A Framework to Exploit Data Sparsity in Tile Low-Rank Cholesky Factorization

Qinglei Cao[1,5], Rabab Alomairy[2,7], Yu Pei[1,5], George Bosilca[1,6], Hatem Ltaief[2,7], David Keyes[2,7], and Jack Dongarra[1,3,4,6]

[1]Innovative Computing Laboratory, University of Tennessee, US
[2]Extreme Computing Research Center, Division of Computer, Electrical, and Mathematical Sciences and Engineering, King Abdullah University of Science and Technology, KSA
[3]the Oak Ridge National Laboratory, US
[4]University of Manchester, UK
[5]*{qcao3, ypei2}@vols.utk.edu*
[6]*{bosilca, dongarra}@icl.utk.edu*
[7]*{rabab.omairy, hatem.ltaief, david.keyes}@kaust.edu.sa*

***Abstract*—We present a *general* framework that couples the `PaRSEC` runtime system and the `HiCMA` numerical library to solve challenging 3D data-sparse problems. Though formally dense, many matrix operators possess a rank structured property that can be exploited during the most time-consuming computational phase, i.e., the matrix factorization. In particular, this work highlights how a software bundle powered by a task-based programming model can address the heterogeneous workloads engendered by compressing the dense operator. Using Tile Low-Rank (`TLR`) approximation, our approach consists in capturing the most significant information in each tile of the matrix using a threshold which satisfies the application's accuracy requirements. Matrix operations are performed on the compressed data layout, reducing memory footprint and algorithmic complexity. Our proposed software solution accommodates a range of traditional data structures of linear algebra, i.e., from dense and data-sparse to sparse, within a single matrix operation. Separation of concerns is at the heart: hardware-agnostic implementation, asynchronous execution with a dynamic runtime system, and high performance numerical kernels, to prepare scientific applications to embrace exascale opportunities. This ambition necessitates extensions to `PaRSEC` that incorporate information related to data structure and rank distribution into the runtime decision-making. We introduce two runtime optimizations to address the challenges encountered when confronted with a large rank disparity: (1) a trimming procedure performed at runtime to cut away data dependencies from the directed acyclic graph discovered to be no longer required after compression and (2) a rank-aware diamond-shaped data distribution to mitigate the load imbalance overheads, reduce data movement, and conserve memory footprint. We assess our implementation using 3D unstructured mesh deformation based on Radial Basis Function (`RBF`) interpolation. We report performance results on two different high-performance supercomputers and compare against existing state-of-the-art implementation. Our implementation shows up to 7-fold on `Shaheen II` and 9-fold on `Fugaku` performance superiority in situations where the 3D unstructured mesh deformation application renders a matrix operator with low density. Our software framework solves a formally dense 3D problem with 52M mesh points on 65K cores in about half an hour. This multidisciplinary work emphasizes the need for runtime systems to go beyond their primary responsibility of task scheduling on massively parallel hardware system, by synergistically bridging matrix algebra libraries with scientific applications.**

***Index Terms*—Low-rank approximations, Task-based programming model, Dynamic runtime system, HPC, Mesh deformations**

I. Introduction

The last decades have witnessed a rapid improvement of computational capabilities in high-performance computing (HPC) platforms thanks to hardware technology scaling. This has necessitated continuous adaptations across the software stack to maintain high hardware utilization. HPC architectures benefit from mainstream advances on the hardware with many-core systems, deep hierarchical memory subsystem, non-uniform memory access, and an ever-increasing gap between computational power and memory bandwidth. In this HPC landscape of potentially million-way parallelism, task-based programming models associated with dynamic runtime systems are becoming more popular [1]–[7]. They foster developers' productivity at extreme scale by abstracting the underlying hardware complexity. Algorithms powered by a task-based runtime system are divided into two parts: `numerical kernels` carrying the fine-grained computational load and `data dependencies` representing how the data flows between the different kernels. Algorithms can then be represented as a directed acyclic graph (DAG) with vertices as tasks and edges as dependencies. The resulting task-based algorithms expose massive parallelism that can be exploited by a dynamic runtime system. The runtime system can dynamically orchestrate tasks and can map them onto the hardware resources. This separation of concerns may relieve domain scientists from the aforementioned programming burden.

With the advent of big data applications, the high algorithmic complexity and large memory footprint of traditional matrix computations remain major obstacles for their efficient deployments, despite the HPC hardware evolution. Low-rank matrix approximations come to the rescue as a paradigm shift from legacy HPC linear algebra algorithms. These algebraic methods exploit data sparsity of the matrix operator by compressing off-diagonal tiles up to an application-dependent accuracy threshold. Each compressed tile captures the most significant information to eventually deliver the required accuracy. The challenge resides in redesigning the numerical algorithms to operate on a compressed data layout. The opportunities are manifold: reducing the algorithmic complexity, saving memory footprint, and minimizing data movement, while satisfying the application accuracy.

In this paper, we implement a software framework `HiCMA-PaRSEC` that couples the `PaRSEC` task-based runtime system [7] and the `HiCMA` low-rank matrix computations library [8] to accelerate large-scale scientific applications. The synergism of such a software bundle has proven to be highly efficient when tackling challenging problems in many scientific domains [8]–[11]. Herein, we demonstrate the versatility of our software framework by targeting 3D unstructured mesh deformation for computational fluid dynamics (CFD) applications, using the Radial Basis Function (`RBF`) approach [12]. This technique interpolates the displacements of the boundary nodes located on the surface of moving bodies with high fidelity. Matrices from `RBF` kernels are symmetric positive-definite and formally dense, yet exhibit inherent data sparsity properties. This allows for the effective use of low-rank approximation on the off-diagonal tile when using these kernels. This translates into solving a data-sparse linear system using the Cholesky factorization that represents the most time-consuming phase. Depending on the targeted accuracy, the low-rank compression format may actually nullify some of the tiles of these `RBF` matrices. The matrix operator may thus assume various data layouts during the lifespan of the studied CFD application: initially dense during generation, then rank structured after compression, and possibly leading toward sparse. Moreover, the degree of the sparsity varies with the problem types and the desired accuracy. This may further exacerbate the already existing computational load imbalance because of rank heterogeneity, as discussed in [13].

Our resulting framework expands on the capabilities of `Lorapo` [9]—the state-of-the-art in `TLR` matrix computations. Our software makes a leap forward by mixing data structures that traditionally support the broad linear algebra discipline within a single matrix operation. This expansion is made possible thanks to the fundamental design of our approach based on the aforementioned separation of concerns. Once the matrix structure is exposed after compression of the dense matrix, it becomes essential to trim the original DAG from data dependencies on the null tiles that are no longer required. A significant part of the `PaRSEC` runtime overhead can be removed, including task management, scheduling, dependency releases, and temporary memory usage. However, sparsifying the DAG introduces further load imbalance in addition to the rank disparity. We design a new rank-aware diamond-shaped data distribution and deploy it at runtime, while reducing expensive data movement in tasks belonging to the critical path. We break the traditional owner-computes strategy to hide the overheads engendered by the data redistribution with useful computations. We evaluate the resulting `TLR` Cholesky implementation on two large supercomputers. Our codes outperform `Lorapo` by up to a 7-fold speedup and can efficiently solve 3D unstructured mesh deformations up to 52M mesh points. We believe this multidisciplinary symbiosis of low-rank approximation, runtime system and domain applications is fundamental to leverage exascale opportunities.

The remainder of this paper is as follows. We present related work in Section II and list our contributions in Section III. Section IV provides basic backgrounds, and Section V details challenges carried by the `RBF` kernels. The next two sections detail the optimizations proposed in this work, i.e., the DAG trimming and the diamond-shaped distributions in Section VI and Section VII, respectively. Section VIII presents a comprehensive performance analysis of the resulting software ecosystem, followed by conclusions and planned work in Section IX.

II. Related Work

This section briefly recaps runtime systems and `TLR` matrix computations to accelerate large-scale applications.

Runtime Systems. The dynamic runtime programming paradigm is not a new concept. For the purpose of this paper, we are primarily interested in task-based dynamic runtimes that target distributed-memory systems and schedule fine-grained computational tasks running well below the order of a second. `OpenMP` [1] is widely used and provides a portable set of compiler-directives in many programming languages for writing parallel sections of code. Its main goal is to provide functionalities on shared-memory systems; support for inter-process communications and synchronizations is delegated via explicit calls to an external communication substrate, e.g., MPI. On distributed-memory systems, `StarPU` [2] and `OmpSs` [3] provide users with a simple task-insertion API, similar to the task interface in `OpenMP`. The task-graph can be dynamically built and unrolled as computational progress occurs. Besides, peer-to-peer and collective communications are provided in `StarPU` with certain limitations, assuming all related dependencies are discovered before a collective starts [4]. `OmpSs` is based on compiler directives and supports many-core architectures, which is considered a forerunner for `OpenMP`. Distributed memory support in `OmpSs` is offered by COMP Superscalar (COMPSs) [14]. Both aforementioned runtimes suffer from a sequential task-insertion, which along with the DAG pruning phase, may have potential limitations on scalability [15]. `HPX` [5] follows the concepts of the ParalleX execution model, supporting inter-process scheduling and many-core architectures, with communications implicitly described in the language. By describing logical regions of data, `Legion` [6] presents abstractions that allow programmers to describe properties of program data (e.g., independence, locality). It relies on `REALM` [16], an event-based low-level runtime for scheduling on distributed-memory machines.

Tile Low-rank Matrix Computations. Low-rank matrix approximations constitute a *renaissance* for dense linear algebra. Indeed, low-rank matrix approximations in the form of hierarchical matrices ($\mathcal{H}$-matrices) [17]–[19] permit the reduction of arithmetic complexity and memory footprint of direct dense factorizations and solvers. There are currently many state-of-the-art data compression formats for $\mathcal{H}$-matrix approximation supporting weak (e.g., Hierarchically Semi-Separable (HSS) [20], [21], Hierarchically Off-Diagonal Low-Rank (HODLR) [22], [23]) and strong admissibility (e.g., $\mathcal{H}^2$-matrix [24], Block/Tile Low-Rank (BLR / TLR) [25], [26]). A subset of these data compression formats may achieve almost linear arithmetic complexity and memory storage for some matrix kernels/operations [27]. However, these data compression formats have their own computational bottlenecks when dealing with typical 3D problems: either because of the high ranks required for accuracy in the large off-diagonal blocks (i.e., for weak admissibility with HODLR/HSS) or the limited performance scalability on distributed-memory systems (i.e., for strong admissibility with $\mathcal{H}^2$). The authors of [8]–[11] provide extended support for TLR matrix computations in order to tackle 3D problems (e.g., acoustic wave scattering and geospatial statistics) on distributed-memory systems. The main idea is to flatten the recursion and to avoid synchronizations in-between hierarchical steps, while promoting task-based dynamic runtime systems for mitigating load imbalance on distributed-memory systems.

By leveraging data sparsity of the underlying matrix operator, solving a broad class of large-scale scientific applications [28] becomes feasible. For instance, computing the unstructured mesh deformation of moving 3D bodies using `RBF` interpolations leads to solving a large dense system of linear equations, for which the matrix factorization account for most of the time. The `RBF` matrix size of the problem corresponds to the number of the boundary elements located at the surface of the 3D objects. Due to the resulting cubic arithmetic complexity, the `RBF` approach has not traditionally been considered as a solution of choice when targeting large mesh deformation.

III. Contributions

The contributions of this paper are as follows. We propose a framework, powered by the `PaRSEC` runtime [7] and the `HiCMA` linear algebra library [8], to efficiently compute the `TLR` Cholesky factorization on data-sparse matrices, as generated by the challenging CFD dynamic meshing problem from Gaussian `RBF` kernels. Compressing such matrix operators may generate a mixture of data structures (i.e., dense, `TLR`, and sparse), which have not been observed for 3D spatial statistics [8]–[10], [13] or computational electromagnetic [11]. To provide an efficient support for the resulting compressed matrices, `PaRSEC` must revisit the original algorithm dependency graph and reduce the number of tasks with related data dependencies, prior to launching the matrix factorization. At the same time, the use of these compressed matrices raises new challenges with regards to the compute and communication balance between the participating processes. We introduce two runtime-level optimizations related to a rank-aware data distribution, and show their impact on the time-to-solution. In particular, we alter the distribution space for the kernel execution, which breaks the traditional owner-computes strategy supported by most of the existing runtimes. This change allows the `PaRSEC` runtime to transparently rebalance the work while respecting the initial data distribution provided by the user. We demonstrate the scalability and efficiency of our software framework on a very large `SARS-CoV-2` dataset that reveals the various aforementioned data structures after compression. We illustrate our software capability to solve the problem at an unprecedented scale.

To the best of our knowledge, this is the first time a runtime system tackles 3D data-sparse problems morphing into a mixture of data structures on distributed-memory systems.

IV. Background

A. PaRSEC Runtime System

`PaRSEC` [7] is a generic task-based runtime system for asynchronous, architecture-aware scheduling of fine-grained tasks on distributed many-core heterogeneous architectures. In `PaRSEC`, like most task-based runtime systems, concepts of task and dependency are utilized to describe the computations and the corresponding data directions used by these tasks, respectively; therefore, algorithms can be represented as a DAG with vertices as tasks and edges as dependencies. `PaRSEC` can dynamically unfold this DAG on a set of distributed resources and satisfy the declared data dependencies by shepherding data between memory spaces (intra-node, inter-nodes, different devices, etc.) and scheduling tasks across heterogeneous resources. The interactions with the runtime are handled via several Domain-Specific Languages (DSLs) to provide more flexibility and help domain scientists to express algorithms in a more productive way. Parameterized Task Graph (PTG) [29], used in this paper, represents the dependencies between tasks by a concise, yet comprehensive task graph description called Job Data Flow (JDF). The represented DAG can be considered as a collection of task classes which holds information about enabling the creation and execution of the task instances, including the operations to be executed on different computational units. The communications are implicit, derived directly from the dependencies between tasks and supporting many protocols such as peer-to-peer and collective (e.g., broadcast, gather, scatter, reductions) to enhance productivity. Template Task Graph (TTG) [30] provides C++ API and extends the idea of PTG by generalizing the notion of parameters to arbitrary types and enabling data-dependent selection of task dependencies. Other DSLs, such as Dynamic Task Discovery [15], are less domain science-oriented and express DAG by sequential task insertion in nested loops. Due to the sequential discovery of tasks, it may suffer from the same high overhead as other distributed task-insertion runtimes, such as `StarPU`.

B. Tile Low-Rank Cholesky Factorization

TLR Cholesky exploits data sparsity of the matrix by compressing off-diagonal tiles, i.e., capturing the most significant singular values (which define the rank of the tile), up to an application-dependent accuracy threshold. Therefore, each off-diagonal dense tile is compressed to two tall-and-skinny tiles, i.e., U and V of size $b \times k$ with b the tile size and k the rank. The algorithms of TLR Cholesky and its dense counterpart are similar except for the computational kernels [9], i.e., POTRF (Cholesky factorization), TRSM (triangular solve), SYRK (SYRK-Dense or SYRK-TLR, symmetric rank-k update) and GEMM (GEMM-Dense or GEMM-TLR, general matrix multiply). We rely on HiCMA [8] to provide an efficient implementation for these computational kernels. Also, the dense and TLR Cholesky share the same definitions of the critical path and dependencies patterns. In particular, the critical path results in a sequential series of operations on tiles mostly with full dense structure. It repetitively follows the same pattern: POTRF on a dense diagonal tile, a single TRSM on a tile directly below the diagonal and, the first SYRK on the next diagonal tile for the next panel factorization. Hiding the cost of the critical path constitutes one of the main challenges for TLR Cholesky since matrix computations outside of it are limited compared to dense Cholesky.

C. 3D Unstructured Mesh Deformation

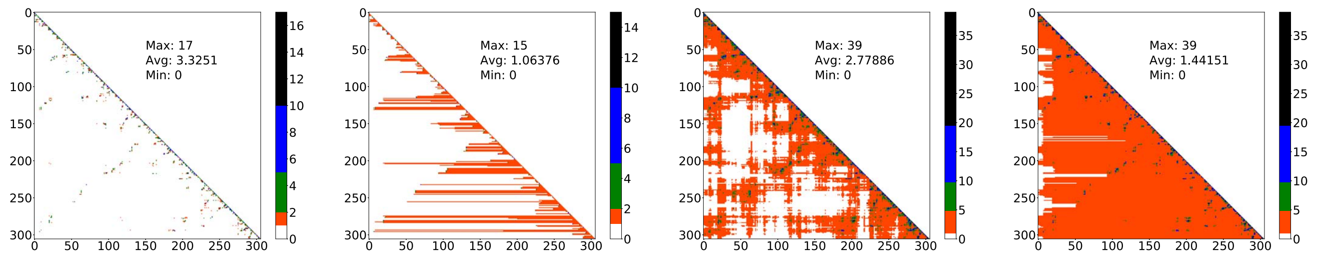
Simulation of fluid-structure interactions involving moving 3D bodies requires computation of large mesh deformations. The RBF technique is an interpolation method that produces high-quality unstructured adaptive meshes. However, the RBF-based boundary problem necessitates solving a large dense linear system that is computationally expensive and prohibitive in terms of storage requirement.

RBF is used to describe the displacement of the internal volume nodes given the displacement of the boundary nodes. As described in [12], an interpolation function describing the displacement d in the whole domain, can be approximated by a sum of basis functions: $d(x) = \sum_{i=1,n_b} \alpha_i \phi(||x - x_{b_i}||) + p(x)$, where $x_{b_i} = [x_{b_i}, y_{b_i}, z_{b_i}]$ are the boundary nodes at which the values are known, p a polynomial, n_b the number of boundary nodes and ϕ a given basis function. The coefficients α_i and the polynomial p are determined by the interpolation conditions $d(x_{b_i}) = d_{b_i}$, where d_b contains the known displacement values at the boundary. The system of unknowns α is subject to the following constraint: $\sum_{i=1,n_b} \alpha_i p(x_{b_i}) = 0$. A unique interpolant is given if the basis function is a conditionally positive definite function. If the basis functions are conditionally positive definite of order $m \leq 2$, one can use a linear polynomial. Herein, we only consider basis functions that satisfy this criterion. A variety of RBF kernels exists in the literature [31], [32], in which a distinction is usually made between global and compact support. The values of the former are always non-zero, while the values of the latter are exactly zero at some distance away from the source (i.e., outside their support radius). This paper focuses on Gaussian RBF from the global support category: $\phi(r) = \exp(-r^2)$, where r is Euclidean distance. The global support functions are usually scaled by a shape parameter δ to avoid excessive condition numbers of the resulting RBF matrices. The scaled version of the RBF function is defined as follows: $\phi_\delta(r) := \phi(r/\delta)$ where δ is chosen as: $\delta = 1/2 \times min||x - x_{b_i}||$. In general, the global support function leads to a more accurate solution because it considers all interactions between mesh points, at the cost of *producing a dense matrix* [12]. We employ mesh reordering based on Hilbert Space Filling Curves in order to preserve a good spatial locality, while improving compression rate and reducing arithmetic complexity.

V. New Challenges

As detailed in Section IV-C, the global support category for RBF kernels produces a dense matrix. This dense matrix can be compressed by low-rank approximation based on an application-dependent accuracy threshold. As the compression happens at the level of each tile, some tiles may disappear if their contributions turn out to be below the application-defined threshold. In addition, the shape parameter may increase the matrix sparsity of the RBF, leading to further null tiles. For the remaining tiles, the rank becomes relatively small compared to the tile size, which reduces the arithmetic intensity of the tile operations [10]. The mixture of data structures introduces new challenges compared to other previous TLR applications [8]–[11], [13]. At the lowest level, a decreasing rank leads to lower task granularity and therefore emphasizes the underlying overheads of the runtime and of the communication layer [33]. At a higher level, the entire critical path of the algorithm as well as the data distribution, load and communication balance are drastically affected. This will require a complete overhaul to maintain the scalability and efficiency of the original algorithm applied on the dense matrix.

To understand exactly how much the compression step and the factorization we apply affect the sparsity of the matrix, Fig. 1 displays the heatmap of the rank distribution on a lower triangular matrix according to two shape parameters. It shows the initial rank distribution, i.e., after matrix compression, and final rank distribution, i.e., after the Cholesky factorization, with matrix size 1.49M and tile size 4880, along with the maximal, average and minimal rank (the average rank is only for non-zero tiles). We define the matrix density as the ratio of non-zero tiles, while sparsity is the complement, i.e., sparsity = 1−density. Three things need to be highlighted: the impact of the shape parameter on the density of the matrix, the variability of the matrix density between the initial and final step, and the sharp decrease in the ranks of the tiles with the distance to the diagonal. These parameters, i.e., shape, density and rank, are problem dependent, which makes even more challenging the construction of an algorithm that works well across the entire spectrum. From Fig. 1, we can see the ranks for all off-diagonal tiles are relatively small compared to the tile size, which means a greater discrepancy between tiles on- and off- diagonal with regard to the computational intensity and communication volume. Also, the shape parameter has a significant impact on matrix sparsity, from a



(a) Initial rank; shape parameter 3.7×10^{-4}, `density` 0.024606.
(b) Final rank; shape parameter 3.7×10^{-4}, `density` 0.206908.
(c) Initial rank; shape parameter 5.0×10^{-2}, `density` 0.552031.
(d) Final rank; shape parameter 5.0×10^{-2}, `density` 0.936969.

Fig. 1: Initial (after compression) and final (after Cholesky) rank distribution of off-diagonal tiles based on the shape parameter.

very sparse matrix, e.g., Fig. 1 (a) and (b), to a quite dense matrix, e.g., Fig. 1 (c) and (d). Matrix sparsity and the variable rank of each tile may cause load imbalance and increase programming efforts, especially on distributed systems. This necessitates new tools focused on user-productivity, while addressing performance and scalability challenges. Therefore, a general framework orchestrated in a versatile runtime PaRSEC is needed to efficiently solve this challenging problem.

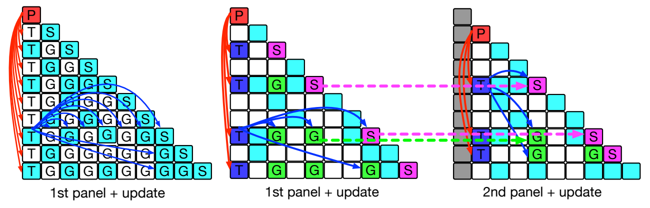
VI. Dynamic DAG Trimming

As previously mentioned, the matrix sparsity resulting from the compression step needs to be exploited to reduce the runtime overhead. Indeed, since the entire dense DAG is exposed to the runtime system, tasks operating on zero-rank tiles and their dependencies are still fed to the runtime decision-making. Unlike other TLR approximation research on distributed memory in the literature [9], [10], we alter *at runtime* the DAG structure taking into account the disappeared tiles, in order to remove overheads due to handling unnecessary tasks and their dependencies. Therefore, the DAG needs to be trimmed: only dependencies related to non-zero or fill-in tiles in that panel factorization should be exposed to the runtime system. An analysis of the compressed matrix is required, so that enough information can be gathered from the structure of the resulting TLR matrix. The outcome is provided to the runtime via the DSL, indicating how to trim the DAG. Algorithm 1 describes such analytical process, which identifies null tiles and deploys the trimming procedure. In this algorithm, array 'rank' is a 1D array to differentiate tiles from non-zero rank to zero rank, and is initialized to the initial rank after matrix compression. NT is the number of tiles in a dimension, and all values in the structure 'analysis' are initialized to 0. The time complexity of Algorithm 1 is $O(max(NT^2, d^2 * NT^3))$, where d is the final `density` after Cholesky factorization. The distributed version of the trimming procedure will only consider the tiles that will be locally updated on each process, therefore, limiting the memory requirements to analyze the matrix's sparsity pattern.

In this way, the DAG can be pruned by reducing the execution space of each task class, i.e., TRSM, SYRK, and GEMM, according to the analyzed information, so that unnec-

Algorithm 1: Matrix analysis for DAG trimming.

```
Input   : initial rank array: rank
Output: hicma_parsec_analysis_t *analysis
Initialize structure: analysis
for ( k = 0; k < NT-1; k++ )
    trsm_index = 0
    for ( m = k+1; m < NT; m++ )
        if ( rank[k*NT+m] > 0 )
            analysis.trsm[k][trsm_index++] = m
            analysis.nb_trsm[k]++
            syrk_index = analysis.nb_syrk[m]
            analysis.syrk[m][syrk_index] = k
            analysis.nb_syrk[m]++
    for ( i = 1; i < analysis.nb_trsm[k]; i++ )
        for ( j = 0; j < i; j++ )
            m = analysis.trsm[k][i]
            n = analysis.trsm[k][j]
            rank[n*NT+m] = 1
            if ( tile(m, n) resides this MPI process )
                gemm_index = analysis.nb_gemm[m][n]
                allocate a piece memory if needed
                analysis.gemm[m][n][gemm_index] = k
            analysis.nb_gemm[m][n]++
```



(a) Before trimming. (b) After trimming.

Fig. 2: Data dependencies before and after DAG trimming with 10×10 tiles for the 1st and 2nd panel factorization. White indicates tiles that have disappeared during the compression, other colors are non-zero or fill-in tiles due to panel factorization. Tiles labeled with P (POTRF), T (TRSM), S (SYRK) and G (GEMM) represent the task to be executed on that tile during factorization. Arrows are data flows between different tasks, with multiple flows from the same source tile representing a broadcast operation. The dashed green and magenta arrows show all dependencies between the 1st and 2nd panel/update after DAG trimming.

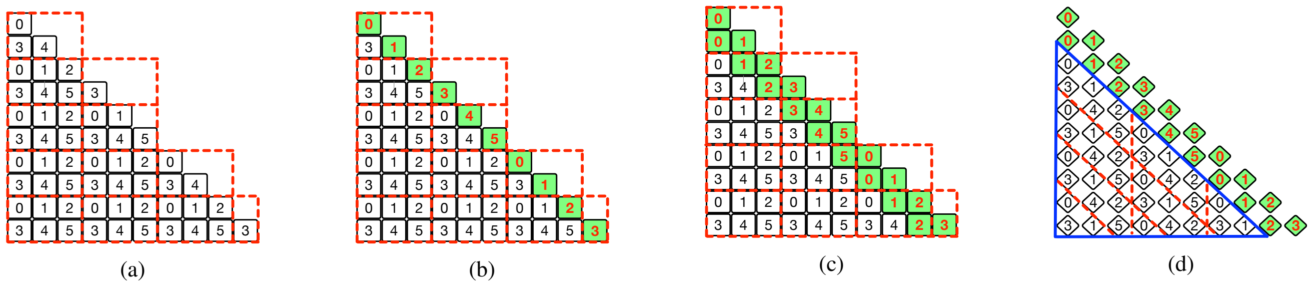


Fig. 3: Data distribution of a matrix of 10×10 tiles using 6 processes. (a) ScaLAPACK 2DBCDD; (b) Lorapo hybrid 1DBCDD+2DBCDD; (c) band distribution to reduce communication in the critical path; (d) diamond-shaped distribution to balance workload for off-band tiles. 1DBCDD/2DBCDD with process grids $1 \times 6/2 \times 3$ for green and white tiles, respectively.

essary dependencies are eliminated. Fig. 2 demonstrates how the DAG is trimmed. Before DAG trimming, the matrix is assumed to have all tiles (in dense or TLR formats) and their dependencies active. Fig. 2a showcases dependencies within a panel factorization without DAG trimming. However, once the sparsity is taken into account, only a fraction of tasks and their dependencies remains operational, as demonstrated in Fig. 2b. All in all, compressing the matrix operator may result also into a compression of its DAG, where only the eligible tasks and their dependencies are kept functional.

VII. Rank-Aware Diamond-Shape Data Distribution

Besides optimizations pioneered in prior work [9], [10], i.e., reducing communication, lookahead, and nested parallelism, we introduce a novel rank-aware data distribution. This distribution not only mitigates the work imbalance produced by the RBF kernel (see Section V) but also generalizes our approach for tackling various 3D data-sparse scientific applications.

A. Reducing Communication in the Critical Path

In the literature, the definition of the critical path in dense Cholesky factorization (see Section IV-B) usually only includes the numerical kernel execution—assuming kernels in the critical path sequentially execute on shared memory—but skips the communications in the critical path. The main reason is that in dense Cholesky, the communications in the critical path are not crucial: either the portion of communication in the critical path is low, and/or there is enough work off the critical path to hide that communication. However, as mentioned hereinbefore, the resulting sparsity of the compressed matrix and the small rank of the compressed tiles lead to the great discrepancy in computational intensity between on- and off-diagonal tiles. Therefore, the critical path becomes cumbersome to get overall performance. Lorapo [9] proposes the concept of "hybrid distribution", which builds upon the traditional two-dimensional block cyclic data distribution (2DBCDD, Fig. 3a). It combines 1DBCDD and 2DBCDD together to balance workload between on- and off-diagonal tiles, respectively, as shown in Fig. 3b. We extend this idea to reduce communication in the critical path by binding the task operating the TRSM in the critical path (the first TRSM in each panel factorization) to the same affinity as the task executing the POTRF in each panel factorization. Fig. 3c highlights this band distribution, which results in having the diagonal and subdiagonal with the same process pattern. In this way, we replace the expensive communication in the critical path due to POTRF-TRSM data dependencies that involves remote nodes with a local communication instead.

B. Diamond-Shaped Data Distribution

Load imbalance is one of the main bottlenecks for TLR matrix computations. It arises because of the rank disparity and eventually the sparsity of the compressed matrix. By applying the previous optimization, load imbalance gets further exacerbated. For general 3D covariance matrix problems, the correlation strength usually decreases as we get farther from the main diagonal. This pattern is even more severe in RBF applications, creating abrupt changes in the rank distribution. We introduce a new diamond-shaped data distribution that exploits this inherent pattern, while maintaining some regularity in the tile distribution. The new distribution assigns process ID for each tile in the diamond shape by skewing the original 2DBCDD. This creates more opportunity to balance the workload than the original rectangular static 2DBCDD, as implemented in ScaLAPACK (see Fig. 3a). Fig. 3d showcases the diamond-shaped distribution using a 2×3 diamond process grid. In addition, it can still keep the column process group as optimal as 2DBCDD, which controls two broadcasts, i.e., POTRF to TRSMs and TRSM to GEMMs in a column. However, more processes may be involved in the row process group. But this is not critical since (1) only one broadcast is involved, i.e., TRSM to GEMMs in a row, and (2) the rank is tiny (see Fig. 1) leading to small message size in this broadcast.

Instead of redistributing the data following this new rank-aware diamond-shaped distribution, which would have required an additional step composed of all the necessary communications to redistribute the data, we take advantage of PaRSEC capability to dissociate the data ownership from the operations on this data, and allow a task execution mapping different than the mainstream owner-compute strategy. Thus, we leave the compressed matrix uses its original distribution, but bind the different task's execution following the rank-aware distribution described above. PaRSEC automatically satisfies all dependencies necessary for tasks execution, and therefore, moves the data from the owner to the location where it is used. Upon completion of all operations on the data, PaRSEC moves it back to its original owner. As a result from the point of view of the user, everything happens as if the data has not been moved and as if the operations are applied in-place.

This approach may cause additional communications, but they happen at most twice per non-zero tile during the entire application execution, when reading and writing the data from and to the original storage. Moreover, since no explicit redistribution stage [34], [35] is required, no additional temporary memory is needed, and the introduced additional communication overheads may be hidden by the computation during runtime. Last but not least, the entire process of moving the data to rebalance the workload is completely transparent from the user perspective, being akin to some of temporary data migration. Our non-invasive approach may eventually permit the user to continue distributing their data using the traditional 2DBCDD in the remaining parts of their applications.

VIII. Performance Results and Analysis

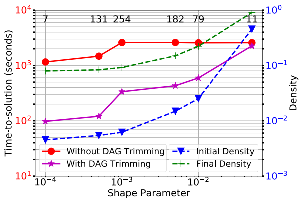
A. Environment Settings

The experiments are conducted on two large clusters. Shaheen II is a Cray XC40 cluster with 6,174 compute nodes, each with two 16-core Intel Haswell CPUs running at 2.30 GHz and 128 GB of DDR4 main memory. Fugaku contains 158,976 compute nodes, each with a 48-core A64FX CPU running at 2.2 GHz and 32 GB of HBM2 memory.

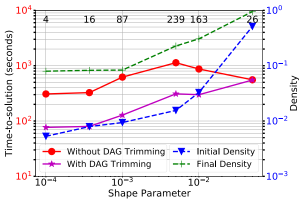
For optimized BLAS and LAPACK kernels: on Shaheen II, Intel compiler suite 19.0.5.281 with sequential Math Kernel Library (MKL) version 2019.5; on Fugaku, Fujitsu compiler with clang mode and SSL2. We simulate a population of SARS-CoV-2 viruses with a resolution of 44932 mesh points. We extract the virus geometry extracted from the Protein Data Bank (PDB) codenamed PDBID 6VXX available at (https://www.rcsb.org/structure/6VXX). We vary the number of viruses in a cube with edge length $1.7\mu m$ from 30 (i.e., $1.49M$ mesh points) virus to 1200 (i.e., $52.57M$). An accuracy threshold of 10^{-4} has been used for subsequent results, unless otherwise specified. This is sufficient to satisfy the displacement accuracy requirements of this 3D unstructured mesh deformation applications during the linear solver. For the data distribution used for off-band tiles towards kernel execution (Fig.3d), we deploy a process grid $P \times Q$ (as square as possible) where $P \leq Q$. Calculations and communications are performed in double-precision floating-point arithmetic. We run our experiments at least three times and since no noticeable performance variability has been identified, the minimum time to solution is reported.

B. Impact of the Shape Parameters

Gaussian RBF kernel formulation contains a free shape parameter that has a significant impact on the overall accuracy. This shape parameter controls the shape of the basis function. A small value decreases the correlation strength while a large value increases it. Finding the optimal shape parameter is a difficult problem, and many researches choose its value by trial and error [36]. We investigate in Fig. 4 the effect of this shape parameter on the compressed RBF operator's density and the TLR Cholesky performance. We vary the shape parameter from $O(10^{-4})$ to $O(10^{-2})$, producing a compressed matrix with a more sparse to a more dense data structure. From Fig. 4, we can see the matrix density increases between compression and TLR Cholesky, as the shape parameter rises due to fill-in occurring during factorization. The labeled ranks get higher with the shape parameter increase, but then eventually decrease since correlations because more scattered across the domain. All in all, the shape parameter has a direct influence on the algorithmic complexity and the elapsed time. Moreover, we notice that the curves with and without DAG trimming converge for large shape parameters. As the shape parameter increases, the number of null tiles eventually decreases, which makes DAG trimming procedure obsolete. This highlights the effectiveness of the matrix analysis in Algorithm 1 used for DAG trimming optimizations. For the remaining experiments, we choose the shape parameter 3.7×10^{-4}, which translates into considering half of the minimum distance between the mesh nodes, as mentioned in section IV-C.



(a) 16 nodes on Shaheen II.

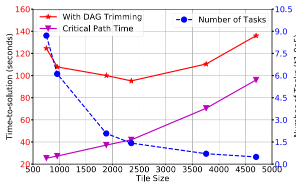


(b) 64 nodes on Fugaku.

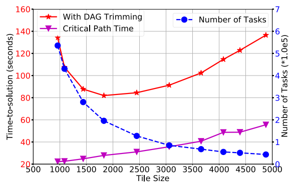
Fig. 4: Impact of the shape parameter on matrix density and time-to-solution when looking at: (1) initial density (after compression) and final density (after Cholesky), (2) with and without DAG trimming optimizations, and (3) labeled max_rank. Matrix size/tile size: (a) 4.49M/2390 (b) 2.99M/2440.

C. Understanding the Impact of Tile Size

Tile size is a crucial parameter in tile algorithms, trading off task granularity, compute intensity and concurrency. The literature covers its importance to TLR matrix computations [10], [13]. The authors in [13] propose a model to calculate the approximate optimal tile size by assuming a first-order approximation——the sequential part (the critical path in the algorithm) at distance one overlapping with the parallel part (everything outside the critical path). Because of matrix sparsity and lower task granularity which emphasizes the importance of communication and runtime overheads, this assumption does not hold for the RBF application. Therefore, we follow the strategy in [10] to tune the tile size b in practice using $b = O(\sqrt{N})$ (with N the matrix size), which theoretically provides a minimal operation count by for TLR matrix computations [37]. This trade-off is pictured by a time-to-solution curve following a bell shape. Fig. 5 demonstrates this pattern on the two platforms and analyzes the reasons behind it by showing the time taken by the critical path and the



(a) 16 nodes on Shaheen II.



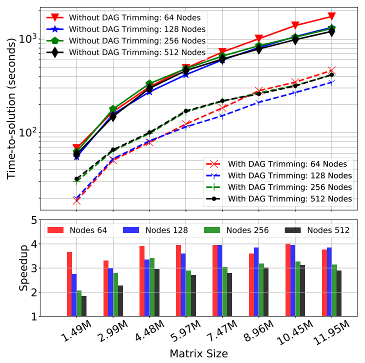
(b) 64 nodes on Fugaku.

Fig. 5: Impact of the tile size. Left y-axis: time-to-solution of TLR Cholesky and the critical path; right y-axis: the number of tasks. Matrix size: (a) 4.49M (b) 2.99M.

number of tasks. Indeed, when the tile size is large, the critical path plays a more significant role because the tiles on the matrix diagonal with full dense data structure account for most of the flops. As the tile size decreases, the number of flops in the critical path decreases, while the amount of computation outside the critical path increases. Therefore, after passing the sweet point where the critical path is balanced by the off-diagonal workloads, the number of tasks continues to increase while their granularity decreases. This leads to a situation where the cost of tasks outside the critical path dominates and eventually exacerbates all overheads of dealing with tasks in the runtime. Auto-tuning the tile size with a model is an important aspect but beyonds the scope of the paper. For this paper's needs it is enough to find, even experimentally, a reasonable value representing a local minima based on the strategy above. We apply this strategy for the rest of the experiments unless specified.

D. Impact of DAG Trimming Optimization

Fig. 6 (left) evaluates the impact of DAG trimming up to 512 Shaheen II nodes with matrix size up to 11.95M. Trimming the DAG always has a net positive impact, with the overhead of the trimming being always significantly smaller than the reduction in runtime overheads. As expected, the benefit is correlated to both the problem size and the number



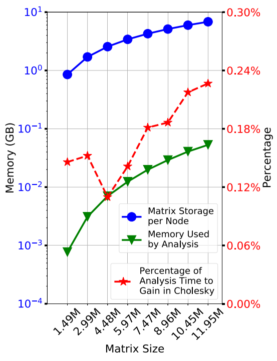


Fig. 6: Effect of DAG trimming optimization on elapsed time (left). Effect of null tile analysis in Algorithm 1 (right) on memory (left y-axis) and on elapsed time in percentage overhead (right y-axis) on 64 Shaheen II nodes.

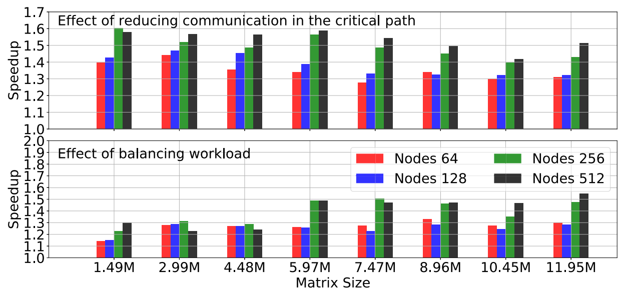


Fig. 7: Incremental effect of the optimizations on Shaheen II.

of compute resources used. This shows that the performance superiority comes directly from the removal of tasks and data dependencies no longer required. This reduces the cost of runtime scheduling and orchestration between the participating processes. Regarding the overheads of Algorithm 1, Fig. 6 (right) demonstrates both time and memory footprint for the trimming analysis are negligible. The subsequent graphs show performance results when DAG trimming is on.

E. Incremental Effect of Proposed Runtime Optimizations

Fig. 7 details the impact of the two proposed optimizations on the time-to-solution up to 512 nodes and up to 11.95M matrix size. Reducing communication in the critical path using a band distribution (Section VII-A) shows a positive impact in Fig. 7 (top) with speedup up to 1.60×. When additionally balancing the workload using the rank-aware diamond-shaped data distribution (Section VII-B), we score further performance improvement in Fig. 7 (bottom) attaining up to 1.55×. From Fig. 7 (top), we can see the impact of the communication reduction increases with the number of processes. Indeed, the most communication intensive parts of the Cholesky factorization are the row and column broadcast operations, as described in Section VII-B. These broadcast operations span across a similar number of processes but the row broadcast moving a low-rank tile has a lesser impact on performance than the column broadcast where a dense, diagonal tile is propagated. The new affinity in the critical path reduces the participants in the column broadcast, and thus reduce its impact. This is even more pronounced when the number of processes increases. By the same token, the additional benefit of load balancing using the diamond-shaped data distribution increases with the matrix sizes and the number of processes. The behavior for small matrices is slightly different since there may not be enough work to feed all processing units.

F. Performance Analysis

Performance comparison. As shown hereinbefore, shape parameter has effect on the compressed RBF operator's density and therefore the TLR Cholesky performance (demonstrated in Fig. 1 and Fig. 4). First, we compare HiCMA-PaRSEC against Lorapo [9] with variable shape parameters about different matrix sizes on 512 nodes Shaheen II, as detailed in Fig. 8.

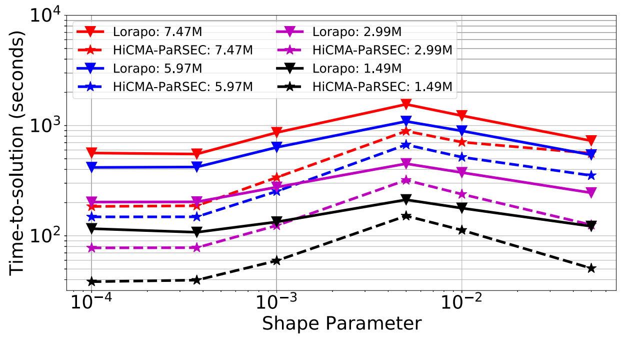


Fig. 8: Comparison with state-of-the-art of different shape parameters about four different matrix sizes on 512 nodes Shaheen II.

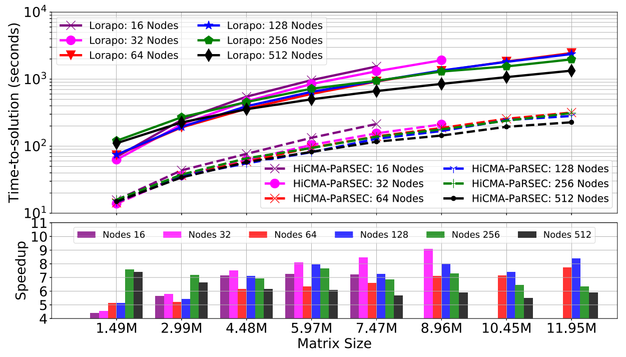


Fig. 10: Comparison with state-of-the-art on Fugaku.

HiCMA-PaRSEC beats Lorapo in all scenarios, from a very sparse matrix (small shape parameter, e.g., 1.0×10^{-4}) to a quite dense matrix (large shape parameter, e.g., 5.0×10^{-2}). Then, while fixing the shape parameter to 3.7×1.0^{-4} (see Section VIII-B), Fig. 9 and Fig. 10 depict the comparison of HiCMA-PaRSEC against Lorapo on more broad settings—up to $11.95M$ matrix size and 512 nodes on two clusters, Shaheen II and Fugaku, respectively—including the detailed time-to-solution and the corresponding speedups. On both systems, HiCMA-PaRSEC consistently outperforms Lorapo: with up to 6.8× speedup and maintaining a steady 6× speedup for matrices larger than 5.97M on Shaheen II; on Fugaku, with up to 9.1× speedup and achieving more than 4× speedup for all matrices. Our generic HiCMA-PaRSEC software bundle is the first approach to efficiently deal with a mixture of data structure on distributed-memory systems, characterized by the challenges of RBF applications, as described in Sections V.

Time Breakdown. Fig. 11 compares the time breakdown of TLR Cholesky of HiCMA-PaRSEC against Lorapo along with matrix compression. HiCMA-PaRSEC reduces the factorization by such a significant factor, that the compression of the initial matrix from dense to TLR becomes the most expensive part, hinting at a possible future work on how to generate the matrix directly in compressed format [38].

Different Accuracy Thresholds. To further highlight the versatility of our framework, Fig. 12 evaluates HiCMA-PaRSEC against Lorapo, using various accuracy thresholds on 512 nodes Shaheen II. As we increase the accuracy threshold from 10^{-5}, to 10^{-7} and to 10^{-9}, the elapsed time increases since we capture more information on each tile resulting in higher ranks. HiCMA-PaRSEC shows significant performance superiority against Lorapo regardless of the accuracy threshold. HiCMA-PaRSEC improves even further the relative performance between accuracy thresholds compared to Lorapo, thanks to the communication reduction in the critical path and the diamond-shaped data distribution for load balancing.

G. *Roofline Algorithmic Model*

A better way to estimate how far an algorithm is from the optimum is to compare it with a known theoretical bound and see how much room for improvement remains. As described in Section IV-B the critical path in Cholesky includes the computational kernels without communication, under the assumption that with enough compute resources everything outside of the critical path would be totally hidden behind the sequential operations in the critical path. This critical path ignores communications, leading to a rather *optimistic bound*. We use it as our roofline algorithmic model. In order to make the critical path similar on a given matrix size for all settings, the tile size is constant in this section, i.e., 4880, which is in the range of empirically tuned tile size during experiments. HiCMA-PaRSEC always achieves more than 70% efficiency for the matrix sizes up to $11.95M$ on Shaheen II. The remaining

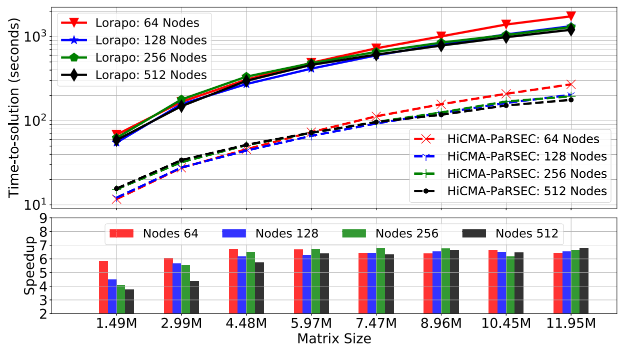


Fig. 9: Comparison with state-of-the-art on Shaheen II.

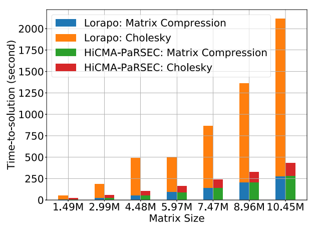


Fig. 11: Time breakdown on 512 Shaheen II nodes.

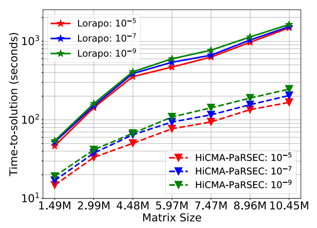


Fig. 12: Time Vs Accuracy on 512 Shaheen II nodes.

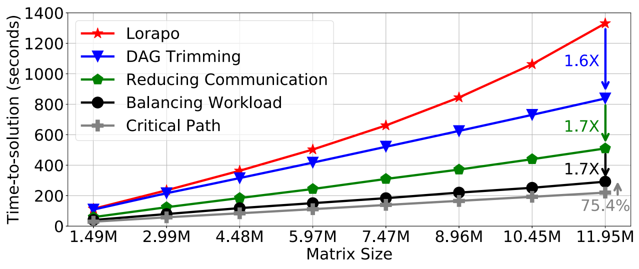


Fig. 13: Performance improvement trace and efficiency on 512 nodes Fugaku. The labeled numbers are the incremental speedup or the ratio of the critical path.

30% includes runtime overheads, communication costs, and off-band computational kernels, which can not be hidden by the sequential critical path.

We now apply our incremental performance optimizations on Fugaku. Fig. 13 depicts the time-to-solution of HiCMA-PaRSEC, the impact of the different optimizations and the critical path, augmented with the achieved efficiency, i.e., the ratio of the critical path to HiCMA-PaRSEC, on 512 nodes Fugaku with multiple matrix sizes. We see remarkable reductions in time-to-solution for each of the proposed optimizations, resulting in a complete solution significantly faster than the state-of-the-art Lorapo. We achieve 75.4% efficiency compared to the optimistic bound defined above. But getting closer to the time taken by the critical path is also an indication that our implementation may run out of concurrency. This situation can already be detected in Fig. 9 when a lower scalability is achieved as the number of nodes increases. For those problem sizes that do not benefit any more from adding more resources, the idea is to switch from dense to sparse direct solver to expose even more parallelism. In fact, PaRSEC provides support of sparse direct solver in PaStiX [39], but research on sparse direct solver powered with low-rank approximation is still limited to shared-memory [37], [40].

H. Performance Evaluation at Extreme Scale

To study the scalability at extreme scale, we push our solution to matrices and number of processors more akin to the desired norms in the target science domain, and present in Fig. 14 results with matrix sizes up to 52.57M and the number of nodes up to 2048. Each matrix size can be considered as a strong scaling experiment, and each number of nodes as a weak scaling one. It must be noted that matrix size of 52.57M cannot routinely be used in the literature of TLR matrix computations. HiCMA-PaRSEC can factorize such matrix size in 36 minutes. This represents a leap forward in TLR matrix approximation for supporting extreme-scale 3D unstructured mesh deformations. All in all, these results show the efficiency and scalability of our generic HiCMA-PaRSEC software bundle, while delivering results with the expected accuracy.

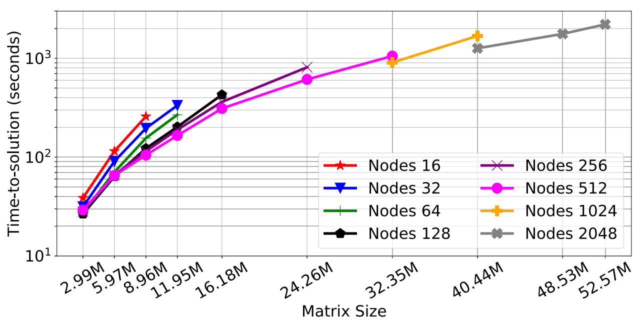


Fig. 14: Extreme scale performance on Shaheen II.

IX. Conclusion and Future Work

This paper demonstrates the efficiency and scalability of HiCMA-PaRSEC, a framework solving challenging 3D data-sparse RBF problems using TLR approximation. The challenge resides in the RBF matrix operator that is initially dense during generation, then rank structured after compression, and possibly leading toward sparse. HiCMA-PaRSEC can support these various data layouts within a single matrix factorization.

By exploiting the matrix sparsity resulting from the TLR compression of a formally dense matrix, the DAG of tasks of a Cholesky factorization is dynamically trimmed to eliminate unnecessary tasks and their data dependencies operating on zero-tiles that are no longer required. The remaining kernel execution space is automatically remapped to a novel rank-aware diamond-shaped distribution space to tackle the work imbalance. The change in execution pattern is completely hidden from the end users, but delivers a significant performance boost. These optimizations enhance the scalability of HiCMA-PaRSEC at extreme-scale and permit to efficiently support RBF application at unprecedented matrix sizes.

As future work, we intend to improve the matrix compression which, as a result of the optimizations presented here, became the most expensive part of the algorithms. In particular, we plan to generate the matrix directly in compressed format [38], without having to generate the full dense structure.

X. Acknowledgments

This research was supported in part by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration. For computer time, this research used the supercomputer Shaheen II at King Abdullah University of Science & Technology (KAUST) in Thuwal Saudi Arabia and the supercomputer Fugaku provided by RIKEN through the HPCI System Research Project (Project ID: hp200310 and ra010009).

References

[1] OpenMP, "OpenMP 4.5 Complete Specifications," 2015. [Online]. Available: https://www.openmp.org/wp-content/uploads/openmp-4.5.pdf
[2] C. Augonnet, S. Thibault, R. Namyst, and P. Wacrenier, "StarPU: A unified platform for task scheduling on heterogeneous multicore architectures," *Concurrency Computat. Pract. Exper.*, 2011.

[3] A. Duran, R. Ferrer, E. Ayguadé, R. Badia, and J. Labarta, "A Proposal to Extend the OpenMP Tasking Model with Dependent Tasks," *International Journal of Parallel Programming*, 2009.

[4] R. Lopes, S. Thibault, and A. Melo, "MASA-StarPU: Parallel Sequence Comparison with Multiple Scheduling Policies and Pruning," in *SBAC-PAD 2020-IEEE 32nd International Symposium on Computer Architecture and High Performance Computing*, 2020.

[5] T. Heller, H. Kaiser, and K. Iglberger, "Application of the ParalleX Execution Model to Stencil-based Problems," *Computer Science - Research and Development*, 2013.

[6] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken, "Legion: Expressing Locality and Independence with Logical Regions," in *International Conference for High Performance Computing, Networking, Storage and Analysis, SC*, 2012.

[7] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, T. Herault, and J. Dongarra, "PaRSEC: A Programming Paradigm Exploiting Heterogeneity for Enhancing Scalability," *Computing in Science and Engineering*, 2013.

[8] K. Akbudak, H. Ltaief, A. Mikhalev, A. Charara, A. Esposito, and D. Keyes, "Exploiting Data Sparsity for Large-Scale Matrix Computations," in *European Conference on Parallel Processing*, 2018.

[9] Q. Cao, Y. Pei, K. Akbudak, A. Mikhalev, G. Bosilca, H. Ltaief, D. Keyes, and J. Dongarra, "Extreme-scale Task-based Cholesky Factorization Toward Climate and Weather Prediction Applications," in *The Platform for Advanced Scientific Computing Conference*, 2020.

[10] Q. Cao, Y. Pei, K. Akbudak, G. Bosilca, H. Ltaief, D. Keyes, and J. Dongarra, "Leveraging parsec runtime support to tackle challenging 3d data-sparse matrix problems," in *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2021.

[11] N. Al-Harthi, R. Alomairy, K. Akbudak, R. Chen, H. Ltaief, H. Bagci, and D. Keyes, "Solving Acoustic Boundary Integral Equations Using High Performance Tile Low-Rank LU Factorization," in *35th International Conference on High Performance*. Springer, 2020.

[12] A. De Boer, M. S. Van der Schoot, and H. Bijl, "Mesh Deformation Based on Radial Basis Function Interpolation," *Computers and Structures*, 2007.

[13] Q. Cao, Y. Pei, T. Herault, K. Akbudak, A. Mikhalev, G. Bosilca, H. Ltaief, D. Keyes, and J. Dongarra, "Performance Analysis of Tile Low-Rank Cholesky Factorization Using PaRSEC Instrumentation Tools," in *IEEE/ACM International Workshop on Programming and Performance Visualization Tools (ProTools) at SC19*. IEEE, 2019.

[14] F. Lordan, E. Tejedor, J. Ejarque, R. Rafanell, J. Alvarez, F. Marozzo, D. Lezzi, R. Sirvent, D. Talia, and R. M. Badia, "Servicess: An Interoperable Programming Framework for the Cloud," *Journal of Grid Computing*, 2014.

[15] R. Hoque, T. Herault, G. Bosilca, and J. Dongarra, "Dynamic Task Discovery in PaRSEC: A Data-flow Task-based Runtime," in *Proceedings of the 8th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*, ser. ScalA '17, 2017.

[16] S. Treichler, "Realm: Performance Portability through Composable Asynchrony," Ph.D. dissertation, Stanford University, 2014.

[17] S. Goreinov, E. Tyrtyshnikov, and A. Y. Yeremin, "Matrix-free iterative solution strategies for large dense linear systems," *Numerical Linear Algebra with Applications*, vol. 4, no. 4, pp. 273–294, 1997.

[18] W. Hackbusch, "A sparse matrix arithmetic based on $\mathcal{H}$-matrices. Part I: Introduction to $\mathcal{H}$-matrices," *Computing*, 1999.

[19] R. Kriemann, "H-LU factorization on many-core systems," *Comput. Vis. Sci. Springer,*, 2013.

[20] E. Corona, P.-G. Martinsson, and D. Zorin, "An $O(N)$ direct solver for integral equations on the plane," *Applied and Computational Harmonic Analysis*, 2015.

[21] F.-H. Rouet, X. S. Li, P. Ghysels, and A. Napov, "A distributed-memory package for dense hierarchically semi-separable matrix computations using randomization," *ACM Transactions on Mathematical Software (TOMS)*, 2016.

[22] A. Aminfar, S. Ambikasaran, and E. Darve, "A fast block low-rank dense solver with applications to finite-element matrices," *Journal of Computational Physics*, 2016.

[23] S. Ambikasaran, D. Foreman-Mackey, L. Greengard, D. W. Hogg, and M. O'Neil, "Fast direct methods for Gaussian processes," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2015.

[24] S. Börm, *Efficient Numerical Methods for Non-Local Pperators: H2-Matrix Compression, Algorithms and Analysis*. European Mathematical Society, 2010.

[25] P. Amestoy, C. Ashcraft, O. Boiteau, A. Buttari, J.-Y. L'Excellent, and C. Weisbecker, "Improving multifrontal methods by means of block low-rank representations," *SIAM Journal on Scientific Computing*, 2015.

[26] K. Akbudak, H. Ltaief, A. Mikhalev, and D. Keyes, "Tile Low Rank Cholesky Factorization for Climate/Weather Modeling Applications on Manycore Architectures," in *32nd International Conference on High Performance*. Springer, 2017.

[27] W. Hackbusch, "Survey on the Technique of Hierarchical Matrices," *Vietnam Journal of Mathematics*, 2016.

[28] S. Börm, L. Grasedyck, and W. Hackbusch, "Introduction to Hierarchical Matrices with Applications," *Engineering Analysis with Boundary Elements*, 2003.

[29] A. Danalis, G. Bosilca, A. Bouteiller, T. Herault, and J. Dongarra, "PTG: An Abstraction for Unhindered Parallelism," *Proceedings of WOLFHPC: 4th International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing*, 2014.

[30] G. Bosilca, R. J. Harrison, T. Herault, M. M. Javanmard, P. Nookala, and E. F. Valeev, "The Template Task Graph (TTG)-an Emerging Practical Dataflow Programming Paradigm for Scientific Simulation at Extreme Scale," in *IEEE/ACM 5th International Workshop on Extreme Scale Programming Models and Middleware (ESPM2)*. IEEE, 2020.

[31] M. D. Buhmann, *Radial Basis Functions: Theory and Implementations*. Cambridge University Press, 2003.

[32] H. Wendland, "Error Estimates for Interpolation by Compactly Supported Radial Basis Functions of Minimal Degree," *Journal of Approximation Theory*, 1998.

[33] E. Slaughter, W. Wu, Y. Fu, L. Brandenburg, N. Garcia, W. Kautz, E. Marx, K. S. Morris, Q. Cao, G. Bosilca *et al.*, "Task Bench: A parameterized Benchmark for Evaluating Parallel Runtime Performance," in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2020.

[34] Q. Cao, G. Bosilca, W. Wu, D. Zhong, A. Bouteiller, and J. Dongarra, "Flexible Data Redistribution in a Task-Based Runtime System," in *2020 IEEE International Conference on Cluster Computing*. IEEE, 2020.

[35] Q. Cao, G. Bosilca, N. Losada, W. Wu, D. Zhong, and J. Dongarra, "Evaluating data redistribution in parsec," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 8, pp. 1856–1872, 2021.

[36] M. Mongillo, "Choosing Basis Functions and Shape Parameters for Radial Basis Function Methods," *SIAM Undergraduate Research Online*, 2011.

[37] T. Mary, "Block Low-Rank Multifrontal Solvers: Complexity, Performance, and Scalability," Ph.D. dissertation, Paul Sabatier University, 2017.

[38] S. Le Borne and M. Wende, "Multilevel interpolation of scattered data using $\mathcal{H}$-matrices," *Numerical Algorithms*, vol. 40, no. 4, pp. 1497–1526, 2020. [Online]. Available: https://doi.org/10.1007/s11075-019-00860-1

[39] X. Lacoste, M. Faverge, G. Bosilca, P. Ramet, and S. Thibault, "Taking Advantage of Hybrid Systems for Sparse Direct Solvers via Task-based Runtimes," in *International Parallel & Distributed Processing Symposium Workshops*. IEEE, 2014.

[40] G. Pichon, E. Darve, M. Faverge, P. Ramet, and J. Roman, "Sparse Supernodal Solver Using Block Low-Rank Compression: Design, Performance and Analysis," *Journal of Computational Science*, 2018.