2018 IEEE International Parallel and Distributed Processing Symposium

BabelFlow: An Embedded Domain Specific Language for Parallel Analysis and Visualization

Steve Petruzza
University of Utah
spetruzza@sci.utah.edu

Sean Treichler
Stanford University
sjt@cs.stanford.edu

Valerio Pascucci
University of Utah
pascucci@sci.utah.edu

Peer-Timo Bremer
Lawrence Livermore National Lab
bremer5@llnl.gov

***Abstract*—The rapid growth in simulation data requires large-scale parallel implementations of scientific analysis and visualization algorithms, both to produce results within an acceptable timeframe and to enable in situ deployment. However, efficient and scalable implementations, especially of more complex analysis approaches, require not only advanced algorithms, but also an in-depth knowledge of the underlying runtime. Furthermore, different machine configurations and different applications may favor different runtimes, i.e., MPI vs Charm++ vs Legion, etc., and different hardware architectures. This diversity makes developing and maintaining a broadly applicable analysis software infrastructure challenging.**

We address some of these problems by explicitly separating the implementation of individual tasks of an algorithm from the dataflow connecting these tasks. In particular, we present an embedded domain specific language (EDSL) to describe algorithms using a new task graph abstraction. This task graph is then executed on top of one of several available runtimes (MPI, Charm++, Legion) using a thin layer of library calls. We demonstrate the flexibility and performance of this approach using three different large scale analysis and visualization use cases, i.e., topological analysis, rendering and compositing dataflow, and image registration of large microscopy scans. Despite the unavoidable overheads of a generic solution, our approach demonstrates performance portability at scale, and, in some cases, outperforms hand-optimized implementations.

***Keywords*-Embedded DSL; User productivity; In-situ analysis; Simulation runtime systems; Programming models**

I. Introduction

Two of the prevailing trends in large-scale scientific computing are the move toward in situ analysis, to avoid the growing I/O bottleneck, and the adoption of new simulation runtimes, such as Legion or Charm++, to manage the increasing parallelism. Unfortunately, when combined, these trends create a significant challenge for developers of analysis packages. The ideal analysis library should be compatible with any relevant application, while simultaneously being highly optimized in order to minimize the impact on the main simulation. Furthermore, developing efficient and scalable algorithms for comparatively unstructured problems such as feature detection, clustering or streamline computation is challenging. While there exist solutions, these are typically specialized implementations, hand tuned for particular software stacks, architectures and host applications [1], [2], [3], [4], [5], [6], [7]. Although it is possible to interoperate between runtimes [8] this significantly increases the complexity of integrating an analysis routine with the main application, adds build complexity and dependencies on additional software stacks, and typically carries a performance penalty. In practice, the burden is placed on the developer of the analysis package to provide native ports or interfaces customized to the chosen runtime of the host application. However, this requires library developers to be proficient in a wide range of runtimes and maintain an ever-growing suite of specialized implementations, which is too time consuming to be practical.

In order to improve user productivity and avoid maintaining multiple implementations for different runtimes, we propose a new task-based abstraction that explicitly separates the description and implementation of an algorithm from the underlying runtime. More specifically, we present: 1) an Embedded Domain Specific Language (EDSL) that describes an algorithm as a *task graph*; and (2), a thin layer of library calls to execute the task graph with different runtime backends. Together, these two components create *BabelFlow*, a unifying framework that allows developers to maintain a single implementation of an algorithm that nevertheless provides a native interface and efficient implementation for a number of different software stacks. To demonstrate the flexibility of our approach, we present results from three disparate use cases: topological feature detection; rendering and image compositing; and image registration of large microscopy scans.

Beyond the immediate benefit of an easy-to-integrate and easy to maintain analysis library, the framework offers a number of additional advantages. First, the description provides an inherent separation of concerns in which the algorithm developer is not exposed to any communication, synchronization or other runtime-related concepts. This allows the communication and algorithm to be developed and tested separately, and the different backends provide an ideal environment for regression testing. Second, the design naturally allows over-decomposition, which is not only useful for runtimes that provide load balancing but also simplifies debugging at scale. Any backend can execute task graphs of arbitrary size, on a single node or even serially, while guaranteeing a correct order of execution.

1530-2075/18/$31.00 ©2018 IEEE
DOI 10.1109/IPDPS.2018.00056

Finally, since the framework guarantees the same tasks are executed, independent of the runtime, it provides an ideal test bed to compare and contrast how different runtimes execute various workloads. Combined, BabelFlow represents a domain specific software layer between analysis algorithms and different simulation runtimes that provides both portable performance and high user productivity.

Our contributions in details are:

- An EDSL that uses a simple C++ API to express large scale parallel algorithms;
- Three different backends to execute task graphs in the native MPI, Charm++ or Legion runtime; and
- Detailed scaling studies for three distinct algorithms implemented using the proposed EDSL and executed on three different runtimes.

II. Related Work

As discussed above, data analysis and visualization is increasingly dependent on parallel implementations both to deal with the growing data sizes as well as to adapt to the needs of in situ deployment. This has given rise to a number of libraries and frameworks to simplify and accelerate the development of analysis algorithms. Some projects like ADIOS [9], Glean [10] or Dataspaces [11] focus on the data movement aspect within parallel applications, between computational resources and to the I/O subsystem. Other techniques, like DIY [12], focus more directly on providing reusable implementation of common analysis patterns, i.e., reductions, binary swaps, etc. There also exist in situ focused APIs to both the VisIt [13], [14] and the Paraview [15], [16] frameworks. However, these efforts are generally focused on simplifying the development of algorithms within a given software stack. For example, expressing an algorithm in DIY will provide an efficient MPI implementation, and working within ADIOS, Glean or Dataspaces will provide an easy integration with simulations already using these frameworks. Finally other approaches, like ParSEC [17], enable portability over heterogenous architectures using a task based algorithm representation but still do not help integrating with large scale simulations where the runtime system has been already chosen. We focus, instead, on the orthogonal issue: how to quickly and transparently port algorithms to different software stacks in order to easily integrate with different applications. In fact, just as BabelFlow provides backends for MPI, Legion and Charm++ currently, it could target higher level frameworks such as ADIOS or Glean in the future. Similarly, the system can exploit new data models such as Conduit [18] to transparently access simulation data and further uncouple the implementation of an algorithm from the specific application that uses it.

Conceptually, our approach is more similar to toolkits, such as VTK-m [19] and its various predecessors [20], [21], [22], which aim to express algorithms for various processor

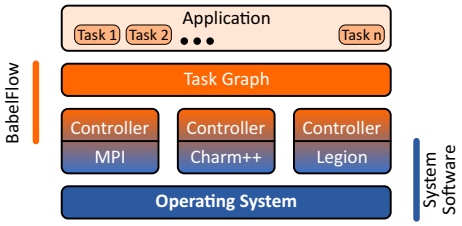


Figure 1: BabelFlow architecture: The application implements a set of tasks and expresses its dataflow using the EDSL in the form of a task graph. A BabelFlow controller, implemented natively in one of several runtimes, then executes the task graph.

architectures, i.e., GPUs vs. many core. However, VTK-m is focused primarily on fine-grain, on-node parallelism with backends to CUDA [23] or OpenMP [24]. Instead, our approach targets more coarse-grain parallelism and distributed parallel runtimes. In this aspect our dataflow matches common serial analysis pipelines, most notably VTK [25], which have long used connected sets of so-called filters to express a dataflow.

III. Dataflow EDSL

The core of our approach is an EDSL to describe parallel algorithms via *task graphs* and execute them via a set of *runtime controllers*. The task graph describes a given algorithm as a set of idempotent tasks and the dataflow connecting them (see Figure 1). The *runtime controller* is responsible for the parallel execution of the workflow using a chosen runtime system. Both the task graph and the controllers use a simple C++ API that defines base classes to be implemented and extended. In particular, the user is required to perform three basic steps: first, implement all tasks used during the algorithm; second, provide deserialization/serialization routines for the objects that are exchanged between tasks; and third, extend the `TaskGraph` class to describe the dataflow. The first two are generic and required in some form for any implementation. The third represents a procedural description of the task graph and can be as simple as a modulo operation for a reduction, as shown in Listing 2 and 3. We also provide prototypical implementations of common task graphs, such as reductions and broadcasts for users to use or modify. Finally, the implementations of the tasks are connected to the task graph by registering the corresponding callbacks, and the graph is passed to a controller for execution.

For example, a visualization workflow to perform a volume rendering followed by an image compositing is shown in Listing 1.

Listing 1: Example of volume rendering and compositing reduction dataflow.

```
int volume_render(vector<Payload>& in,
```

```
                    vector<Payload>& out, TaskId id);
int composite(      vector<Payload>& in,
                    vector<Payload>& out, TaskId id);
int write_image(    vector<Payload>& in ,
                    vector<Payload>& out, TaskId id);

// Reduction tree + additional wrap-up task
Reduction graph(block_decomp, valence);

// Define a TaskMap
ModuloMap task_map(n_procs, graph.size());

// Choose the controller
DataFlow::Runtime_Name::Controller c;
c.initialize(graph, &task_map);

// Register the callbacks
vector<CallbackId> avail_cid = graph.callbacks();
// Leaf task will volume render the local data
c.registerCallback(avail_cid[0], volume_render);
// Internal nodes will composite the image
c.registerCallback(avail_cid[1], composite);
// The wrap-up task will write the image
c.registerCallback(avail_cid[2], write_image);

// Set initial inputs and start execution
map<TaskId,Payload> initial_inputs;
// Populate initial inputs
c.run(initial_inputs);
```

In this example, we want to perform a distributed volume rendering using block-decomposed data. The algorithm has three stages that translate into three different tasks: volume rendering of a local block, compositing multiple images and writing the resulting image. Here we use a simple reduction tree with one additional task at its root. The task graph requires two parameters: the number of blocks used and the reduction factor the user desires. In this case, the `Reduction` task graph provides three different task types: one for the leaf nodes, one for the internal nodes of the reduction and one for the additional root task. A simple `ModuloMap` (defined in Listing 3) is used to allocate the tasks in round robin over the available resources. As discussed below, only the MPI and some variants of the Legion controller require this explicit load balancing. To link the task graph to the implementation, one simply assigns the different callbacks to the task types. Each task implementation uses a generic signature based on the concept of *Payloads*, which is either a pointer to an in-memory object or a binary buffer. Next, a chosen runtime controller is instantiated and initialized with the graph. Finally, the user hands off the input data by assigning Payloads to leaf tasks and starts the execution.

The Listing 1 example shows a post-process type computation in which the entire graph is started by a single controller instance. In practice, the in-situ coupling to a host application would be handled according to each runtime's execution model (see Section V) . For example, in MPI the graph is split across the ranks, and each rank instantiates only its assigned subgraph. Similarly, the subgraph requires only data local to the specific rank. Then, each MPI rank instantiates a controller that executes the local graph. In this respect, Legion uses a similar model based on the concept of a *shard*, while in Charm++ the task graph is entirely defined by the main *chare* (and scalably instantiated by the runtime), but the data assignment is handled by the standard remote procedure calls.

One central advantage of our approach is its flexibility in quickly and easily reassembling different algorithms. For example, changing the callbacks in the listing above, one can also compute global statistics or execute any number of reduction-based algorithms. Similarly, the target runtime can be exchanged, or the height and valence of the reduction can be modified easily. At the core of the EDSL lies the task graph defined as a set of *logical tasks*, each of which stores: a globally unique *task id*, task ids of tasks that will provide inputs and receive outputs and a *task type* identifying which callback to use. Special task ids are reserved for external inputs, e.g., data from a simulation or disk. To simplify the Ids generation, different portions of the graph, such as the embedded reduction or the various broadcast patterns, can be assigned unique prefixes and then can use the traditional modulo type operations to assign postfix Ids. In practice, task graphs may contain millions of nodes. Therefore, fully instantiating a graph on every core or node of a simulation is not scalable. Instead, we typically rely on procedural descriptions, which allow any part of the framework to query the global task graph. A query typically instantiates a set of logical tasks restricted to small local subgraphs, for example the subset of tasks that will be running on a certain rank (i.e., MPI). Ultimately, the runtime controllers convert logical tasks into *physical tasks* with space allocated for inputs and outputs and schedule them for execution according to their internal taskmodel. We currently provide a set of common dataflow graphs for reductions, broadcasts, binary swaps, neighbor and k-way merge dataflows. The user can utilize any of the provided graphs or derive new extensions as needed.

The basic *TaskGraph* interface requires the user to implement only two functions: 1) compute the total number of tasks, and 2) return a logical task corresponding to a task id. In addition, the MPI and some version of the Legion controller use the concept of a *task map* that, given an MPI rank or a shard, provides a list of tasks assigned to it. Listing 2 provides a C++ style description of a reduction task graph implementing all functions needed for all three runtimes. For simplicity, we assume a k-way reduction with k^d many tasks.

Listing 2: All functionality needed for a k-way reduction across all three runtimes. For simplicity we assume there are k^d many leafs

```
Reduction::Reduction(int leafs, int valence) {
  // Set the valence
  k = valence;
  // Assuming k^d leafs
  d = log(leafs,valence);
  n_tasks = (pow(k,(d+1))-1) / (k-1)

  // Add supported task types (i.e., callback ids)
  callback_ids.push_back(LEAF_CB);
```

```
  callback_ids.push_back(REDUCE_CB);
  callback_ids.push_back(ROOT_CB);
}

// Get callbacks Ids
vector<int> Reduction::callbacks()
{ return callback_ids; }

// Create a logical task from an id
Task Reduction::task(int task_id) {
  Task t;
  t.id = task_id;

  // Assign the input for a leaf
  if (task_id >= (n_tasks - pow(k,d)))
      t.callback_id = LEAF_CB;
  else { // Assign inputs for other tasks
      incoming.resize(k);
      for (int i=0; i < k; i++)
        t.incoming[i] = task_id*k+i+1;
  }

  // Assign the output for the root task
  if (task_id == 0)
      t.callback_id = ROOT_CB;
  else {
  // Assign the output for the other tasks
      t.callback_id = REDUCE_CB;
      t.outgoing.resize(1);
      t.outgoing[0].resize(1);
      t.outgoing[0][0] = (task_id-1)/k;
  }
  return t;
}

// Return all tasks for a given shard
vector<Task> Reduction::localGraph(
        TaskMap map, int shard_id) {
  vector<Task> graph;

  // Get a list of all task ids for this group
  vector<int> ids = map.getIds(shard_id);

  for (auto id : ids)
    graph.push_back(task(id));

  return graph;
}

// Return the number of tasks in the graph
int Reduction::size(){ return n_tasks; }
```

Listing 3: TaskMap example that maps the tasks using a simple modulo operation

```
ModuloMap::ModuloMap(int shard_count, int task_count) :
        TaskMap(), mShardCount(shard_count),
        mTaskCount(task_count){}

// Return the shard id for the given task id
int ModuloMap::shard(int task_id) const
{ return (task_id % mShardCount); }

// Return the list of task ids for the given shard id
vector<int> ModuloMap::getIds(int shard_id) const{
  vector<int> back;
  int t = shard_id;

  while (t < mTaskCount) {
    back.push_back(t);
    t += mShardCount;
  }
  return back;
}
```

The definition of `localGraph` and `callbacks`, which is generic, is provided in the base class, as are the default versions of the task maps (e.g., `ModuloMap`). This leaves the user to initialize the graph according to the desired size and to define the function (i.e., *localGraph*) to compute the logical tasks assigned to a specific shard or rank using the given *task map*. In practice, the only unfamiliar aspect of implementing an algorithm using BabelFlow is the definition of the task graph. It requires the user to explicitly define task ids for all tasks and express the necessary communication in terms of these task ids. However, the corresponding index space does not have to be contiguous, which makes it straight forward to define prefixes for different phases of the algorithm and use some intuitive numbering within each phase. For example, the graph of the merge tree computation shown in Figure 5 can be separated into rounds on all leafs (local computation, correction, segmentation), a reduction (all joins) and several broadcasts (relay) patterns, each of which has a simple default ordering. Furthermore, we provide the ability to draw the abstract task graph (or subsets of it) in Dot [26], a graph layout tool that makes debugging simple and intuitive.

IV. Runtime Controllers

Once a (local) task graph has been instantiated and its tasks populated with callbacks, it is used by different *runtime controllers* to generate a set of *physical tasks* and ultimately instantiate these tasks according to the execution model, of the specific runtime. Since the controllers are natively implemented in the chosen runtime model they seamlessly integrate with a host application using the same runtime. However, each runtime system has a different data and execution model. In particular, each runtime has different ways to:

- evaluate dependencies and schedule tasks;
- manage data and communication; and
- distribute the computation over the available resources.

The *task graph* representation, as created by the EDSL, explicitly provides all data dependencies, which in turn determine a (partial) order of execution. The graph does not determine the task mapping, i.e., which particular processors a task should be executed in, or what the optimal scheduling policy might be. Instead, each runtime controller is responsible for translating the high-level EDSL model into its internal representation as well as possible. All runtime controllers share the same interface by deriving from the same base class to make switching between controllers easy. Below we discuss the implementation of three different runtime controllers for MPI, Charm++ and Legion, respectively. Therefore, the implementations represent an initial effort and especially for the less common runtimes (Charm++ and Legion) could likely be optimized further. However, we are closely collaborating with the relevant experts regarding some of the unexpected results of the scaling studies in Sec. V.

A. MPI Controller

The MPI controller uses a static allocation of the tasks and asynchronous point-to-point messages for communication. To determine the task allocation, it uses a *task map* to determine which tasks are assigned to which MPI rank. Note that not all MPI ranks must be assigned tasks nor is there a limit on how many tasks each rank can be assigned. In particular, distributing tasks among fewer ranks provides a direct trade-off between distributed and shared memory parallelism. Furthermore, executing a task graph on fewer (or even a single) ranks has proven useful for debugging a given dataflow.

Each MPI rank instantiates a separate controller in its main thread which will be responsible for scheduling tasks and handling the communication for all its assigned tasks. As discussed before, the user is required to provide a task map that assigns a list of task ids to each rank (e.g., as `getIds` in List. 3). Consequently, each rank creates only the portion of the tasks assigned to it. The controller posts the necessary receives and waits for incoming messages or for a direct input of external data. Since each rank handles only the corresponding local tasks, all data needed to start a dataflow is available locally. Each time new information arrives, the controller checks whether all input requirements for some tasks are met. When a task is ready to execute, it spawns a new thread that is executed in the background. Once the computation has finished, the output Payloads are placed in an outgoing messages buffer for the controller to send. To avoid unnecessary de-/serialization and copying of data, the controller checks explicitly for inter-rank messages for which it skips the serialization and instead transfers the memory directly by passing a pointer. However, each task assumes ownership of its input data and relinquishes ownership of its output data to the MPI thread. This avoids any race conditions on the data but may require additional copies of the data to be made. The in-memory messages are particularly useful for persistent data, as the task graph assumes idempotent tasks with no persistent state. Currently, the MPI controller uses the standard C++ thread API to manage a thread pool. Tasks are scheduled greedily, i.e., each task is started as soon as all its input data has been received, in the order in which this data arrived. To better mesh with the host applications use of threads, other interfaces such as OpenMP could be used.

B. Charm++ Controller

The Charm++ runtime controller implements the tasks as *chares* [27]: migratable-objects that represent the basic unit of parallel computation. The tasks in the task graph are mapped to a collection of chares called a *chare* array. The runtime is able to launch a large number of chares simultaneously and periodically balances the load by migrating chares when necessary. Therefore, no explicit task map is needed and the Charm++ load balancer can be tuned to run manually, periodically or in sync for all the tasks using the runtime APIs. The experiments presented in the next section use periodic load balance.

The current implementation creates a single chare array from the main chare that holds all tasks needed throughout the execution of the task graph. Unlike the MPI and Legion implementation, Charm++ does not explicitly instantiate any local or global task graph. Instead, the chare id is translated into a task id at the execution time of a chare, which determines the callback used, and the communication between chares uses remote procedure calls. For instance, the dataflow execution is started asynchronously by chares containing the input data. Once a task has received all the required inputs, the corresponding callback function is executed and the outputs sent to the other tasks identified through the task ids.

In the future, having multiple chare arrays for the different task types may lead to better performance, but since this would require additional Charm++ specific extensions to the task graph, we opted to use a generic implementation. Similar to the MPI controller's notion of in-memory messages, the Charm++ serialization functionality will avoid unnecessary de-/serializations when possible.

C. Legion Controller

Legion is a data-centric programming system that describes the dependency relationships of a program using so-called *logical regions* [28] that contain the meta-information describing a piece of data but not necessarily the data itself. A region associated with a physical copy of its data is referred to as a *physical region*. The Legion controller uses the given de-/serialization routines to map Payloads to physical regions and vice versa. Each task in Legion has a number of *region requirements*, that represent the inputs/outputs data of the task. In particular, data can be passed to the dataflow by simply specifying the corresponding region requirements, and the runtime will provide the data once it is available. Tasks in Legion are spawned using a *launcher* object. There are three kinds of launchers:

- *single task launcher*, which starts a single task;
- *index task launcher*, which starts a set of tasks that share the same set of region requirements;
- *must parallelism launcher*, which handles a set of *single task launchers*.

Index task launchers and must parallelism launchers are two different concepts used to simultaneously spawn a large number of tasks. Furthermore, each task can spawn subtasks recursively. Note that in Legion the costs for preparing and scheduling tasks is borne by its parent task and roughly proportional to the number of subtasks used. As a result, controlling the granularity of the task graph is crucial to maintain an acceptable ratio of useful work to runtime overhead.

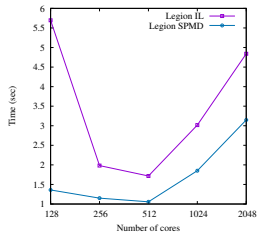


Figure 2: Legion index launches (IL) and SPMD implementations performance comparison executing a parallel merge tree dataflow with an HCCI dataset of size 512x512x512.

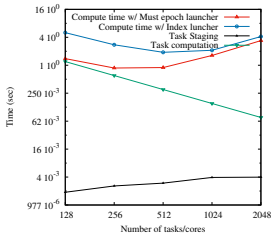


Figure 3: Strong scaling benchmark of Legion index and must epoch launchers. As N tasks are launched on N cores the time per task decreases and the time staging inputs and outputs remains constant at a low level. However, due to runtime overhead in spawning tasks, the overall time increases.

Slaughter et al. [29] suggest that in order to scale an application with a high number of data-parallel tasks, an SPMD (i.e., Single Program Multiple Data) approach is preferable. In order to develop a Legion SPMD runtime controller, we need to separate the computation in a series of independent *shards*. Conceptually, shards are similar to the task map the MPI controller uses to distribute tasks among MPI ranks (i.e., as in the MPI case, the Legion controller makes use of the task map). More specifically, we start one task per shard using a must parallelism launcher to execute a set of independent tasks running in parallel without any runtime synchronization. This significantly reduces the runtime overhead for large task graphs. The per-shard task (i.e., scheduled by the *top level task*) will then schedule its assigned part of the task graph using *single task launchers*.

To manage dependencies between shards, Legion provides synchronization primitives called *phase barriers*. Those are a lightweight producer-consumer synchronization mechanism that allow a set of producer operations to notify a set of consumer operations when data is ready. Note that this semantic is quite different from MPI barriers since there is no global synchronization involved and producers and consumers can be dynamically defined.

As mentioned above, Legion provides a second construct to spawn a large number of tasks: an index launch. To compare both versions, we have developed a second Legion controller using index launchers instead of the SPMD type distribution of tasks. This approach relies more heavily on Legion's ability to distribute and manage large number of tasks (i.e., neither *phase barriers* nor task maps are required). However, Index launches require the task graph to be organized in a set of rounds of similar tasks, all of which can then be processed using a single index launch. The current implementation crawls the graph to group the tasks into *rounds* of noninterfering tasks, i.e., those that do not have dependencies between tasks of the same round. For each round, an *index task launcher* will be executed, mapping the necessary outputs of the previous launch with the inputs of the next.

We performed a series of strong scaling experiments to assess the scalability of both launchers producing an increasing numbers of tasks. Fig. 3 shows execution times for a single launch of a set of data-parallel tasks. The different curves show the compute time for all tasks, the time used to stage task, i.e. prepare input and output regions, set up region requirements, etc., and the total time for the two different implementations. The task computation itself scales almost perfectly for this workload. The staging costs, which represent all operations BabelFlow needs to connect the dataflow to Legion, remains roughly constant, yet the total times shows a significant increase. This is due to the overhead incurred by Legion when spawning a large number of tasks, which in the current version is high compared to the total runtime of our tasks.

Further experiments using a full analysis code have shown that, so far, the index launch approach appears to suffer more (compared to the SPMD) from runtime overheads and demonstrates less scalability. We report some performance results for topological analysis comparing both Legion implementations in Fig. 2. For all remaining experiments discussed in the next section, we used the Legion SPMD implementation. While the index launcher approach at the moment appears less scalable than the SPMD style execution, this may change with future releases of Legion as the runtime evolves. One advantage of our framework is that it is easy to maintain multiple controllers for a given runtime that can be deployed transparently by the algorithm developers. In particular, as the EDSL matures, one might expect certain types of algorithms to prefer particular versions of a controller (i.e., SPMD vs. index launch) or different architectures be more effective than others.

V. Use Cases

The sections above demonstrated the simplicity and flexibility of our approach in both defining task graphs and using them with different runtimes. Here we show that the resulting framework is not only easy to use and portable but can also take advantage of the performance and scalability of the underlying runtimes. To this end, we discuss three

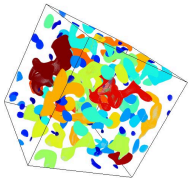


Figure 4: Features extracted from the HCCI dataset using the parallel merge tree algorithm. These features, in the simulation, represent ignition regions.

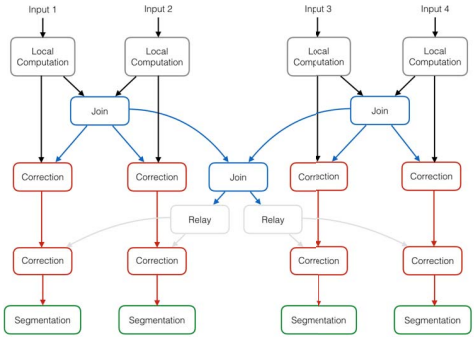


Figure 5: Distributed merge tree dataflow with four input blocks (K=2). The local computation at the leaf nodes of the reduction takes a data block and produces two outputs: a *local tree* and a *boundary tree*. These are sent to the correction and join tasks, respectively. Finally, the correction tasks pass a local tree to a final segmentation tasks.

disparate use cases of large-scale analysis and visualization algorithm: a topological feature extraction with a complex task graph; two versions of a distributed rendering pipeline using a reduction and a binary swap, respectively; and a data-intensive registration of large-scale images. In all cases, we show scaling results for all three runtimes and where possible compare them with existing state-of-the-art implementations. All experiments have been performed on Shaheen II, a Cray XC40 system with 6,174 dual socket compute nodes based on 16 cores Intel Haswell processors with Aries Dragonfly connectivity.

A. Topological Analysis

The first case study is a parallel implementation of segmented merge trees [6] applied to large-scale combustion simulations. As shown in Fig. 5, the task graph of the algorithm is a combination of a global reduction tree and a set of broadcast-like patterns with substantial computation in the reduction as well as at the leaves of the broadcast. The computation is composed of four types of tasks: local computation, join, correction and segmentation. For details and definitions about the algorithm, we refer to the original work by Landge et al. [6]. The local computation at the leaf nodes of the reduction takes a data block generated by the simulation as input and produces two outputs: a *local tree* and a *boundary tree*. These are sent to the correction and join tasks, respectively. All but the leaf tasks of the reduction perform the join of two (or more) boundary trees and send the other boundary tree to the next join and an *augmented boundary tree* to as many correction stages as there are leaves in the subtree of the join. To avoid sending too many messages from a single join task, the dataflow implements its own overlay tree to perform the broadcast (i.e., relay tasks in Fig. 5). The correction uses the augmented boundary tree and the local tree to update the local tree and sends it to the next correction stage. Once all joins have been passed to a correction, each local tree is passed to a final segmentation task. Fig. 5 shows a binary version of this task graph with four leaves. In practice, we typically use 8-way reductions (i.e., k = 8) to reduce the height of the tree.

For testing, we use the output of a large-scale simulation of the autoignition in a Homogeneous-Charge Compression Ignition (HCCI) engine of size $1024 \times 1024 \times 1024$ grid points. The original size of the dataset is $512 \times 512 \times 512$ grid points. In order to perform large-scale experiments at high cores count, we replicate the initial dataset to a larger domain of $1024 \times 1024 \times 1024$ grid points. Since the data is periodic and features are distributed roughly uniformly through the simulation domain (see Fig. 4), the inflated data represents a good proxy for a much larger simulation run. The dataset has been generated with the KAUST Adaptive Reacting Flow Solvers (KARFS) [30] simulator on Shaheen II. Fig. 4 depicts the features extracted by this analysis.

We compare our results to a Lange et al. hand-tuned MPI implementation [6][31], made available to use by the original authors. Fig. 6 shows that the same algorithm executed on Charm++ and MPI runtimes has good scalability and performance. In particular, our generic implementation using the MPI backend outperforms the original implementation, especially at low core counts. The most likely explanation is that the original implementation used blocking communication while our MPI backend uses asynchronous calls and independent threads. Since the computation is naturally load imbalanced (caused by the strong data-dependency of the algorithm), an asynchronous execution is likely more tolerant of delays. This comparison with the original topology implementation is not primarily aimed at comparing run-time overheads but rather at demonstrating that, despite unavoidable overheads, a generic solution can be, sometimes, even faster than typical hand-coded solutions. It is reasonable to consider that a hand-coded version of the algorithm using asynchronous communication, threads, etc. would likely be at least as fast as the generic task. However, developing this code would require significant expertise in MPI whereas our system requires no knowledge of MPI (or any other runtime system). The Legion runtime is comparably fast at low core counts but does not exhibit good scaling. As discussed in Sec. IV-C, the challenge is

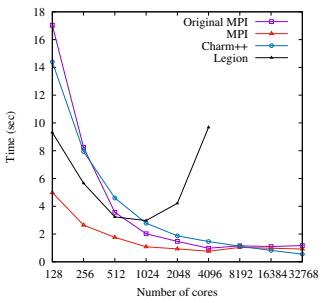


Figure 6: Computation time for the parallel merge tree dataflow using the different runtimes for a HCCI dataset. *Original MPI* is the reference implementation [6].

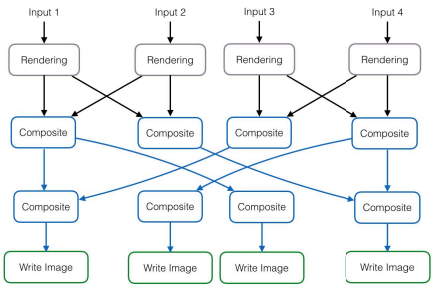


Figure 7: Rendering and binary swap compositing dataflow. The leaves perform the rendering of an input block. The rendered image is split and sent to a pair of composite tasks following the binary swap dataflow. The final tasks perform the final composition and write each tile on the disk.

that at larger core counts many tasks are doing minimal work (or even no work at all), yet the Legion runtime still incurs nontrivial overhead for these tasks.

B. Distributed Rendering

The second use case considers a common two-stage visualization pipeline consisting of a rendering and a compositing stage. We used the same HCCI dataset as input since it represents a challenging test case with complex geometry interspersed with near-empty regions. In the second stage, the local rendering results are composited to either a final image or a set of image tiles, depending on the compositing algorithm used. The first stage is embarrassingly parallel and is implemented using VTK volume rendering (i.e., SmartVolumeMapper with raycasting). Performance scaling results for this stage, equal for all the runtimes, are reported in Fig. 10a. For the compositing stage, we have implemented the two different standard algorithms: reduction and binary swap [32].

As a comparison, we use IceT [33] a high-performance, sort-last parallel rendering library. Note that, to provide a fair comparison with our compositing task, we disabled the interlacing and the background filtering in IceT. These optimization are used to reduce the image sizes that are exchanged and are not available in our implementation. As a result, all tasks will exchange dense images or dense image patches. Using the reduction dataflow, the image compositing follows a simple binary reduction tree where at every stage the input images's z-buffers are reordered and produce the composited image. At the end of the dataflow, a single image is produced (see Fig. 10d). The execution times shown in Fig. 10e report reasonably good scaling for all the runtimes, with MPI showing the lowest increase. Note that, unlike all other results in this section, these results are weak rather than strong scaling, as the number of images to be composited increases with the number of cores.

One of the challenges for the binary reduction is that, by definition, the number of tasks decreases as the algorithm progresses, which severely limits the available parallelism. The traditional solution is the so-called binary swap, where at each stage the tasks pair up and exchange a portion of their current picture. At the end of the dataflow, a number of tasks (i.e., equal to the number of input images to compose) will each own one tile of the final image (see Fig. 7). We see performance increases using MPI and Charm++ runtimes (see Fig. 10f) but a decrease for Legion. One possible explanation is that the number of tasks increases significantly, yet the workload of each task decreases. This may result again in a relatively larger overhead. More in general, the task granularity being too small can cause a large runtime overhead compared to the runtime of the tasks. In particular, the deserialization/serialization of the data structures and the thread management can be avoided in a custom implementation, like IceT, that shows much better timing in the reduction case. However, note that the total execution time for the full dataflow is dominated by the (strongly scaled) rendering tasks, and thus the total time for all runtimes is practically equivalent (see Fig. 10b,10c).

C. Brain Data Registration

The third use case originates in neuroscience and uses BabelFlow to create a dataflow for aligning 3D volumes (see Fig. 8). Each volume of the input data is a stack of 2D images from a laser scan acquisitions of a nonhuman primate brain. Here, we register 25 volumes distributed on a 5x5 grid, each volume containing 1024^3 grid points. The input volumes have an overlapping area of 15%, which is used for evaluating the correct alignment (i.e., offset) of adjacent volumes. The registration algorithm is based on a 3D domain decomposition in *slabs* over the Z axis and a set of $X \times Y$ blocks (i.e., one for each volume). The registration uses a 2D neighbor dataflow (see Fig. 8), where each *slab* exchanges the overlapping sub-volumes data and evaluates the alignment. As this is the first parallel implementation of this particular algorithm, there exists no reference implementation and we report the timing for the three runtimes as is.

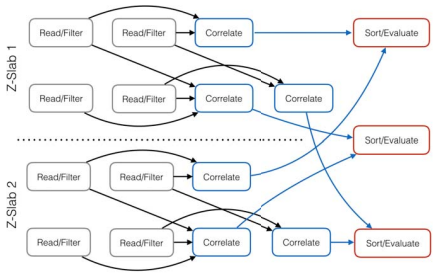


Figure 8: Neighbors registration dataflow for four volumes. For each Z slab, a set of tasks read the blocks that overlaps with the neighbors. These are sent to the correlation tasks to perform the registration. The results are collected by another set of tasks (i.e. sort/evaluate), that will evaluate the final position in space of each volume.

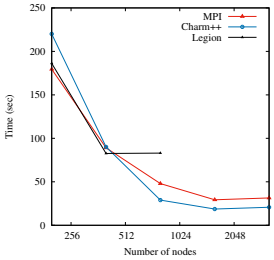


Figure 9: Computation time of brain data registration.

Due to the large data volumes involved, the correlation task is memory limited and we use only 4 of the 32 available cores per node to schedule tasks. Fig. 9 shows the results for up to 3200 nodes. Charm++ and MPI exhibit good scaling with MPI performing better at low and Charm++ better at higher node counts. Legion is on par and even slightly faster for low node counts but levels out as the number of task increases and the workload decreases.

VI. Conclusions

Implementing efficient and scalable analysis and visualization algorithms is challenging and typically relies on solutions specialized for specific software stacks and architectures. As the HPC ecosystem is becoming more diverse, both in terms of runtimes and architectures, hand-tuning solutions in this manner will require substantial integration efforts. Here, we present a framework to address this challenge by combining an EDSL to easily define large-scale algorithms with different runtime controllers to execute algorithms based on different software stacks. We demonstrate that this approach significantly simplifies formulating large-scale parallel algorithms and leads to flexible, portable and scalable implementations in different use cases.

Beyond providing a new approach to design parallel analysis algorithms, our system also represents a flexible test bed to experiment with different strategies to use various runtimes. The MPI and Charm++ controllers both perform well with similar characteristics at scale. The Legion controller works well for low core counts but in its current version does not scale as well as desired. The scalability issues encountered are due to bottlenecks in the Legion runtime that the developers are actively working to address. Note that every runtime system has its quirks and that the goal of our EDSL is not to achieve equivalent performance on all runtimes, but to allow analysis code to be integrated into the runtime system being used by an application, even if it is not a perfect match for the algorithms being used by the analysis code.

Going forward, we believe that the EDSL-driven approach will provide a simple means to develop scalable implementations that are easier to integrate with different host applications. Furthermore, the different task graphs will provide interesting test cases for runtime developers and will allow us to better understand what execution models are most appropriate for analysis and visualization algorithms.

Acknowledgment

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. This work is also supported in part by NSF: CGV: Award:1314896, NSF:IIP Award: 1602127 NSF:ACI:award 1649923, DOE/SciDAC DESC0007446, CCMSC DE-NA0002375, and PIPER: ER26142 DE-SC0010498 and by the Department of Energy under the guidance of Dr. Lucy Nowell and Richard Carson. This research used the resources of the Supercomputing Laboratory at KAUST, Saudi Arabia.

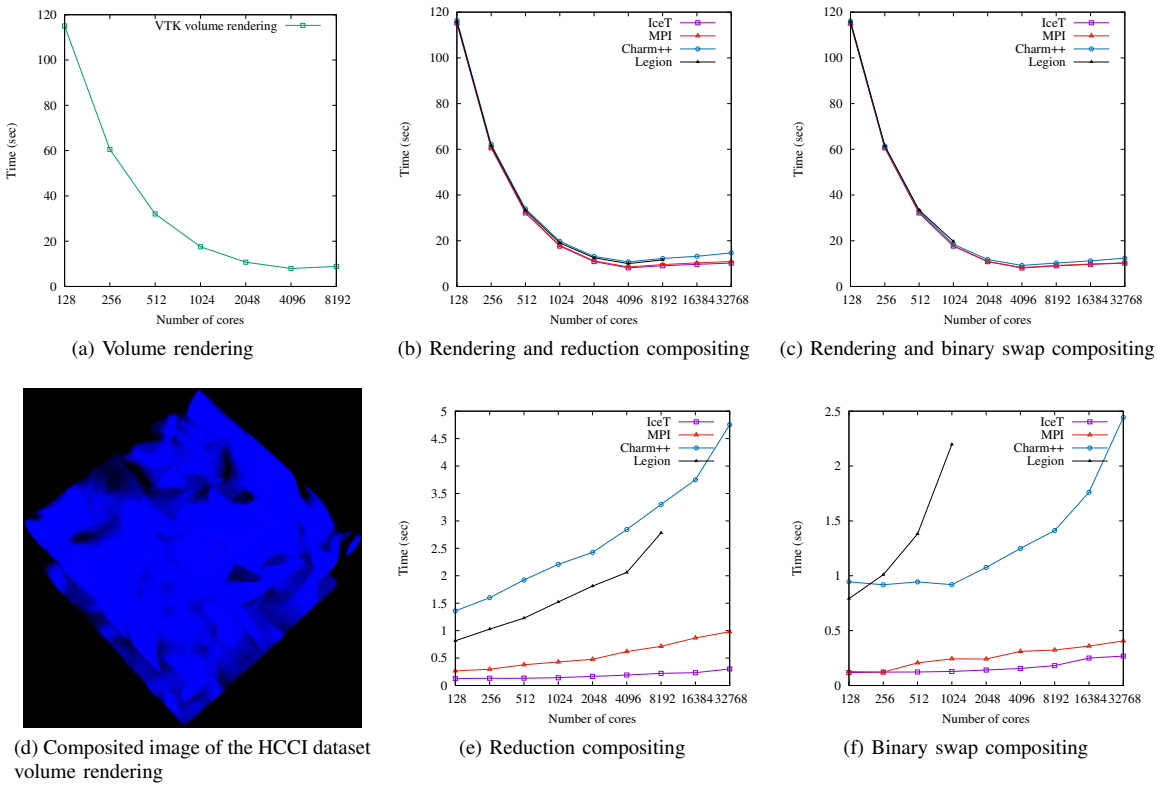
References

[1] J. Ahrens, S. Jourdain, P. O'Leary, J. Patchett, D. H. Rogers, and M. Petersen, "An image-based approach to extreme scale in situ visualization and analysis," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '14. Piscataway, NJ, USA: IEEE Press, 2014, pp. 424–434. [Online]. Available: http://dx.doi.org/10.1109/SC.2014.40

[2] H. Yu, C. Wang, R. W. Grout, J. H. Chen, and K. L. Ma, "In situ visualization for large-scale combustion simulations," *IEEE Computer Graphics and Applications*, vol. 30, no. 3, pp. 45–57, May 2010.

[3] N. Fabian, K. Moreland, D. Thompson, A. C. Bauer, P. Marion, B. Gevecik, M. Rasquin, and K. E. Jansen, "The paraview coprocessing library: A scalable, general purpose in situ visualization library," in *Large Data Analysis and Visualization (LDAV), 2011 IEEE Symposium on*, Oct 2011, pp. 89–96.

[4] M. Dreher and B. Raffin, "A flexible framework for asynchronous in situ and in transit analytics for scientific simulations," in *Cluster, Cloud and Grid Computing (CCGrid), 2014 14th IEEE/ACM International Symposium on*, May 2014, pp. 277–286.



(a) Volume rendering (b) Rendering and reduction compositing (c) Rendering and binary swap compositing

(d) Composited image of the HCCI dataset volume rendering (e) Reduction compositing (f) Binary swap compositing

Figure 10: (a) Volume rendering (i.e., VTK) time for the HCCI dataset. In (b) and (c) we report the execution time for the full dataflow (i.e., volume rendering and compositing), respectively, for a reduction (b) and a binary swap (c) compositing dataflow for an image of size 2048×2048. Although the runtimes seem to behave similarly, they are actually different for the compositing stage (e, f).

[5] J. C. Bennett, H. Abbasi, P. T. Bremer, R. Grout, A. Gyulassy, T. Jin, S. Klasky, H. Kolla, M. Parashar, V. Pascucci, P. Pebay, D. Thompson, H. Yu, F. Zhang, and J. Chen, "Combining in-situ and in-transit processing to enable extreme-scale scientific analysis," in *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*, Nov 2012, pp. 1–9.

[6] A. G. Landge, V. Pascucci, A. Gyulassy, J. C. Bennett, H. Kolla, J. Chen, and P.-T. Bremer, "In-situ feature extraction of large scale combustion simulations using segmented merge trees," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '14. Piscataway, NJ, USA: IEEE Press, 2014, pp. 1020–1031. [Online]. Available: http://dx.doi.org/10.1109/SC.2014.88

[7] B. Whitlock, J. M. Favre, and J. S. Meredith, "Parallel in situ coupling of simulation with a fully featured visualization system," in *Proceedings of the 11th Eurographics Conference on Parallel Graphics and Visualization*, ser. EGPGV '11. Aire-la-Ville, Switzerland, Switzerland: Eurographics Association, 2011, pp. 101–109. [Online]. Available: http://dx.doi.org/10.2312/EGPGV/EGPGV11/101-109

[8] N. Jain, A. Bhatele, J.-S. Yeom, M. F. Adams, F. Miniati, C. Mei, and L. V. Kale, "Charm++ and mpi: Combining the best of both worlds," in *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*. IEEE, 2015, pp. 655–664.

[9] Q. Liu, J. Logan, Y. Tian, H. Abbasi, N. Podhorszki, J. Y. Choi, S. Klasky, R. Tchoua, J. Lofstead, R. Oldfield, M. Parashar, N. Samatova, K. Schwan, A. Shoshani, M. Wolf, K. Wu, and W. Yu, "Hello adios: the challenges and lessons of developing leadership class i/o frameworks," *Concurrency and Computation: Practice and Experience*, vol. 26, no. 7, pp. 1453–1473, 2014. [Online]. Available: http://dx.doi.org/10.1002/cpe.3125

[10] V. Vishwanath, M. Hereld, V. Morozov, and M. E. Papka, "Topology-aware data movement and staging for i/o acceleration on blue gene/p supercomputing systems," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '11. New York, NY, USA: ACM, 2011, pp. 19:1–19:11. [Online]. Available: http://doi.acm.org/10.1145/2063384.2063409

[11] T. Jin, F. Zhang, Q. Sun, H. Bui, M. Parashar, H. Yu, S. Klasky, N. Podhorszki, and H. Abbasi, "Using cross-layer adaptations for dynamic data management in large

scale coupled scientific workflows," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '13. New York, NY, USA: ACM, 2013, pp. 74:1–74:12. [Online]. Available: http://doi.acm.org/10.1145/2503210.2503301

[12] T. Peterka, R. Ross, W. Kendall, A. Gyulassy, V. Pascucci, H.-W. Shen, T.-Y. Lee, and A. Chaudhuri, "Scalable parallel building blocks for custom data analysis," in *Proceedings of Large Data Analysis and Visualization Symposium LDAV'11*, Providence, RI, 2011.

[13] H. Childs, E. Brugger, B. Whitlock, J. Meredith, S. Ahern, D. Pugmire, K. Biagas, M. Miller, C. Harrison, G. Weber, H. Krishnan, T. Fogal, A. Sanderson, C. Garth, W. Bethel, D. Camp, O. Rübel, M. Durant, J. Favre, and P. Navrátil, "VisIt: An End-User Tool For Visualizing and Analyzing Very Large Data," in *High Performance Visualization–Enabling Extreme-Scale Scientific Insight*, Oct 2012, pp. 357–372.

[14] T. Kuhlen, R. Pajarola, and K. Zhou, "Parallel in situ coupling of simulation with a fully featured visualization system," 2011.

[15] N. Fabian, K. Moreland, D. Thompson, A. Bauer, P. Marion, B. Gevecik, M. Rasquin, and K. Jansen, "The paraview coprocessing library: A scalable, general purpose in situ visualization library," in *Proc. of IEEE Symposium on Large Data Analysis and Visualization (LDAV)*, 2011, pp. 89 –96.

[16] U. Ayachit, A. Bauer, B. Geveci, P. O'Leary, K. Moreland, N. Fabian, and J. Mauldin, "Paraview catalyst: Enabling in situ data analysis and visualization," in *Proceedings of the First Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization*, ser. ISAV2015. New York, NY, USA: ACM, 2015, pp. 25–29. [Online]. Available: http://doi.acm.org/10.1145/2828612.2828624

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

[18] LLNL. (2014) Conduit. [Online]. Available: https://software.llnl.gov/conduit/

[19] K. Moreland, C. Sewell, W. Usher, L. t. Lo, J. Meredith, D. Pugmire, J. Kress, H. Schroots, K. L. Ma, H. Childs, M. Larsen, C. M. Chen, R. Maynard, and B. Geveci, "Vtk-m: Accelerating the visualization toolkit for massively threaded architectures," *IEEE Computer Graphics and Applications*, vol. 36, no. 3, pp. 48–58, May 2016.

[20] L.-t. Lo, C. Sewell, and J. P. Ahrens, "Piston: A portable cross-platform framework for data-parallel visualization operators." in *Proc. Eurographics Symp. Parallel Graphics and Visualization*, 2012, pp. 11–20.

[21] J. S. Meredith, S. Ahern, D. Pugmire, and R. Sisneros, "EAVL: The Extreme-scale Analysis and Visualization Library," in *Eurographics Symposium on Parallel Graphics and Visualization*, H. Childs, T. Kuhlen, and F. Marton, Eds. The Eurographics Association, 2012.

[22] K. Moreland, U. Ayachit, B. Geveci, and K. L. Ma, "Dax toolkit: A proposed framework for data analysis and visualization at extreme scale," in *2011 IEEE Symposium on Large Data Analysis and Visualization*, Oct 2011, pp. 97–104.

[23] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with cuda," *Queue*, vol. 6, no. 2, pp. 40–53, Mar. 2008. [Online]. Available: http://doi.acm.org/10.1145/1365490.1365500

[24] L. Dagum and R. Menon, "Openmp: An industry-standard api for shared-memory programming," *IEEE Comput. Sci. Eng.*, vol. 5, no. 1, pp. 46–55, Jan. 1998. [Online]. Available: http://dx.doi.org/10.1109/99.660313

[25] W. Schroeder, K. Martin, and B. Lorensen, *The Visualization Toolkit An Object-Oriented Approach To 3D Graphics, 4th Edition*. Kitware, Inc., 2004.

[26] E. Koutsofios and S. North, "Drawing graphs with dot," AT&T Bell Laboratories, Murray Hill, NJ, Tech. Rep. 910904-59113-08TM, 1991.

[27] L. V. Kale and S. Krishnan, "Charm++: A portable concurrent object oriented system based on c++," *SIGPLAN Not.*, vol. 28, no. 10, pp. 91–108, Oct. 1993. [Online]. Available: http://doi.acm.org/10.1145/167962.165874

[28] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken, "Legion: Expressing locality and independence with logical regions," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '12. Los Alamitos, CA, USA: IEEE Computer Society Press, 2012, pp. 66:1–66:11. [Online]. Available: http://dl.acm.org/citation.cfm?id=2388996.2389086

[29] E. Slaughter, W. Lee, S. Treichler, M. Bauer, and A. Aiken, "Regent: A high-productivity programming language for hpc with logical regions," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '15. New York, NY, USA: ACM, 2015, pp. 81:1–81:12. [Online]. Available: http://doi.acm.org/10.1145/2807591.2807629

[30] B. J. Lee, X. Xiao, F. E. Hernandez Perez, H. G. Im, and R. Sankaran, "KARFS: A combustion DNS solver for hybrid computing architectures," 2012, poster presented at 36th International Symposium on Combustion, Seoul, South Korea, 2016.

[31] I. Rodero, M. Parashar, A. G. Landge, S. Kumar, V. Pascucci, and P.-T. Bremer, "Evaluation of in-situ analysis strategies at scale for power efficiency and scalability," in *Cluster, Cloud and Grid Computing (CCGrid), 2016 16th IEEE/ACM International Symposium on*. IEEE, 2016, pp. 156–164.

[32] K.-L. Ma, J. S. Painter, C. D. Hansen, and M. F. Krogh, "Parallel volume rendering using binary-swap compositing," *IEEE Computer Graphics and Applications*, vol. 14, no. 4, pp. 59–68, 1994.

[33] K. Moreland, W. Kendall, T. Peterka, and J. Huang, "An image compositing solution at scale," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '11. New York, NY, USA: ACM, 2011, pp. 25:1–25:10. [Online]. Available: http://doi.acm.org/10.1145/2063384.2063417