FleCSI 2.0: The Flexible Computational Science Infrastructure Project

Ben Bergen[1(✉)], Irina Demeshko[1(✉)], Charles Ferenbaugh[1], Davis Herring[2(✉)], Li-Ta Lo[1], Julien Loiseau[1], Navamita Ray[1], and Andrew Reisner[1]

[1] Applied Computer Science Group (CCS-7), Los Alamos National Laboratory, Los Alamos, NM 87545, USA
{bergen,irina,cferenbaugh,ollie,jloiseau,nray,areisner}@lanl.gov
[2] Lagrangian Codes Group (XCP-1), Los Alamos National Laboratory, Los Alamos, NM 87545, USA
herring@lanl.gov

Abstract. The FleCSI 2.0 programming system supports multiphysics application development through a runtime abstraction layer, and by providing core topology types that can be customized for specific numerical methods. The abstraction layer provides a single-source programming interface for distributed and shared-memory data parallelism through *task* and *kernel* execution, and has been demonstrated to introduce virtually no runtime overhead. FleCSI's core topology types represent a rich set of basic data structures that can be specialized to create application-facing interfaces for a variety of different physics packages. Using the FleCSI control and data models, it is straightforward to compose multiple packages to create full multiphysics applications. When used with a task-based backend, FleCSI offers extended runtime analysis that can increase task concurrency, facilitate load balancing, and allow for portability across heterogeneous computing architectures.

Keywords: Multiphysics · Computational science · Applied mathematics · Task-based runtimes · Heterogeneity · Performance portability · Accelerators

1 Introduction

FleCSI is a C++ framework designed to support multiphysics application development through a runtime abstraction layer and a collection of useful topology and data types. Many of these capabilities take the form of class and function templates whose behavior is customized by application-specific functions and data and policy types. The abstraction layer insulates developers from underlying complexity, while providing a single-source, integrated programming model that is mapped on top of different low-level backends. FleCSI introduces a functional programming model with runtime, control, execution, and data abstractions that are consistent both with MPI [11] and with state-of-the-art, task-based backends

R. Chaves et al. (Eds.): Euro-Par 2021, LNCS 13098, pp. 480–495, 2022.
https://doi.org/10.1007/978-3-031-06156-1_38

such as Legion [3] and HPX [14]. When configured with one of the task-based backends, FleCSI provides dynamic scheduling of data and task placement for increased concurrency and application performance.

1.1 Structure

This paper describes the main features and goals of the FleCSI 2.0 programming system. It provides an overview and examples of the programming model and its components. These are explained in Sect. 2. Brief descriptions of the core topology types are given in Sect. 3. In Sect. 4, two sample applications that use FleCSI 2.0, *Model for Prediction Across Scales (MPAS)* and *FleCSPH* are discussed. Some performance results regarding runtime overhead of the abstraction layer are presented in Sect. 4.3, with concluding remarks and future work in Sect. 5.

1.2 Related Work

FleCSI is a *unified* runtime in the sense that it supports a single interface for programming both the distributed and shared-memory components of modern computing systems. Two similarly unified programming systems are *Uintah* [12] and the *Multi-Processor Computing runtime (MPC)* [17].

Like FleCSI, Uintah uses a task-based runtime for distributed-memory execution. However, Uintah's task concurrency must be explicitly scheduled by the user. MPC transparently enables shared-memory support with MPI through an *MPI+X* programming model. A significant feature of MPC is its ability to run MPI processes *inside* of threads. This approach can reduce message latency (*memcpy* of messages), and requires fewer communication endpoints. The primary advantage of FleCSI over these systems is that, under FleCSI, tasks and data can be dynamically mapped to compute resources (when using a task-based backend), while Uintah and MPC both employ a static mapping aligned with the runtime processes, i.e., ranks.

2 Programming Model

The FleCSI 2.0 programming model provides explicit runtime, control, data, and execution models that are designed to provide users with a rich environment for application development on state-of-the-art heterogeneous system architectures. These are described in the following sections.

2.1 Runtime Model

The runtime model is how you control the FleCSI runtime system itself. The basic interface includes the functions: `initialize`, `start`, and `finalize`. These are abstractions of the underlying runtime interfaces, e.g., Legion, MPI, HPX, and Kokkos [9], whereby the correct combination of start-up, execution, and

shut-down processes will be invoked, depending on the configuration of FleCSI. The runtime model also provides support for creating command-line options, a logging utility called *flog*, and a timing and profiling interface to Caliper [6].

This model will be extended in a future feature release to provide a richer set of options to allow more precise placement of processes and memory allocations in order to address the performance challenges of running on modern, heterogeneous architectures that have deep, complex memory hierarchies, and which require exploitation of processor–memory affinities to achieve the best throughput.

2.2 Control Model

The FleCSI control model is how the overall execution structure of a FleCSI-based program is defined. FleCSI provides a core control type that can be customized with specialization-defined control points (the skeleton of the application structure). Application developers register actions under the control points, which may have dependencies on each other. FleCSI sorts the actions under a control point to create a runtime program order. Along with *tasks* and *kernels* (parts of the execution model), the control model forms an execution hierarchy with the following relationships:

- **Control Points:** Identify the high-level stages of the application. The execution order of the control points is statically defined. Cycles may be defined over any subset of the control points. The control points form a *control-flow graph (CFG)*.
- **Actions:** Registered under control points. Any two actions under a single control point may have a dependency defined between them. The actions under a single control point form a *directed acyclic graph (DAG)*. Actions allow composition of *contributed* packages. Actions are always executed in a sequential program order defined by: 1.) the order of the control points, and then 2.) the topologically sorted order of the DAG of actions under each control point.
- **Tasks:** Executed from inside of an action. Task launch may be *single* or *index*. Tasks operate on data that are logically distributed over a partitioned address space. Dependency analysis (Legion backend only) allows tasks to be executed concurrently by the runtime. A given *point task* may be executed on any runtime process on any address space (Legion backend only).
- **Kernels:** Executed from inside of a task. Kernels execute data-parallel operations over a local address space. Memory consistency of kernel execution is explicit (*relaxed-consistency*).

This model replaces the normal hard-coded execution structure of an application, instead providing a well-defined, extensible mechanism that can easily be verified and visualized. Figure 1 shows output from a FleCSI-based program using the command-line argument `--control-model`.

The primary advantage of this approach is that FleCSI-based applications can add new actions to any of the DAGs in the model, with associated dependencies, without requiring modification to the core application code. Because of FleCSI's data model (discussed in Sect. 2.3) these new actions will fold seamlessly into the existing control structure, allowing extensibility and experimentation.

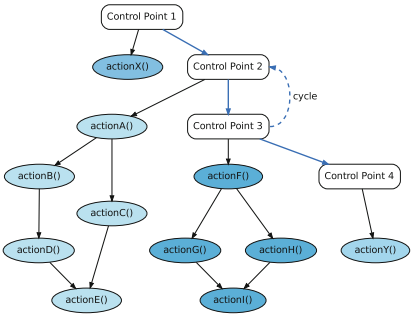

Fig. 1. Example FleCSI control model.

2.3 Data Model

Because FleCSI tasks may be executed on any memory space, all the application's data must be managed by the runtime so that a copy of it may be made available in that memory space and any changes propagated to the next task needing it. (Global variables may also be used, but only for information that is constant across and throughout the simulation, since which process executes which task is arbitrary.) The FleCSI data model closely follows the Legion model, but with some simplifications and elaborations described in the following subsections. Legion provides a data model based on associative arrays called *regions* whose multi-index keys are called *index points*. The domain of keys is called an *index space*, which may have an arbitrary shape but for performance reasons is often hyperrectangular.

A region is a collective object whose storage is typically distributed across multiple memory spaces. A *partition* is a collection of subsets of an index space (or a region based on it) that may be mapped separately; the collection is indexed with integer *colors*, and each subset is called a *subspace*. Multiple unrelated partitions of a single region may exist; it is often the case that the subsets in each are disjoint, that their union is the entire region, or both. In particular, tasks that operate on disjoint subsets can write to them in parallel; read-only access to subsets that overlap those from different partitions can be used to automatically transfer data between sequences of such tasks.

Each region has some number of *fields* defined on it of various types. Legion specifies a type only as a number of bytes and, optionally, a set of serialization functions to marshal heap allocations between memory spaces. Each field is available throughout the index space, but memory is allocated only for the field-subspace pairs that are actually used.

FleCSI organizes multiple related regions, e.g., the cells and vertices of a mesh, into *topologies*. Since FleCSI does not use the Legion feature of creating multiple regions from the same index space, the term "index space" is used in the user interface to describe a selector among the regions for a topology. Each topology type is defined as a *specialization* of one of the *core* topology types defined by FleCSI that specifies a number of properties like the dimensionality of a mesh or which kinds of connectivity information to store explicitly.

Extended Index Space. To allow most tasks to treat its local application data as a traditional, contiguous array, FleCSI stores application data on a particular sort of two-dimensional index space based on *rows*. Each such Extended Index Space is typically partitioned into subsets that are prefixes of each row, such that each color is the index of a (partial) row. The prefixes need not all be the same length, and the partitions can be recreated with different lengths: the underlying index space has a very large number of columns, most of which are never realized in memory. If Legion is in use, it automatically copies the contents of a row that has grown into a prefix of any new allocation. Otherwise, the restricted form of index spaces significantly simplifies the implementation of a backend based on another runtime, e.g., HPX.

For certain topologies, e.g., an unstructured mesh, each row comprises *exclusive*, *shared*, and *ghost* index points: the field values at each ghost point are copies of those at a particular shared point specified by a *copy plan* created by such a topology. Whenever ghost values are required by a task, FleCSI performs the copies specified in the plan if the shared values have been changed since the previous copy. This copying is explicit from the perspective of the runtime but implicit from the perspective of the author of the relevant tasks. Other topologies, e.g., a set of particles, have more dynamic communication patterns; temporarily copying data into a special *buffer* field implements these copies in terms of an ordinary copy plan for the buffers.

Accessors. Fields are registered by creating objects of a *field definition* type, parameterized on the data type as well as the type of the topology, prior to initializing FleCSI. Every instance of the nominated topology type is then equipped with that field, although since each has its own partitions memory is not in general allocated for all of them.

Since the runtime must in general move field data to make it available to a task, ordinary pointers to it are available only inside a task. These are wrapped into *accessors* which provide one of several interfaces for the field values and specify the privileges required by a task. Outside a task, an opaque *field reference* is used instead that nominates a field definition and a topology instance. When a field reference is passed as an argument to a task that declares a compatible accessor as a parameter, the runtime provides a pointer to the relevant array in the task's memory space to use for the parameter.

FleCSI's operation may be described as a distributed version of the C++ memory allocation expression `new T[n]`: `T` is specified by the field definition, `n` is specified by the topology, and the resulting pointer eventually appears as the task parameter.

Layouts. While FleCSI's model for index spaces is simpler and more restricted than the Legion model, its model for data types is more complicated. In addition to restricting accessors to using the type specified in a field definition, FleCSI provides several *layouts* for representing non-trivial objects. The element types used with a layout must themselves be bitwise-copyable; for efficiency, FleCSI does not use mechanisms like Legion's serialization interface for transfer between memory spaces.

The default layout is *dense*, which is simply a one-dimensional array of the specified type `T`. The special case of a *single* element, useful for metadata pertaining to one color of a topology, is provided for convenience. The *ragged* layout emulates a `std::vector<T>` at each index point. The *sparse* layout emulates a `std::map<std::size_t,T>` at each index point. The *particle* layout stores an unordered collection of `T` objects with efficient insertion, removal, and iteration; it is similar to the "bucket array" data structure, albeit with a single bucket.

All of these are implemented recursively in terms of one or more simpler accessors, with a *raw* accessor that manages uninitialized memory (rather than objects) as the base case. The field definitions follow the same structure, automatically registering fields necessary for storing metadata like the sizes of individual ragged arrays. Ragged and sparse fields require support from the topology for allocating memory for the array elements, which is provided centrally for all but the simplest topologies.

Topology Instances. Application developers create a topology instance with the assistance of a *topology slot*, which allows the specialization to contribute to the topology's initialization and allows the client to defer it (so that slots can be declared statically if desired). A topology instance is created from a *coloring* which describes the memory layout required and is constructed using its own slot that automatically utilizes an MPI task (Sect. 2.4), as is commonly required for initialization from external data sources. Listing 1.1 illustrates the typical process of initializing a topology with these types given a specialization Sect. 3.1.

Listing 1.1. Topology initialization

```
1  topology::cslot color;
2  topology::slot topo;
3
4  void setup_color() {
5      // May be distributed rather than duplicated:
6      auto data = load("file name");
7      color.allocate(data.header(), data.body());
8  }
9
10 void setup_topology() {
11     topo.allocate(color.get());
12 }
```

A topology slot may also be passed to a task with a *topology accessor* as a parameter. Such an accessor uses internal fields established by the topology to interpret the contents of flat arrays in terms of its logical structure. Much like accessors for dynamic layouts like ragged, these are implemented in terms of accessors for those structural fields. Field indices are typically supplied by a topology accessor as "strong typedef" objects to reduce the chance of misconstruing them as pertaining to another index space. Flat arrays can be interpreted

as having multiple dimensions through the use of a limited implementation of `mdspan` as proposed for C++23 [7].

2.4 Execution Model

FleCSI has two mechanisms for expressing work: tasks and kernels. Tasks operate on data distributed to one or more address spaces, and use data privileges to maintain memory consistency. Kernels operate on data in a single address space, but require explicit barriers to ensure consistency. This is generally referred to as a relaxed-consistency memory model.

Listing 1.2. Single-Source Execution

```
1  const field<double>::definition<topology,
2    topology::entities> values;
3
4  void init_field(topology::accessor<ro> t,
5     field<double>::accessor<rw> f) {
6        forall(e, t.entities()) {
7            f(e) = 1.0; // dummy initialization
8        };
9  }
10
11 void setup_field() {
12     execute<init_field>(topo, values(topo));
13 }
```

Listing 1.2 shows a continuation of Listing 1.1 that uses the `execute` method to invoke a task (line 12). There are several things to observe about this code example:

- Tasks are invoked using the `execute` function. FleCSI also provides the `reduce`, and `test` functions. Invocations of `reduce` capture the task return value in a `future`, which can be queried like `f.get()` to retrieve the reduced value (distributed-memory). The `test` function executes a task as a reduction and returns the sum of the point task returns, i.e., a value of zero means success.
- Privileges on accessors passed to the task, e.g., the `rw` in `field<double>::accessor<rw>` (line 5), determine the memory consistency operations that will be performed on the respective data upon the next task invocation. Valid privileges are `na` (no access), `ro` (read-only), `wo` (write-only), and `rw` (read-write). The privilege `na` can be used to defer consistency updates.
- The `forall` invocation (line 6) executes the loop body in shared-memory, data-parallel over the specified iterator, e.g., `t.entities()`, potentially handling any data motion that is required to move data to the correct address space. Depending on the backend runtimes used, this code example can seamlessly execute a task in distributed-memory, data parallel over a collection of

nodes, and in shared-memory, data parallel on attached GPU devices on each node.

3 Core Topologies

Each core topology type supports a number of numerical methods with similar data requirements. FleCSI itself implements these types in terms of internal facilities that are not themselves exposed to clients. In contrast, FleCSI defines only the few specializations used by those facilities; others may be defined by applications or by libraries of specializations that are themselves suitably generic.

For example, consider the data representations for the following methods: *particle-in-cell (PIC)*, *molecular dynamics (MD)*, *material point method (MPM)*, *smoothed-particle hydrodynamics (SPH)*, and *Monte Carlo*. These methods are all based on particles, but they vary as to whether interactions between the particles are direct or are mediated by fields stored on a mesh. Here, only MD and SPH feature direct interactions; such methods typically need ghost particles to calculate them, as well as efficient neighbor lookups. The `topo::ntree` topology (Sect. 3.4) supports these methods, organizing the particles into an octree for an $\mathcal{O}(logN)$ nearest-neighbor search.

For the non-interacting particle methods considered here, a primary concern in parallel implementations is to identify *active* particles in the local memory space, and to efficiently move particles between memory spaces. The `topo::set` topology (Sect. 3.5) supports these methods, organizing the particles using the particle layout (Sect. 2.3) for an $\mathcal{O}(1)$ step to the next active particle (using a *low-complexity jump-counting pattern skipfield* data structure) [4,5].

Other combinations are possible. For example, many MD codes reduce search complexity by imposing a *Cartesian* mesh on the domain with the mesh increments set to some *cutoff* metric. The particles for each cell in this approach can be stored in a ragged field (Sect. 2.3) defined on an `topo::narray` topology (Sect. 3.3).

In addition to allowing specialization of the core types, FleCSI's data model also allows easy composition of topologies into more complex types. Consider the design of a data structure to support *block-structured adaptive mesh refinement (AMR)*. The `topo::ntree` topology is well-suited for tracking refinement because of its fast neighbor lookup. This can be combined with a specialization of `topo::narray` to provide a structured mesh interface at each node of the tree. Field data registered at the nodes can be *viewed* using `util::mdspan`.

The primary take-away from this section should be that the FleCSI core topology types and layouts provide data structure support for a wide variety of applied methods, which can be composed to support complex, multi-physics applications development. Table 2 in Appendix A provides some suggested applications of the core types for common numerical methods.

3.1 Specialization Structure

The notion of a *specialization* is formalized in the FleCSI class structure with the `topo::specialization` type, as shown in Listing 1.3. The specialization serves both as a policy for the core topology and as an interface for the application; its general structure is the same for all of the FleCSI core topologies, although the specific details are different for each.

Listing 1.3. Specialization Structure

```
struct my_topo : topo::specialization<topo::topology, my_topo> {

  enum index_space {vertices, integration};

  template<class B>
  struct interface : B {
    // Iterator over an Index Space
    template<index_space IndexSpace>
    auto entities() {/* ... */}

    // Iterator over entities at an entity
    template<index_space To, index_space From>
    auto entities(topo::id<From> from) const {/* ... */ }
  };

}; // struct my_topo
```

All FleCSI topology types define iterators over the *entities* of the topology, with relational types, e.g., `topo::unstructured`, and `topo::ntree`, also defining *sub-entity* iterators.

3.2 topo::unstructured

The `topo::unstructured` topology provides a *graph-like* data structure, suitable for defining unstructured meshes. Specializations can define an arbitrary number of entity types and can control what connectivity information is *stored* (as opposed to what must be *computed*), allowing flexibility in memory vs compute complexity. Coloring utilities are provided with a *mesh definition* abstraction for scalable distribution of input meshes. In addition to the standard *entity*, and *sub-entity* iterators, `topo::unstructured` allows users to define custom iterators using *entity lists*, which are useful for tracking, e.g., domain boundary interfaces.

3.3 topo::narray

The `topo::narray` topology is an n-dimensional array data structure, with support for arbitrary *halo* and *boundary* depths, and optional periodicity in each axis. Halo dependencies optionally include diagonal connections. Listing 1.4 shows an example of a two-dimensional, *Cartesian* mesh interface created using `topo::narray`.

Listing 1.4. *Cartesian* Mesh Example using `topo::narray`

```
void poisson::task::smooth(mesh::accessor<ro> m,
  field<double>::accessor<rw, ro> ua,
  field<double>::accessor<ro, ro> fa) {
  auto u = m.mdspan<mesh::vertices>(ua);
  auto f = m.mdspan<mesh::vertices>(fa);
  const auto dxdy = m.dxdy();
  const auto dx_over_dy = m.xdelta() / m.ydelta();
  const auto dy_over_dx = m.ydelta() / m.xdelta();
  const auto factor = 1.0 / (2 * (dx_over_dy + dy_over_dx));

  for(auto j : m.vertices<mesh::y_axis>()) {
    for(auto i : m.vertices<mesh::x_axis>()) {
      u[j][i] = factor *
                (dxdy * f[j][i] +
                 dy_over_dx * (u[j][i + 1] + u[j][i - 1]) +
                 dx_over_dy * (u[j + 1][i] + u[j - 1][i]));
    }
  }
}
```

3.4 topo::ntree

The `topo::ntree` topology provides special 1, 2, or 3 dimensional hashed tree support, e.g., $n \equiv 3 \Rightarrow$ a *hashed octree*, based on the *Barnes–Hut approximation* [2] with a hashing strategy derived from *Warren & Salmon* [22]. Particle distribution is supported using a naive coloring of a sorted *Morton (Z-Order)*, or *Hilbert* space-filling curve [1].

FleCSI provides several iterator patterns for accessing and modifying the tree. In particular, direct entity and node access, and both *Depth-First Search (DFS)* and *Breadth-First Search (BFS)* traversals are supported, with DFS support for *postorder*, *preorder*, *reverse postorder*, and *reverse preorder* variants. Users can also develop custom traversals using *sub-entity* or *sub-node* iterators. Listing 1.5 provides examples of DFS and BFS traversal iterators.

Listing 1.5. NTree Iterator Example

```
// Depth-First Traversal with reverse preorder
for(auto n : t.dfs<reverse_preorder>()) { /* ... */ }

// Breadth-First Traversal
for(auto n : t.bfs()) { /* ... */ }
```

As with other FleCSI topology types, `topo::ntree` supports field definition using the FleCSI data model (Sect. 2.3) and implicit dependency consistency through accessor permissions. Neighbor interactions are controlled by the user's specialization through rules that define *node-node*, *entity-entity*, and *node-entity* interactions. Using these rules, the runtime automatically retrieves dependencies

from other colors as needed. The tree can be re-sorted and re-distributed as needed to track particle evolution.

3.5 topo::set

The `topo::set` topology is designed to support non-interacting particle methods, i.e., those that do not have direct particle-particle interactions, using the *particle* layout discussed in Sect. 2.3. The topology extends the basic iterators and distributed-memory support of the underlying layout with customizable interfaces for coloring and binning particles, and specialized accessors that can be used to track local particles as they evolve, potentially leaving their original color, with or without *path-dependent* trajectories. The `topo::set` topology is *dependent* in FleCSI's nomenclature because individual particles are *colored* according to their relationship to an *independent* topology, e.g., a distributed mesh.

4 Sample Applications

To demonstrate the use of FleCSI in implementing simulation codes, two applications are described representing disparate FleCSI topologies. In addition to covering multiple FleCSI topologies, these applications demonstrate the implementation of important numerical methods relevant to many simulation codes. A simple investigation of abstraction overhead in the context of these applications is used to verify its minimal impact on application performance.

4.1 MPAS-O-FleCSI

`MPAS-O-FleCSI` is an FleCSI-based implementation of models from MPAS (Model for Prediction Across Scales) [19]. It is part of the CANGA project, that aims at investigating task-based approaches for Earth System Models to achieve maximum performance and better manage architectural and scientific complexity. Currently, several simulation applications are developed on the `MPAS-O-FleCSI` framework, including the shallow water core. The shallow water core of `MPAS-O- FleCSI` implements the numerical scheme from [20] to solve the nonlinear shallow water equations using a C-grid finite-volume discretization on Variable-resolution Spherical Voronoi Tessellations (SCVTs) [13]. The unstructured topology is used to support SCVTs for `MPAS-O-FleCSI` with mesh files read using HDF5 (Fig. 5).

4.2 FleCSPH

FleCSPH [15] is a multiphysics Smoothed Particle Hydrodynamics (SPH) simulation code, using the FleCSI 2.0 `topo::ntree` topology, allowing efficient computation of bulk transport and long-range interactions (Fig. 6). FleCSPH is a capable simulation application, including initial data generators, particle

relaxation with external potentials, *weighted Voronoi tessellations (WVT)* and *artificial pressure method (APM)*, flexible boundary conditions, and complexity reduction using *Fast Multipole Method (FMM)* (up to 4th order). Angular and linear momentum are conserved using a novel FMM algorithm.

With a primary focus on astrophysics, FleCSPH also includes support for gravitational waveform extraction and gravitational radiation reaction, and both analytic and tabulated *equation of state (EOS)* calculations. The application is easily extended to support new features and user-specific problem setups. Standard HDF5 formats based on H5part are used for input and output.

4.3 Abstraction Overhead

Through extensions of its underlying runtime models, FleCSI provides various usability and portability benefits. Leveraging Legion and Kokkos, FleCSI enables applications to be run on diverse hardware and in a heterogeneous environment. Additionally, FleCSI provides type checking and iterators for tasks facilitating increased usability. To be effective, however, the cost of such abstractions must not significantly impact the performance of applications.

The shallow water core from `MPAS-O-FleCSI` is used to investigate abstraction overheads of FleCSI execution and data structures. To this end, we first consider a vector triad from stream [16] (see Algorithm 1).

Algorithm 1. Vector Triad

1: **for** $i = 1, \ldots, N$ **do**
2: $\quad A[i] = B[i] + C[i] * D[i]$
3: **end for**

This provides a baseline for performance investigation as an expectation for performance is easily obtained on a computation relevant to the memory bound nature of second order finite-volume calculations.

Figure 2 shows performance of the vector triad computed over cells in an MPAS mesh on a Power 9 CPU. The FleCSI line shows the vector triad implemented in a FleCSI task using FleCSI fields, iterators (`forall` abstraction using Kokkos with the OpenMP [8] backend), and accessors on the MPAS specialization of `topo::unstructured`. The OpenMP line shows the vector triad implemented in OpenMP using sizes consistent with the number of cells in the given MPAS mesh. Figure 2 demonstrates relatively small overhead when compared to OpenMP.

Figure 3 shows performance of the vector triad on a Volta GPU. The FleCSI line shows the performance of the vector triad implemented in a FleCSI task using the `forall` abstraction with the MPAS unstructured topology specialization. The Kokkos line shows performance of the vector triad using Kokkos with sizes consistent to the number of cells in the corresponding MPAS mesh. Figure 3 shows the abstraction overhead is largest for small mesh sizes with the highest variability.

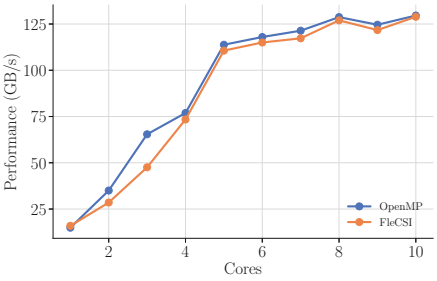

Fig. 2. Vector triad over 64 million cells in an MPAS mesh on Power 9 CPU.

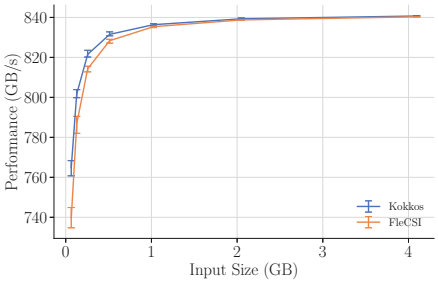

Fig. 3. Vector triad over cells in an MPAS mesh on Volta. Error bars show variation over ten runs

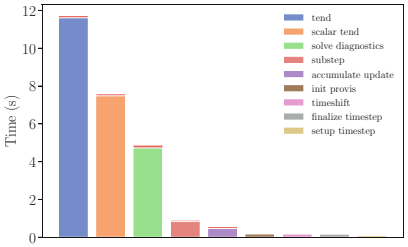

Fig. 4. Shallow water core task execution and runtime overhead.

Figure 4 shows the run time of tasks in the time integration loop of the shallow water core in `MPAS-O-FleCSI`. The FleCSI overhead associated with these tasks is shown above each bar.

Overall, the FleCSI overhead associated with these tasks is insignificant relative to the run time of the task.

5 Conclusions and Future Work

FleCSI 2.0 offers many improvements, and extended capabilities over the original FleCSI 1.4 release, including a completely C++17-compliant interface, new topology types, more flexible topology instance control, better support for scalable topology colorings, a composable internal interface for developing and implementing new topology types, full implementation of FleCSI's explicit programming model components, arbitrary launch domain support for *M-to-N color* to *process* mappings (execution model only), improved profiling utilities, and a fully serializable logging utility (FLOG) that can aid in code development and debugging.

A future minor release of FleCSI 2.0 will add support for arbitrary data mappings, using *color maps* to extend the capability provided by our current *M-to-N* execution model. This will enable straightforward implementations of

complex mapping algorithms, e.g., parallel rendezvous [18]. Future work will also include refinement and integration of a previously-developed model for multi-material representations, additional topology support (In particular, support for *K-D Trees.*), a new interface for controlling *tuneables* in conjunction with several custom mappers targeting upcoming DOE supercomputers, e.g., Crossroads [10] and El Capitan [21], and improved scalability and performance.

Acknowledgments. FleCSI 2.0 is the culmination of several years of research and development, with many important contributors. We would like to acknowledge the following individuals for direct code contributions:

Table 1. Direct Code Contributors to FleCSI

Dani Barrack	Marc Charest	Scot Halverson
Christoph Junghans	Sumathi Lakshmiranganatha	Jonas Lippuner
Nick Moss	Robert Pavel	Jonathan Pieterila Graham
Galen Shipman	Lukas Spies	Martin Staley
Karen Tsai	John Wohlbier	Wei Wu

The initial design and development of FleCSI was funded under the Advanced Technology Development and Mitigation (ATDM) subprogram of LANL's ASC program (NNSA/DOE). This work would not have been possible without close collaborations with the Legion and HPX teams, and the Ristra Project (part of ATDM). We would also like to acknowledge the leadership of the Ristra project: Aimee Hungerford, and David Daniel. The FleCSI project and the Darwin compute cluster are both funded by the Computational Systems and Software Environments (CSSE) subprogram of LANL's ASC program (NNSA/DOE).

FleCSI website: https://flecsi.org.
Source code and issue tracking: https://github.com/flecsi/flecsi.
This work is approved for unlimited release: LA-UR-21-25604

A Topology Applications

As mentioned in Sect. 3, this table gives some suggestions for the particular numerical methods that can be implemented with the various FleCSI core topology types. This list is meant only as an example, and is by no means exhaustive.

Table 2. Suggested FleCSI Topology Applications

Applied Method	Core Topologies
Lagrangian Hydrodynamics	`topo::unstructured`, `topo::narray`
Eulerian Hydrodynamics	`topo::narray`, `topo::ntree`
Smoothed-Particle Hydrodynamics	`topo::ntree`
Finite Element Method	`topo::unstructured`, `topo::narray`
Finite Volume Method	`topo::unstructured`, `topo::narray`
Discontinuous Galerkin	`topo::unstructured`, `topo::narray`
Material-Point Method	`topo::set`
Particle-In-Cell	`topo::set`
Monte Carlo (particle coloring)	`topo::ntree`
Monte Carlo (mesh coloring)	`topo::set`

B Sample Figures

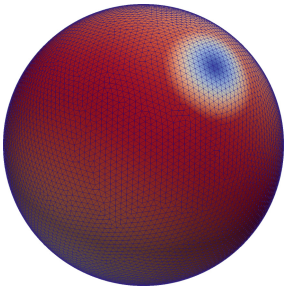

Fig. 5. Example of MPAS mesh used to setup a standard shallow water test case from [23].

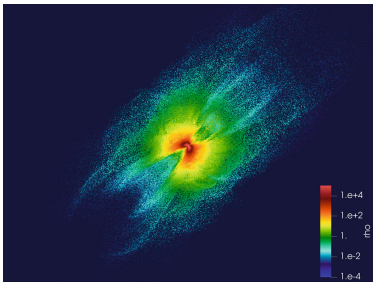

Fig. 6. Simulation of a neutron star merger disk outflow using FleCSPH.

References

1. Bader, M.: Space-Filling Curves: An Introduction with Applications in Scientific Computing. Springer, Cham (2012)
2. Barnes, J.E., Hut, P.: A hierarchical O(n-log-n) force calculation algorithm. Nature **324**, 446 (1986)
3. Bauer, M., Treichler, S., Slaughter, E., Aiken, A.: Legion: expressing locality and independence with logical regions. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, SC 2012, Washington, DC, USA. IEEE Computer Society Press (2012)
4. Bentley, M.: The high complexity jump-counting pattern (2019). https://www.plflib.org. Accessed 10 June 2021
5. Bentley, M.: The low complexity jump-counting pattern (2019). https://www.plflib.org. Accessed 10 June 2021
6. Boehme, D., et al.: Caliper: performance introspection for HPC software stacks. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2016, pp. 550–560 (2016). https://doi.org/10.1109/SC.2016.46
7. Technical Committee in-progress C++23 (2021). https://isocpp.org/std/the-standard. Accessed 14 June 2021
8. Dagum, L., Menon, R.: OpenMP: an industry-standard API for shared-memory programming. IEEE Comput. Sci. Eng. **5**(1), 46–55 (1998). https://doi.org/10.1109/99.660313
9. Edwards, H.C., Trott, C.R., Sunderland, D.: Kokkos: enabling manycore performance portability through polymorphic memory access patterns. J. Parallel Distrib. Comput. **74**(12), 3202–3216 (2014). https://doi.org/10.1016/j.jpdc.2014.07.003. Domain-Specific Languages and High-Level Frameworks for High-Performance Computing

10. The Alliance for Computing at Extreme Scale (ACES): Crossroads: a critical element for improved predictive capability (2021). https://www.lanl.gov/projects/crossroads. Accessed 14 June 2021
11. Message Passing Interface Forum: MPI: a message-passing interface standard. Technical report, USA (1994)
12. Holmen, J.K., Sahasrabudhe, D., Berzins, M.: A heterogeneous MPI+ PPL task scheduling approach for asynchronous many-task runtime systems. In: Proceedings of the Practice and Experience in Advanced Research Computing 2021 on Sustainability, Success and Impact (PEARC21). ACM (2021)
13. Ju, L., Ringler, T., Gunzburger, M.: Voronoi tessellations and their application to climate and global modeling. In: Lauritzen, P., Jablonowski, C., Taylor, M., Nair, R. (eds.) Numerical Techniques for Global Atmospheric Models. LNCSE, vol. 80, pp. 313–342. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-11640-7_10
14. Kaiser, H., Brodowicz, M., Sterling, T.: Parallex an advanced parallel execution model for scaling-impaired applications. In: 2009 International Conference on Parallel Processing Workshops, pp. 394–401 (2009). https://doi.org/10.1109/ICPPW.2009.14
15. Loiseau, J., et al.: FleCSPH: the next generation fleCSIble parallel computational infrastructure for smoothed particle hydrodynamics. SoftwareX **12**, 100602 (2020)
16. McCalpin, J.D.: Memory bandwidth and machine balance in current high performance computers. In: IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter, pp. 19–25, December 1995
17. Pérache, M., Carribault, P., Jourdren, H.: MPC-MPI: an MPI implementation reducing the overall memory consumption. In: Ropo, M., Westerholm, J., Dongarra, J. (eds.) EuroPVM/MPI 2009. LNCS, vol. 5759, pp. 94–103. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-03770-2_16
18. Plimpton, S.J., Hendrickson, B., Stewart, J.R.: A parallel rendezvous algorithm for interpolation between multiple grids. J. Parallel Distrib. Comput. **64**(2), 266–276 (2004). https://doi.org/10.1016/j.jpdc.2003.11.006
19. Ringler, T., Petersen, M., Higdon, R.L., Jacobsen, D., Jones, P.W., Maltrud, M.: A multi-resolution approach to global ocean modeling. Ocean Model. **69**, 211–232 (2013)
20. Ringler, T.D., Thuburn, J., Klemp, J.B., Skamarock, W.C.: A unified approach to energy conservation and potential vorticity dynamics for arbitrarily-structured c-grids. J. Comput. Phys. **229**(9), 3065–3090 (2010)
21. Thomas, J.: LINI and HPE to partner with AMD on El Capitan, projected as world's fastest supercomputer (2021). https://www.llnl.gov/news/llnl-and-hpe-partner-amd-el-capitan-projected-worlds-fastest-supercomputer. Accessed 06 June 2021
22. Warren, M.S., Salmon, J.K.: A parallel hashed oct-tree n-body algorithm. In: Proceedings of the 1993 ACM/IEEE Conference on Supercomputing, Supercomputing 1993, pp. 12–21. ACM, New York (1993). https://doi.org/10.1145/169627.169640
23. Williamson, D.L., Drake, J.B., Hack, J.J., Jakob, R., Swarztrauber, P.N.: A standard test set for numerical approximations to the shallow water equations in spherical geometry. J. Comput. Phys. **102**(1), 211–224 (1992)