ECP Milestone Report
Propose high-order mesh/data format
WBS 2.2.6.06, Milestone CEED-MS18

Jed Brown
Veselin Dobrev
Som Dutta
Paul Fischer
Kazem Kamran
Tzanio Kolev
David Medina
Misun Min
Thilina Ratnayaka
Mark Shephard
Cameron Smith
Jeremy Thompson

June 29, 2018

DOCUMENT AVAILABILITY

Reports produced after January 1, 1996, are generally available free via US Department of Energy (DOE) SciTech Connect.

Website http://www.osti.gov/scitech/

Reports produced before January 1, 1996, may be purchased by members of the public from the following source:

National Technical Information Service
5285 Port Royal Road
Springfield, VA 22161
Telephone 703-605-6000 (1-800-553-6847)
TDD 703-487-4639
Fax 703-605-6900
E-mail info@ntis.gov
Website http://www.ntis.gov/help/ordermethods.aspx

Reports are available to DOE employees, DOE contractors, Energy Technology Data Exchange representatives, and International Nuclear Information System representatives from the following source:

Office of Scientific and Technical Information
PO Box 62
Oak Ridge, TN 37831
Telephone 865-576-8401
Fax 865-576-5728
E-mail reports@osti.gov
Website http://www.osti.gov/contact.html

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

ECP Milestone Report
Propose high-order mesh/data format
WBS 2.2.6.06, Milestone CEED-MS18

Office of Advanced Scientific Computing Research
Office of Science
US Department of Energy

Office of Advanced Simulation and Computing
National Nuclear Security Administration
US Department of Energy

June 29, 2018

ECP Milestone Report
Propose high-order mesh/data format
WBS 2.2.6.06, Milestone CEED-MS18

Approvals

Submitted by:

Tzanio Kolev, LLNL CEED PI	Date

Approval:

Andrew R. Siegel, Argonne National Laboratory Director, Applications Development Exascale Computing Project	Date

Revision Log

Version	Creation Date	Description	Approval Date
1.0	June 29, 2018	Original	

EXECUTIVE SUMMARY

The goal of this milestone was to develop a high-order "Field and Mesh Specification" interface that allows a wide variety of applications and visualization tools to represent unstructured high-order meshes with general high-order finite element fields defined on them.

Currently there is no agreement on how to represent high-order solutions and meshes (or even non-standard finite elements, such as Nedelec and Raviart-Thomas elements) within a common, easy to read and write to format. This is a bottleneck to adopting high-order technologies, as e.g. proper visualization of the high-order information is critical for its use in practice.

In this milestone, the CEED team proposed a new specification, FMS, for high-order fields and meshes together with a simple API library and documentation for it. Initial support for the new FMS interface was added to MFEM and PUMI and we demonstrated its use for data transfer between the codes. Our long-term goal is to work with apps and vis teams within and outside of the ECP to shape the FMS interface in a form that is beneficial for them to adopt for high-order data exchange and for high-order visualization and data analysis.

The artifacts delivered include a simple API library and documentation for the new FMS interface, freely available in the CEED's FMS repository on GitHub. See the CEED website, `http://ceed.exascaleproject.org/fms` and the CEED GitHub organization, `http://github.com/ceed` for more details.

In addition to details and results from the above R&D efforts, in this document we are also reporting on other project-wide activities performed in Q3 of FY18, including: extensive benchmarking of CEED Bake-Off problems and kernels on BG/Q, GPU and AMD/EPYC platforms, improvements in libCEED, results from application engagements, progress on FEM-SEM preconditioning, three new software releases, and other outreach efforts.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

The goal of this milestone was to develop a high-order ”Field and Mesh Specification” interface that allows a wide variety of applications and visualization tools to represent unstructured high-order meshes with general high-order finite element fields defined on them.

Currently there is no agreement on how to represent high-order solutions and meshes (or even non-standard finite elements, such as Nedelec and Raviart-Thomas elements) within a common, easy to read and write to format. This is a bottleneck to adopting high-order technologies, as e.g. proper visualization of the high-order information is critical for its use in practice.

In this milestone, the CEED team proposed a new specification, FMS, for high-order fields and meshes together with a simple API library and documentation for it. Initial support for the new FMS interface was added to MFEM and PUMI and we demonstrated its use for data transfer between the codes. Our long-term goal is to work with apps and vis teams within and outside of the ECP to shape the FMS interface in a form that is beneficial for them to adopt for high-order data exchange and for high-order visualization and data analysis.

The artifacts delivered include a simple API library and documentation for the new FMS interface, freely available in the CEED's FMS repository on GitHub. See the CEED website, `http://ceed.exascaleproject.org/fms` and the CEED GitHub organization, `http://github.com/ceed` for more details.

2. THE CEED HIGH-ORDER FORMAT

The proper specification of the mesh and associated field data for high-order methods is more complex than in the case of low order methods. Since most of the existing tools, file formats, etc., have been defined without full consideration of the needs of high-order methods, it is necessary for CEED to define appropriate representations of, and methods to interact with, high-order meshes and associated fields. The emphasis of the CEED high-order mesh and field representation, and associated APIs, is to support the needs of high-order mesh-based analysis codes and the visualization tools used to examine the results of high-order mesh simulations.

In this section we introduce CEED's new high-order ”Field and Mesh Specification” (FMS) interface that allows a wide variety of applications and visualization tools to represent unstructured high-order meshes with general high-order finite element fields defined on them. We begin this section with a brief discussion of the information needed to support fully automated mesh-based simulations that must start from a general problem definition. Given this, more specific consideration is given to the needs of high-order mesh based simulation codes and the tools used to visualize the simulation results, leading to a description of the overall design of the CEED high-order mesh and field representations and interactions that must be supported. Finally a more detailed discussion of the implementation of the FMS interface is given as well as an initial demonstration of its application.

2.1 Background on Automated Simulations over Complex Domains

An important consideration in the execution of any simulation solving partial differential equations (PDEs) over physical domains is the definition of the domain and specification of loads, material properties, boundary conditions and initial conditions, to be referred to as attributes, that must be specified with respect to that domain to complete the PDE problem definition. Although the input to a mesh-based analysis is the mesh with all attributes associated with that mesh, there are a number of advantages to employing a higher level, ”mesh independent” problem definition. The most convenient precise domain definition is a boundary presentation. Although using a boundary representations for domains defined in CAD systems supports high levels of simulation automation, the use of an overall boundary representation of the domain does not require the inclusion of a CAD system, or any other specific form of domain shape information. However, even in the case where the only domain information is the mesh, a domain boundary representation provides a natural means for the specification of the domain and the associated analysis attribute information.

A boundary representation consists of two components. The first is the model topology which is a general abstract presentation of the domain. The second is the shape information associated with the topological entities in the boundary representation.

2.1.1 Domain Topology

Model topology consists of the topological entities that make-up the boundary and interior of the domain and their adjacencies in terms of the entities that bound a given entity and/or the entities a given entity bounds. Since analysis domains can include multiple material regions and combinations of lower dimension entities that do not bound regions, it is desirable to employ boundary representations that can represent the general combinations of volumes, surfaces, curves and points. The most well known representation for supporting such non-manifold (more strictly speaking not 2-manifold) representations is the radial-edge data structure [13].

2.1.2 Domain Shape Information

When using the abstraction of topological entities in a boundary representation it becomes straight forward to support a wide range of methods for the specification of the geometric information defining the shape of the entities. Limiting our discussion to 3-D geometric domains in which time can be an easily separable dimension (the 3-D geometric domain can evolve in time) one can consider entities being parameterized by the a set of coordinates equal in number to their dimension. Although conceptually a nice idea, there are no known methods to automatically define useful fitted coordinate parametrization in the case of the full range of real 3-D objects of interest. CAD systems representations have a simple one coordinate parametrization of edges, a somewhat complex, unless you interact directly with the CAD system, two coordinate trimmed parametrization of the faces, and no parametrization of the regions.

Instead of attempting to defining a single generalization to define shape information for all boundary entities, a more effective approach is to take advantage of the fact it is straight forward to associate various forms of shape information to model topological entities and to support multiple forms of shape information with the most useful being:

- The shape can be managed and interacted with directly by maintaining a link to a CAD representation that provides APIs to support the required geometric interrogations.
- The shape of the model faces and edges is defined as the union of the shapes of the finite element edges and faces that are used to represent them.
- The shape can be some given analytic expression such as conics or a spline curve or patch.
- The shape may be implicitly defined through some spatial representation (e.g., level sets on a background grid or overlay mesh).

2.1.3 Analysis Attributes

The analysis attributes of loads, material properties and boundary conditions are best described in terms of input tensor fields that are defined by some distribution, written in terms of a discrete number of parameters, that vary over the space of the model entity the attribute is associated with. Thus the only basic difference between an analysis attribute and a solution field is the solution fields are distributions, written in terms of a discrete number of parameters, that vary over the space of the appropriate finite element entities instead of a geometric model entity.

2.1.4 Mesh Definition

The most common description of a mesh used by analysis programs consists of a sets of specific element types defined as ordered list of nodes, plus the coordinates of the nodes. In this form, all shape information is defined in terms of the nodal coordinates. The mesh entity topology is captured in the element type code while the list of nodes defines a single downward adjacency and, when coupled with the nodal coordinates, element type code and ”geometry shape functions” defines the geometry of the mesh entities. In the case of doing a analyses on a fixed mesh this is sufficient information. There are a number of deficiencies with such a minimal representation for use in the execution of high-order finite element simulations, particularly if one want to properly support the visualization of properly represented solution fields and when there is a goal of automating processed of that go from high level problem descriptions adaptively controlled simulation results.

Just as in the case of the overall domain definition, a boundary representation provides an effective abstraction for use in the description of high-order meshes. There are two assumptions made that allow the use of mesh topological representations that are far less complex that a full non-manifold geometric model as is required for the overall analysis domain.

The first assumption is that a single analysis code element is a topological entity with "no holes". That is each mesh edge has vertices at its only its two ends, each mesh face is bounded by a single closed loop of mesh edges, and mesh region is bounded by a single closed set of mesh faces (one shell). Note: in the case of a high-order finite element it is possible that the starting and ending vertex of an edge is the same - however support of this case does require adding complexity to specific mesh manipulation operations. Since each mesh face has a single loop and each mesh region has a single shell, there is no need to maintain the shell and loop in the mesh topology.

The second assumption is actually a requirement that the relationship of the mesh entities to the model entities is defined at the time of mesh generation and is maintained throughout the simulation, including when the mesh is modified due to mesh adaptation or simulation dictated changes. This relationship is, often referred to as the classification, is the unique association of each mesh entity to the lowest dimension possible topological entity in the analysis domain it is associated with [1, 10]. In the case of the classification of a mesh against an analysis domain: Each region is classified to a model region. Each face is classified to the model region it is in, or model face it is on. Each edge is classified to the model region it is in, model face it is on, or model edge it is on. Each vertex is classified to the model region it is in, model face it is on, model edge it is on, or model vertex it represents. This classification information is critical for supporting general mesh adaptation procedures, properly supporting meshes on of non-manifold model situations form the simple case of multiple materials, to a more complex case of representing contact interfaces, without ad-hoc extensions to the mesh representation. Classification is also useful for the high-level specification and tracking of analysis attributes.

In the reminder of this section, the term regions, face, edge and vertex refer to mesh regions, mesh faces, mesh edges and mesh vertices. When discussing the relationship of the mesh entities to the analysis domain topology, the domain topology entities will be referred to as model regions, model faces, model edges and model edges. The relationship of a finite element to a model topological entity is based on the geometric dimension of the elements: 3D elements are regions, 2D element are faces, 1D elements are edges and 0D elements are vertices. There are various nomenclatures that have been defined to precisely describe mesh topologies (e.g., [1, 10]) and operations. We will avoid that level of detail in this document.

The decision of which of the of mesh entities of the four possible dimensions (regions, faces, edges and vertices) to explicitly represent and which of the 12 possible topological adjacencies should be maintained begins with a determination the functions that must be supported by the codes interacting with the mesh and how efficiently one wants to be able to execute those operations.

2.1.5 Design Principles for CEED's Field and Mesh Specification

CEED's FMS description is in terms of topological entities and a selected set of adjacencies of those entities, thus providing a general abstraction for the definition of a mesh. All other information is defined in appropriate fields that are linked to the mesh entities.

Unlike meshes of straight edged mesh entities or even the somewhat common "isoparametric quadratic" element, there are multiple options employed to define the geometric shape information of high-order elements including (i) interpolation through given points, (ii) Bezier or NURBS curves/splines, (iii) analytic expressions (e.g., a circular are) or (iv) implicitly with links to an underlying CAD or implicit function (e.g., a zero level set). To effectively support the variety of options for the definition of element geometry all mesh entity geometric shape information is maintained as mesh entity fields. Note that even in the most common case of interpolating geometry, the there are multiple rules employed with respect to the assignment of the local coordinates to the interpolating points.

To support a full set of capabilities and avoid substantial duplication of field data, field data should be associated with the lowest order mesh entities they are defined with respect to. Given that there will be field data associated with the mesh entities of each dimension, FMS explicitly represents the mesh entities for all four dimensions (vertices, edges, faces and regions). For the effective implementation of FMS one must also determine how flexible they intend to be with respect to the topology of faces and regions. Within the already stated restriction of a single shell for regions and a single loop, the most flexible approach is to define the faces as a loop of any number of edges and the regions as bounded by any number of faces. However, much more efficient implementations can be developed if the face and region topologies are limited to a small set. Since FMS is targeting high-order method on such meshes, it supports faces with only three or four edges and the regions are limited a specific set of face configurations. Given this limited set of topologies, FMS will also employ specific ordering of the entities thus supporting the more efficient execution of functional evaluations.

This approach provides an effective means store and operate on fields defines over a mesh. In all cases, the process of accessing the field information for an element in the mesh begins by accessing the element's core mesh entity, which is the mesh entity of maximum dimension for that element (i.e., if the element is a hexahedron, that will be a region, it the element is a triangle it will be a face, etc.). In the case of a L^2 element the full set of field parameters will be associated with the core mesh entity. However in the case where there is C^0 (or higher) order continuity between elements that continuity is obtained by using shared field parameters that are associated with the mesh entities that bound the core mesh entity. Thus the full description of the field must consider the field parameters associated with the closure of the core mesh entity. Thus for a C^0 high-order hexahedron element the complete field will be defined in terms of field parameters associated with the core mesh region, the faces that bound that region, the edges that bound those faces, and the vertices that bound those edges.

It is clear that to effectively construct C^0 fields based on fields associated with mesh entities of each dimension, there is the need to quickly determine the appropriate mesh adjacencies. Since this operation, and others used when executing an analysis on a given mesh, can be effectively supported by downward mesh adjacencies, the base adjacencies maintained by FMS are the one-level downward adjacencies of regions-to-faces, faces-to-edges and edges-to-vertices. (It should be noted that given these adjacencies and proper mesh classification, it is possible to construct any of the other mesh adjacencies. If other adjacencies are desired they are can be effectively determines and stored during a mesh traversal.) The mesh adjacency information is also effective for supporting common post processing operations. For example in cases where the physics of a problem indicates a field is "continuous", but the mesh-based discretization employed L^2 fields, is common to construct "patch wise" C^0 projection of that L^2 field before visualizing it. The mesh adjacency information effectively supports the determination of the patches.

It is important to recognize that the discrete field parameters, the dof, alone do not fully qualify how a field various over the elements in the mesh. The fields are only fully quantified with a knowledge of the functional form those parameters are used to quantify. Unlike lower order C^0 methods where one common set of functions are used, there are a number of alternative forms used in different finite element codes, and even within a single code infrastructure, there are various combinations of functional forms used. For example MFEM has both C^0 and L^2 discretization methods that an be used in various combination to solve specific problem. Since it is not reasonable to provide a specific enumerated library of functions covering all field types, an API based approach is being adopted to interacting with FMS. The fact that FMS is focused on elements, the abstraction of topology, and the collections of discrete parameters, makes it straight forward to interact through the APIs. The use of the API also makes it quite clear as to what is needed for a new analysis code to interact through FMS, they simply have to provide the portions of the APIs that actually operate on the fields.

CEED's FMS also supports non-confirming adaptive mesh refinement through the application of element level sub-divisions and well as the coarsening of elements back to original starting mesh.

Parallel meshes are supported by placing each of the sets of elements of an non-overlapping partitioning of the mesh into mesh subsets.

2.2 FMS Implementation

This section contains a description of the initial high-order FMS interface which is intended as a lightweight format and API that can represent general finite elements within a common, easy to use framework. This includes high-order solutions and meshes, such as the ones in Figure 1, as well as non-standard finite elements, such as Nedelec and Raviart-Thomas elements.

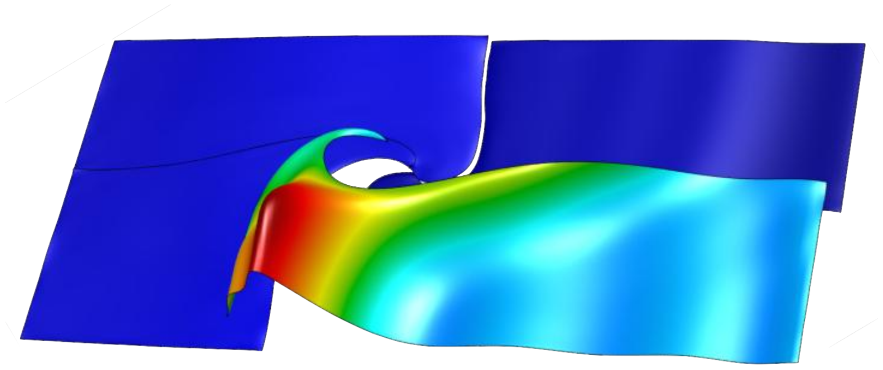


Figure 1: Simple example of a high-order field on a high-order curved mesh. This mesh has four 12th order elements on which an 11th order discontinuous field has been defined. CEED's FMS aims to provide an interface to describe such complex data so it could be exchanged between applications and visualized by tools like ParaView and VisIt.

The data described by FMS is a collection of:

1. Mesh *topology*
2. Set of *fields* defined on that topology

This separation of mesh geometry and topology is important in many applications (e.g. those with moving meshes) and allows the FMS format to handle general high-order meshes in a simple and uniform way. Note that the geometry of the mesh, described by the coordinates of the vertices, for linear meshes, or the coordinates of the nodes for high-order meshes is specified itself as just another finite element field, called *nodes* or *coordinates* (for example as vector field in an H^1-space of appropriate order).

For more details on the CEED FMS interface see `http://ceed.exascaleproject.org/fms/` and the automatically updated Doxygen documentation, `https://codedocs.xyz/CEED/FMS`. We are interested in collaborating with application scientists and visualization teams to further improve FMS and make it a viable option for high-order data exchange and high-order visualization and data analysis.

2.2.1 Mesh Topology

The mesh topology, described by the type `FmsMesh` below, is represented by the following interconnected *mesh entities*:

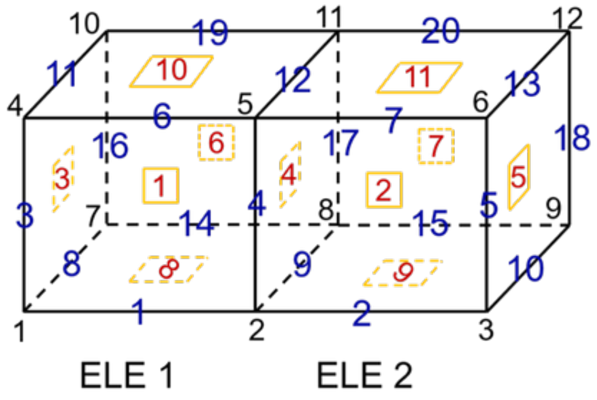
- 0d-entities = Vertices
- 1d-entities = Edges
- 2d-entities = Faces: Triangles and Quads
- 3d-entities = Elements/Volumes/Regions: Tets, Hexes, Wedges and Pyramids

The topology does not include geometric or finite element information, but does include relations between the entities and their orientation with respect to reference configurations. The topology can also be split into *domains* and *components* which form a decomposition (e.g. for parallel computations), or select a subset of physical interest, respectively. In FMS, the Mesh topology is described by objects of type `FmsMesh`, which can be constructed with `FmsMeshConstruct()` and deleted with `FmsMeshDestroy()`.

Mesh Entities

FMS makes the following assumptions about the mesh:

1. . **All entities of all dimensions are represented in the mesh domains**, i.e. we expect explicitly numbered elements, faces, edges and vertices.



2. **An entity is described in terms of its sides**, where a *side* is a one dimension lower entity; in other words an element is described in terms of its faces, a face in terms of its edges and an edge in terms of its vertices.

$$\begin{tikzcd}\boxed{\text{Element}} \arrow[r] & \boxed{\text{Face}} \arrow[r] & \boxed{\text{Edge}} \arrow[r] & \boxed{\text{Vertex}}\end{tikzcd}$$

The first assumption may seem unusual for finite element codes, but it naturally describes the entities with which finite element degrees of freedom are associated, making it easy to describe fields (finite element functions) below.

The second assumption provides a set of *downward adjacencies* which allows easy reconstruction of relations like element-vertex, while providing flexibility for algorithms that need to loop over faces and edges.

More specifically, a mesh entity is described by a tuple of its *side entity* indices. For an entity of dimension d, its side entities are its boundary entities of dimension $d-1$.

For `FMS_TRIANGLE` and `FMS_QUADRILATERAL`, the edges (sides), "abc"/"abcd" and the vertices "012"/"0123" are ordered counterclockwise, as illustrated in the following diagram:

```
3--c--2        2
|     |       / \
d     b      c   b
|     |     /     \
0--a--1    0---a---1
```

For 3D entities, the ordering of the edges inside the faces should follow the counterclockwise ordering when looking the face from outside. This rule is followed by the choices given below. For `FMS_TETRAHEDRON`, the faces (sides), "ABCD", the edges, "abcdef", and the vertices, "0123" are ordered as follows:

```
   z=0         y=0         x=0       x+y+z=1
    2           3           3           3
   / \         / \         / \         / \
  b   c       d   e       f   d       e   f
 /  A  \     /  B  \     /  C  \     /  D  \
1---a---0   0---a---1   2---c---0   1---b---2
```

For example, vertex "0" has coordinates $(x, y, z) = (0, 0, 0)$, vertex "1" has coordinates $(1, 0, 0)$, etc. For `FMS_HEXAHEDRON`, the faces (sides), "ABCDEF", the edges, "abcdefghijkl" and the vertices, "01234567", are ordered as follows:

```
      7--g--6
     /|    /|
    / l   / k   z=0      z=1      y=0      y=1      x=0      x=1
   h  |  f  |   bottom   top      front    back     left     right
  /   3-/c--2   2--c--3  7--g--6  4--e--5  6--g--7  7--h--4  5--f--6
 /   / /   /    |     |  |     |  |     |  |     |  |     |  |     |
4--e--5   /     b  A  d  h  B  f  i  C  j  k  D  l  l  E  i  j  F  k
|  d  |  b      |     |  |     |  |     |  |     |  |     |  |     |
i /   j /       1--a--0  4--e--5  0--a--1  2--c--3  3--d--0  1--b--2
|/    |/
0--a--1
```

For example, vertex “0” has coordinates $(x, y, z) = (0, 0, 0)$, vertex “6” has coordinates $(1, 1, 1)$, etc.

Mesh Domains

Mesh domains describe sets of interconnected mesh entities (0d, 1d, 2d and 3d). All entities in the domain are enumerated locally.

Each mesh domain can be viewed as its own independent mesh, as it provides full description of the entities inside it (the face of all volumes are described as 2-entities, the edges of all faces are described as 1-entities, etc.)

A typical example to keep in mind is parallel computations, where an MPI tasks can contain one (or several) domains, which can be processed independently in the interior. Connections between domains are described using shared entities. Domains are assigned a string name and an integer id. Thus, a domain is identified uniquely by the tuple: (name, id, partition-id) where the partition id is the one assigned to the containing mesh.

In FMS, domains are described using objects of type `FmsDomain` which can only exist as part of an `FmsMesh`; they are created using the `FmsMeshAddDomain()` function. To describe the mesh entities inside the domain, one uses the functions `FmsDomainSetNumVertices()`, `FmsDomainSetNumEntities()`, `FmsDomainAddEntities()`, etc.

Mesh Components

Mesh components are regions of physical interest defined across the mesh domains, such as materials, sections of the boundary, different parts in fluid-structure interaction, etc. The subset of the component on each domain is a set of entities, which is called a *part*. Each part is described as a list of *entity indices* which point to entities of the associated domain. All entities in the component must have (i) the same dimension and (ii) specified orientation (relative to the entity as described in its domain).

Note that different components can overlap and be of different dimensions. Typically, the whole mesh is represented by a special component of all elements (3-entities) on all domains. This is the component, for example, on which the mesh nodes will be defined (see below).

In order to facilitate the definition of fields on the component, the following additional data can be stored in every part of the component: for all lower dimensional entities that are boundary to the *main* entities of the part, define an array that maps the local (to the part) indices to the domain-indices. These arrays plus the main array (the one describing the highest dimensional entities of the component) define local numberings of all entities inside each part. These numberings will be used to define the ordering of the degrees of freedom of a field. When the main entity array is `NULL` (indicating that all entities in the domain are used) then the lower dimensional entities will also be `NULL` because there is no need to have local numbering of the entities — the original numbering defined by the domain can be reused.

In addition to the parts, a component also stores relations to other components. A relation to a component of lower or higher dimension indicates a boundary or interface relation, i.e. the lower dimensional component describes a subset of the boundary entities of the higher dimensional component. A relation to another component of the same dimension is acceptable but has no specified meaning.

A component has an associated *coordinates* or *nodes* field, of type `FmsField`, which may be `NULL`. Mesh *tags* (discussed below) are defined on the set of all *main* entities in a mesh component.

Discrete fields are defined on mesh components. Unlike tags, discrete fields generally associate data not only with the main entities described by the mesh component but also with the lower-dimensional boundary entities of the main component entities.

In FMS, components are described using objects of type `FmsComponent` which only exist as part of an `FmsMesh` and are created using the `FmsMeshAddComponent()` function. Parts and their entities can be added to the component with the functions `FmsComponentAddPart()`, `FmsComponentAddPartEntities()`, `FmsComponentAddRelation()`, etc.

Mesh Tags

A mesh *tag* is an array of integers describing the main entities in a given component. Optionally, the integer tags can be assigned string descriptions. Tags could be used to mark different boundary conditions, different materials in the mesh, store the orders (polynomial degrees) associated with the component entities in variable-order discrete spaces, etc. Each tag naturally defines a non-overlapping decomposition of the associated component. The array with the integer tags is ordered part-by-part within the mesh component.

In FMS, tags are described using objects of type `FmsTag` which exist only as a part of a mesh and can be created using the `FmsMeshAddTag()` function and described by the functions `FmsTagSetComponent()`, `FmsTagSet()`, `FmsTagAddDescriptions()`, etc.

2.2.2 Fields

The fields, described by the type `FmsField` below, are high-order finite element functions given in terms of their degrees of freedom associated with the interior of each of the mesh entities. These fields can be specified only on mesh components, and can describe any scalar or vector function in the de Rham complex (H^1, H(curl), H(div) and L^2 elements). As noted above, the coordinates of the mesh nodes, specifying the actual mesh shape in physical space, are just a special field on the whole mesh.

In FMS, each field is specified by defining a `FmsFieldDescriptor` that contains the associated mesh component, the basis type describing the field and type of the field itself i.e. continuous, discontinuous etc. Two options are available for storing vector field data, either by dimension, `FMS_BY_VDIM` or by nodes `FMS_BY_NODES`. In the former, the pair of indices `(i,j)` of degree of freedom `i` and vector component `j` are mapped to a 1D array by the formula `i*vdim+j`; in latter choice (`FMS_BY_NODES`), the indices `(i,j)` are mapped using the formula `i+num_dofs*j`.

FMS fields and their descriptors are stored in, and only exist as part of, objects of type `FmsDataCollection` which in turn can be created on top of an `FmsMesh` object using the function `FmsDataCollectionCreate()`. Objects of types `FmsField` and `FmsFieldDescriptor` can be created with the functions `FmsDataCollectionAddFieldDescriptor()` and `FmsDataCollectionAddField()`.

2.2.3 Example of a Simple Mesh

Below is a complete example of using the FMS interface to describe a simple mesh.

```c
// Example of using the FMS interface to describe the following 1-element mesh
//
//  <3>----(2)----<2>
//   |             |
//   |             |
//  (3)    [0]    (1)
//   |             |
//   |             |
//  <0>----(0)----<1>
//
// where <V> = vertex id, (E) = edge id and [Q] = quadrilateral id. The edge and
// element orientation are counter-clockwise:
//
// (0) = <0>,<1> ; (1) = <1>,<2> ; (2) = <2>,<3> ; (3) = <3>,<0>
// [0] = (0), (1), (2), (3)
//
// The mesh has one domain and two components: a volume containing element [0]
// and boundary containing edge (2). The volume component is tagged with the
// "material" tag.

#include <fms.h>

```

```c
int main(int argc, const char *argv[]) {

  // Create the Mesh object
  FmsMesh mesh;
  FmsMeshConstruct(&mesh);
  // FmsMeshSetPartitionId(mesh, 0);

  // Mesh Domains
  FmsDomain *domain;
  FmsMeshAddDomains(mesh, "domains", 1, &domain);

  FmsDomainSetNumVertices(domain[0], 4);
  FmsDomainSetNumEntities(domain[0], FMS_EDGE, FMS_INT32, 4);
  FmsDomainSetNumEntities(domain[0], FMS_QUADRILATERAL, FMS_INT32, 1);

  // Edges
  const int edge_vert[] = {0,1, 1,2, 2,3, 3,0};
  FmsDomainAddEntities(domain[0], FMS_EDGE, NULL, FMS_INT32, edge_vert, 4);

  // Faces
  const int quad_edge[] = {0, 1, 2, 3};
  FmsDomainAddEntities(domain[0], FMS_QUADRILATERAL, NULL,
                       FMS_INT32, quad_edge, 1);

  // Mesh Components
  FmsComponent volume;
  FmsMeshAddComponent(mesh, "volume", &volume);
  FmsComponentAddDomain(volume, domain[0]);

  FmsComponent boundary;
  FmsMeshAddComponent(mesh, "boundary", &boundary);
  FmsInt part_id;
  FmsComponentAddPart(boundary, domain[0], &part_id);
  const int bdr_edge[] = {2};
  FmsComponentAddPartEntities(boundary, part_id, FMS_EDGE, FMS_INT32,
                              FMS_INT32, FMS_INT32, NULL, bdr_edge, NULL, 1);

  FmsComponentAddRelation(volume, 1); // 1 = index of "boundary" component

  // Mesh Tags
  FmsTag material;
  FmsMeshAddTag(mesh, "material", &material);
  FmsTagSetComponent(material, volume);
  const int material_tags[] = {1};
  FmsTagSet(material, FMS_INT32, FMS_INT32, material_tags, 1);

  // Finalize the construction of the Mesh object
  FmsMeshFinalize(mesh);
  // Perform some consistency checks.
  FmsMeshValidate(mesh);

  // Coordinates Field
  // Defdine data collection
  FmsDataCollection dc;
  FmsDataCollectionCreate(mesh, "data collection", &dc);

  // Define field descriptor
  FmsFieldDescriptor coords_fd;
  FmsDataCollectionAddFieldDescriptor(dc, "coords descriptor", &coords_fd);
  FmsFieldDescriptorSetComponent(coords_fd, volume);
  FmsInt coords_order = 1;
  FmsFieldDescriptorSetFixedOrder(coords_fd, FMS_CONTINUOUS,
                                  FMS_NODAL_GAUSS_CLOSED, coords_order);

  // Define the coordinates field
  FmsField coords;
  FmsDataCollectionAddField(dc, "coords", &coords);
  const double coords_data[] = {
    0.,0.,
    1.,0.,
    1.,1.,
    0.,1.
  };
  FmsInt sdim = 2; // A 2D mesh embedded in "sdim"-dimensional space.
  FmsFieldSet(coords, coords_fd, sdim, FMS_BY_VDIM, FMS_DOUBLE, coords_data);

  FmsComponentSetCoordinates(volume, coords);

  // Use the mesh to perform computations

  // Destroy the data collection: destroys the FmsMesh and all other linked Fms
  // objects.
  FmsDataCollectionDestroy(&dc);

  return 0;
}
```

2.3 Demonstration of FMS Data Exchange

To illustrate the usage of CEED's FMS interface in practice, we consider a data exchange between the PUMI and MFEM packages in CEED. Given a PUMI mesh, an `FmsMesh` is constructed using PUMI and the FMS API. This mesh is then used to load an MFEM mesh and solve a benchmark problem. This scenario tests the capability of the FMS interface to both write a mesh in its format, given a PUMI mesh as input, and read it to a new format, MFEM in this case.

For the sake of simplicity, a one-domain, one-component mesh is considered. The input is a 3D *quadratic* unstructured tetrahedral mesh in PUMI 'smb' format classified on the corresponding CAD model, see Figure 2.

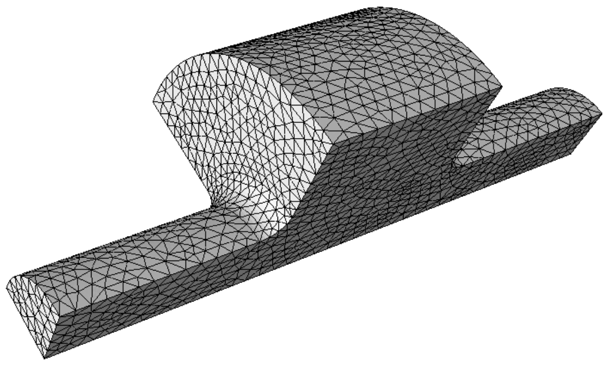


Figure 2: FMS interface example: a curved geometry discretized with a quadratic mesh

The first step after loading the PUMI mesh is to define the topology. FMS requires all mesh entities, i.e. vertices, edges, faces and elements, to be explicitly defined in order to establish the one-level downward adjacencies (the relations element–face, face–edge and edge–vertex). As the PUMI mesh data structure supports one-level downward adjacencies already, this task simplifies to using the PUMI API to loop over each entity type, get the bounding side IDs for each entity and then call the FMS API to construct the corresponding entity. Below is a code segment that writes all triangle faces of an FMS mesh based on their bounding edges.

```
  // Faces
  FmsInt nedges = 3; //Triangle face has 3 edges
  int face[nedges];
  int entDim = 2; // Face dimension
  itr = pumi_mesh->begin(entDim);
  // Loop over PUMI mesh faces
  while ((ent = pumi_mesh->iterate(itr)))
```

```
{
    // Get Edges
    apf::Downward edges;
    pumi_mesh->getDownward(ent, apf::Mesh::EDGE, edges);

    //Get the Edge IDs
    for (FmsInt ed = 0; ed < nedges; ed++){
        face[ed] = apf::getNumber(edge_num_local, edges[ed], 0, 0);
    }

    //Create a face in FmsMesh mesh
    FmsDomainAddEntities(domain[0], FMS_TRIANGLE, 0, FMS_UINT32, face, 1);
}
pumi_mesh->end(itr);
```

Note that `FmsDomainAddEntities()` allows the definition of entity reordering in its third argument. As both PUMI and FMS use the same ordering for faces, i.e. same order in bounding edges, the reordering is set to zero. However, as the codes use different ordering for tetrahedral elements, a `FmsEntityReordering` array needs to be defined. This array describes which face of a tetrahedral element in PUMI matches with which corresponding face of the FMS tetrahedral element:

```
// Define entity reordering for FMS_TETRAHEDRON
FmsEntityReordering EntReord;

// PUMI order for tet element faces
FmsInt nfaces = 4;
int tetOrd[nfaces] = {0,1,3,2};
EntReord[FMS_TETRAHEDRON] = &tetOrd[0];

// Loop over PUMI tetrahedron elements and add
{
  .....
  FmsDomainAddEntities(block[0], FMS_TETRAHEDRON, EntReord, FMS_UINT32, elem, 1);
}
```

Once topology definition is completed, the mesh shape (geometry) is defined using the coordinate field. Each field in FMS has a field descriptor that defines field properties such as continuity, basis type and order of the field. For the coordinate field in this example we choose a second order, continuous field with open nodal Gauss basis:

```
// Define coordinate field
FmsFieldDescriptor CrdFieldDesc;

// Define fixed order field descriptor
FmsFieldDescriptorSetFixedOrder(CrdFieldDesc, FMS_CONTINUOUS, FMS_NODAL_GAUSS_CLOSED, 2)
```

The order of data storage in a field is: first vertex data, followed by data associated with the nodes classified on edges, data associated with the nodes classified on faces, and finally data associated with volume elements. In our example of a quadratic tetrahedral mesh we only have coordinate data associated with vertices and edges.

```
// Vector containing vertices and nodes coordinates
std::vector<double> allPumiCrds;

// Loop over all pumi vertices and add coordinates to allPumiCrds
  ...
// Loop over all pumi edges and add coordinates of node classified on them
  ...
// Set the coordinate field of FmsMesh mesh
FmsFieldSet(FmsCrdField, CrdFieldDesc, numVecComp, FMS_BY_VDIM, FMS_DOUBLE,
            &allPumiCrds)
```

Adding the coordinate field completes the definition of the mesh. The remaining task now is to define possible boundary components and assign material entity tags if needed.

In our elasticity example we are assuming a Dirichlet and a load boundary condition. Each boundary condition is added as a new mesh *component* to FMS, i.e. for the Dirichlet BC we have,

```
FmsComponent Dir_boundary;
FmsMeshAddComponent(mesh, "DirBoundary", &Dir_boundary);
FmsComponentAddPart(Dir_boundary, domain[0], &part);
```

Then the IDs of the corresponding boundary entities are collected. This is done easily in PUMI using reverse classification in which a query of the model face returns mesh faces classified on it, once the IDs are obtained the entities are added to the component as:

```
//Dir_ents contains the FMS_TRIANGLE Dirichlet face IDs
FmsComponentAddPartEntities(Dir_boundary, part, FMS_TRIANGLE, FMS_UINT32,
                            FMS_INT_TYPE, FMS_INT_TYPE,
                            NULL, &Dir_ents, NULL, Dir_cnt);
```

This completes the description of the PUMI mesh in the FMS interface. We next demonstrate, how this mesh can be read in MFEM and used to perform a simple elasticity simulation.

The convertion of the FMS data collection to an MFEM mesh begins with the extraction of the `FmsMesh` from the `FmsDataCollection`, `dc`:

```
// Extract the FmsMesh from the FmsDataCollection
FmsMesh fms_mesh;
FmsDataCollectionGetMesh(dc, &fms_mesh);
```

The second step is to extract the dimension, number of vertices, number of elements, and number of boundary elements in the mesh. Since we assumed a one-domain mesh, we can simply query the first domain in the `FmsMesh`:

```
// Extract the dimension, Dim, number of vertices, NVert and number of
// elements, NElem from domain 0.
FmsDomain *domains;
FmsMeshGetDomains(fms_mesh, 0, &name, &num_domains, &domains);
FmsDomainGetDimension(domains[0], &Dim);
FmsDomainGetNumVertices(domains[0], &NVert);
FmsDomainGetNumEntities(domains[0], FMS_TETRAHEDRON, &NElem);
```

Extracting the number of boundary elements requires reading and adding the number of entries from the mesh components `"DirBoundary"` and `"LoadBoundary"` which have dimension `Dim-1`:

```
NBdrElem = 0;
FmsInt num_comp;
FmsMeshGetNumComponents(fms_mesh, &num_comp);
for (FmsInt i = 0; i < num_comp; i++) {
  FmsComponent comp;
  FmsMeshGetComponent(fms_mesh, i, &comp);
  FmsInt comp_dim;
  FmsComponentGetDimension(comp, &comp_dim);
  if (comp_dim == Dim-1) {
    FmsComponentGetNumEntities(comp, &num_ents);
    NBdrElem += num_ents;
  }
}
```

Next, we can initialize the an `mfem::Mesh` object:

```
// Begin construction of the MFEM mesh.
mfem::Mesh mesh(Dim, NVert, NElem, NBdrElem);
```

After that, we query FMS for the vertices of all tetrahedral entities in domain 0 and push them in the MFEM mesh:

```
FmsInt num_ents;
FmsEntityType et = FMS_TETRAHEDRON;
FmsDomainGetNumEntities(domains[0], et, &num_ents);
mfem::Array<int> ents_verts(num_ents*FmsEntityNumVerts[et]);
FmsDomainGetEntitiesVerts(domains[0], et, NULL, FMS_INT32,
                          0, ents_verts.GetData(), num_ents);
for (FmsInt i = 0; i < num_ents; i++) {
  mesh.AddTet(&ents_verts[4*i]);
}
```

Table 1: Bake-Off Kernel/Problem Summary

	System	Form	BCs	Quadrature Points	Nodal Points
BK1/BP1	$B\underline{u} = \underline{r}$	scalar	homogeneous Neumann	$(p+2)$ GL	$(p+1)$ GLL
BK2/BP2	$B\underline{u}_i = \underline{r}_i$	vector	homogeneous Neumann	$(p+2)$ GL	$(p+1)$ GLL
BK3/BP3	$A\underline{u} = \underline{r}$	scalar	homogeneous Dirichlet	$(p+2)$ GL	$(p+1)$ GLL
BK4/BP4	$A\underline{u}_i = \underline{r}_i$	vector	homogeneous Dirichlet	$(p+2)$ GL	$(p+1)$ GLL
BK5/BP5	$A\underline{u} = \underline{r}$	scalar	homogeneous Dirichlet	$(p+1)$ GLL	$(p+1)$ GLL
BK6/BP6	$A\underline{u}_i = \underline{r}_i$	vector	homogeneous Dirichlet	$(p+1)$ GLL	$(p+1)$ GLL

Proceeding in a similar fashion, we can transfer the boudary elements as well. To complete the transfer, we need to read and convert the mesh coordinates: first we find the unique `Dim`-dimensional component and extract its coordinates field using:

```
FmsField coords;
FmsComponentGetCoordinates(vol_comp, &coords);
```

Then, we extract the data from the `coords` field:

```
FmsInt num_vec_comp;
FmsFieldDescriptor coords_fd;
FmsScalarType coods_data_type;
const void *coords_data;
FmsFieldGet(coords, &coords_fd, &num_vec_comp, NULL, &coods_data_type,
            &coords_data);
```

To interpret the `coords_data`, we can query the field-descriptor `coords_fd` and perform necessary reordering of the `coords_data` which is necessary due to the fact that MFEM generates and uses its own internal numbering for the edges and the faces in the mesh.

3. CEED KERNELS AND BENCHMARKS

Performance test and analysis are central to HPC software development. One of the foundational components of CEED is a sequence of PDE-motivated bake-off problems to establish best practices across a variety of architectures. The main idea is to pool the efforts of several high-order development groups, both internal and external to CEED, in order to identify optimal code optimization strategies for a given architecture. Our first round of tests features comparisons from the CEED projects Nek5000, MFEM, and Holmes, and from the deal.II group in Germany. (The Dune team in Heidelberg has also recently expressed interest in participating.)

In the following sections, we present the bake-off specifications; the first round of results for Nek5000, MFEM, and deal.II using the gcc compiler; updated values using xlc; and very recent results for MFEM using vectorization intrinsics. We also present OCCA-driven Holmes results on Summit, using the Nvidia V100 GPUs on Summit and comparisons of MFEM on EPYC and Skylake processors. In addition to peak performance, we seek to understand *scalable* performance, which means we are concerned with the rate of results produced per node when the amount of work per node is relatively small. Elevating this rate is essential for reducing run time on HPC platforms whenever the problem fits on any subset of the total machine because the user can (and will) in this case use more processors to reduce time to solution.

We note that the MFEM intrinsic developments were motivated by the impressive performance of deal.II. The switch to intrinsics+xlc resulted in a two- to four-fold performance gains for MFEM over the original gcc-only variants. ***We can thus conclude that the original intent of the bake-offs is already being fulfilled: information-sharing is generating performance gains across the spectrum of CEED codes and dependent ECP applications.***

3.1 Bake-Off Specifications

The first suite of problems is focused on run-time performance, measured in degrees-of-freedom per second (DOFs) for bake-off *kernels* (BKs) and bake-off *problems* (BPs). The **BK**s are defined as the application of a local (unassembled) finite-element operator to a scalar or vector field, without the communication overhead associated with assembly. These tests essentially demonstrate the vectorization performance of the PDE operator evaluation (i.e., matrix-free mat-vecs) and provide an upper bound on realizable floating-point performance (MDOFS or MFLOPS) for an application based on such implementations. The **BP**s are mock-up solvers that use the BKs inside a diagonally-preconditioned conjugate gradient (CG) iteration. The point of testing the BKs inside CG is to establish realistic memory-access patterns, to account for communication overhead of assembly (i.e., nearest-neighbor exchange), and to include some form of global communication (i.e., dot products) as a simple surrogate for global coarse-grid solves that will be requisite for exascale linear system solves. To meet these goals in a measurable way, it was decided to run all BPs on 16384 ranks (512 nodes in -c32 mode) of the BG/Q *Cetus* at ALCF. The range of problem sizes was chosen to span from the performance-saturated limit (a lot of work per node) to beyond the strong-scale limit (very little work per node)

To date, there are six BKs and six BPs. Kernels BK1, BK3, and BK5 operate on scalar fields while BK2, BK4, and BK6 are corresponding operators applied to vector fields having three components each. Each BPj, $j = 1, \ldots, 6$ corresponds to CG applied to the assembled BKj system, using a matrix-free implementation if that is faster. The vector-oriented kernels allow amortization of matrix-component memory references across multiple operands, and provide a realistic optimization for vector-based applications such as fluid dynamics (three fields) and electromagnetics (six fields). The vector-based implementations can also benefit from coalesced messaging, thus reducing the overhead of message latency, which is important when running in the fast (strong-scale) limit.

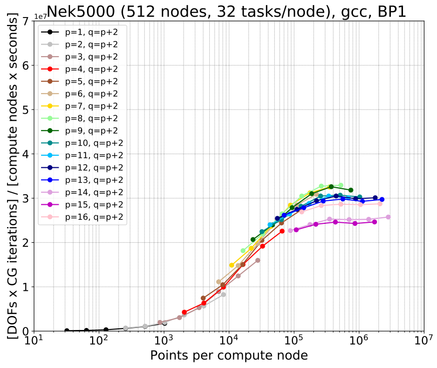
The BPs include solution of the mass matrix, $B\underline{u} = \underline{r}$ (BP1–BP2), and the stiffness matrix, $A\underline{u} = \underline{r}$ (BP3–BP6). Here, $A_{ij} = (\nabla\phi_i, \nabla\phi_j)_q$, and $B_{ij} = (\phi_i, \phi_j)_q$, for nodal basis functions ϕ_i in the standard Q_p approximation spaces in $\mathbb{R}^3$ and $(\cdot,\cdot)_q$ denotes the discrete L^2 inner product based on q points in each direction on the reference domain $\hat{\Omega} := [-1, 1]^3$. Approximation orders are $p = 1, \ldots, 15$, and corresponding quadrature rules for each problem are given in Table 1. There are a total of E elements, $E = 2^{14}$–2^{21}, arranged in a tensor-product array. As the tests are designed to mimic real-world applications, the benchmark codes are to assume that the elements are full curvilinear elements and are not allowed to exploit the global tensor-product structure of the element layout.

3.2 Bake-Off Problems on BGQ

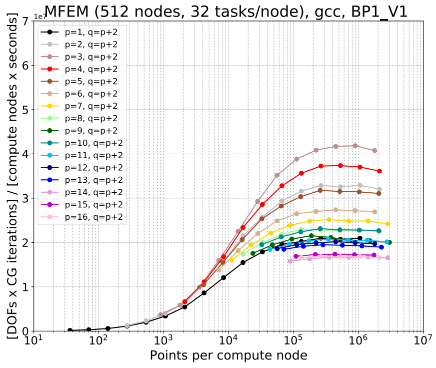
Here we present bake-off results with preliminary test data for Nek5000, MFEM and deal.II on ALCF's BG/Q, *Cetus*, using 512 nodes in -c32 mode (16384 MPI ranks). The initial runs were based on the gcc compiler. On BG/Q, however, the performance of gcc is much lower than the native xlc compiler, so the battery of tests was rerun using xlc, save for deal.II, which at the time of this report still is not linking properly with the xlc compiler on Cetus.

We measure the rate of work in DOFs-per-second (DOFS) or more frequently in millions-of-DOFs-per-second (MDOFS, or *megadofs*). Two of the principal metrics of interest are the peak rate of work per unit resource, $r_{\max}$, and $n_{\frac{1}{2}}$—the local problem size on the node required to realize one-half of the peak rate of work per unit resource. Note that $n_{\frac{1}{2}}$ is defined in terms of points. The importance of $n_{\frac{1}{2}}$, $r_{\max}$, and their scaled ratio, $t_{\frac{1}{2}} = 2n_{\frac{1}{2}}/r_{\max}$, is discussed below. All results are plotted as the rate-of-work per unit-resource, given by [DOFs $\times$ CG Iterations] / [compute nodes $\times$ seconds], versus the number of [Points per compute node]. (To simplify the notation, we will simply refer to the performance variable–the y axis–as MDOFS in the text below.) Choosing number of points (rather than number of DOFs) as the independent variable on the x-axis leads to a data collapse in the case where the systems are solved independently: one obtains a single curve for any number of independent systems. When the systems are solved simultaneously, as in BP2, BP4, and BP6, benefits such as increased data-reuse or amortized messaging overhead should manifest as shifts up-and-to-the-left in the performance curves. We note that, for BP1, BP3, and BP5, points and DOFs are the same.

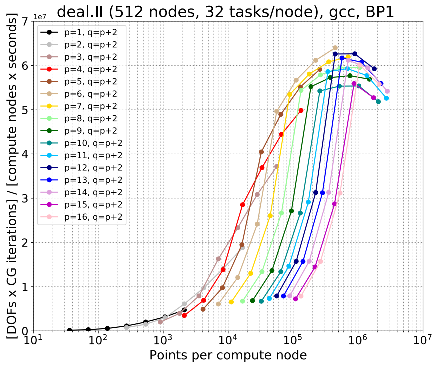
Figures 3–11 present the BP results using the gcc and xlc compilers for Nek5000, MFEM, and deal.II. On each figure, each line represents a different polynomial order. In all cases, performance is strongly tied the number of gridpoints, n. In the case of the gcc compilers, Nek5000 and MFEM generally exhibit a performance plateau as n increases whereas deal.II shows a distinct peak. In the discussion that follows, we focus primarily on the saturated (i.e., peak observable) performance towards the right side of the graphs. On the left side, performance levels drop off to uninteresting values that users would never see. This low



(a) BP1 Nek5000

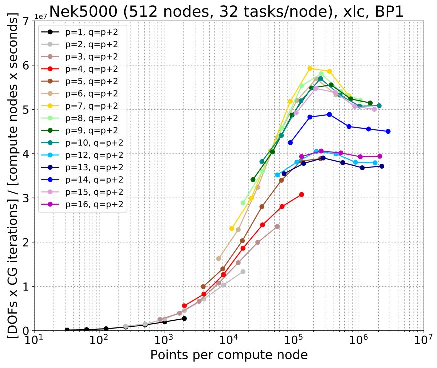


(b) BP1 MFEM

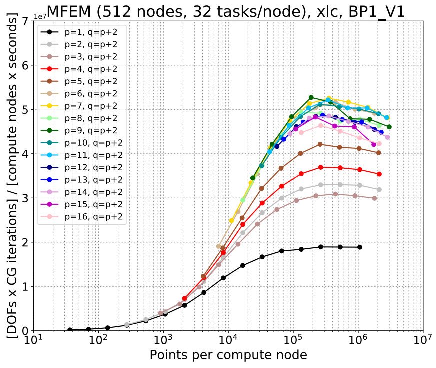


(c) BP1 deal.II

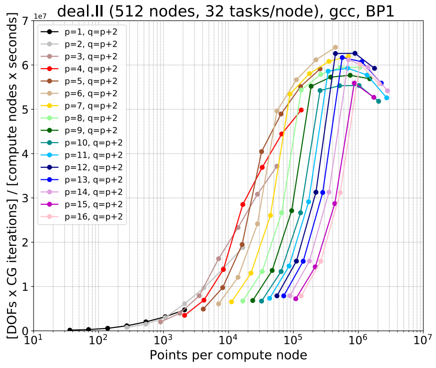
Figure 3: BP1 results of Nek5000 (left), MFEM (center), and deal.ii (right) on BG/Q with varying polynomial order ($p = 1, ..., 16$) with the numbder of quadrature points ($q = p + 2$). The number cpu cores $P = 8,192$.



(a) BP1 Nek5000



(b) BP1 MFEM



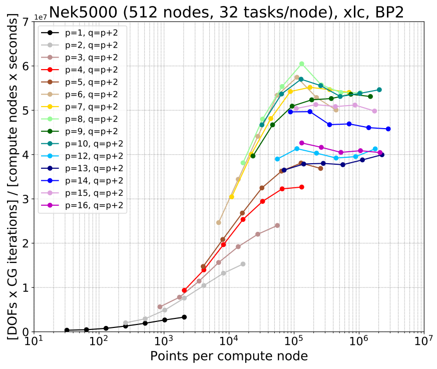
(c) BP1 deal.II

Figure 4: BP1 results of Nek5000 (left), MFEM (center), and deal.ii (right) on BG/Q with varying polynomial order ($p = 1, ..., 16$) with the numbder of quadrature points ($q = p + 2$). The number cpu cores $P = 8,192$.

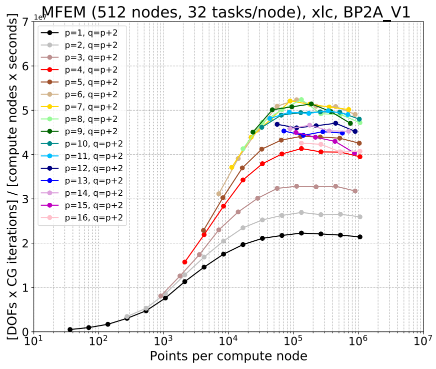
performance regime corresponds to relatively few points per node and is easily avoided on distributed-memory platforms by using fewer processors. While the definition is not precise, the point of rapid performance roll-off represents the *strong-scale limit* to which most users will gravitate in order to reduce time-to-solution. This transition point is thus arguably the most significant part of the domain and its identification is an important part of the BP exercise. A convenient demarcation is $n_{\frac{1}{2}}$, which indicates the number of points per node where the performance is one-half the realizable peak, though most users will choose a number of processors such that $n > n_{\frac{1}{2}}$ in order to realize better than 50% efficiency.

3.2.1 BP1–BP2

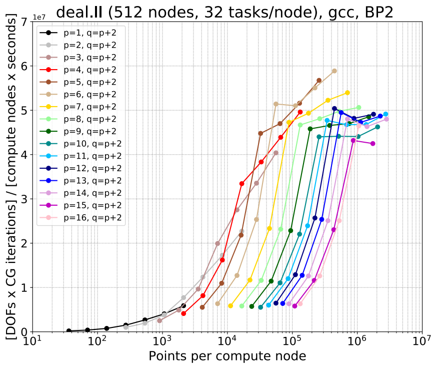
Figures 3–5 present the BP results for the mass-matrix problem, BP1 and BP2. Figure 3 uses the gcc compilers, while Figs. 4–5 are based on xlc for Nek5000 and MFEM. (At this moment deal.II does not link with xlc.) Nek5000+gcc sustains 27-33 MDOFS for polynomial orders $p > 4$, save for $p = 14$ and 15, which saturate around 25 MDOFS. For MFEM, a peak performance of 42 MDOFS is realized for $p = 3$, which corresponds to $4 \times 4 \times 4$ bricks for each element. MFEM realizes > 32 MDOFS for $p = 2$–4, and ≈ 20 MDOFS for the majority of the higher order cases. With gcc, deal.II delivers an impressive 54–64 MDOFS for $p > 4$. The highest values are attained for $n > 450,000$. The $n_{\frac{1}{2}}$ for deal.II is also quite high, however.



(a) BP2 Nek5000

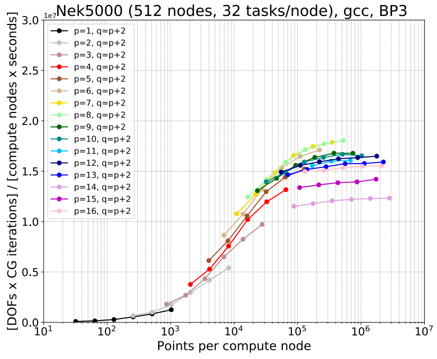


(b) BP2 MFEM

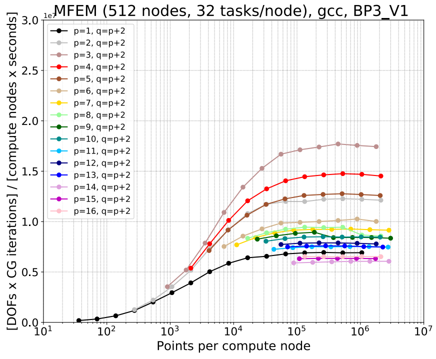


(c) BP2 deal.II

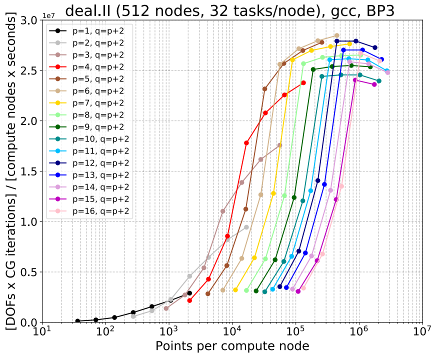
Figure 5: BP2 results of Nek5000 (left), MFEM (center), and deal.ii (right) on BG/Q with varying polynomial order ($p = 1, ..., 16$) with the numbder of quadrature points ($q = p + 2$). The number cpu cores $P = 8,192$.



(a) BP3 Nek5000

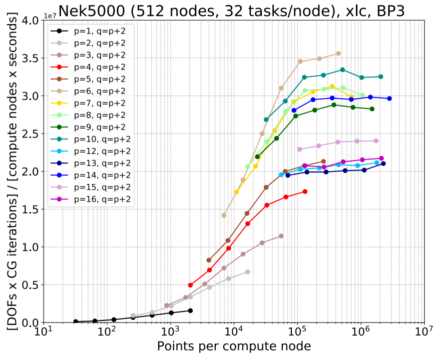


(b) BP3 MFEM

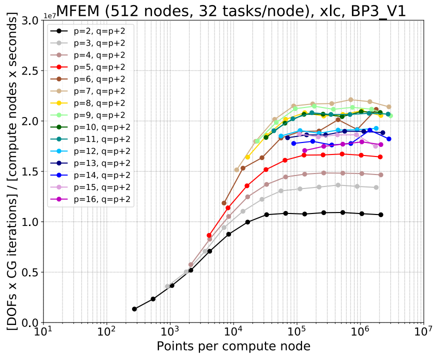


(c) BP3 deal.II

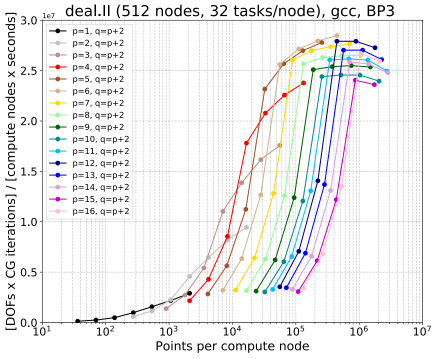
Figure 6: BP3 results of Nek5000 (left), MFEM (center), and deal.ii (right) on BG/Q with varying polynomial order ($p = 1, ..., 16$) with the numbder of quadrature points ($q = p + 2$). The number cpu cores $P = 8,192$.



(a) BP3 Nek5000

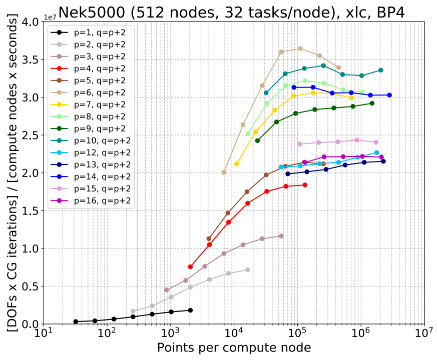


(b) BP3 MFEM

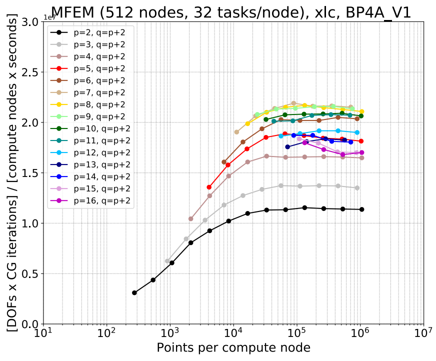


(c) BP3 deal.II

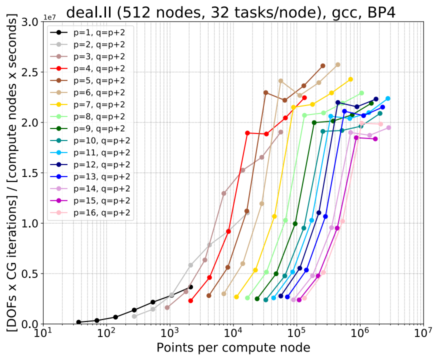
Figure 7: BP3 results of Nek5000 (left), MFEM (center), and deal.ii (right) on BG/Q with varying polynomial order ($p = 1, ..., 16$) with the numbder of quadrature points ($q = p + 2$). The number cpu cores $P = 8,192$.



(a) BP4 Nek5000

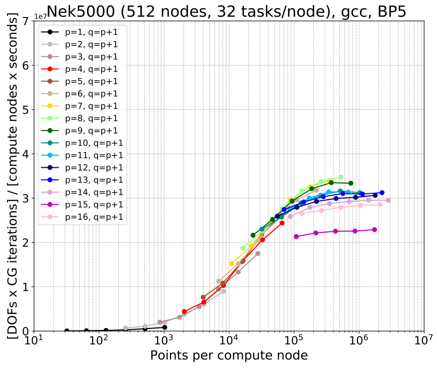


(b) BP4 MFEM

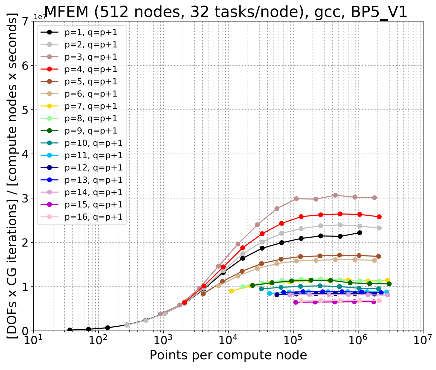


(c) BP4 deal.ii

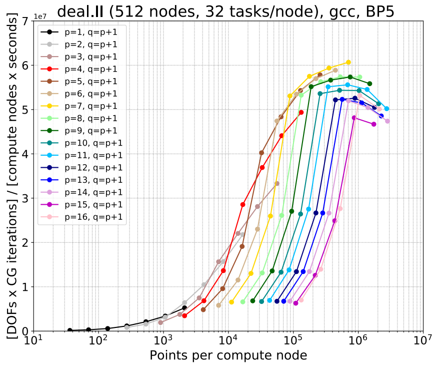
Figure 8: BP4 results of Nek5000 (left), MFEM (center), and deal.ii (right) on BG/Q with varying polynomial order ($p = 1, ..., 16$) with the numbder of quadrature points ($q = p + 2$). The number cpu cores $P = 8,192$.



(a) BP5 Nek5000

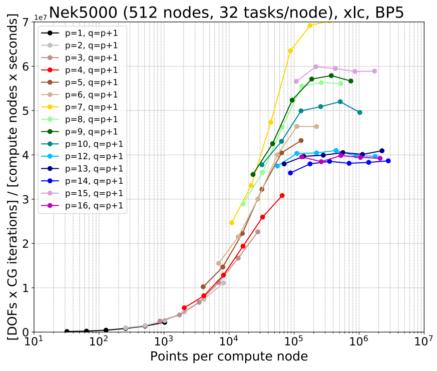


(b) BP5 MFEM

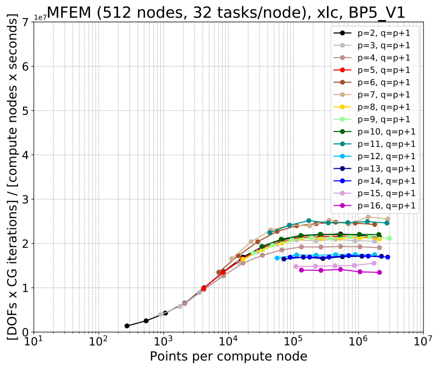


(c) BP5 deal.ii

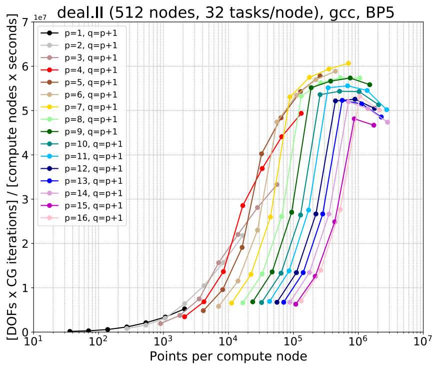
Figure 9: BP5 results of Nek5000 (left), MFEM (center), and deal.ii (right) on BG/Q with varying polynomial order ($p = 1, ..., 16$) with the numbder of quadrature points ($q = p + 1$). The number cpu cores $P = 8,192$.



(a) BP5 Nek5000

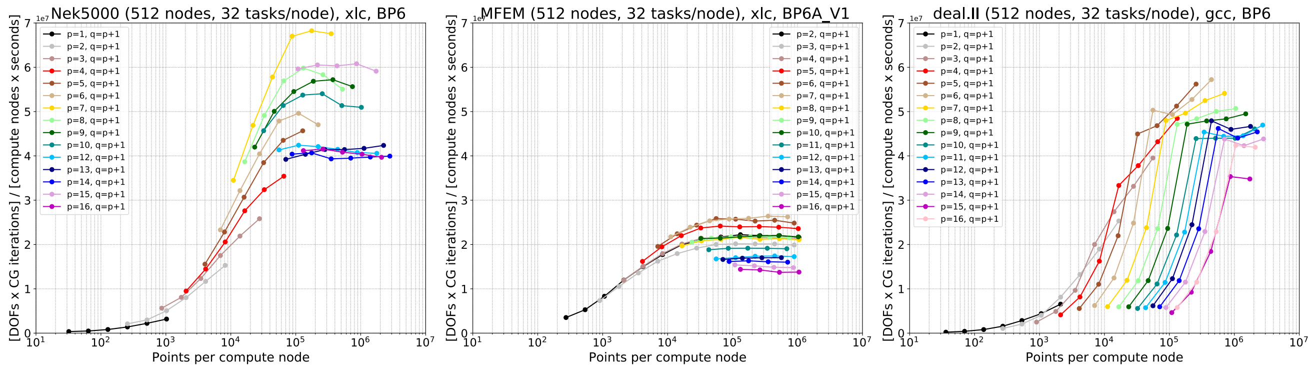


(b) BP5 MFEM



(c) BP5 deal.ii

Figure 10: BP5 results of Nek5000 (left), MFEM (center), and deal.ii (right) on BG/Q with varying polynomial order ($p = 1, ..., 16$) with the numbder of quadrature points ($q = p + 1$). The number cpu cores $P = 8,192$.



(a) BP6 Nek5000 **(b)** BP6 MFEM **(c)** BP6 deal.ii

Figure 11: BP6 results of Nek5000 (left), MFEM (center), and deal.ii (right) on BG/Q with varying polynomial order ($p = 1, ..., 16$) with the numbder of quadrature points ($q = p + 1$). The number cpu cores $P = 8,192$.

For example, for $n = 100,000$, performance is below 30 MDOFS for all $p > 9$. The rapid fall-off is related to the way deal.II distributes elements to MPI ranks. The partitioner insists on having at least 8 elements per rank, rather than insisting on a balanced load. Having 8 elements per rank guarantees that 8-wide vector instructions can be issued for any polynomial order but rails against strong-scale performance.

Figure 4 again shows the BP1 results, but now using the xlc compiler for Nek5000 and MFEM. In addition to the xlc compiler, the Nek5000 results are using assembly-coded matrix-matrix product routines for the tensor contraction. Separate timings (not shown) indicate that most of the performance gains derive from the xlc compiler, with an additional 5 to 30 % coming from the assembly code, depending on p. We see the advantage of the xlc compiler, which boosts the Nek5000 peak to 59 MDOFS for $p = 7$ at $n = 180,000$ and MFEM to 54 MDOFS at $n = 190,000$ for $p = 9$. An interesting observation is that with xlc, $p = 3$ is the lowest performer (30 MDOFS) for MFEM (ignoring $p = 1$), whereas it was the highest (43 MDOFS) with gcc.

For BP2, we consider only the xlc results as shown in Fig. 5. For Nek5000, the peak is now 61 MDOFS ($p = 8$) and there is a significant performance boost in the $n \approx 50,000$ region with $p = 6$–9 realizing $>$ 54 MDOFS. Remarkably, some of the performance curves, such as $p = 15$ show a reduction in peak. For MFEM, the BP2 performance is similar to BP1. The $p = 4$ case is noteworthy in that it shows roughly a 10% performance gain throughout the saturated ($n > 60,000$) regime. BP2-gcc for deal.II leads to a reduction in peak performance, but in some cases (e.g., $p = 5$) there is a shift to the left that indicates more potential for strong scaling.

3.2.2 BP3–BP4

Figures 6–8 present the BP results for the stiffness-matrix problem, $A\underline{u} = \underline{r}$, with integration based on $q = p + 2$ points in each direction for each element. The results are similar to BP1–BP2. With gcc, Nek5000 realizes 15 MDOFS for $p > 5$; MFEM achieves a peak of 18 MDOFS for $p = 4$ and deal.II reaches a peak of 28 MDOFS. With xlc, the peak for Nek5000 reaches 30–35 MDOFS for $p = 6$–8 and 10; MFEM reaches 20–22 MDOFS for $p = 7$–10. For the vector form, BP4-xlc, Nek5000 is boosted to 36 MDOFS for $p = 6$ at $n = 50,000$; MFEM realizes a gain from 18 to 21 MDOFS in moving from BP3 to BP4; and the results are mixed for the deal.II BP4-gcc data.

3.2.3 BP5–BP6

BP5 and BP6 solve a Poisson problem using the standard spectral element stiffness matrix in which quadrature is based on the point set that forms the nodal basis, that is, the Gauss-Lobatto quadrature points. The advantage of this approach is that one only needs to evaluate tensor contractions to apply derivatives and their transposes. No contractions are required for interpolation to quadrature points. The net result is nearly a two-fold increase in MDOFS across all cases. Respectively for Nek5000, MFEM, and deal.II, BP5-gcc realizes peaks of 32, 30, and 60 MDOFS in contrast to 18, 18, and 28 for BP3-gcc. For BP5-xlc, the corresponding peaks are 70 and 25 for Nek5000 and MFEM (something anomalous with MFEM that we are investigating). For BP6, the Nek5000 peak at 68 MDOFS is slightly lower, but is realized for $n = 90,000$.

A particularly salient point, is that Nek5000 BP6-xlc realizes 58 MDOFS for $n = 44,000$ whereas deal.II achieves 58 MDOFS for $n = 180,000$. The net result is that, for the same FLOPS per node, whether measured in total energy consumed or node hours spent, the ability to strong scale implies that the Nek5000 implementation will run **fully 4× faster.** The importance of low $n_{\frac{1}{2}}$ values cannot be underestimated in an HPC environment.

The point of the proceeding argument is not to argue that one code is superior to another. In fact, we do not yet have the xlc data for deal.II and expect that it could improved substantially. The point is to stress the importance of low $n_{\frac{1}{2}}$ in reducing time to solution. In fact, it is a straightforward exercise to show that, for a given set of parameters, time to solution at fixed cost (e.g., charged node hours or energy consumed) is governed by an equation of the following form,

$$T_{\text{wall-clock}} \quad = \quad C\frac{n_{\frac{1}{2}}}{S}, \tag{1}$$

where C is a problem dependent constant, S is the peak realizable speed (e.g., the MDOFS shown in Figs. (3–11), and $n_{\frac{1}{2}}$ is a value of n where performance matches the user's tolerable efficiency level. The smaller the value of $n_{\frac{1}{2}}$ (at fixed efficiency), the more processors that can be used and the faster the calculation will run.

We further remark on some of the cross-code performance variations. One of the optimizations used by deal.II is to exploit the bilateral symmetry of the GL and GLL points to cut the number of operations in the tensor contraction by a factor of two using an even-odd decomposition [4]. The other, as previously mentioned, organizes the data into 8-element blocks to combine favorable vector sizes which helps in realizing improved peak ("S" in (1)) but inhibits strong scaling ("$n_{\frac{1}{2}}$" in (1)). Moreover, as the objective of CEED is to develop poly-algorithmic back-ends that will deliver the best performance to the end users, our objective is to realize the optimal hull of performance over the various implementations. All scaling analysis, therefore, must be based on the best-realized values, not on the values realized by a particular implementation. Thus, $n_{\frac{1}{2}}$ must be based on the peak values such as achieved by deal.II when discussing the prospective performance envelope.

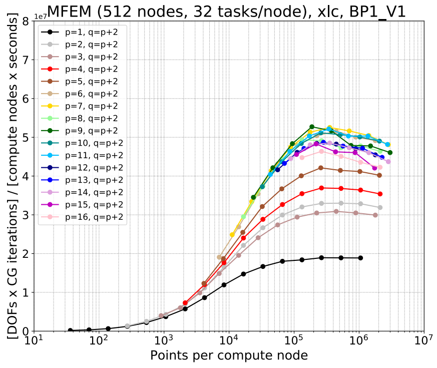
3.2.4 Improvements Due to the Bakeoff Runs

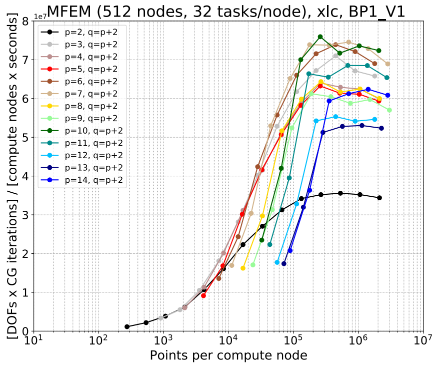
Motivated by the deal.ii results the CEED team added explicit vectorization over the elements to the MFEM code. Coupled with intrinsics, these resulted in a two- to four-fold speed-up over the original gcc-based results and nearly a 50% speed-up attained with xlc alone. These results are very recent and are still being analyzed. **The major point of this story, however, is that cooperation among the BP participants has resulted in performance gains across the board**. The bake-offs are a dynamic process, rather than a static benchmark and we fully expect that they will continue to contribute to performance gains as exascale platforms are deployed. Results for BP1 are presented in Figure 12.

3.3 Bakeoff Kernels on Summit

With multinode scaling issues addressed through the BP studies, we turn now to node scaling on next-generation accelerator-based architectures. Specifically, we consider the performance of BK5 implemented on the NVIDIA V100 core on Summit using OCCA ([7]). In order to understand performance tuning on the V100, we presently consider only the kernel (BK5), not the full miniapp (BP5), which means we are ignoring communication and device-to-host transfer costs. Clearly, these issues will be important in the final analysis. Our intent here is to understand the potential and the limits of GPU-based implementations of the SEM. We seek to understand what is required to get a significant fraction of the V100 peak performance in the context of a distributed memory parallel computing architecture such as Summit.

The BK5 performance kernel amounts to evaluating the matrix-vector product $\underline{w}_L = A_L\underline{u}_L$, where A_L=block-diag(A^e), $e = 1, \ldots, E$. This kernel is fully parallel because A_L represents the *unassembled* stiffness matrix. The local matrix vector products, $\underline{w}^e := A^e\underline{u}^e$ are implemented in the matrix free form outlined in


(a) BP1 MFEM-before


(b) BP1 MFEM-after

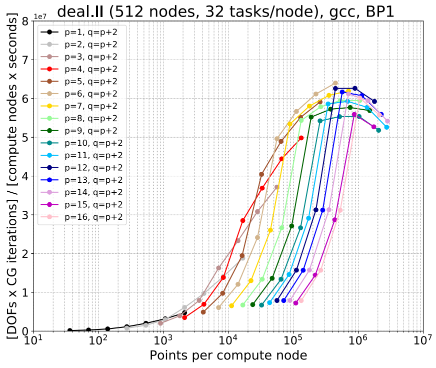

(c) BP1 deal.II

Figure 12: BP1 results of MFEM+xlc (left), MFEM+xlc+intrinsics (center), and deal.ii (right) on BG/Q with varying polynomial order ($p = 1, ..., 16$) with the number of quadrature points ($q = p + 2$). The number cpu cores $P = 8,192$.

(4.4.7) of [3],

$$\underline{w}^e = \begin{pmatrix} D_1 \\ D_2 \\ D_3 \end{pmatrix}^T \begin{pmatrix} G^e_{11} & G^e_{12} & G^e_{13} \\ G^e_{21} & G^e_{22} & G^e_{23} \\ G^e_{31} & G^e_{32} & G^e_{33} \end{pmatrix} \begin{pmatrix} D_1 \\ D_2 \\ D_3 \end{pmatrix} \underline{u}^e. \tag{2}$$

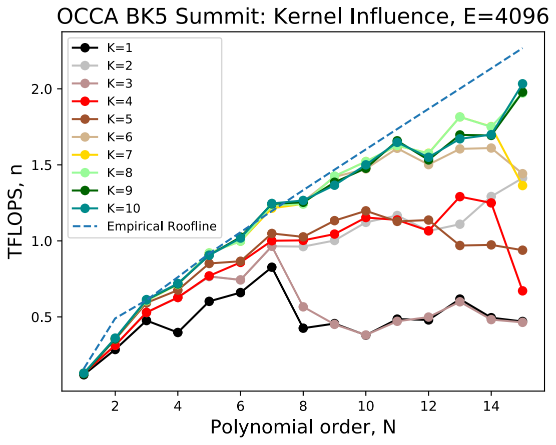
Here, the derivative matrices involve tensor products of the $(N_q \times N_q)$ identity $(\hat{I})$ and derivative $(\hat{D})$ matrices: $D_1 = \hat{I} \otimes \hat{I} \otimes \hat{D}$, $D_2 = \hat{I} \otimes \hat{D} \otimes \hat{I}$, $D_3 = \hat{D} \otimes \hat{I} \otimes \hat{I}$, where $N_q = N + 1$ is the number of nodal points in each direction within an element. The geometric factors are diagonal matrices of size $N_q^3 \times N_q^3$. $\mathbf{G}$ is a symmetric tensor, $G_{ij} = G_{ji}$, so only $6N_q^3$ memory references are required per element, in addition to the N_q^3 reads required to load $\underline{u}^e$. $\hat{D}$ requires only N_q^2 reads, which are amortized over multiple elements and therefore discounted in the analysis. For E elements, the work complexity for (2) is $W = 12N_q^4 + 15N_q^3$, with the leading order $O(N_q^4)$ term comprising tensor contractions that can be cast as efficient BLAS3 kernels.

Figure 13a shows the performance for (2) on a single GPU core of Summit through a sequence of OCCA tuning steps. The number of elements is $E = 4096$ and the polynomial order N varies from 1 to 15 (N_q=2 to 16). Each kernel is run for multiple times and the time for the kernel is taken by dividing total time by the total number of iterations. This is done to smooth out the noise and to be sure that we are not misguided by the clock resolutions on different systems. We note in particular that $N_q = 8$ and 16 see significant performance gains for $K8$.

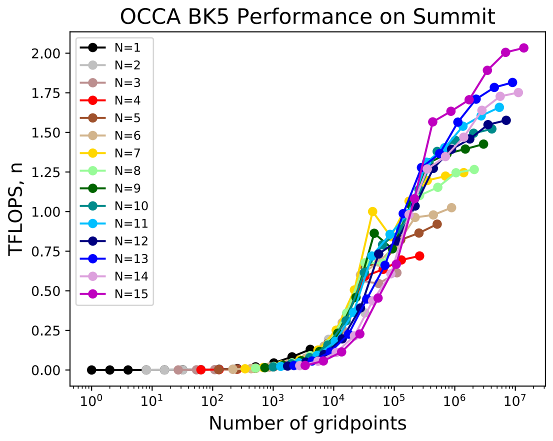
The tuning curves of Fig. 13a were for the case $E = 4096$, which is the largest we considered. On leadership class platforms, users are typically interested in reduced time-to-solution for any given problem (i.e., of fixed global size, n_g), which means they will increase the node (i.e., GPU) count, P, to the point where parallel efficiency reaches an intolerable level (e.g., $< 70\%$). Under this strong-scaling scenario, the local number of points per node, $n = n_g/P$, is reduced. There is ultimately a point of marginal return where each GPU lacks sufficient work to keep busy (or to offset communication). To understand this strong-scale limit, we performed a sequence of timings for $E = 1, 2, 4, \ldots, 4096$, and analyze the data as a function of the local number of gridpoints (for a single GPU), $n := EN^3$.

Figure 13b shows the V100 TFLOPS for BK5 as a function of n for $N = 1$ to 15. The tight data collapse demonstrates that n is a leading indicator of performance, but the polynomial order is also seen to be important, with $N = 15$ realizing a peak of 2 TFLOPS. Cursory inspection of Fig. 13b indicates that $n_{\frac{1}{2}}$ for this kernel is roughly 10^5, implying that a simulation with n_g points could realize a speedup of at most $\frac{1}{2}n_g/10^5$ at the 50% parallel efficiency level. We take a closer look at this question in the next two plots.

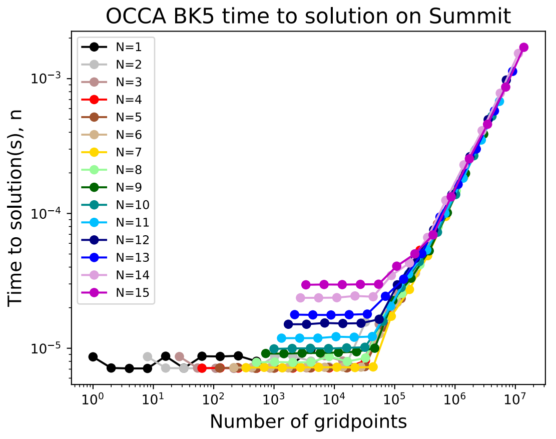
Figure 13c shows the execution time for the cases of Fig. 13b. The expected linear dependence is evident at the right side of these graphs while, to the left, the execution time approaches a constant as $n \longrightarrow 1$. Interestingly, the execution time in the linear regime is very weakly dependent on N, which typically argues



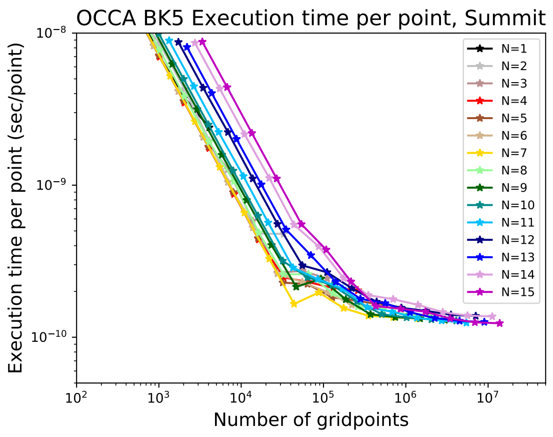
(a) BK5 Tuning



(b) BK5 Tflops



(c) BK5 Timings



(d) BK5 Timings per point

Figure 13: (a) TFLOPS for different kernel tunings. (b) TFLOPS versus problem size n for different polynomial orders, N. (c) Execution time versus n for varying N. (d) Execution time per point versus number of points, n.

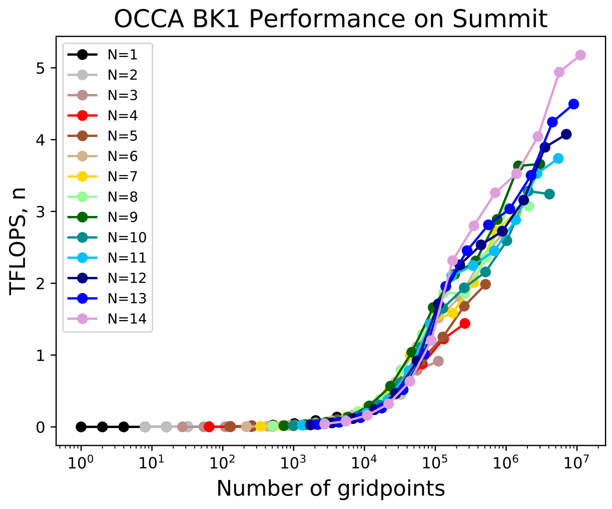
for larger values of N (and hence more accuracy) and correspondingly smaller values of E such that the resolution requirements (n_g) are met for the given simulation. Other considerations, such as meshing or timestep size, inhibit using excessively high polynomial orders and experience has shown $N = 5$ to 9 to be most practical for production simulations.

With this background, deeper insight to the (N, E, P) performance trade-offs can be gained by looking at the execution time per point, shown in Fig. 13d, which also shows the minimal time and the $2\times$ line, which is twice the minimal execution time per point. For a fixed total problem size, n_g, moving horizontally on this plot corresponds to reducing n and increasing P such that $n_g = P\,n$. In the absence of (yet to be included) communication overhead, one gains a full P-fold reduction in the execution if the time per point does not increase when moving to the left. We see in this plot that $N = 7$ appears to offer the best potential for high performance, where even at $n = 30,000$ the execution time per point is within a small multiple of the minimum realized over all cases. This low value of n is in sharp contrast with the $N = 14$ and 15 cases, which cross the $2\times$ line at $n = 200,000$. Thus, through additional inter-node parallelism, the $N = 7$ case affords a potential $200/30 \approx 7$-fold performance gain over the larger N cases.

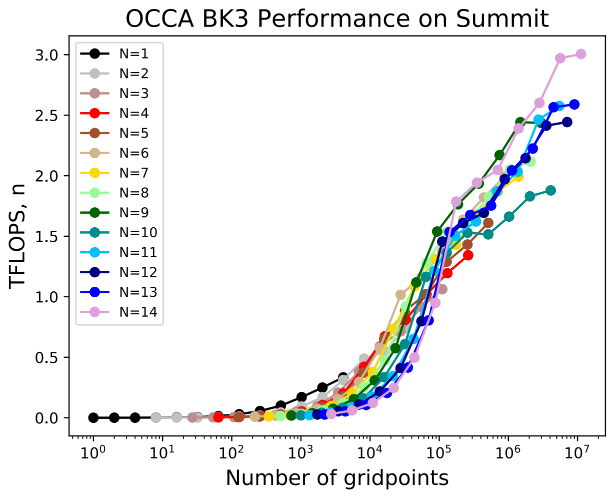
We note that the conclusion that $N = 7$ is superior to $N = 15$ in this context runs contrary to standard intuition, which suggests that, with an $O(N)$ flop-to-byte ratio for (2), performance would be best served by *increasing* N. Such a conclusion would in fact be correct in the saturated limit, where N-independent time is seen at the right of the graphs in Fig. 13c. In an *HPC* context, however, we are interested in running problems that are as *small as possible on each node.* We can see in Fig. 13c that the low-N curves continue to trend downward when the large-N curves flatten out as n is reduced. Thus, in the strong scale limit, a modest value of N proves to be the most scalable.

Naturally, this analysis needs to be repeated to include other parts of the Navier-Stokes solver, including host-device transfers and internode communication, but the analysis approach will be essentially the same. One needs to understand where the strong-scale limits are to make informed discretization and algorithmic choices.

OCCA results on Summit for BK1 and BK3 are shown in Figure 14.



(a) BK1 Summit



(b) BK3 Summit

Figure 14: BK1 and BK3 V100 performance: TFLOPS versus problem size n for different polynomial orders, N.

3.4 AMD EPYC vs. Intel Skylake Performance

The MFEM benchmark from the suite of CEED benchmarks (`https://github.com/CEED/benchmarks/tree/master/tests/mfem_bps`) was executed on dual socket, 48 core, 96 thread, Intel Skylake-SP Xeon Platinum 8160 and AMD EPYC 7451 nodes. These nodes represent the leading x86-compatible processors. The IBM Power9 and ARM processors are excluded from this comparison. Note that, as of this report, the retail cost of the Skylake processor we used is about 60% more than that of the EPYC processor.

Testing indicates that Skylake is up to 3.5 times faster than EPYC (excluding one apparent outlier at 96 processes for the $p = 1, q = p+2$ case). For problems with over 100 thousand points, the Skylake advantage is at most 60%. At problem sizes over 10 million points using one or two processes per core, EPYC outperforms Skylake by about 10%.

A strong scaling test of MFEM across many nodes would be necessary to determine the range of problem sizes for maximum efficiency and time to solution. The largest problem in the benchmark consumes less than 20% of the available memory on the Skylake and EPYC nodes. Using the majority of available memory would be required to study weak scaling performance on large problems. Given the trends in this test, the EPYC system would be better suited to this use case.

Figure 15 shows the performance ratio of Skylake to EPYC on a mesh with up to 25 million hexahedral elements. The left column of plots show the entire range of evaluated problem sizes. The vertical axis in the following plots is the ratio of Skylake, `stampede2`, to EPYC, `ipa`, performance. The horizontal axis is the size of the linear system being solved. The problem size is based on the number of entities in the mesh and the amount of work (flops) per mesh entity; higher values of p have more work per entity. Figure 16

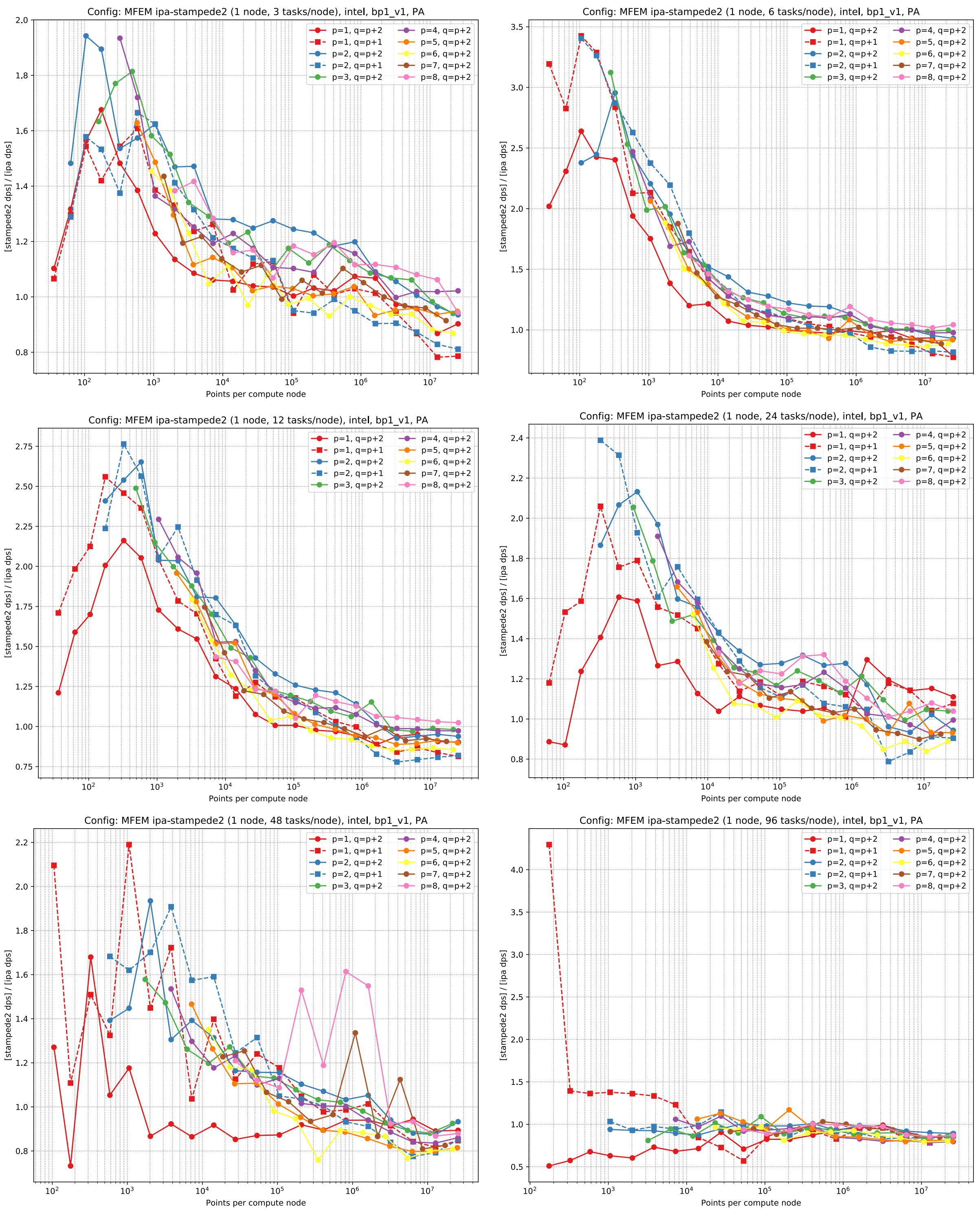


Figure 15: MFEM BP1 Skylake-to-EPYC performance ratio versus problem size.

shows performance versus basis function order. Both systems reach a peak of around a billion DOFs $*$ CG iterations per second (1 GDOF).

The 2018 Intel compilers, with the `-O3` compilation flag, and MPI implementation were used for all MFEM

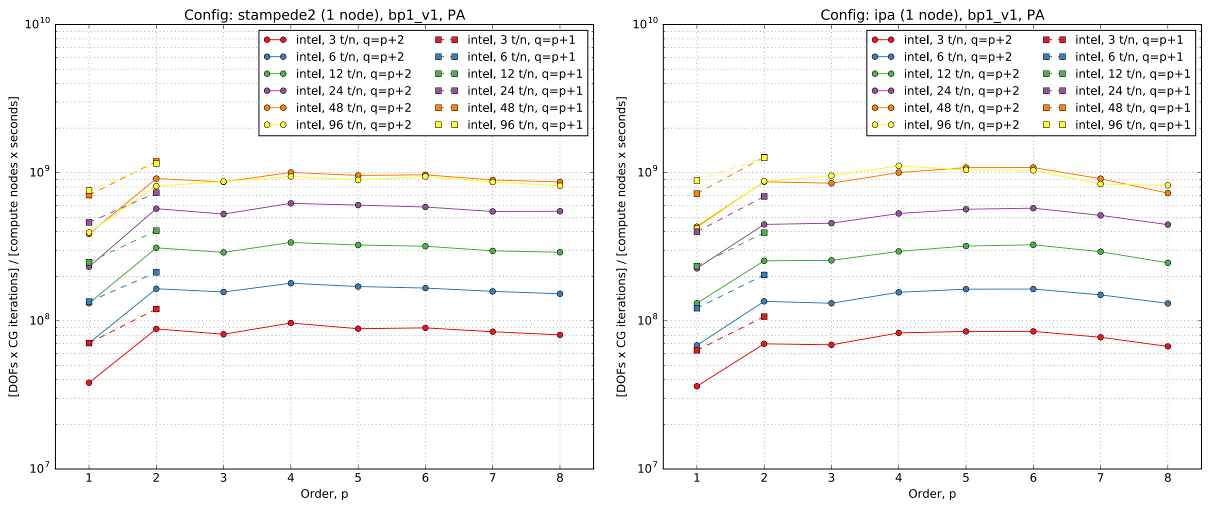


Figure 16: MFEM BP1 performance on Skylake and EPYC versus basis function order.

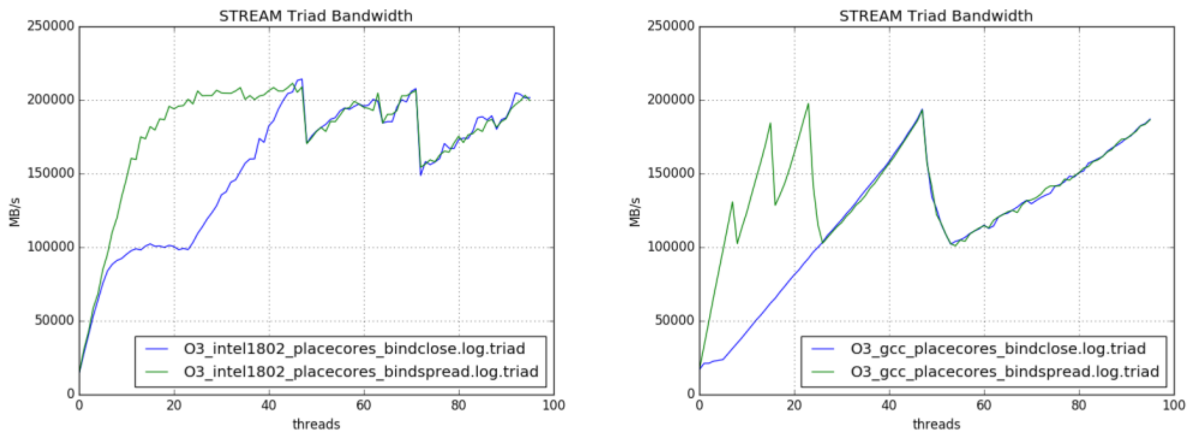


Figure 17: STREAM triad memory bandwidth using two different affinity settings for thread placement on EPYC (left) and Skylake (right).

tests. GCC 7.1, and AOCC 1.2 produced lower performance on EPYC, especially at smaller problem sizes.

The OpenMP variant of the STREAM triad benchmark was also executed. Skylake achieves a peak memory bandwidth of 211 GB/s and EPYC reaches 197 GB/s; a 7% difference. Figure 17 shows the bandwidth on EPYC and Skylake. The GCC 6.1 compiler and 2018 Intel compiler with flags `-O3 -fopenmp -D_OPENMP` were used to compile the STREAM on EPYC and Skylake, respectively.

Additional details, plots, scripts, and run logs from the MFEM BP1 and STREAM tests are available on GitHub: `https://github.com/cwsmith/epyc_vs_skylake`.

4. CEED LIBRARIES AND APPLICATIONS

4.1 Improvements in the CEED API library, libCEED

The CEED API library, libCEED, was released in milestone CEED-MS10 as a lightweight portable library that allows a wide variety of ECP applications to share highly optimized discretization kernels.

A main component of the CEED 1.0 effort, was the continued improvement of libCEED and specifically improving the performance of the MAGMA backend and redesigning the API for operators and qfunctions to facilitate usability and increase robustness.

Active/Passive API

The libCEED API was updated for operators and qfunctions. The opaque qdata field was replaced with the ability to add input and output fields to the qfunctions and operators individually. The example below demonstrates the new API:

```
// Create a QFunction for the mass matrix
CeedQFunctionCreateInterior(ceed, 1, mass, __FILE__ ":mass", &qf_mass);
CeedQFunctionAddInput(qf_mass, "rho", 1, CEED_EVAL_NONE);
CeedQFunctionAddInput(qf_mass, "u", 1, CEED_EVAL_INTERP);
CeedQFunctionAddOutput(qf_mass, "v", 1, CEED_EVAL_INTERP);

// Create an operator for the mass matrix
CeedOperatorCreate(ceed, qf_mass, NULL, NULL, &op_mass);
CeedOperatorSetField(op_mass, "rho", CEED_RESTRICTION_IDENTITY,
                     CEED_BASIS_COLOCATED, rho);
CeedOperatorSetField(op_mass, "u", Erestrictu, bu, CEED_VECTOR_ACTIVE);
CeedOperatorSetField(op_mass, "v", Erestrictu, bu, CEED_VECTOR_ACTIVE);

CeedOperatorApply(op_mass, u, v, CEED_REQUEST_IMMEDIATE);
```

Inputs and outputs are added to qfunctions with and associated field name, dimension, and basis evaluation operation. Fields are added to operators with an associated field name, restriction, basis, and vector.

This new API will facilitate multiphysics coupling. For instance, with a coupled velocity-pressure operator for incompressible flow a single active input vector could be used in multiple operator fields with different restrictions to give both pressure and velocity inputs to the qfunction. Previously, only one basis and restriction could be associated with each operator.

Additionally, this API change allows users to clearly update additional outputs from a qfunction. Modifications to the qdata were less visible in the previous API.

Overall, this redesign of the API is more extensible, provides clearer and more self-documenting code, and is less brittle. Furthermore, it will facilitate future development of the library capabilities and enable composition of operators.

4.2 Application Engagements

In this section we report on the progress in the CEED team engagement with the ExaSMR and Urban ECP applications, We address the algorithmic development and performance improvements, related to the needs of these applications, as well as the potential collaborations extended to the E3SM and ExaWind teams.

4.2.1 ExaSMR

As is the case with many of the exascale applications, ExaSMR is characterized by a wide range of temporal and spatial scales that require modeling to be realistically surmountable, even in the context of an exascale computing platform. For the thermal-hydraulics analysis, hundreds of thousands of flow channels comprise turbulent flow with very fine solution scales. The channels are typically hundreds of hydraulic diameters in length. For full reactor-core simulations, the ExaSMR strategy is to use Reynolds-Averaged Navier Stokes (RANS) in the majority of the core with more detailed large eddy simulations (LES) in critical regions. In addition, while the turbulence is challenging to resolve, it tends to reach a statistically fully-developed state within just a few channel diameters, whereas thermal variations take place over the full core size.

Jacobian-free Newton Krylov Method Implementation into Nek5000. To accelerate the time to solution, CEED is assisting the ExaSMR team in developing fully implicit and steady state solvers for thermal transport and RANS. For thermal transport, we have imported the Jacobian-free Newton Krylov (JFNK) routines from NekCEM drift-diffusion solver [12] to Nek5000 and performed the prelimiary tests for the flow problems demonstrated in Figure 18. Figure 19 shows fast converging to the steady state solutions with the small number of pseudo-time steps (left). Our approach includes an inexact formulation representing the action for the Jacobian matrix-vector multiplication that involves GMRES iterations during the Newton

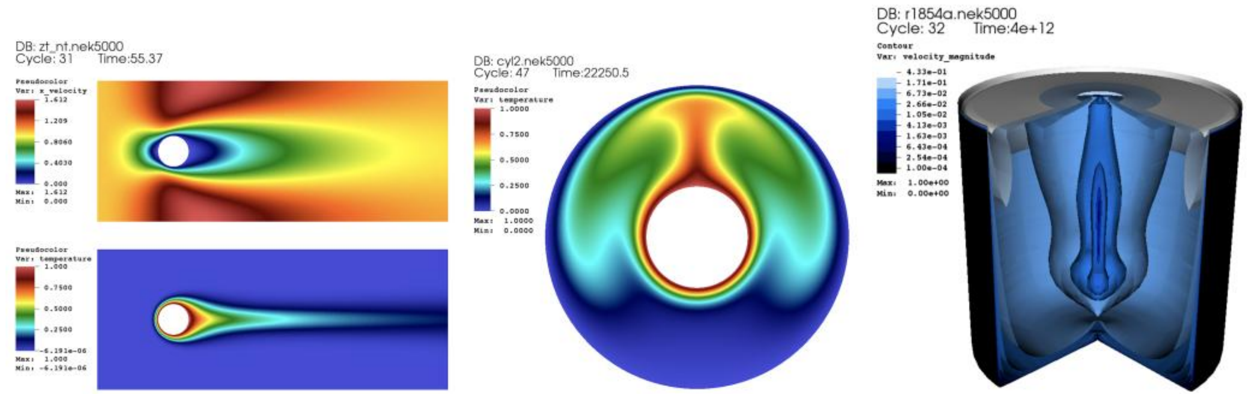


Figure 18: Jacobian-free Newton Krylov pseudo-time stepping, applied for conjugate heat transfer for cylinder, annulus and vortex problems for both $P_N - P_N$ and $P_N - P_{N-2}$ formulations of Nek5000.

iteration step. The iteration counts for Newton and GMRES in each pseudo-time step are demonstrated (right). In order to reduce the GMRES iterations for the convection-diffusion operator, we are currently working on preconditioning techniques involving overlapping Schwarz, tensor-product preconditioners, and spectral-element multigrid.

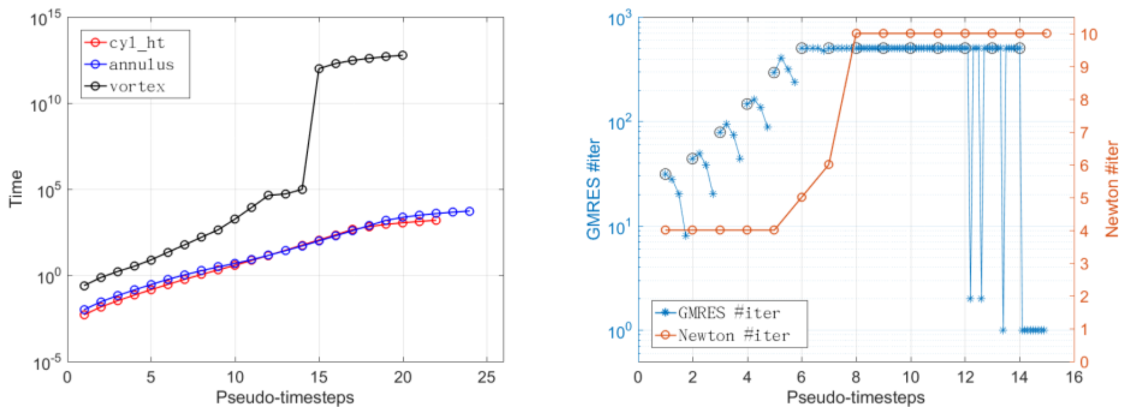


Figure 19: Jacobian-free Newton Krylov pseudo-time steppings, converging to steady-state solutions for cylinder, annulus and vortex problems while demonstrating how the physical time (log scale) increases in each pseudo-time step (left). Newton iteration counts per pseudo-time step and the GMRES iteration counts per each Newton iteration (right) for vortex problem.

RANS Model in Nek5000 with Jacobian-free Newton Krylov Method. We extend the JFNK method for a RANS model [11] that is a 5-equation model, defined as the following with the turbulent kinetic k and the specific dissipation rate ω in addition to the velocity field $\mathbf{v}$, to represent the turbulent properties of the incompressible flows:

$$k = \frac{\langle u'^2 \rangle + \langle v'^2 \rangle + \langle w'^2 \rangle}{2} \tag{3}$$

where u', v' and w' are fluctuation component of velocity vector around the ensemble-averaged mean velocity vector $\mathbf{v} = (u, v, w)$, where

$$\frac{\partial(\rho \mathbf{v})}{\partial t} + \nabla \cdot (\rho \mathbf{v}\mathbf{v}) = -\nabla p + \nabla \cdot \left[(\mu + \mu_t) \left(\nabla \mathbf{v} + \nabla \mathbf{v}^T - \frac{2}{3} \nabla \cdot \mathbf{v} \right) \right] \tag{4}$$

$$\frac{\partial(\rho k)}{\partial t} + \nabla \cdot (\rho k \mathbf{v}) = \nabla \cdot (\Gamma_k \nabla k) + G_k - Y_k + S_k \tag{5}$$

$$\frac{\partial(\rho \omega)}{\partial t} + \nabla \cdot (\rho \omega \mathbf{v}) = \nabla \cdot (\Gamma_\omega \nabla \omega) + G_\omega - Y_\omega + S_\omega, \tag{6}$$

where μ is the molecular viscosity and μ_t is the turbulent viscosity with the continuity equation for incompressible flow

$$\nabla \cdot \mathbf{v} = 0. \tag{7}$$

4.2.2 Urban Systems

Nek5000 Simulations for Goose Island Geometry of Chicago. In collaboration between Urban and CEED teams, Nek5000 simulations were performed for a portion of Goose Island geometry of Chicago. Approximately 1/6 of the target area has been generated as spectral element meshes using Cubit. The initial tests have been conducted with various resolutions on ALCF/Mira. We are currently investigating how to improve the boundary layers near the ground and around the building in order to produce the reference LES solution up to high Reynolds numbers.

Mesh Flexibility with Nek-Nek Overlapping. Next step is to build the various meshes and start increasing the domain size to cover the whole area of Goose Island, improve boundary condition modeling, and run the setup on ALCF/Theta. CEED team's effort will include providing flexibility with mesh generation and design optimization of the city block geometry through Nek-Nek overlapping mesh approach.

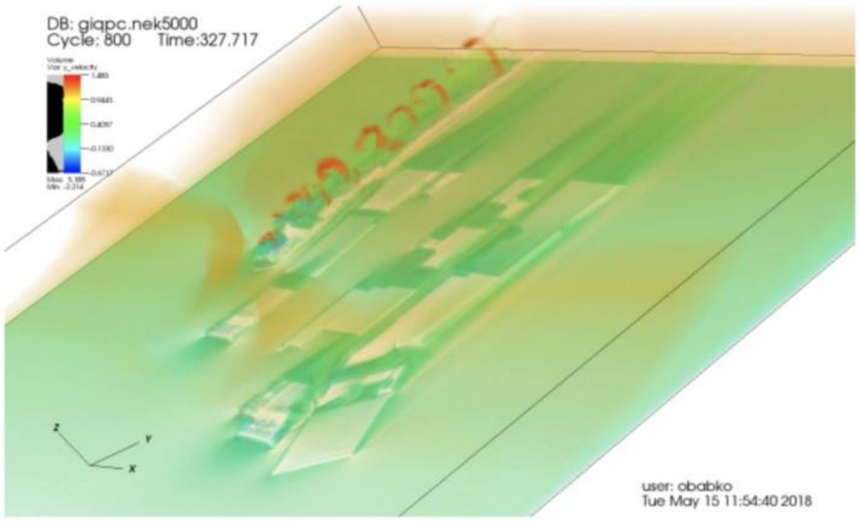


Figure 20: Volume rendering of streamwise velocity from the preliminary Nek5000 LES flow over a portion of Chicagos Goose Island geometry.

4.2.3 Other Applications

We are currently engaging the ExaWind and E3SM teams for potential collaborations. ExaWind has particularly challenging requirements that might be addressed through ongoing developments in Nek5000. In particular, Exawind is exploring LES and RANS models with boundary layer elements that have extremely high ($> 10^3$) aspect ratios. Nek5000 advances in FEM-SEM preconditioners, steady-state and implicit unsteady RANS solvers, and GPU-based solvers have the potential for significant breakthroughs in this area. Concerning E3SM, the CEED team will be participating in a joint mini-symposium with Mark Taylor at the International Conference on Spectral and High-Order Methods (ICOSAHOM 18) next month (July 2018) `http://www.icosahom2018.org`, where we will have a chance to demonstrate the progress made with the bake-off problems and with libCEED as a viable exascale development vehicle.

4.2.4 Nek5000 GPU Porting on OLCF/Summit

Nek5000 supports extended portability to multiple (GPUs), currently using pragma-based OpenACC and CUDA Fortran for local parallelization of the computing-intensive operator evaluations. The Nek5000 communcation library, gslib, `http://github.com/gslib/gslib`, supports all operations (local-gather, global-scatter, global-gather, local-scatter operations) on the GPU with direct communication between remote GPUs using OpenACC pragmas, which eliminates the data movement between GPU and CPU.

Summit/OLCF Early Science Program (ESP) Proposal with ExaSMR. Together with the ExaSMR team, CEED team members (Fischer, Min) have submitted OLCF Summit Early Science program on *Core-Level High Fidelity Simulations: Toward Exascale Simulations of Nuclear Reactor Flows.*

Nek5000 GPU Performance on Summit without Optimization. Over 95% of the operations in Nek5000 consist of tensor contractions, which are effectively implemented through BLAS3 calls on CPUs. On GPUs, further parallelization is needed, as already developed in our OpenACC work [5, 6, 9] with which we began initial porting of Nek5000 on OLCF Summit. Figure 21, shows the results for Nek5000 on Summit vs. number of nodes, with 42 CPUs per node for the CPU curve and 6 GPUs per node for the GPU curve. The case is a 17×17 pin geometry with $E = 221600$ elements of order $N = 11$ ($n = EN^3 = 294,949,600$). It is clear that the GPU runs outperform the CPU case by a factor of two in the peak-performance-per-node limit. Table 2 shows the solution times, parallel efficiency and number of points per rank for the Summit results of Figure 21. We observe in Table 2(a) superlinear speedup in the CPU case because of performance variance for the 20-node run, which can result from less cache reuse or system noise. For $n/P \approx 70,000$, the CPU continues to deliver order unity efficiency. Table 2(b) reveals that the efficiency for the GPUs is $\approx 60\%$ parallel when $n/P \approx 500,000$, which is also in keeping with expectations, given the standard performance fall-off for GPUs, which have a relatively high $n_{\frac{1}{2}}$.

4.2.5 GPU Support Progress of Nek5000 + libCEED

Plans towards Integration of Nek5000 with libCEED. Current steps towards enhanced GPU performance include incorporation of the libCEED library. Nek5000 will utilize libCEED for further performance

# of Nodes on Summit	Time (seconds)	Efficiency on CPUs	n/P
20	991	1.00	351130
40	415	1.19	175565
60	268	1.23	117043
80	203	1.22	87782
100	172	1.15	70226

(a) CPU Performance (42 Cores)

# of Nodes on Summit	Time (seconds)	Efficiency on GPUs	n/P
20	452	1.00	2457913
40	270	0.84	1228956
60	207	0.73	819304
80	170	0.66	614478
100	152	0.59	495182

(b) GPU Performance (6 GPUs)

Table 2: Strong Scalings on Summit.

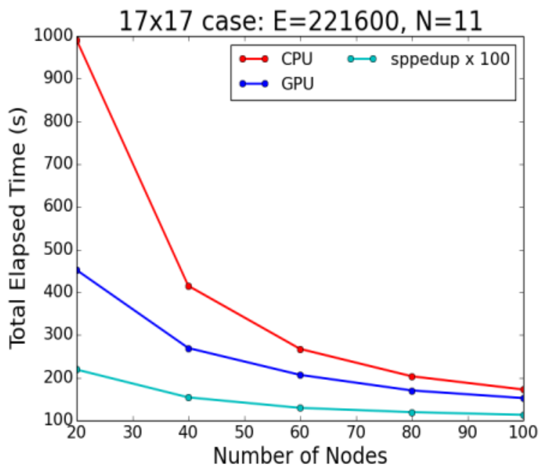


Figure 21: Nek5000 strong-scale results on OLCF/Summit CPUs & GPUs.

improvement. libCEED provides multiple back-ends dedicated to performant hardware, as well as support for heterogeneous MPI configurations, which on Summit would include some cores driving GPUs and some acting as compute cores.

Current Stage of Nek5000 Integration with libCEED. We initiated Nek5000 integration with libCEED using BP1 example which is inline with MFEM and PETSc. We extended the example to BP3, and currently these changes are to be merged with master branch. We note that Nek5000 stores its local data in E-vector format which does not require the conversion step from L-vector to E-vector when applying a `CeedOperator`.

5. OTHER PROJECT ACTIVITIES

5.1 OCCA v1.0 Release

OCCA v1.0 was released with many new features, including:

- Updated API to expose backend-specific features in a generic way.
- New OKL Parser better suited for language transforms and error handling.

See `https://github.com/libocca/occa` for more details. A number of CEED packages, including libCEED and libParanumal are being updated to use the new features.

5.2 MFEM v3.4 Release

MFEM v3.4 was released with many new features, including:

- Significantly improved non-conforming unstructured AMR scalability.
- *Integration with PUMI, the Parallel Unstructured Mesh Infrastructure from RPI.*
- Block nonlinear operators and variable order NURBS.
- Conduit Mesh Blueprint support
- General high-order-to-low-order refined field transfer.
- New specialized time integrators (symplectic, generalized-alpha).
- Twelve new examples and miniapps.

- And many more, see `http://mfem.org` for details.

MFEM is also now officially part of OpenHPC, a Linux Foundation project to provide software components required to deploy and manage HPC Linux clusters, including a variety of scientific libraries.

5.3 MAGMA v2.4.0 Release

We released MAGMA 2.4.0 on June 25, 2018. This release included performance improvements as well as some new dense and sparse numerical linear algebra routines and functionalities (see `http://icl.cs.utk.edu/magma/software/`). Most notably, as related to CEED, we developed and released performance improvements across many batch routines, including batched TRSM, batched LU, batched LU-nopiv, and batched Cholesky.

5.4 FEM-SEM Preconditioning

CEED researchers have performed extensive test and development of finite-element (FE) based preconditioners for Poisson problems based on the spectral element method (SEM). Because of their spectral equivalence, FE preconditioning of the SEM offers the prospect of nearly bounded iteration counts. While the method dates back to the origins of high-order elements [8], practical implementations have to date been lacking because the corresponding FEM problem is difficult to solve. New low-order discretization developments presented in [2], coupled with efficient and scalable algebraic multigrid implementations from Hypre, have led to an approach that, for particularly challenging geometries, can yield as much as a **40% reduction in Navier-Stokes solution times** over the hybrid-Schwarz multigrid solvers currently used in Nek5000.

5.5 Outreach

CEED researchers were involved in a number of outreach activities in Q3 of FY18,, including: the Intel co-design meeting in Santa Clara, Apr 10-12 (six CEED representatives attended), keynote at the 6th European Seminar on Computing and Nek5000's 6th User Meeting held Apr 17-18 at the University of Florida. We submitted 4 papers for publication, and CEED's Panayot Vassilevski was named a SIAM fellow for 2018. The team is also making final preparations for the upcoming CEED-organized minisymposium, "Efficient High-Order Finite Element Discretizations at Large Scale", at the International Conference in Spectral and High-Order Methods (ICOSAHOM18) and CEED's 2nd annual meeting to be held August 8-10, 2018 at the University of Colorado, Boulder.

6. CONCLUSION

In this milestone, we developed a high-order Field and Mesh Specification (FMS) interface that allows a wide variety of applications and visualization tools to represent unstructured high-order meshes with general high-order finite element fields defined on them.

The artifacts delivered include a simple API library and documentation for the new FMS interface, freely available in the CEED's FMS repository on GitHub. See the CEED website, `http://ceed.exascaleproject.org/fms` and the CEED GitHub organization, `http://github.com/ceed` for more details.

In this report, we also described additional CEED activities performed in Q3 of FY18, including: extensive benchmarking of CEED Bake-Off problems and kernels on BG/Q, GPU and AMD/EPYC platforms, improvements in libCEED, results from application engagements, progress on FEM-SEM preconditioning, three new software releases, and other outreach efforts.

ACKNOWLEDGMENTS

This research was supported by the Exascale Computing Project (ECP), Project Number: 17-SC-20-SC, a collaborative effort of two DOE organizations—the Office of Science and the National Nuclear Security Administration—responsible for the planning and preparation of a capable exascale ecosystem—including software, applications, hardware, advanced system engineering, and early testbed platforms—to support the nation's exascale computing imperative.

This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344, LLNL-TR-748602.

REFERENCES

[1] Mark W. Beall and Mark S. Shephard. A general topology-based mesh data structure. *International Journal for Numerical Methods in Engineering*, 40(9):1573–1596, 1997.

[2] P. Bello-Maldonado and P. Fischer. Scalable low-order finite element preconditioners for high-order spectral element Poisson solver. *submitted*, 2018.

[3] M.O. Deville, P.F. Fischer, and E.H. Mund. *High-order methods for incompressible fluid flow.* Cambridge University Press, Cambridge, 2002.

[4] W.S. Don and A. Solomonoff. Accuracy and speed in computing the Chebyshev collocation derivative. Technical Report NASA-CR-4411, 1991.

[5] Jing Gong, Stefano Markidis, Erwin Laure, Matthew Otten, Paul Fischer, and Misun Min. Nekbone performance on GPUs with OpenACC and CUDA fortran implementations. *Special issue on Sustainability on Ultrascale Computing Systems and Applications: Journal of Supercomputing*, 72:4160–4180, 2016.

[6] S. Markidis, J. Gong, M. Schliephake, E. Laure, A. Hart, D. Henty, K. Heisey, and P. Fischer. OpenACC acceleration of the Nek5000 spectral element code. *Int. J. of High Perf. Comp. Appl.*, 1094342015576846, 2015.

[7] David S. Medina, Amik St.-Cyr, and Timothy Warburton. OCCA: A unified approach to multi-threading languages. *CoRR*, abs/1403.0968, 2014.

[8] S.A. Orszag. Spectral methods for problems in complex geometry. *J. Comput. Phys.*, 37:70–92, 1980.

[9] Matthew Otten, Jing Gong, Azamat Mametjanov, Aaron Vose, John Levesque, Paul Fischer, and Misun Min. An MPI/OpenACC implementation of a high order electromagnetics solver with GPUDirect communication. *The International Journal of High Performance Computing Application*, 30(3):320–334, 2016.

[10] E.S. Seol and M.S. Shephard. Efficient distributed mesh data structure for parallel automated adaptive analysis. *Engineering with Computers*, 22(3-4):197–213, 2006.

[11] A. Tomboulidesa, S. M. Aithal, P. F. Fischer, E. Merzari, A. V. Obabkob, and D. R. Shaver. A novel numerical treatment of the near-wall regions in the k-ω class of RANS models. *International Journal of Heat and Fluid Flow*, 2018.

[12] Ping-Hsuan Tsai, Yu-Hsiang Lan, Misun Min, and Paul Fischer. Jacobi-free Newton Krylov method for Poisson-Nernst-Planck equations. *to be submitted*, 2018.

[13] Kevin Weiler. The radial edge structure: a topological representation for non-manifold geometric boundary modeling. *Geometric modeling for CAD applications*, pages 3–36, 1988.