2022 IEEE/ACM International Workshop on Runtime and Operating Systems for Supercomputers (ROSS) | 978-1-6654-7566-2/22/$31.00 ©2022 IEEE | DOI: 10.1109/ROSS56639.2022.00009

Sequential Task Flow Runtime Model Improvements and Limitations

Yu Pei, George Bosilca, Jack Dongarra
University of Tennessee, Innovative Computing Laboratory, USA

***Abstract*—The sequential task flow (STF) model is the mainstream approach for interacting with task-based runtime systems, with StarPU and the Dynamic task discovery (DTD) in PaRSEC being two implementations of this model. Compared with other approaches of submitting tasks into a runtime system, STF has interesting advantages centered around an easy-to-use API, that allows users to expressed algorithms as a sequence of tasks (much like in OpenMP), while allowing the runtime to automatically identify and analyze the task dependencies and scheduling.**

In this paper, we focus on the DTD interface in PaRSEC, highlight some of its lesser known limitations and implemented two optimization techniques for DTD: support for user level graph trimming, and a new API for broadcast read-only data to remote tasks. We then analyze the benefits and limitations of these optimizations with benchmarks as well as on two common matrix factorization kernels Cholesky and QR, on two different systems Shaheen II from KAUST and Fugaku from RIKEN. We point out some potential for further improvements, and provided valuable insights into the strength and weakness of STF model. hoping to guide the future developments of task-based runtime systems.

***Index Terms*—Dynamic Runtime Systems, High Performance Computing, Numerical Linear Algebra**

I. Introduction

As current High performance computing (HPC) systems reach exascale, the number and complexity of nodes (equipped with GPU or other types of accelerators) makes programming and optimal performance with the MPI+X model more difficult. Here MPI+X refers to directly calling MPI for internode communication and on a given node, we can deploy Pthreads, OpenMP, or CUDA/HIP for accelerators. Task-based runtime systems have been developed to manage the challenge of programming at this scale. In this programming model, the runtime is in charge of scheduling the computational tasks on parallel computing resources, as well as the communication between nodes. The user needs to decompose the algorithm into tasks with explicitly specified data dependencies among them, forming a Direct Acyclic Graph (DAG). This programming paradigm was adopted in recent years by many different systems, among them StarPU [1], Legion [2] and PaRSEC [3]. Even within task-based runtime systems, various methods exist to express the algorithm as a task-graph, and subsequently to analyze and schedule the resulting DAG onto HPC systems.

PaRSEC PTG interface [4] provides a domain specific language (DSL) to describe the algorithm by specifying the individual tasks and the data dependencies between those tasks. Given this compact representation of the graph, each node can process the tasks that will be executed on that node. Additionally, nodes can react to the data received from remote nodes without the overhead of building the global knowledge during execution. The STF model provides an easy-to-use API for algorithm formulation for StarPU and PaRSEC DTD interface [5]. A single thread of execution is responsible for inserting the tasks following the sequential execution order of the algorithm. The data usage information is provided (either READ, WRITE, or READ-WRITE) so that internally-independent tasks can be scheduled in parallel. Dependent tasks will follow the correct read-after-write orders. Data usage across nodes will trigger the corresponding data transfers and have them inserted in to the scheduling flow.

Such an interface is easy to use, but the granularity of the tasks needs to be sufficiently high to overlap with the dependency analysis, and this was demonstrated in [5], [6]. This analysis overhead increases equally with the problem size on all the processes. This means that the STF model will face significant scalability issues especially in the exascale era, where the number of compute nodes can be in the range of hundreds of thousands. Still, when making the transition to programming using a task-based runtime system, the STF model is a very attractive target for the new adopters and they can obtain comparable performance to the more commonly used MPI+X model when running on smaller scale [7].

Also unlike in PaRSEC PTG model where parallelism is unleashed eagerly, and can lead to erroneous scheduling decisions: certain control flows are needed to enforce task execution priority. STF models will usually have a parameter specifying the size of the window into the global graph of tasks. It is included with the primary goal of limiting the memory usage resulting from graph exploration. The main thread will keep inserting tasks up-to the window size, then join the computation threads for task execution. When the number of tasks decreases below a prescribed threshold the main thread would go back to the task insertion mode. This window size is a tunable parameter and has the side benefit of acting similarly to the lookahead technique that is common in matrix factorization implementations. It allows the task execution to follow the critical path of the algorithms, ensuring optimal scheduling for the user.

Given the benefits of the STF model, we would like to push the limits of STF model to achieve better scalability and performance, while balancing the ease of use of the model. Previous works have tested similar ideas on Cholesky factorization [8] [9], but we would argue that the dependency graph is relatively simple in the previous studies. As a result,

978-1-6654-7566-2/22/$31.00 ©2022 IEEE
DOI 10.1109/ROSS56639.2022.00009

we also implemented and evaluated trimming and broadcast's impact on QR factorization, which has a tighter dependency graph.

The contributions from this work are:

- Create the sender/receiver key internally to PaRSEC DTD so that user can trim the DAG during task insertion.
- Adopt a two-stage approach for data broadcast operation for the PaRSEC DTD interface.
- Evaluated empirically the changes in writing the algorithms in order to trim the DAG, and the usability of such an interface for more complicated algorithms.
- Evaluated the impacts of graph trimming and broadcast operation on performances of Cholesky and QR factorizations on two HPC systems at scale.

The rest of the paper is organized as follows: Section II introduces the related work. Section III outlines the original design of PaRSEC and PaRSEC DTD interface, and the new implementation to enable graph trimming and broadcast operation. Section IV details the changes to the Cholesky and QR factorization algorithm to have graph trimming and broadcast enabled (and empirical evaluation of the difficulty of user trimming). Section V we analyzed the impact of broadcast on data propagation benchmark, then trimming and broadcast on the factorization algorithms. In section VI we conclude the paper and points out some future work directions.

II. Related Work

A. Dynamic Runtime Systems

To adapt to the rapidly changing and heterogeneous HPC systems while being able to maintain performance portability, task-based dynamic runtimes have been introduced to act as a middle layer on top of multi-threading and MPI communications to schedule fine-grained computational tasks onto the underlying hardware resources. They execute tasks in an asynchronous fashion and break out from the overly constraining bulk-synchronous programming model. They target shared and distributed-memory systems, possibly equipped with GPU accelerators. They reduce process idle time during the execution of imbalanced workloads [10]. They implement various scheduling heuristics to reduce remote and expensive data movement, while favoring data locality. As a result, users programming in this model can separate the problem formulation and performance tuning, and be able to achieve performance portability across machines. In particular, PaRSEC DTD, StarPU and OpenMP [11] provide a convenient task-insertion API or pragma that abstracts the hardware complexity. The user is still in charge of ensuring sequential numerical correctness of the task-based code before these runtimes proceed with the scheduling onto parallel resources. This separation of concerns has enabled wide adoption of these runtimes in the community. These runtimes build the DAG dynamically and unroll it as computational progress occurs. PaRSEC PTG adopts a different approach where the task graph unrolling is done via a high-level description of data dependencies between tasks. There are numerous other runtimes (HPX [12], Charm++ [13], etc.) that employ asynchronous many-tasking executions. In this paper, we focus on PaRSEC DTD as a case of STF model, which shares a lot of the common infrastructures with PaRSEC PTG interface.

B. Numerical Linear Algebra

As a foundational components of many HPC applications and machine learning operations, optimal numerical linear algebra routines are key to efficient system utilization. Numerous libraries provide dense linear algebra routines. Since its initial release nearly 30 years ago, LAPACK [14] has become the de facto standard library for dense linear algebra on a single node. It leverages vendor-optimized BLAS for node-level performance, including shared-memory parallelism. ScaLAPACK [15] built upon LAPACK by extending its routines to distributed computing, relying on both the Parallel BLAS (PBLAS) and explicit distributed-memory parallelism. Some attempts have been made to adapt the ScaLAPACK library for accelerators, but these efforts have shown the need for a new framework. More recently, the DPLASMA [16] and Chameleon libraries [17] both build a task dependency graph and launch tasks as their dependencies are fulfilled. This eliminates the artificial synchronizations inherent in ScaLAPACK's design, and allows for overlap of communication and computation. DPLASMA relies on PaRSEC PTG or DTD to specify and schedule tasks, while Chameleon can use either StarPU or PaRSEC runtime. And they both support GPU-based task executions. SLATE [18] is a recent effort to implement the linear algebra routines in the distributed settings with the goal of replacing ScaLAPACK. It uses modern C++ framework and MPI+OpenMP model, with support for modern accelerated architectures.

In this paper, we showcase two linear algebra algorithms to motivate the need for graph trimming and broadcast features that we added within the DTD interface. Although it is different for each specific application, these two dense solvers are representative to illustrate the user level changes and the impacts from these additional features.

III. User Graph Trimming and Broadcast Operation

A. DTD Model

PaRSEC as a task-based runtime supports multiple interfaces, the PTG interface requires the users to specify the body of the tasks, and the dependencies between tasks via the Job Data Flow (JDF) DSL. On the other hand, DTD interface allows users to write sequential-looking code, including conditionals, for-loops, and code blocks to insert tasks using PaRSEC's API without using a custom DSL. Both methods of task graph definition share the same runtime scheduler, data representation, and communication engine. There are three main concepts that enable expression of a task graph in PaRSEC using DTD: a task, dependency, and data item. A task is any kind of computation that will not block due to communication, data items are regions of main memory used by the computations that will be accessed or modified, and,

finally, dependencies are the ordering relationships between tasks in the graph. To insert a task with any of PaRSEC's API options, users must indicate the data and the mode of operation that will be performed on that data by the task (either read, write or read-write). Dependencies between tasks are created based on the operation-type on the data: a task performing a write before a task performing a read on the same data will create a read-after-write (RAW) dependency between the writing task and reading task, such that the reading task will only execute after the writing task is completed. The properly sequenced expression guarantees the correct ordering of tasks regardless of the parallel execution and any data concurrency interaction.

In distributed memory systems all the participating processes need to have a consistent view of the DAG for DTD to maintain the correct sequential order of tasks and requiring the whole DAG to be discovered by all the processes is one solution. This means that many tasks that are not related to a given node will still need to be inserted and inspected, creating a growing overhead as more nodes are involved. With this kind of implementation guaranteeing the insertion of the entire graph, it allows for creating a *unique key* and thus a consistent naming of each task on all the nodes without involving extra communication. This approach allows for simple message matching across nodes, based on this naming scheme and its unique keys. This is a sufficient solution, but is a stronger requirement than what is needed for the STF model.

B. PaRSEC DTD Tasks and Communications Tracking

PaRSEC communication engine is exposed to the rest of the runtime only through a well-defined interface. The DSLs encapsulate the information of a communication via an object called remote_deps (the circle in Figure 1) for remote dependencies that is passed into the engine. This abstraction allows PaRSEC to adopt different underlying libraries (right now it uses MPI two-sided) for communication. When we are inserting the tasks, DTD keeps track of the remote parent task or the received remote_deps object in a local hash table with the unique key for a given task. Since each task in the entire DAG has a unique key, the communicated data represented in the remote_deps can be matched with the remote task object and continue the task graph execution.

C. Graph Trimming

This unique key generated independently on each node is the link between the task management level and the underlying communication engine. The correct message carrying the data will be provided as the input data to the corresponding task via the key generated independently on each node. By observing that for each send-receive pair of exchanging data between two participating nodes, they only need to keep track of the order of the previous communication instances between the two, then they can correctly generate the next key for the point-to-point transfer between the two. So to remove this artificially stronger requirement of inserting all tasks and labeling them uniquely, thus permitting user level graph trimming, each node keeps internal arrays that will track the sends and receives with respect to other ranks instead. With this approach, users can trim the task graph at the user-level transparently, reducing the overhead of the runtime scheduling and improving performance (an example of the arrays is shown in Figure 1). This change does not affect the existing code that inserts all tasks on each node, since the irrelevant tasks that get trimmed will never have the data IDs assigned to them and thus will not affect the correct ID assignment for retained tasks.

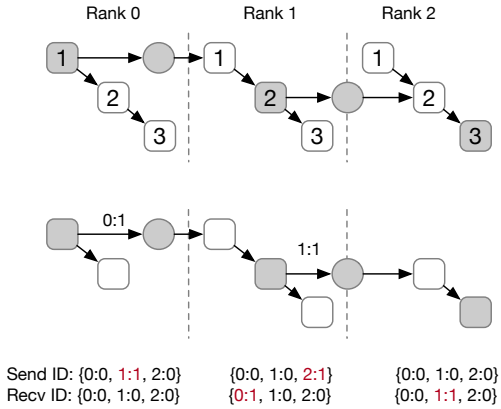

Fig. 1: Top: original DTD, each task has a unique key Bottom: send/recv level key. Grey square represents local task, white square represents remote task. Circle represents the remote_deps structure. In the new scheme, data flow ID is a combination of sender rank and sequence number to uniquely label each data transfer. As long as both the sender and the receiver has the dependent tasks inserted, the data ID will be assigned correctly for the two sides to match the data transferred.

D. Broadcast Operation

Collective operations are a critical part of message delivery optimization, especially for large-scale distributed systems. In a typical MPI-based program, collective operations are done via a predefined communicator, and as a result all the callers know the participant ranks. In a sequential task insertion interface like PaRSEC DTD, tasks are inserted sequentially and the group of nodes/processes/ranks participating in a collective operation are not known beforehand. Previous work [9] implemented implicit broadcast, assuming all the participants are discovered when the data is ready to be send (i.e. the broadcast will cover all the descendant ranks or most of the ranks). The benefit of this approach is that it is transparent to the application writer, your original STF code will benefit without any changes. But the assumption that the task discovery progresses faster than the kernel execution turns out to be a strong one, and risks the possibility of lacking ability to identify collective operations and falling back to doing point-to-point communication. We propose an explicit broadcast API, whereby with the knowledge of the

algorithm writer, the root of a broadcast call can specify all the participating ranks (the dependent tasks that will use the data), and the participants don't need to know each other. This is the same kind of information that is needed when the user trims the task graph with remote read tasks not knowing each other but will have the same writer task in the root inserted. Also, with an explicit collective API, many similar collective calls can be implemented (reduction/allreduce, gather/allgather etc).

PaRSEC PTG and DTD models share the same underlying communication engine, and a version of broadcast has already been implemented for PTG. It is built on top of the MPI point-to-point operations. Since for PTG, the entire graph's information is represented locally, a descendant can replay the task schedule as the root in order to rediscover the participant ranks and the propagation path, thus can continue the data broadcast downstream. There are two different typologies supported, namely: chain and binomial trees. The mechanism to check for direct descendants is based on a bit array representing participant ranks and, as a result, the route will be fixed given a topology and a set of participant ranks.

Since both DSL variants share the same communication engine, the idea then is to adopt a two-stage approach (Figure 2) and reuse many of the same implementations. In this scheme, we first prepare a message containing a global ID, local data keys (the P2P keys between the root and each children) and participating ranks as the first step. This is the propagation of the metadata information representing the broadcast. In the PTG case, we can query the parameterized graph information to obtain this knowledge, but in the STF model, the parent needs to inform the descendants of the global knowledge coming from the root. This metadata is matched via the point-to-point data keys between the root and each of the descendants. For an intermediate node, once it has received the metadata, it can act as the root to continue the propagation of metadata. The actual data broadcast will use the global ID to progress as an independent second step. This is possible because after the first step completes and the global ID is known, the communication engine can match the data received using this ID. By populating the metadata received into the outgoing message, DTD broadcast can reuse PaRSEC collective implementation to continue message propagation using the selected topology.

IV. Evaluation of the Programming Model with Cholesky and QR Factorization

Algorithm 1: Pseudo-code of Cholesky Factorization.

```
for k = 0 to NT − 1 /* Panel Factorization (PF) */
    POTRF(C_kk^RW)
    for m = k + 1 to NT − 1
        TRSM(C_kk^R, C_mk^RW)
    for m = k + 1 to NT − 1
        SYRK(C_mk^R, C_mm^RW)
    for m = k + 2 to NT − 1 /* Trailing Submatrix Update */
        for n = k + 1 to m − 1
            GEMM(C_mk^R, C_nk^R, C_mn^RW)
```

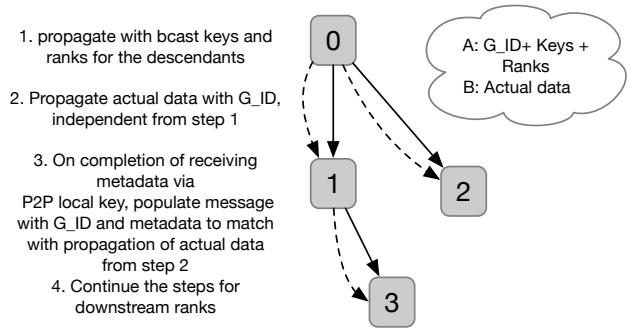

Fig. 2: Two-step broadcast with meta-data transfer as the first, and data payload transfer as the second. They propagate as two separate flows but data reception call can only be matched when the meta-data is received and global ID is known. (The number is the rank location of the task)

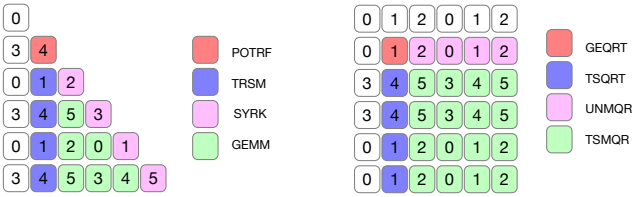

Fig. 3: The four different kernels from Cholesky and QR respectively. Both runs on a 2X3 compute grid with 2-D block cyclic distribution. For QR, a super-tiling of 2 is used on the grid row to reduce cross node P2P communication.

Cholesky and QR factorizations are classic linear algebra algorithms that are widely used for solving linear systems of the form $Ax = b$ with A having special numerical properties benefiting special algorithmic choices. For square matrix, their corresponding floating point operation (FLOP) counts are $\frac{n^3}{3}$ and $4\frac{n^3}{3}$. Their corresponding tile-based algorithms are listed in Algorithm 1 and 2, respectively. They both use four computational kernels that are successively applied on the trailing sub-matrix at each step, as illustrated in Figure 3 for matrices of 6×6 tiles at iteration $k = 2$. In practice, the implementation of these kernels relies on a BLAS library, such as MKL on Intel's x86 CPUs or SSL2 on Fujitsu A64FX CPUs.

Algorithm 2: Pseudo-code of QR Factorization.

```
for k = 0 to NT − 1
    GEQRT(C_kk^RW, T_kk^W)
    for n = k + 1 to NT − 1
        UNMQR(C_kk^R, T_kk^R, C_kn^RW)
    for m = k + 1 to MT − 1
        TSQRT(C_kk^RW, C_mk^RW, T_mk^W)
        for n = k + 1 to NT − 1
            /* Trailing Submatrix Update */
            TSMQR(C_kn^RW, C_mk^R, T_mk^R, C_mn^RW)
```

A. Modifications to the user code

STF model provides a simple-to-use programming interface, but as demonstrated before and later in this study, the task graph overhead will significantly increase as we increase the problem size because the number of tasks is proportional to the problem size when the tile size remains fixed. As a result, the graph trimming is a required step to include in order to achieve good performance at large system and problem scales. Based on the tiled algorithm of Cholesky and QR, here we describe how to both trim the task graphs and to incorporate explicit broadcast operations into the algorithms.

To ensure the correctness of the algorithm, the sender side needs to insert all the remote descendant tasks and on the receiver side, the remote data provider task needs to be inserted as well. For the Cholesky factorization without broadcast, this means that all TRSM tasks need to be inserted on the POTRF task node, and each of the nodes in the current panel need to insert the remote POTRF task in order to receive input data. On the receiving nodes, this means that other remote TRSM tasks can be trimmed (Figure 4, Left). Similarly, for the connections between TRSM and GEMM, each TRSM needs to insert all the GEMMs that are in the same row, as well as the GEMM tasks in the reflective column. On the receiver side, all the GEMM tasks will need to insert the two TRSM tasks from the given row/column. With an explicit API call to a user-level broadcast added, the expression of the program is changed. The destination ranks are iterated to create the metadata, and, as a result, the broadcast operation itself (yellow tasks in Figure 4, Right) can serve as the connection between the sender task and receiver tasks and we don't need to insert the tasks on the other side of the communication exchange, thus simplifying the trimming code.

For the QR algorithm, it has a tighter set of data dependencies between the tasks, where each row has data dependency on the previous row. As a result, for a trailing task TSMQR, we need to discover the TSQRT on that row as well as the UNMQR task or the previous rows' TSMQR task in order to correctly obtain the input data. In the case of 2-dimensional block cyclic data distribution with $P \times Q$ number of nodes (usually with super-tiling on P to reduce row level communication frequency), we only need to insert $(P + Q)/(P \times Q)$ number of the original TSMQR tasks. For broadcast operations, the opportunities are limited in the QR algorithm, as the row-by-row updates naturally translate to point-to-point operations. The only possible broadcasts are the propagation of panel data across a given row of Q processes, either for GEQRT to UNMQR, or TSQRT to TSMQR (Figure 5).

B. Qualitative analysis

The major appealing factor for the STF model is that it is easy to use. Indeed, one can simply write the two algorithms following the pseudo-code with PaRSEC DTD and it will work out-of-the-box. The issue is that in order to obtain good performance and to avoid the overhead of traversing the entire task graph, the user needs to include many conditionals in the user code to evaluate whether we should insert a given task. Here, I will argue that this modification at the user level is not insignificant, rendering the STF model complicated to use (in some ways similar to the SPMD model). This is in contrast to previously brought up suggestions that this modification is easy and can be hidden. For data users, it can insert all the relevant remote tasks that will produce this data. For the data writer tasks, the algorithm writer needs to be aware of the users of output data tasks, and will need to insert those reader tasks correspondingly. In the case of Cholesky and QR factorizations, it is tractable, but when the algorithm becomes more complicated instead of trivially nested for-loops, we can imagine that trimming can produce very error prone codes.

This goes back to some of the difficulties in writing algorithms using PaRSEC PTG. One is that you need to write in a domain specific language, but more importantly, the user needs to think of the algorithm in terms of the DAG and to specify the data dependencies between the tasks explicitly. This includes all the data' input and data' output links of each of the tasks. But to trim the graph correctly, the algorithm writer is essentially expressing the same information as with PaRSEC PTG. As a result, I view the trimming optimization as trying to express the same information on these two interfaces, and they only differ as to when and where the users supply additional information about the relationships between tasks.

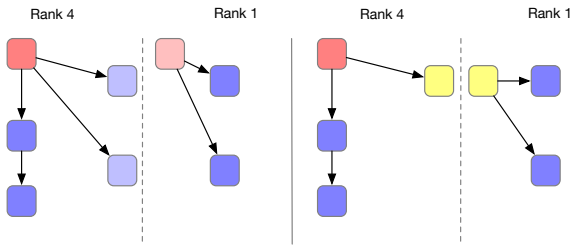

Fig. 4: Left, trimmed task graph without broadcast call; Right, explicit broadcast call to propagate POTRF data. Color scheme and data distribution follows that from Figure 3. Lighter red and purple represent remote tasks, yellow represents broadcast task. Data dependency between TRSM and GEMM omitted.

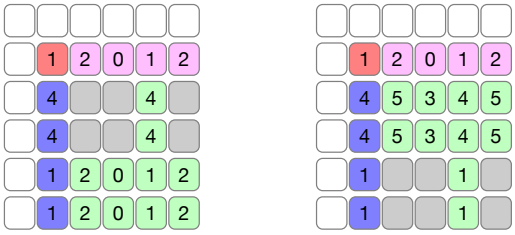

Fig. 5: Since only the TSMQR tasks are of order $O(N^3)$, we can insert all the other tasks in all the nodes while inserting TSMQR only on ranks that are in the same row or column of the current panel tasks. Left is for tasks inserted on rank 1, while right figure is for tasks on rank 4

V. Performance results and analysis

We implemented the features presented above in PaRSEC based on the branch from Nov, 2020. All the results presented in this paper use the IEEE 754 double precision variants DPOTRF and DGEQRF for Cholesky and QR, respectively. We run our experiments on two systems:

- **Shaheen II**, a Cray XC40 supercomputer with 6,174 nodes composed of two-socket 16-core Intel Haswell (AVX2) processor and 128GB of main memory, using the Cray Aries network interconnect. We use Cray MPI and Intel programming environment (MKL).
- **Fugaku system**, a Fujitsu ARM (SVE) system with A64FX nodes composed of four 12-core core memory groups (CMGs) and 32GB of main memory, connected through the TofuD interconnect. We use Fujitsu MPI and SSL2 library.

A. Broadcast benchmark performance

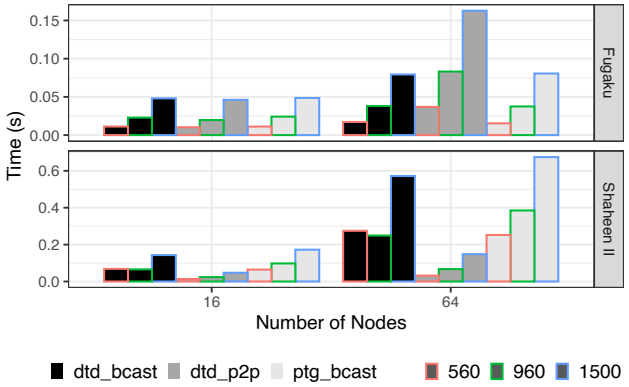

Fig. 6: Benchmark of a broadcast operation for sending a square tile of double floating points. We tested on two set of nodes, and varying the message data size. For comparison, we have the default DTD P2P, the proposed DTD broadcast and finally the broadcast utilized in PTG (the two shared the same mechanism).

We measure the message transfer time of one broadcast operation and compare it with using the default DTD point to point (P2P) to evaluate the benefit from doing the broadcast. We vary the number of nodes, as well as the size of the data we are sending to match with the square tile from Cholesky and QR. The two machines have different networks: Shaheen II uses Cray Aries Interconnect with Dragonfly topology with bandwidth of around 10 GB/s. Fugaku uses TofuD interconnect from Fujitsu with bandwidth of around 40 GB/s.

The results are shown in Figure 6. On Fugaku machine, the result followed our expectations. The broadcast can propagate the data equally or faster than P2P with the root sending the data to each of the descendant. Also, as the message sizes increased, so to did the entire data transfer times.

On Shaheen II, the point-to-point version could finish faster than the collective version for all message sizes. The reason for this is not known, but our hypothesis is that the difference in network topology alleviated the bottleneck of the P2P from the root node. In real applications, the situation can be complicated and the network state can change. For example, the computation threads can create memory contention and reduce network performance [19]. When employing broadcast, the operation can share the network usage across the nodes. Relying on a single root node for data transfers can potentially saturate a single node's outflow bandwidth.

B. Experiment performances

As the baseline to compare our achieved performance, we also ran the ScaLAPACK version of Cholesky and QR factorizations provided by the math libraries on the respective system (MKL from Intel on Shaheen II and SSL2 from Fujitsu on Fugaku). ScaLAPACK is a widely used library that provides distributed versions of common linear algebra operations and its optimized versions are provided by vendors.

Based on the previous descriptions, we implemented different versions of Cholesky, with graph trimming, broadcast operation, or a combination of both. We compared the performance of the different flavors of these algorithms with the original DTD as well as PTG implementations from DPLASMA. For the QR factorization, we have the trimmed-only version as well as trimming with broadcast version (since the broadcast-only version shows no improvement, it is not shown here). We obtained results for matrices varying in size from 100K to 600K, using two different tile sizes. Finally, we show the scalability of the implementations by running on 256 and 512 nodes.

1) Shaheen II results: The results from Shaheen II for the Cholesky and QR factorizations are shown in Figure 7 for 256 and 512 nodes. The black lines represent the results from ScaLAPACK with one MPI rank per core, block size of 64. We tested two different tile sizes, affecting the number of available tasks as well as the degree of parallelism. It should be first noted that overall, the performance from the runtime system-based implementations were better than the ones from ScaLAPACK, especially for the Cholesky factorization. Performance on a single node for DPOTRF is around 860 GFLOP/s, meaning that assuming the perfect scaling, we would have reached 220 TFLOP/s with 256 nodes. Conversely, in terms of the maximum performance achieved, the Cholesky factorization could reach a higher efficiency than QR. This was likely due to a larger degree of parallelism. ScaLAPACK QR performs very well as the problem size increases. Note that since QR has four times the FLOP of Cholesky, the actual execution time is longer for QR factorization.

Additionally, for the tile-based algorithms, the tile size needs to be tuned to obtain better performance. Optimal tile size was dependent on the interface we used and the balance between computation, communication, and runtime overheads. For most cases, a tile size of 560 outpreformed those of 960. However, for the QR implementation this is dependent on the problem size. As the matrix sizes increase, using tile size 560 we observe performance degradation instead of

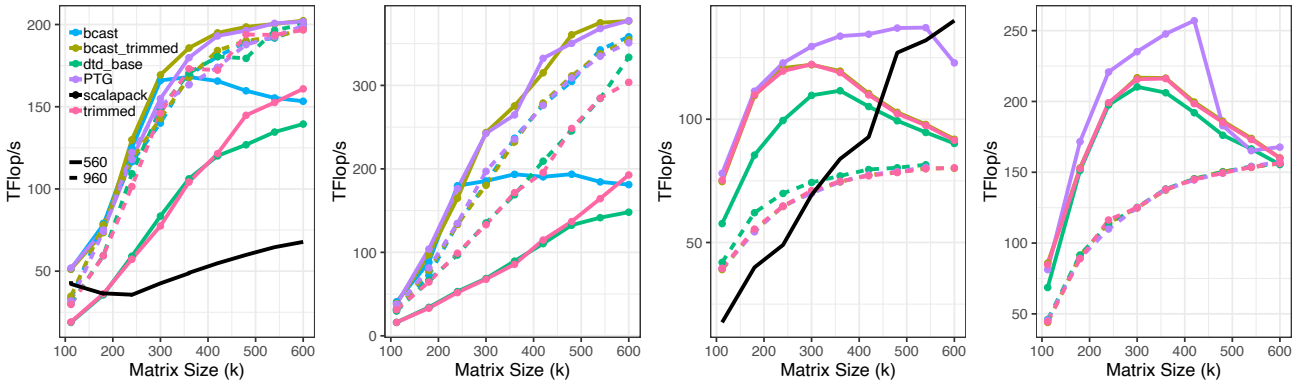

Fig. 7: Performance on Shaheen II. From left to right: Cholesky 256, 512 nodes; QR 256, 512 nodes

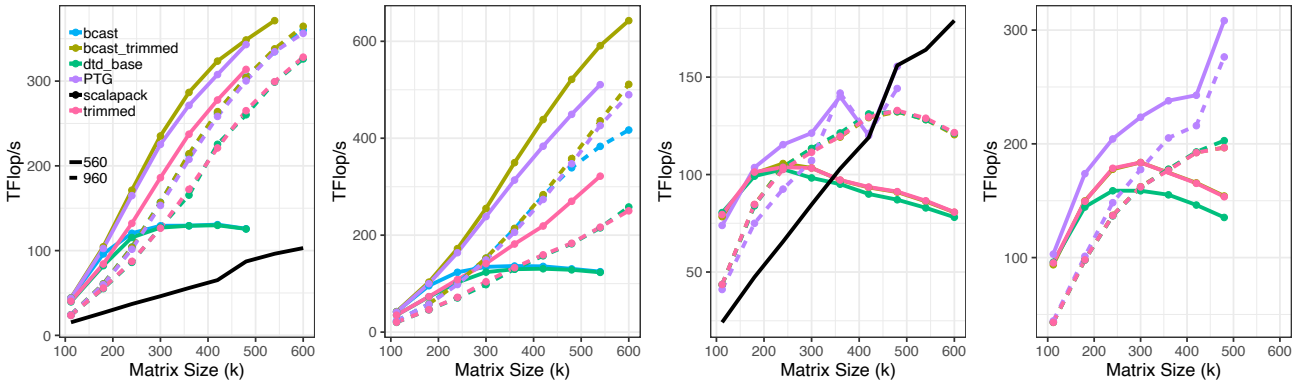

Fig. 8: Performance on Fugaku. From left to right: Cholesky 256, 512 nodes; QR 256, 512 nodes

stabilization, while a larger tile size of 960 shows performance improvements. We suspect this results from the overhead of task insertion and management. This would explain why the trimmed version of QR is faster than the base DTD version when tile size was 560, with the reduction of task analysis overhead.

The two user-level features, that we added, provided various degree of performance improvements. As the number of tasks is cubic with the number of tiles, we could have had 5-fold reduction in the number of tasks when the tile size was 960 instead of 560. As a result, the graph trimming is not providing as much of an impact as with the case of tile size 560 (trimming can reduce the number of inspected tasks by an order of magnitude). Adding broadcast for the Cholesky factorization provided a good performance gain in the case of tile size 560. However, the most gain came from combining the two (even more so than the PTG version of Cholesky implementation).

These two features also changed the optimal tile size for Cholesky from 960 to 560. A tile size of 560 is adequate for obtaining solid performance from Intel's MKL. We noted that the larger tile size was, the more likely it was to compensate for the higher overhead from base DTD overheads.

The interesting thing is that in Figure 6, the P2P is faster than broadcast but results from Figure 7 showed improved performance for Cholesky factorization. Other authors indicated [19] that computation can reduce network bandwidth due to memory contention, we think that the actual P2P bandwidth during Cholesky factorization is less than the benchmark measurement. The network degradation could be remedied by spreading the message propagation across the participating nodes via broadcast.

2) Fugaku results: Similarly, the results from Fugaku are in Figure 8 for 256 and 512 nodes. We generally observed the same trends as in the result from Shaheen II, with good scalability on both 256 and 512 nodes. On one node of Fugaku, we could obtain DPOTRF results of around 1700 GFLOP/s, meaning the result from 256 nodes would have had a ceiling of 435 TFLOP/s. The single node base is lower than other SSL2 results due to an issue calling SSL2 math library from multiple threads, and we had to disable the sector cache optimization to complete the runs. One difference is that the base DTD Cholesky was performing much worse relative to the ones from Shaheen II. And correspondingly, a much smaller effect was observed from just adding broadcast. With 48 cores instead of 32 from Shaheen II, insertion efficiency might have had a larger factor in order to saturate all the cores. And the trimming in this case provided a larger degree of

relieve to this bottleneck. With the two features combined, we actually obtained significantly better result for the Cholesky implementation in comparison with the PTG version from 512 nodes.

For the QR implementation, the broadcast-only version showed a minimum improvement effect, since the dependencies are tighter than for the Cholesky one. Although trimming can provide a small performance boost, we still need to increase the tile size to further reduce the overhead, this in turn diminishes the effect of trimming. In summary, further profiling is needed to understand the exact reason for the performance drop and where the limiting factors were coming from for DTD version.

VI. Conclusion and Future work

In this work, we introduced two new features to PaRSEC DTD interface, namely user level graph trimming and broadcast operations. We demonstrated the user level changes implemented to these two features with Cholesky and QR factorization. Our experience indicates that, although it is a straightforward task to trim the graph when the algorithm is simple, it can be complicated when the user needs to know the exact dependencies among the tasks, leading to messy and error-prone user codes. We also show that with these two added features, we can significantly improve the performance of Cholesky, while providing modest improvement for QR. From both the usability and performance standpoint, we have shown that the STF interface still has abundant opportunities for improvement, but will likely limit the usability of the original interface and create difficult to maintain and debug user code.

There are several opportunities for performance improvements at the implementation level. Including dynamic selection of collective topology, metadata caching and some PaRSEC internal implementation optimizations. Exploring the reason for QR performance drop is beyond the scope of this paper, requiring profiling analysis to understand the bottleneck in those cases and the likely overheads still in the runtime implementation. Additionally, further investigation is required to better understand the applicability of STF interface to a wider range of scientific applications.

Acknowledgment

For computer time, this research used the resources of the Supercomputing Laboratory (KSL) Shaheen II at King Abdullah University of Science & Technology (KAUST) in Thuwal Saudi Arabia and the supercomputer Fugaku provided by RIKEN.

References

[1] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "StarPU: a unified platform for task scheduling on heterogeneous multicore architectures," *Concurrency and Computation: Practice and Experience*, vol. 23, no. 2, pp. 187–198, 2011.

[2] 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*. IEEE, 2012, pp. 1–11.

[3] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, T. Hérault, and J. J. Dongarra, "PaRSEC: Exploiting Heterogeneity to Enhance Scalability," *Computing in Science & Engineering*, vol. 15, no. 6, pp. 36–45, 2013.

[4] A. Danalis, G. Bosilca, A. Bouteiller, T. Herault, and J. Dongarra, "PTG: an Abstraction for Unhindered Parallelism," in *2014 Fourth International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing*. IEEE, 2014, pp. 21–30.

[5] 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.

[6] E. Slaughter, W. Wu, Y. Fu, N. Garcia, W. Kautz, E. Marx, K. S. Morris, Q. Cao, G. Bosilca, S. Mirchandaney *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, pp. 1–15.

[7] Y. Pei, G. Bosilca, I. Yamazaki, A. Ida, and J. Dongarra, "Evaluation of programming models to address load imbalance on distributed multi-core cpus: A case study with block low-rank factorization," in *2019 IEEE/ACM Parallel Applications Workshop, Alternatives To MPI (PAW-ATM)*. IEEE, 2019, pp. 25–36.

[8] E. Agullo, O. Aumage, M. Faverge, N. Furmento, F. Pruvost, M. Sergent, and S. Thibault, "Achieving High Performance on Supercomputers with a Sequential Task-based Programming Model," *TPDS - IEEE Transactions on Parallel and Distributed Systems*, Dec. 2017. [Online]. Available: https://hal.inria.fr/hal-01618526

[9] A. Denis, E. Jeannot, P. Swartvagher, and S. Thibault, "Using Dynamic Broadcasts to improve Task-Based Runtime Performances," in *Euro-Par - 26th International European Conference on Parallel and Distributed Computing*, Rzadca and Malawski. Warsaw, Poland: Springer, Aug. 2020. [Online]. Available: https://hal.inria.fr/hal-02872765

[10] 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 *Proceedings of the Platform for Advanced Scientific Computing Conference (PASC)*, 2020, pp. 1–11.

[11] OpenMP, "OpenMP 4.5 Complete Specifications," 2015. [Online]. Available: https://www.openmp.org/wp-content/uploads/openmp-4.5.pdf

[12] T. Heller, H. Kaiser, and K. Iglberger, "Application of the ParalleX execution model to stencil-based problems," *Computer Science - Research and Development*, vol. 28, no. 2-3, pp. 253–261, 2013.

[13] L. V. Kale and S. Krishnan, "CHARM++: a portable concurrent object oriented system based on C++," in *ACM Sigplan Notices*, vol. 28, no. 10. ACM, 1993, pp. 91–108.

[14] E. Anderson, Z. Bai, C. H. Bischof, L. S. Blackford, J. W. Demmel, J. J. Dongarra, J. J. D. Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. C. Sorensen, *LAPACK User's Guide*, 3rd ed. Philadelphia: SIAM, 1999.

[15] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet *et al.*, *ScaLAPACK users' guide*. SIAM, 1997, vol. 4.

[16] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, A. Haidar, T. Herault, 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 *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*, May 2011, pp. 1432–1441.

[17] "The chameleon project," https://gitlab.inria.fr/solverstack/chameleon, January 2017.

[18] M. Gates, J. Kurzak, A. Charara, A. YarKhan, and J. Dongarra, "Slate: design of a modern distributed and accelerated linear algebra library," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2019, pp. 1–18.

[19] A. Denis, E. Jeannot, and P. Swartvagher, "Interferences between communications and computations in distributed hpc systems," in *50th International Conference on Parallel Processing*, 2021, pp. 1–11.