Extending Camellia for Distributed Stiffness Matrix Determination in DPG

Jesse Chan and Nathan V. Roberts

1 Introduction to DPG

The Discontinuous Petrov-Galerkin (DPG) method is a finite element method introduced by Demkowicz and Gopalakrishnan in [**?**] and [**?**]. The DPG method was first applied to solve the convection-dominated diffusion equation

$$\nabla \cdot (\beta u - \epsilon \nabla u) = f,$$

a prototype for shock and boundary layer problem when $\epsilon \ll 1$, using higher order adaptive meshes in [**?**] and [**?**]. For standard finite element methods, the convection-dominated diffusion problem suffers from large oscillations that grow in amplitude with $\frac{1}{\epsilon}$. This behavior illustrates an example of the *instability* of standard finite element methods when it comes to problems with small parameters.

DPG is a *discontinuous* Galerkin (DG) method, which means that the solution is allowed to be discontinuous across inter-element boundaries, with continuity weakly enforced through numerical fluxes defined on element boundaries. DPG is also a *Petrov*-Galerkin method, a finite element method in which the test and trial spaces are allowed to differ. Petrov-Galerkin methods allow selection of test functions that improve the behavior of the finite element. The Streamline Upwind Petrov-Galerkin method of Hughes and Brooks [**?**] is an example of such a method; it aims to eliminate the unstable oscillations in the convection-dominated diffusion problem for Continuous Galerkin (CG) methods.

DPG has also been successfully applied to problems in wave propagation and linear elasticity [**?**, **?**] using the serial Fortran code `hpDPG`. Roberts has recently developed Camellia [**?**], a toolbox for solving problems using DPG built atop Sandia's Trilinos Project [**?**]. Prior to the present work, Camellia supported only serial execution. The present work extends Camellia to allow distributed stiffness matrix computation, using the convection-dominated diffusion problem as a testbed.

This report is organized as follows. In Section **??**, we provide the technical details of DPG that are important to understand the present work. In Section **??**, we discuss our extensions to Camellia to support distributed stiffness matrix determination. We describe some numerical experiments to examine how well Camellia scales to multiple processors in Section **??**. Finally, in Section **??**, we discuss work that we hope to do in the future to improve Camellia's scalability further.

2 Introduction to DPG

2.1 Optimal Test Functions

The definition of an optimal test function is rigorously defined in [**?**], and has been well-known in the finite element community for some time. However, this idea proved to be impractical, as the cost to computing a single optimal test function was equivalent to the cost of solving the original

problem over the entire mesh. The key contribution of DPG has been to note that the global nature of this problem is due to the continuous nature of the trial space. For a Discontinuous Galerkin method, the test space is also necessarily discontinuous, which decouples the global problem into a local problem over each element.

With respect to optimal test functions, a discontinuous test space allows the test functions to be computed *locally*, on an per-element basis. Not only does this decrease the problem size dramatically, it is also intrinsically parallel - the optimal test functions over each element can be computed completely independently of each other.

With DPG, given any basis $\{u_i\}$ for the trial space, we can compute the ith optimal test function over an element by solving the auxiliary problem

$$(v_i, \delta v)_V = b(u_i, \delta v), \quad \forall \delta v \in V$$

where V is the space of admissible test functions (with proper regularity) on the element. For a trial function whose support spans multiple elements, the optimal test function is solved for independently over each element, then set as the union of the test functions over each element the trial function spans. In practice, we solve this problem approximately by taking V to be the space of admissible test functions of order $p + \Delta p$, where p is the order of the trial space. This results in a fairly small local system

2.2 Hybridized DG formulation

In DG methods, elements are coupled to each other through a numerical flux defined on the boundary; for standard DG methods, this numerical flux is computed using some predetermined function of the finite element solution in the interior. The DPG method is an example of a hybridized DG formulation, where the numerical flux is instead solved for as part of the solution, introducing additional unknowns to the problem (called hybridization).

However, this hybridization produces more loosely coupled blocks in the global stiffness matrix than standard DG methods, which, for certain numerical fluxes, will produce the same coupled sparsity pattern as a CG method. For a hybridized DG formulation, the finite element solution is separated into interior degrees of freedom and boundary/flux degrees of freedom. The interior degrees of freedom (which are usually much more than the boundary degrees of freedom) for each element are completely decoupled from each other, implying that, for global assembly of the stiffness matrices, the amount of data communication needed between different processors is reduced. Thus, we expect the assembly process to scale to large numbers of processors fairly well.

3 Implementation

3.1 Camellia/Trilinos implementation

In this section, we describe the changes that we implemented within Camellia to allow the work to be partitioned and executed efficiently within an MPI environment.

Camellia was originally written to run in serial, so a number of changes were required to allow MPI execution—the most significant of these had to do with element partitioning and partitioning the degree-of-freedom coefficients. Trilinos's Intrepid routines are designed to execute on batches of cells that are alike in topology and polynomial order. We allow meshes of non-uniform topology and polynomial order; thus we had implemented an ElementType class which allowed us to identify elements that could be batched. Our Mesh class maintained data structures such that information

pertaining to a particular ElementType could be accessed quickly—for example, Mesh stored a data structure that contained the vertex coordinates belonging for elements of a given ElementType.

The upshot of this design is that mesh information is already, in a sense, partitioned. However, there is no reason to expect that this will be a good partitioning for dividing up the work of stiffness matrix computation across MPI nodes—for this, we want our mesh partitioning to keep spatially adjacent elements together within a partition to minimize the amount of data that needs to be communicated during global stiffness matrix assembly.

Thus, to maintain both of our desired design features—batching by ElementType and spatially contiguous partitioning across MPI nodes—we require a two-level partitioning; we first divide the elements spatially (the MPI partitions), and then batch together elements of like ElementType within each of these partitions. (Unfortunately, this is not quite the whole story; because other classes depend on the ElementType division of the whole mesh, we have had to maintain the old lookup tables as well. We do hope to eliminate these dependencies in the end.)

To facilitate experiments with multiple partitioning strategies (and to maintain separation of concerns), we designed an abstract MeshPartitionPolicy class which, given a Mesh and the number of partitions desired, returns a partition of the elements in the mesh. The Mesh then uses this to generate a partition of the degree-of-freedom coefficients, which induces a partition of the rows in the stiffness matrix. The latter partition is used in conjunction with an Epetra FECrsMatrix, a distributed stiffness matrix assembly facility within Trilinos. The FECrsMatrix allows storage of data not owned by the current MPI node; this data is communicated to the owning MPI node during global assembly of the stiffness matrix. A convenient consequence of this is that we can separate the cost of global assembly from the cost of local stiffness matrix computations; the latter are independent across MPI nodes, and therefore we expect to see perfect scaling for these. The scaling of the global assembly will depend on the quality of the partition and the cost of communication between MPI nodes. We do not hope for perfect scaling here, but we do hope to see a cost per dof per node that does not grow too quickly in the total number of nodes.

3.2 Partitioning

Since Camellia is tightly integrated with the Trilinos library already, we use the Zoltan package in Trilinos to generate spatially local partitions of the mesh [**?**]. We interface specifically with the Hilbert Space-filling Curve and Reftree partitioning algorithms in Zoltan, both of which can include parallel implementations.

The Hilbert space-filling curve (HSFC) algorithm works by drawing a curve through the geometric mesh in 2 or 3 dimensional space, then mapping that curve to the interval $[0, 1]$. The partitioner then divides up this interval (in an even or weighted manner) into n partitions, and maps each of those partitions back into 2 or 3D. All that's required of the HSFC algorithm is a listing of elements and a geometric identifier associated with a given element. In this case, we use the element centroid as such an identifier.

The Reftree algorithm [**?**] adds complexity in that each processor must store information not only about the active nodes, but the ancestors as well. Additionally, Reftree requires that, if you store an ancestor, you must store all its children as well, increasing the amount of redundancy between distributed refinement trees. However, due to Reftree being more agnostic towards element types (quad vs triangle, tetrahedron vs cube), the same algorithm performs similarly for many different types of refined meshes (as opposed to HSFC, which works best for quadrilateral grids, but not for triangular grids). Additionally, Reftree's algorithm guarantees connected partitions for tetrahedral and triangular grids, and generally performs more robustly on other types of grids as well.

Currently, Reftree has been implemented and tested on local clusters. However, Reftree currently

crashes during runs on Lonestar, our HPC architecture for this report. We use the HSFC partitioner for the scaling tests in this report, and hope to fix issues with Reftree in the near future.

4 Scaling tests

Handicaps - Batch cache size - Using all 12 cores on a single node

5 Future work

5.1 *hp*-meshes

5.2 Parallel solvers

5.2.1 MUMPS

5.2.2 Conjugate gradients/static condensation

One benefit of DPG is that the global stiffness matrices it produces are guaranteed to be symmetric positive-definite, allowing for iterative methods like CG to be used.

Another benefit of the hybridized DG formulation is that the global solve can be statically condensed down to a solve involving only the flux degrees of freedom.