A tutorial for using `Rmpi` on the NCAR/Wyoming supercomputer

Nathan Lenssen, Douglas Nychka, Dorit Hammerling, Seth McGinnis [*]

February 26, 2016

Abstract

The recent growth in the size and complexity of climate data as well as the availability of cluster and supercomputing resources has brought parallel programming to the attention of geostatisticans. The National Center for Atmospheric Research's (NCAR) supercomputer facility is an accessible and powerful resource for any scientist or statistician working in climate-related fields. This technical report presents a tutorial for users of the statistical language `R` to quickly utilize the massively parallel computational resources provided by the NCAR Yellowstone supercomputer through the `Rmpi` package. We provide fully functional scripts on the Yellowstone system that serve as examples of flexible implementations of the capabilities of massively parallel `R` scripts. Additionally, the tutorial serves as a guide for running parallel `R` scripts on any cluster or supercomputer with MPI capabilities.

Keywords: Parallel Computing, R Programming, High Performance Computing, Spatial Statistics

[*]Nathan Lenssen is PhD Student, Columbia University Department of Statistics 1255 Amsterdam Avenue New York, NY 10027 (n.lenssen@columbia.edu), Douglas Nychka, is Senior Scientist, National Center for Atmospheric Research, PO Box 3000, Boulder CO 30307-3000 (nychka@ucar.edu), Dorit Hammerling is Project Scientist II, National Center for Atmospheric Research, PO Box 3000, Boulder CO 30307-3000 (dorith@ucar.edu), Seth McGinnis is Associate Scientist, National Center for Atmospheric Research, PO Box 3000, Boulder CO 30307-3000 (mcginnis@ucar.edu).

Contents

List of Figures

List of Tables

1 Introduction

When working with climate data, we often need to analyze large datasets over time and/or space. Many of these calculations are *embarrassingly parallel*,[1] that is, consist of many independent calculations that can be performed simultaneously. There are various methods and languages available to write parallel algorithms, but they generally require the investment of learning a new language. The `Rmpi` package provides users who have already written data analysis scripts or developed a software base in `R` with a simple and effective way to transition to parallel computing in a morning's work. The existing code can remain in `R` while the package takes care of the lower level parallel functionality. Although using `Rmpi` limits the parallel algorithms that can be implemented, for many computations the speedup is still likely to be significant, provided that the individual tasks are efficient. Moreover, this approach has the advantage that subsequent serial analysis based on the results of the parallel steps can be seamlessly handled in the supervisor `R` session managing the computation.

This report demonstrates how to use `Rmpi` through two examples applied to a large climate data set. The first estimates some statistical parameters for several days of observed daily temperature fields and illustrates a workflow for creating parallel `R` code. The second is a more extensive analysis of the observed temperature data for North America aimed at creating a gridded data product from station observations. This second project is an adjunct to the North American Regional Climate Change Assessment Program (NARCCAP) and is designed to make the creation of gridded climate analyses more transparent and flexible. In both examples, the workflow of submitting a batch file on the supercomputing system, running the supervisor `R` session, and using `Rmpi` to perform parallel calculations remains the same.

This tutorial assumes familiarity with `R` and some spatial statistics. However, it attempts to be largely self-contained with respect to describing the supercomputing environment and the batch processing of jobs. Running the examples requires only a few changes to the supplied code. Besides reproducing the example output the reader can also experiment by modifying the code in different ways. The organization of the `R` code for the `Rmpi` package may also suggest some useful techniques to use in organizing other complex data analysis projects. Finally, although we have written this report for a specific supercomputer environment, we expect the principles to be the same for other supercomputers.

While this paper is designed in a generally linear manner, the large amount of information presented may require multiple read-throughs to get up to speed on how `R` can be used in the *Yellowstone*(YS) environment. It is also important to note that sections 4 and 5 present two different, instructive implementations of `Rmpi`. Section 4 presents a more abstract script that requires few changes to work with other applications. Section 5 outlines a less elegant, but more linear and readable script that may provide better intuition to first time users of `Rmpi`. The following is a brief outline of the paper provided to give the reader a general idea of when key information is given.

- **Section 2** walks the user through the setup of their personal environment on Yellowstone and lists the necessary unix skills. A short example of running a batch script on YS is provided to check that the reader has set up their environment correctly. We leave all explanation of the example to Section 4, so the reader may proceed once they are able to run the script without errors and leave the understanding for later in the paper.

[1]Although *embarrassingly* seems like a negative term, it is not. A more descriptive alternative might be *conveniently* parallel.

- **Section 3** introduces the `Rmpi` package and covers concepts dealing with parallel programming that are critical for any parallel algorithm, especially those dealing with random numbers. We also introduce the function family `mpi.apply` which we use for all `Rmpi` embarrassingly parallel scripts.
- **Section 4** steps slowly through the bash and `R` scripts used in the example first presented in Section 2. As mentioned, this implementation is more abstract and versatile than the Section 5 code. It should serve as a model for a reader's ultimate code, but may be more confusing at first read.
- **Section 5** works through a "real world" geospatial statistics example providing intuition on both the code and the mathematics behind the thin-plate spline. The scripts are verbose and linear allowing the reader to see the relationship between the necessary pieces in a `Rmpi` workflow.

2 Setting up and getting started

In this report we refer to *yellowstone*[2] (YS), the main interactive logon and batch computing facility associated with the NCAR/Wyoming Supercomputing Center (NWSC). This Center provides computing resources for university researchers and NCAR scientists in atmospheric, oceanic, and related sciences that are larger in scale than is typically available at a university. To access these resources, users must go through a formal application procedure to receive an allocation and user account[3]. The allocation policy, however, is structured to reach a broad group of relevant users ranging from individual graduate students to entire research groups pursuing large computational campaigns. Part of the goal of this tutorial is to make it easy for data scientists and statisticians to tackle large projects that are appropriate for YS. YS also serves as an interactive gateway to submit batch jobs on other smaller clusters in NWSC, such as the *geyser* cluster, which is configured specifically for data analysis but limited in the number of cores. YS interactive sessions can also be used to test and debug serial versions of `R` code using the interactive mode for `R`, provided that the computations are restricted to be small.[4]

YS uses the unix operating system, and you will need some knowledge of a few unix shell commands to work there. There are many good tutorials for unix but as an absolute minimum, make note of

- `ls` for listing files in a directory
- `cd` for changing directories,
- `cp` for copying files
- `more` for listing text files
- `scp` or `sftp` to upload and download files between YS and your local system (or PC).

[2]See `https://en.wikipedia.org/wiki/Yellowstone_(supercomputer)` Yellowstone is a 1.5-petaflop IBM iDataPlex cluster computer with 4,518 dual-socket compute nodes that contain 9,036, 2.6-GHz Intel Xeon E5-2670 8-core processors (72,288 cores) with an aggregate memory size of 144.6 terabytes.

[3]Send questions about allocation opportunities to alloc@ucar.edu.

[4]Although one can develop an entire code on YS we recommend that the bulk of development and testing be done on a separate computing platform, such as a local unix cluster or personal computer. The examples have been structured so that only a small portion of the analysis needs to be tested by running on YS in batch mode.

You will also need to have some rudimentary skill with a text editor (such as `vi`, `emacs`, or `pico`) to make small changes to text files on YS. You log in to YS using `ssh`, and start and manage batch jobs from the interactive session using LSF commands such as `bsub`, which submits a batch job to the supercomputer. You can also run `R` as normal in an interactive session, but interactive `R` sessions should be used sparingly, primarily on small tasks that don't use `Rmpi`.

The code examples described throughout this report can be found at `/glade/u/home/nychka/HPC4Stats` on YS's file system. The best way to work through this tutorial is to copy this directory so you can make changes to the examples and run them yourself. Here is how we suggest making a local copy of the tutorial.

```
# move to your home directory
cd ~
# copy the tutorial directory and its contents
cp -r /glade/u/home/nychka/HPC4Stats .
# list the contents of the copied directory
ls -lR HPC4Stats
```

The '-r' option indicates a recursive copy that includes all subdirectories; '-lR' gives a recursive listing with extra details, and of course # indicates a line that is just a comment.

The `HPC4Stats` directory contains two subdirectories: `RmpiExample` and `RmpiMICA`, the first and second examples in this report. As a quick start to check that your environment is set correctly try to run the first example:

- Change directories to RmpiExample:
  `cd ~/HPC4Stats/RmpiExample`
- Edit the file startGeyser.lsf and change P86850053 to the project code for your YS allocation.
- Submit the batch job to geyser:
  `bsub <  startGeyser.lsf`

The job should take a few minutes to run and you can check the status of the job using the `bjobs -w` command. Note that some of this time is spent waiting for the queueing system to schedule the batch job to start running. Therefore your time to wait depends on who else is using the system and can be unpredictable. While the job is running, you may see a number of different logfiles in the directory; they will be deleted when the job is finished.

If the job has run successfully, you will see new files with the `.out` and `.Rout` extensions with file names corresponding to the `$JOBID` shown in the output from `bjobs`. These contain output from the queueing system and the `R` program, respectively. If there is a file ending in `.err`, check it to see if anything went wrong. Since errors can occur at either the unix or R level, looking back and forth between the `.Rout` and `.err` files may be necessary for debugging. An `R` binary output file named `result.rda` should have been written to your user scratch directory[5].

To view the output, make `R` available in your interactive session by typing

```
module load R/3.0.1
```

at the UNIX prompt. Now type `R` and in your R session use the command

[5]Besides their home directory, every user has a directory in `/glade/scratch/`*username* . This directory is reserved for large, temporary files and is convenient for holding intermediate steps in a data analysis project. Individual files are removed from /glade/scratch automatically if they have not been accessed (modified or read, for example) in more than 75 days. So it is important to move these files to a permanent location if they are to be saved.

Table 3.1: Getting started with Rmpi.

```
  library(Rmpi)

# create workers
  mpi.spawn.Rslaves(nworkers = 2)

# a simple task function
  doTask <- function(id, sd = 1.0){
      set.seed(123 + id)
      sd(rnorm(100, sd = sd))
  }

# Do 20 tasks, assigning them to workers as they become available.
  output <- mpi.applyLB(1:20, doTask)

# save results as an R object
  save(output, file = "output.rda")
```

load('/glade/scratch/*username*/result.rda')

to read this output file. Take a look at the objects dataFileNames, result, and timingStats and match these up with what was created in the .Rout file.

3 Rmpi Introduction

The Message Passing Interface (MPI) is a protocol for communicating between individual processors (or *cores*) in parallel systems. It is commonly used in high-performance computing, and nearly every supercomputer will have some form of the MPI library installed. MPI can also be used on personal computers that have multiple processors. Although MPI is primarily invoked by writing low-level C or FORTRAN code, it is accessible in R through the Rmpi package. As the name implies, MPI is a framework for transferring data between separate processes. It is typically faster than writing and reading files to exchange information but slower than accessing information that is held in common memory.

Rmpi takes advantage of MPI by utilizing a supervisor-worker model.[6] The R session started by the batch job is the supervisor, and runs like any other interactive or batch R session. The supervisor uses MPI to spawn workers across the available processors. Each worker is an independent, fully-functional R session. By communicating to the workers using Rmpi, you can load packages, transfer data objects, and execute functions just as in any normal R session. Note that the only way to communicate with a worker is via an Rmpi command in the supervisor session. This limitation greatly simplifies the use of MPI in R for embarrassingly parallel problems, but makes the execution of communication-reliant parallel algorithms difficult to code and nearly impossible to optimize.[7]

The R script in Table 3.1 illustrates a simple use of Rmpi from within a supervisor session. The steps in this short example are common to any use of Rmpi: load the library, spawn the workers, define the tasks, manage the tasks among the workers using a version of apply, and save

[6]Historically this has been referred to as a master-slave model but we prefer the more modern designation supervisor-worker. At the time of writing the Rmpi package still uses master/slave in its functions and documentation.

[7]It is an open question how much and how frequently data can be passed to worker sessions on YS without degrading overall performance.

the results. The worker task is to simulate 100 iid standard normal random variables and return the standard deviation. Note how the `id` variable is used to reset the RNG seed so that different (i.e., independent) streams of random numbers are generated. The script only creates two workers (`nworkers=2`), so the 20 tasks will be split roughly in half between them. The `output` object will be a list of the 20 standard deviations computed by the workers.

There are three important points to keep in mind with regard to this script. First, when working in the YS environment, it is essential to allocate enough cores to match the number of `R` worker sessions spawned by the supervisor. Bad things happen when the supervisor attempts to spawn more workers than the available cores (or threads). Second, this script is general; you can run it on any cluster or personal computer that has multiple cores and `Rmpi` installed. In fact, that's why it only spawns two workers: so it can be tested even on a MacBook Air! Finally, we need to be careful when dealing with parallelization involving randomness, particularly in Monte Carlo applications. Manually allocating each task a unique seed is a simple way to guarantee different sets of random numbers.

3.1 Using `mpi.apply` functions

The primary function we use to make MPI calls is from the `mpi.apply` family of functions. These functions are called similarly to the vanilla `apply()` functions from the `R` base package, and handle all of the parallelism behind the scenes. Where vanilla apply functions loop over the elements of list or rows of a dataframe, the MPI-based apply loops over task IDs.

The exact function used in this project is `mpi.iapplyLB()`, which takes advantage of two MPI features for better performance. The `i` signifies the function is nonblocking, which allows the supervisor process not to use any processor power while it is waiting for the workers. Do not use this feature if the supervisor will be frequently managing communications between workers or spawning new jobs. In our examples, the supervisor never passes communications between nodes, so it makes sense to use non-blocking `iapply` so that a worker can be overlaid on the same core as the supervisor. This feature is useful when the number of available cores is small, and is not as important for YS, where hundreds to thousands of cores are available.

The `LB` signifies that the function is load-balancing. If there are more tasks than workers, the supervisor will act as a scheduler, sending out a new task to a worker whenever the worker completes its current task. If you do not use load balancing, you must ensure that there are as many workers as there are tasks. Generally, `mpi.iapplyLB` is the preferred flavor of `mpi.apply` to use when performing embarrassingly parallel tasks. Because of the complexity of interconnections and system functions on YS, the amount of time taken by a given task can be somewhat unpredictable, and load balancing helps make good use of resources in the face of this variability.

In summary a task assigned to a worker will have a unique task ID and an `R` function to execute. The worker must use the ID, in conjunction with other data, to figure out what it needs to do. This can be as simple as using the ID to determine which file in a list to read. This simple strategy is demonstrated by the example in Section 2. At the other extreme, `R`'s ability to dynamically construct and execute function calls[8] could be used to load unique sets of functions, packages, and data sets, based solely on the task ID. In either case, the key to defining the worker's task is how the top level function uses the task ID.

[8]This is referred to as "computing on the language"; see, for example, `do.call`.

Table 4.1: Batch script `startGeyser.lsf` used to run the first example

```
#!/bin/bash
#BSUB -J exampleTest
#BSUB -q geyser
#BSUB -n 16
#BSUB -R "span[ptile=16]"
#BSUB -P UIMG0001
#BSUB -W 00:30
#BSUB -o %J.out
#BSUB -e %J.err

source /glade/u/apps/opt/lmod/4.2.1/init/bash
module load impi
module load R/3.0.1

HOSTFILE=/glade/scratch/$LOGNAME/.host.$LSB_JOBID
rm -f ${HOSTFILE}
echo $LSB_MCPU_HOSTS|awk '{for(i=1;i<NF;i+=2)print $i}' > ${HOSTFILE}
nmpd=$(wc -l ${HOSTFILE}|awk '{print $1}')
mpdboot -n ${nmpd} -f ${HOSTFILE}
mpiexec -n 1 R CMD BATCH --no-save supervisorBatch.R info.$LSB_JOBID.Rout
mpdallexit
rm -f ${HOSTFILE}
```

4 A spatial statistics example

The small example we ran at the end of Section 2 is designed to run on the data analysis cluster *geyser*, although simple modifications will allow the same code to run on YS nodes.[9] There are three basic parts to this example: the `lsf` script that initiates the `R` batch process for the supervisor session (Table 4.1), the `R` script run by the supervisor (Table 4.2), and the functions used to define the worker tasks. The following subsections explain each of these three parts in more detail.

4.1 `lsf` batch script

The file `startGeyser.lsf` (Table 4.1) has the information needed to start a batch job from an interactive unix session on YS.[10] The unix command

```
bsub < startGeyser.lsf
```

runs the `bsub` command with the file as input. That is, the `bsub` program will take as input the file `startGeyser.lsf` which is reproduced as Table 4.1.

The BSUB comments at the beginning of the script specify options that control how LSF schedules the job. The options are:

-J The name of the job.

[9]Geyser is used here because it was a more ready resource for class access in a STATMOS summer course on this topic.

[10]See `http://www2.cisl.ucar.edu/resources/yellowstone` for an introduction to YS and the LSF batch queuing system.

-q Which resource queue to submit the job.

-n The number of cores to use. (On YS a node has 16 cores, and nodes are not shared between users, so n should always be a multiple of 16 or you'll be charged for core-hours you're not using.)

-R This option specifies how the processes are distributed across the cores in a node. For these applications, we use each of the 16 cores on each node for a separate process.

-P The project code (allocation) to charge core hours against.

-W The maximum runtime for the job in "wall clock" time. LSF cancels the job when it reaches this limit, so it's important not to underestimate it. However, if you give a time that is a large overestimate, the job may have a long wait in the batch queue for a sufficiently large block of time to become available. If this is a problem, you can also use the -We flag to provide an estimated runtime.

-o The output file name. %J becomes the batch job ID.

-e The file name for error messages. (Unix error messages are written to stderr; `R` error messages are written to the .Rout file described below.)

The second part of this script has commands that are executed by a unix shell. In this case the script loads: the `bash` unix shell, the right version of MPI, and a specific version of `R`. The `Rmpi` package must be reinstalled when a new version of `R` is installed, but it must also be matched to the right version of the MPI library, so it's important to check that everything is available and working before updating this part of the script to a new version of R.

The final section of the batch file sets up the environment to execute `R` in batch mode and uses MPI via the `mpiexec` line. Here the `R` batch script (i.e., the supervisor session) is `supervisorBatch.R` and the output from the `R` script is written to a file using the job name and ending in `.Rout`. You can change the names of the scripts and output files if needed, but otherwise the code in this section should not be modified while working on YS.

4.2 `R` code for example

Table 4.2 shows a shortened version (omitting timing blocks) of the file `supervisorBatch.R`, which is used for the supervisor session of `R`. The code up to the comment banner defines task- and user-dependent functions and variables. After the banner, the supervisor sets up the workers and executes the function `doTask` with IDs 1 through `nTasks` in parallel across the workers.

Before the workers can execute tasks, the supervisor must first send them the functions and data they will use in the tasks. For clarity, this code collects the names of all the `R` objects that will be sent to the workers in the `namesDataObjects` vector above the comment banner.

The `nWorkers` and `nTasks` variables control the size of the `Rmpi` application. A potential error here is if the number of workers exceeds the number of cores requested by the `-n` parameter in the lsf batch script.[11]

Before spawning workers, the script sources the `setupSupervisor.R` script, which should contain code that defines the `doTask()` function and sets up any supporting functions and data. After the workers have been spawned, the code loads the `fields` package on each worker. Modify this

[11] We check for this kind discrepancy using the unix environment variable LSB_MAX_NUM_PROCESSORS and `Sys.getenv()` in `R`.

Table 4.2: `R` Batch script to run the `Rmpi` Example

```r
# load  packages in supervisor session
  library( fields)
  library(Rmpi)
# recommend  cleaning out workspace:
  remove( list = ls() )
# task and computer dependent files
  dataDir  <- "/glade/p/work/mcginnis/mica/data/daily/narccap/tmax"
  infoFile <-
  "/glade/p/work/mcginnis/mica/data/daily/narccap/index.tmax.narccap"
  outputDir <- paste0("/glade/scratch/",Sys.getenv("USER"))
  sourceDir <- "~/HPC4Stats/RmpiExample"
# data object names to broadcast to workers:
  namesDataObjects <-
     c( "doTask",
        "getData",
        "stationInfo",
        "dataFileNames",
        "domain"
      )
  nTasks  <- 32
  nWorkers <- 16
####################################################
####### following is independent of tasks #########
####################################################
  source( paste0(sourceDir,"/","setupSupervisor.R") )
# Spawn the workers
    mpi.spawn.Rslaves(nslaves = nWorkers)
# load fields package onto workers
    mpi.bcast.cmd( library(fields) )
# loop over the list of data objects and broadcast to workers
  for( objName in namesDataObjects ) {
    cat("broadcasting: ", objName, fill = TRUE)
    do.call( "mpi.bcast.Robj2slave", list(obj = as.name(objName)))
  }
# here is where the heavy lifting happens
  result <- mpi.iapplyLB( 1:nTasks, doTask)
#
# Don't forget to save!
  save(result,dataFileNames, file = paste0(outputDir,"/","result.rda"))
# close up everything (e.g. close up worker sessions).
  mpi.close.Rslaves()
  mpi.quit()
```

Table 4.3: `doTask` in the `Rmpi` Example

```
# The routine that is parallelized in supervisorBatch.R
doTask <- function(taskID){
  obj <- getData(taskID)
  out <-  Tps( obj$x, obj$y, Z=obj$Z)
        outSummary <- summary( out)
  return( outSummary)
}
```

part of the code if you need to omit `fields` or load other packages. The workers are then assigned tasks using the `mpi.iapplyLB()` function, as in the previous example. After all the tasks have executed, the results are saved as a binary `R` object in the output file.

4.3 The setup script and the `doTask` function

The file `setupSupervisor.R` contains `R` source code that sets up some tables for access to the station data, subsets the spatial domain, and then defines two functions. For this application, `getData` reads in the k^{th} day of the station data where k is the passed task ID. This mechanism is how the worker locates the correct data set to analyze. For a given task ID, `doTask` calls `getData` and then does the analysis on that day's data.

The R function `doTask` (Table 4.3) is a straightforward application of fitting a thin-plate spline to a spatial data set. The data model is

$$\boldsymbol{y}_i = p(\boldsymbol{x}) + g(\boldsymbol{x}_i) + \boldsymbol{e}_i$$

where $\boldsymbol{y}_i$ is the i^{th} spatial observation made at location $\boldsymbol{x}_i$, g is a smooth function over space, and $\boldsymbol{e}_i$ is uncorrelated measurement error with mean 0 and constant variance σ^2. The goal is to estimate the function $p(\boldsymbol{x}) + g(\boldsymbol{x})$ at $\boldsymbol{x}_i$ and also at points, $\boldsymbol{x}$, where observations are not available. The advantage of a statistical model is that one can derive measures of the uncertainty in these estimates.

A thin-plate spline is a form of spatial prediction where p is assumed to be a low order polynomial in the coordinates and g is assumed to be a smooth Gaussian process. The trade-off between the variance in the observations explained by g and the variance coming from the measurement error depends on a non-negative parameter, `lambda`, that sets the amount of smoothing. For `lambda = 0`, the spline surface will fit the data exactly – that is, the measurement error is assumed to be zero. For `lambda` large, the spline surface will tend towards a linear function where the coefficients are fit by least squares. Of course, interesting cases are often when the value for `lambda` suggests an intermediate smoothing between these two extreme cases.

The spline method also has an interpretation in terms of more conventional geostatistical models. Specifically, in this example the covariance of the Gaussian process model for g is a special case of the Matérn covariance as the range parameter increases to infinity, and the smoothness parameter (ν, not to be confused with `lambda`) is 2.0. In addition, p is a linear function of the coordinates. Given this correspondence, the `lambda` parameter can be interpreted as the "noise to signal" ratio in the model, or as the ratio of σ^2 to the variance of g.[12]

[12]The rigorous specification for the covariance of g is that $VAR(\sum_j g(\boldsymbol{x}_j)\beta_j) = \rho \sum_{i,j} K(||\boldsymbol{x}_i - \boldsymbol{x}_j||)\beta_i\beta_j$ where $K(d) = d^2 log(d)$. $\boldsymbol{\beta}$ and $\{\boldsymbol{x}_i\}$ are restricted such that $\sum_j p(\boldsymbol{x}_j)\beta_j = 0$ for all linear p. Here $\lambda = \sigma^2/\rho$.

With this specification, the spline estimator is equivalent to the method of *Kriging* in geostatistics. Estimating the parameter `lambda` is the most computationally demanding part of this analysis and is the focus of this example. Under the assumption that each day is independent, estimating `lambda` is an embarrassingly parallel task, which gives the motivation for using `Rmpi`. The `Tps()` function estimates lambda using both generalized cross-validation and restricted maximum likelihood. The `doTask()` function fits the spline but only returns the summary of the fit instead of a complete description of the spline surface. This summary is enough to examine the estimated parameters in the spatial model for a first exploratory step and avoids passing back large fitted objects. See the help file and examples in the `fields` package for more background on the `Tps()` function.

4.4 Organizing the data

The observational data is derived from the Global Historical Climatology Network (GHCN) dataset distributed by the National Climatic Data Center.[13] It consists of daily values of precipitation and near-surface minimum and maximum air temperatures as recorded by weather stations around the world. The data is distributed in text format as records for individual stations. We have preprocessed the data to reformat it, remove redundant information, and combine all available station data for North America on a given day. These preprocessed data files are available on YS's filesystem:

```
/glade/p/work/mcginnis/mica/data/daily/narccap
```

The maximum temperatures are found in the subdirectory `tmax`. Within `tmax/` the daily data files are named by date in the format YYYYMMDD, so that the observations from January 19, 1970 can be found in the file `19700119`. These files contain only a station ID and the corresponding observation. The file `index.tmax.narccap` contains the station ID, location, elevation, and start and end dates for each station record. This organization of the data is more compact and easier to ingest than its original format, but still makes it easy to associate locations with each observation. The preprocessing also removes any missing values and subsets the available data to a specific subregion of North America. To make this a small size example , we consider only the Western US.

Figure 4.1 is a `quilt.plot` showing `tmax` observations on January 19, 1970. The source code for this plot is given in Table A.1.

4.5 Sample output for a larger example

To better evaluate the computational speed of this approach, we ran a larger example, estimating `lambda` for five years of data using 120 cores (`nWorkers = 120` and `nTasks = 1825`). The batch script `startYS.lsf` calls the slightly modified supervisor script `supervisorBatch2.R`.[14] An important difference here is we use the `regular` queue on YS instead of `geyser`, which allows the use of many more cores. The only changes in the supervisor `R` code are the output filename and the number of workers and tasks. In the lsf batch script, we request 128 cores.[15] Using the timing waypoints, the times in seconds for these different steps are:

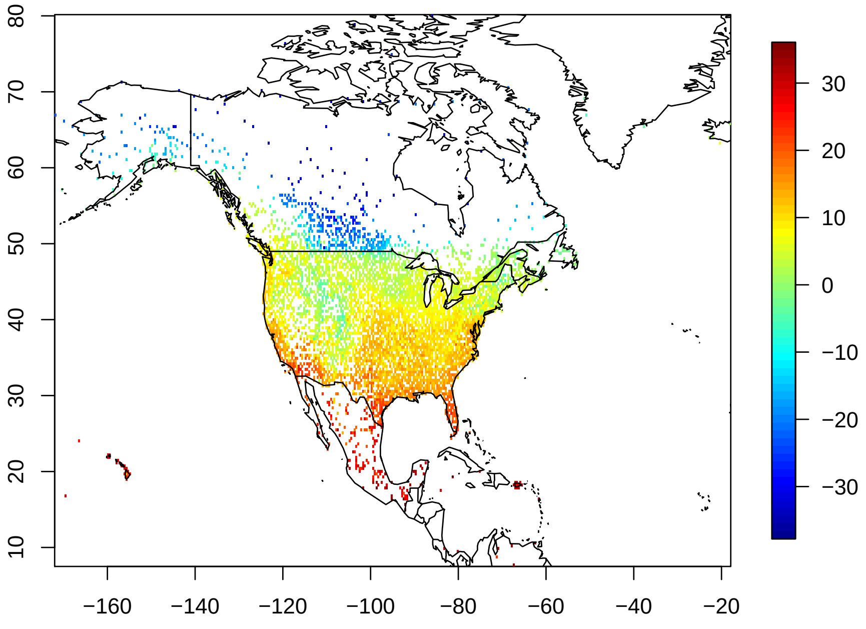
```
Source  Spawn   Broadcast  Apply
0.857   16.385  13.869     684.894
```

[13]See `http://www.ncdc.noaa.gov` → Data Access → Land-based Station → Datasets → GHCN

[14]A quick way to see changes in similar files is to use the unix `diff` utility: `diff supervisorBatch.R supervisorBatch2.R`.

[15]Note that 8 more cores are requested than are used.

Figure 4.1: Observed daily maximum temperatures (C) for February 1, 1970



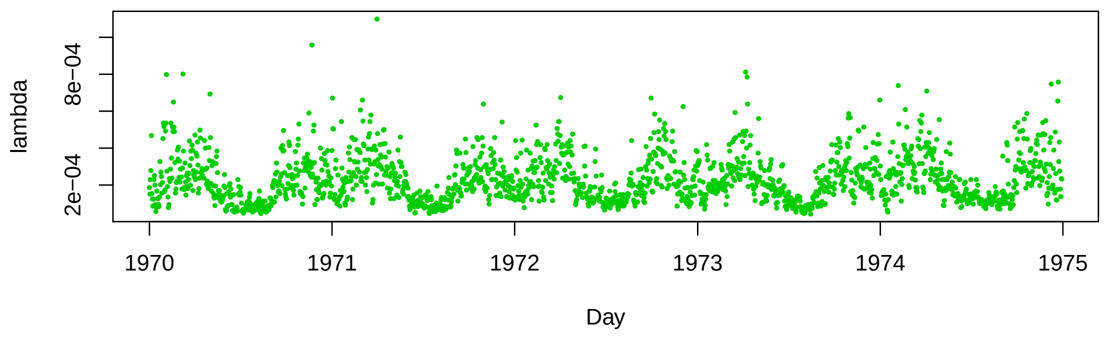
The largest contributor was the MPI apply step, which took about 11 minutes. Note that creating the 120 `R` workers only took about 16 seconds. Based on the `.out` file information, the total "wall clock" time for this job, including waiting in the queue to run, was roughly one half-hour.

The accumulated results of the 1825 tasks form a list where each component is a summary of the `Tps` fit for that day. Of interest are the smoothing parameter (`lambda`) estimates for these five years of daily data. A time series suggesting seasonal dependence of lambda is shown in Figure 4.2. Here we see a clear pattern of seasonality, with smaller `lambda` values indicating greater spatial variability in the summer months. The `R` code used to create Figure 4.2 is given in Table A.2.

5 Workflow for creating a climate data product.

Part of the interest in `Rmpi` arose from the goal of creating climate analyses that could be tailored to specific projects, computed efficiently, and reproduced by other groups. This effort within IMAGe at NCAR is referred to as the Multi-scale Integrated Climate Analysis (MICA) to indicate the flexibility in producing analyses at different spatial scales. In this section a more complex example of `Rmpi` is presented that illustrates the steps of deriving regular gridded fields of daily temperature from the GHCN station data. The subdirectory `RmpiMICA` has the source code and batch files for this example. A difference between this workflow and the previous example is that the `R` code and the use of `Rmpi` are more closely intertwined and specific to this project.

Figure 4.2: Time series plot of estimated lambda parameter using restricted maximum likelihood.



5.1 Spatial prediction for large data sets

The primary spatial method in this example is based on a spatial model that uses a compactly supported covariance function.[16] Such methods are designed to handle larger spatial analysis problems such as all of North America (as opposed to the spatial subsets taken in analysis in Sections 2 and 4). Typically, when the number of spatial observations exceeds several thousand locations, standard statistical models become difficult to compute due to linear algebra on large and dense matrices.[17] The compact correlation used here (the Wendland function) has a limited range of correlations and induces sparse matrices into the statistical computations. The form of this covariance is similar at short distances to the thin plate spline model and so in some sense it represents an approximation to fitting a thin plate spline for larger data sets. With this similarity, a key statistical parameter is again `lambda`, which is interpreted as controlling the smoothness of the fitted surface. This method is implemented as the function `fastTps()` in the fields package.

Although `fastTps()` can handle large spatial datasets on the order of tens of thousands of locations, it does have a limitation for situations where the location density varies greatly. As part of this method one needs to specify the range of the compact covariance, the `theta` argument of the `R` function. Spatial locations separated by more than this distance are assumed uncorrelated because the covariance will be zero beyond this distance. Of course this assumption would rarely hold in practice and typically one would want to choose `theta` as large as possible while keeping the computation feasible. A rough rule of thumb is to increase `theta` until every spatial location has about 50 neighboring locations within a distance of `theta`. Unfortunately, as was shown in Figure 4.1, the station density in North America varies greatly from sparse coverage in Northern Canada to the very dense network in the Northeastern US. Thus it is difficult to choose a range that fits this criterion without greatly decreasing the sparsity of the covariance matrices in areas of dense station locations. As a compromise, for this example we keep the range parameter fairly large but also determine the number of neighboring locations for each grid point in the predicted surface. The idea is to only report spatial predictions where the station density is adequate. There are other methods that can handle larger spatial data sets without this limitation and the overall workflow

[16]That is, a covariance function that drops to zero for locations separated by a distance greater than some threshold

[17]More specifically, the computation time will tend to be dominated by the cube of the number of spatial locations. Thus, where this relation holds, increasing the spatial locations by a factor of 10 will increase the computational time by a factor of 1000.

presented here can be modified to suit these methods as well. More about these alternatives is discussed at the end of this report.

5.2 The MICA setup

As with the instructions for the first example, we assume that you have copied the `HPC4Stats` directory. The source code and lsf script are in the subdirectory `RmpiMICA`, which contains the following files:

startMICA.lsf The master lsf script for submission to the batch queue. Be sure to change the project code to your project code. Note that the bsub arguments are set to request 32 cores (`-n 32`) and a wall clock time of 1 hour (`-W 1:00`). However, typically this job runs in under 5 minutes.

supervisorBatch.R The supervisor `R` script that will create the workers and assign the tasks. This script contains user-dependent directories (the code directory `ddir` and the output directory `outdir`) that are automatically set as long as `HPC4Stats` is in your home directory. Note that this example uses 30 workers and will execute 90 tasks (90 days).

lambdaKrig.R The main function, `lambdaKrig`, which estimates the smoothing parameter, fits the surface to the data, and generates conditional simulations of the surface on the elevation grid used to approximate standard errors. This is analogous to the `doTask` function from the first example.

gridInfo.R The `gridInfo` function reads in the elevation data set and determines the spatial domain for the predicted field.

readData.R The `readData` function reads in the data for the day. This function differs from the first example in that it requires the full path and file name for the data file and also the table of station information. This information is created from the task ID in lambdaKrig.

summarizeFit.R The `summarizeFit` function extracts a short summary from the fastTps fit object.

fromScratch A partial copy of the output files that were originally created in `/glade/scratch/nychka`, included for reference.

To run this example, change directories to RmpiMICA:

```
cd ~/HPC4Stats/RmpiExample
```

substitute your project code in the `startMICA.lsf` script then run:

```
bsub < startMICA.lsf
```

5.3 Details of the `R` code

An amended version of the `R` supervisor script is included as Table A.4, and in this section we highlight some differences with the previous example. The first part of the script defines the input/output file names and directories. Here, 30 workers are created with 90 tasks (days). The `bcast` functions are used to broadcast the libraries and functions to the workers, but additional information is also passed directly through the `mpi.iapplyLB` function to `lambdaKrig`. This has the advantage of making it clear which objects are being used to call this function. The final part of this script writes the summary output and closes the supervisor session.

The main function for this example is `lambdaKrig`, which computes several steps of a spatial analysis. This example has a different style of coding in that information is passed to the main function directly in the `mpi.apply` call. Tables A.5 and A.6 are shortened versions of the `lambdaKrig` function that illustrate this difference. Unlike the `doTask` function from the first example, `lambdaKrig` takes several more arguments and so this information is passed directly rather than being broadcast to each worker.

Another important difference is that the only information passed directly back to the supervisor is a summary of the spatial fit; the main results of the spatial analyses are written directly as `R` binary files in the scratch directory. Saving output within worker processes is critical in large problems for two reasons, both related to the need to avoid passing the entire output dataset back to the supervisor. First and foremost, working with many years of daily data can quickly exceed the memory capabilities of even the largest computers. Dealing with the size of the data is why we have it broken up in the first place. Additionally, communication time between nodes is extremely long and variable. Even if our memory could handle the entire output, we would be including an additional step that would slow down our algorithm and potentially strain the supercomputer.

5.4 Spatial analysis results

The output files in the `RmpiMICA` directory result from executing this script on YS in the `regular` batch queue. The wall clock time is under 10 minutes and each of the 90 `R` binary files created is on the order of 2Mb in size. The summary data is in the form of a list, which is most easily parsed using a for-loop in `R`. As an example, we include code to find the statistics for the summary values across the days. Note that in the code below we load the files from the `fromScratch` directory, which is a copy of some of the output files run for this report. The output for a new run will actually be written to the user's `scratch` directory and the user should load the files from this directory to look at new results.

```r
library(fields)
load("fromScratch/mpiTestDetails.rda")
N <- length( result)
tab <- NULL
for(k in 1:N){
  temp <- result[[k]]
  tab <- rbind(tab, temp$summary$summary)
}
statsTab<- t(stats(tab))
ic <- c(1:4, 8)
ir <- c(1, 5, 7:9)
print( signif((statsTab)[ir,ic]) )
```

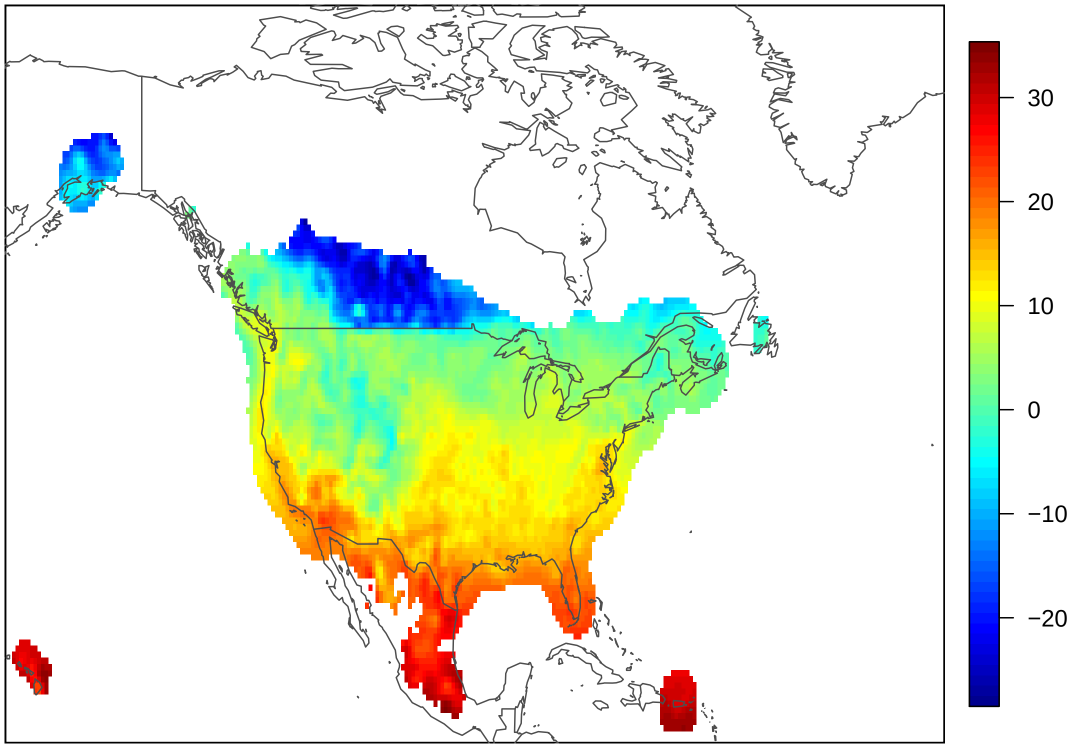
The object `statsTab` is listed below

The estimated fields are saved in separate binary files. Figure 5.1 shows the estimated surface for February 1, 1970 for grid locations with at least 25 stations in range. The `R` code to create this figure is given in Table A.3.

Table 5.1: Statistics from the `result` list of the MICA example

	N	mean	Std.Dev.	min	max
Number of Observations	90	7687.480	43.302	7567	7746
Eff. degrees of freedom	90	1841.070	166.876	1474.000	2225.000
Smoothing parameter	90	0.299	0.095	0.1396	0.596
MLE sigma	90	2.302	0.279	1.8720	3.325
MLE rho	90	19.181	5.931	10.2600	39.670

Figure 5.1: Estimated maximum daily temperature (C) for February 1, 1970



6 Discussion and Future Work

These two illustrations of `Rmpi` are designed to provide an introduction to using `R` on YS with a minimum of programming outside of `R`. The first example attempts to isolate the `Rmpi` step and provides a script that handles parallelizing tasks and passing information to workers in a generic way that does not require many changes for other types of analyses. The second example is more tailored to the task at hand and passes information to the workers directly as part of the `Rmpi` apply function. An important feature of the second example is that each worker creates an output file from the spatial analysis that it was assigned. This mechanism avoids the need to communicate a large amount of information back to the supervisor. In our experience, managing many input and output data files in `R` is one of the main challenges in completing a large spatial analysis across many independent time points. The strategies taken here with file names and station tables represent one way to handle this issue.

Although these examples have been kept to a small number of cores, in principle it is possible to use this method on YS with a much larger number. Keep in mind, however, that as the number of cores increases, the possibility of losing communication with workers and other reliability issues become more likely. More production oriented uses of `Rmpi` would need to consider methods for tracking whether workers have responded within a reasonable time and reassigning tasks or otherwise adjusting when workers fail.

The spatial analysis methods used in this report are simple relative to other Gaussian process estimates, are robust computationally, and are only dependent on a limited number of statistical parameters. A more general model that is closer to standard Kriging is the `spatialProcess` function in `fields` . This supports the Matérn covariance function where for a fixed smoothness parameter the remaining parameters (sill, range, and nugget) are found by maximum likelihood. Unfortunately the Matérn covariance can not be applied to large spatial datasets and so approximations are needed. Typically, on a single core with shared memory, Matérn spatial models are limited to several thousand observations. For larger datasets one possibility is to use the `LatticeKrig` package to estimate the spatial model and generate predicted fields. The `LatticeKrig` model takes advantage of sparse matrix linear algebra to fit the computations onto a single core. Although the numerical method differs, the work flow would follow the same steps as the examples in this report.

In summary, we hope this short tutorial encourages data scientists using `R` to take advantage of high performance computing on YS to accelerate their analyses in `R`. We feel that there are many opportunities for embarrassingly parallel analysis of geophysical data and this report may spur other applications in addition to spatial analysis.

7 Acknowledgements

This work was partially completed while Nathan was employed as an Associate Scientist at NCAR. This material is based upon work supported by the National Science Foundation (NSF) Graduate Research Fellowship under Grant No. (DGE 1144155). We would like to thank the Consulting Services Group at NCAR, especially Siddhartha Ghosh who developed the `.lsf` scripts and solved numerous MPI/R compatibility issues on Yellowstone. Also we are grateful for the careful reading of this report and testing of the R code by Sophia Chen. This work was also supported by the National Center for Atmospheric Research (NCAR). NCAR is managed by the University Corporation for Atmospheric Research under the sponsorship of NSF.

A Supplemental Code

Table A.1: Source code for plotting observations for a given day

```r
library(fields)
dataDir  <- "/glade/p/work/mcginnis/mica/data/daily/narccap/tmax"
infoFile <-
"/glade/p/work/mcginnis/mica/data/daily/narccap/index.tmax.narccap"
sourceDir<- "/glade/u/home/nychka/HPC4Stats/RmpiExample"

# source R function for reading data
source(paste0(sourceDir,"/","setupSupervisor.R"))

# redefine spatial domain to be all of North America
domain<- list(lat = c(7.6, 82.6), lon = c(-172, -17.4))
obj  <- getData(32)

# can check that the date is right using dir(dataDir)[32]
quilt.plot(obj$x, obj$y, nx = 400, ny = 200)
world(add = TRUE)
```

Table A.2: Source code for time series plot of estimated lambda parameter

```r
N <- length(result2)
lambdaRMLE <- rep( NA, N)

# extract the data from the result list
for(k in 1:N){
  summaryFit <- (result2[[k]])
  lambda.est <- summaryFit$lambda.est
  lambdaRMLE[k] <- lambda.est[ 6, 1]
}

# a method to handle the date conversions
day <-substring(dataFileNames, 53, 61)
day <- day[1:N]
dayDate <- as.Date(day, format = "%Y%m%d")

# use new date object for final plot
plot(dayDate, lambdaRMLE, xlab = "Day", ylab = "lambda",
          pch = 16, cex = .5, col = "green3")
```

Table A.3: Source code for plotting estimated daily maximum temperature

```
library(fields)
load("fromScratch/19700201.output.rda")

# mask out grid points with less than 25 neighboring observations.
mask<- outList$pgridInfo$z < 25
tmaxField <- outList$meanField
tmaxField$z[mask] <- NA

# plot using par to set margins for the X window
par( mar = c( 1,1,1,1))
image.plot( tmaxField, axes = FALSE, xlab = "", ylab = "")
box( lwd = 2)
map("world", add = TRUE, col = "grey30")
```

Table A.4: Supervisor script (`supervisorBatch.R`) to setup computations and control workers.

```
# Load packages
  library(fields)
  library(Rmpi)
  library(ncdf4)
  ddir <- "/glade/u/home/nychka/HPC4Stats/RmpiMICA"
# Read in R functions
  source(paste0(ddir,"/lambdaKrig.R"))
  source(paste0(ddir,"/gridInfo.R"))
  source(paste0(ddir,"/readData.R"))
  source(paste0(ddir,"/summarizeFit.R"))
# Constants to control run sizes
  N     <- 90
  procs <- 30
  dataDir  <- "/glade/p/work/mcginnis/mica/data/daily/narccap/tmax"
  orogfile <- "/glade/p/work/mcginnis/mica/orog.grid.nc"
# Set directory to save output
  outdir   <- "/glade/scratch/nychka"
# elevation file
  orogfile <- "/glade/p/work/mcginnis/mica/orog.grid.nc"
#Build data frame with station information
  infoFile <-
  "/glade/p/work/mcginnis/mica/data/daily/narccap/index.tmax.narccap"
  stationInfo<- read.table(infoFile)
  names(stationInfo)<- c("ID", "lat", "lon", "elev", "start", "end")
  stationInfo$lon<- stationInfo$lon - 360
  fileList<- system( paste("ls ", dataDir), intern = TRUE)
# Create the output grid
# and read in elevations at these grid points
  orthoList <- gridInfo(orogfile)
# Spawn the workers
  mpi.spawn.Rslaves(nslaves = procs)
# Load packages on the workers
  mpi.bcast.cmd(library(fields))
  mpi.bcast.cmd(library(ncdf4))
# Broadcast the main function and the supporting functions to
# the workers
  mpi.bcast.Robj2slave(lambdaKrig)
  mpi.bcast.Robj2slave(readData)
  mpi.bcast.Robj2slave(gridInfo)
  mpi.bcast.Robj2slave(summarizeFit)
# Apply lambdaKrig to the IDs 1:N
  result <- mpi.iapplyLB(1:N, lambdaKrig, orthoList = orthoList,
            outdir = outdir, dataDir = dataDir,
            stationInfo = stationInfo, fileList = fileList)
  save(result, file = paste0(outdir,"/mpiTestDetails.rda"))
# Close workers and quit cleanly out of R/Rmpi
  mpi.close.Rslaves()
  mpi.quit()
```

Table A.5: First part of the `lambdaKrig` function called from the supervisor script and executed by each worker.

```r
lambdaKrig <- function(index, orthoList, outdir,
                              dataDir, stationInfo, fileList){
require(fields)
dataFile  <- fileList[index]
#output file includes name of data file
outFile <-paste0(dataFile,".output.rda")
# see Seth's notes for details on this choice
# this is in units of degrees
thetaRadius <- 3.0
# Read in synoptic daily summary data
FN<- paste0(dataDir,"/", dataFile)
dataIn<- readData(FN, stationInfo)

####### Trim station list to output region ########
glon<- orthoList$glon
glat<- orthoList$glat
ind1<- (dataIn$x[,1] >= min(glon)) &
       (dataIn$x[,1] <= max(glon)) &
       (dataIn$x[,2] >= min(glat)) &
       (dataIn$x[,2] <= max(glat))
# omit NAs
ind2<- !is.na(dataIn$y)
# omit duplicated locations (this may not be right)
ind3<- !duplicated(dataIn$x)
xin <- dataIn$x[ind1&ind2&ind3,]
yin <- dataIn$y[ind1&ind2&ind3]
# Zin are the elevation at these locations
# Zin <-dataIn$Z[ind1&ind2]

####### Estimate lambda using MLE ########
lobj <- fastTps.MLE(xin, yin, theta = thetaRadius, relative.tolerance = 1e
    -08)
lambdaMLE <- lobj$lambda.best

####### Fit thin-plate spline surface to data ########
fitObject <- fastTps(xin, yin, lambda = lambdaMLE, theta = thetaRadius)
fitSummary<- summarizeFit( fitObject )
pgrid <- list(x = orthoList$glon, y = orthoList$glat)

# next part is finding out how many neighbors each grid point
# has that is within thetaRadius. This works because of the
# convenient way spam stores sparse matrices.
sparseDist<- nearest.dist( make.surface.grid(pgrid), xin,
                                    delta = thetaRadius)
pgridInfo <- as.surface( pgrid, diff( sparseDist@rowpointers))
meanField <- predictSurface( fitObject, grid.list = pgrid)
```

Table A.6: `lambdaKrig` function continued (Source code continued from Table A.5)

```
####### Generate ensemble of  M realizations of surface
####### on the  pgrid of lon/lats
M <- 100

# time conditional simulation
set.seed(444) # for ensemble generation
tic <- Sys.time()
  ensemble <- sim.fastTps.approx(fitObject, pgrid, M = M)
toc <- Sys.time()
cputime<- toc-tic

####### Generate output list ########
# calculate predict standard error for mean field
stderr = apply(ensemble$Ensemble,1,sd)
# keep ensemble output list format but only save first 10 members
ensemble$Ensemble <- ensemble$Ensemble[,1:10]
# the big output list with everything the needs to be saved
outList <- list(ensemble = ensemble,
                   pgrid = pgrid,
               pgridInfo = pgridInfo,
               meanField = meanField,
                  stderr = stderr,
              fitSummary = fitSummary,
                     day = dataFile,
                    type = "tmax")

# Save this within the MPI loop to avoid passing back
# large amounts of data to the supervisor process
save(outList, file = paste0(outdir, "/", outFile))

# smaller list to return to supervisor
return(list(time = cputime, summary = fitSummary, day = dataFile) )
# all done!
}
```