International Journal of Data Science and Analytics
https://doi.org/10.1007/s41060-020-00215-3

REGULAR PAPER

A fast scalable distributed kriging algorithm using Spark framework

Chandan Misra[1,2] · **Sourangshu Bhattacharya**[3] · **Soumya K. Ghosh**[3]

Received: 11 March 2019 / Accepted: 15 March 2020

Abstract
Environmental and climate models used for weather prediction require evenly spaced meteorological datasets at a very high spatial and temporal resolution to facilitate the analysis of recent climatic changes. However, due to the small number of weather stations available, often the data collected from them are scattered and inadequate for such model creation. For this reason, very high-resolution gridded meteorological surface is developed by interpolating the available scattered data points to fulfill the need of various ecological and climatic applications. Among various interpolation techniques, Ordinary Kriging (OK) is one of the most popular and widely used gridding methodologies with a sound statistical basis providing a possibility to obtain highly accurate results. However, OK interpolation on large unevenly spaced data points is computationally demanding and has a computational cost that scales as the cube of the number of data points as it involves multiplication and inversion of matrices of large cardinalities infeasible for computation on a single node. Additionally, its standard implementation involves complex model fitting and function minimization steps which make automatic kriging analysis from raw data a considerable challenge. Meanwhile, Apache Spark has emerged as a large-scale data processing engine with a dedicated Machine Learning Library (MLLib) for processing large matrices and thereby can be used for large-scale kriging analysis with considerable time. In this paper, we present a new fast distributed OK algorithm on Apache Spark framework and provide an efficient and simple distributed matrix inversion scheme to accelerate the execution of distributed OK algorithm. We have employed Strassen's direct method for matrix inversion and the acceleration is achieved by exploiting the symmetry nature of the variance–covariance matrix of the OK equation to invert the matrix. We show experimentally that our distributed inversion scheme enables us to invert a 16,000 $\times$ 16,000 matrix with 51% and 38% less wall clock time than distributed Spark-based LU and Strassen's inversion scheme, respectively.

Keywords Linear algebra · Matrix multiplication · Strassen's algorithm · Apache Spark

1 Introduction

1.1 Motivation

Numerical weather prediction requires scalable numerical interpolation techniques that act on very large climatic datasets, containing points with different attributes (e.g., elevation, maximum and minimum land surface temperatures, precipitation, humidity, and other) at very high spatial and temporal resolution [1,3] to analyze the trend in climatic changes. However, the climatic model creation is hindered as the weather stations, and hence the observations from them are not arranged in a grid. As a way out, very high-resolution gridded meteorological datasets (or surfaces) are created using interpolation methods to fulfill the need of various ecological and climatic applications [1]. These gridded datasets are then archived online [23,25], and with the help of data processing and delivery system applied scientists can

Electronic supplementary material The online version of this article (https://doi.org/10.1007/s41060-020-00215-3) contains supplementary material, which is available to authorized users.

✉ Chandan Misra
chandan.misra@iitkgp.ac.in

Sourangshu Bhattacharya
sourangshu@cse.iitkgp.ernet.in

Soumya K. Ghosh
skg@cse.iitkgp.ac.in

[1] Advanced Technology Development Centre, Indian Institute of Technology Kharagpur, Kharagpur, India

[2] School of Computer Science and Engineering, Xavier University Bhubaneswar, Bhubaneswar, India

[3] Department of Computer Science and Engineering, Indian Institute of Technology Kharagpur, Kharagpur, India

Published online: 03 April 2020

have on-demand access to these datasets. There is an increasing demand for a high-resolution gridded dataset [55] for several reasons. It is used as a validation tool in interpreting the impact of global warming on local environments [56]. It also validates the simulation products of numerical models and satellite-based high-resolution precipitation products [52].

Another reason for large datasets being handled by spatial interpolation algorithms is the increase in the size of the region of interest. For some studies, e.g., those at country and continent levels [10,13,15,17,19,23,36,40,55,56], smaller region cannot fulfill the requirement for the verification of climate model simulation. In 2008, the Climate Prediction Center (CPC)[1] released analysis product [4] covering the entire globe at a daily interval. The daily weather forecast is also expected to be improved using high spatial resolution [27]. Other sources of large geospatial and remote sensing data include *Light Detection and Ranging* point cloud data (LiDAR), raster images from satellites like *LandSat, Moderate Resolution Imaging Spectroradiometer* (MODIS), etc. As a consequence, traditional algorithms for spatial interpolation techniques, e.g., kriging, which requires linear algebraic operations like matrix multiplications and inversions for model creation, cannot be executed on a single node with limited memory and large execution time. Hence, there is a need for the scalable algorithms and implementation of these interpolation techniques as well as matrix algorithms in a distributed setting for easy and faster execution of the interpolation methods.

1.2 Methods and challenges

Many spatiotemporal interpolation methods have been used in the literature, e.g., Shepard's method or inverse distance weighting (IDW) [44] has been used in [10,38–40,56], and geostatistical methods like kriging [7] have been used in [17,23]. Also, machine learning approaches like linear and nonlinear regression [19], three-dimensional thin-plate smoothing splines (TPS) [15,17,19,23], etc., have been used. In this paper, we focus on the *Ordinary Kriging* (OK) interpolation technique [7,20,34,49], which is a widely used geostatistical approach in the case of data that are spatially or temporally correlated. The choice of OK is motivated by the fact that it is the best linear unbiased estimator (BLUE), i.e., the linear combinations of available data with minimum prediction variance or mean squared prediction error (MSPE).

Ordinary Kriging interpolant is computationally expensive since computing the value of a single prediction point involves all the interpolating points and solving a dense linear system of order $(n \times n)$ which takes $O(n^3)$ where n refers to the number of scattered data points. To overcome this problem, local approaches, called the moving neighborhood, [20,53], have been proposed by approximating kriging equation considering locally a relatively small number of points that are closest to the prediction point. Parallel versions of the method have also been studied extensively in [28,42,43,45,50]. However, there are many problems associated with the local approach as pointed out in [31]. Firstly, the selection of neighborhood size is not straightforward and is performed either by selecting a fixed number of closest points or by considering all the points inside a search radius centering the prediction point. Such neighborhood size may not produce good approximation for all the query points because it depends on the local density of the point distribution. Secondly, local methods behave discontinuously at interface boundaries where the neighborhood changes [32]. Hence, in this paper, we study the problem of parallelizing global neighborhood method for Ordinary Kriging and not consider local approaches.

The computational complexity of the global kriging algorithm is dominated by the calculation of OK weight matrix which is done by two costly matrix operations—matrix inversion of size $(n \times n)$ and matrix multiplication of sizes $(n \times n) \times (n \times m)$, where n is equal to the number of interpolating points and m is equal to the number of prediction points. HPC frameworks like OpenMP [6], MPI [18,37], and GPUs enabled with CUDA [5] have been suggested for implementing OK on large datasets. However, HPC-based systems suffer from disadvantages like vertical scaling, expensive, and non-fault-tolerant and GPUs are a non-scalable centralized system which requires special hardware. Alternatively, large-scale distributed data processing framework like Hadoop MapReduce [14] and Spark [47] along with distributed file system, called HDFS, overcomes almost all the above shortcomings by its distributed storage, data locality, replication, and the use of cheaper commodity servers. As a result, they have emerged as the next-generation parallel dataflow programming platform for data-intensive complex analytics and building parallel applications. Further, Spark has gained its popularity for its in-memory data processing ability to run programs faster (100 times for certain computations) than Hadoop MapReduce. Its general-purpose engine supports a wide range of applications including batch, interactive, iterative algorithm, and streaming, and it offers simple APIs and rich built-in libraries like MLLib and GraphX for data science tasks and data processing applications. As a consequence, Spark is being used for data-intensive as well as computational intensive tasks. In view of these advantages, we use Spark as our framework of choice in this paper.

LU decomposition is the most widely used technique for distributed matrix inversion, possibly due to its efficient block recursive structure. Xiang et al. [54] proposed a Hadoop-based implementation of inverting a matrix relying on computing the LU decomposition and discussed many

[1] https://www.cpc.ncep.noaa.gov/.

Hadoop-specific optimizations. Recently, Liu et al. [30] proposed several optimized block recursive inversion algorithms on Spark based on LU decomposition. In the block recursive approach [30], the computation is broken down into sub-tasks that are computed as a pipeline of Spark tasks on a cluster. The costliest part of the computation is the matrix multiplication, and the authors have given a couple of optimized algorithms to reduce the number of multiplications. However, in spite of being optimized, the implementation requires 9 $O(n^3)$ operations on the leaf node of the recursion tree, 12 multiplications at each recursion level of LU decomposition, and 7 additional multiplications after the LU decomposition to invert the matrix, which makes the implementation perform slower. Misra et al. [33] implemented a distributed approach, called SPIN, using Strassen's matrix inversion procedure in Spark to reduce the number of multiplications to 6 which is arguably the fastest distributed inversion to date. We have noticed that the variance–covariance matrix which needs to be inverted is a symmetric positive definite in nature and thus the property can be utilized in reducing the execution time further.

In this study, we implemented the distributed OK interpolation algorithm on Spark and provided an efficient yet simple distributed matrix inversion scheme which gave a solution toward accelerating OK interpolation for the large volume of datasets. Although parallel processing approaches have been incorporated as a solution for speeding up the OK interpolation method for large datasets, no in-depth implementation details have yet been provided in a distributed environment like Spark. In this paper, we have developed a block recursive distributed matrix inversion algorithm in Spark which exploits the symmetry nature of the matrix to reduce the number of multiplications to 4 at each recursion level of the computation tree compared to 6 multiplication of its non-symmetric approach. We show experimentally that our approach outperforms other state-of-the-art distributed inversion techniques significantly, thus reducing the overall OK running time.

1.3 Organization of the article

After presenting a detailed related work in Sect. 2, we provide a brief on the OK method in Sect. 3. A thorough description of our distributed OK implementation is given in Sect. 4. We introduce our distributed block recursive symmetric matrix inversion in Sect. 4.6 and provide a detailed description of the algorithm. In Sect. 6, we evaluate the performance of our inversion approach along with two other competing approaches—LU-based block recursive approach and Strassen's block recursive approach for general matrices through experimental results, and Sect. 7 summarizes the results.

2 Related works

All the prior works that have been carried out for parallelizing OK are based on, as mentioned earlier, two primary directions—the local neighborhood approach [11,42,43,45,50,51] that considers only a small subset of interpolation points surrounding a prediction point and the global one [21,22,25,26,48,58] which considers all the interpolating points for making prediction. Local approaches have been developed and implemented in various parallel and distributed environments. For example, Strzelczyk et al. in [50,51] have developed a new parallel kriging algorithm and tested for 5000–15000 points taken from satellite *Permanent Scatterer Interferometry Synthetic Aperture Radar* (PSInSAR) data used for deriving information about the velocity of slow ground deformation. They overcome the shortcomings of prior works [11], designed to deal with evenly spaced data, to parallelize kriging on a scattered dataset in a multi-CPU environment. To make sure each CPU gets a near-equal number of points and even amount of computation, the approach groups the known interpolating points into ten classes in terms of the same number of neighbors. The candidates for each class are then scattered over all the CPUs and processed in parallel. To reduce the computation time further, they skipped the interpolation process of some points which have near similar neighborhood and kriged value.

Shi et al. [45] showed that the kriging computation of the local approach if implemented in heterogeneous architecture (cluster of nodes with multiple CPUs and GPUs) can be accelerated to a great extent. They identified the parts of the algorithm which need parallelization and are implemented using CUDA and parts which must be kept sequential with C implementation and executed them in GPU and CPU, respectively. The parallelization is done over the number of prediction points where each prediction point is assigned to a single CUDA core processed in parallel for a certain number of neighboring points for each prediction point.

Permata et al. [42] used the concept of Spark's resilient distributed dataset (RDD) to process LiDAR data which enables DEM generation quickly and efficiently. They implemented kriging in Spark and compared it in terms of processing time, CPU, and cluster memory utilization with the Hadoop-based approach and showed that their method was 4.8 times faster than Hadoop-based kriging implementation. Further, in [43], the authors implemented local OK interpolation algorithm on three platforms (MPI, MapReduce, and GPU) and observed the performance of each method. They showed that under the different environment (no. of LiDAR points, no. of grid points) GPGPU is the recommended method for processing kriging interpolation as it outperformed others if maximum *Degree of Parallelism* (DoP) is provided to each of them. However, they also pointed out that the usability and reliability of MapReduce platform are much greater than MPI

and GPU, as it provides easier and simpler parallelism to regular non-computer science domain researchers. Although the above works have demonstrated that GPU can considerably reduce the computational time of OK implementation, they are non-scalable centralized methods and need special hardware.

Global approaches like [25,26] were one of the preliminary works to parallelize the kriging method. They discussed parallel kriging technique for converting a large scattered set of rainfall observations to grid form to prepare derived data products like rainfall surfaces. The kriging algorithm was implemented in High-Performance Fortran which exploits parallel computation through the data parallel programming model using matrix inversion and computing resources like CM5 (Connection Machine) and Farm of Alpha Workstations. As this work was one of the early works to implement parallel kriging, the number of known values (100–500 random points) and size of target grid surface (50×50 to 700×700) are quite smaller than the size of the large dataset being processed nowadays. The central objective of the work was to provide a framework for spatial data storage interpolation and delivery system, and thus a parallel algorithm for kriging and matrix inversion was not well documented.

Similarly, Jardak et al. [22] discussed the suitability of the MapReduce framework for analyzing a large amount of sensor data. They have given a case study of large-scale kriging analysis in MapReduce for their model which is a complete framework for processing sensor data. It includes implementation of modeling semivariogram, indexing network nodes to reduce spatial search time, and kriging equation solving and prediction of unknown attribute values of points to be interpolated. The prediction is done by using one map and one reduce step. The map step is used for creating the variance–covariance matrices, and the reduce step is for making the predictions. Further, in [21], they dealt with performing prediction on large amounts of measurements that are associated with particular locations of the wireless sensors. They give three application scenarios, computational challenges with spatiotemporal big data. By doing that, they give a scalable solution for spatiotemporal estimation algorithms or kriging in cloud-based infrastructure or using Hadoop MapReduce infrastructure. However, no details on parallelizing kriging equation solving have been presented in their work.

Srinivasan et al. [48] reduced the OK computational cost by formulating the prediction problem in a different way. They observed that OK equation solving has a computational complexity of $O(mn^3)$ or $O(n^4)$ when $m \approx n$, where n is the number of interpolating points and m is the number of prediction points. They viewed the problem as a two-step process—(1) solving a simultaneous linear equation with a computational complexity of $O(n^3)$ followed by (2) matrix–vector product with a computational complexity of $O(n^2)$, making the overall complexity as $O(n^3+n^2)$. However, they emphasized the acceleration of matrix–vector product over GPUs rather than reducing the computation cost of the equation solver.

Another R-based MPI and Math Kernel Library (MKL)-driven parallel implementation has been presented in [58] using multiple nodes and multiple core environment. They applied it to the precipitation anomaly dataset and synthetic dataset to prove better accuracy than IDW and weak scaling, respectively. They used Cholesky decomposition and forward solving strategy for linear equation solving. There are other studies (for example [16] and [6]) which solve the kriging equation using Gauss–Jordan elimination and LU decomposition. These methods are standard methods for solving a simultaneous linear equation and can achieve significant performance gain when implemented in parallel architectures. However, some part or the entire algorithm of these methods is intrinsically sequential in nature. In other words, each step of the algorithm depends on the previous step of the algorithm, making it almost impossible to divide the input matrix and distribute it in Spark. Additionally, if we utilize Spark's iterative solution to process RDD in a loop, it fails as the lineage of the computation becomes memory intensive and requires breaking the chain momentarily and thus we lose the fault tolerance. Also, each such loop, one communication or shuffle phase is required, which negates the use of in-memory computation.

Authors in [41,57] have presented large-scale kriging processing as part of the cloud-based spatial data analysis and visualization system and scalable storage and processing framework for pervasive computing, respectively. However, these works provided a spatial data storage and processing system and thus particular parallel processing scheme for kriging was not presented.

3 Kriging formulation

Kriging, named after *D. G. Krige*, is one of the fundamental spatial prediction tools in geo-statistics and found applications in many fields like mining, environmental sciences, hydrogeology, remote sensing, etc. It is defined as a method for interpolating the value of a random field at an unobserved location based on available surrounding measurements. In this paper, we have implemented Ordinary Kriging (OK), which is also termed as BLUE or best linear unbiased estimator. It is named like that because it uses linear combinations of available data, tries to have the mean residual or error m_R equal to 0, and tries to minimize the prediction variance or mean squared prediction error (MSPE) σ_R^2.

In OK, we use probability model which calculates the residual means and prediction variance and then chooses weights of the nearby sample points so that mean is zero and variance is minimized. In the probabilistic approach,

the model is a stationary random function which consists of several random variables at each of the interpolating points, i.e., $V(x_1), \ldots, V(x_n)$, and the point to be interpolated, i.e., $V(x_0)$. Kriging estimate, which is a random variable at every value in the model, is the outcome of a random variable, and the estimate is also a random variable since it is a weighted linear combination of the random variables at the interpolating points. The value of the estimate is

$$\hat{V}(x_0) = \sum_{i=1}^{n} w_i . V(x_i) \tag{1}$$

which minimizes the variance of the modeled error

$$\text{Var}\{R(x_0)\} = \text{Var}\{\hat{V}(x_0) - V(x_0)\} \tag{2}$$

instead of minimizing the variance of actual error σ_R^2 using a semivariogram model

$$\gamma(h) = \frac{1}{2} E[(Vx_i) - V(x_j))^2], \quad \forall i, j. \tag{3}$$

This gives the OK equation

$$\tilde{\gamma}_{i0} = \sum_{j=1}^{n} w_j \tilde{\gamma}_{ij} - \mu \tag{4}$$

in terms of semivariogram, where $\tilde{\gamma}_{ij}$ is the semivariance between samples at location i and j, respectively, w_j is the kriging weighted coefficient of point j, and μ is the Lagrange parameter. We can also express the kriging equation in terms of covariogram as

$$\tilde{C}_{i0} = \sum_{j=1}^{n} w_j \tilde{C}_{ij} - \mu. \tag{5}$$

The system can be written as matrix notation as

$$\underbrace{\overbrace{\begin{bmatrix} \hat{C}_{11} & \cdots & \hat{C}_{1n} & 1 \\ \vdots & \ddots & \vdots & \vdots \\ \hat{C}_{n1} & \cdots & \hat{C}_{nn} & 1 \\ 1 & \cdots & 1 & 0 \end{bmatrix}}^{C}}_{(n+1)\times(n+1)} \cdot \underbrace{\overbrace{\begin{bmatrix} w_1 \\ \vdots \\ w_n \\ \mu \end{bmatrix}}^{w}}_{(n+1)\times 1} = \underbrace{\overbrace{\begin{bmatrix} \hat{C}_{10} \\ \vdots \\ \hat{C}_{n0} \\ 1 \end{bmatrix}}^{D}}_{(n+1)\times 1} \tag{6}$$

$\tilde{C}_{ij}$ is the covariance between ith and jth random variable, and μ is the Lagrange parameter. Therefore, the kriging weights w_i can be solved as

$$w = C^{-1} \cdot D \tag{7}$$

by choosing the above $(n+1)^2$ covariances, calculated from a chosen function $\hat{C}(h)$. The covariances and semivariances can be obtained using this carefully chosen function, also termed as population covariogram and population variogram, respectively. These functions are theoretical models that are fitted through empirical or sample semivariogram or sample covariogram. Commonly used semivariogram models used to define all sorts of sample semivariogram or covariogram patterns and are spherical, exponential, Gaussian, linear, power, and wave or hole effect model. The model parameters are estimated to fit one of the stated variogram models. For this work, we use the spherical covariance model which is defined as

$$\gamma(h) = \begin{cases} a + \sigma^2 \left[\frac{3h}{2r} - \left(\frac{h}{2r}\right)^3\right], & 0 < h \leqslant r \\ a + \sigma^2, & h > r \end{cases} \tag{8}$$

having model parameters

$$a = \text{nugget effect}, \sigma^2 = \text{sill and } r = \text{range}. \tag{9}$$

Based on the formulation given above, an OK sequential algorithm can be implemented by using the following six stages:

1. Calculation of empirical semivariogram
2. Fitting empirical semivariogram to one of the theoretical semivariogram models ($\tilde{\gamma}(h, \theta)$) for extracting the model parameters (a =nugget effect, σ^2 =sill and r =range).
3. Construction of the covariance matrix C.
4. Construction of the D vector.
5. Calculation of the kriging weight vector (w). It requires a matrix inversion of matrix C and a matrix–vector multiplication of matrix C^{-1} and vector D as shown in Eq. 7.
6. Getting the interpolated value ($\tilde{v}$) at the point with unknown values using true sample values ($v_1 \ldots v_k$, where k is the number of samples) and kriging weights.

Note that, in case of several points to be interpolated, D will be a matrix instead of a vector. In that case, kriging weight calculation involves matrix–matrix multiplication of matrix C^{-1} and matrix D.

4 Distributed kriging in Spark

In this section, we discuss the implementation details of each of the stages mentioned in the last section and emphasize on a distributed matrix inversion scheme that exploits the symmetry of the covariance matrix (C) to reduce the overall running time of the OK process. We provide a brief on the state of the art in this regard and try to furnish the performance analysis of our approach for establishing its superiority. Later,

in Sect. 6, we conduct experiments to prove that our approach indeed outperforms the baseline approaches conforming our analysis.

4.1 Data storage

Before describing the individual stages, let us first consider the underlying data structures which are used for storing and processing datasets as well as for matrix computation. The fundamental distributed data structure used in Spark implementation is the resilient distributed datasets or RDDs. RDDs are a collection of data items that are split into partitions stored on Hadoop distributed file system (HDFS). RDD contains key–value pairs and can be transformed into another RDD by operations called transformation. RDDs can also distribute basic data structures like matrix and vectors to be used for matrix and/or vector operations distributedly. We use two flavors of distributed matrix data structure: *IndexedRowMatrix* and *BlockMatrix*.

4.1.1 Indexed row matrix

Indexed row matrix is a wrapper over a RDD of *IndexedRows* spread in the cluster. *IndexedRow* is a row of a matrix with an attached row index represented as a tuple *(rowIndex, Vector)*. *Vector* refers to a local vector which represents the row of the matrix. This data structure is necessary to convert an *RDD* to a *BlockMatrix* to do matrix operations.

4.1.2 Block matrix

Block matrix is basically an *RDD* of *MatrixBlocks* spread in the cluster. Distributing the matrix as a collection of blocks makes them easy to be processed in parallel and follow divide-and-conquer approach. *MatrixBlock* is a block of matrix represented as a tuple *((rowIndex, columnIndex), Matrix)*. Here, *rowIndex* and *columnIndex* are the row and column indices of a block of the matrix. *Matrix* refers to a one-dimensional array representing the elements of the matrix arranged in a column major fashion.

4.2 Data preprocessing

In the data preprocessing stage, we convert the input raw image into an RDD so that it can be consumed in the proceeding job pipeline. We take the raw image and convert it into a corresponding NDVI image using QGIS software and export it as a comma-separated value file. Each line of the csv file consists of the coordinates of a point and the NDVI value associated with it. We then store the csv file in the HDFS and process it using a Spark job which converts the input file into an RDD. The spark job consists of one map transformation which maps each line of the file onto a record, and a collection of records thus produced generates the RDD of records.

4.3 Empirical semivariogram calculation

Sample or empirical semivariogram ($\gamma(h)$) calculation is used to examine the spatial correlation in data at various distances of separation. The h parameter describes the orientation separating pairs of points because spatial correlation is characterized by both distance and direction. However, for the sake of simplicity, we implement the empirical semivariogram calculation through the use of omnidirectional variogram. This means that calculation will be carried out as a function of distance ignoring any directional differences.

According to [2] if the input data are more than 5000 and data are scattered, it is not possible to calculate and visualize the semivariogram due to the computing time and memory limits of the modern computers, because in that scenario we will get more than 12 million pairs of points with different distances. Though this task can be done in a distributed setting, it is often, however, difficult to visualize any pattern because there are so many pairwise lags [2,12]. To solve the problem geostatistical, analysts select 5000 observations randomly for semivariogram model fitting. Alternatively, another method, known as *binning* method, is employed to reduce the number points by grouping pairs of locations based on their distance from one another. This method is incorporated in software packages like *GSLIB* [9], *Splus* [24], *SAS* [46], *ArcGIS* [2], and *GSTAT* R package [35] and therefore has been incorporated in our implementation as well.

Algorithm 1: Empirical Semivariogram Calculation

1 **Procedure** EMPSEM(*RDD inputRDD, double cuoff*)
  Result: RDD $\langle Block \rangle$ resultRDD
2 Generate **trainingIndexed** by attaching an unique id to every record of the **inputRDD**
3 Generate all pairs of points **cartesianProd**
4 Calculate **trimMap** to obtain $dist_{ij}$ and $(v_i - v_j)^2 \; \forall i, j$
5 Filter records where $i < j$ and $dist_{ij} >$ cutoff
6 Calculate the number of points **counts** in each bin and broadcast it to all the nodes
7 Generate key–value pairs where key=**bin** and value is a tuple having value $dist_{ij}, (v_i - v_j)^2$
8 Group the records by **bin** and add all the values to calculate $sum = \sum dist_{ij}, \sum (v_i - v_j)^2$ for individual bins
9 Generate **semVarRDD** by dividing $\sum dist_{ij}$ and $\sum (v_i - v_j)^2$ by **counts** for individual bins
10 **return** semVarRDD

EMPSEM procedure, as presented in Algorithm 1 (detail algorithm is given as Algorithm 1 in the supplemental material) implements the binning approach in a distributed way. It takes an RDD which contains the input dataset and makes

a Cartesian product to itself to obtain records consisting of every point with every other point. The resulting RDD contains n^2 records where n is equal to the number of records in the sample. Then, we calculate the distance between each pair of points using a *mapToPair* transformation (line 4). We also calculate the expression $(val_j - val_i)^2$ and the bin in which each pair of points falls. However, this emits duplicate records for the same pair of points, i.e., one for (i, j) and the other for (j, i) for all i, j. To overcome this, we employ a filter function (line 5) which checks if $i < j$ and discards any record that violates it. In this way, the number of rows in the file becomes $n(n+1)/2$. The filtering step also excludes any record that contains the distance greater than the cutoff distance.

The number of points for each bin is obtained using *countByKey* transformation (line 6) which counts the number of records for each key. We maintain a hashmap in the memory to accommodate these bin–counts pairs and broadcast the hashmap to be processed in a distributed manner. We then group the records based on bin index which groups all the values $dist_{ij}$ and $< (val_i - val_j)^2 >$ related to a particular bin and sums them together. The last *mapToPair* transformation (line 9) divides the sum of distances and semivariance by the number of points in that particular bin and thereby provides the distance and semivariance for each bin.

4.4 Covariances among points in the sample

To calculate the kriging weight matrix (w), according to Eq. 7, we build the covariance matrix using one of the semivariogram model equations. (For our case, see Eq. 8.) The covariance matrix computation procedure COV in Algorithm 2 (detailed algorithm is given as Algorithm 2 in the supplemental material) uses the spherical model to build the covariance matrix and returns a *BlockMatrix* which later can be inverted using our inversion algorithm. To calculate the covariances of all pairs of points, we need the distances of all pairs of points and thus we use *trimMap RDD* (see line 4 in Algorithm 1) to get all the distances and maps each such distance onto covariance using spherical model equation.

Note that the size of the covariance matrix is $m+1$, where m is the number of interpolating points, and the entries are shown in Eq. 6. Therefore, in *rowRDD*, we have m^2 entries instead of $(m+1)^2$ and thus to include last column and row vectors, we group each row with row indices and do a *flatMap* transformation which adds the last row and column to the RDD. All the rows are transformed into *rows* which is a *RDD* of *IndexedRow*. To create the covariance block matrix, we transform *rows* into *CRowMatrix* of type *IndexedRowMatrix*, a wrapper over an *RDD* of *IndexedRow*. At last, the *CRowMatrix* is converted into block matrix C and returned.

Algorithm 2: Covariance Matrix Calculation

1 **Procedure** COV (*RDD trimMap, sill, range*)
  Result: BlockMatrix C
2 Calculate **covarianceRDD** to obtain the covariance $C_{ij}\ \forall i, j$
3 Generate $< key, value >$ pairs **rowRDD** where key=i and value is a tuple of (j, C_{ij})
4 Group the records based on key i or the row index
5 Generate <key,value> pairs **rows** where key=i and value is a vector of $C_{ij}\ \forall i, j$
6 Convert **rows** to **CRowMatrix** where each record is a **IndexedRow**
7 Convert **CRowMatrix** to **BlockMatrix** C
8 return C

4.5 Covariances between sample points and the prediction point

We calculate the covariances between the sample and prediction points much like the same way as we have done for C matrix described in the last section. The only difference is that we require all pairs of points between the sample points and the prediction points.

Algorithm 3: D Matrix Calculation

1 **Procedure** DMAT (*RDD indexedTest, RDD indexedTrain, sill, range*)
  Result: BlockMatrix D
2 Generate all pairs of points (i, j) where i is a interpolating point and j is a point to be interpolated.
3 Calculate **covarianceRDD** to obtain the covariance $D_{ij}\ \forall i, j$
4 Generate $< key, value >$ pairs **rowRDD** where key=i and value is a tuple of (j, D_{ij})
5 Group the records based on key i or the row index
6 Generate $< key, value >$ pairs **rows** where key=i and value is a vector of $D_{ij}\ \forall i, j$
7 Convert **rows** to **DRowMatrix** where each record is a **IndexedRow**
8 Convert **DRowMatrix** to **BlockMatrix** D
9 **return** D

To obtain that, we feed *DMAT* procedure (see Algorithm 3 and detailed algorithm is given as Algorithm 3 in the supplemental material) with a couple of RDDs: *indexedTest* and *indexedTrain* for prediction and sample data points, respectively, where each record is indexed with a unique and meaningful number (line number in this case). The *cartesian* transformation generates all pairs of points which then undergoes a *mapToPair* which calculates the covariance for each pair of points. Next, we group the records according to the row index of the D matrix and use the same *flatMap* transformation as in *COV* procedure with one difference. Instead of creating one additional row and column, we create only one additional row, i.e., an additional single entry for each column vector of D. The resulting RDD of *IndexedRow*

is converted into *IndexedRowMatrix* and to *BlockMatrix D* subsequently.

4.6 Symmetric matrix inversion

We have seen in [33] that the block recursive-based divide-and-conquer approach for matrix inversion can achieve substantial performance gain and outperforms LU-based divide-and-conquer approach in a considerable margin. It distributes sub-matrices across nodes and exploits each large matrix multiplication in a parallel fashion. In this section, we will look into the same block recursive approach in light of matrix which is symmetric and positive definite and try to reduce the running time by reducing the number of matrix multiplications at each recursion level. For reference, Strassen's method for matrix inversion of a square matrix A is given in Algorithm 4 in Sect. 4.6.1. Finally, Sect. 4.6.2 describes the distributed inversion algorithm and its implementation strategy using *Blockmatrix* for symmetric and positive definite matrix.

4.6.1 Strassen's algorithm for matrix inversion

Strassen's matrix inversion algorithm appeared in the same paper in which the well-known Strassen's matrix multiplication was published. The algorithm splits both the input matrix A and its inverse $C = A^{-1}$ into half-sized sub-matrices as

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}^{-1} = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} \tag{10}$$

and the result C can be calculated according to Algorithm 4, and then it computes each sub-matrix, $(C_{11}, \ldots, C_{22})$, in terms of input sub-matrices, $(A_{11}, \ldots, A_{22})$ using a series of matrix multiplications, additions, and scalar multiplications in a recursive way until it reaches a certain size, called *threshold*. Matrix inversions below the threshold can be performed using any inversion algorithm. Algorithm 4 is realized according to Eq. 11 after simplifying Eq. 10.

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} P & Q \\ R & S \end{bmatrix}$$
where $P = A_{11}^{-1} - A_{11}^{-1} A_{12} E^{-1} A_{21} A_{11}^{-1}$
$Q = A_{11}^{-1} A_{12} E^{-1}$
$R = E^{-1} A_{21} A_{11}^{-1}$
$S = -E^{-1}$ and
$E = A_{21} A_{11}^{-1} A_{12}$
$- A_{22}$ Schur Complement of A_{11}. (11)

Equation 10 and Algorithm 4 require that the matrix A is square. For odd-sized matrix $M \times N$, where $M = 2m + 1$ and $N = 2n + 1$ and m, n are even, A can be partitioned as equation 12.

$$A = \begin{bmatrix} A_{2m \times 2n} & A_{2m \times 1} \\ A_{1 \times 2n} & A_{1 \times 1} \end{bmatrix} \tag{12}$$

Algorithm 4: Strassen's Serial Inversion Algorithm

```
1  function Inverse();
   Input  : Matrix A (input matrix of size n × n),
            int threshold
   Output: Matrix C (invert of matrix A)
2  begin
3      if n=threshold then
4          invert A in any approach;
5      else
6          Compute A11, B11, ..., A22, B22 by computing n = n/2;
7          I ← Inverse(A11)
8          II ← A21.I
9          III ← I.A12
10         IV ← A21.III
11         V ← IV − A22
12         VI ← Inverse(V)
13         C12 ← III.VI
14         C21 ← VI.II
15         VII ← III.C21
16         C11 ← I − VII
17         C22 ← −VI
18     return C
```

4.6.2 Distributed block recursive matrix inversion algorithm

Algorithm 4 cannot be performed in parallel as multiple matrix operations cannot be done in parallel. However, if the matrices of consideration are large, then each such matrix computation step can be done in parallel or in a distributed manner. The distributed block recursive algorithm can be represented as a recursion tree as depicted in Fig. 1, where upper left sub-matrix is divided recursively until it can be inverted serially on a single machine. After the leaf node inversion, the inverted matrix is used to compute intermediate matrices, where each step is done distributively. Another recursive call

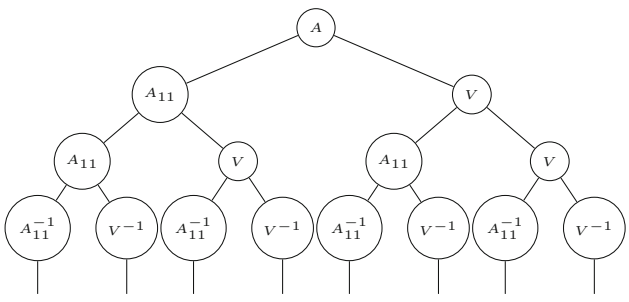

Fig. 1 Recursion tree for Algorithm 4

Algorithm 5: Strassen's Serial Inversion Algorithm for Symmetric matrices

```
function Symm-Inverse();
  Input  : Matrix A (input matrix of size n × n),
           int threshold
  Output: Matrix C (invert of matrix A)
begin
    if n=threshold then
        invert A in any approach
    else
        Compute A11, B11, ..., A22, B22 by computing n = n/2;
        I ← Symm-Inverse(A11)
        II ← A21.I
        III ← II^T
        IV ← A21.III
        V ← IV − A22
        VI ← Symm-Inverse(V)
        C12 ← III.VI
        C21 ← C12^T
        VII ← III.C21
        C11 ← I − VII
        C22 ← −VI
    return C
```

is performed for the matrix VI until the leaf node is reached. Like A_{11}, it is also inverted on a single node when the leaf node is reached. The computation of Algorithm 4 is dominated by the cost of 6 matrix multiplications. Therefore, the computation time can be reduced significantly if the number of matrix multiplications can be minimized. For a symmetric positive definite matrix A, the inversion of A is also symmetric and the splitting into sub-matrices can be done using Eq. 13.

$$\begin{bmatrix} A_{11} & A_{21}^T \\ A_{21} & A_{22} \end{bmatrix}^{-1} = \begin{bmatrix} C_{11} & C_{21}^T \\ C_{21} & C_{22} \end{bmatrix}. \tag{13}$$

Therefore, it is sufficient to calculate C_{21}, and therefore C_{12} can be obtained by just transposing C_{21}. Moreover, variable III can be calculated using matrix transpose operation substituting matrix multiplication because of the following equality:

$$\begin{aligned} II^T &= (A_{21}.I)^T = (A_{21}.A_{11}^{-1})^T \\ &= (A_{11}^{-1})^T.A_{21}^T = A_{11}^{-1}.A_{12} = III. \end{aligned} \tag{14}$$

Thus, matrix inversion for the symmetric matrix can be done using only 4 multiplications instead of 6 which is given in Algorithm 5.

The core inversion algorithm (described in Algorithm 5) takes a matrix (say A) represented as *BlockMatrix*, as input and the computation performed by the algorithm is based on six distributed methods, which are as follows:

- *breakMat* Breaks a matrix into four equal-sized sub-matrices
- *xy* Returns one of the four sub-matrices after the breaking, according to the index specified by x and y.
- *multiply* Multiplies two *BlockMatrix*
- *subtract* Subtracts two *BlockMatrix*
- *scalarMul* Multiples a scalar with a *BlockMatrix*
- *arrange* Arranges four equal quarter *BlockMatrices* into a single full *BlockMatrix*.

Below, we describe the methods in little bit more detail and also provide the algorithm for each.

breakMat method (see Algorithm 4 in supplemental material) breaks a matrix into four sub-matrices but does not return four sub-matrices to the caller. It just prepare the input matrix to a form which helps filtering each part easily. It takes a *BlockMatrix* and returns a *PairRDD* of *tag* and *Block* using a *mapToPair* transformation. First, the *BlockMatrix* is converted into an RDD of *MatrixBlocks*. Then, each *MatrixBlock* of the RDD is mapped onto tuple of *(tag, MatrixBlock)*, resulting in a *pairRDD* of such tuples. Inside the *mapToPair* transformation, we carefully tag each *MatrixBlock* according to which quadrant it belongs to.

xy method (see Algorithm 5 in supplemental material) is a generic method signature for four methods used for accessing one of the four sub-matrices of size 2^{n-1} from a matrix of size 2^n. Each method consists of two transformations—*filter* and *map*. *filter* takes the matrix as a *pairRDD* of *(tag, MatrixBlock)* tuple which was the output of *breakMat* method and filters the appropriate portion against the tag associated with the *MatrixBlock*. Then, it converts the *pairRDD* into *RDD* using the *map* transformation.

multiply method multiplies two input sub-matrices and returns another sub-matrix of *BlockMatrix* type. Multiply method in our algorithm uses a naive block matrix multiplication approach, which replicates the blocks of matrices and groups the blocks together to be multiplied in the same node. It uses co-group to reduce communication cost.

subtract method subtracts two *BlockMatrix* and returns the result as *BlockMatrix*.

scalarMul method (see Algorithm 6 in supplemental material) takes a *BlockMatrix* and returns another *BlockMatrix* using a *map* transformation. The *map* takes blocks one by one and multiplies each element of the block with the scalar.

arrange method (see Algorithm 7 in supplemental material) takes four sub-matrices of size 2^{n-1} which represents four coordinates of a full matrix of size 2^n and arranges them in later and returns it as *BlockMatrix*. It consists of four *map*s, each one for a separate *BlockMatrix*. Each *map* maps the block index to a different block index that provides the final position of the block in the result matrix.

4.7 Kriging weights calculation and interpolated value generation

The final step of OK implementation is to solve the OK equation to get the weight matrix and multiply the transpose of the weight matrix to the data vector as depicted in Algorithm 7. The variance–covariance matrix C is generally large and has size equal to $(m+1) \times (m+1)$, where m is the number of interpolating points. D matrix has a size of $(m+1) \times p$, where p is the number of prediction points. Data vector is the vector of attribute values of the known interpolating points which are transformed into BlockMatrix according to Algorithm 6. The predicted values are basically a row vector represented as a BlockMatrix, and each column of the matrix represents the predicted value for a single prediction point.

Algorithm 6: Data Matrix Calculation

1 **Procedure** DATAMAT(*PairRDD trainingIndexed*)
  Result: BlockMatrix *Data*
2 Generate $< key, value >$ pairs where key = i and value is a tuple of $< j, v_{ij} >, \forall$ point i, j
3 Group the records based on key i or the row index
4 Generate $< key, value >$ pairs **rows** where key=i and value is a vector of $Data_{ij}$ $\forall i, j$
5 Convert **rows** to **dataRowMatrix** where each record is a **IndexedRow**
6 Convert **dataRowMatrix** to **BlockMatrix** *Data*
7 **return** Data

Algorithm 7: Ordinary Kriging

1 **Procedure** ORDINARY-KRIGING(*BlockMatrix C, BlockMatrix D, BlockMatrix Data*)
2 BlockMatrix **CInverse** $\leftarrow C^{-1}$
3 BlockMatrix **w** $\leftarrow C^{-1}.D$
4 BlockMatrix **wTrans** $\leftarrow w^T$
5 BlockMatrix **predictions** $\leftarrow wTrans.Data$
6 **return** *predictions*

5 Performance analysis

In this section, we attempt to estimate the performances of the block recursive approach of both the non-symmetric and proposed symmetric one for distributed matrix inversion. In this work, we are interested in the *wall clock running time* of the algorithms for a varying number of nodes, matrix sizes, and other algorithmic parameters e.g., partition/block sizes. We are interested in the practical efficiency of our algorithm which includes only the time spent by the processes in the CPU as the algorithms are different only in the number of matrix computations. Therefore, the wall clock time depends only on two independently analyzed quantities: total *computational complexity* of the sub-tasks to be executed and *parallelization factor* of each of the sub-tasks or the total number of processor cores available.

Table 1 Summary of the cost analysis of our distributed inversion algorithm

Method	Comp. Cost	P.F.
leafNode	$\frac{n^3}{b^2}$	—
breakMat	$2b^2 - 2b$	$min\left[\frac{b^2}{4^i}, cores\right]$
xy (filter)	$8b^2 - 4b$	$min\left[\frac{b^2}{4^i}, cores\right]$
xy (map)	$2b^2 - 2b$	$min\left[\frac{b^2}{4^{i+1}}, cores\right]$
multiply	$\frac{n^3}{6b^2}(b^2 - 1)$	$min\left[\frac{n^2}{4^{i+1}}, cores\right]$
transpose	$2b^2 - 2b$	$min\left[\frac{b^2}{4^i}, cores\right]$
subtract	$\frac{n^2}{2b}(b - 1)$	$min\left[\frac{n^2}{4^{i+1}}, cores\right]$
scalarMul	$\frac{b}{2}(b - 1)$	$min\left[\frac{b^2}{4^{i+1}}, cores\right]$
arrange	$\frac{b}{2}(b - 1)$	$min\left[\frac{b^2}{4^{i+1}}, cores\right]$

Comp. Cost computation cost, *P.F.* parallelization factor

We consider only square matrices of dimension 2^p for all the derivations. The key input and tunable parameters for the algorithms are:

- $n = 2^p$: number of rows or columns in matrix A
- b = number of splits for square matrix
- $2^q = \frac{n}{b}$ = block size in matrix A
- $cores$ = total number of physical cores in the cluster
- i = current processing level of algorithm in the recursion tree.
- m = total number of levels of the recursion tree.

Therefore,

- Total number of blocks in matrix $A = b^2$
- $b = 2^{p-q}$

Before going into the details of the analysis, we give the performance analysis of the methods described in Sect. 4.6.2. A summary of the independently analyzed quantities is given in Table 1.

There are two primary parts of the algorithm—*if* part and *else* part. *If* part does the calculation of the leaf nodes of the recursion tree as shown in Fig. 1, while *else* part does the computation for internal nodes. It is clearly seen from the figure that, at level i, there are 2^i nodes and the leaf level contains 2^{p-q} nodes.

There is only one transformation in *if* part which is *map*. It calculates the inverse of a matrix block in a single node using the serial matrix inversion method. The size of each block is n/b, and we need $\approx (n/b)^3$ time to perform each

such method. Therefore, the computation cost to process all the leaf nodes is

$$Comp_{leafNode} = 2^{p-q} \times \left(\frac{n}{b}\right)^3 = \frac{n^3}{b^2}. \tag{15}$$

In our algorithm, leaf nodes process one block on a single machine of the cluster. In spite of being small enough to be accommodated in a single node, we do not collect them in the master node for the communication cost. Instead, we do a *map* which takes the only block of the RDD, do the calculation, and return the RDD again.

else part consists of seven distributed methods as specified in Sect. 4.6.2. Here, we try to compute the computation cost and parallelization factor of each one of them. ***breakMat*** method maps each block of a *BlockMatrix* onto a $< tag, Block >$ using a *mapToPair* transformation. There are b^2 such blocks, and $(1/4)$th of them are processed in each recursion level. Therefore,

$$Comp_{breakMat} = \sum_{i=0}^{m-1} 2^i \times \left(\frac{b^2}{4^i}\right) = 2b\,(b-1)\,. \tag{16}$$

Two transformations of ***xy*** method process $\left(\frac{b^2}{4^i}\right)$ and $\left(\frac{b^2}{4^{i+1}}\right)$ blocks for ith level, respectively. The reason is that the former has to check the tag associated with each block and the latter only transforms only $(1/4)$th of it. Therefore,

$$\begin{aligned} Comp_{xy} &= \left[\frac{\sum_{i=0}^{m-1} 2^i \times \left(\frac{b^2}{4^i}\right)}{min\left[\left(\frac{b^2}{4^i}\right), cores\right]} + \frac{\sum_{i=0}^{m-1} 2^i \times \left(\frac{b^2}{4^{i+1}}\right)}{min\left[\left(\frac{b^2}{4^{i+1}}\right), cores\right]}\right] \\ &= \left[\frac{8b^2 - 4b}{min\left[\left(\frac{b^2}{4^i}\right), cores\right]} + \frac{2b^2 - 2b}{min\left[\left(\frac{b^2}{4^{i+1}}\right), cores\right]}\right]. \end{aligned} \tag{17}$$

multiply method multiplies two *BlockMatrices* each having $b^2/4^{i+1}$ number of blocks at ith level. Every single block in the final *blockMatrix* requires $b/2^i$ block multiplications. Each such multiplication incurs $(n/b)^3$ computational cost. We exempt addition cost because it has lower running time bound than multiplication. Therefore,

$$Comp_{multiply} = \sum_{i=0}^{m-1} 2^i \times \left(\frac{n^3}{8^{i+1}}\right) = \frac{n^3\left(b^2-1\right)}{6b^2}. \tag{18}$$

transpose method transposes a *BlockMatrix* and returns the transposed *BlockMatrix*. The distributed method comprises only one map transformation. The map transformation transposes each matrix block and swaps its row and column indices and returns them. The computational complexity of the map transformation is

$$Comp_{transpose} = \sum_{i=0}^{m-1} 2^i \times \left(\frac{b^2}{4^{i+1}}\right) = 2b\,(b-1)\,. \tag{19}$$

Similar to ***multiply*** method, ***subtract*** subtracts two *BlockMatrices* each having $b^2/4^{i+1}$ number of blocks. Every single block in the final *blockMatrix* requires a single block addition. Each such addition incurs $(n/b)^2$ computational cost. Therefore,

$$Comp_{subtract} = \sum_{i=0}^{m-1} 2^i \times \left(\frac{n^2}{4^{i+1}}\right) = \frac{n^2\,(b-1)}{2b}. \tag{20}$$

scalarMul method takes a *BlockMatrix* and returns another *BlockMatrix* using a *map* transformation. The *map* takes blocks one by one and multiply each element of the block with the scalar. Therefore, the computation cost of *scalarMul* is

$$Comp_{scalarMul} = \sum_{i=0}^{m-1} 2^i \times \left(\frac{b^2}{4^{i+1}}\right) = \frac{b}{2}\,(b-1)\,. \tag{21}$$

arrange method takes four sub-matrices of size 2^{n-1} which represents four coordinates of a full matrix of size 2^n and arranges them in later and returns it as *BlockMatrix*. It consists of four *map*s, each one for a separate *BlockMatrix*. Each *map* maps the block index onto a different block index that provides the final position of the block in the result matrix. The computation cost for *maps* is same as *scalarMul*, which can be found in Eq. 21.

Each method experiences one of the three variants of *parallelization factor* based on how much data are being computed in parallel and the total number of physical cores available in the cluster. ***breakMat*** method and *filter* transformation of ***xy*** method process matrices of $b^2/4^i$ blocks at ith recursion level, and each such block is processed in parallel. Therefore, each one has the following parallelization factor:

$$PF_A = min\left[\left(\frac{b^2}{4^i}\right), cores\right]. \tag{22}$$

Similarly, *map* transformation of ***xy*** method, ***transpose***, ***scalarMul***, and ***arrange*** method process matrices of $b^2/4^{i+1}$ blocks at ith recursion level and each such block is processed in parallel. Therefore,

$$PF_B = min\left[\left(\frac{b^2}{4^{i+1}}\right), cores\right]. \tag{23}$$

In ***multiply*** and ***subtract*** methods, the processing to obtain every element of a block of the result matrix can be done in parallel giving the PF as

$$PF_C = min\left[\frac{n^2}{4^{i+1}}, cores\right]. \tag{24}$$

The non-symmetric inversion approach requires 1 breakmat, 4 *xy* method calls, 6 multiplications, 2 subtractions, 1 scalar multiplication, and 1 arrange method call for each recursion level. When summed up, it gives the following cost:

$$Cost_{NS} = \left(\frac{n^3}{b^2}\right) + \frac{34b^2 - 18b}{PF_A} + \frac{9b^2 - 9b}{PF_B} + \frac{n^2(b-1)[n(b+1)+b]}{b^2 \times PF_C}. \tag{25}$$

The symmetric inversion approach requires 1 breakmat, 4 *xy* method calls, 4 multiplications, 2 transposes, 2 subtractions, 1 scalar multiplication, and 1 arrange method call for each recursion level. When summed up, it gives the following cost:

$$Cost_S = \left(\frac{n^3}{b^2}\right) + \frac{34b^2 - 18b}{PF_A} + \frac{13b^2 - 13b}{PF_B} + \frac{n^2(b-1)[2n(b+1)+3b]}{3b^2 \times PF_C} \tag{26}$$

If we consider only the highest terms of both the equations, it will lead to the following inequality:

$$Cost_S \approx 2/3 \times Cost_{NS}. \tag{27}$$

Later, in Sect. 6, we will show that this is true when the algorithms are run in the cluster.

6 Experiments

In this section, we report results from experiments performed to evaluate the performance of our OK implementation considering different matrix inversion techniques. Before conducting the experiments, we first describe the dataset used for the experiments in Sect. 6.1. Then, in Sect. 6.3, we test the accuracy of the kriging algorithm to show that the accuracy of the kriging system increases as we increase the number of interpolating points to build the model. In Sect. 6.4, we measure the wall clock execution time of matrix inversion and multiplication separately with increasing interpolating points and prediction points and show that matrix inversion is the most dominant part of the entire algorithm. At last, we measure the combined wall clock execution time of both inversion and multiplication using three distributed and one single node implementation.

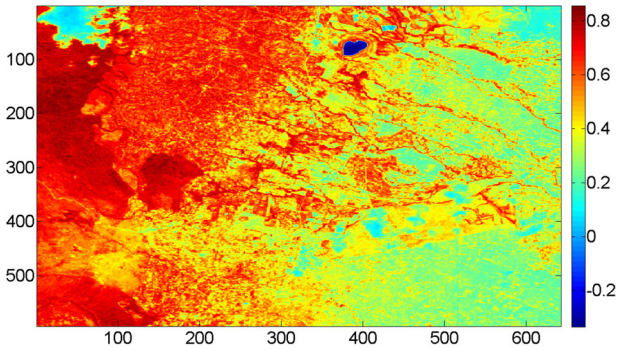

Fig. 2 NDVI image of the whole study area

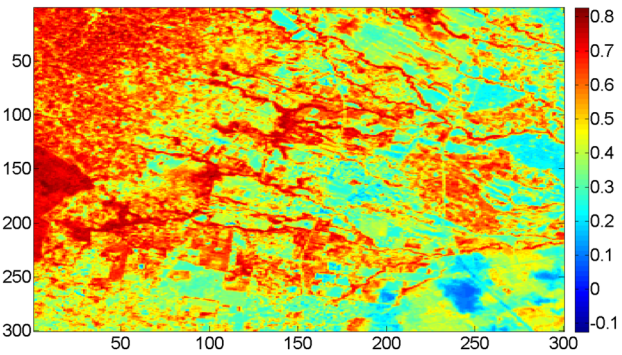

Fig. 3 Cropped (150 × 150) NDVI image

6.1 Landsat data and extraction

For this paper, we have taken the Landsat image data for our experiment from Landsat 8 captured during the month of March 2016. All the bands are merged in a TIFF file and have size 642 × 593 with 30 m × 30 m resolution. Therefore, a total of 380,706 pixels are present in the image. The image extends from 251,940 to 269,730 along latitude and −382,140 to −362,880 along longitude on $WGS84$ reference frame with the meter as a unit of measurements. The NDVI for the whole image is shown in Fig. 2.

To get the study area, we have cropped the whole image to get a smaller one with size 150 × 150 with 22500 cells. After that, we converted the raster into NDVI image and saved it to the disk. The minimum and maximum NDVI value are −0.1257988 and 0.7970431, respectively. The cropped NDVI image is shown in Fig. 3.

6.2 Test setup

All the experiments are carried out on a dedicated cluster of 3 nodes. Software and hardware specifications are summarized in Table 2. For block-level multiplications, we use Breeze, a

Table 2 Test setup components specifications

Comp. name	Comp. size	Spec.
Processor	2	Intel Xeon 2.60 GHz
Core	6 per processor	NA
Physical memory	132 GB	NA
Ethernet	14 Gb/s	Infini Band
OS	NA	CentOS 5
File system	NA	Ext3
Apache Spark	NA	1.6.0
Apache Hadoop	NA	2.6.0
Java	NA	1.7.0 update 79

Comp. Name component name, *Comp. Size* component size, *Spec.* specification), *NA* not applicable

single-node low-level linear algebra library. Breeze provides Java APIs to the implementation through Spark, and it calls C/Fortran-implemented native library, such as BLAS, to execute the linear algebra computation through the Java Native Interface (JNI).

We choose the block size of matrices to be 500 so that it can provide a reasonable amount of time to complete execution of the matrix operations. We avoid too small block size which generates large matrix blocks and also too large block size which takes too much time to complete.

6.3 Interpolation error with input data size

In the first experiment, we measure the reconstruction error as a function of increasing input dataset size. To conduct the experiment, we increase the number of training points from 1000 to 20,000 and kept the number of points to be interpolated fixed at 1000 and plot the error measured in terms of mean squared error (MSE) as shown in Fig. 4.

It is clearly seen that the reconstruction error of the model decreases as we include more interpolating points to create the model. Although the slope of the plot is decreasing, the error is prominent even at 20,000 points, a typical input dataset size for geospatial modeling as has been given in [8,29,31]. However, creating models using such large input dataset requires servers with large memory and thus may be expensive for small research institutions. For example, a typical desktop having 8 GB memory starts experiencing thrashing for as many as 15,000 datasets, while a server having 120 GB memory takes 974 s using an optimized package like R GStat. Therefore, smaller reconstruction error provides sufficient support to build kriging system model on big data platforms which can scale to such large geospatial datasets, using multiple workstations to get better estimated values at locations of unknown attribute values.

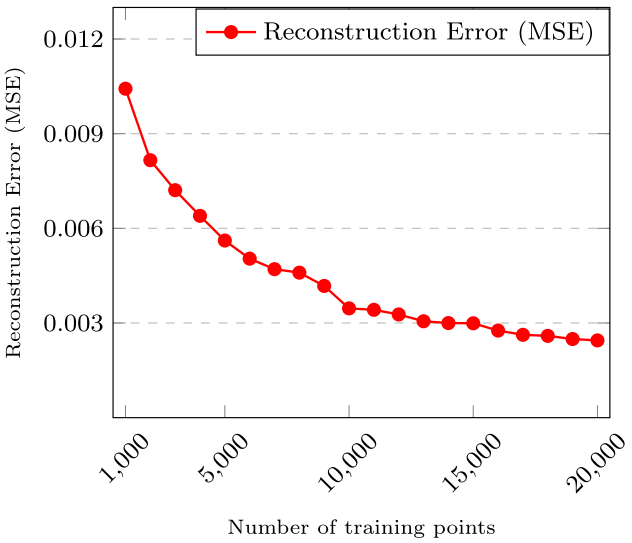

Fig. 4 Reconstruction error (MSE) with increasing number of interpolating points

6.4 Execution time of Ordinary Kriging algorithm

In this section, we calculate the wall clock execution time for solving OK equation, most time-consuming operation, and require two matrix operations—(1) matrix inversion of covariance matrix (CINV) and (2) matrix multiplication of $C^{-1}.D$ to get the weight matrix w (MATMUL). By calculating wall clock execution time of these two operations, we can perform runtime comparative analysis between them. We conduct two sets of experiments—first with an increasing number of interpolating points and then with an increasing number of points to be interpolated. For the first experiment, we increase the number of interpolating points from $1000 - 20{,}000$ keeping the number of points to be interpolated fixed at 1000. In the second experiment, we keep the number of interpolating points fixed at 20,000 and increase the number of prediction points from 1000 to 10,000. Tables 3 and 4 show the wall clock execution time breakdown for both the cases.

From the results, it is clear that matrix inversion is the dominant operation in solving OK equation within a threshold of the number of prediction points. Obviously, if we further increase the number of prediction points the multiplication operation will surpass the inversion cost as both have a computational complexity of $O(n^3)$, where n is the dimension of the matrix. But in situations where the model required to be tested on a subset of data frequently before testing on the final prediction points, the inversion cost is the primary bottleneck and thus needed to be addressed to reduce the overall running time of the OK algorithm.

Table 3 Wall clock execution time of matrix inversion and matrix multiplication with increasing number of interpolating points

No. of interpolating points	CINV	MATMUL
1000	9	4
2000	21	6
3000	36	9
4000	48	11
5000	67	13
6000	84	15
7000	104	18
8000	118	21
9000	157	24
10,000	187	27
11,000	238	36
12,000	250	38
13,000	301	36
14,000	351	39
15,000	411	44
16,000	452	49
17,000	762	50
18,000	802	57
19,000	1004	58
20,000	1127	60

Table 4 Wall clock execution time of matrix inversion and matrix multiplication with increasing number of points to be interpolated

No. of Prediction Points	CINV	MATMUL
1000	1127	38
2000	1120	48
3000	1125	90
4000	1105	96
5000	1122	129
6000	1097	141
7000	1112	163
8000	1100	175
9000	1092	195
10,000	1102	206

6.5 Comparison with state of the art

In this section, we compare the wall clock execution time of the OK equation implemented using three distributed matrix inversion methods as shown in Fig. 5. The distributed methods are—(1) LU-based block recursive matrix inversion, (2) Strassen's block recursive inversion for a general matrix, and (3) our symmetric matrix-based implementation. We estimate that a target surface containing 1000 points with the number of interpolating points varying from 1000 to 16,000

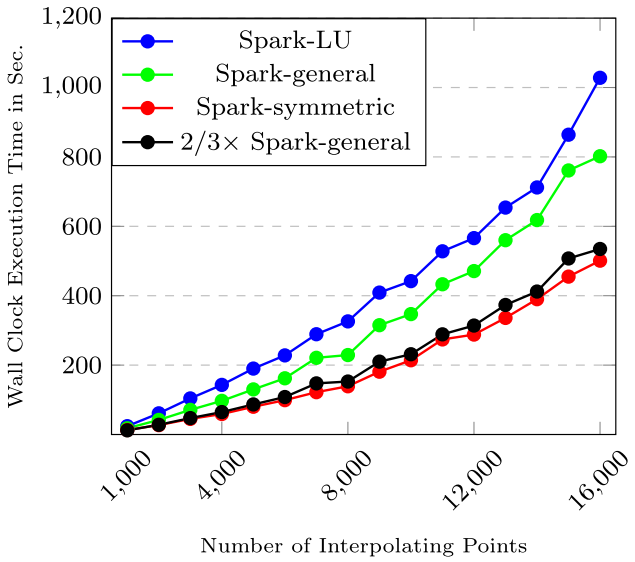

Fig. 5 Wall clock execution time for solving kriging equation using four competing approaches—(1) distributed approaches using Spark-based matrix inversion **a** based on LU-based block recursion, **b** Strassen's general block recursion, and **c** Strassen's symmetric block recursion and (2) (2/3)× Strassen's general block recursion

would be a standard test case for the OK algorithm. It can be seen that the proposed approach has the best time performance for solving the OK system. Also, note that time taken by the symmetric approach is almost 2/3 times of the non-symmetric one (see the black line) which supports our theoretical analysis. Moreover, it supports the idea of being able to reduce the overall execution time of OK algorithm considerably by using our inversion approach.

7 Conclusion

In this paper, we have focused on the problem of Ordinary Kriging where the number of interpolating points is large and provided an Apache Spark-based end-to-end distributed implementation of steps involved in the process which was not present in earlier works. While attempting to reduce the overall running time of the Ordinary Kriging algorithm, we noticed that the inversion of the variance–covariance matrix is the most time-consuming operation and reducing the inversion time affects the overall running time the most. We decrease the running time of the inversion using a distributed block recursive approach and also exploited the symmetric nature of the matrix to further reduce the cost. Through experimentation on the wall clock execution time, we found that matrix inversion is indeed the costly operation and our distributed symmetric block recursive approach outperformed other baseline distributed approaches with a substantial margin. We have performed the theoretical analysis of a non-symmetric and symmetric version of

the inversion approaches and found that our approach is 2/3 times faster than its non-symmetric counterpart owing to less number of matrix multiplications required in each recursion level which is also supported by the results shown in Fig. 5.

Compliance with ethical standards

Conflict of interest On behalf of all authors, the corresponding author states that there is no conflict of interest.

References

1. Abatzoglou, J.T.: Development of gridded surface meteorological data for ecological applications and modelling. Int. J. Climatol. **33**(1), 121–131 (2013)
2. ArcGIS: creating empirical semivariograms. http://desktop.arcgis.com/en/arcmap/latest/extensions/geostatistical-analyst/creating-empirical-semivariograms.htm/ (2016). Accessed 30 Dec 2019
3. Caesar, J., Alexander, L., Vose, R.: Large-scale changes in observed daily maximum and minimum temperatures: creation and analysis of a new gridded data set. J. Geophys. Res. Atmos. **111**(D5), D05101 (2006)
4. Chen, M., Shi, W., Xie, P., Silva, V., Kousky, V.E., Wayne Higgins, R., Janowiak, J.E.: Assessing objective techniques for gauge-based analyses of global daily precipitation. J. Geophys. Res. Atmos. **113**(D4), D04110 (2008)
5. Cheng, T.: Accelerating universal kriging interpolation algorithm using CUDA-enabled GPU. Comput. Geosci. **54**, 178–183 (2013)
6. Cheng, T., Li, D., Wang, Q.: On parallelizing universal kriging interpolation based on OpenMP. In: 2010 Ninth International Symposium on Distributed Computing and Applications to Business Engineering and Science (DCABES), pp. 36–39. IEEE (2010)
7. Cressie, N.: Statistics for Spatial Data. Wiley, New York (2015)
8. Cressie, N., Kang, E.L.: High-resolution digital soil mapping: Kriging for very large datasets. In: ViscarraRossel, R.A., McBratney, A.B., Minasny, B. (eds.) Proximal Soil Sensing, pp. 49–63. Springer, New York (2010)
9. Deutsch, C.V., Journel, A.G., et al.: Geostatistical Software Library and User's Guide, vol. 119, 147th edn. Oxford University Press, New York (1992)
10. Di Luzio, M., Johnson, G.L., Daly, C., Eischeid, J.K., Arnold, J.G.: Constructing retrospective gridded daily precipitation and temperature datasets for the conterminous united states. J. Appl. Meteorol. Climatol. **47**(2), 475–497 (2008)
11. Gebhardt, A.: PVM kriging with R. In: Proceedings of the 3rd International Workshop on Distributed Statistical Computing, Vienna. Citeseer (2003)
12. Gribov, A., Krivoruchko, K., Ver Hoef, J.M.: Modified weighted least squares semivariogram and covariance model fitting algorithm. Stoch. Model. Geostat. Princ. Methods Case Stud. **2** (2000)
13. Gyalistras, D.: Development and validation of a high-resolution monthly gridded temperature and precipitation data set for Switzerland (1951–2000). Clim. Res. **25**(1), 55–83 (2003)
14. Hadoop, A.: Apache Hadoop project. https://hadoop.apache.org/ (2016). Accessed 30 Dec 2019
15. Haylock, M., Hofstra, N., Klein Tank, A., Klok, E., Jones, P., New, M.: A European daily high-resolution gridded data set of surface temperature and precipitation for 1950–2006. J. Geophys. Res. Atmos. **113**(D20), D20119 (2008)
16. Hernandez-Penaloza, G., Beferull-Lozano, B.: Field estimation in wireless sensor networks using distributed kriging. In: 2012 IEEE International Conference on Communications (ICC), pp. 724–729. IEEE (2012)
17. Herrera, S., Gutiérrez, J.M., Ancell, R., Pons, M., Frías, M., Fernández, J.: Development and analysis of a 50-year high-resolution daily gridded precipitation dataset over Spain (Spain02). Int. J. Climatol. **32**(1), 74–85 (2012)
18. Hu, H., Shu, H.: An improved coarse-grained parallel algorithm for computational acceleration of ordinary Kriging interpolation. Comput. Geosci. **78**, 44–52 (2015)
19. Hutchinson, M.F., McKenney, D.W., Lawrence, K., Pedlar, J.H., Hopkinson, R.F., Milewska, E., Papadopol, P.: Development and testing of Canada-wide interpolated spatial models of daily minimum–maximum temperature and precipitation for 1961–2003. J. Appl. Meteorol. Climatol. **48**(4), 725–741 (2009)
20. Isaaks, E.H., Srivastava, R.M.: An Introduction to Applied Geostatistics. Oxford University Press, Oxford (1989)
21. Jardak, C., Mahonen, P., Riihijarvi, J.: Spatial big data and wireless networks: experiences, applications, and research challenges. Netw. IEEE **28**(4), 26–31 (2014)
22. Jardak, C., Riihijarvi, J., Oldewurtel, F., Mahonen, P.: Parallel processing of data from very large-scale wireless sensor networks. In: Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing, pp. 787–794. ACM (2010)
23. Jeffrey, S.J., Carter, J.O., Moodie, K.B., Beswick, A.R.: Using spatial interpolation to construct a comprehensive archive of Australian climate data. Environ. Model. Softw. **16**(4), 309–330 (2001)
24. Kaluzny, S.P., Vega, S.C., Cardoso, T.P., Shelly, A.A.: S+ SpatialStats: User's Manual for Windows ® and UNIX®. Springer, New York (2013)
25. Kerry, K., Hawick, K.: Spatial interpolation on distributed, high-performance computers. In: Proceedings of High-Performance Computing and Networks (HPCN) Europe, vol. 98 (1997)
26. Kerry, K., Hawick, K.A.: Kriging interpolation on high-performance computers. In: International Conference on High-Performance Computing and Networking, pp. 429–438. Springer (1998)
27. Krishnamurti, T., Mishra, A., Simon, A., Yatagai, A.: Use of a dense gauge network over india for improving blended TRMM products and downscaled weather models. J. Meteorol. Soc. Jpn. **87**, 395416 (2009)
28. Krishnan, S., Baru, C., Crosby, C.: Evaluation of MapReduce for gridding LIDAR Data. In: 2010 IEEE Second International Conference on Cloud Computing Technology and Science (CloudCom), pp. 33–40. IEEE (2010)
29. Lesniak, A., Porzycka, S.: Geostatistical computing in PSInSAR data analysis. In: Computational Science—ICCS 2009, pp. 397–405. Springer (2009)
30. Liu, J., Liang, Y., Ansari, N.: Spark-based large-scale matrix inversion for big data processing. IEEE Access **4**, 2166–2176 (2016)
31. Memarsadeghi, N., Raykar, V.C., Duraiswami, R., Mount, D.M.: Efficient kriging via fast matrix-vector products. In: 2008 IEEE Aerospace Conference, pp. 1–7. IEEE (2008)
32. Meyer, T.H.: The discontinuous nature of Kriging interpolation for digital terrain modeling. Cartogr. Geogr. Inf. Sci. **31**(4), 209–216 (2004)
33. Misra, C., Haldar, S., Bhattacharya, S., Ghosh, S.K.: Spin: A fast and scalable matrix inversion method in apache spark. In: Proceedings of the 19th International Conference on Distributed Computing and Networking, ICDCN '18, pp. 16:1–16:10. ACM, New York (2018). https://doi.org/10.1145/3154273.3154300
34. Oliver, M.A., Webster, R.: Kriging: a method of interpolation for geographical information systems. Int. J. Geogr. Inf. Syst. **4**(3), 313–332 (1990)

35. Pebesma, E.J.: Multivariable geostatistics in S: the GSTAT package. Comput. Geosci. **30**(7), 683–691 (2004)
36. Perry, M., Hollis, D.: The generation of monthly gridded datasets for a range of climatic variables over the UK. Int. J. Climatol. **25**(8), 1041–1054 (2005)
37. Pesquer, L., Cortés, A., Pons, X.: Parallel ordinary kriging interpolation incorporating automatic variogram fitting. Comput. Geosci. **37**(4), 464–473 (2011)
38. Rajeevan, M., Bhate, J.: A high resolution daily gridded rainfall dataset (1971–2005) for mesoscale meteorological studies. Curr. Sci **96**(4), 558–562 (2009)
39. Rajeevan, M., Bhate, J., Jaswal, A.: Analysis of variability and trends of extreme rainfall events over India using 104 years of gridded daily rainfall data. Geophys. Res. Lett. **35**(18), L18707 (2008)
40. Rajeevan, M., Bhate, J., Kale, J., Lal, B.: High resolution daily gridded rainfall data for the Indian region: analysis of break and active monsoon spells. Curr. Sci. **91**(3), 296–306 (2006)
41. Riihijarvi, J., Mahonen, P.: Highly scalable data processing framework for pervasive computing applications. In: 2013 IEEE International Conference on Pervasive Computing and Communications Workshops (PERCOM Workshops), pp. 306–308. IEEE (2013)
42. Rizki, P.N.M., Eum, J., Lee, H., Oh, S.: Spark-based in-memory DEM creation from 3D LiDAR point clouds. Remote Sens. Lett. **8**(4), 360–369 (2017)
43. Rizki, P.N.M., Lee, H., Lee, M., Oh, S.: High-performance parallel approaches for three-dimensional light detection and ranging point clouds gridding. J. Appl. Remote Sens. **11**(1), 016011 (2017)
44. Shepard, D.: A two-dimensional interpolation function for irregularly-spaced data. In: Proceedings of the 1968 23rd ACM National Conference, ACM '68, pp. 517–524. ACM, New York (1968). https://doi.org/10.1145/800186.810616
45. Shi, X., Ye, F.: Kriging interpolation over heterogeneous computer architectures and systems. GISci. Remote Sens. **50**(2), 196–211 (2013)
46. Software, S.: Statistical analysis software, SAS/STAT. https://www.sas.com/en_us/software/stat.html (2016). Accessed 30 Dec 2019
47. Spark, A.: Apache spark, lightning-fast cluster computing. https://spark.apache.org/ (2016). Accessed 30 Dec 2019
48. Srinivasan, B.V., Duraiswami, R., Murtugudde, R.: Efficient kriging for real-time spatio-temporal interpolation. In: Proceedings of the 20th Conference on Probability and Statistics in the Atmospheric Sciences, pp. 228–235. American Meteorological Society, Atlanta (2010)
49. Stein, M.L.: Interpolation of Spatial Data: Some Theory for Kriging. Springer, New York (2012)
50. Strzelczyk, J., Porzycka, S.: Parallel kriging algorithm for unevenly spaced data. In: Manninen, P., Öster, P. (eds.) Applied Parallel and Scientific Computing, pp. 204–212. Springer, New York (2012)
51. Strzelczyk, J., Porzycka, S., Lesniak, A.: Analysis of ground deformations based on parallel geostatistical computations of PSInSAR data. In: 2009 17th International Conference on Geoinformatics, pp. 1–6. IEEE (2009)
52. Turk, F.J., Arkin, P., Sapiano, M.R., Ebert, E.E.: Evaluating high-resolution precipitation products. Bull. Am. Meteorol. Soc. **89**(12), 1911–1916 (2008)
53. Wackernagel, H.: Multivariate Geostatistics: An Introduction with Applications. Springer, New York (2013)
54. Xiang, J., Meng, H., Aboulnaga, A.: Scalable matrix inversion using MapReduce. In: Proceedings of the 23rd International Symposium on High-Performance Parallel and Distributed Computing, pp. 177–190. ACM (2014)
55. Yatagai, A., Arakawa, O., Kamiguchi, K., Kawamoto, H., Nodzu, M.I., Hamada, A.: A 44-year daily gridded precipitation dataset for Asia based on a dense network of rain gauges. Sola **5**, 137–140 (2009)
56. Yatagai, A., Xie, P., Alpert, P.: Development of a daily gridded precipitation data set for the middle east. Adv. Geosci. **12**, 165–170 (2008)
57. Zhang, M., Wang, H., Lu, Y., Li, T., Guang, Y., Liu, C., Edrosa, E., Li, H., Rishe, N.: TerraFly GeoCloud: an online spatial data analysis and visualization system. ACM Trans. Intell. Syst. Technol. TIST **6**(3), 34 (2015)
58. Zhuo, W., Paciorek, C., Kaufman, C., Bethel, W., et al.: Parallel kriging analysis for large spatial datasets. In: 2011 IEEE 11th International Conference on Data Mining Workshops (ICDMW), pp. 38–44. IEEE (2011)