Exceptional service in the national interest

Sandia National Laboratories

July 12, 2022 SIAM Annual

A DPG Space-Time Vlasov-Poisson Discretization with Adaptive Mesh Refinement

Nathan V. Roberts, Stephen D. Bond, and Eric C. Cyr
nvrober@sandia.gov
Sandia National Laboratories

Outline

The full 3D3V Vlasov-Maxwell equations take the form:

$$\frac{\partial f}{\partial t} + \mathbf{v} \cdot \frac{\partial f}{\partial \mathbf{x}} + \frac{q}{m} \left(\mathbf{E} + \mathbf{v} \times \mathbf{B} \right) \cdot \frac{\partial f}{\partial \mathbf{v}} = 0 \tag{1}$$

$$\mathbf{J} = q \int \mathbf{v} \, f d^3 v \tag{2}$$

$$\nabla \cdot \mathbf{E} = \frac{\rho}{\epsilon_0} = \frac{q}{\epsilon_0} \int f d^3 v \tag{3}$$

$$\nabla \cdot \mathbf{B} = 0 \tag{4}$$

$$\nabla \times \mathbf{E} = -\frac{\partial \mathbf{B}}{\partial t} \tag{5}$$

$$\nabla \times \mathbf{B} = \mu_0 \left(\mathbf{J} + \epsilon_0 \frac{\partial \mathbf{E}}{\partial t} \right) \tag{6}$$

These are our ultimate target. For this talk, we simplify by using an electrostatic assumption, yielding the Vlasov-Poisson equations.

The 3D3V Vlasov-Poisson equations take the form:

$$\frac{\partial f}{\partial t} + \mathbf{v} \cdot \frac{\partial f}{\partial \mathbf{x}} + \frac{q}{m}\mathbf{E} \cdot \frac{\partial f}{\partial \mathbf{v}} = 0 \tag{7}$$

$$\nabla \cdot \mathbf{E} = \frac{q}{\epsilon_0} \int f d^3 v \tag{8}$$

$$\mathbf{E} + \nabla \phi = 0 \tag{9}$$

Here, we have introduced a potential ϕ such that $\mathbf{E} = -\nabla\phi$ (convenient for BCs). We simplify further by restricting to 1D1V:

$$\frac{\partial f}{\partial t} + v_x \frac{\partial f}{\partial x} + \frac{q}{m} E \cdot \frac{\partial f}{\partial v_x} = 0 \tag{10}$$

$$\frac{\partial E}{\partial x} = \frac{q}{\epsilon_0} \int f dv_x \tag{11}$$

$$E + \frac{\partial \phi}{\partial x} = 0 \tag{12}$$

High-Level Introduction to DPG

Suppose you have a bilinear form $b(\,\cdot\,,\,\cdot\,)$ with load $l(\,\cdot\,)$, and (group) trial variable $u \in U^h$, test $v \in V$:

$$b(u, v) = l(v)$$

U, V Hilbert; V endowed with inner product $(\,\cdot\,,\,\cdot\,)_V$.

High-Level Introduction to DPG

Suppose you have a bilinear form $b(\cdot,\cdot)$ with load $l(\cdot)$, and (group) trial variable $u \in U^h$, test $v \in V$:

$$b(u,v) = l(v)$$

U, V Hilbert; V endowed with inner product $(\cdot,\cdot)_V$.
For each trial basis function $e \in U^h$, we define $v_e^{\text{opt}} \in V$ by

$$(v_e^{\text{opt}}, w)_V = b(e,w) \quad \forall w \in V;$$

that, is $v_e^{\text{opt}} \in V$ is the *Riesz representative* of $b(e,\cdot)$. Using these *optimal test functions* as our test space, we immediately see that a stiffness matrix $K_{ij} = b(e_i, v_{e_j}^{\text{opt}})$ is symmetric (Hermitian) positive definite:

$$K_{ij} = b(e_i, v_{e_j}^{\text{opt}}) = \left(v_{e_i}^{\text{opt}}, v_{e_i}^{\text{opt}}\right)_V = \overline{\left(v_{e_j}^{\text{opt}}, v_{e_i}^{\text{opt}})_V\right)} = \overline{b(e_j, v_{e_i}^{\text{opt}})} = \overline{K_{ij}}.$$

High-Level Introduction to DPG

Using infinite-dimensional V is called the *ideal* DPG method; the *practical* DPG method breaks V element-wise and uses a high-order $V_h(K) \subset V(K)$.

- We express the polynomial order of the test space as $p + \Delta p$, where p is the (H^1) order of the trial space.

High-Level Introduction to DPG

Using infinite-dimensional V is called the *ideal* DPG method; the *practical* DPG method breaks V element-wise and uses a high-order $V_h(K) \subset V(K)$.

- We express the polynomial order of the test space as $p + \Delta p$, where p is the (H^1) order of the trial space.
- polynomial enrichment of the test space $\implies$ *inherently* high-order method.

High-Level Introduction to DPG

Using infinite-dimensional V is called the *ideal* DPG method; the *practical* DPG method breaks V element-wise and uses a high-order $V_h(K) \subset V(K)$.

- We express the polynomial order of the test space as $p + \Delta p$, where p is the (H^1) order of the trial space.
- polynomial enrichment of the test space $\implies$ *inherently* high-order method.
- We **solve (dense) element-local problems** to determine optimal test functions.

High-Level Introduction to DPG

Using infinite-dimensional V is called the *ideal* DPG method; the *practical* DPG method breaks V element-wise and uses a high-order $V_h(K) \subset V(K)$.

- We express the polynomial order of the test space as $p + \Delta p$, where p is the (H^1) order of the trial space.
- polynomial enrichment of the test space $\implies$ *inherently* high-order method.
- We **solve (dense) element-local problems** to determine optimal test functions.
- Method minimizes the error in an energy norm determined by test inner product (user choice).

High-Level Introduction to DPG

Using infinite-dimensional V is called the *ideal* DPG method; the *practical* DPG method breaks V element-wise and uses a high-order $V_h(K) \subset V(K)$.

- We express the polynomial order of the test space as $p + \Delta p$, where p is the (H^1) order of the trial space.
- polynomial enrichment of the test space $\implies$ *inherently* high-order method.
- We **solve (dense) element-local problems** to determine optimal test functions.
- Method minimizes the error in an energy norm determined by test inner product (user choice).
- Natural error indicator: Riesz representative of residual $b(u_h, \cdot) - l(\cdot) \implies$ can use to drive (robust) **AMR**.

Vision for DPG Vlasov Solver

The goal: **flexible**, **robust**, **accurate** plasma physics solver for regimes that PIC does not address well.

Our approach: DPG for Vlasov.

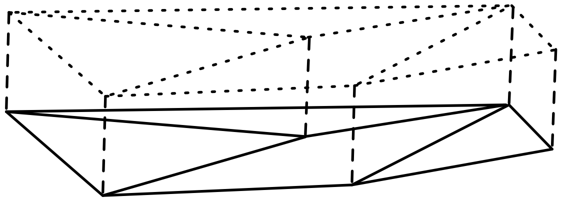
DPG has many attractive features:

- discrete stability is automatic
- almost total flexibility in solution basis (can go high-order)
- "minimum-residual method": solution error is minimized in an energy norm
- comes with a built-in error indicator: AMR is natural and robust

Vlasov in Camellia

Camellia is my Trilinos-based FEM library, with support for DPG + AMR.

- For Vlasov, we need hyper-dimensional meshes, up to 7D total.
- Key feature: allow **orthogonal extrusion** of any mesh in new dimensions.
  - Assume orthogonal: simplifies Jacobian computations, etc.
  - Do not assume uniform divisions: allow AMR in the new dimensions.



Challenges: Computational Cost and the Curse

The curse of dimensionality looms. We have three key mitigations:

1. Adaptive Mesh Refinement
   - Full support for isotropic h-adaptivity.
   - Anisotropic adaptivity: necessary for performance in high dimensions.
2. Underway: Hyperdimensional Serendipity bases[1]
3. Smart Assembly
   - Structure of Vlasov allows most terms to be integrated in lower dimensions, and multiplied by a pre-computed integral corresponding to remaining dimensions.
   - Not yet implemented.

[1]Serendipity basis support in Intrepid2; Trilinos master SHA1 22d0482, 7/7/22.

Temporal Discretization

Two basic approaches:

- time-marching
- space-time

We are pursuing **both** of these. Concerns for time-marching:

- Theory suggests we can accumulate error; in practice we may observe this when under-resolved.
- Very little theory for standard schemes beyond backward Euler; experiments for other PDEs suggest any implicit scheme is reasonable, though.
- For Vlasov, we have only implemented backward Euler so far.
- Best refinement strategy isn't as clear; may depend on the problem.

In favor of time-marching:

- High-order RK schemes can be pretty efficient/effective.
- Toolset (ParaView, etc.) is set up for time-marching.

Temporal Discretization

Concerns for space-time:

- Adds another (curséd) dimension to the mesh.
- Harder to visualize.

In favor of space-time:

- DPG theory supports it very well.
- Can do localized adaptivity in temporal dimension.
- Refinement strategy is clear(er).
- Can run parallel-in-time.
- Strategies, optimizations we develop for velocity dimensions are likely to carry over to time dimension as well: it's another orthogonal extrusion.

Space-Time Formulation: Vlasov

We may write the 1D1V Vlasov equation as:

$$\nabla_{xtv} \cdot \begin{bmatrix} v_x f \\ f \\ \frac{q}{m} E_x f \end{bmatrix} = 0.$$

Multiplying by test $w \in H^1$ and integrating by parts:

$$\langle \hat{t}_n, w \rangle - \left(\begin{bmatrix} v_x f \\ f \\ \frac{q}{m} E_x f \end{bmatrix}, \nabla_{xtv} w \right) = 0,$$

where formally

$$\hat{t}_n = \operatorname{tr} \left(\begin{bmatrix} v_x f \\ f \\ \frac{q}{m} E_x f \end{bmatrix} \cdot \begin{bmatrix} n_x \\ n_t \\ n_v \end{bmatrix} \right).$$

We use the graph norm on the test space.

Space-Time Formulation: Poisson

Our space-time Poisson Formulation:

$$\begin{aligned} \langle \hat{V}_E, \tau\, n_x \rangle - (V_E, \partial_x \tau) + (E_x, \tau) &= 0 \\ \langle \hat{E}_x, q\, n_x \rangle - (E_x, \partial_x q) &= \left(\frac{\rho}{\epsilon_0}, q \right). \end{aligned}$$

Note that the traces $\hat{V}_E, \hat{E}_x$ are only defined at the spatial interfaces (those for which $n_x \neq 0$). Note also that ρ is two-dimensional: it varies in time as well as space. The usual situation is that BCs are imposed on $\hat{V}_E$ at the left and right boundaries; for the cold diode, we impose $\hat{V}_E = 0$ at each.

We use the graph norm on the test space.

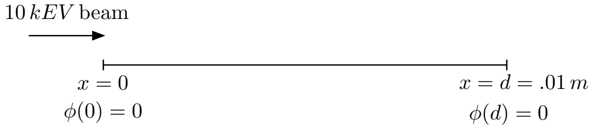
Solution Strategy: Fixed Point Iteration

We use a fixed-point iteration with a set maximum number of iterations:

- up to 15 fixed-point iterations per solve, with early exit if the relative norm of the update falls below a tolerance (10^{-6}).
- Linear solves performed with Geometric-Multigrid-preconditioned conjugate gradient solver, tolerance between 10^{-7} and 10^{-9}.

The Cold Diode Problem

In the cold diode problem, a beam of electrons is emitted across a 1D anode-cathode gap, with an applied voltage across the gap.



- We have an exact solution due to Jaffé.
- EMPIRE-PIC has very accurate results for this problem.
- Tom Smith provided me the Python scripts used in EMPIRE's analysis; I've adapted these.

The Cold Diode Problem and Vlasov

Some notes on our approach:

- We nondimensionalize for computations, such that $v^*_{\text{beam}} = 1$ and $t^*_{\text{final}} = 1$.
- We rescale on output for comparison to exact solution.
- Inflow BC: approximated with a Maxwellian with thermal velocity $\sigma = 0.025\, v_{\text{beam}}$.
- $\sigma > 0 \implies$ solving a slightly different problem; can expect some error due to that difference.
- Important to **resolve** the BC; we perform initial refinements to resolve to a given tolerance.

The Cold Diode Problem and Vlasov

For space-time, there is a *corner discontinuity* in the BCs: the initial condition disagrees with the injection BC at $x = 0$.

- Our approach *can* handle this, but it costs us in the test space degree.
- We therefore use a linear *temporal ramp* $w_{\text{ramp}}(t)$ to weight the inflow BC; $w_{\text{ramp}}(0) = 0, w_{\text{ramp}}(0.25) = 1$.

Space-Time Results: Uniform Refinement Studies

Table: Relative L^2 errors

f order	Mesh Size	E err.	ϕ err.	n_e err.	v_x err.
0	$4 \times 40 \times 40$	2.458E-01	2.228E-01	2.276E-02	2.386E-02
0	$8 \times 80 \times 80$	1.228E-01	1.133E-01	1.130E-02	1.198E-02
0	$16 \times 160 \times 160$	6.137E-02	5.690E-02	5.630E-03	5.998E-03
1	$4 \times 20 \times 40$	2.481E-03	2.505E-02	2.446E-03	2.200E-03
1	$8 \times 40 \times 80$	7.065E-04	6.266E-03	6.660E-04	6.212E-04
1	$16 \times 80 \times 160$	3.924E-04	1.605E-03	3.641E-04	3.399E-04
2	$4 \times 10 \times 40$	5.021E-04	4.206E-04	2.586E-03	6.109E-04
2	$8 \times 20 \times 80$	3.660E-04	3.673E-04	4.753E-04	3.365E-04
2	$16 \times 40 \times 160$	3.618E-04	3.635E-04	4.016E-04	3.138E-04
3	$4 \times 5 \times 40$	6.151E-03	2.189E-03	2.614E-02	3.178E-03
3	$8 \times 10 \times 80$	3.624E-04	3.632E-04	4.126E-04	3.133E-04
3	$16 \times 20 \times 160$	3.619E-04	3.637E-04	3.353E-04	3.126E-04

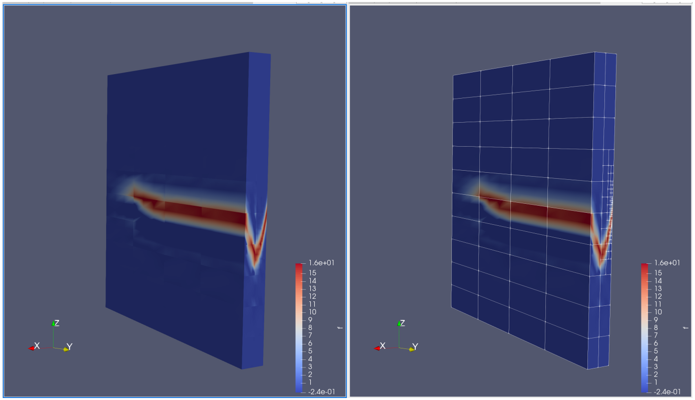
Uniform refinement study for space-time, for poly orders from 0 to 3. As with our finest time-marching solves, we see error of roughly 3×10^{-4} in each variable, due to the nonzero value for σ. Note that the second dimension is time; we use coarser discretizations in time for higher polynomial orders so that we have roughly the same number of temporal nodes as in the time-marching scheme.

Adaptive Space-Time Results

For this AMR run, we perform a set of initial refinements, driven by the error in the boundary condition, until that error is less than a specified tolerance in the relative L^2 norm on the boundary. In this run, we use the following setup:

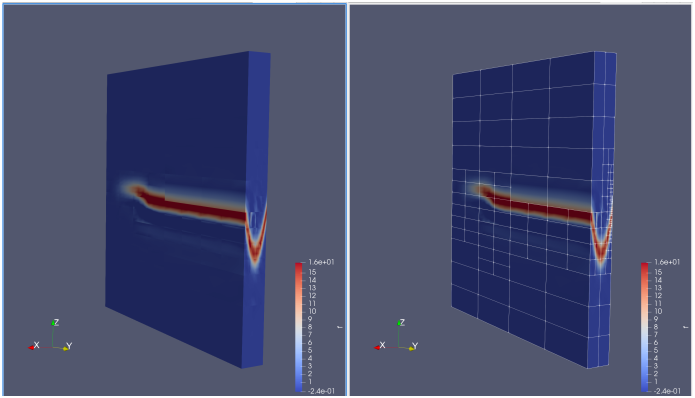
- coarse mesh: $2 \times 4 \times 10$ elements
- $\sigma = 0.025$
- BC tol: 10^{-5}
- quadratic field variables
- test space enrichment $\Delta p = 4$
- greedy refinement parameter $\theta = 0.2$

Adaptive Space-Time Results: Vlasov



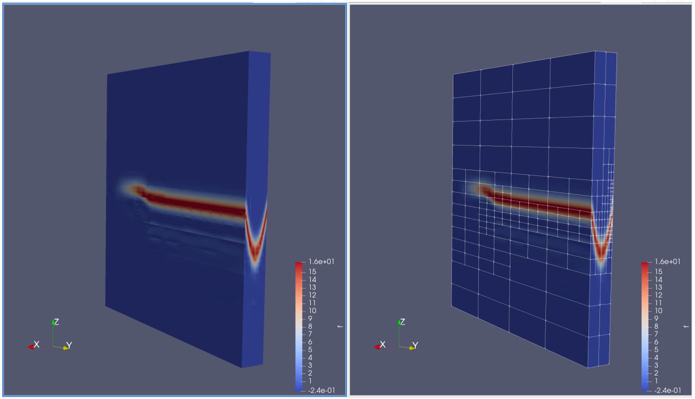
Vlasov solution for the cold diode problem, after 0 energy-error refinements. Time dimension is coming out of the screen; the left side is the spatial outflow.

Adaptive Space-Time Results: Vlasov



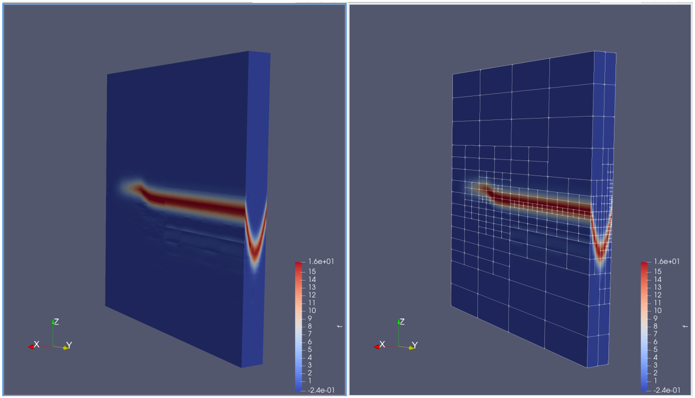
Vlasov solution for the the cold diode problem, after 1 energy-error refinement. Time dimension is coming out of the screen; the left side is the spatial outflow.

Adaptive Space-Time Results: Vlasov



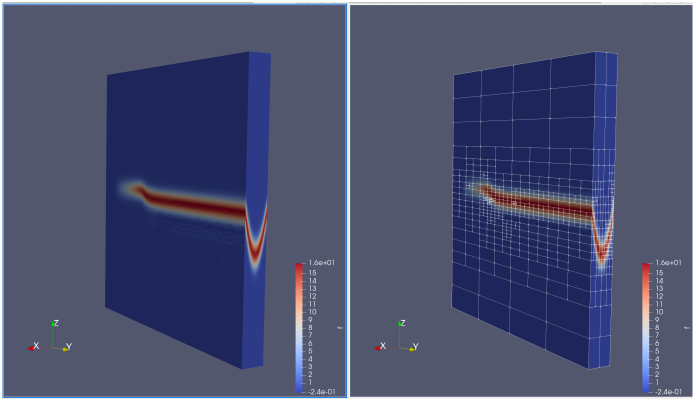
Vlasov solution for the cold diode problem, after 2 energy-error refinements. Time dimension is coming out of the screen; the left side is the spatial outflow.

Adaptive Space-Time Results: Vlasov



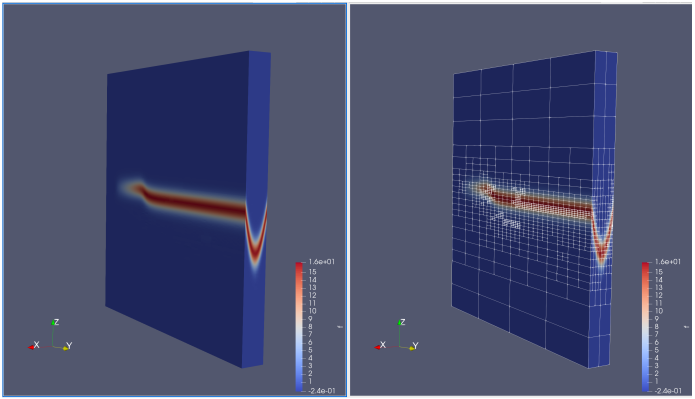
Vlasov solution for the cold diode problem, after 3 energy-error refinements. Time dimension is coming out of the screen; the left side is the spatial outflow.

Adaptive Space-Time Results: Vlasov



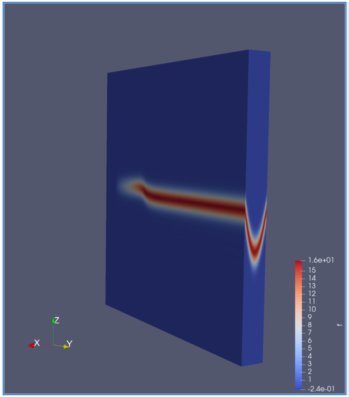
Vlasov solution for the cold diode problem, after 4 energy-error refinements. Time dimension is coming out of the screen; the left side is the spatial outflow.

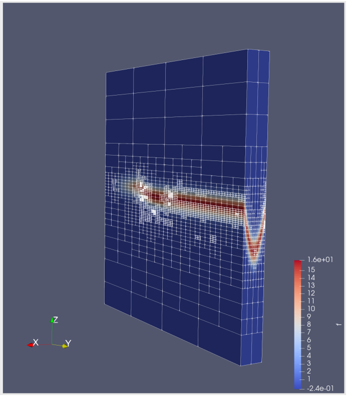
Adaptive Space-Time Results: Vlasov



Vlasov solution for the cold diode problem, after 5 energy-error refinements. Time dimension is coming out of the screen; the left side is the spatial outflow.

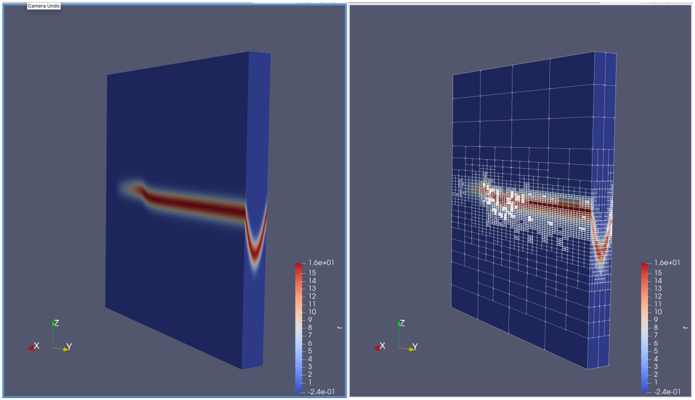
Adaptive Space-Time Results: Vlasov





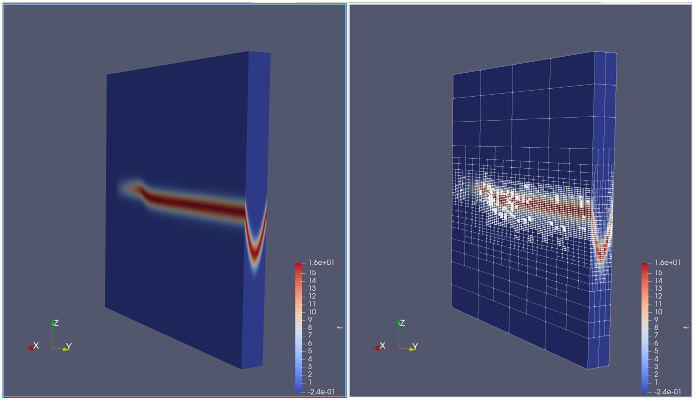
Vlasov solution for the cold diode problem, after 6 energy-error refinements. Time dimension is coming out of the screen; the left side is the spatial outflow.

Adaptive Space-Time Results: Vlasov



Vlasov solution for the cold diode problem, after 7 energy-error refinements. Time dimension is coming out of the screen; the left side is the spatial outflow.

Adaptive Space-Time Results: Vlasov



Vlasov solution for the cold diode problem, after 8 energy-error refinements. Time dimension is coming out of the screen; the left side is the spatial outflow.

Further Cost Mitigation: Serendipity Basis

Table: Number of dofs/element for full tensor H^1 basis.

p	1	2	3	4	5	6	7
Dim.							
2	4	9	16	25	36	49	64
3	8	27	64	125	216	343	512
4	16	81	256	625	1296	2401	4096
5	32	243	1024	3125	7776	16807	32768
6	64	729	4096	15625	46656	117649	262144
7	128	2187	16384	78125	279936	823543	2097152

Table: Number of dofs/element for Serendipity basis.

p	1	2	3	4	5	6	7
Dim.							
2	4	8	12	17	23	30	38
3	8	20	32	50	74	105	144
4	16	48	80	136	216	328	480
5	32	112	192	352	592	952	1472
6	64	256	448	880	1552	2624	4256
7	128	576	1024	2144	3936	6960	11776

Questions:

- Can we use Serendipity for test as well as trial?
- How well can we approximate optimal test functions with Serendipity basis?
- How well can we approximate (typical) solutions?

Further Cost Mitigation: Smart Assembly

We can take advantage of the structure of Vlasov to perform assembly more efficiently. The simplest terms in our formulation take the form

$$\begin{aligned}(C\phi_i, \psi_j)_K &= (C\phi^{\mathbf{x}}_{i_x}\phi^{\mathbf{v}}_{i_v}, \psi^{\mathbf{x}}_{j_x}\psi^{\mathbf{v}}_{j_v}) \\ &= \int_K C\phi^{\mathbf{x}}_{i_x}\phi^{\mathbf{v}}_{i_v}\psi^{\mathbf{x}}_{j_x}\psi^{\mathbf{v}}_{j_v}\,\partial\mathbf{x}\partial\mathbf{v}\end{aligned}$$

where ϕ_i is the trial function and ψ_j is the test function, and C some constant. Each velocity dimension is an orthogonal extrusion $\implies$ ref.-to-physical Jacobians diagonal, so we may write:

$$\left(\int_{K_x} C\phi^{\mathbf{x}}_{i_x}\psi^{\mathbf{x}}_{j_x}\,\partial\mathbf{x}\right)\left(\int_{K_v} \phi^{\mathbf{v}}_{i_v}\psi^{\mathbf{v}}_{j_v}\,\partial\mathbf{v}\right).$$

The velocity-space integral itself decomposes into a product of integrals along each velocity-space dimension; these integrals may be performed in reference space and multiplied by the cell measure in the corresponding velocity dimension to obtain a physical integral. Similar tricks can be performed for most terms in our formulation.

Conclusion

- In Intrepid2, we have some pretty good building blocks for structure-dependent algorithms.
- In Vlasov, we have a highly structured problem — especially so for a space-time 3D3V discretization!
- Still to do:
  - Use Serendipity bases (recently added to Intrepid2)
  - Smart Assembly
  - Anisotropic adaptivity: vital for higher dimensions
- We *do not* have a robust, local *anisotropic* error indicator. An area for future research!

Time-Marching Results: Uniform Refinement Study

Table: Quadratic f, Time-Marching, Relative L^2 errors

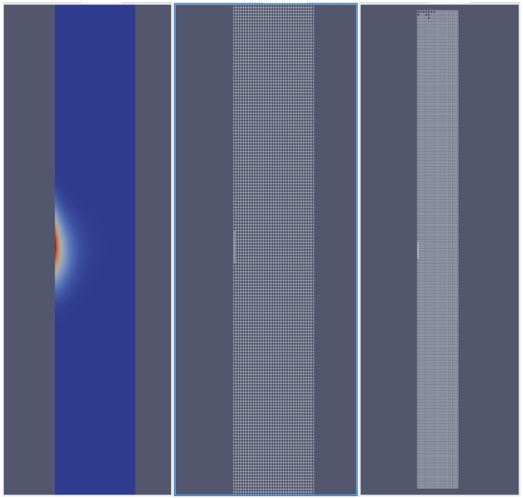
Mesh Size	Num Time Steps	E err.	ϕ err.	n_e err.	v_x err.
4x40	20	3.951E-04	3.715E-04	1.206E-03	5.041E-04
8x80	25	3.620E-04	3.638E-04	3.361E-04	3.133E-04
16x160	50	3.616E-04	3.634E-04	3.350E-04	3.126E-04
32x320	100	3.322E-04	3.333E-04	3.117E-04	3.069E-04

Adaptive Solve

To give just one adaptive solve example:

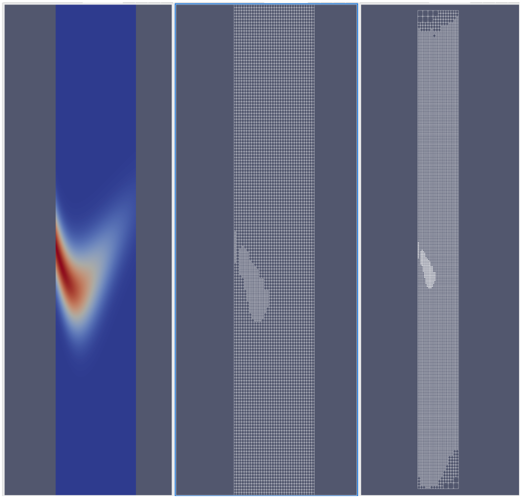
- start with a fine mesh identical to the finest fixed-size quadratic solution, 32×320 elements
- each time step, refine according to energy error, and unrefine an equal number of elements
- test space enrichment $\Delta \mathrm{p} = 5$.

Adaptive Solve



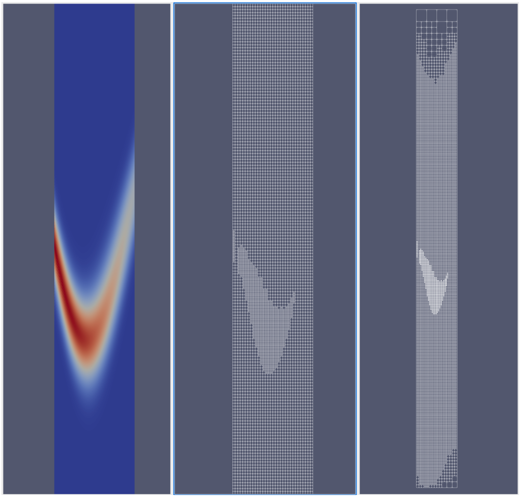
Time step 1.

Adaptive Solve



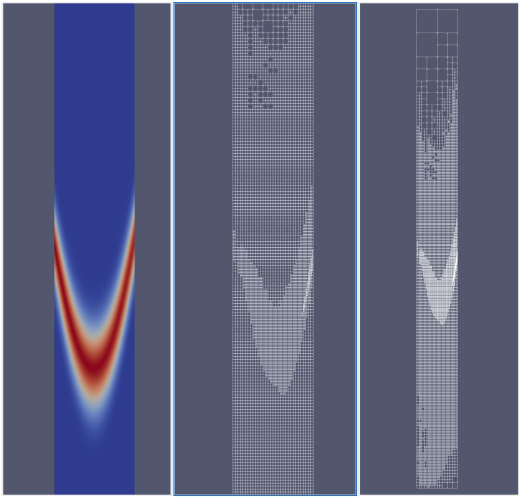
Time step 5.

Adaptive Solve



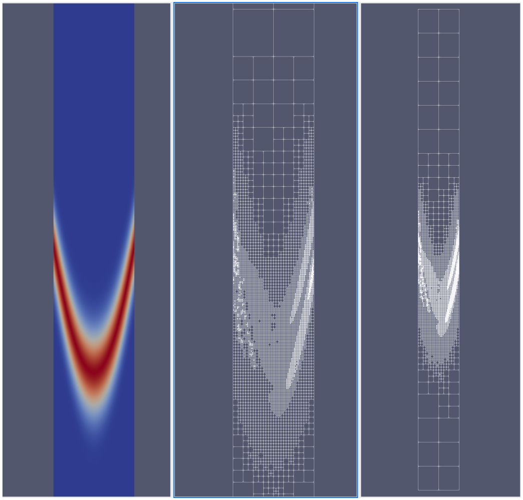
Time step 10.

Adaptive Solve



Time step 20.

Adaptive Solve



Time step 100. In contrast to the fixed-mesh solution, here there is **no** visible error accumulation at inflow (or elsewhere).

Motivation: Sum Factorization

For hexahedral elements in 3D:

- standard assembly: $O(p^9)$ flops
- sum factorization: $O(p^7)$ flops in general; $O(p^6)$ flops for constant-Jacobian case.

Savings increase for higher dimensions...

Basic idea: save flops by factoring sums.

	Adds	Multiplies	Total Ops
$\sum_{i=1}^N \sum_{j=1}^N a_i b_j$	$N^2 - 1$	N^2	$2N^2 - 1$
$\sum_{i=1}^N a_i \sum_{j=1}^N b_j$	$2N - 2$	N	$3N - 2$

Intrepid2's Basis Class

- Principal method: `getValues()` — arguments: points, operator, Kokkos View for values
- Fills the View with basis values at each ref. space quadrature point.

Structure has been lost:

- points: flat container discards tensor structure of points.
- values: each basis value is the product of tensorial component bases; we lose that by storing the value of the product.

Both points and values will generally require (a lot) more storage than a structure-preserving data structure would allow.
But our main interest is in the impediment to algorithms that take advantage of the structure.

Structure-Preserving Data Classes in Intrepid2

- `CellGeometry`: general class for specifying geometry, with support for low-storage specification of regular grids, as well as arbitrary meshes.

Structure-Preserving Data Classes in Intrepid2

- `CellGeometry`: general class for specifying geometry, with support for low-storage specification of regular grids, as well as arbitrary meshes.
- `Data`: basic data container, with support for expression of regular and/or constant values without requiring redundant storage of those.

Structure-Preserving Data Classes in Intrepid2

- `CellGeometry`: general class for specifying geometry, with support for low-storage specification of regular grids, as well as arbitrary meshes.
- `Data`: basic data container, with support for expression of regular and/or constant values without requiring redundant storage of those.
- `TensorData`: tensor product of `Data` containers; allows storage of tensor-product basis evaluations such as those from H^1 value basis evaluation.

Structure-Preserving Data Classes in Intrepid2

- `CellGeometry`: general class for specifying geometry, with support for low-storage specification of regular grids, as well as arbitrary meshes.
- `Data`: basic data container, with support for expression of regular and/or constant values without requiring redundant storage of those.
- `TensorData`: tensor product of `Data` containers; allows storage of tensor-product basis evaluations such as those from H^1 value basis evaluation.
- `VectorData`: vectors of `TensorData`, possibly with multiple families defined within one object. Allows storage of vector-valued basis evaluations.

Structure-Preserving Data Classes in Intrepid2

- `CellGeometry`: general class for specifying geometry, with support for low-storage specification of regular grids, as well as arbitrary meshes.
- `Data`: basic data container, with support for expression of regular and/or constant values without requiring redundant storage of those.
- `TensorData`: tensor product of `Data` containers; allows storage of tensor-product basis evaluations such as those from H^1 value basis evaluation.
- `VectorData`: vectors of `TensorData`, possibly with multiple families defined within one object. Allows storage of vector-valued basis evaluations.
- `TensorPoints`: tensor point container defined in terms of component points.

Structure-Preserving Data Classes in Intrepid2

- `CellGeometry`: general class for specifying geometry, with support for low-storage specification of regular grids, as well as arbitrary meshes.
- `Data`: basic data container, with support for expression of regular and/or constant values without requiring redundant storage of those.
- `TensorData`: tensor product of `Data` containers; allows storage of tensor-product basis evaluations such as those from H^1 value basis evaluation.
- `VectorData`: vectors of `TensorData`, possibly with multiple families defined within one object. Allows storage of vector-valued basis evaluations.
- `TensorPoints`: tensor point container defined in terms of component points.
- `BasisValues`: abstraction from `TensorData` and `VectorData`; allows arbitrary reference-space basis values to be stored.

Structure-Preserving Data Classes in Intrepid2

- `CellGeometry`: general class for specifying geometry, with support for low-storage specification of regular grids, as well as arbitrary meshes.
- `Data`: basic data container, with support for expression of regular and/or constant values without requiring redundant storage of those.
- `TensorData`: tensor product of `Data` containers; allows storage of tensor-product basis evaluations such as those from H^1 value basis evaluation.
- `VectorData`: vectors of `TensorData`, possibly with multiple families defined within one object. Allows storage of vector-valued basis evaluations.
- `TensorPoints`: tensor point container defined in terms of component points.
- `BasisValues`: abstraction from `TensorData` and `VectorData`; allows arbitrary reference-space basis values to be stored.
- `TransformedVectorData`: `VectorData` object alongside a transformation matrix, stored in a `Data` object, that maps it to physical space.

Two Sum Factorization Approaches

In N-dimensional hypercube integration, we can have $N+2$ nested summations; we want to compute and store these in an efficient manner.

We implement two sum factorization algorithms:

1. **Basis-indexed**:
   - standard approach (see e.g. Mora & Demkowicz)
   - loop nesting structure: point loops contain basis loops
   - intermediates are indexed by basis ordinals, with implicit reference to quadrature indices
2. **Point-indexed**:
   - our design, based on Intrepid2 data layout: we attempt to improve data locality.
   - loop nesting: basis loops contain point loops
   - intermediates are indexed by point ordinals, with implicit reference to basis ordinals

Estimated Flops for Each Algorithm

We use Poisson assembly on a 16^3 grid, with elementwise integrals of the form

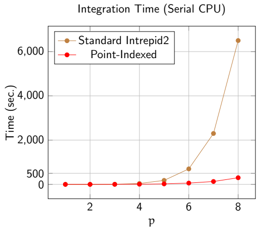
$$K_{ij} = \int_K \nabla \phi_i \cdot \nabla \phi_j \, \partial K,$$

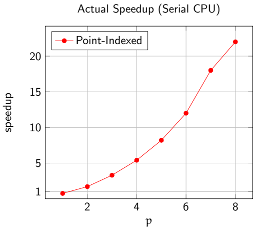
as our test problem. We implement a flop estimator (counting each add or multiply as one flop), with results:

p	Standard	Basis-Indexed	Speedup	Point-Indexed	Speedup
1	1.6e+07	2.7e+07	0.60x	2.9e+07	0.55x
2	5.3e+08	3.6e+08	1.5x	3.8e+08	1.4x
3	6.7e+09	2.4e+09	2.8x	2.5e+09	2.7x
4	4.9e+10	1.1e+10	4.5x	1.1e+10	4.5x
5	2.5e+11	3.7e+10	6.8x	3.9e+10	6.4x
6	1.0e+12	1.1e+11	9.1x	1.1e+11	9.1x
7	3.3e+12	2.7e+11	12x	2.7e+11	12x
8	9.6e+12	6.0e+11	16x	6.1e+11	16x

(Speedup values here are theoretical, based only on flop counts.)

Poisson Results: Serial





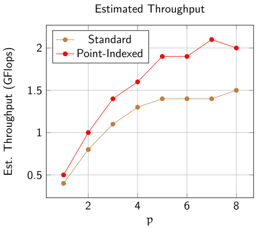
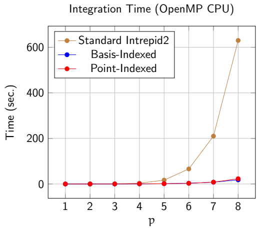
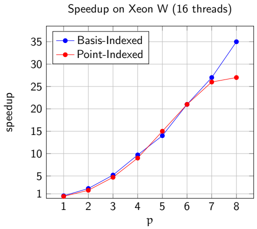


Figure: Serial (Intel Xeon W, 2.3 GHz) timing comparison for 3D Poisson integration, 4096 elements. (Optimal workset sizes for each case determined experimentally.)

Poisson Results: OpenMP





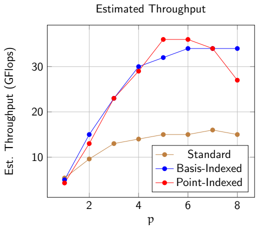


Figure: OpenMP (Intel Xeon W, 2.3 GHz, 16 threads) timing comparison for 3D Poisson integration, 4096 elements. (Optimal workset sizes for each case determined experimentally.)

Poisson Results: CUDA P100

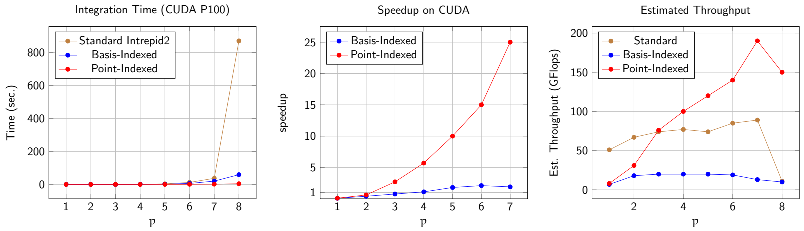


Figure: CUDA (P100) timing comparison for 3D Poisson integration, 4096 elements. (Optimal workset sizes for each case determined experimentally.)

Note: The $p = 8$ case has a dramatic slowdown for standard (for this case, the only workset size that ran to completion was 1); we exclude it from the speedup plot so as to not to throw off the scaling.