Journal of Statistical Software

January 2015, Volume 63, Issue 10. *http://www.jstatsoft.org/*

Parallelizing Gaussian Process Calculations in R

Christopher J. Paciorek
University of California, Berkeley

Benjamin Lipshitz
University of California, Berkeley

Wei Zhuo
Georgia Institute of Technology

Prabhat
Lawrence Berkeley National Laboratory

Cari G. Kaufman
University of California, Berkeley

Rollin C. Thomas
Lawrence Berkeley National Laboratory

Abstract

We consider parallel computation for Gaussian process calculations to overcome computational and memory constraints on the size of datasets that can be analyzed. Using a hybrid parallelization approach that uses both threading (shared memory) and message-passing (distributed memory), we implement the core linear algebra operations used in spatial statistics and Gaussian process regression in an R package called **bigGP** that relies on C and **MPI**. The approach divides the covariance matrix into blocks such that the computational load is balanced across processes while communication between processes is limited. The package provides an API enabling R programmers to implement Gaussian process-based methods by using the distributed linear algebra operations without any C or **MPI** coding. We illustrate the approach and software by analyzing an astrophysics dataset with $n = 67,275$ observations.

Keywords: distributed computation, kriging, linear algebra.

1. Introduction

Gaussian processes are widely used in statistics and machine learning for spatial and spatio-temporal modeling (Banerjee, Gelfand, and Sirmans 2003), design and analysis of computer experiments (Kennedy and O'Hagan 2001), and non-parametric regression (Rasmussen and Williams 2006). One popular example is the spatial statistics method of kriging, which is equivalent to conditional expectation under a Gaussian process model for the unknown spatial field. However standard implementations of Gaussian process-based methods are computa-

tionally intensive because they involve calculations with covariance matrices of size n by n where n is the number of locations with observations. In particular the computational bottleneck is generally the Cholesky decomposition of the covariance matrix, whose computational cost is of order n^3.

For example, a basic spatial statistics model (in particular a geostatistical model) can be specified in a hierarchical fashion as

$$\begin{aligned} \boldsymbol{Y}|\boldsymbol{g},\boldsymbol{\theta} &\sim \mathcal{N}(\boldsymbol{g}, \boldsymbol{C}_y(\boldsymbol{\theta})) \\ \boldsymbol{g}|\boldsymbol{\theta} &\sim \mathcal{N}(\boldsymbol{\mu}(\boldsymbol{\theta}), \boldsymbol{C}_g(\boldsymbol{\theta})), \end{aligned}$$

where $\boldsymbol{g}$ is a vector of latent spatial process values at the n locations, $\boldsymbol{C}_y(\boldsymbol{\theta})$ is an error covariance matrix (often diagonal), $\mu(\boldsymbol{\theta})$ is the mean vector of the latent process, $\boldsymbol{C}_g(\boldsymbol{\theta})$ is the spatial covariance matrix of the latent process, and $\boldsymbol{\theta}$ is a vector of unknown parameters. We can marginalize over $\boldsymbol{g}$ to obtain the marginal likelihood,

$$\boldsymbol{Y}|\boldsymbol{\theta} \sim \mathcal{N}(\boldsymbol{\mu}(\boldsymbol{\theta}), \boldsymbol{C}(\boldsymbol{\theta})),$$

where $\boldsymbol{C}(\boldsymbol{\theta}) = \boldsymbol{C}_y(\boldsymbol{\theta}) + \boldsymbol{C}_g(\boldsymbol{\theta})$. This gives us the marginal density,

$$f(\boldsymbol{y}) \propto |\boldsymbol{C}(\boldsymbol{\theta})|^{-1/2} \exp\left\{-\frac{1}{2}(\boldsymbol{y}-\boldsymbol{\mu}(\boldsymbol{\theta}))^\top (\boldsymbol{C}(\boldsymbol{\theta}))^{-1}(\boldsymbol{y}-\boldsymbol{\mu}(\boldsymbol{\theta}))\right\},$$

which is maximized over $\boldsymbol{\theta}$ to find the maximum likelihood estimator. At each iteration in the maximization, the expensive computations are to compute the entries of the matrix $\boldsymbol{C}(\boldsymbol{\theta})$ as a function of $\boldsymbol{\theta}$, calculate the Cholesky decomposition, $\boldsymbol{L}\boldsymbol{L}^\top = \boldsymbol{C}(\boldsymbol{\theta})$, and solve a system of equations $\boldsymbol{L}^{-1}(\boldsymbol{y}-\boldsymbol{\mu}(\boldsymbol{\theta}))$ via a forwardsolve operation. Given the MLE, $\hat{\boldsymbol{\theta}}$, one might then do spatial prediction, calculate the variance of the prediction, and simulate realizations conditional on the data. These additional tasks involve the same expensive computations plus a few additional closely-related computations.

In general the Cholesky decomposition will be the rate-limiting step in these tasks, although calculation of covariance matrices can also be a bottleneck. In addition to computational limitations, memory use can be a limitation, as storage of the covariance matrix involves n^2 floating points. For example, simply storing a covariance matrix for $n = 20,000$ observations in memory uses approximately 3.2 GB of RAM. As a result of the computational and memory limitations, standard spatial statistics methods are typically applied to datasets with at most a few thousand observations.

To overcome these limitations, a small industry has arisen to develop computationally-efficient approaches to spatial statistics, involving reduced rank approximations (Kammann and Wand 2003; Banerjee, Gelfand, Finley, and Sang 2008; Cressie and Johannesson 2008), tapering the covariance matrix to induce sparsity (Furrer, Genton, and Nychka 2006; Kaufman, Schervish, and Nychka 2008), approximation of the likelihood (Stein, Chi, and Welty 2004), and fitting local models by stratifying the spatial domain (Gramacy and Lee 2008), among others. In the kriging context, estimation of the parameters using the variogram, a method of moments estimator, combined with prediction within local neighborhoods is computationally tractable for large datasets.

At the same time, computer scientists have developed and implemented parallel linear algebra algorithms that use modern distributed memory and multi-core hardware. In our work, rather

than modifying the statistical model, as statisticians have focused on, we consider the use of parallel algorithms to overcome computational limitations, enabling analyses with much larger covariance matrices than would be otherwise possible.

We present an algorithm and R (R Core Team 2014) package, **bigGP**, for distributed linear algebra calculations focused on those used in spatial statistics and closely-related Gaussian process regression methods. The approach divides the covariance matrix (and other necessary matrices and vectors) into blocks, with the blocks distributed amongst processors in a distributed computing environment. The algorithm builds on that encoded within the widely-used parallel linear algebra package, **ScaLAPACK** (Blackford *et al.* 1987), a parallel extension to the standard **LAPACK** (Anderson 1999) routines. The core functions in the **bigGP** package are C functions, with R wrappers, that rely on standard **BLAS** (Dongarra, Croz, Hammarling, and Duff 1990) functionality and on **MPI** (Gropp, Lusk, and Skjellum 1999) for message passing. This set of core functions includes Cholesky decomposition, forwardsolve and backsolve, and crossproduct calculations. These functions, plus some auxiliary functions for communication of inputs and outputs to the processes, provide an API through which an R programmer can implement methods for Gaussian-process-based computations. Using the API, we provide a set of methods for the standard operations involved in kriging and Gaussian process regression, namely

- likelihood optimization,
- prediction,
- calculation of prediction uncertainty,
- unconditional simulation of Gaussian processes, and
- conditional simulation given data.

These methods are provided as R functions in the package. We illustrate the use of the software for Gaussian process regression in an astrophysics application.

We close this introduction by situating our software within the context of other software for Gaussian process modeling. A broad variety of software, both within and outside of R, is available for working with Gaussian processes, much of it implementing various approximate methods. However, there is little parallelized software for working with Gaussian processes without approximations. R provides a wide variety of tools for parallelization, best summarized in the "High Performance Computing" task view (Eddelbuettel 2014) on the Comprehensive R Archive Network (CRAN), as well as a variety of tools for Gaussian process models. The Gaussian process-related packages available on CRAN do not provide parallelized software for exact calculations, with packages such as **mlegp** (Dancik and Dorman 2008), **gptk** (Kalaitzis, Gao, Honkela, and Lawrence 2014), **laGP** (Gramacy 2014), and **GPfit** (MacDonald, Ranjan, and Chipman 2013) implementing either approximate methods or non-parallel algorithms for exact computations. The packages **MAGMA** (Tomov, Nath, Du, and Dongarra 2011) and **openCL** (Stone, Gohara, and Shi 2010) provide general purpose interfaces to GPU libraries that could be used to develop user-friendly tools for parallel Gaussian process calculations. Finally, and most promisingly, the recently-released **pbd** packages (Ostrouchov, Chen, Schmidt, and Patel 2012) for working with large datasets wrap **ScaLAPACK** to implement parallel

linear algebra. **pbd** allows one to distribute matrices amongst multiple processors, with distributed linear algebra computations carried out without collecting intermediate results back to a single processor. Thus, an alternative to the core functions in our software would make use of the **pbd** functions in place of our API, with our methods for kriging and Gaussian process regression instead using **pbd** on the back end. Note that while our software is based upon the blocked approach of **ScaLAPACK** (Section 2.1), we tailor the algorithm to achieve better load-balancing and limit communication for Cholesky factorization (Section 2.2). Looking outside of R, the **ScalaGAUSS** package (Anitescu, Chen, and Stein 2014) implements (in C++ and MATLAB) likelihood maximization for Gaussian processes using a stochastic approximation to the likelihood that allows for optimization using matrix-vector calculations that are of order $n \log n$ (Anitescu, Chen, and Wang 2012). Recent advances in communication-avoiding linear algebra (Ballard, Demmel, Holtz, and Schwartz 2011) in general, and especially for Cholesky decomposition (Georganas, González-Domínguez, Solomonik, Zheng, no, and Yelick 2012) may prove useful in speeding up parallel Gaussian process computation, but have not yet been incorporated into standard libraries like **ScaLAPACK**.

2. Parallel algorithm and software implementation

2.1. Distributed linear algebra calculations

Parallel computation can be done in both shared memory and distributed memory contexts. Each uses multiple CPUs. In a shared memory context (such as computers with one or more chips with multiple cores), multiple CPUs have access to the same memory and so-called 'threaded' calculations can be done, in which code is written (e.g., using the **openMP** protocol, Chapman, Jost, and Pas 2008) to use more than one CPU at once to carry out a task, with each CPU having access to the objects in memory. In a distributed memory context, one has a collection of nodes, each with their own memory. Any information that must be shared with other nodes must be communicated via message-passing, such as using the **MPI** standard. Our distributed calculations use both threading and message-passing to exploit the capabilities of modern computing clusters with multiple-core nodes.

We begin by describing a basic parallel Cholesky decomposition, sometimes known as Crout's algorithm, which is done on blocks of the matrix and is implemented in **ScaLAPACK**. Figure 1 shows a schematic of the block-wise Cholesky factorization, where the covariance matrix is divided into 6 blocks, a $B = 3$ by $B = 3$ array of blocks (storing only the lower blocks of the symmetric matrix). The arrows show the dependence of each block on the other blocks; an arrow connecting two blocks stored on different nodes indicates that communication is necessary between those nodes. For the Cholesky decomposition, the calculations for the diagonal blocks all involve Cholesky decompositions (and symmetric matrix multiplication for all but the first block), while those for the off-diagonal blocks involve forwardsolve operations and (for all but the first column of blocks) matrix multiplication and subtraction. Most of the total work is in the matrix multiplications.

The first block must be factored before the forwardsolve can be applied to blocks 2 and 3. After the forwardsolves, all the remaining blocks can be updated with a matrix multiplication and subtraction. At this point the decomposition of blocks 1–3 is complete and they will not be used for the rest of the computation. This procedure is repeated along each block

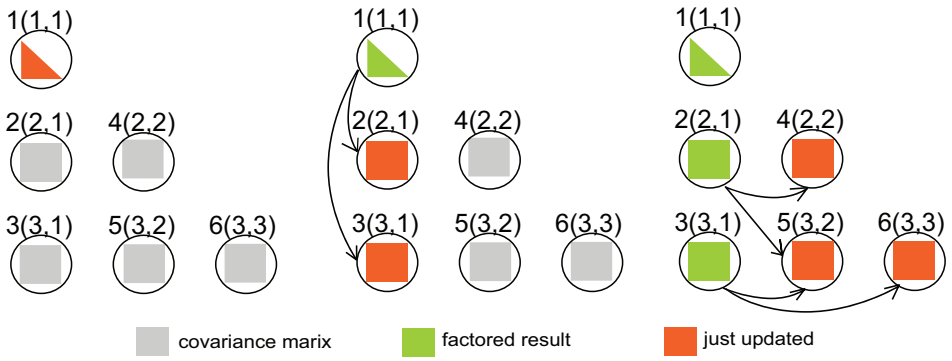

Figure 1: Diagram of the first steps in the distributed Cholesky factorization. Arrows indicate dependencies between processes. Orange coloring indicates computations carried out in each step, and green indicates parts of the matrix that are completely factored as of the previous step. The labels of the form "id(x,y)" indicate the process ID and Cartesian coordinates of the process.

column, so for example block 6 must wait for blocks 4 and 5 to finish before all the necessary components are available to finish their own calculations.

To specify the distributed algorithm, there are several choices to be made: the number of blocks, B, how to distribute these blocks amongst computational processes, how to distribute these processes amongst the nodes, and how many nodes to use. We discuss the tradeoffs involved in these choices next and the choices that our algorithm makes in Section 2.2.

Given a matrix of a given size, specifying the number of blocks is equivalent to choosing the size of the blocks. Larger blocks allow for more efficient local computations and less total communication. The largest effect here is the on-node cache subsystem, which allows each node to run near its peak performance only if the ratio of computation to memory traffic is high enough. The computational efficiency will increase with the block size until the blocks are large enough to fill the cache available to one process. For example, if 8MB of cache are available, one would like to have a block size at least 1024×1024. However, smaller blocks allow the algorithm to better balance the computational load between the processes (and therefore ultimately among the cores and nodes of the computer) by assigning multiple blocks to each process. The first block must finish before anything else can be done; assuming that each block is assigned to only one process, all the other processes must wait for this block to be factorized before they can begin computation. More generally, the diagonal and first off-diagonal blocks form a critical path of the algorithm. In Figure 1, this critical path is of blocks 1, 2, 4, 5, 6. The decomposition, forwardsolves, and multiplicative updates of each of these blocks must be done sequentially. Decreasing the block size decreases the amount of work along this critical path, thus improving the load balance. Put another way, decreasing the block size decreases how long the majority of the processes wait for the processes in a given column to finish before they can use the results from that column to perform their own computation.

Given a matrix of a fixed size and a fixed number of nodes, if we were to use the maximum block size, we would distribute one block per process and one process per node. If we use a smaller block size, we can accommodate the extra blocks either by assigning multiple blocks to a each process, or multiple processes to each node. Consider first running just one process

per node. For the linear algebra computations, by using a threaded BLAS library (such as **openBLAS**, Zhang 2013, **MKL**, Intel 2013, or **ACML**, AMD 2013), it is still possible to attain good multi-core performance on a single process. However, any calculations that are not threaded will not be able to use all of the cores on a given node, reducing computational efficiency. An example of this occurs in our R package where the user-defined covariance function (which is an R function) will typically not be threaded unless the user codes it in threaded C code (e.g., using **openMP**) and calls the C code from the R function or uses a parallel framework in R.

Alternatively, one could specify the block size and the number of nodes such that more than one process runs on each node. For non-threaded calculations, this manually divides the computation amongst multiple cores on a node, increasing efficiency. However it reduces the number of cores available for a given threaded calculation (presuming that cores are assigned exclusively to a single process, as when using the **openMPI** (Gabriel *et al.* 2004) implementation with **Rmpi** (Yu 2002)) and may decrease efficiency by dividing calculations into smaller blocks with more message passing. This is generally a satisfactory solution in a small cluster, but will lose efficiency past a few tens of nodes. Finally, one can assign multiple blocks per process, our chosen approach, described next.

2.2. Our algorithm

Our approach assigns one process per node, but each process is assigned multiple blocks. This allows each process access to all the cores on a node to maximally exploit threading. We carefully choose which blocks are assigned to each process to achieve better load-balancing and limit communication. We choose an efficient order for each process to carry out the operations for the blocks assigned to it. Our package is flexible enough to allow the user to instead run multiple processes per node, which may improve the efficiency of the user-defined covariance function at the expense of higher communication costs in the linear algebra computations.

We require that the number of processes is $P = D(D+1)/2 \in \{1, 3, 6, 10, 15, \ldots\}$ for some integer value of D. We introduce another quantity h that determines how many blocks each process owns. The number of blocks is given by $B = hD$, and so the block size is $\lceil \frac{n}{hD} \rceil$, where n is the order of the matrix. See Figure 2 for an example of the layout with $D = 4$ and either $h = 1$ or $h = 3$. Each "diagonal process" has $h(h+1)/2$ blocks, and each "off-diagonal process" has h^2 blocks of the triangular matrix.

As discussed above, small values of h increase on-node efficiency and reduce communication, but large values of h improve load balance. On current architectures, a good heuristic is to choose h so that the block size is about 1000, but the user is encouraged to experiment and determine what value works best for a given computer and problem size. When using more than 8 cores per process, the block size should probably be increased. Note that we pad the input matrix so that the number of rows and columns is a multiple of hD. This padding will have a minimal effect on the computation time; if the block size is chosen to be near 1000 as we suggest, the padding will be at most one part in a thousand.

Note that when $h > 1$, there are essentially two levels of blocking, indicated by the thin black lines and the thick blue lines in Figure 2. Our algorithm is guided by these blocks. At a high level, the algorithm sequentially follows the Cholesky decomposition of the large (blue) blocks as described in the previous section. Each large block is divided among all the processors, and all the processors participate in each step. For example, the first step is to perform Cholesky

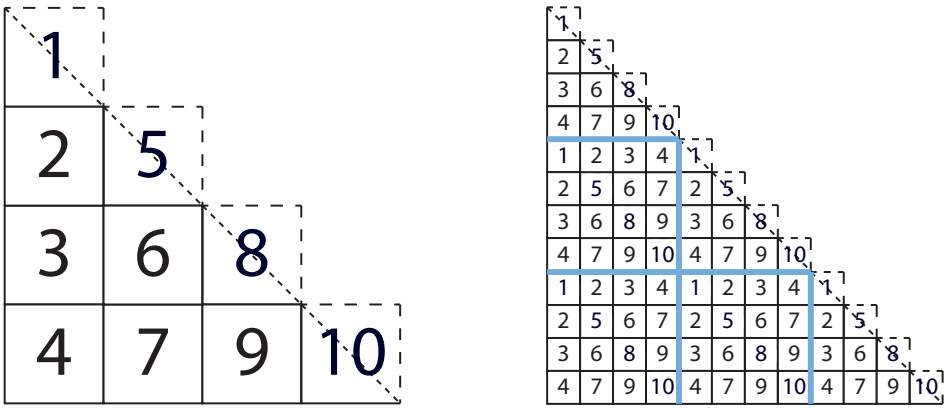

Figure 2: The matrix layout used by our algorithm with $D = 4$ and $h = 1$ (left) or $h = 3$ (right). The numbers indicate which process owns a given block. When $h = 1$, each of the 10 processes owns one block of the matrix. When $h = 3$, the blocks are a third the size in each dimension. The diagonal processes (1, 5, 8, 10) each own $h(h+1)/2 = 6$ blocks, and the off-diagonal processes (2, 3, 4, 6, 7, 9) each own $h^2 = 9$ blocks.

decomposition on the first large block. To do so, we follow exactly the $h = 1$ algorithm (making use of the Cartesian coordinate identification system indicated in Figure 1):

for $i = 1$ to D **do**
  Processor (i, i) computes the Cholesky decomposition of its block
  parallel for $j = i + 1$ to D **do**
    Processor (i, i) sends its block to processor (j, i)
    Processor (j, i) updates its block with a triangular solve
    parallel for $k = i + 1$ to D **do**
      if $k \leq j$ **then**
        Processor (j, i) sends its block to processor (j, k)
      else
        Processor (j, i) sends its block to processor (k, j)
      end if
    end parallel for
  end parallel for
  parallel for $j = i + 1$ to D **do**
    parallel for $k = j + 1$ to D **do**
      Processor (k, j) updates its block with a matrix multiplication
    end parallel for
  end parallel for
end for

The $h = 1$ algorithm is poorly load-balanced; for example going from $D = 1$ to $D = 2$ (one process to three processes), one would not expect any speedup because every operation is along the critical path. However, because it is a small portion of the entire calculation for $h > 1$, the effect on the total runtime is small. Instead, most of the time is spent in matrix multiplications of the large blue blocks, which are well load-balanced.

Memory use

The number of informative entries in a triangular or symmetric $n \times n$ matrix is $n(n+1)/2$. Ideally, it would be possible to perform computations even if there is only barely this much memory available across all the nodes, that is if there were enough memory for $n(n+1)/(D(D+1))$ entries per node. Our algorithm does not reach this ideal, but it has a small memory overhead that decreases as D or h increase. The maximum memory use is by the off-diagonal nodes that own h^2 blocks. Additionally, during the course of the algorithm they must temporarily store up to 4 more blocks. Assuming for simplicity that hD evenly divides n, the maximum memory use on a node is then

$$\begin{aligned} M &\leq \left(\frac{n}{hD}\right)^2 (h^2+4) = \frac{n(n+1)}{D(D+1)} \left(1 + \frac{4nD + n^2h^2 + 4n - Dh^2}{Dh^2n + Dh^2}\right) \\ &< \frac{n(n+1)}{D(D+1)} \left(1 + \frac{4}{h^2} + \frac{1}{D} + \frac{4}{Dh^2}\right). \end{aligned}$$

For example when $h = 3$ and $D = 4$, the memory required is about 1.8 times the memory needed to hold a triangular matrix. Increasing h and D decreases this overhead factor toward 1.

Advantages of our approach

So far we have focused our discussion on the Cholesky factorization, as this is generally the rate-limiting step in Gaussian process methods. Our approach and software also improve computational efficiency by exploiting the sequential nature of Gaussian process calculations, in which each task relies on a sequence of linear algebra calculations, many or all of which can be done in a distributed fashion. Our framework generates matrices in a distributed fashion and keeps them distributed throughout a sequence of linear algebra computations, collecting results back to the master process only at the conclusion of the task. For example in likelihood calculation, we need not collect the full Cholesky factor at the master process but need only collect the (scalar) log-likelihood value that is computed using a sequence of distributed calculations (the Cholesky factorization, followed by a forwardsolve, calculation of a sum of squares, and calculation of a log-determinant). For prediction, if we have computed the Cholesky during likelihood maximization, we can use the distributed Cholesky as input to distributed forwardsolve and backsolve operations, collecting only the vector of predictions at the master process. This feature is critical both for avoiding the large communication overhead in collecting a matrix to a single processor and to allowing computations on matrices that are too big to fit on one node.

ScaLAPACK is an alternative to our approach and uses a very similar algorithmic approach. In Section 3 we show that our implementation is as fast or faster than using **ScaLAPACK** for the critical Cholesky decomposition. In some ways our implemenatation is better optimized for triangular or symmetric matrices. When storing symmetric matrices, **ScaLAPACK** requires memory space for the entire square matrix, whereas our implementation only requires a small amount more memory than the lower triangle takes. Furthermore, unlike the **RScaLAPACK** (Yoginath, Samatova, Bauer, Kora, Fann, and Geist 2005) interface (which is no longer available as a current R package on CRAN) to **ScaLAPACK**, our implementation carries out multiple linear algebra calculations without collecting all the results back to the master process and calculates the covariance matrix in a distributed fashion.

2.3. The bigGP R package

Overview

The R package **bigGP** implements a set of core functions, all in a distributed fashion, that are useful for a variety of Gaussian process-based computational tasks. In particular we provide Cholesky factorization, forwardsolve, backsolve and multiplication operations, as well as a variety of auxiliary functions that are used with the core functions to implement high-level statistical tasks. We also provide additional R functions for distributing objects to the processes, managing the objects, and collecting results at the master process.

This set of R functions provides an API for R developers. A developer can implement new tasks entirely in R without needing to know or use C or **MPI**. Indeed, using the API, we implement standard Gaussian process tasks: log-likelihood calculation, likelihood optimization, prediction, calculation of prediction uncertainty, unconditional simulation of Gaussian processes, and simulation of Gaussian process realizations conditional on data. Distributed construction of mean vectors and covariance matrices is done using user-provided R functions that calculate the mean and covariance functions given a vector of parameters and arbitrary inputs.

API

The API consists of

- basic functions for listing and removing objects on the slave processes and copying objects to and from the slave processes: `remoteLs`, `remoteRm`, `push`, `pull`;
- functions for determining the lengths and indices of vectors and matrices assigned to a given slave process: `getDistributedVectorLength`, `getDistributedTriangularMatrixLength`, `getDistributedRectangularMatrixLength`, `remoteGetIndices`;
- functions that distribute and collect objects to and from the slave processes, masking the details of how the objects are divided amongst the processes: `distributeVector`, `collectVector`, `collectDiagonal`, `collectTriangularMatrix`, `collectRectangularMatrix`; and
- functions that carry out linear algebra calculations on distributed vectors and matrices: `remoteCalcChol`, `remoteForwardsolve`, `remoteBacksolve`, `remoteMultChol`, `remoteCrossProdMatVec`, `remoteCrossProdMatSelf`, `remoteCrossProdMatSelfDiag`, `remoteConstructRnormVector`, and `remoteConstructRnormMatrix`. In addition there is a generic `remoteCalc` function that can carry out an arbitrary function call with either one or two inputs.

The package must be initialized after loading, which is done with the `bigGP.init` function. During initialization, slave processes are spawned and R packages loaded on the slaves, parallel random number generation is set up, and blocks are assigned to slaves, with this information stored on each slave process in the `.bigGP` object. Users need to start R in such a way (e.g., through a queueing system or via `mpirun`) that P slave processes can be initialized, plus one for the master process, for a total of $P + 1$. P should be such that $P = D(D + 1)/2$ for

integer D, i.e., $P \in 3, 6, 10, 15, \ldots$. One may wish to have one process per node, with threaded calculations on each node via a threaded BLAS (Basic Linear Algebra Subprograms), or one process per core (in particular when a threaded BLAS is not available). The determination of the number of cores per process is system-specific and not set at the level of R or by **bigGP**. Rather, the user must request a total number of processes and cores per process via whatever queueing system is in place (if any) on the system they are using or by specifying the hosts if simply using `mpirun`. We note that apart from requesting resources from the system, the user must specify only a single number, P, in R.

Our theoretical assessment and empirical tests suggest that the blocks of distributed matrices should be approximately of size 1000 by 1000. To achieve this, the package chooses h given the number of observations, n, and the number of processes, P, such that the blocks are approximately that size, i.e., $n/(hD) \approx 1000$. However the user can override the default and we recommend that the user test different values on their system.

Kriging implementation

The kriging implementation is built around two reference classes.

The first is a `krigeProblem` class that contains metadata about the problem and manages the analysis steps. To set up the problem and distribute inputs to the processes, one instantiates an object in the class. The metadata includes the block replication factors and information about which calculations have been performed and which objects are up-to-date (i.e., are consistent with the current parameter values). This allows the package to avoid repeating calculations when parameter values have not changed. Objects in the class are stored on the master process.

The second is a `distributedKrigeProblem` class that contains the core distributed objects and information about which pieces of the distributed objects are stored in a given process. Objects in this class are stored on the slave processes. By using a reference class we create a namespace that avoids name conflicts amongst multiple problems, and we allow the distributed linear algebra functions to manipulate the (large) blocks by reference rather than by value.

The core methods of the `krigeProblem` class are a constructor; methods for constructing mean vectors and covariance matrices given user-provided mean and covariance functions; methods for calculating the log determinant, calculating the log density, optimizing the density with respect to the parameters, prediction (with prediction standard errors), finding the full prediction variance matrix, and simulating realizations conditional on the data. Note that from a Bayesian perspective, prediction is just calculation of the posterior mean, the prediction variance matrix is just the posterior variance, and simulation of realizations is just simulation from the posterior. All of these are conditional on the parameter estimates, so this can be viewed as empirical Bayes.

It is possible to have multiple `krigeProblem` objects defined at once, with separate objects in memory and distributed amongst the processes. However, the partition factor, D, is constant within a given R session.

Code that uses the `krigeProblem` class to analyze an astrophysics data example is provided in Section 4.

Using the API

To extend the package to implement other Gaussian process methodologies, the two key elements are construction of the distributed objects and use of the core distributed linear algebra functions. Construction of the distributed objects should mimic the `localKrigeProblemConstructMean` and `localKrigeProblemConstructCov` functions in the package. These functions use user-provided functions that operate on a set of parameters, a list containing additional inputs, and a set of indices to construct the local piece of the object for the given indices. As a toy example, the package may set the indices of a matrix stored in the first process to be $(1,1),(2,1),(1,2),(2,2)$, namely the upper 2×2 block of a matrix. Given this set of indices, the user-provided function would need to compute these four elements of the matrix, which would then be stored as a vector, column-wise, in the process. Once the necessary vectors and matrices are computed, the distributed linear algebra functions allow one to manipulate the objects by name. As we have done in the `krigeProblem` class, we recommend the use of reference classes to store the various objects and functions associated with a given methodology.

3. Timing results

We focus on computational speed for the Cholesky factorization, as this generally dominates the computational time for Gaussian process computations. We run the code underlying the package as a distributed C program, as R serves only as a simple wrapper that calls the local Cholesky functions on the worker processes via the `mpi.remote.exec` function. We use Hopper, a Cray system hosted at the National Energy Research Scientific Computing center (NERSC). Each Hopper node consists of two 12-core AMD "MagnyCours" processors with 24 GB of memory. Hopper jobs have access to a dedicated Cray Gemini interconnect to obtain low-latency and high bandwidth inter-process communication. While Hopper has 24 cores per node, each node is divided into 4 NUMA regions each with 6 cores; in our experiments we try running one process per node, one process per NUMA region (4 per node), or one process per core (24 per node).

We start by considering timing as a function of the size of the problem, illustrating that our approach greatly reduces computational time and that it allows one to do computations for matrices too large to work with on an individual machine. We then consider the choice of h and compare our implementation with **ScaLAPACK**, illustrating that our implementation is equivalent in terms of timing. Next we explore the question of how to assign one's available cores: fewer processes and more cores per process or the reverse. Finally, we consider the use of GPUs.

3.1. Timing and comparison with increasing problem size

As the matrix size n increases, the arithmetic count of computations required for Cholesky decomposition increases as a function of n^3. For small problem sizes, this increase is mitigated by the greater efficiency in computing with larger matrices. Figure 3 shows how runtime varies with n.

As a practical illustration, if 100 Cholesky decompositions were required for likelihood optimization for a problem with $n = 8192$, one could carry out the optimization using our

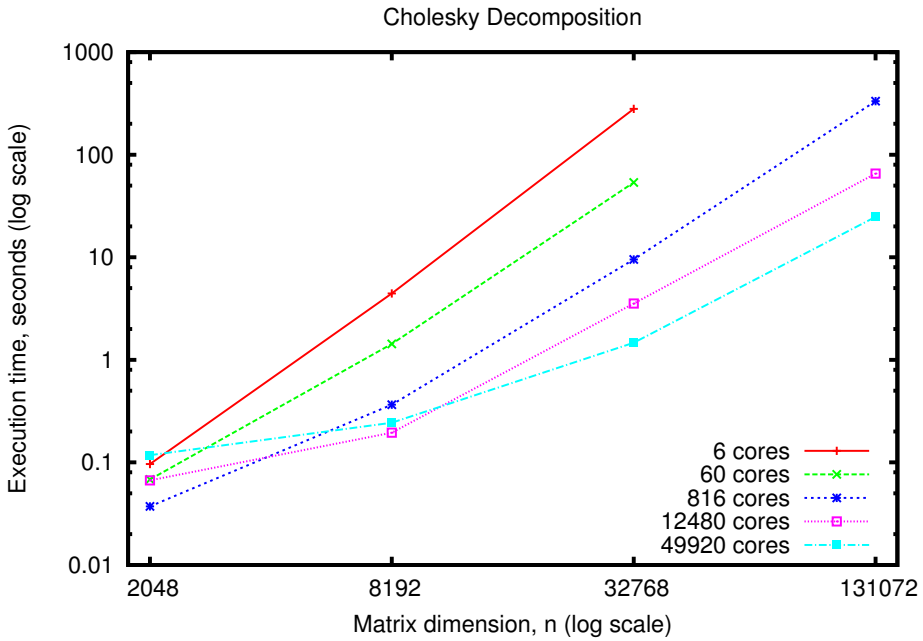

Figure 3: Runtimes as a function of n for Cholesky decomposition on Hopper, for a variety of numbers of cores. For 49920 cores, we used 24 cores per process; in the other cases we used 6 cores per process. For each data point, we use the optimal value of h, as determined in Section 3.2.

implementation with six cores on Hopper in approximately 440 seconds. This is comparable to using a fast threaded BLAS to do a standard Cholesky on a desktop machine. In particular using R on a desktop Linux machine with 16 GB RAM and 8 cores linked to **openBLAS** or on a Mac Mini with 8 GB RAM and 4 cores linked to the **vecLib** BLAS, the 100 decompositions would take approximately 500 and 610 seconds respectively. In contrast, by using 816 cores on Hopper, computational time is reduced to 40 seconds for 100 decompositions. Furthermore, with 816 cores, such an optimization with $n = 32768$ would take approximately 950 seconds with our implementation, but neither of the desktop machines could factorize the matrix because of memory constraints.

3.2. Choice of h and comparison to ScaLAPACK

In Figure 4 we compare the performance at different values of h. One notable feature is that for $h = 1$ there is no performance improvement in increasing from $P = 1$ to $P = 3$, because there is no parallelism. Allowing larger values of h makes a speedup possible with $P = 3$. Generally, larger values of h perform best when P is small, but as P grows the value of h should decrease to keep the block size from getting too small.

Figure 4 also compares our performance to **ScaLAPACK**, a standard distributed-memory linear algebra library. Performance for **ScaLAPACK** (using the optimal block size) and our algorithm (using the optimal value of h) is similar. We are thus able to get essentially the same performance on distributed linear algebra computations issued from R with our framework as if the programmer were working in C and calling **ScaLAPACK**.

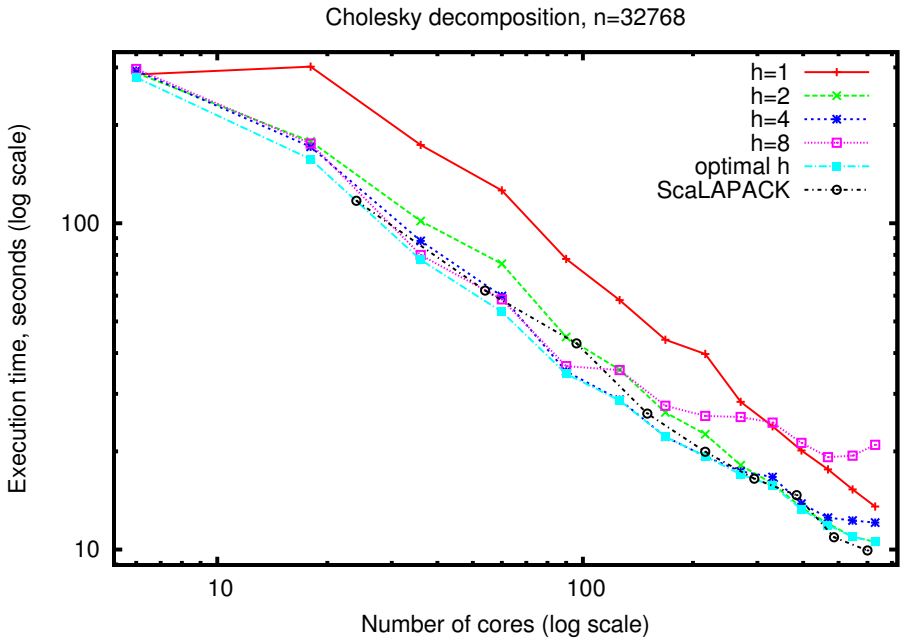

Figure 4: Runtimes for 32768×32768 Cholesky decomposition on Hopper with various values of h using 6 cores per process. The last line shows **ScaLAPACK** as a benchmark. The optimal value of h was chosen by trying all values between 1 and 8. The blocksize for **ScaLAPACK** corresponds to the best performance using a power of 2 blocks per process.

3.3. Effect of number of cores per process

Our framework gives the user the freedom to choose how many cores to assign to each process, up to the number of cores on a node. Whatever choice the user makes, all of the cores will be active most of the time. When multiple cores are assigned to a single process, parallelism between cores comes from the threaded BLAS, whereas parallelism between the processes comes from our package. Both use similar techniques to achieve parallelism. The main difference is in the communication. When running with many processes per node, each one is sending many small messages to processes on other nodes, but when running with one process per node one is more efficient in sending fewer, larger messages. As Figure 5 shows, the number of cores per process is not very important when using a small number of cores, where the calculation is computation-bound (up to about 480 cores). As the number of cores increases, the calculation becomes communication-bound, and better performance is attained with fewer processes per node (more cores per process). Note that the ideal choice of block size is affected by the number of cores per process, since efficiently using more cores requires larger blocks.

3.4. Using GPUs to speed up the linear algebra

There is growing use of GPUs to speed up various computations, in particular linear algebra, with Franey, Ranjan, and Chipman (2012) exploring the use of GPUs for likelihood maximization of Gaussian process models. Our framework can be modified to run on a single GPU or a cluster of nodes with GPUs by using **CUBLAS** (nVidia 2012) and **MAGMA** instead of

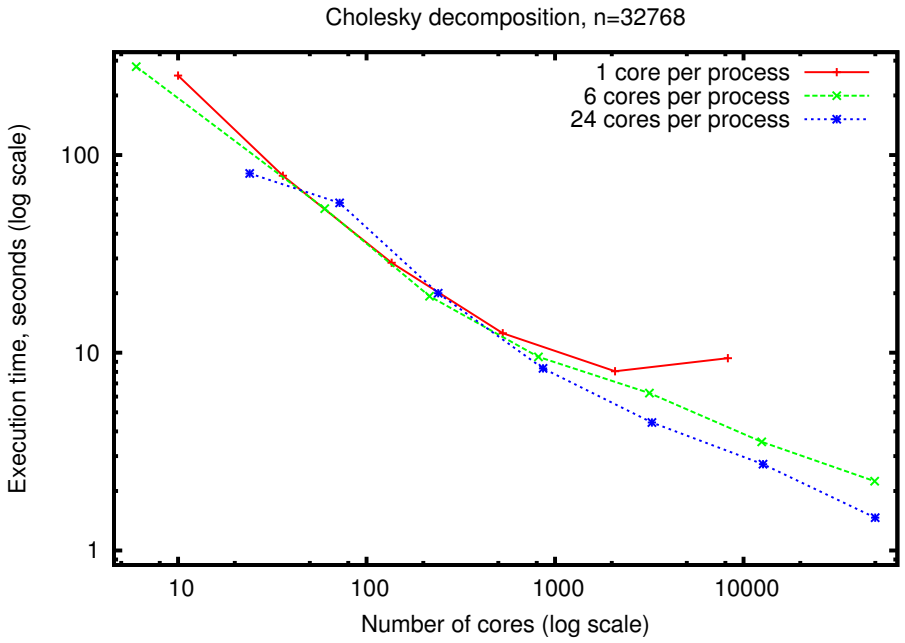

Figure 5: Runtimes for 32768 × 32768 Cholesky decomposition on Hopper using 1 core, 6 cores, or 24 cores (full node) per process. For each data point, we use the optimal value of h.

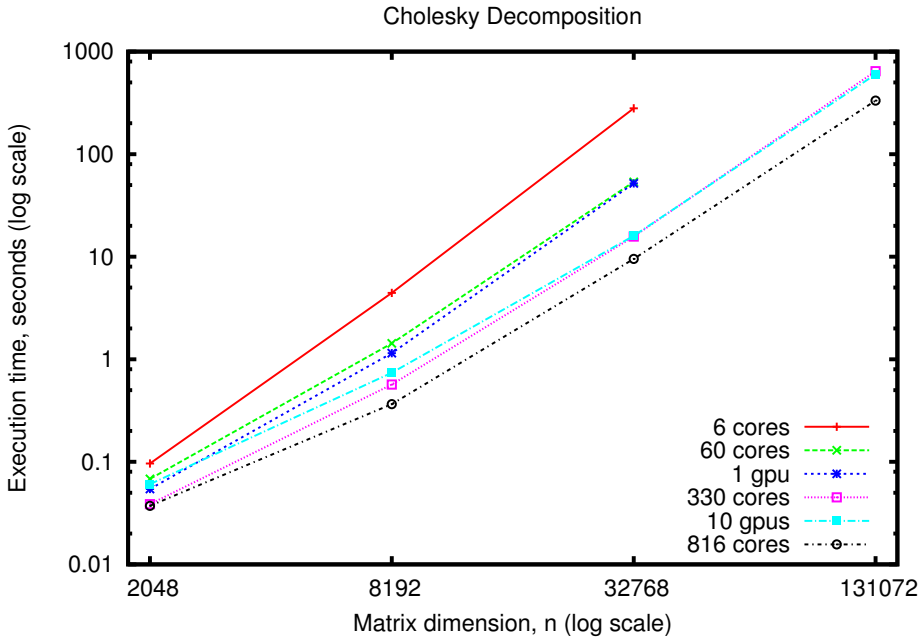

Figure 6: Runtimes as a function of n for Cholesky decomposition using 1 and 10 GPUs on Dirac with results using Hopper for a variety of number of cores. The CPU lines correspond to using 6 cores per process.

BLAS and **LAPACK**. We implemented Cholesky decomposition and tested on the NERSC machine Dirac, a small cluster with one NVIDIA Tesla C2050 GPU and two Intel 5530 CPUs per node. Theoretically each GPU has the equivalent performance of about 60 cores of Hopper, although the interconnects are slower and so more problems are communication-bound. Figure 6 compares the performance on 1 and 10 GPUs on Dirac to the performance on Hopper. One GPU is roughly the same speed as 60 cores on Hopper (matching the theoretical result), whereas 10 GPUs gives roughly the same speed as 330 cores on Hopper (showing the slow-down due to communication). When running on Dirac, the computation is entirely done on the GPUs; CPUs are only used for transferring data between nodes. In principle one could try to divide the computation between the CPU and GPU, but, since the theoretical peak of the CPUs on each node is only 15% that of the GPU, this would only yield slight performance improvements.

4. Astrophysics example

4.1. Background

Our example data set is the public spectrophotometric time series of the Type Ia supernova SN 2011fe (Pereira *et al.* 2013), obtained and reduced by the Nearby Supernova Factory (Aldering *et al.* 2002). The time series itself is a sequence of spectra, each consisting of flux and flux error in units of erg s^{-1} cm^{-2} $Å^{-1}$ tabulated as a function of wavelength in Å. Each spectrum was obtained on a different night. There are 25 unique spectra, each of which contains 2691 flux (and flux error) measurements. The total size of the data set is thus 67,275 flux values. The time coverage is not uniform, but the wavelength grid is regularly spaced and the same from night to night. The flux values themselves are calibrated so that differences in the brightness of the supernova from night to night and wavelength to wavelength are physically meaningful. The data are shown in Figure 7.

We are interested in obtaining a smoothed prediction of the flux of SN 2011fe as a function of time and wavelength along with an estimate of the prediction error. The spectrum of a supernova contains broad absorption and emission features whose appearance is the result of physical processes and conditions in the expanding stellar ejecta. The widths, depths, and heights of such features change with time as the supernova expands and cools. The wavelengths of absorption feature minima are examples of physically interesting quantities to extract from spectral time series as a function of time. These translate to a characteristic ejecta velocity that provides an estimate of the kinetic energy of the supernova explosion, something of great interest to those that study exploding stars.

To demonstrate our algorithms and software on a real data set, we extract the velocity of the Si II (singly ionized silicon) absorption minimum typically found near 6150 Å in Type Ia supernovae. This is done by finding the minimum absorption in the feature using the smoothed representation of the supernova spectrum. Realizations of the spectrum, sampled from the posterior Gaussian process, are used to produce Monte Carlo error estimates on the position of the absorption minimum. Rather than measuring the position of each minimum only at points where the data have been obtained (represented by the solid dots in Figure 7), this procedure yields a smooth estimate of the absorption minimum as a function of time interpolated between observations, while also taking observation errors into account.

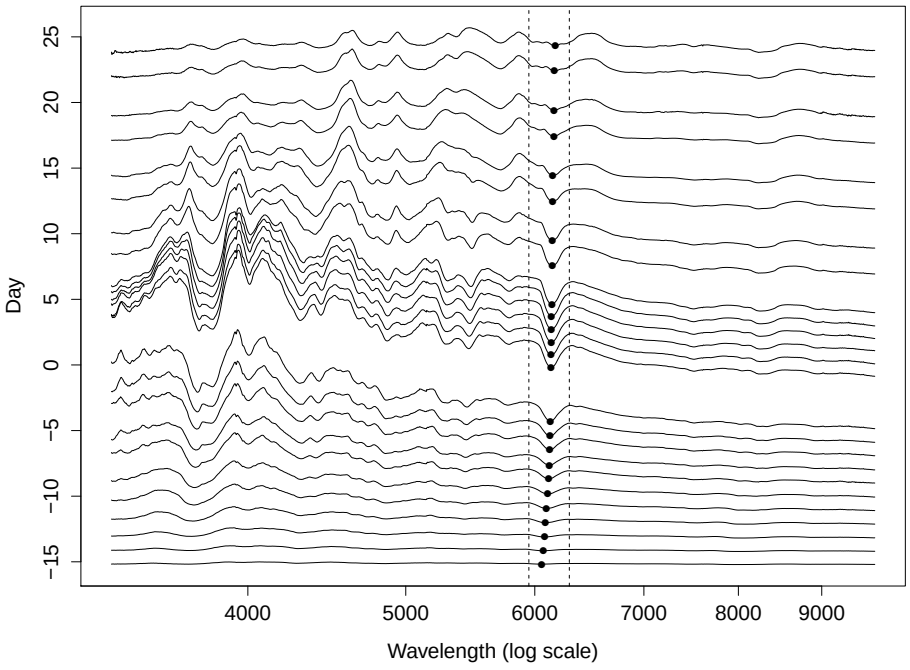

Figure 7: Flux observations for the Type Ia supernova. Offset on the y-axis corresponds to time of observation in days. The scale for the flux measurements is not indicated due to the offset, but these measurements range from -0.003 to 1. Each spectrum contains measurements corresponding to 2691 wavelengths (Å), shown on the x-axis. Wavelengths 5950 Å and 6300 Å are indicated by dotted vertical lines; this is the range over which we make predictions. The empirical minimum within this range for each spectrum is indicated with a solid dot.

4.2. Statistical model

We model the flux measurements $Y_1, \ldots, Y_{67275}$ as being equal to a GP realization plus two error components: random effects for each phase (time point) and independent errors due to photon noise. We denote these three components by

$$Y_i = Z(t_i, w_i) + \alpha_{t_i} + \epsilon_i,$$

where t_i represents the time corresponding to Y_i and w_i the log wavelength, α_{t_i} is the random effect corresponding to time t_i, and ϵ_i is measurement error for the i^{th} observation. The models for these components are

$$\begin{aligned} Z &\sim GP(\mu(\cdot;\kappa,\lambda), \sigma^2 K(\cdot,\cdot;\rho_p,\rho_w)) \\ \alpha_1, \ldots, \alpha_{25} &\overset{iid}{\sim} N(0, \tau^2) \\ \epsilon_i &\sim N(0, v_i), \quad \epsilon_1, \ldots, \epsilon_{67275} \text{ mutually independent.} \end{aligned}$$

Z has mean function μ, a function of time t only, derived from a standard template Type Ia supernova spectral time series (Hsiao *et al.* 2007), with κ and λ controlling scaling in magnitude and time. We take the correlation function, K, to be a product of two Matérn correlation functions, one for both the phase and log wavelength dimensions, each with smoothness parameter $\nu = 2$. Note that the flux error variances v_i are known, leaving us with six parameters to be estimated.

4.3. R code

For this problem we chose $P = 465$, requesting 466 processes (including one for the master) with 6 cores per process on Hopper. In the R code, the first steps are to load the package, set up the inputs to the mean and covariance functions, and initialize the kriging problem object, called `prob`. Note that in this case the mean and covariance functions are provided by the package, but in general these would need to be provided by the user.

```
R> library("bigGP")
R> nProc <- 465
R> n <- nrow(SN2011fe)
R> m <- nrow(SN2011fe_newdata)
R> nu <- 2
R> inputs <- c(as.list(SN2011fe), as.list(SN2011fe_newdata), nu = nu)
R> prob <- krigeProblem$new("prob", numProcesses = nProc, h_n = NULL,
+    h_m = NULL, n = n, m = m, meanFunction = SN2011fe_meanfunc,
+    predMeanFunction = SN2011fe_predmeanfunc,
+    covFunction = SN2011fe_covfunc,
+    crossCovFunction = SN2011fe_crosscovfunc,
+    predCovFunction = SN2011fe_predcovfunc, inputs = inputs,
+    params = SN2011fe_initialParams, data = SN2011fe$flux,
+    packages = "fields", parallelRNGpkg = "rlecuyer")
```

We then maximize the log likelihood, followed by making the kriging predictions and generating a set of 1000 realizations from the conditional distribution of Z given the observations and fixing the parameters at the maximum likelihood estimates. The predictions and realizations are over a grid, with days ranging from -15 to 24 in increments of 0.5 and wavelengths ranging from 5950 to 6300 in increments of 0.5. The number of prediction points is therefore $79 \times 701 = 55379$.

```
R> prob$optimizeLogDens(method = "L-BFGS-B", verbose = TRUE,
+    lower = rep(.Machine$double.eps, length(SN2011fe_initialParams)),
+    control = list(parscale = SN2011fe_initialParams))
R> pred <- prob$predict(ret = TRUE, se.fit = TRUE, verbose = TRUE)
R> realiz <- prob$simulateRealizations(r = 1000, post = TRUE, verbose = TRUE)
```

4.4. Results

The MLEs are $\hat{\sigma}^2 = 0.0071$, $\hat{\rho}_p = 2.33$, $\hat{\rho}_w = 0.0089$, $\hat{\tau}^2 = 2.6 \times 10^{-5}$, and $\hat{\kappa} = 0.33$. Figure 8 shows the posterior mean and pointwise 95% posterior credible intervals for the wavelength corresponding to the minimum flux for each time point. These are calculated from the 1000 sampled posterior realizations of Z. For each realization, we calculate the minimizing wavelength for each time point. To translate each wavelength value to an ejecta velocity via the Doppler shift formula, we calculate $v = c(\lambda_R/w - 1)$ where $\lambda_R = 6355$ is the rest wavelength of an important silicon ion transition and c is speed of light, $3 \times 10^8 m/s$. The posterior mean and pointwise 95% posterior credible intervals for the ejecta velocities are shown in Figure 8 (b).

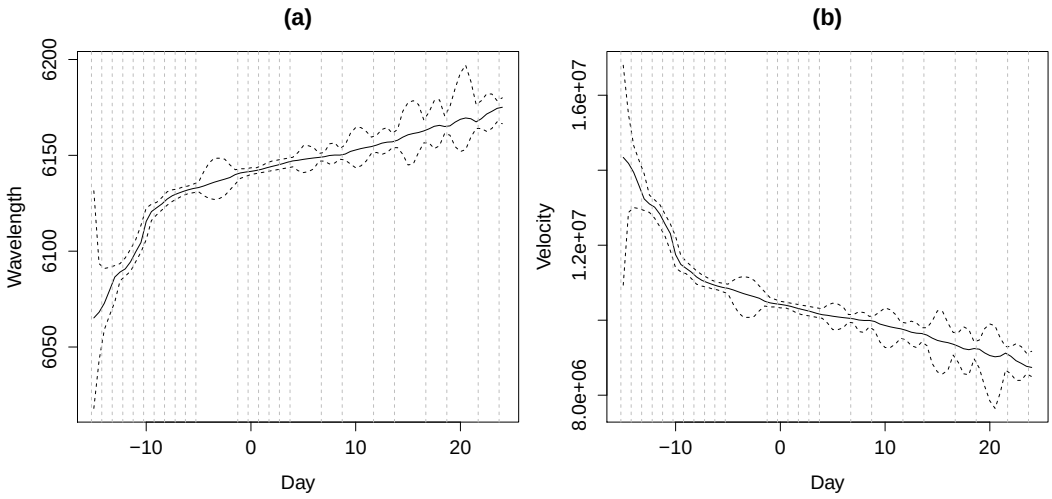

Figure 8: Posterior means (solid black lines) and pointwise 95% posterior credible intervals (dotted black lines) for the wavelength (Å) corresponding to the minimum flux (a) and the corresponding ejecta velocity (m/s) (b). The vertical gray dotted lines indicate observation times.

5. Discussion

Our software allows one to carry out standard Gaussian process calculations, such as likelihood maximization, prediction, and simulation of realizations, off the shelf, as illustrated in the astrophysics example, in situations in which calculations using threaded linear algebra on a single computer are not feasible because the calculations take too long or use too much memory. The software enables a user to implement standard models and related models without approximations. One limitation of our implementation is that we do not do any pivoting, so Cholesky factorization of matrices that are not numerically positive definite fails. This occurred in the example when simulating realizations on fine grids of wavelength and phase.

Of course with large datasets, one has the necessary statistical information to fit more complicated models, including hierarchical models, than the standard kriging methodology we implement. The package is designed to be extensible, providing a core set of distributed linear algebra functions common to Gaussian process calculations as an API usable from R without any knowledge of C or **MPI**. This allows others to implement other Gaussian process methodologies that rely on these functions. These might include Bayesian methods for nonstationary covariance models and spatio-temporal models among others. For example, MCMC updating steps might be done from the master process, with the linear algebra calculations done using the API. As can be seen from our timing results, a Cholesky decomposition for $32,768$ observations can be done in several seconds with a sufficient number of processors (a speed-up of 2–3 orders of magnitude relative to a single computational node), potentially enabling tens of thousands of MCMC updates. Or for a separable space-time model, the spatial and temporal covariance matrices could be manipulated separately using the package. One might also consider handling even larger datasets (e.g., millions of observations) by use of the core functions within the context of computationally-efficient methods such as low-rank approximations.

One useful extension of our approach would be to sparse matrices. Sparse matrices are at the core of computationally-efficient Markov random field spatial models (Banerjee *et al.* 2003; Rue, Martino, and Chopin 2009; Lindgren, Rue, and Lindström 2011) and covariance tapering approaches (Furrer *et al.* 2006; Kaufman *et al.* 2008; Sang and Huang 2012), and implementing a distributed sparse Cholesky decomposition could allow calculations for problems with hundreds of thousands or millions of locations. To get a feel for what is possible, we benchmarked the PaStiX package (Hénon, Ramet, and Roman 2000) on Hopper on sparse matrices corresponding to a two-dimensional grid with five nonzeros per row. With one million locations, a Cholesky decomposition could be done in about 5 seconds using 96 cores (4 nodes) of Hopper; for 16 million locations, it took about 25 seconds using 384 cores (16 nodes). Using more than this number of cores did not significantly reduce the running time. We also note that with these improvements in speed for the Cholesky decomposition, computation of the covariance matrix itself may become the rate-limiting step in a Gaussian process calculation. Our current implementation takes a user-specified R function for constructing the covariance matrix and therefore does not exploit threading because R itself is not threaded. In this case, specifying more than one process per node would implicitly thread the covariance matrix construction, but additional work to enable threaded calculation of the covariance matrix may be worthwhile.

Acknowledgments

This work is supported by the Director, Office of Science, Office of Advanced Scientific Computing Research, of the U.S. Department of Energy under Contract No. AC02-05CH11231. This research used resources of the National Energy Research Scientific Computing Center.

References

Aldering G, Adam G, Antilogus P, Astier P, Bacon R, Bongard S, Bonnaud C, Copin Y, Hardin D, Henault F, Howell DA, Lemonnier JP, Levy JM, Loken SC, Nugent PE, Pain R, Pecontal A, Pecontal E, Perlmutter S, Quimby RM, Schahmaneche K, Smadja G, Wood-Vasey WM (2002). "Overview of the Nearby Supernova Factory." In JA Tyson, S Wolff (eds.), *Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series*, volume 4836 of *Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series*, pp. 61–72.

AMD (2013). *Core Math Library (**ACML**)*. URL http://developer.amd.com/tools-and-sdks/cpu-development/amd-core-math-library-acml/.

Anderson E (1999). ***LAPACK** Users' Guide*, volume 9. Society for Industrial and Applied Mathematics.

Anitescu M, Chen J, Stein M (2014). "An Inversion-Free Estimating Equation Approach for Gaussian Process Models." *Technical Report ANL/MCS-P5078-0214*, Argonne National Laboratory.

Anitescu M, Chen J, Wang L (2012). "A Matrix-Free Approach for Solving the Parametric

Gaussian Process Maximum Likelihood Problem." *SIAM Journal on Scientific Computing*, **34**(1), A240–A262.

Ballard G, Demmel J, Holtz O, Schwartz O (2011). "Minimizing Communication in Numerical Linear Algebra." *SIAM Journal on Matrix Analysis and Applications*, **32**(3), 866–901.

Banerjee S, Gelfand AE, Finley AO, Sang H (2008). "Gaussian Predictive Process Models for Large Spatial Data Sets." *Journal of the Royal Statistical Society B*, **70**(4), 825–848.

Banerjee S, Gelfand AE, Sirmans CF (2003). "Directional Rates of Change Under Spatial Process Models." *Journal of the American Statistical Association*, **98**(464), 946–954.

Blackford LS, Choi J, Cleary A, D'Azevedo E, Demmel J, Dhillon I, Dongarra J, Hammarling S, Henry G, Petitet A, others (1987). ***ScaLAPACK*** *Users' Guide*, volume 4. Society for Industrial and Applied Mathematics.

Chapman B, Jost G, Pas RVD (2008). *Using* ***OpenMP****: Portable Shared Memory Parallel Programming*, volume 10. MIT Press.

Cressie N, Johannesson G (2008). "Fixed Rank Kriging for Very Large Spatial Data Sets." *Journal of the Royal Statistical Society B*, **70**(1), 209–226.

Dancik GM, Dorman KS (2008). "**mlegp**: Statistical Analysis for Computer Models of Biological Systems Using R." *Bioinformatics*, **24**(17), 1966–1967.

Dongarra JJ, Croz JD, Hammarling S, Duff IS (1990). "A Set of Level 3 Basic Linear Algebra Subprograms." *ACM Transactions on Mathematical Software*, **16**(1), 1–17.

Eddelbuettel D (2014). "CRAN Task View: High-Performance and Parallel Computing with R." Version 2014-12-05, URL http://CRAN.R-project.org/view=HighPerformanceComputing.

Franey M, Ranjan P, Chipman H (2012). "A Short Note on Gaussian Process Modeling for Large Datasets Using Graphics Processing Units." http://arXiv.org/abs/arXiv:1203.1269/.

Furrer R, Genton MG, Nychka D (2006). "Covariance Tapering for Interpolation of Large Spatial Datasets." *Journal of Computational and Graphical Statistics*, **15**, 502–523.

Gabriel E, Fagg GE, Bosilca G, Angskun T, Dongarra JJ, Squyres JM, Sahay V, Kambadur P, Barrett B, Lumsdaine A, others (2004). "Open **MPI**: Goals, Concept, and Design of a Next Generation **MPI** Implementation." In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pp. 97–104. Springer-Verlag.

Georganas E, González-Domínguez J, Solomonik E, Zheng Y, no JT, Yelick K (2012). "Communication Avoiding and Overlapping for Numerical Linear Algebra." In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pp. 100:1–100:11. IEEE Computer Society Press, Los Alamitos.

Gramacy RB (2014). ***laGP****: Local Approximate Gaussian Process Regression.* R package version 1.1-1, URL http://CRAN.R-project.org/package=laGP.

Gramacy RB, Lee HKH (2008). "Bayesian Treed Gaussian Process Models with an Application to Computer Modeling." *Journal of the American Statistical Association*, **103**(483), 1119–1130.

Gropp WD, Lusk EL, Skjellum A (1999). *Using **MPI**: Portable Parallel Programming with the Message-Passing Interface*, volume 1. MIT Press.

Hénon P, Ramet P, Roman J (2000). "**PaStiX**: A Parallel Sparse Direct Solver Based on a Static Scheduling for Mixed 1D/2D Block Distributions." In *Proceedings of the 15 IPDPS 2000 Workshops on Parallel and Distributed Processing*, IPDPS '00, pp. 519–527. Springer-Verlag, London.

Hsiao EY, Conley A, Howell DA, Sullivan M, Pritchet CJ, Carlberg RG, Nugent PE, Phillips MM (2007). "K-Corrections and Spectral Templates of Type Ia Supernovae." *The Astrophysical Journal*, **663**(2), 1187.

Intel (2013). *Intel Math Kernel Library – Documentation*. URL http://software.intel.com/en-us/articles/intel-math-kernel-library-documentation.

Kalaitzis A, Gao A, Honkela P, Lawrence ND (2014). ***gptk**: Gaussian Processes Tool-Kit*. R package version 1.08, URL http://CRAN.R-project.org/package=gptk.

Kammann EE, Wand MP (2003). "Geoadditive Models." *Journal of the Royal Statistical Society C*, **52**, 1–18.

Kaufman C, Schervish M, Nychka D (2008). "Covariance Tapering for Likelihood-Based Estimation in Large Spatial Datasets." *Journal of the American Statistical Association*, **103**, 1556–1569.

Kennedy MC, O'Hagan A (2001). "Bayesian Calibration of Computer Models." *Journal of the Royal Statistical Society B*, **63**, 425–464.

Lindgren F, Rue H, Lindström J (2011). "An Explicit Link between Gaussian Fields and Gaussian Markov Random Fields: The Stochastic Partial Differential Equation Approach." *Journal of the Royal Statistical Society B*, **73**, 423–498.

MacDonald B, Ranjan P, Chipman H (2013). "**GPfit**: An R Package for Gaussian Process Model Fitting Using a New Optimization Algorithm." URL http://arXiv.org/abs/1305.0759/.

nVidia (2012). ***CUBLAS** Library User Guide*. nVidia, v5.0 edition. URL http://docs.nvidia.com/cublas/.

Ostrouchov G, Chen WC, Schmidt D, Patel P (2012). *Programming with Big Data in R*. URL http://r-pbd.org/.

Pereira R, Thomas RC, Aldering G, Antilogus P, Baltay C, Benitez-Herrera S, Bongard S, Buton C, Canto A, Cellier-Holzem F, Chen J, Childress M, Chotard N, Copin Y, Fakhouri HK, Fink M, Fouchez D, Gangler E, Guy J, Hillebrandt W, Hsiao EY, Kerschhaggl M, Kowalski M, Kromer M, Nordin J, Nugent P, Paech K, Pain R, P écontal E, Perlmutter S, Rabinowitz D, Rigault M, Runge K, Saunders C, Smadja G, Tao C, Taubenberger S,

Tilquin A, Wu C (2013). "Spectrophotometric Time Series of SN 2011fe from the Nearby Supernova Factory." URL http://arXiv.org/abs/arXiv:1302.1292/.

Rasmussen CE, Williams CKI (2006). *Gaussian Processes for Machine Learning*. MIT Press, Cambridge.

R Core Team (2014). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. URL http://www.R-project.org/.

Rue H, Martino S, Chopin N (2009). "Approximate Bayesian Inference for Latent Gaussian Models by Using Integrated Nested Laplace Approximations." *Journal of the Royal Statistical Society B*, **71**(2), 319–392.

Sang H, Huang JZ (2012). "A Full Scale Approximation of Covariance Functions for Large Spatial Data Sets." *Journal of the Royal Statistical Society B*, **74**, 111–132.

Stein ML, Chi Z, Welty LJ (2004). "Approximating Likelihoods for Large Spatial Data Sets." *Journal of the Royal Statistical Society B*, **66**(2), 275–296.

Stone JE, Gohara D, Shi G (2010). "**OpenCL**: A parallel Programming Standard for Heterogeneous Computing Systems." *Computing in Science & Engineering*, **12**(3), 66.

Tomov S, Nath R, Du P, Dongarra J (2011). "**MAGMA** Users' Guide." ICL, UTK.

Yoginath SB, Samatova NF, Bauer D, Kora G, Fann G, Geist A (2005). "**RScaLAPACK**: High-Performance Parallel Statistical Computing with R and **ScaLAPACK**." In *ISCA PDCS*, pp. 61–67.

Yu H (2002). "**Rmpi**: Parallel Statistical Computing in R." *R News*, **2**(2), 10–14. URL http://CRAN.R-project.org/doc/Rnews/.

Zhang X (2013). ***OpenBLAS**: An Optimized BLAS Library Based on **GotoBLAS2***. URL https://github.com/xianyi/OpenBLAS/.

Affiliation:

Christopher J. Paciorek
Cari G. Kaufman
Department of Statistics
University of California, Berkeley
Berkeley, California, United States of America
E-mail: paciorek@alumni.cmu.edu, cgk@stat.berkeley.edu
URL: http://www.stat.berkeley.edu/~paciorek/, http://www.stat.berkeley.edu/~cgk/

Benjamin Lipshitz
Department of Electrical Engineering and Computer Science
University of California, Berkeley
Berkeley, California, United States of America
E-mail: benjamin.lipshitz@gmail.com

Wei Zhuo
College of Computing
Georgia Institute of Technology
Atlanta, Georgia, United States of America
E-mail: wzhuo3@cc.gatech.edu

Prabhat
Computational Research Division
Lawrence Berkeley National Laboratory
Berkeley, California, United States of America
E-mail: prabhat@lbl.gov

Rollin C. Thomas
Computational Cosmology Center
Lawrence Berkeley National Laboratory
Berkeley, California, United States of America
E-mail: rcthomas@lbl.gov

Journal of Statistical Software http://www.jstatsoft.org/
published by the American Statistical Association http://www.amstat.org/
Volume 63, Issue 10 *Submitted:* 2013-06-03
January 2015 *Accepted:* 2014-09-29