ECP Milestone Report
Improved Support for Parallel Adaptive Simulation in CEED
WBS 2.2.6.06, Milestone CEED-MS29

Mark Shephard
Valeria Barra
Jed Brown
Jean-Sylvain Camier
Veselin Dobrev
Yohan Dudouit
Paul Fischer
Tzanio Kolev
David Medina
Misun Min
Cameron Smith
Morteza H. Siboni
Jeremy Thompson
Tim Warburton

July 12, 2019

ECP Milestone Report
Improved Support for Parallel Adaptive Simulation in CEED
WBS 2.2.6.06, Milestone CEED-MS29

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

July 12, 2019

ECP Milestone Report
Improved Support for Parallel Adaptive Simulation in CEED
WBS 2.2.6.06, Milestone CEED-MS29

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	July 12, 2019	Original	

EXECUTIVE SUMMARY

As part of its discretization mandate, CEED is developing adaptive algorithms for mesh refinement, coarsening and parallel rebalancing needed in general unstructured adaptive mesh refinement (AMR) of high-order hexahedral and/or tetrahedral meshes.

This milestone provides an update on our developments of adaptive mesh control methods for both conforming and non-conforming mesh adaptation procedures. We report on alternative load balancing methods and demonstrate the execution of a simulation workflow for the RF simulation of a tokamak fusion system including the geometrically complex antenna geometry.

In addition to details and results from these efforts, in this document we report on other project-wide activities performed in Q3 of FY19, including: the first MFEM release with GPU support, work with ECP applications, libCEED performance optimization, work on accelerator-oriented solvers, general interpolation, and other outreach efforts.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

1. INTRODUCTION

This milestone provides an update on our developments of adaptive mesh control methods for both conforming and non-conforming mesh adaptation procedures. As part of this discussion we also demonstrate the execution of a simulation workflow for the RF simulation of a tokamak fusion system including the geometrically complex antenna geometry. The milestone also reports on an investigation of alternative load balancing methods.

A number of other recent CEED advances are reported on including: the first MFEM release with GPU support, the latest libCEED performance results, the development of an improved point location algorithm for Nek5000, recent Nek5000 results generated for ECP applications, work on accelerator-oriented solvers, general interpolation, and other outreach efforts.

2. ADAPTIVE ALGORITHMS FOR HIGH-ORDER SIMULATIONS

Mesh adaptation methods are central to the effective control of the equation discretizations errors introduced by the finite dimensional trial and weighting spaces used in the finite element method. CEED technologies include both conforming and nonconforming mesh adaption methods. In this report we review several ongoing efforts in CEED mesh adaptation technologies targeting high-order meshes and discretizations, including:

- Advances to the conforming mesh adaptation procedures for higher order meshes including extensions to the mesh curving operators and the introduction of two mesh modification operators.
- A recovery-based error estimator for vector-basis Nedelec elements commonly used for $H(curl)$ operators.
- Application of conforming mesh adaptation demonstrating the ability to address complex geometric domains defined in terms of CAD solid models.
- Advances to the MFEM nonconforming mesh adaptive refinement procedures.
- Parallel load balancing performance on GPU architectures.

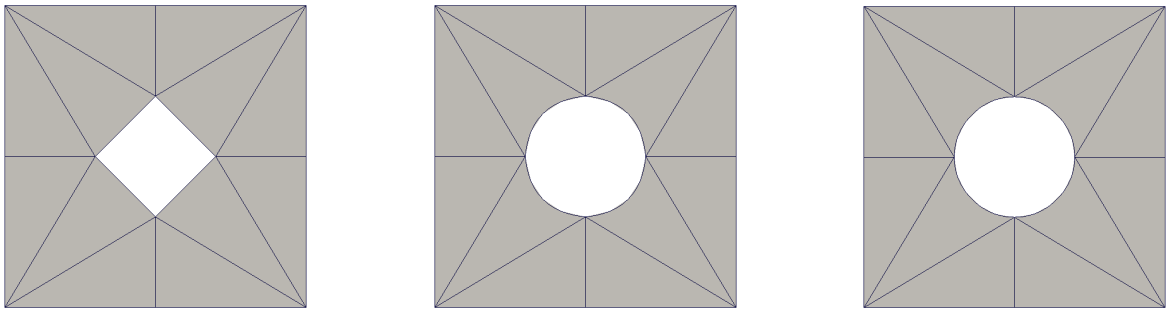
2.1 High-Order Mesh Curving in Conforming Mesh Adaptation

The application of conforming mesh adaptation in combination with high-order methods must account for the fact that individual mesh elements can be curved. In the case of curved domains the mesh entities used to represent the curved boundaries must be curved to the appropriate level of geometric approximation as dictated by the order of the finite element basis functions used. In addition to mesh entities that represent the domain boundaries being curved to those boundaries, interior mesh entities are also often curved, either due to the use of the Lagrangian reference frames in large motion problems, or because the use of large elements to represent curved boundaries requires the curving of nearby interior mesh entities to maintain acceptable element shapes. Thus the conforming mesh adaptation procedures, originally based on straight sided elements [20], have to be extended to deal with curved elements. Some of the recent efforts in the PUMI-based [18, 28] curved mesh adaptation procedures have been focused on extending the procedures that could do quadratic element geometry [22, 23] to support high-order geometry. The specific developments carried out include:

- Curving mesh entities on the boundary up to sixth order to improve geometric approximation.
- Introduction of a higher order mesh entity curving operator.
- Introduction of a cavity re-triangulation operator.

To maintain the maximum possible rate of convergence in higher-order finite elements, the meshes must satisfy specific requirements in terms of geometric approximations of the domain. Specifically, the level of the geometric approximation of mesh entities classified on curved domain boundaries must increase as the order of the finite element basis functions increase. Otherwise, the geometric approximation errors yield any further increase in finite element order meaningless [24]. The process of increasing the order of geometric approximation of a mesh is referred to here as *geometry inflation*. The importance of "geometry inflation"

when using higher-order finite element solvers such as MFEM is best demonstrated by means of an example. More specifically, we consider a simple Maxwell eigenproblem over a domain shown in Figure 1. In particular we solve this problem using 3rd order finite elements over three different meshes: (a) a linear geometry mesh, (b) a quadratic geometry mesh, and (c) a cubic geometry mesh, as shown in Figure 1. As can be seen in this figure all meshes used for this example have the same number of elements. Table 1 shows the first 4 eigenvalues for this eigenproblem as obtained by the finite element analysis over the three meshes shown in the figure. For comparison, we also list the corresponding eigenvalues for a reference (much finer) mesh.



(a) **(b)** **(c)**

Figure 1: Proper "geometry inflation" is important when using higher-order finite element basis functions. Cubic finite elements were used to solve a Maxwell eigenproblem on mesh linear (a), quadratic (b), and cubic (c) geometry.

Mesh Geometry	λ_1	λ_2	λ_3	λ_4
Linear	7.18	7.18	18.4	37.7
Quadratic	6.65	6.65	18.0	34.3
Cubic	6.68	6.68	18.1	34.3
Reference	6.70	6.70	18.2	34.8

Table 1: The first 4 eigenvalues for the Maxwell operator solved using the meshes shown in Figure 1 and a refined reference mesh.

It should be noted that the process of entity order inflation can lead to the creation of invalid elements. Procedures to address the elimination of the invalid elements has been addressed in the case of going from linear elements to quadratic elements which is where the creation of invalid elements frequently occurs. Although the creation of invalid element is less frequent in inflating from quadratic to higher order, it can, and does occur. Initial efforts toward addressing this issue are discussed below.

For the purposes of "geometry inflation", we use Bèzier mesh entity geometry because of their useful properties, such as the convex hull property [14]. In order to inflate a mesh entity to the desired order, we have to identify the locations of internal control points for the given Bèzier representation (for example for an order p edge we need to determine $p-1$ internal points). To obtain the control points for a given mesh entity we make sure that the geometric representation of the mesh entity coincides with the model boundary (or field lines in case of the field following meshes) at a specific set of points. The choice of these points, must be such that the desired geometric approximation is achieved. For our geometric inflation methods, we make use of the Babuška-Chen points which are known to be optimal for polynomial interpolation of real functions in an interval or a triangle [11]. We note here that other choices for the placement of these points include the Gauss-Lobatto and Gauss-Legendre points. As can be seen in Table 2 the difference between these different point placement schemes is not significant especially for the case of the Babuška-Chen and Gauss-Lobatto points.

In addition to needing curved mesh entities for the mesh edges and faces classified on curved model boundaries, there are times when it is desirable, or necessary, to have and control curved mesh entities interior

Point Placement Scheme	1st point	2nd point
Babuška-Chen	0.274804	0.725196
Gauss-Lobatto	0.276393	0.723607
Gauss-Legendre	0.211325	0.788675

Table 2: Different point placement options in the interval $[0, 1]$ for 3rd order polynomial interpolation of real functions.

to the domain. Although the current mesh generation and mesh adaptation procedures can effectively support the needs for quadratically curved mesh entities [22, 23, 25], there are times when higher that quadratic interior mesh entities are needed. One such case is shown in Figure 20 where a coarse tetrahedral mesh is used to represent a highly curved geometry. Another case is when applying conforming mesh adaptation in conjunction with high-order Lagrangian finite elements codes, such as MARBL, where elements can become highly curved in the process of following material during the simulation.

To support the ability to curve interior mesh entities to higher than quadratic order a high-order entity curving operator has been added to a test mesh adaptation procedure. Our conforming mesh adaptation procedure operates by the application of a set of local mesh modification operations that coarsen, refine or improve the element quality as dictated by a defined mesh size field [20]. As with the other mesh modification operators, the entity curing operator first identifies the local set of mesh entities that will be affected by the operation. This set of mesh entities is called a mesh cavity. In the case of the entity curving operator the cavity is the closure of all the mesh entities the entity to be curved bounds. The entity curving operator executes a local optimization procedure using a quasi-Newton iteration in which the degrees of freedom are the positions of the control points interior to the mesh entity being curved. The steps of the above optimization-based algorithm for determining the location of internal nodes in the cavity is shown in Algorithm 1.

Algorithm 1 Optimization based node re-positioning

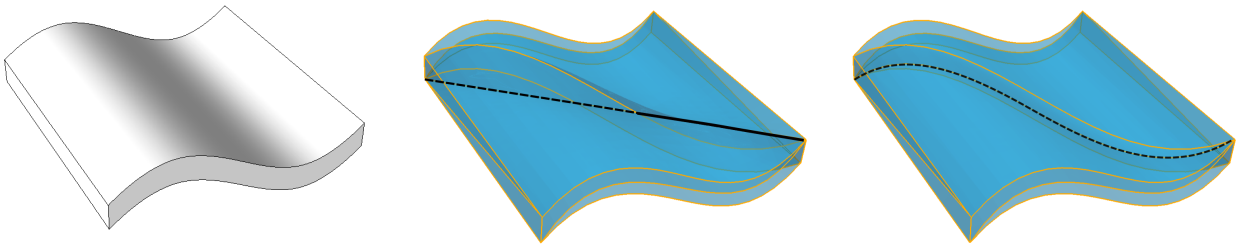
1: **input(s):** *cavity*, x (degrees of freedom), f (merit function)
2: $x_0 \leftarrow$ initial coordinates of the nodes to be re-positioned
3: *iter* $\leftarrow$ number of iterations for the optimization algorithm
4: *tol* $\leftarrow$ convergence criteria for the optimization algorithm
5: $x \leftarrow \text{LBFGS}(f, iter, tol, x_0)$ ▷ limited-memory BFGS optimization algorithm
6: *cavity* $\leftarrow$ UpdateCavity(x) ▷ update the cavity using new node coordinates
7: **if** *cavity* is valid **then**
8: **return** Success
9: **else**
10: *cavity* $\leftarrow$ UpdateCavity(x_0) ▷ revert the cavity back to its original shape
11: **return** Failure
12: **end if**

In practice, the choice of the optimization merit function depends on the desired properties. For example, one might prefer curved meshes in which the variation of the Jacobian determinant is minimal throughout each entity in the mesh, or one might prefer the Jacobian determinant to be larger than a threshold across the mesh. We are currently using the merit function defined in reference [15] as given by

$$f(\mathbf{x}) = \sum_{e \in \text{cavity}} \sum_{i+j+k+l=3(P-1)} \omega_{ijkl} \left(\frac{N_{ijkl}}{6|K_s|} - 1 \right)^2 . \tag{1}$$

Here $\mathbf{x}$ is a vector containing the locations of all the internal control points in the cavity, e is an entity in the cavity, ω_{ijkl} are weights associated with the control points of e, $|K_s|$ is the volume of the straight sided e and N_{ijkl} are the control coefficients of the Jacobian determinant for the entity e (see for example, [6] for the definition of these control coefficients in terms of the control points of the underlying Bèzier shapes). The specific merit function in equation (1) tends to minimize the variation of the Jacobian over the elements in the cavity. Other merit functions can be found in reference [31].

Figure 2 shows the result of applying the entity curving operator to a simple example. Specifically, we consider a thin slab in which the cross section follows a cubic profile. As can be seen in this figure after inflating the bounding mesh geometry to cubic Bezier in order to approximate the model boundaries an internal edge becomes invalid (see Figure 2b). One can then use the above operator to obtain the location of internal nodes for the edge in question. This results in a valid mesh shown in Figure 2c. Note that for this specific example a cubic representation for the mesh geometry is sufficient to achieve a reasonable geometric approximation without introducing any mesh refinement. It is worthwhile to mention that achieving the result in Figure 2c would not have been possible using only quadratic meshes and without introducing mesh refinement. This shows the importance of using higher order geometric approximation to represent complex model boundaries with as few finite elements as possible. Additional efforts are required to determine the most effective use of the entity curving operator during mesh adaptation. Such efforts include:



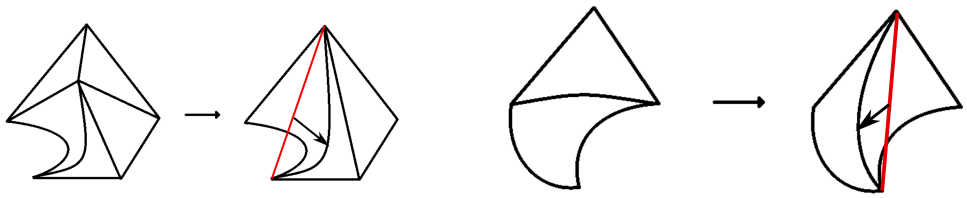
(a) **(b)** **(c)**

Figure 2: A simple example where interior mesh entities must be curved to higher than quadratic order (c) due to a coarse mesh (b) on a highly curved geometry model (a).

1. Combining the above reshaping operator with more basic operators such as edge collapse and edge/face swap operations in case of curved meshes (see Fig 3 for an illustration).
2. Reshaping mesh entities (e.g., mesh edges) that are classified on curved model faces. Note that, in general this requires a constrained optimization procedure to ensure that the reshaped entities conform to the model face. However, by rewriting the optimization problem in terms of the parametrization of the underlying model surface we are able to avoid the constrained optimization.
3. Improving the mesh curving and mesh adapt logic (such as the order in which different operations are performed) to achieve a valid curved mesh in minimal time.

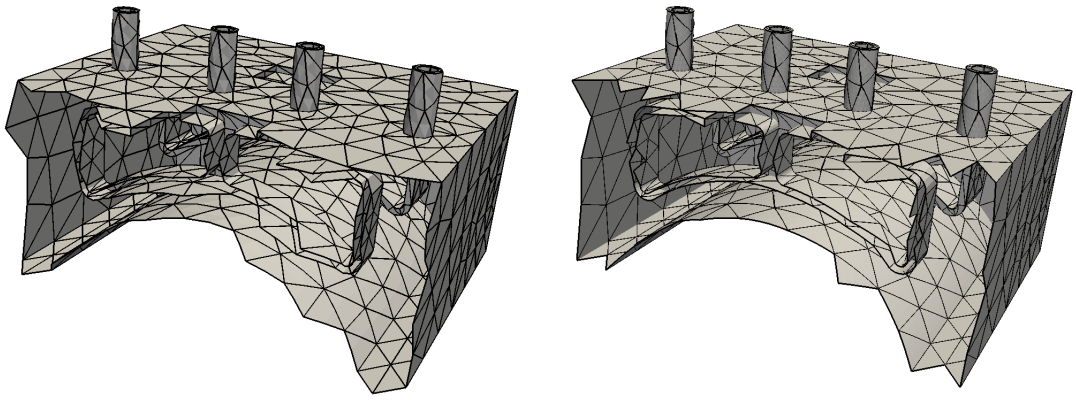
As mentioned previously, the boundary mesh edge and mesh face order inflation process can create invalid elements (elements with a negative Jacobian somewhere). Fig 4 shows the mesh entities on the surface that were inflated from quadratic to cubic. Even with those small changes in shape a small number of invalid elements resulted when entity inflation only was applied. A procedure that employs the mesh entity curving operation, including further curving of mesh entities classified on the boundary, is under development and is in initial testing. Its application did yield a valid mesh of cubic elements (Fig 4(b).

An evaluation of the existing mesh adaptation procedure determined that the application of the primary mesh coarsening operator, edge collapse, would often fail due to local topological or geometric restrictions. The result would be an adapted mesh that had more elements than necessary. To increase the ability to coarsen the mesh a new operator was implemented that retriangulates cavities by the application of element removal operations. The current implementation is limited to cavities with straight mesh edges. It can be extended to curved mesh entities. It is worthwhile to mention that although the cavity retriangulation by element removal is a more expensive operator than the edge collapse operator, the low success rate of the edge collapse operator introduced additional computational cost, as seen further below.



(a) **(b)**

Figure 3: Schematics for basic operators that result in invalid elements due to curved cavity boundaries which can be fixed by using the entity reshaping operator described in this section: collapse (a) and swap (b) operations.



(a) **(b)**

Figure 4: The coarse cubic curved mesh for an RF antenna: initial quadratic mesh generated in Simmetrix (a), and curved cubic mesh (b).

The element removal operator operates on a given mesh cavity by first emptying it and then adding one element at time by attaching it to faces of the cavity boundary thus removing them from the cavity boundary. The process examines edges of the current cavity and create an element using a selected edge and the two adjacent faces. The element to remove first is based on geometric properties of the cavity. Starting at one edge in the cavity the desired geometric properties are checked. If the element that would be created satisfies the desired geometric properties, the element is created and the edge and its adjacent faces are eliminated from the cavity and replaced my newly created ones interior to the original cavity. If the current edge being examined does not meet the desired geometric properties, the process considers the next edge in the queue. The process continues until the cavity is filled, or it cannot meet the required conditions. This basic procedure is successful in most cases. When it does fail the cavity is returned to its original form. Since it is currently being used to coarsen meshes during adaptation, this is acceptable. Two items to note: (1) its failure rate is much less than edge collapse; (2) there are additional algorithmic refinements possible that can ensure successful termination of the process so long as point insertion is allowed.

In the current implementation, the edges to be removed are considered in a priority order based on the lowest metric dihedral angle for the edge: i.e. the angle between the adjacent boundary faces after they have been transformed according to the given metric [4]. The rationale behind this approach is that the removal of an element also reduces the dihedral angle of other boundary edges around the removed element typically providing the opportunity for the definition of well shaped elements in subsequent steps. Since there are times when this selection will create an element that intersects the cavity, leaves a situation where a future element must be poorly shaped, or creates a topological situation currently not handled, the procedure includes the required geometric checks and rejects a removal if any of them are detected. The overall algorithm is described in Algorithm 2.

Algorithm 2 Element Removal Operation

procedure MAKENEWELEMENTS($cavity, qualToBeat$)
  $bEdges \leftarrow$ a queue of all boundary edges based on priority
  while $bEdges$ is not empty **do**
    $e \leftarrow pop(bEdges)$ ▷ pick out the next edge in queue
    $\{v_0, v_1\} \leftarrow$ vertices of $edge$
    $\{f_0, f_1\} \leftarrow$ boundary faces adjacent to $edge$
    $\{v_2, v_3\} \leftarrow$ vertices of $\{f_0, f_1\}$ resp. opposite $edge$
    $elem \leftarrow$ element made out of $\{v_0, v_1, v_2, v_3\}$
    if QUAL($elem$) ¡ $qualToBeat$ **then** ▷ All elements must be better than $qualityToBeat$, which could be 0 (i.e. just valid).
      Reject this element and move to next edge in $bEdges$
    end if
    if $elem$ intersects any other edge of the cavity **then** ▷ Additional "closeness" checks can also be performed here
      Reject this element and move to next edge in $bEdges$
    end if ▷ The new cavity is valid and not too thin
    Update boundary and reconstruct $bEdges$ (reinserting all popped-out edges)
  end while
  if Cavity is non-empty at this point **then** ▷ algorithm failed
    Attempt point insertion or any other element removal procedure
  end if
end procedure

Tests were performed on a model of a dam-break with obstacle, which has experimental data (commonly known as the MARIN case). In this simulation, run in the Proteus CFD code, adaptations were performed at regular intervals during the simulation in order to obtain isotropic meshes for subsequent steps. Here we show the results at a late stage of the simulation, after water has already hit the obstacle and far wall (see Figure 5 for the adapted mesh). In the adapted mesh, we intend for the elements to be 8 times smaller in the neighborhood of the air-water interface as compared to those further away. A comparison of the mesh adaptation procedure with or without the element removal operator included indicates (see the histograms in

Figure 6 for more details):

- The number of undesired short edges decreased from 36K to 2.7K.
- The number of elements in the mesh decreased from 7.1M to 6.9M.
- The number of low quality elements decreased from 98 to 30.
- The minimum edge length (in metric space) increased from 0.13 to 0.17.
- The time required to execute the mesh adaptation step decreased by 13%.

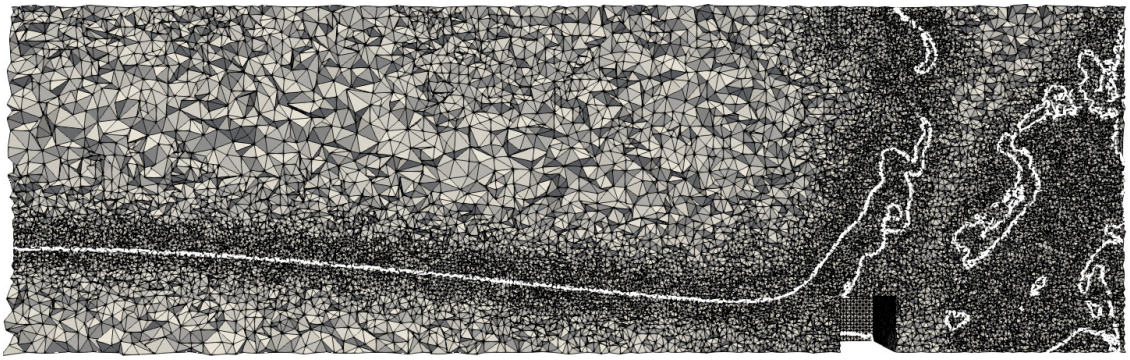


Figure 5: Cross section of mesh for a later step in the MARIN (dam-break with obstacle) case.

2.2 Error Indicator for Electromagnetics

A key step in the application of adaptive mesh control methods is the determination of the portions of the domain where the mesh must be enriched to attain the level of solution accuracy desired and well as determination of areas where the mesh can be coarsened while still providing the desired levels of accuracy. Error estimation/indication procedures are used perform a posteriori evaluation of the local error contributions and to indicate where the mesh needs enrichment and where it can be coarsened.

A super convergent patch recovery based error indicator has been implemented for use in the adaptive FR application example discussed below. Specific care was taken in the implementation of the procedure to properly account for the vector basis functions used by the $H(curl)$ edge based Nedelec elements. This class of error indicator has been found satisfactory for driving a mesh adaptation procedure.

2.3 RF Application Example

Increasingly the scientific questions being investigation by DOE researchers require the execution of simulations that not only account for complex physics, they must also account for geometrically complex domains that must be properly represented by the meshes used in the simulation. Without the application of fully automatic mesh generation procedures that can operate on high level geometric domain definitions, the initial mesh generation process can dominate the time required to execute the desired simulations. By combining the CEED finite element analysis and adaptive meshing procedures with advanced geometric modeling and automatic mesh generation techniques, it becomes possible to execute reliable simulations of complex physics problems over general domains of interest. As a demonstration of such a capability a MFEM based adaptive RF simulation of a portion of a tokamak reactor with full RF antenna geometry is presented. Simulations of these types (current under development for plasma and RF simulations) are of great importance to ITER design and operation.

The combination of complex physics and complex geometry associated with these simulations requires the execution of an overall work flow that includes the following steps:

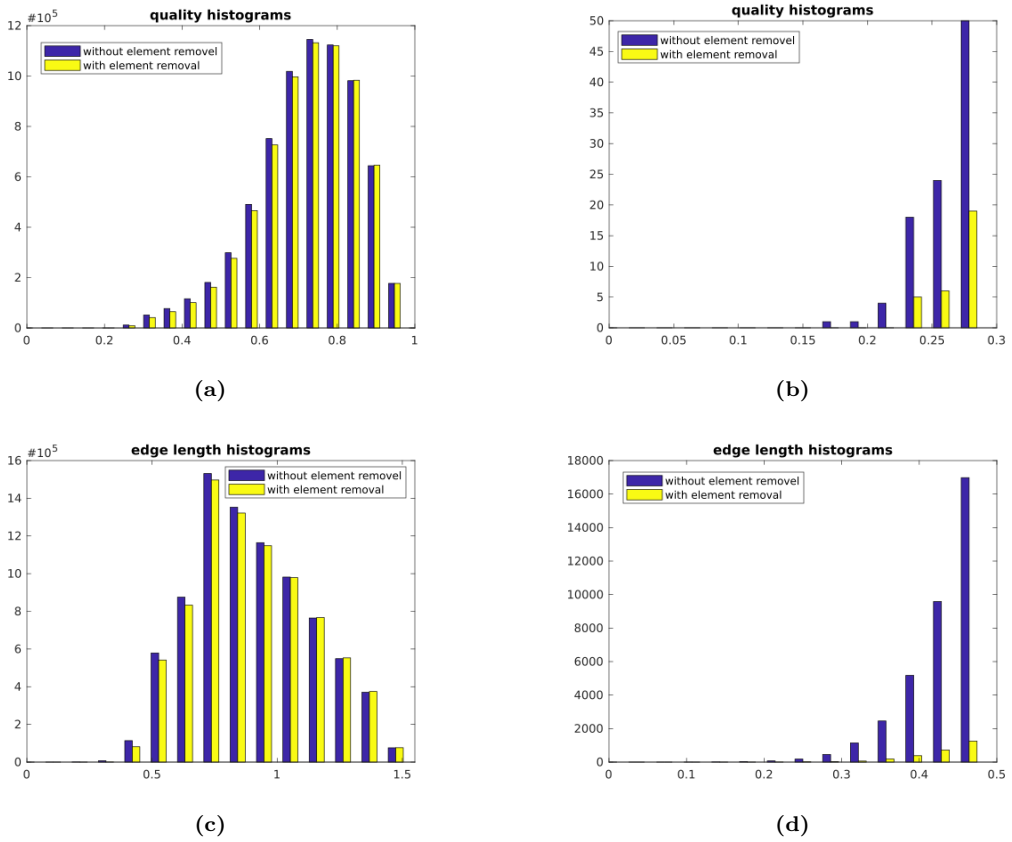
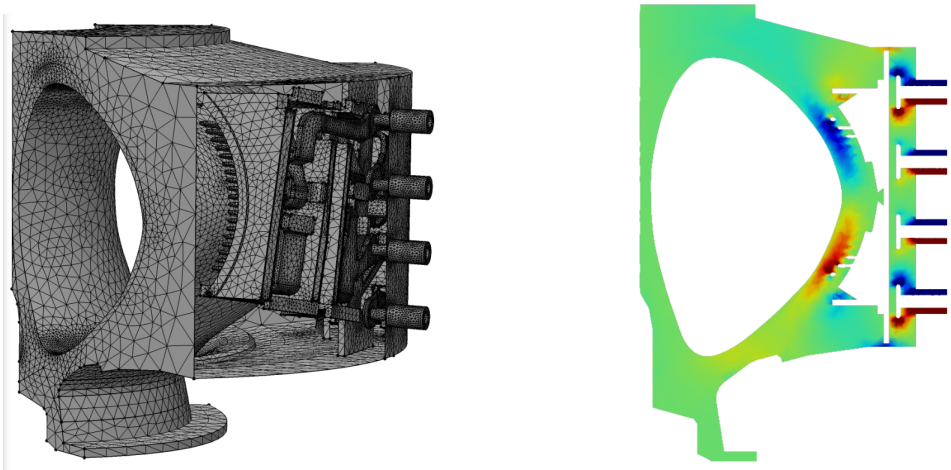


Figure 6: Mesh statistics for the adapted MARIN case with and without the element removal procedure: quality histogram (a) with close-up (b), and edge length histogram (c) with close-up (d).

1. Obtaining detailed CAD models of the antenna assemblies and performing operations on those models to represent the level of geometric detail needed for the simulation, while eliminating unneeded details, and ensuring the geometry can be integrated with the other geometric components required to define the final analysis geometric model to be meshed.
2. Combining the cleaned-up CAD antenna models with the reactor geometry and any desired physics geometry (e.g., flux surfaces) to create the needed analysis geometric domain.
3. Associating any desired meshing attributes to the analysis model.
4. Applying the required physical attributes needed and defining the analysis case to be executed.
5. Automatically generating the required mesh.
6. Executing the finite element analysis.
7. Estimating the mesh discretization errors. If the error is within the given tolerance level, terminate the simulation and mark it as complete.
8. Adapting the mesh and returning to step 6.

Steps 1-5 are executed using tools from Simmetrix [2] and MIT [1]. Steps 6-8 are executed by CEED developed technologies.

An RF simulation consists of specifying the needed analysis attributes in Petra-M [1] and invoking the appropriate finite element analysis. The LLNL (MFEM developers) and MIT teams involved with the RF fusion SciDAC have developed RF simulation capability in MFEM and a tool to prescribe the needed RF analysis attributes through Petra-M respectively. The Simmetrix mesh, as well as its classification against the geometric model, has been integrated into Petra-M for the association of the needed analysis attributes. The mesh and attribute information has been integrated into MFEM to perform the RF simulations. Figure 7 shows the 1 million element mesh used to run an MFEM simulation and the resulting electric field.



(a) **(b)**

Figure 7: MFEM simulation using combined model. Shown are the Simmetrix mesh (a) and the z-component of the electric field (b).

The most recent developments have been focused on addressing Steps 7 and 8 of the workflow indicated above and executing these steps in an automated loop using in-memory integration methods. In memory integration methods [29] eliminate the need to employ costly file I/O in the process of communicating information between components, in this case between Petra-M, MFEM, the error estimation procedure and the mesh adaptation procedure.

At this point a basic adaptive loop is in place that is using a combination of in-memory and file based information transfer (full in-memory transfer will be completed shortly). A super convergent patch recovery error estimator is used to generate element level error contributions. The element level errors are converted into

a mesh size field which represents the input to the mesh adaptation procedures. An initial test demonstration of an adaptive simulation applied to a two antenna problem is shown in Figure 8. The adapted mesh for this example is shown in Figure 9.

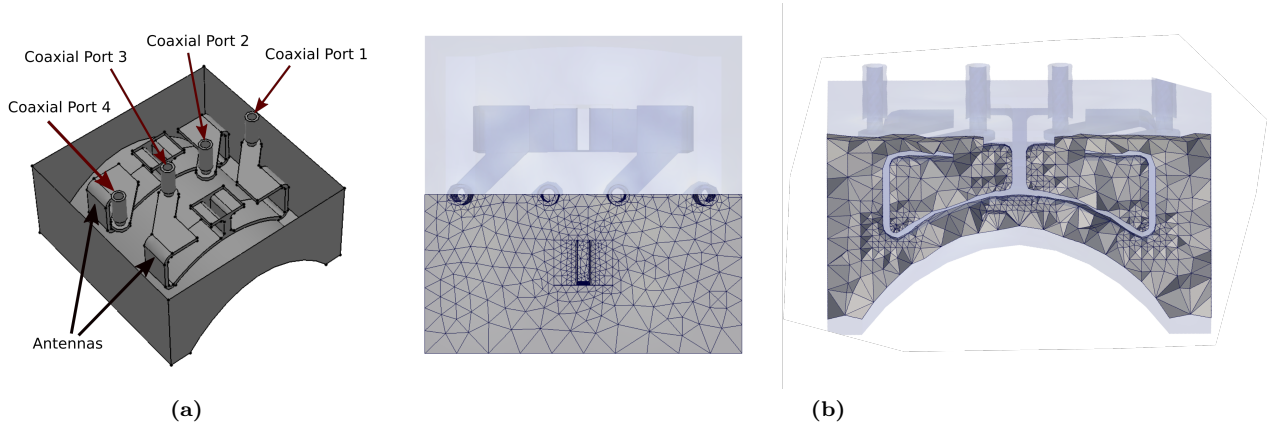


Figure 8: Demonstration example for a two antenna RF case. Shown are the analysis geometry with physics parts (a) and the automatically generated initial mesh (b).

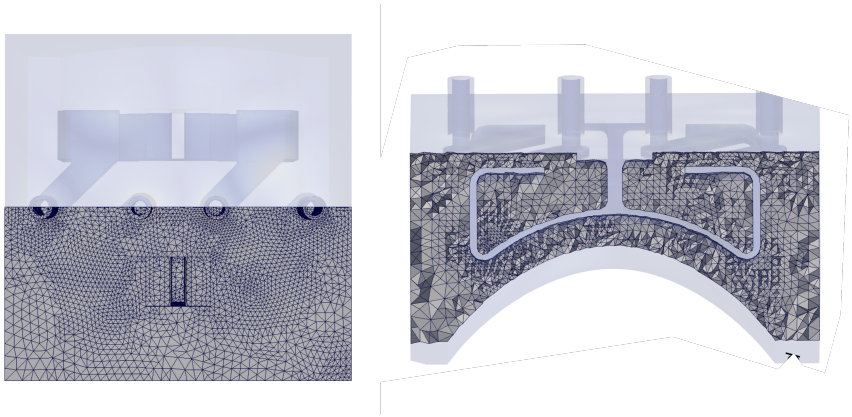


Figure 9: The adapted mesh for the above two antenna example.

2.4 Nonconforming Mesh Adaptation

Tensor-product mesh elements (quadrilateral in 2D and hexahedral in 3D) are attractive in many high-order applications, because the tensor product structure enables efficient operator evaluation, as well as refinement flexibility (e.g. *anisotropic* refinement). Unlike the conforming case however, *hanging* nodes that occur after local refinement of quadrilaterals and hexahedra are not easily avoided by further refinement. Therefore, CEED researchers are also interested in *non-conforming* (irregular) meshes, in which adjacent elements need not share a complete face or edge and where some finite element degrees of freedom (DOFs) need to be constrained to obtain a conforming solution.

In this section we review the algorithms for handling parallel non-conforming meshes in the MFEM library. This review is a just a summary of [10], see the paper for all details.

In order to support the entire de Rham sequence of finite element spaces, at arbitrarily high-order, we use a variational restriction approach to AMR, which is described below. This approach naturally supports high-order curved meshes, as well as finite element techniques such as hybridization and static condensation.

It is also highly scalable, easy to incorporate into existing codes, and can be applied to complex (anisotropic, n-irregular) 3D meshes.

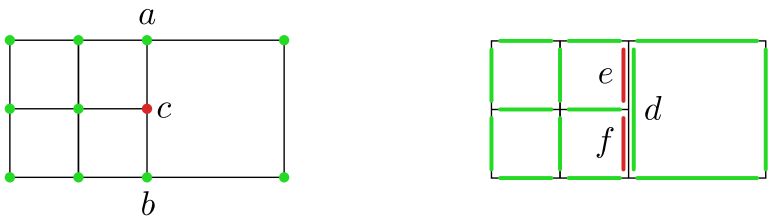


Figure 10: Illustration of conformity constraints for lowest order nodal elements in 2D. Left: Nodal elements (subspace of H^1), constraint $c = (a + b)/2$. Right: Nedelec elements (subspace of $H(curl)$, constraints $e = f = d/2$. In all cases, fine degrees of freedom on a coarse-fine interface are linearly interpolated from the values at coarse degrees of freedom on that interface.

To construct a standard finite dimensional FEM approximation space $V_h \subset V$ on a given non-conforming mesh, we must ensure that the necessary conformity requirements are met between the slave and master faces and edges so that we get V_h that is a (proper) subspace of V. For example, if V is the Sobolev space H^1, the solution values in V_h must be kept continuous across the non-conforming interfaces. In contrast, if V is an $H(curl)$ space, the tangential component of the finite element vector fields in V_h needs to be continuous across element faces. More generally, the conformity requirement can be expressed by requiring that values of V_h functions on the slave faces (edges) are interpolated from the finite element function values on their master faces (edges). Finite element degrees of freedom on the slave faces (and edges) are thus effectively constrained and can be expressed as linear combinations of the remaining degrees of freedom. The simplest constraints for finite element subspaces of H^1 and $H(curl)$ in 2D are illustrated in Figure 10.

The degrees of freedom can be split into two groups: *unconstrained (or true)* degrees of freedom and *constrained (or slave)* degrees of freedom. If z is a vector of all slave DOFs, then z can be expressed as $z = Wx$, where x is a vector of all true DOFs and W is a global interpolation matrix, handling indirect constraints and arbitrary differences in refinement levels of adjacent elements. Introducing the *conforming prolongation matrix*

$$P = \begin{pmatrix} I \\ W \end{pmatrix},$$

we observe that the coupled AMR linear system can be written as

$$P^T A P x_c = P^T b, \tag{2}$$

where A and b are the finite element stiffness matrix and load vector corresponding to a finite element discretization on the broken space $\widehat{V}_h = \cup_i (V_h|_{K_i})$. After solving for the true degrees of freedom x_c we recover the complete set of degrees of freedom, including slaves, by calculating $x = Px_c$. An illustration of this process is provided in Figure 11.

MFEM's non-conforming AMR algorithms have been heavily optimized for both weak and strong parallel scalability as illustrated in Figure 12, where we report results from a 3D Poisson problem on the unit cube with exact solution having two shock-like features. We initialize the mesh with 32^3 hexahedra and repeat the following steps, measuring their wall-clock times (averaged over all MPI ranks): 1) Construct the finite element space for the current mesh (create the P matrix); 2) Assemble locally the stiffness matrix A and right hand side b; 3) Form the products $P^T AP$, $P^T b$; 4) Eliminate Dirichlet boundary conditions from the parallel system; 5) Project the exact solution u to u_h by nodal interpolation; 6) Integrate the exact error $e_i = ||u_h - u||_{E,K_i}$ on each element; 7) Refine elements with $e_j > 0.9 \max\{e_i\}$; 8) Load balance so each process has the same number of elements (± 1). Out of the approximately 100 iterations of the AMR loop, we select 8 iterations that have approximately 0.5, 1, 2, 4, 8, 16, 32 and 64 million elements in the mesh at the beginning of the iteration and plot their times as if they were 8 independent problems. We run from 64 to 393,216 (384K) cores on LLNL's Vulcan BG/Q machine. The solid lines in Figure 12 show strong scaling, i.e. we follow the same AMR iteration and its total time as the number of cores doubles. The dashed lines

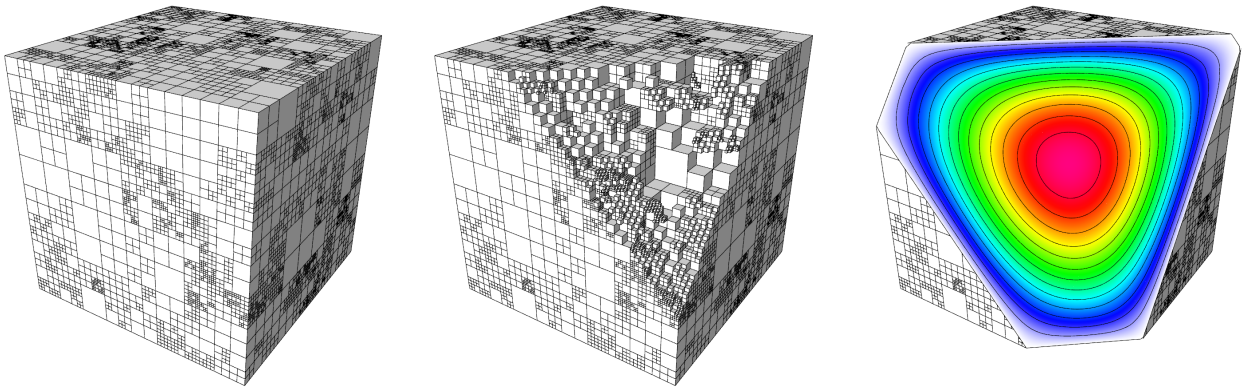


Figure 11: Illustration of the variational restriction approach to forming the global AMR problem. Randomly refined non-conforming mesh (left and center) where we assemble the matrix A and vector b independently on each element. The interpolated solution $x = Px_c$ (right) of the system (2) is globally conforming (continuous for an H^1 problem).

skip to a double-sized problem when doubling the number of cores showing weak scaling, and should ideally be horizontal.

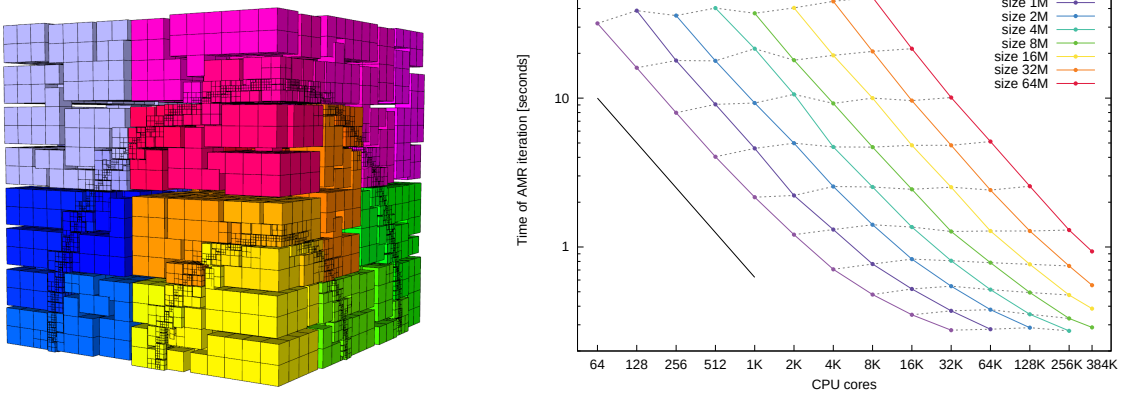


Figure 12: Left: One octant of the parallel test mesh partitioned by the Hilbert curve (2048 domains shown). Right: Overall parallel weak and strong scaling for selected iterations of the AMR loop. Note that these results test only the AMR infrastructure, no physics computations are being timed.

2.5 Load Balancing

Partitioning and load balancing on accelerated systems must account for the disparity between the network and accelerator memory bandwidth. On the Summit system at Oak Ridge National Laboratory this ratio is 205:1 if peak bandwidth from the accelerator memory is attained on all six accelerators (6 * 855 GBs = 5130 GBs/node) and peak bandwidth is reached during inter-node communications (25GBs) (5130 GBs / 25 GBs = 205) [12]. On Sierra at Lawrence Livermore National Laboratory (LLNL) with four accelerators per node the ratio is 130:1. Figure 13 depicts the logical processing and network components of the Summit and Sierra IBM AC922 nodes. In contrast, on an IBM BGQ this ratio was 1.4 (27 GBs/node and 20GBs inter-node

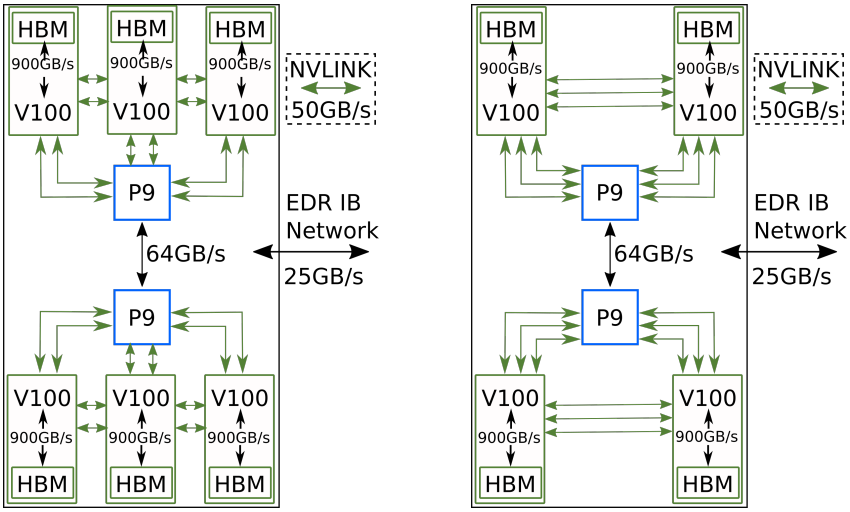


Figure 13: Logical diagram of the IBM AC922 Summit (left) and Sierra (right) nodes.

bandwidth). On a Skylake cluster like Stampede2 at TACC, the ratio is approximately 17 (194 GBs for two socket Skylake [17] and 12 GBs Omnipath). Within a Summit node the V100 memory to NVLink 2.0 channel bandwidth ratio is 8.6. An additional NVLink 2.0 channel is used between V100s on a Sierra node which decreases its intra-node ratio to 5.7.

The low performance of the network relative to the accelerators suggests that partitions that induce a low communication cost should perform well assuming they achieve a sufficient work load balance. Towards this the EnGPar load balancing library has been integrated with MFEM to provide graph and hyper-graph based multi-level partitioning via ParMETIS and native diffusive partition improvement procedures [13]. Integration entailed modifying the MFEM CMake build system and calling the requisite MFEM mesh topology query APIs to pass information into the EnGPar hyper-graph construction routines.

The EnGPar integration is used to compare the performance of GPU accelerated assembly and solution of the Laplace equation $-\Delta u = 1$ on a nonconforming mesh partitioned with a space-filling curve and a multi-level graph method. The Gecko library [21] quickly computes the space-filling curve ordering then computes a prefix sum to provide a near perfect element imbalance. ParMETIS [19], via EnGPar, computes a partition of the mesh dual-graph (mesh elements define graph vertices and mesh faces define graph edges) with a multi-level procedure. ParMETIS partitions satisfy the user requested imbalance target for mesh elements while minimizing the number of mesh faces duplicated across process boundaries.

The distribution of work to processes is measured by the vertex, edge, face and volume mesh entity imbalances (properly weighted) as each of these levels of mesh entity will have degrees of freedom associated with them when using high-order elements. These degrees of freedom are the principle source of computational load. The imbalance of a given mesh entity order is defined as the maximum number of entities across all processes divided by the average number of entities (total entities divided by number of processes).

The communications induced by a partition is measured with the count of mesh entities that are duplicated on multiple processes. For example, two triangles sharing a common edge but assigned to different processes will each have duplicate copies of the common edge and the vertices that bind it. In this case the shared entity count would be one edge and two vertices.

Table 3 lists the average time of three runs for each partition of a 2.5 million hex element mesh with linear, quadratic, and cubic finite elements on the LLNL Lassen system (a 20 PetaFLOP version of Sierra). The time for a given run is recorded as the maximum time spent by any process. One MPI process is associated with each GPU. Table 4 lists the vertex and element imbalances for each partition. ParMETIS was run with a target element imbalance of 3%, which it achieves, and maintains a vertex imbalance of less than 8%. Gecko produces a near perfect element imbalance and a vertex imbalance of up to 5%. Timings at all GPU counts slightly favor the ParMETIS partition over the Gecko partition. The largest difference is observed on 16 GPUs for quadratic shape functions where MFEM is 5.57% faster with ParMETIS than Gecko. Figure 14

order	1			2			3		
GPUs	sfc (s)	graph (s)	graph/ sfc (%)	sfc (s)	graph (s)	graph/ sfc (%)	sfc (s)	graph (s)	graph/ sfc (%)
4	6.71	7.03	4.48	144	148	2.70	489	490	0.11
8	3.49	3.59	2.80	75.8	78.5	3.41	262	266	1.39
12	2.49	2.51	1.05	52.6	54.5	3.44	173	177	2.23
16	1.90	1.95	2.67	39.2	41.5	5.57	133	138	3.73
20	1.57	1.60	1.89	32.5	33.2	1.94	111	112	0.82

Table 3: The time spent in GPU accelerated MFEM assembly and equation solution with Gecko and ParMETIS partition, respectively labeled as 'sfc (s)' and 'graph (s)'. The 'sfc/graph %' column lists the percent decrease in time spent in MFEM on the ParMETIS partition versus the Gecko partition.

GPUs	Gecko Vtx. Imb.	Gecko Elm. Imb.	ParMETIS Vtx. Imb.	ParMETIS Elm. Imb.
4	1.02	1.00	1.01	1.00
8	1.03	1.00	1.03	1.00
12	1.04	1.00	1.03	1.00
16	1.04	1.00	1.06	1.03
20	1.05	1.00	1.07	1.03

Table 4: Mesh entity imbalance. Lower is better.

depicts the performance improvement in MFEM with ParMETIS partitions.

Figure 15 depicts the ratio of shared entities in Gecko-to-ParMETIS partitions. The Gecko shared entity count is between 30% and 60% higher at all GPUs counts. Figure 16 shows the higher surface area, which results in the higher shared entity count, of a Gecko partition relative to a ParMETIS partition. The reduction in work and induced communications of the ParMETIS partitions results in MFEM performing slightly better than it does on Gecko partitions.

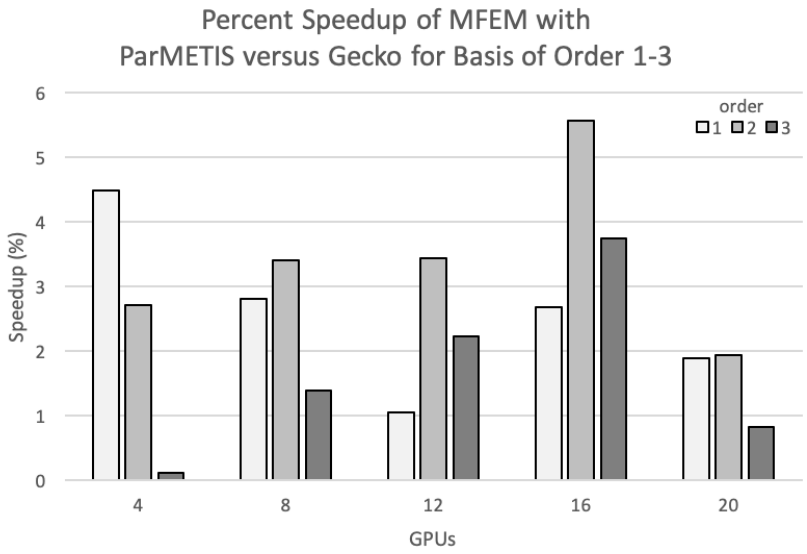


Figure 14: Percent speedup of MFEM assembly and solution procedures with a ParMETIS partition versus a Gecko partition.

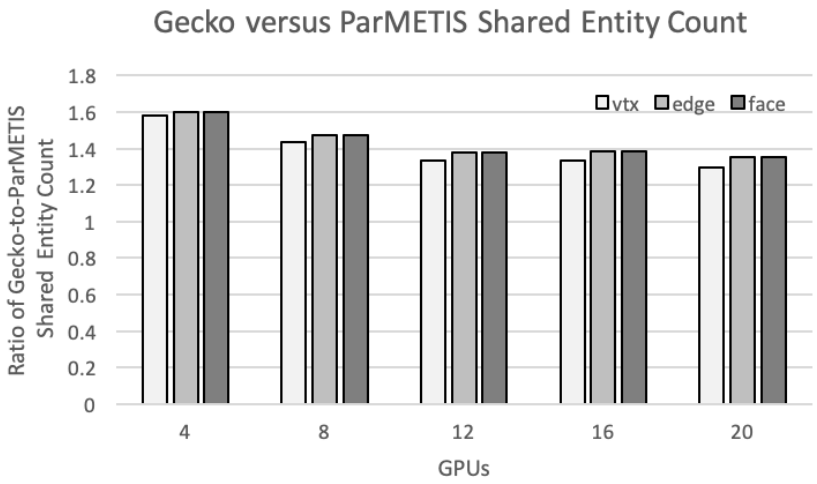


Figure 15: Ratio of Gecko-to-ParMETIS shared entity counts. A value greater than one indicates that Gecko has more shared entities than ParMETIS.

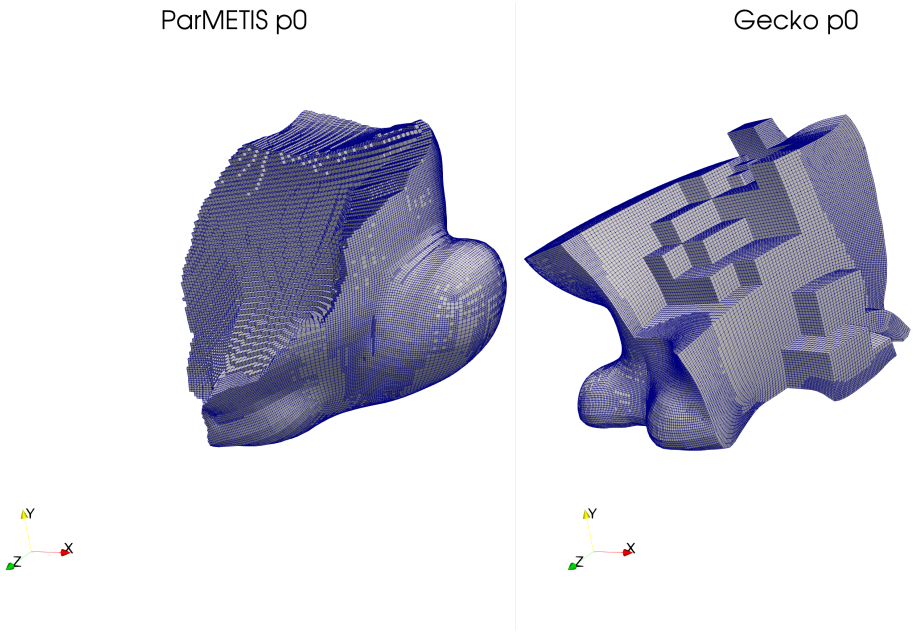


Figure 16: Part zero of the ParMETIS (left) and Gecko (right) four part partitions.

3. CEED DISCRETIZATION LIBRARIES

3.1 MFEM GPU Support

Version 4.0 of MFEM introduces initial support for hardware accelerators, such as GPUs, and programming models, such as CUDA, OCCA, libCEED, RAJA and OpenMP in the library. This support is based on new backends and kernels working seamlessly with a new lightweight memory spaces manager. Several of the MFEM example codes (ex1, ex1p, ex6, and ex6p) and the Laghos miniapp can now take advantage of this GPU acceleration.

One main feature of the MFEM performance portability approach is the ability to select the backends at runtime: they can be mixed to take full advantage of heterogeneous architectures. Algorithms well suited for multi-core CPUs can still be used, while kernels more suited for GPUs can be launched on these architectures.

Most of the kernels are based on a single source, while still offering good performance and being able to use efficiently hardware resources at the developer level. This modularity follows MFEM's design philosophy and is made possible by integrating the support directly at the library's finite element components level. Many linear algebra and finite element operations can now benefit fully from the new GPU acceleration.

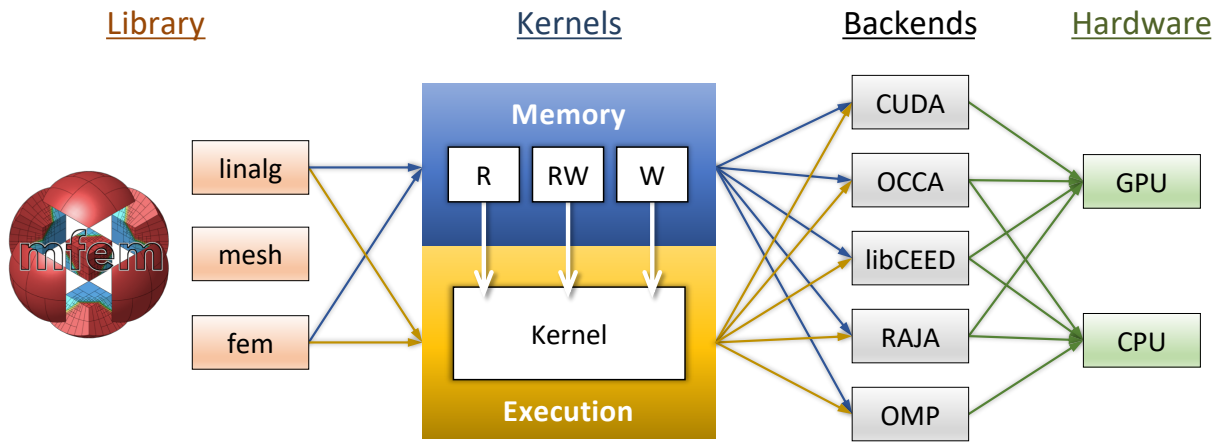


Figure 17: Diagram of MFEM's modular design for accelerator support, combining flexible memory management with runtime-selectable backends for executing key finite element and linear algebra kernels.

Figure 17 illustrates the main components of MFEM's modular design for accelerator support. The *Library* side of MFEM (on the left) represents the software components where new kernels have been added. The following components have been extended with new accelerated *kernels*:

- The `linalg` directory: most operations in class `Vector` and some operations (e.g. matvec) in class `SparseMatrix`. Other classes, such as the Krylov solvers and time-stepping methods, are automatically executed on the device because they are written in terms of `Vector` operations.
- The `mesh` directory: the computation of the so-called *geometric factors*.
- The `fem` directory: the *mass* and *diffusion* `BilinearFormIntegrator`s, the *element restriction* operators associated with class `FiniteElementSpace`, and the matrix-free action of the `BilinearForm` class.

The integration of the kernels has been made at the *for-loop* level. Existing code has been transformed to use a new *for-loop* abstraction defined as a set of new `MFEM_FORALL` macros, in order to take advantage of various *backends* supported via the new macros. This approach allows for gradual code transformations that are not too disruptive for both, MFEM developers and users. Existing applications based on MFEM should be able to continue to work as before and have an easy way to transition to accelerated kernels. Another requirement is to allow interoperability with other software components or external libraries that MFEM could be used in conjunction with: HYPRE, PETSc, SUNDIALS, etc.

The main challenge in this transition to *kernel-centric* implementation is the need to transform existing algorithms to take full advantage of the increased levels of parallelism in the accelerators while maintaining good performances on standard CPU architectures. Another important aspect is the need to manage memory allocation and transfers between the CPU (host) and the accelerator (device). In MFEM, this is achieved using a new `Memory` class that manages a pair of host+device pointers and provides a simple interface for copying or moving the data when needed. An important feature of this class is the ability to work with externally allocated host and/or device pointers which is essential for interoperability with other libraries.

Lambda-capturing for-loop *bodies*

There are multiple ways to write kernels, but one of the easiest ways, from the developer's point of view, is to turn *for-loop* bodies into *kernels* by keeping the bodies unchanged and having a way to *wrap* and *dispatch* them toward native backends. This can be easily done for the first outer for-loop using standard `C++11` features, however, additional care is required when one wants to address deeper levels of parallelism.

```
1: void PADiffusionSetup2D(const int Q1D, const int NE, const double COEFF,
2:                         const Array<double> &w, const Vector &j, Vector &op) {
3:   const int NQ = Q1D*Q1D;
4:   auto W = w.Read();
5:   auto J = Reshape(j.Read(), NQ, 2, 2, NE);
6:   auto y = Reshape(op.Write(), NQ, 3, NE);
7:   MFEM_FORALL(e, NE, {
8:     for (int q = 0; q < NQ; ++q) {
9:       const double J11 = J(q,0,0,e);
10:      const double J21 = J(q,1,0,e);
11:      const double J12 = J(q,0,1,e);
12:      const double J22 = J(q,1,1,e);
13:      const double c_detJ = W[q] * COEFF / ((J11*J22)-(J21*J12));
14:      y(q,0,e) =  c_detJ * (J12*J12 + J22*J22); // 1,1
15:      y(q,1,e) = -c_detJ * (J12*J11 + J22*J21); // 1,2
16:      y(q,2,e) =  c_detJ * (J11*J11 + J21*J21); // 2,2
17:    }
18:  });
19: }
```

Figure 18: MFEM kernel: diffusion setup kernel for partial assembly.

Figure 18 provides an example of a typical kernel in MFEM: the diffusion setup (partial assembly) in 2D. The kernel is structured as follows:

- Lines `4` to `6` are the portion of the kernel where the pointers are requested from the memory manager (presented in the next sub-section) and turned into tensors with given shapes.
- Line `7` holds the `MFEM_FORALL` *wrapper* of the first outer *for-loop*, with the iterator, the range, and the *for-loop* body.
- Lines `9` to `16` are the core of the computation and show how to use the tensors declared before entering the kernel.

Table 5 presents two versions of the 2D diffusion action (matvec) kernel when using partial assembly. On the left is the baseline version that employs one-level parallelism over mesh elements. Lines `10` to `16` are the memory requests, line `18` is the *for-loop* wrapper, and the rest is typical kernel code with inner nested for-loops. On the right side of the table, an improved version of the same kernel is presented which makes use of additional capabilities when supported by the respective backend. These include: (i) the use of another level of parallelism withing the lambda body – a block of threads of arbitrary size (from 1 to thousands) – allowing for parallelization within a mesh element; (ii) utilize *shared* memory as a fast scratch memory shared withing the thread block. This kernel is the one used both for the OpenMP and the CUDA backends.

Table 5: MFEM diffusion kernel for partial assembly in 2D. The left version does not use inner for loops. The right version takes advantage of shared memory, inner for-loops mapped to blocks of threads with arbitrary sizes.

```
template<int T_D1D = 0, int T_Q1D = 0>
void PADiffusionApply2D(const int NE,
                        const Array<double> &b, const Array<double> &g,
                        const Array<double> &bt, const Array<double> &gt,
                        const Vector &_op, const Vector &_x, Vector &_y,
                        const int d1d = 0, const int q1d = 0) {
  const int D1D = T_D1D ? T_D1D : d1d;
  const int Q1D = T_Q1D ? T_Q1D : q1d;

  auto B = Reshape(b.Read(), Q1D, D1D);
  auto G = Reshape(g.Read(), Q1D, D1D);
  auto Bt = Reshape(bt.Read(), D1D, Q1D);
  auto Gt = Reshape(gt.Read(), D1D, Q1D);
  auto op = Reshape(_op.Read(), Q1D*Q1D, 3, NE);
  auto x = Reshape(_x.Read(), D1D, D1D, NE);
  auto y = Reshape(_y.ReadWrite(), D1D, D1D, NE);

  MFEM_FORALL(e, NE,
  {
    constexpr int max_D1D = T_D1D ? T_D1D : MAX_D1D;
    constexpr int max_Q1D = T_Q1D ? T_Q1D : MAX_Q1D;
    double grad[max_Q1D][max_Q1D][2];
    for (int qy = 0; qy < Q1D; ++qy)
      for (int qx = 0; qx < Q1D; ++qx)
        grad[qy][qx][0] = grad[qy][qx][1] = 0.0;
    for (int dy = 0; dy < D1D; ++dy) {
      double gradX[max_Q1D][2];
      for (int qx = 0; qx < Q1D; ++qx) {
        gradX[qx][0] = gradX[qx][1] = 0.0;
      }
      for (int dx = 0; dx < D1D; ++dx) {
        const double s = x(dx,dy,e);
        for (int qx = 0; qx < Q1D; ++qx) {
          gradX[qx][0] += s * B(qx,dx);
          gradX[qx][1] += s * G(qx,dx);
        }
      }
      for (int qy = 0; qy < Q1D; ++qy) {
        const double wy  = B(qy,dy);
        const double wDy = G(qy,dy);
        for (int qx = 0; qx < Q1D; ++qx) {
          grad[qy][qx][0] += gradX[qx][1] * wy;
          grad[qy][qx][1] += gradX[qx][0] * wDy;
        }
      }
    }
    // Calculate Dxy, xDy in plane
    for (int qy = 0; qy < Q1D; ++qy) {
      for (int qx = 0; qx < Q1D; ++qx) {
        const int q = qx + qy * Q1D;
        const double O11 = op(q,0,e);
        const double O12 = op(q,1,e);
        const double O22 = op(q,2,e);
        const double gradX = grad[qy][qx][0];
        const double gradY = grad[qy][qx][1];
        grad[qy][qx][0] = (O11 * gradX) + (O12 * gradY);
        grad[qy][qx][1] = (O12 * gradX) + (O22 * gradY);
      }
    }
    for (int qy = 0; qy < Q1D; ++qy) {
      double gradX[max_D1D][2];
      for (int dx = 0; dx < D1D; ++dx)
        gradX[dx][0] = gradX[dx][1] = 0;
      for (int qx = 0; qx < Q1D; ++qx) {
        const double gX = grad[qy][qx][0];
        const double gY = grad[qy][qx][1];
        for (int dx = 0; dx < D1D; ++dx) {
          const double wx  = Bt(dx,qx);
          const double wDx = Gt(dx,qx);
          gradX[dx][0] += gX * wDx;
          gradX[dx][1] += gY * wx;
        }
      }
      for (int dy = 0; dy < D1D; ++dy) {
        const double wy  = Bt(dy,qy);
        const double wDy = Gt(dy,qy);
        for (int dx = 0; dx < D1D; ++dx) {
          y(dx,dy,e) += ((gradX[dx][0] * wy) + (gradX[dx][1] * wDy));
        }
      }
    }
  });
}
```

```
template<int T_D1D = 0, int T_Q1D = 0, int T_NBZ = 0>
static void SmemPADiffusionApply2D(const int NE,
                        const Array<double> &_b, const Array<double> &_g,
                        const Array<double> &_bt, const Array<double> &_gt,
                        const Vector &_op, const Vector &_x, Vector &_y,
                        const int d1d = 0, const int q1d = 0) {
  const int D1D = T_D1D ? T_D1D : d1d;
  const int Q1D = T_Q1D ? T_Q1D : q1d;
  constexpr int NBZ = T_NBZ ? T_NBZ : 1;
  constexpr int MQ1 = T_Q1D ? T_Q1D : MAX_Q1D;
  constexpr int MD1 = T_D1D ? T_D1D : MAX_D1D;

  auto b = Reshape(_b.Read(), Q1D, D1D);
  auto g = Reshape(_g.Read(), Q1D, D1D);
  auto op = Reshape(_op.Read(), Q1D*Q1D, 3, NE);
  auto x = Reshape(_x.Read(), D1D, D1D, NE);
  auto y = Reshape(_y.ReadWrite(), D1D, D1D, NE);

  MFEM_FORALL_2D(e, NE, Q1D, Q1D, NBZ,
  {
    const int tidz = MFEM_THREAD_ID(z);
    MFEM_SHARED double sBG[2][MQ1*MD1];
    double (*B)[MD1] = (double (*)[MD1]) (sBG+0);
    double (*G)[MD1] = (double (*)[MD1]) (sBG+1);
    double (*Bt)[MQ1] = (double (*)[MQ1]) (sBG+0);
    double (*Gt)[MQ1] = (double (*)[MQ1]) (sBG+1);
    MFEM_SHARED double Xz[NBZ][MD1][MD1];
    MFEM_SHARED double GD[2][NBZ][MD1][MQ1];
    MFEM_SHARED double GQ[2][NBZ][MD1][MQ1];
    double (*X)[MD1] = (double (*)[MD1])(Xz + tidz);
    double (*DQ0)[MD1] = (double (*)[MD1])(GD[0] + tidz);
    double (*DQ1)[MD1] = (double (*)[MD1])(GD[1] + tidz);
    double (*QQ0)[MD1] = (double (*)[MD1])(GQ[0] + tidz);
    double (*QQ1)[MD1] = (double (*)[MD1])(GQ[1] + tidz);
    MFEM_FOREACH_THREAD(dy,y,D1D) {
      MFEM_FOREACH_THREAD(dx,x,D1D) { X[dy][dx] = x(dx,dy,e); }
    }
    if (tidz == 0) {
      MFEM_FOREACH_THREAD(d,y,D1D) {
        MFEM_FOREACH_THREAD(q,x,Q1D) {
          B[q][d] = b(q,d); G[q][d] = g(q,d); } }
    }
    MFEM_SYNC_THREAD;
    MFEM_FOREACH_THREAD(dy,y,D1D) {
      MFEM_FOREACH_THREAD(qx,x,Q1D) {
        double u = 0.0;
        double v = 0.0;
        for (int dx = 0; dx < D1D; ++dx) {
          const double coords = X[dy][dx];
          u += B[qx][dx] * coords;
          v += G[qx][dx] * coords;
        }
        DQ0[dy][qx] = u;
        DQ1[dy][qx] = v;
      }
    }
    MFEM_SYNC_THREAD;
    MFEM_FOREACH_THREAD(qy,y,Q1D) {
      MFEM_FOREACH_THREAD(qx,x,Q1D) {
        double u = 0.0;
        double v = 0.0;
        for (int dy = 0; dy < D1D; ++dy) {
          u += DQ1[dy][qx] * B[qy][dy];
          v += DQ0[dy][qx] * G[qy][dy];
        }
        QQ0[qy][qx] = u;
        QQ1[qy][qx] = v;
      }
    }
    MFEM_SYNC_THREAD;
    MFEM_FOREACH_THREAD(qy,y,Q1D) {
      MFEM_FOREACH_THREAD(qx,x,Q1D) {
        const int q = (qx + ((qy) * Q1D));
        const double O11 = op(q,0,e);
        const double O12 = op(q,1,e);
        const double O22 = op(q,2,e);
        const double gX = QQ0[qy][qx];
        const double gY = QQ1[qy][qx];
        QQ0[qy][qx] = (O11 * gX) + (O12 * gY);
        QQ1[qy][qx] = (O12 * gX) + (O22 * gY); } }
    // ...
  });
}
```

Memory management

Before entering each kernel, the pointers that will be used in it have to be requested from the new `Memory` class which acts as the *frontend* of the internal lightweight MFEM memory manager. Access to the pointers stored by the `Memory` class is requested using three modes: `Read`-only, `Write`-only, and `ReadWrite`. These access types allow the memory manager to seemlessly copy or move data to the device when needed. Portions of the code that do not use acceleration (i.e. run on CPU) need to request access to the `Memory` using the *host* versions of the three access methods: `HostRead`, `HostWrite`, and `HostReadWrite`. The use of these access types allows the memory manager to minimize memory transfers between the host and the device. The pointers returned by the three access methods can be reshaped as *tensors* with some given geometry using the function `Reshape` which then allows for easy multi-dimensional indexing inside the computational kernels, see e.g. Table 5.

In addition to holding the *host+device* pointers, the `Memory` class keeps extra meta-data in order to keep track of the usage of the different memory spaces. For example, if a vector currently residing in device memory is temporarily needed on the host where it will not be modified (e.g. to save the data to a file), the host code can use `HostRead` to tell the memory manager to copy the data to the host while also telling it that the copied data will not be modified; using this information, the memory manager knows that a subsequent call to, say, `Read` will not require a memory copy from host to device.

Initial results

Figure 19 and Table 6 present initial performance results measured on a Linux desktop with a Quadro GV100 GPU, sm_70, CUDA 10.1, and Intel(R) Xeon(R) Gold 6130 CPU @ 2.10GHz. Note that this configuration is very similar to a compute nodes of LLNL's Sierra machine. Single-core, multi-core CPU, and single-GPU performance for different discretization orders is shown, keeping the total number of degrees of freedom (DOFs) constant at 1.3 millions in 2D. Results from all backends supported in MFEM 4.0, as well as recent results based on the libCEED library (integrated with MFEM) are included. The plot clearly shows that GPU acceleration offers a significant gain in performance relative to multi-core CPU. Comparing the single-core CPU backends, we see that libCEED (`ceed-avx` and `ceed-libxsmm`) brings better performance for orders above two. For GPUs, libCEED (`ceed-cuda`) presents an improvement over all other GPU backends.

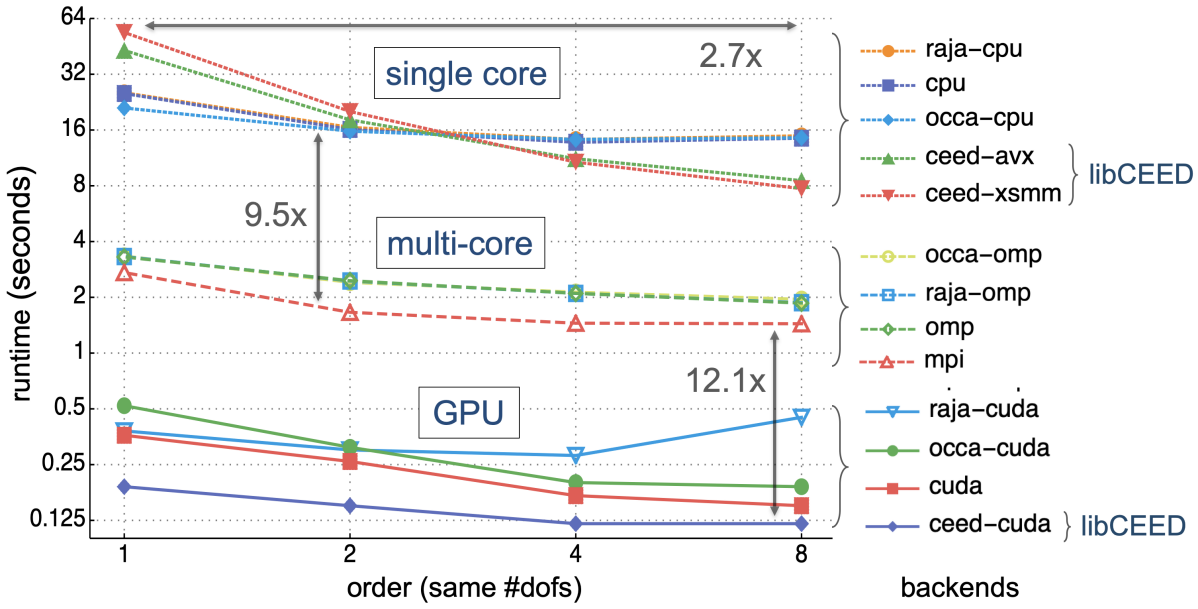


Figure 19: Initial results with MFEM-4.0: Example 1, 200 CG-PA iterations, 2D, 1.3M dofs, GV100, sm_70, CUDA 10.1, Intel(R) Xeon(R) Gold 6130@2.1GHz

	1	2	4	8
occa-cuda =	00.52	00.31	00.20	00.19
raja-cuda =	00.38	00.30	00.28	00.45
cuda =	00.36	00.26	00.17	00.15
ceed-cuda =	**00.19**	**00.15**	**00.12**	**00.12**
occa-omp =	03.34	02.41	02.13	01.95
raja-omp =	03.32	02.45	02.10	01.87
omp =	03.30	02.46	02.10	01.86
mpi =	**02.72**	**01.66**	**01.45**	**01.44**
occa-cpu =	**21.05**	**15.77**	14.23	14.53
raja-cpu =	25.42	16.53	14.22	14.88
cpu =	25.18	16.11	13.73	14.45
ceed-avx =	43.04	18.16	11.20	08.53
ceed-xsmm =	53.80	20.13	**10.73**	**07.72**

Table 6: Initial results with MFEM-4.0: Example 1, 200 CG-PA iterations, 2D, 1.3M dofs, GV100, sm_70, CUDA 10.1, Intel(R) Xeon(R) Gold 6130@2.1GHz. Underlined are the best performing backends in the CPU (blue), multi-core (green) and GPU (red) categories.

3.2 libCEED Benchmarking

CPU backend optimizations and expansion of the libCEED benchmarking was incorporated this quarter. The CPU backends were updated to more efficiently handle the l-vectors in memory, and the benchmarking suite was expanded to include more CEED benchmark problems and more comprehensive summary outputs. Also, a p-multigrid example was developed using PETSc's multigrid preconditioner interface.

3.2.1 CPU Backend Optimization

The CPU backends were updated to use partial l-vectors for the current block of elements rather than forming full l-vectors. This enhancement offered mild performance improvement. The previous full l-vector capability was retained as the `/cpu/self/ref` backends to allow for future performance comparisons. The improvements can be seen in the `/cpu/self/opt` backends.

3.2.2 Benchmarking libCEED Performance

The libCEED team has worked to improve the overall performance of the library with different backends. Benchmarking was performed using PETSc implementations of several CEED Benchmark Problems. In particular, we are going to show here results obtained for BP1, on a Knight Landing node (Intel Xeon Phi 7230 SKU 1.3 GHz), on the Theta (ALCF) cluster, for libCEED built with the intel-18 compiler, and on a Skylake node (2x Intel Xeon Platinum 8180M CPU 2.50GHz), tested with both the intel-19 and the gcc-8 compilers.

We begin by showing the results obtained on the KNL node (Intel Xeon Phi 7230 SKU 1.3 GHz), for a totality of 64 ranks. The first eight plots show the results of BP1 using the reference backend in pure C, and three optimized backends: the `/cpu/self/opt/*` backend that does not form the E-vector, but processes the desired element(s) directly; the `/cpu/self/avx/*` backend that relies upon AVX instructions to provide vectorized CPU performance; and the `/cpu/self/xsmm/*` backend that relies upon the LIBXSMM package to provide vectorized CPU performance. Each of these backends have been tested in serial and blocked implementations, that use internal vectorization for each element and external vectorization across blocks of eight elements, respectively.

We proceed by showing results obtained on the Skylake (2 Intel Xeon Platinum 8180M CPU 2.50GHz), for a totality of 56 ranks. For these results, we choose to plot the performance with respect to time on the x-axis. We see that the left edge of the graphs defines strong scaling. We can see how the blocked implementation is

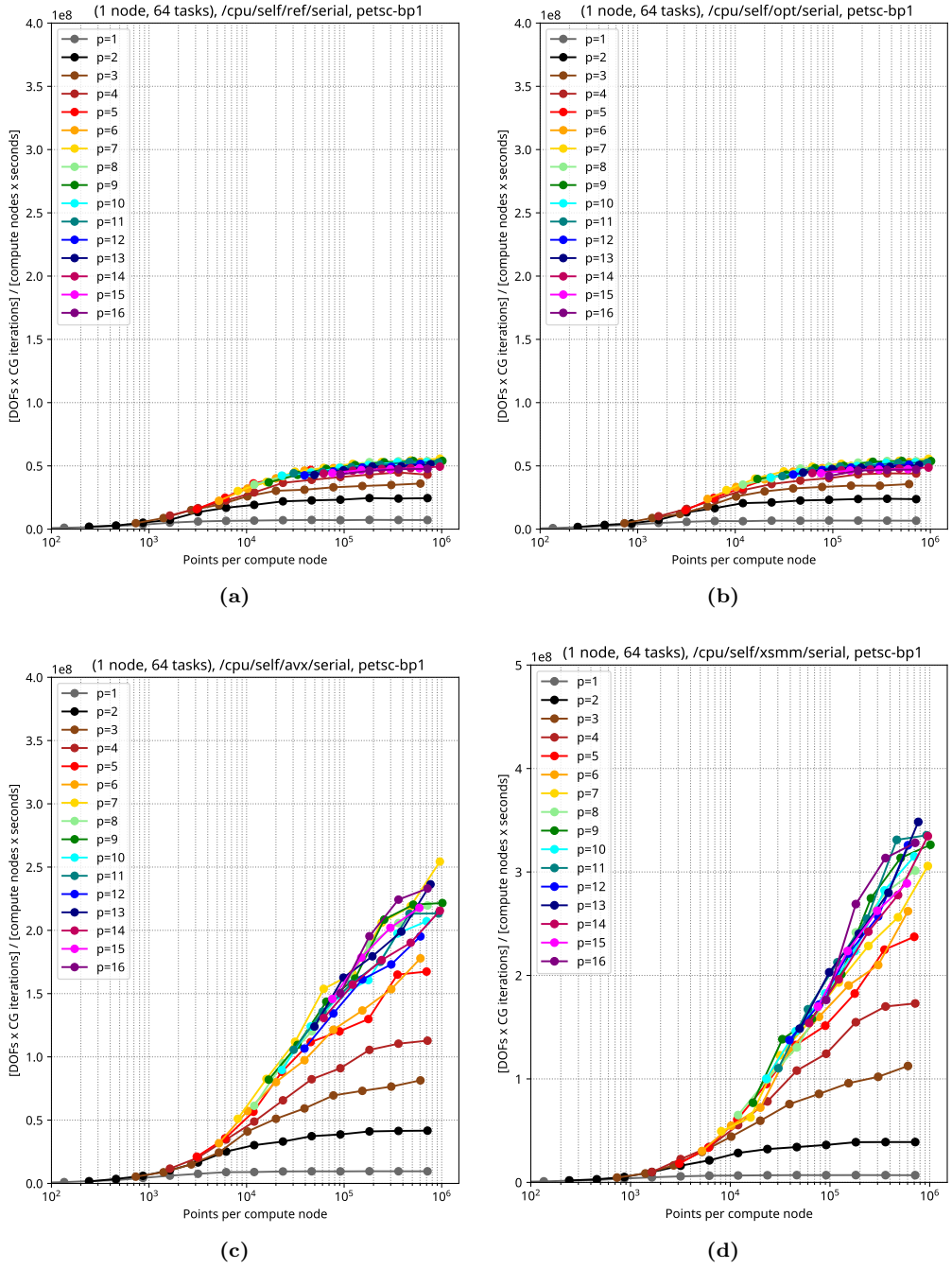


Figure 20: BP1 for `/cpu/self/*/serial` for the `ref`, `opt`, `avx`, and `xsmm` backends respectively, on Intel Xeon Phi 7230 SKU 1.3 GHz with intel-18 compiler. Note the different y-axis for the `xsmm` resutls.

better for low order, but eventually the serial one is better and attains faster minimum time to solution (left edge of the performance curve).

We notice that in all our results, the blocked implementation outperforms the serial one (compare figures 20 and 21, and 22(a)-(c) with 22(d)-(f)), except for the case of the `/cpu/self/xsmm/blocked` on the Skylake node with the gcc-8 compiler (compare 23(c) with 23(f)).

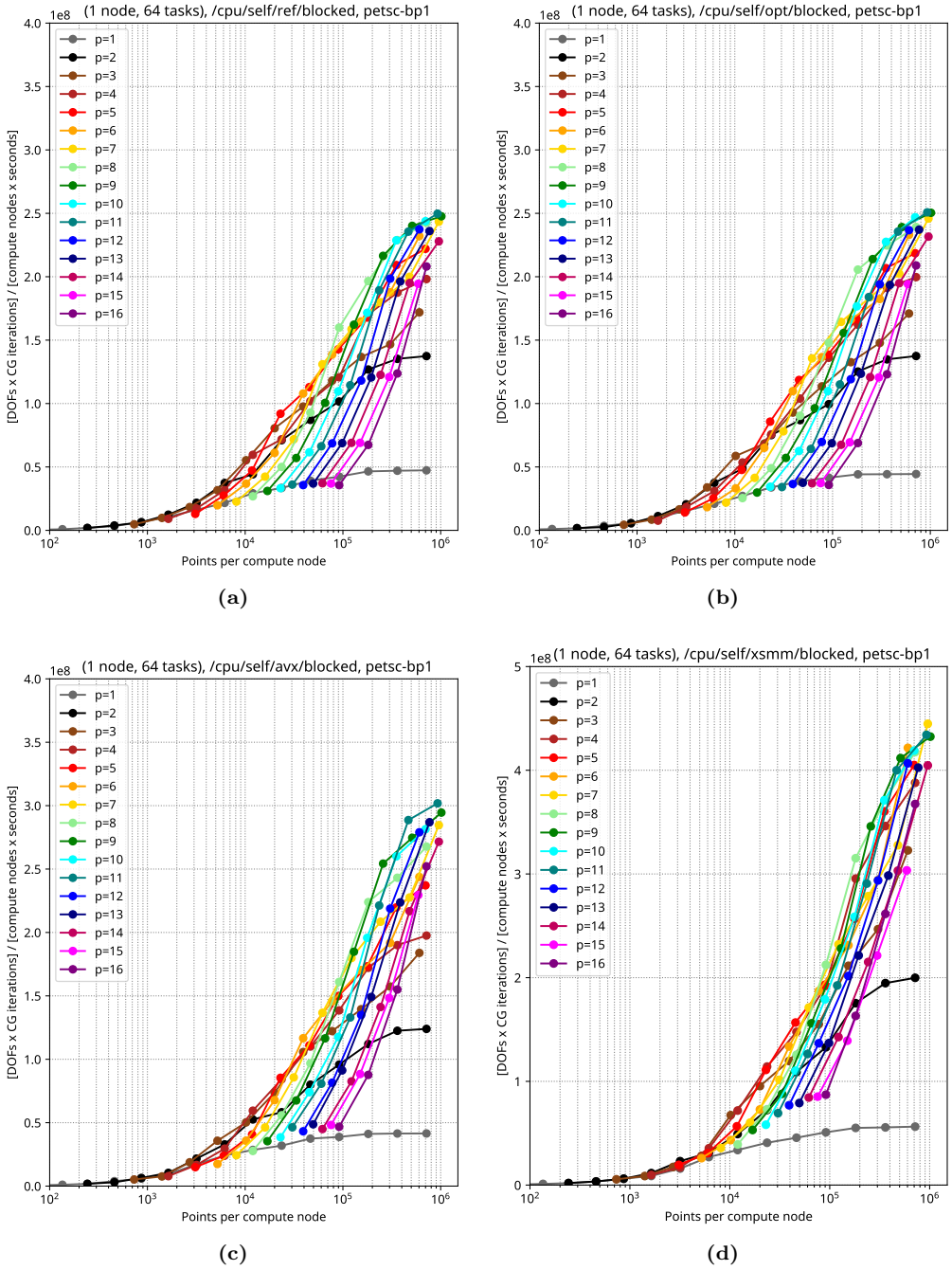


Figure 21: BP1 for `/cpu/self/*/blocked` for the `ref`, `opt`, `avx`, and `xsmm` backends respectively, on Intel Xeon Phi 7230 SKU 1.3 GHz with intel-18 compiler. Note the different y-axis for the `xsmm` resutls.

In addition to these results, benchmark problems 2, 4, 5, and 6 were added to libCEED. A more comprehensive summary output and improved plotting was also added.

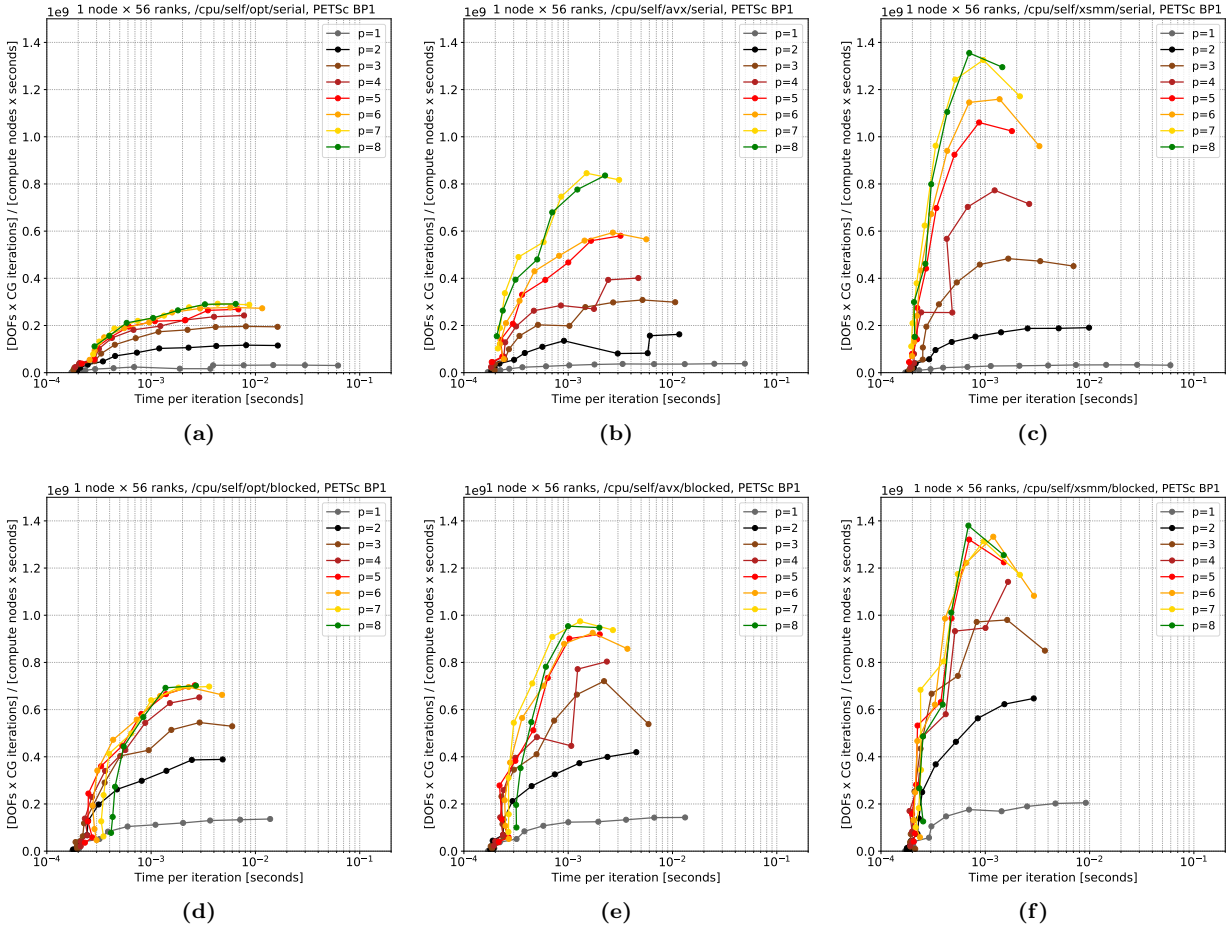


Figure 22: BP1 for `/cpu/self/*/serial` (top) and `/cpu/self/*/blocked` (bottom) for the `opt`, `avx`, and `xsmm` backends respectively, on Skylake (2x Intel Xeon Platinum 8180M CPU 2.50GHz) with the intel-19 compiler.

3.2.3 Multigrid Example

A p-multigrid example based on the Poisson problem was added to libCEED. This example demonstrates the flexibility of the CEED operator decomposition, as all operators are implemented in libCEED, to include p-restriction, p-prolongation, and the diffusion operator. Initial performance results highlight the need for preconditioning based upon the CEED operator decomposition.

3.3 NekRS Developments

NekRS is a newly developed accelerator-oriented version of Nek5000 that is based on the libParanumal library (`https://www.paranumal.com`), which is being developed with CEED support by the Warburton group at Virginia Tech. libParanumal (libP) is based on the open concurrent compute abstraction (OCCA), which is a library and thread programming language to portably program CPUs and GPUs that is also coming from the Warburton group. OCCA uses JIT compilers to generate optimized CUDA, OpenCL, and OpenMP kernels targeting the architectures that are planned for the ECP. The libP kernels are tuned to sustain the maximal throughput (as established through roofline analysis) on current-generation accelerators and are able to sustain 2 (of a peak-possible 7) double-precision TFLOPS for key spectral element operators on a single Nvidia V100 on Summit.

The overarching strategy for NekRS is to exploit the performance and portability of the libP/OCCA

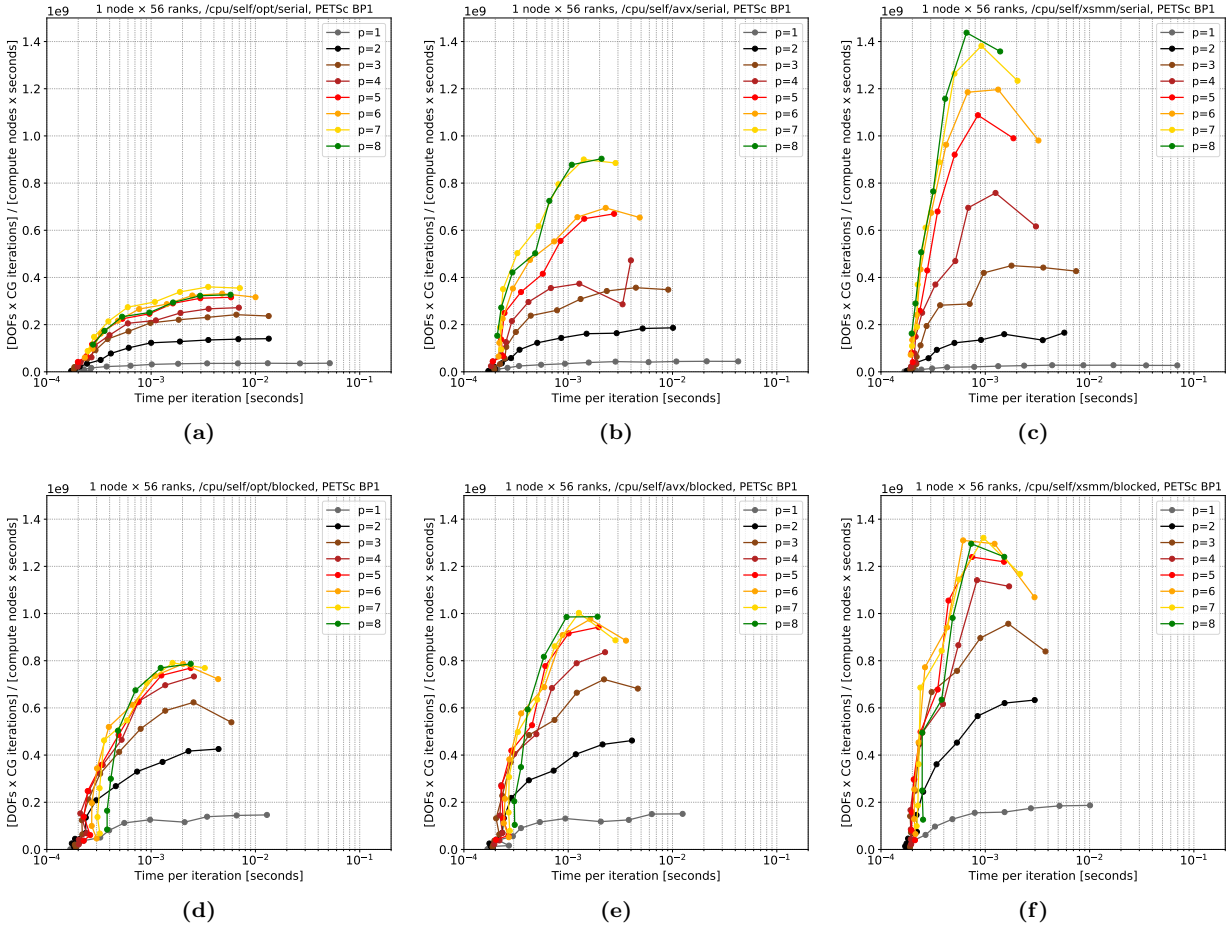


Figure 23: BP1 for `/cpu/self/*/serial` (top) and `/cpu/self/*/blocked` (bottom) for the `opt`, `avx`, and `xsmm` backends respectively, on Skylake (2x Intel Xeon Platinum 8180M CPU 2.50GHz) with the gcc-8 compiler.

constructs within the scalable Nek5000 framework that has been developed over the past decades. The ANL and Virginia Tech teams have worked closely to implement stable, exponentially-convergent, Navier-Stokes formulations that are 2nd- or 3rd-order accurate in time. The formulation includes characteristics-based algorithms that are able to exceed traditional Courant stability-limited timestep sizes by a factor of five to ten. Extensive testing is underway, with exponential convergence established for analytical solutions to the Navier-Stokes equations for both steady and unsteady 3D problems using either Dirichlet or periodic boundary conditions.

Large-scale tests on Summit have also been performed with NekRS. Figure 24 demonstrates validation on the ExaSMR 17×17 rod bundle test problem (pictured on the right). Figure 24 (left) shows the pressure iteration counts for 30 million elements (total 10.4 billion grid points). NekRS performed the simulation with 0.5144 sec per step using 3960 GPUs. We see that the current Chebyshev-smoothed p-multigrid algorithm in libP does an excellent job in yielding order-independent iteration counts for this benchmark.

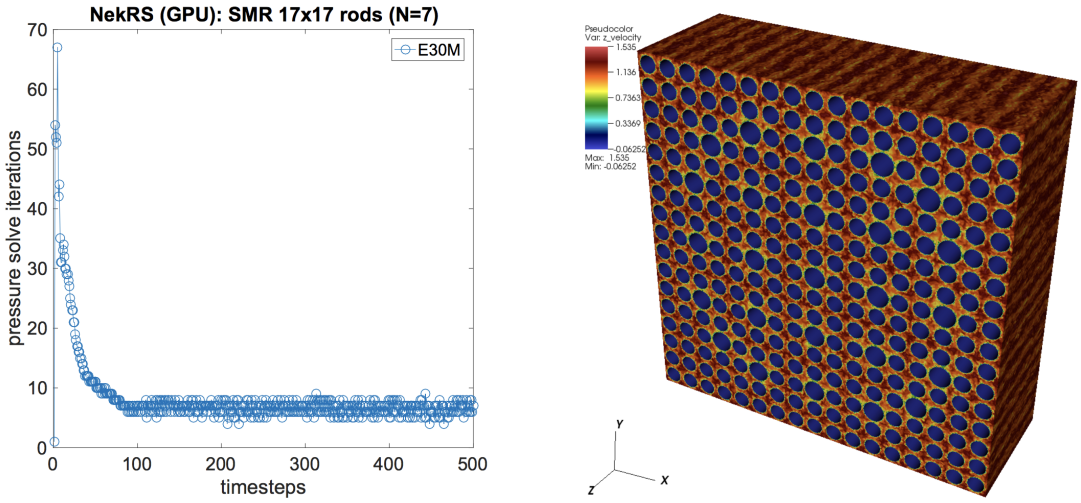


Figure 24: NekRS 17×17 rod-bundle validation on OLCF Summit V100 GPUs: (left) NekRS pressure iteration history for $E = 30M$ spectral elements (on 3960 GPUs) of order $N = 7$, (total 10.4 billion grid points); (right) computational domain and solution for $E = 3M$ case at $Re = 5000$.

4. ECP APPLICATION COLLABORATIONS

4.1 ExaSMR

Rod bundles represent an essential component of both classical and modern nuclear power plants. In most concepts, fuel assemblies comprise a large number of packed cylinder arrays of rods containing the nuclear fuel and surrounded by the liquid coolant. The prediction of the temperature distribution within the rod bundle is of major importance in the reactor design and is determined primarily by the flow characteristics in the subchannels bounded by adjacent rods. The problem lies however in the high Reynolds number that characterize these flows, which dictates the required local resolution and also an increment in the time-scale separation. This leads to smaller time steps and longer transients, eventually making these simulations unfeasible.

In collaboration with the ECP-ExaSMR team, CEED team explored overcoming the excessive cost of large eddy simulations (LES) of full-length heated rod bundle calculations. The recently developed CEED team's steady-state solver has been applied for solving the temperature field in their pseudo-RANS (Reynolds-averaged Navier-Stokes) approach [26].

While LES of an entire rod bundle is prohibitively expensive in most cases, the simulation of a small axial section of a bare rod is typically affordable. ExaSMR team's strategy is to use average and fluctuating information from a small but highly resolved LES of a single rod to build custom models of the turbulent thermal stresses that can be employed in the simulation of the entire full-length rod (see Figure 25). The idea reported here consists of using the information granted by the small LES calculation to determine the appropriate turbulence viscosity or turbulent thermal diffusivity that could be used to solve only for the temperature field in their pseudo-RANS approach.

While the fluid viscosity does not depend on temperature under a valid assumption for the cases that are considered for the proposed methodology, velocity equations are completely unlinked to the temperature field. Using the Reynolds-averaging operator, one may divide instantaneous fields in an average part (time average for statistically steady flows) and a fluctuating part (i.e., $u_i = \bar{u}_i + u'_i$). When this is applied to the NS equations, the RANS equations are obtained as

$$\frac{\partial \overline{u}_i}{\partial t} + \frac{\partial}{\partial x_j}\left(\overline{u}_i \overline{u}_j\right) = -\frac{1}{\rho}\frac{\partial \overline{p}}{\partial x_i} + \frac{\partial}{\partial x_j}\left[\nu\left(\frac{\partial \overline{u}_i}{\partial x_j} + \frac{\partial \overline{u}_j}{\partial x_i}\right)\right] + \frac{\partial}{\partial x_j}\left(-\overline{u'_i u'_j}\right), \tag{3}$$

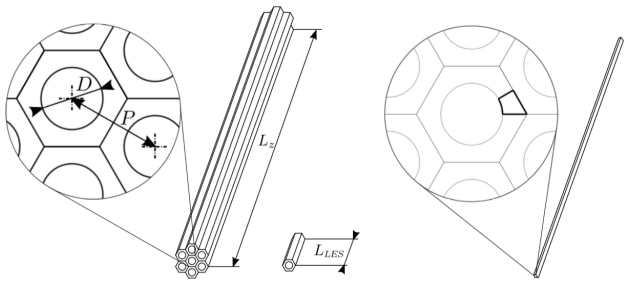


Figure 25: Geometry for bare rod in triangular array [26]. L_{LES} represents the length used for the pseudo-RANS approach, which is shorter than the full length L_z.

$$\frac{\partial \overline{u}_i}{\partial x_i} = 0, \tag{4}$$

$$\frac{\partial \overline{T}}{\partial t} + \overline{u}_j \frac{\partial \overline{T}}{\partial x_j} = \frac{\partial}{\partial x_j}\left(\alpha \frac{\partial \overline{T}}{\partial x_j}\right) + \frac{\partial}{\partial x_j}\left(-\overline{u_j' T'}\right), \tag{5}$$

where the turbulent heat fluxes can be modeled following a gradient diffusion hypothesis,

$$-\overline{u_j' T'} = \alpha_t \frac{\partial \overline{T}}{\partial x_j}. \tag{6}$$

Table 7: CPU time and number of iterations required to reach L_2 errors at the level of 10^{-7}, using 8 KNL nodes (512 MPI ranks) on Argonne's Bebop cluster.

Case	No. of Iterations			CPU Time (min)		
tol	10^{-2}	10^{-4}	10^{-6}	10^{-2}	10^{-4}	10^{-6}
BDF1/EXT	75275	34456	23026	154	97	114
BDF2/EXT	63854	31854	22371	130	90	106
BDF3/EXT	57163	30063	22095	117	85	103
SS	60			12		

Considering the proposed methodology, once the average velocity field and the turbulent diffusivity are obtained by the pseudo-RANS approach, one may solve for the Reynolds-averaged temperature field by considering the transient problem as described in Eq.(5), until a steady state is reached. Standard transient solvers in Nek5000 would need to advect the solution over times $\tau \gg \Delta t$, where Δt is the step size, limited by stability. Alternatively, one approach to accelerate the convergence of the thermal advection is to consider a steady-state solver. In this framework, the equation to be solved by a steady-state solver is

$$\overline{u}_j \frac{\partial \overline{T}}{\partial x_j} = \frac{\partial}{\partial x_j}\left((\alpha + \alpha_t)\frac{\partial \overline{T}}{\partial x_j}\right). \tag{7}$$

Nek5000's steady-state solver is based on a tensor product-based fast diagonalization method implemented for an overlapping Schwarz smoothing technique, integrated into a spectral element multigrid preconditioner [7, 8]. The tensor-product overlapping Schwarz spectral element multigrid preconditioner is formulated for the advection-diffusion operators based on spectral element discretizations and applied as a preconditioner for the GMRES procedure for solving Eq.(7).

We demonstrate the performances of the steady-state (SS) solver and the transient solvers. In particular, the solution of the temperature field with the pseudo-RANS approach was obtained based on the kth-order time discretization schemes involving the BDFk with extrapolations (EXT). Table **??**. The temperature field was initialized to T_0 in the entire domain. To obtain a reference solution, we ran the SS solver until full convergence (machine precision) to produce a steady solution $\overline{T}_{ref}$. In the subsequent simulations, $\overline{T}_{ref}$ was

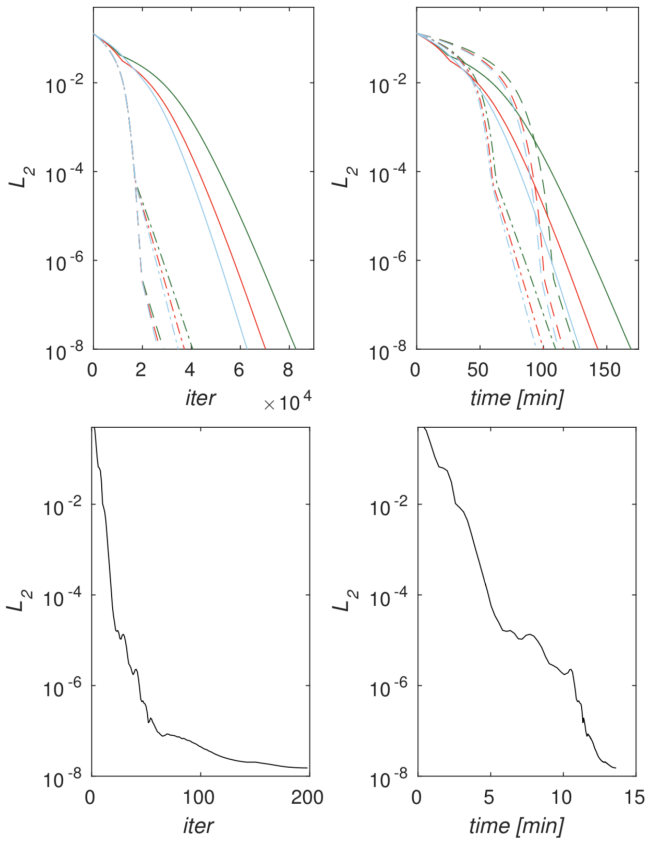


Figure 26: Error norm evolution for the transient solvers with BDF1/EXT (green), BDF2/EXT (red), BDF3/EXT (blue), and the SS solver (black). Transient solver results are shown for the linear solver tolerances of 10^{-2} (—), 10^{-4} (- · -), and 10^{-6} (- - -) with the timestep iterations and simulation time. The SS solver results are shown with the GMRES iterations and simulation time using $E = 21660$ spectral elements with $N = 7$ [26].

taken as a reference, and the error of the predicted $\overline{T}$ with respect to $\overline{T}_{ref}$ was evaluated for each approach at each time step in the L_2 norm.

For the transient solver, the time step was fixed so that the maximum Courant number based on the frozen velocity field was $CFL = 0.5$. Three tolerances of the linear Helmholtz solver were chosen for the convergence criteria of each time step: $tol = 10^{-2}$, 10^{-4}, or 10^{-6}. For both the SS and BDFk/EXT schemes, the simulations were run until they reached the level of errors 10^{-7} in the L_2 norm. We calculated the evolution of the error norm, the required number of iterations, and the simulation time using 8 KNL nodes (512 MPI ranks) on Argonne's Bebop cluster. The results are shown in Figure 26 and listed in Table 7.

Figure 26 shows the convergence behaviors of the transient solution compared with the reference solution for the BDFk/EXT schemes. We observe a significant increase in the computational cost as the tolerance gets tighter with $tol = 10^{-6}$. The significant computational cost of increasing the convergence of each time step (tightening the tolerance to $tol = 10^{-6}$) or increasing the order of accuracy can be clearly extracted from Figure 26 and Table 7. While slower at the beginning, the tighter tolerance in the linear solver might be worthwhile, since the required number of iterations to reach a given level of convergence might be reduced, as demonstrated in Figure 26. For the simulations performed, considering the fastest time discretization scheme and larger tolerance (BDF3/EXT with $tol = 10^{-4}$), the SS solver is still about 7 times faster.

Table 8: Maximum Δt.

Model	case	Re	N+1	max Δt (old)	max Δt (new)	ratio
regularized k-ω	2D channel	1E5	8	1E-3(1E-3)	3E-3(1E-3)	3
			12	1E-4(1E-4)	1E-3(1E-3)	10
			16	4E-5(5E-5)	7E-4(1E-4)	17.5

4.2 ExaWind

In collaboration with ExaWind team, we investigate fast algorithms for airfoil simulations. We currently focus on several different RANS models (such as k-ω, k-τ, k-ϵ) and time stepping approaches for NACA012 geometries. We consider the incompressible RANS approach with 5-equations which can be described as

$$\frac{\partial \rho \mathbf{v}}{\partial t} + \nabla \cdot (\rho \mathbf{v}\mathbf{v}) = -\nabla p + \nabla \cdot \left[(\mu + \mu_t) \left(\nabla \mathbf{v} + \nabla \mathbf{v}^T - \frac{2}{3} \nabla \cdot \mathbf{v} \right) \right], \tag{8}$$

$$\nabla \cdot \mathbf{v} = 0, \tag{9}$$

and the turbulent kinetic energy k and the specific dissipation rate ω described as

$$\frac{\partial (\rho k)}{\partial t} + \nabla \cdot (\rho k \mathbf{v}) = \nabla \cdot (\Gamma_k \nabla k) + G_k - Y_k + S_k, \tag{10}$$

$$\frac{\partial (\rho \omega)}{\partial t} + \nabla \cdot (\rho \omega \mathbf{v}) = \nabla \cdot (\Gamma_\omega \nabla \omega) + G_\omega - Y_\omega + S_\omega, \tag{11}$$

where the diffusion coefficients are

$$\mu_t = \frac{k}{\omega}, \quad \Gamma_k = \mu + \frac{\mu_t}{\sigma_k}, \quad \Gamma_\omega = \mu + \frac{\mu_t}{\sigma_\omega}. \tag{12}$$

Our regularized $k - \omega$ model [30] is based on $\omega = \omega_w + \omega'$, where $\omega_w = \frac{a\nu}{\beta^*_\infty y_w^2}$ is the asymptotic solution of ω at wall, which can be computed by the distance to the wall. The regularized ω' equation is defined as

$$\frac{\partial \rho \omega'}{\partial t} + \nabla \cdot (\rho \omega' \mathbf{v}) = \nabla \cdot (\Gamma_\omega \nabla \omega') + G_\omega - Y_\omega + S_\omega + \nabla \cdot (\Gamma_\omega \nabla \omega_w) - \rho \mathbf{v} \cdot \nabla \omega_w. \tag{13}$$

We improved the stability of this model by linearizing several of the RANS source terms and treating these implicitly. The results are shown in Table 8, demonstrating the maximum timestep size using this approach with the improved ratio for 2D RANS channel flows. We applied this approach to NACA012 simulations for different model approaches and their drag and lift profiles for the zero angle of attack shown in Figure 27 demonstrate good agreement to the reference solutions [3].

4.3 Urban

The urban challenge problem considers the assessment of extreme heat events on buildings in dense urban environments, with up to a few 1000 buildings being modeled during an event. This challenge problem involves coupling of WRF (to define initial weather conditions), Nek5000 (to model heat transfer near buildings), and EnergyPlus (to model heat emissions and energy performance).

In collaboration with the ECP-Urban team, CEED team performed LES simulations of Chicago downtown block, consisting of 20 buildings as shown in Figure 28. Our simulations use the initial and boundary conditions, provided from the simulations by the WRF code. The 20 building block spectral element mesh consists of the total number of elements $E = 143340$ and its simulation with $N = 13$ is performed on ALCF/Mira using 512 to 1024 nodes with $Re \sim 10,000$.

5. OTHER PROJECT ACTIVITIES

5.1 General Interpolation

General purpose interpolation is important for a wide range of science and engineering applications. Examples include Lagrangian particle tracking, nonconforming instances of overlapping Schwarz methods (i.e., overset

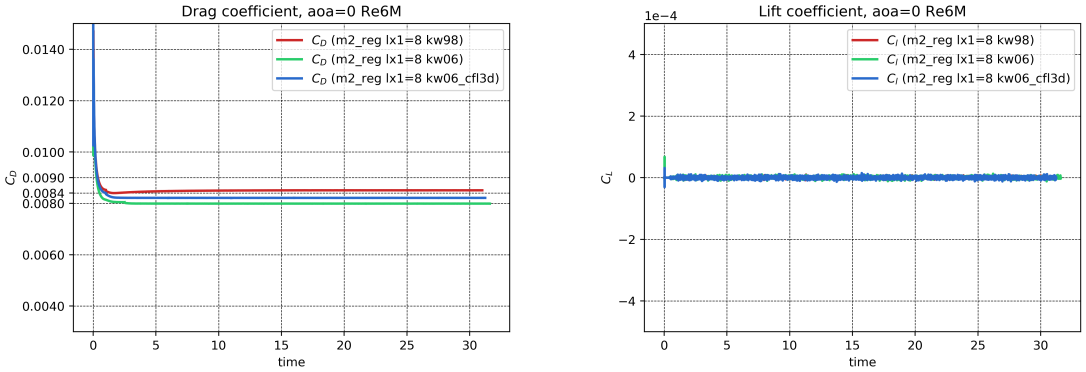


Figure 27: NACA012 drag and lift coefficient profiles for the angle of attack (aoa)=0 with $Re = 6M$ for three different model approachs. lx1=$N + 1$ ($N = 7$).

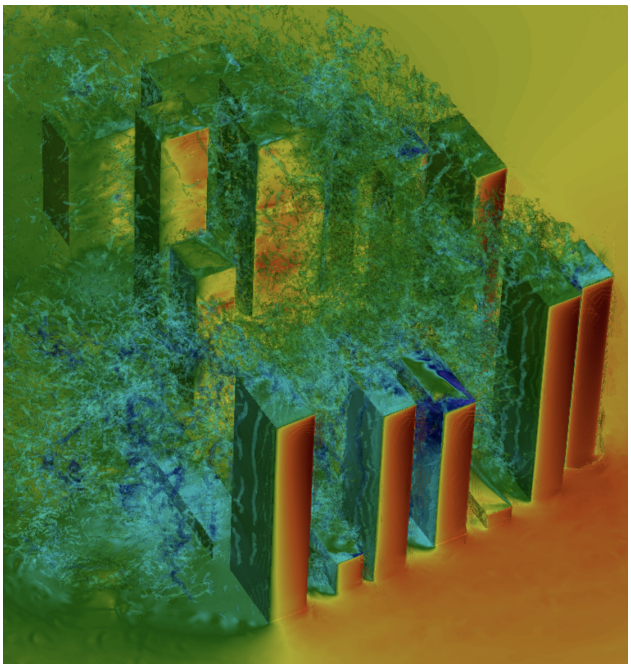


Figure 28: Urban simulations using a spectral element mesh with $E = 143340$ and $N = 13$: vortex flow around the 20 buildings with mixed tall and short heights in Chicago downtown block.

grid methods), and solution interrogation for generating profiles and statistics. Exascale implementations of this statement should be robust, fast, and scalable to millions of processors.

The mathematical statement of the interpolation problem is simply *Find* $u^* = u(\mathbf{x}^*)$ *for any* $\mathbf{x}^*$ *in* Ω. In finite element applications, a scalar field u is represented by piecewise polynomials on non-overlapping subdomains Ω^e such that $\Omega = \cup_e \Omega^e$. The geometry associated with each element is denoted by $\mathbf{x}^e = \mathbf{x}^e(\mathbf{r})$, $\mathbf{r} \in \hat{\Omega}$. For Nth-order curvilinear brick (hexahedral) elements in $\mathbb{R}^3$, we have $\hat{\Omega} = [-1, 1]^3$, and

$$\mathbf{x}^e(\mathbf{r}) \quad = \quad \sum_{k=0}^{N} \sum_{j=0}^{N} \sum_{i=0}^{N} \mathbf{x}^e_{ijk} l_i(r) l_j(s) l_k(t), \tag{14}$$

for one-dimensional basis functions, $l_j \in \mathbb{P}_N(r)$, and $(N+1)^3$ nodal coordinates, $\mathbf{x}^e_{ijk}$. We use a similar expansion for $u^e(\mathbf{r})$, which is the restriction of u to Ω^e. To solve the interpolation problem $u^* = u(\mathbf{x}^*) = u^{e^*}(\mathbf{r}^*)$ we must first find the element, e^*, and coordinates, $\mathbf{r}^* = (r^*, s^*, t^*) \in \hat{\Omega}$, such that $\mathbf{x}^{e^*}(\mathbf{r}^*) = \mathbf{x}^*$. In a distributed-memory parallel computing context, we must also find the processor, $p^* \in [0, P-1]$, which holds element e^*.

We have developed a scalable general purpose interpolation library, *findpts*, written in C that readily links with any Fortran, C, or C++ code. The library provides two key capabilities. First, it determines the computational coordinates $\mathbf{q}^*_j = (e^*_j, p^*_j, \mathbf{r}^*_j)$ associated with any given set of points $\{\mathbf{x}^*_j\} \in \mathbb{R}^3$. Second, through an auxiliary routine, *findpts_eval*, it interpolates any field (or fields) for a given set of computational coordinates. *findpts* and *findpts_eval* require interprocessor communication and must be called by all processors in a given communicator. All communication is effected in $\log_2 P$ message exchanges by virtue of the *gslib*[1] generalized and scalable all-to-all utility, *gs_crystal*, which is based on the crystal router algorithm of [16].

To find $\mathbf{q}^*$ for a given point $\mathbf{x}^*$, *findpts* first uses a hash table to identify processors that could potentially hold the data-source element, e^*. A call to *gs_crystal* exchanges copies of the $\mathbf{x}^*_j$ entries between the originating processors and potential sources. Once there, element-wise bounding boxes further discriminate to determine candidates for e^*. At that point, a trust-region based Newton optimization is used to solve $\mathbf{r}^* = \text{argmin}\|\mathbf{x}^e(\mathbf{r}) - \mathbf{x}^*\|$ Once the Newton-search is complete the full coordinates $\mathbf{q}^*$ are returned to the originating processor. Additionally, findpts also indicates whether a point was found inside an element, on a border, or was not found within the mesh. For points with more than one candidate processor, the originator resolves ties by choosing the source processor contribution for which $\|\mathbf{r}^*\|_\infty$ is minimal. Subsequent to establishment of $\mathbf{q}_j$, calls to *findpts_eval*$(u, \mathbf{q})$ will return the value set $\{u_j\}$ to the originating processor. Here, each originator sends the sets $(e^*_j, \mathbf{r}^*_j)$ to corresponding processors p_j in $\log_2 P$ time through *gs_crystal*. Interpolation of u is performed on the source processors and data is returned to the originators via *gs_crystal*.

We have recently made several improvements in *findpts*. For interpolation, we now retain $(e^*_j, \mathbf{r}^*_j)$ on the source processors so that *findpts_eval* requires only the interpolation step and the second communication phase. With this improvement, the cost of repeated field evaluations (i.e., other fields or the same fields at subsequent time steps), are reduced roughly four-fold. Specifically, for a 5th-order mesh with E =70,400 elements evaluation times for a total of 275,000 uniformly-distributed points were reduced from 0.2 seconds to .05 seconds on the BG/Q Cetus at Argonne, for processor counts ranging from P=4096 to 32768. This new development is part of the latest *gslib* release.

Another development in *findpts* is to exploit the fact that the geometry is often represented as a polynomial of lower degree than the solution. While u^e is represented in $\hat{\Omega}$ by Nth-order polynomials, it's quite possible that $\mathbf{x}^e$ (14) may only be a linear or quadratic polynomial, at least away from the domain boundaries. In that case, the $O(N^3)$ cost incurred for each evaluation of $\mathbf{x}^e(\mathbf{r})$ can be reduced by a factor of $((N+1)/3)^3$ in the Newton search phase of *findpts*. Test runs have shown a four-fold savings for the commonly used value of N=7, and much greater savings for higher values of N.

5.2 Accelerator-Oriented Solvers

Minimizing communication is central to realizing high performance for efficient execution of parallel algorithms. On traditional distributed memory platforms, communication can be broadly characterized as either latency dominated (meaning the messages are of length $m < m_2$, where m_2 is the message size having twice the communication cost of a one-word message), or bandwidth limited, meaning $m > m_2$. There are, however, additional details that depend on the nature of the communication exchange. For example, data transposes (i.e., all-to-alls) can tax the bisection bandwidth of the network, so one must understand the overall capacity between processor groups that are exchanging large volumes of data under such circumstances. For GPUs there is another concern, which is the significant latency between the device and host and the relatively (with respect to processing speed) limited bandwidth in and out of the GPU.

Incompressible Navier-Stokes simulations are predominantly limited by the solution time required for large sparse linear systems with multiple right hand sides (in succession, not simultaneously) and thus can amortize most any reasonable set up cost. Our focus, therefore, is on reducing communication in the *execute* phase of iterative solvers, with particular attention to GPU-based systems. The idea is to exchange local work for a reduction in the *number* of communications. In this vein, we pursue a multilevel strategy due to

[1] *gslib* is a separate library for gather-scatter and other PDE-oriented communications.

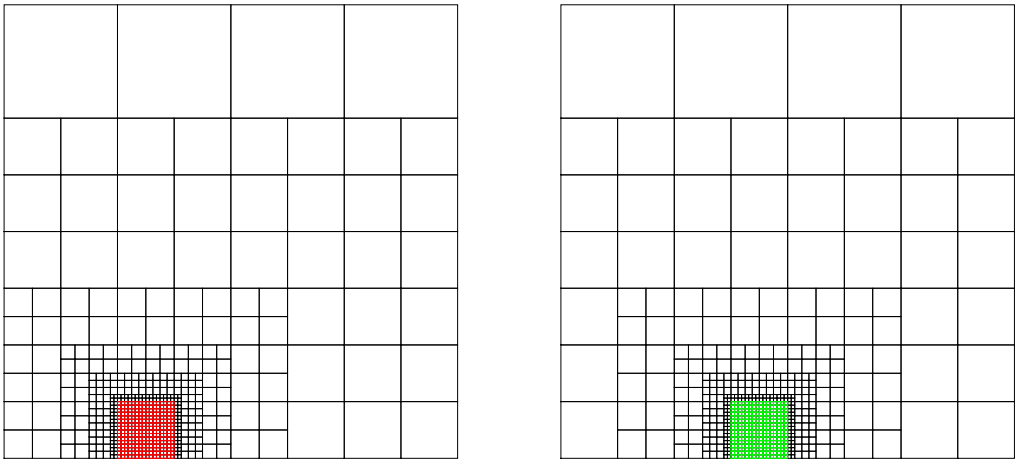


Figure 29: A range decomposition example showing two (of P) local domains (red and green) with their respective extensions.

Randy Bank [5] and, more recently, Mitchell and Manteuffel [27] that has the potential for a low number of iterations and thus a low number of communication phases. The idea is to solve systems on each processor that correspond to the PDE discretized over the full domain with *varying* resolution. This approach, known as range decomposition, leads to systems that are somewhat more expensive to solve than those that arise with standard domain decomposition (i.e., overlapping Schwarz) approaches but they do not require a coarse grid solve and they require relatively few outer iterations when embedded in a Krylov subspace projection (KSP).

To illustrate the approach, we begin by considering the Poisson equation on a uniform domain $\Omega = [0,1]^2$ with $E_x \times E_y$ bilinear finite elements. The domain is partitioned in the usual way across P processors and each processor, $p = 0, \ldots, P-1$, is responsible for updating the solution in its subdomain, Ω_p. The range decomposition strategy is to extend Ω_p to all of Ω with a sequence of successively coarse grids as the distance from Ω_p increases. For a given processor p, we assign a Lagrangian basis function, $\phi_i^p(\mathbf{x})$ to each of these extended-domain gridpoints, $\mathbf{x}_i^p$, $i = 1, \ldots, n_p$, and seek solutions to the Poisson equation in $X_0^{N,p} = \text{span}\{\phi_1^p, \ldots, \phi_{n_p}^p\}$. The right-hand side entries,

$$b_i^p \quad := \quad \int_\Omega \phi_i^p \, f \, dV, \tag{15}$$

require integration of *nonlocal* data, exterior to processor p, as indicated in Fig. 29. With establishment of the $\underline{b}^p$s (15), one solves the globalized problems, $A^p \underline{u}^p = \underline{b}^p$, locally on each processor and then combines the solutions. In this last step, we opt for a *restricted additive Schwarz* (RAS) approach [9], in which each processor p retains only its local domain solution and discards the supporting part of the solution. The localized problems are also solved with a multilevel-preconditioned KSP.

On GPUs, the localized problems are solved to relatively tight tolerances in an effort to reduce the number of outer iterations, each of which requires parallel communication. Fortunately, for the outer KSP, the bulk of the communication occurs at one time—one must transfer a hierarchy of data out of and into each GPU. Properly orchestrated, the inbound and outbound data volume is a fraction of the resident values (and may be in lower precision). On summit, local data volumes will be $\approx 10^6$ words (the estimated strong-scale and memory-bound limit for PDE solvers). Assuming that half of this data must move leads to an estimated data volume of 0.5×10^6 words $\times$ 4 bytes/word $= 2 \times 10^6$ bytes per V100. Summit's internode bandwidth is $\approx 45GB/s$, implying a transfer time of 40μs, which is on par with the latency. Having latency-bound messages that are fully packed with information is precisely the goal of a communication-minimal solution strategy. Any message exchange that is on par with the minimal (e.g., one-word) transfer cost, but which exchanges sufficient information to enable a reduction in iteration count and, hence, *number* of messages is a worthwhile strategy for accelerator-enabled platforms such as Summit.

The CEED team at UIUC has been working on a test suite to implement range decomposition at scale. They are targeting millions of degrees-of-freedom per rank and hundreds of thousands of MPI ranks. Initial runs with 268 million elements on 16384 ranks require only 11 outer iterations to achieve a 10^{-8} reduction in error for the 2D Poisson problem on the square. The team is extending this test suite to 3D and will then port the suite to GPUs. Subsequent to successful testing, the algorithm will be implemented for the pressure solve in NekRS, which is the GPU-oriented version of Nek5000 that is built on top of libParanumal/OCCA.

5.3 Outreach

CEED researchers were involved in a number of outreach activities including: participation in the PETSc User meeting, Second Conceptualization Workshop on Fluid Dynamics Software Infrastructure, FASTMath all hands meeting, PASC19, the Salishan Conference on High Speed Computing, and the inaugural North American High Order Methods Conference (NAHOMCon) where Paul Fischer and Tim Warburton serve as the Scientific Committee and Tzanio Kolev and Misun Min presented keynote talks. There has also been ongoing planning for CEED's 3rd annual meeting to be held at Virginia Tech in August 6-8, 2019.

6. CONCLUSION

The usefulness of PDE models to DOE domain scientist requires the ability of those technologies to model the physics of interest. However, this alone is not sufficient, particularly in large complex (both from a physics and geometric domain) problems that must employ advanced equation discretization method that are solved on will exascale computing systems.

Methods that ensure that the discretizations used solve the mathematical models to a sufficient level of accuracy are applied. Since we are applying discretization methods to the problem because we do not know, and cannot solve for, its exact solution, ensuring that the discretization errors are small enough is a difficult problem (both mathematically and computationally).

In the case of mesh-based discretization technologies, adaptive mesh control techniques methods based on a-posteriori estimates to the local discretization errors provide the best option for ensuring domain scientists that the mathematical model underlying the simulations they perform are solves to a sufficient degree of accuracy.

This milestone report presents the recent advances to CEED adaptive mesh control technologies and load balancing procedures as needed for parallel execution. In addition, we report on other project wide activities including: the first MFEM release with GPU support, work with ECP applications, libCEED performance optimization, work on accelerator-oriented solvers, general interpolation, 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-780300.

REFERENCES

[1] Petra-m (physics equation translator for mfem). `http://piscope.psfc.mit.edu/index.php/Petra-M`. Accessed: 2019-06-21.

[2] Simmetrix inc. - mesh generation, geometry access. `http://www.simmetrix.com/`. Accessed: 2019-06-21.

[3] Turbulence modeling resource. `https://turbmodels.larc.nasa.gov/naca0012_val_w06.html`. Langley Research Center.

[4] Frédéric Alauzet. Size gradation control of anisotropic meshes. *Finite Elements in Analysis and Design*, 46(1-2):181–202, 2010.

[5] R.E. Bank. A domain decomposition solver for a parallel adaptive meshing paradigm. In *Proc. 16th Int. Symp. on Domain Decomposition Methods for Part. Diff. Eqs.*, pages 3–14. Springer-Verlag, 2006.

[6] Houman Borouchaki and Paul Louis George. *Meshing, Geometric Modeling and Numerical Simulation 1: Form Functions, Triangulations and Geometric Modeling*. John Wiley & Sons, 2017.

[7] P. Brubeck and P. Fischer. Approximate fast diagonalization for the spectral element method. *to be submitted*, 2019.

[8] P. Brubeck, K. Kaneko, Y.H. Lan, L. Lu, P. Fischer, and M. Min. Schwarz preconditioning for spectral element simulations of steady incompressible navier-stokes problems. *to be submitted*, 2019.

[9] Xiao-Chuan Cai and Marcus Sarkis. A restricted additive Schwarz preconditioner for general sparse linear systems. *SIAM J. Sci. Comput*, 21:792–797, 1999.

[10] Jakub Cerveny, Veselin Dobrev, and Tzanio Kolev. Non-conforming mesh refinement for high-order finite elements. *SIAM Journal on Scientific Computing*, 2019. to appear.

[11] Qi Chen and Ivo Babuška. Approximate optimal points for polynomial interpolation of real functions in an interval and in a triangle. *Computer Methods in Applied Mechanics and Engineering*, 128(3-4):405–417, 1995.

[12] Jack Choquette. Volta: Programmability and Performance. `https://www.hotchips.org/`, 2017.

[13] Gerrett Diamond, Cameron W. Smith, Eisung Yoon, and Mark S. Shephard. Dynamic load balancing of plasma and flow simulations. In *Proceedings of the 9th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*, ScalA '18, pages 73–80, New York, NY, USA, November 2018. ACM.

[14] Gerald Farin. *Curves and surfaces for computer-aided geometric design: a practical guide*. Elsevier, 2014.

[15] Rémi Feuillet, Adrien Loseille, David Marcum, and Frédéric Alauzet. Connectivity-change moving mesh methods for high-order meshes: Toward closed advancing-layer high-order boundary layer mesh generation. In *2018 Fluid Dynamics Conference*, page 4167, 2018.

[16] G. C. Fox, M. A. Johnson, G. A. Lyzenga, S. W. Otto, J. K. Salmon, and D. W. Walker. *Solving Problems on Concurrent Processors*. Prentice-Hall, Englewood Cliffs, NJ, 1988.

[17] Antonio Gómez-Iglesias, Feng Chen, Lei Huang, Hang Liu, Si Liu, and Carlos Rosales. Benchmarking the Intel Xeon Platinum 8160 Processor. Technical Report TR-17-01, Texas Advanced Computing Center, The University of Texas at Austin, 2017.

[18] Daniel A. Ibanez, E. Seegyoung Seol, Cameron W. Smith, and Mark S. Shephard. Pumi: Parallel unstructured mesh infrastructure. *ACM Transactions on Mathematical Software (TOMS)*, 42(3):17, 2016.

[19] George Karypis and Vipin Kumar. Parallel multilevel series k-way partitioning scheme for irregular graphs. *Siam Rev.*, 41(2):278–300, June 1999.

[20] X. Li, M. S. Shephard, and M. W. Beall. 3d anisotropic mesh adaptation by mesh modifications. *Comp. Meth. Appl. Mech. Engng*, 194(48-49):4915–4950, 2005.

[21] Peter Lindstrom. The minimum edge product linear ordering problem. Technical report, 08 2011.

[22] Qiukai Lu, Mark S. Shephard, Saurabh Tendulkar, and Mark W. Beall. Parallel mesh adaptation for high-order finite element methods with curved element geometry. *Engineering with Computers*, 30(2):271–286, 2014.

[23] Xiao-Juan Luo, Mark S. Shephard, Lie-Quan Lee, Lixin Ge, and Cho Ng. Moving curved mesh adaptation for higher-order finite element simulations. *Engineering with Computers*, 27(1):41–50, 2011.

[24] Xiao-Juan Luo, Mark S Shephard, Robert M Obara, Rocco Nastasia, and Mark W Beall. Automatic p-version mesh generation for curved domains. *Engineering with Computers*, 20(3):273–285, 2004.

[25] Xiao-Juan Luo, Mark S Shephard, Lu-Zhong Yin, Robert M OBara, Rocco Nastasi, and Mark W Beall. Construction of near optimal meshes for 3d curved domains with thin sections and singularities for p-version method. *Engineering with Computers*, 26(3):215–229, 2010.

[26] Javier Martinez, Yu-Hsiang Lan, Elia Merzari, and Misun Min. On the use of les-based turbulent thermal-stress models for rod bundle simulations. *Internaltional Journal of Heat and Mass Transfer*, submitted, 2019.

[27] W. Mitchell and T. Manteuffel. Advances in implementation, theoretical motivation, and numerical results for the nested iteration with range decomposition algorithm. *Num. Lin. Alg. with Applications*, 2018.

[28] PUMI: Parallel unstructured mesh infrastructure, 2016. http://www.scorec.rpi.edu/pumi.

[29] Cameron W. Smith, Brian Granzow, Gerrett Diamond, Dan A. Ibanez, Onkar Sahni, Kenneth E. Jansen, and Mark S. Shephard. In-memory integration of existing software components for parallel adaptive unstructured mesh workflows. *Concurrency and Computation: Practice and Experience*, pages 1–19, 2017 accepted.

[30] A. tomboulides, S.M. Aithal, P.F. Fischer, E. Merzari, A.V. Obabko, and D.R. Shaver. *A novel numerical treatment of the near-wall regions in the k-ω class of RANS models*, volume 72. 2018.

[31] Thomas Toulorge, Christophe Geuzaine, Jean-François Remacle, and Jonathan Lambrechts. Robust untangling of curvilinear meshes. *Journal of Computational Physics*, 254:8–26, 2013.