# Mint: A Performance Portable Mesh Data Model Enabling the Development of Next-Generation Massively Parallel Multi-Physics Applications

George Zagaris*, Kevin Holst†, Jay Sitaraman‡, Shirzad Hosseinverdi§, William Tyson¶, Vinod Lakshminarayan‖, Andrew Bodling**, Taylor Erwin††, Patrick McNally‡‡, Steven E. Lamberson§§, Ryan Bond¶¶, Andy Wissink***
*DoD HPCMP CREATE$^{TM}$-AV*

Keith Obenschain†††,
*Naval Research Laboratory, Washington, DC 20375*

**The increasingly diverse hardware architecture landscape that characterizes modern high-performance scientific computing presents significant software engineering challenges for the development of large-scale multi-physics codes. Performance portability libraries, such as RAJA and Kokkos, have emerged as powerful tools that help mitigate some of these challenges. However, leveraging these libraries for large-scale multi-physics applications requires substantial effort, refactoring and software engineering expertise. This paper introduces *Mint*, a C++ library that serves as a foundational software infrastructure, consisting of a comprehensive mesh data model and a mesh-aware, fine-grained parallel execution model that underpins the development of massively parallel and performance portable multi-physics applications. *Mint* enables seamless integration with RAJA and Kokkos and can be extended to support other performance portability libraries and approaches. As a foundational building block, *Mint* offers a flexible and sustainable approach that streamlines the development of next-generation, multi-physics applications while insulating application developers, scientists and engineers from the complexities of emerging software technologies and diverse hardware architectures. We describe the software architecture of *Mint* and present performance results from benchmark applications that demonstrate the feasibility of our approach.**

## I. Introduction

Over the past decade, developing applications for High-Performance Computing (HPC) has become increasingly challenging due to the rapid pace of disruptive changes in HPC architecture, growing hardware diversity [1, 2], the shift in programming paradigm, and the proliferation of programming models and software technologies that have emerged to address these changes. The increasing demand for better performance and higher throughput, coupled with the requirement to maintain power consumption of HPC systems at sustainable levels, has driven HPC architecture design towards heterogeneous systems [3]. Heterogeneous systems combine conventional multi-core CPUs with specialized accelerators, such as GPUs, that collectively expose thousands of cores to the application within a single compute node. Harnessing the massive parallelism of heterogeneous systems requires a shift to a fine-grained parallel programming

*HPCMP CREATE™-AV Kestrel, Computer Scientist, Air Force Research Laboratory, Wright-Patterson AFB, Dayton, OH 45433
†HPCMP CREATE™-AV Kestrel, Assistant Research Professor, University of Tennessee, Knoxville, TN 37996
‡HPCMP CREATE™-AV Helios Principal Developer, U.S. Army Combat Capabilities Development Command Aviation & Missile Center, Moffett Field, CA, 94035
§HPCMP CREATE™-AV Helios, Research Scientist, Science & Technology Corp., NASA Ames Research Center, Moffett Field, CA 94035
¶HPCMP CREATE™-AV Kestrel, Aerospace Engineer, Naval Air Warfare Center Aircraft Division, Patuxent River, MD 20670
‖HPCMP CREATE™-AV Helios, Aerospace Engineer, Whisper Aero Inc., Crossville, TN 38555
**HPCMP CREATE™-AV Helios, Research Scientist, Science & Technology Corp., NASA Ames Research Center, Moffett Field, CA 94035
††HPCMP CREATE™-AV Kestrel, Computational Scientist, Oak Ridge National Laboratory, Oak Ridge, TN 37830
‡‡HPCMP CREATE™-AV Kestrel Lead Architect, U.S. Army Research Engineer Research & Development Center(ERDC), Vicksburg, MS 39180
§§HPCMP CREATE™-AV Kestrel Principal Developer, Bowhead Total Enterprise Solutions LLC, Springfield, VA 22150
¶¶HPCMP CREATE™-AV Kestrel, University of Tennessee, Knoxville, TN 37996
***HPCMP CREATE™-AV Program Manager, U.S. Army Combat Capabilities Development Command, Aviation & Missile Center, Moffett Field, CA, 94035
†††DoD HPCMP Chief Technology Officer, Naval Research Laboratory, Washington, DC, USA

paradigm for on-node parallel execution. However, transitioning to this new programming paradigm in a portable manner and across a diverse set of architectures, presents significant software engineering challenges.

Currently, nine of the ten most powerful HPC systems in the world are GPU-based heterogeneous systems [4]. While NVIDIA GPUs were dominant in the pre-exascale era with systems like *Sierra* [5] and *Summit* [6], the first exascale system, *Frontier* [7], employs AMD CPUs and AMD MI250X GPUs. Similarly, *Aurora* [8], combines Intel CPUs with Intel GPUs. Emerging hardware trends favor integrated chiplet designs that unify the CPU and GPU on a single chip. NVIDIA's Grace-Hopper and Grace-Blackwell Superchips integrate NVIDIA's ARM-based Grace CPU with Hopper and Blackwell GPUs, respectively, connected via the NVLINK chip-to-chip interconnect. Likewise, AMD's MI300A Accelerated Processing Unit (APU) architecture [9], which debuted in the *El Capitan* system [10], integrates AMD CPUs and GPUs with High-Bandwidth Memory (HBM) on a single integrated chip.

The widespread adoption of GPU-based heterogeneous systems and the rapid evolution of increasingly diverse hardware architectures present significant software engineering challenges. Developers are faced with the challenge of developing applications that are efficient and portable across a wide array of platforms, ranging from laptops and high-end workstations, which may also be equipped with GPU accelerators, to commodity clusters, supercomputers and advanced heterogeneous systems. Achieving this level of performance portability demands:

1) Navigating the complex ecosystem of programming models and software technologies such as, *CUDA* [11], *rocM* [12], *HIP* [13], *SYCL* [14], *OpenMP* [15] and *OpenACC* [16].
2) Developing an understanding of the unique programming idioms, performance characteristics, trade-offs and platform-specific limitations of each programming model, and
3) Carefully devising a performance portability strategy that balances performance, development overhead and maintenance, tailored according to the application requirements.

However, the sheer size and complexity of developing large-scale, production, multi-physics applications impose additional constraints and requirements on an overarching performance portability strategy:

1) **Single-Source Codebase.** Large-scale multi-physics codes can consist of millions of lines of code, with no single *hotspot* or *kernel* dominating the execution time. Maintaining separate versions of the codebase for each architecture is both impractical and unsustainable. Therefore, a unified, single-source codebase that ensures parallelism and portability across diverse architectures is essential [17–20].
2) **Ease of Maintenance and Continuous Development.** The lifetime of multi-physics codes spans several decades, requiring significant investments in on-going research, development, verification and validation. As the code evolves in response to the changes in HPC architectures, it must remain maintainable and extensible, supporting the addition of new algorithms and modeling capabilities to meet the evolving programmatic needs and mission requirements.
3) **Long-Term Sustainability.** The rapid evolution of hardware and software technologies demands a strategy that is resilient to disruptive changes and capable of spanning across multiple platform generations. Leveraging widely supported portable programming models and abstractions reduces the likelihood of costly refactoring and ensures the long-term sustainability of the application.
4) **Adaptable and Extensible to Future Technologies.** The long service lives of large-scale multi-physics applications requires a strategy that is future-proof, capable of adapting to hardware advancements, software innovations, emerging programming models and new programming language features.

The impetus to maintain a unified, single-source codebase has prompted the development of performance portability libraries such as *Kokkos* [21–23], *RAJA* [24], *Alpaca* [25], and *OCCA* [26]. These libraries serve as essential building blocks for performance portability, abstracting the complexities of the low-level programming models and the hardware-specific details of the corresponding architectures. However, selecting the most suitable approach is neither clear nor straightforward due to the range of available options, each with its own strengths, weaknesses, and developer community support. Furthermore, leveraging these libraries for the development of large-scale multi-physics applications requires substantial effort, refactoring and software engineering expertise.

This paper introduces *Mint*, a C++ library that serves as a foundational software infrastructure for the development of large-scale, distributed and massively parallel multi-physics applications. *Mint* provides a comprehensive mesh data model and a mesh-aware, fine-grained parallel execution model that seamlessly integrates with *RAJA* [24] and *Kokkos* [21–23] while remaining extensible to support other performance portability libraries and approaches. This flexibility ensures that applications are not tied to a specific software stack or implementation. As a foundational building block, *Mint* provides a flexible and sustainable infrastructure that streamlines the development of next-generation, multi-physics applications while insulating application developers, scientists and engineers from the complexities of emerging software technologies and diverse hardware architectures.

The remainder of this paper is organized as follows. **Section II** provides a concise overview of the background and related work. **Section III** details the core features, capabilities and software architecture of *Mint*. Performance results from benchmark applications, highlighting the feasibility and effectiveness of our approach are presented in **Section IV**. Finally, **Section V** concludes the paper with a summary and closing remarks.

## II. Background & Related Work

This section provides an overview of the prominent approaches for fine-grained parallelism on heterogeneous systems. The advantages and disadvantages of each approach are discussed, drawing comparisons and contrasts between them. In addition, a review of mesh discretization libraries that share similarities with *Mint* is provided, highlighting some of the unique aspects and distinguishing characteristics of this work.

### A. Prominent Approaches for Fine-Grained Parallelism on Heterogeneous Systems

Effective utilization of heterogeneous systems requires application developers to shift to a new programming paradigm. In addition to exploiting coarse-grained, distributed parallelism with MPI [27] on the CPU cores across the nodes of the system, developers must adapt their algorithms and data structures for fine-grained on-node parallel execution to harness the massive parallelism offered by the accelerators, as illustrated in Figure 1. This requires the use of the MPI+X approach, where "X" represents the programming model used to offload computation on the accelerators for on-node, fine-grained parallel execution, which can vary depending on the target architecture. For example, CUDA [11] may be used for NVIDIA GPUs, while HIP [13] is used with AMD GPUs.

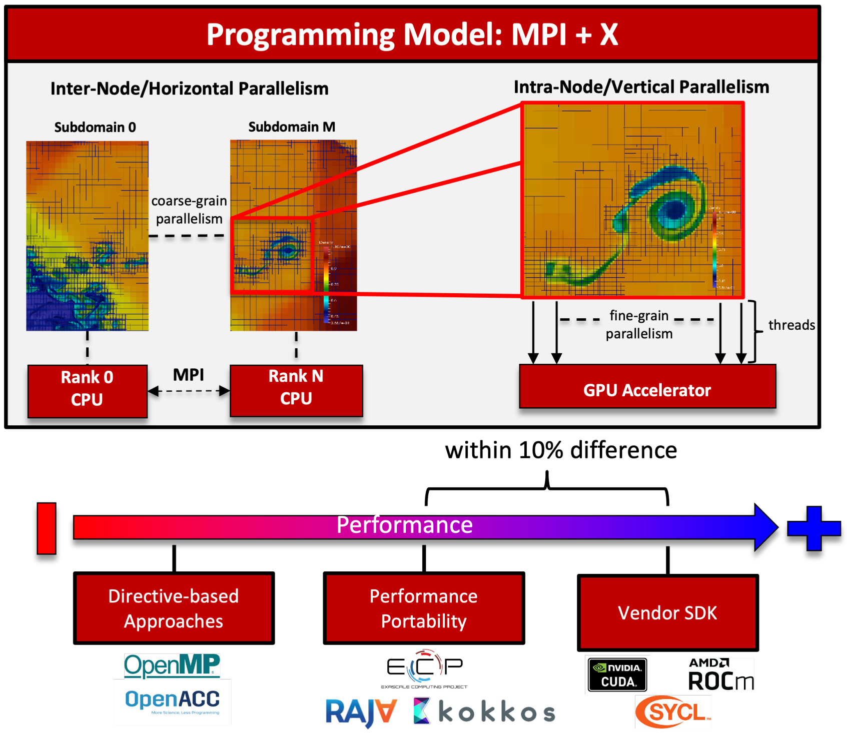


**Fig. 1 The MPI+X approach for heterogeneous systems. MPI [27] handles coarse-grained distributed parallelism across the nodes, while fine-grained, on-node parallelism offloads computation to accelerators using a suitable programming model for the target architecture. Vendor SDKs deliver the best performance but risk vendor lock-in. Directive-based approaches such as *OpenMP* [15], offer portability, but, generally do not provide the best performance. Performance portability libraries such as *Kokkos* [23] and *RAJA* [24] offer a compelling alternative for C++ applications, achieving performance that is within 10% of the vendor SDKs.**

Using a vendor SDK approach for fine-grained parallelism, such as *CUDA* [11] for NVIDIA GPUs can typically deliver the best performance, but this approach is subject to vendor lock-in, which would require application developers to maintain multiple versions of the code for each architecture. Directive-based approaches, such as OpenMP [15] and OpenACC [16], provide a portable programming approach, but, heavily rely on the compiler and generally do not achieve the desirable levels of performance.

The *Thrust library* [28], is one of the earliest library approaches focused on promoting developer productivity and application performance. It has proven its effectiveness in large-scale multi-physics applications [29] and, through the *rocThrust port* [30], it can be utilized to target AMD GPUs. However, to the best of our knowledge, a port for Intel GPUs is not currently available.

OCCA [26] is another library approach. It defines the OCCA Kernel Language (OKL) that enables the creation of portable device kernels using a directive-based extension to the C-language. OCCA has demonstrated good performance and support for multiple backends in large-scale multi-physics applications [31]. However, the OKL syntax may be a bit obscure for some developers, which can hinder adoption, as well as, make debugging and fine-tuning of kernels more challenging due to difficulties in utilizing existing debugger and performance profiling tools.

Performance portability libraries such as *Kokkos* [23] and *RAJA* [24] offer a compelling alternative for C++ applications, delivering performance that is within 10% of vendor SDKs. Both Kokkos and RAJA are C++ libraries that provide abstractions for high-level parallel constructs via templated functions. Developers compose their algorithms using these constructs and specify the target execution space (e.g., CUDA, HIP or other). The libraries then dispatch the user-supplied code to the appropriate device backend, translating it using the corresponding programming model for the specified execution space.

SYCL [14] is another library that shares common goals with RAJA and Kokkos. However, whereas RAJA and Kokkos development is driven primarily by the HPC community, SYCL has a much broader charter and consequently limits itself to a smaller feature set. For a more detailed comparison and discussion on SYCL see reference [32].

## B. Mesh and Discretization Libraries

A robust and efficient distributed mesh representation plays a crucial role as a fundamental infrastructure component for multi-physics codes. Over time, several libraries have emerged as valuable building blocks to streamline the development of large-scale multi-physics applications. While conducting a comprehensive literature survey of all the contributions in this domain exceeds the scope of this paper, we provide an overview of some notable works in this area.

### *1. Mesh Infrastructure Libraries*

The Mesh Toolkit (MSTK) [33, 34] and Parallel Unstructured Mesh Infrastructure (PUMI) [35] provide a flexible infrastructure for management of distributed unstructured meshes including core operations such as partitioning and ghost communication. However, they do not provide support for a structured mesh representation or fine-grained parallelism for heterogeneous systems.

The Mesh Oriented Database (MOAB) [36] provides a more generic mesh data management system that can handle both structured and unstructured mesh representations. However, like PUMI and MSTK, MOAB does not provide any support for fine-grain parallelism for heterogeneous systems.

### *2. Block-Structured AMR Frameworks*

A number of libraries have been developed for Block-Structured AMR applications. AMReX [37] is a modern block-structured AMR framework for patch-based AMR. It provides a homegrown performance portability layer that uses CUDA [11] for NVIDIA GPUs, HIP [13] for AMD GPUs and SYCL [14] for Intel GPUs.

Similar to AMReX, the Structured Adaptive Mesh Refinement Application Infrastructure (SAMRAI) [38, 39] provides an object oriented framework for large-scale, patch-based AMR applications. SAMRAI employs RAJA [24] for fine-grained parallelism and Umpire [40] for memory management on heterogeneous systems.

Parthenon [41] provides a cartesian AMR framework that shares similarities with AMReX in terms of data containers and parallel regions. However, instead of a homegrown performance portability layer, it adopts Kokkos [22] as its performance portability backend.

### *3. Finite Element Discretization Libraries*

For Finite Element applications, libMesh [42] provides a powerful framework that serves as the basis for many applications, targeting multi-scale, multi-physics finite element simulations. It offers a comprehensive set of finite element basis functions and quadrature rules and leverages PETSc [43] for its linear solvers. However, it does not have any support for structured mesh representations or fine-grained parallelism for heterogeneous systems.

A fairly recent development in this area is the ELEMENTS [44] library. ELEMENTS is a C++ high-order finite element library that serves as a research tool for both continuous and discontinuous finite element methods. Although, it does not support structured mesh representations, it adopts Kokkos [22] as its performance portability backend.

MFEM [45] provides a modular, massively parallel and scalable software library for arbitrary high-order finite element applications. It provides interfaces to various other solver libraries including PETSc [43] and Hypre [46]. Since version 4.0, MFEM supports GPU acceleration through a flexible interface that can incorporate multiple backends, including CUDA [11], OCCA [26], RAJA [24] and OpenMP [15].

While existing technologies are tied to a particular mesh representation, discretization method and software stack for performance portability, key distinctions and a differentiating characteristic of *Mint* and the work presented herein are:

1) *Mint* aims to provide a comprehensive mesh data model that supports structured, unstructured and particle-based (i.e., *meshless*) representations.
2) *Mint* provides a foundational infrastructure that is agnostic to the physics and numerical discretization method used by the application.
3) *Mint* provides an execution model that serves as an intermediate portability layer, seamlessly integrating with *RAJA* [24] and *Kokkos* [21–23] while remaining extensible to support other performance portability libraries and approaches. This flexibility ensures that applications are not tied to a specific software stack or implementation.

Recent results underscore the feasibility of our approach. Hosseinverdi et al. [47] detail the development of a performance portable Unstructured Reynolds-Averaged Navier-Stokes (RANS) flow solver that is leveraging *Mint*. Similarly, Bodling et al. [48] reported preliminary progress on the development of a Mint-based performance portable RANS flow solver for structured, body-fitted grids. Furthermore, Sitaraman et al. [49] highlight the use of *Mint* for the development of a performance portable overset grid assembly code, PUNDIT-G, used in the DoD HPCMP CREATE$^{TM}$-AV codes to couple the near-body and off-body Cartesian AMR solvers targeting moving body calculations.

## III. Software Architecture Overview of Mint

*Mint* is a C++ library, developed under the DoD HPCMP CREATE$^{TM}$ program that provides a comprehensive mesh data model, depicted in Figure 2, and a mesh-aware, fine-grain, parallel execution model that underpins the development of next-generation, massively parallel, multi-physics applications, targeting new and emerging architectures.

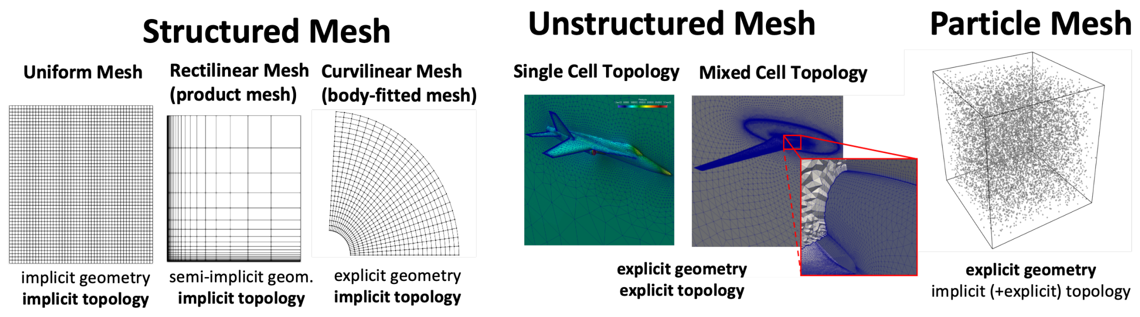


**Fig. 2 Mint Mesh Data Model. Mint provides a comprehensive mesh data model, supporting structured, unstructured and particle-based mesh representations. The model organizes the mesh into distinct mesh types based on the representation of the geometry and topology of the mesh.**

*Mint* serves as a foundational infrastructure designed to streamline the development of large-scale, multi-physics applications. The key features that *Mint* provides include the following:

1) Support for 1D, 2D, and 3D mesh geometry
2) Efficient data-structures for structured, unstructured and particle mesh representations.
3) Support for common cell types and an extensible infrastructure to add new cell types.
4) Mesh Partitioning/Re-partitioning and Ghost Cell Communication
5) Finite Element shape functions for each supported cell type and corresponding Gauss-Quadratures.
6) Parallel Mesh I/O

7) A mesh-aware execution model that supports on-node, fine-grain parallelism, enabling the implementation of performance portable numerical schemes and computational kernels that are born parallel and portable.

The two main components that support the mesh data model representation are: (a) the Memory Management Substrate and (b) the Parallel Execution Model. These two components are discussed in the following sections in more detail.

## A. Memory Management Substrate

*Mint* provides a light-weight central memory management substrate, consisting of a small set of functions for managing memory in different memory spaces. These operations are facilitated by leveraging the Umpire library [40], which provides functionality to conveniently manage memory resources on heterogeneous platforms.

Memory spaces are represented via an integer handle, which can be queried using the following functions:

**Listing 1 Mint's Memory Space Query Methods**

```
int getHostMemorySpace();
int getDeviceMemorySpace();
int getUnifiedMemorySpace();
int getConstantDeviceMemorySpace();
int getPinnedMemorySpace();
```

The memory space can then be passed as an argument to `mint::allocate` to allocate memory in a particular memory space and perform other operations, such as copy the memory to a different memory space, etc. The code snippet below illustrates some of these operations:

**Listing 2 Mint's Memory Management Operations**

```
 // allocate array of 10 integers in global device memory
 int* A_d = mint::allocate< int >(10, mint::getDeviceMemorySpace());
 assert(A_d != nullptr);

 // allocate array of 10 integers on the host
 // Operation defaults to host if 2nd argument for memspace is not specified
 int* A_h = mint::allocate< int >(10);
 assert(A_h != nullptr);

 // copy the data from device to host
 mint::memCopy(A_h, A_d, 10*sizeof(int));

 // de-allocate memory, supplied pointer is set to nullptr on retrun
 mint::deallocate(A_h);
 mint::deallocate(A_d);
```

In Mint-based applications, allocation and de-allocation of memory for C++ objects is accomplished using C++ `new` and `delete` operations. However, memory managements for all array data employs Mint's central memory management substrate component, as discussed in this section.

## B. Fine-Grained Parallel Execution Model

Mint's fine-grained parallel execution model serves as an intermediate layer that provides an *application-oriented*, *mesh-aware* and *physics-focused* API that insulates computational scientists and engineers from the programming details and nuances of performance portability. The fine-grained parallel execution model can be compiled with either RAJA [24] or Kokkos [22] as the backend performance portability library. Kernels in Mint are defined either as functors or C++ lambda closures. When a Kernel is launched, *Mint* forwards the kernel to the performance portability backend to launch on the device using an appropriate device backend, e.g., CUDA for NVIDIA GPUs, or, HIP for AMD GPUs.

The fine-grained parallel execution model, depicted in Figure 3, is built upon a set of data-parallel primitive operations. These foundational operations include a generic loop traversal (`for_all`), parallel scan, sort, reductions and

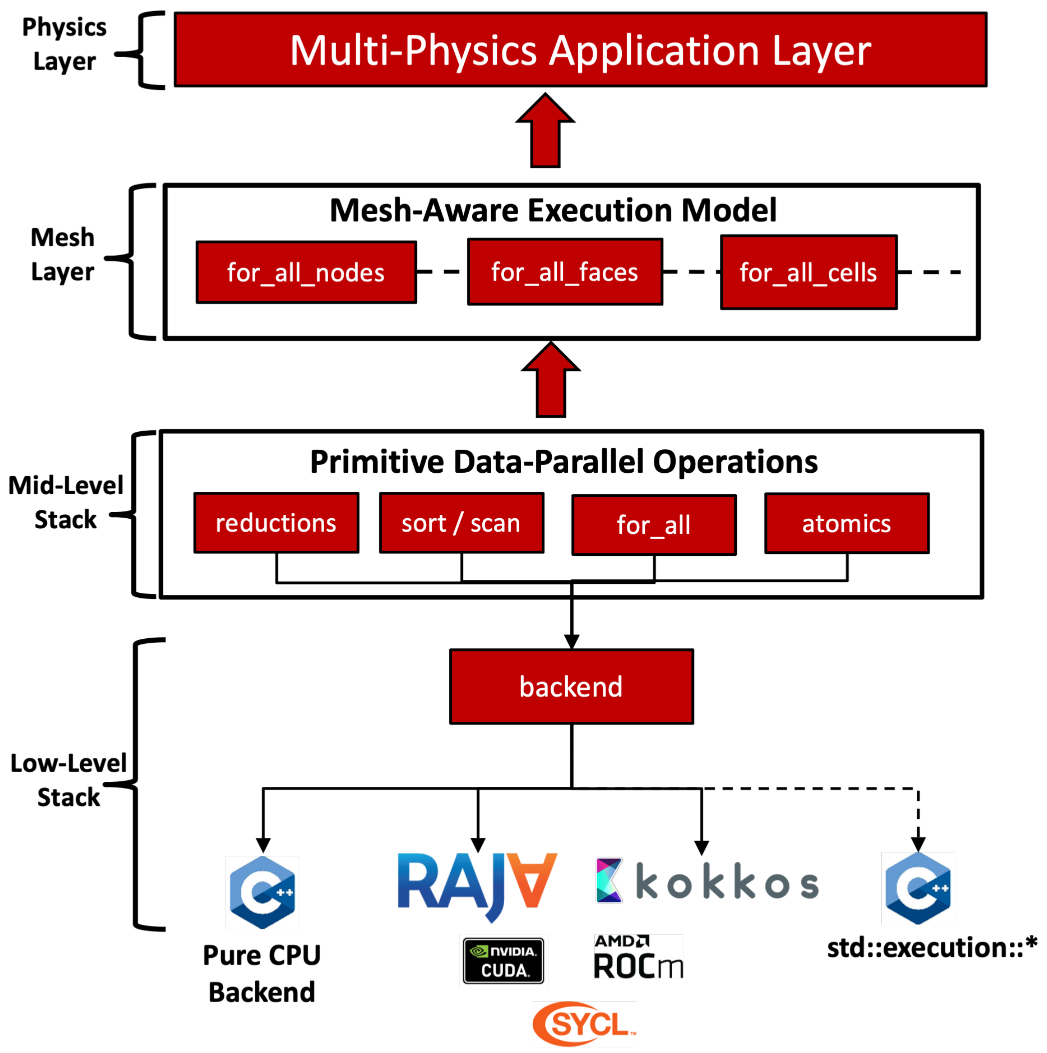


**Fig. 3 Fine-Grained Parallel Execution Model. Built upon a set of data-parallel primitive operations, which include, a generic loop traversal (`for_all`), parallel scan, sort, reductions and atomic operations. Leveraging these foundational operations the mesh-aware execution model defines mesh traversal operators that can be applied on `mint::Mesh` object. Developers use these operations to compose the physics algorithms at the application layer, thereby, insulate the physics application from the details of the underlying software technologies and hardware architectures.**

atomic operations. Leveraging these primitives, the mesh-aware execution model then defines a set of mesh traversal operations that can be applied on a given mesh object. The mesh traversal operations support looping over constituent entities of the mesh such as, the mesh nodes, faces or cells. Developers compose the physics algorithms using the mesh traversal and data-parallel primitive operations. This stratified approach provides a number of benefits to the application:

1) Insulates the physics algorithms at the application level from any API changes at the low-level and from the details and complexities of the underlying software technologies and hardware architectures.
2) Simplifies the use of RAJA or Kokkos and enhances their adoption by application developers.
3) Provides built-in flexibility towards adoption of emerging technologies and new programming language features in the future (e.g., C++ `std::execution::*`)

*1. Execution Space*

The Execution Space defines where and how a kernel is executed. Both the data-parallel primitive operations and the mesh traversal operations can be called by specifying the execution space either at compile-time, by specifying the execution space as the first template argument of the function, or at runtime, by passing a `mint::ExecutionSpace` object as the second function argument, after the name of the kernel.

The list of currently supported execution spaces is summarized in Table 1.

| **Execution Space** | **Description** | **Availability** |
|---|---|---|
| `SEQ_EXEC` | Sequential execution on the CPU | Always |
| `OMP_EXEC` | OpenMP execution on the CPU | When compiled with OpenMP |
| `CUDA_EXEC<int BLOCKSIZE>` | Parallel execution on NVIDIA GPUs (synchronous, CPU blocks) | When compiled with CUDA |
| `CUDA_EXEC<int BLOCKSIZE, ASYNC>` | Parallel execution on NVIDIA GPUs (asynchronous – CPU does not block) | When compiled with CUDA |
| `HIP_EXEC<int BLOCKSIZE>` | Parallel execution on AMD GPUs (synchronous, CPU blocks) | When compiled with HIP |
| `HIP_EXEC<int BLOCKSIZE, ASYNC>` | Parallel execution on AMD GPUs (asynchronous – CPU does not block) | When compiled with HIP |

**Table 1 Summary of available Executions Spaces**

To illustrate the mechanics of how to use the execution space, consider the following *DAXPY* kernel that sets the execution space at compile-time:

**Listing 3 DAXPY Kernel example Using An Execution Space set at Compile-Time**

```
// set the execution space to device at compile time
using EXEC = mint::CUDA_EXEC< 256, ASYNC >;

// launch the DAXPY kernel
mint::for_all< EXEC >( "daxpy", N, MINT_LAMBDA(const mint::index_t i) {
  y[ i ] +=  alpha * x[ i ];
} );
```

The equivalent version of the *DAXPY* kernel with an execution space that is set at run-time is shown below:

**Listing 4 DAXPY Kernel example Using An Execution Space set at Runtime**

```
// create an ExecutionSpace object
mint::ExecutionSpace exec;

// set the execution space to device at runtime
exec.useDeviceAsync();

// launch the DAXPY kernel
mint::for_all( "daxpy", exec, N, MINT_LAMBDA(const mint::index_t i) {
  y[ i ] +=  alpha * x[ i ];
} );
```

*2. Mesh Traversal Operations*

Building on top of the generic loop traversal operator (i.e., `for_all`), which was illustrated in the previous section, the mesh-aware execution model defines a set of mesh traversal operations that operate on a given mesh object.

These traversal routines provide an additional layer of abstraction that hides the details of the underlying mesh type, data-structure and organization and streamline the development of mesh-based algorithms in scientific computing applications.

The list of available mesh traversal operations is summarized in Table 2.

| **Traversal Operation** | **Description** |
|---|---|
| `for_all_nodes` | Loop over the mesh nodes |
| `for_all_faces` | Loop over the mesh faces |
| `for_all_cells` | Loop over the mesh cells |

**Table 2 Summary of available Executions Spaces**

Similar to the generic `for_all` traversal operator, the execution space for the mesh traversal operations may also be prescribed either at compile-time, as the first template argument to the function, or at runtime, by passing a `mint::ExecutionSpace` object as the second function argument, after the name of the kernel. In addition, the mesh traversal operations can optionally take an additional template argument that indicates any extra mesh information that needs to be supplied to the user-specified kernel at runtime. These optional arguments are organized under the `mint::xargs::*` namespace and are summarized in Table 3.

| **Additional Template Argument** | **Description** | **Usage** |
|---|---|---|
| `mint::xargs::x` | Supply the node's x-coordinate | Use with node-traversals on 1D meshes |
| `mint::xargs::xy` | Supply the node's x,y coordinates | Use with node-traversals on 2D meshes |
| `mint::xargs::xyz` | Supply the node's x,y,z coordinates | Use with node-traversals on 3D meshes |
| `mint::xargs::nodeids` | Supply the cell or face node IDs | Use with face or cell traversals |
| `mint::xargs::coords` | Supply the cell or face node coordinates | Use with face or cell traversals |
| `mint::xargs::faceids` | Supply the IDs of constituent cell faces | Use with cell traversals |
| `mint::xargs::facenbrs` | Supply immediate cell face neighbors (FN1) | Use with cell traversals |
| `mint::xargs::cellids` | Supply the IDs of the abutting cell IDs | Use with face traversals |

**Table 3 Summary of optional template arguments to Mesh Traversal routines.**

The code snippet depicted in 5 shows an example 2D Flux update kernel that illustrates the usage Mint's mesh-aware execution model. The kernel loops over the cells of the mesh. For each cell, the flux contributions from the constituent cell faces are accumulated and stored as a cell-centered value. Since, the kernel requires the face IDs as an additional argument, `mint::xargs::faceids` is supplied as a template argument when the `for_all_cells` is called. The execution space is selected at runtime specified by supplying a `mint::ExecutionSpace` object as the second argument to the `for_all_cells` function call. Inside the kernel, there are various memory reads and writes of array variables that are captured by the C++ lambda capture. The kernel requires that the captured array pointers all point to a memory space that is accessible in the specified execution space, e.g., device memory.

The Mesh traversal Operators, in conjunction with the data-parallel primitives, memory management API and execution space definitions are used to compose the kernels for the physics algorithms. This approach enables development of single-source, performance portable code. The next section provides numerical results from various applications highlighting the feasibility and effectiveness of our approach.

**Listing 5 Sample 2D Flux Update Kernel**

```
// create an execution space object
mint::ExecutionSpace exec;

// select the execution space at runtime
exec.useDeviceAsync();

// grab pointer to the cell-centered flux field on the mesh
double* flux = mesh->getFieldPtr< double >( "flux", CELL_CENTERED );

mint::for_all_cells< mint::xargs::faceids >(
 "flux_update",
 exec,
 mesh,
 MINT_LAMBDA( int iCell, const int* faceIDs, const int nFaces ) {
  double iflux = 0.;
  MINT_PRAGMA_UNROLL(3)
  for ( mint::index_t iface=0; iface < nFaces; ++iface )
  {
    mint::index_t faceIdx = faceIDs[ iface ];
    double mult = diff_face[ faceIdx ] * area[ faceIdx ] * sn(iCell, iface);
    iflux += mult * gradfx[ faceIdx ] * nx[ faceIdx ];
    iflux += mult * gradfy[ faceIdx ] * ny[ faceIdx ];
  }
  flux[ iCell ] = iflux;
} );
```

## IV. Performance Results

This section presents numerical results from various benchmarks that illustrates the feasibility of our approach. The performance results presented in this section were obtained using DoD HPCMP computational resources. The node architecture of the platforms that were used is summarized in Table 4.

| Platform | CPU Sockets / Node | GPUs / Node |
|---|---|---|
| Narwhal [50] | 2 AMD EPYC Rome (128 cores) | 2 NVIDIA V100 GPUs (1 GPU per socket) |
| Nautilus [51] | 2 AMD EPYC Milan (128 cores) | 4 NVIDIA A100 GPUs (2 GPUs per socket) |
| Ruth | 1 AMD EPYC Trento (64 cores) | 4 AMD MI250X GPUs |

**Table 4 Platforms used for Evaluating Performance**

### A. Performance Comparison for Steady-State Calculations Over an Unstructured Sphere Mesh

This section presents results using a mint-based mint-app, *MCFD*, that solves the laminar Navier-Stokes equations on an unstructured mesh of a sphere configuration. Figure 4 shows the steady-state solution and convergence plot for the 10K-cell and 1.6M-cell sphere mesh configuration respectively.

The performance of *MCFD* is compared to FVSAND [52], which provides native-CUDA and native-HIP baseline implementations for our evaluation. The execution times from *MCFD* and FVSAND, across various programming models and hardware configurations, including CPUs (MPI and OpenMP) and GPUs (NVIDIA V100, NVIDIA A100, and AMD MI250X), for different mesh sizes. are summarized in Table 5.

Our preliminary results demonstrate that the MCFD implementation using RAJA or Kokkos incurs negligible overhead compared to the native CUDA and HIP implementations in FVSAND. Using a single-source codebase, MCFD can be compiled with either RAJA [24] or Kokkos [22] and executed using the OpenMP [15] backend targeting CPUs,

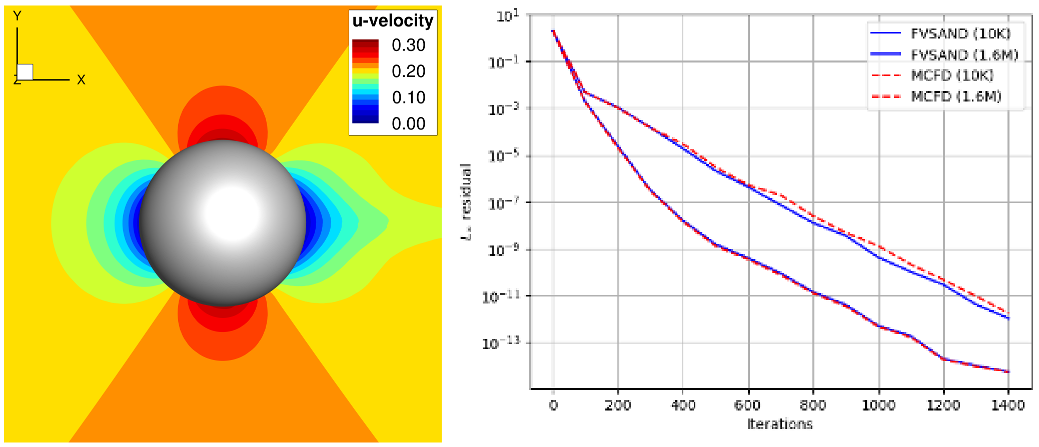


**Fig. 4 Steady-state Solution and convergence plot**

| Code / Programming Model | Execution Time (s) | | |
|---|---|---|---|
| | 100K Mesh | 1.6M Mesh | 6.4M Mesh |
| **Baseline MPI (full node / 128 CPU cores) (Nautilus)** | | | |
| FVSAND | **5.33** | **69.71** | **280.70** |
| MCFD | 4.34 | 83.62 | 285.98 |
| **OpenMP CPU (128 threads) (Narwhal)** | | | |
| MCFD – (RAJA/OpenMP) | 5.77 | 79.58 | 341.61 |
| MCFD – (Kokkos/OpenMP) | 6.18 | 85.52 | 358.64 |
| **NVIDIA V100 GPU (Narwhal)** | | | |
| FVSAND – (Native-CUDA) | **2.60** | **37.63** | **152.03** |
| MCFD – (RAJA/CUDA) | 2.21 | 36.02 | 152.02 |
| MCFD – (Kokkos/CUDA) | 2.28 | 35.79 | 154.56 |
| **NVIDIA A100 GPU (Nautilus)** | | | |
| FVSAND – (Native-CUDA) | **1.52** | **20.96** | **84.49** |
| MCFD – (RAJA/CUDA) | 1.52 | 21.99 | 92.61 |
| **AMD MI250X GPU (Ruth)** | | | |
| FVSAND – (Native-HIP) | **1.82** | **26.15** | **104.77** |
| MCFD – (RAJA/HIP) | 2.06 | 19.66 | 79.36 |
| MCFD – (Kokkos/HIP) | 2.12 | 19.63 | 79.14 |

**Table 5 Execution time comparison of FVSAND and MCFD across various programming models and hardware configurations, including CPUs (MPI and OpenMP) and GPUs (NVIDIA V100, NVIDIA A100, and AMD MI250X), for different mesh sizes.**

CUDA [11] for NVIDIA GPUs, or HIP [13] for AMD GPUs.

The baseline MPI performance on a full node of Nautilus [51] for *MCFD* and FVSAND is comparable, with FVSAND being approximately 10 seconds faster on the 1.6M-cell mesh. The performance difference for the 1.6M-cell mesh case is likely attributed to variations in how the two codes handle mesh partitioning. FVSAND reads a surface mesh and generates a Strand mesh in parallel, leveraging the inherent semi-structured nature of Strand meshes for efficiency. In contrast, MCFD reads the corresponding UGRID file and partitions the mesh using Zoltan's graph partitioner, which may introduce some imbalances relative to FVSAND.

The OpenMP CPU performance on a full node of Narwhal [50] is comparable to, though slightly slower than, the baseline MPI performance. Achieving optimal performance with OpenMP often requires careful tuning of environment variables, such as setting `OMP_PROC_BIND=spread` and `OMP_PLACES=threads`. However, it has been our experience that using distributed parallelism with MPI on the CPU cores across the compute nodes of a system generally performs better than exposing fine-grained parallelism via OpenMP on the CPU cores within a compute node. Both RAJA and Kokkos deliver similar performance with OpenMP, though we observe that RAJA consistently shows a slight edge.

On GPUs, MCFD using Kokkos and RAJA achieve performance closely aligned with the corresponding native CUDA and HIP implementations in FVSAND. The GPU execution times between RAJA or Kokkos are nearly identical, underscoring the effectiveness of portable programming models in delivering high performance. The GPU performance improves with increasing mesh sizes, as expected. This trend is particularly evident with newer GPUs such as the NVIDIA A100 and AMD MI250X.

### B. Performance Evaluation using the Taylor-Green Vortex Case

This section presents preliminary performance results from a node-to-node comparison study, obtained using XCFD, a next-generation, massively parallel and performance portable, unstructured, cell-centered, finite volume flow solver, developed under the DoD HPCMP CREATE$^{\text{TM}}$-AV program that uses *Mint* as its foundational infrastructure.

For our assessment we used the Taylor-Green Vortex benchmark problem and carried out a transient calculation for a 1000 iterations, using an explicit Runge-Kutta scheme. We compare the performance between CPU nodes and GPU nodes on the Narwhal and Nautilus systems. The node architecture of these systems is summarized in Table 4. Throughput scaling is used to characterize the performance of each compute node, where we vary the problem size from $32^3$ to $512^3$ mesh cells and measure the performance on a fixed computational resource, i.e., a CPU node versus a GPU node. As shown in Figure 5a, due to the limited computational resources of CPU nodes, the Nautilus CPU node is saturated quickly. The throughput curve is relative flat. In contrast, GPU nodes have thousands of cores, delivering higher performance and require large amounts of work in order to saturate. For the Taylor-Green Vortex Case, a Nautilus dual-socket GPU node, with 2 GPUs per socket (4 GPUs total within the node) is saturated when the mesh size is roughly at $256^3$ cells.

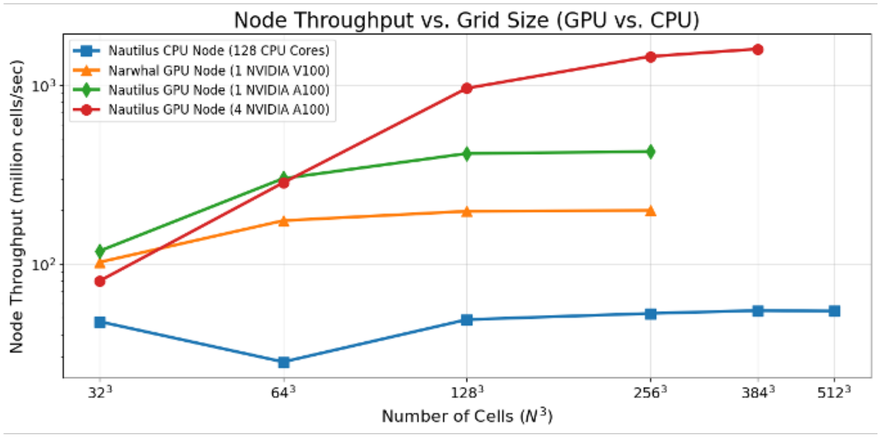
Wall clock time and speedup was also measured for the Taylor-Green Vortex cases across the different platforms. The Nautilus CPU node (128 cores) is used as a baseline and we set the mesh size to $256^3$ cells is at which the GPU node is saturated for the node-to-node comparison. The results are summarized in Figure 5b. The Narwhal GPU node, consisting of a single NVIDIA V100 GPU is **3.64x** faster. Similar performance is observed with both RAJA and Kokkos performance portability backends. A single NVIDIA A100 GPU on Nautilus is **7.78x** faster than the baseline CPU performance, which is roughly **2x** faster than an NVIDIA V100 GPU. Using all four GPUs of the Nautilus GPU node we observe **26.4x** speedup over the baseline CPU performance.

## V. Summary & Concluding Remarks

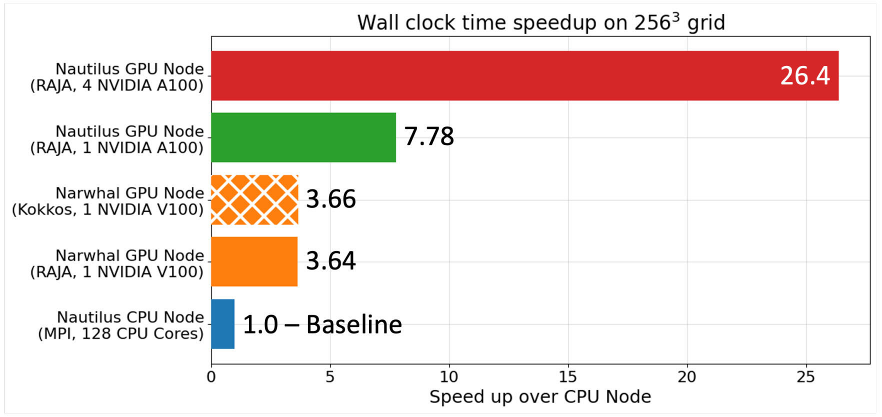
This paper introduces *Mint*, which serves as a foundational software infrastructure for the development of large-scale, distributed and massively parallel multi-physics applications. *Mint* provides a comprehensive mesh data model and a mesh-aware, fine-grained parallel execution model that seamlessly integrates with *RAJA* [24] and *Kokkos* [21–23] while remaining extensible to support other performance portability libraries and approaches. This flexibility ensures that applications are not tied to a specific software stack or implementation. As a foundational building block, *Mint* provides a flexible and sustainable infrastructure that streamlines the development of next-generation, multi-physics applications while insulating application developers, scientists and engineers from the complexities of emerging software technologies and diverse hardware architectures.

The development of large-scale, production-grade multi-physics applications is inherently complex and challenging. A robust performance portability strategy is essential to achieve a single-source codebase that is easy to maintain, ensures long-term sustainability of the application, and offers the flexibility to adapt to evolving hardware and software advancements. Mint provides a solid foundation that can help address these needs and streamline development.

The work presented herein informs the evolution, development and overarching performance portability strategy for the Next Generation Multi-Physics solvers in the DoD HPCMP CREATE$^{TM}$-AV program.



**(a) Node-to-Node Throughput for the Taylor-Green Vortex Case.**



**(b) Wall clock time & Speedup for Taylor-Green Vortex Case.**

**Fig. 5 Performance Plots from the Taylor-Green Vortex Case.**

## Acknowledgements

This work is supported by the Computational Research and Engineering for Acquisition Tools and Environments (CREATE) Program, sponsored by the U.S. Department of Defense HPC Modernization Program (HPCMP). The authors thank the CREATE Leadership for their keen interest and support in this work and the DoD Supercomputing Resource Centers (DSRC) for providing computational resources.

## References

[1] Deakin, T., McIntosh-Smith, S., Price, J., Poenaru, A., Atkinson, P., Popa, C., and Salmon, J., "Performance Portability across Diverse Computer Architectures," *2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, 2019, pp. 1–13. https://doi.org/10.1109/P3HPC49587.2019.00006.

[2] Deakin, T., Poenaru, A., Lin, T., and McIntosh-Smith, S., "Tracking performance portability on the yellow brick road to exascale," *2020 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, IEEE, 2020, pp. 1–13.

[3] Parnell, L. A., Demetriou, D. W., Kamath, V., and Zhang, E. Y., "Trends in High Performance Computing: Exascale Systems and Facilities Beyond the First Wave," *2019 18th IEEE Intersociety Conference on Thermal and Thermomechanical Phenomena in Electronic Systems (ITherm)*, 2019, pp. 167–176. https://doi.org/10.1109/ITHERM.2019.8757229.

[4] "Top500: November 2024 List," https://top500.org/lists/top500/2024/11/, 2024. Online; Accessed: 2024-11-18.

[5] "Sierra Supercomputer," Lawrence Livermore National Laboratory, 2023. URL https://hpc.llnl.gov/hardware/platforms/sierra, accessed on 17th May 2023.

[6] "Summit Supercomputer," Oak Ridge National Laboratory, 2023. URL https://www.olcf.ornl.gov/olcf-resources/compute-systems/summit/, accessed on 17th May 2023.

[7] "Frontier Supercomputer," Oak Ridge National Laboratory, 2023. URL https://www.olcf.ornl.gov/frontier/, accessed on 17th May 2023.

[8] "Aurora Supercomputer," Argonne National Laboratory, 2023. URL https://www.alcf.anl.gov/aurora, accessed on 17th May 2023.

[9] Smith, A., Loh, G. H., Schulte, M. J., Ignatowski, M., Naffziger, S., Mantor, M., Kalyanasundharam, M. F. N., Alla, V., Malaya, N., Greathouse, J. L., Chapman, E., and Swaminathan, R., "Realizing the AMD Exascale Heterogeneous Processor Vision : Industry Product," *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*, 2024, pp. 876–889. https://doi.org/10.1109/ISCA59077.2024.00068.

[10] "El Capitan Supercomputer," Lawrence Livermore National Laboratory, 2023. URL https://www.llnl.gov/news/llnl-hpe-partner-amd-el-capitan-projected-worlds-fastest-supercomputer, accessed on 17th May 2023.

[11] Cook, S., *CUDA Programming: A Developer's Guide to Parallel Computing with GPUs*, 1$^{st}$ ed., Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2012.

[12] Advanced Micro Devices, Inc., *rocM: Radeon Open Compute*, Advanced Micro Devices, Inc., 2023. URL https://rocmdocs.amd.com/en/latest/, version 4.3.

[13] Advanced Micro Devices, Inc., *HIP: Heterogeneous-Compute Interface for Portability*, Advanced Micro Devices, Inc., 2022. URL https://github.com/ROCm-Developer-Tools/HIP, version 4.2.

[14] Group, K., "SYCL - C++ Single-source Heterogeneous Programming for OpenCL," *Khronos Group*, 2014.

[15] OpenMP Architecture Review Board, "OpenMP Application Programming Interface, Version 4.5," http://www.openmp.org/mp-documents/spec4.5.pdf, 2015.

[16] OpenACC Organization, "OpenACC Application Programming Interface, Version 3.0," https://www.openacc.org/sites/default/files/spec_3_0.pdf, 2020.

[17] Harrell, S. L., Kitson, J., Bird, R., Pennycook, S. J., Sewall, J., Jacobsen, D., Asanza, D. N., Hsu, A., Carrillo, H. C., Kim, H., and Robey, R., "Effective Performance Portability," *2018 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, 2018, pp. 24–36. https://doi.org/10.1109/P3HPC.2018.00006.

[18] Bhattacharya, M., Calafiura, P., Childers, T., Dewing, M., Dong, Z., Gutsche, O., Habib, S., Ju, X., Kirby, M., Knoepfel, K., Kortelainen, M., Kwok, M., Leggett, C., Lin, M., Pascuzzi, V. R., Strelchenko, A., Viren, B., Yeo, B., and Yu, H., "Portability: A Necessary Approach for Future Scientific Software," , 2022.

[19] Pennycook, S. J., Sewall, J. D., and Lee, V. W., "A Metric for Performance Portability," , 2016.

[20] Pennycook, S. J., and Sewall, J. D., "Revisiting a Metric for Performance Portability," *2021 International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, IEEE, 2021, pp. 1–9.

[21] Edwards, H. C., Trott, C. R., and Sunderland, D., "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns," *Journal of Parallel and Distributed Computing*, Vol. 74, No. 12, 2014, pp. 3202 – 3216. https://doi.org/https://doi.org/10.1016/j.jpdc.2014.07.003, URL http://www.sciencedirect.com/science/article/pii/S0743731514001257, domain-Specific Languages and High-Level Frameworks for High-Performance Computing.

[22] Trott, C., Berger-Vergiat, L., Poliakoff, D., Rajamanickam, S., Lebrun-Grandie, D., Madsen, J., Al Awar, N., Gligoric, M., Shipman, G., and Womeldorff, G., "The Kokkos EcoSystem: Comprehensive Performance Portability for High Performance Computing," *Computing in Science & Engineering*, Vol. 23, No. 5, 2021, pp. 10–18. https://doi.org/10.1109/MCSE.2021.3098509.

[23] Trott, C. R., Lebrun-Grandié, D., Arndt, D., Ciesko, J., Dang, V., Ellingwood, N., Gayatri, R., Harvey, E., Hollman, D. S., Ibanez, D., Liber, N., Madsen, J., Miles, J., Poliakoff, D., Powell, A., Rajamanickam, S., Simberg, M., Sunderland, D., Turcksin, B., and Wilke, J., "Kokkos 3: Programming Model Extensions for the Exascale Era," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 33, No. 4, 2022, pp. 805–817. https://doi.org/10.1109/TPDS.2021.3097283.

[24] Beckingsale, D. A., Burmark, J., Hornung, R., Jones, H., Killian, W., Kunen, A. J., Pearce, O., Robinson, P., Ryujin, B. S., and Scogland, T. R., "RAJA: Portable Performance for Large-Scale Scientific Applications," *2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, 2019, pp. 71–81. https://doi.org/10.1109/P3HPC49587.2019.00012.

[25] Zenker, E., Worpitz, B., Widera, R., Huebl, A., Juckeland, G., Knüpfer, A., Nagel, W. E., and Bussmann, M., "Alpaka - An Abstraction Library for Parallel Kernel Acceleration," IEEE Computer Society, 2016. URL http://arxiv.org/abs/1602.08477.

[26] Medina, D. S., St-Cyr, A., and Warburton, T., "OCCA: A unified approach to multi-threading languages," *arXiv preprint arXiv:1403.0968*, 2014.

[27] Message Passing Interface Forum, "MPI: A Message-Passing Interface Standard, Version 3.1," Tech. rep., MPI Forum, 2015. URL https://www.mpi-forum.org/docs/mpi-standards/mpi-3.1/mpi31-report.pdf.

[28] Bell, N., Hoberock, J., Nguyen, D., and Yalamanchili, S., "Thrust: A Productivity-Oriented Library for CUDA," *GPU Computing Gems Jade Edition*, Morgan Kaufmann, 2012, pp. 359–371. https://doi.org/10.1016/B978-0-12-385963-1.00025-3.

[29] Corrigan, A., Kailasanath, K., Liu, J., Ramamurti, R., and Schwer, D., "A Hybrid Grid Compressible Flow Solver for Large-Scale Supersonic Jet Noise Simulations on Multi-GPU Clusters," *50th AIAA Aerospace Sciences Meeting Including the New Horizons Forum and Aerospace Exposition*, 2011, pp. L5006–. https://doi.org/10.2514/6.2012-564.

[30] ROCmSoftwarePlatform, "rocThrust: A Parallel Programming Library for Heterogeneous Systems," https://github.com/ROCmSoftwarePlatform/rocThrust, 2021.

[31] Min, M., Brazell, M., Tomboulides, A., Churchfield, M., Fischer, P., and Sprague, M., "Towards Exascale for Wind Energy Simulations," , 2022.

[32] Hammond, J. R., Kinsner, M., and Brodman, J., "A comparative analysis of Kokkos and SYCL as heterogeneous, parallel programming models for C++ applications," *Proceedings of the International Workshop on OpenCL*, 2019, pp. 1–2.

[33] Garimella, R. V., "MSTK - A Flexible Infrastructure Library for Developing Mesh Based Applications," *International Meshing Roundtable Conference*, 2004.

[34] "MSTK," https://github.com/MeshToolkit/MSTK, 2023. Online; Accessed: 2023-04-28.

[35] Ibanez, D., Seol, E. S., Smith, C. W., and Shephard, M. S., "PUMI: Parallel Unstructured Mesh Infrastructure," *ACM Trans. Math. Softw.*, Vol. 42, 2016, pp. 17:1–17:28.

[36] Tautges, T. J., "MOAB-SD: integrated structured and unstructured mesh representation," *Engineering with Computers*, Vol. 20, No. 3, 2004, pp. 286–293. https://doi.org/10.1007/s00366-004-0296-0, URL https://doi.org/10.1007/s00366-004-0296-0.

[37] Zhang, W., Almgren, A., Beckner, V., Bell, J., Blaschke, J., Chan, C., Day, M., Friesen, B., Gott, K., Graves, D., Katz, M., Myers, A., Nguyen, T., Nonaka, A., Rosso, M., Williams, S., and Zingale, M., "AMReX: a framework for block-structured adaptive mesh refinement," *Journal of Open Source Software*, Vol. 4, No. 37, 2019, p. 1370. https://doi.org/10.21105/joss.01370, URL https://doi.org/10.21105/joss.01370.

[38] Wissink, A. M., Hornung, R. D., Kohn, S. R., Elliott, N., and Smith, S. S., "Large Scale Parallel Structured AMR Calculations Using the SAMRAI Framework," *SC Conference*, IEEE Computer Society, Los Alamitos, CA, USA, 2001, p. 22. https://doi.org/10.1109/SC.2001.10029, URL https://doi.ieeecomputersociety.org/10.1109/SC.2001.10029.

[39] LLNL, "SAMRAI: Structured Adaptive Mesh Refinement Application Infrastructure," https://github.com/LLNL/SAMRAI, 2023.

[40] Beckingsale, D. A., McFadden, M. J., Dahm, J. P. S., Pankajakshan, R., and Hornung, R. D., "Umpire: Application-focused management and coordination of complex hierarchical memory," *IBM Journal of Research and Development*, Vol. 64, No. 3/4, 2020, pp. 00:1–00:10. https://doi.org/10.1147/JRD.2019.2954403.

[41] Grete, P., Dolence, J. C., Miller, J. M., Brown, J., Ryan, B., Gaspar, A., Glines, F., Swaminarayan, S., Lippuner, J., Solomon, C. J., Shipman, G., Junghans, C., Holladay, D., Stone, J. M., and Roberts, L. F., "Parthenon—a performance portable block-structured adaptive mesh refinement framework," *The International Journal of High Performance Computing Applications*, Vol. 0, No. 0, 2022, p. 0. https://doi.org/10.1177/10943420221143775, URL https://doi.org/10.1177/10943420221143775.

[42] Kirk, B. S., Peterson, J. W., Stogner, R. H., and Carey, G. F., "libMesh: a C++ library for parallel adaptive mesh refinement/coarsening simulations," *Engineering with Computers*, Vol. 22, No. 3, 2006, pp. 237–254. https://doi.org/10.1007/s00366-006-0049-3, URL https://doi.org/10.1007/s00366-006-0049-3.

[43] Balay, S., Abhyankar, S., Adams, M. F., Brown, J., Brune, P., Buschelman, K., Eijkhout, V., Gropp, W. D., Kaushik, D., Knepley, M. G., McInnes, L. C., Rupp, K., Smith, B. F., and Zhang, H., "PETSc: Portable, Extensible Toolkit for Scientific Computation," *Concurrency and Computation: Practice and Experience*, Vol. 28, No. 4, 2016, pp. 1870–1905. https://doi.org/10.1002/cpe.3595.

[44] Moore, J. L., Morgan, N. R., and Horstemeyer, M. F., "ELEMENTS: A high-order finite element library in C++," *SoftwareX*, Vol. 10, 2019, p. 100257. https://doi.org/https://doi.org/10.1016/j.softx.2019.100257, URL https://www.sciencedirect.com/science/article/pii/S235271101930113X.

[45] Anderson, R., Andrej, J., Barker, A., Bramwell, J., Camier, J.-S., Cerveny, J., Dobrev, V., Dudouit, Y., Fisher, A., Kolev, T., Pazner, W., Stowell, M., Tomov, V., Akkerman, I., Dahm, J., Medina, D., and Zampini, S., "MFEM: A Modular Finite Element Methods Library," *Computers & Mathematics with Applications*, Vol. 81, 2021, pp. 42–74. https://doi.org/10.1016/j.camwa.2020.06.009.

[46] Falgout, R., Jones, J., and Meier Yang, U., "Hypre: A library of high performance preconditioners," *Computing in Science & Engineering*, Vol. 4, No. 3, 2002, pp. 24–28. https://doi.org/10.1109/5992.998641.

[47] Hosseinverdi, S., Sitaraman, J., Zagaris, G., Lakshminarayan, V. K., and Holst, K. R., *Development of a Performance Portable Massively Parallel Unstructured Compressible Flow Solver*, ???? https://doi.org/10.2514/6.2023-3427, URL https://arc.aiaa.org/doi/abs/10.2514/6.2023-3427.

[48] Bodling, A. L., Zagaris, G., Jude, D., Sitaraman, J., and Hosseinverdi, S., *Development of a Performance Portable Structured Grid Based Compressible Flow Solver*, ???? https://doi.org/10.2514/6.2024-3537, URL https://arc.aiaa.org/doi/abs/10.2514/6.2024-3537.

[49] Sitaraman, J., Hosseinverdi, S., Jude, D., Roget, B., Abras, J., Lakshminarayan, V., and Zagaris, G., *Progress in Massively Parallel and Performance Portable Overset Implementations in HPCMP CREATE A/V Helios*, ???? https://doi.org/10.2514/6.2024-1940, URL https://arc.aiaa.org/doi/abs/10.2514/6.2024-1940.

[50] Department of Defense High Performance Computing Modernization Program (DoD HPCMP), "Narwhal High Performance Computing System," Available at https://centers.hpc.mil/systems/unclassified.html#Narwhal, ????. 2024.

[51] Department of Defense High Performance Computing Modernization Program (DoD HPCMP), "Nautilus High Performance Computing System," Available at https://centers.hpc.mil/systems/unclassified.html#Nautilus, ????. 2024.

[52] Jay Sitaraman, "FVSAND: A Finite Volume Sandbox," https://github.com/jsitaraman/fvsand, 2022. Accessed: 2023.