\quad + 2\rho\langle M(z^- - z^+), z^+ - z_\rho\rangle \\
&= -\|z^- - z_\rho\|_M^2 + 2\rho(f(x^+) - f(x^-) - \langle \nabla f(x^-), x^+ - x^-\rangle) \\
&\quad + 2\langle z^- - z_\rho, z^+ - z_\rho\rangle_M \\
&= -\|z^- - z_\rho\|_M^2 + 2\rho(f(x^+) - f(x^-) - \langle \nabla f(x^-), x^+ - x^-\rangle) \\
&\quad + 2(1 - 1/\rho)\langle z^- - z_\rho, z^- - z_\rho\rangle_M \\
&\leq (1 - 2/\rho)\|z^- - z_\rho\|_M^2 + \rho L_f\|x^- - x^+\|_2^2 \\
&= (1 - 2/\rho)\|z^- - z_\rho\|_M^2 + (L_f/\rho)\|x^- - x_\rho\|_2^2
\end{aligned}
$$

where the third equality follows from (B.2); the fourth and fifth equalities are from (A.1); the first inequality is due to the Lipschitz continuity of ∇f; the

final equality is again from (A.1). □

We need the following fact to prove Theorem 1.

Proposition 5. *Let M be a symmetric, positive definite matrix in $\mathbb{R}^{(p+l)\times(p+l)}$ and G as given in* (2.11)*. Then, for $\mu > 0$ such that*

$$\|(x,0)\|_{M^{-1}}^2 \le (1/\mu)\|x\|_2^2, \quad \forall x \in \mathbb{R}^p, \tag{B.7}$$

operator $M^{-1}G$ is μ/L_f-cocoercive in $\langle\cdot,\cdot\rangle_M$.

Proof.

$$\begin{aligned}
\|M^{-1}Gz - M^{-1}Gz'\|_M^2 &= \|Gz - Gz'\|_{M^{-1}}^2 \\
&= \|(\nabla f(x) - \nabla f(x'), 0)\|_{M^{-1}}^2 \\
&\le (1/\mu)\|\nabla f(x) - \nabla f(x')\|_2^2 \\
&\le (L_f/\mu)\langle \nabla f(x) - \nabla f(x'), x - x'\rangle \\
&= (L_f/\mu)\langle Gz - Gz', z - z'\rangle \\
&= (L_f/\mu)\langle M^{-1}Gz - M^{-1}Gz', z - z'\rangle_M.
\end{aligned}$$

Note that we used $1/L_f$-cocoercivity of ∇f in the third line. □

Proof of Proposition 1. Note that $\|L^T \cdot\|_{M_{\mathsf{LV}}}^2 = \langle M_{\mathsf{LV}} L^T \cdot, L^T \cdot\rangle = \|\cdot\|_{M_{\mathsf{CV}}}^2$ and likewise $\langle L^T\cdot, L^T\cdot\rangle_{M_{\mathsf{LV}}} = \langle\cdot,\cdot\rangle_{M_{\mathsf{CV}}}$. Then,

$$\begin{aligned}
&\langle M_{\mathsf{LV}}^{-1}L^{-1}GL^{-T}w - M_{\mathsf{LV}}^{-1}L^{-1}GL^{-T}w', w - w'\rangle_{M_{\mathsf{LV}}} \\
&\quad= \langle (L^T M_{\mathsf{CV}}^{-1}L)(L^{-1}GL^{-T})(L^T z) \\
&\qquad - (L^T M_{\mathsf{CV}}^{-1}L)(L^{-1}GL^{-T})(L^T z'), L^T z - L^T z'\rangle_{M_{\mathsf{LV}}} \\
&\quad= \langle L^T(M_{\mathsf{CV}}^{-1}Gz - M_{\mathsf{CV}}^{-1}Gz'), L^T(z - z')\rangle_{M_{\mathsf{LV}}} \\
&\quad= \langle M_{\mathsf{CV}}^{-1}Gz - M_{\mathsf{CV}}^{-1}Gz', z - z'\rangle_{M_{\mathsf{CV}}} \\
&\quad\ge (\mu/L_f)\|M_{\mathsf{CV}}^{-1}Gz - M_{\mathsf{CV}}^{-1}Gz'\|_{M_{\mathsf{CV}}} \\
&\quad= (\mu/L_f)\|L^T(L^{-T}M_{\mathsf{LV}}^{-1}L^{-1}GL^{-T}(L^T z) - L^{-T}M_{\mathsf{LV}}^{-1}L^{-1}GL^{-T}(L^T z')\|_{M_{\mathsf{LV}}}^2 \\
&\quad= (\mu/L_f)\|M_{\mathsf{LV}}^{-1}L^{-1}GL^{-T}w - M_{\mathsf{LV}}^{-1}L^{-1}GL^{-T}w'\|_{M_{\mathsf{LV}}}^2,
\end{aligned}$$

where the inequality and μ come from Proposition 5, as follows. From (2.13), we see that

$$M_{\mathsf{CV}}^{-1} = L^{-T}M_{\mathsf{LV}}^{-1}L^{-1} = \begin{bmatrix} I & \tau K \\ & I \end{bmatrix} \begin{bmatrix} \tau I & \\ & (\frac{1}{\sigma}I - \tau K K^T)^{-1} \end{bmatrix} \begin{bmatrix} I & 0 \\ \tau K & I \end{bmatrix}$$

$$= \begin{bmatrix} \tau I + \tau^2 K^T (\frac{1}{\sigma} I - \tau K K^T)^{-1} K & \tau K^T (\frac{1}{\sigma} I - \tau K K^T)^{-1} \\ \tau (\frac{1}{\sigma} I - \tau K K^T)^{-1} K & (\frac{1}{\sigma} I - \tau K K^T)^{-1} \end{bmatrix}$$

and can choose $\mu = 1/\tau - \sigma\|K\|_2^2$, because $\lambda_{\max}(\tau I + \tau^2 K^T (\frac{1}{\sigma} I - \tau K K^T)^{-1} K) = \tau + \tau^2 \|K\|_2^2 / (1/\sigma - \tau \|K\|_2^2) = \frac{1}{1/\tau - \sigma\|K\|_2^2}$. Therefore $M_{\mathsf{LV}}^{-1} L^{-1} G L^{-T}$ is $(1/\tau - \sigma\|K\|_2^2)/L_f$-cocoercive with respect to $\|\cdot\|_{M_{\mathsf{LV}}}$. □

Proof of Proposition 2. For M given by (2.17), its inverse is given by

$$\begin{aligned} M^{-1} = \tilde{L}^{-T} M_{\mathsf{LV}} \tilde{L}^{-1} &= \begin{bmatrix} I & -\tau C \\ & I \end{bmatrix} \begin{bmatrix} \tau I & \\ & (\frac{1}{\sigma} I - \tau K K^T)^{-1} \end{bmatrix} \begin{bmatrix} I & \\ -\tau C & I \end{bmatrix} \\ &= \begin{bmatrix} \tau I + \tau^2 C^T (\frac{1}{\sigma} I - \tau K K^T)^{-1} C & -\tau C^T (\frac{1}{\sigma} I - \tau K K^T)^{-1} \\ -\tau (\frac{1}{\sigma} I - \tau K K^T)^{-1} C & (\frac{1}{\sigma} I - \tau K K^T)^{-1} \end{bmatrix}. \end{aligned}$$

Since $\lambda_{\max}(\tau I + \tau^2 C^T (\frac{1}{\sigma} I - \tau K K^T)^{-1} C) = \tau + \tau^2 \|C\|_2^2 / (1/\sigma - \tau \|K\|_2^2)$, we see that (B.7) holds with $\mu = \big(\tau + \tau^2 \|C\|_2^2 / (1/\sigma - \tau\|K\|_2^2)\big)^{-1} = \frac{1/\tau - \sigma\|K\|_2^2}{1 - \sigma\tau(\|K\|_2^2 - \|C\|_2^2)}$. This shows that the operator $M^{-1}G$ is $\frac{1/\tau - \sigma\|K\|_2^2}{L_f(1 - \sigma\tau(\|K\|_2^2 - \|C\|_2^2))}$-cocoercive with respect to $\langle\cdot,\cdot\rangle_M$. Hence, Algorithm (2.18) meets the condition for (A.4) with $t = 1$ if with $\gamma = \frac{1/\tau - \sigma\|K\|_2^2}{L_f(1 - \sigma\tau(\|K\|_2^2 - \|C\|_2^2))} > 1/2$. Required positive definiteness of M implies $\frac{1}{\tau\sigma} > \|K\|_2^2$. Thus the result (2.20) follows. □

Proof of Theorem 1. From the convexity-concavity of $\mathcal{L}(x, y)$, we have

$$\begin{aligned} \mathcal{L}(\bar{x}^N, y) - \mathcal{L}(x, \bar{y}^N) &\le \frac{1}{\sum_{k=0}^N \rho_k} \sum_{k=0}^N \rho_k (\mathcal{L}(\tilde{x}^k, y) - \mathcal{L}(x, \tilde{y}^k)) \\ &\le \frac{1}{2\sum_{k=0}^N \rho_k} \left(\|z^0 - z\|_M^2 + \sum_{k=0}^N \frac{L_f}{\rho_k} \|x^{k+1} - x^k\|_2^2 \right), \end{aligned}$$

where the second inequality comes from Lemma 1 by putting $z^- = z^k$, $z^+ = \tilde{z}^k$, $\rho = \rho_k$, $z_\rho = z^{k+1}$, and noting that $1 < 1/\alpha < 2$ by the assumption $\mu > L_f/2$. Now by Proposition 5 we see that $R_{M^{-1}F}(I - M^{-1}G)$ is α-averaged with respect to $\|\cdot\|_M$, thus by (A.3b) we have

$$\begin{aligned} \frac{1 - \alpha\bar{\rho}}{\alpha} \sum_{k=0}^\infty \frac{1}{\rho_k} \|x^{k+1} - x^k\|_2^2 &\le \sum_{k=0}^\infty \frac{1 - \alpha\rho_k}{\alpha\rho_k} \|x^{k+1} - x^k\|_2^2 \\ &\le \sum_{k=0}^\infty \frac{1 - \alpha\rho_k}{\alpha\rho_k} \|z^{k+1} - z^k\|_2^2 \le \frac{1}{\lambda_{\min}(M)} \|z^0 - z^\star\|_M^2. \end{aligned}$$

Therefore

$$\mathcal{L}(\bar{x}^N, y) - \mathcal{L}(x, \bar{y}^N) \leq \frac{1}{2\sum_{k=0}^N \rho_k} \left(\|z^0 - z\|_M^2 + \sum_{k=0}^N \frac{L_f}{\rho_k} \|x^{k+1} - x^k\|_2^2 \right)$$
$$\leq \frac{1}{2\sum_{k=0}^N \rho_k} \left(\|z^0 - z\|_M^2 + \frac{\alpha L_f}{(1-\alpha\bar{\rho})\lambda_{\min}(M)} \|z^0 - z^\star\|_M^2 \right).$$

□

Proof of Corollary 1. The proof closely follows that of Loris and Verhoeven (2011, Theorem 1), given for f being quadratic. Because $z^k = (x^k, y^k) \to (x^\star, y^\star) = z^\star \in \mathbf{Fix}\, T$ where $T = R_{M^{-1}F}(I - M^{-1}G)$, we see $\tilde{z}^k = Tz^k \to z^\star$ and thus $\bar{z}^N = (\bar{x}^N, \bar{y}^N) = (\sum_{k=1}^N \rho_k \tilde{z}^k)/(\sum_{k=1}^N \rho_k) \to z^\star$. Also because $(x^\star, y^\star)$ is a saddle-point of $\mathcal{L}(x, y)$, we have $\mathcal{F}^\star = \mathcal{F}(x^\star) = \mathcal{L}(x^\star, y^\star) \geq \mathcal{L}(x^\star, y)$ for all $y \in \mathbb{R}^l$. Then

$$0 \leq \mathcal{F}(\bar{x}^N) - \mathcal{F}^\star = \mathcal{F}(\bar{x}^N) - \mathcal{L}(x^\star, y^\star) \leq \mathcal{F}(\bar{x}^N) - \mathcal{F}(x^\star, \bar{y}^N)$$
$$= \sup_{y \in \mathbb{R}^l} \mathcal{L}(\bar{x}^N, y) - \mathcal{L}(x^\star, \bar{y}^N).$$

The $\sup_{y\in\mathbb{R}^l} \mathcal{L}(\bar{x}^N, y) = f(\bar{x}^N) + \sup_{y\in\mathbb{R}^l} \langle K\bar{x}^N, y\rangle - h^*(y)$ is attained at a $\hat{y}^N \in \partial h(K\bar{x}^N)$ because under the assumption $\mathbf{dom}\, h = \mathbb{R}^l$, h^* is 1-coercive, thus $-\langle K\bar{x}^N, \cdot\rangle + h^\star(\cdot)$ is coercive (Hiriart-Urruty and Lemaréchal, 1993, Prop.X.1.3.9; Bauschke and Combettes, 2011, Proposition 11.14). As $\bar{x}^N$ converges, $K\bar{x}^N$ is bounded independent of N. Now because h is real-valued, it follows that h is locally Lipschitz in the neighborhood of $K\bar{x}^N$ (see, e.g., Bertsekas, 2009, Proposition 5.4.2). Let the local Lipschitz constant be Q. It also follows that $\partial h(K\bar{x}^N)$ is bounded by Q, i.e. $\|\hat{y}^N\|_2 \leq Q$. Therefore

$$0 \leq \mathcal{F}(\bar{x}^N) - \mathcal{F}^\star = \mathcal{F}(\bar{x}^N) - \mathcal{L}(x^\star, y^\star) = \sup_{y\in\mathbb{R}^l} \mathcal{L}(\bar{x}^N, y) - \mathcal{L}(x^\star, \bar{y}^N)$$
$$= \max_{\|y\|_2 \leq Q} \mathcal{L}(\bar{x}^N, y) - \mathcal{L}(x^\star, \bar{y}^N)$$
$$\leq \max_{\|y\|_2 \leq Q} \frac{1}{2\sum_{k=0}^N \rho_k} \left(\|(x^0, y^0) - (x^\star, y)\|_M^2 + \frac{\alpha L_f}{(1-\alpha\bar{\rho})\lambda_{\min}(M)} \|z^0 - z^\star\|_M^2 \right)$$
$$= C_1/(\sum_{k=0}^N \rho_k).$$

□

We need the following lemma to prove Theorem 2.

Lemma 2 (Davis (2015), Theorem 4.1)**.** *Suppose $\mathcal{T} : \mathbb{R}^n \to \mathbb{R}^n$ is an α-averaged operator with respect to $\|\cdot\|_M$, where $0 < \alpha < 1$ and $M \succ 0$. Let $z^\star \in \mathbf{Fix}\,\mathcal{T}$ and $z^0 \in \mathbb{R}^n$. For $\{\rho_k\} \subset (1, 1/\alpha)$, consider a sequence $\{z^k\}$ generated by the KM iteration:*

$$z^{k+1} = z^k + \rho_k(\mathcal{T}z^k - z^k).$$

If $\tau = \sup_{k\geq 0}(1 - \alpha\rho_k)\rho_k/\alpha > 0$, then we have

$$\|\mathcal{T}z^k - z^k\|_M^2 \leq \frac{\|z^0 - z^\star\|_M^2}{\tau(k+1)} \quad \textit{and} \quad \|\mathcal{T}z^k - z^k\|_M^2 = o\left(\frac{1}{k+1}\right). \tag{B.8}$$

Proof of Theorem 2. By condition (2.21), $\|z'\|_M^2 \geq \nu\|x'\|_2^2 + \epsilon\|y'\|_2^2$ for all $z' = (x', y')$. Then, in the same manner as the proof of Theorem 1, we put $z^- = z^k$, $z^+ = \tilde{z}^k$, $\rho = \rho_k$, $z_\rho = z^{k+1}$ in Lemma 1 and note that $1 < 1/\alpha < 2$ by the assumption $\nu > L_f/2$ to have

$$\begin{aligned} 2\rho_k(\mathcal{L}(\tilde{x}^k, y) - \mathcal{L}(x, \tilde{y}^k)) \leq \|z^k - z\|_M^2 - \|z^{k+1} - z\|_M^2 + \epsilon(1 - 2/\rho_k)\|y^k - y^{k+1}\|_2^2 \\ + \left(\nu - \tfrac{2\nu - L_f}{\rho_k}\right)\|x^k - x^{k+1}\|_2^2. \end{aligned} \tag{B.9}$$

The rest of the proof closely follows that of Davis (2015, Theorem 4.2). Note ν satisfies (B.7) and hence by Proposition 5, $R_{M^{-1}F}(I - M^{-1}G) : z^k \mapsto \tilde{z}^k$ is α-averaged with respect to $\|\cdot\|_M$. Let $z_\rho = (1-\rho)z^k + \rho\tilde{z}^k =: T_\rho z^k$ for any $\rho \in (0, 1/\alpha)$; for $\rho = \rho_k$, we have $z_\rho = z^{k+1}$. Then the map $T_\rho : z^k \mapsto z_\rho$ is $\alpha\rho$-averaged with respect to $\|\cdot\|_M$ and hence $\|z_\rho - z^\star\|_M \leq \|z^k - z^\star\|_M$. From (A.3a), we have $\|z_\rho - z^\star\|_M \leq \|z^0 - z^\star\|_M$, thus by the triangle inequality $\|z_\rho - z\|_M \leq \|z^0 - z^\star\|_M + \|z^\star - z\|_M$ for any $z \in \mathbb{R}^{p+l}$. Then we have

$$\begin{aligned} (1/\rho)\langle z^k - z_\rho, z_\rho - z\rangle_M &= \langle \tilde{z}^k - z^k, z_\rho - z\rangle_M \\ &\leq \|\tilde{z}^k - z^k\|_M\|z_\rho - z\|_M \\ &\leq \frac{\|z^0 - z^\star\|_M}{\sqrt{\tau(k+1)}}(\|z^0 - z^\star\|_M + \|z^\star - z\|_M) \end{aligned} \tag{B.10}$$

for all $\rho \in (0, 1/\alpha)$, where the last inequality is from Lemma 2.

Note that Lemma 1 (with the improvement (B.9) above) still holds if ρ_k is replaced by any $\rho \in (0, 1/\alpha)$ and z^{k+1} is replaced by z_ρ. Therefore we have

$\mathcal{L}(\tilde{x}^k, y) - \mathcal{L}(x, \tilde{y}^k)$

$$
\begin{aligned}
&\le \inf_{0<\rho<1/\alpha} \frac{1}{2\rho}\left(\|z^k - z\|_M^2 - \|z_\rho - z\|_M^2 - \epsilon(\tfrac{2}{\rho}-1)\|y_\rho - y^k\|_2^2 + (\nu - \tfrac{2\nu - L_f}{\rho})\|x_\rho - x^k\|_2^2\right)\\
&= \inf_{0<\rho<1/\alpha} \frac{1}{2\rho}\Big(2\langle z^k - z_\rho, z_\rho - z\rangle_M + \|z_\rho - z^k\|_M^2 - \epsilon(\tfrac{2}{\rho}-1)\|y_\rho - y^k\|_2^2\\
&\qquad\qquad + (\nu - \tfrac{2\nu - L_f}{\rho})\|x_\rho - x^k\|_2^2\Big)\\
&\le \inf_{0<\rho<1/\alpha} \frac{1}{2\rho}\left(2\langle z^k - z_\rho, z_\rho - z\rangle_M + (\bar\lambda + \epsilon - \tfrac{2\epsilon}{\rho})\|y_\rho - y^k\|_2^2 + (\bar\lambda + \nu - \tfrac{2\nu - L_f}{\rho})\|x_\rho - x^k\|_2^2\right)\\
&= \inf_{0<\rho<1/\alpha} \frac{1}{\rho}\langle z^k - z_\rho, z_\rho - z\rangle_M + \frac{1}{2\rho}\left((\bar\lambda + \epsilon - \tfrac{2\epsilon}{\rho})\|\tilde y^k - y^k\|_2^2 + (\bar\lambda + \nu - \tfrac{2\nu - L_f}{\rho})\|\tilde x^k - x^k\|_2^2\right)\\
&\le \frac{1}{\tilde\rho}\langle z^k - z_{\tilde\rho}, z_{\tilde\rho} - z\rangle_M
\end{aligned}
$$

by choosing a small $\tilde\rho \in (0, 1/\alpha)$ such that $\bar\lambda + \epsilon \le 2\epsilon/\tilde\rho$ and $\bar\lambda + \nu \le (2\nu - L_f)/\tilde\rho$, where $\bar\lambda = \lambda_{\max}(M)$. The first equality uses the cosine rule

$$
2\langle a - b, c - b\rangle_M = -\|a - c\|_M^2 + \|a - b\|_M^2 + \|c - b\|_M^2
$$

for any $a, b, c \in \mathbb{R}^{p+l}$. The desired result follows from (B.10).

The $o(1/\sqrt{k+1})$ rate is also from (B.10) and Lemma 2. □

Proof of Proposition 3. We first show that Condition 2 is equivalent to

$$
0 \prec M^{-1} \prec \begin{bmatrix} \frac{2}{L_f} I & \\ & \infty \end{bmatrix}, \tag{B.11}
$$

or $z^T M^{-1} z < \frac{2}{L_f}\|x\|_2^2 + \delta_{\{0\}}(y)$ for all $z = (x, y) \neq 0$. To see this, let $g_1(z) = (1/2) z^T M z$ and $g_2(z) = \frac{1}{2} z^T \begin{bmatrix} \frac{L_f}{2} I & \\ & 0 \end{bmatrix} z = \frac{L_f}{4}\|x\|_2^2$. Then Condition 2 ensures that $g_1(z) > g_2(z)$ for all $z \neq 0$. Take the convex conjugates of g_1 and g_2. Observe that for $w = (w_1, w_2)$, $g_1^*(w) = \sup_z \langle w, z\rangle - (1/2) z^T M z = (1/2) w^T M^{-1} w$ and

$$
\begin{aligned}
g_2^*(w) = \sup_z \langle w, z\rangle - g_2(z) = \sup_x \langle w_1, x\rangle - \tfrac{L_f}{4}\|x\|_2^2 + \sup_y \langle w_2, y\rangle&\\
= \begin{cases} \frac{1}{L_f}\|w_1\|_2^2, & \text{if } w_2 = 0,\\ \infty, & \text{otherwise.} \end{cases}&
\end{aligned}
$$

Conjugacy asserts that $g_1^*(w) \le g_2^*(w)$, or equivalently

$$
0 \prec M^{-1} \preceq \begin{bmatrix} \frac{2}{L_f} I & \\ & \infty \end{bmatrix}.
$$

Now for $w = (w_1, 0)$ $(w_1 \neq 0)$, $f_1^*(w) = \langle w, \hat{z}\rangle - (1/2)\hat{z}^T M \hat{z} = (1/2) w_1 \bar{M}_{11} w_1$, where

$$M^{-1} = \begin{bmatrix} \bar{M}_{11} & \bar{M}_{12} \\ \bar{M}_{12}^T & \bar{M}_{22} \end{bmatrix}, \quad \hat{z} = M^{-1} w = \begin{bmatrix} \bar{M}_{11} w_1 \\ \bar{M}_{12}^T w_1 \end{bmatrix} \neq 0,$$

because $\bar{M}_{11} \succ 0$. Then

$$\begin{aligned} \tfrac{1}{2} w_1^T \bar{M}_{11} w_1 = g_1^*(w) = \langle w, \hat{z}\rangle - g_1(\hat{z}) &< \langle w, \hat{z}\rangle - g_2(\hat{z}) \\ &\leq \sup_z \langle w, z\rangle - g_2(z) = g_2^*(w) = \tfrac{1}{L_f}\|w_1\|_2^2, \end{aligned}$$

or $\bar{M}_{11} \prec \frac{2}{L_f} I$. It follows (B.11). Because both g_1 and g_2 are convex, closed, and proper, the same logic applies to g_1^* and g_2^*, meaning that the above matrix inequality implies Condition 2, establishing the equivalence.

Now Condition 1 implies $(x, 0)^T M^{-1} (x, 0) < \frac{2}{L_f}\|x\|_2^2$ for all $x \neq 0$ and $z^T M^{-1} z < \infty$, implying (B.11), thus Condition 2. That Condition 2 implies Condition 1 is straightforward, by choosing $1/\mu \in [\lambda_{\max}(\bar{M}_{11}), 2/L_f)$.

Condition 3 is equivalent to

$$0 \prec M^{-1} \preceq \begin{bmatrix} \frac{1}{\nu} I & \\ & \frac{1}{\epsilon} I \end{bmatrix}, \tag{B.12}$$

thus $(x, 0)^T M^{-1} (x, 0) \leq \frac{1}{\nu}\|x\|_2^2$ where $\nu > L_f/2$. This implies Condition 1. Finally, note that

$$\begin{aligned} z^T M^{-1} z = x^T \bar{M}_{11} x + 2x^T \bar{M}_{12}^T y + y^T \bar{M}_{22} y \\ \leq \lambda_{\max}(\bar{M}_{11})\|x\|_2^2 + 2x^T \bar{M}_{12}^T y + \lambda_{\max}(\bar{M}_{12})\|y\|_2^2, \end{aligned}$$

or

$$M^{-1} \preceq \begin{bmatrix} \lambda_{\max}(\bar{M}_{11}) I & \bar{M}_{12} \\ \bar{M}_{12}^T & \lambda_{\max}(\bar{M}_{22}) I \end{bmatrix}.$$

Both $\lambda_{\max}(\bar{M}_{11})$ and $\lambda_{\max}(\bar{M}_{22})$ are positive because $\bar{M}_{11}, \bar{M}_{22} \succ 0$. Then the second inequality in (B.12) holds if and only if either $\frac{1}{\nu} = \lambda_{\max}(\bar{M}_{11})$, $\bar{M}_{12} = 0$, $\frac{1}{\epsilon} - \lambda_{\max}(\bar{M}_{22}) \geq 0$ or $\frac{1}{\nu} > \lambda_{\max}(\bar{M}_{11})$, $\bar{M}_{12} = 0$, $\frac{1}{\epsilon} - \lambda_{\max} \geq (\frac{1}{\nu} - \lambda_{\max}(\bar{M}_{11}))^{-1}\|\bar{M}_{12}\|_2^2$ (Boyd and Vandenberghe, 2004, Appendix A). Now because Condition 1 implies $\lambda_{\max}(\bar{M}_{11}) \leq \frac{1}{\mu} < \frac{2}{L_f}$, we can choose ν and ϵ so that $\frac{1}{\mu} \leq \frac{1}{\nu} < \frac{2}{L_f}$ and $\frac{1}{\epsilon} \geq \lambda_{\max}(\bar{M}_{22}) + (\frac{1}{\nu} - \lambda_{\max}(\bar{M}_{11}))^{-1}\|\bar{M}_{12}\|_2^2$. This implies (B.12) and thus Condition 3. □

B.2 Optimal acceleration

The following proposition plays a central role in proving Theorems 3 and 4. The following proposition is a key in proving the above results.

Proposition 6. *Assume that $\rho_k \le 1$ for any k. If $z^k = (x^k, y^k)$ is generated by (2.23), then for any $z = (x, y) \in \mathcal{Z}$,*

$$\begin{aligned}
&\rho_k^{-1}\mathcal{G}(z^{k+1}, z) - (\rho_k^{-1} - 1)\mathcal{G}(z^k, z) \\
&\quad \le \langle \nabla f(x_{md}^k), \tilde{x}^{k+1} - x\rangle + \frac{\rho_k L_f}{2}\|\tilde{x}^{k+1} - \tilde{x}^k\|_2^2 \\
&\quad + g(\tilde{x}^{k+1}) - g(x) + h^*(\tilde{y}^{k+1}) - h^*(y) \\
&\quad + \langle K\tilde{x}^{k+1}, y\rangle - \langle Kx, \tilde{y}^{k+1}\rangle.
\end{aligned}$$

Upcoming Lemmas 4 and 6 are derived from Proposition 6. Theorems 3–6 follow from these lemmas.

Proof. By the convexity of f and L_f-Lipschitz smoothness of ∇f,

$$\rho_k^{-1} f(x^{k+1}) \le \rho_k^{-1} f(x_{md}^k) + \rho_k^{-1}\langle \nabla f(x_{md}^k), x^{k+1} - x_{md}^k\rangle + \frac{\rho_k^{-1} L_f}{2}\|x^{k+1} - x_{md}^k\|_2^2.$$

From equation (2.23c), $x^{k+1} - x_{md}^k = \rho_k(\tilde{x}^{k+1} - \tilde{x}^k)$. Thus,

$$\begin{aligned}
\rho_k^{-1} f(x^{k+1}) &\le \rho_k^{-1} f(x_{md}^k) + \rho_k^{-1}\langle \nabla f(x_{md}^k), x^{k+1} - x_{md}^k\rangle + \frac{\rho_k L_f}{2}\|\tilde{x}^{k+1} - \tilde{x}^k\|_2^2 \\
&\overset{(2.23h)}{=} \rho_k^{-1} f(x_{md}^k) + (\rho_k^{-1} - 1)\langle \nabla f(x_{md}^k), x^k - x_{md}^k\rangle \\
&\quad + \langle \nabla f(x_{md}^k), \tilde{x}^{k+1} - x_{md}^k\rangle + \frac{\rho_k L_f}{2}\|\tilde{x}^{k+1} - \tilde{x}^k\|_2^2 \\
&= (\rho_k^{-1} - 1)[f(x_{md}^k) + \langle \nabla f(x_{md}^k), x^k - x_{md}^k\rangle] \\
&\quad + [f(x_{md}^k) + \langle f(x_{md}^k), \tilde{x}^{k+1} - x_{md}^k\rangle] + \frac{\rho_k L_f}{2}\|\tilde{x}^{k+1} - \tilde{x}^k\|_2^2 \\
&= (\rho_k^{-1} - 1)[f(x_{md}^k) + \langle \nabla f(x_{md}^k), x^k - x_{md}^k\rangle] \\
&\quad + [f(x_{md}^k) + \langle f(x_{md}^k), x - x_{md}^k\rangle] \\
&\quad + \langle f(x_{md}^k), \tilde{x}^{k+1} - x\rangle + \frac{\rho_k L_f}{2}\|\tilde{x}^{k+1} - \tilde{x}^k\|_2^2 \\
&\le (\rho_k^{-1} - 1) f(x^k) + f(x) + \langle \nabla f(x_{md}^k), \tilde{x}^{k+1} - x\rangle + \frac{\rho_k L_f}{2}\|\tilde{x}^{k+1} - \tilde{x}^k\|_2^2,
\end{aligned} \tag{B.13}$$

where the last inequality again uses the convexity of f.

Now using the convexity of g and (2.23h), we have $g(x^{k+1}) \leq (1-\rho_k)g(x^k) + \rho_k g(\tilde{x}^{k+1})$. Thus,

$$\rho_k^{-1}[g(x^{k+1}) - g(x)] \leq (\rho_k^{-1} - 1)[g(x^k) - g(x)] + [g(x^{k+1}) - g(x)]. \tag{B.14}$$

Likewise,

$$\rho_k^{-1}[h^*(x^{k+1}) - h^*(x)] \overset{(2.23i)}{\leq} (\rho_k^{-1} - 1)[h^*(x^k) - h^*(x)] + [h^*(x^{k+1}) - h^*(x)]. \tag{B.15}$$

Combining inequalities (B.13), (B.14), and (B.15), it follows that

$$\begin{aligned}
\rho_k^{-1}\mathcal{G}(z^{k+1}, z) - (\rho_k^{-1} - 1)\mathcal{G}(z^k, z) &= \rho_k^{-1}\left\{[f(x^{k+1}) + g(x^{k+1}) + \langle Kx^{k+1}, y\rangle - h^*(y)]\right. \\
&\qquad \left. -[f(x) + g(x) - \langle Kx, y^{k+1}\rangle - h^*(y^{k+1})]\right\} \\
&\quad + (\rho_k^{-1} - 1)\left\{[f(x^k) + g(x^k) + \langle Kx^k, y\rangle - h^*(y)]\right. \\
&\qquad \left. -[f(x) + g(x) - \langle Kx, y^k\rangle - h^*(y^k)]\right\} \\
&= \rho_k^{-1} f(x^{k+1}) - (\rho_k^{-1} - 1) f(x^k) - f(x) \\
&\quad + \rho_k^{-1}[g(x^{k+1}) - g(x)] - (\rho_k^{-1} - 1)[g(x^k) - g(x)] \\
&\quad + \rho_k^{-1}[h^*(y^{k+1}) - h^*(y)] - (\rho_k^{-1} - 1)[h^*(y^k) - h^*(y)] \\
&\quad + \langle K[\rho_k^{-1}x^{k+1} - (\rho_k^{-1} - 1)x^k], y\rangle - \langle Kx, \rho_k^{-1}y^{k+1} - (\rho_k^{-1} - 1)y^k\rangle \\
&\leq f(x) + \langle \nabla f(x_{md}^k, \tilde{x}^{k+1} - x\rangle + \frac{\rho_k L_f}{2}\|x^{k+1} - x^k\|_2^2 \\
&\quad + g(\tilde{x}^{k+1}) - g(x) \\
&\quad + h^*(\tilde{y}^{k+1}) - h(y) \\
&\quad + \langle K[\rho_k^{-1}x^{k+1} - (\rho_k^{-1} - 1)x^k], y\rangle - \langle Kx, \rho_k^{-1}y^{k+1} - (\rho_k^{-1} - 1)y^k\rangle \\
&\overset{(2.23h),(2.23i)}{\leq} f(x) + \langle \nabla f(x_{md}^k, \tilde{x}^{k+1} - x\rangle + \frac{\rho_k L_f}{2}\|x^{k+1} - x^k\|_2^2 \\
&\quad + g(\tilde{x}^{k+1}) - g(x) + h^*(\tilde{y}^{k+1}) - h(y) \\
&\quad + \langle Kx^{k+1}, y\rangle - \langle Kx, y^{k+1}\rangle
\end{aligned}$$

□

The following lemmas find an upper bound for $\mathcal{G}(\tilde{z}^{k+1}, z)$.

Lemma 3 (Loris and Verhoeven (2011), Lemma 1)**.** *If* $y^+ = \mathbf{prox}_{\sigma h^*}(y^- + \sigma\Delta)$, *then*

$$\langle y - y^+, \Delta\rangle - h^*(y) + h^*(y^+) \leq \frac{1}{2\sigma}\left(\|y - y^-\|_2^2 - \|y - y^+\|_2^2 - \|y^- - y^+\|_2^2\right) \tag{B.16}$$

for any y.

Lemma 4. *If $z^{k+1} = (x^{k+1}, y^{k+1})$ is obtained by (2.23), we have the following under the condition (2.25) if $g \equiv 0$ or $A = -K$:*

$$\begin{aligned}\rho_k^{-1}\gamma_k\mathcal{G}(z^{k+1}, z) \le \mathcal{D}_k(z, \tilde{z}^{[k]}) - \gamma_k\langle\tilde{x}^{k+1} - x, B^T(\tilde{y}^{k+1} - \tilde{y}^k)\rangle + \gamma_k\langle A(\tilde{x}^{k+1} - \tilde{x}^k), \tilde{y}^{k+1} - y\rangle \\ + \tau_k\gamma_k\langle(K+B)^T(\tilde{y}^{k+1} - \tilde{y}^k), (K+A)^T(\tilde{y}^{k+1} - y)\rangle - \gamma_k\left(\frac{1-q}{2\tau_k} - \frac{L_f\rho_k}{2}\right)\|\tilde{x}^{k+1} - \tilde{x}^k\|_2^2 \\ - \gamma_k\left(\frac{1-r}{2\sigma_k} - \frac{\|K+A\|_2\|K+B\|_2\tau_{k-1}}{2}\right)\|\tilde{y}^{k+1} - \tilde{y}^k\|_2^2, \qquad \text{(B.17)}\end{aligned}$$

where γ_k is defined by

$$\gamma_k = \begin{cases} 1 & \text{if } k = 1 \\ \theta_k^{-1}\gamma_{k-1} & \text{if } k \ge 2 \end{cases}, \qquad \text{(B.18)}$$

and $\mathcal{D}_k(z, \tilde{z}^{[k]})$ is defined by

$$\mathcal{D}_k(z, \tilde{z}^{[k]}) := \sum_{i=1}^{k}\left[\frac{\gamma_i}{2\tau_i}(\|x - \tilde{x}^i\|_2^2 - \|x - \tilde{x}^{i+1}\|_2^2) + \frac{\gamma_i}{2\sigma_i}(\|y - \tilde{y}^i\|_2^2 - \|y - \tilde{y}^{i+1}\|_2^2)\right]. \qquad \text{(B.19)}$$

Proof. For iteration (2.23), the following relation holds by Lemma 3 and Lemma 3:

$$\langle y - \tilde{y}^{k+1}, \tilde{u}^{k+1}\rangle + h^*(\tilde{y}^{k+1}) - h^*(y) \le \frac{1}{2\sigma_k}\left(\|y - \tilde{y}^k\|_2^2 - \|\tilde{y}^{k+1} - \tilde{y}^k\|_2^2 - \|y - \tilde{y}^{k+1}\|_2^2\right),$$

$$\langle\tilde{x}^{k+1} - x, \nabla f(x_{md}^k) + \tilde{v}^{k+1}\rangle + g(\tilde{x}^{k+1}) - g(x) = \frac{1}{2\tau_k}\left(\|x - \tilde{x}^k\|_2^2 - \|\tilde{x}^{k+1} - \tilde{x}^k\|_2^2 - \|x - \tilde{x}^{k+1}\|_2^2\right).$$

Using the above relationship along with Proposition 6, we obtain the following.

$$\begin{aligned}&\rho_k^{-1}\mathcal{G}(z^{k+1}, z) - (\rho_k^{-1} - 1)\mathcal{G}(z^k, z) \\ &\qquad \le \frac{1}{2\tau_k}\left(\|x - \tilde{x}^k\|_2^2 - \|x - \tilde{x}^{k+1}\|_2^2\right) - \left(\frac{1}{2\tau_k} - \frac{L_f\rho_k}{2}\right)\|\tilde{x}^{k+1} - \tilde{x}^k\|_2^2 \\ &\qquad + \frac{1}{2\sigma_k}\left(\|y - \tilde{y}^k\|_2^2 - \|y - \tilde{y}^{k+1}\|_2^2\right) - \frac{1}{2\sigma_k}\|\tilde{y}^{k+1} - \tilde{y}^k\|_2^2 \\ &\qquad - \langle\tilde{x}^{k+1} - x, \tilde{v}^{k+1}\rangle + \langle\tilde{u}^{k+1}, \tilde{y}^{k+1} - y\rangle + \langle K\tilde{x}^{k+1}, y\rangle - \langle Kx, \tilde{y}^{k+1}\rangle. \qquad \text{(B.20)}\end{aligned}$$

The sum of the four inner products on the last line, namely, $-\langle\tilde{x}^{k+1} - x, \tilde{v}^{k+1}\rangle + \langle\tilde{u}^{k+1}, \tilde{y}^{k+1} - y\rangle + \langle K\tilde{x}^{k+1}, y\rangle - \langle Kx, \tilde{y}^{k+1}\rangle$, multiplied by γ_k can be computed

as follows.

$$\begin{aligned}
\gamma_k[-\langle\tilde{x}^{k+1}-x,&\tilde{v}^{k+1}\rangle+\langle\tilde{u}^{k+1},\tilde{y}^{k+1}-y\rangle+\langle K\tilde{x}^{k+1},y\rangle-\langle Kx,\tilde{y}^{k+1}\rangle]\\
&=\gamma_k[-\big(\langle\tilde{x}^{k+1}-x,B^T(\tilde{y}^{k+1}-\tilde{y}^k)\rangle-\theta_k\langle\tilde{x}^{k+1}-x,B^T(\tilde{y}^k-\tilde{y}^{k-1})\rangle\big)\\
&\quad+\big(\langle A(\tilde{x}^{k+1}-\tilde{x}^k),\tilde{y}^{k+1}-y\rangle-\theta_k\langle A(\tilde{x}^k-\tilde{x}^{k-1}),\tilde{y}^{k+1}-y\rangle\big)\\
&\quad+\tau_k\langle(K+A)(K+B)^T(\tilde{y}^{k+1}-\tilde{y}^k),\tilde{y}^{k+1}-y\rangle\\
&\quad-\tau_{k-1}\theta_k\langle(K+A)(K+B)^T(\tilde{y}^k-\tilde{y}^{k-1}),\tilde{y}^{k+1}-y\rangle]\\
&=-\big(\gamma_k\langle\tilde{x}^{k+1}-x,B^T(\tilde{y}^{k+1}-\tilde{y}^k)\rangle-\gamma_{k-1}\langle\tilde{x}^k-x,B^T(\tilde{y}^k-\tilde{y}^{k-1})\big)\\
&\quad+\big(\gamma_k\langle A(\tilde{x}^{k+1}-\tilde{x}^k),\tilde{y}^{k+1}-y\rangle-\gamma_{k-1}\langle A(\tilde{x}^k-\tilde{x}^{k-1}),\tilde{y}^k-y\rangle\big)\\
&\quad+\tau_k\gamma_k\langle(K+B)^T(\tilde{y}^{k+1}-\tilde{y}^k),(K+A)^T(\tilde{y}^{k+1}-y)\rangle\\
&\quad-\tau_{k-1}\gamma_{k-1}\langle(K+B)^T(\tilde{y}^k-\tilde{y}^{k-1}),(K+A)^T(\tilde{y}^k-y)\rangle\\
&\quad+\gamma_{k-1}\langle\tilde{x}^{k+1}-\tilde{x}^k,B^T(\tilde{y}^k-\tilde{y}^{k-1})\rangle-\gamma_{k-1}\langle A(\tilde{x}^k-\tilde{x}^{k-1}),\tilde{y}^{k+1}-\tilde{y}^k\rangle\\
&\quad-\gamma_{k-1}\tau_{k-1}\langle(K+B)^T(\tilde{y}^k-\tilde{y}^{k-1}),(K+A)^T(\tilde{y}^{k+1}-\tilde{y}^k)\rangle.
\end{aligned}$$

We used the relation

$$\begin{aligned}
\tilde{u}^{k+1}=&K\tilde{x}^{k+1}+A(\tilde{x}^{k+1}-\tilde{x}^k)-\theta_kA(\tilde{x}^k-\tilde{x}^{k-1})\\
&+\tau_k(K+A)(K+B)^T(\tilde{y}^{k+1}-\tilde{y}^k)-\theta_k\tau_{k-1}(K+A)(K+B)^T(\tilde{y}^k-\tilde{y}^{k-1}),
\end{aligned}$$

which holds if $g\equiv 0$ or $A=-K$ in the first equality.

By upper bounding the inner product terms, and noting that $\theta_k=\gamma_{k-1}/\gamma_k=\tau_{k-1}/\tau_k=\sigma_{k-1}/\sigma_k$, we have:

$$\begin{aligned}
|\gamma_{k-1}\langle\tilde{x}^{k+1}-\tilde{x}^k,B^T(\tilde{y}^k-\tilde{y}^{k-1})\rangle|&\le\frac{\gamma_kq}{2\tau_k}\|\tilde{x}^{k+1}-\tilde{x}^k\|_2^2+\frac{\|B\|_2^2\gamma_{k-1}\tau_{k-1}}{2q}\|\tilde{y}^k-\tilde{y}^{k-1}\|_2^2\\
|\gamma_{k-1}\langle\tilde{x}^k-\tilde{x}^{k-1},A^T(\tilde{y}^{k+1}-\tilde{y}^k)\rangle|&\le\frac{\|A\|_2^2\gamma_{k-1}\sigma_{k-1}}{2r}\|\tilde{x}^k-\tilde{x}^{k-1}\|_2^2+\frac{\gamma_kr}{2\sigma_k}\|\tilde{y}^{k+1}-\tilde{y}^k\|_2^2\\
|\gamma_{k-1}\tau_{k-1}\langle(K+B)^T(\tilde{y}^k-\tilde{y}^{k-1}),&(K+A)^T(\tilde{y}^{k+1}-\tilde{y}^k)\rangle|\\
&\le\frac{\|K+A\|_2\|K+B\|_2\gamma_{k-1}\tau_{k-1}\theta_k}{2}\|\tilde{y}^k-\tilde{y}^{k-1}\|_2^2\\
&\quad+\frac{\|K+A\|_2\|K+B\|_2\gamma_{k-1}\tau_{k-1}}{2\theta_k}\|\tilde{y}^{k+1}-\tilde{y}^k\|_2^2
\end{aligned}\tag{B.21}$$

for some positive q and r. Thus

$$\begin{aligned}
\rho_k^{-1}\gamma_k\mathcal{G}(z^{k+1},z)&-(\rho_k^{-1}-1)\gamma_k\mathcal{G}(z^k,z)\\
\le&\frac{\gamma_k}{2\tau_k}\left(\|x-\tilde{x}^k\|_2^2-\|x-\tilde{x}^{k+1}\|_2^2\right)+\frac{\gamma_k}{2\sigma_k}\left(\|y-\tilde{y}^k\|_2^2-\|y-\tilde{y}^{k+1}\|_2^2\right)\\
&-\left(\gamma_k\langle\tilde{x}^{k+1}-x,B^T(\tilde{y}^{k+1}-\tilde{y}^k)\rangle-\gamma_{k-1}\langle\tilde{x}^k-x,B^T(\tilde{y}^k-\tilde{y}^{k-1})\right)
\end{aligned}$$

$$
\begin{aligned}
&+ \left(\gamma_k\langle \tilde{x}^{k+1} - \tilde{x}^k, A^T(\tilde{y}^{k+1} - y)\rangle - \gamma_{k-1}\langle \tilde{x}^k - \tilde{x}^{k-1}, A^T(\tilde{y}^k - y)\rangle\right) \\
&+ \tau_k\gamma_k\langle (K+B)^T(\tilde{y}^{k+1} - \tilde{y}^k), (K+A)^T(\tilde{y}^{k+1} - y)\rangle \\
&- \tau_{k-1}\gamma_{k-1}\langle (K+B)^T(\tilde{y}^k - \tilde{y}^{k-1}), (K+A)^T(\tilde{y}^k - y)\rangle \\
&- \gamma_k\left(\frac{1-q}{2\tau_k} - \frac{L_f\rho_k}{2}\right)\|\tilde{x}^{k+1} - \tilde{x}^k\|_2^2 + \frac{\|A\|_2^2\gamma_{k-1}\sigma_{k-1}}{2r}\|\tilde{x}^k - \tilde{x}^{k-1}\|_2^2 \\
&- \gamma_k\left(\frac{1-r}{2\sigma_k} - \frac{\|K+A\|_2\|K+B\|_2\tau_{k-1}}{2}\right)\|\tilde{y}^{k+1} - \tilde{y}^k\|_2^2 \\
&+ \frac{\gamma_{k-1}\tau_{k-1}}{2}\left(\frac{\|B\|_2^2}{q} + \|K+A\|_2\|K+B\|_2\theta_k\right)\|\tilde{y}^k - \tilde{y}^{k-1}\|_2^2.
\end{aligned}
$$

Recursively applying the above relation, we obtain:

$$
\begin{aligned}
&\rho_k^{-1}\gamma_k\mathcal{G}(z^{k+1}, z) \\
&\quad\le \mathcal{D}_k(z, \tilde{z}^{[k]}) - \gamma_k(\langle \tilde{x}^{k+1} - x, B^T(\tilde{y}^{k+1} - \tilde{y}^k)\rangle - \langle \tilde{x}^{k+1} - \tilde{x}^k, A^T(\tilde{y}^{k+1} - y)\rangle \\
&\qquad - \tau_k\gamma_k\langle (K+B)^T(\tilde{y}^{k+1} - \tilde{y}^k), (K+A)^T(\tilde{y}^{k+1} - y)\rangle) \\
&\qquad - \gamma_k\left(\frac{1-q}{2\tau_k} - \frac{L_f\rho_k}{2}\right)\|\tilde{x}^{k+1} \\
&\qquad - \tilde{x}^k\|_2^2 - \gamma_k\left(\frac{1-r}{2\sigma_k} - \frac{\|K+A\|_2\|K+B\|_2\tau_{k-1}}{2}\right)\|\tilde{y}^{k+1} - \tilde{y}^k\|_2^2 \\
&\qquad - \sum_{i=1}^{k-1}\gamma_i\left(\frac{1-q}{2\tau_i} - \frac{L_f\rho_k}{2} - \frac{\|A\|_2^2\sigma_i}{2r}\right)\|\tilde{x}^{i+1} - \tilde{x}^i\|_2^2 \\
&\qquad - \sum_{i=1}^{k-1}\gamma_i\Bigg(\frac{1-r}{2\sigma_i} - \frac{\|K+A\|_2\|K+B\|_2\tau_{i-1}}{2} \\
&\qquad\quad - \frac{\tau_i}{2}\left(\frac{\|B\|_2^2}{q} + \|K+A\|_2\|K+B\|_2\theta_i\right)\Bigg)\|\tilde{y}^{i+1} - \tilde{y}^i\|_2^2.
\end{aligned}
$$

Thus by the conditions (2.25), the desired result holds. □

Proof of Theorem 3. First we find an upper bound of $\mathcal{D}_k(z, \tilde{z}^{[k]})$.

$$
\begin{aligned}
\mathcal{D}_k(z, \tilde{z}^{[k]}) = \frac{\gamma_1}{2\tau_1}\|x - \tilde{x}^1\|_2^2 - \sum_{i=1}^{k-1}\frac{1}{2}\left(\frac{\gamma_i}{\tau_i} - \frac{\gamma_{i+1}}{\tau_{i+1}}\right)\|x - \tilde{x}^{i+1}\|_2^2 - \frac{\gamma_k}{2\tau_k}\|x - \tilde{x}^{k+1}\|_2^2 \\
+ \frac{\gamma_1}{2\sigma_1}\|y - \tilde{y}^1\|_2^2 - \sum_{i=1}^{k-1}\frac{1}{2}\left(\frac{\gamma_i}{\sigma_i} - \frac{\gamma_{i+1}}{\sigma_{i+1}}\right)\|y - \tilde{y}^{i+1}\|_2^2 - \frac{\gamma_k}{2\sigma_k}\|y - \tilde{y}^{k+1}\|_2^2
\end{aligned}
$$

$$
\begin{aligned}
&\leq \frac{\gamma_1}{\tau_1}\Omega_X^2 - \sum_{i=1}^{k-1}\left(\frac{\gamma_i}{\tau_i} - \frac{\gamma_{i+1}}{\tau_{i+1}}\right)\Omega_X^2 - \frac{\gamma_k}{2\tau_k}\|x - \tilde{x}^{k+1}\|_2^2 \\
&\quad + \frac{\gamma_1}{\sigma_1}\Omega_Y^2 - \sum_{i=1}^{k-1}\left(\frac{\gamma_i}{\sigma_i} - \frac{\gamma_{i+1}}{\sigma_{i+1}}\right)\Omega_Y^2 - \frac{\gamma_k}{2\sigma_k}\|y - \tilde{y}^{k+1}\|_2^2 \\
&= \frac{\gamma_k}{\tau_k}\Omega_X^2 + \frac{\gamma_k}{\sigma_k}\Omega_Y^2 - \gamma_k\left(\frac{1}{2\tau_k}\|x - \tilde{x}^{k+1}\|_2^2 + \frac{1}{2\sigma_k}\|y - \tilde{y}^{k+1}\|_2^2\right), \qquad \text{(B.22)}
\end{aligned}
$$

where we used (2.24) for the inequality.

Consider the following upper bounds of the three inner product terms in (B.17):

$$
\begin{aligned}
|\gamma_k\langle \tilde{x}^{k+1} - x, B^T(\tilde{y}^{k+1} - \tilde{y}^k)\rangle| &\leq \frac{\gamma_k q}{2\tau_k}\|\tilde{x}^{k+1} - x\|_2^2 + \frac{\|B\|_2^2\gamma_k\tau_k}{2q}\|\tilde{y}^{k+1} - \tilde{y}^k\|_2^2 \\
|\gamma_k\langle \tilde{x}^{k+1} - \tilde{x}^k, A^T(\tilde{y}^{k+1} - y)\rangle| &\leq \frac{\|A\|_2^2\gamma_k\sigma_k}{2r}\|\tilde{x}^{k+1} - \tilde{x}^k\|_2^2 + \frac{\gamma_k r}{2\sigma_k}\|\tilde{y}^{k+1} - y\|_2^2 \\
|\tau_k\langle (K+B)^T(\tilde{y}^{k+1} - \tilde{y}^k), (K+A)^T(\tilde{y}^{k+1} - y)\rangle| & \\
&\leq \frac{\|K+A\|_2\|K+B\|_2\gamma_k\tau_k}{2}\|\tilde{y}^{k+1} - \tilde{y}^k\|_2^2 \\
&\quad + \frac{\|K+A\|_2\|K+B\|_2\gamma_k\tau_k}{2}\|\tilde{y}^{k+1} - y\|_2^2. \qquad \text{(B.23)}
\end{aligned}
$$

Then (2.25a), (B.17), (B.22), and (B.23) imply that

$$
\begin{aligned}
\rho_k^{-1}\gamma_k\mathcal{G}(z^{k+1}, z) &\leq \frac{\gamma_k}{\tau_k}\Omega_X^2 + \frac{\gamma_k}{\sigma_k}\Omega_Y^2 - \gamma_k\frac{1-q}{2\tau_k}\|x - \tilde{x}^{k+1}\|_2^2 \\
&\quad - \gamma_k\left(\frac{1-r}{2\sigma_k} - \frac{\|K+A\|_2\|K+B\|_2\tau_k}{2}\right)\|y - \tilde{y}^{k+1}\|_2^2 \\
&\quad - \gamma_k\left(\frac{1-q}{2\tau_k} - \frac{L_f\rho_k}{2} - \frac{\|A\|_2^2\sigma_k}{2}\right)\|\tilde{x}^{k+1} - \tilde{x}^k\|_2^2 \\
&\quad - \gamma_k\left(\frac{1-r}{2\sigma_k} - \frac{\tau_k}{2}\left(2\|K+A\|_2\|K+B\|_2 + \|B\|_2^2\right)\right)\|\tilde{y}^{k+1} - \tilde{y}^k\|_2^2 \\
&\leq \frac{\gamma_k}{\tau_k}\Omega_X^2 + \frac{\gamma_k}{\sigma_k}\Omega_Y^2.
\end{aligned}
$$

That is, (2.27). □

Proof of Corollary 2. First check (2.28) and (2.29) satisfy (2.25):

$$
\frac{1-q}{\tau_k} - L_f\rho_k - \frac{\|A\|_2^2\sigma_k}{r} \geq \left((1-q)P_2 - \frac{a^2}{r}\right)\frac{\Omega_X L_K}{\Omega_Y} \geq 0,
$$

$$\frac{1-r}{\sigma_k} - \tau_k \left(2\|K+A\|_2 \|K+B\|_2 + \frac{\|B\|_2^2}{q} \right) \geq \left(1 - r - \frac{2cd + b^2/q}{P_2} \right) \frac{\Omega_X L_K}{\Omega_Y} \geq 0,$$

Then by (2.27), we have

$$\begin{aligned}
\mathcal{G}^\star(z^k) &\geq \frac{\rho_{k-1}}{\tau_{k-1}} \Omega_X^2 + \frac{\rho_{k-1}}{\sigma_{k-1}} \Omega_Y^2 \\
&= \frac{4P_1 L_f + 2P_2(k-1) L_K \Omega_Y / \Omega_X}{k(k-1)} \Omega_X^2 + \frac{2L_K \Omega_X / \Omega_Y}{k} \Omega_Y^2 \\
&= \frac{4P_1 \Omega_X^2}{k(k-1)} L_f + \frac{2\Omega_X \Omega_Y (P_2+1)}{k} L_K.
\end{aligned}$$

□

We need the following lemma to prove Theorem 4.

Lemma 5. *Consider a saddle point $\hat{z} = (\hat{x}, \hat{y})$ of the problem (2.2), and the parameters ρ_k, θ_k, τ_k, and σ_k satisfying the conditions for Theorem 4. Then*

$$\|x - \tilde{x}^1\|_2^2 + \frac{\tau_k}{\sigma_k} \|y - \tilde{y}^1\|_2^2 \geq (1-q) \|x - \tilde{x}^{k+1}\|_2^2 + \frac{\tau_k}{\sigma_k} \left(\frac{1}{2} - r \right) \|y - \tilde{y}^{k+1}\|_2^2 \tag{B.24}$$

and

$$\tilde{\mathcal{G}}(\tilde{z}^{k+1}, v^{k+1}) \leq \frac{\rho_k}{2\tau_k} \|x^{k+1} - \tilde{x}^1\|_2^2 + \frac{\rho_k}{2\sigma_k} \|y^{k+1} - \tilde{y}^1\|_2^2 =: \delta_{k+1} \tag{B.25}$$

for all $t \geq 1$, where $\tilde{\mathcal{G}}$ is defined in (2.31), and

$$\begin{aligned}
v^{k+1} = \Bigg(& \frac{\rho_k}{\tau_k} (\tilde{x}^1 - \tilde{x}^{k+1}) - B^T (\tilde{y}^{k+1} - \tilde{y}^k), \\
& \frac{\rho_k}{\sigma_k} (\tilde{y}^1 - \tilde{y}^{k+1}) + A(\tilde{x}^{k+1} - \tilde{x}^k) + (K+A)(K+B)^T (\tilde{y}^{k+1} - \tilde{y}^k) \Bigg)
\end{aligned} \tag{B.26}$$

Proof. First, let us prove (B.24). The conditions for Lemma 4 clearly holds.

Note that

$$\mathcal{D}_k(z,\tilde{z}^{[k]}) = \frac{\gamma_1}{2\tau_1}\|x-\tilde{x}^1\|_2^2 - \sum_{i=1}^{k-1}\left(\frac{\gamma_i}{2\tau_i}-\frac{\gamma_{i+1}}{2\tau_{i+1}}\right)\|x-\tilde{x}^{k+1}\|_2^2 - \frac{\gamma_k}{2\tau_k}\|x-\tilde{x}^{k+1}\|_2^2 + \frac{\gamma_1}{2\sigma_1}\|y-\tilde{y}^1\|_2^2 - \sum_{i=1}^{k-1}\left(\frac{\gamma_i}{2\sigma_i}-\frac{\gamma_{i+1}}{2\sigma_{i+1}}\right)\|y-\tilde{y}^{k+1}\|_2^2 - \frac{\gamma_k}{2\sigma_k}\|y-\tilde{y}^{k+1}\|_2^2. \tag{B.27}$$

By (2.32), one may see that

$$\gamma_k^{-1}\mathcal{D}_k(z,z^{[k]}) = \frac{1}{2\tau_k}\|x-\tilde{x}^1\|_2^2 - \frac{1}{2\tau_k}\|x-\tilde{x}^{k+1}\|_2^2 + \frac{1}{2\sigma_k}\|y-\tilde{y}^1\|_2^2 - \frac{1}{2\sigma_k}\|y-\tilde{y}^{k+1}\|_2^2.$$

Thus (B.17) is equivalent to

$$\begin{aligned}\rho_k^{-1}\mathcal{G}(\tilde{z}^{k+1},z) \le{}& \frac{1}{2\tau_k}\|x-\tilde{x}^1\|_2^2 - \frac{1}{2\tau_k}\|x-\tilde{x}^{k+1}\|_2^2 + \frac{1}{2\sigma_k}\|y-\tilde{y}^1\|_2^2 - \frac{1}{2\sigma_k}\|y-\tilde{y}^{k+1}\|_2^2\\ &- \langle\tilde{x}^{k+1}-x, B^T(\tilde{y}^{k+1}-\tilde{y}^k)\rangle + \gamma_k\langle A(\tilde{x}^{k+1}-\tilde{x}^k), \tilde{y}^{k+1}-y\rangle\\ &+ \tau_k\langle (K+B)^T(\tilde{y}^{k+1}-\tilde{y}^k), (K+A)^T(\tilde{y}^{k+1}-y)\rangle\\ &- \left(\frac{1-q}{2\tau_k}-\frac{L_f\rho_k}{2}\right)\|\tilde{x}^{k+1}-\tilde{x}^k\|_2^2\\ &- \left(\frac{1-r}{2\sigma_k}-\frac{\|K+A\|_2\|K+B\|_2\tau_{k-1}}{2}\right)\|\tilde{y}^{k+1}-\tilde{y}^k\|_2^2.\end{aligned}$$

Note that

$$\begin{aligned}&\left|\langle A(\tilde{x}^{k+1}-\tilde{x}^k), \tilde{y}^{k+1}-y\rangle\right| \le \frac{\|A\|_2^2\sigma_k}{2r}\|\tilde{x}^{k+1}-\tilde{x}^k\|_2^2 + \frac{r}{2\sigma_k}\|\tilde{y}^{k+1}-y\|_2^2\\ &|\tau_k\langle (K+B)^T(\tilde{y}^{k+1}-\tilde{y}^k), (K+A)^T(\tilde{y}^{k+1}-y)\rangle|\\ &\qquad\le \tau_k^2\sigma_k\|K+A\|_2^2\|K+B\|_2^2\|\tilde{y}^{k+1}-\tilde{y}^k\|_2^2 + \frac{1}{4\sigma_k}\|\tilde{y}^{k+1}-y\|_2^2\\ &\left|\langle\tilde{x}^{k+1}-x, B^T(\tilde{y}^{k+1}-\tilde{y}^k)\rangle\right| \le \frac{q}{2\tau_k}\|\tilde{x}^{k+1}-x\|_2^2 + \frac{\|B\|_2^2\tau_k}{2q}\|\tilde{y}^{k+1}-\tilde{y}^k\|_2^2.\end{aligned} \tag{B.28}$$

Thus

$$\begin{aligned}\rho_k^{-1}\mathcal{G}(z^{k+1},z) \le{}& \frac{1}{2\tau_k}\|x-\tilde{x}^1\|_2^2 - \frac{1-q}{2\tau_k}\|x-\tilde{x}^{k+1}\|_2^2\\ &+ \frac{1}{2\sigma_k}\|y-\tilde{y}^1\|_2^2 - \frac{1}{2\sigma_k}\left(1-r-\frac{1}{2}\right)\|y-\tilde{y}^{k+1}\|_2^2\\ &- \left(\frac{1-q}{2\tau_k}-\frac{L_f\rho_k}{2}-\frac{\|A\|_2^2\sigma_k}{2r}\right)\|\tilde{x}^{k+1}-\tilde{x}^k\|_2^2\end{aligned}$$

$$- \left(\frac{1-r}{2\sigma_k} - \frac{\|K+A\|_2\|K+B\|_2\tau_{k-1}}{2} - \frac{\|B\|_2^2\tau_k}{2q} - \|K+A\|_2^2\|K+B\|_2^2\tau_k^2\sigma_k \right) \|\tilde{y}^{k+1} - \tilde{y}^k\|_2^2.$$

It can be easily seen that

$$\begin{aligned}
&\frac{1-r}{2\sigma_k} - \frac{\|K+A\|_2\|K+B\|_2\tau_{k-1}}{2} - \frac{\|B\|_2^2\tau_k}{2q} - \|K+A\|_2^2\|K+B\|_2^2\tau_k^2\sigma_k \\
&\qquad \geq \frac{1-r}{2\sigma_k} - \frac{\tau_k\|B\|_2^2}{2q} - \tau_k\|K+A\|_2\|K+B\|_2 \\
&\qquad \geq 0.
\end{aligned}$$

Hence

$$\rho_k^{-1}\mathcal{G}(z^{k+1}, z) \leq \frac{1}{2\tau_k}\|x-\tilde{x}^1\|_2^2 - \frac{1-q}{2\tau_k}\|x-\tilde{x}^{k+1}\|_2^2 + \frac{1}{2\sigma_k}\|y-\tilde{y}^1\|_2^2 - \frac{1/2-r}{2\sigma_k}\|y-\tilde{y}^{k+1}\|_2^2.$$

Since $\mathcal{G}(z^{k+1}, \hat{z}) \geq 0$, we obtain

$$\|x-\tilde{x}^1\|_2^2 + \frac{\tau_k}{\sigma_k}\|y-\tilde{y}^1\|_2^2 \geq (1-q)\|x-\tilde{x}^{k+1}\|_2^2 + \frac{\tau_k}{\sigma_k}(1/2-r)\|y-\tilde{y}^{k+1}\|_2^2.$$

Next, we prove (B.25). Note that

$$\begin{aligned}
\|x-\tilde{x}^1\|_2^2 - \|x-\tilde{x}^{k+1}\|_2^2 &= 2\langle \tilde{x}^{k+1} - \tilde{x}^1, x - x^{k+1}\rangle + \|x^{k+1}-\tilde{x}^1\|_2^2 - \|x^{k+1}-\tilde{x}^{k+1}\|_2^2 \\
\|y-\tilde{y}^1\|_2^2 - \|y-\tilde{y}^{k+1}\|_2^2 &= 2\langle \tilde{y}^{k+1} - \tilde{y}^1, y - y^{k+1}\rangle + \|y^{k+1}-\tilde{y}^1\|_2^2 - \|y^{k+1}-\tilde{y}^{k+1}\|_2^2.
\end{aligned} \tag{B.29}$$

From this, we have:

$$\begin{aligned}
&\rho_k^{-1}\mathcal{G}(z^{k+1}, z) - \frac{1}{\tau_k}\langle \tilde{x}^1 - \tilde{x}^{k+1}, x^{k+1} - x\rangle - \frac{1}{\sigma_k}\langle \tilde{y}^1 - \tilde{y}^{k+1}, y^{k+1} - y\rangle \\
&\quad - \langle x - x^{k+1}, B^T(\tilde{y}^{k+1} - \tilde{y}^k)\rangle + \langle A(\tilde{x}^{k+1} - \tilde{x}^k), y - y^{k+1}\rangle \\
&\quad + \tau_k\langle (K+B)^T(\tilde{y}^{k+1} - \tilde{y}^k), (K+A)^T(y - y^{k+1})\rangle \\
&\leq \frac{1}{2\tau_k}\left(\|x^{k+1}-\tilde{x}^1\|_2^2 - \|x^{k+1}-\tilde{x}^{k+1}\|_2^2\right) + \frac{1}{2\sigma_k}\left(\|y^{k+1}-\tilde{y}^1\|_2^2 - \|y^{k+1}-\tilde{y}^{k+1}\|_2^2\right) \\
&\quad - \left(\frac{1-q}{2\tau_k} - \frac{L_f\rho_k}{2}\right)\|\tilde{x}^{k+1} - \tilde{x}^k\|_2^2 \\
&\quad - \left(\frac{1-r}{2\sigma_k} - \frac{\|K+A\|_2\|K+B\|_2\tau_{k-1}}{2}\right)\|\tilde{y}^{k+1} - \tilde{y}^k\|_2^2 \\
&\quad - \langle \tilde{x}^{k+1} - x^{k+1}, B^T(\tilde{y}^{k+1} - \tilde{y}^k)\rangle + \langle A(\tilde{x}^{k+1} - \tilde{x}^k), \tilde{y}^{k+1} - y^{k+1}\rangle \\
&\quad + \tau_k\langle (K+B)^T(\tilde{y}^{k+1} - \tilde{y}^k), (K+A)^T(\tilde{y}^{k+1} - y^{k+1})\rangle \\
&\leq \frac{1}{2\tau_k}\|x^{k+1} - \tilde{x}^k\|_2^2 + \frac{1}{2\sigma_k}\|y^{k+1} - \tilde{y}^1\|_2^2 \\
&\quad - \frac{1-q}{2\tau_k}\|x^{k+1} - \tilde{x}^{k+1}\|_2^2 - \frac{1/2-r}{2\sigma_k}\|y^{k+1} - \tilde{y}^{k+1}\|_2^2
\end{aligned}$$

$$
\begin{aligned}
&- \left(\frac{1-q}{2\tau_k} - \frac{L_f \rho_k}{2} - \frac{\|A\|_2^2 \sigma_k}{2r} \right) \|\tilde{x}^{k+1} - \tilde{x}^k\|_2^2 \\
&- \left(\frac{1-r}{2\sigma_k} - \frac{\|K+A\|_2 \|K+B\|_2 \tau_{k-1}}{2} - \frac{\|B\|_2^2 \tau_k}{2q} - \|K+A\|_2^2 \|K+B\|_2^2 \tau_k^2 \sigma_k \right) \|\tilde{y}^{k+1} - \tilde{y}^k\|_2^2 \\
&\leq \frac{1}{2\tau_k} \|x^{k+1} - \tilde{x}^1\|_2^2 + \frac{1}{2\sigma_k} \|y^{k+1} - \tilde{y}^1\|_2^2.
\end{aligned}
$$

In the penultimate inequality, the upper bound for inner product terms similar to (B.28) was used. □

Proof of Theorem 4. It is sufficient to find upper bounds of $\|v^{k+1}\|_2$ and δ_{k+1}. From the definition of R and (B.24), we have $\|\hat{x} - \tilde{x}^{k+1}\|_2 \leq \mu R$ and $\|\hat{y} - \tilde{y}^{k+1}\|_2 \leq \sqrt{\frac{\sigma_k}{\tau_k}} \nu R$. For v^{k+1} defined in (B.26),

$$
\begin{aligned}
\|v^{k+1}\|_2 &\leq \rho_k \Big(\frac{1}{\tau_k} \|\tilde{x}^1 - \tilde{x}^{k+1}\|_2 + \|B\|_2 \|\tilde{y}^{k+1} - \tilde{y}^k\|_2 \\
&\quad + \frac{1}{\sigma_k} \|\tilde{y}^1 - \tilde{y}^{k+1}\|_2 + \|A\|_2 \|\tilde{x}^{k+1} - \tilde{x}^k\|_2 + \|K+A\|_2 \|K+B\|_2 \tau_k \|\tilde{y}^{k+1} - \tilde{y}^k\|_2 \Big) \\
&\leq \rho_k \Big(\frac{1}{\tau_k} (\|\hat{x} - \tilde{x}^1\|_2 + \|\hat{x} - \tilde{x}^{k+1}\|_2) + \frac{1}{\sigma_k} (\|\hat{y} - \tilde{y}^1\|_2 + \|\hat{y} - \tilde{y}^{k+1}\|_2) \\
&\quad + \|A\|_2 (\|\hat{x} - \tilde{x}^{k+1}\|_2 + \|\hat{x} - \tilde{x}^k\|_2) \\
&\quad + (\|B\|_2 + \|K+A\|_2 \|K+B\|_2 \tau_k) (\|\hat{y} - \tilde{y}^{k+1}\|_2 + \|\hat{y} - \tilde{y}^k\|_2) \\
&\leq \frac{\rho_k}{\tau_k} \|\hat{x} - \tilde{x}^1\|_2 + \frac{\rho_k}{\sigma_k} \|\hat{y} - \tilde{y}^1\|_2 \\
&\quad + \rho_k \left(\frac{1}{\tau_k} + 2\|A\|_2 \right) \mu R + \rho_k \left(\frac{1}{\sigma_k} + 2\|B\|_2 + 2\|K+A\|_2 \|K+B\|_2 \tau_k \right) \nu R \\
&= \frac{\rho_k}{\tau_k} \|\hat{x} - \tilde{x}^1\|_2 + \frac{\rho_k}{\sigma_k} \|\hat{y} - \tilde{y}^1\|_2 \\
&\quad + R \left[\frac{\rho_k}{\tau_k} \left(\mu + \frac{\tau_1}{\sigma_1} \nu \right) + 2\rho_k \left(\|A\|_2 \mu + \|B\|_2 \nu \right) + 2\tau_k \rho_k \|K+A\|_2 \|K+B\|_2 \nu \right],
\end{aligned}
$$

i.e., (2.34). In the last equality, we used

$$
\frac{1}{\sigma_k} = \frac{\tau_k}{\sigma_k} \frac{1}{\tau_k} = \frac{\tau_1}{\sigma_1} \frac{1}{\tau_k}.
$$

Now, we find an upper bound for δ_{k+1} defined in (B.25).

$$
\begin{aligned}
\delta_{k+1} &= \frac{\rho_k}{2\tau_k} \|x^{k+1} - \tilde{x}^1\|_2^2 + \frac{\rho_k}{2\sigma_k} \|y^{k+1} - \tilde{y}^1\|_2^2 \\
&\leq \frac{\rho_k}{\tau_k} \left(\|\hat{x} - x^{k+1}\|_2^2 + \|\hat{x} - \tilde{x}^1\|_2^2 \right) + \frac{\rho_k}{\sigma_k} \left(\|\hat{y} - y^{k+1}\|_2^2 + \|\hat{y} - \tilde{y}^1\|_2^2 \right) \\
&= \frac{\rho_k}{\tau_k} (R^2 + (1-q) \|\hat{x} - x^{k+1}\|_2^2 + \frac{\tau_k}{\sigma_k} (1/2 - r) \|\hat{y} - y^{k+1}\|_2^2 \\
&\quad + q \|\hat{x} - x^{k+1}\|_2^2 + \frac{\tau_k}{\sigma_k} (r + 1/2) \|\hat{y} - y^{k+1}\|_2^2)
\end{aligned}
$$

$$
\begin{aligned}
&\leq \frac{\rho_k}{\tau_k}[R^2 + \frac{\rho_k}{\gamma_k}\sum_{i=1}^{k}\gamma_i[(1-q)\|\hat{x}-\tilde{x}^{i+1}\|_2^2 + \frac{\tau_k}{\sigma_k}(1/2-r)\|\hat{y}-\tilde{y}^{i+1}\|_2^2 \\
&\quad + q\|\hat{x}-\tilde{x}^{i+1}\|_2^2 + \frac{\tau_k}{\sigma_k}(r+1/2)\|\hat{y}-\tilde{y}^{i+1}\|_2^2]] \\
&\leq \frac{\rho_k}{\tau_k}[R^2 + \frac{\rho_k}{\gamma_k}\sum_{i=1}^{k}\gamma_i[R^2 + q\|\hat{x}-\tilde{x}^{i+1}\|_2^2 + \frac{\tau_k}{\sigma_k}(r+1/2)\|\hat{y}-\tilde{y}^{i+1}\|_2^2]] \\
&\leq \frac{\rho_k}{\tau_k}\left[2 + q\mu^2 + (r+1/2)\nu^2\right]R^2 \\
&= \frac{\rho_k}{\tau_k}\left[2 + \frac{q}{1-q} + \frac{r+1/2}{1/2-r}\right]R^2,
\end{aligned}
$$

i.e., (2.33). In the second and third inequalities, we used

$$
x^{k+1} = \frac{\rho_k}{\gamma_k}\sum_{i=1}^{k}\gamma_i\tilde{x}^{i+1}, \quad y^{k+1} = \frac{\rho_k}{\gamma_k}\sum_{i=1}^{k}\gamma_i\tilde{y}^{i+1}, \quad \text{and} \quad \frac{\rho_k}{\gamma_k}\sum_{i=1}^{k}\gamma_i = 1.
$$

□

Proof of Corollary 3. First check if (2.36) and (2.37) satisfy (2.25) and (2.32). Conditions (2.32) and (2.25a) are trivial to see. To prove (2.25b) and (2.25c):

$$
\begin{aligned}
\frac{1-q}{\tau_k} - L_f\rho_k - \frac{\|A\|_2^2\sigma_k}{r} &\geq L_K\left((1-q)P_2\frac{N}{k} - \frac{a^2k}{rN}\right) \\
&\geq L_K\left((1-q)P_2 - \frac{a^2}{r}\right) \geq 0,
\end{aligned}
$$

and

$$
\begin{aligned}
\frac{1-r}{\sigma_k} - \tau_k\left(2\|K+A\|_2\|K+B\|_2 + \frac{\|B\|_2^2}{q}\right) & \\
\geq \left(\frac{(1-r)N}{k} - \frac{(2cd+b^2/q)k}{P_2N}\right)L_K & \\
\geq \left((1-r)P_2 - (2cd+b^2/q)\right)L_K/P_2 \geq 0. &
\end{aligned}
$$

Condition (2.37) also implies that $\tau_k \leq \sigma_k$.

Note that

$$\begin{aligned}\frac{\rho_N}{\tau_N} &\le \frac{4P_1L_f}{N^2} + \frac{2P_2L_K}{N} \\ L_K^2\rho_N\tau_N &\le \frac{2NL_K^2}{(2P_1L_f + P_2NL_K)(N+1)} \le \frac{2L_K}{P_2N} \\ \rho_N L_K &\le \frac{2L_K}{N}.\end{aligned} \tag{B.30}$$

When we put $\|A\|_2 \le aL_K$, $\|B\|_2 \le bL_K$, $\|K+A\|_2 \le cL_K$, and $\|K+B\|_2 \le dL_K$, $\|v^{k+1}\|_2$ is bounded above by

$$\begin{aligned}\|v^{k+1}\|_2 \le{}& \frac{\rho_k}{\tau_k}\left(\|\hat{x}-\tilde{x}^1\|_2 + \|\hat{y}-\tilde{y}^1\|_2\right) \\ &+ R\left[\frac{\rho_k}{\tau_k}\left(\mu + \frac{\tau_1}{\sigma_1}\nu\right) + 2\rho_k L_K(a\mu + b\nu) + 2\tau_k\rho_k L_K^2 cd\nu\right].\end{aligned}$$

Thus by (B.30), we have

$$\epsilon_{N+1} \le \delta_{N+1} \le \left(\frac{4P_1L_f}{N^2} + \frac{2P_2L_K}{N}\right)\left[2 + \frac{q}{1-q} + \frac{r+1/2}{1/2-r}\right]R^2,$$

which is (2.38), and

$$\begin{aligned}\|v^{N+1}\|_2 \le{}& \frac{4P_1L_f}{N^2}\left[\left(\|\hat{x}-\tilde{x}^1\|_2 + \|\hat{y}-\tilde{y}^1\|_2\right) + R\left(\mu + \frac{\tau_1}{\sigma_1}\nu\right)\right] \\ &+ \frac{L_K}{N}\Big[2P_2\left(\left(\|\hat{x}-\tilde{x}^1\|_2 + \|\hat{y}-\tilde{y}^1\|_2\right) + R\left(\mu + \frac{\tau_1}{\sigma_1}\nu\right)\right) \\ &+ 4R(a\mu + b\nu) + \frac{4Rcd\nu}{P_2}\Big],\end{aligned}$$

which is (2.39). □

B.3 Optimal stochastic acceleration

We obtain a bound similar to Lemma 4 first. The following lemma provides an upper bound on $\rho_k^{-1}\gamma_k\mathcal{G}(z^k, z)$.

Lemma 6. *Assume that $z^k = (x^k, y^k)$ is the iterates generated by the iteration* (2.43)*. Also assume that the parameters satisfy* (2.25a) (2.32)*, and* (2.45)*. Then for any $z \in \mathcal{Z}$, we have*

$$\begin{aligned}\rho_k^{-1}\gamma_k\mathcal{G}(z^{k+1},z) \leq\ & \mathcal{D}_k(z,\tilde{z}^{[k]}) - \gamma_k\langle \tilde{x}^{k+1}-x, B^T(\tilde{y}^{k+1}-\tilde{y}^k)\rangle \\ & + \gamma_k\langle A(\tilde{x}^{k+1}-\tilde{x}^k), \tilde{y}^{k+1}-y\rangle \\ & + \tau_k\gamma_k\langle (K+B)^T(\tilde{y}^{k+1}-\tilde{y}^k), (K+A)^T(\tilde{y}^{k+1}-y)\rangle \\ & - \gamma_k\left(\frac{s-q}{2\tau_k} - \frac{\rho_k L_f}{2}\right)\|\tilde{x}^{k+1}-\tilde{x}^k\|_2^2 \\ & - \gamma_k\left(\frac{t-r}{2\sigma_k} - \frac{\|K+A\|_2\|K+B\|_2\tau_{k-1}}{2}\right)\|\tilde{y}^{k+1}-\tilde{y}^k\|_2^2 \\ & + \sum_{i=1}^{k}\Lambda_i(z),\end{aligned} \tag{B.31}$$

where γ_k *and* $\mathcal{D}(z,\tilde{z}^{[k]})$ *are defined in* (B.18) *and* (B.19)*, respectively, and*

$$\Lambda_i(z) := -\frac{(1-s)\gamma_i}{2\tau_i}\|\tilde{x}^{i+1}-x^i\|_2^2 - \frac{(1-t)\gamma_i}{2\sigma_i}\|\tilde{y}^{i+1}-y^i\|_2^2 - \gamma_i\langle\Delta^i, z^{i+1}-z\rangle. \tag{B.32}$$

Proof. Analogous to the proof of Lemma 4, except for that we start with

$$\begin{aligned}&\langle -\tilde{u}_{k+1}, \tilde{y}^{k+1}-y\rangle + h^*(\tilde{y}^{k+1}) - h^*(y) \\ &\qquad \leq \frac{1}{2\sigma_k}\|y-\tilde{y}^k\|_2^2 - \frac{1}{2\sigma_k}\|\tilde{y}^{k+1}-\tilde{y}^k\|_2^2 - \frac{1}{2\sigma_k}\|y-\tilde{y}^{k+1}\|_2^2\end{aligned}$$

$$\begin{aligned}&\langle \hat{\mathcal{F}}(x_{md}^k), \tilde{x}^{k+1}-x\rangle + \langle \tilde{x}^{k+1}-x, \tilde{v}_{k+1}\rangle + g(\tilde{x}^{k+1}) - g(x) \\ &\qquad \leq \frac{1}{2\tau_k}\|x-\tilde{x}^k\|_2^2 - \frac{1}{2\tau_k}\|\tilde{x}^{k+1}-\tilde{x}^k\|_2^2 - \frac{1}{2\tau_k}\|x-\tilde{x}^{k+1}\|_2^2.\end{aligned}$$

□

Now we define $\Delta_{x,f}^k := \hat{\mathcal{F}}(x_{md}^k) - \nabla f(x_{md}^k)$, $\Delta_{x,K}^k := \tilde{v}_{k+1} - \tilde{v}_{k+1,o}$, $\Delta_y^k := -\tilde{u}_{k+1} + \tilde{u}_{k+1,o}$, and $\Delta^k := (\Delta_x^k, \Delta_y^k)$, where $\tilde{u}_{k+1,o}$ and $\tilde{v}_{k+1,o}$ is the result from (2.23) calculated with the recent iterates $(\tilde{x}^{k+1}, \tilde{y}^{k+1})$, $(\tilde{x}^k, \tilde{y}^k)$ and $(\tilde{x}^{k-1}, \tilde{y}^{k-1})$ from (2.43).

We need the following lemmas.

Lemma 7 (Lemma 4.5, Chen et al., 2011)**.** *Let τ_i, σ_i, and $\gamma_i > 0$. For any $\tilde{z}^1 \in Z$, define $\tilde{z}_v^1 = \tilde{x}^1$ and*

$$z_v^{i+1} = \underset{z=(x,y)\in Z}{\arg\min} -\tau_i\langle\Delta_x^i, x\rangle - \sigma_i\langle\Delta_y^i, y\rangle + \frac{1}{2}\|z - z_v^i\|_2^2, \tag{B.33}$$

then

$$\sum_{i=1}^{k}\gamma_i\langle-\Delta^i, z_v^i - z\rangle \le \mathcal{D}_k(z, \tilde{z}_v^{[k]}) + \sum_{i=1}^{k}\frac{\tau_i\gamma_i}{2}\|\Delta_x^i\|_2^2 + \sum_{i=1}^{k}\frac{\sigma_i\gamma_i}{2}\|\Delta_y^i\|_2^2, \tag{B.34}$$

where $\tilde{z}_v^{[k]} := \{z_v^i\}_{i=1}^k$.

Lemma 8. *The following holds for $\mathbb{E}[\|\Delta_{x,f}^i\|_2^2]$, $\mathbb{E}[\|\Delta_{x,K}^i\|_2^2]$, and $\mathbb{E}[\|\Delta_y^i\|_2^2]$.*

$$\mathbb{E}[\|\Delta_{x,f}^i\|_2^2] \le \chi_{x,f}^2 \tag{B.35a}$$
$$\mathbb{E}[\|\Delta_{x,K}^i\|_2^2] \le \chi_{x,K}^2 + \chi_B^2 \tag{B.35b}$$
$$\mathbb{E}[\|\Delta_y^i\|_2^2] \le \chi_y^2 + \chi_A^2 + \tau_i^2\|K + A\|_2^2(\chi_x^2 + \chi_B^2). \tag{B.35c}$$

If $A = -K$ and $B = bK$, after rearranging terms in (2.43)*, we have*

$$\begin{aligned}\mathbb{E}[\|\Delta_{x,f}^i\|_2^2] &\le \chi_{x,f}^2\\ \mathbb{E}[\|\Delta_{x,K}^i\|_2^2] &\le \chi_{x,K}^2\\ \mathbb{E}[\|\Delta_y^i\|_2^2] &\le \chi_y^2.\end{aligned} \tag{B.36}$$

Proof. (B.35a) is trivial, by (2.42). Note that

$$\begin{aligned}\Delta_{x,K}^i = \hat{\mathcal{K}}_y(\tilde{y}^{k+1}) - K^T\tilde{y}^{k+1} + \hat{\mathcal{B}}(\tilde{y}^{k+1} - \tilde{y}^k - \theta_k(\tilde{y}^k - \tilde{y}^{k-1}))\\ - B^T(\tilde{y}^{k+1} - \tilde{y}^k - \theta_k(\tilde{y}^k - \tilde{y}^{k-1})),\end{aligned}$$

and as separate calls for the stochastic oracle are independent, we obtain (B.35b). If we define

$$\Delta_v^i := \hat{\mathcal{F}}(\tilde{x}^k) - \nabla f(\tilde{x}^k) + \bar{v}_k - \bar{v}_{k,o},$$

then one may easily check that

$$\mathbb{E}[\|\Delta_v^i\|_2^2] \le \chi_{x,f}^2 + \chi_{x,K}^2 + \chi_B^2.$$

Then we have:

$$\begin{aligned}\Delta_y^i = &\hat{\mathcal{K}}_x(\tilde{x}^k - \tau_k(\nabla f(\tilde{x}^k) + \bar{v}_{k,0} + \Delta_v^i)) - \hat{\mathcal{A}}(\theta_k(\tilde{x}^k - \tilde{x}^{k-1}) + \tau_k(\nabla f(\tilde{x}^k) + \bar{v}_{k,0} + \Delta_v^i))\\ &- K(\tilde{x}^k - \tau_k(\nabla f(\tilde{x}^k) + \bar{v}_{k,0} + \Delta_v^i)) + A(\theta_k(\tilde{x}^k - \tilde{x}^{k-1}) + \tau_k(\nabla f(\tilde{x}^k) + \bar{v}_{k,0} + \Delta_v^i))\\ &- \tau_k(K + A)\Delta_v^i,\end{aligned}$$

thus

$$\mathbb{E}[\|\Delta_y^i\|_2^2] \le \chi_y^2 + \chi_A^2 + \tau_k^2\|K + A\|_2^2(\chi_x^2 + \chi_B^2).$$

When $A = -K$ and $B = bK$, we may rearrange (2.43) to include only one call to either $\hat{\mathcal{K}}_x$ or $\hat{\mathcal{K}}_y$, as

$$\begin{aligned}\tilde{u}_{k+1} &= \hat{\mathcal{K}}_x(\tilde{x}^k + \theta_k(\tilde{x}^k - \tilde{x}^{k-1}))\\ \tilde{v}_{k+1} &= \hat{\mathcal{K}}_y(\tilde{y}^{k+1} + b((\tilde{y}^{k+1} - \tilde{y}^k) - \theta_k(\tilde{y}^k - \tilde{y}^{k-1}))).\end{aligned}$$

Then using the approach similar to above, we may obtain (B.36). □

Proof of Theorem 5. First we use the bounds in (B.23) to obtain

$$\rho_k^{-1}\gamma_k\mathcal{G}(z^{k+1}, z) \le \frac{\gamma_k}{\tau_k}\Omega_X^2 + \frac{\gamma_k}{\sigma_k}\Omega_Y^2 + \sum_{i=1}^k \Lambda_i(z),$$

following the steps to prove Theorem 3. Then by the definition of $\Lambda_i(z)$, we have

$$\begin{aligned}\Lambda_i(z) &= -\frac{(1-s)\gamma_i}{2\tau_i}\|\tilde{x}^{i+1} - x^i\|_2^2 - \frac{(1-t)\gamma_i}{2\sigma_i}\|\tilde{y}^{i+1} - y^i\|_2^2 + \gamma_i\langle\Delta^i, z - z^{i+1}\rangle\\ &= -\frac{(1-s)\gamma_i}{2\tau_i}\|\tilde{x}^{i+1} - x^i\|_2^2 - \frac{(1-t)\gamma_i}{2\sigma_i}\|\tilde{y}^{i+1} - y^i\|_2^2 + \gamma_i\langle\Delta^i, z^i - z^{i+1}\rangle\\ &\quad + \gamma_i\langle\Delta^i, z - z^i\rangle\\ &\le \frac{\tau_i\gamma_i}{2(1-s)}\|\Delta_x^i\|_2^2 + \frac{\sigma_i\gamma_i}{2(1-t)}\|\Delta_y^i\|_2^2 + \gamma_i\langle\Delta^i, z - z^i\rangle,\end{aligned}$$

where the last line is due to Young's inequality. By this result and Lemma 7, we have

$$\begin{aligned}\sum_{i=1}^k \Lambda_i(z) &\le \sum_{i=1}^k \left[\frac{\tau_i\gamma_i}{2(1-s)}\|\Delta_x^i\|_2^2 + \frac{\sigma_i\gamma_i}{2(1-t)}\|\Delta_y^i\|_2^2 + \gamma_i\langle\Delta^i, z_v^i - z^i\rangle + \gamma_i\langle-\Delta^i, z_v^i - z\rangle\right]\\ &\le \mathcal{D}_k(z, \tilde{z}_v^{[k]}) + \frac{1}{2}\sum_{i=1}^k \left[\frac{(2-s)\tau_i\gamma_i}{1-s}\|\Delta_x^i\|_2^2 + \frac{(2-t)\sigma_i\gamma_i}{1-t}\|\Delta_y^i\|_2^2 + \gamma_i\langle\Delta^i, z_v^i - z^i\rangle\right].\end{aligned} \tag{B.37}$$

Let us define U_k as

$$U_k := \frac{1}{2}\sum_{i=1}^{k}\left[\frac{(2-s)\tau_i\gamma_i}{1-s}\|\Delta_x^i\|_2^2 + \frac{(2-t)\sigma_i\gamma_i}{1-t}\|\Delta_y^i\|_2^2 + \gamma_i\langle\Delta^i, z_v^i - z^i\rangle\right] \quad \text{(B.38)}$$

for later use.

Note that Δ^i and z^i are independent by the assumptions of stochastic oracle. By this fact and Lemma 8,

$$\mathbb{E}[U_k] \le \frac{1}{2}\sum_{i=1}^{k}\left[\frac{(2-s)\tau_i\gamma_i(\chi_x^2+\chi_B^2)}{1-s} + \frac{(2-t)\sigma_i\gamma_i(\chi_y^2+\chi_A^2+\tau_k^2\|K+A\|_2^2(\chi_x^2+\chi_B^2))}{1-t}\right]. \quad \text{(B.39)}$$

Similar to (B.22), $\mathcal{D}_k(z, \tilde{z}_v^{[k]}) \le \frac{\Omega_X^2\gamma_k}{\tau_k} + \frac{\Omega_Y^2\gamma_k}{\sigma_k}$. Thus we have:

$$\mathbb{E}[\rho_k^{-1}\gamma_k\mathcal{G}^\star(z^{k+1})] \le \frac{2\gamma_k}{\tau_k}\Omega_X^2 + \frac{2\gamma_k}{\sigma_k}\Omega_Y^2 + \mathbb{E}[U_k].$$

The above relation along with (B.39) implies the condition (a).

Proof of part (b) is analogous to the proof of Theorem 3.1 in Chen et al. (2014). This uses a large-deviation theorem for martingale-difference sequence. □

Proof of Corollary 4. First we check (2.45a) and (2.45b).

$$\frac{s-q}{\tau_k} - \rho_k L_f - \frac{\|A\|_2^2\sigma_k}{r} \ge \frac{L_K\Omega_Y}{\Omega_X}\left((s-q)P_2 - \frac{1}{r}\right) \ge 0$$

$$\frac{t-r}{\sigma_k} - \tau_k\left(2\|K+A\|_2\|K+B\|_2 + \frac{\|B\|_2^2}{q}\right) \ge \left((t-r)R - \frac{b^2/q}{P_2}\right)\frac{\Omega_X}{\Omega_Y} \ge 0,$$

by (2.52). Note that $\gamma_k = k$, $\sum_{i=1}^k\sqrt{i} \le \int_1^{k+1}\sqrt{u}du \le \frac{2}{3}(k+1)^{3/2} \le \frac{2\sqrt{2}}{3}(k+1)\sqrt{k}$, so

$$\frac{1}{\gamma_k}\sum_{i=1}^{k}\tau_i\gamma_i \le \frac{\Omega_X}{kP_3\chi_x}\sum_{i=1}^{k}\sqrt{i} \le \frac{2\sqrt{2}\Omega_X(k+1)\sqrt{k}}{3kP_3\chi_x} \text{ and}$$
$$\frac{1}{\gamma_k}\sum_{i=1}^{k}\sigma_i\gamma_i \le \frac{\Omega_Y}{kP_3\chi_y}\sum_{i=1}^{k}\sqrt{i} \le \frac{2\sqrt{2}\Omega_Y(k+1)\sqrt{k}}{3kP_3\chi_y},$$

which in turn implies

$$\begin{aligned}\mathcal{C}_0(k) \le \frac{2}{k+1}\Bigg[&\frac{2(2P_1L_f\Omega_X + P_2L_K\Omega_Y k + P_3\chi_x k^{3/2})}{k}\Omega_X \\ &+ 2L_K\Omega_X\Omega_Y + P_3\chi_y\Omega_Y k^{1/2} \\ &+ \frac{\sqrt{2}(2-s)\Omega_X\chi_x^2(k+1)\sqrt{k}}{3(1-s)P_3\chi_x k} + \frac{\sqrt{2}(2-t)\Omega_Y\chi_y^2(k+1)\sqrt{k}}{3(1-t)P_3\chi_y k}\Bigg] \\ \le\ &\frac{8P_1L_f\Omega_X^2}{k(k+1)} + \frac{4L_K\Omega_X\Omega_Y(P_2+1)}{k} + \frac{4P_3(\chi_x\Omega_X + 4\chi_y\Omega_Y)}{\sqrt{k}} \\ &+ \frac{\sqrt{2}(2-s)\Omega_X\chi_x}{3(1-s)\sqrt{k}} + \frac{\sqrt{2}(2-t)\Omega_Y\chi_y}{3(1-t)\sqrt{k}}\end{aligned}$$

and

$$\begin{aligned}\mathcal{C}_1(k) \le \frac{2}{k+1}\Bigg[&\frac{\sqrt{2}}{k}\left(\sqrt{2}\chi_x\Omega_x + \chi_y\Omega_y\right)\sqrt{\sum_{i=1}^{k} i^2} \\ &+ \frac{\sqrt{2}(2-s)\Omega_X\chi_x^2(k+1)\sqrt{k}}{3(1-s)P_3\chi_x k} + \frac{\sqrt{2}(2-t)\Omega_Y\chi_y^2(k+1)\sqrt{k}}{3(1-t)P_3\chi_y k}\Bigg] \\ \le\ &\left(4 + \frac{\sqrt{2}(2-s)}{3(1-s)}\right)\frac{\Omega_X\chi_x}{\sqrt{k}} + \left(2\sqrt{2} + \frac{\sqrt{2}(2-s)}{3(1-s)}\right)\frac{\Omega_Y\chi_y}{\sqrt{k}}.\end{aligned}$$

□

We need the following lemma to prove Theorem 6.

Lemma 9. *For a saddle point $\hat{z} = (\hat{x}, \hat{y})$ of (2.2), and the parameters ρ_k, θ_k, τ_k, and σ_k satisfy (2.25a), (2.32), and (2.45), then*

$$\begin{aligned}&(1-q)\|\hat{x} - \tilde{x}^{k+1}\|_2^2 + \|\hat{x} - \tilde{x}_v^{k+1}\|_2^2 + \frac{\tau_k(1/2-r)}{\sigma_k}\|\hat{y} - \tilde{y}^{k+1}\|_2^2 + \frac{\tau_k}{\sigma_k}\|\hat{y} - \tilde{y}_v^{k+1}\|_2^2 \\ &\le 2\|\hat{x} - \tilde{x}^1\|_2^2 + \frac{2\tau_k}{\sigma_k}\|\hat{y} - \tilde{y}^1\|_2^2 + \frac{2\tau_k}{\gamma_k}U_k, \qquad \text{(B.40)}\end{aligned}$$

where $(\tilde{x}_v^{k+1}, \tilde{y}_v^{k+1})$ is defined in (B.33), and U_k is defined by (B.38).

Furthermore,

$$\tilde{\mathcal{G}}(z^{k+1}, v^{k+1}) \le \frac{\rho_k}{\tau_k}\|x^{k+1} - \tilde{x}^1\|_2^2 + \frac{\rho_k}{\sigma_k}\|y^{k+1} - \tilde{y}^1\|_2^2 + \frac{\rho_k}{\gamma_k}U_k := \delta'_{k+1}, \quad \text{(B.41)}$$

for $k \geq 1$, where

$$v_{k+1} = \rho_k \bigg(\frac{1}{\tau_k}(2\tilde{x}^1 - \tilde{x}^{k+1} - \tilde{x}_v^{k+1}) - B^T(\tilde{y}^{k+1} - \tilde{y}^k), \\ \frac{1}{\sigma_k}(2\tilde{y}^1 - \tilde{y}^{k+1} - \tilde{y}_v^{k+1}) + A(\tilde{x}^{k+1} - \tilde{x}^k) + \tau_k(K+A)(K+B)^T(\tilde{y}^{k+1} - \tilde{y}^k) \bigg).$$

Proof. By applying the bounds (B.28) and (B.37) to (B.31), we obtain:

$$\rho_k^{-1}\gamma_k \mathcal{G}(z^{k+1}, z) \leq \bar{\mathcal{D}}_k(z, \tilde{z}^{[k]}) + \frac{q\gamma_k}{2\tau_k}\|x - \tilde{x}^{k+1}\|_2^2 + \frac{(r+1/2)\gamma_k}{2\sigma_k}\|y - \tilde{y}^{k+1}\|_2^2 \\ + \bar{\mathcal{D}}_k(z, \tilde{z}_v^{[k]}) + U_k,$$

where

$$\bar{\mathcal{D}}_k(z, \tilde{z}^{[k]}) = \frac{\gamma_k}{2\tau_k}(\|x - \tilde{x}_1\|_2^2 - \|x - \tilde{x}_{k+1}\|_2^2) + \frac{\gamma_k}{2\sigma_k}(\|y - \tilde{y}_1\|_2^2 - \|y - \tilde{y}_{k+1}\|_2^2).$$

Letting $z = \hat{z}$ and using $\mathcal{G}(z^{k+1}, \hat{z}) \geq 0$ leads to (B.40). If we only use (B.37) on (B.31), we get:

$$\begin{aligned} \rho_k^{-1}\gamma_k \mathcal{G}(z^{k+1}, z) \leq{}& \bar{\mathcal{D}}_k(z, \tilde{z}^{[k]}) - \gamma_k\langle \tilde{x}^{k+1} - x, B^T(\tilde{y}^{k+1} - \tilde{y}^k)\rangle \\ &+ \gamma_k\langle A(\tilde{x}^{k+1} - \tilde{x}^k), \tilde{y}^{k+1} - y\rangle \\ &+ \tau_k\gamma_k\langle (K+B)^T(\tilde{y}^{k+1} - \tilde{y}^k), (K+A)^T(\tilde{y}^{k+1} - y)\rangle \\ &+ \bar{\mathcal{D}}_k(z, \tilde{z}_v^{[k]}) + U_k. \end{aligned}$$

Applying (B.29) and following the steps of Lemma 5 results in (B.41). □

Proof of Theorem 6. Note that (B.39) holds by Lemma 8. By the definition of S in (2.59) and (B.39), we have

$$\mathbb{E}[U_k] \leq \frac{\gamma_k}{2\tau_k}S^2.$$

By the above, (B.40), and (2.35), we have

$$\mathbb{E}[\|\hat{x} - \tilde{x}^{k+1}\|_2^2] \leq \frac{2R^2 + S^2}{1-q} \text{ and } \mathbb{E}[\|\hat{y} - \tilde{y}^{k+1}\|_2^2] \leq \frac{(2R^2 + S^2)\sigma_1}{\tau_1(1/2 - r)}.$$

By Jensen's inequality, this leads to

$$\mathbb{E}[\|\hat{x} - \tilde{x}^{k+1}\|_2] \leq \sqrt{\frac{2R^2 + S^2}{1-q}} \text{ and } \mathbb{E}[\|\hat{y} - \tilde{y}^{k+1}\|_2] \leq \sqrt{\frac{(2R^2 + S^2)\sigma_1}{\tau_1(1/2 - r)}}.$$

Similarly, we have

$$\mathbb{E}[\|\hat{x} - \tilde{x}_v^{k+1}\|_2] \leq \sqrt{2R^2 + S^2} \text{ and } \mathbb{E}[\|\hat{y} - \tilde{y}_v^{k+1}\|_2] \leq \sqrt{\frac{(2R^2 + S^2)\sigma_1}{\tau_1}}.$$

Thus

$$\begin{aligned}
\mathbb{E}[\|v^{k+1}\|_2] \leq \rho_k \, \mathbb{E}[&\frac{1}{\tau_k}(2\|\hat{x} - \tilde{x}^1\|_2 + \|\hat{x} - \tilde{x}^{k+1}\|_2 + \|\hat{x} - \tilde{x}_v^{k+1}\|_2) \\
&+ \frac{1}{\sigma_k}(2\|\hat{y} - \tilde{y}^1\|_2 + \|\hat{y} - \tilde{y}^{k+1}\|_2 + \|\hat{y} - \tilde{y}_v^{k+1}\|_2) \\
&+ \|A\|_2(\|\hat{x} - \tilde{x}^{k+1}\|_2 + \|\hat{x} - \tilde{x}^k\|_2) \\
&+ (\|B\|_2 + \|K + A\|_2\|K + B\|_2\tau_k)(\|\hat{y} - \tilde{y}^{k+1}\|_2 + \|\hat{y} - \tilde{y}^k\|_2)] \\
\leq &\frac{2\rho_k\|\hat{x} - \tilde{x}^1\|_2}{\tau_k} + \frac{2\rho_k\|\hat{y} - \tilde{y}^1\|_2}{\sigma_k} \\
&+ \sqrt{2R^2 + S^2}\left[\frac{\rho_k}{\tau_k}(1 + \mu') + \frac{\rho_k}{\sigma_k}\sqrt{\frac{\sigma_1}{\tau_1}}(1 + \nu')\right. \\
&+ \rho_k(2\|A\|_2\mu' + 2\|B\|_2\nu'\sqrt{\frac{\sigma_1}{\tau_1}}) \\
&\left.+ 2\rho_k\tau_k\|K + A\|_2\|K + B\|_2\nu'\sqrt{\frac{\sigma_1}{\tau_1}}\right]
\end{aligned}$$

where $\mu' = 1/\sqrt{1-q}$ and $\nu' = 1/\sqrt{1/2 - r}$. Now we find an upper bound of $\mathbb{E}[\delta'_{k+1}]$.

$$\begin{aligned}
\mathbb{E}[\delta'_{k+1}] &\leq \mathbb{E}\left[\frac{2\rho_k}{\tau_k}(\|\hat{x} - x^{k+1}\|_2^2 + \|\hat{x} - \tilde{x}^1\|_2^2) + \frac{2\rho_k}{\sigma_k}(\|\hat{y} - y^{k+1}\|_2^2 + \|\hat{y} - \tilde{y}^1\|_2^2)\right] + \frac{\rho_k}{2\tau_k}S^2 \\
&= \frac{\rho_k}{\tau_k}\,\mathbb{E}\left[(2R^2 + 2(1-q)\|\hat{x} - x^{k+1}\|_2^2 + \frac{2\tau_k(1/2 - r)}{\sigma_k}\|\hat{y} - y^{k+1}\|_2^2)\right. \\
&\quad \left.+ 2q\|\hat{x} - x^{k+1}\|_2^2 + \frac{2\tau_k(r + 1/2)}{\sigma_k}\|\hat{y} - y^{k+1}\|_2^2\right] + \frac{\rho_k}{2\tau_k}S^2 \\
&\leq \frac{\rho_k}{\tau_k}\left[2R^2 + \frac{2\rho_k}{\gamma_k}\sum_{i=1}^{k}\gamma_i[(2R^2 + S^2) + q\mu'^2(2R^2 + S^2) + (r + 1/2)\nu'^2(2R^2 + S^2)] + \frac{S^2}{2}\right] \\
&= \frac{\rho_k}{\tau_k}\left[6R^2 + \frac{5}{2}S^2 + \frac{2q}{1-q}(2R^2 + S^2) + \frac{2(r + 1/2)}{1/2 - r}(2R^2 + S^2)\right] \\
&= \frac{\rho_k}{\tau_k}\left[\left(6 + \frac{4q}{1-q} + \frac{4(r + 1/2)}{1/2 - r}\right)R^2 + \left(\frac{5}{2} + \frac{2q}{1-q} + \frac{2(r + 1/2)}{1/2 - r}\right)S^2\right]
\end{aligned}$$

□

Proof of Corollary 5. First we check (2.45a) and (2.45b):

$$\frac{s-q}{\tau_k} - \rho_k L_f - \frac{\|A\|_2^2 \sigma_k}{r} \geq L_K \left((s-q)P_2 - \frac{1}{r} \right) \geq 0,$$

$$\frac{t-r}{\sigma_k} - \tau_k \frac{b^2 L_K^2}{q} \geq L_K \left((t-r) - \frac{b^2}{qP_2} \right) \geq 0,$$

by (2.61).

Now, if we put $\eta = 2P_1 L_f + P_2 L_K (N-1) + P_3 N\sqrt{N-1}\chi'$,

$$\begin{aligned} S &\leq \sqrt{\sum_{i=1}^{N-1} \frac{(2-s)\chi_x^2 i^2}{(1-s)\tau^2} + \sum_{i=1}^{N-1} \frac{(2-t)\chi_y^2 i^2}{(1-t)\eta^2}} \\ &\leq \sqrt{\frac{N^2(N-1)}{3\eta^2} \left(\frac{(2-s)\chi_x^2}{1-s} + \frac{(2-t)\chi_y^2}{1-t} \right)} = \frac{\chi' N\sqrt{N-1}}{\sqrt{3}\eta} \\ &\leq \frac{\chi' N\sqrt{N-1}}{\sqrt{3}N\sqrt{N-1}\chi'/\tilde{R}} = \frac{\tilde{R}}{\sqrt{3}}. \end{aligned} \tag{B.42}$$

Thus ϵ_N is bounded above by

$$\epsilon_N \leq \frac{\rho_{N-1}}{\tau_{N-1}} (\zeta R^2 + \xi S^2) \leq \frac{\rho_{N-1}}{\tau_{N-1}} (\zeta R^2 + \xi \frac{\tilde{R}^2}{3}),$$

where $\zeta = 6 + \frac{4q}{1-q} + \frac{4(r+1/2)}{1/2-r}$ and $\xi = \frac{5}{2} + \frac{2q}{1-q} + \frac{2(r+1/2)}{1/2-r}$.

Note that

$$\frac{\rho_{N-1}}{\tau_{N-1}} \|\hat{x} - \tilde{x}^1\| \leq \frac{\rho_{N-1}}{\tau_{N-1}} R, \quad \frac{\rho_{N-1}}{\sigma_{N-1}} \|\hat{y} - \tilde{y}^1\| \leq \frac{\rho_{N-1}}{\tau_{N-1}} \sqrt{\frac{\sigma_1}{\tau_1}} R$$

and that

$$\begin{aligned} \rho_{N-1} L_K &\leq \frac{2L_K}{N}, \\ \frac{\rho_{N-1}}{\tau_{N-1}} &\leq \frac{2\eta}{N(N-1)} = \frac{4P_1 L_f + 2P_2 L_K (N-1) + 2N\sqrt{N-1}\chi'/\tilde{R}}{N(N-1)} \\ &= \frac{4P_1 L_f}{N(N-1)} + \frac{2P_2 L_K}{N} + \frac{2\chi'/\tilde{R}}{\sqrt{N-1}} \end{aligned} \tag{B.43}$$

Thus

$$\begin{aligned}\epsilon_N &\leq \frac{\rho_{N-1}}{\tau_{N-1}}(\zeta R^2 + \xi S^2) \\ &\leq \left(\frac{4P_1L_f}{N(N-1)} + \frac{2P_2L_K}{N} + \frac{2\chi'/\tilde{R}}{\sqrt{N-1}}\right)\left(\zeta R^2 + \frac{\xi\tilde{R}^2}{3}\right).\end{aligned}$$

Now note that $\sqrt{2R^2+S^2} \leq \sqrt{2}R + S$.

$$\begin{aligned}\mathbb{E}[\|v^N\|_2] &\leq \frac{2\rho_{N-1}}{\tau_{N-1}}\left(1+\sqrt{\frac{\sigma_1}{\tau_1}}\right)R \\ &\quad + (\sqrt{2}R+S)\left[\frac{\rho_{N-1}}{\tau_{N-1}}\left(1+\mu'+\sqrt{\frac{\sigma_1}{\tau_1}}(1+\nu')\right) + \rho_{N-1}L_K(2\mu'+2b\nu')\right] \\ &= \frac{\rho_{N-1}}{\tau_{N-1}}\left(2\left(1+\sqrt{\frac{\sigma_1}{\tau_1}}\right)R + (\sqrt{2}R+S)\left(1+\mu'+\sqrt{\frac{\sigma_1}{\tau_1}}(1+\nu')\right)\right) \\ &\quad + \rho_{N-1}L_K(\sqrt{2}R+S)\left(2\mu'+2b\nu'\sqrt{\frac{\sigma_1}{\tau_1}}\right) \\ &\leq \left(\frac{4P_1L_f}{N(N-1)} + \frac{2P_2L_K}{N} + \frac{2\chi'/\tilde{R}}{\sqrt{N-1}}\right)\times \\ &\quad \left(2R\left(1+\sqrt{\frac{\sigma_1}{\tau_1}}\right) + \left(\sqrt{2}R + \frac{\tilde{R}}{\sqrt{3}}\right)(2+\mu'+\nu')\right) \\ &\quad + \frac{2L_K}{N}(\sqrt{2}R+\tilde{R}/\sqrt{3})\left(2\mu'+2b\nu'\sqrt{\frac{\sigma_1}{\tau_1}}\right),\end{aligned}$$

thus we obtain the desired order for both ϵ_N and $\mathbb{E}[\|v_N\|]$. □

Appendix C

AWS EC2 and ParallelCluster

We used AWS Elastic Compute Cloud (EC2) via CfnCluster throughout our multi CPU-node experiments, which is updated to ParallelCluster after we had completed the experiments. In this section, we instruct how to use ParallelCluster via Amazon Web Services. This section is structured into three parts: setting up AWS account and how to configure and run a job on ParallelCluster. We refer the readers to the official documentation[1] and an AWS whitepaper[2] for further details.

C.1 Overview

A virtual cluster created by ParallelCluster consists of two types of *instances* in EC2: a *master* node and multiple *worker* instances. The master instance manages *jobs* through a queue on a *job scheduler* and several AWS services such as *Simple Queue Service* and *Auto Scaling Group*. When a virtual cluster is

[1] https://docs.aws.amazon.com/parallelcluster/index.html

[2] https://d1.awsstatic.com/Projects/P4114756/deploy-elastic-hpc-cluster_project.a12a8c61339522e21262da10a6b43a3678099220.pdf

created, the *shared file system*. The software necessary for the jobs are installed in this file system, and a script to set up the environment variables for the tools is utilized. While the master instance does not directly take part in the actual computation, the speed of network on the shared file system depends on the instance type of the master instance. If the jobs depend on the shared dataset, the master instance has to allow fast enough network speed. The actual computation is performed on the worker instances. Each worker has access to the shared file system where the necessary tools and data reside. The network speed between workers depends on the worker instance type.

C.2 Glossary

We briefly introduce some of the key concepts regarding the AWS and cluster computing in this subsection.

Some of the basic concepts from AWS are shown below:

- Instance: a virtual computer on AWS EC2. There are various types of instances determines number of cores, memory size, network speed, etc. `c5.18xlarge` is prominently utilized in our experiments.[3]
- Region: a region, e.g., North Virginia, Ohio, North California, Oregon, Hong Kong, Seoul, Tokyo is completely independent from other regions, and data transfer between regions are charged.
- Availablity zone: there are a handful of availability zones in each region. Each availability zone is isolated, but availability zones in the same region is interconnected with a low-latency network. Note that a virtual cluster created by ParallelCluster is tied to a single availity zone.

[3]See `https://aws.amazon.com/en/ec2/instance-types/` for the full list of types of instances.

Listed below are some, but not all, of the AWS services involved in ParallelCluster. They are all managed automatically through ParallelCluster and can be modified through the AWS console.

- Elastic Compute Cloud (EC2): the core service of AWS that allows users to rent virtual computers. There are three methods of payment available:
  - On-demand: hourly charged, without risk of interruption.
  - Spot: bid-based charging. Serviced at up to 70%-discounted rate, but is interrupted if the price goes higher than the bid price.
  - Reserved: one-time payment at discounted rate.
- Elastic Block Store (EBS): persistent block storage volume for EC2 instances, e.g. a solid-state drive (SSD). In ParallelCluster, each instance is started with a root EBS volume exclusive to each instance.
- CloudFormation: An interface that describes and provisions the cloud resources.
- Simple Queue Service: the actual job queue is served through message passing between EC2 instances.
- CloudWatch: monitors and manages the cloud.
- Auto Scaling Group: a collection of EC2 instances with similar characteristics. The number of instances is automatically scaled based on criteria defined over CloudWatch.
- Identity and Access Management (IAM): An IAM user is an "entity that [one] creates in AWS to represent the person or application that uses it

to interact with AWS."[4] Each IAM user is granted certain permissions determined by the root user. As there are many services involved in ParallelCluster, it is recommended to use an IAM user with full permission.

- Virtual Private Cloud (VPC): a VPC is a dedicated virtual network exclusive to the user, isolated from any other VPCs, which spans all the availability zones in one region. A subnet is a subnetwork in VPC exclusive to a single availity zone.[5]

- Security Group (SG): A security group acts as a "virtural firewall that controls the traffic for one or more instances."[6]

Here are some of the concepts related to cluster computing:

- Shared file system: for multiple instances to work on the same data, it is convenient to have a file system that can be accessed by all the instances involved. In ParallelCluster, it is implemented as an additional EBS volume attached to the master instance. All the worker instances can access this volume, and its speed of network depends on the instance type of the master instance.

- Job: a unit of execution. defined by either a single command or a job script.

- Queue: a data structure containing jobs to run. Jobs in a queue is managed and prioritized by a job scheduler.

- Master: an instance that manages the job scheduler.

[4] https://docs.aws.amazon.com/IAM/latest/UserGuide/id_users.html
[5] https://docs.aws.amazon.com/vpc/latest/userguide/VPC_Subnets.html
[6] https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using-network-security.html

- Worker: an instance that executes the jobs.
- Job scheduler: an application program that controls the execution of jobs over a cluster. e.g. Sun Grid Engine, Torque, Slurm, etc. The Sun Grid Engine (SGE) was used for our experiments.

Several SGE commands are as follows:

- `qsub`: submits a job to the job queue
- `qdel`: removes a job on the job queue
- `qstat`: shows the current status of the queue
- `qhost`: shows the current list of workers

C.3 Prerequisites

The following are needed before we proceed. Most of these might be considered the first steps to use AWS.

- Access keys with administrative privileges: Access keys are credentials for IAM users and root users. They consist of access key ID (analogous to username) and secret access key (analogous to passwords). They should be kept confidential. It is recommended to create a temporary IAM user with administrative privilage and create an access key ID and a secret access key for the IAM user. They can be created in the AWS console (or the IAM console for an IAM user).[7]
- A VPC and a subnet: A VPC for each region and a subnet for each availability zone is created by default. One may use these default VPC and subnet or newly-created ones.

[7] `https://docs.aws.amazon.com/IAM/latest/UserGuide/id_credentials_access-keys.html`

- A security group: One may use a default security group or a newly-created one.

- A key pair that allows the user to access the cloud via SSH: Amazon EC2 uses public-key cryptography for login credentials. Each EC2 instance is configured with a public key, and the user has to access this instance using the matching private key. It can be generaged and managed on AWS EC2 console as well as the user's terminal.[8]

C.4 Installation

First, we install the ParallelCluster command line interface (CLI) on a local machine. ParallelCluster command line interface is distributed through the standard Python Package Index (PyPI), so one may install it through `pip`, the standard package-installing command for Python. One may install ParallelCluster by executing the following on the command line:

```
sudo pip install aws-parallelcluster
```

C.5 Configuration

Once ParallelCluster is installed on a local machine, an initial configuration is needed. It can be done by various ways, but the easiest way is through the command below:

```
pcluster configure
```

Then, the interactive dialog to setup ParallelCluster appears:

[8] `https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ec2-key-pairs.html`

```
ParallelCluster Template [default]: <a name desired>
AWS Access Key ID []: <copy and paste the access key>
AWS Secret Access Key ID []: <copy and paste the key>
Acceptable Values for AWS Region ID:
    eu-north-1
    ap-south-1
    eu-west-3
    eu-west-2
    eu-west-1
    ap-northeast-2
    ap-northeast-1
    sa-east-1
    ca-central-1
    ap-southeast-1
    ap-southeast-2
    eu-central-1
    us-east-1
    us-east-2
    us-west-1
    us-west-2
AWS Region ID [ap-northeast-2]: <the region to use>
VPC Name [<default name>]: <a name desired>
Acceptable Values for Key Name:
    <the registered key names appear here>
Key Name []: <enter the EC2 key pair name>
Acceptable Values for VPC ID:
    <the list of VPC appears here>
VPC ID []: <enter one of the vpc above>
Acceptable Values for Master Subnet ID:
    <the list of subnet ids appears here>
Master Subnet ID [subnet-<default value>]: <subnet id>
```

Now examine the files in the directory `~/.parallelcluster` (a hidden directory under the home directory). The file `pcluster-cli.log` shows the log and the file `config` shows the configuration. One can modify the file `config` to fine-tune the configuration per user's need. The following is the `config` corresponding to our CfnCluster experiments:

```
[global]
update_check = true
sanity_check = true
cluster_template = test

[aws]
aws_region_name = ap-northeast-2

[cluster test]
vpc_settings = testcfn
key_name = <key name>
initial_queue_size = 0
max_queue_size = 20
ebs_settings = expr_ebs
scheduler = sge
compute_instance_type = c5.18xlarge
master_instance_type = c5.18xlarge
cluster_type = spot
spot_price = 1.20
base_os = centos7
scaling_settings = custom
extra_json = {"cluster" : { "cfn_scheduler_slots" : "2"} }
master_root_volume_size = 20
compute_root_volume_size = 20

[ebs expr_ebs]
ebs_snapshot_id = < a snapshot id >
volume_size = 40

[vpc testcfn]
master_subnet_id = < a subnet id >
vpc_id = < a vpc id >

[aliases]
ssh = ssh {CFN_USER}@{MASTER_IP} {ARGS}

[scaling custom]
scaling_idletime = 20
```

In the `[global]` section, we set global configurations. The `cluster_template`

names the cluster section to be used for the cluster.
update_check check for the updates to ParallelCluster, and sanity_check validates that resources defined in parameters.

In the [aws] section, the region is specified. AWS access key and secret key may appear here unless specified in the base AWS CLI.

In the [cluster] section, we define the detailed specification of the virtual cluster. The vpc_settings names a setting for VPC, detailed in the [vpc] section, and the ebs_settings names the setting for EBS, detailed in [ebs] section. The key_name defines the key name to use. The initial_queue_size defines the number of worker instances at the launch of the cluster. We used zero for our experiments, as we often needed to check if the configuration is done properly on master before running actual jobs. The worker instances are launched upon submission of a new job into the queue, and they are terminated when the workers stay idle for a while (not exactly defined, but often around five to ten minutes).
We set the max_queue_size, the maximum number of worker instances to 20. We used CentOS 7 as the base_os for our instances. The master_root_volume_size and the compute_root_volume_size determine the size of root volume of the master instance and each of the worker instance, respectively. For the scheduler, we used the Sun Grid Engine (sge). For the compute_instance_type, we used c5.18xlarge, an instance with 36 physical cores (72 virtual cores with hyperthreading). It consists of two non-uniform memory access (NUMA) nodes with 18 physical cores each. In NUMA memory design, an access to local memory of a processor is faster than an access to non-local memory within a shared memory system. master_instance_type defines the instance type of the master. Sometimes it is fine to be as small as t2.micro, a single-core instance, but we needed an instance with good net-

work performance when many instances simultaneously accessed a large data file on shared storage. The `cluster_type` is either `ondemand` (default) or `spot`. For `c5.18xlarge` in Seoul region (`ap-northeast-2`), on-demand price was $3.456 per instance-hour, while the spot price was at $1.0788 per instance-hour throughout the duration of our experiments. Budget-constrained users may use spot instances for worker instances. In case of this scenario, the `spot_prices` was set to $1.20 per instance-hour, so if the actual price went above this value, our worker instances would have been terminated. Only the on-demand instance could be used as the master instance, so smaller instance might be desirable for lower cost. The setting `extra_json = {"cluster" : { "cfn_scheduler_slots" : "2"} }` sets number of slots that an instance bears to two. Each computing job is required to declare the number of "slots" to occupy. By default, the number of slots per instance is the number of virtual cores the instance has. This default setting is natural, but a problem arises if we intend to utilize shared-memory parallelism in NUMA node-level, as the number of slots occupied is tied to the number of instances launched. We assigned one slot per NUMA node that an instance has (i.e., 2 slots per instance), and utilized all 18 physical cores per NUMA node.

The `[ebs]` section defines the configuration for the EBS volume mounted on the master node and shared via NFS to workers. The `ebs_snapshot_id` defines the ID of the EBS snapshot to be used. We had datasets and packages necessary for our jobs pre-installed in an EBS volume and created a snapshot. The size of the volume was 40 GB. By default, the volume is mounted to the path `/shared`.

We refer the readers to the manual `https://docs.aws.amazon.com/parallelcluster/` for further details.

C.6 Creating, accessing, and destroying the cluster

We can create a virtual cluster named `example` by issuing the following command on a local machine:

```
pcluster create example
```

To access the master instance through ssh, one needs the location of the private key (`.pem`) file. The command to use is:

```
pcluster ssh example -i <private key file>
```

The default username for instances with CentOS is `centos`. The default username depends on the Amazon Machine Image (AMI) being used to create a virtual machine, which is determined by the `base_os` selected on the configuration. The names of the existing clusters can be listed using the command `pcluster list`, and we may completely remove a cluster `example` using the command `pcluster delete example`.

C.7 Installation of libraries

Now we can access the master node through secure shell(SSH). We have a shared EBS volume mounted at `/shared`, and we are to install necessary software there. For our experiments, we installed anaconda, a portable installation of Python, in the directory `/shared`. A script to set up environment variables is also created and saved in `/shared`:

```
# setup.sh
module load mpi/openmpi-x86_64 # loads MPI to the environment
source /shared/conda/etc/profile.d/conda.sh
export PATH=/shared/conda/bin:$PATH
export LD_LIBRARY_PATH=/shared/conda/lib:$LD_LIBRARY_PATH
```

We issued the command:

```
source setup.sh
```

to set up the environment variables. We installed PyTorch from source[9], as it is required to do so in order to incorporate MPI.

To download our code, one can issue the command:

```
git clone https://github.com/kose-y/dist_stat /shared/dist_stat
```

C.8 Running a job

To provide instructions on how to define the environment to each instance, we need a script defining each job. The following script `mcpi-2.job` is for running the program for Monte Carlo estimation of π in Section 3.3 (Listing 3.1) using two instances (four processes using 18 threads each).

```
#!/bin/sh
#$ -cwd
#$ -N mcpi
#$ -pe mpi 4
#$ -j y
date
source /shared/conda/etc/profile.d/conda.sh
export PATH=/shared/conda/bin:$PATH
export LD_LIBRARY_PATH=/shared/conda/lib:$LD_LIBRARY_PATH
export MKL_NUM_THREADS=18
mpirun -np 4 python /shared/dist_stat/examples/mcpi-mpi-pytorch.py
```

The line `-pe mpi 4` tells the scheduler that we are using four slots. Setting the value of the environment variable `MKL_NUM_THREADS` to 18 means that MKL runs with 18 threads or cores for that process. We launch four processes in the cluster, two per instance, as defined by our ParallelCluster setup, in parallel using MPI. We can submit this job to the Sun Grid Engine (the job scheduler) using the command:

[9] `https://github.com/pytorch/pytorch#from-source`

```
qsub mcpi-2.job
```

When we submit a job, a message similar to the following appears:

```
Your job 130 ("mcpi") has been submitted
```

One may see the newly submitted job in the queue using the command `qstat`.

```
qstat
```

```
job-ID  prior   name       user         state submit/start at     queue      slots ja-task-ID
-----------------------------------------------------------------------------------------------
    130 0.55500 mcpi       centos       qw    02/28/2019 03:58:54                 4
```

If we want to delete any job waiting for the queue or running, use the command `qdel`.

```
qdel 130
```

```
centos has deleted job 130
```

Once the job is completed, the output is saved as a text file named such as `mcpi.o130`. For example:

```
Thu Feb 28 04:07:54 UTC 2019
3.148
```

The scripts for our numerical examples are in `/shared/dist_stat/jobs`.

C.9 Miscellaneous

To keep what is on the EBS volume on the cloud and access later, we need to create a snapshot for the volume. We can later create a volume based on

this snapshot[10], and mount it on any instance[11]. In ParallelCluster, this is done automatically when we give an ID of a snapshot in the `config` file.

[10] https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ebs-creating-snapshot.html

[11] https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ebs-using-volumes.html

Appendix D

Code for memory-efficient ℓ_1-regularized Cox proportional hazards model

For CPU code, the following code accelerates the computation of $P_{(n+1)}\delta$ for ℓ_1-regularized Cox regression in Section 3.5.4 using the AVX.

```
using LoopVectorization
function pi_delta!(out, w, W_dist, delta, W_range)
    # fill `out` with zeros beforehand.
    m = length(delta)
    W_base = minimum(W_range) - 1
    W_local = W_dist.localarray
    @avx for i in 1:m
        outi = zero(eltype(w))
        for j in 1:length(W_range)
            outi += ifelse(i <= j + W_base,
                delta[j + W_base] *
                w[i] / W_local[j], zero(eltype(w)))
        end
        out[i] = outi
    end
    DistStat.Barrier()
```

```
    DistStat.Allreduce!(out)
    return out
end
```

`DistStat.Allreduce!(out)` computes the elementwise sum of `out` in all ranks, and saves it in the place of `out`. For GPU, the kernel function can be written as follows:

```
function pi_delta_kernel!(out, w, W_dist, delta, W_range)
    idx_x = (blockIdx().x-1) *
        blockDim().x + threadIdx().x
    stride_x = blockDim().x * gridDim().x
    W_base = minimum(W_range) - 1
    for i in idx_x:stride_x:length(out)
        for j in W_range
            @inbounds if i <= j
                out[i] += delta[j] * w[i] /
                    W_dist[j - W_base]
            end
        end
    end
end
```

And the host function to compute $P_{(n+1)}\delta$ is:

```
function pi_delta!(out::CuArray, w::CuArray,
            W_dist, delta, W_range)
    fill!(out, zero(eltype(out)))
    numblocks = ceil(Int, length(w)/256)
    CuArrays.@sync begin
        @cuda threads=256 blocks=numblocks pi_delta_kernel!(
            out, w, W_dist.localarray, delta, W_range)
    end
    DistStat.Allreduce!(out)
    out
end
```

Appendix E

Details of SNPs selected in ℓ_1-regularized Cox regression

Figure E.1 shows the solution path for SNPs within the range we used for the experiment in Section 3.5.5. Tables E.1 and E.2 list the 111 selected SNPs with `dist_stat`.

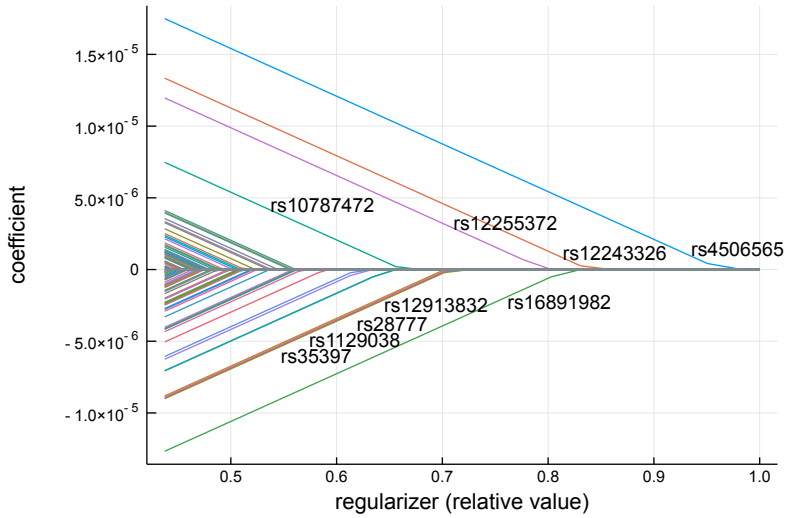


Figure E.1: Solution path for ℓ_1-regularized Cox regression on the UK Biobank dataset. Signs are with respect to the reference allele: positive value favors alternative allele as the risk allele.

Table E.1: SNPs selected by ℓ_1-penalized Cox regression: #1-#56

Rank	SNP ID	Chr[A]	Location	A1[B]	A2[C]	MAF[D]	Mapped genes	Sign[E]	Known[F]
1	rs4506565	10	114756041	A	**T**	0.238	TCF7L2	+	Yes
2	rs12243326	10	114788815	**C**	T	0.249	TCF7L2	+	Yes
3	rs16891982	5	33951693	G	**C**	0.215	SLC45A2	−	
4	rs12255372	10	114808902	**T**	G	0.215	TCF7L2	+	Yes
5	rs12913832	15	28365618	G	**A**	0.198	HERC2	−	
6	rs28777	5	33958959	**C**	A	0.223	SLC45A2	−	
7	rs1129038	15	28356859	**C**	T	0.343	HERC2	−	
8	rs35397	5	33951116	T	**G**	0.304	SLC45A2	−	
9	rs10787472	10	114781297	**C**	A	0.430	TCF7L2	+	Yes
10	rs2470890	15	75047426	T	**C**	0.429	CYP1A2	−	
11	rs2472304	15	75044238	A	**G**	0.460	CYP1A2	−	
12	rs1378942	15	75077367	A	**C**	0.401	CSK, MIR4513	−	
13	rs34862454	15	75101530	T	**C**	0.416	LMAN1L	−	
14	rs849335	7	28223990	C	**T**	0.406	JAZF1, JAZF1-AS1	−	Yes
15	rs864745	7	28180556	C	**T**	0.316	JAZF1	−	Yes
16	rs12785878	11	71167449	T	**G**	0.251	NADSYN1, DHCR7	−	
17	rs4944958	11	71168073	G	**A**	0.237	NADSYN1, DHCR7	−	
18	rs8042680	15	91521337	**A**	C	0.277	PRC1, PRC1-AS1, Y_RNA	+	
19	rs35414	5	33969628	**T**	C	0.188	SLC45A2	−	
20	rs1635852	7	28189411	**T**	C	0.423	JAZF1	−	
21	rs10962525	9	16659863	**T**	C	0.321	BNC2	+	
22	rs1446585	2	136407479	**G**	A	0.322	R3HDM1	+	
23	rs7570971	2	135837906	**A**	C	0.327	RAB3GAP1	+	
24	rs36074798	15	91518800	**ACT**	A	0.328	PRC1, PRC1-AS1, Y_RNA	+	Yes
25	rs10962612	9	16804167	G	**T**	0.088	BNC2	−	
26	rs10962612	2	135911422	T	**C**	0.097	RAB3GAP1, ZRANB3	+	
27	rs941444	17	17693891	**C**	G	0.073	RAI1	−	Yes
28	rs6769511	3	185530290	T	**C**	0.045	IGF2BP2	+	Yes
29	rs916977	15	28513364	**T**	C	0.044	HERC2	−	
30	rs35390	5	33955326	**C**	A	0.062	SLC45A2	−	
31	rs35391	5	33955673	**T**	C	0.374	SLC45A2	−	
32	rs1470579	3	185529080	A	**C**	0.436	IGF2BP2	+	Yes
33	rs2862954	10	101912064	**T**	C	0.488	ERLIN1	−	
34	rs2297174	9	16706557	A	**G**	0.346	BNC2	−	
35	rs1667394	15	28530182	T	**C**	0.274	HERC2	−	
36	rs12440952	15	74615292	**G**	A	0.279	CCDC33	+	
37	rs56343038	9	16776792	G	**T**	0.318	BNC2, LSM1P1	−	
38	rs9522149	13	111827167	**T**	C	0.395	ARHGEF7	−	
39	rs343092	12	66250940	**T**	G	0.463	HMGA2, HMGA2-AS1	−	Yes
40	rs10733316	9	16696626	**T**	C	0.436	BNC2	−	
41	rs823485	1	234671267	T	**C**	0.488	LINC01354	+	
42	rs12910825	15	91511260	A	**G**	0.384	PRC1, PRC1-AS1, RCCD1	+	Yes
43	rs2959005	15	74618128	T	**C**	0.222	CCDC33	−	
44	rs10756801	9	16740110	T	**G**	0.494	BNC2	−	
45	rs12072073	1	3130016	**C**	T	0.497	PRDM16	+	
46	rs7039444	9	20253425	T	**C**	0.360	(intergenic variant)	+	
47	rs7899137	10	76668462	**A**	C	0.289	KAT6B	−	
48	rs11078405	17	17824978	**T**	G	0.291	TOM1L2	+	
49	rs830532	5	142289541	C	**T**	0.333	ARHGAP26	+	
50	rs833283	3	181590598	**G**	C	0.352	(intergenic variant)	−	
51	rs10274928	7	28142088	**A**	G	0.365	JAZF1	−	Yes
52	rs13301628	9	16665850	A	**C**	0.412	BNC2	−	
53	rs885107	16	30672719	**C**	T	0.353	PRR14, FBRS	+	
54	rs8180897	8	121699907	A	**G**	0.445	SNTB1	+	
55	rs23282	5	142270301	G	**A**	0.225	ARHGAP26	+	
56	rs6428460	1	198377460	C	**T**	0.229	(intergenic variant)	+	

[A] Chromosome, [B] Minor allele, [C] Major allele, [D] Minor allele frequency, [E] Sign of the regression coefficient, [F] Mapped gene included in Mahajan et al. (2018). The boldface indicates the risk allele determined by the reference allele and the sign of the regression coefficient.

Table E.2: SNPs selected by ℓ_1-penalized Cox regression: #57-#111

Rank	SNP ID	Chr[A]	Location	A1[B]	A2[C]	MAF[D]	Mapped genes	Sign[E]	Known[F]
57	rs11630918	15	75155896	**C**	T	0.383	SCAMP2	−	
58	rs7187359	16	30703155	G	**A**	0.335	(intergenic variant)	+	
59	rs2183405	9	16661933	G	**A**	0.271	BNC2	+	
60	rs2651888	1	3143384	G	**T**	0.411	PRDM16	+	
61	rs2189965	7	28172014	**T**	C	0.340	JAZF1	+	Yes
62	rs12911254	15	75166335	A	**G**	0.344	SCAMP2	−	
63	rs757729	7	28146305	**G**	C	0.441	JAZF1	−	Yes
64	rs6495122	15	75125645	C	**A**	0.478	CPLX3, ULK3	−	
65	rs4944044	11	71120213	A	**G**	0.426	AP002387.1	−	
66	rs6856032	4	38763994	G	**C**	0.109	RNA5SP158	+	
67	rs1375132	2	135954405	**G**	A	0.478	ZRANB3	+	
68	rs2451138	8	119238473	T	**C**	0.314	SAMD12	−	
69	rs6430538	2	135539967	**T**	C	0.470	CCNT2-AS1	+	
70	rs7651090	3	185513392	**G**	A	0.281	IGF2BP2	+	Yes
71	rs4918711	10	113850019	T	**C**	0.285	(intergenic variant)	−	
72	rs3861922	1	198210570	A	**G**	0.466	NEK7	−	
73	rs7917983	10	114732882	T	**C**	0.481	TCF7L2	+	Yes
74	rs1781145	1	1388289	A	**C**	0.362	ATAD3C	+	
75	rs7170174	15	94090333	**T**	C	0.246	AC091078.1	−	
76	rs7164916	15	91561446	T	**C**	0.246	VPS33B, VPS33B-DT	+	
77	rs696859	1	234656596	T	**C**	0.430	(intergenic variant)	+	
78	rs28052	5	142279870	C	**G**	0.166	ARHGAP26	+	
79	rs1408799	9	12672097	**T**	C	0.277	(intergenic variant)	−	
80	rs10941112	5	34004707	**C**	T	0.355	AMACR, C1QTNF3-AMACR	−	
81	rs11856835	15	74716174	**G**	A	0.261	SEMA7A	−	
82	rs4768617	12	45850022	**T**	C	0.259	(intergenic variant)	−	
83	rs8012970	14	101168491	**T**	C	0.179	(intergenic variant)	−	
84	rs4402960	3	185511687	G	**T**	0.187	IGF2BP2	+	Yes
85	rs1695824	1	1365570	A	**C**	0.164	LINC01770, VWA1	+	
86	rs934886	15	55939959	**A**	G	0.360	PRTG	−	
87	rs7083429	10	69303421	**G**	T	0.367	CTNNA3	+	
88	rs4918788	10	114820961	G	**A**	0.348	TCF7L2	+	Yes
89	rs7219320	17	17880877	A	**G**	0.318	DRC3, AC087163.1, ATPAF2	+	
90	rs61822626	1	205118441	C	**T**	0.478	DSTYK	−	Yes
91	rs250414	5	33990623	**C**	T	0.361	AMACR, C1QTNF3-AMACR	−	
92	rs11073964	15	91543761	C	**T**	0.362	VPS33B,PRC1	+	Yes
93	rs17729876	10	101999746	**G**	A	0.352	CWF19L1, SNORA12	−	
94	rs2386584	15	91539572	T	**G**	0.360	VPS33B, PRC1	+	Yes
95	rs683	9	12709305	**C**	A	0.430	TYRP1, LURAP1L-AS1	−	
96	rs17344537	1	205091427	**T**	G	0.462	RBBP5	−	
97	rs10416717	19	13521528	A	**G**	0.470	CACNA1A	+	
98	rs2644590	1	156875107	**C**	A	0.453	PEAR1	−	
99	rs447923	5	142252257	**T**	C	0.384	ARHGAP26, ARHGAP26-AS1	+	
100	rs2842895	6	7106316	C	**G**	0.331	RREB1	−	Yes
101	rs231354	11	2706351	**C**	T	0.329	KCNQ1, KCNQ1OT1	+	Yes
102	rs4959424	6	7084857	T	**G**	0.410	(intergenic variant)	−	
103	rs2153271	9	16864521	T	**C**	0.411	BNC2	−	
104	rs12142199	1	1249187	A	**G**	0.398	INTS11, PUSL1, ACAP3, MIR6727	−	
105	rs2733833	9	12705095	**T**	G	0.272	TYRP1, LURAP1L-AS1	−	
106	rs1564782	15	74622678	A	**G**	0.283	CCDC33	−	
107	rs9268644	6	32408044	**C**	A	0.282	HLA-DRA	+	
108	rs271738	1	234662890	A	**G**	0.395	LINC01354	+	
109	rs12907898	15	75207872	T	**C**	0.391	COX5A	−	
110	rs146900823	3	149192851	GC	**G**	0.344	TM4SF4	−	
111	rs1635166	15	28539834	T	**C**	0.118	HERC2	−	

[A] Chromosome, [B] Minor allele, [C] Major allele, [D] Minor allele frequency,
[E] Sign of the regression coefficient, [F] Mapped gene included in Mahajan et al. (2018). The boldface indicates the risk allele determined by the reference allele and the sign of the regression coefficient.

Bibliography

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. (2015). TensorFlow: Large-scale machine learning on heterogeneous systems. Software available from tensorflow.org.

Amazon Web Services (2019). AWS ParallelCluster documentation. Accessed: 2019-10-14.

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. *Journal of Machine Learning Research*, 18(1):310–342.

Bahrampour, S., Ramakrishnan, N., Schott, L., and Shah, M. (2015). Com-

parative study of deep learning software frameworks. *arXiv preprint arXiv:1511.06435*.

Baker, J., Fearnhead, P., Fox, E. B., and Nemeth, C. J. (2018). `sgmcmc`: An R package for stochastic gradient markov chain monte carlo. *Journal of Statistical Software*, 91(3).

Ballard, G., Demmel, J., Holtz, O., and Schwartz, O. (2011). Minimizing communication in numerical linear algebra. *SIAM Journal on Matrix Analysis and Applications*, 32(3):866–901.

Bauschke, H. H. and Combettes, P. L. (2011). *Convex analysis and monotone operator theory in Hilbert spaces*. Springer Science & Business Media.

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 Journal on Imaging Sciences*, 2(1):183–202.

Belloni, A., Chernozhukov, V., and Wang, L. (2011). Square-root lasso: pivotal recovery of sparse signals via conic programming. *Biometrika*, 98(4):791–806.

Bergstra, J., Bastien, F., Breuleux, O., Lamblin, P., Pascanu, R., Delalleau, O., Desjardins, G., Warde-Farley, D., Goodfellow, I., Bergeron, A., et al. (2011). Theano: Deep learning on GPUs with Python. In *NIPS 2011, BigLearning Workshop, Granada, Spain*, volume 3, pages 1–48.

Bertsekas, D. P. (2009). *Convex optimization theory*. Athena Scientific.

Besard, T., Churavy, V., Edelman, A., and De Sutter, B. (2019). Rapid software prototyping for heterogeneous and distributed platforms. *Advances in Engineering Software*, 132:29–46.

Besard, T., Foket, C., and De Sutter, B. (2018). Effective extensible programming: Unleashing Julia on GPUs. *IEEE Transactions on Parallel and Distributed Systems*.

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 Transactions on Mathematical Software*, 28(2):135–151.

Boţ, R. I. and Csetnek, E. R. (2015). On the convergence rate of a forward-backward type primal-dual splitting algorithm for convex optimization problems. *Optimization*, 64(1):5–23.

Boţ, R. I. and Csetnek, E. R. (2016). An inertial forward-backward-forward primal-dual splitting algorithm for solving monotone inclusion problems. *Numerical Algorithms*, 71(3):519–540.

Boţ, R. I., Csetnek, E. R., Heinrich, A., and Hendrich, C. (2015). On the convergence rate improvement of a primal-dual splitting algorithm for solving monotone inclusion problems. *Mathematical Programming*, 150(2):251–279.

Boyd, S., Parikh, N., Chu, E., Peleato, B., and Eckstein, J. (2010). Distributed optimization and statistical learning via the alternating direction method of multipliers. *Foundations and Trends in Machine Learning*.

Boyd, S. P. and Vandenberghe, L. (2004). *Convex optimization*. Cambridge University Press, Cambridge, UK.

Breslow, N. E. (1972). Discussion of the paper by D. R. Cox. *Journal of the Royal Statistical Society: Series B (Methodological)*, 34(2):216–217.

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.

Buckner, J., Wilson, J., Seligman, M., Athey, B., Watson, S., and Meng, F. (2010). The gputools package enables gpu computing in r. *Bioinformatics*, 26(1):134–135.

Chambolle, A. and Pock, T. (2011). A first-order primal-dual algorithm for convex problems with applications to imaging. *Journal of Mathematical Imaging and Vision*, 40(1):120–145.

Chambolle, A. and Pock, T. (2016). On the ergodic convergence rates of a first-order primal-dual algorithm. *Mathematical Programming*, 159(1-2):253–287.

Chen, P., Huang, J., and Zhang, X. (2013). A primal-dual fixed point algorithm for convex separable minimization with applications to image restoration. *Inverse Problems*, 29(2):025011.

Chen, P., Huang, J., and Zhang, X. (2016). A primal-dual fixed point algorithm for multi-block convex minimization. *arXiv preprint arXiv:1602.00414*.

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, X., Lin, Q., Kim, S., Carbonell, J. G., and Xing, E. P. (2012). Smooth-

ing proximal gradient method for general structured sparse regression. *The Annals of Applied Statistics*, 6(2):719–752.

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

Cheng, B. and Titterington, D. M. (1994). Neural networks: A review from a statistical perspective. *Statistical Science*, pages 2–30.

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

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 Biology*, 9(7):e1001091.

Collobert, R., Kavukcuoglu, K., and Farabet, C. (2011). Torch7: A Matlab-like environment for machine learning. In *BigLearn, NIPS workshop*, number EPFL-CONF-192376.

Combettes, P. L. (2018). Monotone operator theory in convex optimization. *Mathematical Programming*, 170:177–206.

Combettes, P. L., Condat, L., Pesquet, J.-C., and Vũ, B. C. (2014). A forward-backward view of some primal-dual optimization methods in image recovery. In *2014 IEEE International Conference on Image Processing (ICIP)*, pages 4141–4145. IEEE.

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*, pages 185–212. Springer.

Combettes, P. L. and Pesquet, J.-C. (2012). Primal-dual splitting algorithm for solving inclusions with mixtures of composite, Lipschitzian, and parallel-sum type monotone operators. *Set-Valued and Variational Analysis*, 20(2):307–330.

Combettes, P. L. and Vũ, B. C. (2014). Variable metric forward-backward splitting with applications to monotone inclusions in duality. *Optimization*, 63(9):1289–1318.

Combettes, P. L. and Wajs, V. R. (2005). Signal recovery by proximal forward-backward splitting. *Multiscale Modeling & Simulation*, 4(4):1168–1200.

Condat, L. (2013). A primal-dual splitting method for convex optimization involving lipschitzian, proximable and linear composite terms. *Journal of Optimization Theory and Applications*, 158(2):460–479.

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. *Journal of Investigative Dermatology*, 129(2):392–405.

Cox, D. R. (1972). Regression models and life-tables. *Journal of the Royal Statistical Society: Series B (Methodological)*, 34(2):187–202.

Dalcin, L. D., Paz, R. R., Kler, P. A., and Cosimo, A. (2011). Parallel distributed computing using Python. *Advances in Water Resources*, 34(9):1124–1139.

Davis, D. (2015). Convergence rate analysis of primal-dual splitting schemes. *SIAM J. Optim.*, 25(3):1912–1943.

Davis, D. and Yin, W. (2017). A three-operator splitting scheme and its optimization applications. *Set-valued and variational analysis*, 25(4):829–858.

de Leeuw, J. (1977). Applications of convex analysis to multidimensional scaling. In *Recent Developments in Statistics (Proc. European Meeting of Statisticians, Grenoble, 1976)*, pages 133–145.

de Leeuw, J. and Heiser, W. J. (1977). Convergence of correction matrix algorithms for multidimensional scaling. *Geometric Representations of Relational Data*, pages 735–752.

Dempster, A. P., Laird, N. M., and Rubin, D. B. (1977). Maximum likelihood from incomplete data via the EM algorithm. *Journal of the Royal Statistical Society. Series B (methodological)*, pages 1–38.

Dillon, J. V., Langmore, I., Tran, D., Brevdo, E., Vasudevan, S., Moore, D., Patton, B., Alemi, A., Hoffman, M., and Saurous, R. A. (2017). `Tensorflow` distributions. *arXiv preprint arXiv:1711.10604*.

Donoho, D. (2017). 50 years of data science. *Journal of Computational and Graphical Statistics*, 26(4):745–766.

Drori, Y., Sabach, S., and Teboulle, M. (2015). A simple algorithm for a class of nonsmooth convex–concave saddle-point problems. *Operations Research Letters*, 43(2):209–214.

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. *Nature Genetics*, 42(2):105.

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

Elrod, C. (2020). *LoopVectorization.jl: Macro(s) for vectorizing loops*. Julia package version 0.6.21.

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 Journal on Imaging Sciences*, 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 *In The 1st Workshop on Cloud Computing and its Applications (CCA)*.

Fawcett, K. A., Wheeler, E., Morris, A. P., Ricketts, S. L., Hallmans, G., Rolandsson, O., Daly, A., Wasson, J., Permutt, A., Hattersley, A. T., Glaser, B., Franks, P. W., McCarthy, M. I., Wareham, N. J., Sandhu, M. S., and Barroso, I. (2010). Detailed investigation of the role of common and low-frequency wfs1 variants in type 2 diabetes risk. *Diabetes*, 59(3):741–746.

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

Friedman, J., Hastie, T., Höfling, H., and Tibshirani, R. (2007). Pathwise coordinate optimization. *The Annals of Applied Statistics*, 1(2):302–332.

Friedman, J., Hastie, T., and Tibshirani, R. (2010). Regularization paths for generalized linear models via coordinate descent. *Journal of Statistical Software*, 33(1):1–22.

Gabay, D. and Mercier, B. (1976). A dual algorithm for the solution of non-linear variational problems via finite element approximation. *Computers & Mathematics with Applications*, 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, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary.

Gentzsch, W. (2001). Sun Grid Engine: Towards creating a compute power grid. In *Proceedings First IEEE/ACM International Symposium on Cluster Computing and the Grid*, pages 35–36. IEEE.

Ghadimi, S. and Lan, G. (2012). Optimal stochastic approximation algorithms for strongly convex stochastic composite optimization i: A generic algorithmic framework. *SIAM Journal on Optimization*, 22(4):1469–1492.

Glowinski, R. and Marroco, A. (1975). Sur l'approximation, par éléments finis d'ordre un, et la résolution, par pénalisation-dualité d'une classe de problèmes de dirichlet non linéaires. *Revue française d'automatique, informatique, recherche opérationnelle. Analyse numérique*, 9(2):41–76.

Goldstein, T., Li, M., and Yuan, X. (2015). Adaptive primal-dual splitting methods for statistical learning and image processing. In *Advances in Neural Information Processing Systems 28*, pages 2089–2097.

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

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

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.

Hiriart-Urruty, J.-B. and Lemaréchal, C. (1993). *Convex analysis and minimization algorithms II: Advanced Theory and Bundle Methods*. Springer-Verlag Berlin Heidelberg.

Hu, C., Pan, W., and Kwok, J. T. (2009). Accelerated gradient methods for stochastic optimization and online learning. In *Advances in Neural Information Processing Systems*, pages 781–789.

Huang, D. W., Sherman, B. T., and Lempicki, R. A. (2009a). Bioinformatics enrichment tools: paths toward the comprehensive functional analysis of large gene lists. *Nucleic Acids Research*, 37(1):1–13.

Huang, D. W., Sherman, B. T., and Lempicki, R. A. (2009b). Systematic and integrative analysis of large gene lists using DAVID bioinformatics resources. *Nature Protocols*, 4(1):44.

Hunter, D. R. and Lange, K. (2004). A tutorial on MM algorithms. *The American Statistician*, 58(1):30–37.

Hunter, D. R. and Li, R. (2005). Variable selection using MM algorithms. *Annals of statistics*, 33(4):1617–1642.

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

IEEE Standards Committee (2008). 754-2008 ieee standard for floating-point arithmetic. *IEEE Computer Society Std*, 2008:517.

Jacob, L., Obozinski, G., and Vert, J.-P. (2009). Group lasso with overlap and graph lasso. In *Proceedings of the 26th Annual International Conference on Machine Learning*, pages 433–440. ACM.

Jain, A. K. (1989). *Fundamentals of Digital Image Processing*. Englewood Cliffs, NJ: Prentice Hall.

Janssens, B. (2018). *MPIArrays.jl: Distributed arrays based on MPI onesided communication*.

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 *Proceedings of the 22nd ACM international conference on Multimedia*, pages 675–678. ACM.

Jordan, M. I., Lee, J. D., and Yang, Y. (2019). Communication-efficient distributed statistical inference. *Journal of the Americal Statistical Association*, 114(526):668–681.

Juditsky, A., Nemirovski, A., and Tauvel, C. (2011). Solving variational inequalities with stochastic mirror-prox algorithm. *Stochastic Systems*, 1(1):17–58.

Julia Contributors (2020). *Julia 1.4 Documentation*. Julia version 1.4.

JuliaGPU Contributors (2020). `CUDA.jl: CUDA` *programming in* Julia. Julia package version 1.0.2.

JuliaParallel Contributors (2019). *DistributedArrays.jl: Distributed arrays in Julia*. Julia package version 0.6.4.

JuliaParallel Contributors (2020). *MPI.jl: MPI wrappers for Julia*. Julia package version 0.11.0.

Keys, K. L., Zhou, H., and Lange, K. (2019). Proximal distance algorithms: Theory and practice. *Journal of Machine Learning Research*, 20(66):1–38.

Klimentidis, Y. C., Zhou, J., and Wineinger, N. E. (2014). Identification of allelic heterogeneity at type-2 diabetes loci and impact on prediction. *PloS one*, 9(11).

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 Computing*, 38(3):157–174.

Ko, S. and Won, J.-H. (2019). Optimal minimization of the sum of three convex functions with a linear operator. In *The 22nd International Conference on Artificial Intelligence and Statistics*, pages 1185–1194.

Ko, S., Yu, D., and Won, J.-H. (2019+). Easily parallelizable and distributable class of algorithms for structured sparsity, with optimal acceleration. *Journal of Computational and Graphical Statistics*, (to appear).

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 *International Conference on Artificial Intelligence and Statistics*, pages 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 International Parallel and Distributed Processing Symposium (IPDPS)*, pages 842–853. IEEE.

Lam, S. K., Pitrou, A., and Seibert, S. (2015). Numba: A LLVM-based Python JIT compiler. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, page 7. ACM.

Lan, G. (2012). An optimal method for stochastic composite optimization. *Mathematical Programming*, 133(1-2):365–397.

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

Lange, K. and Carson, R. (1984). EM reconstruction algorithms for emission and transmission tomography. *Journal of Computer Assisted Tomography*, 8(2):306–16.

Lange, K., Hunter, D. R., and Yang, I. (2000). Optimization transfer using surrogate objective functions. *Journal of Computational and Graphical Statistics*, 9(1):1–20.

Langenberg, C., Sharp, S. J., Franks, P. W., Scott, R. A., Deloukas, P., Forouhi, N. G., Froguel, P., Groop, L. C., Hansen, T., Palla, L., et al. (2014). Gene-lifestyle interaction and type 2 diabetes: the epic interact case-cohort study. *PLoS Med*, 11(5):e1001647.

Latafat, P. and Patrinos, P. (2017). Asymmetric forward–backward–adjoint splitting for solving monotone inclusions involving three operators. *Computational Optimization and Applications*, 68(1):57–93.

Lattner, C. and Adve, V. (2004). LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004.*, pages 75–86. IEEE.

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

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

Lee, D. D. and Seung, H. S. (2001). Algorithms for non-negative matrix factorization. In *Advances in neural information processing systems*, pages 556–562.

Lee, T., Won, J.-H., Lim, J., and Yoon, S. (2017). Large-scale structured sparsity via parallel fused lasso on multiple gpus. *Journal of Computational and Graphical Statistics*, (to appear).

Li, C. (2019). JuliaCall: an R package for seamless integration between R and Julia. *The Journal of Open Source Software*, 4(35):1284.

Li, Q., Kecman, V., and Salman, R. (2010). A chunking method for Euclidean distance matrix calculation on large dataset using multi-GPU. In *2010 Ninth International Conference on Machine Learning and Applications*, pages 208–213. IEEE.

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. *Physics in Medicine & Biology*, 63(3):035042.

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

Lin, Q., Chen, X., and Peña, J. (2014). A smoothing stochastic gradient method for composite optimization. *Optimization Methods and Software*, 29(6):1281–1301.

Lin, Z., Liu, R., and Su, Z. (2011). Linearized alternating direction method with adaptive penalty for low-rank representation. In *Advances in Neural Information Processing Systems*, pages 612–620.

Liu, C., Yang, H.-c., Fan, J., He, L.-W., and Wang, Y.-M. (2010a). Distributed nonnegative matrix factorization for web-scale dyadic data analysis on MapReduce. In *Proceedings of the 19th International Conference on World Wide Web*, pages 681–690. ACM.

Liu, J., Yuan, L., and Ye, J. (2010b). An efficient algorithm for a class of fused lasso problems. In *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 323–332. ACM.

Lorenz, D. A. and Pock, T. (2015). An inertial forward-backward algorithm for monotone inclusions. *Journal of Mathematical Imaging and Vision*, 51(2):311–325.

Loris, I. and Verhoeven, C. (2011). On a generalization of the iterative soft-thresholding algorithm for the case of non-separable penalty. *Inverse problems*, 27(12):125007.

Ma, E., Gupta, V., Hsu, M., and Roy, I. (2016). dmapply: A functional primitive to express distributed machine learning algorithms in R. *Proceedings of the VLDB Endowment*, 9(13):1293–1304.

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. *Nature Genetics*, 50(11):1505.

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 Biology*, 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.

Monteiro, R. D. and Svaiter, B. F. (2011). Complexity of variants of tseng's modified fb splitting and korpelevich's methods for hemivariational inequalities with applications to saddle-point and convex optimization problems. *SIAM Journal on Optimization*, 21(4):1688–1720.

Munshi, A. (2009). The OpenCL specification. In *2009 IEEE Hot Chips 21 Symposium (HCS)*, pages 1–314. IEEE.

Nakano, J. (2012). Parallel computing techniques. In *Handbook of Computational Statistics*, pages 243–271. Springer.

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. *Statistical Science*, 27(4):538–557.

Nemirovski, A. (2004). Prox-method with rate of convergence O(1/t) for variational inequalities with Lipschitz continuous monotone operators and smooth convex-concave saddle point problems. *SIAM Journal on Optimization*, 15(1):229–251.

Nemirovski, A., Juditsky, A., Lan, G., and Shapiro, A. (2009). Robust stochastic approximation approach to stochastic programming. *SIAM Journal on optimization*, 19(4):1574–1609.

Nemirovsky, A. (1992). Information-based complexity of linear operator equations. *Journal of Complexity*, 8(2):153–175.

NERSC (2019). Distributed TensorFlow. Accessed: 2019-12-03.

Nesterov, Y. (2004). *Introductory lectures on convex optimization: A basic course*, volume 87. Springer Science & Business Media.

Nesterov, Y. (2005). Smooth minimization of non-smooth functions. *Mathematical Programming*, 103(1):127–152.

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 Genetics*, 10(8):e1004517.

Nitanda, A. (2014). Stochastic proximal gradient descent with acceleration techniques. In *Advances in Neural Information Processing Systems*, pages 1574–1582.

Nvidia (2007). Nvidia CUDA compute unified device architecture programming guide, Version 1.0.

NVIDIA (2013). Basic linear algebra subroutines (cuBLAS) library. Accessed: 2019-11-28.

NVIDIA (2018). Sparse matrix library (cuSPARSE). Accessed: 2019-11-28.

Ouyang, H. and Gray, A. (2012). Stochastic smoothing for nonsmooth minimizations: accelerating sgd by exploiting structure. In *International Coference on International Conference on Machine Learning*, pages 1523–1530.

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*, volume 26, pages 80–113. Wiley Online Library.

Parikh, N. and Boyd, S. (2014). Proximal algorithms. *Foundations and Trends in Optimization*, 1(3):127–239.

Paszke, A., Gross, S., Chintala, S., Chanan, G., Yang, E., DeVito, Z., Lin, Z., Desmaison, A., Antiga, L., and Lerer, A. (2017). Automatic differentiation in PyTorch. In *NIPS Autodiff Workshop*.

Polson, N. G., Scott, J. G., and Willard, B. T. (2015). Proximal algorithms in statistics and machine learning. *Statistical Science*, 30(4):559–581.

R Core Team (2018). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. Accessed: 2019-11-28.

Raina, R., Madhavan, A., and Ng, A. Y. (2009). Large-scale deep unsupervised learning using graphics processors. In *Proceedings of the 26th Annual International Conference on Machine Learning*, pages 873–880. ACM.

Ramani, S. and Fessler, J. A. (2011). Parallel mr image reconstruction using augmented lagrangian methods. *IEEE Transactions on Medical Imaging*, 30(3):694–706.

Ramdas, A. and Tibshirani, R. J. (2016). Fast and flexible ADMM algorithms for trend filtering. *Journal of Computational and Graphical Statistics*, 25(3):839–858.

Riobello, C., Gómez, J., Gil-Peña, H., Tranche, S., Reguero, J. R., Jesús, M., Delgado, E., Calvo, D., Morís, C., Santos, F., Coto-Segura, P., Iglesial, S., Alonso, B., Alvarez, V., and Coto, E. (2016). Kcnq1 gene variants in the risk for type 2 diabetes and impaired renal function in the spanish renastur cohort. *Molecular and cellular endocrinology*, 427:86–91.

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

Rosasco, L., Villa, S., and Vũ, B. C. (2014). Convergence of stochastic proximal gradient algorithm. *arXiv preprint arXiv:1403.5074*.

RStudio (2019). tensorflow: R interface to TensorFlow. Accessed: 2019-11-28.

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

Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1988). Learning representations by back-propagating errors. *Cognitive Modeling*, 5(3):1.

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

Salman, M., Dasgupta, S., Cholendra, A., Venugopal, P., Lakshmi, G., Xaviour, D., Rao, J., and D'Souza, C. J. (2015). Mtnr1b gene polymorphisms and susceptibility to type 2 diabetes: A pilot study in south indians. *Gene*, 566(2):189–193.

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 *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 2135–2135. ACM.

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

Sidky, E. Y., Jørgensen, J. H., and Pan, X. (2012). Convex optimization problem prototyping for image reconstruction in computed tomography with the chambolle–pock algorithm. *Physics in Medicine & Biology*, 57(10):3065.

Solo.io (2019). Gloo: an Envoy-powered API gateway. Accessed: 2019-11-28.

Staples, G. (2006). Torque resource manager. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 8. ACM.

Suchard, M. A., Holmes, C., and West, M. (2010a). 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 Transactions on Modeling and Computer Simulation*, 23(1):10.

Suchard, M. A., Wang, Q., Chan, C., Frelinger, J., Cron, A., and West, M. (2010b). Understanding GPU programming for statistical computation: Stud-

ies in massively parallel massive mixtures. *Journal of Computational and Graphical Statistics*, 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 Medicine*, 12(3):e1001779.

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(7145):661.

Tibshirani, R. (1996). Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society: Series B (Methodological)*, 58(1):267–288.

Tibshirani, R., Saunders, M., Rosset, S., Zhu, J., and Knight, K. (2005). Sparsity and smoothness via the fused lasso. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 67(1):91–108.

Tibshirani, R. J. and Taylor, J. (2011). The solution path of the generalized lasso. *The Annals of Statistics*, 39(3):1335–1371.

Tieleman, T. (2010). Gnumpy: an easy way to use GPU boards in Python. *Department of Computer Science, University of Toronto*.

University of Zurich (2019). ElastiCluster. Accessed: 2019-12-06.

Ushey, K., Allaire, J., and Tang, Y. (2019). *reticulate: Interface to 'Python'*. R package version 1.13, Accessed: 2019-11-28.

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*.

Vardi, Y., Shepp, L. A., and Kaufman, L. (1985). A statistical model for positron emission tomography. *Journal of the American Statistical Association*, 80(389):8–20.

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. *Nature Genetics*, 42(7):579.

Vũ, B. C. (2013). A splitting algorithm for dual monotone inclusions involving cocoercive operators. *Advances in Computational Mathematics*, 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™*, pages 167–188. Springer.

Wang, Y.-X. and Zhang, Y.-J. (2013). Nonnegative matrix factorization: A comprehensive review. *IEEE Transactions on Knowledge and Data Engineering*, 25(6):1336–1353.

Whitehead, N. and Fit-Florea, A. (2011). Precision & performance: Floating point and IEEE 754 compliance for NVIDIA GPUs. *Technical Report, Nvidia Corporation*.

Wu, T. T. and Lange, K. (2010). The MM alternative to EM. *Statistical Science*, 25(4):492–505.

Xianyi, Z., Qian, W., and Chothia, Z. (2014). OpenBLAS: An optimized BLAS library. Accessed: 2019-11-28.

Xin, B., Kawahara, Y., Wang, Y., and Gao, W. (2014). Efficient generalized fused lasso and its application to the diagnosis of Alzheimer's disease. In *AAAI*, pages 2163–2169.

Xue, L., Ma, S., and Zou, H. (2012). Positive-definite ℓ_1-penalized estimation of large covariance matrices. *Journal of the American Statistical Association*, 107(500):1480–1491.

Yan, M. (2018). A new primal–dual algorithm for minimizing the sum of three functions with a linear operator. *Journal of Scientific Computing*, 76(3):1698–1717.

Yoo, A. B., Jette, M. A., and Grondona, M. (2003). Slurm: Simple linux utility for resource management. In *Workshop on Job Scheduling Strategies for Parallel Processing*, pages 44–60. Springer.

Yu, D., Won, J.-H., Lee, T., Lim, J., and Yoon, S. (2015). High-dimensional fused lasso regression using majorization–minimization and parallel processing. *Journal of Computational and Graphical Statistics*, 24(1):121–153.

Yu, H. (2009). Rmpi: interface (wrapper) to MPI (message-passing interface).

Yuan, M. and Lin, Y. (2006). Model selection and estimation in regression with grouped variables. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 68(1):49–67.

Yurtsever, A., Vũ, B. C., and Cevher, V. (2016). Stochastic three-composite convex minimization. In *Advances in Neural Information Processing Systems*, pages 4329–4337.

Zhao, R. and Cevher, V. (2018). Stochastic three-composite convex minimization with a linear operator. In *International Conference on Artificial Intelligence and Statistics*, pages 765–774.

Zhong, W. and Kwok, J. (2014). Accelerated stochastic gradient method for composite regularization. In *International Converence on Artificial Intelligence and Statistics*, pages 1086–1094.

Zhou, H., Lange, K., and Suchard, M. A. (2010). Graphics processing units and high-dimensional optimization. *Statistical Science*, 25(3):311.

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

Zhu, Y. (2017). An augmented ADMM algorithm with application to the generalized lasso problem. *Journal of Computational and Graphical Statistics*, 26(1):195–204.

Zou, H. and Hastie, T. (2005). Regularization and variable selection via the elastic net. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 67(2):301–320.

국문초록

지난 10년간의 하드웨어와 소프트웨어의 기술적인 발전은 고성능 컴퓨팅의 접근장벽을 그 어느 때보다 낮추었다. 이 학위논문에서는 병렬화 용이하고 역행렬 연산이 없는 변수 분리 알고리즘과 그 통계계산에서의 구현을 논의한다. 첫 부분은 볼록 함수 두 개 또는 세 개의 합으로 나타나는 구조화된 희소 통계 추정 문제에 대해 다룬다. 이 때 함수들 중 하나는 비평활 함수와 선형 함수의 합성으로 나타난다. 그 예시로는 그래프 구조를 통해 유도되는 희소 융합 Lasso 문제와 한 변수가 여러 그룹에 속할 수 있는 그룹 Lasso 문제가 있다. 이를 풀기 위해 역행렬 연산이 없는 두 종류의 원시-쌍대 (primal-dual) 알고리즘을 단조 연산자 이론 관점에서 통합하며 이를 통해 병렬화 용이한 precondition된 전방-후방 연산자 분할 알고리즘의 집합을 제안한다. 이 통합은 점근적으로 최적 수렴률을 갖는 가속 알고리즘의 집합을 구성하는 데 활용된다. 두 번째 부분에서는 PyTorch와 Julia를 통해 사용하기 쉬운 분산 행렬 자료 구조를 제시한다. 이 구조는 사용자들이 코드를 한 번 작성하면 이것을 노트북 한 대에서부터 여러 대의 그래픽 처리 장치 (GPU)를 가진 워크스테이션, 또는 클라우드 상에 있는 슈퍼컴퓨터까지 다양한 스케일에서 실행할 수 있게 해 준다. 아울러, 이 자료 구조를 비음 행렬 분해, 양전자 단층 촬영, 다차원 척도법, ℓ_1-벌점화 Cox 회귀 분석 등 다양한 병렬화 가능한 통계적 문제에 적용한다. 이 예시들은 8대의 GPU가 있는 워크스테이션과 720개의 코어가 있는 클라우드 상의 가상 클러스터에서 확장 가능했다. 한 사례로 400,000명의 대상과 500,000개의 단일 염기 다형성 정보가 있는 UK Biobank 자료에서의 제2형 당뇨병 (T2D) 발병 나이를 ℓ_1-벌점화 Cox 회귀 모형을 통해 분석했다. 500,000개의 변수가 있는 모형을 적합시키는 데 50분 가량의 시간이 걸렸으며 알려진 T2D 관련 다형성들을 재확인할 수 있었다. 이러한 규모의 전유전체 결합 생존 분석은 최초로 시도된 것이다.

주요어: 단조 연산자 이론, 원시-쌍대 알고리즘, 고성능 컴퓨팅, 다중 GPU, 분산 컴퓨팅, 클라우드 컴퓨팅
학번: 2014-30997