Sandia National Laboratories

Exceptional service in the national interest

Developing a GPU-enabled DPG Toolkit

Experiences with *Intrepid2*

J. Plews, G. Bunting, J. Rouse, C. Dohrmann

Trilinos User Group (TUG) Meeting

2021 December 01

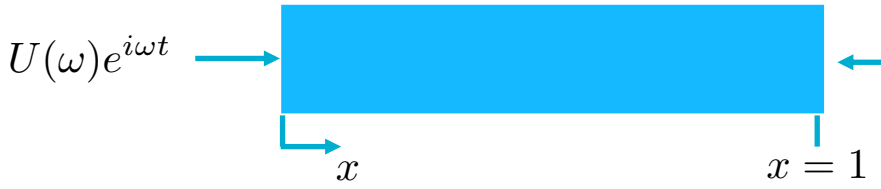
SAND2021-15074 C

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia LLC, a wholly owned subsidiary of Honeywell International Inc. for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

U.S. DEPARTMENT OF ENERGY

NNSA

Motivation: High-frequency wave propagation



$$\frac{\partial^2 p(x)}{\partial x^2} + k^2 p(x) = 0$$

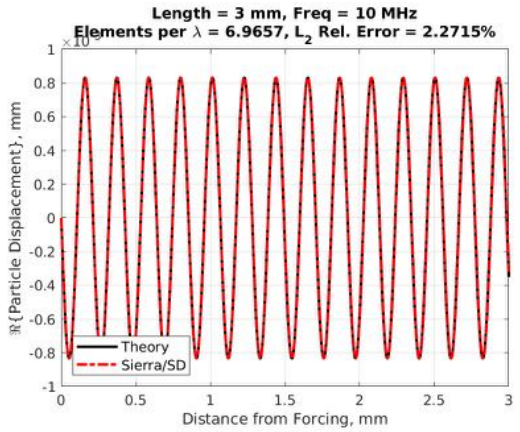
Helmholtz equation

Wavenumber: $k = \frac{\omega}{c}$

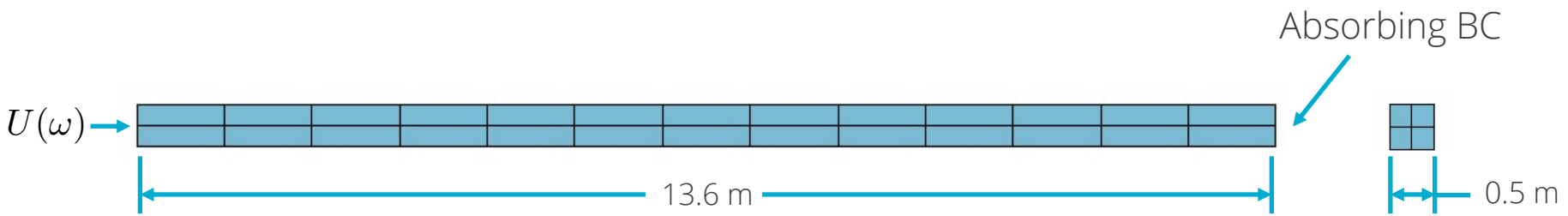
$$\left.\frac{\partial p}{\partial x}\right|_{x=0} = -i\omega\rho_o U$$

$$\left.\frac{\partial p}{\partial x}\right|_{x=1} = -ikp(1)$$

Boundary conditions

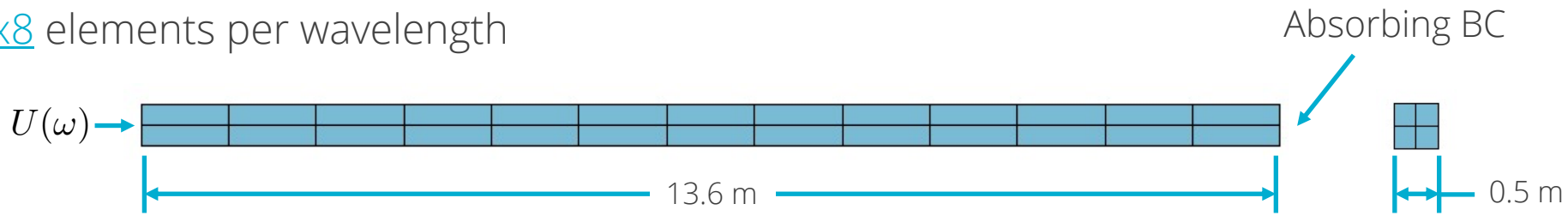


Example: Semi-infinite duct



Motivation: High-frequency wave propagation

- Excitation frequency: 20 Hz
- 16 Hex8 elements per wavelength

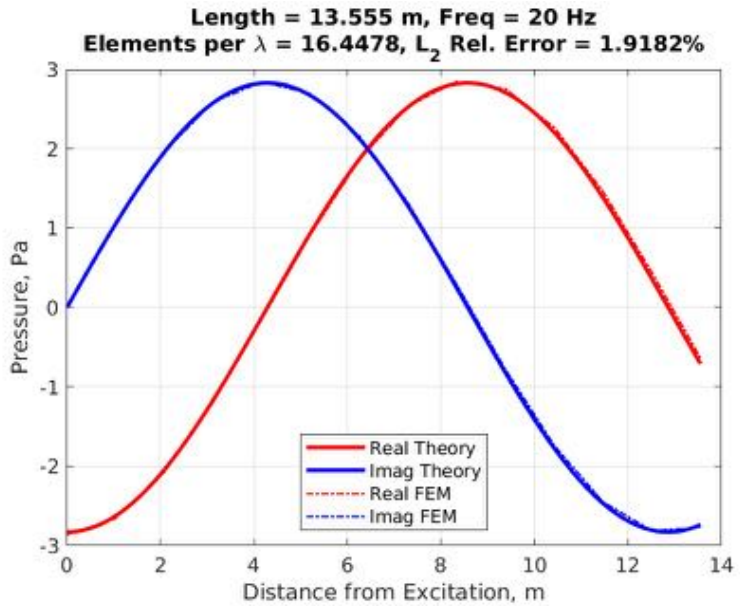


- Analytical solution: $p(z) = \rho_o c U(\omega) e^{-ikz}$

L_2 relative error = 1.9 %

$$\tilde{e}_2 = \frac{||p - p_h||_2}{||p||_2}$$

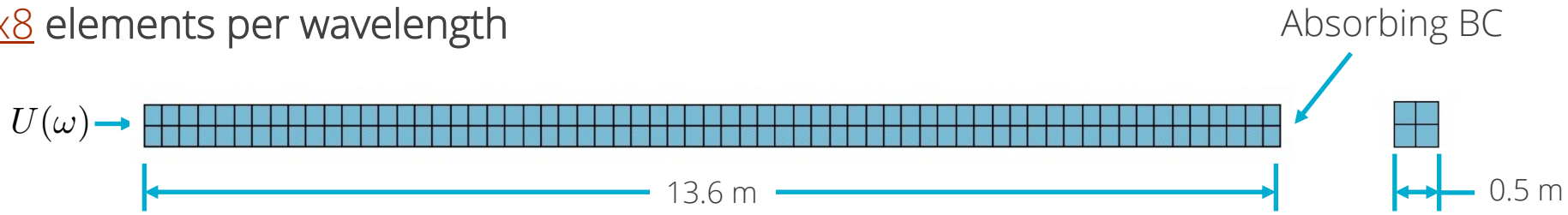
$$||u||_2 = \left[\int_0^L |u(z)|^2 dz \right]^{1/2}$$



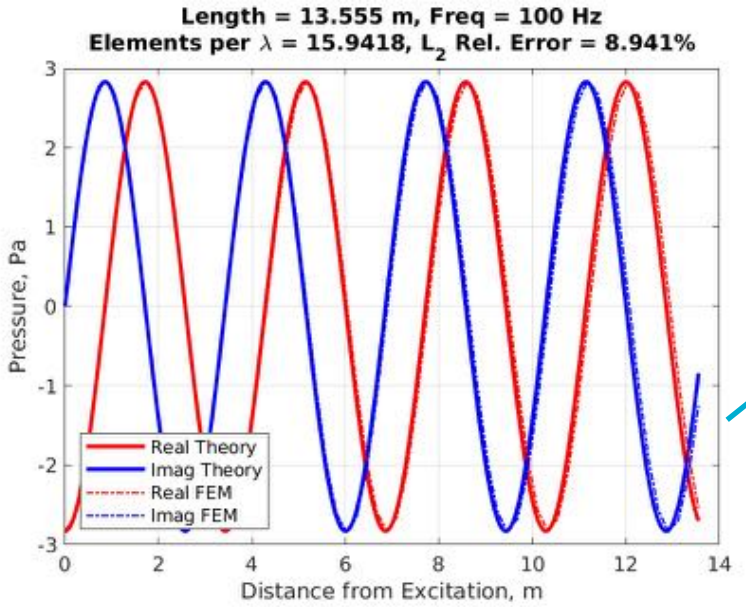
Percent error at end = 2.9 %

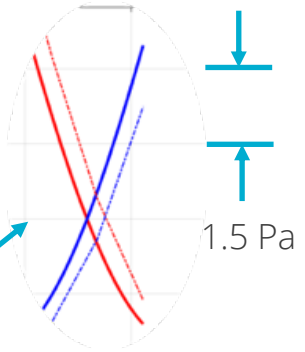
Motivation: High-frequency wave propagation

- Excitation frequency: 100 Hz
- 16 Hex8 elements per wavelength



L_2 relative error = 8.9 %





Error at end = 15.5 %

Same number of elements per wavelength as 20 Hz case, 4.7x larger L_2 error!

Motivation: Address pollution in high-frequency problems

- Relative error for classical Galerkin finite element solution to the Helmholtz equation:

Element size: *h*
Element order: *p*
Structure length: *L*

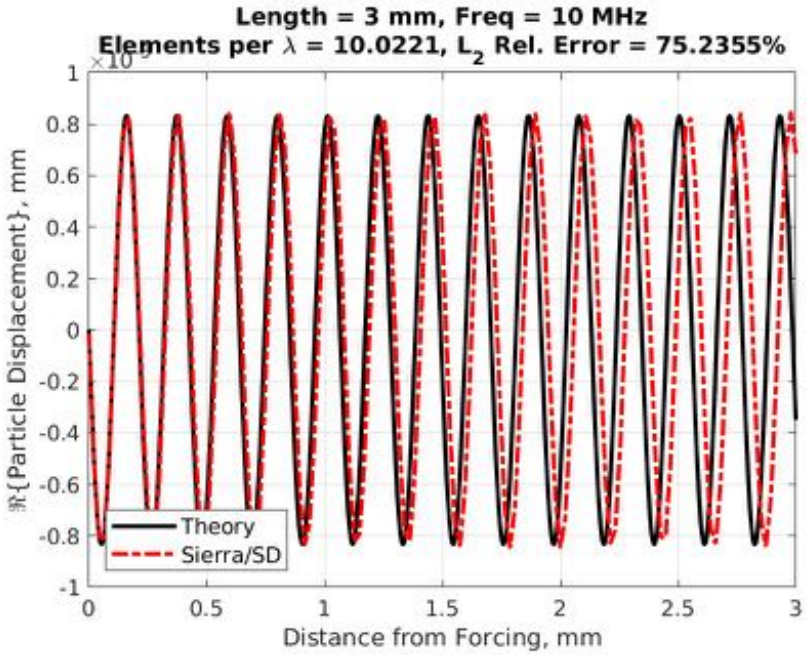
$$\frac{|u-u_h|_1}{|u|_1} \leq C_1\left(\frac{kh}{p}\right)^p + C_2 Lk\left(\frac{kh}{p}\right)^{2p}, \quad kh < 1$$

Interpolation Error

'Pollution Effect'

The issue for high frequencies

- Interpolation error: how well the shape functions approximate the solution
- *'Pollution effect'*: the accumulation of interpolation error over the length of the structure
- For constant relative error the second term must remain constant with increasing *L*

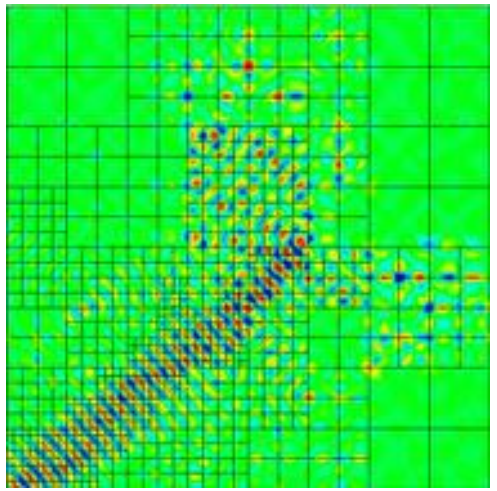


Motivation: Address pollution in high-frequency problems

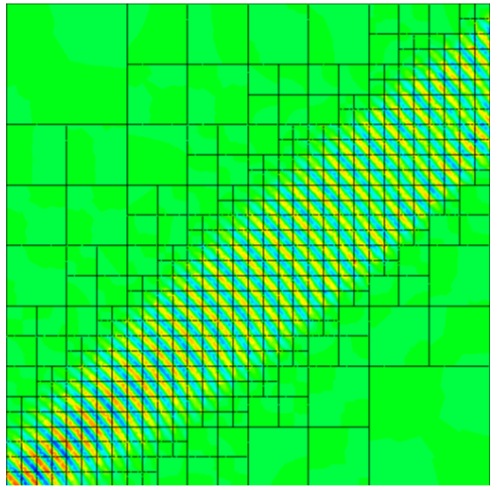
- Brute force: Domain decomposition and massively parallel FE
  - Pushed as far as possible with Galerkin FE and traditional element formulations
  - Sierra Mechanics implementation is the state of the art
- Higher-order FE methods
  - Going beyond quadratic approximations with *p*-elements drastically improves efficiency
  - Enriched methods
- Discontinuous Petrov-Galerkin
  - Developed by Prof. Demkowicz at UT Austin
- Others in the literature …

Discontinuous Petrov-Galerkin (DPG) discretizations

- Trial functions are continuous, test functions are discontinuous
  - Compute optimal test functions on the fly
  - Built-in error indicator
- Ultraweak formulation
  - Positive definite matrices
- Pollution is diffusive rather than dispersive
  - Pollution free in 1D
- Unconditionally stable
  - Allows for *hp*-adaptivity



Galerkin FEM



DPG

L. Demkowicz and J. Gopalakrishnan. *ICES REPORT 15-20, The University of Texas at Austin*, 2015.

J. Zitelli and I. Muga and L. Demkowicz and J. Gopalakrishnan and D. Pardo and V. M. Calo. *Journal of Computational Physics*, 230(7): 2406-2432, 2011.

S. Petrides. *PhD Dissertation, The University of Texas at Austin*, 2019.

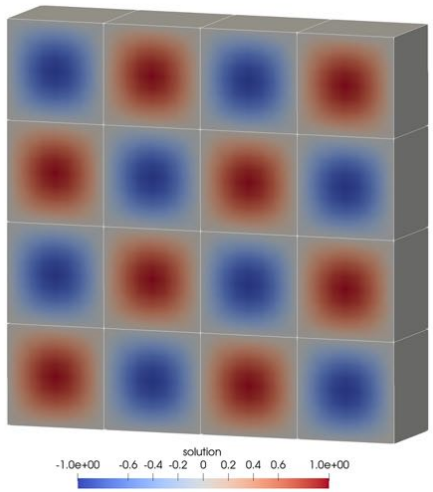
`mini_dpg`: A Trilinos-based toolkit for FE and DPG discretizations

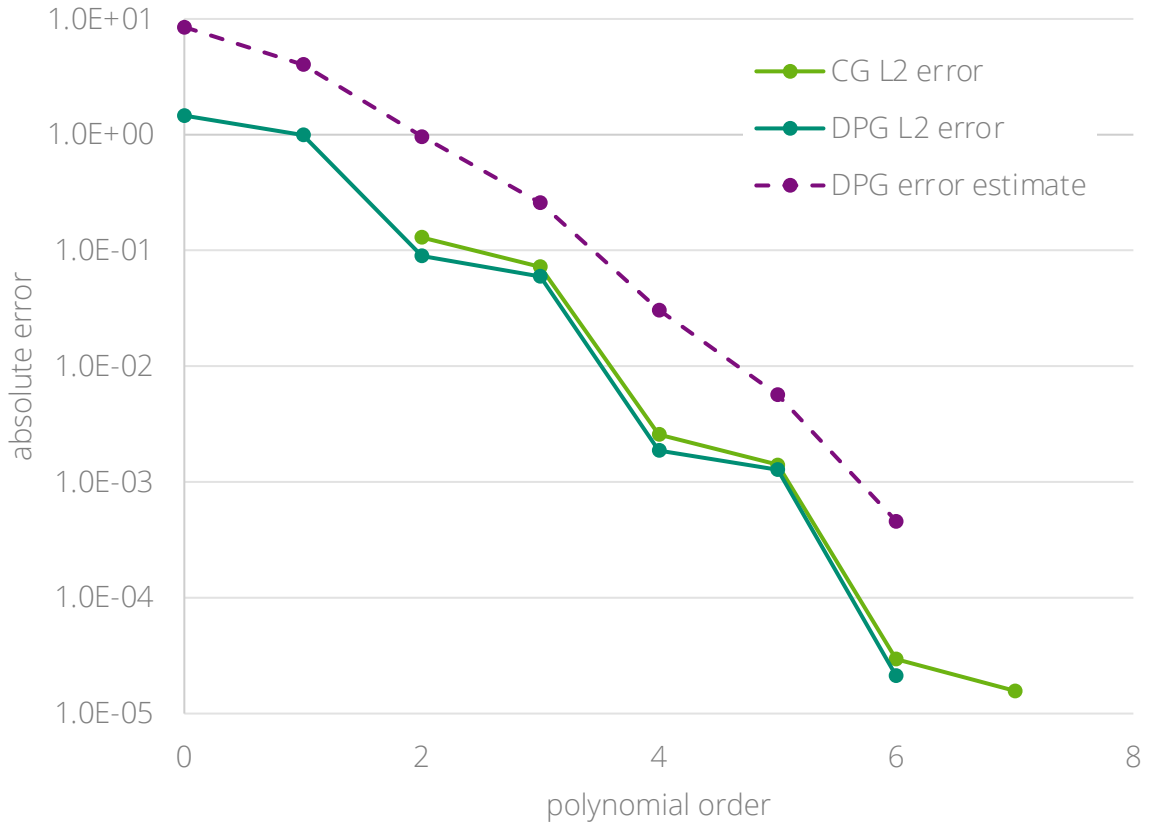
- `mini_dpg` mimics Sierra apps like Structural Dynamics and Solid Mechanics
  - Three-dimensional, MPI-parallel, heterogeneous (GPU-accelerated)
  - ...but with advanced discretizations in mind
- Intrepid2: finite element spaces for interpolation and integration
  - High-order approximations: nodal (Lagrange) and *p*-hierarchical bases
  - *H-div*, *H-grad*, L^2 polynomial spaces
- STK mesh and IO
  - Parallel data structure for large FE meshes
  - Cell topologies, computing and defining field data on nodes, edges, faces, and cells
- Kokkos parallelism, batch dense linear algebra, data management
- Tpetra, Amesos2, and Belos sparse, parallel linear solves
  - Also, *p*-preconditioning strategies, work in progress by Clark Dohrmann

Numerical results: Verification

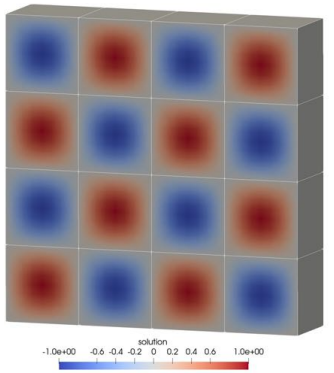
- p-convergence of DPG vs. CG (Galerkin) formulation of Poisson's equation
  - 4x4x1 hex grid
- Method of manufactured solutions for testing and verification:

$$\nabla^2 \phi = f$$
$$\phi = \sin(\pi x) sin(\pi y) \sin(\pi z)$$
$$\rightarrow f = -3\pi^2 \sin(\pi x) sin(\pi y) \sin(\pi z)$$



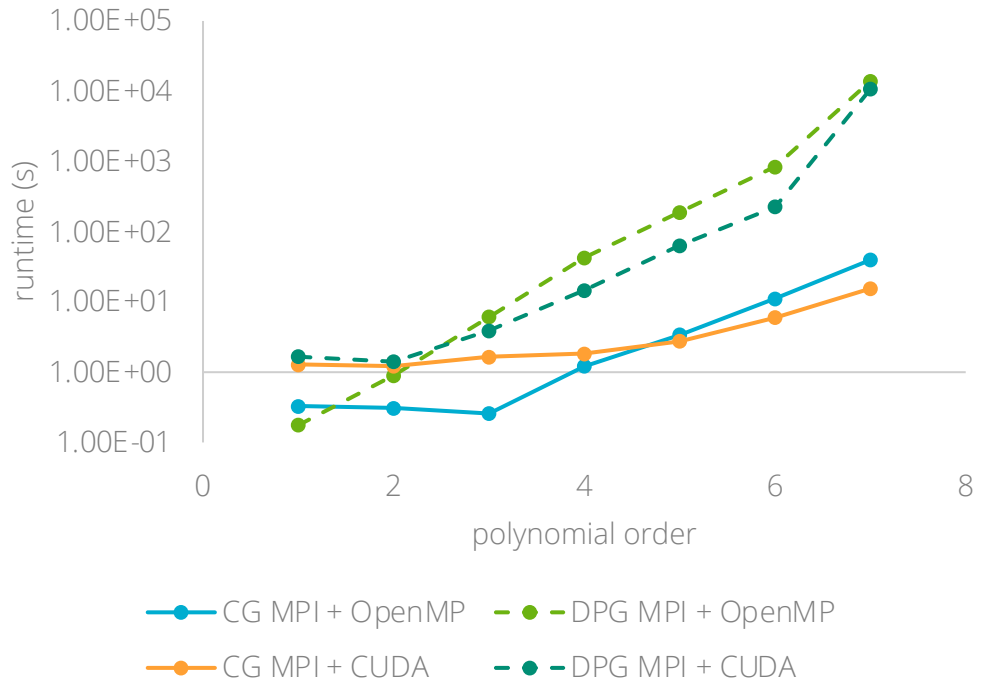


Numerical results: Verification problem performance

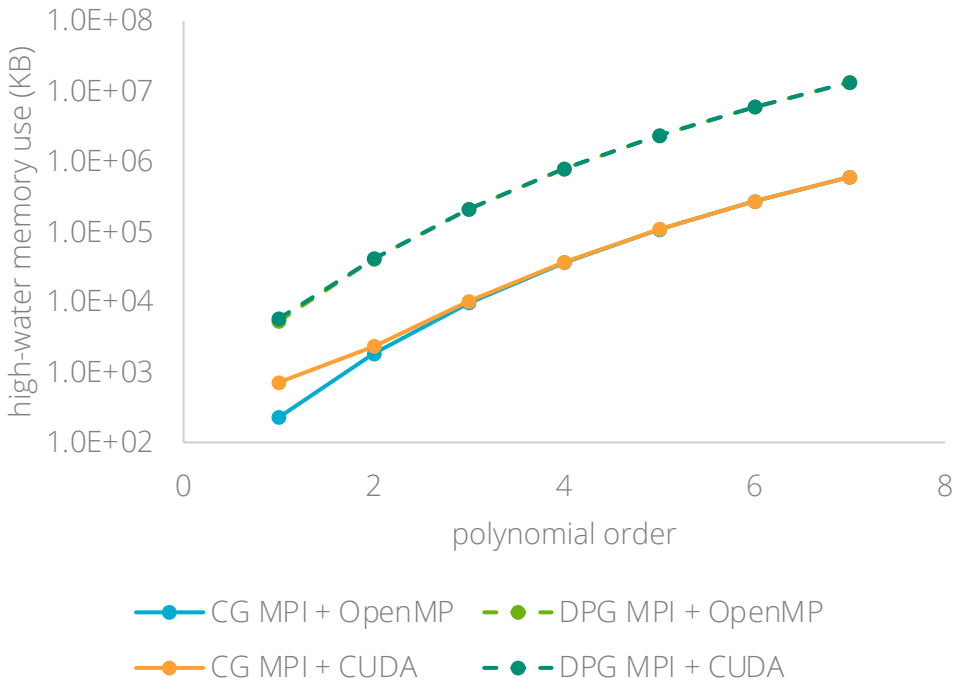


- GPU acceleration has benefits (even at low *p*-orders for DPG)
- Memory usage in DPG is a concern

Timing



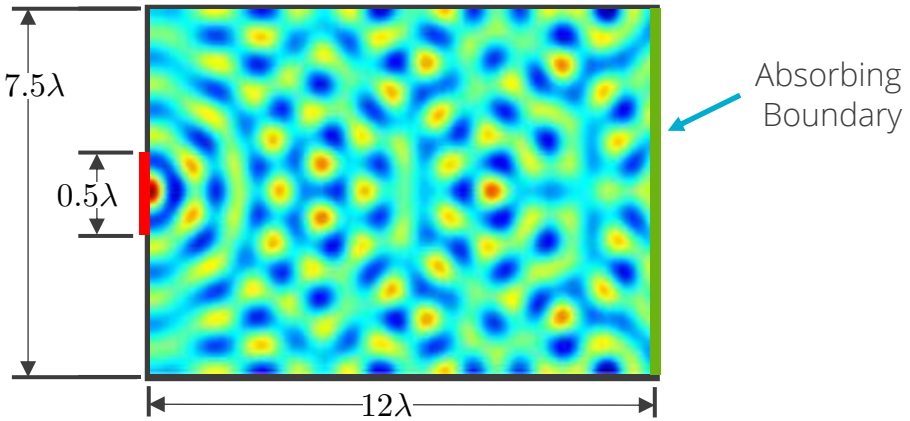
Memory



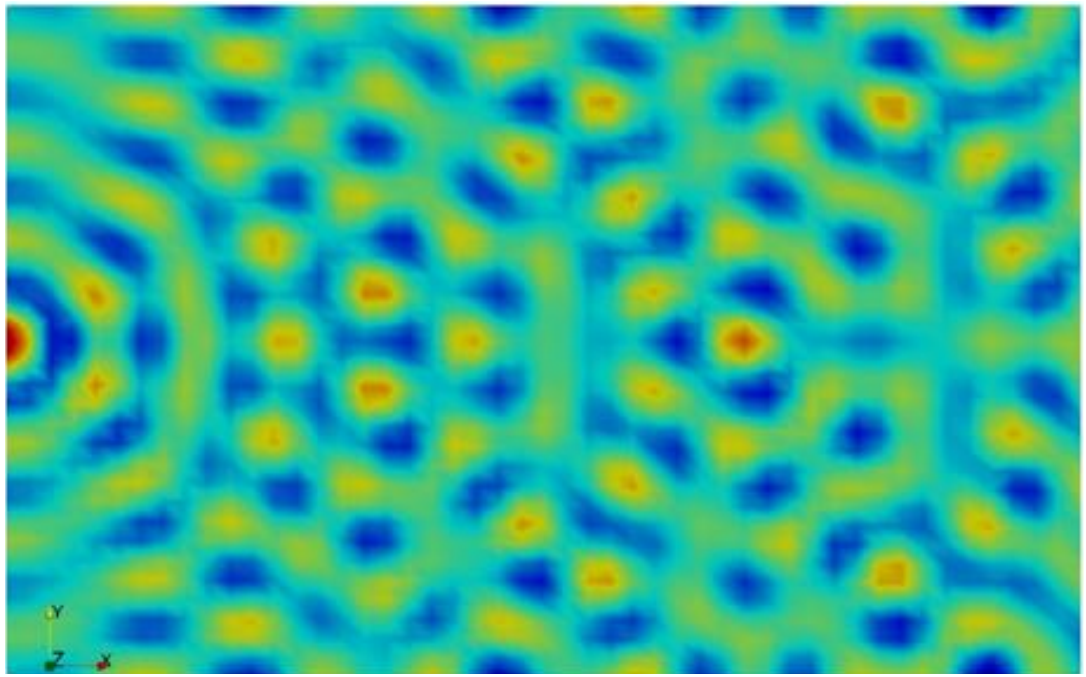
MPI + OpenMP: 2 MPI ranks x 4 threads per rank
MPI + CUDA: 2 MPI ranks + 1 Nvidia Volta GPU

Numerical results: High-order adaptive visualization

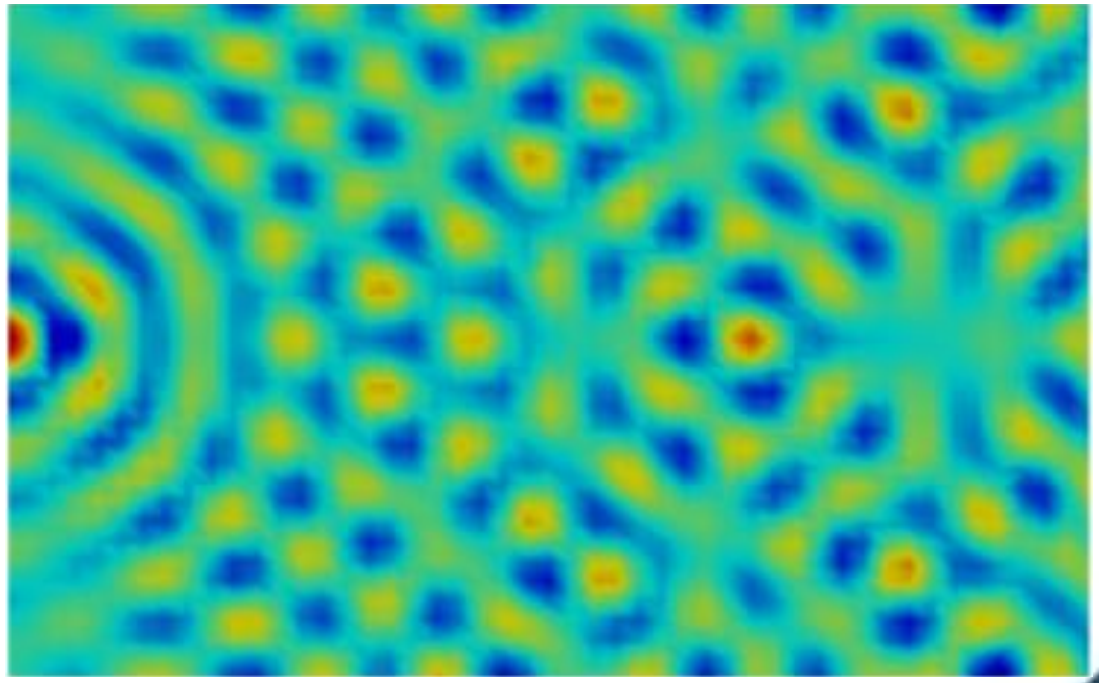
- High-order visualization/output
- Strategy: *h*-adapt "viz" mesh and interpolate to points based on desired error
- Example: 2D acoustic duct (Helmholtz), 1 element per wavelength



p=6 CGFE solution (Re)



p=7 DPG solution (Re)



Learning & using Intrepid2

- Discretizing Poisson's equation

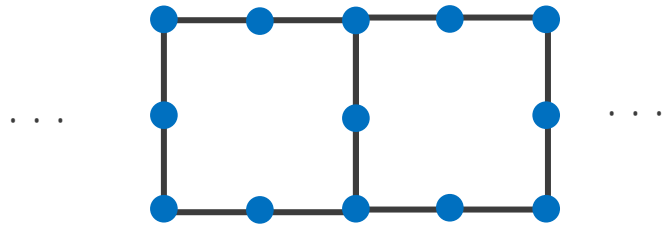
$$\nabla^2 \phi = f$$

$$\vec{\psi} = \vec{\nabla}\phi$$

Traditional Galerkin Weak Form:

$$-\int_\Omega \vec{\nabla}\phi \cdot \vec{\nabla} v \, d\Omega + \int_\Gamma v(\vec{\nabla}\phi \cdot \hat{n}) \, d\Gamma = \int_\Omega f v \, d\Omega$$

Expressed for entire domain

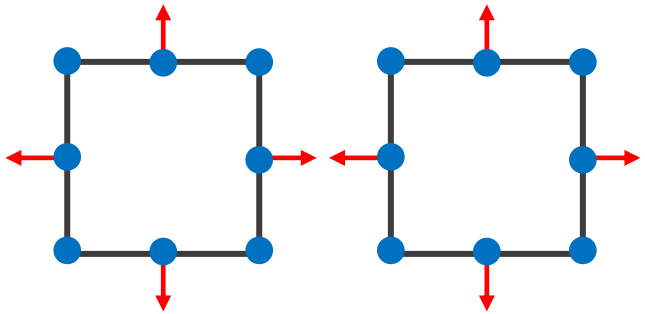


DPG Ultraweak Form:

$$-\int_{\Omega_k} \phi(\vec{\nabla} \cdot \vec{q}) \, d\Omega_k - \int_{\Omega_k} \vec{\psi} \cdot \vec{q} \, d\Omega_k +$$

$$\int_{\Gamma_k} \widehat{\phi}(\vec{q} \cdot \hat{n}) \, d\Gamma_k + \int_{\Gamma_k} \widehat{\psi}_n v \, d\Gamma_k = \int_{\Omega_k} f v \, d\Omega_k$$

Trace — Flux — For each element *k*



Learning & using Intrepid2

- Example: Discretizing Poisson's equation
  - Intrepid2-based assembly of PDE terms is simple for continuous Galerkin…

```
intrepid2_basis::Basis hex = create_p_hierarchical_hex_basis("hierarchical", "hgrad", 1);

auto fs = intrepid2_basis::FunctionSpace::create(iface, hex, hexNodes, hexNodeGids);

auto transformDShape = fs->compute_transformed_derivatives();

// stiffness
fs->compute_weighted_integral(stiffness, transformDShape, transformDShape);
```

Learning & using Intrepid2

- Example: Discretizing Poisson's equation
  - DPG discretizations introduce more complexity and bookkeeping
  - Introduce collections of function spaces and bases

Multiple bases

Function spaces corresponding to trace and element spaces

```
BasisCollection b;
short pInteg = 2 * porderTest;
b.add_basis(Basis("hex8", "dg_hierarchical", "hgrad", porderTest, pInteg), false);
b.add_basis(Basis("hex8", "dg_hierarchical", "hdiv", porderTest, pInteg), false);
b.add_basis(Basis("hex8", "hierarchical", "l2", porder - 1, pInteg), true);
b.add_basis(Basis("hex8", "hierarchical", "hdiv", porder, pInteg), false);
b.add_basis(Basis("hex8", "hierarchical", "hgrad", porder, pInteg), false);

FunctionSpaceCollection fc(cellCoords, cellVertexGids, bases);
fc.add_function_space_instance("v", "dg_hierarchical", "hgrad", 1);
fc.add_function_space_instance("q", "dg_hierarchical", "hdiv", 1);
fc.add_function_space_instance("phi", "hierarchical", "l2", 1);
fc.add_function_space_instance("psi", "hierarchical", "l2", 3);
fc.add_trace_function_space_instance("phiHat", "hierarchical", "hgrad", 1);
fc.add_trace_function_space_instance("psiHatN", "hierarchical", "hdiv", 1);
```

Learning & using Intrepid2

- Example: Discretizing Poisson's equation
  - Intrepid2-based implementation exploits integrals over sub-matrices to accommodate diverse range of PDE terms

Element contributions

Side contributions

```
// B
auto stiffPhiQ = Kokkos::subview(stiffness, Kokkos::ALL(), phiDofs, qDofs);
auto stiffPsiQ = Kokkos::subview(stiffness, Kokkos::ALL(), psiDofs, qDofs);
auto stiffPsiV = Kokkos::subview(stiffness, Kokkos::ALL(), psiDofs, vDofs);

// Bhat
auto stiffPhiHatQ = Kokkos::subview(stiffness, Kokkos::ALL(), phiHatDofs, qDofs);
auto stiffPsiHatNV = Kokkos::subview(stiffness, Kokkos::ALL(), psiHatNDofs, vDofs);

...

// B
fsHvol->compute_weighted_integral(stiffPhiQ, transformShapeHvol, transformDerivHdiv, -1.0);
fsHvol->compute_weighted_integral(stiffPsiQ, transformShapeHvolVec, transformShapeHdiv, -1.0);
fsHvol->compute_weighted_integral(stiffPsiV, transformShapeHvolVec, transformDerivHgrad, -1.0);

for (unsigned iface = 0; iface < hex.num_faces(); ++iface)
{
  ...

  auto qDotN = compute_values_dotted_with_normals(transformShapeHdivOnFace, hexFaceNormals);
  auto psiHatDotN = compute_values_dotted_with_normals(transformShapeFluxCgHdiv, hexFaceNormals);

  // Bhat
  fsTraceCgHgrad->compute_weighted_integral(stiffPhiHatQ, transformShapeTraceCgHgrad, qDotN);
  fsFluxCgHdiv->compute_weighted_integral(stiffPsiHatNV, psiHatDotN, transformShapeHgradOnFace);
}
```

Learning & using Intrepid2

- Example: Discretizing Poisson's equation
  - *Aside*: `mini_dpg` is also leveraging
    - Kokkos batched BLAS/LAPACK operations
    - and Intrepid2 projection utilities

```
// BTGinvl <- B^H * Ginv * l
// BTGinvB <- B^H * Ginv * B
dpgk::dense_kokkos_gemm(false, false, alpha, BT, get_ginv_b_subview(), beta, BTGinvB);
dpgk::dense_kokkos_gemv(false, alpha, BT, get_ginv_l_subview(), beta, BTGinvl);
```

Learning & using Intrepid2

- Room for improvement: Return values, handling array sizes, avoiding excess memory use/repeat allocations

Each allocates a `Kokkos::DynRankView<>` —convenient sizing

```
auto transformShapeFluxCgHdiv = fsFluxCgHdiv->compute_transformed_values();

auto hexFacePoints = fsFluxCgHdiv->get_integration_rule().points;
auto hexFaceNormals = hex.element_face_normals(elemCoords, hexFacePoints, iface);

// q dot n = transformShapeHDiv . cellSideNormals
auto psiHatDotN = compute_values_dotted_with_normals(transformShapeFluxCgHdiv, hexFaceNormals);
```

- Memory overhead due to arrays
- Strategies for array allocation and reuse?
- Opportunities for more expression templates?

Learning & using Intrepid2

- Room for improvement: Opportunities to exploit parallel asynchrony

Operations on non-overlapping arrays could be asynchronous and non-blocking

```
// B
fsHvol->compute_weighted_integral(stiffPhiQ, transformShapeHvol, transformDerivHdiv, -1.0);
fsHvol->compute_weighted_integral(stiffPsiQ, transformShapeHvolVec, transformShapeHdiv, -1.0);
fsHvol->compute_weighted_integral(stiffPsiV, transformShapeHvolVec, transformDerivHgrad, -1.0);

for (unsigned iface = 0; iface < hex.num_faces(); ++iface)
{
  ...

  // Bhat
  fsTraceCgHgrad->compute_weighted_integral(stiffPhiHatQ, transformShapeTraceCgHgrad, qDotN);
  fsFluxCgHdiv->compute_weighted_integral(stiffPsiHatNV, psiHatDotN, transformShapeHgradOnFace);
}
```

Takeaways and topics for future work

- `mini_dpg` is leveraging Trilinos packages to enable high-order discretizations on three-dimensional PDE discretizations of interest
  - More general than expected thanks to flexibility in Intrepid2
  - Elastodynamics implementation pending

- GPU acceleration in Intrepid2 and Kokkos batched linear algebra prove valuable at higher polynomial orders
  - Work to do to prove performance on realistic problems

- Memory high-water mark, usability, and performance
  - Matrix-free computations and efficient preconditioners
  - Avoid repeated memory allocations…especially on GPUs
  - Asynchronous, nonblocking kernels?