Extending Camellia for Distributed Stiffness Matrix Determination in DPG

Jesse Chan Nathan V. Roberts

1 Introduction

The Discontinuous Petrov-Galerkin (DPG) methodology is a class of finite element methods introduced by Demkowicz and Gopalakrishnan in [3] and [4]. DPG 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 [4] and [5]. 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 [2] 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 computes test functions on the fly which are *optimal* in that they deliver the best available approximation to the solution, as measured in an energy norm defined by the problem and a test space which may be chosen by the analyst.

DPG has also been successfully applied to problems in wave propagation and linear elasticity [10, 1] using the serial Fortran code `hpDPG`. Roberts has recently developed Camellia [9], a toolbox for solving problems using DPG built atop Sandia's Trilinos Project [7]. 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 2, we provide the technical details of DPG that are important to understand the present work. In Section 3, 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 4. Finally, in Section 5, we discuss work that we hope to do in the future to improve Camellia's scalability further.

2 DPG: Some Technical Details

2.1 Optimal Test Functions

The concept of an optimal test function is rigorously defined in [4], and has been well-known in the finite element community for some time. However, the idea was considered impractical, as the cost of computing a single optimal test function was equivalent to the cost of solving the original problem over the entire mesh. A 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 element-local problems.

A discontinuous test space allows the test functions to be computed *locally*, on a per-element basis. Not only does this decrease the problem size dramatically, it is also intrinsically parallel—the optimal test functions for each element can be computed 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, and $(v_i, \delta v)_V$ is some inner product on V (choices for this inner product are discussed in [4], [10]). 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 small local system on each element which is solved to determine a test function corresponding to each degree of freedom on the element. Δp is taken to be 2 in this report.

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, which means that the numerical flux is solved for as part of the solution by introducing additional unknowns to the problem.

Whereas standard DG methods can (for certain numerical fluxes) produce global stiffness matrices whose elements are as tightly coupled as a CG method would, hybridization produces more loosely coupled blocks in the matrix. 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 (of which there are usually many more than boundary degrees of freedom) for each element are decoupled from each other. Thus the inter-node communication required during global stiffness matrix assembly is reduced. We therefore expect the assembly process to scale to large numbers of processors, particularly if we use a good partitioner.

3 Implementation

3.1 Extending Camellia

In this section, we describe the changes that we implemented within Camellia to allow the stiffness matrix determination to be 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 degree of freedom per node that does not grow too quickly in the total number of nodes.

The MeshPartitionPolicy partitions the mesh into sets of elements assigned to an MPI node; for the interface to Epetra, we need a partition of rows of the global stiffness matrix (each of which corresponds to a degree of freedom). Degrees of freedom corresponding to numerical fluxes are shared by the elements along whose boundaries they lie. We adopt a simple greedy policy for this—shared d.o.f.s are assigned to the element with the smallest global identifier. This means that even if the element partitioning were perfectly balanced, some MPI nodes might still have more d.o.f.s assigned to them than others. This will not affect the amount of work involved in local stiffness matrix computations, but it may affect MPI communication costs during global assembly.

3.1.1 Design Limitations: where Amdahl's law might come to haunt us

Here, we note a few features of the present design that may become bottlenecks as we scale up to larger problems and larger numbers of MPI nodes.

At present, our global solve is implemented using the KLU implementation provided by the Amesos package within Trilinos. This is a direct solve, executed in serial. Amesos gathers the global stiffness matrix to rank 0, solves, and then scatters the solution according to the partition we have defined. For our serial computations prior to the present work, the cost of the global solve has been dominated by the cost of the local optimal test function determination; however, now that the latter cost has been distributed using MPI and the former has not, we expect the global solve to become a bottleneck, both in terms of overall execution time and in terms of the sizes of problems that can be solved, since the KLU solve must fit into a single node's memory to be practical.

Similarly, at present the entire mesh is determined and stored on every MPI node. This has the advantage that no MPI communication is required during mesh construction, but for very large meshes it may become impractical, because the time cost of mesh construction may grow relative to the distributed parts of the computation, and perhaps (for *extremely* large meshes) because the memory required for the mesh may grow too large for storage on a node.

Finally, certain features of the original code depend upon all the solution coefficients being available to the current processor, so for the time being we distribute the entire solution to every MPI node, after KLU has scattered according to the original partitioning. We hope to eliminate this dependence on having all solution coefficients available, allowing the solution to be distributed according to the same mapping as was used for the stiffness matrix.

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 [6]. 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 2D 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 [8] 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 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

We tested the scalability of our code on the Lonestar machine at the Texas Advanced Computing Center (TACC), verifying whether or not we observe strong/weak scaling for various parts of the program. We solve the convection-diffusion equation on a square mesh, with boundary conditions such that the solution develops a boundary layer (sharp gradient) near the north and east edges of the square. The solution for a refined mesh is shown in Figure 1.

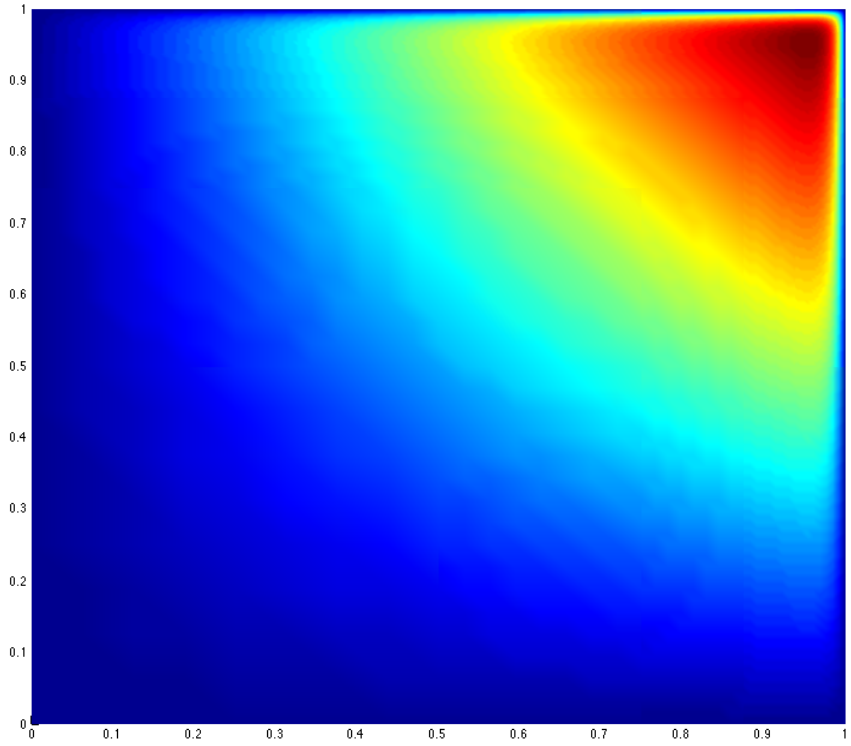


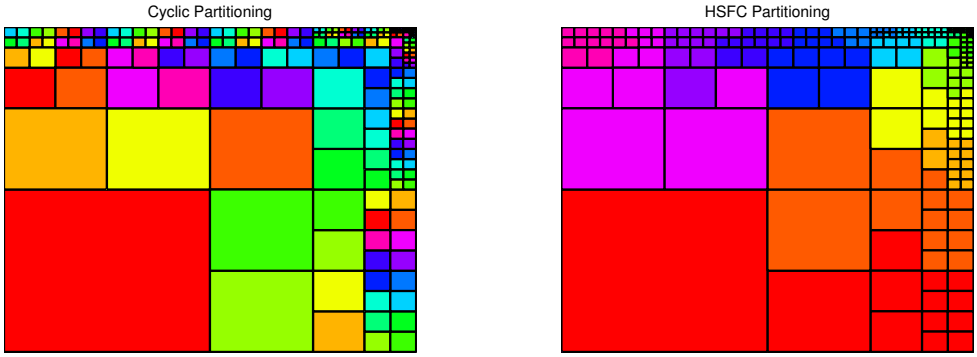
Figure 1: Computed solution for $\epsilon = 10^{-2}$ using a 12,928-element mesh. L^2 error is $O(10^{-5})$.

We set up our computational experiments by first refining all cells that share an edge with the north and east sides, and repeating this process four times. We then constructed three additional meshes, by uniformly refining every element in the mesh (quadrupling the number of elements each time). For each of these meshes, we solved on Lonestar, using 1, 4, 16, and 64 nodes, tracking the timing of various portions of the code. From this data, we extracted strong and weak scaling statistics, discussed below in Sections 4.1 and 4.2. The runtimes plotted in each of our figures are the mean of the times recorded at each MPI node.

We also used contiguous and discontiguous partitions of each mesh to measure the effect of partition quality on the communication costs in global stiffness matrix assembly. We created contiguous partitions of the mesh using a HSFC partitioner. To create the discontiguous partitioning,

we used the Zoltan cyclic partitioner, which loops through the list of active elements, sending the first element to the first processor, the next to the next processor, and so on. After reaching the last processor, the cyclic partitioner cycles back to the first processor and repeats this process.[1] The cyclic partitioning is a "worst case" scenario in the sense that each of the current MPI node's elements' neighbors are owned by some other MPI node, with the consequence that information concerning each element boundary must be sent via MPI. An example cyclic partitioning can be seen in Figure 2(a); Figure 2(b) shows an HSFC partitioning.

It's worth noting that the runtimes reported here do not represent Camellia running at peak efficiency. A single Lonestar node contains 12 cores. Because MPI communication has much higher bandwidth intra-node than inter-node, and for analysis we wanted approximately equal bandwidth between MPI processes, we use only one core per node. Utilizing all 12 cores on a node would decrease communication costs. Additionally, there is a parameter within Camellia that controls the batch size for the Trilinos Intrepid batching routines. Tuning this parameter based on machine-specific cache sizes should further improve performance.



(a) Cyclic partitioning

(b) HSFC partitioning

Figure 2: Discontiguous and contiguous partitions of the mesh.

4.1 Strong scaling

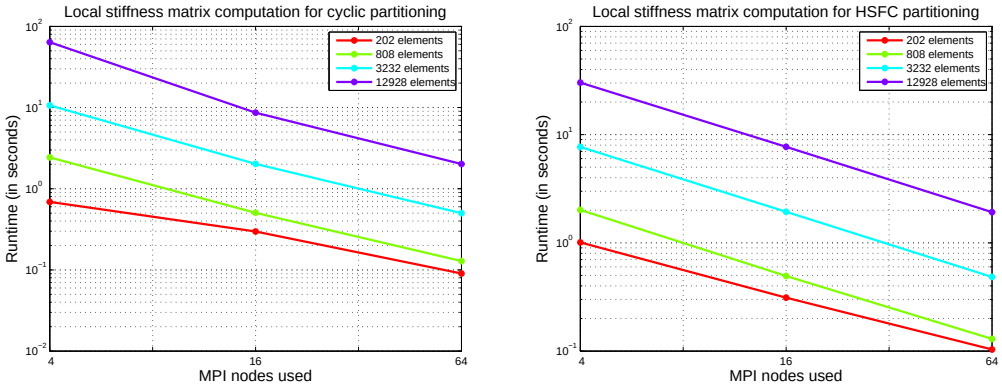
In our strong scaling tests, we considered problems of fixed size, distributing the workload for each among increasing numbers of MPI nodes. Here, we hope to see runtimes that decrease linearly in the number of MPI nodes—this is what we mean by achieving strong scalability.

Because the local stiffness computations are entirely independent of each other, requiring no communication among MPI nodes, we expect to see perfect scaling for these. As can be seen in Figures 3(a) and 3(b), we do achieve something very close to this, although the cyclic partitioning took longer in almost every case, for reasons that are unclear to us. It may have something to do with data locality on each node—by the manner of our construction, data concerning contiguous

[1] Due to the nature of our refinements in this test, a blocked partitioner (equally partitioning the ordered list of active elements by element number) happened to produce nearly-contiguous partitions. We therefore have omitted this partitioner from our analysis, since we do not believe that its performance here represents its performance in general.

mesh elements is more likely to be close together in memory than the data for mesh elements in the cyclic case.

It is less clear what we should expect for the global stiffness matrix assembly; we expect the HSFC partitioning to outperform the cyclic partitioning, and we would like to see the runtimes for assembly decrease as the number of MPI nodes increases. As can be seen in Figures 4(a) and 4(b), for the larger problems in both cases, we do have something approaching strong scalability. For the 202- and 808-element problems, we see the time cost increase (or fail to decrease, in the case of the 808-element problem for the cyclic partitioner) between 16 and 64 nodes. However, it's worth noting that in these cases, the real times involved are small: less than a second in each case. If our concern is our ability to scale up to large problems on large numbers of processors, it does not appear that the global assembly will be a bottleneck. The HSFC partitioning does show its strength here compared with the cyclic; for the 12,928 mesh on the 4-node run, the cyclic partitioner took an average of 49.1 seconds per node, compared with 1.63 seconds for the HSFC partition.



(a) Cyclic Partitioning

(b) HSFC Partitioning

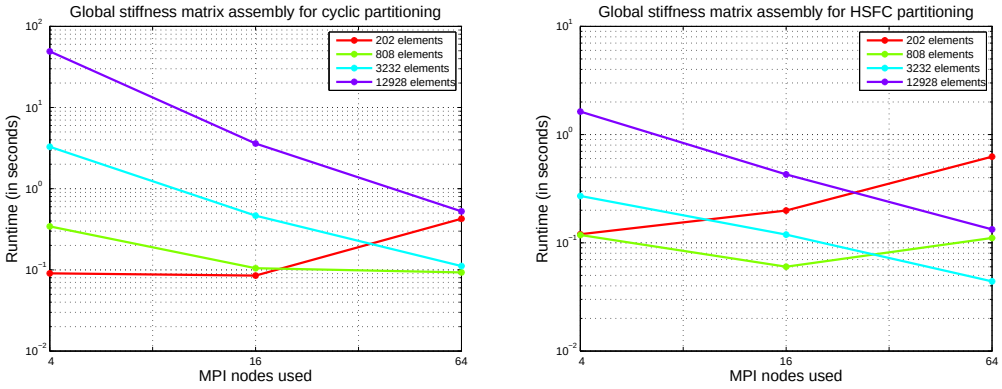
Figure 3: Strong scaling plots for the local stiffness matrix computation.

4.2 Weak scaling

In our weak scaling tests, we aimed to fix the size of the problem *per MPI node*, and examined the runtime of various portions of the execution—we hope to see a constant runtime as the problem size and number of processors increase, and this is what we mean by achieving weak scalability.

Both partition types achieve weak scalability in the local stiffness matrix computation, as can be seen in Figure 5(a). In fact, we see the total runtime *decrease* slightly as the total workload and number of MPI nodes increase—we are unsure of the reason for this. Similarly, although we would expect the partitioning not to affect this portion of the computation at all (since no inter-node communication occurs at this stage), for some reason the HSFC partitioning slightly outperforms the cyclic on each of the multi-node runs.

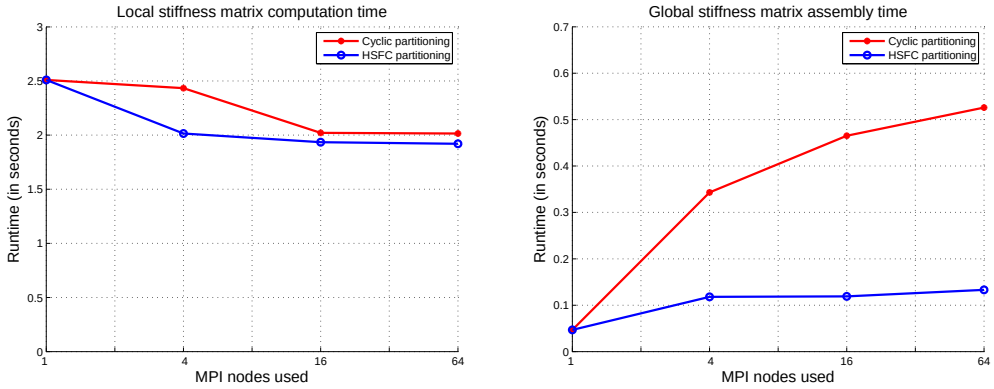
Assembly of the global stiffness matrix achieves weak scalability only for the contiguous case, as can be seen in Figure 5(b). It's worth noting, however, that though the poor mesh partitioning imposes much higher communication costs than the HSFC partitioning, the maximum amount of time spent on assembly is still only a small fraction of the time spent on computation of local stiffness matrices.



(a) Cyclic Partitioning

(b) HSFC Partitioning

Figure 4: Strong scaling plots for global stiffness matrix assembly. (Note the differing scales.)



(a) Local stiffness computation

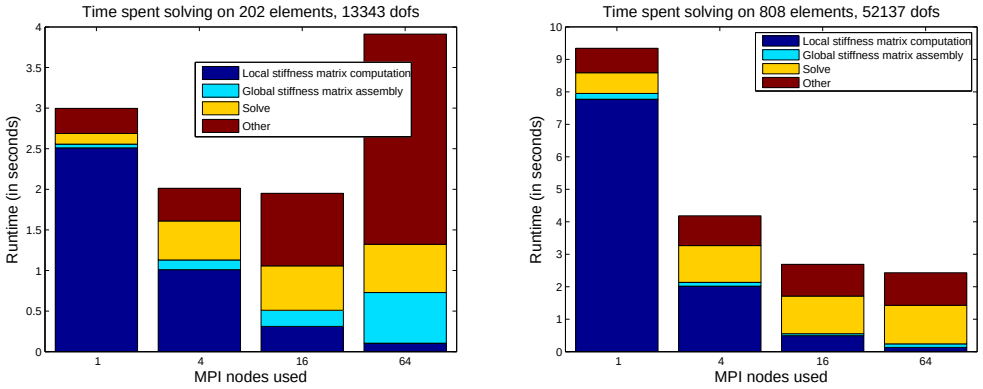
(b) Global stiffness assembly

Figure 5: Weak scaling results for computation of local stiffness matrices and global stiffness matrix assembly. For each run, both the number of MPI nodes and the number of elements increased by a factor of four.

4.3 Total runtime analysis

As the final component of our performance analysis, we examine components of the total runtime for each of our four problems using 1, 4, 16, and 64 MPI nodes; here we use the HSFC partitioner. This can be seen in Figure 6. We see that for the largest MPI runs the combined time spent on local stiffness computation and global assembly is a small fraction of the total time spent—since these were the targets of the present work, we can count it a success. The graphs also show that the cost of the global solve dominates on each of these runs, making this the obvious next target for optimization. In the "Other" category, the largest cost is the mesh determination. In the 12,928-element case, this takes about 5 seconds. Thus, a distributed mesh might be our next target after

the solve is optimized.



(a) 202 elements, 13,343 dofs

(b) 808 elements, 52,137 dofs

(c) 3,232 elements, 206,081 dofs

(d) 12,928 elements, 819,393 dofs

Figure 6: Breakdown of time spent in a solve for a series of HSFC-partitioned meshes.

5 Future work

There are several ways in which we hope to extend Camellia further to allow it to solve larger problems still; we hope to:

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

As discussed in Section 4.3 above, the lowest-hanging fruit for further improving the performance for the experimental setup considered in the present work is distributing the solve. We have already successfully interfaced with MUMPS—a parallel sparse direct solver—on one of our laptops; we expect that getting MUMPS to work on Lonestar will just be a matter of building Trilinos and Camellia with MUMPS there. One of the advantages of DPG is that its stiffness matrices are guaranteed to be symmetric positive definite, which opens the door to using iterative methods such as conjugate gradient methods. DPG also produces matrices whose entries are coupled only through the flux degrees of freedom; thus static condensation techniques can be used to reduce the size of the global system to be solved. (See Appendix A for a brief explanation of what's involved in static condensation.) We believe that iterative solvers and static condensation would allow us to solve larger problems than MUMPS will allow—we have heard that MUMPS only scales to about 128 processors.

In the present work, we considered a mesh of uniform polynomial order and considered all elements to be of equal weight; Camellia supports meshes with variable order, and Zoltan provides facilities for weighting elements differently. By taking advantage of this feature of Zoltan (using a weight corresponding to the number of degrees of freedom on each element), we can improve Camellia's load balancing for meshes of variable polynomial order. In fact, in doing so, we expect to improve slightly the load balancing for the setup considered here, because elements with hanging nodes do have slightly more degrees of freedom than those without.

At present, the mesh is constructed and stored on each node, as is the solution. It should be a relatively small extension to the present work to allow the solution to be distributed. Allowing mesh construction to be distributed is a bigger challenge, largely because the nodes must agree on the global stiffness matrix row indices. Given that we are working with an adaptive mesh, it will likely make sense to distribute the work of mesh refinement according to the partition of the previous mesh and use Zoltan to do the load rebalancing.

In summary, we have achieved good overall speedup for Camellia using a Hilbert space-filling curve partitioner, and have identified the next steps for improving its parallel performance further.

References

[1] Antti H. Niemi, Jamie A. Bramwell, and Leszek F. Demkowicz. Discontinuous Petrov-Galerkin Method with Optimal Test Functions for Thin-body Problems in Solid Mechanics. Technical Report 17, ICES, 2010.

[2] A.N. Brooks and T.J.R. Hughes. Streamline upwind/petrov-galerkin formulations for convection dominated flows with particular emphasis on the incompressible navier-stokes equations. *Comp. Meth. Appl. Mech. Engr*, 32:199–259, 1982.

[3] L. Demkowicz and J. Gopalakrishnan. A Class of Discontinuous Petrov-Galerkin Methods. Part I: The Transport Equation. *Comput. Methods Appl. Mech. Engrg.*, 2009. accepted, see also ICES Report 2009-12.

[4] L. Demkowicz and J. Gopalakrishnan. A Class of Discontinuous Petrov-Galerkin Methods. Part II: Optimal Test Functions. Technical Report 16, ICES, 2009. Numer. Meth. Part. D. E., in review.

[5] L. Demkowicz, J. Gopalakrishnan, and A. Niemi. A Class of Discontinuous Petrov-Galerkin Methods. Part III: Adaptivity. Technical Report 1, ICES, 2010.

[6] Karen Devine, Erik Boman, Robert Heaphy, Bruce Hendrickson, and Courtenay Vaughan. Zoltan data management services for parallel dynamic applications. *Computing in Science and Engineering*, 4(2):90–97, 2002.

[7] Michael Heroux, Roscoe Bartlett, Vicki Howle Robert Hoekstra, Jonathan Hu, Tamara Kolda, Richard Lehoucq, Kevin Long, Roger Pawlowski, Eric Phipps, Andrew Salinger, Heidi Thornquist, Ray Tuminaro, James Willenbring, and Alan Williams. An Overview of Trilinos. Technical Report SAND2003-2927, Sandia National Laboratories, 2003.

[8] W.F. Mitchell. A refinement-tree based partitioning method for dynamic load balancing with adaptively refined grids. *Journal of Parallel and Distributed Computing*, 67:417–429, 2007.

[9] Nathan V. Roberts, Denis Ridzal, Pavel B. Bochev, and Leszek D. Demkowicz. A Toolbox for a Class of Discontinuous Petrov-Galerkin Methods Using Trilinos. Technical Report SAND2011-6678, Sandia National Laboratories, 2011.

[10] J. Zitelli, I. Muga, L. Demkowicz, J. Gopalakrishnan, D. Pardo, and V. Calo. A Class of Discontinuous Petrov-Galerkin Methods. Part IV: Wave Propagation Problems. Technical Report 17, ICES, 2010.

A Static Condensation

One benefit of DPG's hybridized DG formulation is that the global solve can be reduced down to a solve involving only the flux degrees of freedom. Since the interior degrees of freedom are completely decoupled from each other, the system can be reordered to yield

$$\begin{pmatrix} A & B \\ B^T & C \end{pmatrix} \begin{pmatrix} u \\ f \end{pmatrix} = \begin{pmatrix} F \\ G \end{pmatrix}$$

where A is block diagonal. Noting that $u = A^{-1}(F - Bf)$, we can substitute this into the second solution to obtain a problem purely in terms of the flux degrees of freedom

$$(C - B^T A^{-1} B) f = G - B^T A^{-1} F$$

Since A is easy to invert, and B is a much smaller system, this *static condensation* would reduce the cost of the solve.