High Performance Machine Learning through Codesign and Rooflining

by

Huasha Zhao

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Engineering - Electrical Engineering and Computer Sciences

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor John F. Canny, Chair
Professor Anthony D. Joseph
Associate Professor Haiyan Huang

Fall 2014

High Performance Machine Learning through Codesign and Rooflining

Abstract

High Performance Machine Learning through Codesign and Rooflining

by

Huasha Zhao

Doctor of Philosophy in Engineering - Electrical Engineering and Computer Sciences

University of California, Berkeley

Professor John F. Canny, Chair

Machine learning (ML) is a cornerstone of the new data revolution. Most attempts to scale machine learning to massive datasets focus on parallelization on computer clusters. The BIDMach project instead explores the untapped potential (especially from GPU and SIMD hardware) inside individual machines. Through careful codesign of algorithms and "rooflining", we have demonstrated multiple orders of magnitude speedup over other systems. In fact, BIDMach running on a single machine exceeds the performance of cluster systems on most common ML tasks, and has run computer-intensive tasks on 10-terabyte datasets. We can further show that BIDMach runs at close to the theoretical limits imposed by CPU/GPU, memory or network bandwidth. BIDMach includes several innovations to make the data modeling process more agile and effective: likelihood "mixins" and interactive modeling using Gibbs sampling.

These results are very encouraging but the greatest potential for future hardware-leveraged machine learning appears to be on MCMC algorithms: We can bring the performance of sample-based Bayesian inference up close to symbolic methods. This opens the possibility for a general-purpose "engine" for machine learning whose performance matches specialized methods. We demonstrate this approach on a specific problem (Latent Dirichlet Allocation), and discuss the general case.

Finally we explore scaling ML to clusters. In order to benefit from parallelization, rooflined nodes require very high network bandwidth. We show that the aggregators (reducers) on other systems do not scale, and are not adequate for this task. We describe two new approaches, butterfly mixing and "Kylix" which cover the requirements of machine learning and graph algorithms respectively. We give roofline bounds for both approaches.

To my parents.

Contents

List of Figures

List of Tables

Acknowledgments

It is a great pleasure to spend the past four years at Berkeley. I am very honored and grateful to have Professor John Canny as my advisor during my PhD study. John is a great advisor and mentor who provides both visionary guidance and hands-on instructions. His advice is always constructive and extremely helpful, and I enjoy every single meeting with him. He is also incredibly patient and tolerant for my mistakes. I give my most heartfelt thanks to John.

I feel fortunate to also have Anthony Joseph and Haiyan Huang on my dissertation committee - I want to thank Anthony who, throughout my study, has provided many insightful advices on problem formulation, choice of technologies, and thesis writing, and Haiyan who has taught me the statistics foundations on which many of the applications in this thesis are based. In addition, I also owe my gratitude to Peter Bartlett, Pieter Abbeel, Kurt Keutzer and Venkat Anantharam, for their guidance on problem solving and encouragement to explore new frontiers. Finally, I want to give special thanks to Kannan Ramchandran for recruiting me to this program and for his great mentoring and support during my first year of study.

My thanks also go to Tom Barrett from Goldman Sachs and Ye Chen from Microsoft, for offering the internship opportunities at the two great companies. The internships connect my research to the industry applications that have direct business impact. They also offer inspirations on new research directions. In addition, the work experience makes my life as a graduate student more exciting and colorful.

This dissertation marks the end of my twenty years of school life - from compulsory education, to high school, to Tsinghua, and to Berkeley. I would like to thank all the teachers and fellow students who has taught me the knowledge, courage and perseverance that lay the foundation of the accomplishment of the thesis.

Finally, I am deeply indebted to my parents and my family for their love and patience. I could not have gotten started without them.

Chapter 1

Introduction

Machine learning has been a key component of modern data analytics. With the rapid growth of the amount of data, machine learning algorithms are required to run at very large scale. Our goal is to develop tools that supports such analysis as fast and efficiently as possible. Many groups are developing tools on clusters. While cluster approaches have produced useful speedups, they have generally not leveraged single machine performance. In fact, efficiency, measured as operations/watt or operations/unit cost, cannot be improved by scaling up on clusters: since performance scaling is inevitably sub-linear with cluster size, sufficiency goes down.

We design and develop BIDMach with the philosophy of focusing on single machine speed first and cluster scale-up second. Two key design ideas are incorporated in the development of our tool: codesign, which means choosing the right combination of hardware (CPU/GPU, memory, power and network) and designing algorithms to best leverage the hardware, and roofline design, which means quantifying the theoretical performance limits of each component of a machine learning algorithm and making sure to get close to them.

Full performance of single node is achieved using GPUs. We argue that GPUs are now mature and fully-functional data analysis engines, and offer large gains on most machine learning algorithms. To further scale up accelerated node, we develop two algorithm-aware network protocols butterfly mixing and Kylix for dense (relatively small) and sparse (power law) model updates respectively. We conclude the thesis with a case of building a graphical model inference system (using Gibbs sampling) on top of BIDMach.

BIDMach is currently the fastest system for a large and growing list of machine learning tasks (see Chapter 2). On a single GPU-equipped node, BIDMach outperforms the fastest cluster systems running on up to a few hundred nodes for these problems. BIDMach also scales well, and has run topic models with hundreds of topics on several terabytes of data.

In the following chapters, we present our design choices and principles that enable such performances. They include system design choices that fully unleash the hardware computing power for machine learning and algorithm design patterns to improve performance of two most powerful machine learning solvers, i.e. Stochastic Gradient Descent and Markov chain Monte Carlo.

Chapter 2 and 3 presents two key design ideas that guides the overall development of BIDMach: codesign, which means choosing the right combination of hardware and designing algorithms to best leverage the hardware, and roofline design, which means quantifying the theoretical performance limits of each component of a ML algorithm and making sure to get close to them. When implementing a algorithm, we need its hardware mappings. This includes computation pattern required by the algorithm, memory/IO pattern and communication pattern when designing a distributed algorithm.

To reach performance limits, careful and often iterative design and coding is needed. This is time-consuming. It would be problematic to do such optimization for every machine learning algorithm. Instead, we create an intermediate layer - a set of common computation and communication kernels/primitives, between BIDMach and the hardware. This includes BIDMat, butterfly mixing and Kylix. These deal respectively with matrix algebra, model synchronization across the network, and algorithms on graphs. An important role of BIDMat is to manage memory with matrix caching, which we will describe shortly. The software stack is shown in Figure 3.1.

Chapter 4 and 5 applies the design principles to Gibbs sampling - a very general method for Bayesian inference and demonstrates great benefit. Gibbs sampling is simple to apply, but has several limitations when used for parameter estimation, and is often much slower than non-sampling inference methods. SAME (State Augmentation for Marginal Estimation) [75, 29] is an approach to MAP parameter estimation which gives improved parameter estimates over direct Gibbs sampling. SAME can be viewed as cooling the posterior parameter distribution and allows annealed search for the MAP parameters, often yielding very high quality (lower loss) estimates. But it does so at the expense of additional samples per iteration and generally slower performance. On the other hand, SAME dramatically increases the parallelism in the sampling schedule, and is an excellent match for modern (SIMD) hardware. In this paper we explore the application of SAME to graphical model inference on modern hardware. We show that combining SAME with factored sample representation (or approximation) gives throughput competitive with the fastest symbolic methods, but with potentially better quality. We describe experiments on Latent Dirichlet Allocation, achieving speeds similar to the fastest reported methods (online Variational Bayes) and lower cross-validated loss than other LDA implementations. The method is simple to implement and should be applicable to many other models. In Chapter 5, we show that the Gibbs sampling procedure can be reduced to a couple of highly optimized matrix operations (from BIDMach) on generic Bayesian network with conditional probability tables and exponential family emissions. The sampling routine can therefore leverage on full hardware accelerations we have developed so far, and is much faster comparing with other systems we have benchmarked.

Finally we explore scaling ML to clusters. Chapter 6 describes bufferfly mixing which shows how to scale up BIDMach for incremental model-update algorithms with relatively small models. Incremental model-update strategies are widely used in machine learning and data mining. By "incremental update" we refer to models that are updated many times using small subsets of the training data. Two well-known examples are stochastic gradient and MCMC. Both provide fast sequential performance and have generated many of the

best-performing methods for particular problems (logistic regression, SVM, LDA etc.). But these methods are difficult to adapt to parallel or cluster settings because of the overhead of distributing model updates through the network. Updates can be locally batched to reduce communication overhead, but convergence typically suffers as the batch size increases. In this paper we introduce and analyze *butterfly mixing*, an approach which *interleaves* communication with computation. We evaluate butterfly mixing on stochastic gradient algorithms for logistic regression and SVM, on two datasets. Results show that butterfly mix steps are fast and failure-tolerant on an Amazon EC2 cluster.

Chapter 7 describes Kylix - a Sparse Allreduce primitive that scales models at web scale. Allreduce is a basic building block for parallel computing. Our target here is "Big Data" processing on commodity clusters (mostly sparse power-law data). Allreduce can be used to synchronize models, to maintain distributed datasets, and to perform operations on distributed data such as sparse matrix multiply. We first review a key constraint on cluster communication, the minimum efficient packet size, which hampers the use of direct all-to-all protocols on large networks. Our allreduce network is a nested, heterogeneous-degree butterfly. We show that communication volume in lower layers is typically much less than the top layer, and total communication across all layers a small constant larger than the top layer, which is close to optimal. A chart of network communication volume across layers has a characteristic "Kylix" shape, which gives the method its name. For optimum performance, the butterfly degrees also decrease down the layers. Furthermore, to efficiently route sparse updates to the nodes that need them, the network must be nested. While the approach is amenable to various kinds of sparse data, almost all "Big Data" sets show power-law statistics, and from the properties of these, we derive methods for optimal network design. Finally, we present experiments showing with Kylix on Amazon EC2 and demonstrating significant improvements over existing systems such as PowerGraph and Hadoop.

Chapter 8 concludes the thesis. The BIDMach code is open-source and freely available on github, `https://github.com/BIDData/BIDMach/wiki`. We also have a public image for launching GPU instances on Amazon EC2, available at our project website, `http://bid2.berkeley.edu/bid-data-project/download/`.

Chapter 2

Hardware Software Co-design

2.1 Introduction

The primary motive for the BID Data Suite is *exploratory data analysis*. We argue that data analysis is inevitably a complex and iterative process - models can be continually refined and improved through exploration. The faster this exploration, the more thoroughly and rapidly a solution space can be explored. Furthermore, real-world data analysis problems are not "pure" and the best solution is usually a mixture of many simpler models. For instance, on surely the most well-explored behavioral data mining task, all of the most successful Netflix® contest competitors were complex hybrids [8]. Finally, complex models require substantial parameter tuning, which today is an ad-hoc process. There is need for tools that support both manual and automatic exploration of the solution space.

A secondary goal is to support *rapid deployment and live tuning of models* in commercial settings. That is, there should be a quick, clean path for migrating prototype systems into production. The prototype-production transition often results in degradations in accuracy to meet performance constraints. To better support this transition, the performance of the prototype system must meet or exceed that of other production systems, and so high performance is a necessary component of the BID Data Suite.

The BID Data Suite is designed to work in single-user mode, or in small clusters. We argue next that this usage model supports most "Behavioral Data" problems, and many other problems that fit under a petabyte.

The Scale of Behavioral Data

By behavioral data we mean data that is generated by people. This includes the web itself, any kind of text, social media (Facebook®, Twitter®, Livejournal), digital mega-libraries, shopping (Amazon®, Ebay®), tagging (Flickr®, Digg®) repositories (Wikipedia, Stack Overflow), MOOC data, server logs and recommenders (Netflix®, Amazon® etc). These datasets have an enormous variety of potential uses in health care, government, education, commerce, and cover the larger part of the commercial applications of data mining,

and a fair slice of research applications. We exclude data in specialized formats like images and videos, and high-density sensor data. The later require specialized processing and are often orders of magnitude larger. However, meta-data extracted from images and video (who is in them, what is being said etc.) are very much back in the realm of behavioral data. The figure below gives a feel for the scale of these datasets:

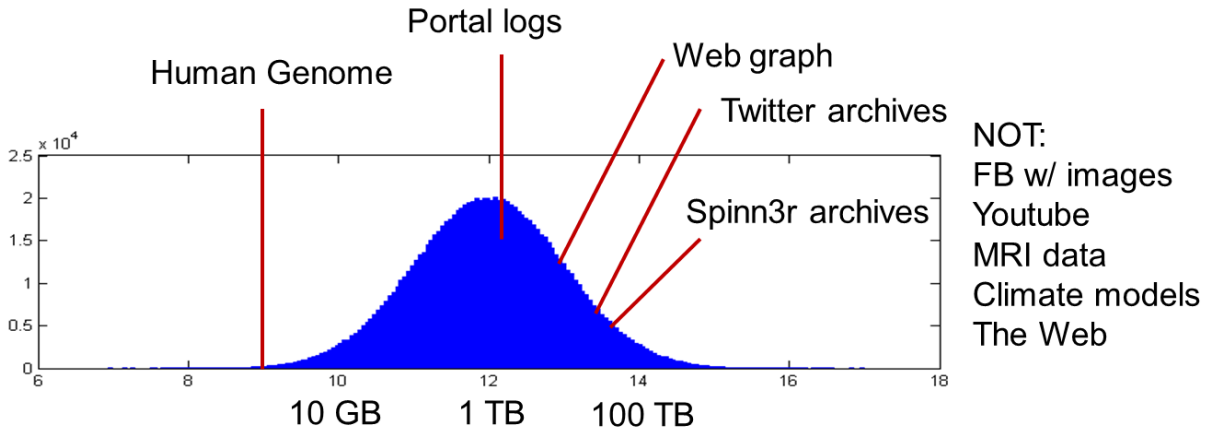


Figure 2.1: The scales of behavioral data

Spinn3r is an aggregator of all the blogs, news, and news-related social media posts around the world. FB is FaceBook. We placed "portal logs" in the middle of the curve acknowledging that many web server clusters generate much larger log sets. However, these logs comprise an enormous amount of un-normalized meta-data, and encode relatively few behaviorally relevant events (e.g. the presence of a particular image on the page). Once simplified and featurized, the logs are much smaller and in the terabyte range for most portals. There are only 6 billion people on earth, and a small fraction are active on computers generating data at any given time. This places a fairly stable limit on the amount of behavioral data that will be generated over time. These dataset sizes are practical to process on suitably configured single machines or small clusters. A key aspect of behavioral data is that it involves a large number of (ideally) interchangable samples - the data from particular people - or individual documents. This enables a very fast family of model optimization strategies to be integrated into the suite, at the same time supporting a very broad (and extensible) set of model designs.

Data Mining Workloads

Most large-scale data mining workloads tend to be either I/O bound or compute-bound. Since these are the two primary resources this is almost a tautology. But there is more to it, and in practice they are either *heavily* I/O bound or *heavily* compute bound. By this we mean that the fraction of time spent in one phase is much larger than the other. Examples of I/O bound workloads are regression and linear classification. Examples of compute-bound

workloads are clustering models, factor models (aka matrix completion), random forests and ensemble models. In the first case, we touch each input datum "lightly" (i.e. once or twice). Since computation (Gflops-Teraflops) is much faster than I/O (0.1-1GB/s), there is a big disparity between data retrieval and use. For compute-bound workloads, we touch each datum many times. Compute-bound models are much more complex, and if they are valuable at all, their value tends to improve with complexity. Complexity is tuned as high as possible - and is often set by machine performance limits. Thus each datum is used many times - often thousands - and the overall time spent processing a datum dominates the time to retrieve it.

Patterns for Model Optimization

In recent years, two of the simplest model inference algorithms: Stochastic Gradient Descent (SGD) and Gibbs Sampling (GS) have been refined to the point that they often provide the best speed and acceptable accuracy for many inference problems. Machine learning has drawn ever more heavily on optimization theory, which has lead to much better scaling/projection strategies for SGD with provably fast convergence. SGD and GS are now preferred tools for a very wide range of gradient-based model optimizations. By developing very fast local and distributed code for SGD and GS, we can support the rapid development and optimization of existing or new models. Convergence of these algorithms has improved to the point where on large datasets, convergence is reached in few passes over the data and sometimes in less than one pass. For this reason, our preferred I/O design pattern is to read the dataset directly from disk for each pass over it (if more than one is needed), and process it in blocks - both SGD and GS typically perform equally well when gradients are updated on small blocks rather than individual samples.

SGD and GS are naturally sequentially algorithms and parallelization has been a challenge. Recently however, we demonstrated a new technique called Butterfly mixing [96] which achieves a typical order-of-magnitude reduction in communication time with minimal impact on convergence. By supporting Butterfly Mixing in the toolkit we free the designer from having to distribute a complex algorithm, instead requiring only callbacks that compute either gradient of loss on a block of data (for SGD) or change in likelihood for a given state change (for GS). Butterfly mixing is an example of the "loop interleaving" pattern we will describe shortly.

Impact of GPU acceleration

Graphics Processors contrast with CPUs in many ways, but the most important for us are:

- Much higher levels of parallelism - the number of cores is in the thousands.
- Significantly higher memory bandwidth.
- Low-level support for transcendental functions (exp, log etc)

GPUs have seen much service in scientific computing, but several factors have hampered their use as general computing accelerators, especially in machine learning. Primarily these were slow transfers from CPU to GPU memory, and limited GPU memory size. However, transfer speeds have improved significantly thanks to PCI-bus improvements, and memory capacity now is quite good (2-4 GBytes typical). GPU development has also focused most on optimizing single-precision floating point performance, and for typical data mining tasks this is adequate. In fact GPUs now provide significant (order-of-magnitude) improvements in almost every step that bottlenecks common machine learning algorithms. The table below lists some performance details from the Nvidia GTX-690 cards used in our prototype data engine.

Task	Java	1C	8C	1G	4G	imp
CPU copy(gbs)	8	12	*	*	*	*
GPU copy(gbs)	*	*	*	150	*	*
CPU-GPU(gbs)	*	12	*	12	*	*
8k SGEMM(gf)	2	44	270	1300	3500	10^3
SPMV(gf)	1	1	9	0.3	1.1	1
512 SPMM(gf)	1	1	6	30	100	100
exp,ln(ge)	0.02	0.3	4	10	35	10^3
rand(ge)	0.08	0.07	0.8	26	90	10^3

In the table above, we contrast the performance of standard Java builtins, accelerated CPU primitives (using Intel® MKL) and GPU primitives (using Nvidia® CUDA). The machine has a single 8-core CPU (Intel E5-2660) and 4 GPUs (two Nvidia GTX-690s). 1C denotes MKL-accelerated performance with a single CPU thread. 8C gives accelerated performance with multi-threading enabled (speedups are sometimes ¿ 8 since the CPU has 16 hyper-threads). 1G gives the performance on one GPU, while 4G is performance with all 4 GPUs running.

The test tasks include memory operations (scored in gbs = GigaBytes/sec), matrix arithmetic (scored in gf = Gigaflops/sec), and transcendental function and random number generation (scored in ge = billion elements per sec). All operations are single-precision. 8k SGEMM is dense matrix-matrix multiply of two 8k x 8k matrices. SPMV is sparse matrix-vector product. 512 SPMM is sparse matrix / dense matrix product where the dense matrix has 512 columns. This can also be treated as 512 SPMV operations in parallel (important for multi-model inference). These tasks were chosen to represent common bottleneck operations in most data mining tasks.

Note the large speedups from GPU calculation (10^2-10^3) for all compute tasks except SPMV. For most problems, these gains are practical and realizable. Note also that (multi-threaded) CPU acceleration by itself accounts for large speedups, and does better than the GPU at SPMV. Both CPU and GPU acceleration can be applied in cluster environments but care must be taken if multiple tasks are run on the same machine - sharing main memory can easily destroy the coherence that makes the CPU-accelerated primitives fast. Since GPUs each have separate memories, it is actually simpler to share them among cluster tasks.

The exception is SPMV which is the "bottleneck" for regression and SVM models computed from sparse data. However, when the data are being read from disk, I/O time dominates calculation and so any performance gains from good SPMV performance are lost. In fact we believe this creates an opportunity: to do as much computation as possible on the I/O stream without slowing it down. The data in the table imply that one can do blocked SPMM (512 columns) in real-time on a single-disk data stream at 100 MB/s using 4 GPUs. In other words, on a 4-GPU system, one can run 512 regression models in the same time it takes to run a single model. One can make a training pass with about 256 models in this time. One goal of the BID Data project is to make ensemble and other multi-model learning strategies as natural as regression, since with suitable tools they cost no more time to run. Normally there will be accuracy gains from ensemble learning on behavioral datasets, and there is no practical penalty to doing so.

But even if the goal is to deliver just one regression model, we can use the available cycles to speed up learning in a variety of ways. The first is simple rate exploration: it is known that SGD algorithm convergence is quite sensitive to the gradient scaling factor, and the optimal factor changes with time. By running many models in parallel with different constants, one can find the best at each step (through cross-validation) and evolve the model using the best constant at each time.

A second approach is to use limited-memory second order methods such as online L-BFGS which fit neatly in the computational space we have $O(kd)$ with d feature dimensions and a memory of the last $k < 512$ gradients. These methods have demonstrated two- to three- orders of magnitude speedup compared to simple first-order SGD [15].

2.2 The Suite

The Data Engine

As we saw above, typical data mining workloads are either I/O bound or compute-bound. By contrast, both the cost and power consumption of a typical cluster server are concentrated in the CPU+memory combination. This is an order of magnitude higher in both cost and power than typical storage or GPU (although we are conflating cost and power, in recent years energy prices imply that the TCO for servers is increasingly dominated by power consumption anyway). Yet the I/O performance of a server is limited by the disk bandwidth (or network bandwidth if storage is distributed), and as we argued above compute performance for DM workloads is dictated by the GPU. A *data engine* is a commodity workstation (PC) which is optimized for a mix of I/O bound and compute-bound operations.

Note that relative to a standard cluster server, a Data engine achieves an order of magnitude increase in TB/watt or TB/dollar. It also achieves (at least) and order of magnitude increase in TFlops/watt or TFlops/dollar. By using a large array of disks in JBOD/Software RAID configuration, the performance on I/O bound operations is similarly improved by at least an order of magnitude.

We have built a data engine prototype in a 4U rackmount case. It was used for all the non-cluster benchmarks in this article. This version includes a single 8-core CPU (Intel E5-2660) with 64 GB ram, 20 x 2TB Sata disks with a JBOD controller, and two dual-GPUs (Nvidia GTX-690), i.e. four independent GPUs. Total cost was under $8000. This design is almost perfectly balanced in terms of the cost ($2000 for each of CPU/mem, GPUs and storage). Power is approximately balanced as well. The GPUs can draw 2-3x more power during (fairly rare) full use of all GPUs. This version fits in a 4U rack although more exotic case combinations allow the same contents to fit in a 2U space.

We have not seen a similar configuration among cloud servers, although Amazon now offers both high-storage and GPU-assisted instances. These are suitable separately for I/O-bound or compute-bound workloads, and also leverage the software tools described below.

BIDMat: A Fast Matrix Toolkit

The discussion above implies several desiderate for a machine learning toolkit. One consequence of our goal of agility is that algorithms should be expressed at high-level in terms of familiar mathematical objects (e.g. matrices), and to the extent possible, implementation details of low-level operations should be hidden. The very widespread use of Matlab®, R, SciPy etc. underscores the value of having a matrix layer in a learning toolkit. Therefore we began by implementing the matrix layer with these goals:

Interactivity: Interactivity allows a user to iteratively perform calculations, inspect models, debug, and build complex analysis workflows one step at a time. It allows an analyst to "touch, feel, and understand" data. It was a rather easy choice to go with the Scala language which includes an efficient REPL.

Natural Syntax: Code that looks like the mathematical description it came from is much easier to develop, debug and maintain. It is also typically much shorter. Scala includes familiar mathematical operators and allows new operators to be defined with most combinations of non-alphanumeric characters (e.g. `+@, *!, *|`)

CPU and GPU acceleration: As we saw earlier, GPU performance now strongly dominates CPU performance for most expensive operations in machine learning. BIDMat uses GPU acceleration in several ways, including GPU-resident matrix types, and operators that use the GPU on CPU-resident data. Generic matrices allow algorithms to be written to run in either CPU or GPU.

Simple Multi-threading: BIDMat attempts to minimize the need for explicit threading (matrix primitives instead are multithreaded), but still it is important for non-builtin CPU or GPU functions. Scala has an extremely elegant and simple threading abstraction (actors) which support multi-threaded coding by non-expert programmers.

Reuse: Since Scala targets the Java Virtual Machine and can call Java classes, we are able to reuse an enormous codebase for: (i) GPU access using the JCUDA library, (ii) File I/O

using HDF5 format and the hdf5-java library or hdfs via hadoop, (iii) communication using the JMPI library and (iv) clustering using JVM-based tools like Hadoop, Spark, Hyracks etc.

A number of custom kernels (functions or operators) were added to the library to remove bottlenecks when the toolkit was applied to particular problems. These are described next:

Custom Kernels

Custom kernels are non-standard matrix operations that provide significant acceleration for one or more learning algorithms. We discuss three here:

Sampled Dense-Dense matrix product
The sampled dense-dense matrix product (SDDMM) is written

$$P = A *_S B = (AB) \circ (S > 0)$$

Where A and B are respectively $m \times p$ and $p \times n$ dense matrices, S is an $m \times n$ sparse matrix, and $\circ$ is the element-wise (Hadamard) product. $S > 0$ denotes a matrix which is 1 at non-zeros of S and zero elsewhere. P is also an $m \times n$ sparse matrix with the same nonzeros as S. Its values are the elements of the product AB evaluated at the nonzeros of S, and zero elsewhere. SDDMM is a bottleneck operation in all of the factor analysis algorithms (ALS, SFA, LDA and GaP) described later. In each case, S is a input data matrix (features x users, or features x docs etc.) and is extremeley sparse. Direct evaluation of AB is impractical. Naive (Java) implementation of SDDMM only achieves about 1 Gflop, while CPU- and GPU-assisted custom kernels achieve around 9 Gflops (8C), 40 Gflops (1G) and 140 Gflops (4G) respectively. Since SDDMM is the bottleneck for these algorithms, the speedups from custom kernels lead to similar speedups in overall factor model performance. These are discussed in the benchmark section later.

Edge Operators
Most matrix toolkits support scalar arguments in elementwise operators such as $C = A \otimes B$ where $\otimes$ can be $+$, $-$, $\circ$ (elementwise product) etc. However, a common operation on matrices is to scale all the rows or columns by a constant, or to add row- or column-specific constants to a matrix. The BIDMat library realizes these operations with *edge operators*. If A is an $m \times n$ matrix, it is legal to perform an element-wise operation on a B which is either:

An $m \times n$ matrix The standard element-wise operation.

An $m \times 1$ column vector Applies the operation $\otimes$ to each of m rows of A and the corresponding element of B.

An $1 \times n$ row vector Applies the operation $\otimes$ to each column of A and the corresponding element of B.

A 1×1 scalar This applies the operation $\otimes$ to B and all elements of A.

Edge operators lead to modest performance improvements (a factor of two is typical) over alternative realizations. They also lead to more compact and more comprehensible code.

Multivector Operators
One of our goals is to support efficient multi-model and ensemble learning. For these tasks its more efficient to perform many same-sized vector operations with a matrix operation rather than separate vector operations. An $m \times n$ *multivector* is a set of n m-vectors. We currently represent $m \times n$ multvectors as normal $m \times n$ matrices. Multivector operators perform the same operation on each of the n vectors. In fact most multivector operations are already realized through edge operators.

For instance, suppose we wish to multiply a scalar a and an m-vector b. We can perform this operation on n vectors using an $n \times 1$ matrix A representing the values of a in each submodel, and an $m \times n$ multivector B representing the values of b. In BIDMat this is written

$$C = A \ *@ \ B$$

since $*@$ is element-wise multiplication (this same operator performs scalar multiplication if applied to a scalar a and a vector b). The use of edge operators means that in most cases, the single-model and multi-model Scala code is identical.

Multivectors are assumed to be stored as columns. This forces a unique interpretation of vector operators like the dot poduct (literally "dot" in BIDMat) and norm when applied to matrices:

$$u \ dot \ v = sum(u \circ v, 1) \qquad norm(u) = \sqrt{sum(u \circ u, 1))}$$

where $sum(\cdot, 1)$ denotes the sum over the first dimension.

BIDMach: Local/Peer Machine Learning

BIDMach is a machine learning toolkit built on top of BIDMat. BIDMach may evolve into a general-purpose machine learning toolkit but for now it is focused on large-scale analytics with the most common inference algorithms that have been used for behavioral data: regression, factor models, clustering and some ensemble methods. The main elements are:

Frequent-update inference Stochastic gradient optimizers and MCMC, especially Gibbs samplers, have emerged as two of the most practical inference methods for big data. BIDMach provides generic optimizers for SGD and Gibbs sampling. Both methods involve frequent updates to global models (ideally every sample but in practice on mini-batches). The high cost of communication to support minibatch updates on a cluster at the optimal rate led us to develop a new method called *Butterfly Mixing* described later. We are also developing lower-level kernels to support exploration of, and improvements to these methods. In particular, GPU-based parametric random number generators enable substantial speedups to common Gibbs sampler patterns.

Multimodel and Ensemble Inference The BIDMat library provides low-level primitives to support many-model inference in a single pass over the dataset. BIDMach leverages this to support several higher-level tasks: (i) parameter exploration, where different instances of the same model are run with different parameter choices, (ii) parameter tuning, where different initial parameter choices are made, and the parameter set is periodically changed dynamically (i.e. contracted to a small range around one or more optima at some stage during the optimization) based on the most recent loss estimates, (iii) k-fold cross-validation where k models are trained in parallel, each one skipping over an n/k-sized block of training data, (iv) ensemble methods (especially bagging) which are built on many independent models which are trained in parallel. Advanced ensemble methods such as super-learning [57] use several of these techniques at once.

Algorithm Design Patterns

In our experience implementing machine learning algorithms with the BID Data toolset, its clear that tools by themselves are not enough. Its rather the application of appropriate design patterns that leverage the individual tools. One non-obvious pattern that we have had several successes with is *loop interleaving*, illustrated schematically in figure 2.2. The pattern begins with an overall iteration with an expensive block step, show at left. The block step is *fragmented* into smaller substeps (middle), which are then *interleaved* with the steps in the main iteration (right). The net effect is that it is possible to do many more iterations in a given amount of time, while there is *typically* enough similarity between data across iterations that the substeps still function correctly.

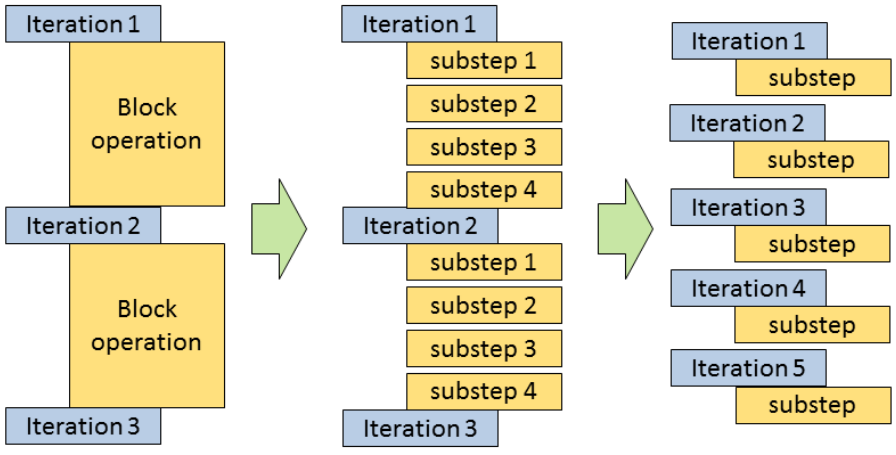


Figure 2.2: The loop interleaving pattern

The loop-interleaving pattern is deliberately presented at high-level. It is not a formalizable construction. The two applications we give of it require their own proofs of soundness, which use completely different methods. Intuitively, the method will often work because

the model is changing by small increments in the main loop, and iterative methods in the inner loop can use state from the previous (outer) iteration that will be "close enough". And further there is little need to complete the inner loop iterations (middle part of the figure) since the final model will change anyway in the next outer iteration.

Butterly Mixing

The first instantiation of the loop interleaving pattern is *butterfly mixing*, a technique for rapid sharing of stochastic gradient or MCMC updates in a cluster. The problem is succinctly captured with a specialized version of the loop-interleaving pattern. In the figure below, the left figured shows the time trajectory of a gradient update step, and a distributed "allreduce" step. Allreduce is a parallel primitive in which an aggregate (in this case the average) of all locally-updated models is computed and distributed to all machines. Allreduce is quite expensive in practice (one-two orders of magnitude slower than the local updates), and its cost grows with the size of the network. The idea is to fragment the allreduce into a series of simpler (in this case constant time) operations, namely the individual layers of a butterfly communication network. See the figure below.

A butterfly network calculates an allreduce in optimal time $O(\log_2(n))$ steps for a network with n nodes. Then the butterfly steps are interleaved with gradient steps in the right part of the figure above. This arrangement is show layered on the network below.

Its clear that this approach dramatically reduces communication overhead compared to allreduce every step (part (a) of the figure), and many more gradient updates are being done in a given amount of time. Its less clear what impact this has on convergence. In [96] we showed that this impact is minimal and in fact the convergence (in number of update steps) of butterfly mixing on typical datasets is almost as fast as an allreduce at every step.

Butterfly mixing has many practical advantages. It requires only synchronization between communicating pairs of nodes. It is extremely simple to implement: we implemented it in MPJ (a java implementation of MPI), which required only blocking SendReceive calls on each pair of communicating nodes. Unlike tree allreduce, there are no "hot" edges that can lead to high latency through the entire network. Each communication step is constant time (at least through a single switch). Butterfly mixing does require 2n messages be exchanged between n nodes, which is not an issue for rack-level clusters, but does become prohibitive in larger clusters. In future work, we expect to experiment with hybrid topologies of full butterfly exchanges (within rack) backed with "thinner" butterfly exchanges (using subsets of nodes in each rack) across racks.

2.3 Benchmarks

Pagerank

Give the adjacency matrix G of a graph on n vertices with normalized columns (to sum to 1), and P a vector of vertex scores, the Pagerank iteration in matrix form is:

$$P' = \frac{1}{n} + \frac{n-1}{n} GP \tag{2.1}$$

A benchmark dataset used by several other groups for Pagerank is the Twitter® Followers graph. This graph has 60M nodes (40M non-isolated) and 1.4B edges. The graph in binary form on disk is about 18 GB. We wrote a simple Pagerank iteration which leverages the I/O performance of the Data Engine RAID, and leverages Intel® MKL acceleration for the sparse matrix-vector multiply. Single-node performance for this problem is competitive with the best reported cluster implementation.

System	Graph VxE	Time(s)	Gflops	Procs
Hadoop	?x1.1B	198	0.015	50x8
Spark	40Mx1.5B	97.4	0.03	50x2
Twister	50Mx1.4B	36	0.09	60x4
PowerGraph	40Mx1.4B	3.6	0.8	64x8
BIDMat	60Mx1.4B	6	0.5	1x8
BIDMat+disk	60Mx1.4B	24	0.16	1x8

The "Procs" column lists num nodes x num cores per node. The first line for BIDMat is performance with the entire graph in memory (18GB). The second line shows the performance including the time to read the graph from disk (about 18 seconds), showing that the RAID achieved a throughput of about 1 GB/sec. All the other systems except Hadoop use memory-resident data, and so the number of processors presumably must scale with the size of the graph. BIDMat on the other hand can handle much larger graphs that are disk resident on a single node in reasonable running time.

LDA and GaP

Latent Dirichlet Allocation [12] is a widely-used topic model. LDA models documents with a generative process in which topic distribution for each document is chosen from a Dirichlet process, topics are chosen independently word-by-work according to this distribution, and then words are finally chosen from a multinomial distribution for each topic. GaP (Gamma Poisson) is a derivative of LDA which instead models the topic mixture as contiguous bursts of text on each topic [21]. Both the original LDA and GaP models are optimized with alternating updates to topic-word and topic-document matrices. For LDA it is a variational EM iteration, for GaP it is an alternating likelihood maximization. Variational LDA was described with a simple recurrence for the E-step (Figure 6 in [12]). Following the notation

of [12], we develop a matrix version of the update. First we add subscripts j for the j^{th} document, so γ_{ij} is the variational topic parameter for topic i in document j, and ϕ_{nij} is the variational parameter for word in position n being generated by topic i in document j. Then we define:

$$F_{ij} = \exp(\Psi(\gamma_{ij})) \tag{2.2}$$

The update formula from figure 6 of [12] can now be written:

$$\gamma_{ij} = \alpha_i + \sum_{w=1}^{M} \beta_{iw} F_{ji} C_{wj} / \sum_{i=1}^{k} \beta_{iw} F_{ij} \tag{2.3}$$

where C_{wj} is the count of word w in document j. Most such counts are zero, since C is typically very sparse. The above sums have been written with w ranging over word values instead of word positions as per the original paper. This shows that LDA factorizations can be computed with bag-of-words representation without explicit word labels in each position. M is the vocabulary size, and k is the number of topics. Writing the above in matrix form:

$$\gamma' = \alpha + F \circ \left(\beta * \frac{C}{\beta^T *_C F} \right) \tag{2.4}$$

where the quotient of C by $\beta^T *_C F$ is the element-wise quotient. Only terms corresponding to nonzeros of C (words that actually appear in each document) need to be computed, hence the denominator is a SDDMM operation. The quotient results in a sparse matrix with the same nonzeros as C, which is then multiplied by β. The dominant operations in this update are the SDDMM, and the multiplication of β by the quotient. Both have complexity $O(kc)$ where c is the number of nonzeros of C. There is also an M-step update of the topic-word parameters (equation 9 of [12]) which can be expressed in matrix form and has the same complexity.

The GaP algorithm has a similar E-step. Using the notation from [21], the matrix Λ in GaP plays the role of β in LDA, while X in GaP plays the role of γ in LDA. With this substitution, and assuming rows of β sum to 1, the GaP E-step can be written:

$$\gamma' = \left(a - 1 + \gamma \circ \left(\beta * \frac{C}{\beta^T *_C \gamma} \right) \right) / \left(1 + \frac{1}{b} \right) \tag{2.5}$$

where a and b are $k \times 1$ vectors which are respectively the shape and scale parameters of k gamma distributions representing the priors for each of the k dimensions of γ. This formula is again dominated by an SDDMM and a dense-sparse multiply and its complexity is $O(kc)$. Not all the matrix operations above have matching dimensions, but the rules for edge operators will produce the correct results. So the above formulas for LDA and GaP can be entered directly as update formulas in BIDMat. LDA/GaP are compute-intensive algorithms. Fortunately, GPU implementations of the dominant steps (SDDMM and SPMM) are very efficient, achieving 30-40 gflops/sec on each GPU.

The table below compares the throughput of our variational LDA implementation with two previously-reported cluster implementations of LDA. The document sets are different in each case: 300-word docs for Smola et al. [80] and 30-word docs for PowerGraph [39]. Both methods use Gibbs samplers applied to each word and so document length is the true document length. We are able to use bag-of-words which cuts the document length typically by about 3x. The latent dimension is 1000 in all cases. We tested on a dataset of 1M wikipedia articles of average length 60. Since we used the variational method instead of Gibbs sampling, we made many passes over the dataset. 30 iterations gave good convergence. The per-iteration time was 20 seconds for 1M documents, or about 10 minutes total. Performance is given as length-normalized (to length 100) docs/second.

System	Docs/hr	Gflops	Procs
Smola[80]	1.6M	0.5	100x8
PowerGraph	1.1M	0.3	64x16
BIDMach	3.6M	30	1x8x1

The "Procs" field lists machines x cores, or for BIDMach machines x cores x GPUs. The Gibbs samplers carry a higher overhead in communication compared to the variational method, and their gflop counts are lower. We assign a total of 10 flops to each sample to represent random number generation and updates to the model counts. However, since these methods need multinomial samples there are typically many additional comparisons involved. Still the gflops counts indicate how much productive model-update work happens in a unit of time. We are currently developing some blocked, scalable random number generators for GPUs which we believe will substantially improve the Gibbs sampler numbers above. In the mean time we see that in terms of overall performance, the single-node GPU-assisted (variational) LDA outperforms the two fastest cluster implementations we are aware of. We hope to improve this result 3-4x by using the additional GPUs in the data engine.

ALS and SFA

ALS or Alternating Least Squares [52] is a low-dimensional matrix approximation to a sparse matrix C at the non-zeros of C. Its a popular method for collaborative filtering, and e.g. on the Netflix challenge achieves more than half the lift (¿ 6%) of of the winning entry. Sparse Factor Analysis (SFA) [20] is a closely-related method which uses a generative probabilistic model for the factorization. We will borrow the notation from LDA, and write $C \approx \beta^T * \gamma$ as the matrix approximation. Both methods minimize the regularized squared error at non-zeros of C

$$l = \sum (C - \beta^T *_C \ \gamma)^2 + \sum w_\beta \circ \beta^2 + \sum w_\gamma \circ \gamma^2 \tag{2.6}$$

where squares are element-wise and sums are taken over all elements. w_β and w_γ are row-vectors used to weight the regularizers. In ALS, the i^{th} element of w_β is proportional to the number of users who rated movie i, while the j^{th} element of w_γ is proportional to the number of movies rated by user j. In SFA, uniform weights are used for w_γ. Weights are applied as edge operators.

First of all, it is easy to see from the above that the gradient of the error *wrt* γ is

$$\frac{dl}{d\gamma} = 2\beta * (C - \beta^T *_C \ \gamma) + 2w_\gamma \circ \gamma \tag{2.7}$$

and there is a similar expression for $dl/d\beta$. So we can easily optimize the γ and β matrices using SGD. Similar to LDA and GaP, the gradient calculation involves an SDDMM operation and a sparse-dense matrix multiply. The complexity is again $O(kc)$. We can similarly compute the gradient *wrt* β. An SGD implementation is straightforward if the data arrays can be stored in memory. In practice since typically nsamples (users) >> nfeatures (movies) and the feature array β can fit in memory, we process the data array C in blocks of samples.

Closed Form Updates

Since ALS/SFA use a squared error loss, there is a closed form for each latent factor γ and β given the other. The alternating iteration using these expressions typically converges much faster (in number of iterations) than an SGD iteration. However, because of the irregularity of the matrix C, the closed form solution requires inversion of a different matrix for each user's γ_i, given below:

$$\gamma_i = \left(\lambda n_i I + \beta \ \mathrm{d}iag(C_i)\beta^T\right)^{-1} \beta C_i \tag{2.8}$$

where C_i is the column of data for user i, and n_i is the number of items rated by user i. There is a similar update for each column of the matrix β. Dense matrix inverses make good use of hardware acceleration, and the MKL Lapack routines on our test machine achieved over 50 gflops for the inverse. But still the complexity of closed-form ALS is high, $O(k^2c+(m+n)k^3)$ per iteration where m and n are the number of users and movies respectively. It is a full factor of k slower than LDA or GaP for a given dataset. In practice, best performance on collaborative filtering (and certainly on the Netflix dataset) is obtained at values of $k = 1000$ and above. And this is only for one iteration. While the update to β and γ is closed-form given the other, the method still requires iteration to alternately update each factor. So the $O(k^2c + (m + n)k^3)$ complexity per iteration is problematic.

Accelerating ALS/SFA with Loop Interleaving

Closed form ALS/SFA is a good match for the loop interleaving pattern. Each iteration involves an expensive step (matrix inversion) which we can break into cheaper steps by iteratively solving for γ_i, e.g. by using conjugate gradient. i.e. we solve the equation:

$$\left(\lambda n_i I + \beta \ \mathrm{d}iag(C_i)\beta^T\right)\gamma_i = M_i\gamma_i = \beta C_i \tag{2.9}$$

where M_i is the matrix in parentheses. Solving $M_i\gamma_i = \beta C_i$ using an iterative method (conjugate gradient here) requires only black-box evaluation of $M_i\hat{\gamma}_i$ for various query vectors $\hat{\gamma}_i$. It turns out we can compute all of these products in a block using SDDMM:

$$v = \lambda w_\gamma \circ \hat{\gamma} + \beta * (\beta^T *_C \ \hat{\gamma}) \tag{2.10}$$

where $v_i = M_i\hat{\gamma}_i$. The conjugate gradient updates are then performed column-wise on $\hat{\gamma}$ using v. There is a similar black box step in conjugate gradient optimization of β:

$$w = \lambda w_\beta \circ \hat{\beta} + \gamma \ *^T \ (\hat{\beta}^T *_C \ \gamma) \tag{2.11}$$

where $*^T$ is multiplication of the left operand by the transpose of the right, an operator in BIDMat. This update is not "local" to a block of user data γ, however the w for the entire dataset is the sum of the above expression evaluated on blocks of γ. Thus in one pass over the dataset, we can perform b steps of conjugate gradient updates to γ and one CG step of update to β.

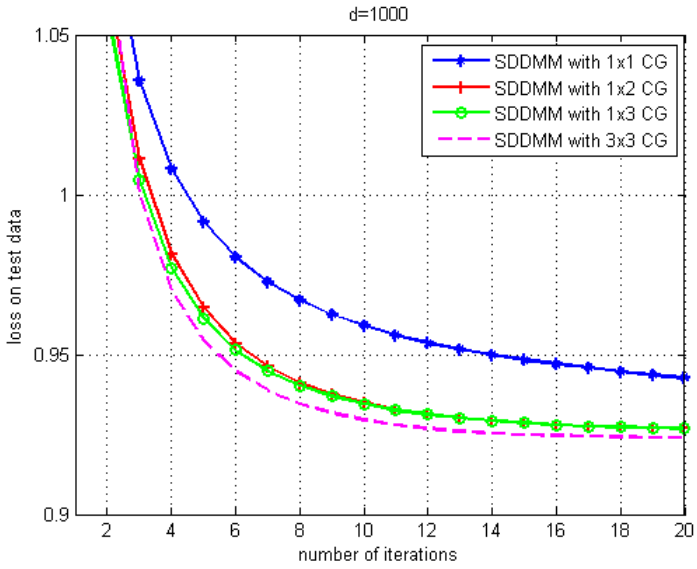


Figure 2.3: Loss vs. num iterations vs. CG updates

The complexity of one pass of conjugate gradient is then $O(ck)$ matching the complexity of LDA and GaP. Whereas normally conjugate gradient steps would be performed in sequence on one of γ or β, in our ALS/SFA implementations, we assume the data are stored only in user blocks and make a single pass over the dataset to perform both steps. The calculation in 2.10 is local to each user's data and so multiple steps can be done on each block of data read from disk. For equation 2.11 we note that the global update is a sum of updates from blocks of C, and so we accumulate the w update over the iteration. The upshot is that we can do k updates to the user model, and 1 update to the movie model, in a single pass over user-blocked data on disk. Figure 2.3 shows that multiple conjugate gradient updates to the user model only converges almost as well as multiple updates to both the movie and user models (which is turn is almost identical to convergence for the closed form solution).

The table below shows the normalized performance of the CG method and a prior method on a testset described in [39] (11M docs, 30 terms/doc). The time given is for 20 iterations

of BIDMach which is adequate for convergence. We note that there is both a machine-performance difference and an asymptotic difference. ALS as implemented by [39] and all other implementations we are aware of, has complexity $O(ck^2 + dk^3)$ for one iteration where c is the total number of terms, d is the number of documents (assumed larger than number of features), and k is the number of dimensions. Our CG implementation by contrast has complexity $O(ck)$. As can be seen in the table, running time does indeed grow linearly with k. Standard ALS running time grows cubically with k, and so large-dimensional calculations often take days to complete. Our CG implementation converges in almost the same number of iterations as the closed-form method, so we obtain full benefit of the $O(k + (d/c)k^2)$ speedup.

System \ Dimension	20	100	1000	Procs
Powergraph (closed)	1000s	**	**	64x8
BIDMach (CG)	150s	500s	4000s	1x8x1

PAM Clustering

As part of a collaborative project, we needed to solve a PAM (Partitioning Around Mediods) clustering task which was taking many hours. The PAM code was in Matlab® and called C mex files for compute-intensive tasks. Standard PAM has high complexity: with n samples and f features, clustering into k clusters naively has complexity $O(n^2 f + n^2 k)$ per iteration. The $O(n^2 f)$ term comes from PAMs initial step of computing all pairwise distances between samples. Fortunately, for the case we were interested in: dense data and euclidean distance, the pairwise distance can be implemented on GPU with a dense matrix-matrix multiply. Our multi-GPU SGEMM achieves 3.5 teraflops on the 4-GPU data engine, and this improved the absolute time of distance computation by almost four orders of magnitude over the C mex code. The second step involves iteration over all current mediods and all other points to find a swap that best improves the current clustering. i.e. this step is repeated kn times. In the standard PAM algorithm, computing the new cost after the swap normally takes $O(n)$ time. We realized that this step could be greatly accelerated by sorting the pairwise distance matrix. When this is done, the average cost of the swap update dropped to $O(n/k)$, and the overall cost for our new PAM implementation is:

$$O(n^2 f + n^2 \log n)$$

i.e. it was dominated by the initial distance calculation and sort. A further benefit of this approach is that it is no longer necessary to store the entire $n \times n$ pairwise distance matrix, since finding nearest mediods on average only requires looking $O(n/k)$ elements into each sorted column of distances. So the storage required by PAM drops to $O(n^2/k)$.

The constant factors were still important however, and CPU-based sort routines slowed down the calculation substantially. We therefore added a GPU sort based on the thrust library radixsort. Out routine sorts the columns of an input matrix and produces a set of integer indices so that the same permutation can be applied to other data (here we needed

the indices of other points corresponding to each distance). The GPU sort is able to sort 200 million elements/second with indices. The time breakdown for clustering 20000 points with 20000 features into 200 clusters is: 10 secs distances, 30 secs sorting and 9 secs searching. This is more than a 1000-fold improvement over the previous implementation.

Regression and SVM

We briefly summarize the results from the paper [96] here. We tested the scaling improvements with butterfly mixing with two algorithms and two datasets. The algorithms were the Pegasos SVM algorithm [78], and a logistic regression model. Both used SGD and the convergence of both algorithms improved with update frequency, making parallelization difficult. The two datasets where the RCV1 news dataset and a custom dataset of Twitter data. The twitter dataset was a mix of tweets containing positive emoticons, and a sample of tweets without. The emoticons serve as cues to positive sentiment and so this dataset can be used to build sentiment models. This sensitivity of convergence to batch size is shown below for RCV1.

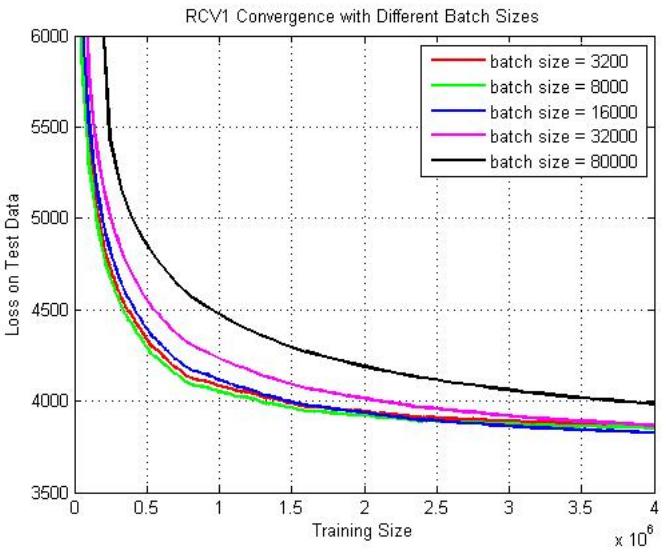


Figure 2.4: Impact of Minibatch Size on Convergence

The optimal sequential mini-batch size was 16000 (1M tokens), and a batch of this size takes about 10msec to process and 100msec or 10msec respectively to load from a single disk, or the Data Engine RAID. Synchronizing the model over a 64-node EC2 cluster using MPI allreduce took over half a second, so communication would completely swamp computation if done at this rate. A single butterfly step by contrast takes about 25 msec. With butterfly mixing updates we can increase the mini-batch size somewhat to better balance communication and computation without a large increase in convergence time. The overall speedups are shown in figure 2.5.

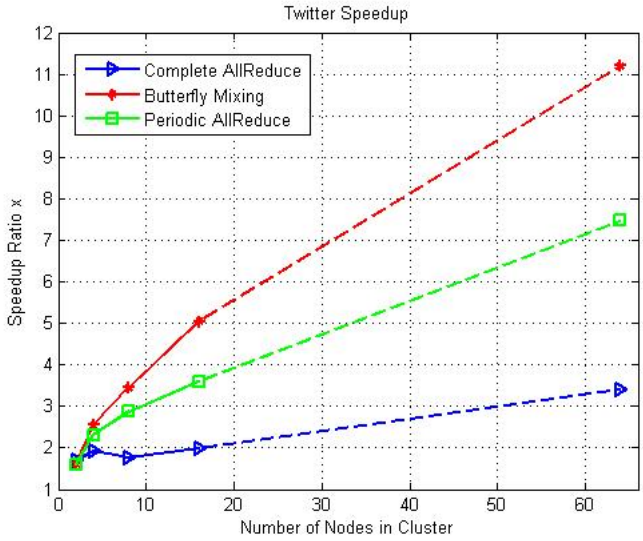


Figure 2.5: Speedup Relative to Single Processor

In figure 2.5, speedup ratios are given for 4, 8, and 16 nodes in a local cluster, and 64 nodes in EC2 (linked by dotted lines to indicate the difference in platform). While the speedup is less than the number of nodes, this is an encouraging result given the strong dependence of SGD on update frequency.

2.4 Squaring the Cloud

Parallelizing machine learning is one approach to improve speed and extend problem scale. But it does so often at high cost. Fast-mixing algorithms (SGD and MCMC) in particular suffer from communication overhead. The speedup is typically a sublinear function $f(n)$ of n, since network capacity decreases at larger scales (typical approximations are $f(n) = n^{\alpha}$ for some $\alpha < 1$). This means that the cost of the computation in the cloud *increases* by a factor of $n/f(n)$ since the total work has increased by that factor. Energy use similarly increases by the same factor. By contrast, a single-node speedup by a factor of k implies a simple k-fold saving in both cost and power.

Interestingly, node speedups in tasks already running on clusters can generate superlinear savings. Imagine a task whose cluster speedup function is $f(n) = n^{0.5}$ running on 1000 nodes, and completing in time t. If we can achieve a 10-fold speedup in single node performance, the same task running on 10 accelerated nodes will achieve the same overall running time. The 10-fold single-node speedup has generated *a 100-fold decrease* in total work and power. More generally, the savings for a k-fold single node speedup will be $f^{-1}(k)$ or k^2 for our example. This super-linear behavior motivated the subtitle of the paper: "squaring the cloud"

2.5 Related Work

Many other big data toolkits are under active development [39], [62], [94], [33], [87], [42], [73], [53] [7]. Our system is perhaps closest to GraphLab/PowerGraph [62, 39] which has a generic model layer (in their case graphs, in ours matrices) and a high-level collection of machine-learning libraries, and in the use of custom communication vs. reliance on a MapReduce or Dataflow layer. Our system also resembles Jellyfish [73] in its deep integration of fast iterative solvers (SGD and eventually MCMC). There are many unique aspects of our system however, as articulated in the introduction. It's the combination of these factors that lead to performance/cost and performance/energy advantages of one to two orders of magnitude over other systems at this time.

2.6 Summary

We made a case for the importance of single-node enhancements for cost and power efficiency in large-scale behavioral data analysis. We showed that enhancements based on CPU and GPU acceleration, and in some cases algorithmic improvements, provide similar or better speedups to medium scale cluster implementations. However, they do so at far lower cost and power. The paper focused on standard machine learning algorithms, whereas the toolkit in future will include a large suite of algorithms for causal analysis. We did not describe any MCMC algorithms, but these also are an important part of the BID Data roadmap. Finally, we did not discuss disk-scale, GPU-assisted sorting which enables a host of other queries such as joins, group bys, and indexing. We hope to achieve similar enhancements for those tasks soon.

Chapter 3

Roofline Design

3.1 Introduction

"Big data" is a broad term that covers many types of data and analysis. Increasingly it includes machine learning algorithms running at very large scale. Our goal has been to develop tools that support such analysis as efficiently as possible. That means focusing on single-machine speed first, and cluster scale-up second. Efficiency, measured as operations/watt or operations/unit cost, cannot be improved by scaling up on a cluster: and in fact since performance scaling is inevitably sub-linear with cluster size, efficiency goes down. As has been shown previously [23], orders of magnitude performance gains are possible on single nodes. It is still possible to scale up accelerated single node algorithms, but this requires new network protocols which have been developed recently [96] and [97]. Full performance is achieved using GPUs, although BIDMach with CPU acceleration is also faster than the other systems we have benchmarked against. We argue that GPUs are now mature and fully-functional data analysis engines, and offer large gains on most ML algorithms. A final and very important goal is usability and interactivity. Practical data mining systems require a considerable amount of tuning and incorporation of business logic constraints. The more iteration and exploration during design time, the better the solution and this often translates directly into higher revenue. Ease of use, interactivity, and high single-machine performance all contribute significantly to better exploratory analysis and tuning.

Related Work

The ML tools in common use fall in two classes: Many groups [39, 81, 63, 58, 4, 50] are developing tools to analyze these datasets on clusters. While cluster approaches have produced useful speedups and scaled to thousand of machines, their single-machine performance is often poor. Recent work has shown that for many common machine learning problems single node benchmarks now dominate the cluster benchmarks that have appeared in the literature [22, 23]. Furthermore, the usability of these systems has not been a priority. At the very least, the level of interactivity is low and they are not generally integrated with

an interpreter/visualization system. At the other end of the tools spectrum: data analysis tools such as Matlab, R, Scikit-Learn provide an interactive environment for analysis and visualization, and are generally easy to learn and use. While these system often have some CPU acceleration (typically through BLAS/LAPACK libraries) they lack thorough integration with GPUs. Other recent work has considered scale-up of those systems on clusters [25, 45]. But relatively little work has considered single-node optimization, whereas we believe that is where some of the biggest opportunities lie. We also believe that there is no conflict between high usability and high performance, and that it is possible to deliver both.

Contributions

In this paper, we describe BIDMach, which incorporates two key design ideas: codesign, which means choosing the right combination of hardware and designing algorithms to best leverage the hardware, and roofline design [89], which means quantifying the theoretical performance limits of each component of a ML algorithm and making sure to get close to them. When implementing a algorithm, we need its hardware mappings. This includes computation pattern required by the algorithm, memory/IO pattern and communication pattern when designing a distributed algorithm.

To reach performance limits, careful and often iterative design and coding is needed. This is time-consuming. It would be problematic to do such optimization for every machine learning algorithm. Instead, we create an intermediate layer - a set of common computation and communication kernels/primitives, between BIDMach and the hardware. This includes BIDMat, butterfly mixing and Kylix. These deal respectively with matrix algebra, model synchronization across the network, and algorithms on graphs. An important role of BIDMat is to manage memory with matrix caching, which we will describe shortly. The software stack is shown in Figure 3.1.

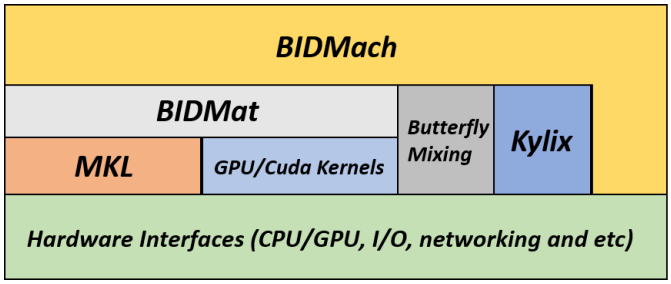


Figure 3.1: Software Architecture

BIDMach is written in the beautiful Scala language. Scala can run compiled or interactively (it includes a Read-Eval-Print Loop) and it can interpret uncompiled Scala scripts. Scala is a "functional-style" language with lambda expressions, and supports mutable and immutable objects. Scala runs on the JVM (Java Virtual Machine) and is fully interoperable with Java. For GPU support, we use the JCUDA library to expose NVIDIA CUDA kernels

though the Java Native Interface, and into Scala. BIDMach can run on Java-based distributed platforms such as Hadoop and Spark. Our code is open-source and freely available on github. We also have a public image for launching GPU instances on Amazon EC2.

BIDMach is similarly designed to best support minibatch algorithms (although it can handle any learning algorithm). A diagram of its core classes appears in Figure 3.2.

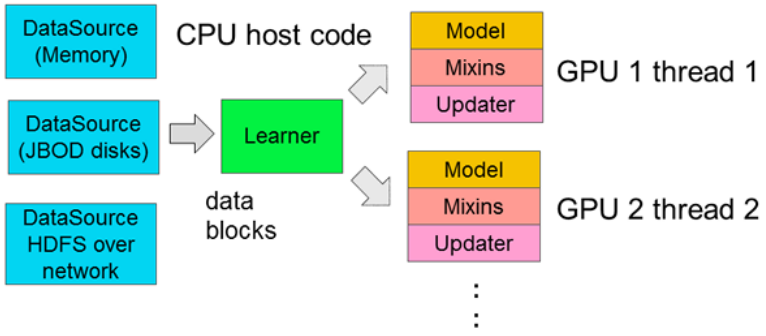


Figure 3.2: The BIDMach Framework

To develop a new learning algorithm, a developer needs to define a new **M**odel class which specifies an update step on a (typically mini-) batch of data. For simple gradient-based algorithms, this is all that needs to be done. The other classes manage data delivery, pre-processing, optimization and regularization (via mixin classes). Code in the model class should written using generic matrix operations, and the developer never need be concerned about where it is run (on CPU, GPU, multiple GPUs or across a cluster).

The rest of the paper is organized as follows. Section 2 describes a set of computational kernels for ML. Section 3 explains the roofline design principle and the system architecture of BIDMach. We show a number of new benchmarks in Section 4 and several real world applications in Section 5. And finally Section 6 concludes the paper.

3.2 Computational Kernels for Machine Learning

Matrix presentations (including sparse representations, adjacency matrices for graphs etc) are used in many machine learning toolkits, and provide convenient abstractions for aggregates of elements. They are also a quick path to high performance, since optimization need only deal with the matrix operators used by ML algorithms. The next step then, is to enumerate the matrix kernels needed. We can assume that common matrix algebra kernels are included, but it will be important to consider performance on typical "big data". For instance, we have found that sparse matrix operators developed for scientific computing do not perform well on power-law data (which is found in text, social networks, web graphs etc.), so we should enumerate these anyway.

Multi-Class GLM with SGD

Generalized Linear Models (GLM)s are the most widely used class of models for classification tasks. It includes linear regression, logistic regression and are closely related to SVM, which is implemented in the GLM package in BIDMach. For many practical problems (e.g. Advertisement targeting, User profiling, Message classification) the goal is to classify an instance using multiple classes, either independently or to a single best-match label. Our implementation performs multi-class classification efficiently using matrix operations.

In GLM, the object is to minimize the loss function (or equivalently maximize the likelihood function) in the following form,

$$l = f(\beta^T X), \tag{3.1}$$

where X is feature-by-example matrix, β a vector that parameterizes the model, and f is the GLM mean function for that model type. The model gets updated according to,

$$\beta_{t+1} = \beta_t + \gamma_t \frac{dl}{d\beta_t}, \tag{3.2}$$

where γ_t is the learning rate at time t (some variants use less than unit weight for the previous β). The gradient of the likelihood is defined as,

$$\frac{dl}{d\beta_t^T} = f'(\beta_t^T X_t) X_t^T, \tag{3.3}$$

where X_t is the mini-batch of data at time t.

In a multi-class regression model, the matrix formulation is to pool β vectors for all n models into a matrix $B = (\beta^1, \beta^2, \cdots, \beta^n)$, so that all the models are trained in parallel with each pass over the dataset.

Inspection of the formulas above shows that to implement multi-class regression we need only matrix-matrix multiplication, addition, and the function $f'(...)$ above. In all cases, we assume the model β is dense (for large datasets, it has few zeros) but the input matrix X may be dense or sparse. The dense operations are already part of BLAS libraries. From (3.3), we can most conveniently store β in transposed form, and then we need matrix multiply operations $D \times S$ and $D \times S^T$ on one dense and one sparse matrix argument respectively. And if we implement $D \times S^T$ directly as a single primitive, we avoid the very expensive sparse transpose step.

Latent Dirichlet Allocation

We start with a matrix formulation of LDA, which is easily obtained from the element-wise representation from [11]. We give here only the update for the topic-document matrix U, the update for the topic-term matrix V is similar. Let S be a block of input data (term x

document matrix). Then the U update can be computed as:

$$U \leftarrow \exp\left(\Psi\left(U \circ \left(V\frac{S}{VU}\right) + \alpha\right)\right) \tag{3.4}$$

where the products are matrix products, the division is element-wise, and $\circ$ denotes Hadamard (element-wise) product, and α is the topic-document prior. The bottleneck step in this calculation is the term $V\frac{S}{VU}$. Now VU is an nterms x ndocuments dense matrix, but when we compute the element-wise ratio $R = \frac{S}{VU}$, we don't need any terms of VU where S is zero (the matrix formula is a shorthand for the original element-wise formula, results of VU where S is zero are never needed and can be zero). For typical data, the density of S may be much less than 1%, and so we need only compute those values. We developed a kernel (SDDMM) for this step, which turns out to be a key step in other factor models such as Gibbs sampling LDA and Sparse Factor Analysis [23]. The result R is a sparse matrix with similar structure to S, and we then need to compute the product VR, which is a dense-sparse product as we saw with GLM. We need element-wise division, multiply and element-wise function application for exp and Ψ. Finally when the V model is updated we need a normalizing vector which requires a column-wise accumulate $sum(...)$ operation.

Data Rearrangement: Sorting

A variety of kernels, in particular random Forests, require "long-range" data movement, i.e. rearrangement which wont fit in fast processor memory. We use sorting for these tasks which involve a variety of key types and values. Details will be explained shortly.

3.3 System Architecture

Roofline Design

From the discussion above, we have the following kernels. Together they comprise more than 90% of the computing time across our algorithms.

1. Matrix-matrix multiply, especially $D \times S$ and $D \times S^T$ on sparse, power-law S.
2. SDDMM, the product of two dense matrices evaluated at the non-zeros of a sparse matrix.
3. Element-wise matrix operations: add, subtract, multiply, divide, power.
4. Element-size functions exp, log, Ψ etc.
5. Sorting.
6. Row-wise and column-wise reduction operations, $sum()$, $mean()$, $sdev()$ etc.

For CPU acceleration, we rely on Intel MKL (Math Kernel Library) and Scala kernels. MKL already includes sparse BLAS which implement (1). We explored C native code implementations of (1) but did not achieve a performance gain. Both (1) and (2) are memory-intensive operations. The most efficient implementation we found used level-1 BLAS sub-steps. i.e. Matrix-matrix multiply of $D \times S$ was accomplished by multiplying a column of D by an element of S. The SDDMM operation $SDDMM(A, B, S)$ was implemented by computing the inner product of the i^{th} column of A and the j^{th} column of B corresponding to a sparse non-zero element S_{ij}. For very sparse S, there is little re-use of either row or column data. Large model matrices are too big to cache, so the maximum throughput will be limited by main memory bandwidth. For each element of a column for either algorithm, we perform a multiply and add, and we read and write a total of 8 bytes. From here we can easily derive a bound on the maximum throughput from roofline arguments [89], $GF = 2/8 * BW$. On our test machine, a mid-range 2012 desktop workstation, the CPU main memory throughput was 28 GB/s. Substituting the numbers from above, we would expect a limit of 7 gflops on either dense-sparse MM or SDDMM. This value is much lower than the ALU throughput (several hundred gflops), and there is no power limit that applies. So our ideal performance is 7 gflops. The measured performance was in fact slightly higher at 6 gflops for MM, and 5 gflops for DDS. Its unlikely that these kernels could be made much faster.

Similar arguments apply to the design of these kernels on a GPU. The GPUs on our test machine are NVIDIA GTX-680s. They have a main memory bandwidth of around 150 GB/s. Based on memory B/W limits, the maximum kernel throughput for MM and SDDMM on sparse data should be $2/8 * 150 = 37$ gflops. We tried the Nvidia SPBLAS sparse BLAS implementation of dense-sparse multiply, but measured a throughput of only about 10 gflops. We implemented our own dense-sparse MM, and SDDMM in CUDA C code. We summarize the results below, along with the CPU kernel results. Note the asymmetry between normal and transposed multiply on the GPU.

Operation	$D * S$	$D * S^T$	$SDDMM$
CPU	6 (7)	6 (7)	5 (7)
GPU	23 (37)	30 (37)	32 (37)

Table 3.1: Memory roofline for matrix multiply operators. All numbers show in gflops, estimated roofline values in parentheses.

For basic element-wise matrix operators, the analysis is simpler. Using a 28 GB/s memory B/W bound for our CPU, an element-wise operation must read two elements (4 bytes each for single precision) and write one for each flop it achieves. This suggests a roofline of 28/12 or 2.3 gflop. The measured throughput is 1.7 gflops. On the 680 GPU with 150 GB/s B/W, we derive a limit for element-wise operations of $150/12 = 13$ gflops. The measured throughput for all the operators we tried was 11 gflops.

Measurement of element-wise function evaluation is more complicated because the arithmetic complexity of the function may not be known (in fact usually is not known since it depends on an internal implementation). Let's fix two functions that are commonly used, exp (a low complexity function) and the log of the gamma function (a high complexity function). We cant easily give roofline CPU estimates for these, so our limits come from memory considerations only. We also measure throughput with function evaluations / second instead of flops.

Operation	$op(A, B)$	$\exp(A)$	$gammaln(A)$
CPU	1.7 (2.3)	2.3 (3.4)	0.15 (3.4)
GPU	11 (13)	12 (18)	7 (18)

Table 3.2: Memory roofline for element-wise operations and function evaluation. Numbers shown in billions of operations or billions of function evaluations/sec, estimated roofline values in parentheses.

For reducing operations, performance may be quite different depending on whether reduction runs along columns (i.e. for a column-major system like BIDMach, that means along contiguous memory locations), or along rows. We summarize the results for the $sum()$ operator below

Operation	sum along cols	sum along rows
CPU	0.7 (7)	0.05 (7)
GPU	28 (37)	28 (37)

Table 3.3: Memory roofline for the sum reduction. Numbers shown in gflops, or billions of element reductions/sec. Estimated roofline values in parentheses.

The slow performance of CPU reducers is a "known issue" in the toolkit at this point, and requires writing vector operations for each reduction.

Sorting is very easy to program in CUDA which includes templated C++ sorting libraries. It has proved much harder to do in available CPU libraries which are available only for a few key types. The table below shows sorting throughput in GB/s. To define an appropriate memory roofline, we need to figure out how many times data needs to be moved. Merge sorting requires $\log_2 N$ rounds, or 28 rounds for 1 GB of 32-bit data. Quicksort requires approximately 1/4 as many memory moves. Radix sort is also often fast on large datasets because it requires fewer rounds. For 4-bit radices, it uses 1/4 as many rounds with full copies, which gives the same roofline as Quicksort. The table below shows best observed performance on CPU (which uses a hybrid sort) and GPU (using Thrust radix sort) on 1GB of 32-bit keys.

System	Best throughput	Quicksort/Radixsort roofline
CPU	0.032 GB/s	2 GB/s
GPU	4 GB/s	10 GB/s

Table 3.4: Memory roofline for sorting

Discussion

The above tables show that the GPU kernels consistently approach their memory roofline limits. The CPU kernels do in many cases, but fall behind in a significant number of others. In some cases, e.g. sum reductions, this gap should be easy to close. In others, i.e. sorting, it is much harder.

Matrix Caching

BIDMat is a Scala matrix library intended to support large-scale machine learning. In particular it has rooflined kernels for all the operators discussed in the last section, and is well-adapted to sparse, power-law data. In BIDMat, `Mat` is a generic matrix type that can be backed by both CPU and GPU, dense or sparse, single or double precision, and integer or float data. This allows users to focus on algorithm logic without worrying about the implementation. BIDMat provides an interactive environment reminiscent of Matlab, R and NumPy but with a more natural syntax (standard math operators are available directly), full native support for multi-threading, a clean syntax, and considerably improved performance on compiled code.

GPUs offers high throughput on the key kernels for machine learning as shown in the last section. However, there is no garbage collector in the standard GPU libraries. And more importantly, memory allocation on typical matrix sizes on a GPU is much slower (sometimes two to three orders of magnitude, depending on the size of block to be allocated) than GPU memory access. So even if a GPU garbage collector were available it would completely swamp the calculations that use it. Instead, we have implemented a caching scheme which supports re-use of intermediate results, and which achieves no memory allocation after the first use of a container. This approach complements mini-batch processing as we describe later.

To implement caching, each matrix has a unique GUID which is set when the matrix is created. A basic operation like $C = A + B$, rather than allocating new storage for C, first checks the matrix cache using a key `(A.guid, B.guid, "+".hash)`. If A and B are immutable (`val` in Scala), the output is fully determined by their values and the operator $+$. It is therefore safe to use a container for C within any code segment where A and B are immutable. A and B are actually mutable which is important for us to be able to re-use their storage. But once set, we can derive arbitrary expression DAGs from them with recycled storage.

Some operations return matrices whose dimensions depend on other matrices. For instance, an expression $A(I;J)$ with matrix A and integer matrices I and J returns a matrix whose size is the product of the number of elements in I times the number of elements in J. Since these matrices have immutable dimensions, the following hash key will find a correctly-sized container if one has been saved: `(A.guid, I.guid, J.guid, hash("apply"))`.

Scalars (floating point or integer) can be mixed with matrices in expressions and have the usual mathematical interpretation. e.g. $A + 1$, $B * 2$ etc. Since scalars are values rather than references (as for matrices) there are no GUIDs associated with scalars, and caching needs to be done carefully. To avoid aliasing the scalar values must be incorporated into keys. e.g. $A + 1$ is cached with key `(A.guid, 1, hash("+"))`. But these can lead to a failure of caching in common functional expressions like $a * A$ where a is a scalar which is changing from one iteration to the next. Each evaluation leads to a separate cached value even though the symbol a is the same, and its value is assumed fixed in that scope. The simplest solution is to use 1x1 matrices for scalars. Setting as contents and keeping it fixed inside model methods will generate only one cached value.

Mini-Batch Processing

BIDMach is optimized for minibatch processing. While not every algorithm fits this framework, a very large number of algorithms have been shown to achieve optimal performance in a mini-batch implementation. Recent results suggest in fact that SGD is "data-optimal" in terms of early, approximate convergence [15]. And many other algorithms (LDA, NMF) have been instantiated as "online" minibatch algorithms whose performance is as good or better than batch versions. And finally, mini-batch design is natural for MCMC algorithms which involve continuous model updates in the limit.

The BIDMach Framework

BIDMach has a modular design intended to make it very easy to create new models, to run diverse datasources, and tailor the performance measures that are optimized in training. The architecture is shown in Figure 3.2. Here we describe in detail each of the core classes of BIDMach:

DataSource

DataSource support a "next" method which produces a minibatch of data i.e. a block of samples of specified size. The datasource itself may be backed by an array in memory, a collection of files on disk, or an HDFS source. Datasources in general output multiple matrices in response to the next method: for instance a datasource for training regression models outputs a block of k samples as a sparse matrix and a block of k class membership vectors as a dense matrix. Some datasources also support a "putBack" method, which allows data to be pushed back into the Datasource. Such sources are therefore both sources and

sinks. For instance, a datasources for regression prediction has two "output" matrices: one contains includes the data instances to predict from, and the second matrix contains the predictions.

Model

Model implements particular learning algorithms in a minibatch framework. Models support an update method, which often produce a model gradient from the current model and input data block, and a likelihood method which returns the likelihood of a data block given the current model. Models are represented with one or more matrices, which are stored in `model.modelmats`. Not all updates are gradients, and e.g. multiplicative updates involve two matrices whose ratio defines the final model.

Mixins

Mixins are additional terms in the loss function that the learner minimizes. Mixins include L1 and L2 regularizers, but also richer terms like within-topic entropy, cross-topic divergence, distributional skew (borrowed from ICA and used to improve topic independence). Any number of mixins may be used with a particular learner. Mixins also include an individual loss term that is evaluated and save during learning. They are natural "KPIs" (Key Performance Indicators), and are very useful for model tuning.

Updater

Updater implements particular optimization strategies. They include simple batch updates and many kinds of minibatch update. Minibatch updaters use dynamic windows and decreasing weights to integrate gradient updates with an evolving model.

Butterfly Mixing

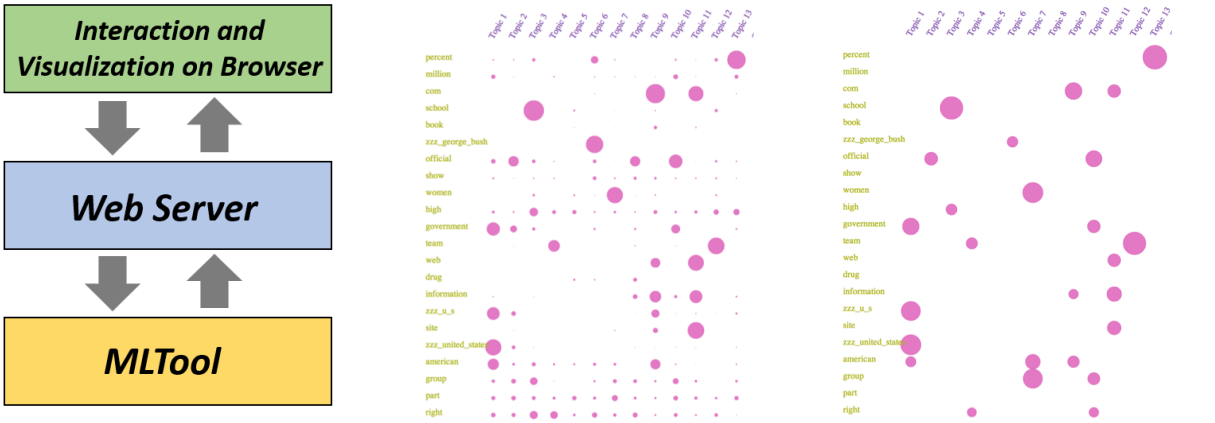
Buttefly mixing is a technique for rapid sharing of minibatch model updates in a cluster. An Allreduce computes the reduction (Aggregate) of models across the cluster (sum, avg, max etc.), and then redistributes to all hosts. Allreduce operations are used in most Map-Reduce implementations of machine learning algorithms, but comprise a single allreduce at the end of each pass over the dataset. These primitives are too slow for minibatch applications, where the goal it to perform many allreduce operations per minute, or even several per second. A simple solution to this problem is "Butterfly Mixing". Butterfly networks (or equivalently, rounds of communication long specific dimensions of a hypercube), are latency-optimal allreduce networks. Butterfly Mixing breaks the AllReduce operation into layers of such a butterfly network and interleaves model updates with each layer.

Its clear that this approach dramatically reduces communication overhead compared to allreduce every step, and many more gradient updates are being done in a given amount of time. Its less clear what impact this has on convergence. In [96], Zhao et al has showed that

this impact is minimal and in fact the convergence (in number of update steps) of butterfly mixing on typical datasets is almost as fast as an allreduce at every step.

Kylix

The Kylix primitive supports distributed graph algorithms on "hard-to-separate" graphs such as power-law graphs, and distributed algorithms with massive models which are too big to fit on each machine. The reduce operations in several popular frameworks [63, 81, 39] turn out not to be scalable, and performance degrades on large clusters. These systems use randomized all-to-all communication. But as the network grows, packet sizes shrink at some point hit a packet-size "floor" below which communication throughput slows down considerably. This packet size floor is about 5MB on EC2. To avoid this problem we use an elaboration of butterfly allreduce. But this time the solution is a nested, heterogeneous-degree butterfly network. By heterogeneous we mean that the butterfly degree d differs from one layer of the network to another. This allows us to tune packet sizes for each layer. By nested, we mean that values pass "down" through the network to implement an scatter-reduce, and then back up through the same nodes to implement an allgather. Nesting allows the return routes to be collapsed down the network, so that communication is greatly reduced in the lower layers. Because reduction happens in stages, the total data and communication volume in each layer almost always decreases (caused by collapsing of sparse elements with the same indices). This network works particularly well with power-law data which have high collision rates among the high-frequency head terms.



(a) Interaction/Visualization Architecture

(b) Original LDA

(c) LDA with L1 Regularization

Figure 3.3: LDA Interaction and Visualization

Interactive Learning and Visualizations

Model training is fast with BIDMach, and its possible to interact with the model using appropriate visualization/interaction widgets while it is being trained.

The visualization architecture is described in Figure 3.3a. BIDMach provides a web interface for live tuning parameters during model training. The graphical interface is built with `D3.js` -a JavaScript library for manipulating documents based on data. The web server hosts the data to be visualized, and syncs the model data with BIDMach 10 times per second to show the live stream of the model visualization. The web server also passes user requests (from the browser) for parameter changing to BIDMach.

Figure 3.3b and 3.3c show two visualization examples of LDA. In the figures, each row represents a word and each column represents a topic. The size the bubble in the figure is proportional to the weight of the corresponding word-topic weight in the LDA model. The two figures visualize two LDA models with and without regularization. We can clearly see that noise weights are removed when applying L1 regularization. In the real system, such visual feedbacks are live while training the model, and user can adjust parameters according to such feedback. We hope this can help users to identify the right set of parameters.

Recent work in [6] presents an inference method using "cooled" Gibbs sampling. The approach allows tuning temperatures of individual parameters in a model. The temperature controls the variance of the sampling distribution as in simulated annealing. In LDA, user can click on a bubble (word-topic weight) and adjust the temperature for that parameter. Increasing the temperature heats up the distribution, and this acts as re-evaluating the parameter because the next sample will have high variance. The interaction enables human-guided search and may help remove potential defects in the model.

3.4 Benchmarks

We benchmark the performance of BIDMach on several common ML problems and compare with many other systems. All the experiments are performed on a PC with an Intel E5-2660 CPU, a GTX-680 GPU, 64GB RAM and Redhat Enterprise Linux 6, or otherwise explained.

Logistic Regression

The first benchmark studies performance on multi-class classification tasks using logistic regression - arguably the most common data analysis tasks that has many applications. We first look at sparse data, which is the common data structure for text, web, click, social media links etc. The result can be seen in Table 3.5. We use the RCV1[61] dataset for the test. All the systems here that have support for MKL are compiled with MKL support linked in.

In the table, "until conv." means until convergence. For this problem, BIDMach with GPU (minibatch size = 10000) is about two orders of magnitude faster than Vowpal Wabbit

System	Num Classes	Time	Num Passes	Gflops
BIDMach GPU	103	0.7s	1	11
BIDMach MKL only	103	19s	1	0.4
Vowpal Wabbit	103	44s	1	0.15
Scikit-Learn	103	576s	until conv.	N/A
Mahout	20	1560s	until conv.	N/A

Table 3.5: Logistic Regression (sparse) on 276k x 100k block of RCV1 date set, single machine

[58], the next-fastest system. All algorithms converge to the same accuracy. BIDMach is able to converge in 1-2 passes.

We Further test the performance on dense input data. The result can be seen in Table 3.6.

System	Num Classes	Time	Num Passes	Gflops
BIDMach GPU	103	0.3s	1	100
BIDMach MKL only	103	2.4s	1	16
Vowpal Wabbit	103	30s	1	N/A
Scikit-Learn	103	84s	until conv.	N/A
Mahout	1	55s	1	0.006
Spark/ MLLib	1	20s	1	0.015

Table 3.6: Logistic Regression (dense) on 1000 x 100000 block of RCV1 date set, single machine

Finally we perform a experiment on 100GB dataset - a benchmark of a similar dataset is reported in [95] on a 100-node cluster. The authors train a single logistic model in 10 iterations. We assume this gives good convergence. The results can be seen in Table 3.7. Again BIDMach is able to converge in 1 pass for this dataset using mini-batch updates. BIDMach on one GPU-accelerated node is slightly faster than Spark on the 100-node cluster, while training 103x as many models . BIDMach is faster when only training a single model, this time about 4x faster than Spark on the cluster.

K-Means

We test k-means on the RCV1 dense data set. BIDMach achieve 280 Gflops when using GPU acceleration, which is 46 times higher than MLLib. The results can be seen in Table

System	Machines	Num Classes	Time	Num Passes	Gflops
BIDMach GPU	1	103	60s	1	100
BIDMach MKL only	1	103	400s	1	16
BIDMach GPU	1	1	20s	10	3
Spark/ MLLib	100	1	80s	10	2

Table 3.7: Logistic Regression (dense) on 100 GB dataset

3.8.

System	Time	Gflops
BIDMach GPU	7s	280
BIDMach (MKL only)	47s	50
Scikit-Learn	130s	20
Spark/MLLib	315s	6

Table 3.8: K-Means clustering (dense) on 1k x 100k block of RCV1, 256 dimensions, 20 passes, single machine

We also test a larger dense data set based on a benchmark in [5], to compare our system with a 40-node cluster. The results can be seen in Table 3.9. For such a larger problem, BIDMach is still 20x faster than Spark on the cluster.

System	Machines	Time	Gflops
BIDMach GPU	1	61s	9
BIDMach MKL only	1	520s	1
Spark/ MLLib	40	1200s	0.4
Mahout	40	2200s	0.2

Table 3.9: K-Means distributed clustering (dense) on 100M x 13-element random vectors,10 passes

Latent Dirichlet Allocation

BIDMach includes several implementations of LDA: an online Variational Bayes (VB) [44], a batch variational Bayes [11], and a cooled Gibbs sampler [6] implementation. We compare

against 3 systems, which are all custom LDA implementations: 1) Blei et al's original C++ implementation of batch VB [11]. 2) Griffiths et al's implementation of CGS in Matlab/C [40]. 3) Yan et al.'s implementation of CGS with GPU acceleration [91] (on a Geforce 280 GTX). To the best of our knowledge, [91] is the fastest single-machine implementation of LDA other than BIDMach.

We conduct the benchmark tests on the widely used UCI NYTimes dataset. The results can be seen in Table 3.10. All systems are trained to convergence. The "Iteration" column shows the number of iterations taken to converge. The "Time" column shows the total time. BIDMach is at least two orders of magnitude faster than the other systems.

System	Time	Iteration	Gflops
BIDMach online VB	40s	2	25
BIDMach batch VB	400s	20	25
BIDMach Cooled GS	90s	3	30
Yan GPU CGS	5400s	1000	0.4
Blei batch VB	252000s	20	0.05
Griffiths CGS	225000s	1000	0.008

Table 3.10: Latent Dirichlet Allocation (sparse) on NYTimes dataset

BIDMach is also able to run LDA with 1000 topics on 1 terabyte of twitter data, in 3 hours until convergence. We are aware of no other system can run LDA on this scale. Later in Section 3.5, we compared our LDA with a cluster implementation.

PageRank

This benchmark evaluates the scalability of BIDMach/Kylix on clusters. We run PageRank on two well known graphs - Yahoo's web graph [74] and Twitter Follower's graph [56], on a 64-node Amazon EC2 clusters. We compare BIDMach/Kylix with Hadoop/Pegasus and Powergraph. We chose Hadoop because it is a widely-known system, and Powergraph because it has demonstrated the highest performance to date for the Pagerank problem [39]. Figure 3.11 plots runtime per iteration for different systems. PowerGraph is run on a 64-node EC2 cluster with 10Gb/s interconnect - the same as us. Each Powergraph node has a two quad core Intel Xeon X5570 for the Twitter benchmark and dual 8-core CPUs for the Yahoo benchmark [39]. Pegasus runs on a 90-node Yahoo M45 cluster. We estimate Pegasus runtime for Twitter and Yahoo graph by using their runtime result [50] on a power-law graph with 0.3 billion edges and assuming linear scaling in number of edges. We believe that the estimate is sufficient since we are only interested in the runtime in terms of order of magnitude for Hadoop-based system. From the table it can be seen that Kylix runs 3-7x faster than PowerGraph, and about 400x times faster than Hadoop.

System	Yahoo graph	Twitter graph
BIDMach/Kylix	2.5s	0.55s
PowerGraph	7s	3.6s
Pegasus/Hadoop	1000s (est)	250s (est)

Table 3.11: PageRank runtime per iteration

Economic Analysis

Finally, we compare the cost of running regressions (as in Table 3.7 for training 103 models), k-means (as in Table 3.9) and LDA for processing 20 million news articles with 1000 topics, using different systems. The LDA benchmark of Ahmed et al. [4] (CGS) is reported on 100 machines, and we assume it converges in 1000 iterations. We use the hourly rate on Amazon EC2 for this cost estimation. A GPU instance (`g2.2xlarge`) is priced at $0.65/hr, which is used by BIDMach. We assume all other systems use a general purpose large instance (`m3.large`) which is priced at $0.14/hr - the second cheapest among all instances offered. The results are shown in Table 3.12. BIDMach has a huge cost advantage over other systems.

System	Regression	K-Means	LDA
BIDMach GPU	0.01	0.01	5.4
BIDMach MKL only	0.07	0.09	N/A
Spark/ MLLib	31	1.9	N/A
Mahout	N/A	3.4	N/A
Ahmed et al. [4]	N/A	N/A	466

Table 3.12: Costs of running different algorithms (US $)

3.5 Applications

Here we briefly describe the performance gain of deploying BIDMach at a US major technology company, for two online advertisement applications.

Offer/Query Categorization

In offer/query categorization, the goal is to train a classification model for predicting product categories for incoming offers/retail queries. The training dataset has a few million examples and 500k features, and there are 2000 categories in the product taxonomy. We trained 2000

independent models using logistic regression with BIDMach. On a single GPU, training can be completed in 2 min for one set of tuning parameters. This is 500x faster than the original in-house implementation of logistic regression with batch SGD in C++. We also improve the accuracy by 7%. This is because we are able to explore the hyper-parameter space more rapidly and thoroughly.

Offer Clustering

The second task is to train a clustering model for offer descriptions (average length of 40 words). We applied our LDA with online VB inference. BIDMach is able to process at a speed of 3 million full offer descriptions per minutes (until convergence), for 60 million offers. As a comparison, Ahmed et al. [4] - the state-of-art cluster implementation of CGS LDA, processes 100k news articles per minute on a 100 node cluster. Assuming news article is 10 times the length offer description on average, we are still 3x faster on a single GPU machine than a cluster of size 100.

Auction Simulation and Optimization

Sponsored search auctions allow advertisers to bid in real-time on each search query. They include several parameters ($N_{auction}$) that can be tuned to improve advertiser and publisher-directed metrics. These parameters are tuned using simulation based on recent live auction logs. The logs contain histories of search queries and bids, and the simulator simulates each query auction counterfactually many times using N_{params} different set of parameters. The parameters are further refined by query cluster of which there are $N_{clusters}$. A large number of parameters are explored:

$$N_{auction} * N_{params} * N_{clusters}$$

The full parameter set is approximately 1GB in size. The auction simulation is run over several weeks of data, producing a large array of basic performance indicators. From this array of basic indicators, optimal parameters choices for desired KPIs (Key Performance Indicators) are derived using a simple lagrange multiplier step. This latter step takes a few minutes, and can be done online.

Auction simulation is quite complex so we developed first a Scala reference implementation for debugging. Then we wrote two GPU implementations. The first relied on shared memory for working storage, and achieved an 70x speedup over the Scala reference implementation. The second used more aggressive coding, moving almost all the state of each auction to register memory, and e.g. sorting bids in registers. This gave a full 400x speedup over the Scala implementation. The GPU implementation is now 50x faster than the previous production simulation running on a cluster of 250 nodes. This example hints at the (unexplored) potential speedups on graphics processors by using register implementations. Whereas GPUs have roughly 10x the computing power and perhaps 5x the memory speed vs. standard PCs, they have at least 1000x the register storage. This storage supports single-

cycle access with no contention, and for suitable algorithms, opens the door to multiple order-of-magnitude speedups.

3.6 Summary

This paper describes the design principles and system architecture of BIDMach. We provide several concrete examples on roofline design and how it guides the development of our system. We present several benchmarks and applications of BIDMach, and compare with other systems. The conclusion is BIDMach is orders of magnitude faster and scalable both on single node and clusters. BIDMach also supports interactive ML.

The roofline design principle is applicable to many other problems, and we are currently developing kernels for inference on general graphical models. It is likely to lead to similar large improvement in speed.

Chapter 4

SAME but Different: Fast and High-Quality Gibbs Parameter Estimation

4.1 Introduction

Many machine learning problems can be formulated as inference on a joint distribution $P(X, Z, \Theta)$ where X represents observed data, Θ a set of parameters, and Z represents latent variables. Both Θ and Z are latent in general, but it is useful to distinguish between them - Θ encodes which of a class of models represents the current situation, while Z represent local labels or missing data. One generally wants to optimize over Θ while marginalizing over Z. And the output of the algorithm is a value or distribution over Θ while the Z are often ignored.

Gibbs sampling is a very general approach to posterior estimation for $P(X, Z, \Theta)$, but it provides samples only rather than MAP estimates. But therein lies a problem: sampling is a sensible approach to marginal estimation, but can be a very inefficient approach to optimization. This is particularly true when the dimension of Θ is large compared to X (which is true e.g. in Latent Dirichlet Allocation and probabilistic recommendation algorithms). Such models have been observed to require many samples (thousands to hundreds of thousands) to provide good parameter estimates. Hybrid approaches such as Monte-Carlo EM have been developed to address this issue - a Monte-Carlo method such as Gibbs sampling is used to estimate the expected values in the E-step while an optimization method is applied to the parameters in the M-step. But this requires a separate optimization strategy (usually gradient-based), a way to compute the dependence on the parameters symbolically, and analysis of the accuracy of the E-step estimates.

SAME (State Augmentation for Marginal Estimation) [75, 29] is a simple approach to MAP parameter estimation that remains within the Gibbs framework[1]. SAME replicates

[1]SAME is a general approach to MCMC MAP estimation, but in this paper we will focus on its realization

the latent state Z with additional states. This has the effect of "cooling" the marginal distribution on Θ, which sharpens its peaks and causes Θ samples to approach local optima. The conditional distribution $P(Z|X,\Theta)$ remains the same, so we are still marginalizing over a full distribution on Z. By making the temperature a controllable parameter, the parameter estimates can be annealed to reach better local optima. In both [75, 29] and the present paper we find that this approach gives better estimates than competing approaches. The novelty of the present paper is showing that SAME estimation can be *very fast*, and competitive with the fastest symbolic meethods. Thus it holds the potential to be the method of choice for many inference problems.

Specifically, we define a new joint distribution

$$P'(X,\Theta,Z^{(1)},\ldots,Z^{(m)}) = \prod_{j=1}^{m} P(X,\Theta,Z^{(j)}) \tag{4.1}$$

which models m copies of the original system with tied parameters Θ and independent latent variable blocks $Z^{(1)},\ldots,Z^{(m)}$. The marginalized conditional $P'(\Theta|X) = P'(X,\Theta)/P(X)$. And

$$P'(X,\Theta) = \int_{Z^{(1)}} \cdots \int_{Z^{(m)}} \prod_{j=1}^{m} P(X,\Theta,Z^{(j)})\; dZ^{(1)}\cdots dZ^{(m)} = \prod_{j=1}^{m} P(X,\Theta) = P^m(X,\Theta) \tag{4.2}$$

where $P(X,\Theta) = \int_Z P(X,\Theta,Z)\; dZ$. So $P'(\Theta|X) = P^m(X,\Theta)/P(X)$ which is up to a constant factor equal to $P^m(\Theta|X)$, a power of the original marginal parameter distribution. Thus it has the same optima, including the global optimum, but its peaks are considerably sharpened. In what follows we will often demote X to a subscript since it fixed, writing $P(\Theta|Z,X)$ as $P_X(\Theta|Z)$ etc.

This new distribution can be written as a Gibbs distribution on Θ, as $P^m(\Theta,X) = \exp(-mg_X(\Theta)) = \exp(-g_X(\Theta)/(kT))$, from which we see that $m = 1/(kT)$ is an inverse temperature parameter (k is Boltzmann's constant). Increasing m amounts to cooling the distribution.

Gibbs sampling from the new distribution is usually straightforward given a sampler for the original. It is perhaps not obvious why sampling from a more complex system could improve performance, but we have added considerable parallelism since we can sample various "copies" of the system concurrently. It will turn out this approach is complementary to using a factored form for the posterior. Together these methods gives us orders-of-magnitude speedup over other samplers for LDA.

The rest of the paper is organized as follows. Section 2 summarizes related work on parameter inference for probabilistic models and their limitations. Section 3 introduces the SAME sampler. We discuss in Section 4 a factored approximation that considerably accelerates sampling. A hardware-optimized implementation of the algorithm for LDA is

in Gibbs samplers

described in Section 5. Section 6 presents the experimental results and finally Section 7 concludes the paper.

4.2 Related Work

EM and related Algorithms

The Expectation-Maximization (EM) algorithm [28] is a popular method for parameter estimation of graphical models of the form we are interested in. The EM algorithm alternates between updates in the expectation (E) step and maximization (M) step. The E-step marginalizes out the latent variables and computes the expectation of the likelihood as a function of the parameters. The E step computes a Q function $Q(\Theta'|\Theta) = E_{Z|\Theta}(\log P_X(Z, \Theta'))$ to be optimized in the M-step.

For EM to work, one has to compute the expectation of the sufficient statistics of the likelihood function, where the expectation is over $Z|\Theta$. It also requires a method to optimize the Q function. In practice, the iterative update equations can be hard to derive. Moreover, the EM algorithm is a gradient-based method, and therefore is only able to find locally-optimal solutions.

Variational Bayes (VB) [49] is an EM-like algorithm that uses a parametric approximation to the posterior distribution of both parameters and other latent variables, and attempts to optimize the fit (e.g. using KL-divergence) to the observed data. Parameter estimates are usually taken from the means of the parameter approximations (the hyperparameters), unlike EM where point parameter estimates are used. It is common to assume a coordinate-factored form for this approximate posterior. The factored form simplifies inference, but makes strong assumptions about the distribution (effectively eliminating interactions). It makes most sense for parameter estimates, but is a strong constraint to apply to other latent variables. VB has been criticized for lack of quality (higher loss) on certain problems because of this. Nevertheless it works well for many problems (e.g. on LDA). This motivated us to introduce a similar factored approximation in our method described laster.

Gibbs Sampling

Gibbs samplers [37] work by stepping through individual (or blocks of) latent variables, and in effect perform a simulation of the stochastic system described by the joint distribution. The method is simple to apply on graphical models since each variable is conditioned locally by its Markov blanket. They are easy to define for conjugate distributions and can generalize to non-conjugate graphical models using e.g. slice sampling. Gibbs samplers are often the first method to be derived, and sometimes the only practical learning strategy for complex graphical models.

Gibbs samplers only require that the joint density is strictly positive over the sample space which is known as the Gibbs distribution. However, Gibbs sampling only gives samples from

the distribution of Θ, it can be difficult to find ML or MAP values from those samples. Furthermore, it can be very slow, especially for models with high dimensions. Our results suggest this slow convergence is often due to large variance in the parameters in standard samplers.

Monte Carlo EM

Monte Carlo EM [86] is a hybrid approach that uses MCMC (e.g. Gibbs sampling) to approximate the expected value $E_{Z|\Theta}(\log P_X(Z, \Theta'))$ with the mean of the log-likelihood of the samples. The method has to optimize $Q(\Theta'|\Theta)$ using a numerical method (conjugate gradient etc.). Like standard EM, it can suffer from convergence problems, and may only find a local optimum of likelihood.

Message-Passing Methods

Belief propagation [88] and Expectation propagation [67] use local (node-wise) updates to infer posterior parameters in graphical models in a manner reminiscent of Gibbs sampling. But they are exact only for a limited class of models. Recently variational message-passing [90] has extended the class of models for which parametric inference is possible to conjugate-exponential family graphical models. However similar to standard VB, the method uses a coordinate-factored approximation to the posterior which effectively eliminates interactions (although they can be added at high computational cost by using a factor graph). It also finds only local optima of the posterior parameters.

4.3 SAME Parameter Estimation

SAME estimation involves sampling multiple Z's independently and inferring Θ using the aggregate of Z's.

Method

We use the notation $Z_{-i} = Z_1, \ldots, Z_{i-1}, Z_{i+1}, \ldots, Z_n$ and similarly for Θ_{-i}.

Algorithm 1 Standard Gibbs Parameter Estimation

1: initialize parameters Θ randomly, then in some order:
2: Sample $Z_i \sim P_X(Z_i|Z_{-i}, \Theta)$
3: Sample $\Theta_i \sim P_X(\Theta_i|\Theta_{-i}, Z)$

Sampling $Z_i^{(j)}$ in the SAME sampler is exactly the same as for the standard sampler. Since the groups $Z^{(j)}$ are independent of each other, we can use the sampling function for the original distribution, conditioned only on the other components of the same group: $Z_{-i}^{(j)}$.

Algorithm 2 SAME Parameter Estimation

1: initialize parameters Θ randomly, and in some order:
2: Sample $Z_i^{(j)} \sim P_X(Z_i^{(j)}|Z_{-i}^{(j)}, \Theta)$
3: Sample $\Theta_i \sim P_X(\Theta_i|\Theta_{-i}, Z^{(1)}, \ldots, Z^{(m)})$

Sampling Θ_i is only slightly more complicated. We want to sample from

$$P_X(\Theta_i|\Theta_{-i}, Z^{(1)}, \ldots, Z^{(m)}) = P_X(\Theta, \overline{Z})/P_X(\Theta_{-i}, \overline{Z}) \tag{4.3}$$

where $\overline{Z} = Z^{(1)}, \ldots, Z^{(m)}$ and if we ignore the normalizing constants:

$$P_X(\Theta, \overline{Z})/P_X(\Theta_{-i}, \overline{Z}) \propto P_X(\Theta, \overline{Z}) = \prod_{j=1}^{m} P_X(\Theta, Z^{(j)}) \propto \prod_{j=1}^{m} P_X(\Theta_i|\Theta_{-i}, Z^{(j)}) \tag{4.4}$$

which is now expressed as a product of conditionals from the original sampler $P_X(\Theta_i|\Theta_{-i}, Z^{(j)})$. Inference in the new model will be tractable if we are able to sample from a product of the distributions $P_X(\Theta_i|\Theta_{-i}, Z^{(j)})$. This will be true for many distributions. e.g. for exponential family distributions in canonical form, the product is still an exponential family member. A product of Dirichlet distributions is Dirichlet etc., and in general this distribution represents the parameter estimate obtained by combining evidence from independent observations. The normalizing constant will usually be implied from the closed-form parameters of this distribution.

Adjusting sample number m at different iterations allows annealing of the estimate.

4.4 Coordinate-Factored Approximation

Certain distributions (including LDA) have the property that the latent variables Z_i are independent given X and Θ. That is $P(Z_i|Z_{-i}, X, \Theta) = P(Z_i|X, \Theta)$. Therefore the Z_i's can be sampled (without approximation) in parallel. Furthermore, rather than a single sample from $P(Z_i|X, \Theta)$ (e.g. a categorical sample for a discrete Z_i) we can construct a SAME Gibbs sampler by taking m samples. These samples will now have a multinomial distribution with count m and probability vector $P(Z_i|X, \Theta)$. Let $\hat{Z}_i(v)$ denote the count for $Z_i = v$ among the m samples, and $P(Z_i = v|X, \Theta)$ denote the conditional probability that $Z_i = v$.

We can introduce still more parallelism by randomizing the order in which we choose which Z_i from which to sample. The count m for variable Z_i is then replaced by random variable $\hat{m} \sim \text{Poisson}(m)$ and the coordinate-wise distributions of $\hat{Z}_i$ become independent Poisson variables:

$$\hat{Z}_i(v) \sim \text{Poisson}(mP(Z_i = v|X, \Theta)) \tag{4.5}$$

when the Z_i are independent given X, Θ, the counts $\hat{Z}_i(v)$ fully capture the results of taking the m (independent) samples. These samples can be generated very fast, and completely

in parallel. m is no longer constrained to be an integer, or even to be > 1. Indeed, each sample no longer corresponds to execution of a block of code, but is simply an increment in the value m of the Poisson random number generator. In LDA, it is almost as fast to generate all m samples for a single word for a large m (up to about 100) as it is to generate one sample. This is a source of considerable speedup in our LDA implementation.

For distributions where the Z_i are not independent, i.e. when $P(Z_i|Z_{-i}, X, \Theta) \neq P(Z_i|X, \Theta)$, we can still perform independent sampling **a**s an approximation. This approach is quite similar to the coordinate-factored approximation often used in Variational Bayes. We leave the details to a forthcoming paper.

4.5 Implementation of SAME Gibbs LDA

Our SAME LDA sampler implementation is described in Algorithm 3. Samples are taken directly from Poisson distributions (line 7 of the algorithm) as described earlier.

Latent Dirichlet Allocation

LDA is a generative process for modeling a collection of documents in a corpus. In the LDA model, the words $X = \{x_{d,i}\}$ are observed, the topics $Z = \{z_{d,i}\}$ are latent variables and parameters are $\Theta = (\theta, \phi)$ where θ is document topic distributions and ϕ is word topic distributions. Subscript d, i denotes document d and its i^{th} word. Given X and Θ, $z_{d,i}$ are independent, which also implies that we can sample from Z without any information about Z from previous samples. A similar result holds for the parameters, which can be sampled given only the current counts from the Z_i. We can therefore alternate parameter and latent variable inference using a pair of samplers: $P_X(Z|\theta, \phi)$ and $P_X(\theta, \phi|Z)$. Such blocked sampling maximizes parallelism and works very well with modern SIMD hardware.

The sampling of $z_{d,i}$'s uses the Poisson formula derived earlier. The conditional distributions $P_X(\theta, \phi|z_{d,i})$ are multiple independent Dirichlet's. In practice, we collapse out (θ, ϕ), so we in effect sample a new $z^t_{d,i}$ given the z^{t-1} from a previous iteration. The update sampler follows a Dirichlet compound multinomial distribution (DCM) and can be derived as,

$$P_X(z^t_{d,i}|z^{t-1}, \alpha, \beta) = \frac{c_{k,d,\cdot}/m + \alpha}{c_{\cdot,d,\cdot}/m + K\alpha} \frac{c_{k,\cdot,w}/m + \beta}{c_{k,\cdot,\cdot}/m + W\beta}, \tag{4.6}$$

where W and K are numbers of words and topics respectively, c are counts across all samples and all documents which are defined as,

$$c_{k,d,w} = \sum_{j=1}^{m} \sum_{i=1}^{N_d} 1(z^{(j)}_{d,i} = k \text{ and } x_{d,i} = w), \tag{4.7}$$

where N_d is the number of documents and superscript (j) of z denotes the j^{th} sample of the hidden topic variable. In Equation 4.6, dot(s) in the subscript of c denotes integrating

(summing) over that dimension(s), for example, $c_{k,d,\cdot} = \sum_{w=1}^{W} c_{k,d,w}$. As shown in Equation 4.6, sample counts are sufficient statistics for the update formula.

Mini-batch Processing

For scalability (to process datasets that will not fit in memory), we implement LDA using a mini-batch update strategy. In mini-batch processing, data is read from a dataset and processed in blocks. Mini-batch algorithms have been shown to be very efficient for approximate inference [17, 16].

In Algorithm 3, D_t is a sparse (nwords x ndocs) matrix that encodes the subset of documents (mini-batch) to process at period t. The global model (across mini-batches) is the word-topic matrix ϕ. It is updated as the weighted average of the current model and the new update, see line 14. The weight is determined by $\rho_t = (\tau_0 + t)^{-\gamma}$ according to [44]. We do not explicitly denote passes over the dataset. The data are treated instead as an infinite stream and we examine the cross-validated likelihood as a function of the number of mini-batch steps, up to t_{max}.

Algorithm 3 SAME Gibbs LDA

1: **for** $t = 0 \rightarrow t_{max}$ **do**
2: $\hat{\theta} = 0$; $\hat{\phi} = 0$; $\rho_t = (\tau_0 + t)^{-\gamma}$
3: $\mu = SDDMM(\theta, \phi, D_t)$
4: **for all** document-word pair (d, w) in mini-batch D_t **parallel do**
5: **for** $k = 1 \rightarrow K$ **parallel do**
6: $\lambda = \theta_{d,k}\phi_{k,w}/\mu_{d,w}$
7: sample $z \sim \text{Poisson}(\lambda \cdot m)$
8: $\hat{\theta}_{d,k} = \hat{\theta}_{d,k} + z/m$
9: $\hat{\phi}_{k,w} = \hat{\phi}_{k,w} + z/m$
10: **end for**
11: **end for**
12: $\theta = \hat{\theta} + \alpha$; $\phi = \hat{\phi} + \beta$
13: normalize $\hat{\phi}$ along the word dimension
14: $\phi = (1 - \rho_t)\phi + \rho_t\hat{\phi}$
15: **end for**

GPU optimizations

GPUs are extremely well-suited to Gibbs sampling by virtue of their large degree of parallelism, and also because of their extremely high throughput in non-uniform random number generation (thanks to hardware-accelerated transcendental function evaluation). For best performance we must identify and accelerate the bottleneck steps in the algorithm. First, computing the normalizing factor in equation 4.6 is a bottleneck step. It involves evaluating

the product of two dense matrix $A \cdot B$ at only nonzeros of a sparse matrix C. Earlier we developed a kernel (SDDMM) for this step which is a bottleneck for several other factor models including our online Variational Bayes LDA and Sparse Factor Analysis [23].

Second, line 7 of the algorithm is another dominant step, and has the same operation count as the SDDMM step. We wrote a custom kernel that implements lines 4-11 of the algorithm with almost the same speed as SDDMM. Finally, we use a matrix-caching strategy [22] to eliminate the need for memory allocation on the GPU in each iteration.

4.6 Experiments

LDA

In this section, we evaluate the performance of the SAME/Factored Gibbs sampler against several other algorithms and systems. We implemented LDA using our SAME approach, VB online and VB batch, all using GPU acceleration. The code is open source and distributed as part of the BIDMach project [22, 23] on github [2]. We compare our systems with four other systems: 1) Blei et al's VB batch implementation for LDA [11], 2) Griffiths et al's collapsed Gibbs sampling (CGS) for LDA [40], 3) Yan et al's GPU parallel CGS for LDA [91], 4) Ahmed et al's cluster CGS for LDA [4].

1) and 2) are both C/C++ implementations of the algorithm [3] on CPUs. 3) is the state-of-the-art GPU implementation of parallel CGS. To our best knowledge, 3) is the fastest single-machine LDA implementation to date, and 4) is the fastest cluster implementation of LDA.

All the systems/algorithms are evaluated on a single PC equipped with a single 8-core CPU (Intel E5-2660) and a dual-core GPU (Nvidia GTX-690), except GPU CGS and cluster CGS. Each GPU core comes with 2GB memory. Only one core of GPU is used in the benchmark. The benchmark for GPU CGS is reported in [91]. They run the algorithm on a machine with a GeForce GTX-280 GPU. The benchmarks we use for cluster CGS were reported on 100 and 1000 Yahoo cluster nodes [4].

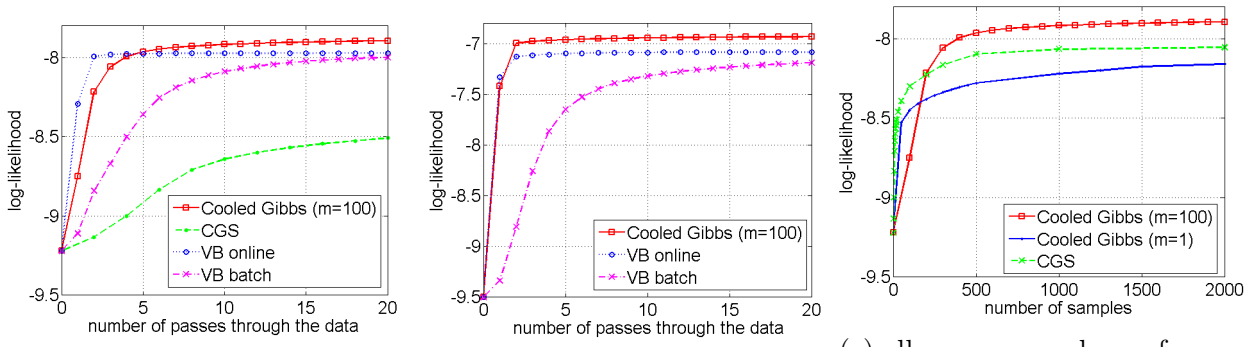
Two datasets are used for evaluation. 1) **N**YTimes. The dataset has approximately 300k New York Times news articles. There are 100k unique words and 100 million tokens in the corpus. 2) **P**ubMed. The dataset contains about 8.2 million abstracts from collections of US National Library of Medicine. There are 140k unique tokens, and 730 million words in the corpus.

Convergence

We first compare the convergence of SAME Gibbs LDA with other methods. We choose $m = 100$ for the SAME sampler, because convergence speed only improves marginally beyond

[2] `https://github.com/BIDData/BIDMach/wiki`

[3] 2) provides a Matlab interface for CGS LDA



(a) ll vs. n passes on NYTimes (b) ll vs. n passes on PubMed (c) ll vs. number of samples/word

Figure 4.1: Convergence Comparison

$m = 100$, however, runtime per pass over the dataset starts to increase noticeably beyond $m = 100$ while being relatively flat for $m < 100$. The mini-batch size is set to be $1/20$ of the total number of examples. We use the per word log likelihood $ll = \frac{1}{N^{test}} \log(P(X^{test}))$ as a measure of convergence.

Figure 4.1 shows the cross-validated likelihood as a function of the number of passes through the data for both datasets, up to 20 passes. As we can see, the SAME Gibbs sampler converges to a higher quality result than VB online and VB batch on both datasets. The SAME sampler and VB online converged after 4-5 passes for the NYTimes dataset and 2 passes for the PubMed dataset. VB batch converges in about 20 passes.

All three methods above are able to produce higher quality than CGS within 20 passes over the dataset. It usually takes 1000-2000 passes for CGS to converge for typical datasets such as NYTimes, as can be seen in Figure 4.1c.

Figure 4.1c plots log-likelihood against the number of samples taken per word. We compare the SAME sampler with $m = 100$, $m = 1$ and CGS. To reach the same number of samples CGS and the SAME sampler with $m = 1$ need to run 100 times as many passes over the data as the SAME sampler with $m = 100$. That is Figure 4.1c shows 20 passes of SAME sampler with $m = 100$ and 2000 passes of the other two methods. At the beginning, CGS converges faster. But in the long run, SAME Gibbs leads to better convergence. Notice that SAME with $m = 1$ is not identical to CGS in our implementation, because of minibatching and the moving average estimate for Θ.

Runtime Performance

We further measure the runtime performance of different methods with different implementations for LDA. Again we fix the sample size for SAME GS at $m = 100$. All the runtime number are generated by running each method/system on the NYTimes dataset with $K = 256$, except that Yan's parallel CGS [91] reports their benchmark for $K = 128$.

Table 4.1: Runtime Comparison on NYTimes (Seconds)

	SAME GS BIDMach	VB online BIDMach	VB batch BIDMach	VB batch Blei et al	CGS Griffiths et al	GPU CGS Yan et al
Runtime/Pass	30	20	20	12600	225	5.4
Time to Conv.	90	40	400	252000	225000	5400

We also report time to convergence, which is defined as the time to reach the log-likelihood for standard CGS at the 1000^{th} iteration.

Results are illustrated in the Table 4.1. The SAME Gibbs sampler takes 90 seconds to converge on the NYTimes dataset. As comparison, the other CPU implementations take around 60-70 hours to converge, and Yan's GPU implementation takes 5400 second to converge for $K = 128$. Our system demonstrates two orders of magnitude improvement over the state of the art. Our implementation of online VB takes about 40 seconds to converge on the NYTimes dataset. The Gibbs sampling method is close in performance (less than a factor of 3) to that.

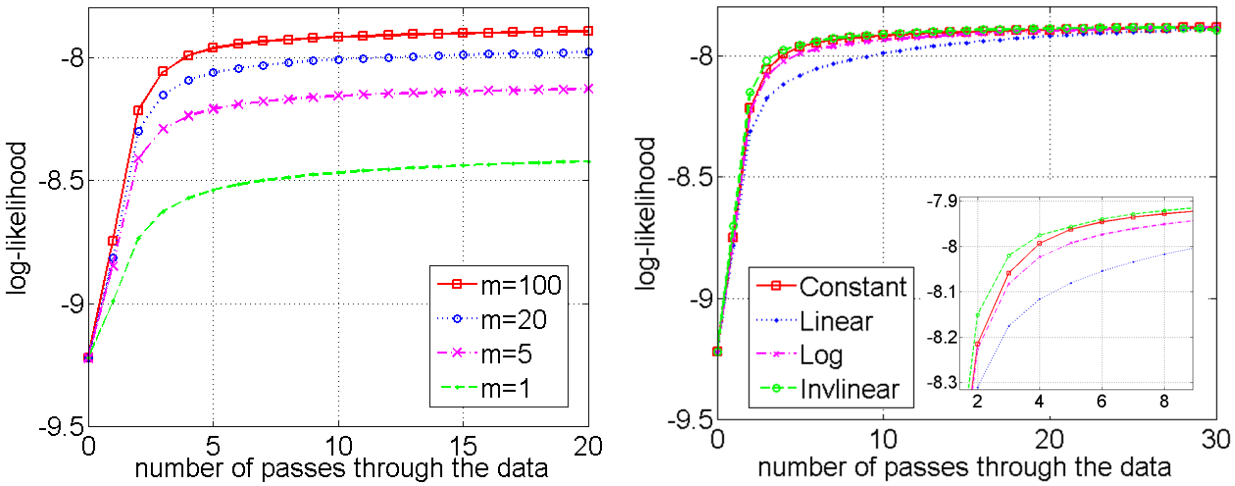
Finally, we compare our system with the state-of-art cluster implementation of CGS LDA [4], using time to convergence. Ahmed et al. [4] processed 200 million news articles on 100 machines and 1000 iterations in 2mins/iteration = 2000 minutes overall = 120k seconds. We constructed a repeating stream of news articles (as did [4]) and ran for two iterations - having found that this was sufficient for news datasets of comparable size. This took 30k seconds, which is 4x faster, on a single GPU node. Ahmed et al. also processed 2 billion articles on 1000 machines to convergence in 240k seconds, which is slightly less than linear scaling. Our system (which is sequential on minibatches) simply scales linearly to 300k seconds on this problem. Thus single-machine performance of GPU-accelerated SAME GS is almost as fast as a custom cluster CGS on 1000 nodes, and 4x faster than 100 nodes.

Multi-sampling and Annealing

The effect of sample number m is studied in Figure 4.2 and Table 4.2. As expected, more samples yields better convergences given fixed number of passes through the data. And the benefit comes almost free thanks to the SIMD parallelism offered by the GPU. As shown in Table 4.2, $m = 100$ is only modestly slower than $m = 1$. However, the runtime for $m = 500$ is much longer than $m = 100$.

We next studied the effects of dynamic adjustment of m, i.e. annealing. We compare constant scheduling (fixed m) with

1. linear scheduling, $m_t = \frac{2mt}{t_{max}+1}$,
2. logarithmic scheduling, $m_t = mt_{max} \log t / \sum_{t=1}^{t_{max}} \log t$,
3. invlinear scheduling, $m_t = \frac{2m(t_{max}+1-t)}{t_{max}+1}$.



(a) ll vs. sample size m

(b) annealing schedule

Figure 4.2: Effect of sample sizes

Table 4.2: Runtime per iteration with different m

	m=500	m=100	m=20	m=5	m=1
Runtime per iteration (s)	50	30	25	23	20

t_{max} is the total number of iterations. The average sample size per iteration is fixed to $m = 100$ for all 4 configurations. As shown in Figure 4.2b, we cannot identify any particular annealing schedule that is significantly better then fixed sample size. Invlinear is slighter faster at the beginning but has the highest initial sample number.

4.7 Summary

This paper described hardware-accelerated SAME Gibbs Parameter estimation - a method to improve and accelerate parameter estimation via Gibbs sampling. This approach reduces the number of passes over the dataset while introducing more parallelism into the sampling process. We showed that the approach meshes very well with SIMD hardware, and that a GPU-accelerated implementation of cooled GS for LDA is faster than other sequential systems, and comparable with the fastest cluster implementation of CGS on 1000 nodes. The code is at `https://github.com/BIDData/BIDMach`

SAME GS is applicable to many other problems, and we are currently exploring the method for inference on general graphical models. The coordinate-factored sampling approximation is also being applied to this problem in conjunction with full sampling (the approximation reduces the number of full samples required) and we anticipate similar large improvements in speed. The method provides the quality advantages of sampling and an-

nealed estimation, while delivering performance with custom (symbolic) inference methods. We believe it will be the method of choice for many inference problems in future.

Chapter 5

Fast Parallel Bayesian Network Inference with Gibbs Sampling

5.1 Introduction

A Bayesian network [48] is defined by a directed acyclic graph (DAG) $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ and a set of local parameters Θ. Vertex set $\mathcal{V}$ in DAG $\mathcal{G}$ represents variables of interest and edge set $\mathcal{E}$ encodes direct dependencies between those variables. Without loss of generality, we assume the variables are binary in this paper. Parameter Θ describes local conditional probability distributions (CPD). Denote $\theta_{X_v|\pi_v} = \log P(X_v|\pi_v)$. Here we use X_* to denote random variable(s) that corresponds to the subscript $*$, where $*$ can either be a node $v \in \mathcal{V}$ or a set of nodes $\mathcal{C} \subseteq \mathcal{V}$. We further define π_v a set of random variables that are parents of v and $\pi_\mathcal{C}$ the union of parent sets of every $v \in \mathcal{C}$.

Given $\mathcal{G}$ and Θ, the joint distribution of the graph is uniquely determined by,

$$\log P(X_\mathcal{V}) = \sum_{v \in \mathcal{V}} \log P(X_v|\pi_v) = \sum_{v \in \mathcal{V}} \theta_{X_v|\pi_v}. \tag{5.1}$$

The inference problem is to find the distribution of unobserved variables given partial observations for a network $(\mathcal{G}, \Theta)$. This also a key step of estimating Θ when the parameter is unknown (using EM algorithm). Here we apply Gibbs sampling because exact inference (junction tree) is usually intractable.

Gibbs sampling [37] is a special case of MCMC simulation. It works by stepping through each unobserved variables of $X - \mathcal{V}$ and sampling from its full conditionals (given the full configuration of the graph except the current variable to be sampled). Gibbs sampling is by nature a sequential problem. Before formulate the algorithm into matrix operations, we first look for parallelization opportunities of the problem by exploiting the conditional dependency of the graph $\mathcal{G}$.

5.2 Graph Coloring

The idea is to apply graph coloring to the moralized graph $\tilde{\mathcal{G}}$ of the network. One property of the Baysian network is that $u, v \in \mathcal{V}$ are independent conditioning on a set of variable $\mathcal{C}$ if $\mathcal{C}$ includes at least one variable on every path connecting u and v in $\tilde{\mathcal{G}}$ [48]. That is vertex set $\mathcal{C}$ separate the dependency between u and v.

Assume there is a k-coloring of $\tilde{\mathcal{G}}$ such that each vertex is assigned one of k colors and adjacent vertices have different colors. Denote $\mathcal{V}_c$ the set of variables assigned color c where $1 \leq c \leq k$. Our new algorithm samples sequentially from $\mathcal{V}_1$ to $\mathcal{V}_k$. Within each color group, we sample all the variables in parallel. Combining the above property and a classic result [10] from parallel scheduling, we can conclude that the new parallel sampler corresponds exactly to the execution of a sequential scan Gibbs sampler for some permutation over the variables. That is the new algorithm will converge to the desired distribution. Intuitively, this is because variables within one color group are independent to each other given all the other variables.

Finding the optimal coloring of a graph is NP-complete in general. However, we find efficient heuristics in [54] for balanced graph coloring perform well in many real word problems. Constructing balanced partition is as important as finding the smallest k.

5.3 Efficient Parallel Inference

Going back to our inference problem, sampling all variables in a color group can be formulated into matrix operations. To sample any unobserved $u \in \mathcal{V}$ using Gibbs sampling, we need to find $f_u(1)$ and $f_u(0)$ which can be evaluated in the following,

$$\begin{aligned} f_u(X_u) &\doteq \log P(X_u | X_{\mathcal{V}\setminus\{v\}}) \\ &\overset{(1)}{=} \log P(X_u | X_{\mathcal{V}\setminus\mathcal{V}_c}) \\ &\overset{(2)}{=} \log P(X_u, X_{\mathcal{V}\setminus\mathcal{V}_c}) + C_1 \\ &\overset{(3)}{=} \log \sum_{\mathcal{V}_c\setminus\{u\}} \prod_{v\in\mathcal{V}} P(X_v|\pi_v) + C_1 \end{aligned} \tag{5.2}$$

(1) is true by the dependency encoded in the network and the construction of the coloring. (2) comes from Bayes rule and C_1 does not depend on X_u. (3) expands the marginal distribution by definition.

It turns out the marginal distribution can be factorized, and $f_u(X_u)$ can be simplified,

$$\begin{aligned} f_u(X_u) &\overset{(4)}{=} \log P(X_u|\pi_u) \prod_{v\in\xi_u} P(X_v|\pi_v) \\ &\quad \times \sum_{\mathcal{V}_c\setminus\{u\}} \prod_{v\in\mathcal{V}\setminus(\xi_u\cup\{u\})} P(X_v|\pi_v) + C_1 \\ &\overset{(5)}{=} \log P(X_u|\pi_u) + \sum_{v\in\xi_i} \log P(X_v|\pi_v) + C_2 \\ &\overset{(6)}{=} \theta_{X_u|\pi_u} + \sum_{v\in\xi_i} \theta_{X_v|\pi_v} + C_2. \end{aligned} \tag{5.3}$$

(4) is a critical step for the matrix formulation. Here ξ_u is defined as the children set of node u. By the construction of the graph coloring, a key obesration is that,

$$(\{u\} \cup \pi_u \cup \xi_u \cup \pi_{\xi_u}) \cap (\mathcal{V}_c \setminus \{u\}) = \emptyset. \tag{5.4}$$

That is, node u, its parents, its children and the parents of all its children have no intersect with any nodes in the color group except u itself. This allows us to take out the term $\log P(X_u|\pi_u) \prod_{v\in\xi_u} P(X_v|\pi_v)$ from the summation, and gives us a factorized form of the marginal distribution.

(5) rearranges log and product terms. $C_2 = C_1 + \sum_{\mathcal{V}_c\setminus\{u\}} \prod_{v\in\mathcal{V}\setminus(\xi_u\cup\{u\})} P(X_v|\pi_v)$ which again does not depend on X_u, and finally (6) plugs in the definition of θ.

To sample X_u, the only term to be evaluated is the odd ratio,

$$\alpha_u = e^{(f_u(0)-f_u(1))} = e^{(g_u(0)-g_u(1))}, \tag{5.5}$$

where $g_u(X_u) = \theta_{X_u|\pi_u} + \sum_{v\in\xi_i} \theta_{X_v|\pi_v}$ and C_2 is canceled out. As a sanity check, the sampling of X_u does not depend on the assignments of any other variables in the same color group by Equation 5.4.

Finally, we formulate the sampling procedure of Bayes Net inference into parallel implementations with matrix operations. Odd ratios $\alpha_{\mathcal{V}_c}$ for any $u \in \mathcal{V}_c$ can be computed simultaneously with matrix-vector multiplies, and all $X_u's$ can be sampled in fully parallel afterwards. The implementation of the parallel BayesNet inference is described in Algorithm 4.

The key step to compute α is described in line 12 of Algorithm 4. It is implementing Equation 5.5 for all nodes in $\mathcal{V}_c$ in parallel. $A_{\mathcal{V}_c}$ is computed in line 7, where A is the adjacency matrix of $\mathcal{G}$ and I is the identical matrix. Subscripts $\mathcal{V}_c$ and $\mathcal{V}_c \cup \xi_{\mathcal{V}_c}$ are matrix slicing indices operated in rows and columns respectively. $\phi^{0,1}$ are column vectors with dimension $|\mathcal{V}|$. They represent the probabilities of each node given its parent *under the current configuration*. Subscript $\mathcal{V}_c \cup \xi_{\mathcal{V}_c}$ is again slicing indices. $\phi^{0,1}$ are computed in the

Algorithm 4 Parallel BayesNet Inference

1: A = adjacency matrix of graph $\mathcal{G}$
2: B = encoding matrix of parameter θ
3: C = offset vector of parameter θ
4: Initialize the network configuration $X^1_{\mathcal{V}}$: random for unobserved nodes, fixed for observed nodes
5: **for** $t = 1 \to t_{max}$ **do**
6: **for** $c = 1 \to k$ **do**
7: $A_{\mathcal{V}_c} = (A^T + I)_{\mathcal{V}_c, \mathcal{V}_c \cup \xi_{\mathcal{V}_c}}$
8: $Y^0 =$ set $(X^t_{\mathcal{V}_c} = 0)$ for $X^t_{\mathcal{V}}$
9: $Y^1 =$ set $(X^t_{\mathcal{V}_c} = 1)$ for $X^t_{\mathcal{V}}$
10: $\phi^0 = \theta$-lookup$(C + B \times Y^0)$
11: $\phi^1 = \theta$-lookup$(C + B \times Y^1)$
12: $\alpha_{\mathcal{V}_c} = exp\left(A_{\mathcal{V}_c} \times \left(\phi^0_{\mathcal{V}_c \cup \xi_{\mathcal{V}_c}} - \phi^1_{\mathcal{V}_c \cup \xi_{\mathcal{V}_c}}\right)\right)$
13: Sample $Z_{\mathcal{V}_c}$ uniformly (from [0,1]) at random
14: $X^{t+1}_{\mathcal{V}_c} = \mathbf{1}(Z_{\mathcal{V}_c} < \frac{1}{1+\alpha_{\mathcal{V}_c}})$
15: **end for**
16: **end for**

line $10 - 11$, with matrix multiplication and array lookups. To see how, we first realize that values in $\phi^{0,1}$ are just a subset of parameter values θ. θ can be stored in an array. $\theta_{X_u|\pi_u}$ is stored in position i_u where

$$i_u = \sum_{v \in \mathcal{S}_\sqcap} 2^{1+|\pi_v|} + hash(X_u, \pi_u). \tag{5.6}$$

Here we assume the vertex set $\mathcal{V}$ is an ordered set and set $\mathcal{S}_u$ contains all the vertices smaller than u. The first term in Equation 5.4 determines the offset C in Algorithm 4. It is the total number of parameters needed to be stored before u. And the second term - the hash function determines the encoding matrix B in Algorithm 4. We use the decimal conversion of the binary number constructed by the permutation of (X_u, π_u) as the hash function.

Notice that the actual sampling (line 13-14) is also done in matrix operations, which fully utilizes the parallelism provided by the hardware. To further increase parallelism in practice, we sample multiple observations in a batch.

5.4 Experiments

In this section, we evaluate the performance of our parallel Gibbs sampler on a real world Bayesian network application. The sampler code is open source and distributed as part of the BIDMach project [22, 23] on github [1]. We compare our system with two other popular

[1] `https://github.com/BIDData/BIDMach/wiki`

systems: Jags [72] and Microsoft Infer.net [68]. Jags is the most popular and efficient tool for Bayesian inference using the BUGS [83] model format. Jags uses Gibbs sampling as the primary inference algorithm - the same as this paper. Stan [82], a competitor of Jags, is under rapid development. However, Stan is currently lacking the support on efficient inference of discrete models, and we decide not to include it for the benchmark. Infer.net another Bayesian inference engine written in the C# language. It uses Expectation Propagation (EP) algorithm for most of inference tasks.

All the systems/algorithms are evaluated on a PC equipped with a single 4-core CPU (Intel Core i7-3667U) and a dual-core GPU (Nvidia GTX-690). Only one core of GPU is used in the benchmark. We also use Intel VTune Amplifier [46] to profile each program and measure the flops performance. VTune is very power tool: It allows user to attach any running process and monitor the all the hardware instructions executed, including both X87 legacy floating point operations and SSE operations.

Dataset

We benchmark all the systems on fitting a Bayesian network with a nation-wide examination dataset from the Dynamic Learning Maps (DLM) [32] project. The dataset contains the assessment (correct or not) of 30,000 students' responses to questions from the DLM Alternate Assessment System. There are 4000 students and 340 unique questions in the pilot experiment ,and the overall completion rate of the questions is only 2.2 % (assessment questions are tailored for each student). Each of the 340 questions is considered to be derived from a set of 15 basic concepts, and relations between questions and concepts and within concepts are given. Each question is considered as a observed node in the Bayesian network (with very high missing value rate), and each concept is considered as a hidden node which never gets observed. Each node takes a binary value. The inference task is to learn the parameter of the network on 80% of the response assessment and predict on the rest 20% of the response. We use the prediction accuracy to measure the quality of the model.

Performance and Runtime

We first look at the efficiency of each system measured by giga floating point operations per second (Gflops). Intel VTune Amplifier is used to measure the flops numbers. As presented in Figure 5.1, BIDMach achieves 5 Gflops and 1 Gflops for GPU and CPU respectively. The Gibbs sampler is bottlenecked by the calculation of sampling probability vectors which is implemented using SpMV operations. Such flops numbers are the hardware limit of SpMV operation. Jags and Infer.net operates at much lower flops rates. Note that the y-axis of the figure is in log-scale. The VTune profile results also show that Jags spend 70 % of the runtime on disk IO, which is highly inefficient. We also observe that the memory usage of Infer.net is not efficient: on our PC with 8G memory, it cannot scale up to 10000 students (with the same statistics as the DLM pilot dataset we use).

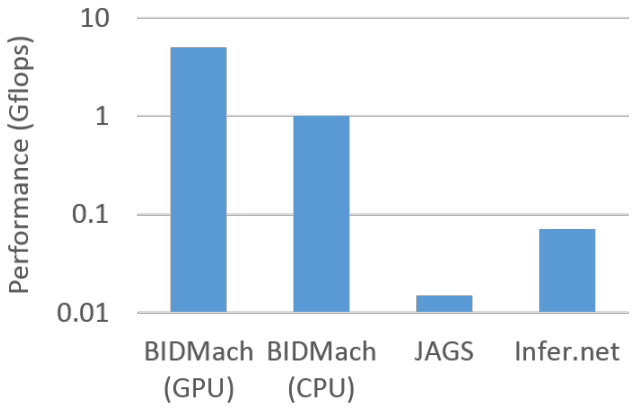


Figure 5.1: Performance Comparison

Figure 5.2 shows the runtime until convergence for each inference engine. Again, time is in log-scale. The Gibbs sample approach converges in about 200 iterations, while the EP algorithm converges in 50 iterations. Infer.net is 3.5x faster than Jags. This is expected as symbolic method is usually more efficient than sampling approach. BIDMach is 2-3 orders of magnitude faster than the other systems.

We can also verify that BIDMach is doing the same amount of work (floating point operations) as Jags by multiplying the gflops number in Figure 5.1 with the run time in Figure 5.2.

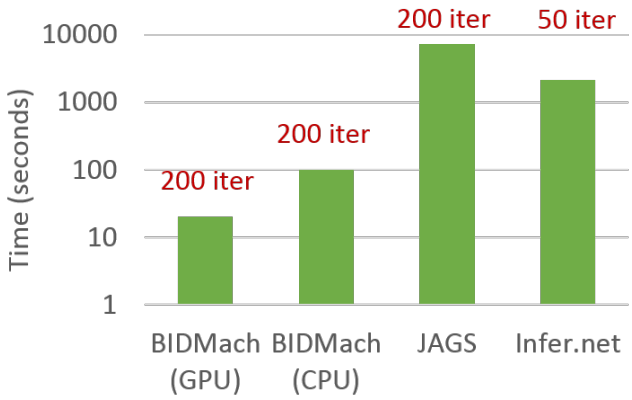


Figure 5.2: Runtime Comparison

Prediction Accuracy

For each node to be predicted, we sample 50 instances for that node from the learned network and observed values, and then take the majority as the predicted value. The accuracy is measured as the percentage of corrected predictions. A random guess will give an accuracy of 50%. Figure 5.3 shows the predicted accuracy as a function of number of iterations in training. Both BIDMach and Jags achieves 65-67% accuracy in around 200 iterations. However, BIDMach has a huge advantage in terms of speed as shown in Figure 5.2.

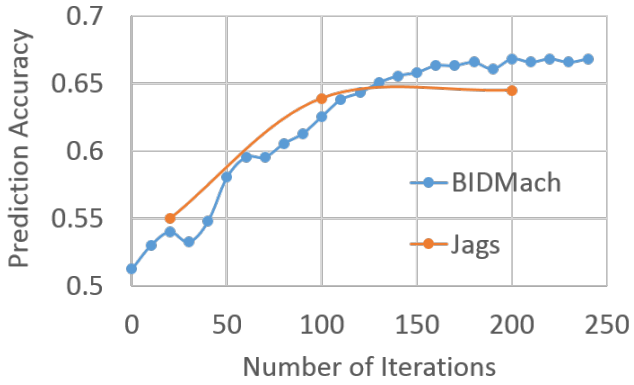


Figure 5.3: Accuracy Comparison

5.5 Summary

This paper proposes a parallel Gibbs sampling algorithm for the inference of Bayesian networks with discrete distributions. We show that the calculation of the probability distribution to sample from can be formulated as a sparse matrix - vector multiplication (SpMV), and we applied full hardware accelerations on this operation. Experiment results show that our implementation is orders of magnitude faster than other existing systems.

Chapter 6

Butterfly Mixing: Accelerating Incremental-Update Algorithms on Clusters

6.1 Introduction

The availability of massive data sources creates tremendous opportunities for business and the sciences. But harnessing this potential is challenging. Most machine learning methods involve some iteration to infer the best model, and many can be formulated directly as optimization problems. But traditional (batch) gradient-based optimization methods are too slow to run many passes over large datasets. Much faster convergence is typically obtained using stochastic gradient descent [9] - continuous model updates on relatively small subsets of the data. MCMC methods also follow this paradigm with updates ideally performed after each sample.

Hence we consider "incremental update" strategies in this paper. Both stochastic gradient and MCMC methods provide good sequential performance and have generated many of the best-performing methods for particular problems (logistic regression, SVM, LDA etc.). But these methods are difficult to adapt to parallel or cluster settings because of the overhead of distributing frequent model updates through the network. Updates can be locally batched to reduce communication overhead, but convergence typically suffers as the batch size increases - see Figure 6.3 for an example. In this paper we introduce and analyze *butterfly mixing*, an approach which *interleaves* communication with computation.

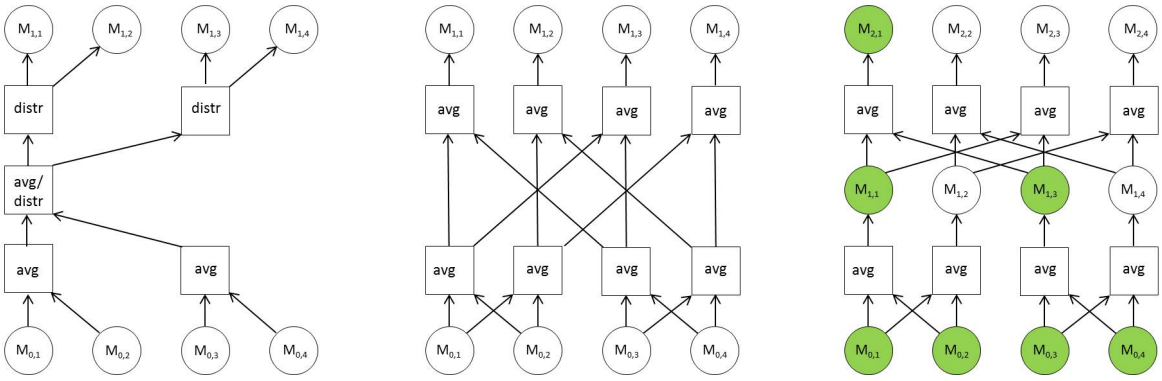
Butterfly mixing uses a butterfly network on 2^k nodes, and executes one *constant time* communication step (a butterfly shuffle) for each computation step. Butterfly mixing fully distributes model updates after k steps, but data from smaller subsets of nodes travels with lower latency. Convergence of butterfly mixing is intermediate between full model synchronization (an AllReduce operation) on every cycle which has k times the communication cost, and full AllReduce every k compute cycles which has the same communication cost as butter-

fly mixing. We show through simulation and experiment that butterfly mixing comes close to offering *the best of both worlds*: i.e. low communication cost similar to periodic AllReduce but convergence which is closer to AllReduce on every cycle. Our implementation uses a hardware accelerated (through Intel MKL) matrix library written in the Scala language to achieve state-of-the-art performance on each node.

Stochastic gradient descent is a simple, widely applicable algorithm that has proved to achieve reasonably high performance on large-scale learning problems [14]. There has been a lot of research recently on stochastic gradient [31, 14] improving gradient approximations, update schedules, and some work on distributed implementations [98, 69]. However, stochastic gradient is most efficient with small batches i.e. it is a "mostly sequential" method. When batch updates are parallelized, the batch size is multiplied by the number of parallel nodes. Local updates must then be combined by an additive or average reduction, and then redistributed to the hosts. This process is commonly known as AllReduce. AllReduce can be implemented using peer communication during gradient updates, and this has been used to improve the performance of gradient algorithms in the Map-Reduce framework[58]. However, on all but the smallest, fastest networks, AllReduce is much more expensive than a gradient update. As we will show later, AllReduce is an order of magtitude slower then an optimal-sized gradient step given both communication and computation capacity at gigabytes scale (Gbps and Gflops).

AllReduce is commonly implemented with either tree [58] or butterfly [70] topologies, as shown in Figure 6.1a and 6.1b. The tree topology uses the lowest overall bandwidth, but effectively maximizes latency since the delay is set by the slowest path in the tree. It also has no fault-tolerance. A butterfly network can be used to compute an AllReduce with half the worst-case latency. Faults in the basic butterfly network still affect the outputs, but on only a subset of the nodes. Simple recovery strategies (failover to the sibling just averaged with) can produce complete recovery since every value is computed at two nodes. Butterfly networks involve higher bandwidth, but this is not normally a problem in switched networks in individual racks. For larger clusters, a hybrid approach can use butterfly communication within each rack and then multi-node communication between racks as limited by the available bandwidth. In any case, the same strategy of interleaving communication and computation can be used.

Whatever AllReduce strategy is used, typical AllReduce communication times are significantly higher than the time to perform model updates for optimal-sized blocks of data. In the example presented in Figure 6.3, block sizes larger than 16000 (on 16 nodes that means 1000 samples per node) lead to significant increases in convergence time. Processing a 1k block of data takes less than 15 msec per node, but communicating model updates through AllReduce takes about 50 msecs (Figure 6.4a) or 200msecs (Figure 6.4b) - these numbers were for butterfly AllReduce, and the tree AllReduce times are much higher. As Figure 6.4 shows, we can achieve much lower (effectively constant) communication costs by using either butterfly reduce steps or doing a full butterfly AllReduce every k steps. The latter approach however delays global model updates by a factor of k, effectively increasing block size by k which is far from optimal. With butterfly mixing communication time is still significant, but



(a) Tree AllReduce (b) Butterfly AllReduce (c) Butterfly mixing

Figure 6.1: Different parallelization schemes for $N = 4$ nodes. Each node (circle) M_{ij} performs model update on new data at time i for node j. (a) and (b) synchronize model before every gradient step, with tree-based and butterfly AllReduce respectively. They suffer from overwhelmingly high communication overhead. (c) reduces synchronization overhead without losing convergence performance by mixing model update with communication.

we can use a batch size which is closer to optimal and achieve better overall running time for a given loss.

Contributions

In this paper, we introduce butterfly mixing and validate its performance on two stochastic gradient algorithms on two datasets. The method should provide similar gains for MCMC methods, but that is the subject of future work. It should also be of value for other optimization methods that admit incremental optimization of a model. An MPI version of butterfly mixing was implemented with high performance libraries (Intel MKL) in the Scala language. We evaluated the system for training a topic classifier on RCV1 and a sentiment predictor on a proprietary twitter dataset featuring 170 million automatically labelled tweets. We tested the system on a 64-node Amazon EC2 cluster. Experiments show that butterfly mix steps is fast and failure-tolerant, and overall achieves a 3.3x speed-up over AllReduce.

6.2 Related Works

In this section, we briefly survey some of the previous works on scalable machine learning and how it informed the goals for this paper.

In recognition of the communication and computation trade-off in stochastic gradient method, Zinkevich et al. [98] propose a simple algorithm in which multiple gradient descents run in parallels and their outputs are averaged in the end. It has also been shown that in many problems there is no advantage to running this averaging method without communi-

cation [69]. Niu et al. [69] consider a lock-free approach to parallelize stochastic gradient descent, but their focus is on single multi-core processors where access/communication times are minimal. Similar to us, Agarwal et al. [3] describe and analyse a gradient-based optimization algorithm based on delayed stochastic gradient information. However, they consider an arbitrary fixed network ignoring the costs of routing messages, and leaving out the efficiency gains of routing dynamically with a butterfly.

Hadoop MapReduce [26] is the most popular platform for distributed data processing, and Chu et al. [24] describes a general framework to run machine learning algorithms on top of it. However, Hadoop has often been criticised for using "disk for communication", and practical running times for machine learning algorithms are typically much higher than algorithms using the network for communication. In recognition of this, Spark [84, 94] was developed to allow in-memory models as RDDs. However, Spark was optimized for relatively infrequent "batch" operations on these datasets. RDDs are immutable and each iteration involves creation of a new RDD with a distributed dataflow scheduled by the Mesos system. We believe this is a much longer operation than an AllReduce step.

While one can use Hadoop and Spark for basic data management, efficient Stochastic gradient implementation requires custom communication code, which can still be done on MapReduce jobs. That was the approach used for Hadoop-compatible AllReduce [58]. These authors added a communication primitive to the map phase that allows map tasks to perform multiple AllReduce steps throughout the map phase. This approach has the same problems with AllReduce addressed by our work, and we believe would benefit from replacing the AllReduce step with a butterfly mix step. While we have used MPI for butterfly mixing, similar gains should be possible in systems that support more flexible message routing such as Hyracks [13] and DraydLINQ [93], since these systems can implement batterfly mixing directly.

6.3 Training Models with Stochastic Gradient Descent

Stochastic gradient is typically used to minimize a loss function L written as a sum of differentiable functions over data instances $L : \mathbb{R}^d \mapsto \mathbb{R}$,

$$L(\mathbf{w}) = \frac{1}{n} \sum_{i=1}^{n} L(\mathbf{w}; \mathbf{x}_i, y_i), \tag{6.1}$$

where $\mathbf{w}$ is a $d-$dimensional weight vector to be estimated, and $\mathbf{x}_i \in \mathbb{R}^d$, $y_i \in \{+1, -1\}$ are the feature vector and the label of the i^{th} example respectively.

Stochastic gradient can then be used to minimize the loss function iteratively according to

$$\mathbf{w}(t+1) = \mathbf{w}(t) - \gamma(t) H(t) \hat{\nabla} L(\mathbf{w}(t)). \tag{6.2}$$

In the formula above, $\gamma(t)$ is a time varying step size, and $H(t)$ is called preconditioner, which attempts to reduce the condition number of the system and, as a result, to ensure the fast convergence of the gradient method [79]. In this paper, we use $\gamma(t) = \frac{\gamma_0}{\sqrt{t}}$ and Jacob preconditioner to be the inverse feature frequencies [31]. This simple weighting scheme shows very good convergence on most problems.

In stochastic gradient method, $\nabla L(\mathbf{w}(t))$ is estimated by partial average of the empirical loss function,

$$\hat{\nabla} L(\mathbf{w}(t)) = \frac{1}{|\mathcal{T}_j|} \sum_{i \in \mathcal{T}_j} \nabla L(\mathbf{w}; \mathbf{x}_i, y_i) \tag{6.3}$$

where $\mathcal{T}_j$ is the j^{th} "batch", and $|\mathcal{T}_j|$ is called batch size. Stochastic gradient converges fastest with batch size 1, but in practice batch size can be increased until there is significant interaction between sample updates. This will occur when there is a lot of overlap between the (sparse) model coefficients being changed by different sample updates.

Logistic Regression

Logistic regression model [35] is among the most widely used supervised learning models.

The loss function is defined as the negative log-likelihood of the model, which can be written as,

$$L(\mathbf{w}) = \quad \frac{1}{n} \sum_{i=1}^{n} \left\{ -y_i \mathbf{w}^T \mathbf{x}_i + \log(1 + e^{\mathbf{w}^T \mathbf{x}_i}) \right\} \tag{6.4}$$

The gradient step to find optimal weight $\mathbf{w}$ in logistic regression model can be computed according to Equation 6.3 and

$$\nabla L(\mathbf{w}; \mathbf{x}_i, y_i) = \mathbf{x}_i (y_i - \frac{e^{\mathbf{w}^T \mathbf{x}_i}}{1 + e^{\mathbf{w}^T \mathbf{x}_i}}). \tag{6.5}$$

Support Vector Machine (SVM)

SVM is perhaps the most popular algorithm for training classification models. The goal of linear SVM is to find the minimal value of the following optimization problem,

$$\min_{\mathbf{w}, b} \frac{\lambda}{2} ||\mathbf{w}||^2 + \sum_{i=1}^{n} \max\{0, 1 - y_i \mathbf{w}^T \mathbf{x}_i\} \tag{6.6}$$

Shalev-Shwartz et al. [78] have shown that the primal problem with hinge loss can be solved by first updating $\mathbf{w}(t)$ using the following sub-gradient,

$$\nabla L(\mathbf{w}; \mathbf{x}_i, y_i) = \lambda \mathbf{w} - y_i \mathbf{x}_i, \tag{6.7}$$

and then scaling $\mathbf{w}(t)$ by a factor of $\min\{1, \frac{1}{\sqrt{\lambda} ||\mathbf{w}(t)||}\}$. While many algorithms for SVM have been developed, the fastest in recent years have used stochastic gradient methods [14].

6.4 Butterfly Mixing Algorithm

In this section, we present our butterfly mixing algorithm, and briefly study its convergence behaviour.

Butterfly Network

In a butterfly network, every node computes some function of its own value and one other node and outputs to itself and one other node. The neighbors at one level of the network are all neighbors in a particular dimension of a hypercube over the nodes. For our purposes, the node operation will always be an average. At the end of the process every output node holds the average of the inputs. The latency is k for 2^k nodes which is best possible for point-to-point communication. Butterfly network is failure tolerant: A fault or delay at one step only affects the subtree above it.

The resulting butterfly communication structure is illustrated in Figure 6.1c. All N nodes execute the same average algorithm so that all N partial averages are in motion simultaneously. After k steps, the average is replicated on every single node. As highlighted in Figure 6.1c with $N = 4$ nodes, at $t = 2$ step, each node already contains gradient information from all the other nodes.

Butterfly Mixing

Butterfly mixing interleaves the above average operation in butterfly network with iterative stochastic gradient updates. Denote $\mathbf{w}^k(t)$ as the weight vector available on node k at time t, and $\mathbf{g}^k(t)$ the gradient evaluated at the current position. We now present in detail the model of butterfly updates of weight vector $\mathbf{w}$.

Let S^{kj} be the set of times that weight vector is received by node j from node k. In our algorithm, S is determined by butterfly reduction structure. For instance, S^{kk} includes all time ticks up to the end of the algorithm for all $k \in \{1, 2, \ldots, N\}$, because each node "sends" gradient update to its own at each iterative step. As another example, according to butterfly structure, $S^{12} = \{1, 3, 5, \ldots\}$ for $N = 4$, which is an arithmetic sequence with common difference $k = 2$. The full reduce algorithm is presented in Algorithm 5.

Butterfly mixing is initialized with zero at the beginning. At time t, each node updates its weight vector according to messages $\mathbf{w}^j(t), \{j|t \in S^{ij}\}$ it receives, and incorporates new training examples coming in to compute its current gradient and new position. Specifically, $x^k(t)$ is updated according to the formula,

$$\mathbf{w}^k(t+1) = \frac{1}{2} \sum_{\{j|t \in S^{ij}\}} \mathbf{w}^j(t) + \gamma^k(t) H(t) \mathbf{g}^k(t+1), \tag{6.8}$$

where set $\{j|t \in S^{ij}\}$ is of cardinality 2 for all t, so that the expectation of stochastic process $\mathbf{w}^k(t)$ remains stable.

Algorithm 5 Butterfly reduce algorithm that aggregate weight vectors in a balanced pattern

function ButterflyReduce($\mathbf{W}, k, t, N$)
  $i \leftarrow mod(t, \log N)$
  $j \leftarrow k + 2^{i-1}$
  if $j > 2^i \times \lceil \frac{k-0.5}{2} \rceil$ **then**
    $j \leftarrow j - 2^i$
  end if
  return $mean(\mathbf{w}^k, \mathbf{w}^j)$
end function

Algorithm 6 Distributed stochastic gradient descent with butterfly mixing

Require: Data split across N cluster nodes
$\mathbf{w} = \mathbf{0}, t = 0$, H = inverse feature frequencies
repeat
  for all nodes k **parallel do**
    for $j = 0 \rightarrow \lfloor \frac{n}{m} \rfloor - 1$ **do**
      $\mathbf{w}^k \leftarrow$ ButterflyReduce($\mathbf{W}, k, t, N$)
      $\mathbf{g}^k \leftarrow \frac{1}{m} \sum_{i=jm}^{jm+m-1} \nabla L^i(\mathbf{w^k}; \mathbf{x}^i, y^i)$
      $\mathbf{w^k} \leftarrow \mathbf{w^k} - \gamma_t H \mathbf{g^k}$
      $t \leftarrow t + 1$
    end for
  end for
until p pass over data

The detailed butterfly mixing is presented in Algorithm 6, where $\mathbf{W}$ is an aggregation of $\mathbf{w}^k, \forall k \in \{1, 2, \ldots, N\}$. Notice that the distributed iterative update model does not guarantee the agreement on the average of weight vector $\mathbf{w}$ across nodes at any time t. However, as we will show later, the final average of $\mathbf{w}^k$ does converge in a reasonably small number of iterations. An intuitive explanation would be that butterfly reduction accelerates the convergence of $\mathbf{w}^k(t)$ to a small neighbourhood of the optimal through efficient aggregation of gradient steps across the network, while timely update of asynchronous mixing provides refined gradient direction by introducing new training examples at each mixing.

Comparisons between different reduction and mixing schemes are illustrated in Figure 6.1. The communication latency for the butterfly mixing network is the same as performing a tree AllReduce every $2k$ steps, and a butterfly AllReduce every k updates. Butterfly mixing guarantees that data are fully mixed after k steps, but because the mixing is continuous the average over smaller sub-cubes of data is available at lower latencies. We would therefore expect the butterfly mix network convergence rate to be somewhere between a network with AllReduce on every step, and periodic AllReduce every k steps. As we will see, performance is in fact closer to Allreduce on every step in terms of convergence, while the communication

cost is the same as AllReduce every k steps.

Convergence Results

We briefly present the convergence analysis of our algorithm in this subsection. A similar proof and analysis could be found in Sec. 7 of [10]. The proof consists of two major components. We first find a single vector $\mathbf{z}(t)$ to keep track of all vectors $\mathbf{w}^1(t), \mathbf{w}^2(t), \ldots, \mathbf{w}^N(t)$, simultaneously and analyze its convergence; then we show that $\mathbf{w}^k(t)$ is actually converge to $\mathbf{z}(t)$ at a certain rate. The overall convergence performance is a mixture of the above two.

In Section 6.4, vector $\mathbf{w}^k(t)$ is defined recursively in Equation 6.8. It will be useful for the analysis if we explicitly expand $\mathbf{w}^k(t)$ in terms of gradient estimates $\mathbf{g}^j(t), \forall j \in \{1, 2, \ldots, N\}, 0 < \tau < t$, that is,

$$\mathbf{w}^k(t) = \sum_{\tau=1}^{t-1} \sum_{j=1}^{N} \Phi^{kj}(t, \tau) \gamma^j(\tau) \mathbf{g}^j(\tau). \tag{6.9}$$

It can be shown that the limit of coefficient scalar $\Phi^{kj}(t, \tau), \forall k$ converges to $\Phi^j(\tau)$ linearly with rate ρ, as follows,

$$|\Phi^{kj}(t, \tau) - \Phi^j(\tau)| \leq A\rho^{t-\tau}, \forall t > \tau > 0. \tag{6.10}$$

On the other hand, it is natural to define $\mathbf{z}(t)$ that summarizes all $\mathbf{w}^k(t), \forall k \in \{1, 2, \ldots, N\}$ using the limit of $\Phi^{kj}(t, \tau)$,

$$\mathbf{z}(t) = \sum_{\tau=1}^{t-1} \sum_{j=1}^{N} \Phi^j(\tau) \gamma^j(\tau) \mathbf{g}^j(\tau), \tag{6.11}$$

Note that $\mathbf{z}(t)$ can also be expressed in a recursive way to apply Lipschiz properties,

$$\mathbf{z}(t+1) = \mathbf{z}(t) + \sum_{j=1}^{N} \Phi^k(t) \gamma^j(t) \mathbf{g}^j(t), \tag{6.12}$$

Under Lipschitz continuity assumptions on loss function L and some bounded gradient conditions, we have

$$||\mathbf{z}(t) - \mathbf{w}^k(t)||_2 \leq A \sum_{\tau=1}^{t-1} \frac{1}{\tau} \rho^{t-\tau} b(\tau), \tag{6.13}$$

$$L(\mathbf{z}(t+1)) \leq L(\mathbf{z}(t)) - \frac{1}{t} G(t) + C \sum_{\tau=1}^{t} \rho^{t-\tau} \frac{b^2(\tau)}{\tau^2}, \tag{6.14}$$

where $b(t) = \sum_{k=1}^{N} ||\mathbf{g}^k(t)||_2$ and $G(t) = -\sum_{k=1}^{N} \Phi^k(t) ||\mathbf{g}^k(t)||_2^2$, for $t \geq 1$. This concludes the convergence of the algorithm.

6.5 System Implementation

Butterfly mixing is a rather general design pattern that can be implemented on top of many systems. We start with a simple MPI version written on top of the BIDMat matrix toolkit in the Scala language: `http://bid.berkeley.edu/BIDMat/index.php/`. BIDMat inherits the REPL (command interpreter) from Scala but also runs on a JVM and so can be used in cluster toolkits like Hadoop and Spark. Its uses native code linkage to high-performance libraries including Intel MKL and HDF5 for file IO. Our system can be configured for training widely used logistic regression model and SVM. And, it can be easily extendible and suitable to any cumulative update method, and particular any gradient-based optimization algorithms.

Communication Module

We build our MPI communication framework on top of MPJ Express [76], which is an open-source implementation of Java bindings for the MPI standard. The performance of MPJ Express has been completely studied in [77] and shown to be close to a more widely used C/C++ MPI interface, Open MPI [36]. Furthermore, MPJ Express provides seamless integration with to butterfly mixing developing environment.

Four communication patterns, including butterfly mixing, have been implemented and detailed below. We also enforce synchronization using mpi barrier at the end of each communication step.

- **N**oReduce: there was no communication between nodes in the cluster.
- **C**omplete AllReduce: an MPI AllReduce was performed on every step, which is equivalent to a sequential batch SGD where batch size was single-node batch size times number of nodes. AllReduce is implemented with the MPI standard API, `public void Allreduce(...)`. We use the butterfly implementation of AllReduce in MPJ Express package with `EXOTIC_ALLREDUCE` tag turned on.
- **B**utterfly mixing: a Butterfly mix step occurred on each round, where each node send and receive updated weight $\mathbf{w}$ to/from its pair node. This procedure is implemented using the API `public Status Sendrecv(...)`.
- **P**eriodic AllReduce: A complete butterfly AllReduce was performed every $log_2 N$ steps. Periodic AllReduce has the same communication cost as butterfly mixing.

6.6 Experiments

We chose to train a standard logistic regression model on a widely used sparse training set, Reuters RCV1 [61] and a linear SVM on a filtered subset of about 6 months of data from Twitter.

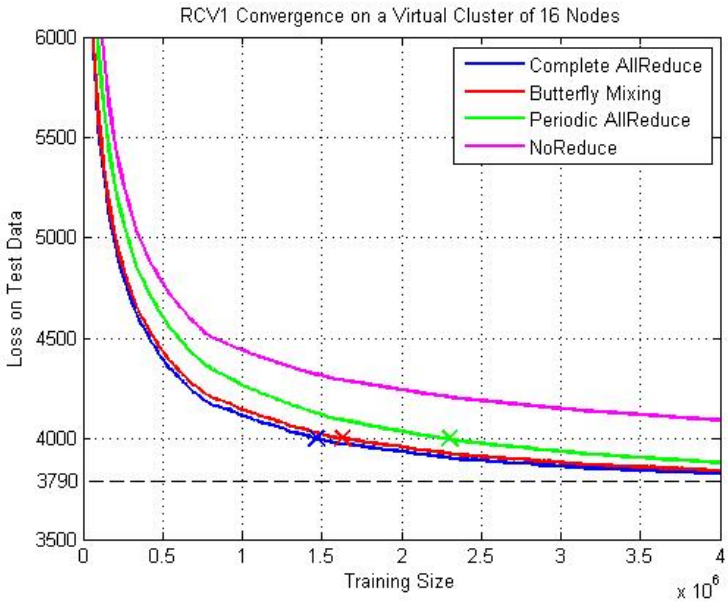


Figure 6.2: Convergence Performance with Different Communication Schemes.

Description of Datasets

The RCV1 news is standard data set for algorithm benchmarking. It consists of a collection of approximately 800,000 news articles, each of which is assigned to one or more categories. We are focusing on training binary classifier for CCAT (Commerce) category with logistic regression model of dimension 50k. The input of the classifier is an article bag-of-words feature vector with tf-idf values, and the output is a binary variable indicating whether an individual article belongs to CCAT or not. Articles are randomly shuffled and split into approximately 760,000 training examples and 40,000 test examples. Training examples are further evenly portioned to N cluster nodes.

To further evaluate the performance of butterfly mixing, we build a tweets sentiment classifier using SVM with 250k uni-gram features. We collected a filtered stream of about 6 months of tweets which contain emoticons such as:

- Positive sentiment: ":-)", ":D", ";)" etc.
- Negative sentiment: ":-(", "=(", ";(" etc.

There are about 170 million unique tweets in this dataset. Since emoticons have known positive or negative sentiment, they are used as training labels.

Simulation Study

We first study the convergence rates between different parallization schemes given training sizes. Experiments are performed on RCV1 dataset on a virtual cluster with $N = 16$ nodes.

We measured the logistic loss as a function of the number of examples processed. As is shown in Figure 6.2, in terms of loss at a fixed amount of data, the butterfly mix loss is indeed closer to complete AllReduce than periodic AllReduce. More importantly, butterfly mixing is even closer to complete AllReduce when the graph is sliced at a fixed loss value. As marked in Figure 6.2, at $loss = 4000$, butterfly mixing consumes almost as much data as complete AllReduce, while periodic AllReduce needs 60% more. Similar ratios occur at other loss values.

This result is very encouraging. On this relatively small cluster, we have been able to approach the convergence rate of complete AllReduce using only $1/log_2 N$ as much communications. We also reduced the time to reach a given loss value by roughly 30% relative to periodic AllReduce by simply reordering the gradient and mixing step.

System Performance on Real Cluster

We tested our system on both Berkeley CITRIS cluster for high-performance computing and Amazon EC2 cluster. The CITRIS cluster is made up of 33 IBM Dataplex server nodes. Each node has two Intel Xeon quad core processors (Nehelam's) at 2.7 Ghz with about 24GB of RAM. All the nodes are connected with QDR Infiniband interconnect. For the EC2 cluster, we use M3 Extra Large instances with 15G memory and 13 EC2 compute units.

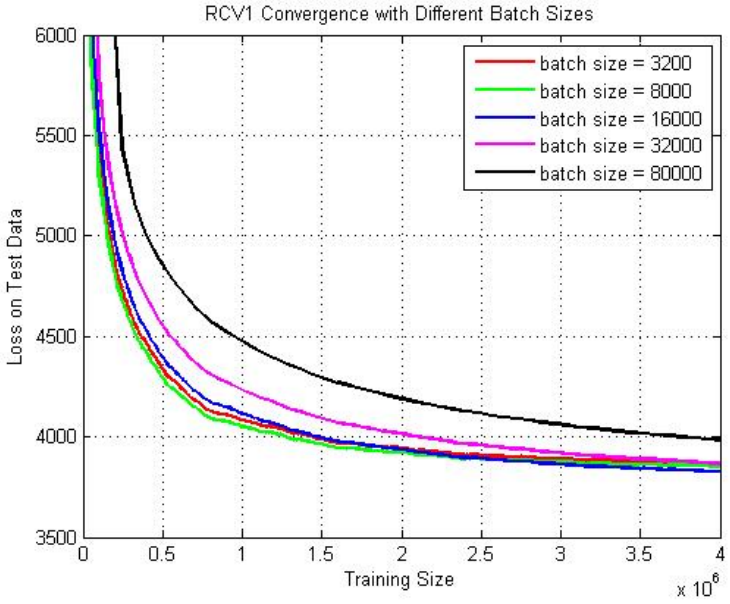
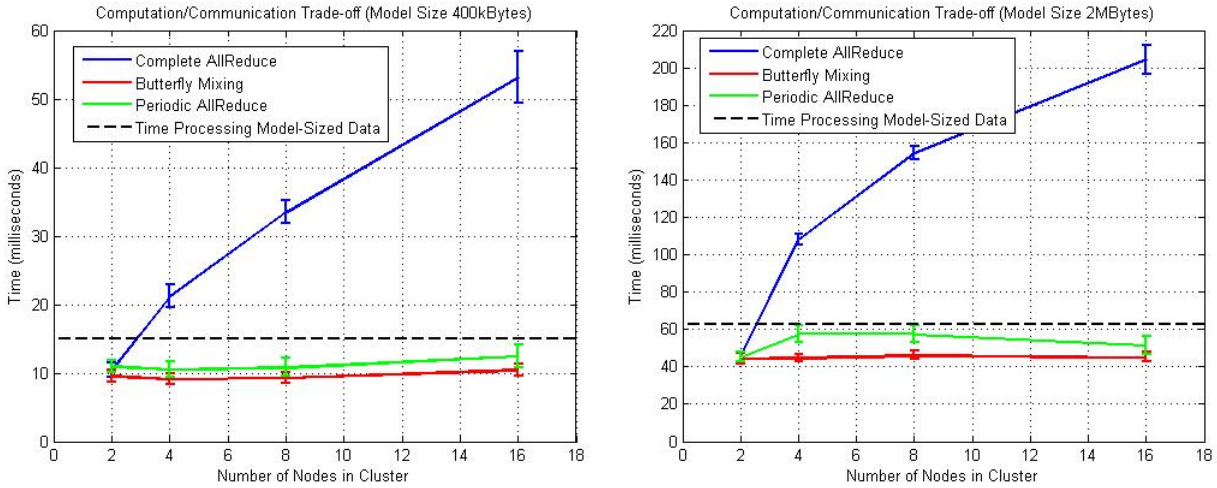


Figure 6.3: Impact of Aggregate Batch Sizes on Convergence Performance.

We evaluated the performance of the system in terms of "time to reach targeted loss"



(a) Model Size: 400kBytes (b) Model Size: 2Mbyes

Figure 6.4: Communication Overhead per Gradient Step. Dash lines show the times it takes to process model-sized training data on a single node.

T_{loss}. We can break down T_{loss} into the following two parts,

$$T_{loss} = \alpha^m \frac{S_{loss}}{N} + \beta^m(N) \frac{S_{loss}(S_{batch})}{S_{batch}} \tag{6.15}$$

where S_{batch} is the aggregate batch size across N nodes, and S_{loss} is the training size required to reach the target loss value. The first part of the formula measures computation time, where α^m is a model specific parameter that quantifies the time of processing unit size of training data for model m; and the second part is communication time, in which β^m is also model relevant and a function of cluster size N for Complete AllReduce. Note above that parallelization will provide benefit only when α^m is comparable to, if not much bigger than β^m.

It is important to notice that the training size required for convergence S_{loss} is a function of S_{batch}. As illustrated in Figure 6.3, the smaller S_{batch} is, the fewer training examples are required to reach certain loss. Interestingly, we can observe the saturation effect when S_{batch} reaches 16000: further decrease on S_{batch} does not improve the convergence performance. Actually, S_{batch} is a key parameter that determines the overall system performance. Larger batch size will indicate less communication time, however, it will also worsen the convergence performance in terms of S_{loss} which will eventually lift up T_{loss}.

To further understand the trade-off between communication and computation, we present in Figure 6.4 the benchmark of wall clock times taken for computing/communication per gradient step. Computation time should break down to CPU time for gradient step and disk time, while communication time should be a function of model sizes (model dimension $\times$ size of Double) as well as the size of cluster. We report the computation time for processing

model-sized data, this number can provide some insight on CPU/network trade-off, because optimal batch size should be multiples of model size, and "mini-batch" size processed by individual node in one round should be comparable to model size after paralellization. Results are shown in Figure 6.4, as we can see, communication time per butterfly mixing is stable across different sizes of cluster N and comparable to that of computation for both models. At the same time, communication time is much worse for AllReduce, where it increases logarithmically with N.

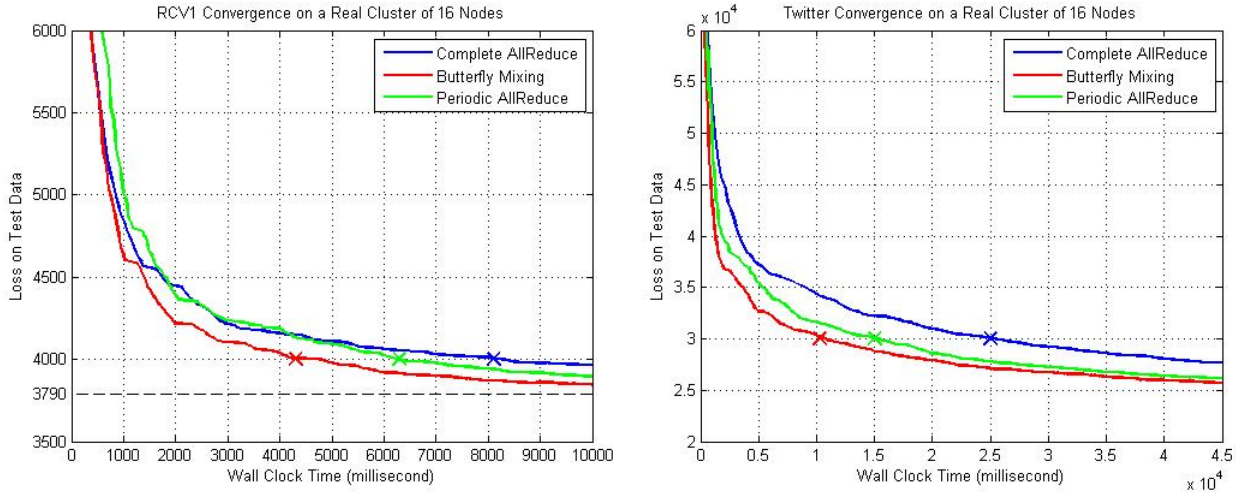
We test the algorithm performance with different communication schemes on the CITRIS cluster. S_{batch} are tuned to minimize T_{loss} for different schemes according to 6.15. Loss on test data is plotted against wall clock time for both RCV1 and Twitter datasets in Figure 6.5. This results are very promising. Butterfly mixing have been able to achieve 2.5x and 2x speed-up in comparison with complete AllReduce (at $loss = 30000$ and 4000) on Twitter and RCV1 datasets respectively. We have also saved 30% time to reach a given loss value relative to periodic AllReduce.

Finally, we explore the running time as a function of the number of nodes. We changed the number of nodes from 2 to 64 and computed the speedup ratio relative to the run with single node [1]. Speedup ratio is defined as the ratio between T_{loss}'s of different cluster setups. S_{batch} for each cluster is optimized individually to balance the communication trade-off as in 6.15. Results on Twitter dataset are reported in Figure 6.6. Butterfly mixing scales well on both clusters, providing a 5x speedup on the 16-node CITRIS cluster and 11.5x speedup on the 64-node EC2 cluster. In contrast, complete AllReduce performs badly with merely 3.5x gain on the 64-node cluster, and this verifies the argument earlier that iterative algorithms are in general difficult to adapt to parallel. In fact, AllReduce spend most time to communicate model updates across nodes, while butterfly mixing successfully mitigates this communication dilemma and achieves a 3.3x (out of 6x theoretically) performance gain over AllReduce on the 64-node cluster.

6.7 Summary

In this paper, we described butterfly mixing as an approach to improve the cluster performance with incremental update algorithms such as stochastic gradient and MCMC. Experiments were done with stochastic gradient implementations of logistic regression and SVM. Work remains to quantify the performance of butterfly mixing on different algorithms especially MCMC and on clusters with dynamic node scheduling. We will also explore the uses of butterfly mixing in conjunction with more advanced gradient estimation methods (e.g. LBFGS) which may admit larger optimal block sizes and reduce the communication "pressure".

[1] We run the experiments for cluster size 2-16 on the CITRIS cluster, and cluster size 64 on Amazon EC2.



(a) RCV1 (b) Twitter

Figure 6.5: Convergence Performance including Communication Overhead on a 16-node Cluster.

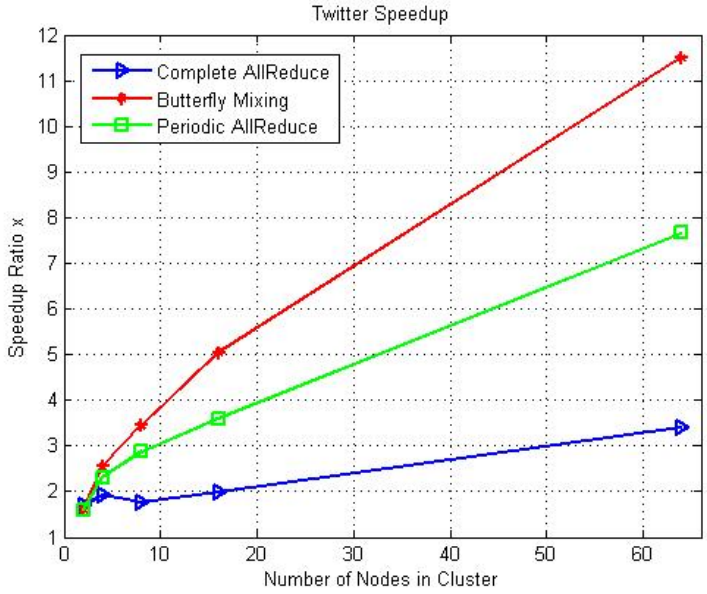


Figure 6.6: Speedup Ratio Relative to Single Processor.

Chapter 7

Kylix: A Sparse Allreduce for Commodity Clusters

7.1 Introduction

Much of the world's "Big Data" is sparse: web graphs, social networks, text, clicks logs etc. Furthermore, these datasets are well fit by power-law models. By power-law, we mean that the frequency of elements in one or both (row and column) dimensions of these matrices follow a function of the form

$$F \propto r^{-\alpha} \tag{7.1}$$

where r is the rank of that feature in a frequency-sorted list of features [2]. These datasets are large: 40 billion vertices for the web graph, terabytes for social media logs and news archives, and petabytes for large portal logs. Many groups are developing tools to analyze these datasets on clusters [62, 39, 64, 96, 50, 33, 47]. While cluster approaches have produced useful speedups, they have generally not leveraged single-machine performance either through CPU-accelerated libraries (such as Intel MKL) or using GPUs. Recent work has shown that very large speedups are possible on single nodes [23]. In fact, for many common machine learning problems single node benchmarks now dominate the cluster benchmarks that have appeared in the literature [23].

It is natural to ask if we can further scale single-node performance on clusters of fully-accelerated nodes. However, this requires proportional improvements in network primitives if the network is not to be a bottleneck. In this work we are looking to obtain an order of magnitude improvement in the throughput of the allreduce operation.

Allreduce is a general primitive that is integral to distributed graph mining and machine learning. In an Allreduce, data from each node, which can be represented as a vector (or matrix) v_i for node i, is reduced in some fashion (say via a sum) to produce an aggregate

$$v = \sum_{i=1,\ldots,m} v_i$$

and this aggregate is then shared across all the m nodes.

In many applications, and in particular when the shared data is large, the vectors v_i are sparse. And furthermore, each cluster node may not require all of the sum v but only a sparse subset of it. We call a primitive which provides this capability a *Sparse Allreduce*. By communicating only those values that are needed by the nodes Sparse Allreduce can achieve orders-of-magnitude speedups over dense approaches.

Many Big Data analysis toolkits: GraphLab/Powergraph, Hadoop, Spark etc., all use direct all-to-all communication for the Sparse Allreduce operation. That is, every feature has a home node and all updates to that feature are forwarded to the home node. The updates are accumulated and then sent to all the nodes who request the new value of that feature. Home nodes are distributed in balanced fashion across the network. Unfortunately this approach is not scalable: as the number of nodes m increases, the packet size for each message decreases as $1/m$ assuming fixed data per node. If instead the total dataset size is fixed, then message size decreases as $1/m^2$. Eventually, the time to send each message hits a floor value determined by overhead in the TCP stack and switch latencies. We show later that this limit is easily hit in practice, and is present in published benchmarks on these systems.

The aim of this paper is to develop a general efficient sparse allreduce primitive for computing on commodity clusters. The solution, called "Kylix", is a nested, heterogeneous-degree butterfly network. By heterogeneous we mean that the butterfly degree d differs from one layer of the network to another. By nested, we mean that values pass "down" through the network to implement an scatter-reduce, and then back up through the same nodes to implement an allgather. Nesting allows the return routes to be collapsed down the network, so that communication is greatly reduced in the lower layers. Because reduction happens in stages, the total data and communication volume in each layer almost always decreases (caused by collapsing of sparse elements with the same indices). This network works particularly well with power-law data which have high collision rates among the high-frequency head terms. From an analysis of power-law data, we give a method to design the optimal network (layer degrees and number of layers) for a given problem.

We next show how sparse allreduce primitives are used in algorithms such as PageRank, Spectral Clustering, Diameter Estimation, and machine learning algorithms that train on blocks (mini-batches) of data, e.g. those that use Stochastic Gradient Descent(SGD) or Gibbs samplers.

Applications

MiniBatch Machine Learning Algorithms

Recently there has been considerable progress in sub-gradient algorithms [59, 31] which partition a large dataset into mini-batches and update the model using sub-gradients. Such models achieve many model updates in a single pass over the dataset, and several benchmarks on large datasets show convergence in a single pass [59]. Most machine learning models are

amenable to subgradient optimization, and in fact it is often the most efficient method for moderate accuracy. Finally, MCMC algorithms such as Gibbs samplers involve updates to a model on every sample. To improve performance, the sample updates are batched in very similar fashion to sub-gradient updates [80].

All these algorithms have a common property in terms of the input mini-batch: if the mini-batch involves a subset of features, then a gradient update commonly uses input only from, and only makes updates to, the subset of the model that is projected onto those features. This is easily seen for factor and regression models whose loss function has the form

$$l = f(X_i v)$$

where X_i is the input mini-batch of the entire data matrix X, v is a vector or matrix which partly parametrizes the model, and f is in general a non-linear function. The derivative of loss, which defines the SGD update, has the form

$$dl/dv = f'(X_i v)X_i^T$$

from which we can see that the update is a scaled copy of X, and therefore involves the same non-zero features.

Graph Mining Algorithms

Many graph mining algorithms use repeated matrix/vector multiplication, which can be implemented using sparse allreduce: Each node i holds a subgraph whose adjacency matrix is X_i, and the input and output vectors are distributed using the allreduce. Each node requests vector elements for non-zero columns of X_i, and outputs elements corresponding to non-zero rows of X_i. Connected components, breadth-first search, and eigenvalues can be computed from such matrix-vector products. Diameter estimation algorithm in [51], the probabilistic bit-string vector is updated using matrix-vector multiplications.

To present one of these examples in a bit more detail: PageRank provides an ideal motivation for Sparse Allreduce. The PageRank iteration in matrix form is:

$$v' = \frac{1}{n} + \frac{n-1}{n} Xv \tag{7.2}$$

The dominant step is computing the matrix-vector product Xv. We assume that edges of adjacency matrix X are distributed across machines with X_i being the share on machine i, and that vertices v are also distributed (usually redundantly) across machines as v_i on machine i. At each iteration, every machine first acquires a sparse *input* subset v_i corresponding to non-zero columns of its share X_i - for a sparse graph such as a web graph this will be a small fraction of all the columns. It then computes the product $u_i = X_i v_i$. This product vector is also sparse, and its nonzeros correspond to non-zero *rows* of X_i. The input vertices v_i and the output vertices u_i are passed to a sparse (sum) Allreduce, and the result loaded into the vectors v_i' on the next iteration will be the appropriate share of the matrix

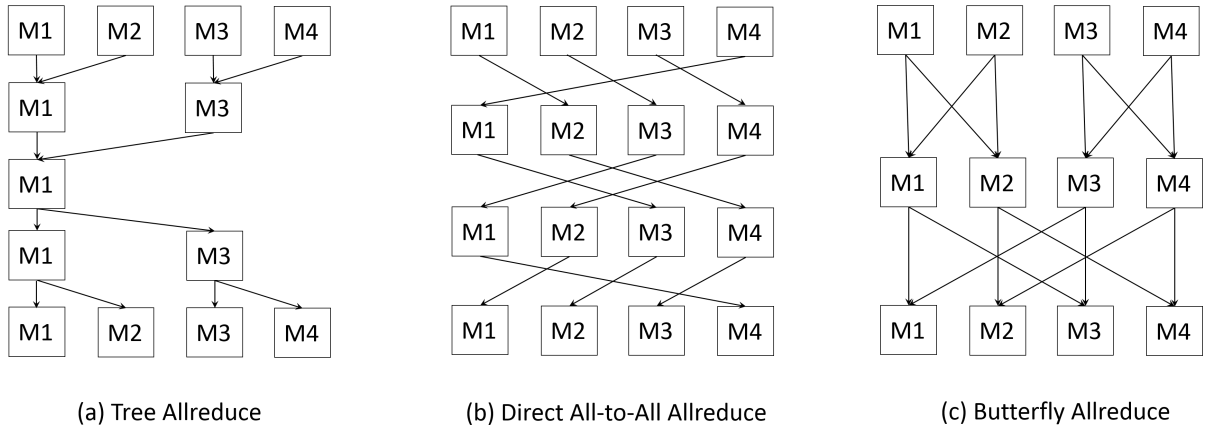


Figure 7.1: Allreduce Topologies

product Xv. Thus a requirement for Sparse Allreduce is that we be able to specify a vertex subset going in, and a different vertex set going out (i.e. whose values are to be computed and returned).

Main Contributions

The key contributions of this paper are the following:

- **K**ylix, a general and scalable sparse allreduce primitive that supports big data analysis on commodity clusters.
- A design workflow for selecting optimal parameters for the network for a particular problem.
- A replication scheme that provides a high-degree of fault-tolerance with modest overhead.
- Several experiments on large datasets. Experimental results suggest that Kylix (heterogeneous butterfly) is 3-5x faster than direct all-to-all communication (our implementation) on typical datasets. The gains for Kylix over other systems for this benchmark are somewhat higher: 3-7x (PowerGraph) and about 400x (Hadoop).

Kylix is modular and can be run self-contained using shell scripting (it does not require an underlying distributed middleware like Hadoop or MPI). Our current implementation is in pure Java, making it easy to integrate with Java-based cluster systems like Hadoop, HDFS, Spark, etc.

The rest of the paper is organized as follows. Section II reviews existing allreduce primitives for commodity data clusters, and highlights their scalability difficulties. Section III

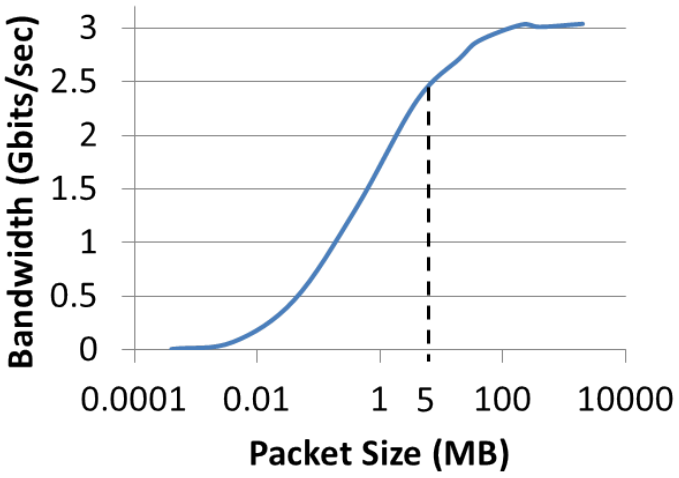


Figure 7.2: Bandwith vs. packets size

introduces Kylix, its essential features, and an example network. A design workflow for choosing its optimal parameters is discussed in Section IV. Section V and VI describe fault tolerance and optimized implementation of Kylix respectively. Experimental results are presented in Section VII. We summarize related works in Section VIII, and finally Section IX concludes the paper.

7.2 Background: Allreduce on Clusters

Cloud computing is a cost-effective, scalable technology with many applications in business and the sciences. Cloud computing is built on commodity hardware - inexpensive computer nodes and moderate performance interconnects. Cloud hardware may be private or third party as typified by Amazon EC2, Microsoft Azure and Google Cloud Platform. Cloud computers can be configured as parallel or distributed computing services but systems that use them must deal with the realities of cloud computing: (i) variable compute node performance and external loads (ii) networks with modest bandwidth and high (and variable) latency and (iii) faulty nodes and sometimes faulty communication and (iv) power-law data for most applications. Kylix addresses these challenges. Like many cloud systems, Kylix is "elastic" in the sense that its size and topology can be adapted to the characteristics of particular sparse workloads. It is fault-tolerant and highly scalable. We next discuss the tradeoffs in its design.

AllReduce

Allreduce is commonly implemented with 1) tree structure[58], 2) direct communications [30] or 3) butterfly topologies[71].

Tree Allreduce

The tree reduce topology is illustrated in Figure 7.1(a). The topology uses the lowest overall bandwidth for atomic messages, although it effectively maximizes latency since the delay is set by the slowest path in the tree. It is a reasonable solution for small, dense (fixed-size) messages. A serious limitation for sparse allreduce applications is that intermediate reductions grow in size - while some sparse terms collapse, others do not. The middle (full reduction) node will have complete (fully dense) data which will often be intractably large. It also has no fault tolerance and extreme sensitivity to node latencies anywhere in the network. It is not practical for the problems of interest to us, and we will not discuss it further.

Direct All-to-All Allreduce

In direct all-to-all allreduce (from now on we will use the term "direct allreduce" for this method), each processor communicates with all other processors. Every feature is first sent to a unique home node where it is aggregated, and the aggregate is then broadcast to all the nodes. Random partitioning of the feature set is commonly used to balance communication. Messages are typically scheduled in a circular order, as presented in Figure 7.1(b). Direct allreduce achieves asymptotically optimal bandwidth, and optimal latency *when packets are sufficiently large* to mask setup-teardown times. But there is a minimum efficient packet size that must be used, or network throughput drops (see figure 7.2). It is easy to miss this target size on a large network, and there is no way to tune the network to avoid this problem. Also, the very large (quadratic in m) number of messages make this network more prone to failures due to packet corruption, and sensitive to latency outliers.

In our experiment setup of a 64-node (`cc2.8xlarge`) Amazon EC2 cluster with 10Gb/s inter-connect, the smallest efficient packets size is 5M to mask message sending overhead (Figure 7.2). For smaller packets, latency dominates and throughput will drop. Similar observations have also been discussed in [43, 85]. In fact, scaling the cluster much beyond this limit actually *increases* the total communication time because of the increasing number of messages, reversing the advantages of parallelism.

Butterfly Network

In a butterfly network, every node computes a reduction of values from its *in* neighbours (including its own) and outputs to its *out* neighbours. In the binary case, the neighbours at layer i lie on the edges of hypercube in dimension i with nodes as vertices. The cardinality of the neighbor set is called the degree of that layer. Figure 7.1 demonstrates a 2×2 butterfly network and Figure 6.1c shows a 3×2 network. A binary butterfly gives the lowest latency for allreduce operations *when messages have fixed cost.*

In a generalized butterfly, nodes in layer i are partitioned into groups of size d_i and allreduce is performed within each group using direct allreduce. Equivalently, the $\prod d_i$

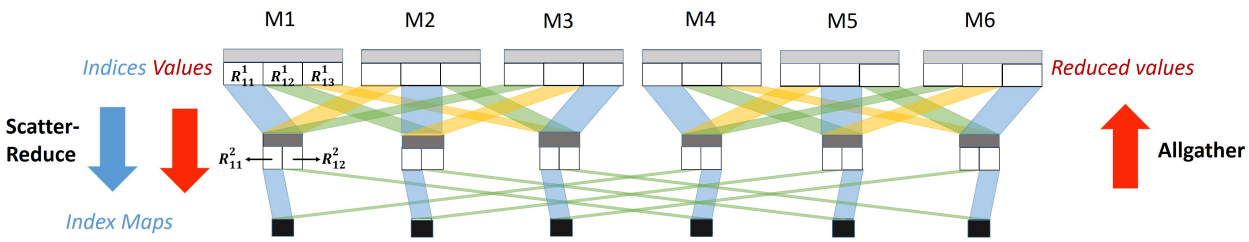


Figure 7.3: Nested Sparse Allreduce within a heterogeneous-degree (3×2) butterfly network for 6 machines. Widths of the rectangles are proportional to the range lengths for that layer; R^i_{jk} is the range of vertices sent from machine j to k at layer i. The densities (proportion of non-zeros) of the ranges are color-coded; darker indicates higher density.

nodes can be laid out on a unit grid within a hyper-rectangle of length d_i along dimension i. Then a reduction in layer i is reduction along dimension i.

Partitions of Power-Law Data

It is known that most big data (power-law or "natural graph" data [1, 60]) is difficult to partition. Thus distributed algorithms on these graphs are communication-intensive, and it is very important to have an efficient allreduce. For matrix multiply, it was shown in [39, 92], edge partitioning is more effective for power-law datasets than vertex partitioning (vertices represent row/column indices and edges represent non-zeros of the matrix) . Paper [39] describes two edge partitioning schemes, one random and one greedy. Here we will only use random edge partitioning - the precomputation needed to partition is quite significant compared to the application running time (e.g. PowerGraph takes 300s for configuration and 3.6s for runtime per iteration in [39]).

7.3 Sparse Allreduce

A sparse allreduce operation for an n-vector on a network of m nodes should have the following properties:

1. Each network node $i \in \{1, \ldots, m\}$ specifies a set of input indices $in_i \subseteq \{1, \ldots, n\}$ that it would like to receive, and a set of output indices $out_i \subseteq \{1, \ldots, n\}$ that it would like to reduce data into.

2. Node i has a vector of values $vout_i$ (typically in $\mathbb{R}^{|out_i|}$) that correspond to the indices out_i. It pushes these values into the network and receives reduced values vin_i corresponding to the indices in_i that it has asked for.

In general, in_i and out_i will be different. It must be the case that $\cup_i in_i \subseteq \cup_i out_i$ or there will be some input nodes with no data to draw from. This is typically ensured by the type of calculation: for pagerank the indices are fixed and the union of row indices should cover $\{1, \ldots, n\}$. For distributed models, every model feature should have a "home machine" which always sends and receives that feature.

Steps 1 and 2 can be performed separately or together. We call step 1 *configuration* and step 2 *reduction*. For pagerank, step 1 is done just once (in and out vertex sets are fixed throughout the calculation), with step 2 performed on every iteration. For minibatch updates, the in and out vertices change on every allreduce. In that case, it is more efficient to do configuration and reduction concurrently with combined network messages. For our nested butterfly method, we define in addition:

- l is the number of layers of the network (layers of communication), and d_i the degree of the network at layer $i \in \{1, \ldots, l\}$. We will also slightly abuse notation to refer to "node layer" i, which is used to denote the contents of a node which are the results of communication layer i. Node layers are numbered $\{0, \ldots, l\}$ with 0 at the top.
- $m_j^i \subseteq \{1, \ldots, m\}$ is the set of neighbors of machine j at layer i, then $|m_j^i| = d_i$.
- in_k^i and out_k^i are the set of in (resp. out) indices hosted by node layer i. We initialize $in_k^0 = in_k$ and $out_k^0 = out_k$.
- in_{jk}^i and out_{jk}^i are the set of in (resp. out) indices sent from node j to node k at layer i during configuration.

Configuration

Configuration has a downward pass only (Figure 4 is helful in this discussion). In this pass, indices are partitioned in node layer $i-1$, transmitted in communication layer i, and merged (by union) in node layer i. More precisely:

$(in^i_{jm^i_j(1)}, \ldots, in^i_{jm^i_j(k_i)}) = partition(in_j^{i-1})$ and

$(out^i_{jm^i_j(1)}, \ldots, out^i_{jm^i_j(k_i)}) = partition(out_j^{i-1})$

Partitioning is done into equal-size ranges of indices (this is unbalanced in general but we ensure that the original indices are hashed to the values used for partitioning). This ensures that all the indices merged in the node layer below lie in the same range, to maximize overlap.

Then the in_{jk}^i and out_{jk}^i index sets are sent to node k. Node k receives data from all its neighbors and computes the unions:

$in_k^i = \bigcup_{j \in m_k^i} in_{jk}^i$ and $out_k^i = \bigcup_{j \in m_k^i} out_{jk}^i$.

For efficiency, these union operations also generate maps $f_{jk}^i : \mathbb{N} \to \mathbb{N}$ from positions in the in-feature index sets in_{jk}^i into their union in_k^i, and $g_{jk}^i : \mathbb{N} \to \mathbb{N}$ from positions in the out-feature sets into their unions. These are used during reduction to add and project vectors in constant time per element.

Reduction

Reduction involves both a downward pass and an upward pass. The downward pass proceeds from layer $i = 1, \ldots, l$.

In the downward pass, values $vout^i_{jk}$ are sent from node j to node k in layer i. These values correspond to the indices out^i_{jk}.

As values from its layer-i neighbors are received by node k, they are summed into the total v^i_k for node k using the index map f^i_{jk}.

After l such steps, layer l contains a single copy of fully-reduced data distributed across all the nodes.

The upward pass proceeds from layer $i = l, \ldots, 1$. We define $vin^l_k = vout^l_k$ for $k = 1, \ldots, m$ to start the upward pass.

The map g^i_{jk} is used to extract the vector of values vin^i_{jk} to be sent to node j by node k from vin^i_k.

When node j receives these vectors from all of its layer-i neighbors, it simply concatenates them to form vin^{i-1}_j.

Finally vin^0_j is the desired reduced data for node j.

Configreduce

Configuration and Reduction can be performed in a single down-up pass through the network. During the downward pass from $i = 1, \ldots, l$, configuration is done on layer i, and then the downward reduction step for layer i. When we reach the bottom layer, we perform the upward reduction steps as usual. Combined configure/reduce are used for communication in algorithms where in and out vertices change on every allreduce, such as minibatch updates.

Heterogeneous Degree Butterfly

The goal of network design is to make the allreduce operation as efficient as possible. The first goal is to minimize the number of layers since layers increase latency. The total network size is the product of the layer degrees, so the larger the degrees, the fewer layers we will need. We adjust d_i for each layer to the largest value that avoids saturation (packet sizes below the minimum efficient size discussed earlier). Figure 7.3 illustrates a 3×2 network, where each processor talks to 3 neighbors in layer 1 and 2 neighbors in layer 2. In direct allreduce, packet size in each round of communication is constrained to be E_m/m where E_m is the data size on each machine. This may be too small - smaller than the minimum efficient packet. For example, in the Twitter followers' graph, the packet size is around 0.4 MB in a 64 node direct allreduce network. Figure 3 suggests that the smallest efficient packet is around 5MB for EC2, and that the B/W for 0.4MB packets is only about 30% of the efficient packet B/W.

The heterogeneity of layer degrees allows us to tailor packet size layer-by-layer. For the first layer, we know the amount of data at each node (total input data divided by m), and

we can chose d_1 accordingly. It should be the largest integer such that (node MB)/d is larger than 5MB. Later layers involve intervals of merged sparse data. The total amount of data decreases layer-by-layer because of sparse index set overlap during merges. But to determine these sizes analytically, we need to do be able to model the expected number of merged indices. We do this for power-law data next.

7.4 Tuning Layer Degrees for Power-Law Data

In this section, we describe a systematic method to determine the degrees of the network for optimal performance on power-law data.

Assume the frequency of rank-ordered features in the data vector follows a Poisson distribution with power-law rate. That is,

$$f_r \sim Poisson(\lambda r^{-\alpha}), \tag{7.3}$$

where r is the feature rank in descending order of frequency, α is the exponent of the power-law rate and λ is a scaling factor. Larger λ gives a denser vector. Let λ_0 be the scaling factor for the initial random partition of data across m machines. The expected message size for communication at each layer of butterfly network can be determined by the following proposition.

Proposition 7.4.1 *Given a dataset of n features, and a sparse allreduce butterfly network of degrees $d_1 \times d_2 \cdots \times d_l$. Define $d_0 = 1$ and $K_i = \prod_{j=0}^{i} d_i$. The vector density, i.e. the proportion of non-zero features in the vector to be reduced for each machine at layer i is*

$$D_i = \frac{1}{n} \sum_{r=1}^{n} \left(1 - \exp\left(-K_i \lambda_0 r^{-\alpha}\right)\right). \tag{7.4}$$

And the message size is

$$P_i = \frac{1}{K_i} \sum_{r=1}^{n} \left(1 - \exp\left(-K_i \lambda_0 r^{-\alpha}\right)\right). \tag{7.5}$$

Proof

Denote X_r the indicator function of the event the r^{th} feature occurs at least once in the vector at the initial partition, then

$$Pr(X_r = 1) = 1 - \exp\left(-\lambda_0 r^{-\alpha}\right) \tag{7.6}$$

according to the Poisson distribution. Then the scaling factor λ_0 is implicitly determined by the density of the initial partition at each node which is measurable:

$$\begin{aligned} D_0 &= \frac{1}{n}\mathbb{E}\left(\sum_{r=1}^{n} X_r\right) \\ &= \frac{1}{n}\sum_{r=1}^{n} 1 \cdot Pr(X_r = 1) + 0 \cdot Pr(X_r = 0) \\ &= \frac{1}{n}\sum_{r=1}^{n} \left(1 - \exp\left(-\lambda_0 r^{-\alpha}\right)\right) := f(\lambda_0). \end{aligned} \tag{7.7}$$

Notice that the density D is a function of only the scaling factor λ given the number of features n. So we could define $D := f(\lambda)$ according to Equation 7.7.

At the i^{th} layer of the network, the Poisson rate of each feature becomes $K_i\lambda_0 r^{-\alpha}$ and the scaling factor becomes $K_i\lambda_0$ since data there is a sum of data from K_i nodes. The density at layer i is $D_i = f(K_i\lambda_0)$.

At layer i the range of the vector to be reduced is $\frac{n}{K_i}$ according to our index partition. Then $P_i = D_i\frac{n}{K_i}$, which gives Equation 7.5. ∎

Figure 7.4 plots the density function f with regards to normalized scaling factor $\hat{\lambda}$, for different α. The scaling factor in the figure is normalized by $\lambda_{0.9}$, where $f(\lambda_{0.9}) = 0.9$ for the purpose of better presentation. We also zoom in the figure to show the relationship between density and λ for very low densities. Notice that the shape of the curve has only a modest dependence on α (α concentrates from 0.5 to 2 for most real world datasets).

To use Figure 7.4:

- Measure the density of the input data (could be either in our out features), i.e. the fraction of non-zeros over the number of features n. Draw a horizontal line through this density on the y-axis of the curve.
- Read off the λ value from the x-axis for this density.
- Multiply this x-value by the layer degree, to give a new x-value.
- Read off the new density from the y-axis.

This gives the density of the next layer. To compute the optimal degree for that layer, we need to know the amount of data per node. That will be the length of the partition range at that node times the density. The partition range at each layer is the total data vector length divided by all the layer degrees above that layer. i.e. the expected data size at layer i is $P = nD/\prod_{j=1,\ldots,i-1} d_j$ where D is the density we just computed (Proposition 4.1). Given this data size, we find the largest d such that P/d is at least 5 MB. Given this new d we can repeat the process one layer below etc. The steps in the method can easily be automated by tabulating the graph as a 1-1 function and its inverse to perform the needed lookups.

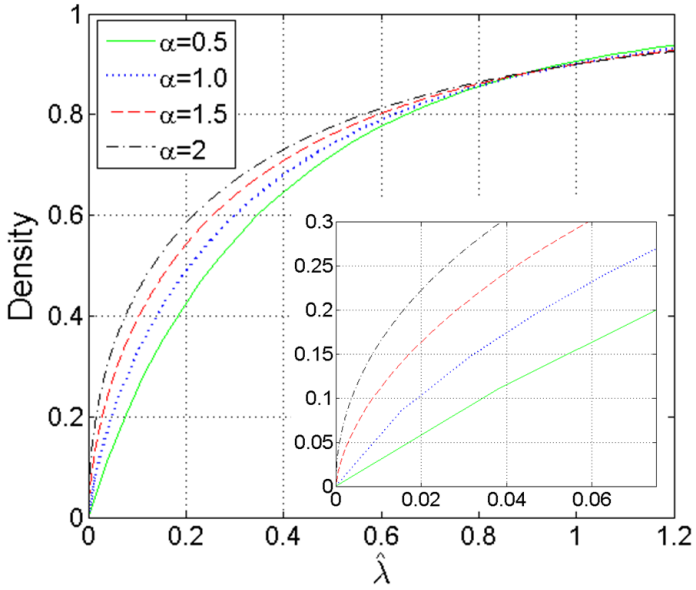


Figure 7.4: Density curve for different α

This approach gives a good starting point for optimal layer design. It will still be necessary to do some experimental tuning. For the Twitter example presented later, the experimentally optimal degree network has messages of 4-5 MB in each layer. For the larger Yahoo dataset, the optimal message sizes were 10-12 MB in the two layers of that network. Attempting to reduce the first-layer message size leads to larger messages in the second layer, which gave lower overall throughput.

The same method can be used for other sparse datasets without power-law structure. It will be necessary to construct an approximate density curve similar to figure 7.4. This involves drawing p samples from the sparse set for various p, and measuring the density. A scaled version of p should be plotted on the x-axis, with the density on y.

7.5 Fault Tolerance

Machine failure is a reality of large clusters. We next describe a simple but relatively efficient fault tolerance mechanism using replication to deal with multiple node failures [41].

Data Replication

Our approach is to replicate by a replication factor s, the data on each node, and all messages. Thus data on machine i also appears on the replicas $m+i$ through $i+(s-1)*m$. Similarly every config and reduce message targeted at node j is also sent to replicas $m+j$ through $j+(s-1)*m$. When receiving a message expected from node j, the other replicas are also listened to. The first message received is used, and the other listeners are canceled.

This protocol completes unless all the replicas in a group are dead. The expected number of node failures before this occurs is about $\sqrt{m}$ by the birthday paradox [34] for a system of replication factor 2.

Packets Racing

Replication by s increases per-node communication by s in the worst case (cancellations will reduce it somewhat). There is some performance loss because of this, as shown in the next section. On the other hand, replication offers potential gains on networks with high latency or throughput variance, because they create a race for the fastest response (in contrast to the non-replicate network which is instead driven by the *slowest* path in the network).

7.6 Implementation

On 10 Gbit networks, the overhead of computing and memory access can dominate communication. It is important for all operations to be as fast as possible. We describe below some techniques to remove potential bottlenecks.

Tree Merging

The dominant step in Kylix is merging (union of) the index sets during configuration. This is a linear time operation using hash tables, but in reality the constants involved in random memory access are too high. Instead we maintain each index set in sorted order and use merging to combine them. The merged sets must be approximately equal in length or this will not be efficient (cost of a merge is the length of the longer sequence). So we use a tree-merge. Each sequence is assigned to a leaf of a full binary tree. Nodes are recursively merged with their siblings. This was 5x faster than a hash implementation.

Multi-Threading and Latency Hiding

Scientific computing systems typically maintain a high degree of synchrony between nodes running a calculation. In cluster environments, we have to allow for many sources of variability in node timing, latency and throughput. While our network conceptually uses synchronized messages to different destinations to avoid congestion, in practice this does not give the best performance. Instead we use multi-threading and communicate opportunistically. i.e. we start threads to send all messages concurrently, and spawn a thread to process each message that is received. In the case of replicated messages, once the first message of a replicate group is received, the other threads listening for duplicates are terminated and those copies discarded. Still, the network interface itself is a shared resource, so we have to be careful that excessive threading does not hurt performance through switching of the active message thread. The effects of thread count is shown in Figure 7.7.

Language and Networking Libraries

Kylix is currently implemented using standard Java sockets. We explored several other options including OpenMPI-Java, MPJexpress, and Java NIO. Unfortunately the MPI implementations lacked key features that we needed to support multi-threading, asynchronous messaging, cancellation etc., or these features did not work through the Java interface. Java NIO was simply more complex to use without a performance advantage. All of the features we needed were easily implemented with sockets, and ultimately they were a better match for the level of abstraction and configurability that we needed.

We acknowledge that the network interface could be considerably improved. The ideal choice would be RDMA over Ethernet (RoCE), and even better RoCE directly from GPUs when they are available. This feature in fact already exists (as GPUdirect for NVIDIA CUDA GPUs), but is currently only available for infiniband networks.

7.7 Experiments

In this section, we evaluate Kylix and compare its performance with two other systems: Hadoop and PowerGraph. Two datasets are used:

1. Twitter Followers Graph [56]. The graph consists of 40 million vertices and 1.4 billion edges.

2. Yahoo! Altavista web graph [74]. This is one of the largest publicly available web graphs with 1.4 billion vertices and 6 billion edges.

The average densities of the 64-way partitioned datasets are 0.18 and 0.04 respectively. All the experiments with Kylix are performed on the Amazon EC2 commodity cluster which comprises 64 `cc2.8xlarge nodes`. Each node has an eight-core Intel Xeon E5-2670 processor and all nodes are interconnected by 10Gb/s Ethernet.

Optimal Degrees

The optimal degrees are $8 \times 4 \times 2$, and 16×4 for the Twitter followers' graph and Yahoo web graph respectively. They are determined using the method discussed in Section IV. Figure 7.5 shows the total communication volume across the network (including packets to its own) for different layers. The last layer in the figure is the total volume of fully reduced values at the bottom layer of scatter-reduce (it is the communication volume if there were an additional layer for reduce). The total communication volume is decreasing from layer to layer. The Twitter graph shrinks very fast at lower layers, because vectors communicated are dense and the collision rate is close to a hundred percent. For the Yahoo's graph, the collision rate is much lower because it is more sparse, and the volume shrinking is less significant. The shape of the total communication volume by layer looks like a Kylix, which coins the name of our system.

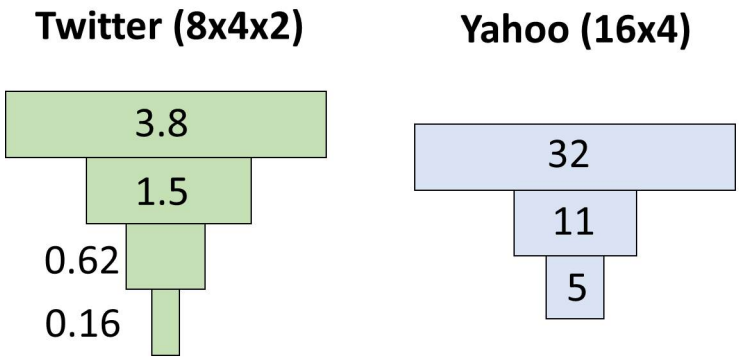
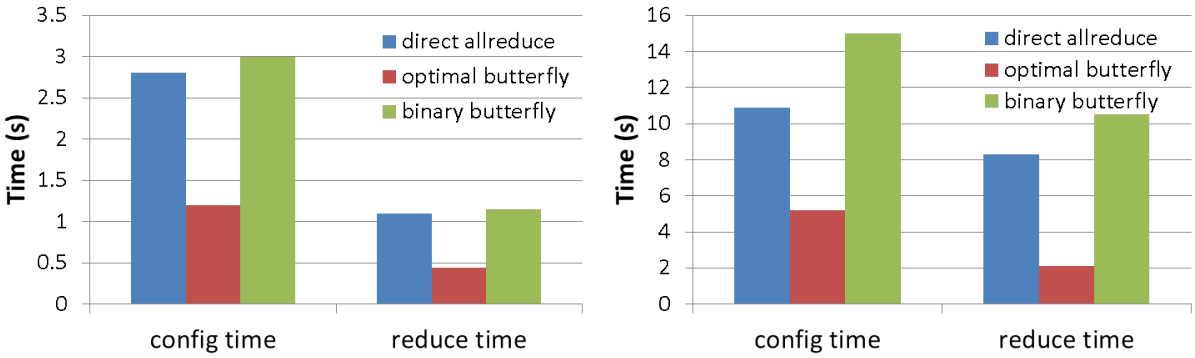


Figure 7.5: Data volumes (GB) at each layer of the protocol, resembling a Kylix. These are also the total communication volumes for all but the last layer.



(a) Twitter followers' graph

(b) Yahoo web graph

Figure 7.6: Allreduce time per iteration

Figure 7.6 plots the average config time and reduce time per iteration for different network configurations, including, direct all-to-all communication, optimal butterfly and binary butterfly, for both graphs. From the figure, we can see that optimal butterfly is 3-5x faster than the other configurations. Optimal butterfly fully utilized the network bandwidth with tuned packet sizes, whereas, the direct allreduce topology sends 0.4MB of packets each round (for the Twitter graph) which is below the optimal packet size; this packet size utilizes about 30% of the full bandwidth as we can tell from Figure 7.2. Optimal butterfly is also faster than binary butterfly since it has fewer layers and both latency and size of replicated messages to be routed are reduced.

Effect of Multi-Threading

We compare the Allreduce runtime for different thread levels in Figure 7.7. All the results are run under the $8 \times 4 \times 2$ configuration on the Twitter dataset. Significant performance improvement can be observed by increasing from single thread up to 4 threads, and it is also clear from the figure that the benefit of adding thread level is marginal beyond 16 threads

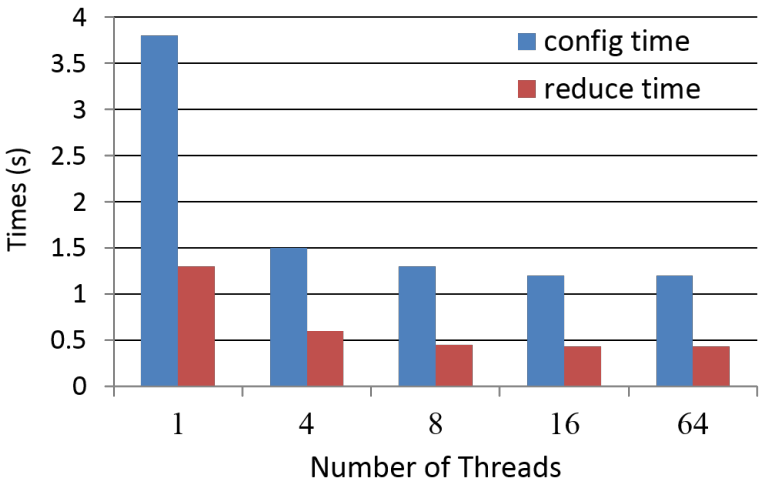


Figure 7.7: Runtime comparison between different thread levels.

Table 7.1: Cost of Fault Tolerance

System config	$8 \times 4 \times 2$ network replication=1 (64 nodes)	8×4 network replication=1 (32 nodes)	8×4 network replication=2 (64 nodes)	8×4 network replication=2 (64 nodes)	8×4 network replication=2 (64 nodes)	8×4 network replication=2 (64 nodes)
# of dead nodes	0	0	0	1	2	3
Config time (s)	1.2	1.3	1.51	1.49	1.52	1.51
Reduce time (s)	0.44	0.60	0.75	0.73	0.76	0.74

(each machine has 16 CPU threads).

Performance with Fault Tolerance

Table 7.1 demonstrates the performance with data replication on Twitter dataset. As shown in the table, the impact of data replication on runtime is moderate. The first column shows the performance of the optimal, unreplicated network on 64 nodes. The last four columns show the optimal replicated network on 64 nodes (which has data partition into 32 parts) with 0,1,2,3 failures. The second column shows an unreplicated 32-node network for reference purposes. Notice first that the runtime with failures is apparently independent of the number of failures. Notice second that replication increases the configuration time by only about 25%, and reduction time by about 60%. While the replicated network potentially does twice the work of the unreplicated network, it benefits from packet racing and early termination. For the network to fail completely, it would require about $\sqrt{m} = 8$ failures by our earlier analysis. This is very unlikely to happen for a 64-node cluster in real production settings.

We also compare the runtime for different number of machine failures in Table 7.1. Runtimes with replication are the same regardless of the number of dead nodes up to 3 tested.

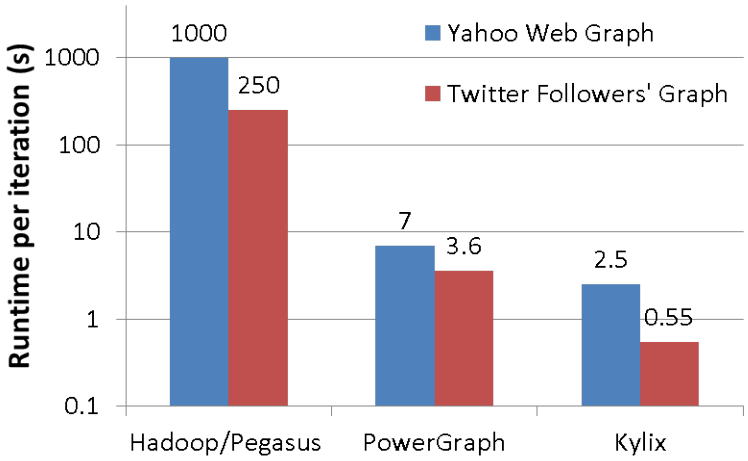


Figure 7.8: PageRank runtime comparison (log scale).

Performance and Scalability

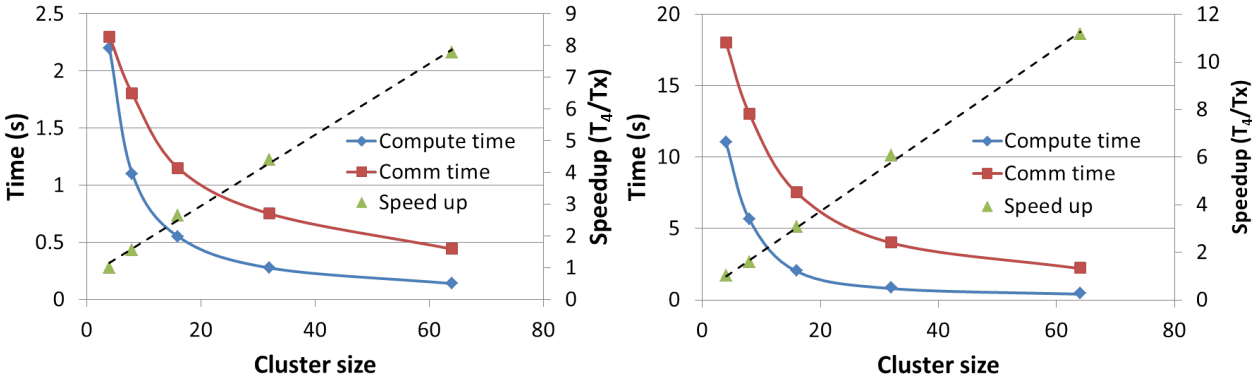
We next compared Kylix with two other systems on the PageRank algorithm. PageRank is a widely benchmarked problem for Power-law graphs and is known to be communication intensive. Our version of PageRank is implemented using BIDMat+Kylix. BIDMat [23] is an interactive matrix library written in Scala that includes hardware acceleration (Intel MKL).

The scalability of Kylix is illustrated in Figure 7.9. The figure plots the runtimes (broken down into computation and communication time per iteration) against cluster sizes. It also plots the speed up over runtime of 4 nodes (4 is the minimal cluster size such that number of edges in each partition of Yahoo graph fits into the range of `int`), which is defined as $speedup = T_4/T_x$ for cluster of size x. The butterfly degrees are optimally tuned individually for different cluster sizes.

As shown in the figure, the speed up ratio is 7-11 on 64 nodes. The optimal speedup is 16, since we are comparing with runtime on 4 nodes as baseline. The system achieves roughly linear scaling with the number of machines in the cluster. Scaling beyond this should be possible, but from the graph it can be seen that communication starts to dominate the runtime for both datasets after 32 nodes. Particularly, for our 64-node experiments, communication takes up to 75-90% of overall runtime. However, there is no computational explanation for this and we believe it is actually due to lack of synchronization (which is absorbed in the communication time measurements) of the protocol across larger networks. We are currently exploring improvements on larger networks.

Finally, we compare our system with Hadoop/Pegasus and Powergraph [1]. We chose

[1]MPI is not compared because it does not support a *Sparse* Allreduce operation. MPI dense Allreduce must take at least 100s per iteration on the Yahoo dataset assuming full (10Gb/s) bandwidth, which is orders of magnitude slower.



(a) Twitter followers' graph

(b) Yahoo web graph

Figure 7.9: Compute/Comm time break-down and speedup on a 64-node EC2 commodity cluster (512 cores in total)

Hadoop because it is a widely-known system, and Powergraph because it has demonstrated the highest performance to date for the Pagerank problem [39]. Figure 7.8 plots runtime per iteration for different systems. PowerGraph was run on a 64-node EC2 cluster with 10Gb/s interconnect - the same as this paper. Each Powergraph node has a two quad core Intel Xeon X5570 for the Twitter benchmark [39] and dual 8-core CPUs for the Yahoo benchmark [38]. Pegasus runs on a 90-node Yahoo M45 cluster. We estimate Pegasus runtime for Twitter and Yahoo graph by using their runtime result [50] on a power-law graph with 0.3 billion edges and assuming linear scaling in number of edges. We believe that the estimate is sufficient since we are only interested in the runtime in terms of order of magnitude for Hadoop-based system. The y-axis of the plot is log-scale. Kylix spends 0.55 second for on PageRank iterations on the Twitter followers' graph and 2.5 seconds for the Yahoo graph. From the graph it can be seen that Kylix runs 3-7x faster than PowerGraph, and about 400x times faster than Hadoop. We point out that the benchmark quoted for Pagerank on PowerGraph uses greedy partitioning which its authors note saves 50% runtime compared to the random partition we used [38]. Thus one would expect the performance ratio between Kylix and Powergraph to be closer to 6-14x when run on the same partioned dataset. The gap for Twitter (7x) is larger than for the Yahoo graph (3x). We expect this is because Twitter is a smaller dataset and direct messaging in Powergraph falls significantly below the efficient message minimum size. This suggests that the gap is likely to widen if calculations are run on larger clusters where message sizes will once again be small in direct allreduce.

The overall achieved bandwidth for the reduce operation is around 3 Gb/s per node on EC2 which is somewhat lower than the rated 10Gb/s of the network. It is known that standard TCP/IP socket software has many memory-to-memory copy operations, whose overhead is significant at 10Gb/s. There are several technologies available which would better this figure, however at this time there are barriers to their use on commodity clusters. RDMA over Converged Ethernet would be an ideal technology. This technology bypasses

copies in several layers of the TCP/IP stack and uses direct memory-to-memory copies to deliver throughputs much closer to the limits of the network itself. It is available currently for GPU as GPUdirect (which communicates directly from on GPU's memory to another over a network), and in Java as Sockets Direct. However, at this time both these technologies are only available for Infiniband networks. We will monitor these technologies, and we also plan to experiment with some open source projects like RoCE (RDMA over Converged Ethernet) which offer more modest gains.

7.8 Related Work

Many other distributed learning and graph mining systems are under active development at this time [62, 39, 64, 96, 47, 50]. Our work is closest to the GraphLab [62] project. GraphLab improves upon the Hadoop MapReduce infrastructure by expressing asynchronous iterative algorithms with sparse computational dependencies. PowerGraph is an improved version of GraphLab, which particularly tackles the problem of power-law data in the context of graph mining and machine learning. We focus on optimizing the communication layer of the distributed learning systems, isolating the Sparse Allreduce primitive from other matrix and machine learning modules. There are a variety of other similar distributed data mining systems [63, 51, 18] built on top of Hadoop. However, the disk-caching and disk-buffering philosophy of Hadoop, along with heavy reliance on reflection and serialization, cause such approaches to fall orders of magnitude behind the other approaches discussed here. Finally, we also distinguish our work with other dense Allreduce systems such as [65].

Our work is also related with research in distributed SpMV [27, 19], distributed graph mining [92, 19] and optimized all-to-all communications [66, 55, 65] in the scientific computing community. However, they usually deal with matrices with regular shapes (tri-diagonal), matrices desirable partition properties such as small surface-to-volume ratio, or dense matrices. We are more focused on communicating intensive algorithms posed by sparse power law data which is hard to partition. We also distinguish our work by concentrating on studying the performance trade-off on commodity hardwares, such as on Amazon EC2, as opposed to scientific clusters featuring extremely fast network connections, high synchronization and exclusive (non-virtual) machine use.

7.9 Summary

In this paper, we described Kylix, a Sparse Allreduce primitive for efficient and scalable distributed machine learning. The primitive is particularly well-adapted to the power-law data common in machine learning and graph analysis. We showed that the best approach is a hybrid between butterfly and direct all-to-all topologies, using a nested communication pattern and non-homogeneous layer degrees. We added a replication layer to the network which provides a high degree of fault tolerance with modest overhead. Finally, we presented

a number of experiments exploring the performance of Kylix. We showed that it is significantly faster than other primitives. In the future we hope to achieve further gains by using more advanced network layers that use RDMA over Converged Ethernet (RoCE), and more attention to potential clock skew across the network. Our code is open-source and freely-available, and is currently in pure Java. It is distributed as part of the BIDMat suite, but can be run standalone without other BIDMat features.

Chapter 8

Conclusions and Future Work

This dissertation presents BIDMach - a high performance toolkit for machine learning. BIDMach currently holds the performance records for a large and growing list of machine learning algorithms. The tool has been released on github under BSD-style licenses, and deployed in production at a major US technology company.

The performance of the tool is achieved by the co-design of the machine learning algorithms and the underlying hardware they run on. All the key and common kernels of the algorithms are optimized for hardware parallelism, memory throughput and networking to the theoretical limit. We also modify the algorithms to better match hardware mappings, without compromising convergence property of the algorithm. The designing guideline we use - co-design and roofline, shows success in the majority of popular machine learning algorithms we have looked at, including regression, sparse factor analysis, general graphical model inference and etc.. We believe these principles are general and can apply to other algorithms as well.

In the future, we would like to include human into the co-design loop. In fact, human is the critical part of the machine learning pipeline and human-guide learning always delivers better prediction accuracy and model interpretation at least empirically. One key feature of such system is interactivity. In practice, interaction is possible only when the system is fast, which is addressed by this thesis. We also need to build an environment that provides meaningful feedbacks and visualizations that monitors the progress of the learning process. We look forward to see such features in the future BIDMach releases.

Bibliography

[1] Amine Abou-Rjeili and George Karypis. "Multilevel algorithms for partitioning power-law graphs". In: *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*. IEEE. 2006, 10–pp.

[2] Lada A Adamic. "Zipf, power-laws, and pareto-a ranking tutorial". In: *Xerox Palo Alto Research Center, Palo Alto, CA, http://ginger. hpl. hp. com/shl/papers/ranking/ranking. html* (2000).

[3] A. Agarwal and J.C. Duchi. "Distributed delayed stochastic optimization". In: *arXiv preprint arXiv:1104.5525* (2011).

[4] Amr Ahmed et al. "Scalable inference in latent variable models". In: *Proceedings of the fifth ACM international conference on Web search and data mining*. ACM. 2012, pp. 123–132.

[5] UC Berkeley AMPLab. "Spark Kmeans Benchmark". In: (). URL: `http://apache-spark-user-list.1001560.n3.nabble.com/K-means-faster-on-Mahout-then-on-Spark-td3195.html`.

[6] Authors Anonymized. "Cooled Gibbs Parameter Estimation". In: *Submission*. 2014.

[7] Alexander Behm et al. "ASTERIX: towards a scalable, semistructured data platform for evolving-world models". In: *Distributed and Parallel Databases* 29.3 (2011), pp. 185–216.

[8] R. Bell and Y. Koren. "Scalable Collaborative Filtering with Jointly Derived Neighborhood Interpolation Weights". In: *ICDM'07*. IEEE, 2007.

[9] D.P. Bertsekas. "Nonlinear programming". In: (1999).

[10] D.P. Bertsekas and J.N. Tsitsiklis. "Parallel and distributed computation". In: (1989).

[11] David M Blei, Andrew Y Ng, and Michael I Jordan. "Latent dirichlet allocation". In: *the Journal of machine Learning research* 3 (2003), pp. 993–1022.

[12] David Blei, Andrew Ng, and Michael Jordan. "Latent Dirichlet Allocation". In: *NIPS*. 2002.

[13] V. Borkar et al. "Hyracks: A flexible and extensible foundation for data-intensive computing". In: *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*. IEEE. 2011, pp. 1151–1162.

[14] L. Bottou and O. Bousquet. "The tradeoffs of large scale learning". In: *Advances in neural information processing systems* 20 (2008), pp. 161–168.

[15] Leon Bottou. "Large-Scale Machine Learning with Stochastic Gradient Descent". In: *Proc. COMPSTAT 2010*. 2010, pp. 177–187.

[16] Léon Bottou. "Large-scale machine learning with stochastic gradient descent". In: *Proceedings of COMPSTAT'2010*. Springer, 2010, pp. 177–186.

[17] Léon Bottou and Olivier Bousquet. "The Tradeoffs of Large Scale Learning." In: *NIPS*. Vol. 4. 2007, p. 2.

[18] Yingyi Bu et al. "HaLoop: Efficient iterative data processing on large clusters". In: *Proceedings of the VLDB Endowment* 3.1-2 (2010), pp. 285–296.

[19] Aydın Buluç and John R Gilbert. "The Combinatorial BLAS: Design, implementation, and applications". In: *International Journal of High Performance Computing Applications* 25.4 (2011), pp. 496–509.

[20] John Canny. "Collaborative Filtering with Privacy via Factor Analysis". In: *ACM SIGIR 2002*. 2002.

[21] John Canny. "GAP: A Factor Model for Discrete Data". In: *ACM SIGIR*. 2004, pp. 122–129.

[22] John Canny and Huasha Zhao. "BIDMach: Large-scale Learning with Zero Memory Allocation". In: *BigLearn Workshop, NIPS*. 2013.

[23] John Canny and Huasha Zhao. "Big data analytics with small footprint: Squaring the cloud". In: *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM. 2013, pp. 95–103.

[24] C.T. Chu et al. "Map-reduce for machine learning on multicore". In: *Advances in neural information processing systems* 19 (2007), p. 281.

[25] Sudipto Das et al. "Ricardo: integrating R and Hadoop". In: *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. ACM. 2010, pp. 987–998.

[26] J. Dean and S. Ghemawat. "MapReduce: Simplified data processing on large clusters". In: *Communications of the ACM* 51.1 (2008), pp. 107–113.

[27] James Demmel et al. "Avoiding communication in sparse matrix computations". In: *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*. IEEE. 2008, pp. 1–12.

[28] Arthur P Dempster, Nan M Laird, Donald B Rubin, et al. "Maximum likelihood from incomplete data via the EM algorithm". In: *Journal of the Royal statistical Society* 39.1 (1977), pp. 1–38.

[29] Arnaud Doucet, Simon Godsill, and Christian Robert. "Marginal maximum a posteriori estimation using Markov chain Monte Carlo". In: *Statistics and Computing* 12 (2002), pp. 77–84.

[30] Jose Duato, Sudhakar Yalamanchili, and Lionel M Ni. *Interconnection networks: An engineering approach*. Morgan Kaufmann, 2003.

[31] John Duchi, Elad Hazan, and Yoram Singer. "Adaptive subgradient methods for online learning and stochastic optimization". In: *The Journal of Machine Learning Research* 12 (2011), pp. 2121–2159.

[32] "Dynamic Learning Maps". In: (). URL: `http://dynamiclearningmaps.org/`.

[33] Jaliya Ekanayake et al. "Twister: a runtime for iterative mapreduce". In: *Proc. 19th ACM HPDC*. 2010, pp. 810–818.

[34] Philippe Flajolet, Daniele Gardy, and Loÿs Thimonier. "Birthday paradox, coupon collectors, caching algorithms and self-organizing search". In: *Discrete Applied Mathematics* 39.3 (1992), pp. 207–229.

[35] J. Friedman, T. Hastie, and R. Tibshirani. *The elements of statistical learning, second edition*. Springer Series in Statistics, 2009.

[36] Edgar Gabriel et al. "Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation". In: *Proceedings, 11th European PVM/MPI Users' Group Meeting*. Budapest, Hungary, Sept. 2004, pp. 97–104.

[37] Stuart Geman and Donald Geman. "Stochastic relaxation, Gibbs distributions, and the Bayesian restoration of images". In: *Pattern Analysis and Machine Intelligence, IEEE Transactions on* 6 (1984), pp. 721–741.

[38] Joseph Gonzalez. "Thesis defense presentation". In: (2012). URL: `http://www.cs.berkeley.edu/~jegonzal/talks/jegonzal_thesis_defense.pptx`.

[39] Joseph E Gonzalez et al. "PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs." In: *OSDI*. Vol. 12. 1. 2012, p. 2.

[40] Thomas L Griffiths and Mark Steyvers. "Finding scientific topics". In: *Proceedings of the National academy of Sciences of the United States of America* 101.Suppl 1 (2004), pp. 5228–5235.

[41] Rachid Guerraoui and André Schiper. "Fault-tolerance by replication in distributed systems". In: *Reliable Software Technologies Ada Europe'96*. Springer. 1996, pp. 38–57.

[42] Joseph M Hellerstein et al. "The MADlib analytics library: or MAD skills, the SQL". In: *Proceedings of the VLDB Endowment* 5.12 (2012), pp. 1700–1711.

[43] Zach Hill and Marty Humphrey. "A quantitative analysis of high performance computing with Amazon's EC2 infrastructure: The death of the local cluster?" In: *IEEE Grid Computing*. 2009, pp. 26–33.

[44] Matthew Hoffman, Francis R Bach, and David M Blei. "Online learning for latent dirichlet allocation". In: *advances in neural information processing systems*. 2010, pp. 856–864.

[45] MathWork Inc. "Matlab distributed computing server". In: (). URL: `http://www.mathworks.com/products/distriben`.

[46] Intel. "Intel VTune Amplifier 2015". In: (). URL: `https://software.intel.com/en-us/intel-vtune-amplifier-xe`.

[47] Michael Isard et al. "Dryad: distributed data-parallel programs from sequential building blocks". In: *ACM SIGOPS Operating Systems Review* (2007).

[48] Finn V Jensen. *An introduction to Bayesian networks*. Vol. 210. UCL press London, 1996.

[49] Michael I Jordan et al. "An introduction to variational methods for graphical models". In: *Machine learning* 37.2 (1999), pp. 183–233.

[50] U Kang, Charalampos E Tsourakakis, and Christos Faloutsos. "Pegasus: A peta-scale graph mining system implementation and observations". In: *ICDM*. IEEE. 2009, pp. 229–238.

[51] U Kang et al. *Hadi: Fast diameter estimation and mining in massive graphs with hadoop*. Carnegie Mellon University, School of Computer Science, Machine Learning Department, 2008.

[52] Yehuda Koren, Robert Bell, and Chris Volinksy. "Matrix Factorization Techniques for Recommender Systems". In: *IEEE Computer* 42.8 (2009), pp. 30–37.

[53] Tim Kraska et al. "MLbase: A distributed machine-learning system". In: *Conf. on Innovative Data Systems Research*. 2013.

[54] Marek Kubale. *Graph colorings*. Vol. 352. American Mathematical Soc., 2004.

[55] Sameer Kumar et al. "Optimization of All-to-all communication on the Blue Gene/L supercomputer". In: *Parallel Processing, 2008. ICPP'08. 37th International Conference on*. IEEE. 2008, pp. 320–329.

[56] Haewoon Kwak et al. "What is Twitter, a social network or a news media?" In: *Proceedings of the 19th international conference on World wide web*. ACM. 2010, pp. 591–600.

[57] Mark van der Laan and Sherri Rose. *Targeted Learning: Causal Inference for Observational and Experimental Data*. Springer-Verlag, 2011.

[58] J Langford, L Li, and A Strehl. *Vowpal wabbit online learning project*. 2007.

[59] Leon Bottou Yann Le Cun and L Bottou. "Large scale online learning". In: *Advances in neural information processing systems* 16 (2004), p. 217.

[60] Jure Leskovec et al. "Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters". In: *Internet Mathematics* 6.1 (2009), pp. 29–123.

[61] D.D. Lewis et al. "Rcv1: A new benchmark collection for text categorization research". In: *The Journal of Machine Learning Research* 5 (2004), pp. 361–397.

[62] Yucheng Low et al. "Graphlab: A new framework for parallel machine learning". In: *arXiv preprint arXiv:1006.4990* (2010).

[63] Apache Mahout. "Mahout". In: (). URL: `http://mahout.apache.org/`.

[64] Grzegorz Malewicz et al. "Pregel: a system for large-scale graph processing". In: *Proceedings of the 2010 international conference on Management of data*. ACM. 2010.

[65] Amith R Mamidala, Jiuxing Liu, and Dhabaleswar K Panda. "Efficient Barrier and Allreduce on Infiniband clusters using multicast and adaptive algorithms". In: *Cluster Computing, 2004 IEEE International Conference on*. IEEE. 2004, pp. 135–144.

[66] Kapil K Mathur and S Lennart Johnsson. *All-to-all communication algorithms for distributed BLAS*. Harvard University, Center for Research in Computing Technology, Aiken Computation Laboratory, 1993.

[67] Thomas Minka. "Expectation Propagation for Approximate Bayesian Inference". In: (2001). Ed. by Jack S. Breese and Daphne Koller.

[68] Tom Minka et al. *Infer .NET 2.4, 2010. Microsoft Research Cambridge*.

[69] Feng Niu et al. "Hogwild!: A lock-free approach to parallelizing stochastic gradient descent". In: *arXiv preprint arXiv:1106.5730* (2011).

[70] P. Patarasuk and X. Yuan. "Bandwidth efficient all-reduce operation on tree topologies". In: *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*. IEEE. 2007, pp. 1–8.

[71] Pitch Patarasuk and Xin Yuan. "Bandwidth optimal all-reduce algorithms for clusters of workstations". In: *Journal of Parallel and Distributed Computing* 69.2 (2009), pp. 117–124.

[72] Martyn Plummer et al. "JAGS: A program for analysis of Bayesian graphical models using Gibbs sampling". In: *Proceedings of the 3rd International Workshop on Distributed Statistical Computing (DSC 2003). March*. 2003, pp. 20–22.

[73] Benjamin Recht and Christopher Ré. "Parallel stochastic gradient algorithms for large-scale matrix completion". In: *Optimization Online* (2011).

[74] Yahoo! Academic Relations. *Yahoo! AltaVista Web Graph*. URL: `http://webscope.sandbox.yahoo.com`.

[75] Christian Robert, Arnaud Doucet, and Simon Godsill. "Marginal MAP Estimation using Markov Chain Monte-Carlo". In: *IEEE Int. Conf. on Acoustics, Speech and Signal Processing*. Vol. 3. IEEE, 1999, pp. 1753–1756.

[76] Aamir Shafi, Bryan Carpenter, and Mark Baker. "Nested parallelism for multi-core HPC systems using Java". In: *J. Parallel Distrib. Comput.* 69.6 (2009), pp. 532–545. URL: `http://dx.doi.org/10.1016/j.jpdc.2009.02.006`.

[77] A. Shafi et al. "A comparative study of Java and C performance in two large-scale parallel applications". In: *Concurrency and Computation: Practice and Experience* 21.15 (2009), pp. 1882–1906.

[78] S. Shalev-Shwartz, Y. Singer, and N. Srebro. "Pegasos: Primal estimated sub-gradient solver for svm". In: *Proceedings of the 24th international conference on Machine learning*. ACM. 2007, pp. 807–814.

[79] J.R. Shewchuk. "An Introduction to the Conjugate Gradient Method Without the Agonizing Pain". In: (1994).

[80] Alexander Smola and Shravan Narayanamurthy. "An architecture for parallel topic models". In: *Proceedings of the VLDB Endowment* 3.1-2 (2010), pp. 703–710.

[81] Evan R Sparks et al. "MLI: An API for distributed machine learning". In: *ICDM*. IEEE. 2013, pp. 1187–1192.

[82] Stan Development Team. *Stan Modeling Language Users Guide and Reference Manual, Version 2.4*. 2014. URL: `http://mc-stan.org/`.

[83] Andrew Thomas, David J Spiegelhalter, and WR Gilks. "BUGS: A program to perform Bayesian inference using Gibbs sampling". In: *Bayesian statistics* 4.9 (1992), pp. 837–842.

[84] A. Thusoo et al. "Hive: a warehousing solution over a map-reduce framework". In: *Proceedings of the VLDB Endowment* 2.2 (2009), pp. 1626–1629.

[85] Edward Walker. "Benchmarking Amazon EC2 for high-performance scientific computing". In: *Usenix Login* 33.5 (2008), pp. 18–23.

[86] Greg CG Wei and Martin A Tanner. "A Monte Carlo implementation of the EM algorithm and the poor man's data augmentation algorithms". In: *Journal of the American Statistical Association* 85.411 (1990), pp. 699–704.

[87] Markus Weimer, Tyson Condie, Raghu Ramakrishnan, et al. "Machine learning in ScalOps, a higher order cloud computing language". In: *NIPS 2011 Workshop on parallel and large-scale machine learning (BigLearn)*. Vol. 9. 2011, pp. 389–396.

[88] Yair Weiss. "Correctness of Local Probability Propagation in Graphical Models with Loops". In: *Neural Computation* 12.1 (2000), pp. 1–41.

[89] Samuel Williams, Andrew Waterman, and David Patterson. "Roofline: an insightful visual performance model for multicore architectures". In: *Communications of the ACM* 52.4 (2009), pp. 65–76.

[90] J.M. Winn and C. Bishop. "Variational Message Passing". In: *Journal of Machine Learning Research* 6 (2005), p. 661.

[91] Feng Yan, Ningyi Xu, and Yuan Qi. "Parallel Inference for Latent Dirichlet Allocation on Graphics Processing Units." In: *NIPS*. Vol. 9. 2009, pp. 2134–2142.

[92] Andy Yoo et al. "A scalable distributed parallel breadth-first search algorithm on BlueGene/L". In: *Supercomputing, 2005. Proceedings of the ACM/IEEE SC 2005 Conference*. IEEE. 2005, pp. 25–25.

[93] Y. Yu et al. "DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language". In: *Proceedings of the 8th USENIX conference on Operating systems design and implementation*. USENIX Association. 2008, pp. 1–14.

[94] Matei Zaharia et al. "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing". In: *Proceedings of the 9th USENIX NSDI*. 2011.

[95] Matei Zaharia et al. "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing". In: *NSDI*. USENIX Association. 2012, pp. 2–2.

[96] Huasha Zhao and John Canny. "Butterfly mixing: Accelerating incremental-update algorithms on clusters". In: *SIAM Conf. on Data Mining*. SIAM. 2013.

[97] Huasha Zhao and John Canny. "Kylix: A Sparse Allreduce for Commodity Clusters". In: *Proceedings of the 46th International Conference on Parallel Processing*. 2014.

[98] M. Zinkevich et al. "Parallelized stochastic gradient descent". In: *Advances in Neural Information Processing Systems* 23.23 (2010), pp. 1–9.