2022 IEEE/ACM Parallel Applications Workshop: Alternatives To MPI+X (PAW-ATM) | 978-1-6654-5410-0/22/$31.00 ©2022 IEEE | DOI: 10.1109/PAW-ATM56565.2022.00008

Composition of Algorithmic Building Blocks in Template Task Graphs

Thomas Herault
Innovative Computing Laboratory
The University of Tennessee
Knoxville, TN, USA

Joseph Schuchart
Innovative Computing Laboratory
The University of Tennessee
Knoxville, TN, USA

Edward F. Valeev
Department of Chemistry
Virginia Tech
Blacksburg, Virginia, USA

George Bosilca
Innovative Computing Laboratory
The University of Tennessee
Knoxville, TN, USA

***Abstract*—In this paper, we explore the composition capabilities of the Template Task Graph (TTG) programming model. We show how fine-grain composition of tasks is possible in TTG between DAGs belonging to different libraries, even in a distributed setup. We illustrate the benefits of this fine-grain composition on a linear algebra operation, the matrix inversion via the Cholesky method, which consists of three operations that need to be applied in sequence.**

Evaluation on a cluster of many core shows that the transparent fine-grain composition implements the complex operation without introducing unnecessary synchronizations, increasing the overlap of communication and computation, and thus improving significantly the performance of the entire composed operation.

***Index Terms*—Task-Based Runtime System, Template Task Graph, Layered software design, Library composition**

I. Introduction

The Template Task Graph (*TTG*) concept has recently been introduced [1], [2] as the key abstraction of a homonymous programming model targeting scalable distributed micro task-based applications over modern high-performance computing platforms. The power of *TTG* derives from its ability to encode a finite directed acyclic graph (DAG) of tasks (or, rather, families of DAGs) in a compressed form by exploiting the regularity of structure of typical DAGs.

A *TTG* program consists of definitions of at least one TTG, which includes the type of operations the graph can generate (the Template Tasks, TTs), the types of input and output of each of these TTs (for static error checking), and the definition of how data (or control) messages can travel between the TTs. Computation is kicked off by initial data or control messages triggering a cascade of tasks dynamically instantiated by the flow of messages through the graph, through any predecessor task, executing on any thread across the entire execution environment. For TTGs composed of "normal" tasks (those that depend solely on replicated deterministic knowledge) the static replicated definition of each TTG allows any thread on any process to 'discover' new tasks (or provide the input of previously discovered tasks) without any communication, a capability critical to ensure high scalability.

However, when designing software libraries that rely on TTGs to implement their task-based algorithms, this introduces a new challenge: as the actual DAG of tasks is discovered dynamically during the execution from a predecessor task that belongs to the same DAG, how can we ensure that tasks discovered in one library can use data that has been produced by tasks from another library? And how can we detect how a successorship dependency can be established between tasks from different DAGs, allowing a tasks to forward their output to tasks belonging to another library?

A simple approach, embraced by many programming paradigms as a mean to compose libraries, is to use synchronizing fences between libraries: a DAG is fully unrolled and executed, storing the results of its computations in designated memory, before the DAG of the next library starts to execute, pulling the data from the same, well designated, memory. These fences introduce unnecessary synchronization in the parallel application and lead to sub-optimal use of resources, i.e., less concurrency and potentially higher storage requirements, and lower occupancy of computing resources for imbalanced algorithms. Therefore, such synchronization points and the ensuing serial portions of code may severely limit scalability [3].

Many existing DAG runtime systems allow a fine-grain composition of tasks, but when designing an application that composes two or more operations, where each operation is itself a distributed DAG of tasks, the programmer is faced with different situations depending on the task programming interface:

- While Sequential Task Flow (STF) approaches (e.g., OpenMP [4], OmpSs [5], StarPU [6], PaRSEC DTD [7]) conceptually provide fine-grain task composition by sequential construction of task graphs, the practical level of composition is typically limited. Because of the sequential nature of task discovery, STF-based systems introduce a window of submission: once a number of tasks have been discovered, tasks must be executed before new tasks can be discovered by the sequential thread. Unbounding this submission window, indeed opens up

978-1-6654-5410-0/22/$31.00 ©2022 IEEE
DOI 10.1109/PAW-ATM56565.2022.00008

opportunities for composition in exchange of a significant increase in memory consumption by the runtime, but only once all the tasks of the preceding operation have been discovered and taken into account. This hampers the fine-grain task composition because the full DAG of a previous operation must be discovered before any task of the next DAG. Moreover, the sequential discovery quickly becomes a bottleneck, even at moderate scales. The Template Task Graph approach, which unrolls a synthetic graph of template tasks, is impervious to this limitation because the sequential part of the code only needs to discover a number of template tasks that is independent of the problem size (and is typically constant per operation).

- On the other hand, future-based systems (e.g., HPX [8], MADNESS [9]) require the programmer to manage one future per dependency, which quickly becomes intractable when composing very large task graphs. A future represents a single value / dependency, and thus the approach lacks scalability, as a set of futures, linear in the problem size, must be defined as the output and input of each exposed DAG.
- Finally, existing approaches closest to TTG and based on unrolling of a synthetic graph of task classes (such as PTG) do not usually provide a concept that allows fine-grain task composition, as the generating graph does not know about the successor or predecessor graph. In PTG, the current approach either promotes coarse-grain operation composition (the first task of a successor DAG is only started after the last task of a predecessor DAG is completed), or it falls back on a non-scalable future-based systems: each data element of the input and output of an operation needs to be exposed via a future.

In this paper we demonstrate that by virtue of a full task graph satisfying the same concept as individual template tasks, TTGs naturally compose with other TTs and/or TTGs to form dynamic and more complex TTGs. Execution of such composite TTGs has the same favorable properties as execution of a single TTG composed of primitive TTs. This allows composition of complex fine-grained task DAGs from simpler reusable DAGs without **any** loss in performance. To illustrate the benefits of such TTG-based composition and the resulting performance benefits, we used a paradigmatic example from the Linear Algebra community, Matrix Inversion via the Cholesky method.

Such composition of task graphs extends properties known from software library design such as encapsulation and separation of concerns to the area of fine-grain task graph composition. It promises to further improve resource utilization by extending the view of the task scheduler across other separately developed algorithmic building blocks expressed in individual task graphs.

The rest of the paper is structured as follows. Section II provides an overview of the *TTG* programming model using a motivating example for the the composition of TTs and TTGs. Section III discusses the composition of linear algebra building blocks to form higher-level algorithms such as the Cholesky method for matrix inversion. Section IV discusses features in *TTG* and the underlying runtime systems that are useful to analyzing and improving the performance of composed task graphs. Section V provides an evaluation of the composed matrix inversion. Section VI discusses related work before Section VII concludes the paper.

II. Template Task Graph

Template Task Graph (*TTG*) is a new task programming interface built on top of C++ that promotes a flow programming model inspired by earlier innovations such as Flow-Based Programming (FBP) [10]. *TTG* targets modern scientific applications on modern high-performance computing (HPC) systems. Thus, efficiency in resource utilization, distributed computing, and scalability are key elements in the design and implementation of *TTG*. *TTG* defines a portable task-based interface, and features multiple implementations over different runtime systems. At the time of writing this paper two implementations exists: an implementation over the *PaRSEC* runtime system [11], that is the main focus for the high-performance implementation of *TTG*, and another one over the *MADNESS* runtime system [9] that validates the portability of the design and its openness towards other programming models.

TTG is designed with scalability in mind. In the context of distributed task programming paradigms, this implies that the discovery of tasks cannot be a sequential process, but must be distributed not only between the processes but even between threads (and tasks) that compose the application. However, the DAG of tasks remains, in essence, a common object, shared among all computing resources. Thus, to avoid bottlenecks it is important to create a structure over which the different computing units can distribute the load.

In [12], the authors introduce a task-based programming paradigm, called parameterized task graphs (PTG), running on top of *PaRSEC* where parameterized DAGs of tasks are created at compile time, thus providing a rigid structure on which the actual DAG of tasks that is executed can be discovered in a fully distributed manner: each thread in the parallel application can instantiate any subsection of the DAG of tasks independently, and only the tracking of dependencies between these subsections needs to be shared between threads.

TTG builds on a similar idea, but takes in account that modern scientific applications are not easily amenable to the rigid structure of a parameterized DAG. For example, parameterized DAGs cannot easily deal with data-dependent applications where the flow of data depends on the computed data itself. *TTG* preserves the central idea of a common shared structure of graph that describes classes of tasks and all *potential* data flow between them, but it opens this approach to irregular and data-dependent applications by allowing each task to instantiate its *actual* successors at runtime.

There are 3 core concepts in *TTG*:

a) Template task and task identifier: template tasks (TT) constitute the nodes of template task graphs. A template task

```
1  struct Key { int pos, step; };
2  auto stencilOperation =
3   [=](const Key &key,  // identifies the task
4       Cell&& current,   // cell to compute
5       const Cell& left, // left and right neighbors
6       const Cell& right) {
7     auto position = key.pos;
8     auto step = key.step;
9     current = compute(position, timestep,
10                      current, left, right);
11    if(step < maxStep) {
12      // send to next timestep, left and right neighbor
13      ttg::broadcast<0, 1, 2>(
14        std::make_tuple(
15          Key{(position + nbCell - 1) % nbCell, step+1},
16          Key{position, step+1},
17          Key{(position + 1) % nbCell, step+1)},
18        std:move(current));
19    } else {
20      // send to output
21      ttg::send<3>(position, std::move(current));
22    }
23  };
24
25  ttg::Edge<Key, Cell> current("current");
26  ttg::Edge<Key, Cell> l2r("left2right");
27  ttg::Edge<Key, Cell> r2l("right2left");
28  ttg::Edge<int, Cell> output("result");
29
30  auto stencilTT =
31            ttg::make_tt(stencilOperation,
32                         // input edges
33                         ttg::edges(current, r2l, l2r),
34                         // output edges
35                         ttg::edges(current, l2r, r2l,
36                                    output),
37                         "stencilOperation");
38
39  // connect the output edge to some template task
40  // storing results, omitted for simplicity
41
42  ttg::make_graph_executable(stencilTT);
43  // kick-off computation on all cells
44  for(auto n = 0; n < nbCell; n++) {
45    if(isLocal(n)) {
46      Cell cell(n);
47      Cell neutral();
48      stencilTT->set_arg<0>(Key{n, 0}, cell);
49      stencilTT->set_arg<1>(Key{n, 0}, neutral);
50      stencilTT->set_arg<2>(Key{n, 0}, neutral);
51    }
52  }
53  // wait for completion
54  ttg::fence(world);
```

Listing 1: TTG Example: 1D Stencil on a circular domain

represents a class of operations to apply to a set of data elements. Template tasks are instantiated into tasks using a task identifier. For example, in a simple stencil application (see the code in Listing 1), a single template task, `stencilTT` (Lines 30–37), defines a function to apply on each element of the stencil (`stencilOperation`, Lines 2–23). In this example, a task identifier is a pair of integers (Line 1) that defines the cell position in the stencil and the step of the iterative operation.

b) Terminals and Edges: Data flows through the graph along edges (Lines 25–28 in Listing 1) that connect the terminals of tasks. A task becomes ready for execution once all its input terminals have received a value. The edges `current` (next step for this cell), `l2r` (receives from left neighbor and sends to right neighbor) and `r2l` (receives from right neighbor and sends to left neighbor) connect the 3 input terminals of `stencilOperation` with the first 3 output terminals of the same template task. This connection is created by passing edges as input and output edges to `ttg::make_tt` (Lines 30–37), which creates a new template task. *TTG* will automatically connect the outputs and inputs and ensure that every input is connected to at least one output terminal (in `ttg::make_graph_executable`). Data is sent to the task's output terminals using either `ttg::send` (Line 21) or `ttg::broadcast` (Lines 13–18). The numeric template parameters specify the output terminals to use, i.e., `ttg::broadcast<1,2,3>` sends the cell along the `current`, `l2r`, and `r2l` edges.

c) Template Task Graphs and Dynamic instantiation of tasks: One or multiple Template Tasks connected together form a *Template Task Graph*, which can be implicit, like in the stencil example, or explicit (see below for examples). This graph can be made executable by notifying the runtime system (via the `make_executable` method), which then allows the user to start instantiating tasks explicitly via the `set_arg` method of a template task (Lines 48–50). In the stencil example, the data for the first iteration of the stencil operation is injected into the data flow via the loop of invocations in Lines 44–52. The user provides the task identifiers for each task invocation (a `Key` with the cell ID and step 0) and the data on which it will start to operate. From then on, until all tasks are done (Line 11 tests if the iteration reached a maximum number of steps for each task), each task instantiates one or more successor tasks that become ready once all their inputs are provided. Ready tasks are executed by the runtime in parallel, over the distributed resources. Data transfers required by the data flow happen asynchronously in the background.

The *TTG* API features many other elements, like traits to tune scheduling policies; process maps to specify where tasks execute; pull terminals to trigger operations when they are necessary instead of executing them proactively; reducing and gathering terminals to work with a variable amount of input flows; control flows to add additional constraints on the order of execution of tasks. These features have been studied in detail in [1] and [2].

III. Composition of Template Task Graphs

A straightforward approach to using different template task graphs in an application is to execute them one after the other. This, however, introduces serial code paths into applications whose individual parts are parallelized using *TTG*. Instead, edges can be used to combine both individual template tasks as well as whole template task graphs. For example, instead of writing each cell into a shared data structure (omitted in Listing 1) and passing this data structure between TTGs, it is possible to link individual TTGs using edges. Thus, the output of the stencil task graph could easily become the input of another task graph.

A. Extending the 1D Stencil Example

Listing 2 illustrates how the initial generation of the 1D stencil algorithm can be implemented in parallel with another

TT, and how it is composed with the 1D stencil TT to form a small TTG that provides the results of the stencil computation through a single edge. The `stencilOperation` is unchanged from the previous example in Listing 1, and we add a `generateOperation` (Lines 6–20) and a `genTT` (Lines 26–29). The `generateOperation` iterates over all possible cells in the stencil and instantiates the local cells with the initial value (Line 10) before it broadcasts the new cells to its output terminals, targeting tasks connected to its output terminals. The creation of the `stencilTT` TT is slightly changed. Its input terminals are connected to multiple edges using `ttg::fuse`. Each input terminal is connected to an output terminal of `genTT` and a corresponding output terminal of `stencilTT` itself (similar to Listing 1). Thus, the resulting `stencilTT` can accept inputs from either a previous iteration of `stencilTT` or from an output of `genTT`.

In Lines 43–50, `makeStencilWithInit` builds a template task graph that contains the generation task connected to the stencil computation task. The inputs to that TTG are empty while its outputs are a single terminal (the output terminal of the `stencilTT`). This TT hides all the complexities of connecting the `genTT` and `stencilTT` and provides a well-defined interface for its users to interact with it. For example, the resulting `stencilTTG` (Line 53) could be connected to another TTG through its output terminal and can be invoked (Line 55) to start the computation.

Internal operations, terminals, and edges that compose these operations are encapsulated inside the TTG, and a user of this operation can connect to it via its terminals, by providing edges that will forward the outputs of the stencil operation to a new set of tasks. Note, however, that this composition remains fine grain: data flows through the output edge as it is generated, distributed between the processes and the threads of each process.

B. A more complete example: matrix inversion via the Cholesky method

To illustrate further the capabilities provided by the composition of TTGs, we introduce a more complex example, the inversion of a matrix via the Cholesky method (POINV). This operation comprises two successive parts: the Cholesky factorization (POTRF) followed by POTRI, the inversion of a matrix factorized by the Cholesky decomposition. The Cholesky factorization is the decomposition of a Hermitian, positive-definite matrix into the product of a lower triangular matrix and its conjugate transpose: given a matrix A, where A is positive-definite of dimension $N \times N$, it computes L such that $A = LL^T$, where L is a lower triangular matrix with real and positive diagonal entries, and L^T denotes the conjugate transpose of L.

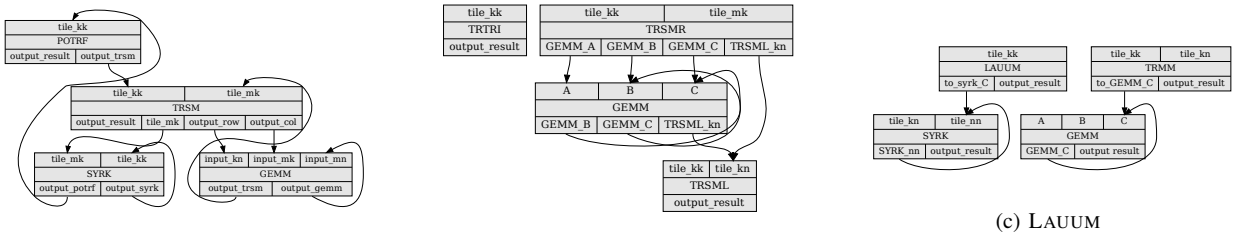
The Cholesky factorization can be computed efficiently using a well-known tile algorithm that is amenable to task-based representation [13], [14]. This algorithm decomposes the operation in four different base *kernels*, namely POTRF, TRSM, HERK (or SYRK for real values), and GEMM, that

```
1  using namespace std;
2  auto makeStencilWithInit(int nbCell,
3                           ttg::Edge<int, Cell>& output) {
4    auto stencilOperation = // unchanged from Listing 1
5
6    auto generateOperation =
7      [=]() {
8        for(auto n = 0; n < nbCell; n++) {
9          if(isLocal(n)) {
10           Cell cell(n);
11           // broadcast new cell to the first step
12           ttg::broadcast<0, 1, 2>(
13             make_tuple(
14               Key2{n, 0},
15               Key2{(n+nbCell-1) % nbCell, 0},
16               Key2{(n+1) % nbCell, 0}),
17               std::move(cell));
18         }
19       }
20   };
21   // Keep edges from Listing 1, and define additional:
22   ttg::Edge<Key, Cell> gen2middle("g2c");
23   ttg::Edge<Key, Cell> gen2left("g2l");
24   ttg::Edge<Key, Cell> gen2right("g2r");
25
26   auto genTT =
27         ttg::make_tt(generateOperation, ttg::edges(),
28                      ttg::edges(g2c, g2l, g2r),
29                      "generateOperation");
30
31   auto stencilTT =
32         ttg::make_tt(stencilOperation,
33                      // fused input edges from init
34                      // and a previous step
35                      ttg::edges(ttg::fuse(current, g2c),
36                                 ttg::fuse(left, g2l),
37                                 ttg::fuse(right, r2l)),
38                      // output edges
39                      ttg::edges(current, left,
40                                 right, output),
41                      "stencilOperation");
42
43   auto ins  = make_tuple();
44   auto outs = make_tuple(stencilTT->template out<3>());
45   vector<unique_ptr<ttg::TTBase>> ops(2);
46   ops[0] = std::move(genTT);
47   ops[1] = std::move(stencilTT);
48
49   return ttg::make_ttg(std::std::move(ops),
50                        ins, outs, "Stencil TTG");
51 }
52
53 stencilTTG = makeStencilWithInit(nbCell, output);
54 ttg::make_graph_executable(stencilTTG);
55 stencilTTG->invoke();
56 ttg::fence(world);
```

Listing 2: 1D stencil generation and computation combined into a single TTG

are basic operations of the BLAS library [15]. The actual tile algorithm is outside the scope of this paper and we refer interested readers to [16] for more details. Figure 1a represents the internal connections between the Template Tasks that implement this tile algorithm in TTG and Figure 2a shows the unrolled DAG of applying POTRF to a matrix with 5×5 tiles.

The inversion of a Cholesky matrix, called POTRI, consists of computing the inverse of a real symmetric positive-definite matrix A, A^{-1} once given the Cholesky decomposition of A. Once again, there is a well-known tile algorithm to compute efficiently this operation by applying successively two other operations on the matrix A: first apply TRTRI to compute



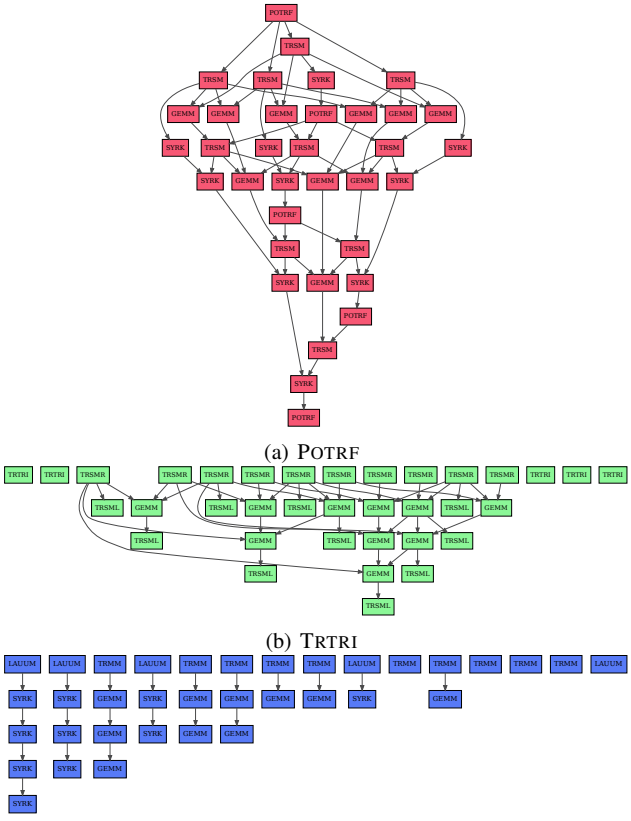
(a) POTRF (b) TRTRI (c) LAUUM

Fig. 1: Graph representation of the TTGs for the tile algorithms of POTRF, TRTRI and LAUUM

the inverse of L, L^{-1}, then apply LAUUM to compute the product ${L^{-1}}^T L^{-1}$ to compute A^{-1}. The TTG representations of these algorithms are given in Figure 1b and Figure 1c and the corresponding DAGs are presented in Figure 2b and Figure 2c. The distributed tile algorithm for TRTRI uses three base kernels from the BLAS: TRTRI (which computes the same operation on a single local tile), TRSM, and GEMM; similarly, the distributed tile algorithm to compute LAUUM is implemented using four BLAS kernels: LAUUM (which computes the product on a single tile), TRMM, SYRK or HERK (depending on the type of the elements), and GEMM.

The composition of the distributed POTRF operation followed by a POTRI operation is thus a common operation used by applications that rely on dense linear algebra operations. This illustrates well how base sequential kernels are composed at the tile level to produce distributed operations and how these operations are composed globally to produce complex results. We can consider three levels of composition: 1) each operation can be executed in sequence over the entire matrix, executing first POTRF, then TRTRI, and finally LAUUM (Figure 2); 2) TRTRI and LAUUM can be composed at the tile level to produce the DAG for POTRI, to execute POTRF followed by POTRI; or 3) all three basic operations can be composed at the tile level into the DAG of the POINV operation.

To provide a consistent interface, each of the TTG implementing these three building blocks is augmented with a new Template Task, that we call a *dispatcher* and that takes a single input terminal on which all tiles of the input matrix are sent. This template task distributes the incoming tiles to the first tasks of each algorithm that require it. It is thus similar in nature to the injection of cells into the task graph discussed in Section II, Listing 1. An edge is connected to the output terminal of each template task that applies a final operation in each algorithm. Through this edge, tiles are sent from one algorithm to the next, where they are again distributed by the *dispatcher* task. This provides a single entry and exit point for each TTG and TTGs can be composed as required by adding an edge connecting them. For example, in Listing 3 the output terminal of TRTRI is connected to the input terminal of LAUUM through the `trtri_to_lauum` edge to form the TTG of the POTRI operation (Line 6), whose visual representation is shown in Figure 3.



(a) POTRF (b) TRTRI (c) LAUUM

Fig. 2: Task graphs for individual operations on a matrix of 5×5 tiles if no composition is applied (some management tasks left out for clarity).

This representation exposes all the internal dependencies of each algorithm, but a user of any of these TTGs does not need to understand those internal complexities to use them: it is sufficient to provide each tile of the input matrix on the input edge and to consume them on the output edge to use these operations.

A DAG of POTRI applied to a matrix with 5×5 tiles is shown in Figure 4. It is notable that the composition at

```
using namespace std;
auto make_potri_ttg(ttg::Edge<Key2, MatrixTile>&input,
                    ttg::Edge<Key2, MatrixTile>&output)
{
  ttg::Edge<Key2, MatrixTile>
                    trtri_to_lauum("trtri_to_lauum");

  auto ttg_trtri = make_trtri_ttg(input, trtri_to_lauum);
  auto ttg_lauum = make_lauum_ttg(trtri_to_lauum, output);

  auto ins  = make_tuple(ttg_trtri->template in<0>());
  auto outs = make_tuple(ttg_lauum->template out<0>());
  vector<unique_ptr<ttg::TTBase>> ops(2);
  ops[0] = std::move(ttg_trtri);
  ops[1] = std::move(ttg_lauum);

  return ttg::make_ttg(std::move(ops),
                       ins, outs, "POTRI TTG");
}
```

Listing 3: Composition of POTRI in TTG, combining TRTRI and LAUUM.

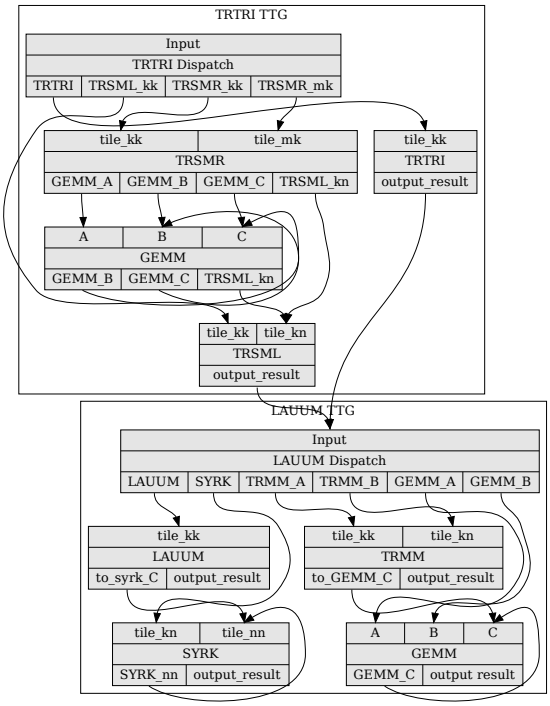


Fig. 3: Graph representation of the TTG for the tile algorithm of POTRI.

this level already eliminates the shrinking of concurrency by merging the two DAGs into a single graph that exhibits significant concurrency from top to bottom.

Similarly, the DAG for the fully composed POINV is provided in Figure 5. In contrast to executing POTRF (Figure 2a) and TRTRI (Figure 2b), the serializing tail has been eliminated, providing significant concurrency throughout the task graph.

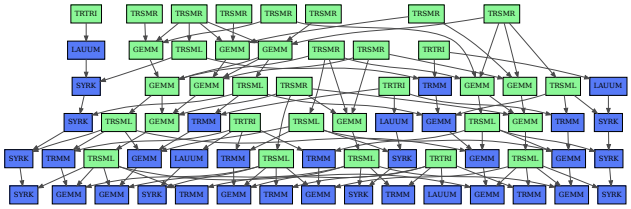


Fig. 4: DAG for the TTG of POTRI (Figure 3) on a matrix of 5×5 tiles, composing TRTRI (green) and LAUUM (blue). Some tasks (such as the dispatcher) have been omitted for clarity.

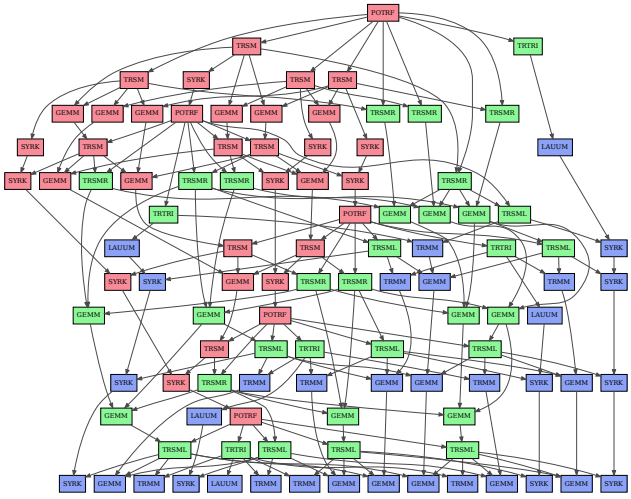


Fig. 5: Single task-graph for POINV on a matrix of 5×5 tiles, composing POTRF (red) and POTRI, in turn composed of TRTRI (green) and LAUUM (blue).

A notable advantage of the composition as it is done between TTGs is that the user can connect multiple edges (fused edges) to outputs of TTGs, to forward the data produced to multiple operations that run in parallel. In the context of the inverse of a Cholesky matrix, for example, it is frequent that the user requires both the factorized representation of the matrix (output of POTRF) and the inverse (output of POTRI). In the traditional approach, the factorization is computed, a copy of the output is created, and the inversion operation is applied (which modifies its inputs). With the fine grain composition of TTGs, the user can create two edges and fuse them. On one edge, the data will flow from the POTRF operation to the POTRI operation, while on the other edge the output of the POTRF operation can be read and stored for further (independent) computations. The runtime system ensures that the order of execution provides consistent views of the data, preventing tasks from modifying tiles while they are read by other tasks (more details are provided below in Section IV). In all cases, tiles are made available one by one, as they are produced, instead of waiting for the entire matrix operation to complete.

IV. Integration of composition features in the runtime ecosystem

A. New policy in the backend runtime system

As noted in Section III, composition features introduce new behaviors that favor concurrent access of tasks to the same units of data. For example, a user may need to keep an intermediary result, without interrupting the flow of data to the next phase of the algorithm. These competing requests can introduce concurrent accesses where one of the tasks accessing a data element requires write access (or modify access) to it. In that case, both backend runtime systems, *MADNESS*, and *PaRSEC* implement a private-copy-based policy: they create an additional copy for this data element before scheduling the write/modify task and provide the private copy to it, thus ensuring the correctness of the execution of read-only tasks.

However, the read accesses will eventually complete, and when a data element that was flowing in the task system is neither required by any new task for reading or writing, it is garbage collected and freed by the system. In the example of an intermediary result, the data elements provided to the reading tasks are typically consumed by an additional computation or by copying them into designated memory for later processing.

This private-copy-based policy is efficient in terms of parallelism, as it ensures the number of tasks that can run in parallel remains equal to all tasks whose input data are ready to process. However, it puts significant pressure on the memory, as many private, short lived, copies might be created, copies that will live only during the execution of a few tasks.

In the *PaRSEC* runtime system, we have introduced a new policy, that impacts the scheduling of tasks: if a task selected for execution by the dynamic scheduler requires write-access that conflicts with any other ready task, the writing task is deselected and marked to be re-scheduled when existing read accesses are completed. To avoid starvations, when the same task is selected again for execution, if new read accesses have been discovered on its input data, then a private copy is allocated (fallback to the private-copy-based policy). This new policy, which we call defer-writer, can be optionally set by users on a given TTG or TT. It limits the amount of temporary memory required for the execution of the TTGs while delaying update operations by favoring tasks that read the data elements.

Such concurrent accesses also occur within TTGs: the TRTRI and LAUUM operations, for example, exhibit parallel tasks that either read or modify the same tiles. In typical task-based systems, a control flow is added to ensure those accesses are done in sequence (all reads execute before the tasks that modify the data, e.g., DPLASMA [13]), or the sequential discovery of tasks ensures that the DAG built by the runtime system will execute all tasks that read the data before any task that writes it (e.g., *StarPU* [6]). With TTG, and the defer-writer and private-copy-based policies, the user can control how these cases are managed, and the choice of approach does not require modifying the code of the algorithm.

B. Profiling and debugging tools

TTG implementations come with a set of tools to help debug and understand the performance of TTG applications. The first tool is illustrated with Figure 1 and Figure 3: and provides a graphical representation of the Template Task Graph, where each Template Task appears as a box with input terminals on top and output terminals at the bottom. Edges are represented by arrows that link an output terminal to an input terminal. Since the Template Task Graph is not a DAG (edges represent all possible connections, and Template Tasks can be instantiated many times with different Task Identifiers during the execution of a single DAG of task out of a TTG), these graphs usually feature cycles.

That graphical representation is extended for TTG compositions by encapsulating all TTs or TTGs belonging to the same TTG inside a cluster subgraph, and adding the name of the TTG to the encapsulating box, as illustrated in Figure 3 for the POTRI TTG, which comprises the TRTRI TTG and the LAUUM TTG. The API of TTG allows the user to specify the depth in the hierarchy of objects they request from this output.

Second, TTG inherits from the tools available in its backend runtime system. Using the *PaRSEC* runtime system, the user can represent the DAG of tasks as it has unrolled during the execution of the TTGs, and as is illustrated in Figure 2 for the individual POTRF, TRTRI, and LAUUM DAGs on a 5×5 matrix. The task graph of the composition of these three algorithms in provided in Figure 5 below. These representations are especially useful to detect dependency errors during the development of new TTGs. The user can define the level of information that is displayed in the nodes and edges, for example, to add task identifiers in the task names, and data identifiers on the arrows.

V. Evaluation of composition capabilities

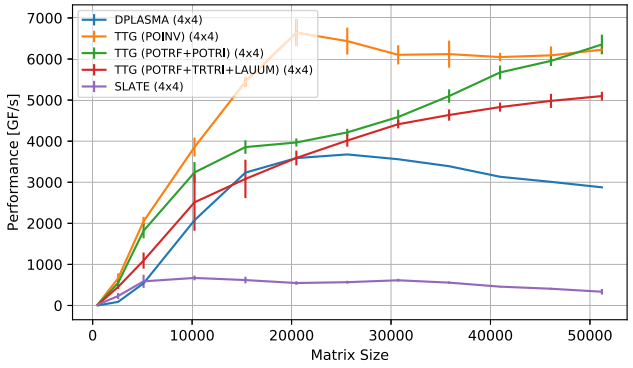
We conducted our evaluation on *Hawk*[1], a Hewlett Packard Enterprise system equipped with 64 core AMD EPYC Rome dual-socket nodes connected through an Infiniband ConnectX-6 fabric. We used GCC 10.2.0, Open MPI 4.1.0, and UCX 1.12.0 for our experiments. All data points reflect at least 5 repetitions and errors bars represent the standard deviation.

A. POINV Performance

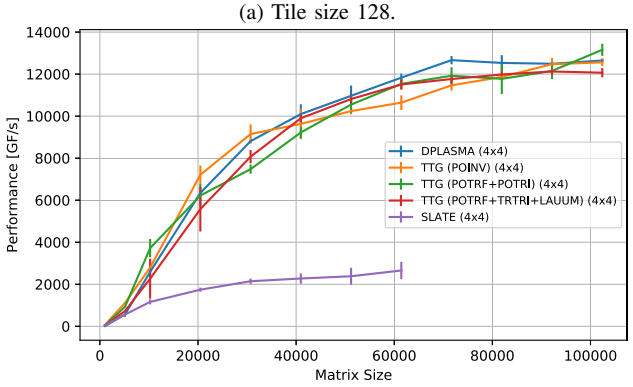
Figure 6 shows the performance of POINV with different levels of compositions described above and compares the performance against DPLASMA [13] and SLATE [17] on 16 nodes of Hawk with different tile sizes. Both SLATE and DPLASMA provide a sequential composition of POTRF, TRTRI and LAUUM to implement POINV.

The data suggests that the fine-grain composition of TTG is especially useful for smaller tiles (128, Figure 6a) while the benefit diminishes for tiles of size 256 (Figure 6b). It is especially notable that for small tiles, the fully composed POINV reaches peak performance at small matrices ($5k^2$ elements per node) while the partially composed POTRF +POTRI requires

[1] https://www.hlrs.de/systems/hpe-apollo-hawk/

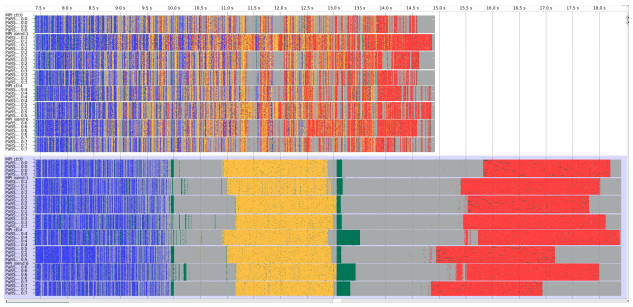


(a) Tile size 128.

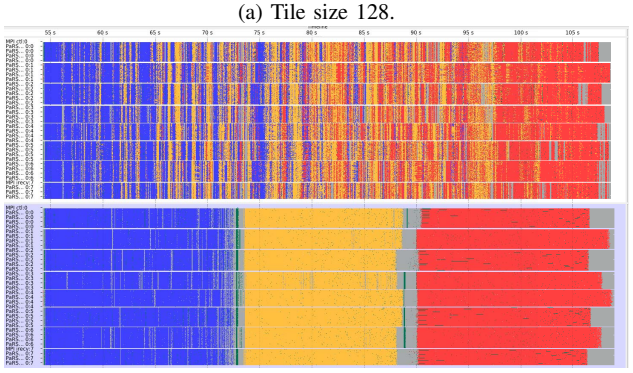


(b) Tile size 256.

Fig. 6: Problem scaling on 16 nodes with different tile sizes.



(a) Tile size 128.



(b) Tile size 256.

Fig. 7: Comparison of traces of fine-grain composition (top each) and sequential composition (bottom each) POINV on 8 processes with 60 threads each. The GEMM of POTRF are shown in blue, those of TRTRI are shown in yellow, and those of LAUUM are shown in red, and idle/communication tasks are shown as gray. All other tasks are green.

approximately double the size. As expected due to the need for fences, all sequential compositions provide lower performance than the fine-grain composition implementations.

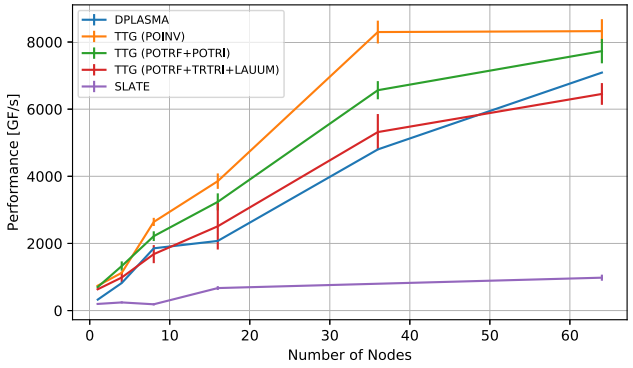
For tiles of 256 elements, the performance benefit of fine-grain composition is significantly diminished. This can be explained by the change in balance between computational density and communication overhead. All three algorithms are dominated by GEMM, whose computational complexity is in $\mathcal{O}(N^3)$ while the communication overhead is in $\mathcal{O}(N^2)$. Thus, by doubling the tile size, the computational efforts are increased by a factor of 8 while the amount of data to be transferred quadruples.

Indeed, looking at a comparison of event traces in Vampir for both tile sizes demonstrates this point. Figure 7 provides a screenshot of a Vampir comparison of traces for both the fine-grain composed TTG version and the sequential composition. The run with tiles of size 128 (Figure 7a) takes about 11 s while the run with tiles of size 256 (Figure 7b) lasts for almost 60 s. With the smaller tile size, the fine-grain composition hides most (although not all) communication overhead and the communication between algorithms takes a significant share of the total runtime. For the larger tiles, the share of communication diminishes and computation becomes dominant, reducing the benefit of the fine-grain composed algorithm.

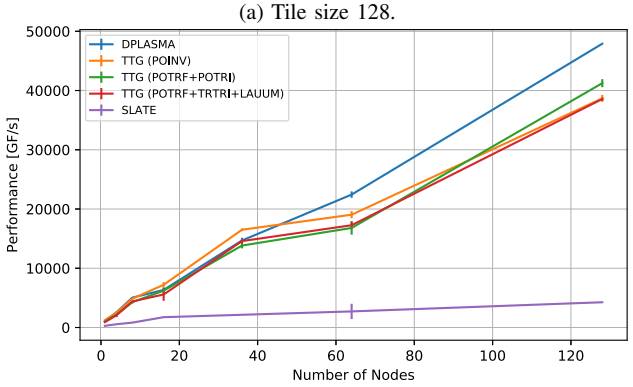
Figure 8 shows the weak-scaling performance of the different POINV implementation with 20^2 tiles per node. Similar to the above results, the fine-grain composition in *TTG* is more beneficial at smaller tile sizes (128), and diminishes when the tile size increases. In fact, it appears that DPLASMA shows slightly better performance than *TTG*. This may be due to the topological communication schemes used by DPLASMA, which appears to improve performance at scale.

We note that *SLATE* consistently yields relatively low performance, and scales poorly. We have noticed similar behavior in the past [2] and have seen that SLATE performs significantly better with larger tiles. However, we decided to include *SLATE* as a state of the art implementation. We have obtained a similar behavior for *ScaLAPACK* [18], which we have omitted here.

Overall, the fine-grain composition of task graphs (or algorithmic building blocks) promises improved latency hiding by exposing increased levels of concurrency to the runtime system. This is especially valuable for applications with a low computation-communication ratio and could become even more significant when applied to algorithms operating on sparse data structures. However, the development of such applications remains as future work.



(a) Tile size 128.



(b) Tile size 256.

Fig. 8: Node scaling performance with 20^2 tiles per node.

B. POTRI *Performance*

The Chameleon library (v1.1.0) runs on top of the *StarPU* runtime system and provides a composed version of POTRI (TRTRI + LAUUM) [19], which we compared against the composed POTRI in *TTG* (Figure 4). The performance on 64 nodes with 60 threads per node on Hawk and different tile sizes are shown in Figure 9. It can be seen that the *TTG* implementation vastly outperforms the Chameleon performance across the range of matrix sizes measured. The speedup with tiles of size 128 and 256 varies between 2x and 3x. This hints at the fact that sequential task discovery in Chameleon induces significant overheads and may even leave threads idling at small tile sizes. For even larger tile sizes (e.g., 512) the gap between the two implementations becomes smaller but we still observe measurable speedups, esp at higher node counts.

The performance achieved when scaling the number of nodes with a fixed number of tiles per node in each dimension of the matrix is shown in Figure 10. With tiles of size 128, *TTG* achieves higher performance even at a single node, which further underlines the idea that the sequential task discovery in Chameleon becomes the bottleneck. At tiles of size 256, the performance of *TTG* and Chameleon on a single node are similar, while *TTG* still achieves 2-3x speedup at higher node counts, mirroring the finding discussed above. We have observed similar behavior for a range of 30-90 tiles per

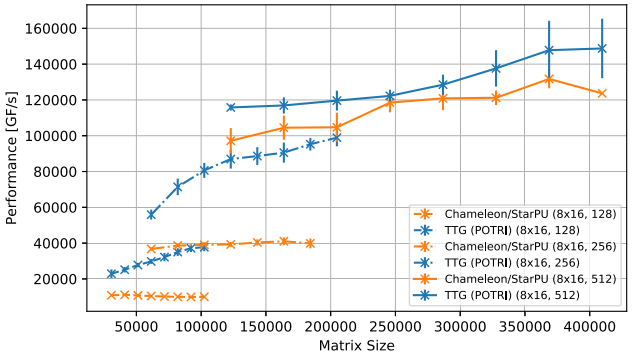


Fig. 9: POTRI performance of *TTG* and Chameleon at different tile sizes on 128 nodes.

dimension per node. The observed speedups at higher node counts and larger tiles can still be explained by the fact that in Chameleon the full task graph has to be discovered by each process. In contrast, the task graph discovery happens in parallel in *TTG* while the template task graph is unrolled.

Chameleon on top of StarPU relies on a Sequential Task Flow discovery of tasks. Such API allows to define a 'window size' via environment variables. The 'window' in this context defines how many tasks in advance the sequential thread that discovers tasks can insert in the DAG of tasks before it is blocked waiting for tasks to execute before the discovery can proceed. By default Chameleon does not define these environment variables, making StarPU run with an infinite window: any number of tasks can be submitted without blocking the execution. This allows the tasks of the LAUUM operation to be discovered, and to run in parallel with tasks of the TRTRI operation. However, because the task discovery is still sequential, the first LAUUM task is still only discovered after the last TRTRI task is discovered. Thus, many tasks that are not ready to execute in the TRTRI DAG are discovered and inserted in the DAG before the first LAUUM task (which becomes ready to execute quickly after the first TRTRI task is completed) is discovered and inserted. This still limits the amount of parallelism, which becomes dependent on the relative speed of the task insertion thread and the progress of the DAG. This explains partly the difference in performance compared to the TTG approach. In the TTG approach, the connection between the final task of TRTRI on any tile of the matrix with the first tak of LAUUM on the same tile is done beforehand, in a scalable way, by simply connecting the output terminal of the TRTRI TTG to the input terminal of the LAUUM TTG. At runtime, as soon as TRTRI is done on any tile of the matrix, the first LAUUM task on that tile becomes ready to execute maintaining maximum parallelism throughout the execution.

VI. RELATED WORK

The composition of task-based systems is highly dependent on the interface used to describe the tasks. Most traditional systems propose a mostly sequential interface, where tasks

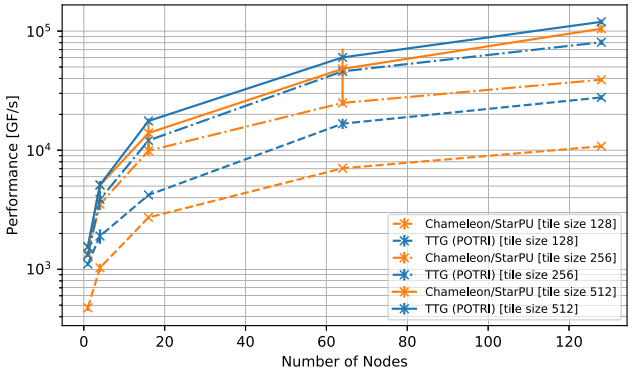


Fig. 10: POTRI performance of *TTG* and Chameleon with increasing node numbers and 50 tiles per node.

are discovered in sequence and executed asynchronously. The runtime system dynamically builds the DAG of tasks using dependency information either given as pragmas (OpenMP [4] or OmpSs [5]) or via the API (Sequential Task Flow model in StarPU [6], [20], Dynamic Task Discovery in *PaRSEC* [7], task dependencies in DASH [21]). In such approaches, the composition of DAGs within the same framework is completely transparent: in fact, there is a single DAG of tasks during the entire execution, and tasks are dynamically added and later removed when they complete their execution. However, the sequential discovery of tasks quickly becomes a bottleneck and the serialization of the discovery of task graphs means that consuming tasks may not be discovered before producing tasks have completed, preventing data from flowing directly from the producer to the consumer. The comparison with Chameleon in the experimental section illustrates this challenge for these approaches, and explains how the composition of TTGs is impervious to this issue.

In other systems, dependencies are made explicit via the creation of futures and promises (Legion [22], MADNESS [9], Charm++ [23], HPX [8]). Tasks are asynchronous functions that can set futures exposed to them and are unscheduled until all the promises they request are set. In this model too, DAGs are not explicit entities, and the composition of work within the same model uses the same future mechanisms. While futures enable data flow between tasks, the composition of task graphs requires sets of futures, which have to be managed by the application.

Last, a few approaches propose to build DAGs as objects that can be scheduled. These DAGs can be built at compile-time (PTG in *PaRSEC* [12], Eventify [24]) or at runtime (Dagger in Charm++ [25], [26]). The composition between two of these DAGs is part of the available API, but it can remain coarse grain (i.e. the first task of a successor DAG can only start after the last task of a predecessor DAG is complete).

VII. CONCLUSION

In this paper, we have introduced the composition feature of Template Task Graphs, describe its capabilities and evaluated its performance on the algorithm to compute the inverse of a matrix via the Cholesky method. The choice of the Inverse Cholesky as a playground was due to its composition of three sequential steps: first, a Cholesky decomposition on a symmetric positive-definite matrix A; then the inverse operation on the resulting triangular matrix. The inverse operation itself consists of two consecutive operations, a triangular inversion and a triangular update. Using the graph composition of TTG, the resulting DAG of tasks are merged at the granularity of tiles, allowing the composed system to execute without synchronizing barriers or fences between two high-level operations. As an added feature, intermediary results can be transparently extracted during the execution of the composed operation.

The TTG composition method hides the internal complexities of individual subgraphs, or building blocks, and thus enables proper software design in the realm of task-based applications by enabling both separation of concerns and encapsulation. The interaction between building blocks is achieved through edges, along which data is transported, giving the opportunity to the underlying runtime system to transparently handles data consistency and data exchange across process boundaries.

Performance evaluation on a distributed system shows that the fine grain composition generally increases the overlap of communication and computation, and the occupancy of computing resources. In particular, it results in significant performance improvement compared to the traditional fenced composition for cases where the computation/communication ratio is small, and a significant performance improvement compared to traditional fine-grain task composition based on sequential task discovery.

ACKNOWLEDGMENT

This research was supported partly by NSF awards #1931347 and #1931384, and by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration. We gratefully acknowledge the provision of computational resources by the High-Performance Computing Center (HLRS) at the University of Stuttgart, Germany.

REFERENCES

[1] G. Bosilca, R. J. Harrison, T. Herault, M. M. Javanmard, P. Nookala, and E. F. Valeev, "The Template Task Graph (TTG) - an emerging practical dataflow programming paradigm for scientific simulation at extreme scale," in *IEEE/ACM 5th Intl. Wksp. on Extreme Scale Programming Models and Middleware (ESPM2)*, Nov. 2020, pp. 1–7. [Online]. Available: https://ieeexplore.ieee.org/document/9307054

[2] J. Schuchart, P. Nookala, M. M. Javanmard, T. Herault, E. F. Valeev, G. Bosilca, and R. J. Harrison, "Generalized Flow-Graph Programming Using Template Task-Graphs: Initial Implementation and Assessment," in *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2022.

[3] G. M. Amdahl, "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities," in *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, ser. AFIPS '67 (Spring). Association for Computing Machinery, 1967.

[4] "OpenMP OpenMP Application Programming Interface, Version 5.2," Sep 2021. [Online]. Available: https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-2.pdf

[5] A. Duran, R. Ferrer, E. Ayguade, R. M. Badia, and J. Labarta, "A Proposal to Extend the OpenMP Tasking Model with Dependent Tasks," *Intl. Journal of Parallel Programming*, vol. 37, no. 3, 2009.

[6] C. Augonnet, S. Thibault, R. Namyst, and P. Wacrenier, "StarPU: A unified platform for task scheduling on heterogeneous multicore architectures," *Conc. Comp. Pract. Exper.*, vol. 23, 2011.

[7] R. Hoque, T. Herault, G. Bosilca, and J. Dongarra, "Dynamic Task Discovery in PaRSEC: A Data-flow Task-based Runtime," in *Proceedings of ScalA'17*, 2017.

[8] 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.

[9] R. J. Harrison, G. Beylkin, F. A. Bischoff, J. A. Calvin, G. I. Fann, J. Fosso-Tande, D. Galindo, J. R. Hammond, R. Hartman-Baker, J. C. Hill, J. Jia, J. S. Kottmann, M. Y. Ou, L. E. Ratcliff, M. G. Reuter, A. C. Richie-Halford, N. A. Romero, H. Sekino, W. A. Shelton, B. E. Sundahl, W. S. Thornton, E. F. Valeev, Á. Vázquez-Mayagoitia, N. Vence, and Y. Yokoi, "MADNESS: A multiresolution, adaptive numerical environment for scientific simulation," *SIAM J. Sci. Comput.*, vol. 38, no. 5, 2016.

[10] J. P. Morrison, *Flow-Based Programming, 2nd Edition: A New Approach to Application Development*. Scotts Valley, CA: CreateSpace, 2010.

[11] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, T. Herault, and J. Dongarra, "PaRSEC: A programming paradigm exploiting heterogeneity for enhancing scalability," *Comp in Sc. and Eng.*, vol. 99, p. 1, 2013.

[12] A. Danalis, G. Bosilca, A. Bouteiller, T. Herault, and J. Dongarra, "PTG: An abstraction for unhindered parallelism," *Proceedings of WOLFHPC'14*, 2014.

[13] G. Bosilca *et al.*, "Flexible Development of Dense Linear Algebra Algorithms on Massively Parallel Architectures with DPLASMA," *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*, 2011.

[14] 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.

[15] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh, "Basic linear algebra subprograms for fortran usage," *ACM Transactions on Mathematical Software (TOMS)*, vol. 5, no. 3, pp. 308–323, 1979.

[16] A. Buttari, J. Langou, J. Kurzak, and J. Dongarra, "A class of parallel tiled linear algebra algorithms for multicore architectures," *Parallel Computing*, vol. 35, no. 1, pp. 38 – 53, 2009.

[17] M. Gates, J. Kurzak, A. Charara, A. YarKhan, and J. Dongarra, "SLATE: Design of a Modern Distributed and Accelerated Linear Algebra Library," in *Supercomputing*, ser. SC '19. Association for Computing Machinery, 2019.

[18] J. Choi, J. Dongarra, R. Pozo, and D. Walker, "Scalapack: a scalable linear algebra library for distributed memory concurrent computers," in *The Fourth Symposium on the Frontiers of Massively Parallel Computation*. IEEE Computer Society, oct 1992.

[19] Institut national de recherche en sciences et technologies du numérique (INRIA). Chameleon—a dense linear algebra software for heterogeneous architectures. [Online]. Available: https://project.inria.fr/chameleon/

[20] E. Agullo, O. Aumage, M. Faverge, N. Furmento, F. Pruvost, M. Sergent, and S. P. Thibault, "Achieving high performance on supercomputers with a sequential task-based programming model," *IEEE Transactions on Parallel and Distributed Systems*, 2017.

[21] J. Schuchart and J. Gracia, "Global Task Data-Dependencies in PGAS Applications," in *High Performance Computing*. Springer International Publishing, 2019.

[22] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken, "Legion: Expressing locality and independence with logical regions," in *Supercomputing*, 2012.

[23] L. V. Kale and S. Krishnan, "CHARM++: A portable concurrent object oriented system based on C++," in *Proceedings of the Conference on Object Oriented Programming Systems, Languages and Applications*, 1993.

[24] D. Haensel, L. Morgenstern, A. Beckmann, I. Kabadshow, and H. Dachsel, "Eventify: Event-Based Task Parallelism for Strong Scaling," in *Proceedings of the Platform for Advanced Scientific Computing Conference*, ser. PASC '20. Association for Computing Machinery, 2020.

[25] A. Gursoy and L. V. Kale, "Dagger: combining benefits of synchronous and asynchronous communication styles," in *Proceedings of 8th International Parallel Processing Symposium*. IEEE, 1994, pp. 590–596.

[26] L. V. Kale and M. A. Bhandarkar, "Structured dagger: A coordination language for message-driven programming," in *European Conference on Parallel Processing*. Springer, 1996, pp. 646–653.

APPENDIX

Appendix: Artifact Description/Artifact Evaluation

SUMMARY OF THE EXPERIMENTS REPORTED

Platform: all experiments were conducted on Hawk (https://www.hlrs.de/systems/hpe-apollo-hawk/), a Hewlett Packard Enterprise system equipped with 64 core AMD EPYC Rome dual-socket nodes connected through an Infiniband ConnectX-6 fabric. We used GCC 10.2.0, Open MPI 4.1.0, and UCX 1.12.0 for our experiments.

Experiments:

- TTG DPOINV, DPOTRI, DTRTRI, DALUUM and DPOTRF runs (all figures in the paper): All TTG runs were conducted using the code available in https://github.com/therault/ttg/tree/potrf-composition2 at commit `ea1b2ae3c2de286dca03a34b91dcd87dfd65536b` by evaleev (comment of the commit: by default prevent find_* commands from looking in INSTALL/STAGING areas to avoid key pain point).
- SLATE (Figures 6 and 8 in the paper): all SLATE runswere conducted using the official repository at https://bitbucket.org/icl/slate.git at commit `c7f5563a1d0d6c580cb7cd9b4cf1a02b3fbb6ca6` (master main)
- DPLASMA (Figures 6 and 8 in the paper): All DPLASMA runs were conducted using DPLASMA available in https://github.com/ICLDisco/dplasma/ at commit `c90b90abeaa67282ccd280572a7e414b79ce6fbc` by bosilca (comment of the commit: Merge pull request #58 from bosilca/fix/concurrent_arenas)

Figures 1 to 5 provide graphical representations of graphs. They are generated using the PaRSEC runtime system, directly from the application. See Experimental Setup below for more detail.

Figures 6 and 8 are performance measurement figures that are extracted from jobs running over the batch scheduler of Hawk. We provide the scripts submitted to the batch scheduler and the scripts used to extract the results and analyze them as Artifacts linked below.

Figure 7 holds traces of executions visualized with Vampir (https://vampir.eu/). The Experimental Setup section below provides more details on how to reproduce such figure.

AUTHOR-CREATED OR MODIFIED ARTIFACTS:

Artifact 1

Persistent ID: https://doi.org/10.5281/zenodo.6968710
Artifact name: Artifact Appendix for Composition of Algorithmic Building Blocks in Template Task Graphs
Reproduction of the artifact with container: We did not provide a container with the necessary software, as the container would not produce the same performance as those obtained in a high-performance environment like the one of Hawk where performance were measured. However, the `CMakeLists.txt` of TTG fetches directly almost all software components necessary for the reproduction of the experiments done in this paper.

To reproduce the environment, you will need:

- CMake 3.22.0 or later
- TTG from https://github.com/therault/ttg at commit `ea1b2ae3c2de286dca03a34b91dcd87dfd65536b` (in branch potrf-composition2)
- MKL version 19.1.0
- GCC version 10.2.0
- DPLASMA from https://github.com/ICLDisco/dplasma/ at commit `c90b90abeaa67282ccd280572a7e414b79ce6fbc` (master branch)
- SLATE from the official repository at https://bitbucket.org/icl/slate.git at commit `c7f5563a1d0d6c580cb7cd9b4cf1a02b3fbb6ca6` (master main)
- GraphViz 2.43.0
- Vampir 9.8.0
- Open MPI version 4.1.0
- hwloc version 2.2.0
- Open Trace Format 2 (OTF2) version 2.3

The run scripts as well as the analysis scripts are provided at https://doi.org/10.5281/zenodo.6968710.
TTG experiments:

- The source is in the `source` directory
- We compiled the tester used for performance runs in a performance build directory
- We compiled the tester used to output the DAGs and profiling other graphs in a profiling directory
- Figures 6 and 8 of the paper: we configured TTG from an empty `performance` directory with the following command line, where `TTG_SOURCE` is an environment variable pointing to the source directory of TTG: `cmake ${TTG_SOURCE} -DTTG_EXAMPLES=1 -DIntelMKL_THREAD_LAYER=sequential` we compiled the tester with the following command executed in the `performance` directory: `make -C examples testing_dpoinv-parsec`
- The run scripts are generated using the `generate-all.sh` generator. The resulting batch scripts have been submitted using `qsub`.
- The resulting output log files have been passed to `plot.py` to generate the plots: `./plot.py /*.log`
- To produce Figures 1 and 3 in the paper (graphs of the TTGs), we started from an empty `profiling` directory, and configured the tester using the following arguments, in the `profiling` directory: `cmake -DTTG_EXAMPLES=1 ${TTG_SOURCE}` We then compiled the tester by issuing the following command in the profiling directory: `make testing_dpoinv-parsec`. Finally, we ran the following commands that outputs DOT representations of the graphs: `./examples/testing_dpoinv-parsec -nruns 0 -N 100 -t 20 -v -seq 2 ./examples/testing_dpoinv-parsec`

`-nruns 0 -N 100 -t 20 -v -seq 1` The outputs were copy/pasted in files and edited to simplify the presentation, then processed via the `dot` tool from GraphViz https://graphviz.org

- To produce Figures 2, 4 and 5, we started from an empty `profiling` directory, and configured the tester with the following arguments from the `profiling` directory: `cmake -DTTG_EXAMPLES=1 -DIntelMKL_THREAD_LAYER=sequential -DPARSEC_PROF_GRAPHER=ON ${TTG_SOURCE}` We then compiled the tester by issuing the following commands in the profiling directory: `make -C examples testing_dpoinv-parsec`. Finally, we ran the following commands that outputs DOT representations of the graphs: `./examples/testing_dpoinv-parsec -nruns 0 -N 100 -t 20 -seq 0 -dag poinv ./examples/testing_dpoinv-parsec -nruns 0 -N 100 -t 20 -seq 1 -dag potri ./examples/testing_dpoinv-parsec -nruns 0 -N 100 -t 20 -seq 2 -dag sequential` Each command generates a local file whose name is the last argument appended with `-0.dot`. These files were edited to simplify them using the `massage_dot.sh` script: `./massage_dot.sh poinv-0.dot > poinv-clean.dot ./massage_dot.sh potri-0.dot > potri-clean.dot ./massage_dot.sh sequential-0.dot > sequential-clean.dot` and produce the figures were created by passing the resulting dot files to the dot tool from GraphViz (https://graphviz.org) `dot -Tpdf -O poinv-clean.dot; dot -Tpdf -O potri-clean.dot; dot -Tpdf -O sequential-clean.dot`
- To produce the lower parts of Figure 7a and 7b, we started from an empty `profiling` directory, and configured the tester with the following arguments from the `profiling` directory: `cmake -DTTG_EXAMPLES=1 -DIntelMKL_THREAD_LAYER=sequential -DPARSEC_PROF_TRACE=ON -DPARSEC_PROF_TRACE_SYSTEM=OTF2 ${TTG_SOURCE}`. The OTF2 library, version 2.1.1 must be discovered by cmake during this configuration phase for the rest to work. We then compiled the tester by issuing the following command in the profiling directory: `make testing_dpoinv-parsec`. To prepare for the run, the file `$HOME/.parsec/mca-params.conf` must be created/edited to include the following lines: `profile_filename = profile mca_pins = task_profiler`. Then, the tester was run with the following command: `mpirun -np 8 ./examples/testing_dpoinv-parsec -N $((20*256)) -t 256 -c 60 -nruns 0`. This produces an OTF2 output in `profile.otf2` and the `profile/` directory. This can be opened with Vampir (https://vampir.eu/) to obtain a figure similar (the actual figure depends on the execution) to the top of Figure 7b To obtain a figure similar to the top of Figure 7a, the run must be `mpirun -np 8 ./examples/testing_dpoinv-parsec -N $((20*128)) -t 128 -c 60 -nruns 0`. Note that the file `profile.otf2` and the directory `profile/` must be deleted between two tests.

DPLASMA experiments:

- DPLASMA was configured in an empty `performance` directory with the following command line (where `DPLASMA_SOURCE` is an environment variable set to the directory holding the soure of DPLASMA). `cmake ${DPLASMA_SOURCE}/`
- It was then compiled with `make`
- And run using the batch scheduler via the script referenced above
- Results were collected from the output of the job, and analyzed using the python script referenced above

SLATE experiments:

- SLATE was configured in an empty `performance` directory with the following command line (where `SLATE_SOURCE` is an environment variable set to the directory holding the soure of SLATE). `cmake ${SLATE_SOURCE}/`
- It was then compile with `make`
- And run using the the batch scheduler script referenced above