The Template Task Graph (TTG) — an emerging practical dataflow programming paradigm for scientific simulation at extreme scale

G. Bosilca*, R.J. Harrison†, T. Herault*, M.M. Javanmard†, P. Nookala†, E.F. Valeev‡
*University of Tennessee, Knoxville, USA. Email:{bosilca,herault}@icl.utk.edu
†IACS, Stony Brook University, USA. Email: {robert.harrison,,mhavanmard,poornimavinaya.nookala}@stonybrook.edu
‡Department of Chemistry, Virginia Tech, Blacksburg, U.S.A. Email: efv@vt.edu

***Abstract*—We describe TESSE, an emerging general-purpose, open-source software ecosystem that attacks the twin challenges of programmer productivity and portable performance for advanced scientific applications on modern high-performance computers. TESSE builds upon and extends the PARSEC DAG/-dataflow runtime with a new Domain Specific Languages (DSL) and new integration capabilities. Motivating this work is our belief that such a dataflow model, perhaps with applications composed in domain specific languages, can overcome many of the challenges faced by a wide variety of irregular applications that are poorly served by current programming and execution models. Two such applications from many-body physics and applied mathematics are briefly explored. This paper focuses upon the Template Task Graph (TTG), which is TESSE's main C++ API that provides a powerful work/data-flow programming model. Algorithms on spatial trees, block-sparse tensors, and wave fronts are used to illustrate the API and associated concepts, as well as to compare with related approaches.**

***Index Terms*—workflow, dataflow, high-performance computing, exascale, graph, DAG**

I. Introduction

TESSE (Task-based Environment for Scientific Simulation at Extreme Scale) [1] attacks the twin challenges of programmer productivity and portable performance for advanced scientific applications on massively-parallel hybrid systems with complex disjoint memories. Of specific interest are *irregular* computations which are hard to compose and execute efficiently with mainstream parallel programming paradigms.

There are several source of the Irregularity in modern advanced scientific applications. First, there is the irregularity of the data itself: as greater simulation size and/or higher precision are targeted dense data structures (uniform meshes, dense tensors) must be replaced by their data-sparse counterparts (adaptively-refined meshes, block-sparse and rank-sparse tensors) to keep the simulation cost tractable. Second, as applications become more complex (e.g., due to multiple physics models being coupled) and as they seek greater concurrency on increasingly heterogeneous hardware, it also becomes necessary to execute multiple (potentially, dissimilar) operations in parallel. The ensuing irregularity and associated resource management and other issues are commonly resolved by static partitioning of processors, or centralized job stealing. Both approaches have been successful as long as the computations involved were large enough to hide the cost of scheduling and/or migration, the two pillars of most of the existing solutions. Unfortunately, as exemplified by the two paradigmatic science applications motivating TESSE (fast tree-based computation on deeply refined numerical meshes, and block-sparse tensor algebra in many-body quantum simulation) exploiting sparsity usually leads to highly irregular fine-grained computation as well as highly data-dependent data/work flows that are very dynamic in nature.

To be able to compose and execute efficiently the irregular computation patterns underlying these and other modern scientific applications in TESSE we adopted the ideas of dataflow and other flow programming approaches to raise the level of abstraction beyond the relatively low-level APIs of modern task-based programming models and runtimes. The key innovation of TESSE is TTG, a flow programming model inspired by earlier innovations such as Flow-Based Programming (FBP) [2]. Unlike the earlier uses of flow programming, the targets of TTG are modern scientific algorithms to be deployed to current and near-future supercomputers, hence the efficient utilization of hardware resources, distributed-memory, and heterogeneity are all first-class concerns. Although the key innovations of TTG are not tied to particular choice of implementation, we implemented TTG as a library in C++ for general applicability and close-to-metal efficiency.

TTG can be viewed as marrying the ideas of flow programming models with the key innovations in the PARSEC runtime [3] for compact specification of task DAGs, namely the Parameterized Task Graph (PTG; section V) [4] in which each edge represents a flow of data associated with a *parameter* identifying the particular data and, equivalently, the receiving task. Simplistically, such a parameter represents a loop index or data structure coordinate (e.g., integer tuple addressing elements of a tensor). Thus, each edge and vertex in PTG encodes several edges and vertices in a DAG of tasks, allowing potentially massive task DAGs to be represented compactly as well as instantiated across a narrow wave front only as needed for execution. The algorithms of dense linear algebra are naturally expressed within the PTG as a workflow over mutable data that fully captures the lifetime of a datum

This work was supported by the National Science Foundation under grants OAC-1931387 and ACI-1450344 at SBU, OAC-1931347 and ACI-1450262 at VT, and OAC-1931384 and ACI-1450300 at UTK.

978-0-7381-1074-5/20/$31.00 ©2020 IEEE
DOI 10.1109/ESPM251964.2020.00011

including whether it is created, read, written, modified, or consumed by a given task. This parameterization combined with the deep understanding of what tasks are doing with data helps reduce resource utilization and data motion, and enables efficient placement and scheduling through use of temporal/spatial locality.

The TTG extends the idea of PTG by generalizing the notion of parameters to arbitrary types and enabling data-dependent selection of task dependencies, which allows to dynamically build the DAG of tasks depending on computations within the predecessor tasks. The TTG implementation replaces the standalone DSL for specifying PTG in PARSEC by a high-level programming API realized as a modern C++ library (the 2017 ISO standard of C++ is used). The use of modern C++ allows for type information to be utilized to ensure correctness when constructing graph as well as for optimizations (e.g. consuming data passed by rvalue references). Lastly, TTG also introduces some concepts from flow programming models, such as programmable terminals, that were not part of PTG. Thus TTG is a major advance of the successful idea of PTG towards general-purpose computation; unfortunately, the general-purpose character of TTG makes it challenging for TTG to retain all of the optimizations feasible within the PTG; making TTG exploit the full capabilities of PARSEC runtime is the focus on the ongoing work.

The main contributions and innovations of TTG are:

- A practical and modern C++ API for dependence programming (spanning work and data flow) of scientific applications on extreme-scale, hybrid supercomputers, which eliminates much of the clutter and programmer overhead that inhibits programmer productivity.
- TTG as a programming model abstracts details of the underlying runtime, and the present TTG implementation is designed to be portable onto multiple runtimes; it currently leverages the very distinct MADNESS and PARSEC parallel. The rich PARSEC ecosystem with its strong support of distributed hybrid architectures is presently our main target for performance portability.
- Use compile time and runtime information to efficiently manage the lifetime of data in different memory spaces.
- Generalizing the parameterization of tasks introduced by PARSEC's PTG to support irregular (sparse) computation in multiple settings.

In the following, we briefly describe the motivating MADNESS and TILEDARRAY scientific applications, and in the related work section describe the PARSEC runtime.

II. MOTIVATING APPLICATIONS

Adaptive Numerical Integro-differential Calculus

MADNESS [5] employs adaptive multiresolution algorithms and separated representations for efficient computation in many dimensions with guaranteed precision [6] with applications in chemistry [7], [8], and multiple fields of physics [9], [10]. To guarantee precision, each function has an independent and dynamically refined "mesh," and composing functions or applying operators can change the mesh refinement. These meshes are represented as 2^d-trees, where d is the dimensionality of the problem, and typically are poorly balanced and change very dynamically. Hence, computing with such trees on a parallel machine poses significant challenges.

Block-Sparse Tensor Algebra for Electronic Structure

Predictive treatment of quantum n-body electronic structure for realistic systems is only feasible by exploiting block-/element/rank sparsity. For example, the simple MP2 method solves (1) for unknown tensor $\mathbf{t}$:

$$0 = v_{ab}^{ij} + t_{ac}^{ij} f_b^c + t_{cb}^{ij} f_a^c - t_{ab}^{ik} f_k^j - t_{ab}^{kj} f_k^i \quad (1)$$

(with Einstein summation convention). A naive algorithm has $\mathcal{O}(n^5)$ cost, with n proportional to the number of atoms. However, with rank-compressed form the cost is $\mathcal{O}(n)$, enabling applications to hundreds of atoms on a single CPU [11].

Data sparsity of $\mathbf{t}$ in (1) unfortunately yields non-uniformly rank-sparse blocks for which there are no known equivalents to highly-efficient dense-matrix algorithms, such as SUMMA [12]. Task-based composition is a natural choice for parallel data-sparse tensor algebra by automating (1) mapping of irregular computation onto physical processor grid, (2) flow of data through the memory hierarchy, and (3) load balance including overlap of algorithmic steps.

To explore the potential of task-based computation for data-sparse tensor algebra some of us developed TILEDARRAY [13], a modern C++ library for parallel tensor algebra. TILEDARRAY is implemented on top of task-based parallel runtime of MADNESS. The current form of TILEDARRAY, already useful in practice, is limited by the relatively simple MADNESS runtime. We expect that moving TILEDARRAY to TESSE and TTG will enable 1) improved performance for both dense and block-sparse tensor algebra, 2) simplified development by automating management of resources, 3) computation on CPU/accelerator platforms, and 4) less-structured work patterns characteristic of data-sparse tensor algebra.

III. TTG— TEMPLATE TASK GRAPH

In this section, we provide a brief introduction to the TTG with more detail provided in Section IV.

Central concepts are:

- `TaskId`: A unique identifier for each task. For example, if computing on a vector it might be the vector/loop index, or if computing on a database it might be the name of a record, or in a matrix-multiplication algorithm it might be the triplet of integers identifying the tiles being operated upon. The only constraint on the type of `TaskId` is that it be hashable. The `TaskId` is used by the runtime to identify the compute resource (process rank, gpu, etc.) for the task using an optional user-defined map, and when a task sends data to a successor the same map is used to route data. This map is thus the primary tool for balancing load and data. A `TaskId` of type `void` implies a singleton task on process rank zero.

- `Terminal`: Each input argument and output result of a (template) task are exposed to the programmer and runtime as a `Terminal`. A task propagates a result or output value to a successor task by sending the value and the successor's `TaskId` to the appropriate output `Terminal`. Broadcast to multiple values of `TaskId` is supported. By default, an input `Terminal` is a single-assignment variable, this property being used by the runtime to determine when arguments of a task are available. However, an input `Terminal` is programmable and, for instance, could perform a reduction operation. If the number of expected input values is fixed, the runtime can determine completion, but with variable length (streaming) data either the user-provided reduction operation or a predecessor task must *finalize* the argument.
- `Edge`: Programs are composed by connecting output terminals with input terminals, currently identified by position but by name is planned. Multiple edges can connect to an input terminal, enabling data to come from multiple sources, and an output terminal might connect to multiple successors implying a broadcast operation.
- `TemplateTask`: This wraps a user-defined function with informal signature `void f(TaskId, Arg0, Arg1, ..., OutputTerminals)`. Again, each input argument is exposed as a `Terminal`, and `OutputTerminals` is a tuple of the output terminals (an alternative interface also provides the input arguments as a tuple of references). The task associated with a specific `TaskId` is instantiated when any input `Terminal` receives a value, and a task is marked ready for execution when all arguments are finalized. If there are no arguments, the task must be created either manually via a special method (`invoke(TaskId)`) of the `TemplateTask`, or via a pull operation as described below. Most users will instantiate a `TemplateTask` by invoking the `make_tt` factory function that deduces type information from the signature of the user's function, as illustrated below. However, a user-defined class can derive from the `TemplateTaskBase` class template using the curiously recurring template pattern that enables the base class to access methods of the derived class. As originally conceived and important for distributed-memory computers, tasks were assumed to only receive data through their input terminals and to have sending data to output terminals as their only side effect. However, there is no constraint on this behavior and work flow over mutable data is also readily composed.
- `CompositeTemplateTask`: This exposes the same API as `TemplateTask` but wraps an entire subgraph exposing input and output terminals as selected by the programmer.
- Push versus pull: As described so far, data must be *pushed* from a task's output terminal into a successor's input terminal. However, many algorithms, such as those operating on pre-existing data structures, can be more easily composed and more efficiently executed by pulling data as needed. This is accommodated by connecting terminals via a *pull*-`Edge`. When a task is instantiated, the runtime checks each input terminal to see if its value should be pulled, in which case the necessary predecessor task (the `TaskId` of which is computed from the current task's `TaskId` via a user-defined function) is instantiated. This can be done recursively and lightweight operations, such as reading a value from local memory, can be directly invoked to avoid the overhead of task creation.

Given a user function (`f`) with the required signature (see `TemplateTask` above), a call to the `make_tt` factory would be used as

```
auto tt = make_tt(f, input_edges, output_edges, task_name,
                  input_terminal_names, output_terminal_names);
```

in which `input_edges` and `output_edges` are possibly empty tuples of edges to connect to each terminal, and `task_name`, `input_terminal_names`, and `output_terminal_names` are optional names for the task and terminals.

IV. Applications

In this section, we examine the TTG implementation of several scientific algorithms: computation on unbalanced spatial trees in MADNESS to illustrate the overall API; a sparse SUMMA matrix multiplication algorithm [12] to illustrate block-sparse matrix/tensor computation from TiledArray; and wavefront computation to contrast TTG with Cpp-Taskflow [14]. In code excerpts (full code is online), the color red denotes types, functions or methods defined in TTG.

A. MADNESS *mini-app*

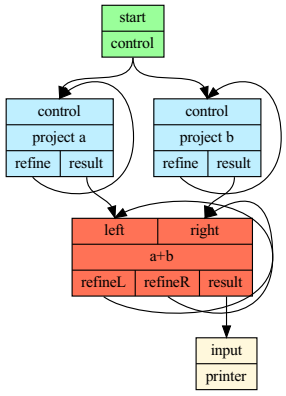


Fig. 1: TTG of the MADNESS mini-app fragment.

In this section, we explore code fragments from a mini-app that represents multiresolution analysis computation in MADNESS. The mini-app fragment adaptively projects two function ($a(x)$ and $b(x)$) into numerical tree representation, these in turn being added ($a(x)+b(x)$) and the result printed. The corresponding TTG is in figure 1 We omit definitions of the simple classes `Key` (the level and translation that labels a node in the tree), `Node` (the value and `has_children` flag associated with a node), and `Control` (empty) data structures, as well as some hopefully obvious typedefs. The listing immediately below adopts several common design motifs.

```
template <typename funcT>
auto make_project(const funcT& func, ctlEdge& ctl,
             nodeEdge& result, const string& name="project")
{
  auto f = [func](const Key& key, Control&& ctl,
                  std::tuple<ctlOut,nodeOut>& out) {
    // compute function value and approximation error
    if (error <= thresh) {
      send<1>(key, Node(value, false), out); // produce leaf
    } else {
      send<0>(key.left_child(), Control(), out);  // recur
      send<0>(key.right_child(), Control(), out); // recur
      send<1>(key, Node(0.0, true), out); // interior node
    }
  };
  ctlEdge refine("refine");
  return make_tt(f, edges(fuse(refine, ctl)),
                 edges(refine, result),
                 name, {"control"}, {"refine", "result"});
}
```

It defines a factory that given a numerical function (here, $a(x)$ or $b(x)$) returns a `TemplateTask` that will recursively perform the projection. The control terminal that triggers computation at a node (there being no other input data) will be connected to the control edge that triggers the overall computation at the top of the tree, and the output terminal to which nodes are sent will be connected to the result edge. To co-locate code definition with use and to facilitate capture of relevant state, the projection operation is defined as a lambda and then wrapped as a `TemplateTask` using the `make_tt` factory described above. Multiple input or output edges that are to be connected to one terminal are logically joined with the `fuse` operation, or could be connected individually in a separate step. The projection operation computes the function value and associated error (code elided) at a node in the tree, and then decides whether to produce a result (sent to output terminal 1), or continue refining by sending control flags to the left and right children and producing an empty interior node.

Similarly, in the next listing we define a factory that given a binary operation acting upon two tree nodes, applies this operation by recurring down the union of the two input trees until a common refinement level is reached.

```
template <typename funcT>
auto make_binary_op(const funcT& op, nodeEdge left,
                    nodeEdge right, nodeEdge Result,
                    const string& name = "binaryop")
{
  auto f = [&op](const Key& key, Node&& left, Node&& right,
                 tuple<nodeOut,nodeOut,nodeOut>& out) {
    if (!(left.has_children || right.has_children)) {
      send<2>(key, Node(op(left.value, right.value), false),
              out);// produce result
    } else {
      auto children = {key.left_child(), key.right_child()};
      if (!left.has_children)
         broadcast<0>(children, left, out);   // recur
      if (!right.has_children)
         broadcast<1>(children, right, out); // recur
      send<2>(key, Node(0.0, true), out); // interior
    }
  };
  nodeEdge L("L"), R("R");
  return make_tt(f, edges(fuse(left, L), fuse(right, R)),
              edges(L, R, Result), name, {"left", "right"},
              {"refineL", "refineR", "result"});
}
```

If both the left and right nodes have data the result (a leaf node with no children) can immediately be computed and sent to the next computational step. Otherwise, if either of the two nodes have data it must be sent down the tree along with an empty interior node. Note that input data can arrive in any order, though execution is more resource efficient in a downward traversal. We have omitted code to compute the value of a node from an ancestor as well as definition of similar factories to create the start and printer operations.

Finally, we examine the main driver.

```
ttg_initialize(argc, argv);
ctlEdge ctl("start␣ctl");
nodeEdge a("a"), b("b"), a_plus_b("a+b");
auto start = make_start(ctl);
auto pA = make_project(&A, ctl, a, "project␣A");
auto pB = make_project(&B, ctl, b, "project␣B");
auto addAB = make_binary_op(&plus<double>, a, b, a_plus_b,
                            "a+b");
auto printer = make_printer(a_plus_b);
if (!make_graph_executable(start.get()))
   error("graph␣is␣not␣connected");
if (ttg_default_execution_context().rank() == 0)
    start->invoke();
ttg_execute(ttg_default_execution_context());
ttg_fence(ttg_default_execution_context());
ttg_finalize();
```

Since TTG strives to be runtime agnostic, basic capabilities (e.g., initialize or finalize the parallel runtime, or access the process rank) are encapsulated, but, presently, anything beyond a simple example would likely need to use runtime-specific features. Each `TemplateTask` is created from the appropriate factory and connected using the declared edges. The graph is checked and made ready for execution. Starting execution presently involves two steps — injecting a start task that will initiate the recursion and then actually executing the TTG. However, this start task with control-edge paradigm is so common that it will likely become built in. During graph execution the main thread of each application process can conduct other work and eventually fence, waiting for the TTG to complete. Once its execution is complete, a TTG may be executed again on other data.

B. Block-Sparse Matrix Multiplication

Block-sparse tensor contraction in TILEDARRAY is implemented as a (block-sparse) matrix multiplication. Here we sketch out the TTG specification of the block-sparse SUMMA algorithm implemented in TILEDARRAY in task-based dataflow-like form explicitly using the MADNESS runtime. [15]

The traditional 2D SUMMA algorithm [12] maps the two inner loops of the familiar matrix multiplication loop nest

```
for(int k=0; k!=K; ++k)
  for(int i=0; i!=I; ++i)
    for(int j=0; j!=J; ++j)
       C[i][j] += A[i][k] * B[k][j]
```

onto a rectangular (2-dimensional) grid of processors, with the outer loop scheduled one-at-a-time, [12] several at a time, [15] or mapped onto the third dimension of the 3-dimensional processor in 2.5D SUMMA. [16] In a given SUMMA "iteration" (i.e. for the given value of k) the involved 1 (block) column of A and 1 (block) row of B are broadcast along the orthogonal directions of the 2-dimensional processor grid computing this k, then multiplied with each matching counterpart.

The TTG representing (block-sparse) matrix multiplication is shown in Figure 2. The TTG actually encodes not just SUMMA, but a family of SUMMA-like algorithms, all having in common the fact that the sum over k is evaluated one value of k at a time; other details, such as the order of summation over k, how computation is mapped onto processors, etc., are configurable attributes of the concrete instance of TTG. By abstracting out the orthogonal concerns from the essential details of the flow of data and computation TTG allows more compact and expressive specification of task-parallel algorithms.

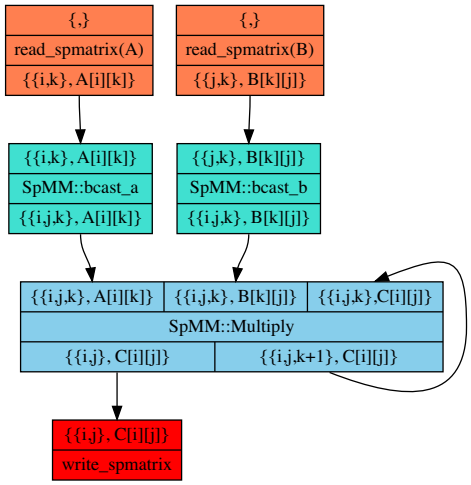


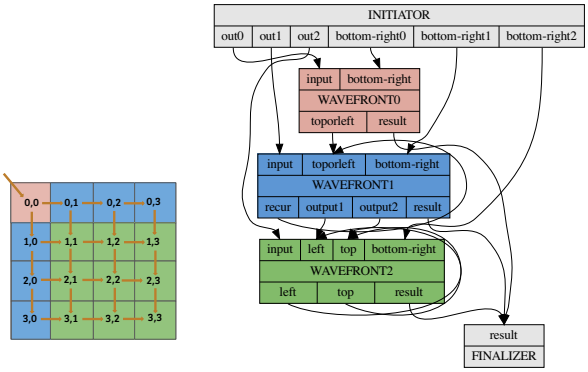
Fig. 2: TTG of the SUMMA-like matrix multiplication.

Note that the TTG also does not exhibit any details related to sparsity. Handling of sparsity (either element-wise and naive block-size) in matrix multiplication is trivially incorporated by skipping the loop iterations for which the requisite data is not available. In TILEDARRAY the sparsity is computed in the inspection stage using replicated meta-data. TTG's implementation of (block-)sparse matrix multiplication also incorporates such inspection stage, producing the metadata used by the reader, broadcast, multiply, and writer tasks to instantiate all non-null tasks. Thus the sparsity in this case is a yet-another concern orthogonal to the data/task flow details specified by the TTG.

C. Wavefront Computation

The input is an $N \times N$ matrix divided into blocks, and we reuse a 2D example provided by Cpp-Taskflow [14] with a 5-point stencil for which computation on block $B[i][i]$ requires data from all four neighbors $B[i-1][j]$, $B[i][j-1]$, $B[i+1][j]$, $B[i][j+1]$ but only has task dependencies on $B[i-1][j]$ and $B[i][j-1]$. Figure 3a shows the task dependencies between the blocks — blocks with same color can run concurrently. Computation starts at the top-left and sweeps the grid diagonally.

Fig. 3b shows the dataflow based TTG graph of wavefront computation. The kernels `WAVEFRONT0`, `WAVEFRONT1`, `WAVEFRONT2` correspond to blocks of the matrix with zero,



(a) Flow of computation (b) TTG graph using dataflow.

Fig. 3: 2D Wavefront Computation

one or two input dependencies, respectively, as color coded in Fig. 3a. The input terminals for each kernel define both the task and data dependencies, so a separate kernel is required for every node with a different number of inputs. Note there are tasks with different input dependencies starting from block $B[0][0]$ with bottom and right dependency, $B[0][1]$ to $B[0][N-2]$ with left, bottom and right dependencies, $B[1][0]$ to $B[N-2][0]$ with top, bottom and right dependencies, $B[0][N-1]$ with left and bottom dependency, $B[N-1][0]$ with top and right dependencies, $B[N-1][j]$ and $B[i][N-1]$ up to $B[N-1][N-1]$ have left and top dependencies. In the current dataflow implementation, we have used an array for holding the bottom/right blocks, which slightly reduces the complexity of the implementation by reducing the number of input/output terminals and connections between the kernels.

In Fig. 3b, the `Initiator` starts the computation by distributing the data blocks to all instances of the kernels along with the bottom and/or right blocks. It instantiates N^2 tasks upfront which can pose resource management challenges for huge problems sizes. However, this is inevitable in the current push-based data-flow approach of TTG. Cpp-Taskflow also requires instantiation of N^2 tasks. The coming ability of TTG to pull data will eliminate this.

```
using BMEdge = Edge<Key, BlockMatrix>;
template <typename funcT>
auto make_wavefront1(const funcT& func, int MB, int NB,
        BMEdge& input, BMEdge& toporleft,
        Edge<Key,std::vector<BlockMatrix>>& bottom_right,
        BMEdge& out1, BMEdge& out2, BMEdge& result)
{
  auto f = [MB, NB, func](const Key& key,
             BlockMatrix&& input, BlockMatrix&& previous,
             std::vector<BlockMatrix>&& bottom_right,
             std::tuple<Out<Key, BlockMatrix>,
             Out<Key, BlockMatrix>, Out<Key, BlockMatrix>,
             Out<Key, BlockMatrix>>& out)
    {
      auto [i, j] = key;
      int next_i = i + 1, next_j = j + 1;
      BlockMatrix res;
      int size = bottom_right.size();
      if (size == 1)
        res = func(i, j, MB, NB, input, previous, previous,
                   bottom_right[0], bottom_right[0]);
      else
        res = func(i, j, MB, NB, input, previous, previous,
                   bottom_right[0], bottom_right[1]);
      send<3>(Key(i,j), res, out);
```

```
      if (next_i < MB) {
        if (j == 0)
          send<0>(Key(next_i, j), res, out); //send top
        else
          send<2>(Key(next_i, j), res, out); //send top
      }
      if (next_j < NB) {
        if (i == 0)
          send<0>(Key(i, next_j), res, out); //send left
        else
          send<1>(Key(i, next_j), res, out); //send left
      }
    };
    return make_tt(f, edges(input, toporleft, bottom_right),
         edges(toporleft, out1, out2, result), "wavefront1",
         {"input", "toporleft", "bottom_right"},
         {"recur", "output1", "output2", "result"});
}
```

The above code fragment of the dataflow wavefront computation defines a `TemplateTask` for `WAVEFRONT1` kernel that recursively propagates along the wavefront. When the computation on a block is complete, it is sent to the "result" output terminal and, as necessary, via other terminals to successor tasks. The other two kernels handle different numbers of input dependencies. Computation starts in the main program by injecting block `(0,0)` which triggers the recursion for propagating the wavefront. A workflow version yields much simpler code since every task depends on zero, one or two tasks, but operates on shared data. The complexity of pushing data is hidden in this approach with task dependency only on $B[i-1][j]$ and $B[i][j-1]$ blocks.

V. Related work

Many efforts exist to provide abstractions via a fine-grain, task-based dataflow programming. Some of the recent task-based runtimes like Legion [17], StarPU [18], QUeuing And Runtime for Kernels (QUARK) [19], HPX [20], Open Community Runtime (OCR) [21], OmpSs [22], SuperMatrix [23], and PaRSEC [24] abstract the available resources to isolate developers from hardware complexity and simplify the writing of parallel applications. We briefly examine six projects.

a) Charm++: Since its introduction in 1993, Charm++ has evolved to support numerous features, including dynamic topology-aware load balancing [25]–[27] , migratable objects [28] , heterogeneous platforms [29], [30], check-pointing [31], [32] , hierarchical load balancing [33] , and integration with bulk-synchronous runtimes . In contrast to Charm++, TTG relies only upon standards-based compilers and libraries and strives to remain fully interoperable with other programming models within the same executable, including nesting and composition of programming models.

b) Concurrent Collections (CnC): [34], [35] CnC decouples control, work, and data with instances of these entities organized into collections. Work is embodied as tasks that produce and consume data instances, with control over task creation via *tag collections* and decoupling of program semantics from application tuning. Extensions to the original, shared memory CnC [36] support distributed platforms [37], CUDA [38] and heterogeneous architectures [39]. For the MADNESS mini-app, a major challenge was the inability of native CnC to reuse the implementation of a computational step, which is essential to build a library of reusable components.

c) High Performance ParallelX: (HPX) [40] is a task Global Address Space (GAS) runtime with a C++ API. The GAS system backing HPX, called AGAS (Active GAS), manages blocks referenced by fake pointers (*hpx_addr_t*). Blocks can be accessed globally through RDMA put/get calls or by using parcels (small functions to be executed on the home node of an associated block). Control flow between tasks is defined by local control objects, and support for accelerators exists through specialized executors, but similarly to OpenMP both the data transfers and the execution transfer is explicit, at the charge of the developer, whereas TTG via PaRSEC manages and optimizes this automatically.

d) Legion: [17] A programming model and runtime conceptually designed for execution on heterogeneous architectures, Legion uses logical regions as first-class citizens to allow data organization. Legion also implements a *software out-of-order processor* task scheduling, hierarchical region partitioning [41] and slicing [42]. The framework is complemented by Regent [43], a high-productivity imperative and structured DSL that simplifies the implementation of Legion programs, and by more specialized DSLs such as Singe [44] and Scout [45]. However, mapping irregular computations and unbalanced trees such as those of the MADNESS miniapp on Legion's regular index/data spaces remains a challenge especially in high dimension spaces.

e) CPP-TASKFLOW: [14] Cpp-Taskflow is a C++ tasking library that handles both for-loops and irregular patterns such as graph traversal. Cpp-Taskflow supports template instantiation to compose dependency graphs which can be dumped to the DOT format for inspection and debugging. However, it is limited to shared-memory architectures and all tasks must be instantiated beforehand to define dependencies.

f) PaRSEC*:* provides a high-efficient distributed low-level task-based runtime supporting a varied set of Domain Specific Languages (DSL) programming languages and APIs [3], [46]. PaRSEC can address any problem that can be expressed as graphs of tasks with labeled edges, where the edges represent data or control dependencies, with few restrictions. Existing usage scenarios for PaRSEC have showed benefit (in terms of performance and composability) for regular and irregular affine algorithms (being used as the underlying runtime for several linear algebra libraries [47], [48]), but most of the existing PaRSEC DSL are not tailored to support irregular dynamic data-dependent applications.

VI. Summary

We have described and illustrated the use of TTG, an emerging C++ API that provides a powerful work/data-flow programming model focused on irregular computation with special focus on efficient execution at extreme scale on current and near-future heterogeneous hardware. It is presently undergoing rapid development but already runs at scale, including use of multi-GPU nodes, on both the MADNESS and PaRSEC runtimes.

REFERENCES

[1] R. Harrison *et al.*, "TTG: Template task graph c++ api," https://tesseorg.github.io and https://github.com/TESSEorg/tesse-cxx, 2019.

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

[3] G. Bosilca *et al.*, "DAGuE: A generic distributed DAG engine for high performance computing," *Parallel Computing*, vol. 38, no. 1-2, pp. 37 – 51, 2012.

[4] A. Danalis, G. Bosilca, A. Bouteiller, T. Herault, and J. Dongarra, "PTG: An abstraction for unhindered parallelism," in *Fourth International WOLFHPC Workshop*, 2014, pp. 21–30.

[5] R. J. Harrison, "MADNESS: Multiresolution adapative environment for numerical scientific simulation written in C++," https://github.com/m-a-d-n-e-s-s/madness, 2019.

[6] R. J. Harrison *et al.*, "MADNESS: A multiresolution, adaptive numerical environment for scientific simulation," *SIAM J. Sci. Comput.*, vol. 38, no. 5, pp. S123–S142, 2016.

[7] R. J. Harrison, "A Krylov subspace accelerated inexact newton method for linear and non-linear equations," *Journal of Computational Chemistry*, vol. 25, p. 328, 2004.

[8] F. Bischoff, R. Harrison, and E. Valeev, "Computing many-body wave functions with guaranteed precision: the first-order moller-plesset wave function for the ground state of helium atom," *J. Chem. Phys.*, vol. 137, 2012.

[9] J. Pei *et al.*, "Coordinate-space hartree-fock-bogoliubov for superfluid fermi systems in large boxes," *J. Phys. Conf. Ser. 402*, 2012.

[10] N. Vence, R. Harrison, and P. Krstić, "Attosecond electron dynamics: A multiresolution approach," *Phys. Rev. A*, vol. 85, p. 033403, Mar 2012.

[11] P. Pinski, C. Riplinger, E. F. Valeev, and F. Neese, "Sparse Maps – A systematic infrastructure for reduced-scaling electronic structure methods. 1. An efficient and simple linear scaling local MP2 method that uses an intermediate basis of pair natural orbitals," *J. Chem. Phys.*, vol. 143, p. 034108, 2015.

[12] R. A. Van De Geijn and J. Watts, "SUMMA: scalable universal matrix multiplication algorithm," *Concurrency: Pract. Exper.*, vol. 9, no. 4, pp. 255–274, Apr. 1997.

[13] J. A. Calvin and E. F. Valeev, "Tiledarray: A massively-parallel, block-sparse tensor library written in c++," 2014. [Online]. Available: https://github.com/valeevgroup/tiledarray

[14] G. G. Tsung-Wei Huang, Chun-Xun Lin and M. Wong, "Cpp-Taskflow: Fast task-based parallel programming using modern c++," *IPDPS'19*, pp. 974–983, 2019.

[15] J. A. Calvin, C. A. Lewis, and E. F. Valeev, "Scalable task-based algorithm for multiplication of block-rank-sparse matrices," in *IA3 '15*. New York, New York, USA: ACM Press, 2015, pp. 1–8.

[16] E. Solomonik and J. Demmel, "Communication-optimal parallel 2.5D matrix multiplication and LU factorization algorithms," in *Euro-Par 2011 Parallel Processing*, 2011, pp. 90–109.

[17] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken, "Legion: Expressing locality and independence with logical regions," in *Supercomputing*. IEEE, 2012, pp. 1–11.

[18] 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, pp. 187–198, 2011.

[19] E. Agullo *et al.*, "Numerical Linear Algebra on Emerging Architectures: The PLASMA and MAGMA Projects," *Journal of Physics: Conference Series*, vol. 180, 2009.

[20] 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.

[21] J. Dokulil, M. Sandrieser, and S. Benkner, "Implementing the Open Community Runtime for Shared-Memory and Distributed-Memory Systems," *Proceedings of PDP 2016*, pp. 364–368, 2016.

[22] 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, pp. 292–305, 2009.

[23] E. Chan, E. S. Quintana-Orti, G. Quintana-Orti, and R. van de Geijn, "Supermatrix out-of-order scheduling of matrix operations for SMP and multi-core architectures," ser. SPAA '07. ACM, 2007, pp. 116–125.

[24] G. Bosilca *et al.*, "PaRSEC: A programming paradigm exploiting heterogeneity for enhancing scalability," *Comp in Sc. and Eng.*, vol. 99, p. 1, 2013.

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

[26] R. Vasudevan, S. S. Vadhiyar, and L. V. Kalé, "G-charm: an adaptive runtime system for message-driven parallel applications on hybrid systems," in *Supercomputing*. ACM, 2013, pp. 349–358.

[27] A. Bhatelé, L. V. Kalé, and S. Kumar, "Dynamic topology aware load balancing algorithms for molecular dynamics applications," in *Supercomputing*. ACM, 2009, pp. 110–116.

[28] L. V. Kale and G. Zheng, "Charm++ and AMPI: Adaptive runtime strategies via migratable objects," *Advanced Computational Infrastructures for Parallel and Distributed Applications*, pp. 265–282, 2009.

[29] M. P. Robson, R. Buch, and L. V. Kale, "Runtime coordinated heterogeneous tasks in Charm++," in *ESPM2*. IEEE, 2016, pp. 40–43.

[30] D. Kunzman, G. Zhang, E. Bohm, and L. V. Kale, "Charm++, offload API, and the cell processor," *Urbana*, vol. 51, p. 61801, 2006.

[31] G. Zheng, L. Shi, and L. V. Kalé, "FTC-Charm++: an in-memory checkpoint-based fault tolerant runtime for Charm++ and MPI," in *Cluster Computing*. IEEE, 2004, pp. 93–103.

[32] G. Zheng, C. Huang, and L. V. Kalé, "Performance evaluation of automatic checkpoint-based fault tolerance for AMPI and charm++," *ACM SIGOPS Op. Sys. Review*, vol. 40, no. 2, pp. 90–99, 2006.

[33] G. Zheng, E. Meneses, A. Bhatele, and L. V. Kale, "Hierarchical load balancing for charm++ applications on large supercomputers," in *ICPPW'10*. IEEE, 2010, pp. 436–444.

[34] Z. Budimlić *et al.*, "Concurrent collections," *Scientific Programming*, vol. 18, no. 3-4, pp. 203–217, 2010.

[35] M. G. Burke, K. Knobe, R. Newton, and V. Sarkar, "Concurrent collections programming model," in *Encyclopedia of Parallel Computing*. Springer, 2011, pp. 364–371.

[36] Z. Budimlic *et al.*, "Multi-core implementations of the concurrent collections programming model," in *CPC'09: 14th International Workshop on Compilers for Parallel Computers*, 2009.

[37] F. Schlimbach, J. C. Brodman, and K. Knobe, "Concurrent collections on distributed memory theory put into practice," in *PDP'13*. IEEE, 2013, pp. 225–232.

[38] M. Grossman, A. S. Sbîrlea, Z. Budimlić, and V. Sarkar, "CnC-CUDA: declarative programming for GPUs," in *Intl. Workshop on Languages and Compilers for Parallel Computing*. Springer, 2010, pp. 230–245.

[39] A. Sbîrlea, Y. Zou, Z. Budimlíc, J. Cong, and V. Sarkar, "Mapping a data-flow programming model onto heterogeneous platforms," in *ACM SIGPLAN Notices*, vol. 47, no. 5. ACM, 2012, pp. 61–70.

[40] H. Kaiser, T. Heller, B. Adelstein-Lelbach, A. Serio, and D. Fey, "HPX: A task based programming model in a global address space," ser. PGAS '14. ACM, 2014, pp. 6:1–6:11.

[41] S. Treichler, M. Bauer, and A. Aiken, "Language support for dynamic, hierarchical data partitioning," in *ACM SIGPLAN Notices*, vol. 48, no. 10. ACM, 2013, pp. 495–514.

[42] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken, "Structure slicing: Extending logical regions with fields," in *Supercomputing*. IEEE Press, 2014, pp. 845–856.

[43] E. Slaughter, W. Lee, S. Treichler, M. Bauer, and A. Aiken, "Regent: a high-productivity programming language for hpc with logical regions," in *Supercomputing*. ACM, 2015, p. 81.

[44] M. Bauer, S. Treichler, and A. Aiken, "Singe: leveraging warp specialization for high performance on gpus," in *ACM SIGPLAN Notices*, vol. 49, no. 8. ACM, 2014, pp. 119–130.

[45] P. McCormick *et al.*, "Exploring the construction of a domain-aware toolchain for high-performance computing," in *WOLFHPC*. IEEE, 2014, pp. 1–10.

[46] G. Bosilca, "ParSEC: a generic framework for architecture aware scheduling and management of micro-tasks on distributed many-core heterogeneous architectures." https://bitbucket.org/icldistcomp/parsec, 2019.

[47] Q. Cao *et al.*, "Extreme-scale task-based cholesky factorization toward climate and weather prediction applications," ser. PASC '20, 2020.

[48] G. Bosilca *et al.*, "Flexible development of dense linear algebra algorithms on massively parallel architectures with DPLASMA," in *Proceedings of PDSEC 2011*, may 2011.