Portable and Composable Flow Graphs for In Situ Analytics

Sergei Shudler*
Lawrence Livermore National Laboratory

Steve Petruzza†
Utah State University

Valerio Pascucci‡
University of Utah

Peer-Timo Bremer§
Lawrence Livermore National Laboratory

Abstract

Existing data analysis and visualization algorithms are used in a wide range of simulations that strive to support an increasing number of runtime systems. The BabelFlow framework has been designed to address this situation by providing users with a simple interface to implement analysis algorithms as dataflow graphs portable across different runtimes. The limitation in BabelFlow, however, is that the graphs are not easily reusable. Plugging them into existing in situ workflows and constructing more complex graphs is difficult. In this paper, we introduce LegoFlow, an extension to BabelFlow that addresses these challenges. Specifically, we integrate LegoFlow into Ascent, a flyweight framework for large scale in situ analytics, and provide a graph composability mechanism. This mechanism is an intuitive approach to link an arbitrary number of graphs together to create more complex patterns, as well as avoid costly reimplementations for minor modifications. Without sacrificing portability, LegoFlow introduces complete flexibility that maximizes the productivity of in situ analytics workflows. Furthermore, we demonstrate a complete LULESH simulation with LegoFlow-based in situ visualization running on top of Charm++. It is a novel approach for in situ analytics, whereby the asynchronous tasking runtime allows routines for computation and analysis to overlap. Finally, we evaluate a number of LegoFlow-based filters and extracts in Ascent, as well as the scaling behavior of a LegoFlow graph for Radix-k based image compositing.

Index Terms: Software and its engineering—Data flow architectures—; Human-centered computing—Visualization—Scientific visualization

1 Introduction

Most high-performance computing systems already made the transition into heterogeneous architectures. This trend is set to increase even more as we approach the limits of the Moore's law. Existing and upcoming diverse hardware architectures lead to a significant challenge of code and performance portability. As a result, developers of simulation codes will typically choose a platform and a family of accelerators to focus on and then spend their efforts to write and tune the code for this chosen set of architectures.

Data and visualization analytics is an integral part of any scientific simulation workflow. Traditionally, the process of analyzing and visualizing the data, with the help of tools such as Paraview [3] and VisIt [7], occurs post-mortem, that is after the simulation has finished running and writing all the data to storage. In such case, analysis routines often run on dedicated systems with fairly standard software stacks. Even if simulation codes run on different architectures and on different runtimes, the post-mortem setting allows analysis code to be optimized for just one architecture.

Post-mortem analysis, however, has a major drawback that stems from the imbalance between compute capacity and I/O bandwidth. An imbalance that is exacerbated by the emergence of extreme scale HPC systems. Performing either partial or full analysis of the simulation data in situ, that is during the computation phase, has the potential to mitigate this problem. In recent years, in situ analysis has been gaining increased focus [10, 11, 14, 36] and was shown to be effective in providing insight into simulations whilst drastically reducing the amount of data that needs to be stored.

With in situ processing, analysis routines become an integral part of the simulation code stack. Unlike simulation code, however, analysis routines are general enough to be applicable to a wide range of domains and problems. If different simulation codes run on different architectures or runtimes, we are faced with the problem of constantly tailoring analysis code to each specific hardware. This incurs significant costs over time as we need to maintain an increasing number of different implementations of the same algorithms. Introducing new features or fixing bugs becomes an ordeal since it has to be replicated for each target runtime and hardware. As a result, most existing tools go down the simulation code route and focus on a single runtime (e.g., MPI). Even libraries designed to simplify the development of new analysis also target eventually a specific runtime. One example is the DIY and DIY2 libraries [27, 30], both of which are based on MPI.

BabelFlow [31] was designed to address these challenges by providing developers with a simple dataflow-based interface to implement parallel algorithms. Essentially, BabelFlow describes task graphs that explicitly identify parallel execution sections of the algorithm and the relations between them. The biggest strength is that it isolates the description of the algorithm from the implementation. In this way, BabelFlow abstracts all aspects of communication from the users, thereby enabling them to focus fully on algorithm design. Most importantly, however, it transparently maps the dataflow graph onto various runtimes to provide a native execution of the corresponding algorithm.

One of the main drawbacks of BabelFlow is that it lacks integration with existing in situ analytics platforms. Another one is that there is no ability to easily combine existing dataflows together. The second aspect is strongly related to the first since in situ analytics requires flexible analysis routines that can handle changing data between iterations. In BabelFlow, users are required to reimplement common patterns for even minor additions, such as attaching an additional file I/O step to the end of an algorithm or adding a preprocessing step before an existing dataflow. Although it might be possible to mitigate some of these challenges through the use of more complex software constructs such as templates or hierarchical design patterns, BabelFlow provides no such tools and their use would also run counter to the desired simplicity for non-experts.

In this work, we introduce LegoFlow, an extension to BabelFlow that integrates with Ascent [22], a flyweight in situ analysis framework, and provides a simple and intuitive interface to compose an arbitrary number of dataflow graphs to create more complex patterns. LegoFlow enables developers to implement a library of

*e-mail: shudler1@llnl.gov
†e-mail: steve.petruzza@usu.edu
‡e-mail: pascucci@sci.utah.edu
§e-mail: bremer5@llnl.gov

communication patterns ranging from simple reductions to complex algorithms, such as computing a merge tree or compositing an image, and then chain these building blocks into new dataflows (and new algorithms) for more specialized situations. Just like BabelFlow, LegoFlow hides all communication from the user whilst maintaining a simple idempotent task model and provides a native implementation of the dataflow in any of the backend runtimes including MPI, Charm++ [19], and Legion [13].

Composability of dataflow graphs is essential in in situ analytics. This allows LegoFlow to be used as a complete communication layer underneath Ascent so that most of the existing filters and extracts can run on non-MPI runtimes with minimal adaptation. It also allows users to adapt dataflows to create multiple variants of a a dataflow for different conditions during the simulation.

Specifically, our contributions are as follows:

- LegoFlow infrastructure that includes mechanisms to compose dataflow graphs and a library of common patterns (e.g., Radix-k, reduction, gather).
- Integration of LegoFlow with Ascent, including providing a number of filters and extracts based on LegoFlow-based.
- Demonstration of the LULESH code execution on top of Charm++ with full LegoFlow-based Ascent pipeline that computes and renders iso-surfaces.

The rest of the paper is organized as follows. We start by reviewing related work in Section 2 and then continue with a brief background of BabelFlow in Section 3. Section 4 describes the design of LegoFlow and goes into greater detail on its methodology. Next, Section 5 explains the LegoFlow-based filters and extracts in Ascent, followed by Section 6 that evaluates various use cases. Finally, we present conclusions and future work in Section 7.

2 Related Work

Dataflow graphs

The concept of dataflow graphs has been extensively researched [18]. Perhaps the earliest use of the dataflow abstraction in the context of scientific data analysis and visualization is the SCIRun system [28]. This system uses dataflow graphs to break analysis and visualization algorithms into a network of separate modules such as mesh reading, Delaunay triangulation, iso surface computation, and so on. Furthermore, users are also provided with a tasking library to offload the computation onto multiple threads on a single node. A later study [33] introduces ready-to-use templates of dataflow graphs that represent common parallel execution patterns. The author acknowledge the gap between the power of dataflow abstractions and ease of programming and aim to mitigate it.

Graph composability

One prominent example for explicit API for graph composability is Intel Threading Building Blocks (TBB) [35]. This library provides users with an interface to create task graphs that are executed on a multicore processor. Similar to other tasking abstractions, the user identifies independent units of computation and the dependencies between them. One of the nodes in a TBB graph could be another TBB graph. This allows users to express nested parallelism in their algorithms and reuse existing graph patterns. Another approach, called Cpp-Taskflow, is similar to TBB and provides users with a library to express task based parallelism on multicore architectures in the form of task graphs [17]. The library leverages modern C++ to provides users with an API for more efficient parallel decomposition. A later development [24] extended Cpp-Taskflow to also allow quick reuse and composition of existing task graphs. Unlike Cpp-Taskflow, however, LegoFlow allows for more complex interconnects between tasks using the graph connector interface explained in the next sections.

In situ analytics.

A number of studies focus on different aspects of in situ (and in transit) analytics. GoldRush [36] targets MPI/OpenMP-based simulations and utilizes periods when OpenMP threads are idle to run in situ algorithms. Damaris/Viz [15] is a framework for in situ visualization that uses the Damaris I/O middleware. By running on dedicated cores it allows simulation to overlap with in situ processing. Bauer et al. [11] survey methods, infrastructures, and a range of applications that use in-situ techniques for analysis and visualization. Ayachit et al. [10] examined the scalability, overhead, and performance aspects related to in situ analysis of mini applications using SENSEI, a framework that is able to channel simulation data into a myriad of analysis or visualization tools such as ParaView and VisIt. These efforts are mainly based on MPI and, in some cases, on OpenMP. However, more recent studies examine in situ analytics in task-based execution environments. Tasking abstraction is more general than threading as it decouples a logical execution package (i.e., a task) from specific implementation concepts such as process or thread. A tasking system, therefore, can be implemented using any combination of processes and threads. Heirich et al. [16] demonstrate that both the simulation and in situ visualization can run on top of Legion. TINS [14] is a framework for in situ analytics that runs on top of TBB and can use dedicated cores for execution, thereby isolating analysis from the simulation. LegoFlow takes a step further by offering a framework of composable parallel patterns for versatile in situ analytics that is not limited to just one non-MPI runtime or just one visualization workflow.

3 Dataflow Graphs in BabelFlow

In this section, we present an overview of BabelFlow [31]. This will provide us with the necessary context to discuss LegoFlow later on.

3.1 Overview

In BabelFlow, the algorithm is described as a task graph in which individual tasks are nodes with inputs and outputs. The inputs are an array of task identifiers that a task receives data buffers from. The outputs, on the other hand, allow for multiple data per destination. Therefore, the outputs are an array of an array of task identifiers. In other words, there is an array of task identifiers per each output buffer. The flow of the data is captured by the edges between the tasks. Parallel execution of the task graph is carried out by a runtime controller that corresponds to the chosen runtime system. Both task graphs and controllers use a C++-based API. Typically, the user will implement a new task graph by deriving from the abstract `TaskGraph` class that provides an interface for describing tasks and other properties of the graph. The existing controllers for MPI, Charm++, and Legion are aware just of this abstract class so they automatically will support any new user defined task graphs.

The interface is designed so that task graphs are specified in a procedural way. This means that individual tasks are only produced by demand and are not preconstructed. Furthermore, users provide implementations for routines that serialize / deserialize task graphs and the objects exchanged between the tasks. Having procedural graphs means serialization is cheap and allows moving the graph efficiently between local and remote processing nodes. BabelFlow, therefore, supports both shared memory and distributed memory runtimes.

3.2 Runtime Controllers

Runtime controllers orchestrate the execution of the dataflow graph on a specific system runtime. Controllers differ in the way they keep track of dependencies, schedule tasks, and distribute tasks between the available computing resources. An individual computing resource is called a *shard*; for example, in MPI case, a shard is an MPI process. Some controllers use an optional class that defines the mapping between tasks and shards. It is called a *task map*. This

mapping is completely orthogonal to the graph definition so that the graph is completely independent from the controller.

The MPI controller statically allocates tasks to ranks according to the provided task map. This enables flexibility and debugging options, e.g., a special task map can map all tasks to one process. Once the local portion of the dataflow graph is instantiated in a process the execution proceeds independently. To communicate data between tasks, asynchronous point-to-point messages are used. The controller first posts receive calls and waits for incoming messages. Once a message for a task arrives it checks whether all the inputs for that task are present and if they are the task is ready for execution. In such case, the controller executes that task in a separate thread. For this purpose, C++ threading API with a thread pool is used. Once the task finishes executing the resulting buffer or buffers can be sent to neighboring (target) tasks connected with outgoing edges from the current (source) task. In case the target task is assigned to the same rank, the controller just assigns pointers instead of performing a redundant copy of the buffer.

The Charm++ parallel programming model is based on C++ and organizes the execution into migratable objects called *chares*. These objects then communicate between each other asynchronously [19]. In other words, chares are units of parallel computation that can be preempted and migrated between worker threads distributed across different nodes. The runtime controller assigns the tasks in the graph to chares; by default, each task is assigned to a single, unique chare. Charm++ launches all the chares simultaneously and manages the load on each thread by migrating chares between the threads. In this case, no task map is needed since the mapping is trivial—one task per chare. The serialized task graph is passed as a parameter to a chare constructor, which then deserializes it and requests the task that corresponds to the current chare. Once a task finishes executing, its output is sent to an outgoing neighbor by means of an entry function call in the corresponding chare. This is the mechanism in Charm++ to send messages between chares and it is implemented as remote procedure calls (RPC).

The Legion controller uses the Legion runtime [13]. It is a data-centric programming model that defines data dependencies in a program using so called *logical regions*. Those regions represent the input and output data of each Legion task, and for each *logical region* a corresponding *physical region* will be instantiated and used at runtime to de-/serialization the data. The controller decomposes the input task graph into rounds of independent tasks (i.e., tasks that do not have data dependencies in the same round) and launches them using an *index task launcher*. This is currently the recommended method for Legion applications to spawn a large number of tasks efficiently. Rounds are executed asynchronously and the runtime is fully responsible for the distribution and execution of the tasks spawned at each round. It also handles the data movement according to the defined data dependencies.

4 Composable Dataflow Graphs

We start this section by presenting the design of LegoFlow and then continue with a discussion about a number of challenges we faced. The goal was to fit LegoFlow into the existing structure of BabelFlow without sacrificing its strengths.

4.1 LegoFlow Design

Figure 1 is a schematic diagram of LegoFlow. It is a middle layer between runtime systems and the application / Ascent filters. A LegoFlow graph specifies the flow of data between nodes, where each node (i.e., task) processes the data independently. The implementation for each task is provided by the user of LegoFlow, be it the application or an Ascent filter. We call these implementations *callbacks* as they are essentially functions that are called by the controller once the task is ready to execute. The graph is an explicit representation of a coarse-grained parallelism, such that

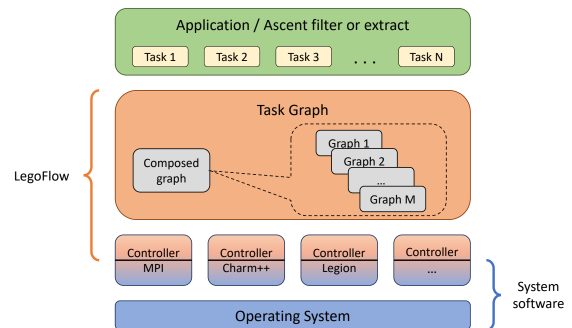


Figure 1: Overview of LegoFlow.

the implementation of each task can use fine-grained parallelism by targeting various CPU architectures or GPU accelerators. In Section 5, we demonstrate an example in which a LegoFlow task uses a portable GPU-accelerated library. It is important to note, however, that the task implementation must preserve portability and not use runtime-specific code.

Some dataflows, such as broadcast and k-way reduction, are already provided by BabelFlow. But as described earlier, there is no easy and flexible way to combine existing graphs and communication patterns into a new graph. Going through this process would involve two steps with compounding complexity. First, code for any common components would have to be copied between combined dataflows with all the downsides regarding code maintenance, debugging, and performance optimization this entails. Second, the key complexity of combining dataflows is managing the task index spaces used to route messages. Many individual dataflows in BabelFlow use pre- and post-fix index operations to simplify creating phases or rounds of communication. Harmonizing the index management across dataflows, therefore, is non-trivial, labor intensive, and error prone.

LegoFlow extends BabelFlow by introducing three new concepts. First, task identifiers that encapsulate individual task ID's and graph ID's and that behave like a single hashable number. Second, a graph connector abstraction that supports arbitrary links between composed graphs and a default implementation for a connector that covers most common use cases. Finally, LegoFlow introduces a composable graph abstraction that encapsulates connected graphs and is itself a task graph. Below we discuss these concepts in more detail.

4.2 Task Identifiers

The tasks in a BabelFlow graph are identified by a local ID and a global ID. The former is used to identify tasks within the same local *shard* (e.g., MPI rank) and the latter identifies tasks globally, across all the *shards*. Both ID's are integers. In LegoFlow, composable graphs need to take into account also the subgraph each task belongs to. For this purpose, we modified the original `TaskId` class to represent a pair of ID's: the subgraph index and the local ID of the task in that graph. By implementing overloaded arithmetic and conditional operators the new task ID can be plugged into existing code as is without substantial code modifications. Furthermore, we implemented a hash function for the `TaskId` class so that we can keep using hash maps in both runtime controllers and graph connectors.

4.3 Graph Connectors

We defined the abstract `TaskGraphConnector` class that specifies how tasks are connected in a composable graph. Essentially, the

```
class ComposableTaskGraph : public TaskGraph
{
public:
  // Uses a default graph connector
  ComposableTaskGraph(std::vector<TaskGraph*>&);

  // Uses the given graph connector
  ComposableTaskGraph(
    std::vector<TaskGraph*>&,
    const TaskGraphConnectorPtr&);

  // ...
private:
  std::vector<TaskGraph*>      m_graphs;
  TaskGraphConnectorPtr        m_connector;
};
```

Listing 1: Composable graph.

class declares three methods. One returns the outgoing connections from a source task to target tasks and the second one—incoming connections into a target task. These two methods reflect the distinction between incoming and outgoing connections in a task. The third method connects the source and target tasks together.

As a default implementation for a task graph connector, we defined the `MultiGraphConnector` class. This connector links graphs according to given pairs of subgraph indices. Specifically, given an array $[g_0,\ g_1,\ \dots,\ g_N]$ of graphs and an array of possible pairs of graph indices $[(i,\ j)\dots]$ $(i < j)$, it connects the graphs according to these indices: $g_i \rightarrow g_j$. If users provide no index pairs, the default behavior is to link consecutive graphs together, i.e., $g_i \rightarrow g_{i+1}$ for $i = 0,\ 1,\ \dots,\ N-1$. This default connector is used for all the use cases in the paper and together with the general functionality in `MultiGraphConnector` would be sufficient for most cases we envision. Should users require more specialized behavior they have the flexibility of developing a new connector for their needs.

The `MultiGraphConnector` class links a pair of graphs together by linking a root (i.e., sink) task in the origin graph to a leaf task in the target graph. A root task is a task without output edges and a leaf task is task without inputs edges. The implementation requests an enumeration of root and leaf ID's from the two graphs and uses a round-robin approach to map a root task to a corresponding leaf task (modulo the total number of leaves). Specifically, the leaf task ID is appended to the array of outgoing task ID's in the root, and vice versa, the ID of the root is appended to the array of incoming task ID's in the leaf.

4.4 Composable Graph

Listing 1 shows a code excerpt that defines the `ComposableTaskGraph` class that represents a composable task graph. It encapsulates an array of subgraphs and contains a pointer to an instance of a graph connector that specifies how these subgraphs are linked together. If no connector is provided in the constructor a default `MultiGraphConnector` is created. This small feature makes it very easy for users to use LegoFlow. They need only to create an array of graphs to be composed and pass that array to the constructor. `ComposableTaskGraph` inherits from the `TaskGraph` class and so behaves exactly like a task graph from the perspective of a runtime controller.

We also implemented the `ComposableTaskMap` class that specifies a default task map that can be used for composable graphs. It works by aggregating the task maps that correspond to each subgraph. In other words, whenever it needs to map a task to a shard it first extracts the subgraph ID from the task ID and then uses that ID to access the subgraph's task map and resolve the mapping.

5 Integration With Ascent

A central contribution of this work is introducing portability and composability of dataflows into in situ analytics. For this purpose, we integrated LegoFlow with Ascent [1] and exposed a number of dataflows as filters and extracts. In the Ascent pipeline the data flows through various filters until it reaches potential extracts and scene operations. A filter transforms the data or augments it in some way, whereas an extract is designed to extract certain parts of the data without passing it further down the pipeline. The extracted data is typically rendered or written to the disk in some form (e.g., an HDF5 dataset). A scene operation visualizes the data using Ascent's builtin renderer.

5.1 Compositing Extract

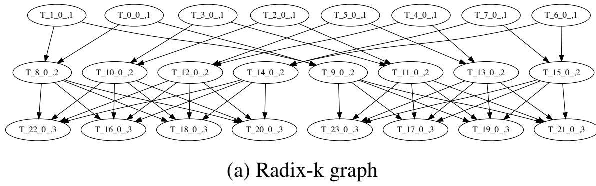
We implemented the Radix-k compositing algorithm [29] using LegoFlow. This algorithm is based on an earlier work that introduced the binary swap algorithm for parallel compositing of locally rendered images [25]. In binary swap, pairs of increasingly distant neighbors exchange decreasing portions of image data until the greatest distance is reached. Radix-k is a generalization that reduces the number of iterations by having dense communication groups within processes. In our implementation, each level has the same number of tasks that are interconnected across adjacent levels according to the corresponding radix. The number of tasks, as well as the array of radices, are provided by the user. This dataflow is essentially the first phase of compositing. The second phase is a gather operation on the pieces of data in each of the root tasks of the Radix-k graph. We reuse the existing k-way reduction graph in BabelFlow and connect it to the Radix-k graph using the LegoFlow mechanism for composing graphs. The reduction pattern can be used both for reduction and gather operations; to switch between the two requires adjusting the task callbacks and the data serialization.

Figure 2 demonstrates an example of the two graphs we connected together into a full dataflow. The first one is a Radix-k graph with 8 tasks per level and a radix array of [2, 4]; whereas, the second one is a 2-way reduction graph. Figure 3 demonstrates the fully composed dataflow graph. The string inside each task has the format: `T_<task ID>_<subgraph ID>, <callback ID>`. Note that for each single graph the subgraph ID is 0, but when these subgraphs are composed the 2-way reduction graph, which follows the first subgraph, gets an ID of 1. The callback ID's are local to each subgraph and describe the function of each task. In some cases, multiple levels of a graph perform the same function and hence have the same callback ID such as callback 2 in Figure 2b. This callback performs the reduction. Callback 1 in this case is just a relay and callback 3 is used to write the resulting image to the disk. As Figure 3 demonstrates, callback ID's can clash with similar ID's in other subgraphs. To avoid the clash, callback ID's are keyed together with the subgraph ID's.

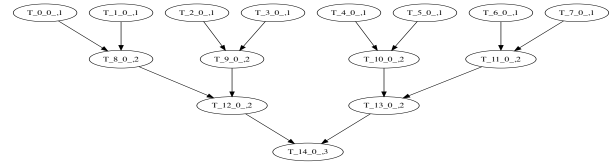
```
RadixKExchange radixk_gr(...);
KWayReduction gather_gr(...);

RadixKExchangeTaskMap radixk_tm(...);
KWayReductionTaskMap gather_tm(...);

std::vector<TaskMap*> tm_vec{
  &radixk_tm, &gather_tm};
std::vector<TaskGraph*> gr_vec{
  &radixk_gr, &gather_gr};

ComposableTaskGraph radixk_compositing(gr_vec);
ComposableTaskMap radixk_compositing_tm(tm_vec);
```

Listing 2: Combining two dataflows in LegoFlow.

Listing 2 is a short code excerpt that demonstrates the ease with which a combined Radix-k compositing dataflow can be produced. The `RadixKExchange` object defines a Radix-k pattern



(a) Radix-k graph



(b) 2-way reduction graph

Figure 2: Two task graphs that constitute the full Radix-k dataflow graph. The strings inside each task have the format: T_<task ID>_<subgraph ID>, <callback ID>.

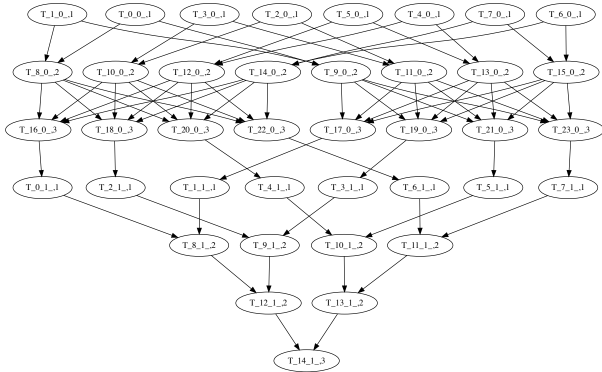


Figure 3: Composed Radix-k and 2-way reduction graph. The strings inside each task have the same format as in Figure 2.

(similar to Figure 2a) and KWayReduction object defines a reduction graph (similar to Figure 2b). The combined pattern is created by just passing an array of subgraphs to the constructor of ComposableTaskGraph. Note that the parameters for each subgraph are passed independently via each subgraph constructor. For example, the Radix-k subgraph will receive the radices array and the reduction subgraph—the number of tasks at the first level and the reduction fan-in (e.g, 2-way, 4-way, etc.). Task maps are created in the same way, that is by combining task maps of the subgraphs into one task map. In total, less than ten lines of code are needed. Compared to the original situation in BabelFlow, we did not need to implement new classes from scratch (e.g., a new class for Radix-k compositing) and did not have to revert to error-prone code rewrite. We also did not have to deal with task ID and indexing issues, which are handled by the LegoFlow insfrastructure. Most importantly, we can easily modify or augment the functionality of the combined compositing dataflow by either using a different reduction pattern or adding some filtering layer. In such case, we just need to change or add another subgraph to the array passed to the ComposableTaskGraph constructor.

The composed dataflow was encapsulated into an Ascent extract. Filters and extracts in Ascent are designed in such a way that they receive their input data as a Conduit Blueprint node [2]. Conduit's goal is to provide an independent data model that allows various applications to share their data with Ascent. Without a common data model, in situ processing cannot be generalized. Specifically, Conduit is a model for describing hierarchical scientific data that is passed between packages either in core or in serialized form. This representation supports structured and unstructured meshes, as well as multi-component arrays.

The extract can be configured through a number of parameters that are passed as part of the Conduit node. We can specify the radices array, the color and depth fields of the image data (that are part of Conduit data fields), and the k parameter in the k-way reduction subgraph.

5.2 Parallel Merge Tree Filter

The Parallel Merge Tree (PMT) algorithm allows users to extract topological features in a dataset [21]. It was implemented in BabelFlow as a dataflow graph in a previous work [31]. The byproduct of the merge tree computation is a segmentation of the data. Essentially, the algorithm assigns a segmentation value to each point in the dataset and the last layer of tasks stores these values to the disk.

We used LegoFlow to add a layer of pre-processing tasks that strip ghost cells from the data before it starts being processed by the PMT tasks that follow in the dataflow. We then encapsulated the full PMT dataflow into an Ascent filter. Compared to an extract, a filter in this case allows us add the segmentation values as a new field in the Conduit data. This data is passed further down the Ascent pipeline, thereby allowing additional Ascent filters to process it and eventually visualize it. The PMT filter can be configured through a number of parameters. We can specify the name of the field from which the data for PMT processing is taken, the features threshold, a reduction fanin, and a flag that specifies whether to create the segmentation field at all.

5.3 Iso-surfaces Extract

In this example, we connected five separate dataflow graphs together to create a graph that computes and renders iso-surfaces of a dataset. Figure 4 depicts a schematic diagram of this composed graph. Two of the subgraphs are the same Radix-k dataflows but with different task functions. Another two subgraphs use VTK-m [26], a scientific visualization toolkit inspired by VTK and that aims to exploit parallelism provided by emerging shared-memory architectures.

Previous work [27] used the Radix-k communication pattern to perform an *allreduce* operation. We take the same approach and by adjusting the callback functions in the Radix-k dataflow we get a subgraph that performs an allreduce operation. This is the topmost subgraph in Figure 4 and it computes the global bounds from the bounds of the local datasets, as well as the global minimum and maximum of the dataset values. The bounds are going to be used in a later stage for the ray tracing rendering, whereas the dataset range is used to automatically calculate iso values. This is needed in case users did not specify explicit iso values but just the number of desired iso levels.

The second subgraph computes iso-surfaces using the marching cubes algorithm in VTK-m. It means that the computation can be accelerated on any available GPUs or potentially make use of additional CPU cores on the node. This is a clear example of how the design of LegoFlow enables us to use multi-level parallelism. In other words, a task can seamlessly use CUDA, OpenMP, TBB or any other fine-grained parallelism library that targets either GPUs or many-core CPUs. Note that the subgraph has two inputs, namely the global bounds (with the data range) and the local dataset, which is passed directly from the controller. The output is then the iso-surface data from the marching cubes algorithm and the global bounds (with the data values range).

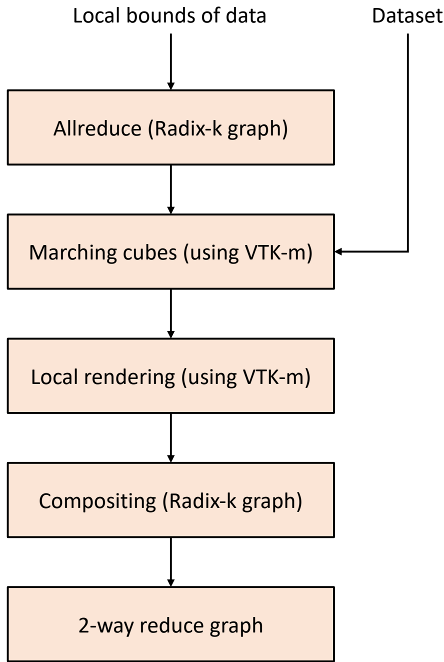


Figure 4: A dataflow graph composed of five separate subgraphs and designed to compute and render iso-surfaces of a dataset.

The third subgraph uses VTK-m to run a ray tracing-based rendering of the local iso-surface data. At this stage the global bounds are used to compute the default position and the direction of the camera. The camera parameters need to be identical between tasks so that we are able to perform image compositing correctly. Finally, the fourth and fifth subgraphs form the exact same compositing pattern used in the first example in 5.1. In the end, the output image is written to the disk.

Creating this dataflow involves not much more code than is presented in Listing 2. Without LegoFlow, this process would have been difficult, tedious, and most importantly, error-prone. Same code (e.g., Radix-k pattern) would have to be copied over, task indices would have to be tailored for this particular case, and future adjustment would cause code changes across all of the implementation. We encapsulated the dataflow as an extract in Ascent with parameters for either the iso values or the number of iso levels, radices array, and the output image size and name. It is also easy to add parameters to adjust the camera configuration.

6 Evaluation of Case Studies

This section focuses on evaluating the contributions explained earlier. Specifically, we evaluate the integration of LegoFlow with Ascent and the filters / extracts that are part of this integration. We then highlight the use case in which a complete application with an Ascent in situ stack executes on top of a non-MPI runtime.

The use cases are as follows:

1. LegoFlow-based compositing extract in Ascent as presented in Section 5.1.
2. LegoFlow-based segmentation filter as described in Section 5.2.
3. Computation and visualization of iso-surfaces in situ (using the extract in Section 5.3) in a LULESH code running entirely on Charm++ without any reliance on MPI.
4. Scaling of the LegoFlow-based Radix-k dataflow running on non-MPI runtimes, i.e, Charm++ and Legion.

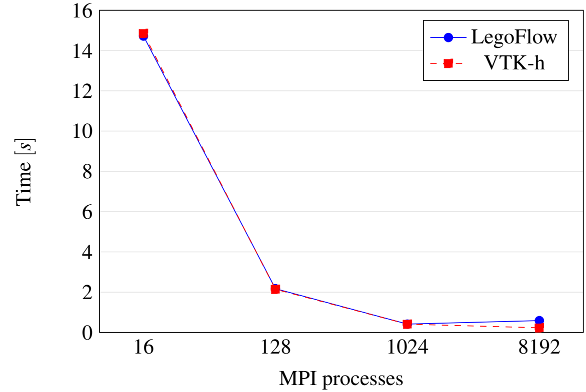


Figure 5: Execution times of Ascent analysis in Laghos. The blue line represents the runs with LegoFlow Radix-k compositing and the red line are runtimes with native Ascent compositing. In both cases, rendering is performed by Devil Ray.

We ran most of the experiments on Summit, a 200 PF heterogeneous system at Oak Ridge National Laboratory [4]. At the moment, this is one of the major pre-exascale machines in the Top500 list. Each node comprises two POWER9 processors with 22 cores each and six NVIDIA Volta V100 GPUs. The system is interconnected by Mellanox EDR InfiniBand network. The vast majority of the floating point compute capability on Summit comes from the GPUs [34] and is aimed to be used by scientific simulation codes. In situ analytics can use either CPUs or GPUs depending on the particular problem at hand. LegoFlow does not place any restrictions in this regard, namely, a callback function in a task can run either CPU or GPU code.

6.1 LegoFlow-based Compositing Extract

In this case study, we look at in situ analysis of Laghos (LAGrangian High-Order Solver) [6], which is a miniapp that solves the time-dependent Euler equations of compressible gas dynamics in a moving Lagrangian frame. It uses high-order 2D and 3D meshes and explicit high-order time-stepping. Laghos captures the basic structure of many other compressible shock hydrocodes such as BLAST [9] and is a proxy-application for the next-generation multi-physics code MARBL [32]. The reason for focusing on Laghos is that higher-order data visualization—particularly one that uses dataflow graphs for volume rendering—is an interesting challenge we aim to tackle. For now, however, we focus on non-volume rendering and parallel compositing with LegoFlow.

The Ascent pipeline consists of a Devil Ray-based [5] filter followed by the LegoFlow compositing extract. Devil Ray is a portable ray-tracing library for visualizing high-order meshes and it performs local, per-rank rendering of the high-order mesh data. It is available in Ascent both as a filter and an extract. For our evaluation we use the Devil Ray filter so that the resulting images can be passed to the compositing extract. The Devil Ray extract is used for comparison since it performs the compositing using VTK-h [8], which is a thin layer on top of VTK-m [26] that adds communication capabilities through MPI and DIY2 [27]. VTK-h is used in Ascent by default for all the renderers.

Laghos offers a choice of scenarios to run. Since our interest lies in visualizing the data, we selected the Taylor-Green vortex scenario and a 3D cube mesh. The ratio between MPI ranks along each axis was set to 2:1:1. This led to selecting MPI process counts to be 16, 128, 1024, and 8192. Using 32 ranks per node, we ran the

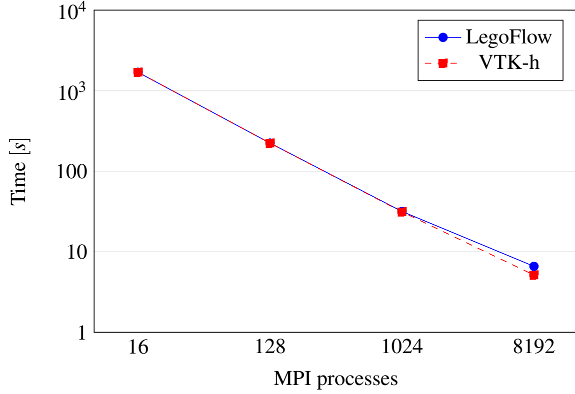


Figure 6: Execution times of Laghos. The blue line represents the runs with LegoFlow Radix-k compositing and the red line are runtimes with native Ascent compositing. In both cases, rendering is performed by Devil Ray.

simulation on 1, 4, 32, and 256 nodes, respectively. Based on the process counts, the radices arrays that we used were: [4, 4], [4, 4, 8], [8, 8, 16], and [8, 8, 8, 16], respectively.

Figure 5 presents the execution times of the Ascent analysis pipeline in the simulation. For lower rank counts, local rendering dominates the runtime and the compositing part remains a much smaller component. The results for LegoFlow-based compositing almost overlap the default VTK-h compositing. It is important to note that Radix-k performance depends on the choice of radices. The longer the array of radices the more levels the graph has, which means more work per process. On the other hand, a shorter array of radices means that some of the radices will be higher, which means more dense communication groups within the graph. We did not tune the radices array and this might explain the slight difference for 8192 processes.

Figure 6 presents the execution time of the complete Laghos simulation run, including computation and analysis. The time scale is logarithmic and, as expected, the analysis part in Ascent is only a small percentage of the total runtime. The results for LegoFlow compositing and the default compositing almost overlap.

6.2 LegoFlow-based Segmentation Filter

This use case evaluates the in situ execution of the PMT filter described in Seciont 5.2. The dataset we used is based on the output of a large-scale simulation of autoignition in a Homogeneous-Charge Compression Ignition (HCCI) engine and has been produced with the KAUST Adaptive Reacting Flow Solvers (KARFS) [23]. The original output was a volume of $512 \times 512 \times 512$ grid points. In order to perform experiments on larger core counts it was replicated to a larger $1024 \times 1024 \times 1024$ grid. The data is periodic and since features are distributed roughly uniformly across the simulation domain the inflated data represents a good proxy for significantly larger simulation runs.

The pipeline consists of the PMT filter followed by Ascent's scene action that volume-renders the segmentation field. Figure 7 shows the rendered output. The transfer function and the color palette are configurable through the parameters passed to the scene action, and we modified them to filter the surrounding feature-less segment (i.e., lower part of the merge tree). Figure 8 presents the execution times of the pipeline for increasing MPI rank counts. In each case we used 32 MPI ranks per node and the times reflect both the execution of the PMT algorithm as well volume rendering. The black dashed line represents perfect scaling and the results show good strong scalability with some slowdown at the highest core count. A number

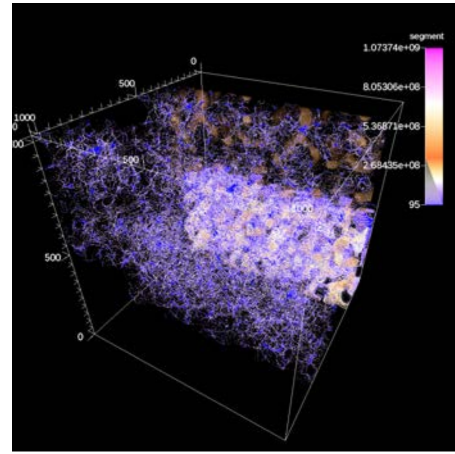


Figure 7: Volume rendered segmentation field produced by the PMT filter.

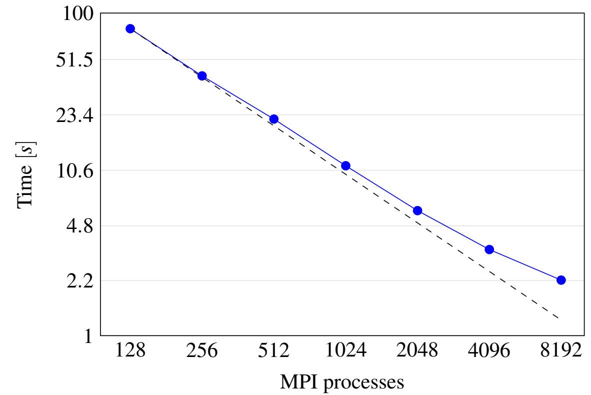


Figure 8: Execution times of the PMT pipeline in Ascent that includes computing the merge tree and the segmentation, as well as performing parallel volume-rendering of the segmentation field.

of factors could be at play, one is the increased overhead and another one is the potential cost the volume rendering of the local data, which at the highest scale has 127K elements. The results, however, are still comparable to the original BabelFlow study that reported similar strong scaling behavior.

6.3 Iso-surfaces Produced by LegoFlow

In this use case, we look at the iso-surfaces extract described in Section 5.3 and run it in an Ascent pipeline in a number of scenarios. This use case highlights both the portability and composability of dataflow graphs in in situ analytics.

In the first scenario, we generated an example dataset using Conduit helper routines and passed it as an input to the LegoFlow-based iso-surfaces extract. The goal was to validate that the resulting image, shown in Figure 9, corresponds to an image produced by a native Ascent pipeline consisting of the *contour* filter and a render scene. In both cases, the same VTK-m routines are used to first compute iso-surfaces using the marching cubes algorithm and then render them using ray tracing. The existing contour filter uses VTK-h to compute the global bounds of the data, which means the communication is delegated to DIY2 [27] and MPI. In our case, the LegoFlow-based extract replaced the MPI-centric communication with a runtime-agnostic dataflow pattern.

In the second scenario, we added Ascent instrumentation to the

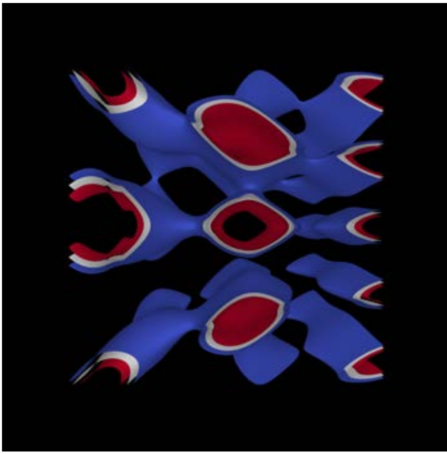


Figure 9: Iso surfaces produced and rendered by a LegoFlow graph composed from five separate subgraphs.

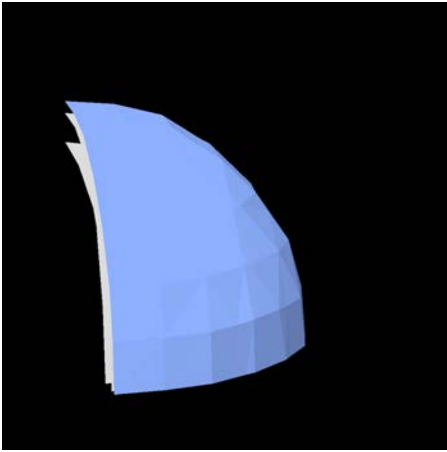


Figure 10: Iso surfaces of LULESH pressure data resulting from the LegoFlow-based extract running on top of Charm++.

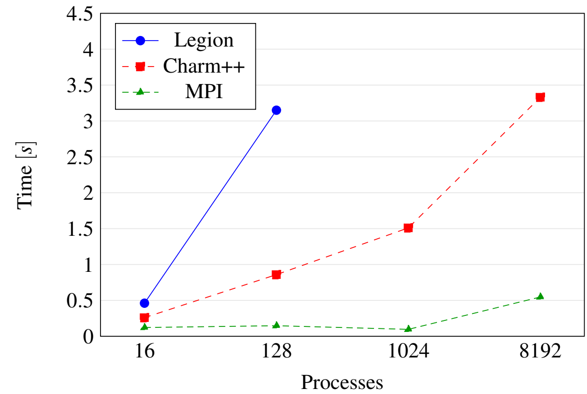


Figure 11: Execution times of LegoFlow Radix-k compositing running on top of Legion, Charm++, and MPI runtimes.

Charm++ port of LULESH [20], which stands for Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics. It is a proxy app that approximates the hydrodynamics equations discretely by partitioning the spatial problem domain into a collection of volumetric elements defined by a mesh. The Charm++ port does not change the physics computations, but it further subdivides the domains and assigns each sub-domain to a separate chare in a 3D chare array. The sub-domain data is used to initialize the Conduit node that is passed to Ascent.

When the LegoFlow extract is executed, the Charm++ controller creates a new chare array that corresponds to the dataflow graph. In a standalone application, as in the first scenario above, the execution of Ascent is driven by the main chare that can directly create new chare arrays without any limitation. When we run Ascent and all of the filters / extracts as part of a LULESH chare, Charm++ places limits on creation of new chare arrays. Specifically, these chare arrays can only be created asynchronously. This means that the first time we execute Ascent we just signal that we want to create a chare array for our dataflow graph, we then provide a callback, which is a chare entry function, and this callback will be called by Charm++ once it finishes creating the new chares. This callback also broadcast the handle for the new chare array so that we are able to invoke functions in the new chares. Our approach, therefore, for managing this interaction was to make minimal changes to existing LULESH chares and create special in situ chares. These chares call Ascent to initiate the creation of the dataflow chare array and then receive the callback invocation with the handle for the new chares. This handle is passed further as a parameter to the iso-surfaces extract in a second call to Ascent along with the Conduit node encapsulating the local LULESH data. The Charm++ controller in the extract then packages and passes the LULESH data to the corresponding dataflow chare. Also, LegoFlow creates a status chare that gets notified whenever dataflow chares start and finish executing. This way it can measure chare execution times and destroy the whole chare array in the end.

Figure 10 depicts three iso-surfaces produced by the chares executing the dataflow graph. The iso-values are computed automatically after the first layer in the graph computes the global data range. Input data are the LULESH pressure values at the 100th iteration. The run was performed on a small $32 \times 32 \times 32$ dataset with 8 LULESH chares and 8 analysis chares.

It is important to note that LULESH chares do not wait for the dataflow chares to finish. The input data is copied when it is passed to in situ chares and there is no danger of data corruption. This allows the simulation to be overlapped with in situ analytics similar to some other in situ frameworks such as Damaris/Viz [15] or TINS [14]. The difference in our case is that we are not bound to a single runtime. Furthermore, we do not need to dedicate cores to either simulation or analytics. A tasking runtime will usually schedule any ready-to-run task on any available core.

6.4 Scaling Radix-k Compositing

In this set of experiments, we compare the weak-scaling execution times of the Radix-k dataflow across the MPI, Legion, and Charm++ runtimes. Secion 6.1 describes that we used Devil Ray to render images of local data as an input to the LegoFlow-based compositing extract. We take the same approach in this case as well. For this purpose, we ran the Devil Ray renderer at the same increasing scales storing the local images at each scale. Then we read these images and used them as input to the Radix-k compositing dataflow that was executed on top of Charm++ and Legion.

The Charm++ processing model identifies a *logical node* (i.e., an OS process) and *processing elements* (i.e., worker threads) on a logical node, such that each processing element can independently execute a chare at any given moment. We ran the experiment with 32 logical nodes and one processing element per node. This is a similar setup to the previous MPI-based case study with 32 processes per Summit node and one thread per process.

For Legion, we have performed minimal modification to the original BabelFlow controller to leverage the new composable task graph.

Also in this case we run experiments using the same configuration with 32 processes per Summit node. As described in Section 3.2, this implementation relies heavily on the runtime's ability to distribute and manage large number of tasks which currently translates to increased runtime overhead at scale.

Figure 11 presents the resulting execution times for the Charm++, Legion, and MPI runtimes. The scaling trend for all three runtimes is on par with the results of executing a reduction-based compositing dataflow reported in the original BabelFlow study [31]. It is clear, however, that there is a room for improving the scaling behavior for both Charm++ and Legion. For the former, this could involve different load balancing strategies or limiting the number of chares for bigger dataflows. For the latter, there is a recent work from the Legion team that attempts to address the issue of scalability of collective operations [12]. This could potentially benefit the performance of the Legion controller in the future.

7 Conclusion

In this work, we introduce LegoFlow, a methodology that integrates portable dataflow graphs into in situ analytics and makes these graphs composable. The work builds upon an earlier effort in BabelFlow, whilst preserving the core strength of decoupling the analysis / visualization algorithm flow from the underlying runtime system. Users can now easily reuse existing algorithms, or quickly construct more complex dataflows for various in situ scenarios.

We explain the design of LegoFlow and demonstrate the ease with which new dataflows can be constructed. The integration with Ascent as well as the various filters and extracts that are part of this integration effort are presented in more detail. We then evaluate the methodology by running two different kinds of extracts and one filter. Most importantly, we highlight the experiment in which LULESH was coupled with LegoFlow-based in situ analytics and both the simulation code and the analysis code ran on top of Charm++. Leveraging the asynchronous task execution, this is an example of overlapping simulation and in situ analytics. This direction could be further expanded by implementing all of VTK-h communication patterns as LegoFlow dataflows, a possibility which is directly enabled by the composability mechanism. Essentially, this can make Ascent, and in situ analytics in general, transparently portable across different runtimes and architectures.

For future work, we plan to optimize the runtime controllers to improve performance. Also, we will investigate the option of making the dataflow graphs dynamic, that is creating additional nodes on the fly. This will enable more algorithms to be expressed as dataflow graphs.

Acknowledgments

This work was funded in part by NSF OAC awards 1842042, 1941085, NSF CMMI awards 1629660, DoE award DE-FE0031880, and the Intel Graphics and Visualization Institute of XeLLENCE. This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344, and UT-Battelle, LLC under contract DE-AC05-00OR22725. This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration. This material is based in part upon work supported by the U.S. Department of Energy NNSA under award DE-NA0002375. We also wish to thank Matt Larsen and Cyrus Harrison from Lawrence Livermore National Laboratory for insightful discussions and all their help with Ascent. LLNL release number: LLNL-CONF-825724.

References

[1] Ascent – A many-core capable lightweight in-situ visualization and analysis infrastructure for multi-physics HPC simulations, 2017.

[2] Conduit – Simplified Data Exchange for HPC Simulations, 2018.

[3] ParaView, 2018.

[4] Summit – Oak Ridge Leadership Computing Facility, 2018.

[5] Devil Ray – A Portably Performant Ray Tracer for High-Order Element Visualization, 2019.

[6] Laghos – High-order Lagrangian Hydrodynamics Miniapp, 2019.

[7] VisIt – An open source, interactive, scalable, visualization, animation, and analysis tool, 2019.

[8] VTK-h – A toolkit of scientific visualization algorithms for emerging processor architectures, 2019.

[9] BLAST – High-Order Finite Element Hydrodynamics, 2021.

[10] U. Ayachit et al. Performance Analysis, Design Considerations, and Applications of Extreme-scale in Situ Infrastructures. In *Proc. of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '16, pp. 79:1–79:12. IEEE Press, Piscataway, NJ, USA, 2016.

[11] A. C. Bauer, H. Abbasi, J. Ahrens, H. Childs, B. Geveci, S. Klasky, K. Moreland, P. O'Leary, V. Vishwanath, B. Whitlock, and E. W. Bethel. In Situ Methods, Infrastructures, and Applications on High Performance Computing Platforms. *Computer Graphics Forum*, 35(3):577–597, 2016.

[12] M. Bauer, W. Lee, E. Slaughter, Z. Jia, M. Di Renzo, M. Papadakis, G. Shipman, P. McCormick, M. Garland, and A. Aiken. Scaling Implicit Parallelism via Dynamic Control Replication. In *Proc. of the 2021 ACM Principles and Practice of Parallel Programming (PPoPP)*, pp. 1–16, 2021.

[13] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken. Legion: Expressing Locality and Independence with Logical Regions. In *Proc. of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12. IEEE Computer Society Press, 2012.

[14] E. Dirand, L. Colombet, and B. Raffin. TINS: A Task-Based Dynamic Helper Core Strategy for In Situ Analytics. In *SCA18 - Supercomputing Frontiers Asia 2018*, pp. 159–178, Mar. 2018.

[15] M. Dorier, R. Sisneros, T. Peterka, G. Antoniu, and D. Semeraro. Damaris/Viz: A nonintrusive, adaptable and user-friendly in situ visualization framework. In *2013 IEEE Symposium on Large-Scale Data Analysis and Visualization (LDAV)*, pp. 67–75, 2013.

[16] A. Heirich, E. Slaughter, M. Papadakis, W. Lee, T. Biedert, and A. Aiken. In Situ Visualization with Task-Based Parallelism. In *Proceedings of the In Situ Infrastructures on Enabling Extreme-Scale Analysis and Visualization*, ISAV'17, p. 17–21. ACM, 2017.

[17] T.-W. Huang, C.-X. Lin, G. Guo, and M. D. F. Wong. Cpp-Taskflow: Fast Task-Based Parallel Programming Using Modern C++. In *Proc. of the 2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 974–983, 2019.

[18] W. M. Johnston, J. R. P. Hanna, and R. J. Millar. Advances in Dataflow Programming Languages. *ACM Comput. Surv.*, 36(1):1–34, Mar. 2004.

[19] L. V. Kale and S. Krishnan. CHARM++: A Portable Concurrent Object Oriented System Based on C++. In *Proc. of the 8th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, OOPSLA '93, p. 91–108. ACM, 1993.

[20] I. Karlin, A. Bhatele, B. L. Chamberlain, J. Cohen, Z. Devito, M. Gokhale, R. Haque, R. Hornung, J. Keasler, D. Laney, E. Luke, S. Lloyd, J. McGraw, R. Neely, D. Richards, M. Schulz, C. H. Still, F. Wang, and D. Wong. LULESH Programming Model and Performance Ports Overview. Technical Report LLNL-TR-608824, December 2012.

[21] 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 *Proc. of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14, pp. 1020–1031. IEEE, 2014.

[22] M. Larsen, J. Ahrens, U. Ayachit, E. Brugger, H. Childs, B. Geveci, and C. Harrison. The ALPINE In Situ Infrastructure: Ascending from the Ashes of Strawman. In *Proc. of the 3rd Workshop In Situ Infrastructures on Enabling Extreme-Scale Analysis and Visualization*, ISAV'17, pp. 42–46. ACM, New York, NY, USA, 2017.

[23] B. J. Lee, X. Xiao, F. E. Hernandez Perez, H. G. Im, and R. Sankaran. KARFS: A Combustion DNS Solver for Hybrid Computing Archi-

tectures. In *Poster presented at the 36th International Symposium on Combustion*, pp. 1–1, 2016.

[24] C.-X. Lin, T.-W. Huang, G. Guo, and M. D. F. Wong. An Efficient and Composable Parallel Task Programming Library. In *Proc. of the 2019 IEEE High Performance Extreme Computing Conference (HPEC)*, pp. 1–7, 2019.

[25] 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*, 14(4):59–68, July 1994.

[26] 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*, 36(3):48–58, 2016.

[27] D. Morozov and T. Peterka. Block-parallel data analysis with DIY2. In *Proc. of the 2016 IEEE 6th Symposium on Large Data Analysis and Visualization*, LDAV '16, pp. 29–36. IEEE, 2016.

[28] S. G. Parker and C. R. Johnson. SCIRun: A Scientific Programming Environment for Computational Steering. In *Proceedings of the 1995 ACM/IEEE Conference on Supercomputing*, Supercomputing '95. ACM, 1995.

[29] T. Peterka, D. Goodell, R. Ross, H.-W. Shen, and R. Thakur. A Configurable Algorithm for Parallel Image-Compositing Applications. In *Proc. of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09. Association for Computing Machinery, 2009.

[30] T. Peterka, R. Ross, A. Gyulassy, V. Pascucci, W. Kendall, H.-W. Shen, T.-Y. Lee, and A. Chaudhuri. Scalable parallel building blocks for custom data analysis. In *Proc. of the 2011 IEEE Symposium on Large Data Analysis and Visualization*, LDAV '11, pp. 105–112. IEEE, 2011.

[31] S. Petruzza, S. Treichler, V. Pascucci, and P.-T. Bremer. BabelFlow: An Embedded Domain Specific Language for Parallel Analysis and Visualization. In *Proc. of 2018 IEEE International Parallel and Distributed Processing Symposium*, IPDPS '18, pp. 463–473. IEEE, 2018.

[32] R. Rieben. The MARBL Multi-physics Code. In *Poster presented at the ECP Annual Meeting 2020*, pp. 1–1, 2020.

[33] A. C. Sena, E. S. Vaz, F. M. Franca, L. A. Marzulo, and T. A. Alves. Graph Templates for Dataflow Programming. In *Proc. of the 2015 International Symposium on Computer Architecture and High Performance Computing Workshop (SBAC-PADW)*, pp. 91–96. IEEE Computer Society, oct 2015.

[34] S. S. Vazhkudai et al. The Design, Deployment, and Evaluation of the CORAL Pre-Exascale Systems. In *Proc. of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, SC '18. IEEE Press, 2018.

[35] M. Voss, R. Asenjo, and J. Reinders. *Flow Graphs: Beyond the Basics*, pp. 451–511. Apress, 2019.

[36] F. Zheng, H. Yu, C. Hantas, M. Wolf, G. Eisenhauer, K. Schwan, H. Abbasi, and S. Klasky. GoldRush: Resource Efficient in Situ Scientific Data Analytics Using Fine-Grained Interference Aware Execution. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13. ACM, 2013.