Parallel Implementation and Computational Analysis of the Multi-resolution Approximation

Lewis R. Blake
Peter Simonson
Dorit Hammerling

NCAR Technical Notes
NCAR/TN-551+STR

National Center for
Atmospheric Research
P. O. Box 3000
Boulder, Colorado
80307-3000
www.ucar.edu

NCAR/TN-551+STR
NCAR Technical Note

2018 - 10

Parallel Implementation and Computational Analysis of the Multi-resolution Approximation

Lewis R. Blake
Department of Applied Mathematics and Statistics
Colorado School of Mines, Golden, CO
Peter Simonson
Department of Applied Mathematics and Statistics
Colorado School of Mines, Golden, CO
Dorit Hammerling
Analytics and Integrative Machine Learning
Computational and Information Systems Laboratory
National Center for Atmospheric Research, Boulder, CO

NCAR Laboratory
NCAR Division

NATIONAL CENTER FOR ATMOSPHERIC RESEARCH
P. O. Box 3000
BOULDER, COLORADO 80307-3000
ISSN Print Edition 2153-2397
ISSN Electronic Edition 2153-2400

Parallel Implementation and Computational Analysis of the Multi-resolution Approximation

Lewis R. Blake [*1], Peter Simonson[†] [‡2], and Dorit Hammerling [§3]

[1]Colorado School of Mines
[2]Colorado School of Mines
[3]National Center for Atmospheric Research

October 23, 2018

Abstract

We design and implement a parallel version of the multi-resolution approximation (MRA) and analyze its computational performance for large geospatial datasets. Our parallelization extends to calculating the prior distribution, posterior distribution, and spatial prediction. We provide an overview of the MRA model and our implementation. A detailed description of our codebase and schematics are included for ease of reference. In an effort to identify efficient model configurations for datasets of various sizes, we perform timing and memory profiling studies including benchmarking against our serial implementation. To test the scalability of our model, remote sensing data of Sea Surface Temperature (SST) recorded by NASA's AMSR-2 and MODIS satellite instruments on the order of 2.4 million and 2.7 million observations, respectively, are used. We show that there is a trade-off between memory consumption and runtime for both smaller and larger datasets. Often utilizing more workers will reduce runtimes in exchange for larger memory requirements associated with parallel computations. These findings highlight the necessity for carefully selecting the model and parallel set-up configurations as a function of the data size and the computational infrastructure.

Keywords: Gaussian Process, Multi-resolution approximation, MATLAB®, geospatial data, parallel computing, Cheyenne supercomputer

[*]lblake@mines.edu
[†]petersimonson@mines.edu
[‡]This author was supported by the National Science Foundation under award DMS-1551229.
[§]dorith@ucar.edu

Contents

1 Introduction

Gaussian Processes (GPs) are ubiquitous in spatial statistics for analyzing spatial data, estimating parameters, and making spatial predictions at unobserved locations. GPs are defined by the property that any finite combination of observations have a multivariate normal (MVN) distribution. The MVN distribution has many convenient features such as being completely determined by its mean and covariance, remaining MVN under operations such as conditioning, and being a good approximation for other distributions.

Directly evaluating the probability density function of a MVN random vector incurs $\mathcal{O}(n^2)$ memory and $\mathcal{O}(n^3)$ floating-point operations (flops), and thus becomes too computationally burdensome when the number of observations, n, is on the order of 10^4 or more. Automation, improvements in sensors, and other factors have significantly increased the amount of collected data. As such, implementation of GPs becomes a substantial computational bottleneck when dealing with big data. Consequently, simplifying assumptions and/or approximations are necessary to model such data. Numerous examples of approaches to approximate GPs in computationally efficient manners are found throughout the literature (see, for example Heaton et al., 2018 [1]). In this technical report we evaluate one such strategy: the Multi-resolution approximation (MRA).

Originally presented in Katzfuss, 2016 [2], we implement the MRA with MATLAB® and test it to scale to large geospatial datasets. Of particular interest is the implementation of the MRA in parallel on the NCAR supercomputer Cheyenne. We test the MRA on both a relatively small dataset on the order of $n = 150,000$ and larger datasets on the order of $n = 2.4$ million and $n = 2.7$ million. We seek to find choices for implementing the MRA for variously sized data across computational environments.

In Section 2 we provide a brief overview of the MRA described in Katzfuss, 2016 [2]. In Section 3 we describe our MRA implementation, its fundamental modeling assumptions, and some imposed restrictions. Section 4 describes the datasets used in model testing. Section 5 is a timing study of our implementation to assess variability in run-times and memory burdens. Section 6 finalizes with a discussion.

2 MRA Model

The MRA presented in Katzfuss, 2016 [2] is the theoretical foundation for the associated codebase of this paper. The study performed in Heaton et al., 2018 [1] showed the MRA to be both computationally efficient and accurate. Following is a brief overview of the MRA (see [2] for a comprehensive description of the model).

The spatial field of interest is modeled via basis function representation of a GP. The true spatial field is denoted $\{y_0(\mathbf{s}) : \mathbf{s} \in \mathcal{D}\}$, or $y_0(\cdot)$, on a continuous domain $\mathcal{D} \subset \mathbb{R}^d$, $d \in \mathbb{N}^+$. Assumed is that $y_0(\cdot) \sim GP(0, C_0)$ is a zero-mean Gaussian process with covariance function C_0, that is known up to a vector of parameters $\boldsymbol{\theta}$.

Once data has been observed at n spatial locations, by the GP assumption, the data follows a n-dimensional MVN distribution. Directly evaluating this distribution results in the computationally burdensome task of inverting and calculating the determinant of an $n \times n$ matrix. To facilitate computationally feasible approximations of the MVN distribution, the following simplifying assumptions are introduced to define the MRA.

To begin with, the MRA defines a recursive partitioning of the spatial domain $\mathcal{D}$, in which each of the J regions is further divided into J smaller regions, and this process is repeated M times. Let $\mathcal{D}_{j_1,\ldots,j_{m-1}}$ be a region at level $m-1$. We can write $\mathcal{D}_{j_1,\ldots,j_{m-1}}$ as a union of regions at one resolution finer (level m) as follows:

$$\mathcal{D}_{j_1,\ldots,j_{m-1}} = \bigcup_{j_m=1,\ldots,J} \mathcal{D}_{j_1,\ldots,j_m}, \quad j_1,\ldots,j_m = 1,\ldots,J; \quad m = 1,\ldots,M.$$

In the MRA, for the GP $y_0(\cdot)$, $[y_0(\cdot)]_{[m]}$ is defined as a "block-independent" version of $y_0(\cdot)$ between regions at resolution m. That is, $[C_0]_{[m]}(\mathbf{s}_1,\mathbf{s}_2) = C_0(\mathbf{s}_1,\mathbf{s}_2)$ if $\mathbf{s}_1,\mathbf{s}_2$ are in the same region $\mathcal{D}_{j_1,\ldots,j_m}$ and $[C_0]_{[m]}(\mathbf{s}_1,\mathbf{s}_2) = 0$ otherwise, where $C_0(\cdot,\cdot)$ is the covariance function. The recursive partitioning of the domain along with the block-independence assumption leads to a natural interpretation of domain partitioning inheriting a parent-child hierarchical structure. In explicit, at each level $m = 1,\ldots,M-1$, we can view each region $\mathcal{D}_{j_1,\ldots,j_m}$ as the parent of the J subregions $\mathcal{D}_{j_1,\ldots,j_{m+1}}$ at level $m+1$ contained within $\mathcal{D}_{j_1,\ldots,j_m}$. Further, by block-independence, at each level the GP within each region $\mathcal{D}_{j_1,\ldots,j_m}$ is only assumed statistically dependent on its parental hierarchy (i.e., all regions at coarser resolutions containing $\mathcal{D}_{j_1,\ldots,j_m}$).

Also defined are a set of r knots (with $r << n$) at each resolution that all lie within a particular subregion $\mathcal{D}_{j_1,\ldots,j_m}$. The knots are the locations at which the basis function attain their maximum. At the finest resolution, M, we define the knots to be the observations within that region. By placing knots within each subregion, instead of working with $n \times n$ matrices, we can reasonably well approximate $y_0(\cdot)$ by working mostly with $r \times r$ matrices in a computationally feasible manner. To model the spatial field as a GP, the MRA iteratively approximates $y_0(\cdot)$ and covariance function $C_0(\cdot,\cdot)$ at resolutions $m = 1,\ldots,M$ dependent on the knots and partitions. At coarser resolutions, the MRA captures large-distance spatial trends in $y_0(\cdot)$. By increasing number of levels used in the approximation, the MRA captures shorter range variability as well.

3 Model Implementation

Our implementation of the MRA was programmed in MATLAB® to take advantage of the Parallel-Computing Toolbox and fast matrix manipulations. The required toolboxes are the Statistical Computing Toolbox and the Parallel-Computing Toolbox. The codebase was adopted from the MRA model implemented by Dorit Hammerling in Heaton, et al. 2018 [1], and optimized to run in parallel. The original MRA codebase executed the algorithm effectively, however we sought to apply the MRA to larger datasets and improve computational efficiency. Since the MRA at each level assumes block-independence between partitions, this results in computational processes at each *level* that are embarrassingly parallel in nature. The MRA algorithm as a whole, however, does not constitute an embarrassingly parallel process.

Throughout, notation from Katzfuss, 2016 [2] will be used interchangeably with their equivalent variables names in the codebase. However, an import difference between our notation and the notation used in Katzfuss, 2016 [2] is that in our implementation we denote the levels as $m = 1, ..., M$ where as Katzfuss denotes the levels going from $m = 0, ..., M$. That is, what is originally presented as M in [2] corresponds to $M + 1$ in our model. This is as a consequence of programming the model in MATLAB® where array indices start at 1, which is reflected in the codebase. Table 1 contains some commonly used notation in Katzfuss, 2016 [2] and their associated codebase variable names.

Table 1: Frequently referenced MRA variables, their associated codebase variable names, and a short description.

Literature Notation	Codebase Notation	Description
M	`NUM_LEVELS_M`	Maximum number of levels to partition $\mathcal{D}$.
J	`NUM_PARTITIONS_J`	Number of partitions for each region.
r	`NUM_KNOTS_r`	Number of knots in each region for levels $m = 1, ..., M - 1$.
$\boldsymbol{\theta}$	`theta`	Vector of parameters for covariance function C.

Restrictions are imposed in our MRA implementation for computational efficiency, mathematical benefit, and desired applicability that are not necessary in the formal derivation of the model. To begin with, we assume that the domain of interest is two-dimensional (i.e., $\mathcal{D} \subset \mathbb{R}^2$). For simplicity's sake, we make the restriction that the number of sub-partitions, `NUM_PARTITIONS_J`, be either 2 or 4. By doing so, we preserve spatial dependence between them at as many levels as possible. By construction of the MRA, for $\mathbf{s}_1, \mathbf{s}_2 \in \mathcal{D}$, the quality of the covariance approximation depends on the smallest partition that contains the two observations, beyond which they are assumed to be statistically independent. If J were quite large at levels $m = 2, ..., M$, $\mathbf{s}_1$ and $\mathbf{s}_2$ could lie in different sub-partitions at a much coarser resolution, and by the block-independence assumption, too much spatial dependence between locations would be lost. That is, locations would share dependence at a fewer number of levels than is desirable resulting in any dependence between them being poorly approximated. See Katzfuss, 2016 [2] Section 2.4 for further details.

3.1 Preliminaries

A stand-alone function, `find_num_levels_suggested`, is included within the codebase. This function takes as inputs the number of data points for a given dataset (n), the number of knots desired at each level (r), and the number of partitions desired within each sub-partitioning of the domain (J), and calculates the number of levels recommended (M). The idea behind this function is to work backwards from the finest resolution level, making the average number of observations per region similar to the number of knots. For example, for a dataset with $n = 1,000,000$ observations, $r = 64$ knots, and $J = 2$ partitions at each level, `find_num_levels_suggested` outputs that $M = 14$ levels are recommended. This function is a guideline for the number of levels to test. For a given dataset, number of knots, and number of partitions, experimenting with the level will be needed in order to assess best practices. We will study this behavior in Section 5.

Another important function to note is `evaluate_covariance`. This is a generic covariance function. For testing purposes, we employ an exponential model. If a different covariance function is preferable for an application, editing this function will be required.

3.2 Implementation Overview

3.2.1 Computational Modes

Our MRA implementation consists of three computational modes: `likelihood`, `optimize`, and `prediction`. The `likelihood` routine calculates the log-likelihood for a given set of parameters. The `optimize` routine uses the built-in `fmincon` function to optimize over the parameter space defined by upper and lower bounds given for $\boldsymbol{\theta}$. The `prediction` routine conducts prediction over a given prediction grid for a given set of parameters. The `optimize` mode can be thought of as a repeated implementation of the `likelihood` mode over the parameter space and then, once a local minimum has been found, executing the `prediction` mode. The `likelihood` and `prediction` modes are two different operations.

3.2.2 Building the Structure

In order to meet the MRA working assumption of a zero-mean GP, prior to conducting any of these modes, a spatial linear model is fit to the data. Having centered the data at zero, the model then utilizes `build_structure` to build the hierarchical structure of partitioning and sub-partitioning the domain as well as placing the knots within each region. At each level, `build_structure` partitions the domain along the longest side of each subregion. In our implementation, we build the hierarchical structure storing all relevant information for a particular subregion in cell arrays which are indexed by a continuous index for each subregion. As a result, these cell arrays are of size $J^M - 1$. Storing these arrays presents a computational limitation. Once the structured domain has been produced, we can move forward to the MRA analysis.

3.2.3 Creating the Prior Distribution and Posterior Inference

The MRA algorithm itself can be thought of as consisting of two major components: creating the prior distribution and computing the posterior inference. In creating the prior distribution, the algorithm works from coarsest to finest resolution (starting at $m = 1$, and going until $m = M$) calculating the prior quantities given by Equation 5 and Equation 6 in Katzfuss, 2016 [2] (implemented within `MRA.m` by the `create_prior` function). Of particular interest is computing the covariance of the knots within a subregion and the knots of the parental hierarchy (that is, all subregions at coarser resolutions containing that subregion). Recall that the MRA assumes block independence across partitions at a given level. So in creating the prior distribution, we are only concerned with partitions at coarser levels that contain a given partition at a finer level, and then computing the covariance between those knots at different resolutions. In our implementation, this procedure is parallelized. See Katzfuss, 2016 [2] Section 3.1 for more details.

Once the MRA algorithm has reached the finest resolution of the prior distribution (level M), the observations enter the algorithm; specifically, the knots within each subregion at the finest level are set to be equal to the observation locations within that subregion. With the knots at the finest resolution, M, set to the data, the MRA algorithm now works from second-finest resolution to coarsest resolution (that is, from $m = M-1$ to $m = 1$), weighting the structure (i.e., calculating the conditional posterior distributions of the weight vectors for the basis functions) and covariance matrices by means of the `posterior_inference` function. This is computed by means of Equations 7, 8, 9, 10, and 11 in Section 3.2 of Katzfuss, 2016 [2]. After the MRA algorithm has worked from coarsest resolution to finest resolution to create the priors, and then from finest to coarsest resolution to compute the posteriors, the approximation of the spatial field $y_0(\cdot)$ is fully determined and the analysis is complete.

3.2.4 Outputs

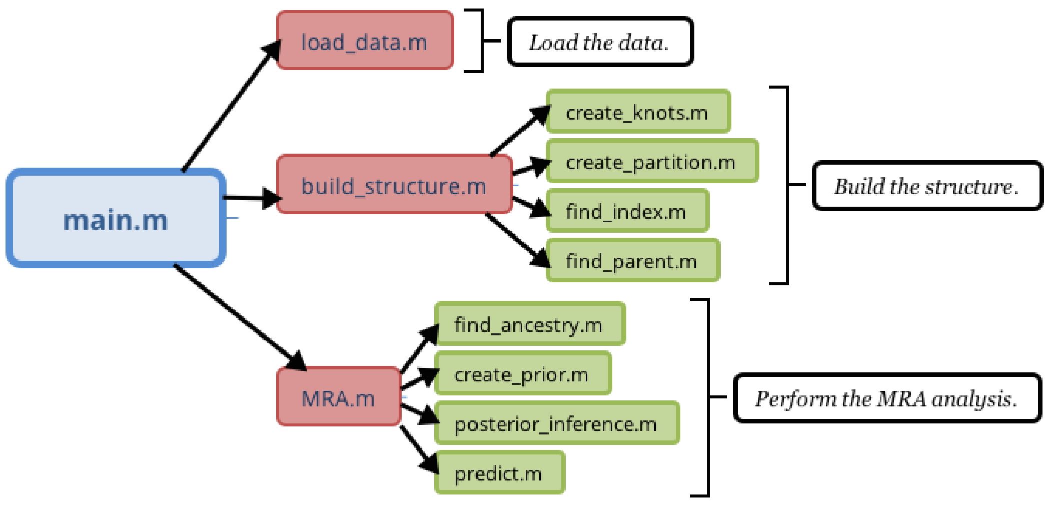
Different structures and plots are returned depending on the computational mode. For the `prediction` and `optimize` modes, the output is a `.mat` file with the MRA spatial prediction results. Moreover, if the boolean variable `plotting` is set to 1, three plots are also produced corresponding to the ground truth, predicted values, and the prediction variance. If computing on a remote server without a GUI, saving these images produced in `main.m` will be needed. Saving the plots produced can be accomplished by uncommenting the `saveas()` command underneath each figure within `main.m`. If saving large plots using this method still fails, slicing into the desired vectors and plotting only a subset of the data may be required. When computing the `likelihood` mode, the log-likelihood for a given set of parameters is calculated and printed in the command window.

A significant benefit of the MRA is that it lends itself to execution in a distributed computing environment. For this reason, the creation of the prior, posterior inference, and spatial prediction in this codebase were designed to run in parallel. A serial implementation of the MRA was also developed and used as a comparison throughout.

3.3 Codebase Structure

Figure 1 is a visual representation of the codebase and how functions therein relate to each other. The software implementation has three modules. The `main.m` script runs these modules in the appropriate order. Within `main.m`, the three modules have responsibilities of loading the data from the data source, building the recursive partitioning structure of the model, and performing the MRA analysis. All three computational modes (i.e., `likelihood`, `optimize`, and `prediction`) utilize the data loading and structure building modules. Then, depending on the computational mode being executed, a `switch` clause within `main.m` determines which mode to execute. Within each of these modes variable inputs are given to the `MRA` function specific to that mode. For instance the `prediction` mode requires the prediction locations in order to execute whereas the `likelihood` routine does not. The `likelihood` and `prediction` modes further differ in the MRA module in that the `likelihood` mode does not execute the `predict` function. See Appendix A for a description of MRA functions and Appendix B for a description of the codebase objects.

Figure 1: MRA codebase flowchart. From left-to-right are function layers, indicated by various colors. Functions are called by their parent in order from top to bottom. Heuristic descriptions are given in brackets on the right.



3.4 User Input

3.4.1 Required Inputs

Most of the user input can be specified within `main.m`. To begin with, the `dataSource` is set with a string. For the two datasets included with this codebase, the options for `dataSource` are `satellite` and `simulated`. These correspond to cases in the `switch` clause within `load_data.m`. Specification of the `calculationMode` is also done within `main.m`. The three types of computation modes, given by `calculationMode`, that our MRA implementation can execute are `prediction`, `optimize`, and `likelihood`.

Next, within `main.m`, the user must specify a boolean variable, `plotting`, to indicate whether the MRA should produce plots when conducting the `prediction` or `optimize` routines. The user must also specify the total number of levels: `NUM_LEVELS_M`, the number of partitions in a subregion: `NUM_PARTITIONS_J`, number of knots per partition: `NUM_KNOTS_r`, and an edge buffer: `offsetPercentage`.

As an aside, the computational speed and accuracy of the MRA model depends in large part on the parameter `NUM_KNOTS_r`. In fact, most of the inference is done by working with $r \times r$ matrices and therefore adjusting `NUM_KNOTS_r` has a large influence on performance. Knot placement has been well studied in spatial statistics. In this codebase, knots are placed rather simply, as a grid within each subregion. Calculation of the number of knots in both the x and y directions takes place within `build_structure.m` and creation of the knots takes place within `create_knots.m` within a subregion. For example, setting the `NUM_KNOTS_r` to 64, forms a 8×8 grid of knot locations within each subregion. Letting r take values that are not a square number results in a knot grid which is $\lceil \sqrt{r} \rceil \times \lceil \sqrt{r} \rceil$, where $\lceil \cdot \rceil$ is the ceiling function.

3.4.2 The Parallel Pool

The parallel pool (collection of workers) is started at the beginning of the `main.m` script with the `parpool` command. This command will launch a parallel pool based on the default preference settings. A specific number of workers is set by the call `parpool(nWorkers)` where `nWorkers` is a positive integer (e.g., `parpool(8)`). For further reference please see `https://www.mathworks.com/help/distcomp/run-code-on-parallel-pools.html`. In most cases, parallel computation is desirable for large datasets where there is sufficient memory.

3.4.3 Prediction Grid

The `prediction` mode allows users to make predictions of $y_0(\cdot)$ at spatial locations within the domain. The MRA selects only those locations where there are observations for the analysis, and then predicts over the prediction grid. The set of prediction locations, $\mathcal{S}^P$, can be given to the MRA by specifying the number of grid points in the x and y directions using `nXGrid` and `nYGrid` in `main.m`. The `prediction` mode will then make predictions over the spatial grid determined by those values.

Alternatively, the user can supply custom prediction locations to the MRA by declaring the `predictionVector` in `load_data.m` to be a supplied object. The variable `predictionVector` takes two vectors as arguments which correspond to the x locations and y locations for prediction. Each row of `predictionVector` is a tuple consisting of two-dimensional coordinates of the locations at which to predict.

3.4.4 Inputting Data

When applying the MRA to a user supplied data set the user must specify the type of data being used and the file path in `load_data.m`. Relative file paths are used for the two data sets provided with the codebase. User supplied data in other folder locations can be loaded using absolute file paths. In order to use a different data set, a new case within the `switch` clause must be added with the case given as a string, a file path to the data set within the `load` function. Additionally, for a user supplied data set, appropriate values for `theta` and `varEps` (the parameters for the `covariance` function and the measurement error variance) should be declared if they are known. If these values are not known, they can be given lower and upper bounds within the `switch` clause of `main.m` and then estimated using the `optimize` routine. To load a user-supplied data set, the `switch` clause case string must match the string given to `dataSource` in `main.m`. The rest of `load_data.m` assumes that the dataset is in matrix format and has columns corresponding to latitude, longitude, observations. If errors for a particular dataset are not known, they can be dropped and rather a constant variance across observations, given by `varEps`, can be used instead. Data stored in other formats, such as a table, can be handled but must be coerced into the `data` matrix in `load_data.m`.

4 Datasets

4.1 Small Data Tested in Heaton et al.

Two smaller data sets are included with the codebase namely the “Small Satellite Data” and the “Small Simulated Data” which each contain about 150,000 data points of daytime land surface temperatures. Both datasets will be referred to by their aforementioned names throughout. These data sets were originally used for the comparison study in Heaton, et al. 2018 [1] and retained here for timing comparisons on smaller datasets. Both the Small Satellite Data and the Small Simulated Data have latitudes ranging from 34.30 to 37.07 and longitudes ranging from -95.91 to -91.28. Within this domain, the mean observation for the Small Satellite data is 44.54 degrees Celsius with a sample standard deviation of 3.97. The mean observation for the Small Simulated Data is 43.48 degreed Celsius with a standard deviation of 3.30. See Heaton et al., 2018 [1] Section 3 for a more complete description of these datasets. For the purposes of testing the MRA, only the Small Satellite Data was used in the timing study.

4.2 NASA Sea Surface Temperature Data

To test the scalability of the MATLAB® implementation of the MRA to large geospatial datasets, data from NASA's Physical Oceanography Distributed Active Archival Center (PODAAC) is used (`https://podaac.jpl.nasa.gov/`). These datasets contain the retrieval of Sea Surface Temperature (SST) from the AMSR-2 instrument with a 25km resolution and the MODIS instrument with 1km resolution. We considered the data available for one day: October 15, 2014. Since the satellites complete two orbits within this time frame, the files contains both nighttime and daytime SST, however for simplicity's sake, we only considered the data corresponding to daytime observations. The daytime October 14, 2105 AMSR data has $n = 2,441,405$ observations with latitudes ranging from -66.15 to 82.78 and longitudes ranging from -179.00 to 180. The mean SST is 289.50 degrees Kelvin with a sample standard deviation of 9.91 degrees. This data will be referenced as AMSR throughout this paper. The MODIS data has $n = 47,567,759$ observations with latitudes ranging from -67.26 to 83.84 and longitudes ranging from -180 to 180. The mean SST over this domain is 292.96 degrees Kelvin with a sample standard deviation of 9.47 degrees. This data will be referenced as MODIS throughout. Both datasets contain eight columns labeled 'bias', 'error', 'lats', 'lons', 'night', 'quality', 'sst', and 'utc'. Table 2 contains descriptions and notes for both the AMSR and MODIS data variables. Figures 2 and 3 are plots of the AMSR and MODIS observations.

The data are obtained in binary format. A Python script transfers the binary files into Python objects and then exports them as a `.csv` file. Once in `.csv` format, it is converted into a MATLAB® table to be more efficiently stored. The tables are then parsed in MATLAB® to a subtable corresponding to day and night measurements. Finally, redundant data-points (i.e., those with duplicate latitude and longitude values) are removed.

Table 2: AMSR and MODIS data variables and their descriptions.

Dataset Variable	Description	Notes
‘bias’	Amount of error that can be attributed to systematic error.	Additive for each measurement/retrieval. Estimated by comparing data to buoys.
‘error’	Square root of the measurement error associated with each data point	Estimated by comparing data to buoys.
‘lats’	Observation latitudes.	
‘lons’	Observation longitudes.	
‘night’	Flag set to either 0 or 1.	Indicates whether the measurements were taken at nighttime or daytime.
‘quality’	Flag ranging from 1 to 5.	This flag indicates the quality of the measurement. Observations with lower quality flags could be the result of cloud-cover or other factors interfering with measurement.
‘sst’	Column of Sea Surface Temperature (SST) observations.	Measured in degrees Kelvin.
‘utc’	Coordinated Universal Time of the measurement.	UTC for October 15, 2014 ranges from 1413331200 to 1413417599.

Figure 2: AMSR Satellite SST data for October 15, 2014 daytime.

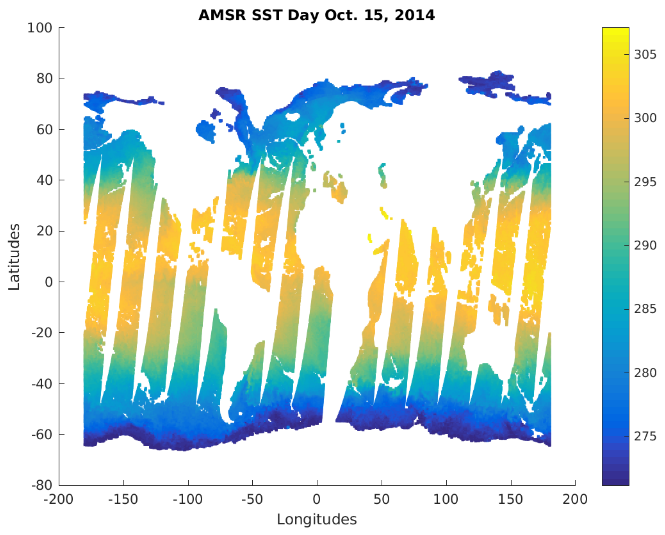
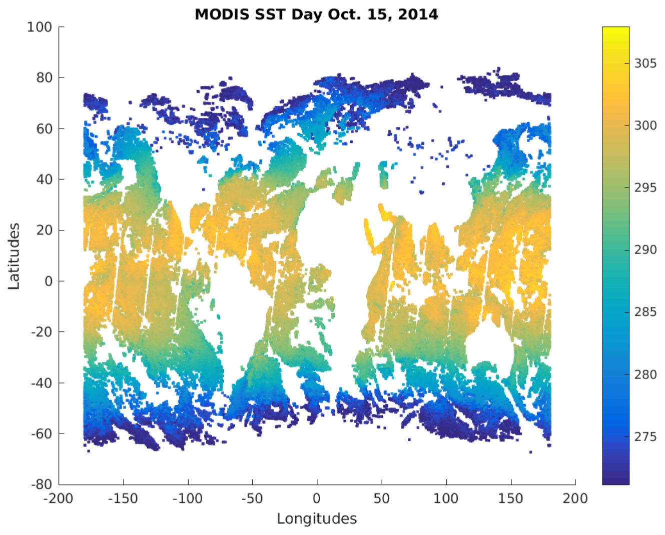


Figure 3: MODIS Satellite SST data for October 15, 2014 daytime. Plot contains every 10th entry from the entire MODIS dataset.



5 Timing Studies

We study how changing both usable memory and number of commissioned cores across computational platforms affect execution time. Moreover, we wish to develop a framework for optimal MRA execution for datasets of various sizes. Presented are experimental results for our MRA implementations with different data sets on computing systems with different amounts of available memory and processing speed. For a particular computational routine and given number of workers, the model is run multiple times in order to assess runtime variability. Throughout, an exclusive node (that is, a node with no other jobs running on it) is used in order to ensure that the entire resources of a given node are available. By doing so, we obtain runtimes that are not affected by competing computational tasks.

5.1 Parallel Processing Environments

We desire to evaluate MRA performance across computing systems ranging from personal computers to high-performance computer (HPC) systems. To do so, we test on three different computing systems and associated parallel processing environments: a personal computer, Cheyenne HPC regular memory nodes, and Cheyenne HPC large memory nodes.

On a 2012 Apple MacBook Pro personal computer with a 2.3 GHz Intel Core i7 processor and 8GB of RAM, the parallel processing environment consists of four workers, one for each available core. Given these hardware limitations, only the Small Satellite Data is tested with MATLAB® 2018a.

We are particularly interested in scaling our MRA to larger datasets and make use of NCAR's Cheyenne HPC. Cheyenne is a 5.34-petaflops HPC platform built by Silicon Graphics International Corporation (SGI) and funded by the National Science Foundations located in the NCAR-Wyoming Supercomputing Center (NWSC). Cheyenne consists of 145,152 2.3-GHz Intel Xeon E5-2697V4 (Broadwell) processors (16 flops per clock) in 4,032 dual socket nodes with 36 cores per node and a combined 313TB memory. The main system consists of 3,186 "regular" nodes with 64 GB memory, and 864 "large memory" nodes each with 128GB memory (`https://www2.cisl.ucar.edu/resources/resources-overview`).

On Cheyenne, for the purposes of comparison, we make use of both types of nodes. On each type of node, various number of workers are commissioned. Tests are replicated five times in order to assess run-time variability. The parallel processing environment on Cheyenne consists of a single node using the MATLAB® Parallel-Computing Toolbox (PCT) license. The PCT allows parallel processing on multiple CPUs or GPUs on one machine with one instance of MATLAB® running. On Cheyenne, the MRA is run using MATLAB® 2018a and we investigate how the number of cores used affects performance.

As will be shown, it is not necessarily advantageous to utilize all workers requested on a node. Utilizing more workers on a node requires portioning memory to each worker regardless of whether they are allocated computations to perform. If the computations are memory constrained, workers must also compete with each other for memory as it becomes available, resulting in runtime spent on communication between the client and workers and some workers being unable to perform computations efficiently due to lack of available memory. The effect of the number of workers commissioned on memory utilization is studied in Section 5.4.

5.2 Small Satellite Data Timing Study

In this portion of the timing study we set $J = 2$, $r = 64$, and $M = 9$. Tests were performed in serial as a baseline for runtimes and then in parallel for comparison. We test both our serial and parallel MRA implementations on computational platforms with increasing numbers of workers and different Cheyenne node types. Where running the `prediction` mode, the prediction grid is set to be a 200×200 grid. Table 3 summarizes the results for both the serial and parallel implementations.

Table 3: Summary of the timing results for the serial and parallel MRA using the Small Satellite Data on both a personal computer and the Cheyenne supercomputer. The names of the MRA modes are given in bold. Each mode was run five times in serial and for a given number of workers (denoted by w). The mean runtime for each mode is given in seconds and its sample standard deviation in parenthesis.

	Likelihood	**Optimize**	**Prediction**
Personal Computer			
Serial	8.62 (0.24)	686.94 (5.24)	13.34 (0.24)
$w = 4$	12.64 (0.26)	655.57 (4.48)	17.61 (1.85)
Regular Cheyenne Node			
Serial	6.75 (0.20)	474.91 (2.08)	9.88 (0.23)
$w = 4$	6.16 (0.18)	398.06 (7.53)	7.81 (0.05)
$w = 12$	4.81 (0.04)	228.67 (2.17)	5.59 (0.06)
$w = 24$	5.00 (0.09)	213.80 (2.15)	5.77 (0.06)
$w = 36$	5.48 (0.05)	223.98 (2.28)	6.18 (0.06)
Large Memory Cheyenne Node			
Serial	6.64 (0.15)	468.32 (2.29)	9.79 (0.34)
$w = 4$	6.08 (0.04)	392.07 (6.74)	7.81 (0.10)
$w = 12$	4.74 (0.05)	223.50 (2.18)	5.57 (0.07)
$w = 24$	4.98 (0.06)	206.76 (2.74)	5.59 (0.08)
$w = 36$	5.36 (0.04)	214.16 (1.96)	6.05 (0.06)

As can be seen in Table 3, for the serial implementation, increasing processor speed significantly decreases run-times across MRA modes where as increasing available memory facilitates only a small decrease in runtime. This is evident from the differences between the personal computer and Cheyenne timings. The relatively small difference in timings between the regular memory node and the large memory node is to be expected, since the execution is not memory-limited. Interestingly, although the `likelihood` mode shows shorter execution times in serial than in parallel with $w = 4$ workers, the `optimize` mode consistently executes faster in parallel than in serial.

For the parallel implementation, runtimes between the regular and large memory nodes are effectively the same within each MRA mode with execution times on the large memory nodes being slightly shorter. That is, increasing the amount of memory available to all the workers does not have a significant effect on runtime performance, which is to be expected as there is no change in processing power between the two types of nodes and the computation is not memory limited. On the other hand, increasing the number of workers available in the parallel inference did decrease the run time, indicating that for smaller datasets, increasing the number of workers can benefit runtimes until a certain threshold is reached. In this case test, employing $w = 24$ workers consistently has the shortest runtimes. This suggests that at least for smaller datasets, there is little benefit gained by increasing the number of workers past a certain threshold. Further experiments to investigate this hypothesis can be found in Section 5.4.

When profiling the MRA on a local machine over the small datasets, we find that the most computationally intensive functions are the `parfor`-loops used for the creation of the prior and posterior inference found in `MRA.m` and `create_prior.m`. The expense associated with the `parfor`-loops is a result of the parallel implementation and the expense associated with calling the `evaluate_covariance` function is due to evaluating the exponential function repeatedly. Further profiling reveals most of this computational expense is a result of "self-time" (calling built-in MATLAB® functions, communication overhead between the client and workers, etc.), which indicates that they are running about as efficiently as possible.

5.3 NASA Sea Surface Temperature Data Timing Studies

Due to the large size of the AMSR and MODIS data, a timing study can only be performed on Cheyenne. We study how changing the number of workers available affected MRA runtime performance.

We begin by testing the serial and parallel implementations of the MRA over both datasets (or subsets thereof) for daytime October 15, 2014 with the `likelihood` and `prediction` modes. The serial implementation (denoted "Serial" in Table 4) and the parallel implementation (denoted by the number of workers used, w) of the MRA are tested. The `likelihood` and `prediction` modes are run on both the large memory and regular Cheyenne nodes using the PCT. The `optimize` mode is not tested as it is essentially repeated application of the `likelihood` routine and runtimes are largely dependent on the upper and lower bounds defined for the parameter space search.

We choose to run the MRA over subsets of the AMSR and MODIS data in order to evaluate runtimes at increasing data sizes. The entire AMSR daytime dataset consists of $n = 2,441,405$ observations on a 25km scale where as the entire MODIS dataset consists of $n = 47,567,759$ observations on a 1km scale. To create a subset of the data of proportion $1/n$, every n^{th} data point is selected. For example, in MATLAB®, `observations = observations(1:2:end);` would create a 1/2 subset of the data. By doing so, the domains between datasets remain fairly consistent, while the density of observations between them varies. Throughout we set `theta = [2.117,1];` and `varEps = 0.001` as determined by running the `optimize` routine over the 1/10 AMSR data. Where the `prediction` mode is tested, the prediction grid is set to be a 200×200 grid.

5.3.1 1/2 Subset AMSR Data Timing Study

Table 4 summarizes results of running the MRA over a 1/2 subset of the AMSR data in serial on both the regular memory and large memory Cheyenne nodes. For these tests, we set $J = 2$, $r = 64$, and $M = 13$.

Table 4: Summary of the results for the MRA using 1/2 the AMSR data ($n = 1,220,703$). Tests run in serial are labeled "Serial" whereas parallel tests are labeled by the number of workers commissioned (denoted by w). Where the MRA execution was possible, each test was run five times. The mean run time for each mode is given in seconds and its sample standard deviation in parenthesis.

	Likelihood	**Prediction**
Regular Cheyenne Node		
Serial	155.67 (2.49)	178.70 (1.60)
$w = 12$	Insufficient Memory	Insufficient Memory
$w = 24$	Insufficient Memory	Insufficient Memory
$w = 36$	Insufficient Memory	Insufficient Memory
Large Memory Cheyenne Node		
Serial	153.99 (2.76)	178.75 (1.91)
$w = 12$	89.13 (0.51)	105.04 (0.94)
$w = 24$	88.99 (0.50)	105.48 (0.99)
$w = 36$	88.91 (0.38)	102.30 (0.71)

From the tests performed on half the AMSR data, it appears that for the specified parameters ($M = 13$, $J = 2$, and $r = 64$), our parallel implementation's memory requirements result in fatal executions of the regular memory nodes. On the large memory nodes there is little-to-no benefit gained by increasing the number of workers from $w = 12$ to $w = 36$. This is likely because as the number of workers increases, portions of available memory must be allocated to the workers regardless of whether they are assigned computations. Additionally, workers must compete with each other for memory as it becomes available resulting in diminished benefit due to inefficient resource allocation.

An important question is to determine the effect of how the number of levels, M, and the number of knots, r, affect runtime. In order to investigate the runtime behavior of the MRA about the runtime minimums observed, we test various settings for M and r over half the AMSR data with 36 workers. Each test is run five times. Results are summarized in Table 5.

Table 5: MRA `likelihood` mode runtime performance for different values of M and r over 1/2 AMSR data on a large memory node with 36 workers. Where execution is nonfatal, each test is run five times. Execution time means are given in seconds and its sample standard deviation in parenthesis.

r \ M	$M = 12$	$M = 13$	$M = 14$
$r = 49$	45.19 (0.41)	57.57 (0.69)	109.52 (1.57)
$r = 64$	56.69 (0.41)	87.52 (1.11)	185.42 (5.90)
$r = 81$	75.38 (0.70)	133.54 (1.37)	Insufficient Memory

In only one configuration (setting $M = 14$ and $r = 81$) was computation not possible over the 1/2 AMSR dataset. Table 5 demonstrates that increasing either M or r generally increases runtime, however increasing M has a more significant increase on runtime than increasing r. The more significant runtime increase associated with increasing M is a result of the number of partitions at each level growing as a power of J. For instance, in this test with $J = 2$, when $M = 13$, there are $J^{M-1} = 2^{12} = 4,096$ partitions at the finest level. By holding J constant and increasing M to $M = 14$, there are $J^{M-1} = 2^{13} = 8,192$ partitions at the finest level, resulting in nearly a double in execution time where evaluation was possible.

Recall that most of the MRA inference depends on calculations with $r \times r$ matrices. Increasing r consequently increases the size of the matrices calculations are performed with. Results from Table 5 show that while increasing r does increase runtime, it does so less significantly because adjustments of r occur on a finer scale. That is, while small adjustments in M increase the total number of partitions as a power of J, r takes on square integer values. As to be expected, holding M constant, roughly doubling r results in roughly doubling the runtime, as consequently calculations are performed with matrices that are roughly four times as large.

5.3.2 Entire AMSR Data Timing Study

Here we test our parallel MRA implementation over the entire AMSR data set with $n = 2,441,405$ observations. For these tests we set $M = 14$, $J = 2$, and $r = 49$. We run the `likelihood` mode over the entire AMSR data in serial and in parallel on a large memory Cheyenne node. Table 6 summarizes the results. Each test is run five times.

Table 6: Summary of the MRA timing results using the AMSR data ($n = 2,441,405$). Tests run in serial are labeled "Serial" whereas parallel tests are labeled by the number of workers commissioned (denoted by w). Each test is run five times. The mean run time for each routine is given in seconds and its sample standard deviation in parenthesis.

	Likelihood	**Prediction**
Large Memory Cheyenne Node		
Serial	321.22 (4.12)	363.28 (4.34)
$w = 12$	135.75 (0.85)	160.99 (2.67)
$w = 24$	136.67 (2.91)	160.47 (3.70)
$w = 36$	143.52 (3.00)	169.10 (5.20)

Again, we see that the parallel MRA has lower execution times where it is possible to execute the MRA in parallel given the available memory. Here, using fewer knots reduces the memory usage sufficiently to allow successful execution, but also approximates the spatial field with fewer basis functions. Interestingly, increasing the number of workers deployed increases runtime, in contrast to previous results from the Small Satellite Data. This is likely due to encountering memory constraints in execution. It is known that employing more workers increases memory overhead. Tests in Table 6 demonstrate that employing more workers can increase runtimes when computation is memory-limited. We test the limits of this observation in the following timing study.

5.3.3 1/17 Subset MODIS Data Timing Study

Here we test the MRA over a 1/17 subset of the MODIS data with $n = 2,798,104$ observations using the `likelihood` and `prediction` modes. We hold $M = 14$, $J = 2$, and $r = 49$ constant. Table 7 summarizes the results. Each test is run five times.

Table 7: Summary of MRA timing results using 1/17 MODIS data ($n = 2,798,104$). Tests run in serial are labeled "Serial" whereas parallel tests are labeled by the number of workers commissioned (denoted by w). Where execution is nonfatal, each test is run five times. The mean run time is given in seconds and its sample standard deviation in parenthesis.

	Likelihood	**Prediction**
Large Memory Cheyenne Node		
Serial	463.67 (8.37)	513.71 (7.28)
$w = 12$	191.92 (2.96)	218.16 (7.61)
$w = 24$	171.17 (2.72)	Insufficient Memory
$w = 36$	Insufficient Memory	Insufficient Memory

As can be seen in Table 7, the parallel MRA hastens the `likelihood` runtime with $w = 12$ and $w = 24$ workers. Alternatively, employing $w = 36$ workers results in fatal execution for $n = 2,798,104$ observations on a large memory node. Utilizing $w = 12$ workers hastens `prediction` runtimes in comparison to the serial implementation, however there is insufficient memory available to execute in parallel with $w = 24$ and $w = 36$ workers. It appears we have found the limit of parallel computation in this processing environment. As a side remark, successful parallel MRA execution of the `likelihood` and `prediction` modes using the 1/17 MODIS dataset was possible on a Dell personal desktop with 32GB memory, two Intel Xeon processors at 2.33 GHz, running Windows 10. This indicates that memory swapping may facilitate similarly sized computations on smaller machines where there is disk space. We examine the effect of increasing the number of workers on memory consumption and runtime in the following section.

5.4 Memory Profiling Tests

The timing studies described in sections 5.1 - 5.3 involved measurements of a single metric (total execution time) over multiple replicates of various datasets/model configurations, in order to gain insight into the total processing burden required by the code in these configurations. These studies, however, do not investigate other aspects of computational expense which may also be of interest (e.g. memory use), nor can they identify which phases of the codebase execution (i.e. which part of the overall algorithm) may be the most costly, in either time or memory.

Profiling the memory use over the execution of the algorithm (that is, tracking how memory use changes during execution, rather than simply logging terminal memory use) can be useful when attempting to identify portions of the code that should be examined for potential improvement. Additionally, the ability to see the actual memory use during execution of a particular configuration is helpful when considering the feasibility of similar executions on other computational platforms (e.g. profiling studies performed on a high performance computing platform can inform decisions about whether a similar volume of data and model configuration should be attempted on a personal laptop). To investigate this aspect of computational expense, a series of memory profiling tests were performed on the MRA codebase in various computing environments, processing modes, and data volumes.

It is worth noting that these profiling tests are more labor intensive to perform than the base timing studies, requiring generation, reconciliation, and analysis of two lengthy data streams (specifically, a memory use time series and a timestamped program process log) for each profile run. In contrast, a base timing study generates only a single value (total duration of execution) for any particular trial. This is the chief reason why the profiles described in this section are not run on as full and varied a set of datasets/model configurations as the base timing studies described earlier.

5.4.1 Standard Cheyenne Node Memory Use Study, 64 Knots

A profiling study was performed on an individual Cheyenne standard computing node on serial and parallel executions (in the single-node Parallel Computing Toolbox schema). The tests followed the following process:

1. An interactive session on a single compute node was initiated (via the request `qsub -I -l select=1:ncpus=36:mpiprocs=36 -l walltime=03:00:00 -q regular -A` *`project_code`*).
2. An additional connection to the interactive session was established in a second terminal via `ssh`.
3. A time-stamped memory logging bash script (based on the `free` command) was started in one terminal, with output piped to a text file.
4. The MRA code to be tested was executed (in likelihood computation mode) in the other terminal (the code was modified to include timestamp outputs at important process milestones).
5. When the MRA code was complete, a log of the terminal output from the MRA terminal (with the process milestone timestamps) was exported, and the memory logging script from the other terminal was killed.
6. The memory logs were imported into a spreadsheet program, reconciled with the milestone logs, and used to create plots, which were then annotated.

The testing scenarios used subsets of the AMSR dataset, with the resolution determined by the `find_num_levels_suggested` function, given the subset size and a fixed knot

Table 7.1: Standard Compute Node Memory Profile Study Test Configurations

Test Environment	Workers	AMSR Fraction	Sample Size (n)	Resolution (M)
Serial	NA	1/10	244,141	12
Serial	NA	1/5	488,281	13
Serial	NA	1/4	610,353	14
Serial*	NA	1/2	1,220,705	15
Parallel	2	1/10	244,141	12
Parallel	3	1/10	244,141	12
Parallel	4	1/10	244,141	12
Parallel	8	1/10	244,141	12
Parallel	12	1/10	244,141	12
Parallel	24	1/10	244,141	12
Parallel	36	1/10	244,141	12
Parallel	2	1/5	488,281	13
Parallel	3	1/5	488,281	13
Parallel	4	1/5	488,281	13
Parallel	8	1/5	488,281	13
Parallel	12	1/5	488,281	13
Parallel	24	1/5	488,281	13
Parallel	36	1/5	488,281	13
Parallel*	2	1/4	610,353	14

and partitioning schema ($r = 64$, $J = 2$). The various scenarios and configurations are summarized in Table 7.1.

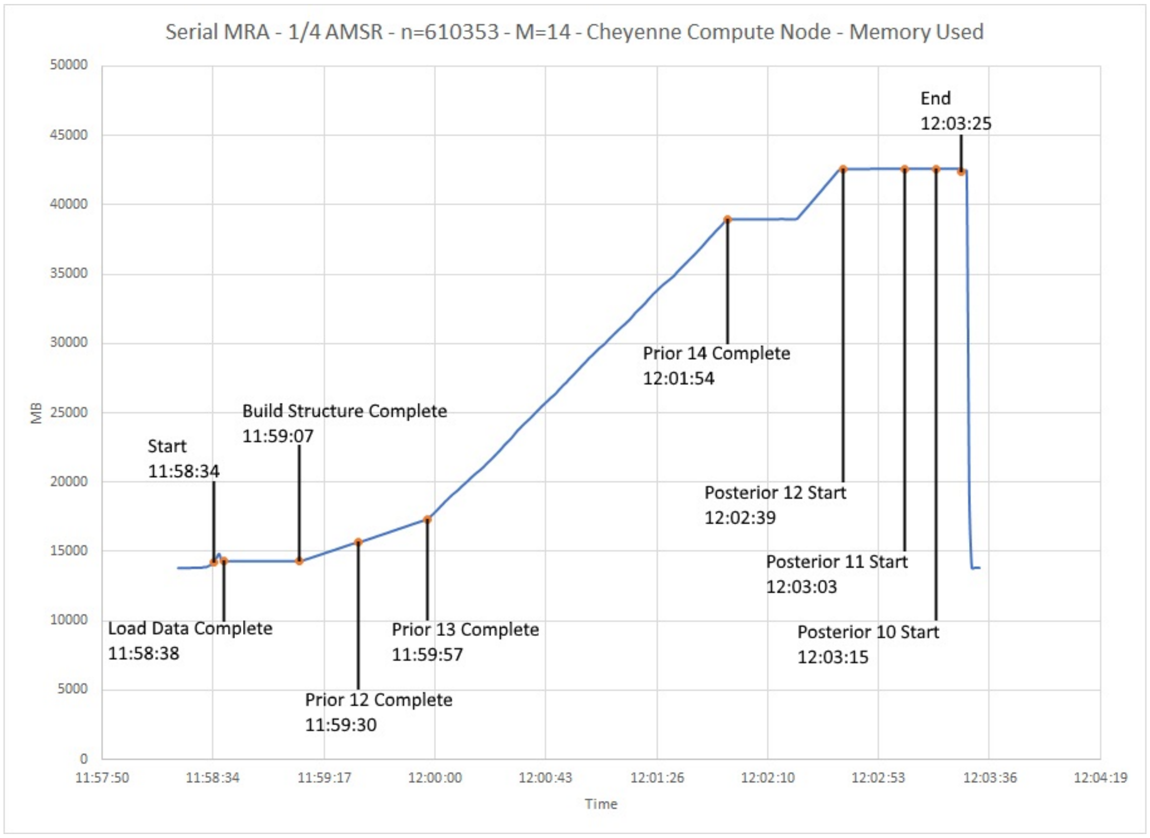
Testing scenarios marked with * in Table 7.1 did not successfully run to completion (due to memory overruns during the computation of the finest level prior).

5.4.2 Serial Tests

An example plot of memory use over time, annotated with program progress, is provided below for the serial execution over the 1/4 subset (Figure 4). This particular profile (taken over a relatively large sample of the AMSR data, with an associated high resolution level) clearly illustrates the changes in memory allocation rate that occur during the prior calculations. The other serial executions are shown as reference profiles in the Parallel Tests portion (5.4.3).

The memory usage pattern shown in Figure 4 is similar to the patterns shown in the other serial executions; a small bump in memory use during the data import phase, followed

Figure 4: 1/4 Subset Serial MRA Runtime Profile

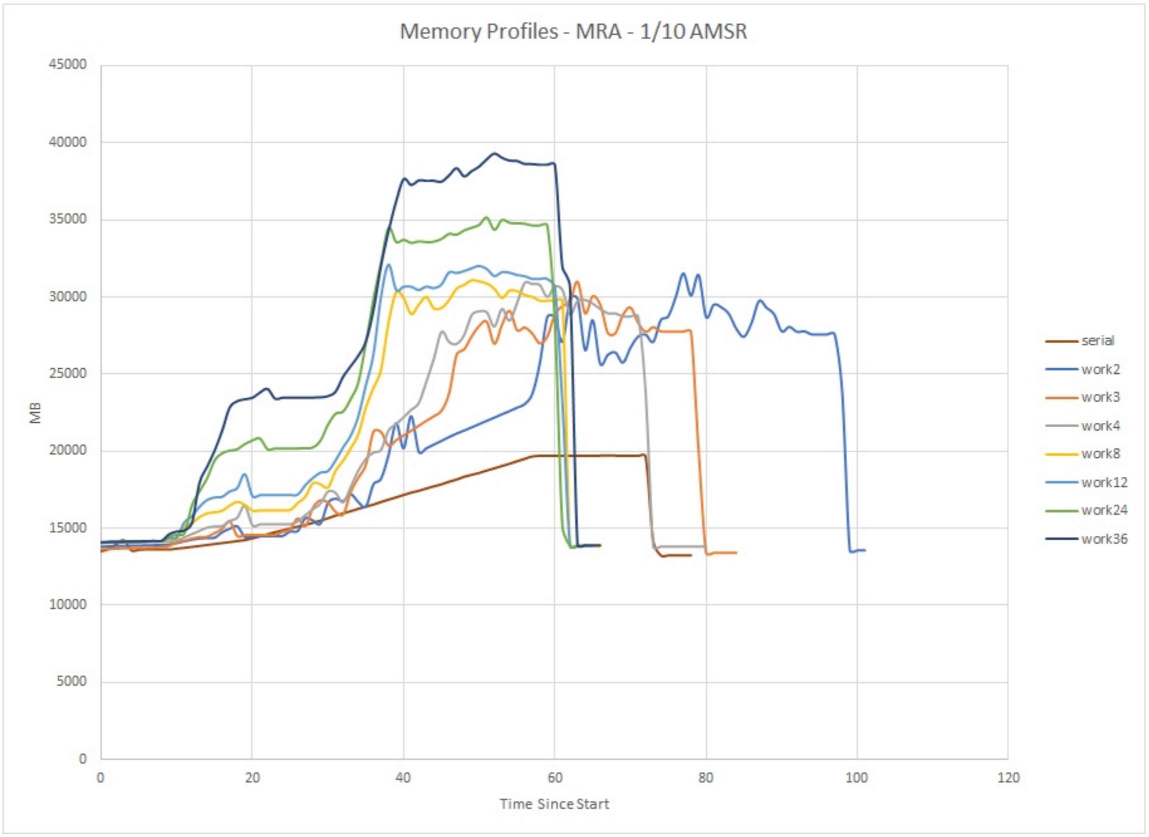


by relatively stable use (due to preallocation) during the build structure phase, followed by linear increase during the prior calculation phases (though at potentially different slopes during different prior levels), with more modest increases mixed with plateaus during the posterior calculation phases, finally followed by memory release in the seconds following completion (and termination of the MATLAB® session).

5.4.3 Parallel Tests

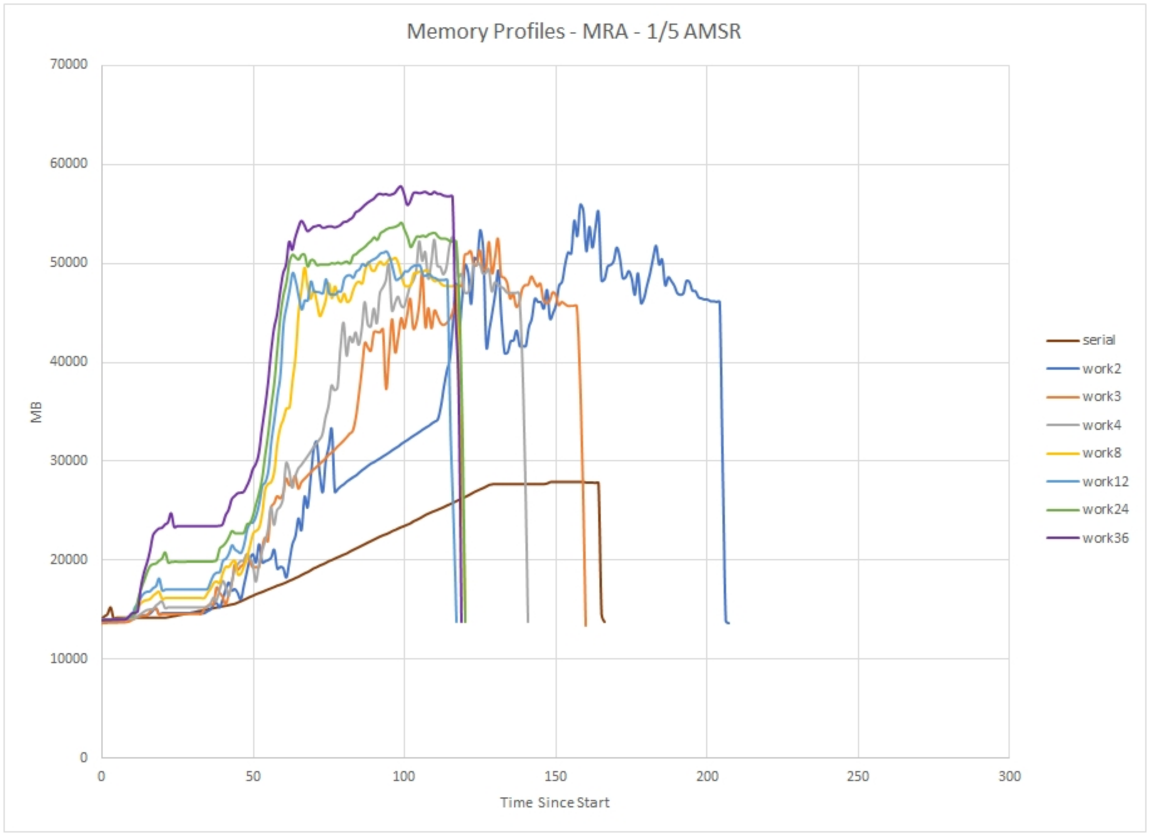
Focusing on the 1/10 subset ($n = 244,141$, $r = 64$, $J = 2$, $M = 12$), the codebase was profiled during parallel execution (in likelihood computation mode) with $w = 2$, $w = 3$, $w = 4$, $w = 8$, $w = 12$, $w = 24$, and $w = 36$ workers. The profiles of these executions are shown in Figure 5, along with the associated serial execution profile for comparison. In this particular test, note that execution times for parallel cases are only faster than the serial case once the worker count hits four, but that execution times cease to improve significantly once the worker count hits eight. Additionally, peak memory use for all parallel cases is substantially higher than for the serial case. Of particular interest are the profiles for the parallel cases $w = 8$, $w = 12$, $w = 24$, and $w = 36$; these show scaling in memory use with no corresponding improvement in execution time, suggesting a bottleneck in parallelization.

Figure 5: 1/10 Subset MRA Parallel Runtime Profiles



Shifting focus to the 1/5 subset ($n = 488{,}281$, $r = 64$, $J = 2$, $M = 13$), the codebase was again profiled during parallel execution with $w = 2$, $w = 3$, $w = 4$, $w = 8$, $w = 12$, $w = 24$, and $w = 36$ workers. The profiles of these executions are shown in Figure 6, along with the associated serial execution profile for comparison. In this scenario (with twice as many observations and a unit increase in resolution level, relative to the 1/10 subset tests), the serial execution is surpassed by the parallel execution at a lower worker count, with only the parallel execution over two workers exhibiting a longer execute time than the serial case. Additionally, these profiles display the same lack of improvement with worker count increase seen in the 1/10 subset tests, the bottleneck again showing up at eight workers. In contrast with the 1/10 subset tests, the memory limits of the standard Cheyenne node (64GB Total, 45GB Usable) begin to pose a concern for the parallel executions (though the serial execution completes quite comfortably, never exceeding even 50% use of the memory available).

Figure 6: 1/5 Subset MRA Parallel Runtime Profiles



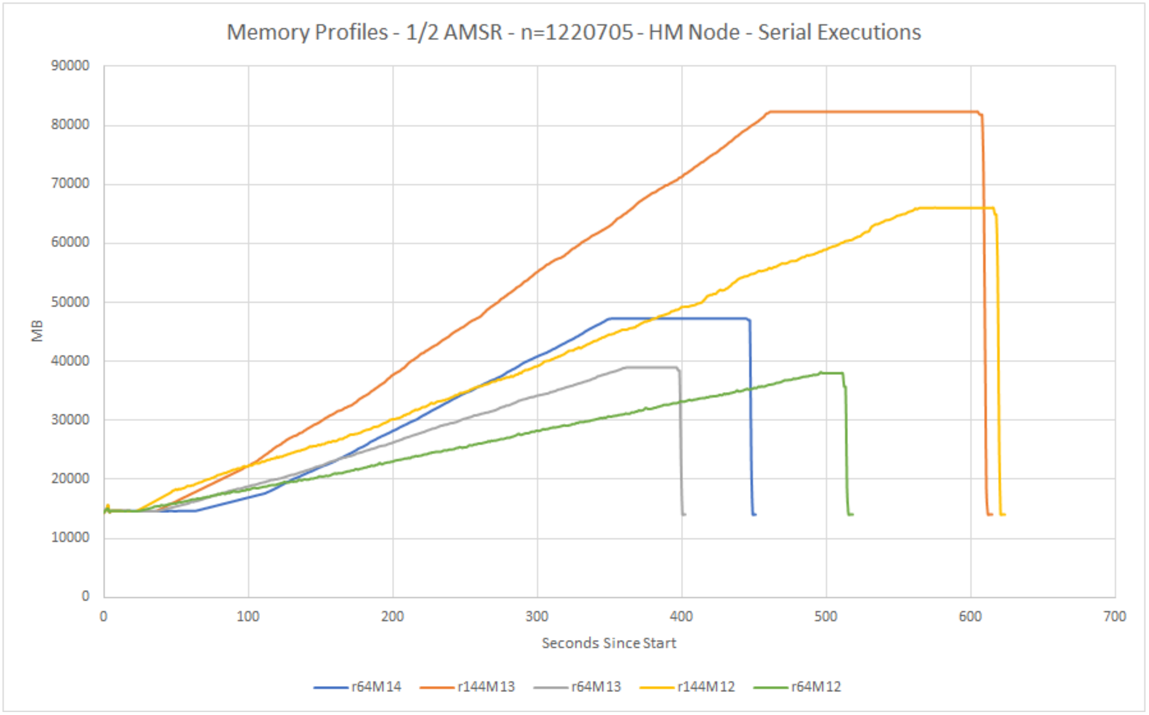
5.5 High Memory Cheyenne Node Memory Use Study, Varying Knots and Levels

Further memory profiling studies were performed on a high-memory node of Cheyenne (128GB Total, 109GB Usable). In this series of tests, half of the AMSR data was imported ($n = 1,220,705$) and the codebase was executed (again, evaluating the likelihood), over two knot counts ($r = 64$ and $r = 144$). In earlier studies, the function `find_num_levels_suggested` was employed to determine the resolution level that produced an average number of observations per finest level subregion approximately equal to the number of knots per subregion, this approach leading to matrix operations involving matrices of approximately the same dimension at all levels (an attempt to keep computational load consistent). In this series of tests, this suggested number of levels was disregarded, and instead, lower resolutions were employed, leading to fewer levels, but involving larger dimension matrices at the finest level, in order to investigate the computational tradeoffs involved. One immediate advantage of using fewer levels is reduced memory usage; this volume of data could not be successfully analyzed at the recommended resolution for either knot count even on high-memory nodes. For reference, `find_num_levels_suggested` suggests $M = 15$ for $n = 1,220,705$, $r = 64$, and $J = 2$. Instead, $M = 12$ and $M = 13$ were evaluated for both knot counts. Additionally, $M = 14$ was evaluated for $r = 64$.

5.5.1 Serial Studies

The memory profiles of the serial tests run are shown in Figure 7. For each of these profiles, the long term linear increase in memory usage corresponds to the computation of the finest level prior, and the plateau in usage that follows corresponds to the computations of the posteriors.

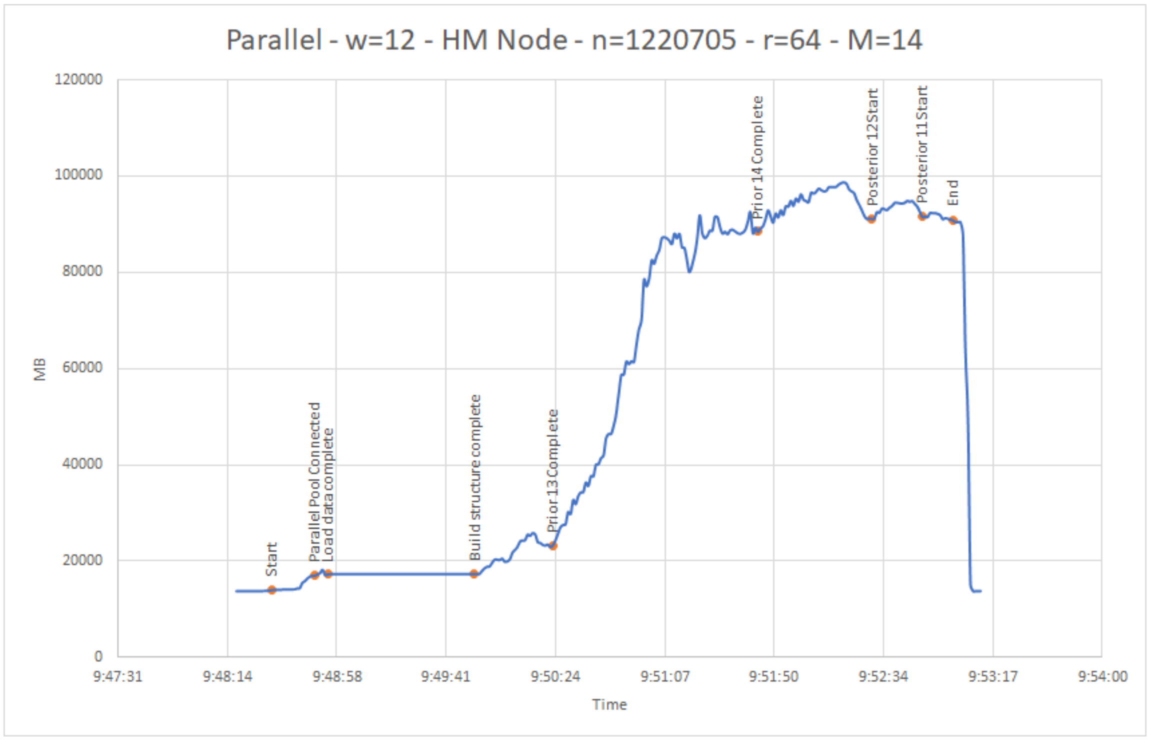
Figure 7: 1/2 Subset MRA Serial Runtime Profiles, High Memory Node



5.5.2 Parallel Studies

With the serial executions available as baselines, some parallel executions were profiled with $r = 64$ knots (an attempt to profile a parallel execution with $r = 144$ knots resulted in a memory overage even when constrained to $w = 2$ workers over $M = 12$ levels). These executions were over $M = 12, M = 13$, and $M = 14$, and with worker pools of $w = 2$ and $w = 12$. A sample annotated memory profile for a parallel execution with $w = 12$ workers over $M = 14$ levels is provided in Figure 8. Note that (just as in the serial case), the largest increases in memory usage occur during the calculation of the the prior at the finest resolution, which is also the most costly phase in terms of execution time (though this phase does represent proportionately less of overall execution time in the parallel case than in the serial). As one might expect, memory usage is much more uneven during the parallel executions than in the serial executions, as individual workers acquire and release memory as they obtain and complete their tasks; in this context, task sizes are unlikely to be well-balanced unless the observed data are distributed over the spatial domain of interest (and subregion division schemes and resolutions are selected) in such a fashion that at the finest level, each subregion contains the same number of observations. It is conceivable that in a remote-sensing context with observations taken over a regular grid, and with an analysis configured by a careful user, this situation may indeed occur, but it should not be expected in general. Thus, during parallel executions, it should be expected that some workers will require larger memory allocations for longer periods than others.

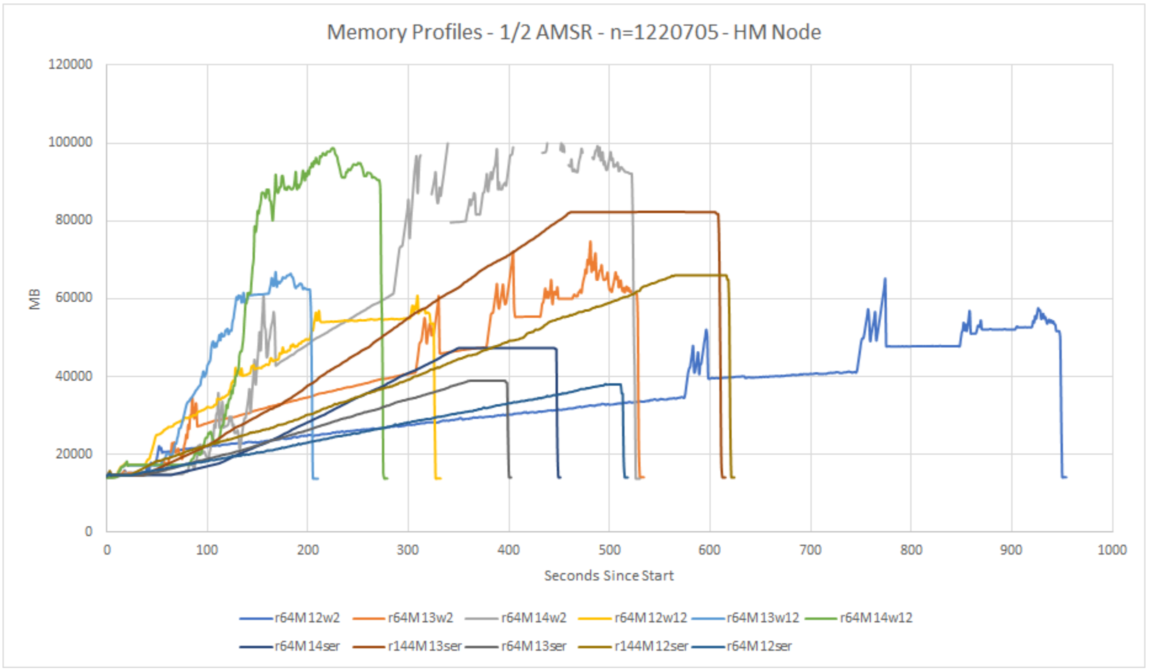
Figure 8: 1/2 Subset Parallel Runtime Profile, High Memory Node



As an augmentation of Figure 7, a plot of all profiles (both serial and parallel) gathered during this series of tests (aligned by starting time) is provided in Figure 9. Note that all of the $w = 12$ worker executions complete in less time than the fastest serial execution ($r = 64$ knots, $M = 13$ levels), particularly impressive is the case of the parallel execution with $r = 64$ knots, $M = 14$ levels, which theoretically would also outperform the aforementioned serial execution in terms of approximation quality (due to the higher resolution employed). Note that the memory usage log associated with the $r = 64$, $M = 14$ parallel case with $w = 2$ workers was incomplete (cause unknown; perhaps execution of the `free` command [the command that underpins the memory log script] may be slightly too costly when so much memory is being accessed in such a volatile pattern), but that test did successfully run to completion. Missing data from that log is represented with discontinuities in the associated line plot.

Generally, in the current codebase implementation, `likelihood` computations can be hastened significantly through parallelization, but this comes at the cost of increased memory use. Additionally, run speed improvement for a particular configuration as a function of workers employed appears to plateau, at least over the testing configurations examined in this series of tests.

Figure 9: 1/2 Subset MRA Runtime Profiles, High Memory Node



5.6 MATLAB® `parallel.pool.Constant` Tests

The MATLAB® function `parallel.pool.Constant()` potentially presents one opportunity for code improvement. From the MathWorks documentation:

"`C = parallel.pool.Constant(X)` copies the value X to each worker and returns a parallel.pool.Constant object, C, which allows each worker to access the value X within a parallel language construct (parfor, spmd, parfeval) using the property C.Value. This can improve performance when you have multiple parfor-loops accessing the same constant set of data, because X is transferred only once to the workers."

A script to test this function in a context similar to that implemented in the MRA codebase (that is, with a `parfor` loop nested inside a standard `for` loop) was written, provided below:

```
%% Parallel Pool Constant Memory and Timing tests
% Req. R2015b or newer

%% Setup
clear all;
disp('Memory Use at Init');
[userview,systemview] = memory;
userview.MemUsedMATLAB

data = rand(1000);
disp('Memory Use after Rand Data Gen');
[userview,systemview] = memory;
userview.MemUsedMATLAB

parpool('local',4);
disp('Memory Use after Pool Startup');
[userview,systemview] = memory;
userview.MemUsedMATLAB

%% Index into data
disp('Memory Use After Each Parfor Worker Indexes Into Data');
tic
for ii = 1:10
    parfor jj = 1:10
        x(ii,jj) = data(ii,jj);
        [userview,systemview] = memory;
        disp(userview.MemUsedMATLAB);
    end
end
```

```
toc
clear x;
clear ii;
clear jj;

%% Pass as constant
disp('Memory Use After Each Parfor Worker Uses par pool const');
tic
c = parallel.pool.Constant(data);
for ii = 1:10
    parfor jj = 1:10
        y(ii,jj) = c.Value(ii,jj);
        [userview,systemview] = memory;
        disp(userview.MemUsedMATLAB);
    end
end
toc
clear y;
clear ii;
clear jj;
clear c;
```

This script was executed as shown (creating an 8 MiB `data` array for passing), and also with the dimensions of the `data` array both increased by a factor of ten (creating an 800 MiB array). Loop execution times are provided in Table 8.1.

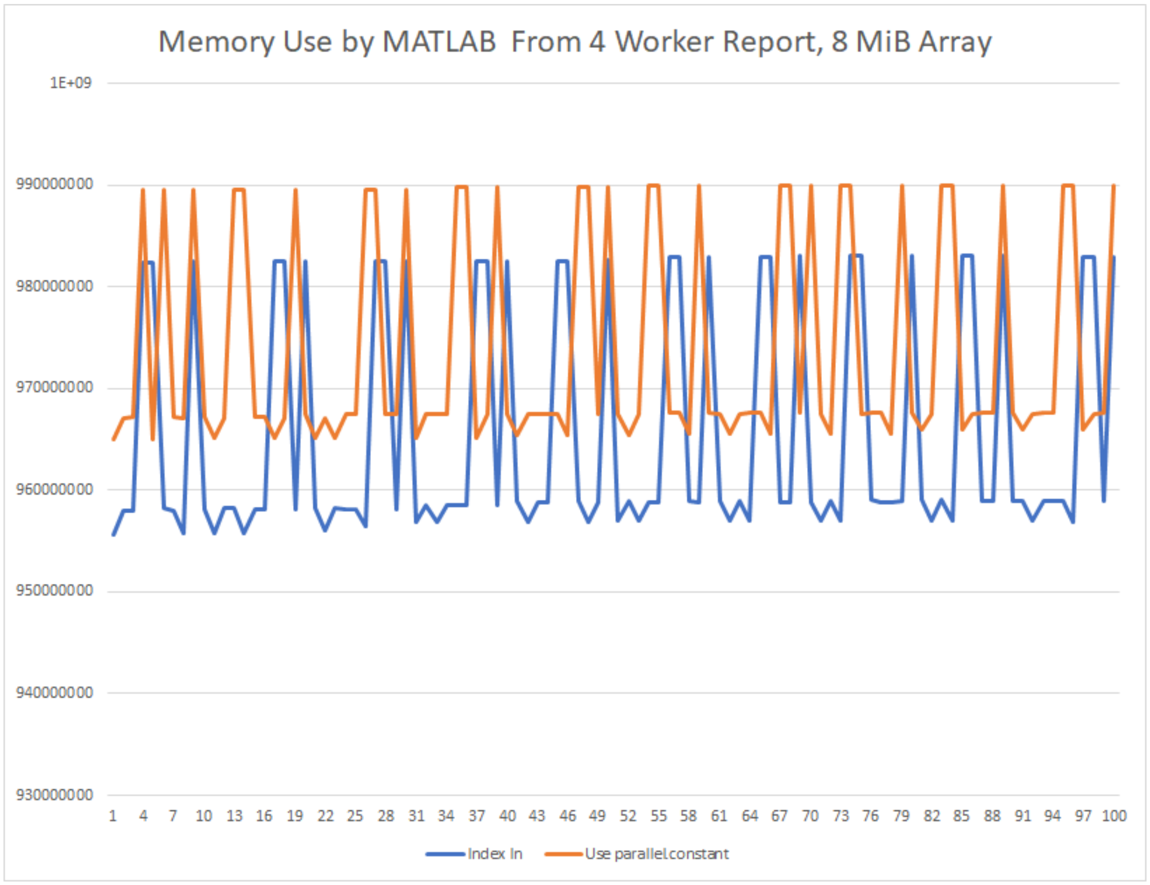
Table 8.1: Loop Execution Times for `parallel.pool.Constant` tests

`data` Array Size	Direct Indexing	`parallel.pool.Constant` Object
8 MiB	1.21s	1.42s
800 MiB	1.60s	9.93s

In this instance, standard array indexing was faster; however, our interest is also in memory use. A similar pattern appears in those data, with the memory use output by each worker at each iteration of the nested loop shown in Figure 10 for the 8MiB case (sorted chronologically), and in Figure 11 for the 800MiB case (sorted by .Constant use ascending).

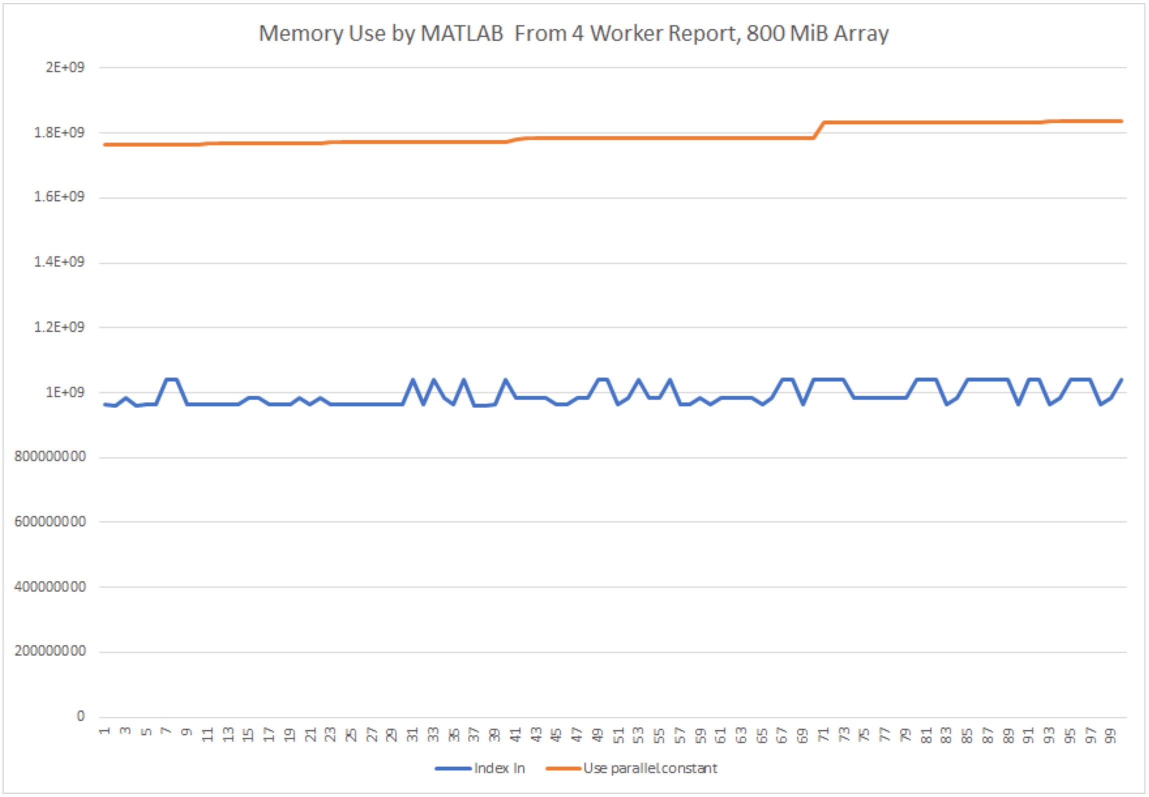
To control for potential order effects, the script was reversed, with the "pass as constant" block executed before the "index in" block. Similar results were observed. Based on these observations, the `parallel.pool.Constant()` approach is unlikely to provide

Figure 10: Memory Use by MATLAB® for `parallel.pool.Constant` Testing, 8MiB Case



an advantage in either runtimes or memory usage for the current implementation of the parallel MRA codebase.

Figure 11: Memory Use by MATLAB® for `parallel.pool.Constant` Testing, 800MiB Case



6 Discussion and Conclusions

Our parallel MRA implementation often provides faster runtimes in comparison to the serial implementation where there is sufficient available memory. If memory is the primary concern, or the dataset of interest is relatively small, serial execution may be preferable. In analyzing the computational costs associated with the parallel MRA we observe a trade-off between runtime and memory consumption. The incurred memory increase in exchange for hastened runtimes was shown to be highly dependent on model parameters and the number of workers employed. This work identified some useful rules-of-thumb when applying our MRA implementations to datasets of various sizes.

We saw in Section 5.3.1 that adjusting values of M, J, and r can have significant effects on runtime. Since the number of partitions at each level grows as a power of J, there are $\sum_{m=0}^{M-1} J^m$ total partitions. Consequently, small changes in the value of M have significant effects on runtimes. Assumed in the calculation of `find_num_levels_suggested` is that observations are spatially distributed uniformly across the domain. Often this assumption may not be valid as is the case with the AMSR and MODIS datasets. At the finest resolution, inference is computed using matrices with dimensions determined by the number of observations within each region. As a result, some workers will perform calculations with fewer numbers of larger matrices than others, which may be preferable to performing computations with many more matrices of the same size.

Additionally, the number of knots, r, plays a large role in runtime. Increasing r allows the user to approximate the spatial field $y_0(\cdot)$ with greater accuracy (i.e., with more basis functions), however doing so will increase runtime, and in some cases can use too much memory. Alternatively, decreasing r approximates $y_0(\cdot)$ with less accuracy but can allow for computations that would otherwise be infeasible for a given M and J. In Section 5.3.3 we saw that adjusting r allows a finer scale runtime adjustments than adjusting M, and may be better for fine tuning model accuracy versus runtime.

Although each dataset must be considered on its own, we recommend first letting $J = 2$, letting $r = 64$, and using the `find_num_levels_sugested` function to determine an estimate of M. Then, execute `likelihood` mode with estimates for the parameters `theta` and `varEps`. Depending on the results of this execution, adjusting r and M appropriately can be used to reduce the `likelihood` runtime. With M, J, and r chosen, upper and lower bounds for the parameters can be set and estimated through the `optimize` mode. Once the `optimize` mode has found a local minimum in the parameter space, the `prediction` mode will execute specifying the MRA for $y_0(\cdot)$.

Future work will likely attempt to avoid the memory overhead created by restructuring the code to work across nodes. Given that our parallel implementation's hardest current limit is the amount of memory for a single node, by distributing the computation across nodes we may be able to alleviate some of these issues and push the MRA to tackle even larger problems efficiently.

Appendix A MRA Functions

Table 10: Functions in the MRA model and their respective inputs and outputs.

Function	Inputs	Outputs
build_structure.m	NUM_LEVELS_M, NUM_PARTITIONS_J, NUM_KNOTS_r, domainBoundaries, offsetPercentage, varargin	knots, partitions, nRegions, outputData, predictionLocations
evaluate_covariance.m	locs1, locs2, theta	R
create_partition.m	xmin0, xmax0, ymin0, ymax0, NUM_PARTITIONS_J	xmin, xmax, ymin, ymax
create_prior.m	theta, NUM_LEVELS_M, knotsb, RpriorCholb, KcBb, dataj, varargin	RpriorCholj, KcBc, Atj, wtj, retlikpred
create_knots.m	xmin, xmax, nx, ymin, ymax, ny, offsetPercentage	X, Y
find_ancestry.m	index, nRegions, NUM_PARTITIONS_J	i_ancestry
find_index.m	level, tileNum, nRegions	index
find_parent.m	index, nRegions, NUM_PARTITIONS_J	l_parent, t_parent, i_parent
load_data.m	dataSource, nXGrid, nYGrid, offsetPercentage	data, regressionModel, domainBoundaries, predictionVector, theta, varEps
MRA.m	theta, data, knots, NUM_LEVELS_M, NUM_PARTITIONS_J, nRegions, isPredicting, varargin	sumLogLikelihood, predictions
posterior_inference.m	RpriorCholj, wtildeChildren, AtildeChildren	wtildeCurrentj, AtildeCurrentj, LogLikelihoodj, RposteriorCholj, Kcwj, KcAj
predict.m	posteriorMeanj, posteriorVariancej, Btildej, RposteriorCholb, KcAb, Kcwb	predictionsj

Table 11: MRA function descriptions.

Function	Description	Functions Used
`build_structure.m`	Builds the nested partitioning structure as a function of the levels.	`create_knots.m`, `create_partition.m`, `find_index.m`, `find_parent.m`
`evaluate_covariance.m`	Generic covariance function. Presently set to an exponential.	∅
`create_partition.m`	Specifies child partitions.	∅
`create_prior.m`	Calculates prior quantities for each region. Contains optional input and output arguments depending on whether at finest level.	`evaluate_covariance.m`
`create_knots.m`	Creates the knots for each partition at each level.	∅
`find_ancestry.m`	Finds the level, tile number, and continuous index of the ancestry for a given region.	`find_parent.m`
`find_index.m`	Finds the continuous index given the level and tile number as inputs.	∅
`find_parent.m`	Finds the level, tile number, and continuous index of the parent for a given region.	∅
`load_data.m`	Data files are loaded and given a variable name to be referenced by `dataType`.	∅
`MRA.m`	Performs the MRA. Begins by calculating the prior quantities and then calculates the posterior distribution.	`find_ancestry.m`, `create_prior.m`, `posterior_inference.m`, `predict.m`
`posterior_inference.m`	Performs the posterior inference.	∅
`predict.m`	Calculates the posterior predictive distribution.	∅

Appendix B Codebase Object Descriptions

Table 12: Objects in the codebase and their descriptions.

`A`	Cell array of the $\mathbf{A}^{k,m}_{j_1,\ldots,j_m}$'s for a given level.
`AtildeCurrent`	Cell array containing current $\mathbf{A}^{k,m}_{j_1,\ldots,j_m}$ for a particular subregion.
`AtildeCurrentj`	`AtildeCurrent` for a particular subregion.
`AtildeFinerLevel`	Cell array containing subset of `AtildePrevious` for one level finer.
`AtildePrevious`	Cell array containing the $\tilde{\mathbf{A}}^{k,l}_{j_i,\ldots,j_m}$'s. See [2] Equation 10.
`Btilde`	Cell array containing the $\tilde{B}^{k,l}_{j_i,\ldots,j_m}$'s. See [2] Equation 13.
`Btildej`	`Btilde` for a particular subregion.
`calculationMode`	String set to either `likelihood`, `optimize`, or `prediction`
`cumulativeRegions`	Vector of the cumulative sum for number of tiles up until a given each level.
`data`	Data matrix created for entries with observations.
`dataSource`	String indicating from which source to load the data. This string must have a corresponding case in the `switch` clause of `load_data.m`
`domainBoundaries`	Vector containing `xMin0`, `xMax0`, `yMin0`, and `yMax0`.
`dummyIndexVec`	Vector containing `index` values for current level.
`finestKnotLevel`	Finest knot level in `build_structure`.
`iLevel`	Counter variable in the `for`-loop over the levels.
`index`	Continuous index of each tile.
`indexAncestry`	Vector containing the continuous `index` of the hierarchical ancestry for a particular subregion.

Table 13: Codebase objects and their descriptions.

`indexBeginningThisLevel`	The beginning continuous index for the level.
`indexCurrent`	Continous `index` for the current tile within the inner most loop of `build_structure()`.
`indexParent`	Continuous `index` for the parent of a given tile.
`indexSmaller`	Beginning index for one resolution coarser.
`isPredicting`	Boolean to determine if MRA is predicting.
`jDummyIndex`	Counter variable in `for`-loop over `nRegionsAtThisLevel`. Used to extract `thisDummyIndex` from `dummyIndexVec`.
`jRegion`	Counter variable used in the `parfor`-loop used to call `create_prior`.
`jTile`	Counter variable in the `for`-loop over the tiles at each level.
`k`	Counter variable used in `for`-loop of `posterior_inference`.
`KcB`	Cell array containing the lower Cholesky factor of $\mathbf{K}_{j_1,\ldots,j_m}$ times $\mathbf{B}^{l}_{j_1,\ldots,j_M}$. See [2] Equation 6.
`KcBContainer`	Cell array used to preallocate space for and store `KcB` for all levels. Construct used for parallelization.
`KcholA`	Cell array containing lower Cholesky factor of $\tilde{\mathbf{K}}_{j_1,\ldots,j_m}$ times $\mathbf{A}^{k,m}_{j_1,\ldots,j_m}$. See [2] Equation 10.
`Kcholw`	Cell array containing lower Cholesky factor of $\tilde{\mathbf{K}}_{j_1,\ldots,j_m}$ times $\omega^{m}_{j_1,\ldots,j_m}$. See [2] Equation 10.
`Kcwj`	$\tilde{\mathbf{K}}^{m}_{j_1,\ldots,j_m}\omega^{m}_{j_1,\ldots,j_m}$ for a particular subregions. Precomputed in `posterior_inference` for later solves.
`KcAj`	Cell array similar to `KcholA`, but for a particular level.
`knots`	Cell array containing all the knots for all levels.
`knotsX`	Vector of the x locations for the knots for a given partition.
`knotsY`	Vector of the y locations for the knots for a given partition.
`kPartition`	Counter variable over the number of partitions across each tile in `build_strcuture`.
`levelParent`	Level for the parent of a particular subregion.

Table 14: Codebase objects and their descriptions.

`logicalInd`	Vector that contains 1 where the observation is a number and 0 where the observation is `NaN`. A particular method to select only the entries from `lat`, `lon`, and `obs` that have have data. May not be appropriate for other datasets.
`logLikelihood`	Cell array containing the log-likelihoods for each subregion.
`logLikelihoodj`	`logLikelihood` for a particular subregion.
`mLevels`	Vector from 0 to $M-1$
`nRegionsAtThisLevel`	Number of regions at a given level.
`nRegions`	Vector of the number of regions at each level.
`nTilesFinestLevel`	Number of tiles at the finest level.
`numVarArgs`	Number of optional input arguments.
`NUM_KNOTS_r`	Number of knots in each partition for levels $m = 1, ..., M-1$.
`NUM_LEVELS_M`	Maximum number of levels at which the MRA partitions and sub-partitions the domain.
`NUM_PARTITIONS_J`	Number of partitions for each region.
`nKnotsX`	Number of knots in x-direction.
`nKnotsY`	Number of knots in y-direction.
`nXGrid`	Number of points in the x-direction for the prediction grid.
`nYGrid`	Number of points in the y-direction for the prediction grid.
`nKnotsX0`	Number of knots in x-direction at coarsest resolution.
`nKnotsY0`	Number of knots in y-direction at coarsest resolution.
`offsetPercentage`	How much (in percentage) to offset the structure from the edge of each region as to avoid edge-effects and overlap of observations between partitions.
`outputData`	Cell array of the data contained within a given region.
`partitions`	Cell array of the partitions (x and y locations) of the domain for each tile.
`plotting`	Boolean. Set to 1 to produce plot, 0 to not produce plots. Plots can only be produced by running the `prediction` routine.

Table 15: Codebase objects and their descriptions.

`posteriorPredictionMean`	Cell array containing posterior means.
`posteriorPredictionVariance`	Cell array containing the posterior variances.
`posteriorMeanj`	`posteriorPredictionMean` for a particular subregion.
`posteriorVariancej`	`posteriorPredictionVariance` for a particular subregion.
`predictionIndex`	Index within `predictionVector` of the prediction locations within a given tile.
`predictionLocations`	Vector containing the prediction locations.
`predictionLocationsj`	Predictions locations for given tile.
`predictions`	Cell array containing `posteriorPredictionMean` and `posteriorPredictionVariance`.
`predictionVector`	Vector containing `xPred` and `yPred`
`regressionModel`	Regression model using `fitlm()` fit to detrend the data.
`residuals`	Residuals found in the regression model. Stored as third column in `data`.
`RposteriorChol`	Cell array containing the lower Cholesky factor of the posterior covariance matrices.
`RposteriorCholb`	`RposteriorChol` for a particular subregion.
`RpriorChol`	Cell array containing the lower Cholesky factor of the prior covariance matrices.
`RpriorCholj`	`RpriorChol` for a particular subregion.
`RpriorCholContainer`	Cell array used to preallocate space for and store `RpriorChol` for all levels. Construct used for parallelization.
`temp`	Temporary variable used to calculate `w{iLevel}`.
`theta`	Vector of parameters for the `evaluate_covariance` function.
`thisAtj`	`Atj` for a particular subregion.
`thisDummyIndex`	Dummy continuous index for a given tile.

Table 16: Codebase objects and their descriptions.

`thisIndexAncestry`	Vector containing the ancestry for a given index. Output of `find_ancestry`.
`thisLevelKcBContainer`	Cell array containing subset of `KcBContainer` needed for this `iLevel`.
`thisLevelKnotsContainer`	Cell array containing subset of `knotsContainer` needed for this `iLevel`.
`thisKcholBchol`	`KcB` for a particular subregion.
`thisRetLikPred`	`retLikelihoodPred` for a particular tile.
`thisRpriorChol`	`RpriorChol` entry for a particular subregion.
`thisLevelRpriorCholContainer`	Cell array containing subset of `RpriorCholContainer` needed for this `iLevel`.
`thiswtj`	`wtj` for a particular subregion.
`varpEps`	Measurement error variance. Nugget term added to the diagonal of `RpriorCholj` in `create_prior.m`
`w`	Cell array of the $\omega^m_{j_1,\ldots,j_m}$'s for a given level.
`wtildeCurrent`	Cell array containing current $\omega^m_{j_1,\ldots,j_m}$ for a particular subregion.
`wtildeCurrentj`	`wtildCurrent` for a particular subregion.
`wtildePrevious`	Cell array containing the $\tilde{\omega}^{k,l}_{j_i,\ldots,j_m}$'s. See [2] Equation 10.
`xMax`	Maximal x location for a given tile.
`xMaxTemp`	Temporary variable for the maximum$x$ location for a given tile in `build_structure`.
`xMax0`	Maximum x value in the loaded data / maximum longitude of the spatial domain.
`xMin`	Minimal x location for a given tile.
`xMinTemp`	Temporary variable for the minimum x location for a given tile in `build_structure`.
`xMin0`	Minimum x value in the loaded data / minimum longitude of the spatial domain.
`yMax`	Maximal y location for a given tile.

Table 17: Codebase objects and their descriptions.

`yMaxTemp`	Temporary variable for the maximum y location for a given tile in `build_structure`.
`yMax0`	Maximum y value in the loaded data / maximum latitude of the spatial domain.
`yMin`	Minimal y location for a given tile.
`yMinTemp`	Temporary variable for the minimum y location for a given tile in `build_structure`.
`yMin0`	Minimum y value in the loaded data / minimum latitude of the spatial domain.

Acknowledgements

We would like thank Brian Vanderwende for HPC assistance throughout this project. Additionally, we would like to thank Doug Nychka and Matthias Katzfuss for informative feedback during the revision process.

References

[1] M. J. Heaton, A. Datta, A. Finley, R. Furrer, R. Guhaniyogi, F. Gerber, R. B. Gramacy, D. Hammerling, M. Katzfuss, F. Lindgren, D. W. Nychka, F. Sun, and A. Zammit-Mangion, "A Case Study Competition Among Methods for Analyzing Large Spatial Data," *ArXiv e-prints*, Oct. 2018.

[2] M. Katzfuss, "A multi-resolution approximation for massive spatial datasets," *Journal of the American Statistical Association*, vol. 117, no. 517, pp. 201–214, 2017. [Online]. Available: https://www.tandfonline.com/doi/full/10.1080/01621459.2015.1123632