Big Data Analytics with Small Footprint: Squaring the Cloud

John Canny Huasha Zhao

University of California
Berkeley, CA 94720
jfc@cs.berkeley.edu, hzhao@cs.berkeley.edu

ABSTRACT

This paper describes the BID Data Suite, a collection of hardware, software and design patterns that enable fast, large-scale data mining at very low cost. By co-designing all of these elements we achieve single-machine performance levels that equal or exceed reported *cluster* implementations for common benchmark problems. A key design criterion is *rapid exploration* of models, hence the system is interactive and primarily single-user. The elements of the suite are: (i) the data engine, a hardware design pattern that balances storage, CPU and GPU acceleration for typical data mining workloads, (ii) BIDMat, an interactive matrix library that integrates CPU and GPU acceleration and novel computational kernels (iii), BIDMach, a machine learning system that includes very efficient model optimizers, (iv) Butterfly mixing, a communication strategy that hides the latency of frequent model updates needed by fast optimizers and (v) Design patterns to improve performance of iterative update algorithms. We present several benchmark problems to show how the above elements combine to yield multiple orders-of-magnitude improvements for each problem.

Categories and Subject Descriptors

G.4 [**Mathematical Software**]: Algorithm design and analysis

General Terms

Algorithms, Design, Experimentation, Performance

Keywords

Data Mining; Toolkit; Machine Learning; Cluster; GPU

1. INTRODUCTION

The motive for the BID Data Suite is *exploratory data analysis*. Exploratory analysis involves sifting through data, making hypotheses about structure and rapidly testing them. The faster this exploration, the higher the performance of the final model in a typically time-limited process. Furthermore, real-world data analysis problems are not "pure" and the best solution is usually a mixture of many simpler models. For instance all of the highest-performing Netflix® contest competitors were complex hybrids [2]. 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.

1.1 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 megalibraries, shopping (Amazon®, Ebay®), tagging (Flickr®, Digg®) repositories (Wikipedia, Stack Overflow), MOOC data, server logs and recommenders (Netflix®, Amazon® etc). These datasets have many uses in health care, government, education, commerce, and cover the larger part of the commercial applications of data mining. We exclude data in specialized formats like images and videos, and high-density sensor data. The later require specialized processing and can be 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. Figure 1 gives a feel for the scale of these datasets:

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

KDD'13, August 11–14, 2013, Chicago, Illinois, USA.

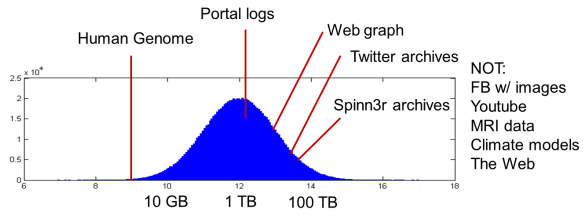


Figure 1: The scales of behavioral data

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.

1.2 Data Mining Workloads

Most large-scale data mining workloads are either I/O-bound or compute-bound. In fact they are typically 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 improves with complexity. Complexity is tuned as high as possible - and is often set by machine performance limits. Each datum is used many times - often thousands - and the overall time spent processing a datum strongly dominates the time to retrieve it.

1.3 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. SGD and GS are now preferred tools for a 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 [19] 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.

1.4 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. "imp" denotes improvement of full GPU acceleration for this machine over Java builtins.

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 data mining tasks.

Note the large speedups from GPU calculation (10^2-10^3) for all tasks except SPMV. Note also that (multithreaded) 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.

SPMV is the bottleneck for regression, SVM and other typical "light touch" 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 may be lost. On the other hand 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.

Even if the goal is to compute one regression model, we can use the available cycles to speed up learning in a variety of ways. The first is parameter exploration: it is known that SGD algorithm convergence is quite sensitive to the gradient scaling factor, and this factor changes with time. Running many models in parallel with different gradient scaling allows the best value to be chosen at each update. 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 [4].

2. THE SUITE

2.1 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. There has been much exploration recently of memory-based cluster frameworks [18], which inherit these performance levels. On the other hand, disk storage is nearly two orders of magnitude lower in power and cost per GBbyte, and GPUs about one order of magnitude lower in power/cost per Gflop. Its therefore natural to explore architectures that shift the I/O and compute loads to disk and GPU. A *data engine* is a commodity workstation (PC) which is optimized for a mix of I/O bound and compute-bound operations.

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 prototype 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 well 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 design achieves order-of-magnitude improvements in both cost and power per Gbyte of storage and per Gflop of computation compared to conventional compute server instances. 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 can leverage the software tools described below.

2.2 BIDMat: A Fast Matrix Toolkit

Our discussion suggests 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 and even unicode math characters: (e.g. `+@, *!, *|`, $\bullet$, $\circ$, $\otimes$, $\vee$, $\wedge$, ...)

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) use of complementary toolkits for tasks such as natural language processing (iii) File I/O using HDF5 format and the hdf5-java library or hdfs via hadoop, (iv) communication using the JMPI library and (v) 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:

2.2.1 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. 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.

2.3 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 (in progress) 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 which are built on many independent models which are trained in parallel. Advanced ensemble methods such as super-learning [16] use several of these techniques at once.

2.4 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 pattern that has recurred in many problems is *loop interleaving*, illustrated schematically in figure 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.

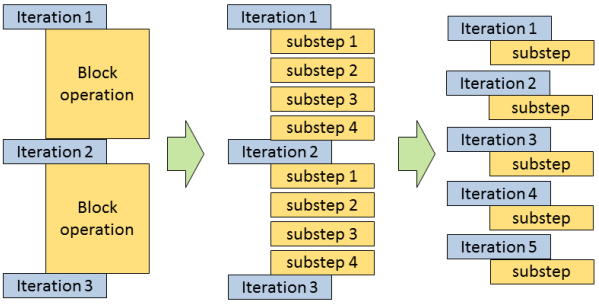


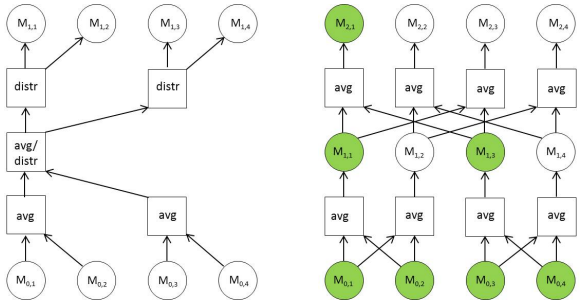
Figure 2: The loop interleaving pattern

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.

2.5 Butterfly Mixing

Most cluster frameworks used for data mining do not use CPU or GPU acceleration. When CPU or GPU acceleration *is* applied to one of these systems, there is often little or no performance improvement for SGD or MCMC or other frequent-update algorithms. The network primitives used by these systems completely dominate the running time. To realize the potential benefits of local acceleration in a cluster we had to develop faster primitives. Butterfly mixing is the first of these. It is also the first instantiation of the loop interleaving pattern. In Figure **??**, the left graphic 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 in butterfly mixing 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 figure 3b.

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. This arrangement is show layered on the network in Figure 3b.



(a) AllReduce (b) Butterfly Mixing

Figure 3: Butterfly mixing interleaves communication with computation

Its clear that this approach dramatically reduces communication overhead compared to allreduce every step in Figure 3a, and many more gradient updates are being done in a given amount of time. Its less clear what impact this has on convergence. In [19] 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 can 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.

3. BENCHMARKS

3.1 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:

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

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 on 64 nodes, and much faster than a Hadoop cluster implementation as seen in the table below.

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.

3.2 LDA and GaP

Latent Dirichlet Allocation [3] 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 [6]. 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 [3]). Following the notation of [3], 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:

$$F_{ij} = \exp(\Psi(\gamma_{ij})) \tag{2}$$

The update formula from figure 6 of [3] can now be written:

$$\gamma_{ij} = \alpha_i + \sum_{w=1}^{M} \beta_{iw} F_{ji} C_{wj} / \sum_{i=1}^{k} \beta_{iw} F_{ij} \tag{3}$$

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:

$$\gamma' = \alpha + F \circ \left(\beta * \frac{C}{\beta^T *_C F} \right) \tag{4}$$

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 [3]) which can be expressed in matrix form and has the same complexity.

The GaP algorithm has a similar E-step. Using the notation from [6], 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:

$$\gamma' = \left(a - 1 + \gamma \circ \left(\beta * \frac{C}{\beta^T *_C \gamma} \right) \right) / \left(1 + \frac{1}{b} \right) \tag{5}$$

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. 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. [15] and 30-word docs for PowerGraph [8]. 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[15]	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.

3.3 ALS and SFA

ALS or Alternating Least Squares [10] 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) [5] 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

$$l = \sum (C - \beta^T *_C \ \gamma)^2 + \sum w_\beta \circ \beta^2 + \sum w_\gamma \circ \gamma^2 \quad (6)$$

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

$$\frac{dl}{d\gamma} = 2\beta * (C - \beta^T *_C \ \gamma) + 2w_\gamma \circ \gamma \quad (7)$$

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 usually nsamples (users) >> nfeatures (movies) and the feature array β can fit in memory, we process the data array C in blocks of samples.

3.3.1 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:

$$\gamma_i = \left(\lambda n_i I + \beta \ \mathrm{diag}(C_i)\beta^T\right)^{-1} \beta C_i \quad (8)$$

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.

3.3.2 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:

$$\left(\lambda n_i I + \beta \ \mathrm{diag}(C_i)\beta^T\right)\gamma_i = M_i\gamma_i = \beta C_i \quad (9)$$

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:

$$v = \lambda w_\gamma \circ \hat{\gamma} + \beta * (\beta^T *_C \ \hat{\gamma}) \quad (10)$$

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:

$$w = \lambda w_\beta \circ \hat{\beta} + \gamma \ *^T \ (\hat{\beta}^T *_C \ \gamma) \quad (11)$$

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.

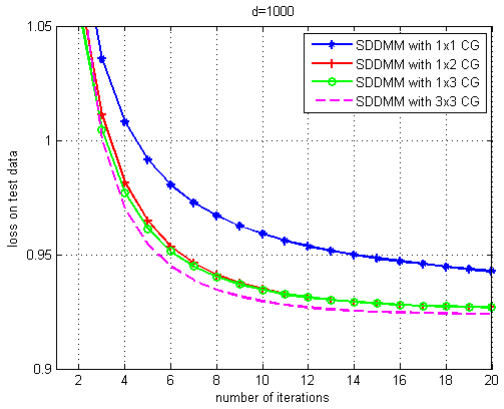


Figure 4: Loss vs. num iterations vs. CG updates

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 is local to each user's data and so multiple steps can be done on each block of data read from disk. For equation 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 4 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.

The table below shows the normalized performance of the CG method and a prior method on a testset described in [8] (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 [8] 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

3.4 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 $O(n^2f + n^2k)$ per iteration. The $O(n^2f)$ 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:

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

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. With this addition the new method achieves more than 1000-fold improvement overall compared to the previous Matlab/C or Fortran implementations.

3.5 Regression and SVM

We briefly summarize the results from the paper [19] here. We tested the scaling improvements with butterfly mixing with two algorithms and two datasets. The algorithms were the Pegasos SVM algorithm [14], 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.

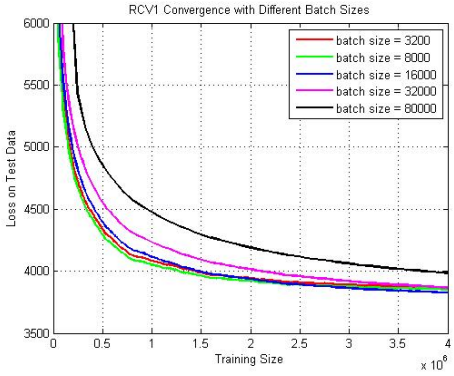


Figure 5: Impact of Minibatch Size on Convergence

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 6.

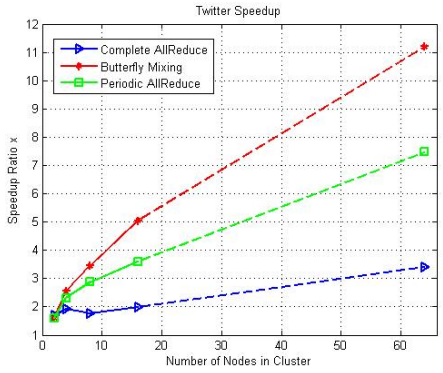


Figure 6: Speedup Relative to Single Processor

In figure 6, 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.

4. SQUARING THE CLOUD

Parallelizing machine learning across a cluster is one approach to improve speed and extend problem scale. But

it does so 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"

5. RELATED WORK

Many other big data toolkits are under active development [8], [12], [18], [7], [17], [9], [13], [11] [1]. Our system is perhaps closest to GraphLab/PowerGraph [12, 8] 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 [13] 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 substantial performance/cost and performance/energy advantages over other systems at this time.

6. CONCLUSIONS AND FUTURE WORK

We made a case for the importance of single-node enhancements for cost and power efficiency in large-scale behavioral data analysis. Single node performance of the BID Data suite on several common benchmarks is faster than generic cluster systems (Hadoop, Spark) with 50-100 nodes and competitive with custom cluster implementations for specific algorithms. We did not describe any MCMC algorithms, but these also are an important part of the BID Data roadmap in future. The toolkit will also include a substantial subset of tools for causal analysis, and disk-scale GPU-assisted sorting for data indexing and search tasks.

7. REFERENCES

[1] Alexander Behm, Vinayak R Borkar, Michael J Carey, Raman Grover, Chen Li, Nicola Onose, Rares Vernica, Alin Deutsch, Yannis Papakonstantinou, and Vassilis J Tsotras. Asterix: towards a scalable, semistructured data platform for evolving-world models. *Distributed and Parallel Databases*, 29(3):185–216, 2011.

[2] R. Bell and Y. Koren. Scalable collaborative filtering with jointly derived neighborhood interpolation weights.

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

[4] Leon Bottou. Large-scale machine learning with stochastic gradient descent. In *Proc. COMPSTAT 2010*, pages 177–187, 2010.

[5] John Canny. Collaborative filtering with privacy via factor analysis. In *ACM SIGIR 2002*, 2002.

[6] John Canny. GAP: a factor model for discrete data. In *ACM SIGIR*, pages 122–129, 2004.

[7] Jaliya Ekanayake, Hui Li, Bingjing Zhang, Thilina Gunarathne, Seung-Hee Bae, Judy Qiu, and Geoffrey Fox. Twister: a runtime for iterative mapreduce. In *Proc. 19th ACM HPDC*, pages 810–818, 2010.

[8] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *OSDI*, 2012.

[9] Joseph M Hellerstein, Christoper Ré, Florian Schoppmann, Daisy Zhe Wang, Eugene Fratkin, Aleksander Gorajek, Kee Siong Ng, Caleb Welton, Xixuan Feng, Kun Li, et al. The madlib analytics library: or mad skills, the sql. *Proceedings of the VLDB Endowment*, 5(12):1700–1711, 2012.

[10] Yehuda Koren, Robert Bell, and Chris Volinksy. Matrix factorization techniques for recommender systems. *IEEE Computer*, 42(8):30–37, 2009.

[11] Tim Kraska, Ameet Talwalkar, John Duchi, Rean Griffith, Michael J Franklin, and Michael Jordan. Mlbase: A distributed machine-learning system. In *Conf. on Innovative Data Systems Research*, 2013.

[12] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M Hellerstein. Graphlab: A new framework for parallel machine learning. *arXiv preprint arXiv:1006.4990*, 2010.

[13] Benjamin Recht and Christopher Ré. Parallel stochastic gradient algorithms for large-scale matrix completion. *Optimization Online*, 2011.

[14] Shai Shalev-Shwartz, Yoram Singer, and Nathan Srebro. Pegasos: Primal estimated sub-gradient solver for SVM. In *Proc. 24th ICML*, pages 807–814, 2007.

[15] Alexander Smola and Shravan Narayanamurthy. An architecture for parallel topic models. *Proceedings of the VLDB Endowment*, 3(1-2):703–710, 2010.

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

[17] 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)*, volume 9, pages 389–396, 2011.

[18] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX NSDI*, 2011.

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