Exact Gaussian Process Regression with Distributed Computations

Duc-Trung Nguyen
EURECOM
Campus Sophia Tech, France
duc-trung.nguyen@eurecom.fr

Maurizio Filippone
EURECOM
Campus Sophia Tech, France
Maurizio.Filippone@eurecom.fr

Pietro Michiardi
EURECOM
Campus Sophia Tech, France
Pietro.Michiardi@eurecom.fr

ABSTRACT

Gaussian Processes (GPs) are powerful non-parametric Bayesian models for function estimation, but suffer from high complexity in terms of both computation and storage. To address such issues, approximation methods have flourished in the literature, including model approximations and approximate inference. However, these methods often sacrifice accuracy for scalability.

In this work, we present the design and evaluation of a distributed method for exact GP inference, that achieves true model parallelism using simple, high-level distributed computing frameworks. Our experiments show that exact inference at scale is not only feasible, but it also brings substantial benefits in terms of low error rates and accurate quantification of uncertainty.

CCS CONCEPTS

• **Computing methodologies** → **Gaussian processes**; **MapReduce algorithms**;

KEYWORDS

Regression, Matrix Factorization, Distributed computing

1 INTRODUCTION

The availability of large amounts of training data, together with the advent of cloud computing and high-level parallel and distributed programming models, have determined unprecedented advances in building large-scale statistical models to solve inference tasks. In this work we are interested in probabilistic machine learning, whereby Bayesian inference offers a principled approach to model complex phenomena allowing accurate quantification of uncertainty.

In particular, we focus on Gaussian processes (GPs) [43], which are powerful non-parametric Bayesian models for function estimation, that do not impose any explicit parametric form. GPs are robust to noisy data, resist overfitting because they are naturally regularized, and produce uncertainty estimations. In applications where *"knowing when the model doesn't know"* is of vital importance (e.g., life and environmental sciences, and autonomous systems), GPs have become a prime choice modeling approach.

SAC '19, April 8–12, 2019, Limassol, Cyprus

ACM ISBN 978-1-4503-5933-7/19/04…$15.00
https://doi.org/10.1145/3297280.3297409

Exact GP inference is inherently computationally demanding, which is a limiting factor that in the past has restricted GP applications to only very small training set sizes, in the order of a few hundreds data points. The main computational bottleneck for exact GP inference is that it requires $O(n^3)$ time complexity and $O(n^2)$ space complexity, with n being the number of training samples.

To overcome this limitation, the machine learning community focused on methods to "sparsify" GPs and to approximate intractable computations. Some work have focused on methods to alleviate the computational burden related to the kernel matrix by using inducing points [15, 22, 41, 48] or structured random features [50]. These methods allow one to transform expensive computations from the entire training data, such as the covariance and inverse covariance matrix calculations, to a small set of representative input points. Other works proposed methods to allow tractable and computationally efficient computations: for example, [7, 23] have shown that training approximate GPs can proceed using small batches of data, thus allowing for the design of embarrassingly parallel algorithms that are also referred to as the "data parallelism" approach [6].

In summary, practical approaches to scale GPs to large training sets resort to various forms of approximations, trading lower accuracy and confidence for faster training times. The problem we address in this work is how to design distributed algorithms that allow *exact inference at scale*. A naive approach of scaling up computational resources to meet the requirements of GP inference with large training sets is not practical: 1) vertical scalability is super linear in terms of costs, 2) the availability of cloud computing makes it both practical and relatively cheap to revert to scale-out solutions, 3) the advent of distributed computing frameworks, which expose simple programming models, suggest that achieving "model parallelism" is conceptually possible. The endeavor of this work is to design and implement a full-fledged distributed GP inference method, which does not resort to any approximate computation.

The challenges we address in this work are as follows. We design distributed, in-memory data representations to store and manipulate large and **dense** covariance matrices, as discussed in section 4. We design efficient, distributed linear algebra primitives, with focus on those required for GP regression (GPR), with access to a high-level only programming model, based on a shared-nothing architecture, as presented in section 5. We combine such primitives to design an exact, distributed GPR algorithm, that scales with the number of training points and computational resources.

In summary, our results indicate that exact GP inference at scale brings several benefits, when compared to approximate solutions, in terms of low error rates and accurate uncertainty quantification, which is important for several application domains.

2 RELATED WORK

This section overviews a fraction of recent advances into the design of scalable Gaussian Process regression, and covers a selection of parallel and distributed linear algebra primitives. A brief introduction to Gaussian Process regression is discussed in section 3.

It is well-known that the primary computational bottleneck for GP regression is the inversion of the kernel matrix, which costs $O(n^3)$ and $O(n^2)$ time and space complexity, respectively. To address this challenge, several approximation techniques have been proposed, that we can group in two categories: local approximations and sparse approximations (including low-rank approximations).

The idea of local approximation is that of partitioning the input domain into a set of local and independent regions. Each region contains a small amount of training points, for which independent GP regression can be computed efficiently [9], since the sample covariance matrices are block diagonal. However, neighboring local regions and their respective GP models can produce conflicting predictions, an issue that has been largely studied in various ways [11, 37, 38, 45].

The idea of sparse approximations, instead, are based on the construction of a set of $m < n$ variables called *inducing variables*, which are representative of the original data [41]. As such, the complexity of these methods is reduced to $O(nm^2)$. Several strategies to compute an approximate posterior distribution of the "full" GP has been proposed. Two popular methods are the Fully Independent Training Conditional (FITC) [41] and the Variational Free Energy (VFE) [48]. In particular, VFE approximates the true posterior by finding a surrogate (and simple) posterior which minimizes the distance to the true posterior. The inducing points are defined as variational parameters which are jointly optimized with the model hyper-parameters, by minimizing the Kullbak-Leibler divergence between the variational distribution and the exact posterior distribution over the latent function values. VFE has inspired many other works, including scalable methods that use mini-batches of data, such as stochastic variational inference [23], variational features [22], structure exploiting inference[50, 51] and distributed learning [15].

Alternative approaches cope with scalability bottlenecks without relying on inducing variables. In [11], Deisenroth *et al.* introduce the robust Bayesian Committee Machine (rBCM), combining Generalized Product of GP Experts and Bayesian Committee Machine (BCM). It uses a hierarchical computation tree whose leaf nodes are named GP experts. The data is divided into small regions and assigned to GP Experts, which perform computations in parallel and independently. The results are recombined by theirs parents recursively. In [8], Zhenwen Dai *et al.* exploit the computational structure of the sparse Gaussian process formulation, to distribute computation across several worker machines, which exploit GPUs to speed-up the inference process. In [39] Hao Peng *et al.* propose to use the parameter server architecture [31], and suggest an asynchronous approach to gradient computation and model update. They introduce a delay limit parameter to cope with the instabilities due to an asynchronous gradient update mechanism.

Although the methods above can handle massively large data, they all rely on some forms of approximations, which imply trading lower training times for higher error rates, as well as larger variance. Our approach is different from such works, because we explicitly avoid approximations, aiming at exact inference at scale.

Exact GP regression for large training data requires scalable linear algebra primitives, whose aim is to distribute matrix operations across several machines. The design of efficient algorithms to implement matrix factorization and inversion has been the subject of several studies. Noteworthy examples are well-known libraries such as LaPACK [1, 2], ScaLaPACK [4, 5], BLAS [28], BLAS level 2 [12] and BLAS level 3 [13]. These libraries operate at low-level and are highly optimized for multi-core processors and are generally integrated within higher-level languages.

The design of parallel algorithms for matrix factorization has also received a great deal of attention by the research community. In particular, prior works focus on specialized hardware such as orthogonal processors [3, 27, 47], on compute grids [19], on shared multi-processor systems [16] on shared memory systems [54] and on database systems [21].

Matrix factorization algorithms designed to execute on distributed systems have also flourished in the past. For example, the works presented in [34, 42] suggest a data-flow programming model that bears many similarities to modern systems such as the one we use in this work, i.e., Apache Spark. In the realm of distributed systems, there exist several variations of factorization algorithms, that target e.g. distributed memory systems [20, 36], as well as asynchronous programming models [26, 35].

Recent works that are closest to our approach present a Map Reduce algorithm to proceed with matrix inversion [52], Apache Spark variants [29, 32] and an algorithm based on the DryadLinq programming model [40]. These approaches differ from our algorithms as they tackle full matrix inversion, whereas we focus on efficient matrix factorization. The work in [53] introduces data structures and basic matrix operations for Apache Spark, which is the system we use in our work. SPIN [33] proposes a distributed version of Strassen's matrix inversion algorithm for Apache Spark: this approach, however, suffers from scalability issues due to the requirement to introduce data redundancy.

Other works, such as MadLINQ [40], require the design of new systems, supporting asynchronous point-to-point communication primitives. Along the same lines, Sparkler is a modified version of Apache Spark that can factorize large low-rank matrices [29]. It uses a distributed hash map to keep track the locations of data and supports point-to-point data transfer to avoid the coordination overheads. In this case, the price to pay for asynchronous communication primitives is a decreased tolerance to failures, thus requiring the implementation of a checkpointing mechanism, which introduces overheads. Along the same lines of the above works that support asynchronous operations, noteworthy examples are MatrixMap [25], YinMem[24], which focus on the optimization of the abstract representation of the computational graphs behind matrix operations.

In summary, what sets apart our work from the literature is that we propose a general, distributed design pattern enabling a wide range of linear algebra operations, that we use for exact GP inference.

3 DISTRIBUTED GP REGRESSION

We now describe our approach to exact Gaussian Process (GP) regression, which we cast as a distributed algorithm operating on

distributed data structures. First, we present the basics of GP inference for regression problems, and emphasize the computational challenges and scalability bottlenecks of GP prediction. Then we outline the structure of our approach, and dedicate subsequent sections to delve into the details of its components.

3.1 Classical GP regression

We briefly review Gaussian processes (GPs) [44], and the computational requirements for predictions and kernel learning. Let's assume a dataset $\mathcal{D}$ of n input vectors $X = [\mathbf{x}_1, \cdots, \mathbf{x}_n]$, each of dimension D, corresponding to a $n \times 1$ vector of target variables $\mathbf{y} = [y(\mathbf{x}_1), \cdots, y(\mathbf{x}_n)]^\top$. If $f(\mathbf{x}) \sim \mathcal{GP}(\mu, k_\theta)$, then any collection of function values f has a joint Gaussian distribution,

$$\mathbf{f} = f(X) = [f(\mathbf{x}_1), \cdots, f(\mathbf{x}_n)]^\top \sim \mathcal{N}(\mu_X, K_{X,X}), \tag{1}$$

with mean vector and covariance matrix defined by the mean vector and covariance function of the Gaussian process: $(\mu_X)_i = \mu(x_i)$, and $(K_{X,X})_{ij} = k_\theta(\mathbf{x}_i, \mathbf{x}_j)$. The covariance function k_θ is parametrized by θ. Assuming additive Gaussian noise, $y(\mathbf{x})|f(\mathbf{x}) \sim \mathcal{N}(y(\mathbf{x}); f(\mathbf{x}), \sigma^2)$, then the predictive distribution of the GP evaluated at the n_* test points indexed by X_*, is given by

$$\mathbf{f}_*|X_*, X, \mathbf{y}, \theta, \sigma^2 \sim \mathcal{N}(\mathbb{E}[\mathbf{f}_*], \mathrm{cov}(\mathbf{f}_*)), \tag{2}$$

$$\mathbb{E}[\mathbf{f}_*] = \mu_{X_*} + K_{X_*,X}[K_{X,X} + \sigma^2 I]^{-1}\mathbf{y}, \tag{3}$$

$$\mathrm{cov}(\mathbf{f}_*) = K_{X_*,X_*} - K_{X_*,X}[K_{X,X} + \sigma^2 I]^{-1}K_{X,X_*}. \tag{4}$$

$K_{X_*,X}$ represents the $n_* \times n$ matrix of covariances between the GP evaluated at X_* and X, and all other covariance matrices follow the same notational conventions. μ_{X_*} is the $n_* \times 1$ mean vector, and $K_{X,X}$ is the $n \times n$ covariance matrix evaluated at the training inputs X. All covariance matrices depend on the kernel parameters θ.

Algorithm 1: Prediction and log marginal likelihood for Gaussian process regression.

Data: X (training data), $\mathbf{y}$ (targets), $k()$ (covariance function), σ^2 (noise level)

1 $L := \mathrm{cholesky}(K_{X,X} + \sigma I)$
2 $\alpha := L^\top \backslash (L \backslash \mathbf{y})$
3 $\mathbb{E}[\mathbf{f}_*] := K_{X_*,X}\alpha$
4 $\mathbf{v} := L \backslash K_{X,X_*}$
5 $\mathrm{cov}(\mathbf{f}_*) := K_{X_*,X_*} - \mathbf{v}^\top\mathbf{v}$
6 $\log p(\mathbf{y}|X) := -\frac{1}{2}\mathbf{y}^\top\alpha - \sum_i \log L_{ii} - \frac{n}{2}\log 2\pi$

Result: $\mathbb{E}[\mathbf{f}_*]$ (predictive mean), $\mathrm{cov}(\mathbf{f}_*)$ (predictive covariance), $\log p(\mathbf{y}|X)$ (log marginal likelihood)

A practical implementation of GP regression (GPR) is shown in Algorithm 1 [44]. The algorithm uses Cholesky decomposition, instead of directly inverting the covariance plus noise matrix, since it is faster and numerically more stable. The algorithm returns the predictive mean and variance for noise free test data: to compute the predictive distribution for noisy test data y_*, we simply add the noise variance σ^2 to the predictive variance of $\mathbf{f}_*$. Model selection through hyper-parameter optimization is discussed in detail in section 7.

3.2 Computational challenges and scalability bottlenecks

The main computational bottleneck for GP inference is solving the linear system $[K_{X,X} + \sigma^2 I]^{-1}\mathbf{y}$, which involves expensive operations related to matrix inversion. The standard procedure for GPR is to compute the Cholesky decomposition of the $n \times n$ kernel plus noise matrix, which requires $O(n^3)$ operations and $O(n^2)$ storage. Afterwards, the predictive mean and variance of the GP cost respectively $O(n)$ and $O(n^2)$ per test point x_*.

In this work, we aim at a *distributed, exact GPR implementation* involving p machines which, in the ideal case, yields $O(\frac{n^3}{p})$ complexity. In a distributed setting, this limit is hardly attainable because the overhead associated to message exchange can dominate computational costs. Additionally, for the training set sizes we are interested in, also the simple construction and storage of the covariance matrix $K_{X,X}$ is challenging, as discussed in section 4.

With this context in mind, the bottlenecks we address in this work are as follows. Since our approach targets large scale training data, we consider **I/O bottlenecks** that are inherent to any *iterative* algorithm that repeatedly accesses large quantities of data: we design new distributed data structures and lookup primitives to quickly iterate over input data. Due to the rigid communication patterns imposed by the programming model we use, we address **network bottlenecks** that affect any distributed algorithm by promoting local, independent operations to be concurrently executed on each machine, to reduce the message passing overheads. Finally, we address the **computational bottlenecks** by a careful algorithm design, which favors the adoption of efficient, low-level numerical libraries.

3.3 Overview of our approach

We begin with a brief description of the programming model and runtime system we assume for our work. We use a distributed, shared-nothing architecture, whereby a collection of machines operate on distributed data structures in parallel, in a synchronous way. The application of a series of high-order functions is handled by a *driver* machine, and materializes as a directed acyclic graph, describing the various *stages* of data transformations: each stage is broken into parallel tasks, that execute the same function on different pieces of data. Stage boundaries require synchronization akin to the "bulk-synchronous processing", by Valiant *et. al.*, in [49]. The driver machine coordinates message broadcast, which is a limiting factor that imposes a strict communication structure to our algorithms.

The main steps of our approach are as follows. Training data reside on a distributed file system. It must be read, and transformed to build the distributed kernel matrix $K_{X,X}$, which we keep in RAM for efficiency, stored in a custom data structure, as described in section 4. Then, the algorithm proceeds with an *in place*, distributed Cholesky factorization, which we discuss in section 5. At this point, we solve the linear system exemplified by Line 2 of Algorithm 1. We achieve this using the factorized matrix, and distributed forward and backward substitution method, which we describe in section 6. Similarly, we solve the linear system exemplified by Line 4 of Algorithm 1, and output the predictive mean and covariance. Finally, we approach model selection by an original method to estimate hyper-parameters, as discussed in section 7.

4 DISTRIBUTED DATA STRUCTURES

To proceed with distributed GPR we need to build and store the covariance matrix K. To do so, we design an efficient, in-memory matrix representation that supports the access patterns of our algorithms, in particular for the distributed Cholesky decomposition.

Irrespectively of the algorithmic variant we may chose for the Cholesky decomposition, there are three kinds of data access patterns we are interested in: *i)* determine if a machine holds a matrix region susceptible of being either factorized or updated; *ii)* access a given matrix region to compute its factorization; *iii)* access the *range* of matrix region(s) to update. To factorize a given matrix K, algorithmic Cholesky variants operate on different matrix elements: rows, columns or blocks. We call such elements *storage units*, which we represent as key/value pairs: thus, $\sigma_i =< k_i, v >$. The *key* of a storage unit is a unique identifier, which carries positional information of the storage unit within the matrix; the *value* of a storage unit is the matrix element itself, e.g. a matrix row.

Storage units are randomly mapped to machines: each holds multiple storage units that we organize in a tree-like structure. Hence, each machine m holds a handler to a *storage tree* T_m, which is designed for fast lookup operations. The storage tree is an enhanced form of self-balancing AVL binary tree [17], whose nodes contain individual storage units. Its root stores, unlike other nodes, the minimum and maximum key identifiers, which we call $minKey_m$ and $maxKey_m$, of the storage units within the tree. A node with key i (which we call node i), in addition to left and right pointers to descendants, stores a pointer, which we call the *next* pointer, to node j that is the closest to i, so that there is no node j' such that $i < j' < j$.

Storage trees are built in a pre-processing step, and only require insert operations. If a machine stores k storage units, then its storage tree height is $\log k$. Insert operations, in addition to finding the right position for the node, also require to update $minKey$, $maxKey$, and the *next* pointer: they thus cost two $O(\log k)$ lookup operations.

Next, we develop the membership and lookup procedures supporting the algorithms we use in GPR. *Membership queries* of the form "$\sigma_i \in T_m$" do not require lookup operations, but only a compare operation with $minKey_m$ and $maxKey_m$ meta-data associated to each storage tree. The first lookup procedure is called `pivotLookup(Key i)`, and it is used to find a "pivot" region i, potentially stored on a machine m: it involves a simple binary search algorithm on the storage tree T_m, and thus costs $O(\log(k))$ for a given storage tree holding k storage units. The second procedure is called `rangeLookup(Key l)`: it is a variant of a binary search algorithm, that uses the *next* pointer as a short-cut mechanism to speed-up traversal of very deep trees. Our goal is to support *range queries* of the form "$\forall \sigma_i, i \in [l, r]$", which are frequent in Cholesky decomposition algorithms. The procedure returns only the σ with the smallest key satisfying the lookup range: that is, $\sigma =< k_i, v >$, with $i \geq l$ and $i < j$, $\forall j \neq i \in [l, r]$. Subsequent storage units within the lookup range are accessed with the *next* pointer.

Computing the covariance matrix. Building the covariance matrix $K_{X,X}$ can conceptually be viewed as a simple Cartesian product among the training inputs X: for all possible input pairs x_i, x_j we apply the kernel function $k(x_i, x_j)$. This seemingly trivial operation is extremely costly if implemented naively on a distributed system.

In this work we design a lightweight mechanism which builds the covariance matrix incrementally. The original dataset D is divided into multiple subsets D_i such that the each one can be stored on a single machine. In iteration i, we only compute the covariance of datapoints in D_i and D. The final covariance matrix results from the union of each of its components and its storage in memory according to our storage unit format.

5 DISTRIBUTED CHOLESKY FACTORIZATION

The Cholesky algorithm is one of the most popular methods to factorize matrices. Given a symmetric, positive-definite real matrix $A \in \mathbb{R}^{n \times n}$, the Cholesky decomposition computes a lower triangular matrix L such that $A = LL^\top$, where matrix L has real and positive diagonal entries.

Designing a distributed Cholesky matrix factorization algorithm requires a careful approach to efficiently exploit parallelism. In this work, we focus on the Block Partitioned Cholesky (BPC) variant.

In the remainder of this section, we assume a generic matrix A to be represented as discussed in section 4. Each machine holds a fraction of the storage units the matrix is divided into: for a given storage unit, we assume its elements to be *sorted* according to their position in the original matrix. Since we focus on the design of a distributed Block-partitioned Cholesky algorithm, we use block matrices as storage units. Figure 1 illustrates how matrix A is subdivided into $b \times b$ block matrices, and how position identifiers are assigned to storage units.

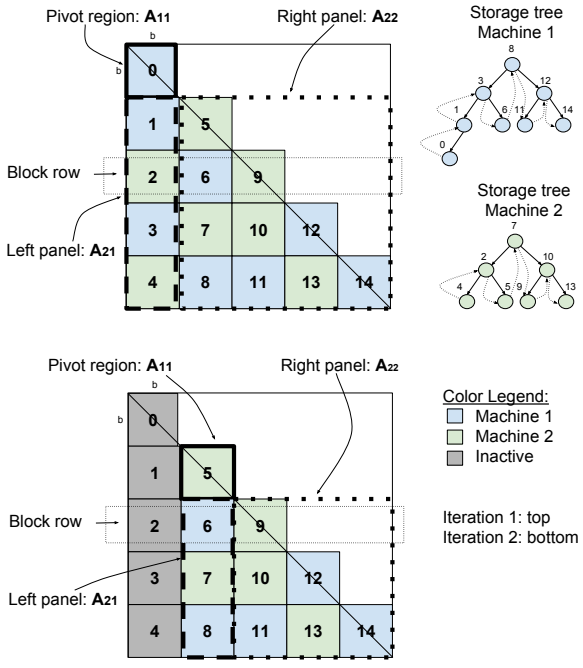

Figure 1: Matrix representation for the BPC algorithm.

Block-partitioned Cholesky operates as follows: at each iteration, it selects a *pivot region* to factorize (in this case, a block matrix), uses

it to update and factorize regions of the matrix we call the *left-panel*, and finally uses the factorized left-panel to update the remaining matrix regions that we call the *right-panel*. The next iteration considers the updated right-panel as the matrix to decompose, and proceeds recursively as illustrated in figure 1.

Algorithm 2: Distributed Block-partitioned Cholesky.

Data: $A \in \mathbb{R}^{n \times n}$, M number of block-rows in A, b block size

1 **Procedure** *Block-partitioned Cholesky(A)*
  /* Executed by the driver */
2 $pid = 0$ /* Current pivot region identifier */
3 $i = 0$
4 **while** $i < M$ **do**
    /* Number of active block rows */
5    $r = M - i - 1$
    /* Parallel lookup on all machines */
6    $A_{11}^{(i)}$ = **pivotLookup** (pid)
    /* Collect region in a single machine */
    /* Region factorization */
    /* LAPAC factorization routine */
7    $L_{11}^{(i)}$ = **dpotrf** $(A_{11}^{(i)})$
    /* One to many communication */
8   **broadcast** $(L_{11}^{(i)})$
9   **if** $i < M - 1$ **then**
      /* Parallel update: left panel */
10      $L_{21}^{(i)}$ = **updateLPanel** (pid, r)
      /* One to many communication */
11   **broadcast** (L_{21})
      /* Parallel update: right panel */
12      $\tilde{A}_{22}^{(i)}$ = **updateRPanel** (pid, r)
13      $pid = pid + r$

Algorithm 2 outlines the main procedure implementing the distributed Block-partitioned Cholesky algorithm. The algorithm is governed by a single parameter, b, which determines the size of a sub-matrix block: b is automatically determined based on the memory constraints of a single machine.

The main procedure is called by the driver machine, that holds information about the total number M of block rows in matrix A, the number r of block rows that are susceptible of being accessed in the current iteration, the identifier *pid* of the pivot region in an iteration, and the current iteration i.

The first operation is a distributed lookup to identify the machine holding the current *pid*, which we call $A_{11}^{(i)}$: this is the current pivot region at iteration i. The pivot region is factorized in a single machine to obtain $L_{11}^{(i)}$, using an efficient serial implementation. Then, the driver distributes $L_{11}^{(i)}$ to all machines.

Next, the algorithm computes the factorization of the left-panel, $A_{21}^{(i)}$. This operation is executed in parallel only by those machines holding relevant storage units to factorize: hence, we use our range lookup, to find storage units to factorize. This step produces the factorized left-panel $L_{21}^{(i)} = (A_{21}^{(i)})^{-1}(L_{11}^{(i)})^{\top}$. Since $L_{21}^{(i)}$ is required to update the right-panel, it needs to be broadcast to all machines.

Finally, the algorithm updates the right-panel $\tilde{A}_{22}^{(i)}$, which requires a similar range lookup as described for the left-panel, and an efficient update procedure to avoid materializing very large matrices, resulting from a naive computation of $\tilde{A}_{22}^{(i)} = A_{22}^{(i)} - L_{21}^{(i)}(L_{21}^{(i)})^{\top}$.

Algorithm 3: Update left panel procedure.

Data: pivot identifier pid, r: active block-rows

/* Note: $L_{11}^{(i)}$ available on all machines */
1 **Procedure** *UpdateLPanel*(pid, r)
  /* Executed in parallel by each machine */
  /* Find the first block in the update range */
2 B = **rangeLookup** (pid)
  /* Scan relevant blocks using next pointer */
3 **while** $B.next \neq NULL \wedge B.key \leq pid + r - 1$ **do**
    /* Update using matrix multiplication */
4    $B = BL_{11}^{(i)}$
    /* Select next block */
5    $B = B$.next

Algorithm 4: Update right panel procedure.

Data: pivot identifier pid, r: active block-rows

/* Note: $L_{21}^{(i)}$ available on all machines */
1 **Procedure** *UpdateRPanel*$(pid, r - 1)$
  /* Executed in parallel by each machine */
  /* Find the first block in the update range */
2 B = **rangeLookup** $(pid + r - 1)$
  /* Scan relevant blocks using next pointer */
3 **while** $B.next \neq NULL$ **do**
4    $\tilde{B} = A_{22} - L_{21}^{(i)}(L_{21}^{(i)})^{\top}$
    /* Select next block */
5    $B = B$.next

Algorithm 3 provides the details of procedure `UpdateLPanel`. The gist of the procedure is to find relevant sub-matrix blocks to update. To do so, we use our efficient storage tree to find the first (in terms of key identifier) storage unit within the update range; then, subsequent blocks are obtained using the next pointer. The update operation consists in a simple matrix multiplication, which we carry out using BLAS level-3 primitives [13].

Algorithm 4 provides the details of procedure `UpdateRPanel`. Each machine determines its state, active or not, depending on the range lookup function: once the first storage unit within range is found, we use the next pointer to update all relevant sub-matrix blocks. At the matrix A level, the update procedure requires to compute $\tilde{A}^{(i)_{22}} = A_{22}^{(i)} - L_{21}^{(i)}(L_{21}^{(i)})^{\top}$. Working at the block level, the update operation avoids materializing the large matrix $L_{21}^{(i)}(L_{21}^{(i)})^{\top}$. Each machine holding a block of matrix $\tilde{A}_{22}^{(i)}$, computes it using the corresponding blocks of matrices $A_{22}^{(i)}$ and $L_{21}^{(i)}(L_{21}^{(i)})^{\top}$.

6 DISTRIBUTED SYSTEMS OF LINEAR EQUATIONS

The distributed BPC algorithm proceeds with an *in place* matrix factorization, and produces the output exemplified by Line 1 of Algorithm 1. At this point, we solve the system of linear equations exemplified by Line 2 and Line 4 of Algorithm 1 using distributed forward and backward substitution algorithms.

The two algorithms, which we describe next, use the same "design pattern" we conceived for the BPC algorithm. They are iterative and proceed by solving (locally) a fraction of the linear system, and by updating (in parallel) the vectors that appear in system to solve.

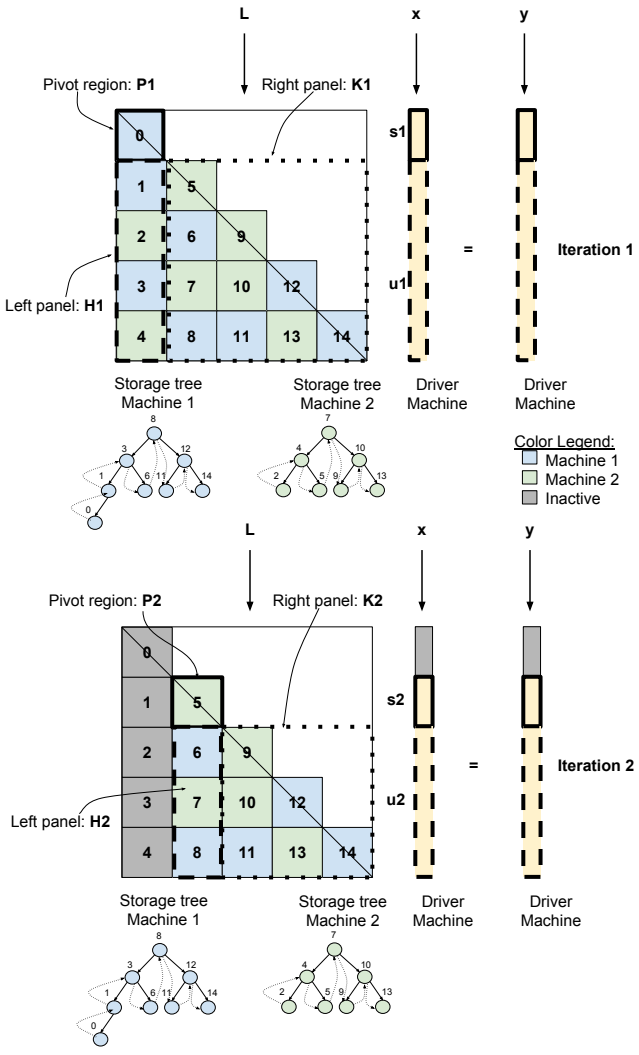

Figure 2: Example of distributed forward substitution and matrix/vector representations. Grayed out areas are "inactive", as they are not involved in computations.

Distributed forward substitution. The forward substitution algorithm solves the problem $L\mathbf{x} = \mathbf{y}$, where L is a $n \times n$ lower triangular matrix, $\mathbf{y}$ is a $n \times 1$ vector. Figure 2 illustrates how matrix L and vectors $\mathbf{x}$, $\mathbf{y}$ are represented, where they are stored, and can be used to describe our distributed forward substitution algorithm. Note that matrix L is distributed across all machines, according to the data structure defined in section 4. Instead, vectors x and y are stored on the driver machine: as also pointed out in [53], it is reasonable to assume such vectors to fit into the memory of a single machine.

Algorithm 5: Distributed forward substitution.

Data: $L \in \mathbb{R}^{n \times n}$ lower triangular matrix, b block size, n rows in L, $\mathbf{y}$ constant vector, $\mathbf{x}$ unknowns vector

1 **Procedure** *Forward Substitution(L, b, n,* $\mathbf{y}$*,* $\mathbf{x}$*)*
2 `/* Executed by the driver */`
2 $i = 0$
3 **while** $i < n/b$ **do**
`/* Define "solve" and "update" regions */`
4 $s^{(i)} = [ib, \min((i+1)b, n)$
5 $u^{(i)} = [\min((i+1)b + 1, n), n]$
6 $\mathbf{x}_s^{(i)} = \mathbf{x}[s^{(i)}]$; $\mathbf{x}_u^{(i)} = \mathbf{x}[u^{(i)}]$
7 $\mathbf{y}_s^{(i)} = \mathbf{y}[s^{(i)}]$; $\mathbf{y}_u^{(i)} = \mathbf{y}[u^{(i)}]$
`/* One to many communication */`
8 **broadcast** $(\mathbf{y}_s^{(i)})$
`/* Parallel lookup on all machines */`
9 $P^{(i)}$ = **pivotLookup** (i)
`/* Local to worker hosting` $P^{(i)}$ `*/`
`/* LAPAC forward substitution routine */`
10 $\mathbf{x}_s^{(i)}$ = **dptrts** $(P^{(i)}, \mathbf{y}_s^{(i)})$
`/* Collect to driver */`
11 $\mathbf{x}_s^{(i)}$ = **collect** $(\mathbf{x}_s^{(i)})$
`/* One to many communication */`
12 **broadcast** $(\mathbf{x}_s^{(i)})$
`/* Distributed matrix vector multiply */`
`/* Collect to driver */`
13 v = **collect** $(H^{(i)}\mathbf{x}_s^{(i)})$
`/* Update for next iterations */`
14 $L = K^{(i)}$
15 $\mathbf{x} = \mathbf{x}_u^{(i)}$
16 $\mathbf{y} = \mathbf{y}_u^{(i)} - v$
17 $i = i + 1$
18 return $\mathbf{x}$

Algorithm 5 illustrates the main steps of our iterative process. The design pattern consists in solving the system one block at the time, on a single machine, while exploiting all distributed machines to update matrix vectors $\mathbf{x}$ and $\mathbf{y}$, before moving forward to explore the next blocks to ingest. In each iteration i, our algorithm identifies a *pivot region* that we denote by $P^{(i)}$: this is the block (i, i) of L. We also identify a *left panel* and a *right panel*, which we denote by $H^{(i)}$ and $K^{(i)}$, respectively (see Figure 2). Vectors $\mathbf{x}$ and $\mathbf{y}$ are partitioned according to the position of the pivot region: $s^{(i)}$ and $u^{(i)}$ denote regions of the vectors that our algorithm solves and updates at each iteration. The key idea to enable a distributed algorithm design is to rewrite the original linear system as:

$$P^{(i)}\mathbf{x}_{s^{(i)}} = \mathbf{y}_{s^{(i)}}$$
$$K^{(i)}\mathbf{x}_{u^{(i)}} = \mathbf{y}_{u^{(i)}} - H^{(i)}\mathbf{x}_{s^{(i)}}$$

Hence, the driver machine broadcasts the vector $\mathbf{y}_{s^{(i)}}$ and triggers a parallel lookup to find the pivot region $P^{(i)}$. The machine holding $P^{(i)}$ performs a *local* forward substitution to compute $\mathbf{x}_{s^{(i)}}$; then, the driver collects the result.[1] Next, the driver broadcasts $\mathbf{x}_{s^{(i)}}$, triggers the distributed matrix-vector multiplication $H^{(i)}\mathbf{x}_{s^{(i)}}$ involving all machines hosting blocks of $H^{(i)}$, collects the resulting vector and computes $\mathbf{y}_{u^{(i)}} - H^{(i)}\mathbf{x}_{s^{(i)}}$. At this point, as shown in Figure 2 (bottom), a new iteration starts: the driver selects a new pivot region $P^{(i+1)}$, which determines $H^{(i+1)}$ and $K^{(i+1)}$, and repeats the above procedure until the whole $\mathbf{x}$ vector is solved.

Distributed backward substitution. The backward substitution algorithm solves the problem $L^T\mathbf{x} = \mathbf{y}$, where L is a $n \times n$ upper triangular matrix, $\mathbf{y}$ is a $n \times 1$ vector. The algorithm is similar to forward substitution, but it applies to the transpose of L: as a consequence, our algorithm is specular with respect to the anti-diagonal of L^T. The first pivot region corresponds to the lower-right corner of the matrix (as opposed to the upper-left corner), and left and right panels H and K are the block-row adjacent to the pivot and sub triangular matrix excluding the pivot, respectively.

The main difference between backward and forward substitution is that it requires our range lookup procedure to be augmented with an additional *next* pointer, which helps navigating the storage tree by row. The construction cost of the new pointer is equivalent to what discussed in section 4, and is payed only once, during the computation of the covariance matrix.

7 MODEL SELECTION

For a statistical model to be a practical tool in an application, it is necessary to make decisions about the details of its specification: with model selection, we address the issue of choosing the continuous hyper-parameters of the covariance function.

A common way to estimate hyper-parameters θ for GPR amounts to maximizing the marginal likelihood, computed as:

$$\log p(\mathbf{y}|X,\theta) = -\frac{1}{2}\mathbf{y}^\top K_y^{-1}\mathbf{y} - \frac{1}{2}\log|K_y| - \frac{n}{2}\log 2\pi, \quad (5)$$

where $K_y = K_{X,X} + \sigma I$. The optimization problem can be solved by taking the partial derivatives of the marginal log likelihood w.r.t. hyper-parameters, which is given by:

$$\frac{\partial}{\partial\theta_i}\log p(\mathbf{y}|X,\theta) = \frac{1}{2}\mathbf{y}^T K_y^{-1}\frac{\partial K_y}{\partial\theta_i}K_y^{-1}\mathbf{y} - \frac{1}{2}\mathrm{Tr}(K_y^{-1}\frac{\partial K_y}{\partial\theta_i}), \quad (6)$$

where Tr() indicates the trace operator. Gradient based optimization of hyper-parameters can be carried out with various methods, including L-BFGS, gradient descent, and conjugate gradient. In this work we use L-BFGS, and define a novel initialization technique to address the efficiency and scalability of model selection.

Since the convergence rate of a gradient-based method to compute hyper-parameter can be slow, our technique proceeds in two distinct phases. In the *initialization* step, we use several small subsets of the full training set, which can fit a single machine, and train **in parallel** multiple-fidelity GPRs [14] to estimate a series hyper-parameters, for an increasing training set size. This step concludes by fitting a simple regression model to the hyper-parameter data, to infer an initialization of the hyper-parameters for the optimization process on the full training set. In the *training* step, we perform gradient-based estimation of hyper-parameters on the full training set, and impose a computational budget, which we label λ, in terms of the number of iterations the optimization is allowed to perform.

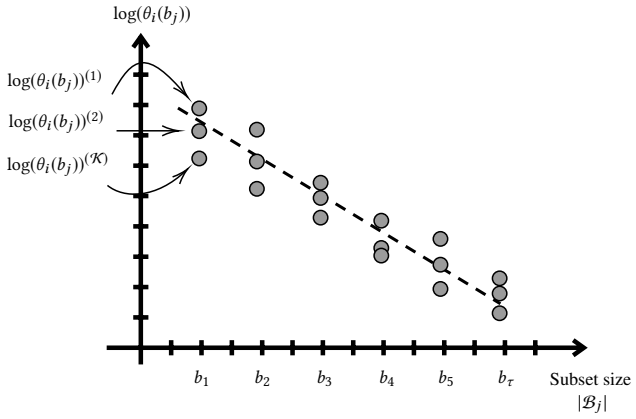

Figure 3: Illustration of the initialization procedure to estimate GP parameters, based on a multi-fidelity approach.

More formally, let $\mathcal{B}_j$ be a subset of $\mathcal{D}$ (the full dataset), sampled uniformly at random with replacement, with $j \in (1, \cdots, \tau)$. Also, let $|\mathcal{B}_j| = b_j$, with:

$$b_j = k_1 n + j\frac{k_2 - k_1}{\tau}n$$

where we have $\min\{b_j\} = k_1 n$, $\max\{b_j\} = k_2 n$, and $k_1, k_2 \ll 1$. The procedure to build the τ subsets $\mathcal{B}_j$ is repeated $\mathcal{K}$ times, such that we can train a regression model using $\mathcal{K}$ data points for each subset size b_j.

Then, the key idea of the initialization phase is to build a simple regression model such that $\theta_i(\mathcal{B}_j) = g(b_j)$, using $b_j, \theta_i^*(\mathcal{B}_j^k)\,\forall j \in \{1\cdots\tau\}, \forall k \in \{1\cdots\mathcal{K}\}$ to train the model, where $\theta_i^*(\mathcal{B}_j^k)$ is the parameter obtained by gradient-based minimization of the marginal log likelihood of the "small" GPRs using input $\mathcal{B}_j$. Figure 3 illustrates the gist of our model. In this work, we use a linear regression model in logarithmic space,[2] that is, we have that:

$$\log\theta_i(\mathcal{B}_j) = \mathbf{w}^T\mathbf{b},$$

where $\mathbf{w} = \{w_1, w_2, \cdots, w_\tau\}^T$ is the vector of model weights, and $\mathbf{b} = \{b_1, b_2, \cdots, b_\tau\}^T$ is the vector of training subset sizes.

We learn model parameters by minimizing the loss function $\mathcal{L} = \sum_{j=1}^{\tau}(\theta_i(\mathcal{B}_j) - \theta_i^*(\mathcal{B}_j))^2 e^{tb_j}$, where t is an additional parameter of the loss function, which weights the (decaying) contribution of each subset $\mathcal{B}_j$ depending on its size.

The initialization step can be easily parallelized. We have implemented a "locality preserving" sampling procedure that builds

[1] Our approach saves on communication costs as broadcasting $\mathbf{y}_{r^{(i)}}$ is cheaper than collecting $P^{(i)}$ at the driver to perform the local forward substitution.

[2] The choice for this model is inspired by Silverman's "rule of thumb" [46].

subsets $\mathcal{B}_j$ using the distributed dataset $\mathcal{D}$, while making sure for subset sizes to be evenly distributed across machines. Each machine contributes to building the training data for our "log-linear" model of the GPR parameters, and to train local models using the local training data. Then, the driver machine collects all local models in each worker machines, and produces an average model.

At this point, the driver machines proceeds by coordinating a distributed execution of the L-BGFS algorithm using Line 18 to find the optimal hyper parameters θ_i of the full GPR model, which is constrained by the "early stopping" mechanism based on an iteration budget λ.

8 EXPERIMENTAL EVALUATION

In this section we proceed with an experimental evaluation of our distributed approach to GPR, and compare it to alternative approaches from the state of the art that use various forms of approximations.

Methodology. In our experiments, we use three well-known, publicly available datasets from UCI,[3] namely Protein, Powerplant and Airline. The Powerplant and Protein datasets were pre-processed according to the methodology in [7], whereas the Airline dataset is pre-processed according to the methodology described in [23]. For each dataset, we build several training sets of different sizes. Powerplant has 4 training sizes: $\{1, 2, 5, 9.469\}\times10^3$; Protein has 6 training sizes: $\{1, 2, 5, 10, 20, 45.515\}\times10^3$; Airline has 7 training sizes: $\{10, 20, 40, 60, 80, 100, 120\}\times10^3$. With each training size, 10 folds of data are constructed. The test dataset sizes are 99, 215 and 10^4 for the Powerplant, Protein and Airline datasets respectively.

The metrics we use in our evaluation are the Standardized Mean Square Error (SMSE) and mean standardized log loss (MSLL), as defined in [44]. We also report the training times of each approach.

We compare our Apache Spark implementation of the distributed GPR (that we label DistGPR)[4], against an optimized TensorFlow implementation of the sparse variational GP (which we label SVGP) from GPFlow [10], and against an approximate GP that uses random Fourier feature expansion of the kernel matrix and stochastic variational inference (which we label DGP-RFF) as presented in [7], which can both run on a single machine.

The experiments are set up as follows: SVGP and DGP-RFF run on a single machine with 128GB RAM, and 32 cores, whereas our DistGPR runs on an Apache Spark cluster with 1 master node and 10 workers, connected by 1 Gbps network. Each worker has 32GB RAM and 6 cores. All experiments are repeated 10 times, and our figures report the average metrics computed across such trials.

Finally, in our experiments, we configure our approach using the following parameters $\mathcal{K} = 30$, $\tau = 10$, $\lambda \in \{0, 3, 7\}$ and $k_2 = max(0.2, 6000/n)$, $k_1 = max(0.3k_2, 1000/n)$. When using SVGP, we use $Z = \sqrt{N}$ inducing points, and for the DGP-RFF approach we use as many Fourier features as inducing points in SVGP, and configure it to be a shallow GP (single layer).

Experimental results. Figure 4 shows the comparison between our DistGPR and both approximate methods, SVGP and DGP-RFF respectively. Each row in the figure corresponds to a different performance metric, while each column indicates a different dataset. The x-axis in the figure reports the training set sizes (in the order of 10^3).

We observe that exact inference pays off in terms of lower error rates, when compared to approximate methods. Indeed, approximate inference techniques might suffer from lack of training data, as visible for the DGP-RFF approach for the Powerplant and Protein datasets. In addition, approximate methods can experience problems to optimize hyper-parameters. As shown in the Airlines dataset, the optimization of inducing points can lead to suboptimal choices, which hurt performance of SVGP. Overall, our approach outperforms or is on par with all alternatives, on both our metrics.

Another key observation is that, for our DistGPR method, costly hyper parameter estimation exhibit diminishing returns: large values of the computational budget λ do not bring substantial benefits, and instead result in long training times. Our multi-fidelity initialization technique is sufficient to estimate good (albeit not optimal) hyper parameters, and allows our method to scale up to problems involving large covariance matrices.

Figure 5 illustrates the training time of each approach. As expected, approximate methods are much faster than exact ones, and the difference amplifies with increasing training set sizes. For the largest dataset, Airlines, SVGP can be up to one order of magnitude faster than our method, whereas DGP-RFF exhibit surprisingly small training times, which are mainly due to the "mini-batch" based stochastic optimization used to compute the posterior. A close look at DistGPR indicates that it scales roughly linearly with training set size (the figure is in log scale). Large values of the computational budget λ imply an order of magnitude larger training times, which cannot be justified by lower error rates. As a concluding remark, we stress that our experimental platform is limited by the 1Gbps network infrastructure: faster networks, e.g. 10 Gbps or more, are becoming the norm, and we expect our training times to decrease significantly with such modern fabrics.

9 CONCLUSION

With the pervasiveness of machine learning in modern society, the study of statistical models enabling a principled approach to the quantification of uncertainty is truly important. Current methods, such as GP modeling, suffer from hard computational challenges, that have largely been addressed through approximation techniques.

In this work, we set to study the design and evaluation of a distributed approach to exact GP inference for regression. Motivated by the abundance of computing resources at a relatively cheap prices, and the advent of widely accessible, high-level distributed programming frameworks, we presented perhaps the first attempt to achieve model parallelism for GP inference. We introduced new, efficient data structures in support to a range of distributed linear algebra primitives, and defined a unifying design pattern to control parallelism and memory consumption.

Using publicly available datasets for regression tasks, we compared our distributed approach to optimized, single machine implementations of state-of-the-art approximate variants, which are known to scale to very large training data. Our results illustrated the trade-off between lower error rates and better quantification of uncertainty offered by exact inference and lower training times of

[3] https://archive.ics.uci.edu/ml/datasets.html
[4] https://github.com/DistributedSystemsGroup/DistGP-code

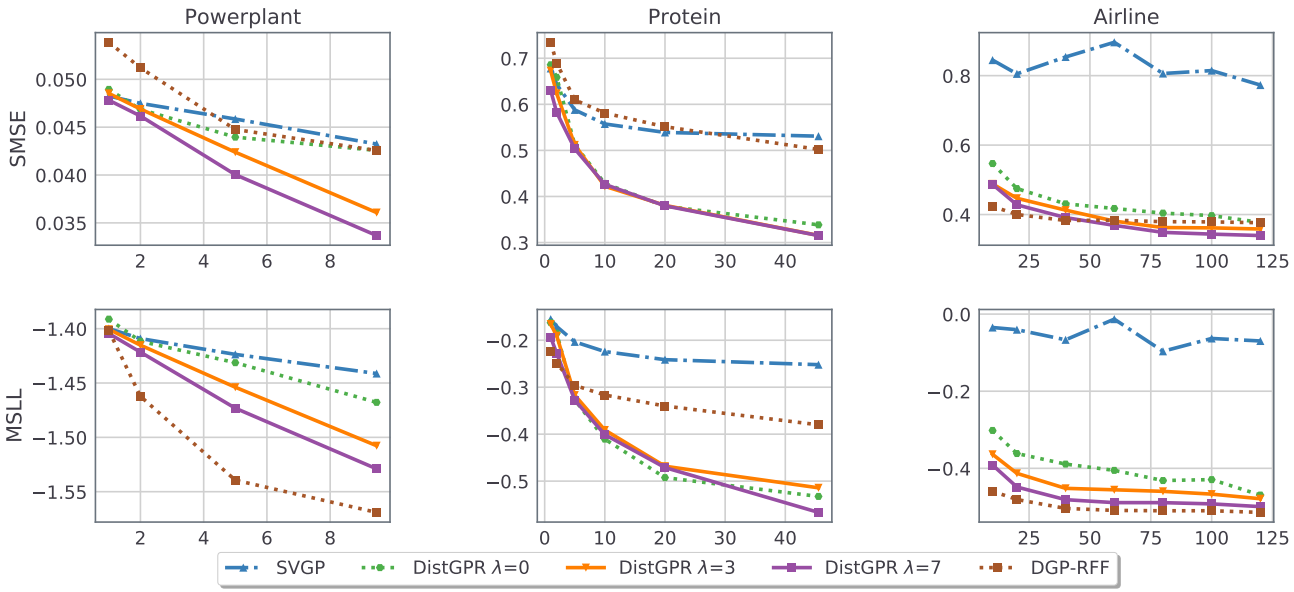

Figure 4: Comparison of different GP regression approaches, as a function of increasing training set sizes ($\times 10^3$).

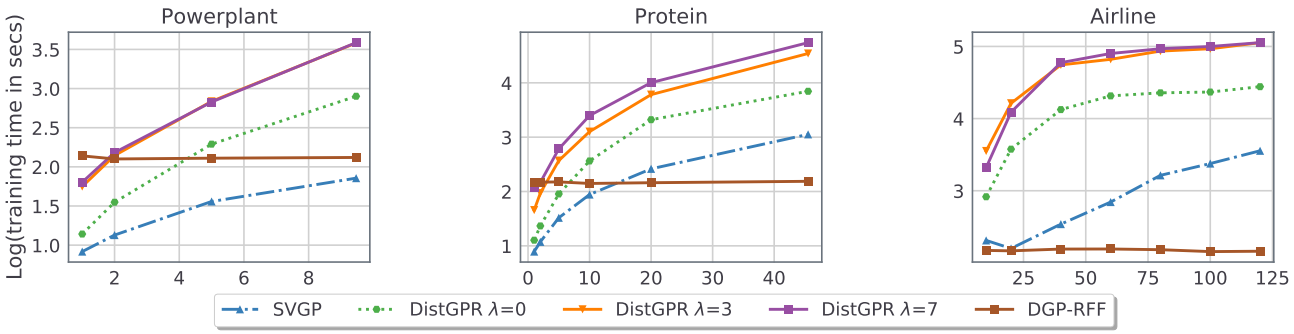

Figure 5: Training times of different GP regression approaches, as a function of increasing training set sizes ($\times 10^3$).

approximate methods. Many critical applications require high predictive confidence, and our work presented a practical approach to exact Gaussian Process regression, which enables such performance.

The performance we measured for our method was bounded by the capacity of our experimental platform. Network speed and memory capacity are key to lower training times and even larger datasets (e.g. MNIST). While additional experiments on public clouds could reveal interesting, our work indicated that modern distributed computing frameworks require further work to introduce simple and safe interfaces for message broadcast. Finally, we believe our work could be combined with approximate approaches, which we plan to study in the future. A detailed comparison to the standard parameter server approach [30] is also truly desirable.

ACKNOWLEDGMENTS

MF gratefully acknowledges support from the AXA Research Fund. Pietro Michiardi was partially supported by KPMG.

REFERENCES

[1] E. Anderson et al. 1999. *LAPACK Users' Guide (Third Ed.)*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA.

[2] E. Anderson, Z. Bai, J. Dongarra, A. Greenbaum, A. McKenney, J. Du Croz, S. Hammarling, J. Demmel, C. Bischof, and D. Sorensen. 1990. LAPACK: A Portable Linear Algebra Library for High-performance Computers. In *Proceedings of the 1990 ACM/IEEE Conference on Supercomputing (Supercomputing '90)*. IEEE Computer Society Press, Los Alamitos, CA, USA, 2–11.

[3] S. S. Bansal, B. Vishal, and R. Gupta. 2002. Near Optimal Cholesky Factorization on Orthogonal Multiprocessors. *Inf. Process. Lett.* 84, 1 (2002), 23–30. https://doi.org/10.1016/S0020-0190(02)00222-3

[4] L. S. Blackford et al. 1997. *ScaLAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA.

[5] Jaeyoung Choi et al. 1996. Design and Implementation of the ScaLAPACK LU, QR, and Cholesky Factorization Routines. *Sci. Program.* 5, 3 (Aug. 1996), 173–184.

[6] Arindam Choudhury, Prasanth B. Nair, and Andy J. Keane. 2002. A Data Parallel Approach for Large-Scale Gaussian Process Modeling. In *Proceedings of the Second SIAM International Conference on Data Mining, Arlington, VA, USA, April 11-13, 2002*. SIAM, 95–111. https://doi.org/10.1137/1.9781611972726.6

[7] Kurt Cutajar, Edwin V. Bonilla, Pietro Michiardi, and Maurizio Filippone. 2017. Random Feature Expansions for Deep Gaussian Processes. In *Proceedings of the 34th International Conference on Machine Learning (Proceedings of Machine Learning Research)*, Doina Precup and Yee Whye Teh (Eds.), Vol. 70. PMLR,

International Convention Centre, Sydney, Australia, 884–893.

[8] Zhenwen Dai, Andreas C. Damianou, James Hensman, and Neil D. Lawrence. 2014. Gaussian Process Models with Parallelization and GPU acceleration. *CoRR* abs/1410.4984 (2014). arXiv:1410.4984

[9] K. Das and A. N. Srivastava. 2010. Block-GP: Scalable Gaussian Process Regression for Multimodal Data. In *2010 IEEE International Conference on Data Mining*. 791–796. https://doi.org/10.1109/ICDM.2010.38

[10] Alexander G. De G. Matthews, Mark Van Der Wilk, Tom Nickson, Keisuke Fujii, Alexis Boukouvalas, Pablo León-Villagrá, Zoubin Ghahramani, and James Hensman. 2017. GPflow: A Gaussian Process Library Using Tensorflow. *J. Mach. Learn. Res.* 18, 1 (Jan. 2017), 1299–1304.

[11] Marc Peter Deisenroth and Jun Wei Ng. 2015. Distributed Gaussian Processes. In *Proceedings of the 32Nd International Conference on International Conference on Machine Learning - Volume 37 (ICML'15)*. JMLR.org, 1481–1490.

[12] Jack J. Dongarra et al. 1988. An Extended Set of FORTRAN Basic Linear Algebra Subprograms. *ACM Trans. Math. Softw.* 14, 1 (March 1988), 1–17. https://doi.org/10.1145/42288.42291

[13] J. J. Dongarra et al. 1990. A Set of Level 3 Basic Linear Algebra Subprograms. *ACM Trans. Math. Softw.* 16, 1 (March 1990), 1–17. https://doi.org/10.1145/77626.79170

[14] Stefan Falkner, Aaron Klein, and Frank Hutter. 2018. BOHB: Robust and Efficient Hyperparameter Optimization at Scale. In *Proceedings of the 35th International Conference on Machine Learning (Proceedings of Machine Learning Research)*, Jennifer Dy and Andreas Krause (Eds.), Vol. 80. PMLR, Stockholmsmassan, Stockholm Sweden, 1437–1446.

[15] Yarin Gal, Mark van der Wilk, and Carl E. Rasmussen. 2014. Distributed Variational Inference in Sparse Gaussian Process Regression and Latent Variable Models. In *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2 (NIPS'14)*. MIT Press, Cambridge, MA, USA, 3257–3265.

[16] Alan George, Michael T Heath, and Joseph Liu. 1986. Parallel Cholesky factorization on a shared-memory multiprocessor. *Linear Algebra and its applications* 77 (1986), 165–187.

[17] Evgenii Landis Georgy Adelson-Velsky. 1962. An Algorithm for the Organization of Information. *Doklady Akademii Nauk USSR* 146, 2 (1962), 263–266.

[18] W.R. Gilks, S. Richardson, and D. Spiegelhalter. 1995. *Markov Chain Monte Carlo in Practice*. Taylor & Francis.

[19] Laura Grigori and Xiaoye S. Li. 2006. Performance Analysis of Parallel Right-Looking Sparse LU Factorization on Two Dimensional Grids of Processors. In *Applied Parallel Computing. State of the Art in Scientific Computing*. Springer Berlin Heidelberg, Berlin, Heidelberg, 768–777.

[20] Fred G. Gustavson, Lars Karlsson, and Bo Kågström. 2007. Three Algorithms for Cholesky Factorization on Distributed Memory Using Packed Storage. In *Applied Parallel Computing. State of the Art in Scientific Computing*, Bo Kågström, Erik Elmroth, Jack Dongarra, and Jerzy Waśniewski (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 550–559.

[21] Joseph Hellerstein et al. 2012. The MADlib analytics library: or MAD skills, the SQL. *Proceedings of the VLDB Endowment* 5, 12 (2012), 1700–1711.

[22] James Hensman, Nicolas Durrande, and Arno Solin. 2017. Variational Fourier Features for Gaussian Processes. *J. Mach. Learn. Res.* 18, 1 (Jan. 2017), 5537–5588.

[23] James Hensman, Nicoló Fusi, and Neil D. Lawrence. 2013. Gaussian Processes for Big Data. *CoRR* abs/1309.6835 (2013). arXiv:1309.6835

[24] Y. Huang, Y. Yesha, M. Halem, Y. Yesha, and S. Zhou. 2016. YinMem: A distributed parallel indexed in-memory computation system for large scale data analytics. In *2016 IEEE International Conference on Big Data (Big Data)*. IEEE, 214–222. https://doi.org/10.1109/BigData.2016.7840607

[25] Y. Huangfu, J. Cao, H. Lu, and G. Liang. 2015. MatrixMap: Programming Abstraction and Implementation of Matrix Computation for Big Data Applications. In *2015 IEEE 21st International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE, 19–28. https://doi.org/10.1109/ICPADS.2015.11

[26] Mathias Jacquelin, Yili Zheng, Esmond Ng, and Katherine A. Yelick. 2016. An Asynchronous Task-based Fan-Both Sparse Cholesky Solver. *CoRR* abs/1608.00044 (2016).

[27] S. G. Kratzer. 1992. Massively parallel sparse LU factorization. In *[Proceedings 1992] The Fourth Symposium on the Frontiers of Massively Parallel Computation*. 136–140. https://doi.org/10.1109/FMPC.1992.234896

[28] Chuck L Lawson, Richard J. Hanson, David R Kincaid, and Fred T. Krogh. 1979. Basic linear algebra subprograms for Fortran usage. *ACM TOMS* 5, 3 (1979), 308–323.

[29] Boduo Li, Sandeep Tata, and Yannis Sismanis. 2013. Sparkler: Supporting Large-scale Matrix Factorization. In *Proceedings of the 16th International Conference on Extending Database Technology (EDBT '13)*. ACM, New York, NY, USA, 625–636. https://doi.org/10.1145/2452376.2452449

[30] Mu Li et al. 2014. Scaling Distributed Machine Learning with the Parameter Server. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI'14)*. USENIX Association, Berkeley, CA, USA, 583–598.

[31] Mu Li, David G Andersen, Alexander J Smola, and Kai Yu. 2014. Communication Efficient Distributed Machine Learning with the Parameter Server. In *Advances in Neural Information Processing Systems 27*, Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger (Eds.). Curran Associates, Inc., Red Hook, NY 12571, USA, 19–27.

[32] Jun Liu, Yang Liang, and Nirwan Ansari. 2016. Spark-based large-scale matrix inversion for big data processing. *IEEE Access* 4 (2016), 2166–2176.

[33] Chandan Misra, Swastik Haldar, Sourangshu Bhattacharya, and Soumya K. Ghosh. 2018. SPIN: A Fast and Scalable Matrix Inversion Method in Apache Spark. In *Proceedings of the 19th International Conference on Distributed Computing and Networking (ICDCN '18)*. ACM, New York, NY, USA, Article 16, 10 pages. https://doi.org/10.1145/3154273.3154300

[34] Dianne P. O'Leary and G. W. Stewart. 1985. Data-flow Algorithms for Parallel Matrix Computation. *Commun. ACM* 28, 8 (Aug. 1985), 840–853. https://doi.org/10.1145/4021.4025

[35] G. Oliva, R. Setola, and C. N. Hadjicostis. 2016. Distributed asynchronous Cholesky decomposition. In *2016 IEEE 55th Conference on Decision and Control (CDC)*. IEEE, 4414–4419. https://doi.org/10.1109/CDC.2016.7798939

[36] Christopher J. Paciorek, Benjamin Lipshitz, Wei Zhuo, Prabhat, Cari G. Kaufman, and Rollin C. Thomas. 2015. Parallelizing Gaussian Process Calculations in R. *Journal of Statistical Software* 63, 10 (2015), 1–23.

[37] Chiwoo Park and Daniel W. Apley. 2018. Patchwork Kriging for Large-scale Gaussian Process Regression. *Journal of Machine Learning Research* 19, 7 (2018), 1–43.

[38] Chiwoo Park and Jianhua Z. Huang. 2016. Efficient Computation of Gaussian Process Regression for Large Spatial Data Sets by Patching Local Gaussian Processes. *Journal of Machine Learning Research* 17, 174 (2016), 1–29.

[39] Hao Peng, Shandian Zhe, Xiao Zhang, and Yuan Qi. 2017. Asynchronous Distributed Variational Gaussian Process for Regression. In *Proceedings of the 34th International Conference on Machine Learning (Proceedings of Machine Learning Research)*, Doina Precup and Yee Whye Teh (Eds.), Vol. 70. PMLR, International Convention Centre, Sydney, Australia, 2788–2797.

[40] Zhengping Qian, Xiuwei Chen, Nanxi Kang, Mingcheng Chen, Yuan Yu, Thomas Moscibroda, and Zheng Zhang. 2012. MadLINQ: Large-scale Distributed Matrix Computation for the Cloud. In *Proceedings of the 7th ACM European Conference on Computer Systems (EuroSys '12)*. ACM, New York, NY, USA, 197–210. https://doi.org/10.1145/2168836.2168857

[41] Joaquin Quiñonero Candela and Carl Edward Rasmussen. 2005. A Unifying View of Sparse Approximate Gaussian Process Regression. *J. Mach. Learn. Res.* 6 (Dec. 2005), 1939–1959.

[42] Padma Raghavan. 1992. *Distributed Sparse Matrix Factorization: QR and Cholesky Decompositions*. Ph.D. Dissertation. University Park, PA, USA. UMI Order No. GAX92-14255.

[43] CE. Rasmussen and CKI. Williams. 2006. *Gaussian Processes for Machine Learning*. MIT Press, Cambridge, MA, USA. 248 pages.

[44] Carl Edward Rasmussen. 2004. *Gaussian Processes in Machine Learning*. Springer Berlin Heidelberg, Berlin, Heidelberg, 63–71. https://doi.org/10.1007/978-3-540-28650-9_4

[45] Carl E. Rasmussen and Zoubin Ghahramani. 2002. Infinite Mixtures of Gaussian Process Experts. In *Advances in Neural Information Processing Systems 14*, T. G. Dietterich, S. Becker, and Z. Ghahramani (Eds.). MIT Press, Cambridge, MA, USA, 881–888.

[46] B. W. Silverman. 1986. *Density estimation for statistics and data analysis*. Monographs on Statistics & Applied Probability, Vol. 26. Chapman and Hall, London.

[47] Przemysław Stpiczyński. 1992. Parallel Cholesky factorization on orthogonal multiprocessors. *Parallel Comput.* 18, 2 (1992), 213 – 219. https://doi.org/10.1016/0167-8191(92)90080-Q

[48] Michalis K. Titsias. 2009. Variational Learning of Inducing Variables in Sparse Gaussian Processes. In *In Proc. of AISTATS (JMLR Proceedings)*, Vol. 5. JMLR.org, 567–574.

[49] Leslie G Valiant. 1990. A bridging model for parallel computation. *Commun. ACM* 33, 8 (1990), 103–111.

[50] Andrew Gordon Wilson, Christoph Dann, and Hannes Nickisch. 2015. Thoughts on Massively Scalable Gaussian Processes. *CoRR* abs/1511.01870 (2015). arXiv:1511.01870

[51] Andrew Gordon Wilson and Hannes Nickisch. 2015. Kernel Interpolation for Scalable Structured Gaussian Processes (KISS-GP). *CoRR* abs/1503.01057 (2015). arXiv:1503.01057

[52] Jingen Xiang, Huangdong Meng, and Ashraf Aboulnaga. 2014. Scalable matrix inversion using MapReduce. In *HPDC*. ACM, 177–190.

[53] Reza Bosagh Zadeh, Xiangrui Meng, Burak Yavuz, Aaron Staple, Li Pu, Shivaram Venkataraman, Evan R. Sparks, Alexander Ulanov, and Matei Zaharia. 2015. linalg: Matrix Computations in Apache Spark. *CoRR* abs/1509.02256 (2015). arXiv:1509.02256

[54] D. Zheng and T.Y.P. Chang. 1995. Parallel cholesky method on MIMD with shared memory. *Computers & Structures* 56, 1 (1995), 25 – 38.