On Aligning Massive Time-Series Data in Splash

Peter J. Haas Yannis Sismanis

IBM Research - Almaden
San Jose, CA, USA
{phaas,syannis}@us.ibm.com

ABSTRACT

Important emerging sources of big data are large-scale predictive simulation models used in e-science and, increasingly, in guiding policy and investment decisions around highly complex issues such as population health and safety. The Splash project provides a platform for combining existing heterogeneous simulation models and datasets across a broad range of disciplines to capture the behavior of complex systems of systems. Splash loosely couples models via data exchange, where each submodel often produces or expects time series having huge numbers of time points and many data values per time point. If the time-series output of one "source" submodel is used as input for another "target" submodel and the time granularity of the source is coarser than that of the target, an interpolation operation is required. Cubic-spline interpolation is the most widely-used method because of its smoothness properties. Scalable methods are needed for such data transformations, because the amount of data produced by a simulation program can be massive when simulating large, complex systems over long time periods, especially when the time dimension is modeled at high resolution. We demonstrate that we can efficiently perform cubic-spline interpolation over a massive time series in a MapReduce environment using novel algorithms based on adapting the distributed stochastic gradient descent (DSGD) method of Gemulla et al., originally developed for low-rank matrix factorization. Specifically, we adapt DSGD to calculate the coefficients that appear in the cubic-spline interpolation formula by solving a massive tridiagonal system of linear equations. Our techniques are potentially applicable to both spline interpolation and parallel solution of diagonal linear systems in other massively parallel data-integration and data-analysis applications.

1. INTRODUCTION

Besides the massive data generated by real-world processes such as sensors and web click-streams, enormous amounts of data are also generated by large-scale, high-resolution predictive simulation models. These models are used for e-science [15] and, increasingly, to guide investment and policy decisions around highly complex issues such as population health and safety [19]. Especially in the latter setting, the challenge of dealing with the massive datasets produced and consumed by these simulation models is compounded by an increasing need to bring together multiple models across a broad range of disciplines. Such model composition is needed to capture the behavior of complex systems of systems and gain synergistic understanding of highly complex problems, avoiding unintended consequences of policy and investment decisions; see, e.g., [10, 18] in the setting of food, climate, and health.

The current work is motivated by the authors' research and development efforts around the Smarter Planet Platform for Analysis and Simulation of Health (Splash). Splash [13, 30] is a platform for combining existing heterogeneous simulation models and datasets to create composite simulation models of complex systems. To facilitate interdisciplinary collaboration and model re-use, Splash facilitates loose coupling of models via data exchange, building upon and extending existing data integration technology. The component models run asynchronously and communicate with each other by reading and writing datasets. In particular, each Splash submodel—e.g., an agent-based simulation model of regional traffic patterns or of disease epidemics—typically produces massive time series having huge numbers of time points and many data values per time point. The time-series dataset output by one "source" submodel is then used as subsequent input for another "target" submodel. Splash applies data transformations as needed for compatibility; the system exploits user-supplied metadata about the submodels and datasets to detect source-target mismatches and help the user design the needed transformations, which are then compiled into code that is invoked at simulation time.

As discussed in [30], Splash uses Clio++, an enhanced version of the Clio data integration tool [12], to let a user semi-automatically design a schema mapping that will "structurally" align the schemas of one or more source models to the schema of a target model. Even when the schemas of a source and target time-series dataset are aligned, however, further *time-alignment* transformations are needed if, for example, the time granularity of a source time series differs from that of the target. Just as the Clio++ GUI allows for semi-automatic design of structural-alignment mappings, a time-aligner GUI—see Figure 1—lets a user choose an appropriate time-alignment mapping for each data item in a time series. The resulting time-series transformation is represented in a formal "time-alignment markup language" (TAML) for future modification or re-use, and the mapping is automatically compiled into runtime code; see [24] for details. Note that the structural and time-alignment transformations are orthogonal: the former mapping is concerned with the structure of the time-series data at each time tick, whereas the latter transformation is concerned about the number of time ticks and the data values at each tick.

Scalable methods are needed for both kinds of data transformations, because the amount of data produced by a simulation

BigData 2012 Istanbul Turkey

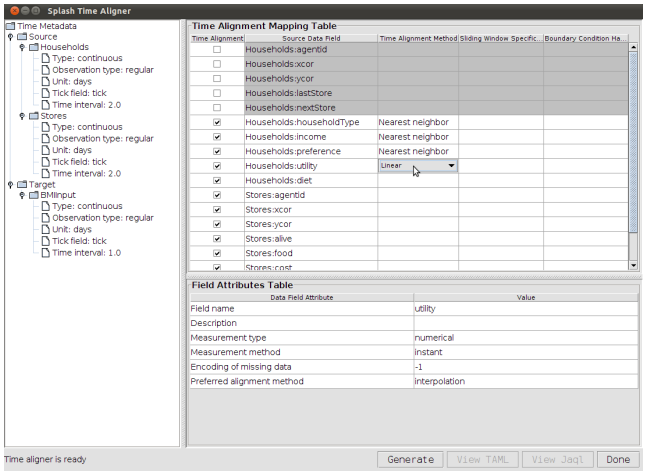


Figure 1: Splash time-alignment GUI [24].

program can be massive when simulating large, complex systems, especially when the time dimension is modeled at high resolution over long time periods. Among the platforms available for large-scale data transformations are MapReduce [7] environments such as Hadoop [1]. A growing number of real-world enterprises are embracing MapReduce systems because of their attractive price/performance characteristics. Even some scientific organizations that have traditionally relied solely on supercomputers for high performance computing (HPC) are increasingly discussing the use of MapReduce systems to connect HPC applications. Splash therefore attempts to provide time-series data transformation techniques suitable for MapReduce environments. Currently, Splash will automatically compile user-specified time-series transformations into JAQL [20] code that will execute the transformations—at scale and on Hadoop—during each simulation run of the composite model.

For time-alignment transformations, a common and important scenario occurs when the granularity of a source time series is coarser than that of the target—e.g., the source model outputs hourly temperatures and the target model needs temperatures every 15 minutes. In this case, a time-series interpolation operation is required. Cubic-spline interpolation is perhaps the most widely-used method because of its smoothness properties. For example, a cubic spline can be shown [22, Prop. 9.2.3] to approximate a function f and its derivative f' with absolute errors of at most $h^{3/2}(\int f''(x)^2\,dx)^{1/2}$ and $h^{1/2}(\int f''(x)^2\,dx)^{1/2}$ at any time point as $h \to 0$, where h is the maximum distance between successive source time ticks. This result assumes only that f is twice continuously differentiable; if $f^{(4)}$ exists and is continuous and the source data ticks are uniformly spaced, then an error bound of $O(h^4)$ can be established. Merit aside, cubic-spline interpolation needs to be supported because it is ubiquitous. For example, the EXPAND function for time-series alignment in the SAS statistical package uses cubic splines as the default interpolation method [28, Ch. 14]. In this paper we focus on the problem of cubic-spline interpolation over massive time series in a MapReduce environment. Besides being important for Splash, scalable cubic-spline interpolation is needed in other data-integration and scientific computing applications; see the references below.

A key challenge in cubic-spline interpolation is that a large-scale tridiagonal linear system $Ax = b$ must first be solved to compute a set of "spline constants" that appear in the interpolation formula. A large literature exists on scalable cubic-spline interpolation and scalable solution of tridiagonal systems. Some of these algorithms impose requirements that are unreasonable or overly stringent in our setting, such as that the source observations be equally spaced [23, 25, 29] or that the inverse of the massive matrix A be (somehow) already computed [4, 25]. Most of the remaining algorithms have focused primarily on vector machines [23], GPUs [33], or message-passing parallel architectures such as MPI [11]. When implemented on a MapReduce platform, these algorithms have large communication overheads, with much data shuffling between mappers and reducers. In recent work on tridiagonal solvers for GPUs, Zhang et al. [33] reviewed a number of prior algorithms and found the Parallel Cyclic Reduction (PCR) algorithm of Hockney and Jesshope [16] to have superior properties, and we focus on this algorithm in the current paper as being exemplary of prior art. Roughly speaking, the original system of m unknowns is partitioned into two systems of $m/2$ unknowns; this partitioning continues recursively, ultimately yielding $\log_2 m$ systems of two unknowns, which are then solved. In a MapReduce setting, the algorithm requires $\log_2 m$ map and reduce jobs, with expensive many-to-many communication occurring over the network during each data shuffling phase.

In this paper, we show how the distributed stochastic gradient descent (DSGD) method of Gemulla et al. [8, 9], originally developed for low-rank matrix factorization, can be adapted to yield novel algorithms for cubic-spline interpolation of massive time series. Specifically, we show how DSGD can be adapted to compute spline constants by solving the system $Ax = b$. By suitable modification of the usual Hadoop `InputFormat` operator, the technique can be implemented as a sequence of map-only jobs, so that network communication is minimal. These techniques are potentially applicable to both spline interpolation problems and solution of k-diagonal and block-diagonal systems in other contexts besides Splash. Our techniques complement those in [17], which implement a broad class of time and geospatial transformations for scientific datasets in an RDBMS.

2. BACKGROUND

We first briefly discuss the general framework for time alignment used by Splash, and then describe the specific problem of distributed cubic-spline interpolation. A thorough discussion of time alignment in Splash can be found in [24].

2.1 A Time-Alignment Framework

We can represent a source dataset comprising a time series as a sequence $S = \big\langle (s_0, d_0), (s_1, d_1) \big\rangle, \ldots, (s_m, d_m) \big\rangle$, where s_i is the time of the ith observation and d_i is the associated data observed at time s_i. Each d_i can be viewed as a k-tuple for some $k \geq 1$. The "ticks", or indices, run from 0 to m. Similarly, we can represent a target dataset as $T = \big\langle (t_0, \tilde{d}_0), (t_1, \tilde{d}_1), \ldots, (t_n, \tilde{d}_n) \big\rangle$. We assume throughout that the source time points are strictly increasing: $s_0 < s_1 < \cdots < s_m$. Moreover, the target time points are required to be evenly spaced and start from simulated time 0, so that $t_i = i\Delta$ for some $\Delta > 0$ and all $i \in \{\, 0, 1, \ldots, n \,\}$. This is usually the case in practice: if there are no requirements at all on the target time points, then no time alignment is needed, and there is usually no natural way to define target time points at irregular intervals. (We make no assumptions about the regularity of the source time points.) Note that no loss of generality is entailed by assuming that $t_0 = 0$, since otherwise the data can be uniformly shifted along the time axis. We also assume that there are no invalid or missing source data values; i.e., we assume that any such values have been fixed or imputed prior to the time-alignment step. Finally, we assume for simplicity that $s_0 = t_0$ and $s_m = t_n$, so that the source data completely spans the target-data interval.

In the spirit of [17], all time-alignment operations can be viewed as applying an "alignment function" over a "window". Specifically, suppose that we wish to compute the target data value $\tilde{d}_i$ at target time t_i for some $i \in \{1, 2, \ldots, n\}$. Then the *window* W_i for t_i is simply a subsequence of S that contains the information needed to perform the computation. Often, the window has the form

$$W_i = \langle (s_j, d_j), (s_{j+1}, d_{j+1}), \ldots, (s_{j+k}, d_{j+k}) \rangle,$$

where $s_j \leq t_i \leq s_{j+k}$, so that the window comprises data observed at contiguous times points of S that span the target time point t_i. If the window width $|W_i|$—that is, the number of (s_i, d_i) pairs in W_i—is the same for each t_i, then the sequence of windows behaves as a sliding window over S. (The window may progress in "bursts" in that $W_{i-1} \neq W_i = W_{i+1} = \cdots = W_{i+l_i} \neq W_{i+l_i+1}$ for one or more values of i, since a given set of source data points might suffice for computing multiple target data points.) The desired aligned data $\tilde{d}_i$ is computed by applying an appropriate *alignment function* to the data in W_i. See [24] for further discussion.

In this paper we focus on time alignments that require cubic-spline interpolation. (Other types of time alignments that Splash can capture include aggregation and allocation; they are needed, e.g., when each data point represents the total rainfall since the last time tick.) For simplicity, we suppose at first that each source-data observation d_i comprises a single attribute value. In principle, an alignment procedure needs to be performed separately for each attribute. In practice, multiple alignment operations might be performed during a single scan over the source data; see also Section 3.5 below.

2.2 Spline Interpolation

The most common types of interpolation are *piecewise linear interpolation* and *natural cubic-spline interpolation*. For linear interpolation, the window is defined as $W_i = \langle (s_j, d_j), (s_{j+1}, d_{j+1}) \rangle$, where $j = \max\{ n : s_n \leq t_i \}$. Note that $s_j \leq t_i < s_{j+1}$. Then the alignment function computes the interpolated data value as

$$\tilde{d}_i = d_j + \frac{t_i - s_j}{s_{j+1} - s_j}(d_{j+1} - d_j).$$

Cubic splines are more complex, but have better smoothness properties. The idea is to define a function $d(x)$ such that (1) $d(s_j) = d_j$ for each j, (2) d is a cubic polynomial over each interval $[s_j, s_{j+1}]$, and (3) the second derivative d'' exists and is continuous throughout the interval $[s_0, s_m]$. We focus on *natural* cubic splines for which $d''(s_0) = d''(s_m) = 0$, so that the interpolating function looks like a straight line to the left and right of $[s_0, s_m]$. Given such a function, the interpolated target value $\tilde{d}_i$ at time t_i is given as $\tilde{d}_i = d(t_i)$. In more detail, set $h_j = s_{j+1} - s_j$ for $0 \leq j \leq m-1$, and let $x = (x_1, x_2, \ldots, x_{m-1})$ be the solution to the linear system $Ax = b$, where A and b are given in Figure 2. Next, set $\sigma_0 = \sigma_m = 0$ and $\sigma_j = x_j$ for $1 \leq j \leq m-1$. Then the window $W_i = \langle (s_j, d_j), (s_{j+1}, d_{j+1}) \rangle$ is defined exactly as for piecewise linear interpolation, but the alignment function now computes the interpolated data value as

$$\tilde{d}_i = \frac{\sigma_j}{6h_j}(s_{j+1} - t_i)^3 + \frac{\sigma_{j+1}}{6h_j}(t_i - s_j)^3 + \Big(\frac{d_{j+1}}{h_j} - \frac{\sigma_{j+1}h_j}{6}\Big)(t_i - s_j) + \Big(\frac{d_j}{h_j} - \frac{\sigma_j h_j}{6}\Big)(s_{j+1} - t_i).$$

See, e.g., [22, Ch. 10] for details.

Suppose for the moment that the σ_j values have been computed in a preprocessing step. Note that the information in the window W_i defined for cubic-spline and linear interpolation in the previous section is not quite enough to compute the interpolated value $\tilde{d}_i$ because we also need the quantities σ_j and σ_{j+1}. We can still fit this interpolation scheme into our general framework if we augment each source tuple d_j by appending σ_j as a new attribute; in the following, we assume that the σ_j's have been appended in this manner. A key observation is that the interpolated target values can be computed in a completely distributed manner, since the W_i windows can be processed in parallel to produce tuples of the form $(t_i, \tilde{d}_i)$. After all such tuples are produced, a parallel sort by t_i value can be used to construct the final target time series. As noted previously, a given window W_i corresponding to successive source time points s_j and s_{j+1} can produce multiple $(t_i, \tilde{d}_i)$ pairs if multiple target time points lie in the interval $[s_j, s_{j+1})$. The remaining (major) question is how to compute the spline constants σ_j in a distributed fashion in MapReduce. This is the focus of the remainder of the paper.

3. COMPUTING SPLINE CONSTANTS

To compute the σ_j spline parameters, we must solve the linear system $Ax = b$, given previously, in a distributed manner. To develop an algorithm well suited for MapReduce architectures, we adapt the distributed stochastic gradient descent algorithm of Gemulla et al. [8, 9].

3.1 Formulation as an SGD Problem

The first step is to recast the problem as a minimization problem. Set $L_i(x) = (A_{i\cdot}x - b_i)^2$ for $1 \leq i \leq m-1$, where $A_{i\cdot}$ denotes the ith row of the $(m-1) \times (m-1)$ matrix A. Solving the linear system $Ax = b$ is equivalent to minimizing the function $L(x) = L_1(x) + \cdots + L_{m-1}(x)$. In the Splash setting, we can tolerate approximate solutions that yield values of L that are close, but not exactly equal, to the minimum possible value: slight deviations from the optimal solution merely cause a slight decrease in smoothness in the cubic-spline interpolation function. This suggests the use of an iterative *gradient descent* procedure that starts with an initial value $x^{(0)}$ and repeatedly takes "downhill" steps that decrease the value of L, using the recursion

$$x^{(n+1)} = x^{(n)} - \epsilon_n \nabla L(x^{(n)}). \tag{1}$$

Here $\{\epsilon_n\}$ is a sequence of decreasing step sizes and ∇L is the gradient of L with respect to x, so that $-\nabla L(x)$ represents the direction of maximum decrease in L, starting from point x. The step sizes must decrease in order to allow convergence to a solution, but must decrease slowly enough so that the algorithm does not get stuck at a solution far from the optimum; a typical sequence is of the form $\epsilon_n = n^{-\alpha}$ for some $\alpha \in (0.5, 1]$. In our setting, a good first approximation $x^{(0)}$ can be obtained by simply ignoring the off-diagonal terms in the matrix A and solving the resulting diagonal system to obtain

$$x_i^{(0)} = \frac{3}{h_{i-1} + h_i}\Big(\frac{d_{i+1} - d_i}{h_i} - \frac{d_i - d_{i-1}}{h_{i-1}}\Big)$$

for $1 \leq i \leq m-1$. It is well known [25] that the inverse matrix A^{-1} tends to be "diagonally dominant" in that the magnitude of the matrix entries decreases exponentially with distance from the main diagonal. Ignoring off-diagonal entries should therefore yield a reasonable first approximation.

For our specific function L, we have

$$\nabla L(x) = \sum_{i=1}^{m-1} \nabla L_i(x) \tag{2}$$

by the linearity of the gradient operator. Denote by $a_{i,j}$ the (i,j)th component of A and by b_i the ith component of b, so that

$$b_i = \frac{d_{i+1} - d_i}{h_i} - \frac{d_i - d_{i-1}}{h_{i-1}} \tag{3}$$

$$A = \begin{bmatrix} \frac{h_0+h_1}{3} & \frac{h_1}{6} & 0 & \cdots & 0 & 0 & 0 \\ \frac{h_1}{6} & \frac{h_1+h_2}{3} & \frac{h_2}{6} & \cdots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & \frac{h_{m-3}}{6} & \frac{h_{m-3}+h_{m-2}}{3} & \frac{h_{m-2}}{6} \\ 0 & 0 & 0 & \cdots & 0 & \frac{h_{m-2}}{6} & \frac{h_{m-2}+h_{m-1}}{3} \end{bmatrix} \quad b = \begin{bmatrix} \frac{d_2-d_1}{h_1} - \frac{d_1-d_0}{h_0} \\ \frac{d_3-d_2}{h_2} - \frac{d_2-d_1}{h_1} \\ \vdots \\ \frac{d_m-d_{m-1}}{h_{m-1}} - \frac{d_{m-1}-d_{m-2}}{h_{m-2}} \end{bmatrix}.$$

Figure 2: Matrix and vector components of linear system for cubic-spline constants.

and

$$a_{i,j} = \begin{cases} h_{i-1}/6 & \text{if } j = i-1 \\ (h_{i-1}+h_i)/3 & \text{if j = i} \\ h_i/6 & \text{if } j = i+1 \\ 0 & \text{otherwise.} \end{cases} \tag{4}$$

For convenience, define $x_0 = x_m = 0$ and define $a_{i,j} = 0$ if $i \notin \{1, 2, \ldots, m-1\}$ or $j \notin \{1, 2, \ldots, m-1\}$. Then the jth component of the ith partial gradient is given by

$$\begin{aligned} \nabla_j L_i(x) &= \nabla_j (A_{i\cdot} x - b_i)^2 = \nabla_j \Big(\sum_{k=1}^{m-1} a_{i,k} x_k - b_i \Big)^2 \\ &= \begin{cases} u_{i,j} & \text{if } i-1 \le j \le i+1 \\ 0 & \text{otherwise,} \end{cases} \end{aligned} \tag{5}$$

where $u_{i,j} = 2a_{i,j}(a_{i,i-1}x_{i-1} + a_{i,i}x_i + a_{i,i+1}x_{i+1} - b_i)$.

Because the gradient ∇L can be expressed as a sum of partial gradients as in (2), the gradient descent procedure can be parallelized in a MapReduce environment by partitioning the data across processing nodes. Each node sums up the partial gradients for its part of the data, and the partial gradients are then aggregated at the reducers. This can be done conveniently by using a query language on top of MapReduce. Once the gradient has been computed, a master node will perform the update of the parameter values [6, 14, 26]. Our version of this algorithm—which uses the sophisticated L-BFGS-B updating formula [3]—is called DGD in the experimental section. Drawbacks of this approach include the need to store the entire L and ∇L arrays in memory, and slow empirical convergence rates.

In their seminal 1951 paper, Robbins and Monro [27] showed that, in terms of the speed of convergence, it is often a better idea to compute a simple, computationally cheap approximation of the gradient at each descent step and use the resulting cost savings to take more steps per unit of computation time. The averaging operation that is implicit in the stochastic recursion (1) has the effect of smoothing out the gradient estimates over time, so that the algorithm converges to a good solution. The resulting class of "stochastic approximation" algorithms has proved extremely successful in practice; see, e.g., the book of Kushner and Yin [21]. In our setting, one simple algorithm along these lines is *stochastic gradient descent (SGD)*. The algorithm is identical to standard gradient descent, but instead of computing a gradient $\nabla L(x)$ as a sum of $m-1$ terms of the form $\nabla L_i(x)$, the sum is approximated by choosing a value of i at random, and then scaling up the partial gradient by a factor of $m-1$. That is, $\nabla L(x)$ is approximated at the nth step by $Y_n(x) = (m-1)\nabla L_{\alpha(n)}(x)$, where each $\alpha(n)$ is sampled randomly and uniformly from $\{1, 2, \ldots, m-1\}$. Thus each Y_n is random, but equal to ∇L in expectation. The recursion is now given by $x^{(n+1)} = x^{(n)} - \epsilon_n Y_n(x^{(n)})$. Under mild conditions [21], the random sequence $\{x^{(n)}\}$ converges to the optimal solution x^* as $n \to \infty$. The main challenge with the SGD algorithm is that the algorithm is purely sequential, and so does not lend itself to a MapReduce implementation.

Gemulla et al. [9] devised a distributed version of SGD in the context of large-scale matrix factorization. We now extend this work to show that SGD can be distributed in the cubic-spline setting.

3.2 A Stratified SGD Algorithm

We start by defining a "stratified" version of stochastic gradient descent, called SSGD. Assume for simplicity that $m-1$ is divisible by 3, and define the following *strata*:

$$\begin{aligned} U^1 &= \{1, 4, 7, \ldots, m-3\} \\ U^2 &= \{2, 5, 8, \ldots, m-2\} \\ U^3 &= \{3, 6, 9, \ldots, m-1\}. \end{aligned}$$

Then set $L^s(x) = 3\sum_{i \in U^s} L_i(x)$ for $s = 1, 2, 3$, so that $L(x) = (1/3)L^1(x) + (1/3)L^2(x) + (1/3)L^3(x)$. We refer to L^1, L^2, and L^3 as *stratum loss functions*. SSGD proceeds in a manner similar to ordinary SGD but, at each iteration, the algorithm takes a downhill step with respect to one of the stratum losses L^s, i.e., approximately in the direction of the negative gradient $-\nabla L^s(x)$. Although each such direction is "wrong" with respect to minimization of the overall loss L, SSGD will converge (under appropriate regularity conditions) to a good solution for L if the sequence of strata is chosen "carefully", which means roughly that over many steps the three strata will be chosen equally often. The motivation for SSGD is that, as discussed in the sequel, the SGD algorithm can be run in a highly parallel manner within each of the three strata.

To develop SSGD, suppose that there is a (potentially random) *stratum sequence* $\{\gamma_n\}$, where each γ_n takes values in $\{1, 2, 3\}$ and determines the stratum to use in the nth iteration. Using a noisy observation Y_n of the gradient $\nabla L^{\gamma_n}(x^{(n)})$, we obtain the update rule

$$x^{(n+1)} = x^{(n)} - \epsilon_n Y_n. \tag{6}$$

Here $Y_n = (m-1)\nabla L_{\alpha(n)}(x^{(n)})$, where $\alpha(n)$ is sampled randomly and uniformly from U^{γ_n}. Note that the stratum size is $(m-1)/3$ and that the partial gradient term is $3\nabla L_{\alpha(n)}(x^{(n)})$, so that the factor of 3 cancels out. Our key assumption on the stratum sequence $\{\gamma_n\}$ is that it is *regenerative* [2, Ch. VI], in that there exists an increasing sequence of almost-surely finite random indices $0 = \beta(0) < \beta(1) < \beta(2) < \cdots$ that serves to decompose $\{\gamma_n\}$ into consecutive, independent and identically distributed (i.i.d.) cycles $\{C_k\}$, with $C_k = \{\gamma_{\beta(k-1)}, \gamma_{\beta(k-1)+1}, \ldots, \gamma_{\beta(k)-1}\}$ for $k \ge 1$. I.e., at each $\beta(i)$, the stratum is selected according to a probability distribution that is independent of past selections, and the future sequence of selections after step $\beta(i)$ looks probabilistically identical to the sequence of selections after step $\beta(0)$. The length τ_k of the kth cycle is given by $\tau_k = \beta(k) - \beta(k-1)$. Letting $I_{\gamma_n = s}$ be the indicator variable for the event that stratum s is chosen in the nth step, set $X_k(s) = \sum_{n=\beta(k-1)}^{\beta(k)-1} \big(I_{\gamma_n = s} - (1/3)\big)$ for $s = 1, 2, 3$. It follows from the regenerative property that the pairs $\{\big(X_k(s), \tau_k\big)\}$ are independent and identically distributed (i.i.d.) for each s.

THEOREM 3.1. *Suppose that A is non-singular, so that the system $Ax = b$ has a unique solution x^*. Also suppose that (i) $\epsilon_n = O(n^{-\alpha})$ for some $\alpha \in (0.5, 1]$, (ii) $(\epsilon_n - \epsilon_{n+1})/\epsilon_n = O(\epsilon_n)$, and (iii) $\{\gamma_n\}$ is regenerative with $E[\tau_1^{1/\alpha}] < \infty$ and $E[X_1(s)] = 0$ for $s = 1, 2, 3$. Then the sequence $\{x^{(n)}\}$ defined by* (6) *converges to x^* with probability 1.*

Thus, under regularity conditions, we may pick any regenerative sequence $\{\gamma_n\}$ such that $E[X_1(s)] = 0$ for all strata. The condition $E[X_1(s)] = 0$ essentially requires that, for each stratum s, the expected fraction of visits to s in a cycle equals $1/3$. By the strong law of large numbers for regenerative processes [2, Sec. VI.3], this condition—in the presence of the finite-moment condition on τ_1—is equivalent to requiring that the long-term fraction of visits to each stratum equals $1/3$. The finite-moment condition is satisfied for schemes in which the number of successive steps taken within a stratum is bounded with probability 1, as is typically the case.

The conditions on $\{\epsilon_n\}$ are often satisfied in practice, e.g., when $\epsilon_n = 1/n$ or when $\epsilon_n = 1/\lceil n/k \rceil$ for some $k > 1$ with $\lceil x \rceil$ denoting the smallest integer greater than or equal to x, so that the step size remains constant for some fixed number of steps. Similarly, a wide variety of strata-selection schemes satisfy the conditions of the theorem. Examples include (1) running precisely $c/3$ steps on stratum s in every "chunk" of c steps, and (2) repeatedly picking a stratum according to some fixed distribution $\{\, p_s > 0 \,\}$ and running $c/(3p_s)$ steps on the selected stratum s. Certain schemes in which the number of steps per stratum is random are also covered by Theorem 3.1; see [8].

A sketch of the proof is as follows. First consider a modification of the recursion in (6) of the form $x^{(n+1)} = \Pi_H[x^{(n)} - \epsilon_n Y^{\gamma_n}(x^{(n)})]$, where $H \subset \Re^{m-1}$ is a (large) hyper-rectangle centered at x^* and containing the initial value $x^{(0)}$, and $\Pi_H[x]$ projects the point $x \in \Re^{m-1}$ onto H. The conditions of the theorem ensure that $\epsilon_n \to 0$, $\sum_n \epsilon_n = \infty$, and $\sum_n \epsilon_n^2 < \infty$. Moreover, the functions L_s and ∇L_s are bounded, continuous, and differentiable on H, as are L and ∇L. Letting $\mathcal{F}_n$ denote the σ-field generated by $\{\, \alpha(i-1), x^{(i)}, \gamma_i,\ i \le n \,\}$, i.e., what is known at step n (just prior to generating the value of Y_n), we have $E[Y_n \mid \mathcal{F}_n] = \nabla L^{\gamma_n}(x^{(n)})$ with probability 1. Finally,

$$\begin{aligned}\sup_n E\big[\|Y_n\|_2^2\big] &\le \sup_n \sup_{x \in H} c^2 \big(L^{\gamma_n}(x)\big)^2 \\ &\le \sup_s \sup_{x \in H} c^2 \big(L^s(x)\big)^2 < \infty,\end{aligned}$$

where $c = (m-1)/3$. Although the direction of each downhill step is "wrong" in that the mean direction is $-\nabla L^{\gamma_n}$ rather than $-\nabla L$, suppose that the directions "average out correctly" in the sense that, for any $x \in H$,

$$\lim_{n \to \infty} \epsilon_n \sum_{i=0}^{n-1} \big[\nabla L^{\gamma_i}(x) - \nabla L(x)\big] = 0 \tag{7}$$

with probability 1. (For example, if ϵ_n were equal to $1/n$, then the nth term would represent the empirical average deviation from the true gradient over the first n steps.) Then by some general results from stochastic approximation theory—see the proof of Theorem 1 in [8]—it follows that $\{x^{(n)}\}$ converges to the set of limit points of the projected ODE $\dot{x} = -\nabla L(x) + z$ for any initial condition, where z is the "minimum force" needed to keep the solution in H [21, Sec. 4.3]. Because of the quadratic nature of the loss function, there is in fact a unique limit point, namely x^*. As shown in [9, Th. 1] using results from regenerative process theory, the condition in (7) is implied by the asserted boundedness and differentiability of L, together with the hypotheses on $\{\epsilon_n\}$.

The final step is to focus on the actual, non-projected algorithm and show that, for a sufficiently large hyper-rectangle H centered on x^*, the $\{\epsilon_n\}$ sequence will hit the set H infinitely often with probability 1, so that we can apply the foregoing arguments to the process observed at these recurrence times. We follow the Liapunov-function approach; see, e.g., [21, Sec. 4.5] or [32]. The basic idea is to identify a non-negative function $V(x)$—which can be viewed as the "distance" of x from the point x^*—such that, for sufficiently large n and for $x^{(n)} \notin H$, the expected one-step drift is negative, i.e., back toward H. In more detail, let $\mathcal{F}_n$ be as above. Then the drift condition requires that there exists $\delta > 0$ such that, for all sufficiently large n,

$$\epsilon_n^{-1} E[V(x^{(n+1)}) - V(x^{(n)}) \mid \mathcal{F}_n] \le -\delta \qquad \text{if } x^{(n)} \notin H$$

with probability 1. If the drift condition holds, then H has the desired recurrence property by Theorem 4.5.4 in [21]. (Indeed, the sequence $\{x^{(n)}\}$ is "positive recurrent" in that the expected number of steps between visits to H is finite.) In our setting, we can take $V(x) = \|x - x^*\|_2^2$. To see this, fix $x \notin H$ and $s \in \{\, 1, 2, 3 \,\}$. Assume for ease of notation that $x^* = (0, 0, \ldots, 0)$. Denote by E_n expectation conditioned on $\mathcal{F}_n$ and, using (2)–(4), write

$$\begin{aligned}&\epsilon_n^{-1} E_n[V(x^{(n+1)}) - V(x^{(n)}) \mid x^{(n)} = x, \gamma_n = s] \\ &\quad = -2x^t E_n[Y_n \mid x^{(n)} = x, \gamma_n = s] \\ &\qquad + \epsilon_n E_n\big[Y_n^t Y_n \mid x^{(n)} = x, \gamma_n = s\big] \\ &\quad = -\frac{3}{m-1} \sum_{i \in U^s} 2\theta_i(\theta_i - b_i) + \epsilon_n \frac{3}{m-1} \sum_{i \in U^s} c_i \theta_i^2,\end{aligned}$$

where $\theta_i = \theta_i(x) = a_{i,i-1}x_{i-1} + a_{i,i}x_i + a_{i,i+1}x_{i+1}$ and $c_i = 4(a_{i,i-1}^2 + a_{i,i}^2 + a_{i,i+1}^2)$. By choosing H sufficiently large, we can ensure that $x \notin H$ implies that $|\theta_i|$ is sufficiently large so that $2\theta_i(\theta_i - b_i) \ge \theta_i^2$ for $i \in U^s$, which implies that

$$\begin{aligned}&E_n[V(x^{(n+1)}) - V(x^{(n)}) \mid x^{(n)} = x, \gamma_n = s] \\ &\quad \le -\frac{3}{m-1} \sum_{i \in U^s} (2 - \epsilon_n c_i)\theta_i^2 \\ &\quad \le -(2 - \epsilon_n c)\theta_*^2,\end{aligned}$$

where $c = \max_{1 \le i \le m-1} c_i$ and

$$\theta_* = \min_{1 \le i \le m-1} \min_{x \notin H} |\theta_i(x)|.$$

Unconditioning on γ_n, we have, with probability 1,

$$E[V(x^{(n+1)}) - V(x^{(n)}) \mid \mathcal{F}_n] \le -(2 - \epsilon_n c)\theta_*^2$$

whenever $x^{(n)} \notin H$. Since $\epsilon_n \to 0$, we have $2 - \epsilon_n c \ge 1$ for sufficiently large n, and the drift condition holds with $\delta = \theta_*^2 > 0$.

3.3 The DSGD Algorithm

The SSGD algorithm given in the previous subsection leads to a distributed SGD algorithm for solving the system $Ax = b$. The basis of this DSGD algorithm is the observation that SGD can be run in a highly parallel manner within each stratum.

As before, assume that $m - 1$ is divisible by 3, and suppose that we have a d-node shared-nothing environment such as MapReduce. Again for simplicity, assume that d divides $m - 1$ and set $r = (m-1)/d$; we assume that $r \ge 3$. We distribute the matrix A and vector b only once across the d nodes, with node q receiving the coefficients $\{\, a_{i,j} : r(q-1) \le i \le rq \text{ and } i - 1 \le j \le i + 1 \,\}$ and $b_{r(q-1)+1}, b_{r(q-1)+2}, \ldots, b_{rq}$. Whereas most $a_{i,j}$ coefficients appear at a unique node, some "boundary" coefficients appear at two successive nodes.

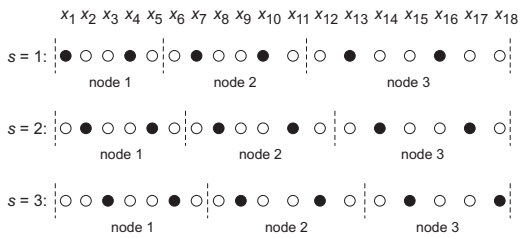


Figure 3: Stratum elements and node assignments for x components in DSGD algorithm ($m-1=18$ components, $d=3$ nodes). Solid circles denote stratum elements for stratum $s=1,2,3$.

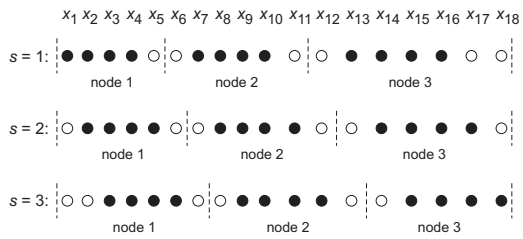


Figure 4: Stratum elements and node assignments for x components in alternative DSGD algorithm ($m-1=18$ components, $d=3$ nodes). Solid circles denote stratum elements for stratum $s=1,2,3$.

The individual steps in DSGD are grouped into *sub-epochs*, each of which amounts to processing one of the strata; we define an *epoch* to comprise a sequence of three sub-epochs. In more detail, DSGD makes use of a sequence $\{(\xi_k, T_k)\}$, where ξ_k denotes the *stratum selector* used in the kth sub-epoch, and T_k the number of steps to run on the selected stratum. Note that this sequence of pairs uniquely determines an SSGD stratum sequence as in Section 3.2: $\gamma_1 = \cdots = \gamma_{T_1} = \xi_1$, $\gamma_{T_1+1} = \cdots = \gamma_{T_1+T_2} = \xi_2$, and so on. The $\{(\xi_k, T_k)\}$ sequence is chosen such that the underlying SSGD algorithm, and hence the DSGD algorithm, is guaranteed to converge. Once a stratum ξ_k has been selected, we perform T_k SGD steps on U^{ξ_k}; this is done in a parallel and distributed manner, as follows.

Suppose that $\xi_1 = 1$, so that we first run T_1 steps of the SGD algorithm on stratum $s=1$. We distribute $x^{(0)}$ across the d nodes, with node q receiving components $x^{(0)}_{r(q-1)}, x^{(0)}_{r(q-1)+1}, \ldots, x^{(0)}_{rq-1}$; see Figure 3. (Node d receives the additional component $x^{(0)}_{m-1}$.) We keep the step size fixed at ϵ during the T_1 processing steps of this stratum.[1] Observe that, by (2)–(4), whenever the nth SGD step randomly selects an index $\alpha(n) = i$ corresponding to a component $x^{(n)}_i$ at a node q and then updates $x^{(n)}$ by adding the term $-\epsilon Y_n = -\epsilon(m-1)\nabla L_i(x^{(n)})$, the only affected components are $x^{(n)}_{i-1}$, $x^{(n)}_i$, and $x^{(n)}_{i+1}$, which are also at node q by design; the coefficients of A and b needed to compute the update also reside at node q. The crucial observation is that updates for different stratum elements have completely disjoint effects. For example, suppose that we run SGD in stratum 1 and that the first four random elements selected are x_7, x_4, x_{10}, and x_{12}. The update obtained by the addition of $-\epsilon(m-1)\nabla L_1(x_7)$ to x affects only elements x_6, x_7, and x_8. The update based on x_4 only affects x_3, x_4, and x_5. These two updates, and indeed all four updates, can thus be applied in any order. More generally, we obtain the same result after T_1 SGD steps in stratum 1 either by executing the T_1 steps in sequence or by executing the steps at the different nodes independently and in parallel and simply summing the updates; see Theorem 4 in [8] for a formal argument.

For the stratification scheme of Figure 3 it is in fact the case that we can apply the SGD updates *within* a node in arbitrary order. Thus, if the sequential algorithm chooses components in U^1 at random, we can correctly "simulate" the algorithm by first generating a random vector $W \approx \text{Multinomial}\big(T_1, (1/d, 1/d, \ldots, 1/d)\big)$ to determine the number of samples to take at each node; then, in parallel at each node q, we draw a sample of W_q components with replacement and process updates corresponding to these components in arbitrary order. The computation of W_q can be performed in a distributed manner by specifying a pseudorandom number generator (PRNG) and running an independent instance of the PRNG at each node, with all PRNGs initialized using the same pseudorandom number seed. An even simpler scheme—which we use in our experiments—samples every element in stratum 1 exactly once. Elements are selected according to a sequence that is randomly and uniformly selected from among all possible $n_i!$ such sequences, where n_i is the number of elements at node i that belong to stratum 1. This approach balances randomness with thorough coverage of the elements in the stratum, and was found to work well in the setting of matrix factorization [9].

The SGD algorithm is run on the other two strata in an analogous manner, with a slight reshuffling of the data required at each change of stratum. If, for example, we have taken T_1 SGD steps (in parallel) on stratum 1, and now wish to take T_2 steps on stratum 2, each node $q > 1$ needs to transmit the leftmost component $x^{(T_1)}_{r(q-1)}$ to node $(q-1)$. After such a transfer, each node q now contains the components $x^{(T_1)}_{r(q-1)+1}, x^{(T_1)}_{r(q-1)+2}, \ldots, x^{(T_1)}_{rq}$. Similar data transfers are required at each sub-epoch. As discussed in Section 4.1 below, the Hadoop implementation of DSGD "hides" these transfers within the standard initialization processing for a map job, minimizing network traffic.

Analogously to our desire to trade randomness and thoroughness when running SGD within a stratum, we also want to trade off these two factors when selecting the strata themselves. At one end of the spectrum, we can visit the strata randomly, in an i.i.d. manner, i.e., so that the sequence $\{\xi_k\}$ is i.i.d.. Then the start of each sub-epoch corresponds to a regeneration point, so that Theorem 3.1 applies and convergence is assured. Our preferred approach is to systematically visit each of the three strata during each epoch. The sequence in which to visit the strata is chosen at random from the set of the $3! = 6$ possible sequences. In this case, the regeneration points correspond to the epochs rather than the sub-epochs.[2] As with the previous intra-stratum sampling scheme, this latter approach was found to work well in the setting of matrix factorization [9].

3.4 An Alternative DSGD Algorithm

An alternate version of the algorithm of the previous section can be obtained by choosing a different set of strata. As before, assume that $m-1$ is divisible by 3, and that we have d nodes, where d

[1] We can handle schemes where the step size changes from step to step rather than at each sub-epoch, but this requires complex bookkeeping and random number synchronization.

[2] The convergence argument in these cases is actually slightly subtle: when applying Theorem 3.1 we take each index $i \in \{1, 2, \ldots, m-1\}$ as its own stratum to ensure the conditional unbiasedness property of the gradient estimates, and observe that the regeneration points—which were defined with respect to the three "big" strata—also correspond to regeneration points with respect to the $m-1$ small strata.

divides $m-1$. Set $r = (m-1)/d$ and

$$U^1 = \bigcup_{q=1}^{d} \{\, r(q-1)+1, r(q-1)+2, \ldots, rq-2 \,\}$$

$$U^2 = \bigcup_{q=1}^{d} \{\, r(q-1)+2, r(q-1)+3, \ldots, rq-1 \,\}$$

$$U^3 = \bigcup_{q=1}^{d} \{\, r(q-1)+3, r(q-1)+4, \ldots, rq \,\}\,.$$

In general, these three strata overlap; see Figure 4. The resulting DSGD algorithm is almost identical to the previous version. The A and b coefficients are distributed in the same manner, as are the components of x. Updates at distinct nodes can still be performed independently and in parallel. The main difference is that, within a node, updates to the various x components are no longer disjoint, so that the order in which updates are processed can no longer be arbitrary. However, we can still systematically sample each stratum element during a sub-epoch and systematically sample each stratum during an epoch, as described previously.

3.5 Multivariate Time Series

Our approach generalizes to multivariate time series, but there are more options for choosing strata in this setting. Supposing that each data value d_i is a vector of length $v > 1$, we now solve a system of the form $AX = B$, where $X, B \in \Re^{(m-1)\times v}$. The loss function is now given by $L(X) = \sum_{i=1}^{m-1} \sum_{j=1}^{v} L_{i,j}(X)$, where $L_{i,j}(X) = (A_{i\cdot}X_{\cdot j} - B_{i,j})^2$. A stratum might take the form $U^s \times \{\, 1, 2, \ldots, v \,\}$, where U^s is a set of i values as in previous sections, or the set $\{\, 1, 2, \ldots, v \,\}$ might be decomposed into a set of q overlapping or non-overlapping strata $Z^1, \ldots, Z^q$ and the overall strata might comprise all sets of the form $U^s \times Z^t$. The preferred stratification scheme may depend on the relative sizes of m and v. We will explore these issues in future work.

4. INTERPOLATION IN MAPREDUCE

The DSGD algorithm, particularly the alternative version of Section 3.4, is well-suited to MapReduce. In this section we describe our Hadoop implementation of DSGD and explain how our modification of the standard `InputFormat` operator leads to a map-only implementation with low network communication overheads. We also describe several other practical implementation details.

4.1 Hadoop Implementation

In Hadoop, data is physically divided into disjoint *blocks* that have a default size of 64 Mb and are distributed across networked machines. The data is also logically divided into *splits* by the Hadoop `InputFormat` operator, and each mapper is assigned exactly one split to process. A split roughly contains one block's worth of data; typically, splits are disjoint, and the last few bytes of data in a split may physically reside in a different block. During the map phase of a MapReduce job, the mapper first obtains, over the network, any missing bytes in its split, then reads and processes the data in the split, and finally writes its output to local files. In the reduce phase, these local files are shuffled over the network to a set of reducers, who complete the data processing; the vast majority of the network traffic is generated in the reduce phase.

We implement the DSGD algorithm as a sequence of map-only jobs to minimize network communication costs. Each abstract "processing node" referred to in Section 3 corresponds to a mapper in Hadoop. Communication of data between mappers is effected by

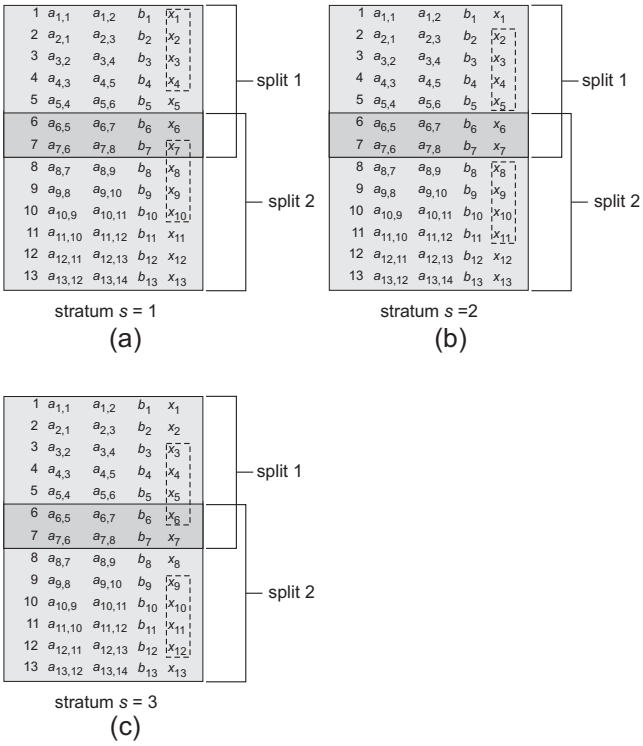


Figure 5: Split boundaries for the three DSGD strata.

configuring the `InputFormat` to allow adjacent splits to overlap each other by a couple of rows.

Specifically, consider the version of DSGD that uses the strata given in Figure 4, and suppose that the first three strata visited by the algorithm are stratum 1, stratum 2, and then stratum 3. The contents of the first two overlapping splits are illustrated in Figure 5; the dark shaded area indicates that rows 6 and 7 belong to both splits. The ith row contains the nonzero coefficients in the ith row of matrix A, along with b_i and the initial value of x_i. Note that, except for $i = 1$, we do not need to explicitly store the coefficient $a_{i,i}$ since it can be computed from the adjacent coefficients as $a_{i,i} = 2(a_{i,i-1}+a_{i,i+1})$. Also note that there is only one physical copy of each of these rows: row 7 and part of row 6 are located in the block that primarily belongs to mapper 2, and the other part of row 6 is located in the block that primarily belongs to mapper 1.

During processing of stratum 1—see Figure 5(a)—mapper 1 retrieves data for rows 6 and 7 over the network, and then updates the values of x_1–x_4 using the data in rows 1–5. Similarly, after obtaining its missing "boundary data", mapper 2 updates the values of x_7–x_{10} using the data in rows 6–11. In the figure, the dashed rectangles indicate the x_i values that are updated. (For this first iteration, the data fetched over the network is not actually used in the computations.) Because of the design of the strata, there are no read/write conflicts between mappers 1 and 2. The new version of split 1 is output by mapper 1 as part 1 and the updated split 2 is output by mapper 2 as part 2. The next map job processes stratum 2; see Figure 5(b). Mapper 1 updates x_2–x_5 based on rows 1–6 and mapper 2 updates x_8–x_{11} based on rows 7–12. When processing stratum 3 as in Figure 5(c), mapper 1 first obtains part of row 6 and row 7 over the network, before updating x_3–x_6 based on rows 2–7. Row 7 has previously been updated by mapper 2, and this modification is therefore "passed" by mapper 2 to mapper 1 during this initialization phase of the map-processing step. Similarly, after ob-

taining its upper-boundary data, mapper 2 updates x_9–x_{12} based on rows 8–13.

The key observation is that the only actual data transfer over a network occurs during the standard initialization phase of each map job, and less than $2M$ rows are transferred overall, where M is the number of mappers. Thus the network overhead for DSGD is minimal relative to algorithms that require reduce phases.

4.2 Other Practical Considerations

The stochastic approximation literature often works with step size sequences roughly of form $\epsilon_n = 1/n^\alpha$ with $\alpha \in (0.5, 1]$ as discussed previously. Theorem 3.1 guarantees asymptotic convergence for such choices. To achieve faster convergence over the finite number of steps that are actually executed, we use an adaptive method for choosing the step size sequence. We exploit the fact that—in contrast to SGD in general—we can determine the current loss after every epoch, and thus can check whether the loss has decreased or increased from the previous epoch. We then employ a heuristic called *bold driver*, which is often used for gradient descent. Starting from an initial step size ϵ_0, we (1) increase the step size by a small percentage (say, 5%) whenever the loss decreases over an epoch, and (2) drastically decrease the step size (say, by 50%) if the loss increases. Within each epoch, the step size remains fixed. Given a reasonable choice of ϵ_0, the bold driver method worked extremely well in our experiments. To pick ϵ_0, we leverage the fact that many compute nodes are available, replicating a small subsequence of the data to each node and trying different step sizes in parallel. Specifically, we try step sizes $1, 1/2, 1/4, \ldots, 1/2^{d-1}$; the step size that gives the best result is selected as ϵ_0. As long as the loss decreases, we repeat a variation of this process after every epoch, trying step sizes within a factor of $[1/2, 2]$ of the current step size. Eventually, the step size will become too large and the loss will increase. Intuitively, this happens when the iterate has moved closer to the global solution than to the local solution. At this point, we switch to the bold-driver method for the rest of the process.

5. EXPERIMENTS

We conducted a set of experiments to study the convergence rate, solution quality, and scalability of the two DSGD variants relative to alternative methods. Overall, the convergence rate and quality of the cubic-spline solution for DSGD was on par or better than the alternatives, and DSGD appeared to scale well.

5.1 Setup

We implemented DSGD with strata as in Figure 3 and the alternative DSGD algorithm (ADGSD) with strata as in Figure 4, along with the DGD and PCR algorithms as in Sections 3.1 and 1. We used two different implementations, one for initial in-memory experiments, using R, and one for scaling experiments over very large datasets, using Hadoop. The in-memory implementation targets datasets that are small enough to fit in main memory. The goal here was to obtain baseline results on convergence rate and spline quality, independent of the particularities of Hadoop. The second implementation, based on Hadoop, runs on a cluster of IBM x3650 M2 racks with 8 nodes interconnected with a high-speed 10Gbit network. Each node has 8 Intel Xeon E5560 cores running at 2.8GHz, 64GB of main memory and 8 SATA attached disks. One node was reserved as the namenode.

We used two time series of IBM stock prices since 1990 for our experiments on real-world data. The *Open* and *Volume* time series comprise daily opening prices and trading volumes. For scaling experiments, we generated massive synthetic datasets by repeatedly replicating the original time-series data. For each of the *Open* and *Volume* time series, we created two kinds of source-target time-series pairs, corresponding to two different spacings of source time ticks. We generated the first type of source time series by taking the first half of the original time series and then, from this subsequence, retaining only those data points corresponding to odd-numbered time ticks. We then constructed a corresponding target time series, denoted TargetTS1, by interpolating data points to replace the ones that were not retained. We also generated a second type of source time series by taking all of the original data and then retaining every fourth time tick. We again interpolated values for the non-retained data points to generate the corresponding target time series, denoted TargetTS2. For an original time series of length m, both types of source time series comprise $m/4$ data points. Thus the number of spline coefficients to compute is the same, allowing fair performance comparisons. (Note that the time to compute the spline constants is independent of the number of interpolated target points.)

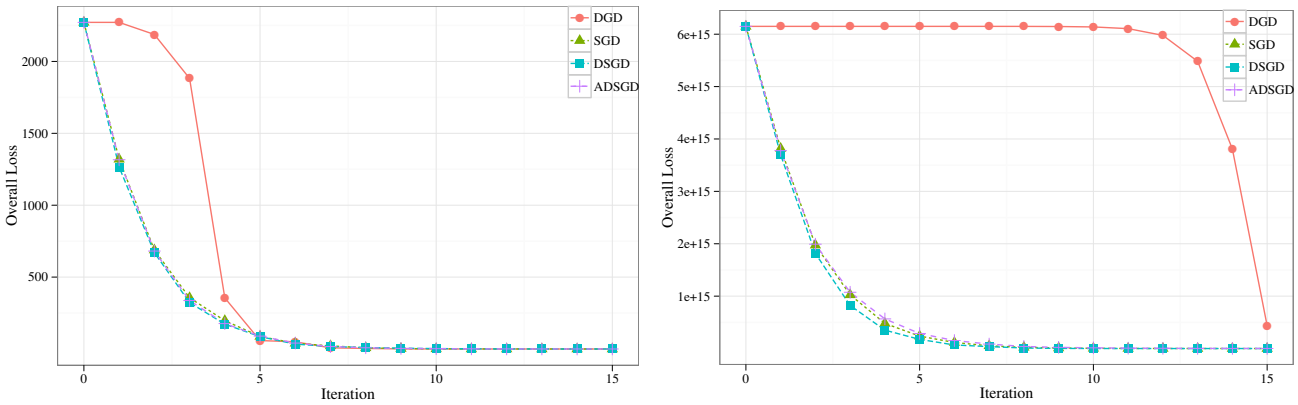
5.2 Convergence Rate and Spline Quality

We evaluated the relative convergence rates of the loss function for sequential (i.e., non-distributed) versions of SGD, DSGD, ADSGD, and DGD, using the R implementation. We also checked the post-convergence quality of the resulting spline interpolations relative to the non-iterative PCR and Thomas [31] algorithms. The latter algorithm is simply Gaussian elimination, tailored to the exploit the tridiagonal matrix structure; it requires only two scans over the data, but is inherently sequential—see [5, pp. 153–156].

We measured the convergence rate by recording the loss-function value—i.e., $\|Ax - b\|_2^2$—after each "iteration", where an iteration corresponds to a full scan over the base data. (Note that DSGD processes three strata per iteration, where each stratum contains roughly one third of the data, whereas ADSGD processes only one stratum per iteration, where each stratum contains virtually all of the base data.) We emphasize that for this comparison, all of the techniques require roughly the same time per iteration, because the processing time is dominated by the I/O required to perform the full scan. Representative results are depicted in Figure 6. Observe that SGD, DSGD and ADSGD typically converge faster than DGD. The *Open* time series is much "smoother" than the *Volume* time series, and we observe that the lack of smoothness significantly decreases the convergence rate of the deterministic DGD algorithm. In contrast, the lack of smoothness has almost no effect on the family of stochastic SGD, DSGD, and ADSGD algorithms. We used the bold-driver technique for adjusting the step-sizes as described in Section 4.2, and an interesting observation is that an initial value of 0.2 for the initial step size resulted in consistently good performance for all time series studied. This phenomenon is probably due to the high quality of the initial guess for $x^{(0)}$, as described in Section 3.1.

In contrast to the gradient descent algorithms, the PCR algorithm is not an iterative approximation algorithm and does not provide any useful intermediate results. Recall that PCR requires $\log_2 m$ scans over the data, where m is the number of source time ticks. In this experiment, PCR required 13 scans to complete. SGD, DSGD and ADSGD all converged after 10 iterations (i.e., after 10 scans).

We assessed the post-convergence quality of the spline interpolations for each source time series by computing interpolation errors. Our error metric was the normalized root mean square deviation (NRMSD) of the differences between the actual and interpolated values in the source and target time series. As a baseline for comparison, we also computed error values for the PCR and Thomas algorithms. As can be seen from Table 1, all of the iterative algorithms achieved essentially the same solution quality upon loss-function convergence as the baseline PCR and Thomas algorithms. The algorithms had lower error on the relatively smooth *Open* time series than on the



(a) *Open* time series (b) *Volume* time series

Figure 6: Convergence Comparison (using R)

Method	NRMSD(%)			
	TargetTS1		TargetTS2	
	Open	*Volume*	*Open*	*Volume*
DGD	0.901	6.28	0.971	5.71
SGD	0.901	6.28	0.971	5.71
DSGD	0.901	6.28	0.971	5.71
ADSGD	0.901	6.28	0.971	5.71
Thomas	0.901	6.28	0.971	5.71
PCR	0.901	6.28	0.971	5.71

Table 1: Quality of interpolating cubic spline

noisier *Volume* time series.

5.3 Scalability

We tested scalability using the Hadoop environment, focusing on the ADSGD and PCR algorithms because DGD runs out of memory and the Thomas algorithm cannot be parallelized. DSGD can be implemented in Hadoop, but not as efficiently as ADSGD, because the time to process one DSGD stratum (which contains roughly one third of the base data) roughly equals the time to process all of the data. This is because Hadoop does not offer an efficient way to avoid I/O of those tuples that DSGD does not touch. In essence, a Hadoop implementation of DSGD would require three scans over the base data in order to process the data once. One way to avoid this problem would be to first execute a MapReduce job that separates the input base data into three files, one for each stratum, and then execute one map job per file, for a total of three map jobs per iteration. This approach, however, would magnify the overheads of Hadoop task spawning and require a full MapReduce job for the preprocessing step. In contrast, ADSGD avoids this complication altogether, since one stratum corresponds roughly to all the base data (except at most two boundary tuples per split).

We found ADSGD to have good scalability properties on Hadoop. Figure 7(a) shows the wall-clock time per ADSGD iteration for varying numbers of concurrent tasks and varying data sizes—the latter measured in GBs as well as the number of equations in the $Ax = b$ system (which essentially equals the number of source time ticks). For comparison, we also depict the performance of PCR. Unlike ADSGD, the PCR algorithm requires a shuffle/reduce phase, which degrades performance. As can be seen, ADSGD is two to three times faster per iteration than PCR on average. Recall that the number of PCR scans over the data increases logarithmically with the number of source ticks; for example, it required 30 full scans for the 96GB (one billion time ticks) dataset. In contrast, ADSGD only required 10 iterations to converge, and hence 10 full scans. The speed-up of both approaches is roughly linear up to 28 concurrent tasks; after that the speed-up performance degrades. The reason is that the amount of data each task has to process becomes quite small, and the actual time to execute ADSGD or PCR is dominated by Hadoop overheads, primarily the time required to spawn tasks. These limits to speed-up are standard for Hadoop processing.

Figure 7(b) depicts the scale-out performance when the dataset size and the number of concurrent tasks are repeatedly doubled. The processing initially remains almost constant as the dataset size and number of concurrent tasks are each scaled up by a factor of two. ADSGD has much better scale-out performance than PCR, because PCR requires a reduce phase that shuffles essentially all of the data over the fixed-bandwidth network. As we move to larger datasets, PCR's performance is limited by the network bandwidth. At 8 billion source time ticks, for example, PCR shuffled 2.1TB of data over the network (about three times the base data, since every equation requires itself plus two others during the reduce phase), whereas ADSGD did not shuffle any data. Note that we used a rather fast and expensive network in our experiments; for cheaper commodity network components, the superior performance of ADSGD over PCR would be even more noticeable.

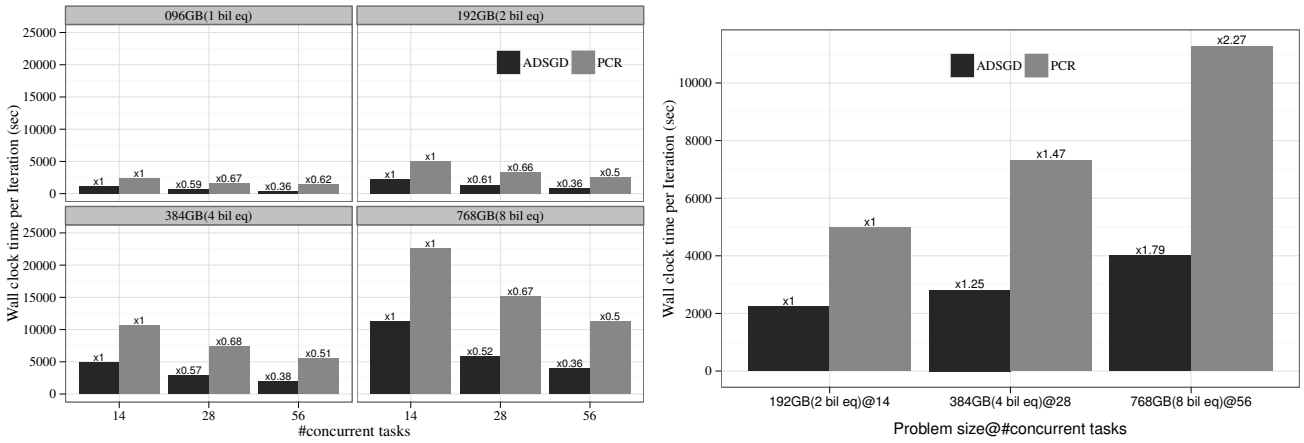
6. CONCLUSIONS

We have indicated the need for transforming massive time-series data at scale when composing large-scale, high-resolution simulation models, as in Splash, and in other data-integration settings. Our novel DSGD algorithm allows scalable cubic-spline interpolation of time series and is well suited to MapReduce environments; specifically, our map-only Hadoop implementation minimizes network communication. Our experiments show that DSGD can yield high-quality interpolations at scale, and has superior empirical performance with respect to the well regarded PCR algorithm.

In future work, we plan to conduct a comprehensive experimental comparison of our DSGD algorithm with other parallel algorithms for spline interpolation. We will also investigate the extension of our methods to multivariate settings and to other useful sparse linear systems of equations.

7. ACKNOWLEDGMENTS

We would like to thank Yinan Li, Wang-Chiew Tan, and Ignacio Terrizzano for their contributions to the Splash time-alignment com-



(a) Speed-Up (b) Scale-Out

Figure 7: Performance Comparison (using Hadoop)

ponent, as well as the other members of the Splash team: Cheryl Kieliszewski, Paul Maglio, and Patricia Selinger.

8. REFERENCES

[1] Apache Hadoop. http://hadoop.apache.org.

[2] S. Asmussen. *Applied Probability and Queues*. Springer, 2nd edition, 2003.

[3] R. H. Byrd, P. Lu, J. Nocedal, and C. Zhu. A limited memory algorithm for bound constrained optimization. *SIAM J. Sci. Comput.*, 16(5):1190–1208, 1995.

[4] I. Chiorean. Remarks on some parallel computation for spline recurrence formulas. *General Math.*, 16(4):35–40, 2008.

[5] S. D. Conte and C. de Boor. *Elementary Numerical Analysis: An Algorithmic Approach*. McGraw-Hill, 3rd edition, 1960.

[6] S. Das, Y. Sismanis, K. S. Beyer, R. Gemulla, P. J. Haas, and J. McPherson. Ricardo: Integrating R and Hadoop. In *SIGMOD*, pages 987–998, 2010.

[7] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. In *OSDI*, pages 137–150, 2004.

[8] R. Gemulla, P. J. Haas, E. Nijkamp, and Y. Sismanis. Large-scale matrix factorization with distributed stochastic gradient descent. Technical Report RJ10481, IBM Almaden Research Center, San Jose, CA, 2011. Available at www.almaden.ibm.com/cs/people/peterh/dsgdTechRep.pdf.

[9] R. Gemulla, E. Nijkamp, P. J. Haas, and Y. Sismanis. Large-scale matrix factorization with distributed stochastic gradient descent. In *KDD*, pages 69–77, 2011.

[10] H. Godfray, J. Pretty, S. Thomas, E. Warham, and J. Beddington. Linking policy on climate and food. *Science*, 331(6020):1013–1014, 2011.

[11] F. G. Gustavson and A. Gupta. A new parallel algorithm for tridiagonal symmetric positive definite systems of equations. In *Proc. PARA '96*, pages 341–349, 1996.

[12] L. M. Haas, M. A. Hernández, H. Ho, L. Popa, and M. Roth. Clio grows up: From research prototype to industrial tool. In *SIGMOD Conference*, pages 805–810, 2005.

[13] P. J. Haas, P. P. Maglio, P. G. Selinger, and W. C. Tan. Data is dead... without what-if models. *PVLDB*, 4(12):1486–1489, 2011.

[14] K. B. Hall, S. Gilpin, and G. Mann. MapReduce/Bigtable for distributed optimization. In *NIPS LCCC Workshop*, 2010.

[15] T. Hey and A. E. Trefethen. The UK e-science core programme and the grid. *Future Generation Comp. Syst.*, 18(8):1017–1031, 2002.

[16] R. W. Hockney and C. R. Jesshope. *Parallel Computers*. Adam Hilger, Bristol, U.K., 1981.

[17] B. Howe and D. Maier. Algebraic manipulation of scientific datasets. *VLDB J.*, 14(4):397–416, 2005.

[18] T. T. Huang, A. Drewnowski, S. K. Kumanyika, and T. A. Glass. A systems-oriented multilevel framework for addressing obesity in the 21st century. *Preventing Chronic Disease*, 6(3), 2009.

[19] Institute of Medicine. *For the Public's Health: The Role of Measurement in Action and Accountability*. The National Academies Press, 2010.

[20] JAQL: Query Language for JavaScript Object Notation (JSON). http://code.google.com/p/jaql, 2009.

[21] H. J. Kushner and G. Yin. *Stochastic Approximation and Recursive Algorithms and Applications*. Springer, 2nd edition, 2003.

[22] K. Lange. *Numerical Analysis for Statisticians*. Springer, second edition, 2010.

[23] J.-L. Larriba-Pey, J. J. Navarro, and A. Jorba. Vector and parallel interpolation by natural cubic splines and B-splines. In *Proc. PDP '96*, pages 385–392, 1996.

[24] Y. Li, P. J. Haas, W.-C. Tan, and I. Terrizzano. Data exchange between simulation models: Getting the time right. Technical report, IBM Research – Almaden, San Jose, CA, 2012. Forthcoming.

[25] A. V. Malevsky and S. J. Thomas. Parallel algorithms for semi-lagrangian advection. *Intl. J. Numerical Methods in Fluids*, 24(4):455–473, 1997.

[26] G. Mann, R. McDonald, M. Mohri, N. Silberman, and D. Walker. Efficient large-scale distributed training of conditional maximum entropy models. In *NIPS*, pages 1231–1239. 2009.

[27] H. Robbins and S. Monro. A stochastic approximation method. *Ann. Math. Statist.*, 22:400–407, 1951.

[28] SAS Institute. *SAS/ETS 9.2 User's Guide*. SAS Publishing, Cary, NC, 2008.

[29] X.-H. Sun. A scalable parallel algorithm for periodic symmetric Toeplitz tridiagonal systems. *Intl. J. Comput. Res.*, 10(1):89–97, 2001.

[30] W. C. Tan, P. J. Haas, R. L. Mak, C. A. Kieliszewski, P. G. Selinger, P. P. Maglio, S. Glissmann, M. Cefkin, and Y. Li. Splash: a platform for analysis and simulation of health. In *ACM Intl. Health Informatics Symp. (IHI)*, pages 543–552, 2012.

[31] L. Thomas. Elliptic problems in linear differential equations over a network. Technical report, Watson Sci. Comput. Lab., Columbia University, New York, 1949.

[32] R. L. Tweedie and S. Meyn. *Markov Chains and Stochastic Stability*. Cambridge University Press, 2nd edition, 2009.

[33] Y. Zhang, J. Cohen, and J. D. Owens. Fast tridiagonal solvers on the GPU. In *Proc. ACM PPoPP*, pages 127–136, 2010.