FleCSI: Flexible Computational Science Infrastructure

Benjamin Bergen [1], Nick Moss[2], Irina Demeshko [3], Davis Herring[1], Marc Charest [4], Julien Loiseau [1], Navamita Ray [1], Jonathan Graham [1], Hartmut Kaiser [5], Li-Ta Lo [1], Karen Tsai [1], Charles Ferenbaugh [1], Richard Berger [1], John Wohlbier [6], Jonas Lippuner [2], Wei Wu [3], Andrew Reisner [1], Christoph Junghans [1], Scott Pakin [1], Brendan K. Krueger [1], Lukas Spies [7], Sumathi Lakshmiranganatha [1], Max Ortner [2], Pascal Grosset [1], David Gunter[2], Maxim Moraru [1], Galen Shipman[1], Jiajia Waters [1], Scot Halverson[3], Onur Çaylak [2], Peter Brady [1], Philipp V. F. Edelmann [1], Mason Delan[2], Brandon Keim [8], Christopher Malone[1], Alex Villa [9], Daniel Holladay [1], Dani Barrack [2], Nikunj Gupta [10], Ondřej Čertík[4], Robert Bird [2], and Melissa Rasmussen [11]

1 Los Alamos National Laboratory, USA **2** Independent researcher **3** NVIDIA, USA **4** Microsoft, USA **5** Louisiana State University, USA **6** Software Engineering Institute, USA **7** INRIA, France **8** University at Buffalo, USA **9** University of California, Merced, USA **10** Databricks, USA **11** Stony Brook University, USA

DOI: 10.xxxxxx/draft

Software

- Review
- Repository
- Archive

Editor:

Submitted: 05 August 2025
Published: unpublished

Summary

FleCSI (Bergen et al., 2021) is a modern C++ framework designed to support the development of multiphysics simulations. It provides a task-based programming model that unifies shared- and distributed-memory programming. FleCSI provides high performance, flexibility, and portability across heterogeneous computing architectures.

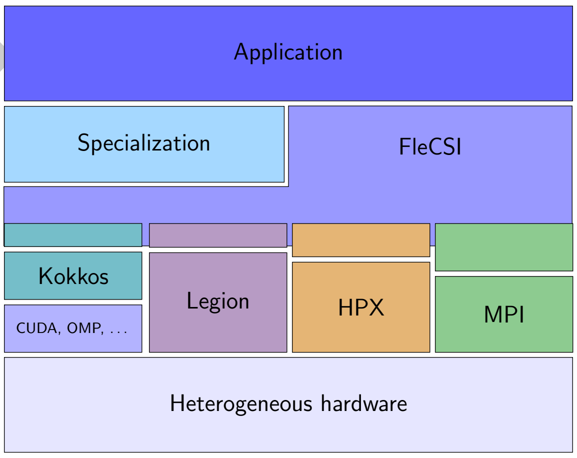

Figure 1: The FleCSI software ecosystem

Statement of need

FleCSI is designed to support the development of multiphysics simulations through a flexible, task-based programming model that enables performance portability across distributed, heterogeneous systems. It advances prior work by integrating dynamic task scheduling, data abstraction, and backend interoperability within a unified C++ framework. Compared to related systems like Uintah (Meng et al., 2012) and MPC (Pérache et al., 2008), FleCSI offers greater extensibility and finer runtime control, placing it at the intersection of portability, scalability, and modern software design for scientific computing.

Software description

FleCSI is designed to abstract away complexity while offering fine control for high-performance computing. The FleCSI runtime system manages initialization, execution, and shutdown. As presented in Figure 1, the FleCSI runtime supports backends such as Legion (Bauer et al., 2012), HPX (Kaiser et al., 2009, 2020), MPI (Message Passing Interface Forum, 2025), and Kokkos (Edwards et al., 2014), enabling code to remain portable across a variety of systems without manually handling the execution environment.

FleCSI's programming model is based on a hierarchy of parallelism: sequential, task-parallel, and data-parallel. The relationships among these is illustrated in Figure 2:

- **Control points** (CP) define an application's sequential backbone.
- **Actions** (A) specify a directed acyclic graph of high-level operations and their dependencies.
- **Tasks** are functions that operate on data distributed across address spaces.
- **Point tasks** (PT) are individual instances of a task that operate on a local fragment of a distributed data structure.
- **Kernels** (K) process a block of local data in a data-parallel fashion on a CPU or GPU.

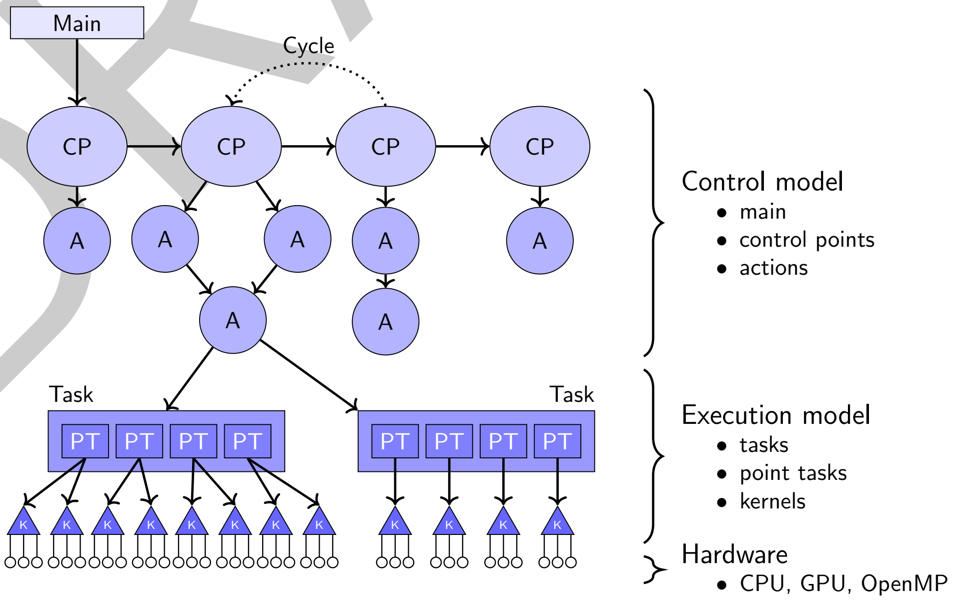

Figure 2: FleCSI control and execution models.

FleCSI's *control model* comprises control points and actions and determines what work is performed and in what order. FleCSI's *execution model*, comprising tasks, point tasks, and kernels, governs where and how that work actually runs. FleCSI's *data model*, not shown in Figure 2, governs how data are distributed and accessed.

Control model

Control points specify an application's sequential control flow and can include conditional branches. For example, one control point may represent "initialization", another "repetition until convergence", and a third "finalization".

Control points provide hooks for a directed acyclic graph (DAG) of *actions* to be attached. An action is a sequential function that defines an application's core numerics or physics routines such as "hydrodynamics" or "viscosity". A new action can be incorporated into an application by specifying its direct dependents and dependencies (control points or other actions). For example, if an existing application defines a "solver" action, a new developer later can create a "preconditioner" action and insert it before the solver in the DAG without having to modify any other code or interfaces. By walking the DAG in topological order, FleCSI ensures a valid program execution.

Execution model

Actions spawn *tasks*, which are functions that are distributed within and across the nodes of the compute cluster and that complete asynchronously. In a computational-science application, a task typically represents updates to a data structure, such as to perform mesh stiffening and relaxing. A task declaration includes the *fields* of a distributed data structure that it will access (e.g., the cells, edges, and vertices of an unstructured mesh) and the access rights it requires on each field: read only, write only, or read/write. Tasks are run concurrently according to field data dependencies. For example, if task A reads x and writes y, task B reads x and writes z, and task C reads y and writes w, then the FleCSI runtime will execute tasks A and B concurrently but require that task A finish before task C can start.

Execution is distributed across logical units called *colors*. Colors are analogous to MPI ranks but do not need to map 1:1 to processes. Rather, the application chooses an appropriate number of colors for each task launch. If colors outnumber processes then some processes simply handle more than one color. Each color is handled by exactly one *point task*—an individual instance of a task. While point tasks are executed on CPUs, the data for readable fields are preloaded into a specified memory space (CPU NUMA domain or GPU device memory), and the data for writable fields automatically will be communicated to dependent tasks.

Point tasks process their data by launching data-parallel *kernels* that operate on the memory space in which the field data was placed. In a computational-science application, these typically perform element updates such as incrementing position, momentum, energy, etc. Kernel can execute in parallel on GPUs, in parallel on CPUs (using OpenMP threads), or serially on CPUs. Kernel code is portable across these three forms of execution; no code modifications are needed to dispatch a kernel to a CPU versus a GPU.

Data model

FleCSI provides several topology types—skeletons of distributed data structures—that applications use to represent physical quantities and their relationships:

- `topo::unstructured` supports graph-based meshes and is suitable for finite element or finite volume methods.
- `topo::narray` provides structured n-dimensional grids with support for boundary conditions and periodicity, making it ideal for Eulerian hydrodynamics.
- `topo::ntree` organizes data in a hashed tree structure that enables fast neighbor searches and is appropriate for particle-based simulations and adaptive mesh refinement.

Although topology data are distributed, all communication and synchronization is implicit and is based on the access rights associated with each field. (See Execution model above.) Fields can be defined with several layouts such as dense (arrays), ragged (vectors), sparse (maps), or particle (buffers).

Acknowledgments

The FleCSI project is supported by the U.S. Department of Energy through Los Alamos National Laboratory (LANL). Los Alamos National Laboratory is operated by Triad National Security, LLC, for the National Nuclear Security Administration of the U.S. Department of Energy (contract no. 89233218CNA000001). This paper has been assigned a Los Alamos Unlimited Release number of LA-UR-25-25479.

The work reported in this paper would not have been possible without close collaborations with the Legion and HPX teams and LANL's Ristra project, FleCSI's initial "customer".

References

Bauer, M., Treichler, S., Slaughter, E., & Aiken, A. (2012). Legion: Expressing locality and independence with logical regions. *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, 1–11. https://doi.org/10.1109/SC.2012.71

Bergen, B., Demeshko, I., Ferenbaugh, C., Herring, D., Lo, L.-T., Loiseau, J., Ray, N., & Reisner, A. (2021). FleCSI 2.0: The flexible computational science infrastructure project. *European Conference on Parallel Processing*, 480–495. https://doi.org/10.1007/978-3-031-06156-1_38

Edwards, H. C., Trott, C. R., & Sunderland, D. (2014). Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *Journal of Parallel and Distributed Computing*, *74*(12), 3202–3216. https://doi.org/10.1016/j.jpdc.2014.07.003

Kaiser, H., Brodowicz, M., & Sterling, T. (2009). ParalleX: An advanced parallel execution model for scaling-impaired applications. *2009 International Conference on Parallel Processing Workshops*, 394–401. https://doi.org/10.1109/icppw.2009.14

Kaiser, H., Diehl, P., Lemoine, A. S., Lelbach, B. A., Amini, P., Berge, A., Biddiscombe, J., Brandt, S. R., Gupta, N., Heller, T., Huck, K., Khatami, Z., Kheirkhahan, A., Reverdell, A., Shirzad, S., Simberg, M., Wagle, B., Wei, W., & Zhang, T. (2020). HPX—the C++ standard library for parallelism and concurrency. *Journal of Open Source Software*, *5*(53), 2352. https://doi.org/10.21105/joss.02352

Meng, Q., Humphrey, A., & Berzins, M. (2012). The Uintah framework: A unified heterogeneous task scheduling and runtime system. *2012 SC Companion: High Performance Computing, Networking, Storage and Analysis (SCC)*, 2441–2448. https://doi.org/10.1109/SCC.2012.6674233

Message Passing Interface Forum. (2025). *MPI: A message-passing interface standard version 5.0*. https://www.mpi-forum.org/docs/mpi-5.0/mpi50-report.pdf

Pérache, M., Jourdren, H., & Namyst, R. (2008). MPC: A unified parallel runtime for clusters of NUMA machines. *Euro-Par 2008 – Parallel Processing*, *5168*, 78–88. https://doi.org/10.1007/978-3-540-85451-7_9