2014 IEEE INTERNATIONAL WORKSHOP ON MACHINE LEARNING FOR SIGNAL PROCESSING, SEPT. 21–24, 2014, REIMS, FRANCE

TRADEOFFS FOR TASK PARALLELIZATION IN DISTRIBUTED OPTIMIZATION

Konstantinos I. Tsianos *Anand D. Sarwate* *Michael G. Rabbat*

Department of ECE
McGill University
Montréal, Québec, Canada

Department of ECE
Rutgers University
Piscataway, NJ, USA

Department of ECE
McGill University
Montréal, Québec, Canada

ABSTRACT

We consider the problem of solving multiple optimization tasks on the same data in a distributed setting. We focus on consensus-based and incremental optimization strategies. Consensus-based distributed optimizers converge in fewer iterations, but the multiple tasks must be run serially. Incremental optimization algorithms, where the iterate is passed from node to node, have slower convergence guarantees but they can be parallelized to work on multiple tasks concurrently. When there are many tasks to solve, this approach can suffer from queuing delay. We provide an analysis of this delay which suggests that incremental algorithms may have superior performance for a moderate number of tasks. The main factor that controls this effect is the communication time. We show experimentally that there is a regime in which parallel instances of incremental algorithms can outperform serial instances of consensus-based algorithms.

Index Terms— distributed optimization, machine learning, consensus algorithms, networks

1. INTRODUCTION

Optimization lies at the heart of many machine learning workflows. In many cases, especially when the volume of data is high, the optimizer has limited time to find a solution [1]. There is an active body of research studying how to harness the power of distributed computing to make optimization methods more efficient for machine learning [2]. Typically, one splits the data across multiple computing nodes to parallelize the processing and communication between the nodes is needed for coordination. In many important cases, however, the cost of local numerical computations is much cheaper than the communication overhead. Consequently, even though communication is needed to obtain a valid solution, it can significantly impact the total time to solve a problem. Another (sometimes overlooked) aspect of working with large datasets is that simply loading the data can take a lot of time. Fortunately, this can be trivially parallelized with modern infrastructure [11]. However, to see the benefit, it may be better to run several optimizations (or *tasks*) over the same data once it is loaded. There are many machine learning scenarios where one naturally needs to solve multiple tasks using the same data. For example, fitting models with free parameters typically requires sweeping through a range of values for parameter tuning. In cross-validation we repeat the same computation on different splits of the data. One approach to multiclass classification is to train several "one-versus-all" classifiers, which could be done in parallel. In exploratory data analysis, one may not know a priori which loss function or kernel function is best suited to the data, and training several algorithms with different characteristics can help determine an appropriate model.

This paper. The communication time, computation time, data dimension, and network size all affect the total time to solve a single optimization problem with a distributed algorithm [3]. In this paper we study issues arising when also parallelizing over *tasks* where the effects of the aforementioned variables are amplified. We consider a standard setting where n nodes are arranged in a network, and the i-th node holds a subset $\mathcal{S}_i$ of the data. The system wishes to solve J different tasks involving the same data. For each task, node i has a local objective $f_i(\mathcal{S}_i)$ and the global objective is to minimize $\sum_{i=1}^{n} f_i(\mathcal{S}_i)$. We consider two classes of asynchronous methods for distributed optimization: consensus-based methods and incremental methods.

In consensus-based methods [4, 5, 6, 7] each node maintains a copy of the optimization variables. Nodes alternate between computing updates using their local objective $f_i(\cdot)$, and exchanging messages with their neighbors so that all nodes' estimates converge to a minimizer of $f(\cdot)$. Each time a node finishes an update it sends its latest state to all of its neighbours. Running J tasks using a consensus-based method is straightforward. One simply invokes the optimizer on the next task once the current one is solved. The overall runtime is J times the runtime required to solve a single task.

Incremental methods [8, 9, 10] are an alternative approach in which only a single node performs an update at any given time, and one single copy of the optimization variables is passed around the network. This reduces the communication cost by having the optimization method "visit" the function components f_i one at a time. Nedic et al. [8] and Bert-

978-1-4799-3694-6/14/$31.00

sekas [9] analyze the situation where every component $f_i(\cdot)$ has to be visited once in every cycle, essentially assuming that the nodes are connected as a complete graph. Johansson et al. [10] generalize this approach for any connected graph in the *Markov incremental gradient descent* (MIGD) algorithm by having the task take a random walk on the graph. In contrast to consensus methods, all J tasks can be solved concurrently using MIGD by having each task take a random walk. However, when two or more tasks arrive at a node, queuing will occur, potentially increasing the total time required to solve all J tasks.

For a single task, incremental methods converge slower in theory and practice compared to consensus methods. For example, on convex and Lipschitz continuous objectives, although the consensus-based method *distributed dual averaging* (DDA) [6] and MIGD both achieve an accuracy that scales as $O(\frac{1}{\sqrt{T}})$ with the number of iterations T, DDA achieves a target error n times faster than MIGD for well-connected graphs such as expanders [6]. Under task parallelization, however, this comparison changes. If $J \leq n$ tasks could be scheduled optimally, an incremental method can potentially solve J tasks in the same time as 1 task, whereas a consensus-based method takes J times longer.

We experimentally verify this prediction by comparing consensus-based serial optimization with parallel incremental optimization. We use DDA and MIGD as exemplars of these two approaches to optimization; other algorithms will have different tradeoffs. We solve a problem in metric learning, where the communication overhead can be significant. A theoretical analysis (deferred to the full version of this paper) predicts the performance for a small-to-moderate number of jobs; empirically, there is a significant range of J for which parallel-MIGD is faster than serial-DDA.

Our results demonstrate that the existing convergence analyses of distributed optimization procedures do not reflect the entire picture. The best (i.e., fastest) optimization algorithm to solve J tasks on the same data depends on the specifics of the tasks and the computing architecture. As in Tsianos et al. [3], the communication time for these algorithms impacts their practical performance. When solving many tasks, amortizing the communication time by *parallelizing over tasks* can potentially be very efficient.

2. FRAMEWORK AND ALGORITHMS

Let $[N] = \{1, 2, \ldots, N\}$. Suppose we have N data points $\{y_1, \ldots, y_N\}$ with each $y_j \in \mathcal{Y}$. The points are divided according to a partition of $[N]$ into n disjoint sets $\mathcal{S}_1, \mathcal{S}_2, \ldots, \mathcal{S}_n$. There are n processors which communicate according to a connected undirected graph $G = (V, E)$ with $V = [n]$. Each processor i has access to data $\mathcal{D}_i = \{y_j : j \in \mathcal{S}_i\}$.

Let $\mathcal{X} \subset \mathbb{R}^d$ be a bounded convex constraint set for which $\max_{x \in \mathcal{X}} \|x\|_2 \leq \sqrt{2}R$. Let $f_i : \mathcal{X} \to \mathbb{R}$ be a convex function with bounded subgradients: $\|g_i(x)\| \leq L$ where $g_i(x) \in \partial_x f_i(x)$ for some $L < \infty$. The aim of the procedure is to minimize the following function over $\mathcal{X}$:

$$f(x) = \frac{1}{n} \sum_{i=1}^{n} f_i(x). \tag{1}$$

We think of $f_i(x)$ as depending on the local data $\mathcal{D}_i$ at node i. For example, in empirical loss minimization a function $l_j(x)$ measures the loss of a model x with respect to the data y_j, and then we simply have $f_i(x) = \frac{n}{N} \sum_{j \in \mathcal{D}_i} l_j(x)$. The global objective being minimized is $F(x) = \frac{1}{N} \sum_{j=1}^{N} l_j(x)$, the average loss over the entire dataset. Let $x^* = \operatorname{argmin}_{x \in \mathcal{X}} f(x)$ and define the error function $\epsilon(x) = f(x) - f(x^*)$.

Distributed optimization methods involving synchronous iterations, such as those built using MapReduce [11] or Spark [12], are susceptible to the "slow node" problem: because all nodes must synchronize between iterations, computation proceeds at the pace of the slowest node. This can be a significant issue in systems where resources are shared among many users or where infrastructure is prone to failure. For this reason, we consider two classes of decentralized optimization methods which can naturally be implemented and analyzed in an asynchronous framework. The two concrete algorithms we consider are DDA [6] and MIGD [10].

In DDA, the nodes use a communication protocol described by a column stochastic matrix P. The matrix P is graph conformant: $p_{i,j} > 0$ iff $(i, j) \in E$. Node i controls the i-th column of P and assigns weights $p_{i,j}$ to its outgoing messages. The algorithm requires the second largest eigenvalue λ_2 of P to calculate the theoretically-optimal step size. To find a solution within the constraint set, each next estimate is projected back into $\mathcal{X}$. The exact form of the projection operator is specific to the problem but is easy to compute when $\mathcal{X}$ has reasonable geometric structure. Finally, the running average $\hat{x}_i(t)$ is maintained locally at each node, and it is the error $\epsilon(\hat{x}_i(t))$ which vanishes as $t \to \infty$. The analysis of DDA [6] shows that all nodes achieve accuracy ϵ after $T = O(\frac{1}{\epsilon^2})$ iterations if nodes communicate over a well-connected graph (e.g., complete or expander).

MIGD [10] is formulated to minimize

$$f(x) = \sum_{i=1}^{n} f_i(x), \quad x \in \mathcal{X}. \tag{2}$$

To match the DDA setting described above, each $f_i(x)$ for MIGD is scaled by $\frac{1}{n}$. Suppose we only have one optimization problem, and node i receives the token for this job. Node i first performs a projected incremental gradient descent update based on its local objective f_i. Once the update computation is completed, i chooses a neighbour j, drawn randomly according to the ith column of a doubly stochastic matrix P, and i passes the token and latest update to j. As explained by Duchi et al. [6], MIGD requires $T = O(\frac{n}{\epsilon^2})$ iterations to reach accuracy $\epsilon > 0$ on a well-connected graph.

3. COMMUNICATION VS. COMPUTATION

Consider solving $J \leq n$ optimization problems using DDA or MIGD. To remove the effect of different graph topologies, we assume that the nodes communicate over a well-connected graph; i.e., either a complete graph or a k-regular expander. To solve J problems with DDA, we run the J jobs one after the other. With MIGD, we start J random walks at different nodes in the network and let them run concurrently. We assume that each processor will only work on one problem at a time, so if two random walks arrive at the same node, one is served and the other is buffered until the node's CPU becomes available to process it.

Let us momentarily assume that all jobs take the same number of iterations to reach the desired accuracy.[1] To solve all problems to ϵ-accuracy, DDA will need $JT = O(\frac{J}{\epsilon^2})$ iterations in total. For MIGD, if the random walks do not collide (so that no job idles in buffer) the time to solve J problems is identical to the time to solve one; i.e., the total number of MIGD iterations is $O(\frac{n}{\epsilon^2})$. Immediately we see that, in this idealized scenario, if $J = \Theta(n)$, MIGD exploits task parallelization to become competitive with DDA. Note, however, that this discussion is in terms of the number of *iterations* or gradient steps taken by the nodes. This abstraction ignores the time for communicating the messages between nodes which, for some problems, may be commensurate with the time to perform a single update. Let us thus refine the analysis following the model of Tsianos et al. [3].

With $J = 1$, at every iteration, each processor i in DDA, or the processor having the token in MIGD, computes a local subgradient on its subset of the data:

$$g_i(x) \in \partial f_i(x) = \frac{n}{N} \sum_{j=1}^{\frac{N}{n}} \partial l_j(x). \tag{3}$$

The cost of this computation increases linearly with the subset size. We normalize time so that one processor computes a subgradient on the full dataset of size N in 1 time unit. Then, dividing the data evenly among n processors, each local gradient computation will take $\frac{1}{n}$ time units. We disregard the time required to compute the projection; often this can be done very efficiently and requires negligible time when N is large compared to n and d.

Next we study the communication cost. One message in either DDA or MIGD is d floating point values (doubles), since this is the dimension of $x(t)$ in MIGD and $z_i(t)$ in DDA. Meta data such as time stamps are $O(1)$ in comparison to d. We account for the cost of communication as follows. In the consensus update of DDA, each node i transmits $p_{ji}z_j$ and receives $p_{ij}z_i$ from each neighbour j in G. Since the message size depends only on the problem dimension d and does not change with N or n, we denote by r the time required to transmit and receive one message, relative to the 1 time unit required to compute the full gradient on all the data. If every node has k neighbors, the cost of one iteration is

$$\frac{1}{n} + kr \quad \text{time units / iteration.} \tag{4}$$

The same approach can be used for MIGD for the node that has the token. Since that node will only transmit the estimate to one neighbor, the cost of a MIGD iteration is simply

$$\frac{1}{n} + r \quad \text{time units / iteration.} \tag{5}$$

Equipped with these time models we can reason about the *time* to achieve ϵ accuracy rather than the number of iterations. Specifically, to reach ϵ accuracy on J problems, DDA will take $J \cdot T$ iterations or

$$\tau_{dda}(\epsilon) = C_{\mathsf{DDA}} \frac{J}{\epsilon^2} \left(\frac{1}{n} + kr \right) \quad \text{time units} \tag{6}$$

where C_{DDA} is a constant that does not depend on n. For MIGD on the other hand, in the ideal case where the random walks do not collide and there are no delays, the total is equivalent to the time to complete T iterations; i.e.,

$$\begin{aligned} \tau_{migd}(\epsilon) =& C_{\mathsf{MIGD}} \frac{n}{\epsilon^2} \left(\frac{1}{n} + r \right) \\ =& C_{\mathsf{MIGD}} \frac{1}{\epsilon^2} (1 + nr) \quad \text{time units.} \end{aligned} \tag{7}$$

This simple manipulation reveals that the relative performance of the two algorithms is more delicate than what the iteration bounds suggest. If G is the complete graph, where $k = n - 1$, then asymptotically as n increases, $\tau_{dda}(\epsilon) = \Theta(\tau_{migd}(\epsilon))$; i.e., the algorithms are equivalent. This is not surprising. Even though MIGD needs n times more iterations, each MIGD iteration takes only $\frac{1}{n}$-th of the time since MIGD requires communication with only one neighbor instead of $n - 1$. Of course, if the message size is very small, the network bandwidth is very high, or computation takes significantly longer than a transmission, then $r \to 0$, and in that case MIGD is indeed n times slower.

For a non-vanishing communication cost, if the graph is a k-regular expander, then as n increases the defining factor is the relative size of Jk and n which multiply the communication/computation tradeoff r in (6) and (7), and it could be the case that either algorithm is faster. Following similar reasoning, these arguments can be generalized to the case where the jobs are not identical. We omit this analysis but refer the reader to the experiments in Section 4 below.

4. EXPERIMENTS

To understand the difference between consensus-based and incremental optimizers when running many jobs, we implement DDA and MIGD on a cluster and compare their performance on logistic regression and metric learning tasks. The

[1] This is the case, e.g., if all jobs are clones of each other. This simplifying assumption helps gaining intuition. Our results do not rely on it.

theoretical analysis of DDA assumes a synchronous algorithm where a barrier mechanism synchronizes all nodes at the end of every iteration [13]. When barriers are used, synchronization forces the algorithm to run at the speed of the slowest node. In our experiments, we use an asynchronous version called Push-Sum DDA (PS-DDA) [14] which converges at the same rate but with different constants. We emphasize that these experiments are designed to illustrate the different regimes for these two procedures, and that the "best" optimization procedure will, in general, depend on the particular problem, data set, and hardware.

Cluster description. Experiments are run on a cluster with 8 machines running Matlab 2009b. Each machine has two 4-core, 2.5GHz Xeon processors with 14GB of RAM. Communication happens over 100Mbps Ethernet. We run 64 worker node processes (one per core), and communication among the nodes is organized as a logical expander graph G. To set G, we sample from the family of Erdős-Rényi random graphs [15] with $p = 0.5$. Such graphs have good expansion properties with high probability. Out of 10^5 samples, we kept the graph for which P has the smallest spectral gap. The implementation of DDA and MIGD is in Matlab using the `labSend` and `labReceive` communication primitives supported by the Parallel Computing Toolbox. No synchronization of the nodes is imposed; both implementations are completely asynchronous and subject to real network conditions and communication delays.

Manipulating the communication time. To demonstrate the tradeoffs discussed above, we first consider a binary classification task. Specifically, we train ℓ_2-regularized logistic regression using the Covertype dataset [16]. We obtain between 1 and 16 different optimization tasks by sweeping the regularization parameter between 0.75 and 0.0001 uniformly on a logarithmic scale. All problems are solved using 32 cores, and all tasks are run until they achieve the same accuracy. In the Covertype dataset we have 522,880 data points with $d = 54$ features each. The time to compute a gradient on the full dataset using a single CPU is roughly 30 seconds, and the time to communicate one 54-dimensional vector to one neighbour is 0.0062 seconds. The relative communication/computation tradeoff is $r = 0.0002$.

Figure 1 depicts the time to completion for DDA and MIGD as a function of the number of tasks for three different communication cost scenarios. The first scenario (solid lines) corresponds exactly to the Covertype problem described above, with the default amount of communication (i.e., each message contains 108 doubles, the model parameters for each class). This is a relatively small amount of communication as compared to the amount of data available, and so DDA consistently finishes faster than MIGD. In order to better understand how the communication-computation tradeoff arises, we repeat the same experiment while artificially increasing the size of each message to either $5,108$ or $10,108$ doubles. The additional communication cost affects

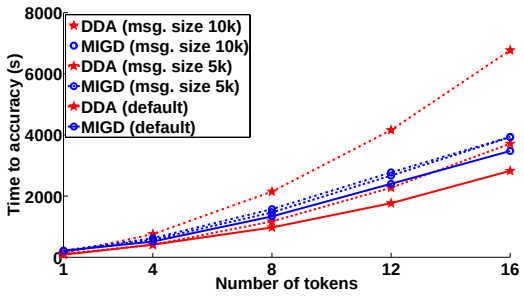

Fig. 1. Solving 1–16 ℓ_2-regularized logistic regression tasks on a cluster of 32 cores. By default (solid lines), messages sizes are small (108 doubles), and DDA is always faster. The two other sets of curves correspond to increasing the message sizes by $5,000$ and $10,000$ doubles, respectively.

the performance of DDA much more significantly than that of MIGD, with the relative amount of time communicating vs. computing determining which algorithm finishes faster.

Large communication time. The previous experiment showed that as the communication time increases for a fixed computation time, the per-iteration cost of DDA can become so large that parallelization in MIGD yields shorter wall time for the optimization of many tasks. To see this effect in a different setting, we evaluate the algorithms on a metric learning task [17] where the message sizes are quite substantial. In metric learning, the input data consists of tuples $\{(u_j, v_j, s_j) : j = 1, \ldots, N\}$ where u_j and v_j are in $\mathbb{R}^d$ and $s_j \in \{-1, 1\}$ is a label indicating whether u_j is "similar" to v_j. The goal is to learn a positive semidefinite symmetric matrix A such that the pseudo-metric

$$D_A(u, v) = \sqrt{(u - v)^T A (u - v)}$$

assigns a small distance to similar points and a large distance to dissimilar points. One way of learning such a matrix is to define a hinge-like loss function

$$l_j(A, b) = \max\{0, s_j(D_A(u_j, v_j)^2 - b) + 1\}$$

where $b \geq 1$ is a bias term, and find A and b which solve:

$$\text{minimize}_{A,b} \frac{1}{N} \sum_{j=1}^{N} l_j(A, b) \qquad \text{subject to } b \geq 1, A \succeq 0.$$

In the distributed setting the data are partitioned into disjoint sets $\mathcal{S}_1, \mathcal{S}_2, \ldots, \mathcal{S}_n$ and the partial objectives are simply $f_i(A, b) = \frac{n}{N} \sum_{j \in \mathcal{S}_i} l_j(A, b)$. Each message contains $\frac{d(d-1)}{2} + d + 1$ doubles to represent the $d \times d$ symmetric matrix A and the bias b.

We use the MNIST digits dataset, where each point is a 28×28 pixel image of handwritten digits 0 to 9. A data point is a $28^2 = 784$-dimensional vector, making each message

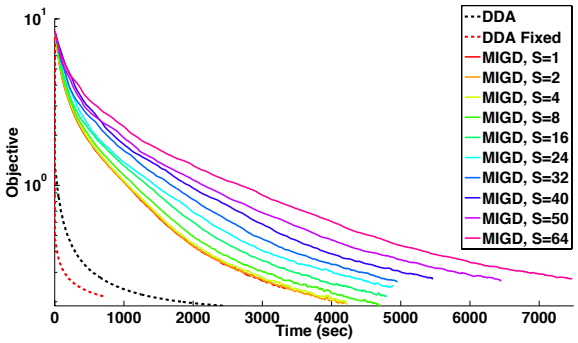

Fig. 2. Evolution of the objective for DDA for diminishing and fixed step size, and MIGD with a varying number of jobs running in parallel.

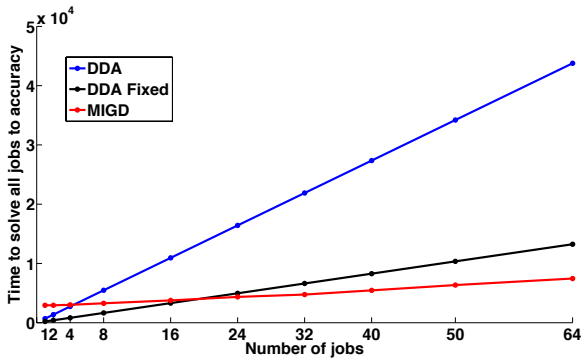

Fig. 3. Cumulative time to solve $J = 1, \ldots, 64$ jobs with MIGD and DDA with fixed and diminishing step size. In both cases there is a crossover point at which MIGD becomes the faster algorithm.

307721 doubles or approximately 2.4 MB in size. For our experiments we use 32000 randomly selected pairs of images, partitioned evenly among the processors.

To solve J instances of this metric learning problem, we run DDA J times serially and run J copies of MIGD in parallel. Because the instances in MIGD move according to a random walk, different instances may collide; if an instance i moves to a node which is already busy processing job j, job i waits in a buffer until the node completes processing job j. When there are many instances, collisions can delay the convergence of MIGD.

Differences when running identical jobs. Our first experiment illustrates the effect of communication cost on distributed optimization. We create a 64 node expander with average node degree 32 ± 2.6. To keep the conditions somewhat symmetric and stay close to the theoretical models discussed earlier, we solve a varying number of *identical* instances of the metric learning problem. Since all instances are identical, they all take approximately the same time to complete. To limit the overhead of monitoring, we focus on one instance. For DDA we measure the time it takes to solve one instance of the problem. For MIGD we track one instance of the problem and measure the total time it takes for this job to converge in the presence of other jobs.

Figure 2 shows the objective value as a function of the total wall time for the two algorithms with varying numbers of jobs. Since MIGD uses a fixed step size, we also show DDA with a fixed step size. As expected, the time for a single DDA instance to complete is very short. MIGD is slower but not by a factor of n. As the number of parallel instances increases there is an additional slowdown for MIGD per instance as a result of buffering from random walk collisions.

In Figure 3 we plot the cumulative time it takes to solve J tasks with DDA and MIGD to a fixed accuracy ϵ, and with J between 1 and 64. We see that there is indeed a tradeoff in wall time between consensus-based and iterative algorithms. For few tasks, DDA is fast enough that it has time to solve 4 tasks one after the other before MIGD solves them in parallel. At that point, MIGD is able to accommodate and solve more and more tasks in parallel and is therefore preferable. Notice also that DDA with fixed step size is faster than DDA with diminishing step size, so the crossover point between DDA and MIGD is moved from 4 to 20 jobs but it still exists.

Example: Sweeping a problem parameter. Next, we turn to a scenario where there is a parameter in the optimization and we wish to run multiple instances for *different* values of the parameter. In this case, instances have different runtimes. As an example we consider ℓ_2-regularized metric learning [20]:

$$\operatorname*{argmin}_{A,b} \frac{1}{N} \sum_{j=1}^{N} l_j(A,b) + \frac{\lambda}{2} \|A\|_F \quad \text{s.t. } b \geq 1, A \succeq 0.$$

Varying the regularization parameter makes the target problem harder (large λ) or easier (small λ). Taking G to be an expander graph of 16 nodes, we solve between 1 and 16 different tasks by sweeping λ from 1 to 0.0001. Figure 4 shows the cumulative amount of time needed to solve all problems with MIGD and DDA with fixed step size. Despite the fact that the jobs are now different from each other, the same tradeoff still exists. With very few jobs, DDA is superior while adding more jobs renders MIGD a better choice. Notice that extrapolating from this figure we see that the benefit of MIGD cannot be sustained. In fact if we let the number of tasks grow, eventually we expect that MIGD would again become slower than DDA due to the increasing number of collisions.

5. DISCUSSION

When solving multiple distributed optimization tasks which use the same data, choosing the right computational algorithm depends on characteristics of the optimization tasks and the system architecture. In this work we illustrate that the

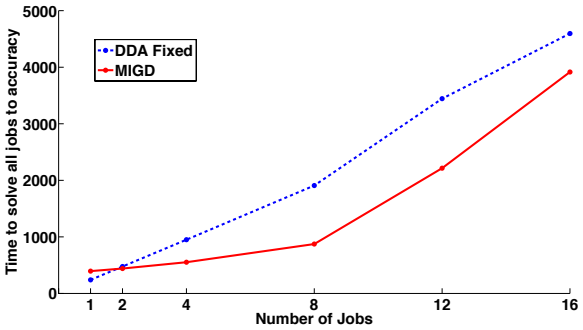

Fig. 4. Cumulative time to solve $J = 1, \ldots, 16$ jobs to accuracy ϵ using DDA with fixed step size and MIGD.

bounds on the iterations for consensus-based and incremental procedures do not necessarily characterize the overall runtime for solving multiple optimizations since there is room for improvement by exploiting task parallelization. In particular, we show that there are regimes where running parallel copies of the incremental MIGD algorithm is faster than running sequential copies of the consensus-based DDA algorithm.

We illustrate this discrepancy via two particular applications, but we believe that the phenomena we exhibit here are general. The two algorithms we consider, DDA and MIGD, are exemplary of two paradigms of asynchronous distributed optimization methods (consensus-based, and incremental, respectively). The regimes for different methods will vary, but we believe it is relatively clear that the tradeoff may appear in many problems of interest.

6. REFERENCES

[1] Leon Bottou, "Large-scale machine learning with stochastic gradient descent," in *Proc. Int'l Conf on Comp. Stat.*, Paris, France, August 2010, pp. 177–187.

[2] Ron Bekkerman, Mikhail Bilenko, and John Langford, *Scaling up Machine Learning, Parallel and Distributed Approaches*, Cambridge University Press, 2011.

[3] Konstantinos I. Tsianos, Sean Lawlor, and Michael G. Rabbat, "Communication/computation tradeoffs in consensus-based distributed optimization," in *NIPS*, 2012.

[4] Angelia Nedic and Asuman Ozdaglar, "Distributed subgradient methods for multi-agent optimization," *IEEE Trans. on Autom. Control*, vol. 54, no. 1, January 2009.

[5] Stephen Boyd, Neal Parikh, Eric Chu, Borja Peleato, and Jonathan Eckstein, "Distributed optimization and statistical learning via the alternating direction method of multipliers," *Foundations and Trends in Machine Learning*, vol. 3, no. 1, pp. 1–122, 2010.

[6] John Duchi, Alekh Agarwal, and Martin Wainwright, "Dual averaging for distributed optimization: Convergence analysis and network scaling," *IEEE Trans. Autom. Control*, vol. 57, no. 3, pp. 592–606, 2011.

[7] S. Sundhar Ram, Angelia Nedic, and Venugopal V. Veeravalli, "Distributed stochastic subgradient projection algorithms for convex optimization," *J. Opt. Th. Appl.*, vol. 147, no. 3, pp. 516–545, 2011.

[8] Angelia Nedic, Dimitri P. Bertsekas, and Vivek S. Borkar, "Distributed asynchronous incremental subgradient methods," in *Inherently parallel algorithms in feasibility and optimization and their applications*, 2000.

[9] Dimitri P. Bertsekas, "Incremental gradient, subgradient, and proximal methods for convex optimization: A survey," Tech. Rep. 2848, MIT-LIDS, 2010.

[10] Bjorn Johansson, Maben Rabi, and Mikael Johansson, "A randomized incremental subgradient method for distributed optimization in networked systems," *SIAM J. Control Opt.*, vol. 20, no. 3, 2009.

[11] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," *Comm. ACM*, vol. 51, no. 1, pp. 107–113, January 2008.

[12] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M.J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *USENIX NSDI*, 2012.

[13] Konstantinos I. Tsianos, Sean Lawlor, and Michael G. Rabbat, "Consensus-based distributed optimization: Practical issues and applications in large-scale machine learning," in *Allerton Conference*, 2012.

[14] Konstantinos I. Tsianos, Sean Lawlor, and Michael G. Rabbat, "Push-sum distributed dual averaging for convex optimization," in *IEEE CDC*, 2012.

[15] Paul Erdős and Alfred Rényi, "On the evolution of random graphs," *Publ. Math. Inst. Hungary. Acad. Sci.*, vol. 5, pp. 17–61, 1960.

[16] Jock A. Blackard and Dennis J. Dean, "Comparative accuracies of neural networks and discriminant analysis in predicting forest cover types from cartographic variables," in *Second Southern Forestry GIS Conference*, Athens, GA, USA, 1998, pp. 189–199.

[17] Eric P. Xing, Andrew Y. Ng, Michael I. Jordan, and Stuart Russell, "Distance metric learning, with application to clustering with side-information," in *NIPS*, 2003.

[18] Kilian Q. Weinberger, Fei Sha, and Lawrence K. Saul, "Convex optimizations for distance metric learning and pattern classification," *IEEE Signal Process. Mag.*, 2010.

[19] Kilian Q. Weinberger and Lawrence K. Saul, "Distance metric learning for large margin nearest neighbor classification," *J. Opt. Th. Appl.*, vol. 10, pp. 207–244, 2009.

[20] Rong Jin, Shijun Wang, and Yang Zhou, "Regularized distance metric learning: Theory and algorithm," in *NIPS*, 2012.