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

July 30, 2015

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.

1 Introduction

Gaussian processes are widely used in statistics and machine learning for spatial and spatio-temporal modeling (Banerjee et al. 2003), design and analysis of computer experiments (Kennedy and O'Hagan 2001). Rasmussen and Williams (2006) illustrate the use of these processes in non-parametric regression. 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 computationally 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

$$\begin{aligned} \boldsymbol{Y}|\boldsymbol{g},\boldsymbol{\theta} &\sim \mathcal{N}(\boldsymbol{g},\boldsymbol{C}_y(\boldsymbol{\theta})) \\ \boldsymbol{g}|\boldsymbol{\theta} &\sim \mathcal{N}(\boldsymbol{\mu}(\boldsymbol{\theta}),\boldsymbol{C}_g(\boldsymbol{\theta})), \end{aligned} \tag{1}$$

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 $\boldsymbol{g}$ in (1) 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.

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 on 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). 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 et al. 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 et al. 2012). Recent advances in communication-avoiding linear algebra (Ballard et al. 2011) in general, and especially for Cholesky decomposition (Georganas et al. 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 et al. 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).

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.

2.2 Our algorithm

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 $\left\lceil \frac{n}{hD} \right\rceil$, 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.

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

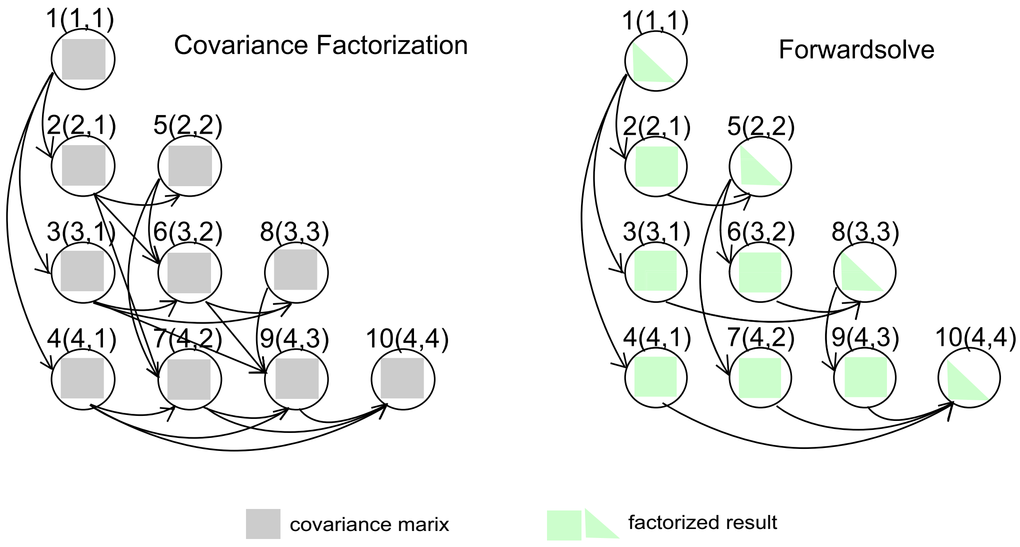

Figure 1: Diagram of the first steps in the distributed Cholesky factorization. Arrows indicate dependencies between processes. Orange coloring indicates computations carried out in each step, and green indicates parts of the matrix that are completely factored as of the previous step. The labels of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

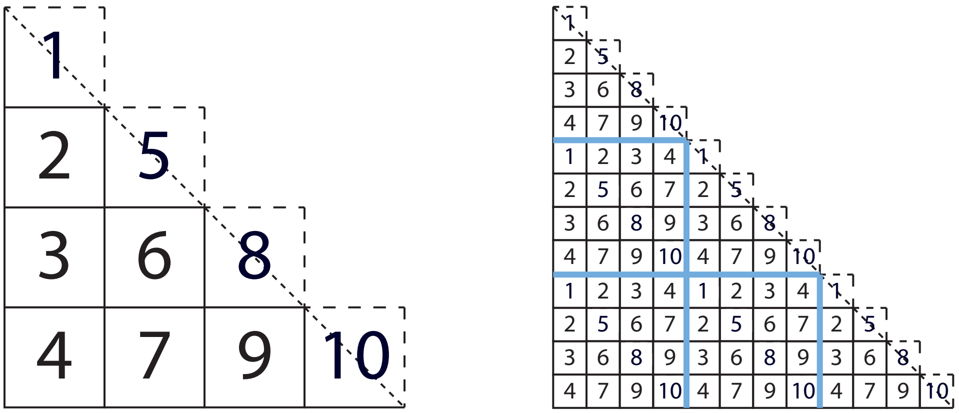

Figure 2: The matrix layout used by our algorithm with $D = 4$ and $h = 1$ (left) or $h = 3$ (right). The numbers indicate which process owns a given block. When $h = 1$, each of the 10 processes owns one block of the matrix. When $h = 3$, the blocks are a third the size in each dimension. The diagonal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

2.2.1 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

$$\begin{aligned} M &\leq \left(\frac{n}{hD}\right)^2 (h^2 + 4) = \frac{n(n+1)}{D(D+1)} \left(1 + \frac{4nD + n^2h^2 + 4n - Dh^2}{Dh^2n + Dh^2}\right) && (2) \\ &< \frac{n(n+1)}{D(D+1)} \left(1 + \frac{4}{h^2} + \frac{1}{D} + \frac{4}{Dh^2}\right). && (3) \end{aligned}$$

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.

would we like two more brief sections that work out the math for load balance and for communication cost? Also do we prefer precise calculations, as above, or asymptotic ones

2.3 The bigGP **R** package

2.3.1 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.

2.3.2 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 `remoteConstruct` 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**, 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**.

2.3.3 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.

2.3.4 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.

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.

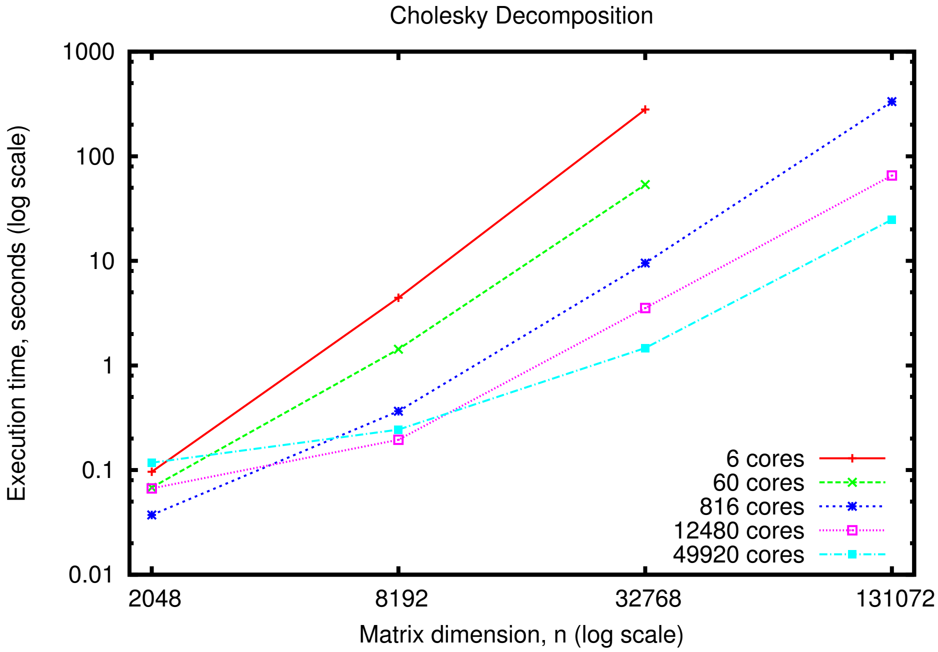

Figure 3: Runtimes as a function of n for Cholesky decomposition on Hopper, for a variety of numbers of cores. For 49920 cores, we used 24 cores per process; in the other cases we used 6 cores per process. For each data point, we use the optimal value of h, as determined in Section 3.2.

3.1 Timing with increasing problem size and comparison with a naive implementation

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 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,

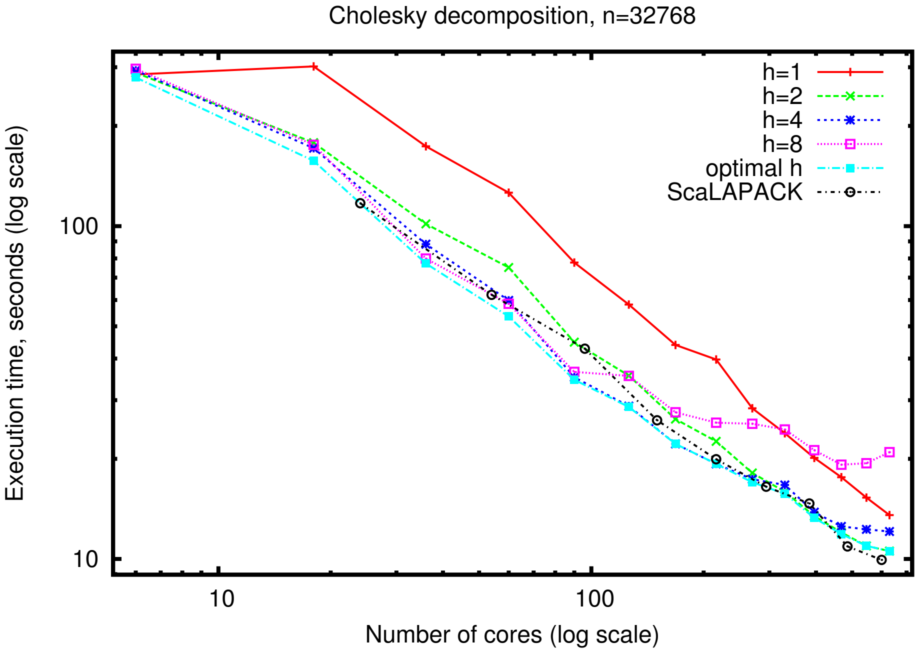

Figure 4: Runtimes for 32768×32768 Cholesky decomposition on Hopper with various values of h using 6 cores per process. The last line shows **ScaLAPACK** as a benchmark. The optimal value of h was chosen by trying all values between 1 and 8. The blocksize for **ScaLAPACK** corresponds to the best performance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

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**.

4 Astrophysics example

4.1 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

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

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.

4.2 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))
```

Table 1: Some results are here.

		Median absolute error					Median relative error				
	n	M1	M2	(p)	M3	(p)	M1	M2	(p)	M3	(p)
Test 1: Holding out all studies from 10% of countries											
mean HAZ	181	0.21	0.20	0.16	0.28	0.00	0.15	0.14	0.56	0.19	0.00
%HAZ<-2	292	6.02	5.94	0.36	8.05	0.00	0.17	0.17	0.19	0.25	0.00
%HAZ<-3	202	3.62	3.40	0.57	4.12	0.00	0.27	0.26	0.26	0.30	0.00
Test 2: Holding out mean and %<-3 when %<-2 is known											
mean HAZ	83	0.10	0.16	0.00	0.10	0.72	0.10	0.15	0.00	0.10	0.89
%HAZ<-3	111	1.76	1.86	0.00	1.76	0.19	0.16	0.19	0.01	0.16	0.30

```
R> pred <- prob$predict(ret = TRUE, se.fit = TRUE, verbose = TRUE)
R> realiz <- prob$simulateRealizations(r = 1000, post = TRUE,
      verbose = TRUE)
```

4.3 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 $\hat{\kappa} = 0.33$. These are calculated from the 1000 sampled posterior realizations of Z. For each realization, we calculate the minimizing wavelength for each time point.

Unrelated results are in Table 1.

Table 2: Results for spatial simulation (1,000 Monte Carlo simulations for each scenario and 100 bootstrap samples). The average out-of-sample R^2 is given in parentheses for each exposure model. The first column is the relative bias in estimating $\beta = 0.1$. This is the same for $\sigma^2 = 200$ and $\sigma^2 = 10$ and is estimated from 100,000 Monte Carlo samples, resulting in negligible Monte Carlo error. The final six columns show the standard deviation, average estimated standard error, and 95% confidence interval coverage, separately for $\sigma^2_\epsilon = 200$ and $\sigma^2_\epsilon = 10$.

		$\sigma^2_\theta = 200$			$\sigma^2_\theta = 10$		
	Rel Bias	SD	E(SE)	Cov	SD	E(SE)	Cov
Scenario 1							
5 degrees of freedom (0.75)							
no correction	-0.027	0.084	0.083	94%	0.02	0.019	93%
bootstrap standard error only	-0.027	0.084	0.084	95%	0.02	0.021	95%
bias correction only	-0.009	0.086	0.083	94%	0.021	0.019	93%
bias correction + bootstrap	-0.009	0.086	0.086	95%	0.021	0.021	96%
10 degrees of freedom (0.79)							
no correction	-0.039	0.08	0.08	95%	0.019	0.018	93%
bootstrap standard error only	-0.039	0.08	0.082	96%	0.019	0.027	98%
bias correction only	-0.025	0.081	0.08	94%	0.019	0.018	93%
bias correction + bootstrap	-0.025	0.081	0.088	97%	0.019	0.03	99%
Scenario 2							
5 degrees of freedom (0.42)							
no correction	-0.125	0.099	0.096	94%	0.025	0.022	87%
bootstrap standard error only	-0.125	0.099	0.097	95%	0.025	0.026	90%
bias correction only	-0.049	0.108	0.096	93%	0.028	0.022	86%
bias correction + bootstrap	-0.049	0.108	0.107	95%	0.028	0.03	94%
10 degrees of freedom (0.59)							
no correction	-0.102	0.087	0.085	93%	0.021	0.019	88%
bootstrap standard error only	-0.102	0.087	0.085	94%	0.021	0.03	94%
bias correction only	-0.061	0.091	0.085	92%	0.023	0.019	88%
bias correction + bootstrap	-0.061	0.091	0.094	95%	0.023	0.036	97%

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.

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

Mihai Anitescu, Jie Chen, and Lei Wang. A matrix-free approach for solving the parametric gaussian process maximum likelihood problem. *SIAM Journal on Scientific Computing*, 34(1): A240–A262, 2012.

Mihai Anitescu, Jie Chen, and Michael Stein. An inversion-free estimating equation approach for gaussian process models. Technical Report ANL/MCS-P5078-0214, Argonne National Laboratory, 2014.

Grey Ballard, James Demmel, Olga Holtz, and Oded Schwartz. Minimizing communication in numerical linear algebra. *SIAM Journal on Matrix Analysis and Applications*, 32(3):866–901, 2011.

S. Banerjee, A.E. Gelfand, and C.F. Sirmans. Directional rates of change under spatial process models. *Journal of the American Statistical Association*, 98(464):946–954, 2003.

B. Chapman, G. Jost, and R. Van Der Pas. *Using **OpenMP**: Portable Shared Memory Parallel Programming*, volume 10. The MIT Press, 2008.

Garrett M Dancik and Karin S Dorman. **mlegp**: Statistical analysis for computer models of biological systems using **R**. *Bioinformatics*, 24(17):1966–1967, 2008.

E. Georganas, J. González-Domínguez, E. Solomonik, Y. Zheng, J. Touriño, and K. Yelick. Communication avoiding and overlapping for numerical linear algebra. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 100:1–100:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press. ISBN 978-1-4673-0804-5. URL `http://dl.acm.org/citation.cfm?id=2388996.2389132`.

E.Y. Hsiao, A. Conley, D.A. Howell, M. Sullivan, C.J. Pritchet, R.G. Carlberg, P.E. Nugent, and M.M. Phillips. K-corrections and spectral templates of Type Ia supernovae. *The Astrophysical Journal*, 663(2):1187, 2007.

M.C. Kennedy and A. O'Hagan. Bayesian calibration of computer models. *JRSSB*, 63:425–464, 2001.

C.E. Rasmussen and C.K.I. Williams. *Gaussian Processes for Machine Learning*, volume 1. The MIT Press, Cambridge, Massachusetts, 2006.