Preprint ANL/MCS-P5015-0913

A Parallel Tree Code for Computing Matrix-Vector Products with the Matérn Kernel

JIE CHEN, Argonne National Laboratory
LEI WANG, University of Wisconsin–Milwaukee
MIHAI ANITESCU, Argonne National Laboratory

The Matérn kernel is one of the most widely used covariance kernels in Gaussian process modeling; however, large-scale computations have long been limited by the expensive dense covariance matrix calculations. As a sequel of our recent paper [Chen et al. 2012] that designed a tree code algorithm for efficiently performing the matrix-vector multiplications with the Matérn kernel, this paper documents the parallel design and the software implementation of the algorithm. The parallelization focuses on data and work load balancing and uses MPI passive one-sided protocols for communications. The software, implemented in C++, provides a flexible interface with rich functionality, together with examples to demonstrate the extraction of performance diagnostics. The code is intended to be used as building blocks for statistical calculations where the matrix-vector multiplication is among the most expensive computational components.

Categories and Subject Descriptors: G.1.0 [**Numerical Analysis**]: General—*Numerical algorithms*; *Parallel algorithms*

General Terms: Design, Algorithms, Performance

Additional Key Words and Phrases: Matérn kernel, matrix-vector multiplication, fast kernel summation, tree code, MPI, one-sided communication

1. INTRODUCTION

The Matérn kernel is one of the most widely used positive-definite covariance kernel in statistical analysis of spatiotemporal data [Stein 1999; Chilès and Delfiner 1999; Wendland 2005; Rasmussen and Williams 2006]. It consists of a family of Matérn functions of arbitrary positive orders ν, including as special cases the exponential function (when $\nu = 1/2$) and the double exponential function, that is, the Gaussian (when $\nu = \infty$). The flexibility in capturing the local smoothness of the data with a suitable order ν contributes to its popularity in various statistical modeling scenarios [Chilès and Delfiner 1999; Wendland 2005]. Associated with the Matérn kernel is a covariance matrix of size $n \times n$ for a set of n points, for each of which a random variable is defined. The covariance matrix is central in the analysis of Gaussian processes, the topics of which include sampling, regression, classification, and model selection, whereby the computation of the matrix-vector product is a common subroutine. Examples of the use of the Matérn matrix-vector product include solving a linear system by using an iterative method [Anitescu et al. 2012] and approximating the matrix square-root function by a least-squares polynomial [Chen et al. 2011].

A fast summation algorithm for computing the Matérn matrix-vector product for a scattered set of points was recently developed by Chen et al. [2012]. This was the first fast summation algorithm dedicated to the Matérn kernel of a general order ν (for an earlier algorithm that handles specifically $\nu = 1.5$, see Anitescu et al. [2012]; and for a software implementation that may be extended to handle the Matérn kernel because of its kernel-independent feature, see Ying et al. [2003]). Our algorithm follows the tree-code framework and entails an empirically $O(n \log n)$ computational cost, which

Author's addresses: J. Chen and M. Anitescu, Mathematics and Computer Science Division, Argonne National Laboratory, {jiechen,anitescu}@mcs.anl.gov. L. Wang, Department of Mathematical Sciences, University of Wisconsin–Milwaukee, wang256@uwm.edu.

is asymptotically lower than the $O(n^2)$ cost as required by a straightforward matrix-vector multiplication. The algorithm addresses various practical needs by adopting several unconventional designs, including a balanced partitioning of the point set according to principal component analysis, a double expansion of the kernel for reducing the storage of intermediate results, an error-estimate strategy based on sampling and fitting, and a precomputation stage for handling multiple vectors. Furthermore, the algorithm can easily extend to the calculations for the differentiated kernels, that is, the partial derivatives of the kernel with respect to parameters, when the kernel is parameterized. Parameterized kernels typically appear in the maximum likelihood methodology for model selection. Of particular interest are the parameters that introduce anisotropy to the kernel by scaling each coordinate of the space [Anitescu et al. 2012; Stein et al. 2013].

As a sequel of the work of Chen et al. [2012], this paper documents the engineering of a parallel implementation of the algorithm. The program is intended to be used as a software library on top of which statistical computing codes are built. Of particular emphases are the following:

— Parallelization strategy that handles data and work load balancing and the implied communication protocol used in a distributed-memory environment (Section 3),
— Interface design that addresses both flexibility and functionality expressiveness (Section 4),
— Novel data structure for storing and retrieving data with d indices where the index sum is bounded, as required by the storage of multidimensional Taylor coefficients (Section 5).

We report several performance results (Section 6) of the code running on a medium-sized parallel computing cluster, with the largest experiment involving $1,024$ MPI processes. The computing cluster was built with several hundred 16-core nodes interconnected by QDR InfiniBand[1] and with the MVAPICH2 compiler[2] support. Thus, the best performance of the code was obtained by enabling asynchronous progress and by running every MPI process with two cores, one of which is dedicated to the MPI passive one-sided communications. Because of the ongoing development of the MPI standard and implementations and hardware features, it is unclear at present whether, and if so, how, the aforementioned restrictions on the complex hardware and software settings can be removed. Nevertheless, the usage of the software poses no difficulty on usual commodity single-core or multicore machines or computing clusters, as long as the compiler supports the MPI-2 standard [Message Passing Interface Forum 2003]. The software is publicly available from the following two websites:

— `http://press3.mcs.anl.gov/scala-gauss/software/`
— `http://www.mcs.anl.gov/~jiechen/software.html`

2. MATÉRN TREE CODE

The family of Matérn functions of order $\nu > 0$ is defined as (see, e.g., Rasmussen and Williams [2006])

$$\phi(r) = \frac{(\sqrt{2\nu}r)^\nu \mathsf{K}_\nu(\sqrt{2\nu}r)}{2^{\nu-1}\Gamma(\nu)},$$

where K_ν is the modified Bessel function of the second kind of order ν and Γ is the Gamma function. When the Matérn function is used as a radial basis kernel, the vari-

[1] `http://en.wikipedia.org/wiki/InfiniBand`
[2] `http://mvapich.cse.ohio-state.edu/overview/mvapich2/`

able r denotes the scaled, elliptical distance between two points $\boldsymbol{x}$ and $\boldsymbol{y}$ in $\mathbb{R}^d$:

$$r = \sqrt{\sum_{i=1}^{d} \frac{r_i^2}{\ell_i^2}} \qquad \text{with} \qquad r_i = x_i - y_i, \tag{1}$$

where the ℓ_i's are scaling factors, one for each coordinate. We use the boldface letter $\boldsymbol{\ell}$ to denote the vector of all ℓ_i's. For convenience of presentation, we will write the Matérn kernel by abuse of notation in different forms: $\phi(r)$, $\phi(\boldsymbol{x}-\boldsymbol{y})$ or $\phi(\boldsymbol{x},\boldsymbol{y})$, which will not cause confusion under a clear context.

Given a set of n points $\{\boldsymbol{x}_i \in \mathbb{R}^d\}$ and a length-n vector $\boldsymbol{q}$, this software computes the matrix-vector product $\boldsymbol{s} = \Phi\boldsymbol{q}$ where the matrix Φ is defined based on the Matérn kernel:

$$\Phi_{ij} = \phi(\boldsymbol{x}_i - \boldsymbol{x}_j).$$

In certain applications, such as maximum likelihood estimation (MLE), the differentiated kernel $\phi^{[k]} \equiv \partial\phi/\partial\ell_k$ is of interest in addition to ϕ (see Anitescu et al. [2012] for the matrix-free technique of performing large-scale MLE that avoids factorizations). In this case, the matrix-vector products $\Phi^{[k]}\boldsymbol{q}$, where

$$\Phi^{[k]}_{ij} = \phi^{[k]}(\boldsymbol{x}_i - \boldsymbol{x}_j),$$

are additionally computed for the same $\boldsymbol{q}$. The organization of the calculations with the differentiated kernels is similar to that of ϕ; thus in what follows the discussions with differentiated kernels are omitted except when necessary.

2.1. Mathematical Background

The tree code algorithm for computing $\Phi\boldsymbol{q}$ is based on a Taylor approximation of the kernel with error estimates. The kernel $\phi(\boldsymbol{x}_i, \boldsymbol{x}_j)$ takes a pair of points, $\boldsymbol{x}_i$ and $\boldsymbol{x}_j$, as arguments. Because it is sometimes confusing when one distinguishes the two points only by using the index, we slightly change the notation of the second argument from $\boldsymbol{x}_j$ to $\boldsymbol{y}_j$ and write the matrix-vector product in a summation form with this change of notation:

$$s_i = \sum_{j=1}^{n} q_j \phi(\boldsymbol{x}_i, \boldsymbol{y}_j), \quad \text{for} \quad i = 1, \ldots, n. \tag{2}$$

By convention, we call $\boldsymbol{x}_i$ the target, $\boldsymbol{y}_j$ the source, and q_j the weight, in recognition of the fact that each source contributes with a certain weight to the target. The kernel admits a Taylor expansion around two distinct centers $\boldsymbol{x}_c$ and $\boldsymbol{y}_c$:

$$\phi(\boldsymbol{x}_c + \Delta\boldsymbol{x}, \boldsymbol{y}_c + \Delta\boldsymbol{y}) = \sum_{\|\boldsymbol{j}\|=0}^{\infty} \sum_{\|\boldsymbol{k}\|=0}^{\infty} \binom{\boldsymbol{j}+\boldsymbol{k}}{\boldsymbol{j}} \frac{\partial_{\boldsymbol{y}}^{\boldsymbol{j}+\boldsymbol{k}} \phi(\boldsymbol{x}_c, \boldsymbol{y}_c)}{(\boldsymbol{j}+\boldsymbol{k})!} (-\Delta\boldsymbol{x})^{\boldsymbol{j}} (\Delta\boldsymbol{y})^{\boldsymbol{k}}. \tag{3}$$

Here, the vectorial notation is standard in multivariate calculus, and $\partial_{\boldsymbol{y}}^{\boldsymbol{k}}\phi$ for any $\boldsymbol{k}$ means the partial derivative of $\phi(\boldsymbol{x},\boldsymbol{y})$ with respect to $\boldsymbol{y}$ of order $\boldsymbol{k}$. We denote by C_t a cluster of target points with centroid $\boldsymbol{x}_c$ and C_s a cluster of source points with centroid $\boldsymbol{y}_c$, and for any $\boldsymbol{x}_i \in C_t$ we define the partial sum

$$s_i(C_s) := \sum_{\boldsymbol{y}_j \in C_s} q_j \phi(\boldsymbol{x}_i, \boldsymbol{y}_j). \tag{4}$$

Using a pair of sufficiently large orders (p_1, p_2), one can approximate $s_i(C_s)$ by truncating the Taylor expansion as in

$$s_i(C_s) \approx \sum_{\|\boldsymbol{j}\|=0}^{p_1} \sum_{\|\boldsymbol{k}\|=0}^{p_2} \underbrace{\binom{\boldsymbol{j}+\boldsymbol{k}}{\boldsymbol{j}}}_{\text{binom. coef.}} \underbrace{\frac{\partial_{\boldsymbol{y}}^{\boldsymbol{j}+\boldsymbol{k}} \phi(\boldsymbol{x}_c, \boldsymbol{y}_c)}{(\boldsymbol{j}+\boldsymbol{k})!}}_{\text{Taylor coef.}} \underbrace{(\boldsymbol{x}_c - \boldsymbol{x}_i)^{\boldsymbol{j}}}_{\text{target momt.}} \underbrace{\left[\sum_{\boldsymbol{y}_j \in C_s} q_j (\boldsymbol{y}_j - \boldsymbol{y}_c)^{\boldsymbol{k}} \right]}_{\text{weighted source momt.}}. \tag{5}$$

Here, we annotate the individual terms of (5) in the summand as binomial coefficients, Taylor coefficients, target moments, and weighted source moments.

We write

$$G_\nu^{\boldsymbol{k}} \equiv \frac{\partial_{\boldsymbol{y}}^{\boldsymbol{k}} \phi(\boldsymbol{x}_c, \boldsymbol{y}_c)}{\boldsymbol{k}!}. \tag{6}$$

These Taylor coefficients can be computed by recursing on $\boldsymbol{k}$ and ν. Because ν denotes a given Matérn order, in the recursions we change the notation of ν to u. The detailed derivation is given by Chen et al. [2012]. In summary, the recurrence is given by

$$G_u^{\boldsymbol{k}} = \frac{2\nu h_u}{\|\boldsymbol{k}\|} \left[\sum_{i=1}^{d} \frac{r_i}{\ell_i^2} G_{u-1}^{\boldsymbol{k}-\boldsymbol{e}_i} - \sum_{i=1}^{d} \frac{1}{\ell_i^2} G_{u-1}^{\boldsymbol{k}-2\boldsymbol{e}_i} \right], \tag{7}$$

where

$$h_u = \begin{cases} \dfrac{1}{2(u-1)}, & u > 1 \\ z(\sqrt{2\nu} r), & u = 1 \\ \dfrac{(\sqrt{2\nu} r)^{2u-2} \Gamma(1-u)}{2^{2u-1}\Gamma(u)}, & 0 < u < 1 \end{cases} \qquad h_u = \begin{cases} \dfrac{1}{2\nu z(\sqrt{2\nu} r)}, & u = 0 \\ -\dfrac{u}{\nu}, & u < 0, \end{cases}$$

$$z(R) = \begin{cases} -\gamma - \log\left(\frac{R}{2}\right), & 0 < R < R_0 \\ 1, & R \geq R_0, \end{cases} \qquad R_0 = 2e^{-\gamma-1},$$

and $\gamma \approx 0.577216$ is the Euler–Mascheroni constant. By convention, $G_u^{\boldsymbol{k}}$ is zero if any of the components of $\boldsymbol{k}$ is negative. When $\boldsymbol{k} = \boldsymbol{0}$, the initial condition is given by

$$G_u^{\boldsymbol{0}} = \begin{cases} \dfrac{(\sqrt{2\nu} r)^u \mathsf{K}_u(\sqrt{2\nu} r)}{2^{u-1}\Gamma(u)}, & u > 0 \\ \dfrac{\mathsf{K}_0(\sqrt{2\nu} r)}{z(\sqrt{2\nu} r)}, & u = 0 \\ \dfrac{(\sqrt{2\nu} r)^{-u} \mathsf{K}_{-u}(\sqrt{2\nu} r)}{2^{-u-1}\Gamma(-u)}, & u < 0. \end{cases} \tag{8}$$

In addition, when the Taylor coefficients for the differentiated kernels $\phi^{[i]}$ are needed to be computed, the following formula is readily applicable:

$$\frac{\partial_{\boldsymbol{y}}^{\boldsymbol{k}} \phi^{[i]}(\boldsymbol{x}_c, \boldsymbol{y}_c)}{\boldsymbol{k}!} = \frac{1}{\ell_i} \left[(k_i + 1) r_i G_\nu^{\boldsymbol{k}+\boldsymbol{e}_i} - k_i G_\nu^{\boldsymbol{k}} \right].$$

The truncation of the Taylor expansion (3) incurs errors. Lacking an analytical bound, we pursue a data analysis approach and fit hypothesized error formulas for estimating the error. Denote by $\|\cdot\|_{\boldsymbol{\ell}}$ the elliptical 2-norm based on the scaling factor $\boldsymbol{\ell}$

(cf. (1)). When the Taylor expansion is truncated at only one of the centers, say $\boldsymbol{y}_c$, we define

$$\delta_p^{\max}(\rho,\tau) = \max_{\substack{\|\boldsymbol{x}_c-\boldsymbol{y}_c\|_{\boldsymbol{\ell}}\le\tau \\ \|\Delta\boldsymbol{y}\|_{\boldsymbol{\ell}}\le\rho}} \left| \phi(\boldsymbol{x}_c, \boldsymbol{y}_c+\Delta\boldsymbol{y}) - \sum_{\|\boldsymbol{k}\|=0}^{p} \frac{\partial_{\boldsymbol{y}}^{\boldsymbol{k}}\phi(\boldsymbol{x}_c,\boldsymbol{y}_c)}{\boldsymbol{k}!}(\Delta\boldsymbol{y})^{\boldsymbol{k}} \right|.$$

Because of symmetry, this definition is equivalent to the one with truncation at $\boldsymbol{x}_c$. An empirical error formula is

$$\log_{10}\delta_p^{\max}(\rho,\tau) = \alpha_1 + \alpha_2\log_{10}\tau + \alpha_3\log_{10}(\rho/\tau), \tag{9}$$

where α_1, α_2, and α_3 are coefficients that may vary with the truncation order p. This formula hypothesizes that in the log scale, the maximum error $\delta_p^{\max}(\rho,\tau)$ scales linearly with the distance τ of two centers and with the ratio ρ/τ between the expansion radius and the center distance. Further, the overall error when the Taylor expansion is truncated at both centers, as defined in

$$\delta_{p_1,p_2}^{\max}(\rho_t,\rho_s,\tau) = \max_{\substack{\|\boldsymbol{x}_c-\boldsymbol{y}_c\|_{\boldsymbol{\ell}}\le\tau \\ \|\Delta\boldsymbol{x}\|_{\boldsymbol{\ell}}\le\rho_t \\ \|\Delta\boldsymbol{y}\|_{\boldsymbol{\ell}}\le\rho_s}} \left| \phi(\boldsymbol{x}_c+\Delta\boldsymbol{x}, \boldsymbol{y}_c+\Delta\boldsymbol{y}) - \sum_{\|\boldsymbol{j}\|=0}^{p_1}\sum_{\|\boldsymbol{k}\|=0}^{p_2} \frac{\partial_{\boldsymbol{y}}^{\boldsymbol{j}+\boldsymbol{k}}\phi(\boldsymbol{x}_c,\boldsymbol{y}_c)}{(\boldsymbol{j}+\boldsymbol{k})!}(-\Delta\boldsymbol{x})^{\boldsymbol{j}}(\Delta\boldsymbol{y})^{\boldsymbol{k}} \right|,$$

is hypothesized to be the maximum of that of two single truncations:

$$\delta_{p_1,p_2}^{\max}(\rho_t,\rho_s,\tau) = \max\{\delta_{p_1}^{\max}(\rho_t,\tau+\rho_s),\ \delta_{p_2}^{\max}(\rho_s,\tau+\rho_t)\}. \tag{10}$$

The rationale of these two hypotheses (9) and (10) is discussed in details by Chen et al. [2012]. In robust software, one must verify them by fitting the coefficients α_1, α_2, and α_3 for p_1 and p_2 separately and quantifying the discrepancies in the fitting. If the absolute differences between both sides of (9) and (10) are less than 1.0 for all fitting samples, we conclude that these hypotheses are valid because the discrepancy between the estimated error and the true error is upper bounded by one order of magnitude.

It is known that when τ exceeds a soft threshold (approximately 1.0 to 5.0), the fitted formulas (9) and (10) largely overestimate the approximation error. From an algorithmic standpoint, this overestimation will affect the timing performance because potentially good approximations between two faraway clusters are conservatively ignored, but the software computes the correct results in any case.

2.2. Algorithmic Flow

The algorithm for the Matérn kernel follows a general tree code but entails several modifications. We start from the tree generation as it forms the foundation of later parallelization designs. The set of d-dimensional points is partitioned recursively to form the hierarchical tree structure. In particular, each partitioning is a bisection, meaning that a set X is separated in two balanced subsets X_l and X_r with $0 \le |X_l| - |X_r| \le 1$. The bisection is performed based on principal component analysis. Then, the recursive bisection forms a *full* binary tree, where each node represents a cluster of points, with the size of the clusters in the same level of the tree differing by at most 1. The root represents the whole set $\{\boldsymbol{x}_i\}$. See Figure 1. If n_0 is the maximum size of a leaf node, then the tree has $h = \lceil\log_2(n/n_0)\rceil + 1$ levels and $2^h - 1 = O(n/n_0)$ nodes. For the convenience of later descriptions, one can visualize that each leaf node contains a target cluster C_t and each tree node contains a source cluster C_s. The concepts of "node" and "cluster" are used interchangeably.

For any target $\boldsymbol{x}_i$ in a certain C_t, we initialize the ith entry of the matrix-vector product, s_i, with zero and perform a tree-walk starting at the root. For each node

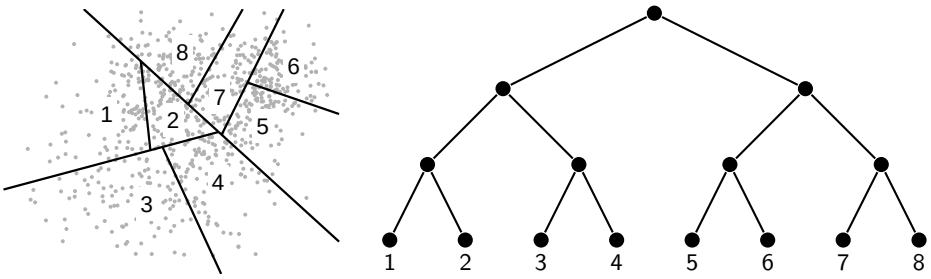


Fig. 1. Hierarchical partitioning of the point set and the *full* binary tree representation.

C_s being visited, we estimate the Taylor approximation error using (9) and (10). If the estimated error is less than a prescribed threshold ϵ, we compute the partial sum $s_i(C_s)$ using the Taylor approximation (5) and accumulate it to s_i. Otherwise, the two children of the node are visited. At the bottom of the recursion when a leaf node (still considered a source cluster C_s) is visited, if the Taylor approximation does not yield an error less than ϵ, we compute the partial sum $s_i(C_s)$ directly according to the definition (4) and accumulate it to s_i. Iterating through all the i's, we compute the overall vector $\boldsymbol{s}$.

3. PARALLEL PROCESSING

In a distributed-memory environment with MPI, the input point set and the resulting tree hierarchy are distributed across MPI processes. The recursive bisection scheme discussed in the preceding section induces a natural distribution for the data. In this section, we discuss the details of parallel processing and the algorithmic aspects incurred.

3.1. Data Distribution

Based on the bisection scheme, the set of n points is evenly distributed among p processes. Here, we require that p be a power of 2. In addition to the obvious reason that this requirement conforms to the full binary tree structure, a further benefit is that all the processes can be fully utilized in a parallel sorting (such as Batcher's bitonic sort [Batcher 1968]) in each bisection. We further restrict the number of points that each process holds to no less than the maximum leaf size, that is, $n/p \geq n_0$. This is equivalent to requiring that the depth of the tree be no less than the depth of the process hierarchy. Figure 2 illustrates the "process level" where there are p tree nodes. Each such node spans a "fan" (an abstraction of a binary subtree). At the bottom of these "fans" are leaf nodes that we visualize as target clusters C_t as usual. Thus, each process holds a partial tree, which consists of exactly one node on the "process level," all its ancestors, and the subtree rooted from this node. In this manner, the nodes above the process level will have a duplicate copy among several processes. Each process will also store the Taylor coefficients—the ones computed for a pair of nodes (C_t, C_s) when Taylor expansion is performed—for all C_t it contains. These coefficients are stored in a hash table that uses the pair (C_t, C_s) as the hashing key.

3.2. Load Balancing

Assume for the moment that a process P holds a target cluster C_t and that it wants to compute the results s_i for all $i \in C_t$. Based on the above data distribution, the computational pattern is that all the processes are looped over as partner processes. For each partner process P_j, P first asks for P_j's tree. Then, P walks through the obtained tree and determines at what source nodes C_s a Taylor expansion or a direct

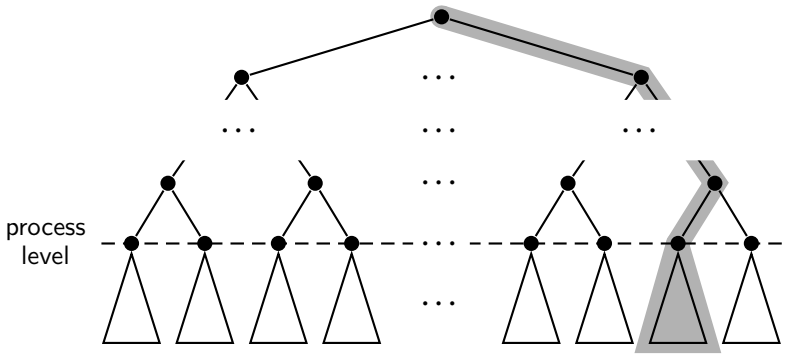


Fig. 2. Parallel distribution of points and tree. Shaded region represents part of the tree held by one process.

summation is used. Next, P asks for further data from P_j, some being weighted source moments (when Taylor expansion is performed) and some being weights and points (when direct summation is performed). As a side note, the weighted source moments have to be computed by P_j first, before any possible P asks for them. After the further data is obtained, P performs the summations and accumulates the results to s_i.

Clearly, if every process P computes the results s_i, $i \in C_t$, for all C_t it holds, the load may be imbalanced. The imbalance comes from two factors: the number of Taylor expansions and direct summations, and the communication cost for data transition. Thus, associating a leaf node C_t to a *unit task*, where s_i for all $i \in C_t$ are computed, we design a load-balancing manager that instructs the task assignments for each process. A direct consequence is that if a process P is assigned a task with C_t not held by P, P must first ask for C_t from some other process.

Before we discuss the algorithm of the task manager, we note a slight complication. When C_t is not held in P, P must also ask for the Taylor coefficients from the process that stores C_t. Since Taylor coefficients are stored in hash tables, the transfer of items is costly if the hash tables are not distributed. We resolve this problem by moving the Taylor coefficients in batch to the processes that actually use them, after the task assignment is made. This strategy causes a slight storage imbalance across processes; but since the task manager is designed to minimize the mismatch between C_t and P, the slight storage imbalance is generally not a problem. Additionally (as discussed in more details in Section 4), the movement of the Taylor coefficients, together with many other computations, are carried out as precomputations that are separate from the main computation, whose performance is the major interest.

3.2.1. Model and Its Impractical Solution. The task assignment is a classic integer program. Denote by an index i a unit task, which ranges from 1 to the number of target nodes—the smallest power of 2 larger than or equal to n/n_0. Denote by j the processes, ranging from 1 to p. We assume that initially the target cluster associated to the task i resides in process $p(i)$. Next, define d_{ij} to be the penalty

$$d_{ij} = \begin{cases} \lambda, & \text{if } j \neq p(i) \\ 0, & \text{otherwise,} \end{cases}$$

where $\lambda > 0$ is a tunable penalty parameter, x_{ij} to be the indicator variable that indicates whether task i is assigned to process j, and c_i to be the cost of completing task i.

Then, the task assignment problem is

$$\begin{aligned} \min \quad & z \\ \text{s.t.} \quad & \sum_i c_i x_{ij} + \sum_i d_{ij} x_{ij} \le z, \quad \forall\, j \\ & \sum_j x_{ij} = 1, \quad \forall\, i \\ & x_{ij} \in \{0,1\}, \quad \forall\, i, \end{aligned} \tag{11}$$

where z is just an auxiliary variable for rewriting the minimax nature of the problem in the standard integer programming form. The first summation term of (11) is the total cost for completing the tasks assigned to process j, whereas the second term is the penalty of assigning foreign tasks to j. We want to minimize the sum of cost and penalty across all processes.

Unfortunately, the total number of variables is so large—not smaller than pn/n_0 (e.g., $p = 128$, $n = 1024^2$, $n_0 = 64$)— that this integer program is unlikely to be solved in a reasonable time with the most efficient optimization solvers currently available, such as CPLEX.[3] Further research might be carried out to design a more sophisticated optimization scheme, but we adopt a simple heuristic that works well in practice.

3.2.2. Heuristic Solution. Assume for the moment that the penalty term is not included. The objective can be adjusted to that of minimizing the variation of $\sum_i c_i x_{ij}$, $j = 1, \ldots, p$, away from the constant mean $\sum_i c_i/p$. One way of achieving this approximately, is to first sort the tasks initially held in each process j in decreasing order and assign as many tasks as possible to j, as long as the accumulated cost has not exceeded $\sum_i c_i/p$. This assignment does not incur any penalty, and all processes can compute the assignment in parallel without any communication. After this pass, a set A of processes is assigned with all its tasks and still has not reached a total of $\sum_i c_i/p$ cost; whereas the remaining set B of processes can be assigned with only part of its tasks and the other tasks have to be moved out.

In the next pass the tasks are moved from the set B of processes to the set A. The task information, such as the cost c_i, has to be gathered by every process, each performing duplicate assignment calculations concurrently. The algorithm of reassigning these tasks is greedy. We maintain a max-heap of all the processes, where the heap keys are the balance of a process, defined as the mean $\sum_i c_i/p$ minus the total cost of tasks it has accumulated so far. We then iterate all the tasks in the queue (sorted in the decreasing order of their costs), assign each task to the process at the root of the heap, and adjust the heap.

This algorithm works well in practice because the number of tasks in the second phase is not too large and these tasks have relatively small costs. (An extreme situation is that some particular task(s) has a cost that already exceeds the mean.) These tasks, since they stand in the front of the sorted order in the first pass, are assigned to the process immediately; the rest of the tasks this process holds enter the second pass. In this case, load balancing is impossible to achieve. In Figure 3, we show that a data structure `LoadStat` is returned after the load-balancing manager computes the task assignment. This data structure is used to inspect the load-balancing statistics.

3.2.3. Simulation of Costs. The cost for a task is estimated based on the following formula:

$$c_i = n_{\text{expand}}(i) \cdot [c_{\text{expand,comp}} + c_{\text{expand,comm}}] + n_{\text{direct}}(i) \cdot [c_{\text{direct,comp}} + c_{\text{direct,comm}}],$$

[3] `http://en.wikipedia.org/wiki/CPLEX`

where $n_{\text{expand}}(i)$ and $n_{\text{direct}}(i)$ are the number of node pairs (fixing the target node i) for Taylor expansions and for direction summations, respectively; $c_{\text{expand,comp}}$ and $c_{\text{expand,comm}}$ are the computation and communication cost for one Taylor expansion, respectively; and similarly for $c_{\text{direct,comp}}$ and $c_{\text{direct,comm}}$. The computation costs, $c_{\text{expand,comp}}$ and $c_{\text{direct,comp}}$, can be straightforwardly simulated. The communication costs, $c_{\text{expand,comm}}$ and $c_{\text{direct,comm}}$, on the other hand, may fluctuate significantly depending on many factors, including the machine architecture and the MPI communication protocol. For example, in the next subsection, we explain our use of the passive one-sided communication protocol. A good simulation of the communication cost happens only when each process uses a separate core for thread communications. In light of this complication in performance tuning, we leave a switch in the code that can be set by the user to indicate whether communication costs are simulated. See the argument `est_comm` in the interface routine `Planning` in Figure 3 and Section 4.

3.3. Passive One-Sided Communications

Augmenting Section 2.2 by parallel processing, we can (incompletely) summarize the computational flow in the following, omitting the precomputations and treating a process executing the flow as the first person:

1: Compute weighted source moments for all the source nodes C_s I have.
2: **for** every set of tasks involving target nodes $\{C_t\}$ held in the same process P_i **do**
3: If I am not P_i, get P_i's tree and the points belonging to the target nodes.
4: **for** every partner process P_j **do**
5: If I am not P_j, get P_j's tree.
6: For each target node C_t, walk P_j's tree, mark the places that require additional data (weighted source moments, points, and weights) from P_j.
7: If I am not P_j, get the required data from P_j.
8: Using the obtained data, compute Taylor expansions and direct summations to accumulate s_i for all $i \in C_t$ and all target nodes C_t.
9: **end for**
10: If I am not P_i, write back the computing results to P_i.
11: **end for**

As is a usual practice, processes P_i and P_j in the two for-loops are permuted to avoid potential congested data requests to the same process.

The data movement occurs irregularly in the above flow. A reason is that the computations with respect to each P_j in the inner loop have different costs. Thus, it is unwise to place MPI send and receive calls (even using the nonblocking versions) for communications in the nested for loops.

This work-flow pattern is best handled by using passive one-sided communications (also known as *remote memory access*), introduced in the MPI-2 standard. In one-sided communications, the MPI calls (e.g., `MPI_Put` and `MPI_Get`) need not be matched between the sending and the receiving parties. Each process opens a memory window containing data that can be accessed by other processes in a manner similar to direct memory addressing. In the active mode,[4] all the processes are explicitly synchronized by using a pair of fences within which a process can remotely access another one's memory window, or part of the processes are synchronized by the post/start/complete/wait mechanism. In fact, the active mode is not truly one-sided to a programmer's eyes because explicit synchronization calls still need to be placed. Putting these

[4]We avoid using the term "target" in the standard MPI terminology (e.g., "active target synchronization" and "target process") and reserve the term for the explanation of the tree code (e.g., "target node" and "target cluster").

synchronization codes anywhere will cause idling of processes finishing the previous computations faster.

The use of the one-sided communications in our program is the passive mode, where no explicit synchronization calls need to be made. When a process wants to access the memory window of another one, it issues a lock before accessing it and releases the lock on completion. The synchronization is done implicitly by the MPI library. A shortcoming of the passive mode is that its best performance achievable by current MPI implementations requires a spare core for handling the thread communications for each process.

4. INTERFACE

We implemented the details discussed above in a C++ class `TreeCodeMatern3D`, which specializes the $\mathbb{R}^3$ case. The tree code algorithm, in principle, works for all practical dimensions, but templating the class over the dimension significantly complicates the coding in many fine details.

The interface of `TreeCodeMatern3D` consists of two sets of public functions: the computational routines and the utility routines. Our design principle is that the computational routines are as neat as possible whereas the utility routines are rich in functionality for users to investigate the various performance statistics.

Whereas one wishes a single function call for the conceptually simple matrix-vector multiplication, it is unlikely because of the complications of the algorithm. For example, precomputations are needed to separate out from the main calculations when multiple vectors $\boldsymbol{q}$ exist. There are more complications as will soon be seen. Thus, we design the computational interface that contains the following routines that must be called in order.

— `TreeCodeMatern3D`: The constructor. It takes as input a distributed array of points and the number of points in this process. The set of points cannot be changed once the class is instantiated.
— `Planning`: The planning routine. It performs the precomputations given the various parameters, including those associated with the Matérn kernel (ν and ℓ), those related to the tree code (p_1, p_2, ϵ and n_0), one that indicates whether the computation includes the differentiated kernels, and one that indicates whether to estimate communication cost in the load-balancing manager. The planning routine performs many calculations: a binomial table is constructed; the error formulas (9) and (10) are fitted; the tree hierarchy is built with points redistributed; Taylor coefficients are computed; the summation cost is simulated; the load balancing manager is invoked; the Taylor coefficients associated with the target nodes are moved to the computing process and stored there; and the reordering information of the points are recorded. The routine returns the indication of whether the fitting of error formulas is successful. This routine can be re-called every time the parameters are changed.
— `Evaluation`: The evaluation routine. This routine performs the actual matrix-vector multiplication. It takes the vector $\boldsymbol{q}$ as input and $\boldsymbol{s}$ as output, and it can be called as many times as desired for multiple $\boldsymbol{q}$'s.
— `CleanupBeforeMPIFinialize`. This routine frees the allocated data structures for several derived MPI datatypes used for data movements. A lack of destructors in the MPI implementation requires such a non-object-oriented-friendly workaround that is otherwise unnecessary.

A caveat in the `Evaluation` routine is that the points are redistributed in the planning phase; thus the ordering of the entries of $\boldsymbol{q}$ (and $\boldsymbol{s}$) might mismatch that of the points after the tree is built. The software could of course implicitly reorder $\boldsymbol{q}$ and $\boldsymbol{s}$, but making the reordering explicit to the user is also justified. Consider, for example, a

Krylov iteration with respect to the matrix Φ. Because the vectors are repeatedly multiplied to Φ and the ordering of the vector entries during the iterations is not relevant, none of the intermediate vectors needs to maintain the correct order for their entries. In other words, reordering needs to be done only before and after the Krylov iteration. Because of this flexibility, we introduce the following two reordering routines.

— `Reorder_InputToTree`: Redistribute the entries (of the input $\boldsymbol{q}$) to match the ordering of the points after the tree is built.
— `Reorder_TreeToInput`: Redistribute the entries (of the computed $\boldsymbol{s}$) to match the original ordering of the input points.

Then, the `Evaluation` routine asks the user to specify whether the redistribution is performed internally or externally. If externally, the user is responsible calling the above reordering routines to ensure the correctness of the computation.

The utility interface contains the following routines.

— `GetNumPtsThisProc`: Return the number of points a process has after the tree is built. This can be used for allocating memory for input/output vectors.
— `GetPlanningTimeStat`: Return the timing statistics of the planning phase.
— `GetEvaluationTimeStat`: Return the timing statistics of the evaluation phase.
— `GetSummationStat`: Return the statistics of the number of direct summations and the number of Taylor approximations. This can be used to verify the $O(n \log n)$ computational complexity as an alternative criterion to wall-clock timing.
— `GetLoadStat`: Return the computational load for each process. This can be used to inspect the load balancing as an alternative criterion to wall-clock timing.

Figure 3 demonstrates a barebone example for using the `TreeCodeMatern3D` interface. The dummy for-loops show the recurrent uses of the `Planning` routine and the `Evaluation` routine when parameters and the input vector change, respectively.

To verify the correctness of the tree code calculation, we implemented a separate class `DirectMatern3D` that computes the matrix-vector product in the straightforward $O(n^2)$ manner. The interface of `DirectMatern3D` is similar to that of `TreeCodeMatern3D` except that no planning is carried out and the points are not redistributed. The interface contains a routine `RelativeErr` that checks the relative difference between two vectors.

5. DATA STRUCTURE FOR MULTIDIMENSIONAL PARTIAL DATA

A straightforward data structure for storing a set of Taylor coefficients $\{G_\nu^{\boldsymbol{k}}\}$ is a d-dimensional array A, as in $A[k_1][k_2]\cdots[k_d] = G_\nu^{\boldsymbol{k}}$. Since the indices have a sum that is bounded by $0 \le k_1 + k_2 + \cdots + k_d \le m$, where $m = p_1 + p_2$ with p_1 and p_2 being the truncation orders, such an array A has a size $(m+1)^d$, of which only $\binom{m+d}{d}$ of them are used. In other words, asymptotically the usage rate is only $1/d!$. Considering that there is a large number of such sets, the storage is significantly wasted.

A more economic storage is a one-dimensional array with exactly $\binom{m+d}{d}$ entries. We thus need an efficient indexing method to position an item $G_\nu^{\boldsymbol{k}}$ with a random $\boldsymbol{k}$. In other words, we seek a one-to-one mapping $f : (k_1, k_2, \ldots, k_d) \mapsto i$ that maps a d-tuple index to a linear index. The mapping in fact serves as an ordering of the d-element integer vectors $\{\boldsymbol{k}\}$, where $0 \le \|\boldsymbol{k}\| \le m$.

A descriptive definition of the ordering is as follows. We first order the tuples in batch according to their sums; that is, $(k_1, k_2, \ldots, k_d)$ is less than $(k'_1, k'_2, \ldots, k'_d)$ if $k_1 + k_2 + \cdots + k_d < k'_1 + k'_2 + \cdots + k'_d$. Then, the tuples with the same sum are ordered lexicographically; that is, $(k_1, k_2, \ldots, k_d)$ is less than $(k'_1, k'_2, \ldots, k'_d)$ if $k_i \le k'_i$ for all i. Algorithmically, the following pseudocode outputs all the d-tuples in this order.

```
long n;         // Total number of points
Point3D *X;     // An array of points (distributed)
double nu;      // Matern order
double ell[3];  // Scaling factors
int p[2];       // Expansion orders
double epsilon; // Error threshold
long n0;        // Maximum number of points a leaf holds
bool kernel[3]; // Compute differentiated kernel wrt the i-th parameter?
bool est_comm;  // Estimate communication cost when planning?
double *q;      // Input vector (distributed)
double *s[4];   // Output vectors (distributed).
                // s[0]: vector for original kernel;
                // s[1] to s[3]: vectors for differentiated kernels

// Set n, X here...

TreeCodeMatern3D TreeCode(n, X, MPI_COMM_WORLD);

for (int i = 0; i < 10; i++) { // a dummy loop

  // Set or change nu, ell, p, epsilon, n0, kernel, est_comm here...

  if (TreeCode.Planning(nu,ell,p,epsilon,n0,kernel,est_comm) == false) {
    std::cout << "Planning failed!";
    exit(1);
  }
  long *sum_stat = NULL;
  int len = TreeCode.GetSummationStat(&sum_stat);
  LoadStat load = TreeCode.GetLoadStat();
  TimeStat t = TreeCode.GetPlanningTimeStat();

  for (int j = 0; j < 10; j++) { // a dummy loop

    // Set or change q here; allocate the right length for s...
    // If s is not reordered, length is TreeCode.GetNumPtsThisProc()

    bool q_reorder = true;
    bool s_reorder = true;
    TreeCode.Evaluation(q, q_reorder, s, s_reorder);
    t = TreeCode.GetEvaluationTimeStat();

    // Or, equivalently...
    // TreeCode.Reorder_InputToTree(&q);
    // TreeCode.Evaluation(q, false, s, false);
    // for (int k = 0; k < 4; k++) {
    //   TreeCode.Reorder_TreeToInput(&s[i]);
    // }

  }

}

TreeCode.CleanupBeforeMPIFinialize();
```

Fig. 3. Example use of the `TreeCodeMatern3D` class.

```
for s ← 0 to m do
    for k_1 ← 0 to s do
        for k_2 ← 0 to s − k_1 do
            ···
            for k_{d−1} ← 0 to s − k_1 − ··· − k_{d−2} do
                Print (k_1, k_2, ..., k_{d−1}, s − k_1 − k_2 − ··· − k_{d−1})
            end for
            ···
        end for
    end for
end for
```

In the appendix, we show that an equivalent mathematical definition of the mapping f is

$$f(k_1, k_2, \ldots, k_d) = \binom{t+d}{d} - \sum_{j=1}^{d-2} \binom{t-(k_1+\cdots+k_j)+d-j-1}{d-j} - k_d - 1,$$

where $t = k_1 + k_2 + \cdots + k_d$. Here, we enforce that $f(0, 0, \ldots, 0) = 0$.

Efficient ways to evaluate the mapping exist. For example, when $d = 3$, we have

$$f(k_1, k_2, k_3) = \binom{t+3}{3} - \binom{t-k_1+1}{2} - k_3 - 1.$$

In computer implementation, we hard-code two arrays b_2 and b_3, with elements $b_2[j] = \binom{j}{2}$, $b_3[j] = \binom{j}{3}$. Then,

```
k'_3 = k_3 + 1;
t' = k_2 + k'_3;
t'' = t' + k_1 + 2;
i = b_3[t''] − b_2[t'] − k'_3.
```

This calculation requires in total six adds/subtracts and two array-lookups.

Figure 4 shows the class `PartialArray3D` that implements such a data structure. It is a templated class with identifier `T`. For our usage `T` is `double`. The value m cannot exceed some predefined maximum `MAX_M`. The operations `Set`, `AddTo`, and `Get` with clear literal meanings all come with two interfaces: one uses the 3-tuple index (k_1, k_2, k_3) to address an element, and the other uses the linear index i. The conversion from a 3-tuple index to a linear index is performed in `Idx3to1`. For convenience, the class overloads the square bracket operator so that the operations with respect to a single element of the array (such as `Set`, `AddTo` and `Get`), using the linear index, can be written in a simpler fashion. The class also overloads the operators such as `=`, `+` and `+=` to perform operations for all the elements in the array.

The two indexing schemes have separate uses. The 3-tuple indexing is used when the array elements are not accessed consecutively. For example, in the recurrence (7) for computing Taylor coefficients, the (k_1, k_2, k_3) element of an array is computed by accessing the $(k_1 - 1, k_2, k_3)$, $(k_1 - 2, k_2, k_3)$, $(k_1, k_2 - 1, k_3)$, $(k_1, k_2 - 2, k_3)$, $(k_1, k_2, k_3 - 1)$, $(k_1, k_2, k_3 - 2)$ elements of another array. In this case, the interfaces with 3-tuple indexing are convenient. On the other hand, in some situations (e.g., performing a summation with respect to $\boldsymbol{k}$ for all $\|\boldsymbol{k}\| \leq m$) the array elements can be accessed consecutively. In this case, using the interfaces with a 3-tuple index incurs overheads because of the index conversion; hence, linear indexing is more favorable.

```
#define MAX_M 50
static const int b2[52] = {0, 0, 1, 3, 6, 10, 15, 21, 28, 36, 45, 55, 66,
    78, 91, 105, 120, 136, 153, 171, 190, 210, 231, 253, 276, 300, 325,
    351, 378, 406, 435, 465, 496, 528, 561, 595, 630, 666, 703, 741, 780,
    820, 861, 903, 946, 990, 1035, 1081, 1128, 1176, 1225, 1275};
static const int b3[54] = { /* omitting contents here */ };

template <class T> class PartialArray3D {

 public:

  // Constructors and destructor
  PartialArray3D();
  PartialArray3D(const PartialArray3D<T>& x);
  PartialArray3D(int m_);
  ~PartialArray3D();

  // Overloaded operators
  T& operator[](int i);
  PartialArray3D<T>& operator = (const PartialArray3D<T>& x);
  PartialArray3D<T>& operator = (const T& x);
  PartialArray3D<T>& operator += (const PartialArray3D<T>& x);
  PartialArray3D<T>& operator += (const T& x);
  PartialArray3D<T> operator + (const PartialArray3D<T>& x) const;

  // Set the i-th element with x
  void Set(int k1, int k2, int k3, const T& x);
  void Set(int i, const T& x); // Same as using the [] operator

  // Add to the i-th element with x
  void AddTo(int k1, int k2, int k3, const T& x);
  void AddTo(int i, const T& x); // Same as using the [] operator

  // Get the i-th element
  T& Get(int k1, int k2, int k3);
  T& Get(int i); // Same as using the [] operator

  // Mapping from 3-tuple index to linear index
  int Idx3to1(int k1, int k2, int k3) {
    int k3p = k3 + 1;
    int tp = k2 + k3p;
    return b3[tp + k1 + 2] - b2[tp] - k3p;
  }

  // More operations

 private:

  T *B;     // The array
  int m;    // k1+k2+k3 <= m
  int len;  // Length of B (should be (m+3)*(m+2)*(m+1)/6)

};
```

Fig. 4. Definition of the `PartialArray3D` class and implementation of the index mapping.

6. EXPERIMENTAL EVALUATION

Partial experimental results regarding the correctness of the program (verified against straightforward summation) and the serial $O(n \log n)$ complexity of the evaluation cost have been shown by Chen et al. [2012]. This section focuses on the effectiveness of the parallelization techniques discussed in Section 3. All the experiments were conducted on the Blues cluster[5] of the Laboratory Computing Resource Center at Argonne National Laboratory. The cluster contains 310 compute nodes, each of which is equipped with 16 Intel Sandy Bridge processors (with hyperthreading disabled) and 64GB of RAM. The nodes are connected through QLogic QDR InfiniBand. The MPI compiler we used was MVAPICH2 built with ICC. All the experimental results were retrieved from the utility interface described in Section 4.

The experimental setup is the following. We ran experiments with a series number of processes and problem sizes, where we always launched 8 MPI processes on each compute node. Four point set configurations were experimented with: (i) uniform distribution in the unit cube, (ii) points on a sphere of unit radius with uniformly random azimuthal angle and polar angle, (iii) 30°N to 60°N segment of this sphere, and (iv) a mixture of four Gaussians with different centers, weights, and scales. Figure 5 shows an illustration of the Gaussian mixture. All these configurations are provided in the test driver accompanied with the released code. Parameters were $\nu = 1.5$, $\boldsymbol{\ell} = [4, 14, 3]$, $p_1 = 4$, $p_2 = 6$, $\epsilon = 10^{-7}$, and $n_0 = 64$. The switch `est_comm` was disabled. Experimental results are reported and compared for the cube and the Gaussian mixture cases, wherein the former produces the most homogeneous points set and the latter heterogeneous.

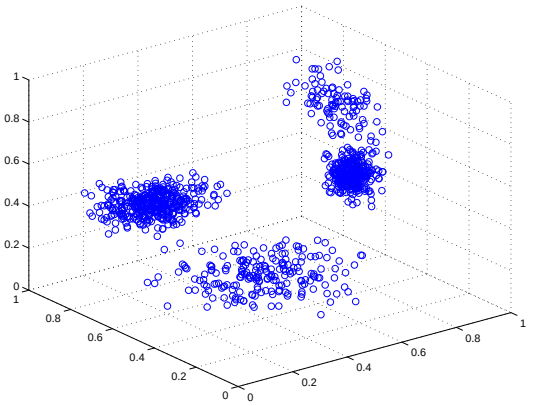


Fig. 5. illustration of the Gaussian mixture.

To demonstrate the effectiveness of the load-balancing manager, we list in Table I the maximum workload across all processes with respect to the average. The perfect case is 100%. Accompanied with it is the cost of the heaviest task normalized by the average process workload, listed inside the parentheses. When the task costs are even, this number should be tiny; a large number indicates unevenness. One sees that in the cube case, task costs are even and load balance is perfect. In the Gaussian mixture case, however, some tasks have exceedingly high cost (see the last column of table). In this case, perfect load balance is impossible to achieve, but our task manager gives the best achievable load balance.

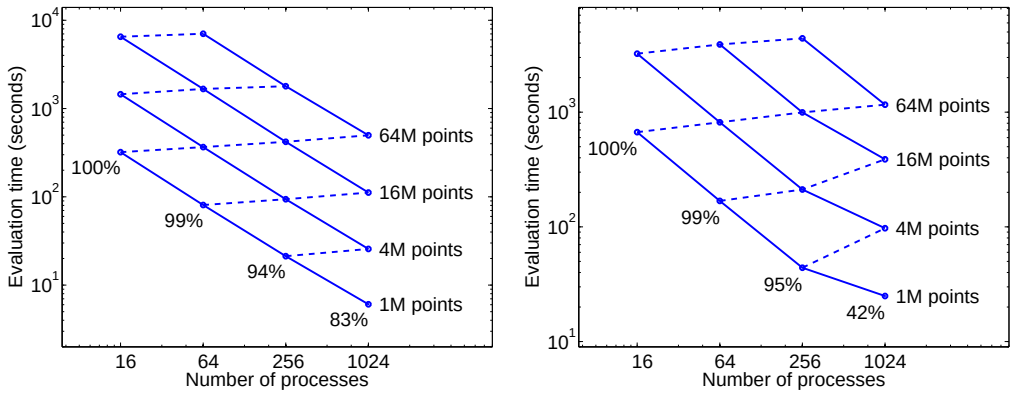
[5] `http://www.lcrc.anl.gov/`

Table I. Maximum workload normalized by the average workload. Inside the parentheses is the cost of the heaviest task normalized by the average workload. All numbers are rounded to 1%. The top table is for cube and the bottom one for Gaussian mixture. The cases marked by "NA" indicate that the required memory has exceeded the machine capacity.

n ($\times 1024^2$)	16	64	256	1024
1	100% (0%)	100% (1%)	101% (2%)	102% (9%)
4	100% (0%)	100% (0%)	100% (1%)	101% (2%)
16	100% (0%)	100% (0%)	100% (0%)	100% (1%)
64	NA	100% (0%)	100% (0%)	100% (0%)

n ($\times 1024^2$)	16	64	256	1024
1	100% (4%)	100% (15%)	104% (58%)	235% (235%)
4	100% (2%)	100% (12%)	100% (48%)	193% (193%)
16	NA	100% (10%)	100% (41%)	163% (163%)
64	NA	NA	100% (22%)	100% (90%)

For timing comparison, Figure 6 plots the strong (solid) and weak (dashed) scalings of the evaluation time together with parallel efficiencies for the smallest problem size (the planning time is typically in several seconds to a few minutes and is unimportant). One sees a good scaling for the cube case and part of the Gaussian mixture case. This result is a consequence of the combined effect of the near-linear complexity of the tree code and the balance of the workload among processes. The part of the Gaussian mixture case that deviates from the perfect strong scaling trend is not surprising. It is caused by the imperfect load balance (see the last column of Table I), which in turn is a result of highly inhomogeneous distribution of the point set.

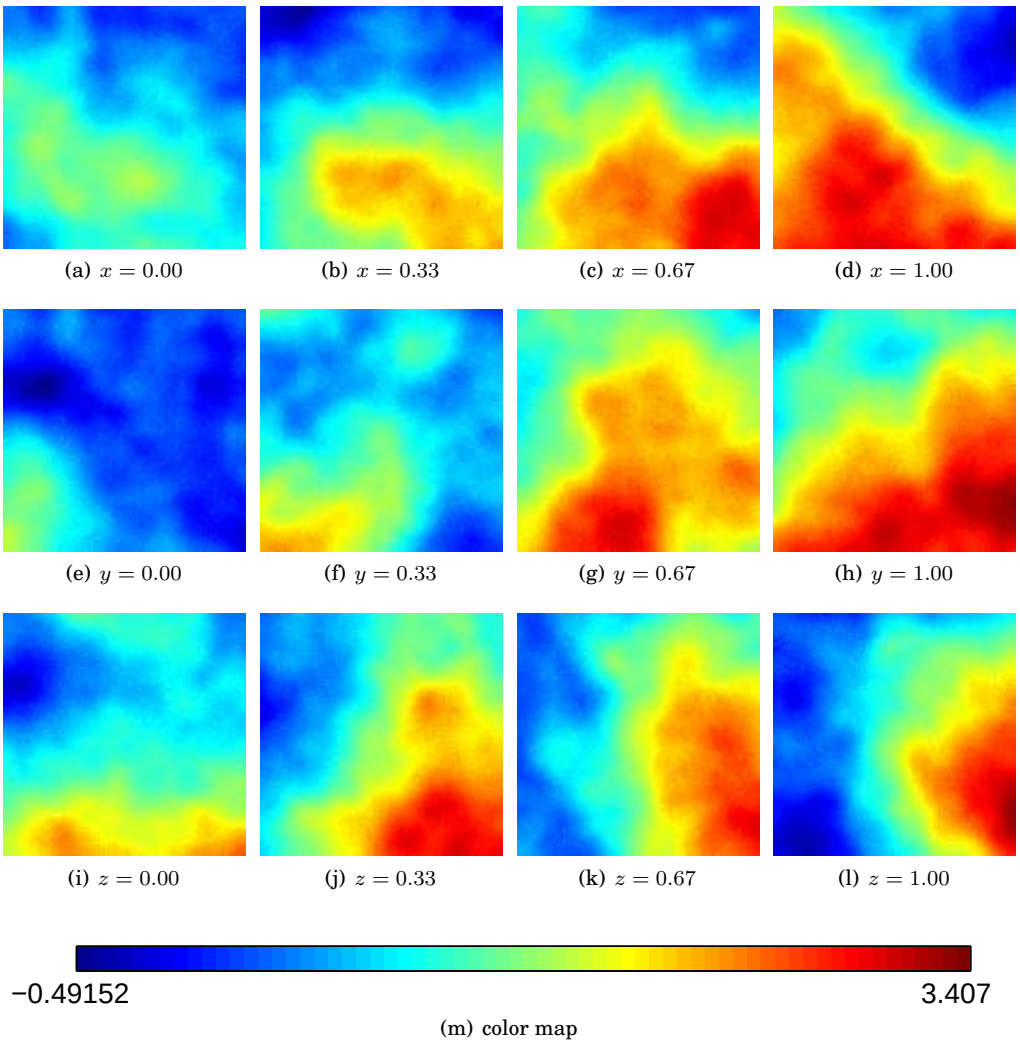


(a) Cube (b) Gaussian mixture

Fig. 6. Scalings of the evaluation time. Solid line: strong scaling; dashed line: weak scaling. Value under the bottom curve: parallel efficiency with respect to strong scaling.

As an application, we used the tree code to build a Gaussian sampling code that generates random Gaussian process data. The traditional method for generating Gaussian data is via a Cholesky factorization of the covariance matrix Φ; however, the cubic time cost and quadratic storage cost make the method extremely difficult to scale beyond $O(10^5)$ data points, even with the use of a supercomputer and the most efficient packages for large-scale linear algebra computations, such as ScaLAPACK [Blackford et al. 1997]. Thus, we have proposed a matrix-free sampling method [Chen et al. 2011] that is based on matrix-vector multiplications and avoids matrix factorizations. Here, we demonstrate the use of the tree code to sample a three-dimensional Matérn process

with smoothness $\nu = 1.0$ and scales $\boldsymbol{\ell} = [1, 1, 1]$. We set $\epsilon =$ `1e-5` and, to prevent the possible loss of positive definiteness, added a nugget `1e-5` to the kernel. Figure 7 shows several slices of a computed random sample in the unit cube on a $100 \times 100 \times 100$ grid. Clearly, the tree code is designed not only for regularly gridded data; however, for visualization purpose we give here a regular grid result. Readers can view an animation of the sample in `http://press3.mcs.anl.gov/scala-gauss/gallery/`.



(a) $x = 0.00$ (b) $x = 0.33$ (c) $x = 0.67$ (d) $x = 1.00$

(e) $y = 0.00$ (f) $y = 0.33$ (g) $y = 0.67$ (h) $y = 1.00$

(i) $z = 0.00$ (j) $z = 0.33$ (k) $z = 0.67$ (l) $z = 1.00$

(m) color map

Fig. 7. Matérn Gaussian sample in the unit cube. $\nu = 1.0$, $\boldsymbol{\ell} = [1, 1, 1]$, nugget = `1e-5`.

7. CONCLUSIONS

We have described the design and parallel implementation of a recently proposed Matérn tree code for computing matrix-vector products. The parallelization with a load-balancing design and MPI passive one-sided communications achieves good work

balance and scaling. The software is designed with an interface focusing on flexibility and functionality expressiveness. It is intended to be used as a library on top of which statistical application codes are built, such as sampling, as demonstrated in the preceding section.

APPENDIX

LEMMA A.1. *The integer equation*

$$k_1 + k_2 + \cdots + k_d = t, \quad \forall\, k_i \geq 0$$

has $\binom{t+d-1}{d-1}$ *solutions.*

PROOF. The number of solutions is equivalent to the number of combinations for using $d-1$ bars to separate a row of t oranges. □

LEMMA A.2. *The integer equation*

$$k_1 + k_2 + \cdots + k_d \leq t, \quad \forall\, k_i \geq 0$$

has $\binom{t+d}{d}$ *solutions.*

PROOF. The number of solutions is equivalent to the number of combinations for using d bars to separate a row of t oranges. □

LEMMA A.3. *The integer equation*

$$k_1 + k_2 + \cdots + k_d = t, \quad \forall\, k_i \geq 0,\ k_1 < s$$

has $\binom{t+d-1}{d-1} - \binom{t-s+d-1}{d-1}$ *solutions.*

PROOF. The number of solutions for the integer equation

$$k_1 + k_2 + \cdots + k_d = t, \quad \forall\, k_i \geq 0,\ k_1 \geq s$$

is $\binom{t-s+d-1}{d-1}$, immediately following Lemma A.2. Without the constraint $k_1 \geq s$, the number of solutions is $\binom{t+d-1}{d-1}$, according to Lemma A.1. □

PROPOSITION A.4. *The position of the* d*-tuple* $(k_1, k_2, \ldots, k_d)$ *in the output of the* d*-level for-loop in Section 5 is*

$$\binom{t+d}{d} - \sum_{j=1}^{d-2} \binom{t-(k_1+\cdots+k_j)+d-j-1}{d-j} - k_d - 1, \tag{12}$$

where $k_1 + \cdots + k_d = t$*. Here, we consider that the position of* $(0, 0, \ldots, 0)$ *is* 0*.*

PROOF. According to Lemma A.2, the number of d-tuples $(k'_1, \ldots, k'_d)$ for which $k'_1 + \cdots + k'_d < t$ is $\binom{t-1+d}{d}$. Next, the number of tuples for which $k'_1 + \cdots + k'_d = t$ but $k'_1 < k_1$ is $\binom{t+d-1}{d-1} - \binom{t-k_1+d-1}{d-1}$, according to Lemma A.3. Iteratively, the number of tuples for which $k'_1 + \cdots + k'_d = t$, $k'_1 = k_1, \ldots, k'_{d'-1} = k_{d'-1}$ and $k'_{d'} < k_{d'}$ is $\binom{t-(k_1+\cdots+k_{d'-1})+d-d'}{d-d'} - \binom{t-(k_1+\cdots+k_{d'})+d-d'}{d-d'}$. Hence, the number of tuples before $(k_1, \cdots, k_{d-1}, k_d)$ is

$$\binom{t-1+d}{d} + \sum_{d'=1}^{d-1} \left[\binom{t-(k_1+\cdots+k_{d'-1})+d-d'}{d-d'} - \binom{t-(k_1+\cdots+k_{d'})+d-d'}{d-d'} \right].$$

Combining every two terms of the above expression when the summation is expanded (similar to telescoping except that the telescoping terms do not cancel), we reach the expression (12). □

ACKNOWLEDGMENTS

We gratefully acknowledge the use of Fusion cluster and Blues cluster in the Laboratory Computing Resource Center at Argonne National Laboratory. This work was supported by the U.S. Department of Energy under Contract DE-AC02-06CH11357.

REFERENCES

ANITESCU, M., CHEN, J., AND WANG, L. 2012. A matrix-free approach for solving the parametric Gaussian process maximum likelihood problem. *SIAM J. Sci. Comput. 34,* 1, A240–A262.

BATCHER, K. E. 1968. Sorting networks and their applications. In *Proceedings of the AFIPS spring joint computer conference*. 307–314.

BLACKFORD, L. S., CHOI, J., CLEARY, A., D'AZEVEDO, E., DEMMEL, J., DHILLON, I., DONGARRA, J., HAMMARLING, S., HENRY, G., PETITET, A., STANLEY, K., WALKER, D., AND WHALEY, R. C. 1997. *ScaLAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA.

CHEN, J., ANITESCU, M., AND SAAD, Y. 2011. Computing $f(A)b$ via least squares polynomial approximations. *SIAM J. Sci. Comput. 33,* 1, 195–222.

CHEN, J., WANG, L., AND ANITESCU, M. 2012. A fast summation tree code for Matérn kernel. Tech. Rep. ANL/MCS-P4001-1212, Argonne National Laboratory. Under revision in SIAM Journal on Scientific Computing.

CHILÈS, J.-P. AND DELFINER, P. 1999. *Geostatistics: Modeling Spatial Uncertainty*. Wiley-Interscience.

MESSAGE PASSING INTERFACE FORUM. 2003. *MPI-2: Extensions to the Message-Passing Interface*. `http://www.mpi-forum.org/docs/mpi2-report.pdf`.

RASMUSSEN, C. AND WILLIAMS, C. 2006. *Gaussian Processes for Machine Learning*. MIT Press.

STEIN, M. 1999. *Interpolation of Spatial Data: Some theory for Kriging*. Springer-Verlag.

STEIN, M. L., CHEN, J., AND ANITESCU, M. 2013. Stochastic approximation of score functions for Gaussian processes. *Annals of Applied Statistics 7,* 2, 1162–1191.

WENDLAND, H. 2005. *Scattered Data Approximation*. Cambridge University Press.

YING, L., BIROS, G., ZORIN, D., AND LANGSTON, H. 2003. A new parallel kernel-independent fast multipole method. In *Proceedings of the 2003 ACM/IEEE conference on Supercomputing*.