Performance Analysis of Tile Low-Rank Cholesky Factorization Using PaRSEC Instrumentation Tools

Qinglei Cao[1,5], Yu Pei[1,5], Thomas Herault[1,6], Kadir Akbudak[2,7], Aleksandr Mikhalev[2,7], 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]{herault, bosilca, dongarra}@icl.utk.edu
[7]{kadir.akbudak, aleksandr.mikhalev, hatem.ltaief, david.keyes}@kaust.edu.sa

Abstract—This paper highlights the necessary development of new instrumentation tools within the `PaRSEC` task-based runtime system to leverage the performance of low-rank matrix computations. In particular, the tile low-rank (TLR) Cholesky factorization represents one of the most critical matrix operations toward solving challenging large-scale scientific applications. The challenge resides in the heterogeneous arithmetic intensity of the various computational kernels, which stresses `PaRSEC`'s dynamic engine when orchestrating the task executions at runtime. Such irregular workload imposes the deployment of new scheduling heuristics to privilege the critical path, while exposing task parallelism to maximize hardware occupancy. To measure the effectiveness of `PaRSEC`'s engine and its various scheduling strategies for tackling such workloads, it becomes paramount to implement adequate performance analysis and profiling tools tailored to fine-grained and heterogeneous task execution. This permits us not only to provide insights from `PaRSEC`, but also to identify potential applications' performance bottlenecks. These instrumentation tools may actually foster synergism between applications and `PaRSEC` developers for productivity as well as high-performance computing purposes. We demonstrate the benefits of these amenable tools, while assessing the performance of TLR Cholesky factorization from data distribution, communication-reducing and synchronization-reducing perspectives. This tool-assisted performance analysis results in three major contributions: a new hybrid data distribution, a new hierarchical TLR Cholesky algorithm, and a new performance model for tuning the tile size. The new TLR Cholesky factorization achieves an $8\times$ performance speedup over existing implementations on massively parallel supercomputers, toward solving large-scale 3D climate and weather prediction applications.

Index Terms—Performance analysis, Profiling tools, Task-based programming model, Dynamic runtime system.

I. INTRODUCTION

Large-scale parallelism is the dominant force behind the improvement of scientific computing. High-performance computing (HPC) architecture development, designed to meet application requirements and achieve new performance levels, must address unprecedented increases in concurrency, heterogeneous hardware design and performance changes. However, as the number of nodes increases, so does the architectural complexity of each node, which makes programming efficiently for the target architecture challenging. Faced with such a daunting challenge, it is becoming increasingly clear that in order to execute at extreme scales, changes to programming model paradigms are needed to help applications to cope with these challenges.

It has been proven that a task-based approach is extremely efficient for load balancing and intelligently using all the resources computational power in heterogeneous platforms for many scientific computing fields—including application libraries built on top of the usual dense [1], [2] and sparse [3] linear algebra solvers with regular, arithmetic/memory-intense computational tasks. In a task-based programming environment, a large amount of parallelism is exposed by representing the algorithm as a continuous set of fine-grained tasks. Then, the runtime system is responsible for scheduling these tasks while satisfying the data dependencies between them. Such a runtime must adapt to the changes in the amount of parallelism available in applications and map tasks to underlying hardware resources under dynamic and unpredictable system conditions. `PaRSEC` [4] is one of the leading runtime systems that are being actively developed.

Although a task-based runtime system is convenient and efficient, from users' perspective a well-designed profiling system is needed to inspect the execution—especially when it suffers from subtle performance problems. The profiling system needs to integrate well with the runtime and be able to extract information that allows reasoning about task costs, scheduling quality, memory usage, and information regarding messages and data transferred in the network. In this paper, we introduce the profiling system of `PaRSEC`: the mechanisms embedded in the runtime system to extract critical information

and produce a trace of the execution, and the tools allowing users to manage this collection of events. Based on this profiling system, we demonstrate the optimization footprints towards Tile Low-Rank (TLR) Cholesky in Section 5 of [5] from a profiling and performance analysis perspective.

The remainder of this paper is as follows. Section II presents related work. Section III describes the necessary background for PaRSEC's dynamic runtime system as well as the TLR Cholesky factorization and the four optimizations where the performace analysis applied. Section IV introduces the design and the novel implementations of the profiling system in PaRSEC. Performance analysis of the incremental optimizations based on this profiling system are reported in Section V. We conclude our work in Section VI.

II. RELATED WORK

In this section, we focus on the profiling and performance instrumentation systems available on the most actively developed task runtime systems.

Legion includes a performance profiling tool, Legion Prof, that generates an in-house format log of the task system execution at runtime [6], [7]. This log can be converted to a set of dynamic HTML pages using a tool provided with the Legion distribution that shows utilization graphs of the memories and processors during the run. These webpages are dynamic, and more detailed information can be obtained, showing what tasks executed on what resource at what time, and showing part of the directed acyclic graph (DAG) connecting these tasks.

StarPU provides multiple approaches for performance analysis: on one hand, the analysis can happen online [8]: dynamic hooks are available for the application developer to connect to task- and communication-related events and write their own tracing or analysis mechanisms, or general statistics on the process status can be read at runtime (*e.g.,*, amount of computing time per core, time spent in the runtime system per core, etc.); or the performance analysis can be conducted offline after creating a trace of the execution [9]. StarPU uses the Fast User/Kernel Tracing (FxT [10]) library to create traces that can then be converted in DOT graph representations or the PAJE trace format [11]. The latter can then be visualized with the VITE tool [12] as a Gantt chart. StarPU provides additional tools to create text files describing the execution of each task in a key/value format for integration with external tools, and allows the user to build application-specific analysis. In [13], the author use a combination of the trace formats (PAJE, DOT, enriched text files), and ad hoc conversion scripts to build a CSV database of the execution and analyze it in R or other statistical tools, using application-specific methods. Last, a set of internal tools are also available in StarPU to measure the efficiency of the performance models built by the runtime system for its scheduling, and to check the accuracy of the simulations, if simulations are conducted with the runtime system.

To the best of our knowledge, QUARK does not provide any profiling or tracing tools within the runtime system to help the performance analysis. In [14], the authors instrumented manually each task in order to collect timing information and build Gantt diagrams and other performance analysis.

OmpSs includes a set of *instrumentation* plug-ins [15], that can be selected at run time, in order to dynamically call functions defined in these plugins when specific events occur. The set of events that trigger a call is controlled at compile time by a variety of options. Available plugins include an Ayudame plugin for the Temanejo graphical debugger [16], a module to compute and output the DAG of tasks, another one (experimental) to provide a trace for execution for a task system simulator, and a module to provide a trace for Paraver [17] that can potentially embed Performance Application Programming Interface (PAPI) counters information. In [18], the authors describe how the parallel trace can visualized as Gantt charts using Paraver, from a variety of perspectives (*e.g.,*, from a task view or a thread perspective, showing the Instruction per cycle achieved by different threads, or the TLB miss ratio).

The HPX Performance Counter Framework [19] defines an API to access internal counters exposed by the HPX runtime. These counters include information about the hardware, but also about the runtime status. Counters are addressed by their names, following a fixed naming scheme. The runtime provides rudimentary tools to regularly read a set of counters and display them on the standard output or send them to an output file, but this approach is only time-driven and does not allow for creating a trace of the execution. The preferred approach is to embed the user analysis directly within the HPX program, or to write our own tracing within the application for offline analysis.

As we describe below, the approach in PaRSEC differs from the other approaches in that a detailed trace of the execution is created and converted into an open format, which encourages the development of small and application-specific analysis tools in simple scripting languages. We illustrate below how this approach allows for taking an application written over PaRSEC and collect a trace of execution with enough fine details to allow a programmer with a good understanding of the application itself to identify the bottlenecks and solve them.

III. BACKGROUND

This section briefly provides background information on the PaRSEC dynamic runtime system, the TLR Cholesky factorization and the optimizations upon which the performance analysis is based. More information are detailed in Sections 4 and 5 of [5].

A. *The PaRSEC Runtime System*

PaRSEC [4] is a task-based runtime for distributed heterogeneous architectures and is capable of dynamically unfolding a description of a graph of tasks on a set of resources and satisfying all data dependencies by efficiently shepherding data between memory spaces (between nodes but also between different memories on different devices) and scheduling tasks

across heterogeneous resources. Domain-specific languages (DSLs) [20] in `PaRSEC` help domain experts to focus only on their domain science by masking required computer science knowledge. The Parameterized Task Graph (PTG) [21] DSL uses a concise, parameterized, task-graph description known as Job Data Flow (JDF) to represent the dependencies between tasks. Other DSLs, such as Dynamic Task Discovery (DTD) [22], are less science-domain oriented and provide alternative programming models to satisfy more generic needs by delivering an API that allows for sequential task insertion into the runtime.

B. `TLR` Cholesky Factorization

In the standard dense Cholesky factorization, data stored in the underlying tile layout is usually executed by four computational kernels: $POTRF$ (Cholesky factorization), $TRSM$ (triangular solve), $SYRK$ (symmetric rank k update), and $GEMM$ (general matrix multiply) on the lower or upper part of the symmetric matrix. The whole factorization translates into a DAG with nodes corresponding to tasks and edges representing data dependencies, with a serial and incompressible critical path of $(NT-1) \times (POTRF+TRSM+SYRK) + POTRF$, where NT is the number of row/column tiles.

The DAG of tasks (and thus the critical path) is the same in both `TLR` and dense Cholesky factorization, but there are two differences:

1) *data format*: all tiles are dense with size of $nb \times nb$ in dense Cholesky factorization, where nb is the tile size; while in `TLR` Cholesky factorization, only tiles on-diagonal are dense, and tiles off-diagonal are approximated up to the application-dependent accuracy threshold by using a variant of the singular value decomposition (SVD) with size of $nb \times rank$ ($rank << nb$ for tiles further away from the diagonal tiles);

2) *computational kernels, as well as arithmetic complexity, of $SYRK$ and $GEMM$*: to work on the compressed data layout of the off-diagonal tiles, the `HiCMA` library of `TLR` Cholesky mainly necessitates the developments of new low-rank LR_SYRK and LR_GEMM kernels, which requires decompression and recompression phases, respectively, as initially introduced in [23]; the arithmetic complexity is $2 \times nb^2 \times rank + 4 \times nb \times rank^2$ for LR_SYRK and $36 \times nb \times rank^2$ for LR_GEMM, instead of nb^3 for $SYRK$ and $2 \times nb^2$ for $GEMM$.

C. Evaluated Optimizations

To be self-contained, we briefly describe the four optimizations made in [5] that relate to the performance analysis applied using the profiling system of `PaRSEC` in Sections V-A to V-D below:

1) *Hybrid Data Distributions*, two intertwined 2D block cyclic data distribution using different process grids are superposed together, as shown in Fig. 2 of [5];

2) *Reduce Communication Volume*, communication is dynamically based on actual $rank$ instead of pre-defined $maxrank$ ($rank * nb$ instead of $maxrank * nb$ per communication);

3) *Lookahead to Emphasize the Critical Path*, a control dependency between tasks LR_SYRK and $TRSM$ of the same panel factorization delays the discovery of parallelism outside the critical path (corresponding to the update operation) to ensure the prioritization of the critical path; and

4) *Hierarchical $POTRF$*, hierarchically creating a node-local task pool that decomposes the $POTRF$ kernel on diagonal dense tiles into smaller subtiles to expose nested parallelism, and ensure work is available for all computational resources to speed up the critical path.

IV. PERFORMANCE TOOLS

`PaRSEC` features a rich development environment including tools to debug programs written in the different DSLs, and to profile the performance of task systems execution. We present in further detail the performance profiling and instrumentation capabilities of `PaRSEC` in this section.

A. Trace Collection Framework

The Trace Collection Framework sits at the root of the performance profiling system. It is part of the `PaRSEC` runtime system and can be enabled through a compile-time option. The framework consists of a runtime support thread and library that provides a generic API to define and store events that occur during the execution. The user program (typically the `PaRSEC` runtime and the different `PaRSEC` DSLs) defines events as identified entities that executed on a given thread at a given time, and is bound with a contiguous structure of arbitrary size that holds information pertaining to the event. For example, for each task of the PTG DSL, the DSL defines task-start and task-end events that store the task class, task identifier, and parameters of the task.

Events are stored in a set of binary files, one per process of the application. In each file, events are grouped in buffers of fixed size, each buffer belonging to a given thread of the process. Buffers are linked one to another, creating as many linked lists of buffers as there were threads in the process during the execution.

The library is designed to be highly scalable for many-thread environments and to incur a minimal overhead when logging events. Logging an event consists of reading a timer, and copying the information related to the event (from a dozen bytes to a few hundreds, depending on the event type) in a buffer of memory that is memory-mapped onto the backend file that stores the binary trace. At runtime, each `PaRSEC` thread owns an independent buffer to log its events, in order to avoid sharing and atomicity issues. When a buffer is filled, the `PaRSEC` thread that is logging an event atomically swaps its current logging buffer with a fresh one. The helping thread that is part of the Trace Collection Framework continuously expands the backend file on which these buffers are mapped, and prepares in advance new buffers for the `PaRSEC` threads

to acquire when needed. The only thread-synchronizing operations occur when requesting a new buffer and releasing the current one, and different `PaRSEC` threads never interact on their tracing structures during the computation.

This approach relies on the availability of a few buffers of memory: one buffer per `PaRSEC` thread for the current buffer, and a few more that are allocated in advance to overlap I/O operations with logging operations. If the `PaRSEC` threads generate events faster than the operating system can truncate the backend file, map new area and unmap completed areas, the system will throttle, slowing down the logging operations in order to complete the preceding ones. This is usually entirely avoided when the backend file is stored on scalable or local I/O systems. The Tracing Framework helping thread is usually left unbounded, to steal idle cycles from computing threads, as all its time is spent waiting or blocking on I/O operations.

In an effort to improve portability with existing performance analysis tools for parallel applications, the tracing interface can also be configured at compile time to produce an OTF2 trace [24] of the execution. In this work, we focus on the binary trace collection and the conversion method described below to build our own ad hoc analysis tools.

B. PINS: *PaRSEC INStrumentation*

The Trace Collection Framework is used within the `PaRSEC` runtime through the `PaRSEC` INStrumentation (PINS) interface: different modules can register callbacks that typically log events in the trace, and are called when the execution reaches critical points in the code. PINS modules are exposed to the final user through the Modular Component Architecture (MCA [25]), and can be selected at run time to decide the type of information logged in the binary profile files.

Typically, PINS registers callbacks for all the important steps of a task life cycle: when it is created, when it becomes ready, when it is selected, potentially when it is assigned to an accelerator, when it starts and ends to execution, and when it enables successor tasks. There are also callbacks pertaining to the state of the runtime: when it allocates or frees resources, when network events are triggered, etc.

The MCA exposes different logging policies, available for the user: for example, the `pins_papi` module allows logging PAPI counters of the user's choice, in addition to basic tracing that records the time and thread that generated each event. This enables augmenting the trace with information on the state of the hardware at the time of the event.

C. Dependency Analysis

The events instrumentation allows us to measure the status of the system at critical times. In order to accomplish a full analysis of the behavior, it is often necessary to connect this information with the actual DAG of tasks that was executed. In order to achieve this, the events trace is completed with another file representing the dependencies as they are expressed to the runtime system in another set of files following the DOT syntax defined by the Graphviz software collection [26] for portability.

For all deterministic problems (typically when the DAG of tasks is not data-dependent, but is entirely defined by the parameters that instantiate it as is the case with the PTG DSL), the collection of the DAG can be done offline—not during the timing of the operation itself, thus completely avoiding the risk of impacting the execution. One DOT file per process is produced, as for the tracing mechanism, and all `PaRSEC` DSLs provide a unique naming of tasks that enables an internal tool to stitch the different DOT files to produce a single one that represents the entire distributed DAG of tasks.

D. Trace Conversion Tools

This is not the case for the binary trace: once a trace is generated, the user has access to a set of binary files, one per process in the application. The format of these files is not exposed to the user, as information in them is kept as close as possible to the architecture, in order to avoid conversion costs to produce a portable trace format during the execution. Timing information, for example, is architecture- and operating system–dependent; each architecture defines its own time reading routine. All information logged by the user (typically integers of various size to store the parameters, PAPI counters, etc.) is also kept in the architecture-specific storage.

As is often the case with tracing systems, a conversion step is necessary to obtain a portable and exploitable file format of the trace. During this step, the different binary files are also merged in a single file, appending the rank that produced the source binary file as an identifier for each record. For portability and ease of use reasons, `PaRSEC` has chosen to export the portable file format in Hierarchical Data Format (HDF5), following the structure required by the popular Pandas Library to describe Data Frames in HDF5.

HDF5 [27] is an open format, self-describing, and efficient to represent large data sets. The self-describing property of HDF5 enables exposing a large variety of data with minimal external documentation. Pandas [28] is a popular Python library providing high-performance, easy-to-use data structures and data analysis tools for the Python programming language. The goal behind these choices is to simplify writing ad hoc analysis tools tailored to their application, as is done in the following. The `PaRSEC` programming environment also provides tools to take the generated trace and convert it into a Gantt chart, or compute basic statistics.

The HDF5 file contains a few Pandas DataFrames or Series: a Series describes what event types have been registered with the application, and associates an identifier to them; another one collects all the architectural information, at the application level, per process and per thread. The largest DataFrame is a relational array that stores all the events logged during the execution (one per row), and provides a tabular view of each record, where the columns define the fields of the events. Some fields are common to all events (*e.g.,* timing of start and end, resource identifier that produced that event, etc.), and other

fields that correspond to the information logged by the PINS module on top of the event are event type–specific.

In order to simplify the development of ad hoc analysis tools in Python with Pandas, the `PaRSEC` environment also provides a library to read the DOT files that are generated into a NetworkX [29] representation that understands the naming scheme of the DSL used, and connects the tasks in the graph object with the records in the Pandas DataFrame. A user can then easily select an event, find the task that relates to it, explore its successors or predecessors and find events relating to those in the DataFrame. The case study in the rest of this paper makes use of this feature.

V. Performance Results and Analysis

The experiments are run on Shaheen II, a Cray XC40 system, which has 6,174 compute nodes, each with two 16-core Intel Haswell CPUs running at 2.30 GHz and 128 GB of DDR4 main memory. Intel compiler suite 18.0.1 along with sequential Math Kernel Library (MKL) version 2018.1 for optimized basic linear algebra subprograms (BLAS) and LAPACK kernels are used in the environment settings. All calculations are performed in double-precision floating-point arithmetic. In all experiments, numerical backward errors have been consistently validated against the application accuracy threshold to ensure correctness. In particular, we compress off-diagonal tiles and retain their most significant singular values (and associated vectors) above the accuracy threshold of 10^{-8}, which ultimately yields absolute numerical error of order 10^{-9} in the solution of linear system in Equation (2) in [5]. This 10^{-9} tolerance is sufficient to satisfy the prediction accuracy requirements of the 3D climate and weather prediction applications, as described in [30]. We employ a process grid $P \times Q$ across computational nodes and make it as square as possible, with $P < Q$ when this square is not possible. Performance analysis utilizing `PaRSEC`'s profiling system for the four optimizations in Section 5 of [5] as well as Section III-C are presented in the following Sections V-A, to Section V-D. To maintain a fair comparison and analysis between the experiments, the tile size for these four sections is chosen to be 2,700, and the three applications mentioned in Section 6.1 of [5], **syn-2D**, **st-2D-sqexp** and **st-3D-sqexp** are measured.

A. Evaluation of Hybrid Data Distributions

Hybrid Data Distributions, as described in Section III-C, is the mixture of two 2D Block Cyclic Data Distribution (2DBCDD). The purpose of Hybrid Data Distributions is to reduce imbalance in terms of computation and memory, caused by the rank disparities between tiles on and off diagonal, as tiles on diagonal are always dense of size $nb \times nb$ while the size of tile off diagonal is $nb \times rank$ (with $rank << nb$ for tiles further away from the diagonal tiles). With this hybrid data distribution, diagonal tiles will be spread into all processes $P \times Q$ instead of just a portion $max(P, Q)$. From a memory perspective, the memory storing diagonal tiles of a certain process will be decreased from $\frac{n \times nb}{max(P,Q)}$ to $\frac{n \times nb}{P \times Q}$, and this

TABLE I: Memory reduction for certain process by Hybrid Data Distributions for **st-2D-sqexp**, **st-2D-sqexp** and **st-3D-sqexp** of tile size 2700.

| No. of Nodes	Matrix Size	Memory Reduced (GB)
16	1080000	4.374
16	2160000	8.748
16	4320000	17.496
64	2160000	5.103
64	4320000	10.206
64	6480000	20.412

reduction does not relate to the kind of applications. That means the saved memory (in doubles) to store diagonal tiles for certain process will be:

$$\frac{n \times nb}{max(P,Q)} - \frac{n \times nb}{P \times Q} \tag{1}$$

We use the event *memory* in the profiling system to detail memory usage of both static matrix allocation and dynamic tempory buffers. Table I demonstrates the memory reduction for the three applications **syn-2D**, **st-2D-sqexp** and **st-3D-sqexp** with different numbers of nodes and matrix size. From the table, it is clear that the memory reduction has a linear dependence with matrix size n and negative correlation with the number of nodes. `PaRSEC`'s profiling system also provides the execution time for each process, as well as each thread, from which we extract the workload for each process to show load balancing. Fig. 1 depicts workloads for each process with and without the hybrid data distributions for **st-2D-sqexp**. If the distribution is the normal 2DBCDD with process grid 4×4, there are only 4 processes of process ID 0, 5, 10, and 15 hosting tiles on diagonal, which causes the imbalance observed in Fig. 1a with the rank hosting only tiles off diagonal.

B. Evaluation of Reducing Communication Volume

This section describes our analysis of the effect of reducing communication volume in Section 5.2 of [5], which sends the actual $rank$ instead of the pre-defined $maxrank$ of all off-diagonal tiles, by showing the rank distributions of off-diagonal tiles for matrices obtained from the profiling system in `PaRSEC`. We used the `PaRSEC` tracing framework API to register a new, application-specific type of event, and at the execution of each task, we logged the $rank$ of the tile on which the task was working. Once the trace was converted, we then wrote application-specific scripts to analyze the HDF5 file, and produce the figures.
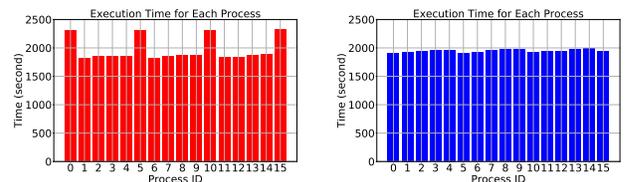
(a) Without the hybrid data distributions.

(b) With the hybrid data distributions.

Fig. 1: Process workload balancing for **st-2D-sqexp** on 16 nodes with process grid 4×4, matrix size $1080K \times 1080K$ and tile size 2700.

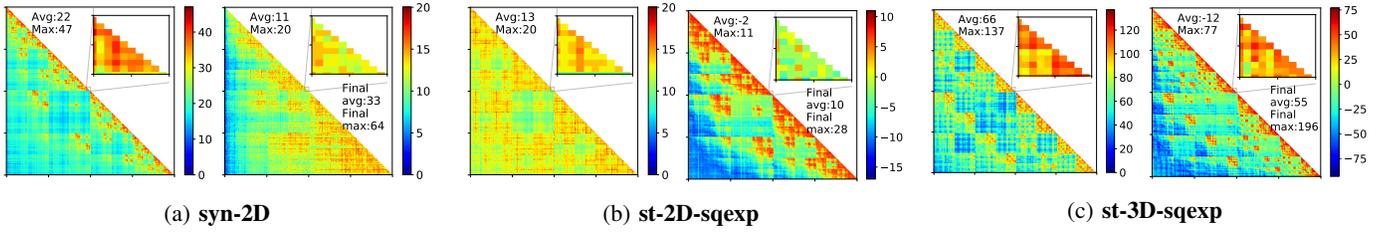| (a) **syn-2D** | (b) **st-2D-sqexp** | (c) **st-3D-sqexp** |

Fig. 2: Initial rank distributions (i.e., before factorization) on the left and the difference between initial and final ranks (i.e., after factorization) on the right; the matrix size is 1080K × 1080K, and the tile size is 2,700.

In Fig. 2, for each application, we show, as heatmaps, the initial rank distribution (i.e., before factorization) on the left, and the extent to which that rank changed by the end of the factorization on the right. Higher rank values are shown in red, and smaller rank values are shown in blue according to the colormap. The figures also display a zoomed area of diagonal tiles to better show the non-uniformity of ranks of tiles close to the diagonal, as well as the average and maximum rank for each configuration. In addition, the difference heatmap shows the average and maximum of the final rank distribution. As opposed to **syn-2D**, the two statistics applications show more discrepancy (more than $2.8\times$ in final average and maximum ranks) in rank distribution. In other words, the ranks in **syn-2D** are observed to be more homogeneous with respect to the statistics applications. Among the statistics problems, the difference between average and maximum ranks is the smallest for **st-2D-sqexp** and the largest for **st-3D-sqexp**, as seen in Fig. 2b and 2c. *Higher discrepancy in ranks results in higher imbalance in computation and communication. Hence, sophisticated task and data distribution heuristics and a dynamic runtime become important to efficiently solve such problems.*

C. Evaluation of Lookahead to Emphasize the Critical Path

This section provides details of performance analysis of the 'lookahead' technique to understand, using profiling tools, how `PaRSEC` executes tasks and how it emphasizes the critical path. By default, as mentioned before, `PaRSEC` eagerly tries to enable tasks to expose the maximum amount of parallel workloads. Although there is a ready queue scheme and priority policy, it can backfire when we overwhelm the scheduler with too many parallel tasks, resulting in delays of tasks in the critical path.

We profiled the execution to ensure that as soon as the data is ready, `PaRSEC` enables the critical tasks first. To be able to compute the average time it takes for data to be produced on one node and consumed on another, we need to connect the task termination, network activation, payload emission, and remote task execution events. This is provided by the `PaRSEC` profiling system through a combination of the trace information and the DOT file. Combining this information, we can identify the time at which diagonal tasks finished and the time when the following triangular updates start executing. Fig. 3 shows the time interval between receiving the diagonal data for $POTRF$ and the start of $TRSM$ tasks in the panels. This representation is application-specific, and is made possible by extracting the appropriate information

from the profile combined with the DAG information, using an application-specific Python script. In the default case, the tasks closest to the diagonal have the largest delay, but with our customized lookahead, we ensure that the critical tasks get executed as soon as the data dependency is fulfilled.

D. Evaluation of Hierarchical POTRF

To understand the impact of the hierarchical approach on the critical path, we compare in Fig. 4a the execution time of a $POTRF$ on a single tile with the corresponding hierarchical version. The hierarchical version using multiple cores is expected to be faster, but its performance is highly dependent on the number of available cores. The gray area encompasses the lower and upper bounds on the time for the hierarchical version according to the number of cores available during its execution.

To quantify these benefits in the context of a real execution, we exploited the basic timing information produced by the tracing system, and used the statistical packages provided by pandas and NumPy to compute our metrics: we compute the occupancy of the computational resources during the original run and then during the hierarchical $POTRF$ run. As described in Section 5.4 of [5], our assumption is that by increasing the parallelism during the dense diagonal $POTRF$, we make more efficient use of the available computational resources. Indeed, instead of waiting for the completion of a single, large granularity task, the computing resource can directly contribute to the diagonal $POTRF$. We compute the
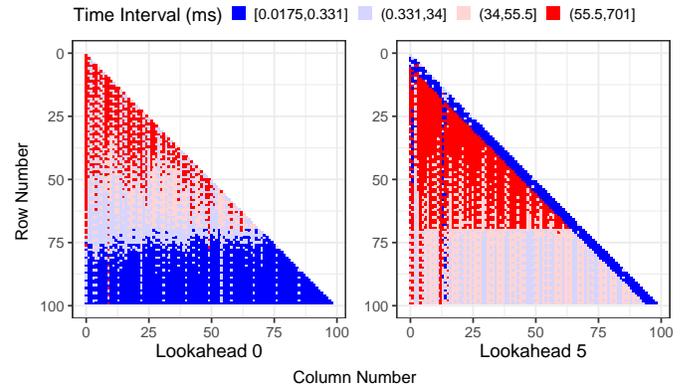

Fig. 3: Time between data is ready and $TRSM$ starts for **st-2D-sqexp**. Left, without lookahead; right, with lookahead of 5; each point represents one $TRSM$; matrix has 100×100 tiles.

occupancy by removing the waste (i.e., the time where computational resources are not actively involved in the execution of kernels) from the execution time.

The number of ready tasks (i.e., tasks with all data dependencies satisfied) decreases as we progress on the Cholesky factorization [31], so it is meaningful to look at different stages independently. Fig. 4b shows the resource occupancy during four different stages of the Cholesky factorization, comparing the hierarchical to the original version. The hierarchical version consistently provides better occupancy. However, as we get closer to the end of the factorization, where there is less potential parallelism in the original version, the hierarchical approach provides a significant boost to the occupancy and proves to be, as expected, an extremely beneficial optimization.

E. Modeling the Most Suitable Tile Size

In addition to the four optimizations mentioned before, we find that the tile size plays a significant role in TLR Cholesky in terms of operation balance between tiles on and off critical path, which could be a result of the profiling tools in PaRSEC using kernel execution time. For tile algorithms, finding the right tile size—the one that trades off performance and level of concurrency—is a critical step, as the tile size is a major factor in the algorithm performance [32]. Unfortunately, this "optimal" tile size depends on many factors other than the algorithm itself (e.g., the computing resources, computer and network performance and capabilities, available memory, matrix size). In addition, few observations are paramount to understand the correlation between tile granularity and performance. Smaller tiles further decrease the computational intensity of the mathematical kernels, while increasing the memory burden and the management overhead imposed on the supporting programming model and execution environment. Oppositely, while providing more computationally intensive operations, larger tiles decrease the degree of parallelism available, limiting the number of tasks that can run in parallel and therefore the resulting occupancy. Assuming square dense tiles, as long as the critical path of the algorithm and the computational costs of the involved tasks are known, the blocking and the distribution of the tiles can be found theoretically [31]. But TLR is not in a regular dense case, and the algorithm exacerbates the challenge of finding the right tile size, because the rank variability across tiles alters the balance of computations around the critical path, weakening the underlying assumptions of the existing optimality proof.
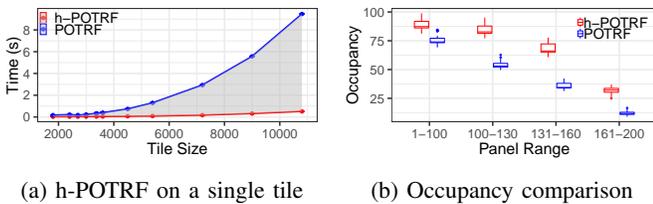
However, we can simplify the problem by reducing the impact of the imbalance introduced by low-rank by restricting the analysis of the critical path to a single tile outside the diagonal—basically providing a 1st-order approximation capable of predicting the most suitable tile size for TLR Cholesky, according to the problem size, the number of compute resources, and the average rank of the off-diagonal tiles. To the best of our knowledge, this is the first time such a theoretical approach has been proposed.

For a general parallel algorithm at a given problem size, we can guarantee an optimal time to solution if the serial part (the critical path in the algorithm) can perfectly overlap with the parallel part (everything outside the critical path). This is also true for TLR Cholesky, where there exists a well-known critical path that can easily be approximated. Thus—putting aside the overhead, hardware limitation, and dependency between the serial path (critical path) and parallel part—to get the best performance, the critical path (L) should be equal to the perfect scaling of the parallel part (D) to the number of available computing resources (C): $L = D/C$. Assume N is the matrix size, $node$ is the number of nodes, k is the average rank of tiles off diagonal; the approximation and proportionality of the best tile size nb can then be approximated as follows:

$$nb \approx \sqrt{\frac{3 \times N \times k \times (3 + \sqrt{9 + 32 \times node})}{4 \times node}} \quad (2)$$

$$nb \propto \sqrt{\frac{N \times k}{\sqrt{node}}} \quad (3)$$

This formula may not be exactly accurate, as it is an approximation, but it gives us insights into how to prune the range of tile sizes containing the optimal value, while enabling us to start with a more pragmatic approach for finding the best tile size by auto-tuning.

To validate our theoretical finding, we collected the tile size from a real execution, by extracting that information from the trace, and used brute force search to find the best tile size for a well-defined setup, N, k, and $nodes$, and compared this result with our approximation. Fig. 5a depicts the execution time for a fixed-size problem depending on the tile size. The proposed model, depicted by the red dashed line, is close to the minima of the black curve. Fig. 5b extends this to a larger

(a) h-POTRF on a single tile   (b) Occupancy comparison

Fig. 4: Impact of Hierarchical POTRF: (a), execution time on a single node; (b), resource occupancy of 540K × 540K matrix on a 3 × 3 process grid with a tile size of 2,700.

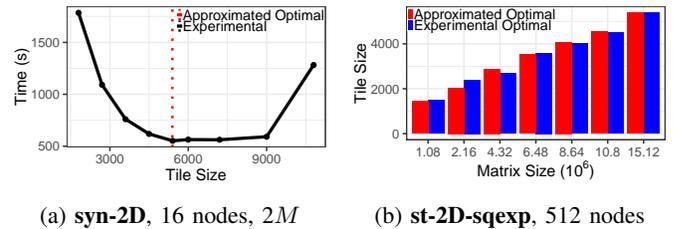(a) **syn-2D**, 16 nodes, $2M$ (b) **st-2D-sqexp**, 512 nodes

Fig. 5: (a) Performance variation with different tile sizes; the red dashed line is the approximated optimal tile size, while the black line the experimental results. (b) Comparison of approximated and experimental optimal tile size.

set of matrices, and highlights the fact that for all cases the approximated value is close to that found experimentally.

VI. CONCLUSION

In this paper, we present the profiling system of `PaRSEC`: the mechanisms embedded in the runtime system to extract critical information and produce a trace of the execution, and the tools allowing users to manage this collection of events. Using the information provided by this profiling system, we demonstrate the performance analysis to show optimization footprints of `TLR` Cholesky factorization from data distribution, communication-reducing and synchronization-reducing perspectives—which accordingly highlights the benefits of `PaRSEC`'s instrumentation tools, providing insights into PaRSEC, details during execution and a good understanding of the application itself, and identifying potential performance bottlenecks. In fact, these contributions may impact the development of a broader class of algorithms in low-rank matrix computations, beyond the herein studied `TLR` Cholesky factorization.

REFERENCES

[1] E. Agullo, J. Demmel, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, H. Ltaief, P. Luszczek, and S. Tomov, "Numerical Linear Algebra on Emerging Architectures: The PLASMA and MAGMA Projects," *Journal of Physics: Conference Series*, vol. 180, 2009.

[2] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, A. Haidar, T. Hérault, J. Kurzak, J. Langou, P. Lemarinier, H. Ltaief, P. Luszczek, A. YarKhan, and J. Dongarra, "Flexible Development of Dense Linear Algebra Algorithms on Massively Parallel Architectures with DPLASMA," in *IPDPS Workshops*. IEEE, 2011, pp. 1432–1441.

[3] 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 *IEEE International Parallel & Distributed Processing Symposium Workshops (IPDPSW)*, May 2014, pp. 29–38.

[4] 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*, vol. 99, p. 1, 2013.

[5] 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."

[6] S. J. Treichler, "Realm: Performance portability through composable asynchrony," PhD Dissertation, Standford University, Dec 2016, https://legion.stanford.edu/pdfs/treichler_thesis.pdf.

[7] Legion Team, "Legion: Performance profiling and tuning," September 2019, https://legion.stanford.edu/profiling/.

[8] StarPU team, "StarPU: Online performance tools," September 2019, http://starpu.gforge.inria.fr/doc/html/OnlinePerformanceTools.html.

[9] ——, "StarPU: Offline performance tools," September 2019, http://starpu.gforge.inria.fr/doc/html/OfflinePerformanceTools.html.

[10] B. Russel, V. Danjean, and S. Thibault, "Fast user/kernel tracing," Septembre 2019, https://savannah.nongnu.org/projects/fkt/.

[11] J. C. de Kergommeaux, B. Stein, and P. Bernard, "Paj, an interactive visualization tool for tuning multi-threaded parallel applications," *Parallel Computing*, vol. 26, no. 10, pp. 1253 – 1274, 2000.

[12] K. Coulomb, M. Faverge, J. Jazeix, O. Lagrasse, J. Marcoueille, P. Noisette, A. Redondy, and C. Vuchener, "Visual trace explorer (ViTE)," Technical report, Tech. Rep., 2009.

[13] V. Garcia Pinto, L. M. Schnorr, L. Stanisic, A. Legrand, S. Thibault, and V. Danjean, "A Visual Performance Analysis Framework for Task-based Parallel Applications running on Hybrid Clusters," *CCPE*, vol. 30, 2018.

[14] B. Haugen, S. Richmond, J. Kurzak, C. A. Steed, and J. Dongarra, "Visualizing execution traces with task dependencies," in *Proceedings of VPA'15*, 2015, pp. 2:1–2:8.

[15] OmpSs Team, "OmpSs: Instrumentation modules," September 2019, https://pm.bsc.es/ftp/ompss/doc/user-guide/run-programs-plugin-instrument.html.

[16] S. Brinkmann, J. Gracia, and C. Niethammer, "Task Debugging with TEMANEJO," in *Tools for High Performance Computing 2012*. Berlin, Heidelberg: Springer, 2013, pp. 13–21.

[17] V. Pillet, V. Pillet, J. Labarta, T. Cortes, T. Cortes, S. Girona, and S. Girona, "PARAVER: A tool to visualize and analyze parallel code," CEPBA/UPC Report No RR-95/03 February 1995, Tech. Rep., 1995.

[18] H. Servat, X. Teruel, G. Llort, A. Duran, J. Giménez, X. Martorell, E. Ayguadé, and J. Labarta, "On the instrumentation of OpenMP and OmpSs tasking constructs," in *Euro-Par'12 Wksh*, 2013, pp. 414–428.

[19] P. Grubel, H. Kaiser, K. Huck, and J. Cook, "Using intrinsic performance counters to assess efficiency in task-based parallel applications," in *IPDPS Workshops*, May 2016, pp. 1692–1701.

[20] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, T. Herault, and J. Dongarra, "PaRSEC: Exploiting heterogeneity to enhance scalability," *Computing in Science Engineering*, vol. 15, no. 6, pp. 36–45, Nov 2013.

[21] A. Danalis, G. Bosilca, A. Bouteiller, T. Herault, and J. Dongarra, "PTG: An Abstraction for Unhindered Parallelism," 2014, pp. 21–30.

[22] 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, pp. 6:1–6:8.

[23] 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, Frankfurt, Germany*. Springer International Publishing, 2017, pp. 22–40.

[24] D. Eschweiler, M. Wagner, M. Geimer, A. Knüpfer, W. E. Nagel, and F. Wolf, "Open trace format 2: The next generation of scalable trace formats and support libraries." in *PARCO*, vol. 22, 2011, pp. 481–490.

[25] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine *et al.*, "Open MPI: Goals, concept, and design of a next generation MPI implementation," in *European Parallel Virtual Machine/Message Passing Interface Users Group Meeting*. Springer, 2004, pp. 97–104.

[26] J. Ellson, E. Gansner, L. Koutsofios, S. C. North, and G. Woodhull, "Graphvizopen source graph drawing tools," in *International Symposium on Graph Drawing*. Springer, 2001, pp. 483–484.

[27] M. Folk, G. Heber, Q. Koziol, E. Pourmal, and D. Robinson, "An overview of the HDF5 technology suite and its applications," in *Proceedings of the EDBT/ICDT 2011 Workshop on Array Databases*. ACM, 2011, pp. 36–47.

[28] W. McKinney, "pandas: a foundational python library for data analysis and statistics," *Python for High Performance and Scientific Computing*, vol. 14, 2011.

[29] A. Hagberg, P. Swart, and D. S Chult, "Exploring network structure, dynamics, and function using NetworkX," Los Alamos National Lab.(LANL), Los Alamos, NM (United States), Tech. Rep., 2008.

[30] S. Abdulah, H. Ltaief, Y. Sun, M. G. Genton, and D. E. Keyes, "Parallel approximation of the maximum likelihood estimation for the prediction of large-scale geostatistics simulations," in *2018 IEEE International Conference on Cluster Computing*. IEEE, 2018, pp. 98–108.

[31] A. Buttari, J. Langou, J. Kurzak, and J. Dongarra, "A Class of Parallel Tiled Linear Algebra Algorithms for Multicore Architectures," 2007. [Online]. Available: http://www.netlib.org/lapack/lawnspdf/lawn191.pdf

[32] J. Kurzak, H. Anzt, M. Gates, and J. Dongarra, "Implementation and Tuning of Batched Cholesky Factorization and Solve for NVIDIA GPUs," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 7, pp. 2036–2048, July 2016.