Generalized Flow-Graph Programming Using Template Task-Graphs: Initial Implementation and Assessment

Joseph Schuchart
Innovative Computing Laboratory
The University of Tennessee
Knoxville, TN, USA
schuchart@icl.utk.edu

Poornima Nookala
Institute for Advanced Computational Science
Stony Brook University
Stony Brook, NY, USA
poornimavinaya.nookala@stonybrook.edu

Mohammad Mahdi Javanmard
Meta Platforms, Inc
New York, NY, USA
mjavanmard@fb.com

Thomas Herault
Innovative Computing Laboratory
The University of Tennessee
Knoxville, TN, USA
herault@icl.utk.edu

Edward F. Valeev
Department of Chemistry
Virginia Polytechnic Institute and State University
Blacksburg, VA, USA
valeev76@vt.edu

George Bosilca
Innovative Computing Laboratory
The University of Tennessee
Knoxville, TN, USA
bosilca@icl.utk.edu

Robert J. Harrison
Institute for Advanced Computational Science
Stony Brook University
Stony Brook, NY, USA
robert.harrison@stonybrook.edu

***Abstract*—We present and evaluate *TTG*, a novel programming model and its C++ implementation that by marrying the ideas of control and data flowgraph programming supports compact specification and efficient distributed execution of dynamic and irregular applications. Programming interfaces that support task-based execution often only support shared memory parallel environments; a few support distributed memory environments, either by discovering the entire DAG of tasks on all processes, or by introducing explicit communications. The first approach limits scalability, while the second increases the complexity of programming. We demonstrate how *TTG* can address these issues without sacrificing scalability or programmability by providing higher-level abstractions than conventionally provided by task-centric programming systems, without impeding the ability of these runtimes to manage task creation and execution as well as data and resource management efficiently. *TTG* supports distributed memory execution over 2 different task runtimes, *PaRSEC* and *MADNESS*. Performance of four paradigmatic applications (in graph analytics, dense and block-sparse linear algebra, and numerical integrodifferential calculus) with various degrees of irregularity implemented in *TTG* is illustrated on large distributed-memory platforms and compared to the state-of-the-art implementations.**

***Index Terms*—Flowgraph programming, Dataflow graph, Template Task Graph, PaRSEC, MADNESS**

I. Introduction

This work is inspired by the belief that flowgraph programming (FGP) is a superior match for (1) high-performance parallel programming of modern computers with complex (distributed/heterogeneous) memory hierarchies and a large number of, potentially heterogeneous, compute resources, (2) *irregular* (scientific) applications characterized by data-dependent operation streams, and (3) combinations thereof. This belief is reflected by many efforts employing data- and control/work-flow programming to simplify parallel programming as an alternative to traditional bulk-synchronous models (see Section IV). The advantages of FGP are due to several traits: (1) specification of only *essential* dependencies between operations maximizes exploitable concurrency and opportunities for hiding latency by overlapping data motion and computation, (2) making the data part of the flow a *dataflow* reduces the need for synchronization, eliminates scheduling delays, and makes operations easier to reuse by eliminating nonessential side effects, and (3) by raising the level of abstraction, programs (often) become easier to write, easier to transform (thereby supporting the development of domain-specific languages), and easier to port. While these advantages are not unique to FGP, achieving them via more conventional means typically involves significant programming costs due to the low-level (and sometimes explicit) management of asynchronous execution. Thus FGP is uniquely positioned to address the tension between programmer productivity and the programming challenges posed by the ever-increasing complexity of hardware and applications.

Recently, a novel flowgraph-based programming model, Template Task Graph (*TTG*), and its implementation as a C++ library was introduced in [8]. The objective of *TTG* is to enable high-level composition and efficient execution of broad classes of *irregular* algorithms on distributed-memory heterogeneous

clusters. Irregular algorithms, characterized by the lack of natural load balance in data and computation and dynamic (typically, data-dependent) operation flow, are of particular importance to modern data science and physical simulation applications, which trade uniform data structures (uniform meshes, dense tensors) for irregular data-sparse counterparts (adaptively-refined meshes, block- and rank-sparse tensors). While irregular algorithms are the natural keystone target for *TTG*, regular algorithms at scale pose similar challenges due to nonuniformity of the hardware (heterogeneous execution units, nonuniform thermal environment) and the execution space (e.g., when multiple dissimilar regular algorithms are executed at once).

The main contributions of this paper are:

- introduction of several new features in *TTG* to enhance the ease of composition and increase execution efficiency;
- the evaluation of their implementation over four paradigmatic regular (dense linear algebra, graph analytics) and irregular (block-sparse linear algebra, adaptive spectral element calculus) applications, and the comparison of the *TTG* implementation of these applications with state of the art implementation in other programming paradigms.

The rest of this paper is organized as follows: in Section II, we present the *TTG* language, its driving concepts, and a discussion of its implementation over *PaRSEC* and *MADNESS*; in Section III, we introduce and evaluate the different applications. We discuss the related work in Section IV before we conclude.

II. Template Task Graph

A detailed description of *TTG* can be found in [8]. For the purpose of this paper, we will provide a brief description of the components in *TTG*, their interaction, and some of the implementation details.

TTG represents an algorithm as a flowgraph (*template task-graph*, TTG) composed of one or more nodes (*template tasks*) equipped with ordered sets of input and output *terminals* connected by directed *edges*. In the current C++ implementation of *TTG*, template tasks, terminals, and edges are explicitly and strongly typed. Edges encode all possible flows of *messages*. Each message consists of a *task ID* and *data*; this idea builds on the concept of the Parameterized Task Graph (PTG) [15]. The task ID represents the task (instance of a task template) for which the data is intended. Thus, messages in the *TTG* model generally contain both a control part (task ID) and data part, allowing to marry the control-flow and data-flow paradigms. Pure control flow can be implemented by omitting the data part, i.e., by using the null type (void) to represent the data part of the message. Pure dataflow can be implemented analogously by using the null type to represent the task ID.

Once every input terminal of a given template task has received one message with the same value of task ID, a task is created with the data parts of the corresponding messages. Tasks define a *task body*, which is a C++ method that will be executed by the runtime system. *TTG* does not constrain the task bodies in any way (i.e., the tasks can be arbitrary, not necessarily pure, functions) but any side effects may require additional synchronization to avoid data races. During its execution, the task may deliver new messages to zero or more output terminals. Introducing the data dependence into the control flow (i.e., by deciding whether a particular output terminal will receive a message or not, or by making the task IDs of the outgoing messages dependent on the data contents of the input messages) allows to implement general data-dependent task flows in *TTG* seamlessly. Thus, the message flow through a TTG generates a set of tasks representing an application. Each TTG can be viewed as encoding a set of possible directed acyclic graphs (DAGs) of tasks with the actual DAG executed being dependent on the data flowing through it.

To illustrate these concepts, we consider the well-known algorithm for (non-pivoted) Cholesky factorization of a dense tiled matrix used in the standard distributed-memory linear algebra package *ScaLAPACK* [14] and whose TTG implementation will be assessed in Section III-B. Figure 1 illustrates its template task graph. The algorithm consists of 4 types of tasks: `POTRF` (Cholesky factorization of diagonal tiles), `GEMM` (generalized matrix multiply), `SYRK` (symmetric rank-k matrix update), and `TRSM` (triangular linear system solver). Each task type is represented by a node in TTG, with two additional nodes representing reading of the input data (`INITIATOR`) and writing the output data (`result`).

Listing 1 illustrates how the TTG is composed by connecting inputs and outputs of each task template to the edges (represented in C++ by `ttg::Edge`). Note that each output terminal may be attached to one or more input terminals. Each task template is typically composed from a free or lambda function by calling `ttg::make_tt` (Lines 9 and 41). Listing 1 illustrates also how the `TRSM` task template is implemented. The lambda (or free function) implementing a task body receives as its arguments the task ID (if non-void), input data (if non-void), and the tuple of output terminals (`ttg::Out`; Lines 14–20). The function body performs arbitrary computation on the data and, if needed, "sends" the data to the output terminals via `ttg::send` (if intended to be an input for a single task) or `ttg::broadcast` (if intended to be an input for multiple tasks; Lines 37–39). Since the edges, input, and output terminals are all explicitly parametrized by the type of data they transport the type safety of TTG's edges and task templates is checked at compile time. Note that the graph built by connecting the nodes that represent task types via edges includes cycles and thus does not represent directly the DAG of tasks. It is during the execution, when tasks are instantiated with their task IDs, that the DAG of task is constructed, distributed across processes, by each task instance that discovers a new task instance.

The task ID of a given task does not have to match, or even be of the same type as, the task IDs of its output terminals. For example, the `TRSM` task IDs are represented by a 2-tuple (`Int2`) and produce data that will be used to create tasks with IDs represented by 3-tuples (`Int3`). Immutable data may be shared between tasks while tasks mutating inputs receive

```
1  /* Edges with 1-tuple task IDs */
2  ttg::Edge<Int1, Tile> init_potrf;
3  /* Edges with 2-tuple task IDs */
4  ttg::Edge<Int2, Tile> potrf_trsm, trsm_result,
5                        trsm_syrk, gemm_trsm;
6  /* Edges with 3-tuple task IDs, encodes the iteration K */
7  ttg::Edge<Int3, Tile> trsm_gemm_row, trsm_gemm_col;
8  auto POTRFOp =
9     ttg::make_tt(potrf_fn /* not shown here */,
10                 /* input edges */ ttg::edges(init_potrf),
11                 /* output edges */
12                 ttg::edges(potrf_results,
13                            potrf_trsm));
14 auto trsm_fn =
15     [](const Int2& id, const Tile<T>& tile_kk,
16        Tile<T>&& tile_mk,
17        std::tuple<ttg::Out<Int2, Tile<T>>,
18                   ttg::Out<Int2, Tile<T>>,
19                   ttg::Out<Int3, Tile<T>>,
20                   ttg::Out<Int3, Tile<T>>>& out){
21   const auto [I, J] = id;
22   const auto K = J;
23   /* call LAPACK library's tsrm function */
24   TRSM(tile_kk, tile_mk);
25   std::vector<Int3> row_ids, col_ids;
26   /* ids for gemms row I */
27   for (int n = J+1; n < I; ++n)
28     row_ids.push_back(Int3(I, n, K));
29   /* ids for gemms column I */
30   for (int m = I+1; m < NROWS; ++m)
31     col_ids.push_back(Int3(m, I, K));
32   /* broadcast the result to 4 output terminals:
33    * 0: to final output task writing back the tile;
34    * 1: to the SYRK kernel;
35    * 2: to the gemm tasks on in row I;
36    * 3: to the gemm tasks in column K; */
37   ttg::broadcast<0, 1, 2, 3>(
38     std::make_tuple(id, Int2(I, K), row_ids, col_ids),
39     std::move(tile_mk), out);
40 };
41 auto TRSMOp = ttg::make_tt(trsm_fn,
42                            /* input edges */
43                            ttg::edges(potrf_trsm, gemm_trsm),
44                            /* output edges */
45                            ttg::edges(trsm_result, trsm_syrk,
46                                       trsm_gemm_row,
47                                       trsm_gemm_col));
```

Listing 1: Select elements of the C++ code specifying the TTG implementation of dense tiled Cholesky factorization in Figure 1.

private copies, which may be passed on to other operations. Thus, applications need not be concerned with protecting access to data under *TTG*'s control. The safety of side-effects of tasks on data outside the control of *TTG* is under the purview of the application.

Once a task template receives all inputs needed for a given task ID the task is scheduled for execution. The process on which a given task will be executed is specified by a user-defined function mapping task IDs to process ranks. Note that creation and execution of tasks is entirely abstracted out in *TTG*. Thus, *TTG* can be viewed as a higher-level abstraction for a low-level task runtime. Current implementation of *TTG* can use one of two task runtimes for distributed task execution: *PaRSEC* and *MADNESS*. Section II-D will discuss the relevant implementation details.

In this work the following features were added to *TTG*:

- the ability to assign *priorities* to tasks by supplying each task template with a *priority map* mapping a task ID to a specific task priority that is provided to the underlying runtime system;
- optimized implementation of `ttg::broadcast`, which appears as a common use case, e.g. in the `TRSM` task template in Listing 1;
- streaming terminals that can receive not just a single message but a (bounded or unbounded) stream of messages;
- support for C++ data types serializable via general-purpose serialization frameworks, as well as support for RMA data transfers where supported by the runtime;
- an improved implementation of *TTG* over two runtimes, focusing on performance.

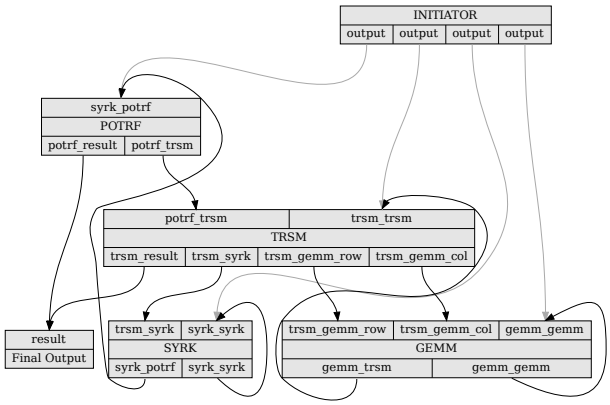


Fig. 1: Template task-graph of the tiled Cholesky factorization. The `INITIATOR` operation is responsible for providing input to tasks that have no direct predecessor in the algorithm.

Several of these features are discussed in detail below.

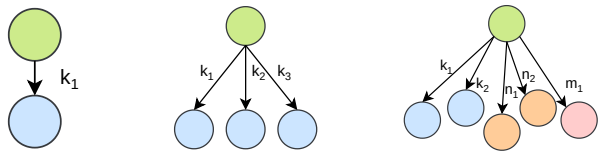
A. Sending and Broadcasting

TTG supports several ways to send data out of tasks:

- to a single output terminal accompanied by a single task ID (`ttg::send`; see Figure 2a);
- to a single output terminal accompanied by several task IDs (`ttg::broadcast`; see Figure 2b);
- to multiple output terminals, each accompanied by one or more task IDs (`ttg::broadcast`; see Figure 2c).

The latter is used in the implementation of the `TRSM` task template shown in Listing 1 (Lines 37–39). The data broadcasting was introduced to optimize data transfers between processes and avoid repeated transfers of the same data.

Note that by default `send` and `broadcast` both copy the argument data; this allows subsequent mutation of the data for sending it to other terminals. Passing data by constant reference indicates that the copying can be bypassed, if possible (e.g., if the lifetime of the object is already tracked by the runtime; see Section II-D). To indicate that the data is no longer going to be used in the task template body the data can be passed by rvalue reference (via `std::move`); for types with efficient rvalue copies this allows to implement efficient (potentially, zero-copy within memory space) data flow through the graph. These customization mechanisms are illustrated in Listing 2.



(a) Send data accompanied by task ID k_1 to 1 output terminal

(b) Broadcasting data accompanied by task IDs $\{k_1, k_2, k_3\}$ to 1 output terminal

(c) Broadcasting data accompanied by task IDs $\{k_1, k_2\}$, $\{n_1, n_2\}$, $\{m_1\}$ to 3 output terminals

Fig. 2: TTG send and broadcast operations.

```
void taskfn(const TaskID& task_id, const MatrixTile& input,
            tuple<Out<TaskID, MatrixTile>,
                  Out<TaskID, MatrixTile>,
                  Out<TaskID, MatrixTile>>& out) {
  MatrixTile output = compute_output_tile(input);
  send<0>(task_id,output,out);//new copy required
  send<1>(task_id,move(output),out);//no copy due to move
  send<2>(task_id,input,out);//no copy as input is const
}
```

Listing 2: Examples of using const-qualified types and `std::move` to signal immutability of objects provided to `ttg:send`.

B. Streaming Terminals

The original design of *TTG* mandated that each input terminal can receive only a single message for a given task ID. For some types of algorithms this restriction produces task templates with large numbers of input terminals. For example, a 1D Jacobi would only require 3 input terminals: the state of the task at the previous iteration as well as the state of the left and right neighbors. However, a 2D Jacobi requires 5 to 9 inputs (depending if neighbors on the diagonal need to be considered), and a 3D Jacobi quickly becomes un-manageable through explicit input terminals defined as independent variables in the user code. In this work, this restriction was lifted by making all input terminals capable of receiving a stream of messages for every task ID. The input messages are *reduced* (e.g., concatenated) using a user-provided function $U \otimes T \to U$ reducing a pair of values into a single value. Each incoming message is processed in a light-weight manner (i.e., without spawning a task) until either the prescribed number of messages has been received or the input terminal is programmatically "finalized" for the given task ID (see Figure 3). An example for using streaming terminals will be provided later in Section III-E.

C. Data serialization

Execution of *TTG* programs involving dataflow requires support for serialization of user data types, both for data between memory spaces (such as between host memories of different processes, or between host and device memory for a single process). The original implementation of *TTG* was limited to serialization of data types that were (1) trivially (bitwise) copyable, or (2) were serializable by the *MADNESS* runtime-provided serialization. In this work,

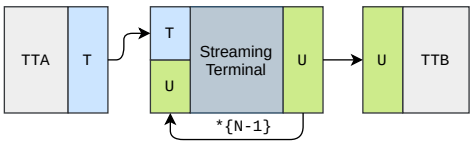


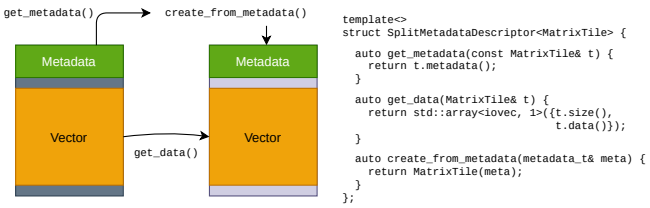
Fig. 3: TTG streaming terminal with input `T`, output `U`, and a size of `N`. The reduction operation of the terminal will be called $N-1$ times on input from `TTA` before before a task of `TTB` will be eligible for execution.

TTG was extended to support data types serializable via the widely-available Boost.Serialization[1] library. Since stock Boost serialization archives provide a number of default features intended for archival purposes (type versioning, pointer tracking, etc.), they are ill-suited for high-performance applications like *TTG*. Therefore, support for Boost.Serialization-compatible types in *TTG* uses custom archives optimized for high-performance serialization into in-memory buffers. *TTG* also provides type traits that detect serializability of a given type via Boost.Serialization, *MADNESS*, or by `memcpy`, and makes the optimal choice of serialization protocol. Thus several mechanisms of serialization are provided for a given flowgraph.

Unfortunately, the default serialization protocols that *TTG* can exploit necessarily involve multiple copies (object to/from serialization buffer to/from MPI message buffer, etc.). To increase the efficiency of data flow, a split-metadata (`splitmd`) mechanism was implemented in *TTG* in the course of this work. Unlike Boost.Serialization and its sibling *MADNESS* serialization protocols, in which the entire object is serialized and transferred as a whole, `splitmd` is a 2-stage protocol (see Figure 4). First, the object's *metadata* (data fields that are sufficient for allocating an object's representation in memory) are serialized and transferred. In addition, the object's contiguous memory is registered with the communication library. The metadata and registration information combined are typically sufficiently small to utilize the eager protocol commonly found in MPI implementations. On the receiving process, the metadata is used to allocate a new object. In the second phase, the received registration information is used to fetch the data into the contiguous memory of the newly created object using remote memory access (RMA). The RMA capability to *TTG* is typically provided by underlying communication libraries such as LCI [16], UCX [38], or GASnet [7]. It can also be emulated using MPI point-to-point operations or use features proposed for MPI RMA [35]. Once the transfer is complete, the sender is notified to release the source object.

Since the `splitmd` serialization fundamentally requires allocated-but-not-yet-initialized to be a valid state, the `splitmd` is intrusive (i.e., typically requires modification of the type definition and/or implementation). Type traits are used to test at compile time whether a given type supports the `splitmd` protocol. Serialization protocols chosen by *TTG* are

[1] https://www.boost.org/doc/libs/1_77_0/libs/serialization/doc/index.html




```
template<>
struct SplitMetadataDescriptor<MatrixTile> {

  auto get_metadata(const MatrixTile& t) {
    return t.metadata();
  }

  auto get_data(MatrixTile& t) {
    return std::array<iovec, 1>({t.size(),
                                 t.data()});
  }

  auto create_from_metadata(metadata_t& meta) {
    return MatrixTile(meta);
  }
};
```

Fig. 4: Schematic depiction of TTG's serialization format for objects containing contiguous data segments (left) and an example implementation for a MatrixTile (right).

selected in this order of preference: `splitmd` (if supported by the *TTG* backend; see Section II-D), trivial (`memcpy`), Boost.Serialization (if the Boost library is available), `madness` (if the *MADNESS* library is available).

D. TTG Execution Backends

As mentioned before, *TTG* as a programming model is a higher-level abstraction over the underlying low-level task runtime. The current C++ implementation of *TTG* can in principle create tasks using many available task runtimes, e.g., standard C++ (`std::async`) or OpenMP. In practice, however, for optimal resource utilization the implementation details of the task runtime matter greatly even in a shared-memory (host-only) setting. For distributed memory operation, additional features are needed to support seamless data transfers, parallel primitives (collective operations, global termination detection), and resource management; extra support is needed for heterogeneous execution and memory spaces within the node.

The implementation details of *TTG* are collectively referred to as a *TTG backend*. A backend provides the ability to schedule and execute tasks as well as resource management and coordination for communication and computation in a distributed setting. There are currently two backends supporting the execution of *TTG* applications on shared- and distributed-memory platforms: *MADNESS* or *PaRSEC*. The *MADNESS* backend served as an early proof of concept for *TTG*, with the *PaRSEC* backend targeted to serve as the main vehicle for efficient performance-portable operation on distributed and heterogeneous platforms. The feature set required to implement *TTG* is not unique to these two backends and is available in other runtimes (e.g., UPC++), thus implementation of additional backends for *TTG* should be straightforward.

MADNESS parallel runtime: started as the foundation for fast integrodifferential numerical calculus with guaranteed precision in up to 6 dimensions, with applications in chemistry and nuclear physics, among others [21]. By now, however, the *MADNESS* parallel runtime has evolved into a powerful general-purpose environment for task-based composition of a wide range of parallel algorithms on distributed data structures as varied as irregular trees in *MADNESS* and the sparse tensors in the *TiledArray* framework [13]. The central elements of the parallel runtime are a) futures for hiding latency and managing dependencies, b) global namespaces with one-sided access, c) remote method invocation in objects in global namespaces, and d) dynamic load balancing and data redistribution. An SPMD model is provided with a single logical main thread per process, a thread pool to execute tasks, and a thread dedicated to serving remote active messages. *MADNESS* can be configured to use its own thread pool implementation, or to use Intel TBB or *PaRSEC*. An application in the *MADNESS* runtime can be viewed as a dynamically constructed DAG, with futures as edges.

PaRSEC [9]: is a task-based runtime for distributed heterogeneous architectures, capable of dynamically unfolding a concise description of a graph of tasks on a set of resources and satisfying all data dependencies by shepherding data between memory spaces (including between nodes) and scheduling tasks on heterogeneous resources. Compared to many runtime systems that support a single way to represent or discover a DAG of tasks, *PaRSEC* is designed to support many Domain Specific Languages (DSLs) or Application Programming Interfaces (APIs). This makes *PaRSEC* a tool of choice to study different APIs or DSLs for distributed task-based programming.

Multiple components constitute the *PaRSEC* runtime: programming interfaces (DSLs/APIs), schedulers, communication engines and data interfaces. The runtime uses a modular component architecture (MCA), allowing different modules or instances to be dynamically selected during runtime, providing a varied set of capabilities to different instances of the runtime (such as scheduling policies, or support for heterogeneity). A well-defined API for these modules transforms them into black boxes, and allows interested developers or users to implement their own, application specific, policies. The different DSLs share the same runtime, data representation, communication engine, scheduler, cohabiting over the same set of hybrid resources and seamlessly inter-operating in the context of the same application.

Several optimizations were introduced in this work specifically for the *PaRSEC* backend, including improvements to the *PaRSEC* runtime itself: a flexible new interface of the *PaRSEC* runtime system to efficiently organize communication between processes, the use of active messages for control signals, the use of a one-sided communication for asynchronous transfers of data, and the use of completion callbacks for notifications. The `splitmd` serialization protocol is also only available when using the *PaRSEC* backend. Most importantly, the *PaRSEC* backend now owns the data flowing through the *TTG* graph and is in charge of managing its life-cycle and marshaling it across memory space boundaries, such as for avoiding copying when data is passed to `ttg:send` or `ttg::broadcast` by `const` reference.

These additions target improving the efficiency and scalability of the *PaRSEC* backend, but have no impact on the correctness and capability of *TTG*, both current *TTG* backends support the full set of *TTG* features. In fact, all *TTG* programs developed in this work are backend independent, with the backend selection performed at compile time by setting a single preprocessor macro. Since the backend can sometimes

TABLE I: Software configurations

Software	*Hawk*	*Seawulf*
MPI	Open MPI 4.1.1, UCX 1.10.0	Intel MPI 20.0.2
Compiler	GCC 10.2.0	GCC 10.2.0
HWLOC	1.11.9	1.11.12
MKL	19.1.0	20.0.2

have substantial impact on the performance, where warranted the performance will be demonstrated for both backends.

III. Benchmarks

A set of paradigmatic algorithms, with varying degree of irregularity in their data and computation traits, was implemented using C++ implementation of the *TTG* programming model. The performance was evaluated against reference implementations using traditional programming models or, where available, against existing state-of-the-art implementations.

A. Test Setup

We performed our evaluation on two systems. The *Hawk* system is a Hewlett Packard Enterprise Apollo[2] installed at the High Performance Computing Center Stuttgart (HLRS) in Stuttgart, Germany, consisting of 5,632 dual-socket 64-core AMD EPYC 7742 nodes equipped with 256 GB main memory and connected through a Mellanox Infiniband HDR 200 fabric. The *Seawulf* system is a Linux cluster installed at StonyBrook University[3] and consists of a variety of nodes equipped with Intel CPUs. In particular, we used up to 32 dual-socket Intel 20-core Xeon Gold 6148 CPUs with 192 GB main memory connected using a Mellanox InfiniBand FDR network. The used software configuration for both systems are listed in Table I.

B. Dense Cholesky Factorization

We implemented the dense tiled Cholesky factorization (POTRF) [12] in *TTG* and compared its performance against state-of-the-art implementations *SLATE* [18], *Chameleon*[4] (running on top of *StarPU* [5]), *ScaLAPACK* [14], and *DPLASMA* [10] (running on top of *PaRSEC*). The templated task-graph of the tiled POTRF algorithm is depicted in Figure 1. To demonstrate the competitive performance *TTG* can deliver, we ran two separate scaling experiments: i) weak scaling across a number of nodes; and ii) problem scaling on a fixed number of nodes. In both cases, we used 60 worker threads pinned to a single NUMA domain per node to avoid interference and to work around issues with process binding observed with some of the reference implementations, leaving 4 cores for the operating system and communication threads.

[2]https://www.hlrs.de/systems/hpe-apollo-hawk/
[3]https://it.stonybrook.edu/help/kb/understanding-seawulf
[4]https://project.inria.fr/chameleon/

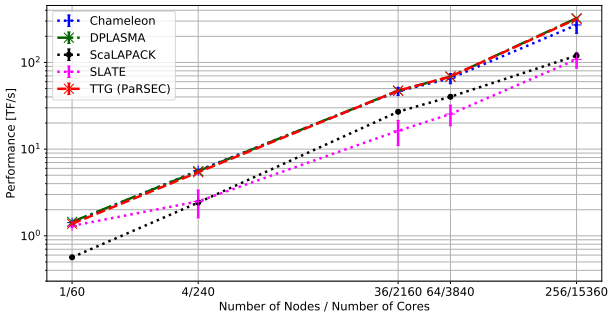


Fig. 5: Weak-scaling of `POTRF` on *Hawk*. Each node holds a submatrix of size $30k^2$. The tile size is 512^2.

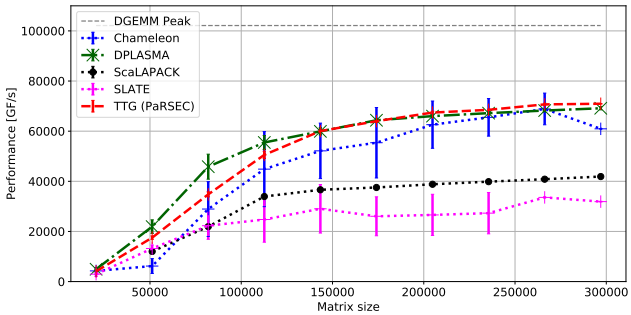


Fig. 6: Scaling the matrix size on 64 nodes performing tiled Cholesky factorization with a tile size of 512^2 on *Hawk*.

1) Node-scaling: Figure 5 shows a clear separation between two sets of scalability trends. *ScaLAPACK* and *SLATE* steadily continue to grow their performance but at a slower pace compared with the others, a behavior that can be explained by the sequentiality induced by the compute flow in the Cholesky algorithm without lookahead implemented in these two libraries. On the other side, all task-based versions benefit from the lack of synchronizations of the tile Cholesky implementation, and see a significant growth in performance with the increase in the number of compute resources in this weak-scale setup. Chameleon slightly trails behind the *TTG* and *DPLASMA* despite having the same potential parallelism due to the same tiled Cholesky implementation. A possible explanation is a more efficient communication substrate in *PaRSEC*, including the collective communication, but a more in-depth analysis would be necessary to confirm this.

2) Problem-scaling: Figure 6 shows a similar outcome, two well-separated groups, both asymptotically reaching their peak for this number of processes. Again, the task-based approaches benefit from the lack of synchronizations, and thus a large potential parallelism that once efficiently mapped into the compute resources lead to a more efficient execution and to reaching the practical peak for smaller matrix sizes.

C. Floyd-Warshall All-Pairs-Shortest Path (FW-APSP)

The FW-APSP algorithm finds the shortest path between every pair of vertices in a directed graph. It is among the most fundamental graph algorithms and has several applications in

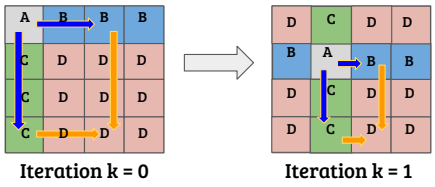


Fig. 7: Flow of data among different kernels in blocked FW-APSP algorithm.

computer networks, logic programming, optimizing compilers, model-checking, social media, transportation, among others.

Prior work proposed different optimization techniques to improve the performance of the algorithm. Venkataraman et al. proposed a single-level tiled algorithm to improve the I/O complexity [40]. Javanmard et al. extended it to a recursive multi-level tiled algorithm to run efficiently on distributed-memory machines as well as GPUs [25], [27]. In the recursive multi-level tiled algorithm, the first level of tiling is used to distribute the underlying adjacency matrix among processes and further parallelism and I/O efficiency were achieved by recursive sub-tiling. Nookala et al. [31] implemented a data-flow version of the standard two-way recursive divide-and-conquer FW-APSP algorithm in Intel CnC [11] and compared the performance with a fork-join implementation in OpenMP. They showed that a data-flow implementation outperforms its fork-join counter-part when, due to artificial dependencies, the fork-join implementation fails to generate enough subtasks to keep all processors busy and does not have enough data locality to compensate for the lost performance.

As shown in Figure 7, the parametric recursive algorithm has four kernels (A, B, C, and D) that each compute the minimum shortest path within the input tiles of the adjacency matrix. Kernel A is only applied to the tiles on the diagonal, followed by kernels B and C applied to the respective row and column. The results of kernels B and C are used as input for kernel D, which is applied to the panels on both sides of the current row and column. In the multi-level MPI+OpenMP implementation, the exchange of super-tiles along rows and columns is performed using MPI broadcast operations while the application of the operations to the sub-tiles is done using OpenMP tasks. In *TTG*, on the other hand, a single-level 2D block-cyclic distribution of tiles is used and tiles are broadcast to all successor operations independent of other tiles. The MPI+OpenMP implementation of [27] puts significant constraints on the available process configurations by requiring process numbers that are both square and multiples of 2. This constraint was later discussed in [25], [26] and virtual padding is mentioned as a potential solution to this constraint but the distributed-memory implementation was not discussed. While the *TTG* implementation of the benchmark does not have these constraints, in the interest of comparability we decided to run the same configuration for both MPI+OpenMP and *TTG*.

Figure 8 depicts the strong-scaling behavior of both the *TTG* and MPI+OpenMP implementation on a 32k matrix with different block sizes. The data shows that the *TTG* implementation clearly outperforms the MPI+OpenMP implementation up to 16 nodes by a factor of almost 2, with *TTG* running on top of *PaRSEC* further scaling to 64 nodes for block sizes of 64 and 128. *TTG* running on top of *MADNESS* benefits from larger tile sizes, presumably due to the lower number of tiles to communicate, but is limited in its scalability.

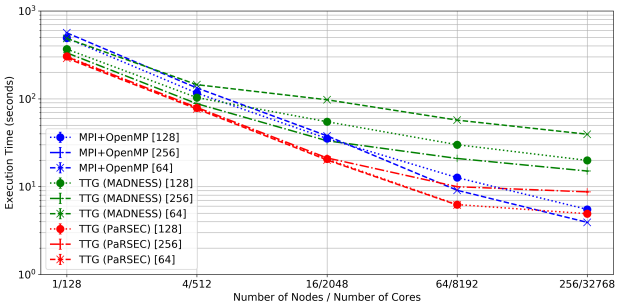


Fig. 8: Strong scaling of the Floyd-Warshall benchmark using *TTG* and MPI+OpenMP on Hawk using 16 processes per node, 8 threads each (block sizes in square brackets).

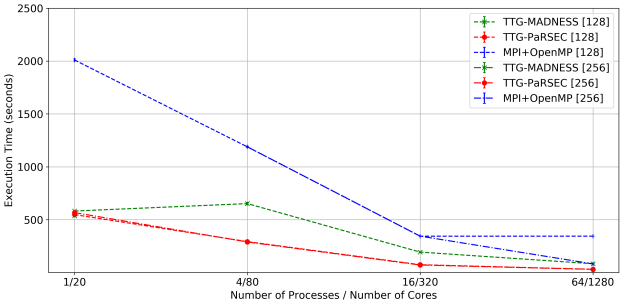


Fig. 9: Strong scaling of the Floyd-Warshall benchmark using *TTG* and MPI+OpenMP on SeaWulf using 2 processes per node, 20 threads each (block sizes in square brackets).

For *TTG* running on top of *PaRSEC*, smaller block sizes lead to better scalability. At 256 nodes, however, *TTG* using blocks of size 128 reaches its scalability limit: $(\frac{32k}{128}) = 256$ blocks in each dimension distributed across $\sqrt{256 \times 16} = 64$ processes per dimension results in $\frac{256}{64} = 4$ blocks per process, less than the number of threads. Unfortunately, an issue in Open MPI prevented us from running with block sizes of 64 with *TTG* on top of PaRSEC on 256 nodes. However, we expect *TTG* to further scale to 256 nodes once this issue is resolved.

Figure 9 shows the strong-scaling behavior on SeaWulf using a 32K matrix with block sizes 128 and 256. *TTG* implementations outperform the MPI+OpenMP implementation on up to 32 nodes by a factor of 4. *TTG* with *MADNESS* performs similar to the *PaRSEC* version with 256 tile size as compared to 128 tile size due to less communication with larger tiles. The running time for benchmarks with 64 tile size exceeded the time-limit and hence are not included in the plot.

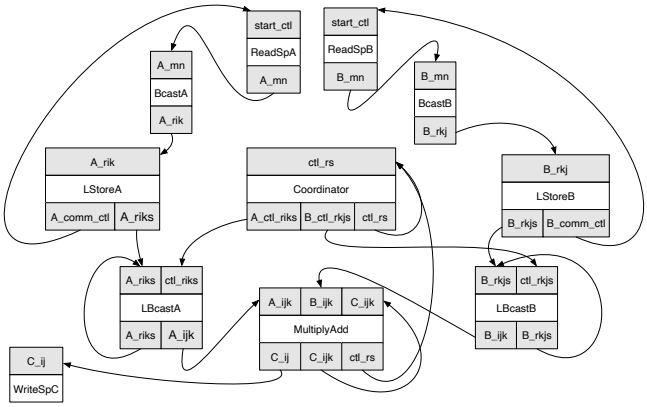


Fig. 10: Template task-graph of the block-sparse matrix-matrix multiply algorithm.

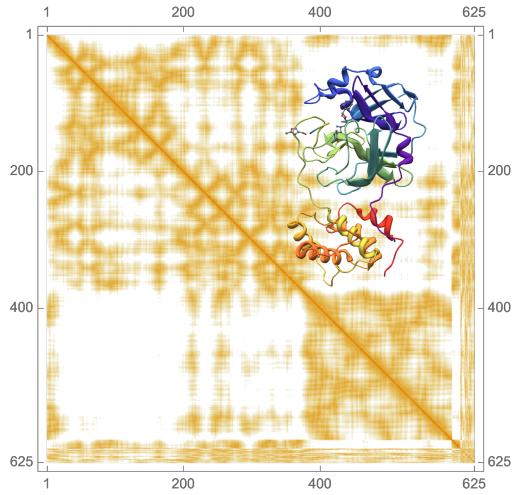


Fig. 11: Nonzero blocks of the block-sparse Yukawa potential matrix used for the `bspmm` benchmark (see text for details).

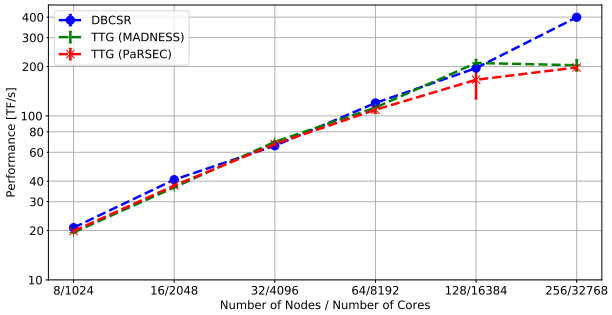


Fig. 12: Strong scaling of block-sparse GEMM.

D. Block-Sparse GEMM

As a first irregular application, we considered a block-sparse matrix-matrix multiplication (`bspmm`). The arguments are matrices tiled in blocks of irregular dimensions, with a significant subset of blocks empty. The bspmm algorithm we implemented follows a 2D SUMMA strategy [39], adapted to the task-based representation, similarly to [23].

The resulting TTG is depicted in Figure 10. Tasks of type `ReadSpA/B` load the tiles from memory and inject them into the flowgraph. The tiles are broadcast to remote nodes via the tasks of type `BcastA/B`, and stored on each node in the tasks of type `LStoreA/B`, to avoid additional communications. There is a control-flow feedback loop from `LStoreA/B` to the `ReadSpA/B` to control how many of these communications can happen in parallel. Then, tiles flow to the main computational kernel in tasks of type `MultiplyAdd` through local broadcast tasks of type `LBcastA/B`. There is another feedback control loop through tasks of type `Coordinator` that wait until multiple `MultiplyAdd` tasks are completed before it allows tasks of type `LBcastA/B` to continue broadcasting local work. This reduces the choices of the scheduler and forces it to focus on a subset of GEMM tasks that work on the same subset of data. Both feedback loops are implemented using streaming terminals discussed in Section II-B.

Compared to the previous applications, BSPMM is irregular and requires dynamic decisions: the DAG of tasks that must be executed depends on each input problem, and there is no universal data placement and scheduling strategy that can guarantee optimal performance without adapting these decisions to each input problem. The task-based approach of *TTG* delegates the dynamic scheduling decision to the underlying runtime system, creating some adaptability. Data placement remains heuristical, based on a 2D block cyclic distribution to balance the load, and the additional control flow is here to manage the high degree of parallelism of the problem.

To evaluate the performance of the `bspmm` implementation, we used the matrix representation of the Yukawa integral operator $(\exp(-r_{12}/5)/r_{12})$ in the cc-pVDZ-RIFIT Gaussian atomic orbital basis for the main protease of the SARS-CoV-2 virus in complex with the N3 inhibitor [28] (total of 2,500 atoms; this size is representative of target problem sizes in biomedical applications). The size of the matrix is 140,440; rows/column panels corresponding to each of the 2,500 atoms are grouped into tiles such that the size of each tile does not exceed the target tile size of 256. Tiles of the matrix with the per-element Frobenius norm of less than 10^{-8} are discarded. We compute the square of the resulting block-sparse matrix A (Figure 11) using the `bspmm` implementation.

We compare the *TTG* implementation with the Distributed Block Compressed Sparse Row library (DBCSR [30]). DBCSR is part of the CP2K quantum chemistry and solid state physics program package; it implements a 2.5D communication-reducing SUMMA algorithm [36] and focuses on block-sparse matrix-matrix multiplication of matrices with a relatively large occupation.

Figure 12 shows the performance obtained for an increasing number of nodes for the Yukawa potential matrix multiplication. From 8 to 128 nodes, DBCSR and both *TTG* backends all exhibit very similar performance, with a linear strong scaling. The *TTG* implementation with both backends stop scaling at this size and for this matrix, while the DBCSR one continues. The *TTG* implementation over the *PaRSEC* backend shows

a high variability at 128 nodes, with some runs significantly slower than others, and a peak at the same speed as the *TTG* implementation over the *MADNESS* backend and the DBCSR one. We are investigating this instability, that we observe only for the specific communication pattern for 128 nodes.

At 256 nodes, each process holds only a few tiles of the product matrix, and communications become the dominant factor of the execution. The 2.5D SUMMA algorithm [36] implemented in DBCSR continues to scale due to its ability to leverage greater cross-section bandwidth compared to the 2D SUMMA variant that was implemented in *TTG*. We expect that by converting the current 2D SUMMA *TTG* implementation to 2.5D SUMMA we will be able to at least match the strong-scaling performance of DBCSR.

E. Multi-Resolution Analysis (MRA)

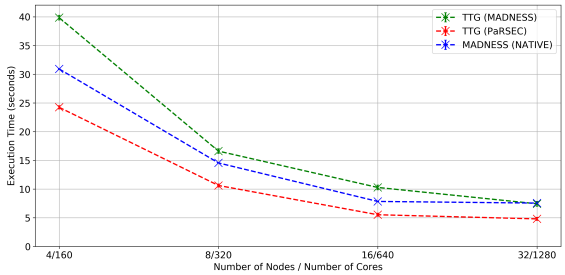
This benchmark computes adaptively the order-10 multiwavelet [3], [4] representation of 3-D Gaussian functions (exponent $30,000$) to precision of 10^{-8} with Gaussian centers distributed randomly in a $[-6,6]^3$ volume. This random distribution leads to substantial clustering and hence load imbalance that is only partially addressed by overdecomposition using a *task ID map* that randomly distributes function tree nodes (and their children) across processes at some target level of refinement. Empirically, the load imbalance is offset by the reduction of communication.

The MRA computation on each function commences by adaptively projecting into the multiwavelet basis by recurring down until the local representation error is below the truncation threshold. The resulting data structure is a 3D spatial tree that extends down about 6 levels of adaptive dyadic refinement. Subsequently, the fast wavelet transform (compression) and inverse transform (reconstruction) are performed and the norm of the function is also computed for verification purposes. Work and data flow down the tree in the projection and reconstruction steps, and flows up the tree for compression. In the compression operation, a parent node needs coefficients from its $2^3 = 8$ children. The code is templated by the number of dimensions, making this a perfect use case of streaming terminals so that a single terminal can process children in arbitrary dimensions. Prior to streaming terminals, the example had to employ complex C++ templates to manage a variable and potentially large number of terminals. The native *MADNESS* implementation computes on each tree in parallel, but there is an explicit barrier after each computational step (projection, compression, reconstruction, norm) as the in-memory data structure is completed. In contrast, the *TTG* implementation eliminates all inessential barriers and streams data through the entire DAG and never stores an explicit representation of all trees. The transition between algorithms that ascend and descend implies that there is a moment for each tree for which all data is stored (as arguments of pending tasks), but computation on other trees proceeds independently in the *TTG* implementation.

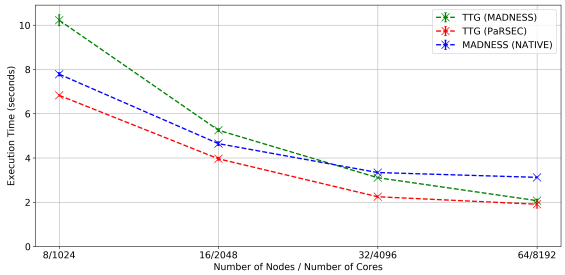
The streaming terminal feature is essential for expressing the MRA numerical calculus algorithms, such as the compress operation, in a manner independent of the number of dimensions d. Since the number of inputs to a compress task is 2^d, changing d would require changing the flowgraph. Listing 3 shows how streaming terminal can be used to implement accumulation of the input node data sent to the compress task. Each compress task expects exactly 2^d inputs, hence the size of the stream expected by the input terminal can be passed directly to the `set_input_reducer` method.

```
reduce_leaves_tt->template set_input_reducer<0>(
  /* the reduction operator */
  [](FunctionReconstructedNode<T,K,NDIM> &&a,
     FunctionReconstructedNode<T,K,NDIM> &&b)
  {
    a.neighbor_coeffs[a.key.childindex()] = a.coeffs;
    a.is_neighbor_leaf[a.key.childindex()] = a.is_leaf;
    a.neighbor_sum[a.key.childindex()] = a.sum;
    a.neighbor_coeffs[b.key.childindex()] = b.coeffs;
    a.is_neighbor_leaf[b.key.childindex()] = b.is_leaf;
    a.neighbor_sum[b.key.childindex()] = b.sum;
    return a;
  },
  1 << NDIM /* the number of reductions to perform */
);
```

Listing 3: Accumulation of child nodes using a streaming terminal on input terminal 0 of the `reduce_leaves_tt` task template in the MRA benchmark.



(a) Strong scaling MRA: 4 to 32 nodes with 120 functions on *Seawulf*, using 2 processes per node with 20 threads each.



(b) Strong scaling MRA: 8 to 64 nodes with 400 functions on *Hawk*, using 8 processes per node with 16 threads each.

Fig. 13: Strong scaling MRA on *Seawulf* and *Hawk*.

Figures 13a and 13b show the results of strong-scaling MRA using *TTG* and native *MADNESS* on *Seawulf* up to 32 nodes and on *Hawk* up to 64 nodes. *TTG* over *PaRSEC* clearly outperforms *TTG* over *MADNESS* and native *MADNESS* on both machines. The benchmark uses plain-old-data (POD) structures for node data and the performance of *TTG* over *MADNESS* suffers due to data copies and high communication

overhead as compared to the efficient communication in *TTG* over *PaRSEC* which avoids unnecessary copying of data. The native *MADNESS* implementation scales up to 32 nodes on both machines. However, it reaches the scalability limit due to the existence of barriers at every step of the computation and re-allocation of data. We are investigating methods for reducing the communication overheads in *TTG* over *MADNESS*.

IV. Related Work

With the increase in hierarchy and complexity of the underlying hardware, maintaining a potential for high performance while abstracting the hardware to a simpler expression became critical. The literature is not short of proposals addressing this problem, including many evolutionary solutions that seek to extend the capabilities of current message passing paradigms with intra-node features (MPI+X). A different, more revolutionary, solution extends the flowgraph programming concepts as a substitute to both local and distributed data dependencies and synchronization management.

a) Flowgraph Programming: Flowgraphs, while ubiquitous as general models of computation (e.g., in compilers), have recently become featured as first-class concepts in programming models and languages aimed at high performance. Control-flow graph models include Taskflow [24], CUDA graphs [1]; TensorFlow [2] and Dask [33] APIs support dataflow graphs; Intel TBB [29] includes support for both control flow and dataflow graphs; CnC [11] and Legion [6] can support control or dataflow graphs through data partitioning and mapping. The most direct influence on *TTG* was Parametrized Task Graph, a programming model supported by *PaRSEC* in which computation is represented as flows of tuple-indexed data through an operation graph. Almost all of these programming models are implemented as C++ libraries. Most implementations limit the support for flowgraphs to shared memory setups, or use explicit communications transformed in tasks to simulate the flowgraph in a distributed setting. The Hume flowgraph DSL focuses on real-time embedded systems [20]. The S-NET DSL [19] is an orchestration language of tasks, strictly decoupling implementation and parallelism.

b) Runtimes: Numerous efforts to provide a similar level of abstraction via a fine-grain task-based dataflow programming exist, adding to those that have transitioned from a grid-based workflow toward a task-based environment. Some of the recent task-based runtimes like Legion [6], *StarPU* [5], HPX [22], CnC [11], *OmpSs* [17], DASH [34], *PaRSEC* [9] and *MADNESS* [21] act as an intermediary between the hardware resources and a programming paradigm, language or API to isolate application developers from the underlying hardware. Some of these programming interfaces have nascent support for distributed execution, e.g., recent versions of the OpenMP specification [32] introduce the *task* and *depend* clauses which can be employed to express control flow graphs. OpenMP is widely used and supports homogeneous, shared memory systems, and its *target* extension to support accelerators is quickly gaining traction. A limitation of the OpenMP model is that distributed memory and inter-node communication need to be explicitly implemented with the use of an external communication library.

In *OmpSs*, tasks are discovered by a single thread and executed by worker threads. The model allows nesting of tasks in individual nodes to relieve the main thread; however it may suffer from scalability issues on large scale distributed systems.

HPX aims to overcome these challenges by replacing explicit communications and synchronizations with asynchronous communication between nodes and lightweight control objects, allowing applications to exploit fine-grained parallelism within the context of a global address space.

Legion, on the other hand, describes logical regions of data and uses those regions to express the dataflow and dependencies between tasks, and defers to its underlying runtime, REALM [37], the scheduling of tasks, and data movement across distributed heterogeneous nodes.

V. Conclusion

Template Task Graph is a new flowgraph programming model that aims to lower the complexity of performance-portable parallel programming of (especially, irregular) complex applications by abstracting many details of the underlying task scheduling and execution as well as associated data and resource management. In this paper, we presented the current status of TTG's C++ distributed-memory implementation using two task-based runtime systems (*MADNESS* and *PaRSEC*). We evaluated these implementations over four paradigmatic applications, ranging from the most regular and compute intensive to applications whose execution is data dependent and memory bound. These evaluations show high performance and scalability, on par and sometimes exceeding the performance of state of the art implementations in other programming paradigms. We presented in details how the features of the language are exploited by the implementations to reduce memory copies and increase data management and communication. It must also be noted that the development cost, while a subjective measure, was certainly significantly lower compared with the state-of-the-art applications, and done by outsiders of the representative domain.

Future work will consider extensions to *TTG* to simplify data injection in the DAG of tasks, to better manage memory and network utilization, to provide some degree of Quality-of-Service with regard to the computation and communication scheduling, and to support heterogeneous platforms.

Acknowledgment

This research was supported partly by NSF awards #1450300, #1450344 and #1450262, 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 also acknowledge Advanced Research Computing at Virginia Tech (www.arc.vt.edu) for providing computational resources and technical support that have contributed to the results reported within this

paper. We gratefully acknowledge the provision of computational resources by the High Performance Computing Center (HLRS) at the University of Stuttgart, Germany.

References

[1] CUDA programming guide - CUDA graphs. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#cuda-graphs, 2021.

[2] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, et al. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, pages 265–283, 2016.

[3] B. Alpert. *Sparse Representation of Smooth Linear Operators*. PhD thesis, Yale University, 1990.

[4] B. Alpert, G. Beylkin, D. Gines, and L. Vozovoi. Adaptive Solution of Partial Differential Equations in Multiwavelet Bases. *Journal of Computational Physics*, 182(1):149–190, 2002.

[5] C. Augonnet, S. Thibault, R. Namyst, and P. Wacrenier. StarPU: A unified platform for task scheduling on heterogeneous multicore architectures. *Conc. Comp. Pract. Exper.*, 23:187–198, 2011.

[6] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. Legion: Expressing locality and independence with logical regions. In *Supercomputing*, 2012.

[7] Dan Bonachea and Paul H. Hargrove. GASNet-EX: A High-Performance, Portable Communication Library for Exascale. 10 2018.

[8] 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)*, pages 1–7, November 2020.

[9] George Bosilca, Aurelien Bouteiller, Anthony Danalis, Mathieu Faverge, Thomas Herault, and Jack Dongarra. PaRSEC: A programming paradigm exploiting heterogeneity for enhancing scalability. *Comp in Sc. and Eng.*, 99:1, 2013.

[10] George 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.

[11] Zoran Budimlić and Kathleen Knobe. CnC: A Dependence Programming Model. In *Proceedings of the Sixth Workshop on Data-Flow Execution Models for Extreme Scale Computing*, DFM'16. ACM, 2016.

[12] Alfredo Buttari, Julien Langou, Jakub Kurzak, and Jack Dongarra. A class of parallel tiled linear algebra algorithms for multicore architectures. *Parallel Computing*, 35(1):38 – 53, 2009.

[13] J Calvin and EF Valeev. TiledArray: A massively-parallel, block-sparse tensor framework written in C++. https://github.com/ValeevGroup/tiledarray, 2018.

[14] 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.

[15] Anthony Danalis, George Bosilca, Aurelien Bouteiller, Thomas Herault, and Jack Dongarra. PTG: An abstraction for unhindered parallelism. *Proceedings of WOLFHPC'14*, pages 21–30, 2014.

[16] H. Dang, R. Dathathri, G. Gill, A. Brooks, N. Dryden, A. Lenharth, L. Hoang, K. Pingali, and M. Snir. A Lightweight Communication Runtime for Distributed Graph Analytics. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2018.

[17] 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*, 37(3):292–305, 2009.

[18] Mark Gates, Jakub Kurzak, Ali Charara, Asim YarKhan, and Jack Dongarra. SLATE: Design of a Modern Distributed and Accelerated Linear Algebra Library. In *Supercomputing*, SC '19. Association for Computing Machinery, 2019.

[19] Clemens Grelck, Jukka Julku, and Frank Penczek. Distributed S-Net: Cluster and Grid Computing without the Hassle. In *2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid 2012)*, 2012.

[20] Kevin Hammond and Greg Michaelson. Hume: A domain-specific language for real-time embedded systems. In *Proceedings of the 2nd International Conference on Generative Programming and Component Engineering*, GPCE '03, page 37–56, Berlin, Heidelberg, 2003. Springer-Verlag.

[21] Robert J. Harrison, Gregory Beylkin, Florian A. Bischoff, Justus A. Calvin, George I. Fann, Jacob Fosso-Tande, Diego Galindo, Jeff R. Hammond, Rebecca Hartman-Baker, Judith C. Hill, Jun Jia, Jakob S. Kottmann, M.-J. Yvonne Ou, Laura E. Ratcliff, Matthew G. Reuter, Adam C. Richie-Halford, Nichols A. Romero, Hideo Sekino, William A. Shelton, Bryan E. Sundahl, W. Scott Thornton, Edward F. Valeev, Álvaro Vázquez-Mayagoitia, Nicholas Vence, and Yukina Yokoi. MADNESS: A multiresolution, adaptive numerical environment for scientific simulation. *SIAM J. Sci. Comput.*, 38(5):S123–S142, 2016.

[22] T. Heller, H. Kaiser, and K. Iglberger. Application of the ParalleX execution model to stencil-based problems. *Computer Science - Research and Development*, 28(2-3):253–261, 2013.

[23] Thomas Herault, Yves Robert, George Bosilca, Robert J. Harrison, Cannada A. Lewis, Edward F. Valeev, and Jack J. Dongarra. Distributed-memory multi-GPU block-sparse tensor contraction for electronic structure. In *35th IEEE International Parallel and Distributed Processing Symposium IPDPS*. IEEE, 2021.

[24] Tsung-Wei Huang, Dian-Lun Lin, Chun-Xun Lin, and Yibo Lin. Taskflow: A Lightweight Parallel and Heterogeneous Task Graph Computing System. *IEEE TPDS*, pages 1–1, 2021.

[25] Mohammad Mahdi Javanmard. *Parametric Multi-Way Recursive Divide-and-Conquer Algorithms for Dynamic Programs*. PhD thesis, State University of New York at Stony Brook, 2020.

[26] Mohammad Mahdi Javanmard, Zafar Ahmad, Martin Kong, Louis-Noël Pouchet, Rezaul Chowdhury, and Robert Harrison. Deriving parametric multi-way recursive divide-and-conquer dynamic programming algorithms using polyhedral compilers. In *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*, pages 317–329, 2020.

[27] Mohammad Mahdi Javanmard, Pramod Ganapathi, Rathish Das, Zafar Ahmad, Stephen Tschudi, and Rezaul Chowdhury. Toward efficient architecture-independent algorithms for dynamic programs. In *International Conference on High Performance Computing*. Springer, 2019.

[28] Zhenming Jin et al. Structure of Mpro from SARS-CoV-2 and discovery of its inhibitors. *Nature*, 582(7811):289–293, June 2020.

[29] Alexey Kukanov and Michael J Voss. The foundations for scalable multi-core software in intel threading building blocks. *Intel Technology Journal*, 11(4), 2007.

[30] Alfio Lazzaro, Joost VandeVondele, Jürg Hutter, and Ole Schütt. Increasing the Efficiency of Sparse Matrix-Matrix Multiplication with a 2.5D Algorithm and One-Sided MPI. In *Proceedings of the Platform for Advanced Scientific Computing Conference*, pages 1–9, Lugano Switzerland, June 2017. ACM.

[31] Poornima Nookala, Zafar Ahmad, Mohammad Mahdi Javanmard, Martin Kong, Rezaul Chowdhury, and Robert Harrison. Understanding Recursive Divide-and-Conquer Dynamic Programs in Fork-Join and Data-Flow Execution Models. In *2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 407–416. IEEE, 2021.

[32] OpenMP Architecture Review Board. OpenMP Application Programming Interface. Version 5.2. Technical report, November 2021.

[33] Matthew Rocklin. Dask: Parallel computation with blocked algorithms and task scheduling. In *Proceedings of the 14th python in science conference*, volume 130, page 136, 2015.

[34] Joseph Schuchart and José Gracia. Global Task Data-Dependencies in PGAS Applications. In *High Performance Computing*. Springer International Publishing, 2019.

[35] Joseph Schuchart, Christoph Niethammer, José Gracia, and George Bosilca. Quo Vadis MPI RMA? Towards a More Efficient Use of MPI One-Sided Communication, 2021.

[36] Edgar Solomonik and James Demmel. Communication-optimal parallel 2.5D matrix multiplication and LU factorization algorithms. In *Link.Springer.Com*. 2011.

[37] Sean Jeffrey Treichler. *Realm: Performance Portability through Composable Asynchrony*. PhD thesis, Stanford University, 2014.

[38] Unified Communication Framework Consortium. *UCX: Unified Communication X API Standard v1.6*. Unified Communication Framework Consortium, 2019.

[39] R. A. Van De Geijn and J. Watts. SUMMA: scalable universal matrix multiplication algorithm. *Concurrency: Practice and Experience*, 9(4), 1997.

[40] Gayathri Venkataraman, Sartaj Sahni, and Srabani Mukhopadhyaya. A blocked all-pairs shortest-paths algorithm. *Journal of Experimental Algorithmics (JEA)*, 8:2–2, 2003.