Scalable I/O-Bound Parallel Incremental Gradient Descent for Big Data Analytics in GLADE

Chengie Qin
UC Merced
5200 N Lake Road
Merced, CA 95343
cqin3@ucmerced.edu

Florin Rusu
UC Merced
5200 N Lake Road
Merced, CA 95343
frusu@ucmerced.edu

ABSTRACT

Incremental gradient descent is a general technique to solve a large class of convex optimization problems arising in many machine learning tasks. GLADE is a parallel infrastructure for big data analytics providing a generic task specification interface. In this paper, we present a scalable and efficient parallel solution for incremental gradient descent in GLADE. We provide empirical evidence that our solution is limited only by the physical hardware characteristics, uses effectively the available resources, and achieves maximum scalability. When deployed in the cloud, our solution has the potential to dramatically reduce the cost of complex analytics over massive datasets.

Categories and Subject Descriptors

H.2 [**Database Management**]: Miscellaneous

1. INTRODUCTION

In the age of big data, businesses compete in extracting the most information out of the immense amount of data they acquire. Since more information translates almost directly into better decisions that provide a much sought-after competitive edge, big data analytics tools promising to deliver this additional bit of information are highly-valued. There are two major issues that have to be addressed by any such tool. First, they have to cope with massive amounts of data. While scalability is the most cited metric in this case, we believe that efficiency is as important, especially in the cloud where billing is based on wall-clock time rather than resource utilization. The ultimate metric should always be the physical hardware limitations. In the case of big data, this almost always corresponds to the limited disk I/O bandwidth. Second, the tools have to be general and extensible. They have to provide a large spectrum of data analysis methods ranging from simple descriptive statistics to complex predictive models. Moreover, the tools should be easily extensible with new methods without major code development.

GLADE [13] is an architecture-independent parallel infrastructure for generic aggregation that extends the User-Defined Aggregation (UDA) mechanism to complex analytics. The generality, scalability and, most importantly, efficiency of this approach was proven on a series of big data analytics tasks ranging from simple to high-cardinality group-by aggregation, k-means clustering, and top-k. GLADE hits the physical bounds imposed by the limited I/O throughput on each of these tasks while it scales linearly with the number of processing nodes for terabyte-size datasets.

Incremental gradient descent is a general technique to solve a large class of analytical models expressed as convex optimization problems, e.g., logistic regression (LR), support vector machines (SVM), low-rank matrix factorization (LMF), and conditional random fields (CRF). Feng et al. [7] show that IGD has a data access pattern identical to the UDA access pattern. Moreover, they provide a UDA-based implementation for all the machine learning models mentioned above. This implementation is targeted at multicore shared-memory architectures which provide limited scalability with the size of the data, as reflected by the corresponding experimental results.

Contributions. In this paper, we present a scalable and efficient parallel IGD implementation in GLADE starting from the centralized shared-memory UDA-based solution proposed in [7]. Our specific contributions are:

- We investigate the modifications required in order to implement IGD in the parallel environment specific to GLADE. We identify passing the model state across iterations, randomizing the input across iterations, and merging the model state across nodes as key factors for an effective implementation. We propose solutions for each, discuss alternatives, and show how they are implemented in GLADE.
- We perform an extensive evaluation of our solution for four different machine learning models – LR, SVM, LMF, and CRF – over truly massive datasets containing up to ten billion examples. The results confirm that GLADE achieves maximum scalability and remains I/O-bound even for these CPU-intensive tasks. Concretely, our implementation converges to the optimal solution over a ten billion dataset in 80 seconds while other analytics tools cannot even handle such a large dataset, less find the optimal solution [7].
- We port the UDA-based IGD implementation given in [7] to GLADE with modifications limited to converting between different interface function names and different internal tuple representations. This confirms GLADE as an extensible general-purpose platform for complex analytics.

The implications our solution has on big data analytics in the cloud are dramatic. We can imagine executing complex analytic tasks over massive datasets at virtually no cost or, equivalently, we are able to execute considerably more tasks for the same price while using the available computational resources to the maximum.

Related work. The work most related to this paper – and our source of inspiration – is Bismarck [7]. The entire idea of unifying

DanaC'13, June 23, 2013, New York, NY, USA.

the solution to a series of convex optimization problems under the IGD umbrella and the in-RDBMS implementation using the UDA mechanism are introduced there. Since GLADE [13, 4] is designed entirely around the UDA abstraction, it was appealing to us to investigate a GLADE solution to IGD. There are considerable differences between our solution and Bismarck though. We focus on a purely parallel IGD implementation not a multi-threaded shared-memory version. The chunk-based extended UDA interface supported in GLADE is considerably richer than the standard UDAs used in Bismarck. Our solution supports both thread- and process-level parallelism. At last, due to the remarkable GLADE performance, we are able to perform experiments on considerably larger datasets containing tens of billions of tuples and gain novel insights on how IGD performs at scale. This was not possible in Bismarck or any other system we are aware of. MADlib [9] is a more general in-RDBMS analytics library that includes IGD and many other statistical methods. Since they are mostly based on the UDA abstraction, they can also be readily implemented in GLADE. HaLoop [3] and Spark [14] present caching mechanisms for iterative processing in Hadoop. Since job scheduling is static in GLADE, these mechanisms do not apply.

Due to its generality and good convergence behavior, IGD received considerable attention in the machine learning optimization literature. As a result, multiple methods to parallelize IGD have been proposed recently. Some of them are multi-core thread-level parallel schemes [11, 10] with or without locking while others are purely parallel solutions [15, 6, 8]. Our goal is not to propose a new parallel IGD method but rather to investigate what methods are scalable to massive data and allow for effective implementation.

2. INCREMENTAL GRADIENT DESCENT

Consider the following optimization problem with a linearly separable objective function (we adopt the notation in [7]):

$$\min_{w \in \mathbb{R}^d} \sum_{i=1}^{N} f\left(w, z_i\right) + P(w) \tag{1}$$

in which a d-dimensional vector $w \in \mathbb{R}^d$, $d \geq 1$ has to be found such that the objective function is minimized. The constants z_i, $1 \leq i \leq N$ correspond to tuples in a database table—materialized or obtained as the result of a query. Essentially, each term in the objective function corresponding to a tuple z_i can be viewed as a separate function $f_i(w) = f\left(w, z_i\right)$.

Gradient descent is an iterative method to solve (1). The main idea is to start from an arbitrary vector $w^{(0)}$ and then to determine iteratively new vectors $w^{(k+1)}$ such that the objective function at each iteration decreases, i.e., $f(w^{(k+1)}) > f(w^{(k)})$ (we consider $P\left(w\right) = 0$ for simplicity).

Multiple details have to be clarified for the method to work. First, how to determine $w^{(k+1)}$? The answer is simple: move in the opposite direction to the gradient or subgradient – if the gradient does not exist – of function f. Formally, this can be written as:

$$w^{(k+1)} = w^{(k)} - \alpha_k \nabla f_{\eta(k)} \left(w^{(k)}\right) \tag{2}$$

where $\alpha_k \geq 0$ is the step size and $\nabla f_{\eta(k)}(w)$ is the approximation to the gradient $\nabla f(w)$ based on a single term $f_{\eta(k)}(w)$ at iteration k, respectively. Taking steps based on the tuple-based approximation instead of the actual gradient is the characteristic property of IGD. Typically, $\alpha_k \to 0$ as $k \to \infty$ and $\eta(k) \in \{1, \ldots, N\}$ represents a random permutation, i.e., all f_i have to be considered before any term is repeated.

Second, how can we determine when the method converges? And how do we know that the method converges to a global minimum of the objective function? The standard method to check for convergence is to compare the objective function for $w^{(k+1)}$ and $w^{(k)}$ obtained in consecutive iterations. If the difference is smaller than a given threshold, convergence can be assumed. An alternative is to fix the number of iterations ahead of time and take the last w as the optimal solution. Convergence to a global optimal solution is theoretically guaranteed when both functions $\sum_{i=1}^{N} f_i(w)$ and $P(w)$ are convex—the case we consider in this paper.

3. GLADE

GLADE is a parallel data processing system executing any computation specified as a Generalized Linear Aggregate (GLA) [4] using a merge-oriented strategy supported by a push-based storage manager that drives the execution. Essentially, GLADE provides an infrastructure abstraction for parallel processing that decouples the algorithm from the runtime execution. The algorithm has to be specified in terms of a clean interface, while the runtime takes care of all the execution details including data management, memory management, and scheduling. Contrary to existent parallel data processing systems designed for a target architecture, typically shared-nothing, GLADE is architecture-independent. GLADE hits the physical limitations both on shared-disk servers as well as on shared-nothing clusters—see [13] for experimental evidence. The reason for this is the exclusive use of thread-level parallelism inside a processing node while process-level parallelism is used only across nodes. There is no difference between these two in the GLADE infrastructure abstraction.

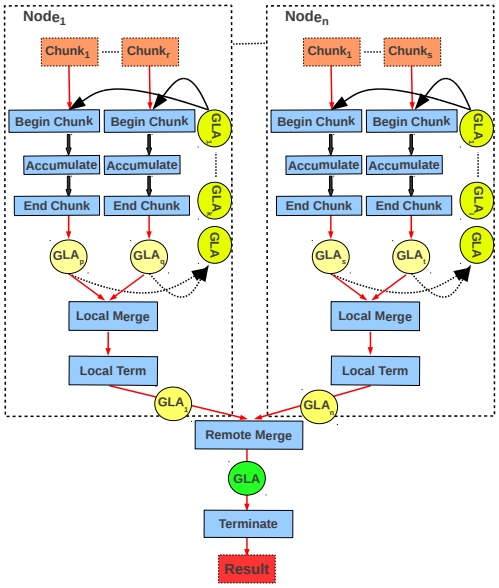

Figure 1: GLADE architecture for chunk-at-a-time parallel processing using a merge-oriented execution strategy.

GLADE consists of two types of entities: a coordinator and one or more executor processes. The coordinator is the interface between the user and the system. Since it does not manage any data except the catalog metadata, the coordinator does not execute any data processing task. These are the responsibility of the executors, typically one for each physical processing node. It is important to notice that the executors act as completely independent entities, in charge of their data and of the physical resources. Each executor runs an instance of the DataPath [1] execution engine enhanced

with a GLA metaoperator for the execution of arbitrary user code specified using the GLA interface.

Figure 1 depicts the stages of the GLADE execution strategy expressed in terms of the GLA interface abstraction [13]. The GLA interface extends the common UDAs implemented in every major RDBMS [7] to parallel chunk-at-a-time or vectorized execution—the processing strategy in GLADE. A chunk contains a set of tuples. It is the basic I/O and processing unit in GLADE. Intuitively, GLAs are objects corresponding to the state of the aggregate upon which the methods in the GLA interface are invoked following a well-defined pattern. The semantic of the UDA methods is standard and is presented elsewhere [5]. `BeginChunk` and `EndChunk` provide access to the entire chunk – the unit of processing – before and after the GLA state is updated. An important usage scenario for these methods in our case is to reorder the tuples in the chunk such that different orders are used across iterations. `Merge` and `Terminate` are split into local (thread-level) and remote (process-level) instances to make the distinction between node- and cluster-level processing clear. This allows for different model merging strategies to be applied inside the node and between nodes.

4. IGD IN GLADE

In this paper, we present a parallel IGD GLADE implementation. In this implementation, a partial model – the vector w – is computed independently for each data partition. This can be a chunk, a series of chunks, or the entire data on a given processing node. The optimal model for an iteration is obtained by merging the partial models together. The reason we focus on the pure UDA implementation is because it is the only implementation that scales to any data volumes and any number of nodes. And scalability is one of the main requirements in cloud computing.

IGD as a GLA. Two GLAs are required to implement IGD in GLADE—one for computing the optimal model (`IGD GLA`) at each iteration and one for computing the objective function (`Loss GLA`). All the action happens in `IGD GLA` – the `Accumulate` method to be precise – where Eq. (2) is implemented. Since the only difference across models is confined to the form of the gradient function $\nabla f(w)$, this is the only place where the code is different for various models. `Loss GLA` can be invoked after each iteration to test for convergence based on the relative drop in the objective function or it can be called only at the end, after a fixed number of iterations, to get the optimal value.

The state of `IGD GLA` contains the model w. GLADE performs optimally when the GLA state fits in memory but that is not a requirement. In `Init` – the constructor in our case – $w^{(k)}$ is initialized either with the original starting point $w^{(0)}$ for the first iteration or with the model computed at the previous iteration $w^{(k-1)}$. `Accumulate` is called with a GLA and a chunk as parameters. For each tuple in the chunk, the approximation to the gradient $\nabla f(w)$ is computed and the model is updated accordingly based on the formula in Eq. (2). Since multiple chunks are processed in parallel – either in different threads or on different processing nodes – `Merge` is called after all the chunks are processed to combine together the partial models in the resulting GLAs. The two versions of `Merge` – local and remote – allow for different merging strategies. Once the complete model is computed, it is post-processed for the subsequent iteration in `Terminate`.

Similar to the pure UDA solution in [7], `IGD GLA` assumes that IGD is commutative and algebraic even though that is not the case. These requirements are needed in order to parallelize the IGD computation. The direct effect of lack of commutativity is the slower convergence rate of the IGD method. This is alleviated to some extent by randomizing the data across iterations. The requirement for IGD to be algebraic is imposed by the need to merge partial models computed on different data partitions. The standard method proposed in the machine learning literature and leveraged GLADE is to average the partial models [15].

Although the GLA interface and execution mechanism provide a high-level abstraction for the parallel IGD implementation, there are a series of challenges that require special consideration in order to devise an effective and efficient implementation. We identify iteration management, data randomization, and model merging as the most important challenges for the purely parallel GLADE IGD implementation. In the following, we present the solution implemented in GLADE for each of them and discuss other alternatives.

Iteration management. Independent of the termination criterion – `Loss GLA` is invoked after every iteration or only at the end – the optimal model computed in one iteration has to be passed as the initial point for the subsequent iteration, i.e., $w^{(k)}$ in Eq. 2. In a distributed environment, this requires broadcasting the model $w^{(k)}$ from the coordinator to all the processing nodes. There are two solutions on how this can be done in GLADE. In the first solution, the model is sent explicitly as part of the job configuration message. In the second, each processing node can directly access the model stored in a common underlying network file system—NFS in GLADE. Data transfer is masked by the NFS operations in this case. The solution can be chosen independently for every task.

Data randomization. The order in which tuples are used to update the model in Eq. (2) determines the IGD convergence rate. Typically, random orders not correlated with the tuple values provide better convergence. Moreover, different random orders across iterations enhance the convergence rate even further. The problem is that randomization is expensive even when executed only once and more so in a shared-nothing environment. For example, in Bismarck [7] data are randomized once and only partially due to the unacceptable increase in execution time randomization incurs. This is not the case in GLADE though where multiple levels of randomization are embedded in the execution strategy.

In GLADE, randomization across processing nodes is realized at data loading. It is a one time process that randomly partitions data across nodes. If executed only as a specific type of loading, e.g., random hash partitioning, it is no different from data partitioning in parallel databases. If executed for each iteration, all-to-all communication between processing nodes is required every time—a time-consuming process that is not guaranteed to result in a significant improvement of the convergence rate. We point the reader interested in specific algorithms for parallel data randomization to [12].

The order in which the data partition at each node is traversed is very likely different from one iteration to another due to the intrinsic execution mechanism. Even more, the order is also quasi-random and impossible to determine prior to runtime. There are two reasons for this phenomenon. First, the DataPath execution engine [1] at each node is push-based. Chunks are continuously read from disk and pushed into processing. If no resources are available – no spare threads in the thread pool – the chunk is dropped and it is regenerated at a later time and in a different order. Due to the CPU-intensive processing required by some IGD models, chunk dropping is quite frequent. As a result, the order in which chunks are processed is non-deterministic even for one iteration, not to mention across iterations. The second reason for the implicit randomness in GLADE processing is the dynamic pairing between chunks and GLAs at runtime. The same GLA is updated multiple times with data from different chunks—the GLA is reused across chunks inside the same task. Randomness results from the non-deterministic assignment of chunks to GLAs which depends on the order in which chunks are processed and the relative chunk process-

ing speed. It is practically impossible to determine which chunks end-up updating the same GLA.

In addition to the implicit randomization mechanisms that are characteristic to GLADE processing and whose behavior is dictated entirely by the runtime context specific to each particular execution, we propose two explicit randomization mechanisms. The first mechanism enforces a random order in how chunks are read from disk—the standard procedure reads chunks sequentially. The random order is determined for each task in part during query admission. While this guarantees different orders across iterations, it also affects the I/O throughput, thus the query execution time. The second mechanism addresses the randomization inside a chunk. Instead of processing tuples in the order in which they are stored in the chunk, tuples are first randomly permuted in `BeginChunk` and only then passed to `Accumulate`. In `EndChunk`, the original order can be restored. The extended GLA interface in Figure 1 supports both these randomization mechanisms seamlessly.

It is important to emphasize that the effect of the proposed explicit mechanisms is hard to quantify in normal GLADE processing due to the interaction with the implicit mechanisms. Specifically, even though a random order to read the chunks from disk is generated, it is not necessary that it is also followed since chunks are dropped and the actual processing order might change. It is also very likely that the implicit mechanisms provide the necessary randomization and the additional mechanisms do not add substantial benefits to compensate for the increase in execution time. The important message is that in GLADE randomization is embedded in the execution strategy and it does not require any special consideration in the normal situation. The experimental results in Section 5 validate this claim.

Model merging. Averaging is the standard method to merge models in parallel IGD computation [15]. The components of w from each GLA are pair-wise averaged, possibly weighted on the number of examples processed by the GLA, in order to obtain the optimal w vector. An alternative is to average the gradients [6] instead of the models and then to compute the model w based on the resulting gradient. These and other merging schemes based on different statistical functions of the sample models and gradients are easily supported in GLADE. `LocalMerge` and `RemoteMerge` allow for different merging strategies to be applied for the GLAs computed at a node and on different nodes, respectively. This is an important distinction that provides an additional level of flexibility and allows for hybrid merging strategies. The number of GLAs computed in GLADE is a variable parameter rather than being determined by the number of data segments. It can vary between one and the number of threads at each processing node, making merging a highly customizable procedure.

5. BENCHMARK RESULTS

We evaluate the efficiency and scalability of the GLADE IGD implementation on massive datasets. According to the qualitative results published in previous work [7], only Bismarck is capable to handle such massive datasets for the tasks we consider. No quantitative evidence, i.e., running time, is provided whatsoever. Since we have already shown [4] that GLADE is two orders of magnitude – a factor of 324 to be precise – faster than Hadoop on similar tasks, we do not consider Hadoop in this study. Thus, we take as absolute reference the physical limitations of the experimental system, i.e., CPU usage, I/O bandwidth, and network bandwidth. Specifically, the experiments we conducted are meant to answer the following questions:

- What is the effect of multi-threading on the parallel IGD implementation? We show that for CPU-intensive models, e.g., LMF, multi-threading has a major impact in GLADE. Essentially, a large number of threads transform the CPU bottleneck into the I/O bottleneck specific to database processing.
- What is the scalability of IGD? GLADE achieves the maximum possible speedup with the number of processing nodes limited only by physical limitations.
- What is the effect of randomization on convergence rate? We show that the default GLADE execution mechanism with implicit randomization is sufficient to achieve the same convergence rate as the explicit randomization mechanisms.
- What is the effect of parallelism on convergence rate? While parallel processing reduces the overall execution time, we show that it also affects negatively the IGD convergence on a per iteration. Nonetheless, the overall effect is positive since the same convergence rate is achieved in shorter time.

Setup. In our experiments, we used an inexpensive $30,000 9-node cluster running Ubuntu SMP 11.04 with Linux kernel 2.6.38-14. One node is configured as the GLADE coordinator while the other eight are workers. Each worker node has 16 cores, 16GB of RAM, and 4 1TB disks. Striping, i.e., RAID-0, is implemented directly in GLADE, not configured in the OS. Notice that only the workers are executing data processing tasks. All the tasks read data from disks with cold caches. We execute ten iterations – optimal model computation and loss calculation – for every task. We report the average execution time per iteration for the model computation phase. Log likelihood is used as the loss function.

Tasks and datasets. We use four machine learning models in our evaluation—LR, SVM, LMF, and CRF. We run experiments over three massive datasets—two synthetic (`classify300M` and `matrix10B`) and one real (`DBLP`)[1]. While a detailed description of the datasets is given in [7], we point out that `matrix10B` is a sparse matrix with 1 million rows and 1 million columns containing 10 billion non-empty cells (the total size is 200GB). The size of the chunk is set to 2^{20} tuples. When running on more than one worker node, data are round-robin partitioned. Due to space limitation, we are not able to include all the experimental results.

Multi-threaded execution. Figure 2a depicts the effect multi-threading has on IGD for the LMF task over the `matrix10B` dataset when executed on a single node. As expected, the execution time per iteration decreases as the number of threads increases. Beyond 8 threads though, the I/O bandwidth becomes the bottleneck and increasing the number of threads further has no effect. The *Ideal CPU* curve corresponds to the execution time required by CPU processing only—this is how the execution time behaves if we ignore the disk I/O. The difference between the real and ideal CPU time for small number of threads – when the processing is CPU-bound – is caused by the scheduling overhead incurred by the GLADE push execution strategy.

Scalability. Figure 2b depicts speedup per iteration as a function of the number of worker nodes starting from the most efficient single-node configuration in Figure 2a. The 10 billion tuples are equally divided between the processing nodes in round-robin fashion. Linear speedup is achieved for up to 4 nodes. The main reason speedup is not linear for the 8-node configuration is that some of the nodes used in this configuration have a significantly lower I/O throughput – 100MB/s vs 120MB/s – than the nodes used in the smaller configurations. Nonetheless, an execution time of 80s per iteration is remarkable for a 10 billion dataset.

Data randomization. The effect of the implicit and explicit GLADE randomization mechanisms on the convergence is depicted in Figure 2c. We use the LR model over the `classify300M`

[1]We thank the Bismarck [7] authors for providing us the datasets.

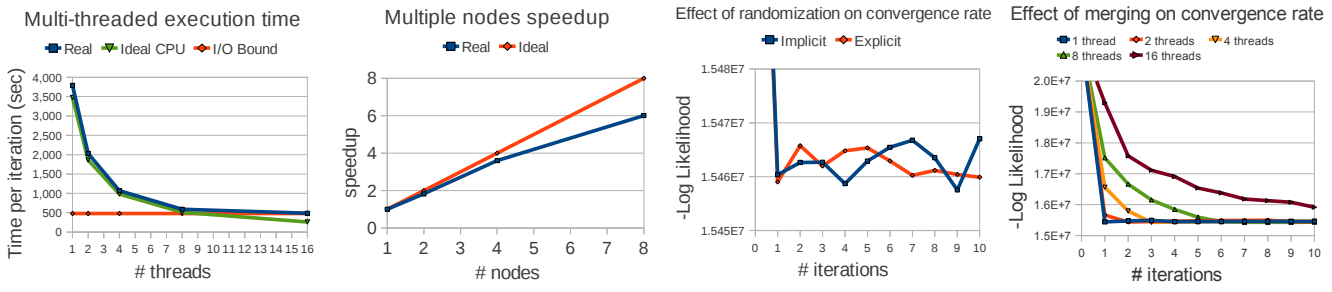

Figure 2: (a) depicts the execution time as a function of the number of threads for the LMF model executed over the `matrix10B` dataset which contains 10 billion examples (200GB). (b) depicts the speedup as a function of the number of processing nodes for `matrix10B`. (c) depicts the effect of randomization on convergence while (d) depicts the effect of merging on convergence for the LR model over the `classify300M` dataset.

dataset in this and the model merging experiment because, due to the large number of examples, LMF converges in one iteration and there is literally no change in the loss function over the subsequent iterations. In order to isolate the effect of randomization, single-thread single-GLA execution is used in this scenario. From Figure 2c we observe that the explicit randomization mechanisms do not bring significant improvement on top of the implicit ones implemented by default in GLADE—the two curves are almost identical. The reason for the increase in the objective function after the first iteration is the unstable behavior of IGD in the neighborhood of the optimal solution [2]. Essentially, the model converges after a single iteration and then oscillates around the optimal solution. The benefit of explicit randomization is that it reduces the amplitude of the oscillations.

Model merging. Parallelization does not come for free. While it reduces the execution time per IGD iteration, it has a negative effect on the convergence rate as depicted in Figure 2d. We observe that as we increase the number of threads, a larger number of iterations is required to achieve convergence. The explanation for this phenomenon is that the smaller number of steps taken by each IGD GLA cannot be compensated by merging a larger number of models. In terms of wall-clock time though, parallelization might still be beneficial since although it takes more iterations to converge to the same objective value the overall time is still shorter than the time taken by fewer longer iterations. For example, in Figure 2c it takes 6 iterations to converge for 8 threads. This is faster than one thread since with 8 threads we can execute 8 iterations in the same time we execute one iteration with a single thread.

Discussion. We have shown that it is possible to optimize a complex LMF model over a 10 billion tuple dataset in 80 seconds. This is a remarkable result for two reasons. First, we achieve this performance on an inexpensive $30,000 9-node cluster. And, most importantly, we are limited only by the I/O throughput of the machine. The execution time on a single node – 480 seconds – is as remarkable. Only a single other system is capable of executing a similar task [7]. No numbers are available though. We have also shown that the intrinsic randomization mechanisms incorporated in GLADE by default are as effective as explicit data randomization.

6. CONCLUSIONS

In this paper, we present a scalable and efficient parallel implementation for incremental gradient descent in GLADE. Our implementation achieves tremendous execution time performance over massive datasets. The most remarkable result is convergence in 80 seconds for the low-rank factorization of a 1 million by 1 million sparse matrix with 10 billion non-empty cells. The implications our solution has on big data analytics in the cloud are dramatic. We can imagine executing complex analytic tasks over massive datasets at virtually no cost or, equivalently, we are able to execute considerably more tasks for the same price. For example, GLADE can execute 324 IGD iterations in the same time Hadoop executes one. We investigate the effect of randomization and parallelism on convergence. The implicit randomization in the push-based GLADE execution strategy is sufficient to achieve convergence. While parallelism speeds-up the execution, it might affect convergence negatively. In future work, we plan to study further the complex interactions between parallelism and convergence.

7. REFERENCES

[1] S. Arumugam and al. The DataPath System: A Data-Centric Analytic Processing Engine for Large Data Warehouses. In *SIGMOD 2010*.

[2] D. P. Bertsekas. Incremental Gradient, Subgradient, and Proximal Methods for Convex Optimization: A Survey. MIT 2010.

[3] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst. HaLoop: Efficient Iterative Data Processing on Large Clusters. *PVLDB*, 3(1), 2010.

[4] Y. Cheng, C. Qin, and F. Rusu. GLADE: Big Data Analytics Made Easy. In *SIGMOD 2012*.

[5] S. Cohen. User-Defined Aggregate Functions: Bridging Theory and Practice. In *SIGMOD 2006*.

[6] J. Duchi, A. Agarwal, and M. J. Wainwright. Distributed Dual Averaging in Networks. In *NIPS 2010*.

[7] X. Feng, A. Kumar, B. Recht, and C. Ré. Towards a Unified Architecture for in-RDBMS Analytics. In *SIGMOD 2012*.

[8] R. Gemulla and al. Large-Scale Matrix Factorization with Distributed Stochastic Gradient Descent. In *KDD 2011*.

[9] J. Hellerstein and al. The MADlib Analytics Library or MAD Skills, the SQL. In *VLDB 2012*.

[10] J. Langford and al. Slow Learners are Fast. In *NIPS 2009*.

[11] F. Niu, B. Recht, C. Ré, and S. J. Wright. A Lock-Free Approach to Parallelizing Stochastic Gradient Descent. In *NIPS 2011*.

[12] C. Qin and F. Rusu. PF-OLA: A High-Performance Framework for Parallel On-Line Aggregation. *CoRR*, abs/1206.0051, 2012.

[13] F. Rusu and A. Dobra. GLADE: A Scalable Framework for Efficient Analytics. *OS Review*, 46(1), 2012.

[14] M. Zaharia and al. Resilient Distributed Datasets: a Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *NSDI 2012*.

[15] M. Zinkevich, M. Weimer, A. Smola, and L. Li. Parallelized Stochastic Gradient Descent. In *NIPS 2010*.