Parallel Graph Analysis and Adaptive Meshing using Graphics Processing Units

Timothy McGuiness and J. Blair Perot

Department of Mechanical and Industrial Engineering, University of Massachusetts, Amherst, MA, USA

Email: *tmcguine@engin.umass.edu*

Abstract

The mathematical concept of a graph can be used to model many real-world problems. Graph theory has applications in the fields of computer science, sociology, engineering and transportation, to name a few. This research aims to not only develop graph analysis algorithms, but to run them in parallel using commodity graphics hardware; a low-cost tool readily available to the computing community. The first part of this study focuses on navigating the graph's data structure, as well as finding an efficient algorithm to evaluate nodal connectivity. In addition, this work examines the application of these general algorithms in the engineering and CFD fields. Parallel versions of mesh smoothing (r-refinement) and partitioning algorithms are explored, drawing on key concepts and techniques developed for the graph analysis software. Both the graph analysis and mesh adaptation algorithms are shown to yield considerable speedups over comparable CPU code. Finally, the use of Message Passing Interface (MPI) is examined in an effort to obtain further performance improvements by running the code simultaneously on multiple graphics processors.

1 Background

For some time, the computing world has been shifting paradigms, moving from serial to parallel processing. Many modern commercial PCs now come equipped with multi-core processors, and the scientific community has used parallel clusters and supercomputers for many years. However, one of the newest venues for parallel computing has come from an unexpected source; the graphics community. Commodity graphics cards have recently proven to be an inexpensive and effective way to provide parallelization to the scientific programmer. Due to their non-standard processor architecture and superior memory bandwidth [14], graphics processing units, or GPUs, are quickly emerging as some of the best processors available today in terms of floating point operations per dollar. General purpose computing on graphics processing units (GPGPU), is a new field which is beginning to draw attention from the engineering community as a way to improve the performance of costly CFD simulations [13, 15].

The mathematical concept of a graph is not much different from that of a mesh; a commonly used tool in engineering applications. A graph consists of a set of nodes, or vertices, and a set of edges that connect them. Not only are these two structures physically similar, but the concept of topology is central to both. The fundamental difference is that graphs are more general since they deal strictly with connectivity. Nodes in a graph can be connected to *any* other node, and edge overlap is not an issue. On the other hand, meshes are restricted by the physical location of constituent nodes, which can only be connected to those which are nearby.

Mesh smoothing, or r-refinement, is a simple yet effective mesh adaptation technique. It involves moving existing mesh points to areas where higher resolution is needed, essentially stretching and compressing the existing mesh. While cell volumes and node locations change in r-refinement methods, an important benefit is that mesh topology does not [4, 5]. In a parallel architecture, this means load balancing is a one-time cost. A significant disadvantage of smoothing can be the creation of low-quality (small cell volume) or

invalid (zero or negative volume) elements. A great deal of effort has been spent in an attempt to prevent the formation of these elements, with solution methods proposed by Farhat and Degand [7, 9], Zeng and Either [19], and most recently Bottaso and Agickoz [5, 1].

Finally, the idea of partitioning is essential in the parallelization of mesh-based algorithms. To work with large meshes on a parallel system, the data structure must be split up and distributed to the system's processors. To run efficiently, it is crucial to break up the workload evenly, and minimize connections between partitions [8, 10]. There are several popular methods for achieving an appropriate partition for a given mesh, and they vary in their complexity and effectiveness. Multilevel partitioning has been researched extensively by Walshaw *et al.* [16, 17, 18], and are also the basis for the popular METIS and ParMETIS programs [11, 12]. A more simplistic method proposed by Farhat [8] involves building Voronoi subdomains, and is the basis for the algorithm used for this research.

2 Algorithms

2.1 Graph Analysis

The Scalable Synthetic Compact Applications (SSCA) benchmarks developed by the High Productivity Computer Systems program (HPCS) are tasks intended to test the performance of parallel machines [2]. The SSCA 2 benchmark is a program designed to analyze very large graphs. In the case of SSCA 2, the graphs are directed and weighted, meaning edges have specific start and end nodes, as well as a cost or weight value.

2.1.1 Scalable Data Generator

The untimed Scalable Data Generator, or SDG, produces the initial list of edges. Given a parameter 'SCALE' as input, the number of nodes in the graph is set as 2^{SCALE}, and the number of edges is eight times the number of nodes.

Each edge entry contains three values; start (or parent) node, end (or child) node, and weight. Lists of start and end nodes are created using a random number generator, and are then tested for "bad" entries. Self-loops and multiple edges - seen in Fig. 1 - are removed from the list during a filtering process. The creation-filtering loop

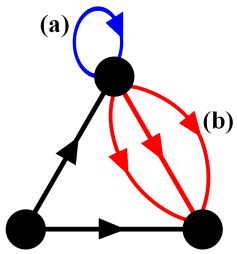

Figure 1: Three-noded graph illustrating **(a)** a self-loop, and **(b)** multiple edges.

continues until the desired number of valid edges has been reached. Building the list of weights is trivial, and it is constructed once the start and end arrays have been built.

2.1.2 Kernel 1 - Graph Construction

The purpose of the first timed kernel is to convert the original edge list data structure into a useable format for the remaining kernels. The graph may be represented in any format, but cannot be altered after its construction in Kernel 1.

Both the HPCS code and our GPU implementation use a node-to-node (N2N) data structure. The N2N structure uses two arrays - a shorter list of pointers, and a longer list of children. In the pointer list, the entry at position p is a location q in the child array; the location where parent p's children are stored. The pointer list is 'number of nodes' long, since each node in the graph can be treated as a parent, and the child list is 'edges' long, because it stores all the end nodes from the original edge list. A simple way to think of building N2N, is by sorting the original list according to start node. In this case, the list of start nodes consists of large groups of 1's, 2's, 3's, etc. This array is condensed into a list of *where* the 1's, 2's and 3's - and hence, their children - begin. An illustration of this conversion is shown in Fig. 2.

Both versions of the code perform this conversion in three steps. First, the number of children for each node is counted - this is known as a node's *degree*. A simple loop increments counters corresponding to edge start nodes. Next, this list of degrees is converted into the pointer list, using an operation known as a *parallel scan*. The scanned pointer value for a node is the sum of degree values for all preceding nodes. The final step builds the child list using the newly created

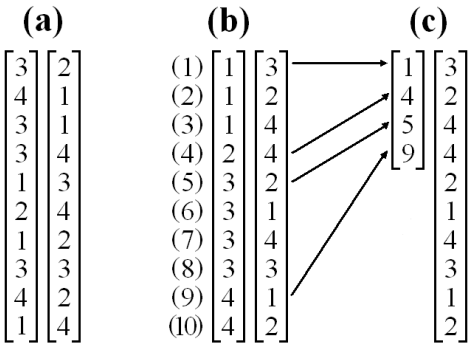

Figure 2: Illustration of conversion from **(a)** the original edge list, to **(b)** list sorted by start node, and finally to **(c)** N2N.

pointer list. A loop finds each edge's start node and the corresponding pointer, and inserts the edge's end node at that location. Care must be taken to offset individual end node entries from their start node's pointer to ensure unique locations in the child list.

It is also worth nothing that Kernel 1 not only builds the standard N2N structure, which follows the true direction of the edges ($N2N_{out}$), but it also builds a second N2N structure that goes "against the grain" ($N2N_{in}$). This second version has proven useful for the parallel implementations of Kernels 2 and 4.

2.1.3 Kernel 2 - Classify Large Sets

Kernel 2 searches through edge weights and picks out those with the largest possible value. This max-weight value is known a priori. In the case of Kernel 2, using the N2N data structure has actually made this task more challenging. In the original edge list, weights are paired with data for both nodes. In the N2N structure, weights are only tied to one node, and finding the second node is non-trivial.

For the Kernel 2 algorithm, threads search the weight list in the $N2N_{in}$ structure for max-weight values. When one is found, the corresponding node (the edge's start node) is saved. Using this start node and the $N2N_{out}$ pointer list, the GPU finds the location of that node's children, and quickly searches that limited region for a max-weight edge. When the weight is found, the corresponding node (the edge's end node) is paired with its start node.

Due to the benchmark's specifications regarding

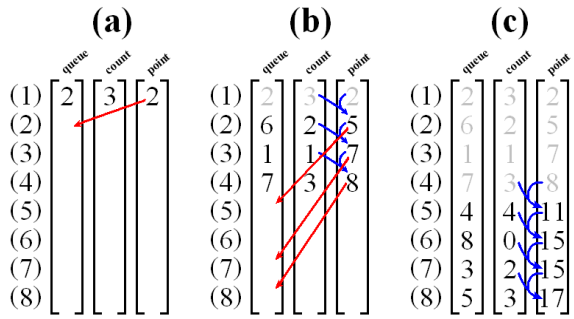

Figure 3: Illustration of queue construction showing **(a)** the first node, its number of children, and the location where they will be inserted. Part **(b)** shows the second generation and the scanned count array. Part **(c)** shows a portion of the third generation.

number of edges and edge weight distribution, there are on average only eight max-weight edges in a graph, regardless of SCALE. As a result, this kernel has relatively little work to do, and is easily the fastest of the four timed tasks.

2.1.4 Kernel 3 - Graph Extraction

The third kernel of SSCA 2 is designed to construct subsets (or subgraphs) of the original graph, using the edges found by Kernel 2 as starting points. Kernel 3 starts at a max-weight edge, and moves out a user-specified number of levels from it.

Kernel 3 produces a list of nodes which represent the members of the subgraph. This *queue* is built in sections, which are filled as the code steps out to each new level of the subgraph. The code reads parent nodes from the current level, and fills in their children in the next level. The most challenging part of parallelizing this code is determining where to insert children into the queue. Each thread needs its own dedicated space in the queue, and must know where that space begins. This is achieved is by keeping *count* and *point* arrays which correspond to the queue. When a node is added to the queue, its number of children is recorded in the count array. After each level is filled, the count array is scanned into the point array, which then contains the queue locations for the children of corresponding nodes. This process is illustrated in Fig. 3.

An important feature of the queuing algorithm is its ability to discern whether or not a node

is already in the subgraph. Before children are added, the code checks an array of nodal flags to make sure they have not previously been added to the queue. If a node has already been visited, its reserved spot in the queue is left empty, and a zero is entered in the count array. This eliminates extra queue entries, and therefore extra work in the next queue level.

2.1.5 Kernel 4 - Betweenness Centrality

The goal of Kernel 4 is to determine which nodes have the highest connectivity, or betweenness centrality (BC). Given a particular starting node, partial BC scores are calculated for all other nodes. This process is repeated using a subset of nodes as starting points. A node's final BC value is the sum of all its partial scores. This is easily the most computationally-taxing portion of the SSCA 2 code.

As proposed by Brandes [6], and Bader and Madduri [3], the BC algorithm consists of two main steps. The outsweep assigns nodal values of distance to and number of shortest paths back to the start node. This process is conducted in the same manner as Kernel 3, moving out visiting new generations, level by level. The only differences are nodal values now need to be assigned in addition to just filling the queue, and the algorithm must continue for as long as it takes to visit *all* nodes. Assigning the depth and shortest path values is a trivial addition to the queuing code, and since the algorithm only permits each node to appear in the queue once, the last level of children will be empty, and the algorithm will stop on its own.

The insweep works through the queue backwards, level by level. As the code moves in along shortest paths (from child to parent), the child's BC and shortest paths values are used to update the BC score for the parent. The $\mathrm{N2N}_{in}$ data structure - carefully marked during the outsweep - is used to determine if parents lie along shortest paths. In equation 1, ν and ω represent parent and child nodes, respectively. A node's temporary BC score (reset after each outsweep/insweep pair) is represented by δ, while σ is a node's shortest path value.

$$\delta[\nu] \leftarrow \delta[\nu] + \frac{\sigma[\nu]}{\sigma[\omega]} \cdot (1 + \delta[\nu]) \tag{1}$$

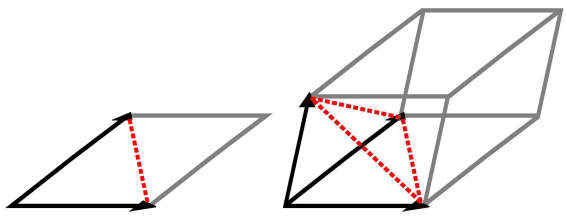

Figure 4: Illustration of face area and cell volume calculations. The face and cell are defined by the black and red lines.

2.2 Mesh Smoothing

The bulk of the smoothing algorithm involves finding the movement, or residual, for each node in an unstructured tetrahedral mesh. To find a cell's contribution to its nodes' residuals, we minimize the derivative of a cellular deformation function with respect to the position of node n, represented by $\vec{x_n}$. The deformation statistic used in the code is a cell's average edge length squared, divided by cell volume. The function and its derivative are shown below in equations 2 and 3, respectively. In the equation 3, the $\vec{L_i}$'s represent lengths of edges touching node n, and $\vec{n}$ represents the outward-facing normal for the face opposite node n. The outer summation is over all cells touching the node.

$$F = \frac{1}{6V} \sum_{edges} L^2 \tag{2}$$

$$\frac{\partial F}{\partial \vec{x_n}} = \sum_{cells}^{node} \frac{1}{6V} [\alpha + \beta] \tag{3}$$

where

$$\alpha = 2\left(\vec{L_a} + \vec{L_b} + \vec{L_c}\right), \quad \beta = \sum L^2 \left(\frac{2\vec{n}}{6V}\right)$$

This optimization procedure attempts to minimize edge lengths while maximizing cell volume. Cross products of specific edge pairs yield values two times the face normals ($2\vec{n}$). The dot product of these normals with a third edge is analogous to cell volume, but produces a quantity six times the true value ($6V$). Graphical representations of these "oversized" values can be seen in Fig. 4.

2.3 Mesh Partitioning

The current serial and parallel algorithms build partitions using a size-weighted Voronoi-type method. Partitions are assigned starting points within the mesh. For a particular node, a loop over partitions finds the distances between the node and each partition point. When all partitions have been checked, nodes are assigned to the closest one. Each partition essentially grows outward from its starting point, "collecting" nodes along the way. This process is iterative, and adjustments are made to partition positions and weights until the partitions are well-balanced in terms of size (number of constituent nodes). Starting points are re-set at each iteration step, as the average location of all nodes currently in the partition. Similarly, at each step, large-size partitions are given higher weights to lower their maximum radius, and vice versa.

2.4 Parallel Programming Challenges

One of the problems frequently encountered during the parallelization of these programs is when multiple GPU threads attempt simultaneous reads or writes to the same memory location. This never occurs in serial code since only a single core is active, working on a single data element. However, in parallel, this happens quite often. For example, in the smoothing code, even though all threads work on unique cells, many share common nodes. Trying to update these residuals simultaneously can lead to overwriting, and incorrect results.

The simplest solution to this problem is atomic functions, which are built into the CUDA API. These perform a simple locking procedure to serialize threads attempting concurrent reads/writes. These were critical in SSCA 2 - particularly for finding nodal degrees in Kernel 1. However, these only work with integer values, and are therefore, somewhat limited.

In Kernel 4, to attain the same level of control adding floating point values to BC scores, we multiplied decimal values by large powers of ten, and typecast them as integers. These integers were added atomically, and the sum was divided by the same power of ten. This too is imperfect, since some precision is lost in the conversion.

In the smoothing and partitioning codes, in some cases we store sums on a per-thread basis, then add thread sums later. For residuals, we calculate and store contributions by cell, then retrieve and add them from a nodal perspective later. This is equivalent to recasting all scatter operations into gather operations. It requires additional connectivity pointers, and reduces the code performance by necessitating additional memory reads/writes.

3 Results

3.1 SSCA 2

Work on SSCA 2 has been completed, and the parallel code uses MPI to run on up to four graphics cards simultaneously. All comparisons to serial or CPU versions refer to the HPCS code as the optimized benchmark. Fig. 5 show speedups relative to the CPU, for SCALE sizes of 16 through 21. A speedup of 10 indicates the GPU code ran ten times faster than the CPU. Plots include data for the CPU, a single GPU, two GPUs and four GPUs (CPU, GPU x1, GPU x2 and GPU x4, respectively). All trials were run on Orion, a machine with an AMD Phenom II Quad-Core CPU (3.2 GHz) with 8 GB of RAM, and four NVIDIA GeForce GTX 295 GPUs.

Several important conclusions can be drawn from this data. First and foremost, *GPU performance is better for larger problem sizes.* Throughout this project, the GPU has only been able to outperform the CPU consistently for very large SCALEs. As noted, the codes were tested on a machine with state of the art CPU components. Even with some of the best commercial GPUs available, it was difficult to match the output of a highly optimized serial processor. Still, it is clear that there exists a point where problems become so large that the CPU cannot compete, and the GPU wins. As problem sizes progress farther beyond this point, the performance gap continues to grow. The expectation is that this trend would continue beyond the current testing limit of SCALE 21.

A second conclusion is that *MPI carries a high cost.* While MPI is extremely useful, it can also slow down a program considerably. This was particularly evident in the results for Kernel 1. On a single GPU, the code achieved a modest speedup of around 2-3x. When a second GPU was introduced - making MPI communication

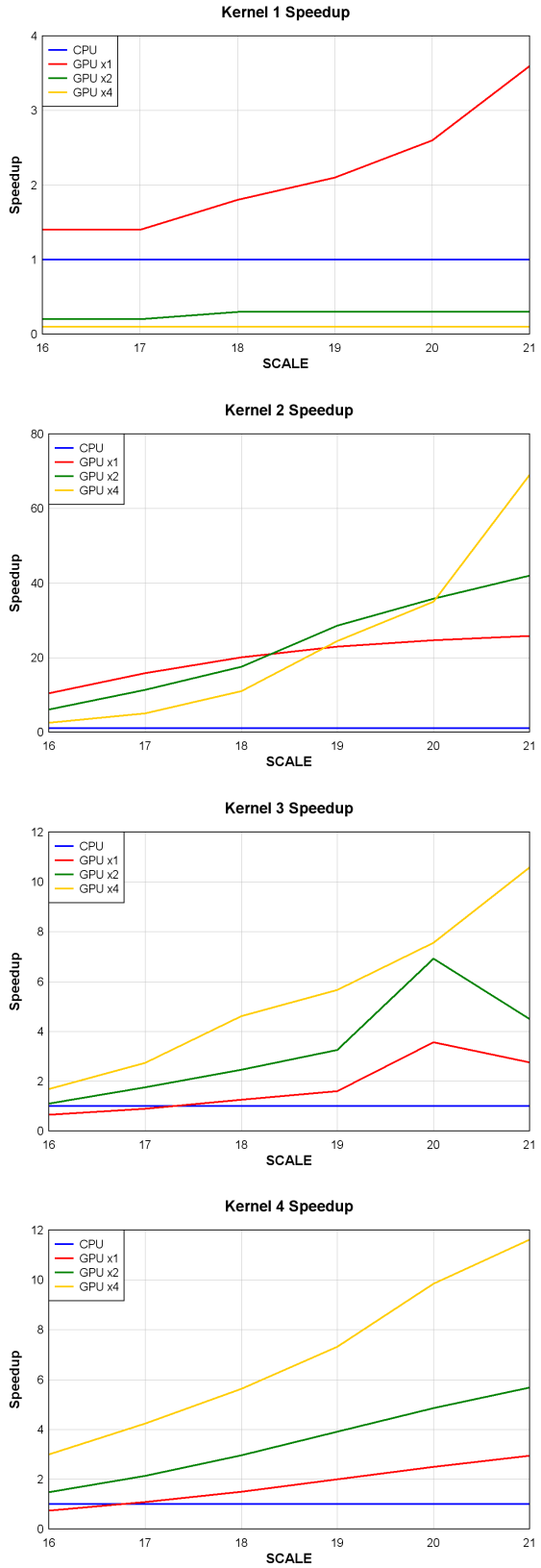

Figure 5: Parallel speedups over serial code for Kernels 1 - 4.

necessary - speedups went from three times faster, to three times slower. Going to four GPUs decreased performance by another factor of three. Compared to the other kernels, Kernel 1 requires the most MPI communication by far. At the end of this kernel, the N2N structure is distributed to all processes, and each uses its own full copy for the remainder of the program. A way around this issue is to attempt to "hide" MPI and CUDA communication using non-blocking MPI sends and receives and asynchronous CUDA memory copies. These operations are then performed in the background while other tasks are worked on at the same time.

Third, the eye-popping speedups from Kernel 2 highlight a simple yet important fact. These results showcase the *tremendous memory bandwidth of the GPU*. This is by far the simplest kernel, since it is composed almost entirely of memory reads. Because there are so few max-weight edges, the vast majority of threads will read a weight from memory, test it, and move on. Only in extremely rare cases will the code need to take further action. The simplicity of this kernel makes for an excellent comparison between the CPU and GPU at the hardware level. For computationally-sparse tasks, the GPU is capable of impressive performance gains, like the 20-60x that were regularly seen from this portion of the code.

Finally, the results for Kernels 3 and 4 show *code performance scales with number of processors used*. While this may seem to be obvious, it is not always an easy relationship to attain. Inefficient algorithms, overhead for memory copies and kernel invocations, and restrictions on problem size can all erode the expected computational efficiency for the GPU. These results show that with large SCALE sizes, doubling processing power does indeed halve the required time almost exactly. It is not surprising that this trend is apparent on the two most computationally-intense kernels. Often, the biggest obstacle in reaching this relationship is the amount of work per processor. Without enough work, the GPU simply cannot reach its full potential. In the case of the last two kernels, even when additional graphics cards are introduced, there is so much work to be done, there is still plenty to go around.

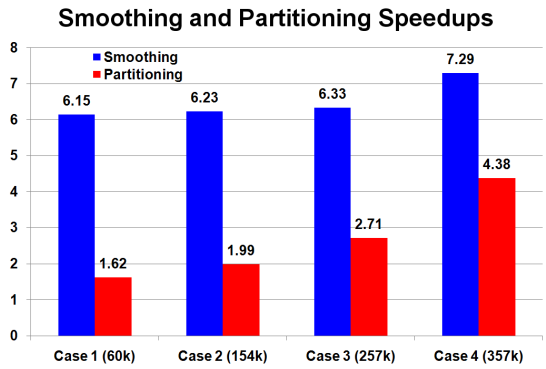

Figure 6: Parallel speedups over serial code for smoothing and partitioning.

3.2 Smoothing and Partitioning

At present, the parallel smoothing and partitioning codes are functioning on a single GPU, and some preliminary testing has been completed. Fig. 6 shows speedups of the parallel codes over the serial versions for four test meshes (60k, 154k, 257k, 357k cells, respectively).

In the case of smoothing, all test cases were non-trivial (node positions were altered by the smoothing algorithm), and maximum errors in the parallel solutions never exceeded 0.5%. For the vast majority of nodes, smoothed GPU nodal coordinates were identical to the CPU solution to six digits. The parallel partitioning results were also highly accurate for all test cases. Maximum errors in partition sizes did not exceed 0.002%, and partition coordinates were accurate to four digits. As seen with the SSCA 2 results, GPU performance shows a strong correlation to problem size.

4 Future Work

4.1 SSCA 2

Currently, the parallel code requires all GPUs to have access to the entire graph data structure, meaning the whole N2N list must be loaded into each card's memory. This means problem size is bound by the amount of memory on each GPU (hence, our SCALE limit of 21). Using more GPUs does not decrease the amount of required memory per card. This problem is most apparent for Kernels 3 and 4, since they require the most memory.

To enhance the scalability of the code, the plan is to give each process a portion of the N2N list, and use MPI to cooperatively build the subgraph or queue. Now, the addition of more GPUs will reduce the required memory per card, meaning problem size will only be limited by the number of cards used.

Implementing this new distributed N2N list requires making modifications to all the kernels. Currently, these changes have been finished for Kernels 1 and 2, but have not been put into place in other parts of the code. The reduced amount of MPI communication in Kernel 1 has translated to better speedups, yet the addition of only a few MPI commands to Kernel 2 has caused drastic reductions in performance. The expectation is that the large amounts of communication required for Kernels 3 and 4 will lead to very poor results relative to the current CPU implementation. However, it is important to keep in mind that using the same distributed data structure and algorithm in a CPU code would likely perform equally poorly. The next step for SSCA 2 is to get the new data structure working in both the CPU and GPU codes.

4.2 Smoothing and Partitioning

The main priority for both the parallel smoothing and partitioning codes is to run more robust test cases. Ideally, the codes should be tested on larger meshes, with more complex geometries. With the largest test case containing only about 360k cells, the codes should really be tested on meshes containing closer to one million cells. Furthermore, the current test meshes are all geometrically simple, as they are either solid cubes, or rectangular prisms. In the future, the intent is to run tests on meshes of more complex - and perhaps more practical - structures.

Another goal for this project is to run the code on multiple GPUs using MPI. With up to four cards, the parallel graph analysis code showed impressive performance gains over the single-GPU implementation. The expectation is that this same trend can be demonstrated with the smoothing and partitioning algorithms.

Acknowledgements

The authors would like to thank the US Department of Energy and Oak Ridge National

Laboratories for their generous funding on this project.

References

[1] N. Acikgoz. *Adaptive and Dynamic Meshing Methods for Numerical Simulations*. Doctor of philosophy, Georgia Institute of Technology, School of Aerospace Engineering, Atlanta, Georgia, mai 2007.

[2] D. A. Bader, J. Feo, J. Gilbert, J. Kepner, D. Koester, E. Loh, K. Madduri, B. Mann, T. Meuse, and E. Robinson. *HPC Scalable Graph Analysis Benchmark*. HPCS, 1.0 edition, 2009.

[3] D. A. Bader and K. Madduri. Parallel algorithms for evaluating centrality indices in real-world networks. In *Parallel Processing*, pages 539–550, Columbus, Ohio, apr 2006.

[4] T. J. Baker. Mesh adaptation strategies for problems in fluid dynamics. *Finite Elements in Analysis and Design*, 25(3-4):243–273, jan 1997.

[5] C. L. Bottasso, D. Detomi, and R. Serra. The ball-vertex method: a new simple spring analogy method for unstructured dynamic meshes. *Computer Methods in Applied Mechanics and Engineering*, 194(39-41):4244–4264, oct 2005.

[6] U. Brandes. A faster algorithm for betweenness centrality. *Journal of Mathematical Sociology*, 25(2):163–177, 2001.

[7] C. Degand and C. Farhat. A three-dimensional torsional spring analogy method for unstructured dynamic meshes. *Computers and Structures*, 80(3-4):305–316, feb 2002.

[8] C. Farhat. A simple and efficient automatic fem domain decomposer. *Computers and Structures*, 28(5):579–602, 1988.

[9] C. Farhat, C. Degand, B. Koobus, and M. Lesoinne. Torsional springs for two-dimensional dynamic unstructured fluid meshes. *Computer Methods in Applied Mechanics and Engineering*, 163(1-4):231–245, sep 1998.

[10] C. Farhat, S. Lanteri, and H. D. Simon. Top/domdec - a software tool for mesh partitioning and parallel processing. *Computing Systems in Engineering*, 6(1):13–26, 1995.

[11] G. Karypis and V. Kumar. *METIS Manual*. University of Minnesota, Department of Computer Science, 4.0 edition, 1998.

[12] G. Karypis, K. Schloegel, and V. Kumar. *ParMETIS Manual*. University of Minnesota, Department of Computer Science, 3.1 edition, 2003.

[13] S. Menon and J. B. Perot. Implementation of an efficient conjugate gradient algorithm for poisson solutions on graphics processors. In *2007 Meeting of the Canadian CFD Society*, pages 468–481, Toronto, Ontario, jun 2007.

[14] NVIDIA Corporation. *NVIDIA CUDA Programming Guide*, 2.2.1 edition, 2009.

[15] J. B. Perot and S. Menon. Cfd computations on multi-gpu configurations. In *60th Meeting of the American Physical Society, Division of Fluid Dynamics*, pages 622–632, Salt Lake City, Utah, nov 2007.

[16] C. Walshaw and M. Cross. Parallel optimisation algorithms for multilevel mesh partitioning. *Parallel Computing*, 26(12):1635–1660, nov 2000.

[17] C. Walshaw, M. Cross, R. Diekmann, and F. Schlimbach. Multilevel mesh partitioning for optimizing domain shape. *International Journal of High Performance Computing Applications*, 13(4):334–353, 1999.

[18] C. Walshaw, M. Cross, and K. McManus. Multiphase mesh partitioning. *Applied Mathematical Modelling*, 25(2):123–140, dec 2000.

[19] D. Zeng and C. R. Ethier. A semi-torsional spring analogy model for updating unstructured meshes in 3d moving domains. *Finite Elements in Analysis and Design*, 41(11-12):1118–1139, jun 2005.