Extending Camellia: Distributed Stiffness Matrix Determination with MPI

Jesse Chan, Nathan V. Roberts

Center for Predictive Engineering and Computational Sciences
Institute for Computational Engineering and Sciences
The University of Texas at Austin

23 January 2012

Outline

1. Status Update in Brief
2. Extending Camellia for MPI
3. Timing Experiments
4. Parallel Adaptivity
5. Future Work

What we had in September 2011

Support for:

- meshes of arbitrary degree, with arbitrary combinations of triangles and quads
- p-refinements
- easy-to-specify bilinear forms
- easy-to-specify inner products, **including automatically specified mathematician's and optimal inner products**
- rudimentary plotting of field variables using MATLAB
- all computations done in serial

What we've added

Support for:

- h-refinements
- automatic adaptivity
- mesh-dependent inner products
- better plotting of fields, plus simultaneous flux plotting, using MATLAB
- **distributed optimal test function and stiffness matrix determination using MPI**
- next up/underway: work on 2D Burgers' equation (following MIT paper)

Goals for Camellia+MPI

Immediately (today):

- distributed optimal test function and stiffness matrix determination
- perform timing experiments to measure speedup and determine where the serial bottlenecks are

Eventually (someday):

- parallel data structures for Mesh, Solution
- parallel solver (MUMPS already works on Nate's laptop)

Distributing Element Computations: Partitioning Strategy

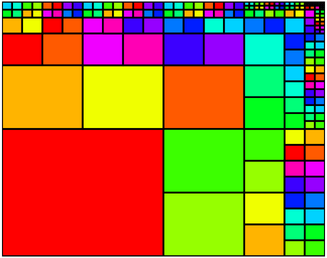
To take advantage of MPI, we need to partition the Mesh somehow. What's a good way to do this?

- Key consideration in designing distributed algorithms: **data locality** and **minimizing inter-node communication**.
- In FEM algorithms, data locality generally follows spatial locality—in DPG, we'll have information to communicate about fluxes and traces along each edge shared by two MPI nodes.
- We use a Hilbert space-filling curve to determine a spatially-local partition of elements, using Trilinos's Zoltan package.

Example Partitions

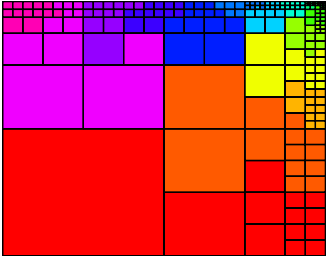
"Cyclic" partitioning

Cyclic Partitioning



Hilbert SFC partitioning

HSFC Partitioning



Details of Original Mesh Implementation: ElementType

- The low-level methods used to do integration, etc. in Trilinos allow **batching** of elements—i.e. computations can be done for several elements at once for better efficiency.
- These methods assume that all elements in a batch are of like type.
- Camellia's meshes have elements of different types.
- Design goal: allow computations to be done on all elements that are of like type.

All of this motivates the introduction of the ElementType class.

Details of Original Mesh Implementation: ElementType

Camellia's ElementType is determined by the following features of the element:

- trial space (a *DofOrdering* object),
- test space (a *DofOrdering* object),
- element topology (a Trilinos *CellTopology* object)

All elements of like type can be addressed as a batch. Supporting this required the creation of a number of structures within Camellia's Mesh class that are indexed by ElementType.

Two Element-Partitioning Mechanisms

The division of the Mesh into elements of like ElementType can be thought of as a partitioning of the Mesh. How should this interact with the partitioning provided by the HSFC?

- effectively, want a **two-level** partition: first, apply HSFC, and then use ElementType within an MPI node for batching
- this means that each of the Mesh's data structures indexed by ElementType should also be indexed by partitionNumber
- unfortunately, we still need to support some mesh-global operations indexed by ElementType: thus we need to duplicate several of our lookups.
- the eventual goal: get rid of those mesh-global operations, replacing them with distributed versions.

Other Code Modifications

Changes required outside of Mesh:

- introduce abstract MeshPartitionPolicy class, with concrete HSFC and Cyclic implementations
- in Solution, determine dof partitioning, and supply that to the Epetra CrsMatrix
- in Solution: after the solve, distribute solution coefficients to all nodes (will not be required once we have a completely distributed implementation of all methods)

Experimental Setup

For a convection-dominated diffusion problem, we used four meshes of size varying from 202 elements to 12928 elements, and timed the overall computation, as well as the following individual components on each node:

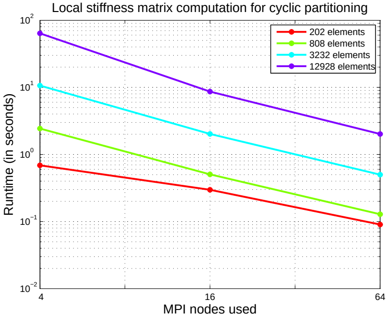
- local stiffness matrix computation
- global stiffness matrix assembly
- solve

We ran this on Lonestar, using between 1 and 64 MPI nodes. For a cleaner experiment, we only used 1 core from each machine.

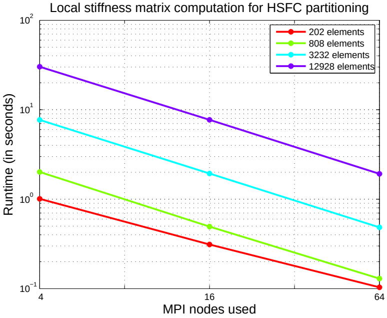
Strong Scaling

Strong scaling: fix the problem size, and distribute the workload across increasing numbers of processors.

"Cyclic" local stiffness computation

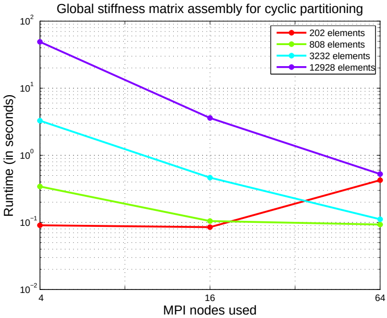


Hilbert SFC local stiffness computation

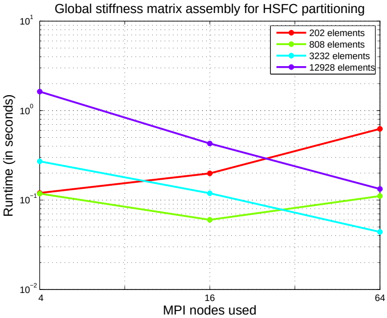


Strong Scaling, Global Assembly

“Cyclic” global assembly



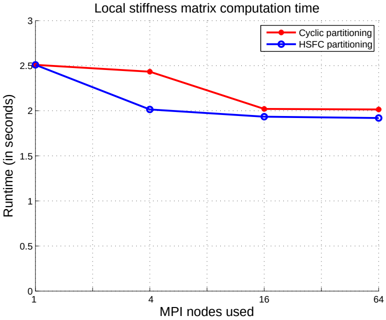
Hilbert SFC global assembly



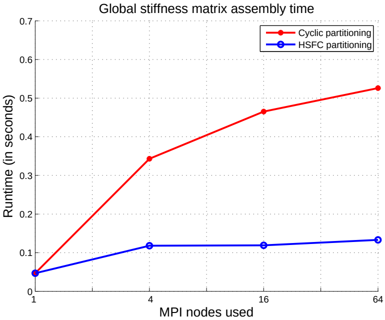
Weak Scaling

Weak scaling: fix the problem size per MPI node, and increase the number of MPI nodes. Hope to see constant runtime.

Local Stiffness



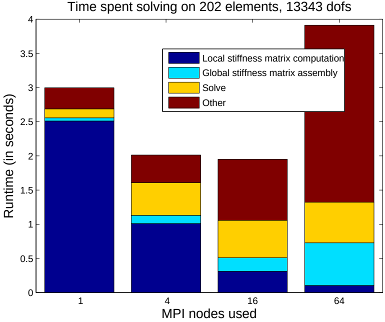
Global Assembly



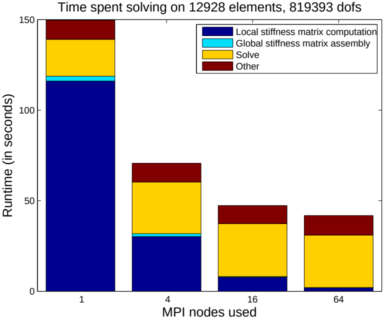
Total Runtime Analysis

Examine the fraction of time spent in various computations as the number of MPI nodes increases.

202 elements, 13,343 does



12,928 elements, 819,393 dofs

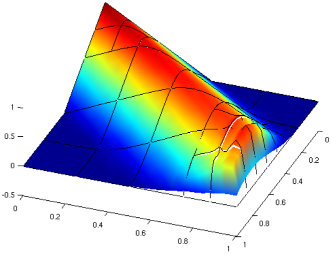


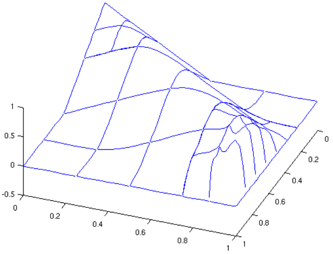
Experiments with parallel adaptivity

We have implemented Heuer and Demkowicz's inner product in Camellia

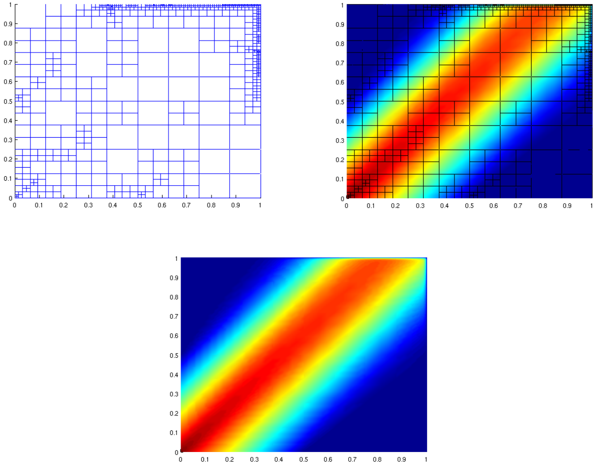
$$((\tau, v), (\delta\tau, \delta v))_V = C(K, \epsilon)\|v\| + \epsilon\|\nabla v\| + \|\beta \cdot \nabla v\|_w + \|\tau\|_w + \|\nabla \cdot \tau\|_w$$

where $C(K, \epsilon) = \min(\epsilon, |J(K)|)$.

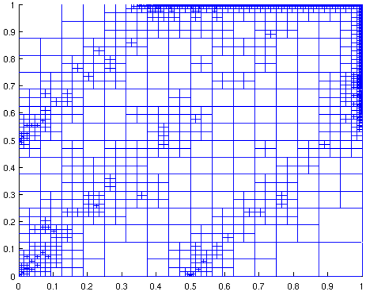


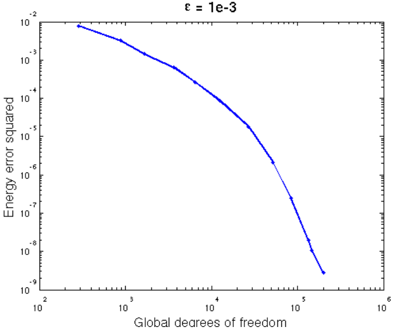


For better pictures, $\epsilon = 5e - 2$, slightly skew advection.



To make sure we still work at smaller scales, $\epsilon = 1e-3$





Future Work

We are pretty satisfied, for now, with our parallel performance: we expect that we will be able to do our 2D Navier-Stokes simulations in reasonable time with the code as it stands. But there is still significant opportunity to extend Camellia to solve larger problems still:

- distribute the solve using MUMPS or an iterative solver,
- improve load balancing for meshes of variable polynomial order,
- distribute mesh construction and storage, and
- distribute solution storage.

Thank you.

Questions?