Parallel Mesh Adaptation and Graph Analysis Using Graphics Processing Units

Timothy P. Mcguiness
University of Massachusetts Amherst, tmcguine@engin.umass.edu

PARALLEL MESH ADAPTATION AND GRAPH ANALYSIS USING GRAPHICS PROCESSING UNITS

A Thesis Presented

by

TIMOTHY P MCGUINESS

Submitted to the Graduate School of the
University of Massachusetts Amherst in partial fulfillment
of the requirements for the degree of

MASTER OF SCIENCE IN MECHANICAL ENGINEERING

February 2011

Mechanical and Industrial Engineering

PARALLEL MESH ADAPTATION AND GRAPH ANALYSIS USING GRAPHICS PROCESSING UNITS

A Thesis Presented

by

TIMOTHY P MCGUINESS

Approved as to style and content by:

____
J. Blair Perot, Chair

____
Stephen de Bruyn Kops, Member

____
Yahya Modarres-Sadeghi, Member

____
Donald L. Fisher, Department Head
Mechanical and Industrial Engineering

ACKNOWLEDGMENTS

I have to thank my parents and my brother, for always being there. I need to thank Jaimee, for being my best friend. I want to thank Ali, for always answering my CUDA programming questions. I *must* thank Professor Perot, not only for his guidance and patience, but more importantly for giving me a job! Lastly, I can’t forget to thank Mike Martell, Dnyanesh Digraskar, Kshitij Neroorkar, Kyle Mooney, Nat Trask, Mike Colarossi, Shivasubramanian Gopalakrishnan, Sandeep Menon, and Tom Furlong, for making the lab and graduate school such a blast.

ABSTRACT

PARALLEL MESH ADAPTATION AND GRAPH ANALYSIS USING GRAPHICS PROCESSING UNITS

FEBRUARY 2011

TIMOTHY P MCGUINESS, B.S., PENNSYLVANIA STATE UNIVERSITY

M.S.M.E., UNIVERSITY OF MASSACHUSETTS AMHERST

Directed by: Professor J. Blair Perot

In the field of Computational Fluid Dynamics, several types of mesh adaptation strategies are used to enhance a mesh's quality, thereby improving simulation speed and accuracy. Mesh smoothing (r-refinement) is a simple and effective technique, where nodes are repositioned to increase or decrease local mesh resolution. Mesh partitioning divides a mesh into sections, for use on distributed-memory parallel machines. As a more abstract form of modeling, graph theory can be used to simulate many real-world problems, and has applications in the fields of computer science, sociology, engineering and transportation, to name a few. One of the more important graph analysis tasks involves moving through the graph to evaluate and calculate nodal connectivity. The basic structures of meshes and graphs are the same, as both rely heavily on connectivity information, representing the relationships between constituent nodes and edges. This research examines the parallelization of these algorithms using commodity graphics hardware; a low-cost tool readily available to the computing community. Not only does this research look at the benefits of the fine-grained parallelism of an individual graphics processor, but the use of Message Passing Interface (MPI) on large-scale GPU-based supercomputers is also studied.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1 Motivation

The cost of conducting physical experiments for a fluid dynamics problem is often prohibitive. The amounts of time and physical resources required can be large. Since the mid-50s, engineers have been using computer simulations, rather than physical ones, to obtain answers to fluid dynamics problems [1]. The field of computational fluid dynamics (CFD) has grown rapidly, and although it has alleviated some of the costs associated with physical experiments, CFD has cost issues of its own.

In order to keep numerical errors small and to ensure accurate results, CFD codes must use small timesteps and small spatial discretizations. These are often unfortunately quite small, leading to dramatically long simulation times. Even for physical phenomena that take only seconds to occur, a corresponding simulation may take several days to calculate. As a result, there is a constant emphasis on improving the numerical methods used to solve these problems, and also on improving the instruments which are used to perform the calculations.

The performance level of the modern computer has been improving for quite some time. Processors speeds have been getting faster, memory sizes have been growing larger, and computational precision has become finer. Moore’s law, posed in the mid 60s, states that the number of transistors on a processor is roughly doubling every two years [2]. While the state of computer technology has in fact followed this trend for some time, it is now being discovered that this actually may not continue for much longer. Physical limitations on processor and wire sizes [3], thermal noise [4], and current leakage [5] are all posing new challenges in making faster, more efficient processors. Furthermore, even with the introduction of new and improved processors, memory bandwidth is proving now to be a limiting factor in many modern scientific computing applications [6].

As a result, the computing community has begun a shift away from serial computing (the use of a single processor core) toward parallel processing (the use of multiple cores). 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 yet effective way to provide parallelization to the scientific programmer. Due to their non-standard processor architecture and superior memory bandwidth [7], graphics processing units, or GPUs, are quickly emerging as some of the best processors available today in terms of floating point operations per second (FLOPS) per dollar. General purpose computing on graphics processing units (GPGPU), is a new field which has caught the eye of many members of the engineering community as a way to achieve performance increases for CFD simulations.

1.2 Background

1.2.1 Meshes and Adaptation

For CFD problems, it is common practice to discretize the domain, so that the governing differential equations can be solved. This discretization is known as a mesh, which consists of distinct points (nodes), their connections (edges, faces) and the subdomains themselves (cells) [8].

In general, smaller subdomains – or cells – are better able to model physical phenomena or areas of interest [8]. At the same time, computational cost is clearly proportional to the number of cells used. It is obvious that an ideal mesh would consist of non-uniform cells which are highly refined in the necessary locations, but coarser in others to help reduce simulation time. Additionally, in problems where the simulated fluid flow or solid bodies undergo significant movements, deformations, or other changes, a given mesh – particularly if it is already non-uniform – may be inadequate for modeling these different states. Finally, the user may not know what an ideal mesh looks like before a simulation is run [9]. These facts lead to the conclusion that some form of mesh modification is desirable to help strike a balance between benefits of non-uniformity and the issues raised by domain changes.

There are two general fixes for this problem. One of which is to rebuild a new mesh from scratch when necessary, however this approach is costly, and the overhead associated with each generation step remains constant. By comparison, the practice of modifying the existing mesh can be much faster, particularly when few changes are needed [10]. It is this second form of mesh adaptation that has gained significant popularity in the CFD community, and it has manifested itself in several forms.

1.2.1.1 Mesh Smoothing

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. The spring analogy is one of the most common ways of describing how r-refinement works [11]. In this analogy, the edges of the mesh are replaced with springs, each with a spring constant inversely proportional to the target length. As the nodes that define an edge move closer than this desired length, the edge becomes more resistant to compression. One important benefit of r-refinement is that mesh topology remains the same [12, 13]. Only cell volumes and node locations change, while connectivity and the number of nodes in the mesh do not. This eliminates the issue of altering the existing data structure, and also means the computational cost remains the same from iteration to iteration. Additionally, in a parallel architecture, this means that the amount of work for each processor remains constant, and load balancing is a one-time cost.

One of the biggest problems with smoothing schemes is the creation of poor-quality elements [13-17]. This is often caused by rapid mesh deformation, when the movement of nodes is large relative to element size. Low quality elements with small cell volumes, and invalid mesh elements with zero or negative volumes can be created in these situations. Mesh deformation occurs at these high speeds in many real-world CFD examples, making this a very important problem to address.

One approach to solving this problem is to try and restrict cell movement in the direction of collapse. Farhat [14] and Degand [15] proposed the torsional spring method. Additional springs are added at each vertex of the triangular cell, which serve to limit the degree of the angle formed by the two edges. While this technique works well in 2D, the extension to three dimensions is cumbersome and inefficient [16, 17]. A similar strategy is given by Zeng and Ethier [16]. In their semi-torsional spring analogy, extra spring values are added along edges, inversely proportional to the degree of the opposing angle. Not only is this method more efficient, but the translation to 3D domains is much simpler. Bottasso [13] and later Acikgoz [17] propose a method in which each node in the mesh is enclosed within the rough circle (2D) or sphere (3D) created by its closest neighboring vertices. This ball-vertex method adds additional springs from the central vertex, to the opposing edge (2D) or face (3D). Figure 1.1 shows the location of these perpendicular springs, which become stiffer as the cell collapses.

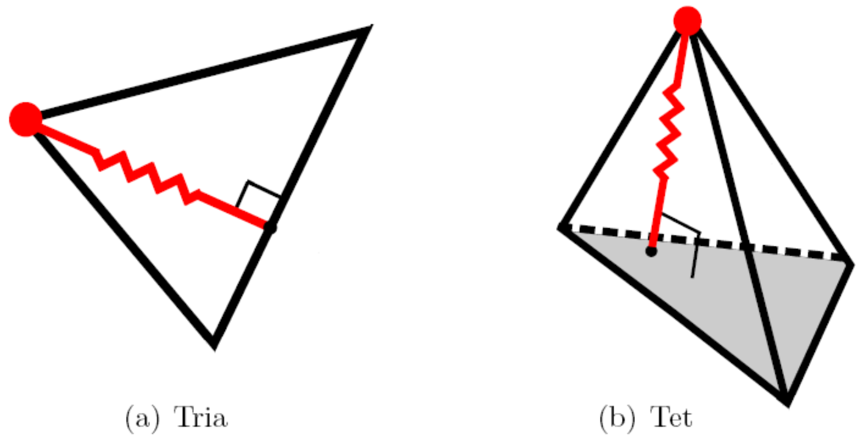

Figure 1.1. Illustration of ball-vertex springs for (a) triangle and (b) tetrahedron. Reproduced from Acikgoz [17].

Figure 1.1. Illustration of ball-vertex springs for (a) triangle and (b) tetrahedron.

1.2.1.2 Edge Reconnection

Edge reconnection, or swapping, compliments mesh motion by optimizing the connectivity, while leaving node locations unchanged. Reconnection can also improve cell quality by increasing the minimum angles of cells with high aspect ratios [18]. In other words, reconnection prefers stout cells over flat, thin ones. It is well known that elements with small angles and volumes, sometimes called sliver cells, can lead to numerical problems [18, 19]. Even when nodes are well-spaced, this does not guarantee the prevention of sliver cells [19, 20]. In two dimensions, when considering two adjacent triangles, their non-shared edges form a quadrilateral which as shown in Figure 1.2. This quadrilateral can be divided into the constituent cells in two unique ways. If the dividing edge is the longer diagonal, a reconnection scheme will replace it with the edge formed by the shorter diagonal. It is worth noting that when programming this type of adaptation, certain edges cannot be altered. For instance, if the quadrilateral is concave, the diagonal cannot be swapped, since the new elements would be invalid [10], as seen on the next page, in Figure 1.3. This 2D approach can be applied easily in three dimensions, swapping the common faces of neighboring tetrahedra [21]. In both cases, the resulting cells will have smaller aspect ratios, and in turn, smaller minimum angles.

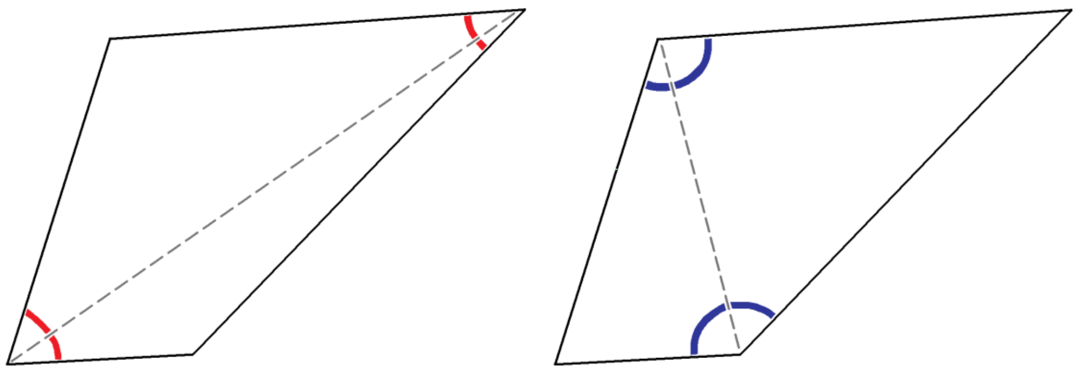

Figure 1.2. Example of edge reconnection for adjacent 2D cells. When divided along the quadrilateral's shorter diagonal, the cells are of higher quality, thanks to their greater minimum angles.

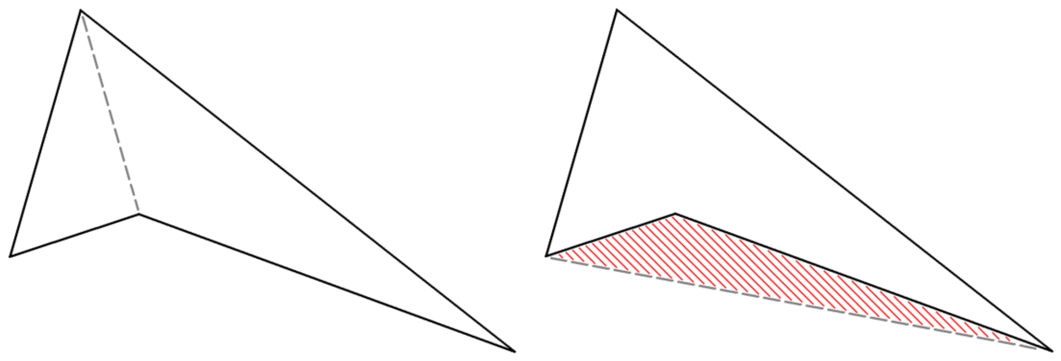

Figure 1.3. Illustration of how edge reconnection can cause invalid elements in some situations. If edge reconnection is performed on a concave quadrilateral, it will result in the inverted cell shown in the second image.

1.2.1.3 Mesh Enrichment

One very popular type of mesh adaptation used in the finite element method, is h-refinement, or h-enrichment. Error values are calculated for the mesh at each refinement stage, and cells with values above a specified threshold are split. As stated, smaller cell size leads to a reduction in cell error, thus improving the accuracy of the simulation. Splitting cells requires adding nodes to the mesh, and although there are different strategies for determining the location of these new points, edge bisection is by far the most common [10, 22, 23]. For quadrilaterals and hexahedra, bisection of cell edges and faces leads to four new cells in 2D, and eight cells in 3D. With triangles and tetrahedra, nodes can be added to one or more edges, and the cell is divided appropriately. One important benefit of h-refinement is that it has been shown to either sustain or improve the quality of the original mesh [12]. On the other hand, a disadvantage of this method is that it can lead to large changes in element size over short distances, as shown in Figure 1.4. If these changes occur too rapidly, they can be problematic for PDE solvers, and can generate poor solutions.

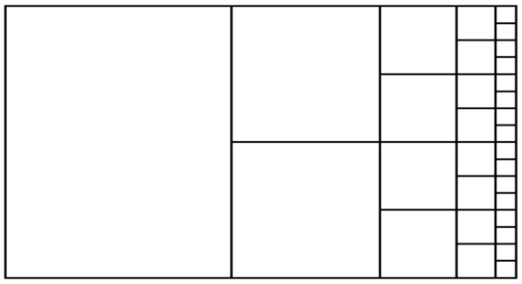

Figure 1.4. Large variations in cell size in nearby cells, created by h-refinement.

Frequently when mesh refinement is used, its counterpart – coarsening – is also employed. Similar to the upper error bounds used for enrichment, lower bounds are put to use for mesh coarsening. If it becomes apparent that mesh resolution is higher than necessary, edges can be removed from the mesh, combining cells and reducing the computational workload. The technique of edge collapse is often used for this process. The meshes shown in Figure 1.5 were adapted using a combination of refinement, coarsening, smoothing and reconnection.

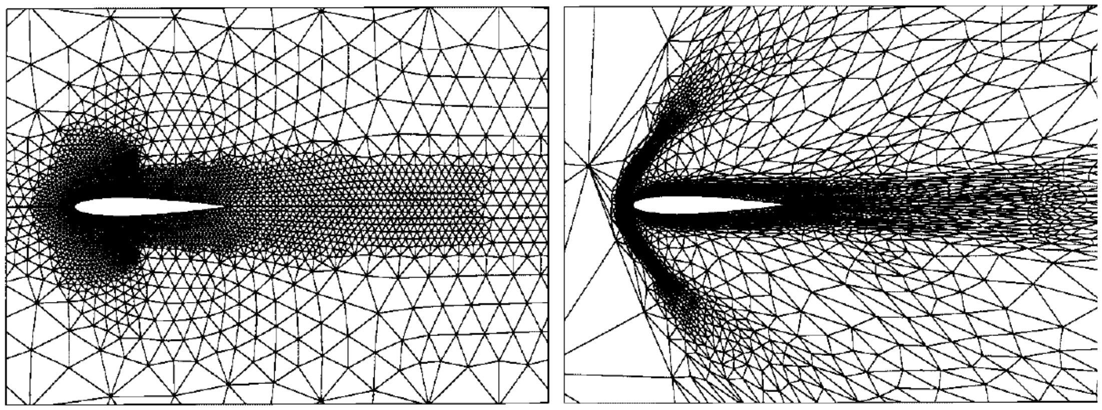

Figure 1.5. Comparison of initial and refined meshes around an airfoil. Reproduced from Dompierre *et al*. [9].

1.2.1.4 P-refinement

Another mesh adaptation technique is p-refinement. In the finite element method, the solution is approximated using a piecewise polynomial in each element in the mesh. Solution accuracy is improved by increasing the degree of these polynomials. It has been shown that in many situations, p-refinement converges exponentially; much faster than h-refinement [24, 25]. This is due to the fact that the number of unknowns increases factorially, or faster than exponentially. The reason for this is that p-refinement typically requires the use of fewer cells to obtain similar levels of accuracy. However, since computational time is roughly proportional to number of mesh elements times number of unknowns per element, there is little practical advantage to this method. Because p-refinement does not change a mesh's topology, parallelization is much easier than with h-refinement, and has already been explored by Ghosh and Basu [26].

One important note on p-refinement is that for flows involving discontinuities, higher-order methods usually lead to numerical instabilities, since the solution is not smooth [12]. With the ever-increasing complexity of the flows modeled by CFD codes, this fact should be taken into consideration when using p-refinement. Also, p-refinement methods are often more challenging from a programming perspective [27].

1.2.2 Mesh Partitioning

An important topic when considering parallelization of mesh-based algorithms is the idea of mesh partitioning. To process large meshes using 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 communication between processors [28, 29]. There are several popular methods for achieving an appropriate partition for a given mesh, and they vary in their complexity and effectiveness.

One of the more well-known approaches is multilevel partitioning. In this technique, certain nodes are removed to create a coarse version of the mesh, at which point an initial partition is inserted. The removed nodes are then replaced in the mesh progressively, and the partition is optimized at each step. These algorithms have been researched extensively by Walshaw *et al*. [30-32], and are also the basis for the popular METIS and ParMETIS programs [33, 34]. While these algorithms tend to produce more efficiently load-balanced partitions that require less communication, they are much more difficult to code.

A more simplistic method proposed by Farhat [28] involves building subdomains like a Voronoi diagram. Partitions are initialized with starting nodes, and the algorithm moves outward to neighboring cells or nodes using the mesh's connectivity. Partitions "collect" the closest un-owned elements, until all have been assigned to a subdomain. In a similar manner, partitions can be constructed using cell and node locations, rather than connectivity. While these techniques are easier to implement, they cannot guarantee the same partition optimality that multilevel methods offer.

1.2.3 Graph Theory

The mathematical concept of a graph is not much different from that of a mesh. 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 between them however, is that graphs are more general. While graphs deal strictly with connectivity, meshes add the concept of physical location. In other words, nodes in a graph can be connected to *any* other node, and edge overlap is not an issue. Meanwhile, connectivity for mesh points is restricted to only those which are physically nearby. While the research into graph analysis algorithms has not proven to be directly useful, several of the more general concepts have proven useful for the mesh adaptation schemes. These algorithms will be described in more detail in the following chapter.

1.2.4 General Purpose Computing on Graphics Processing Units (GPGPU)

The architecture of a graphics processor is fundamentally different than that of a standard processor. Graphics processors have been specifically designed to handle arithmetic-intensive, parallel tasks. Figure 1.6 shows how fewer transistors are dedicated to flow control and cache, leaving more for data processing. Modern GPUs boast impressive advantages over standard processors in terms of both memory bandwidth and giga-FLOPS (GFLOPS), and these gaps continue to grow, as seen in Figure 1.7, also on the following page [7, 35]. This makes the GPU an ideal platform for most scientific applications, since they are often memory-access intensive.

The early 2000s saw the development of several high-level shading languages such as Cg [36], HLSL and OpenGL [37], designed to help exploit the power of graphics hardware. Unfortunately, these languages still dealt heavily in graphics-specific concepts such as textures and fragments, making it necessary for programmers to navigate this additional level of abstraction [35]. In late 2006, NVIDIA released their

Compute Unified Device Architecture (CUDA), which provided a much more user-friendly programming environment. Using only a few extensions to the C language, CUDA allows programmers to easily create code for execution on the GPU [7]. Since the release of CUDA, the GPGPU community has grown considerably, branching out into many different scientific areas.

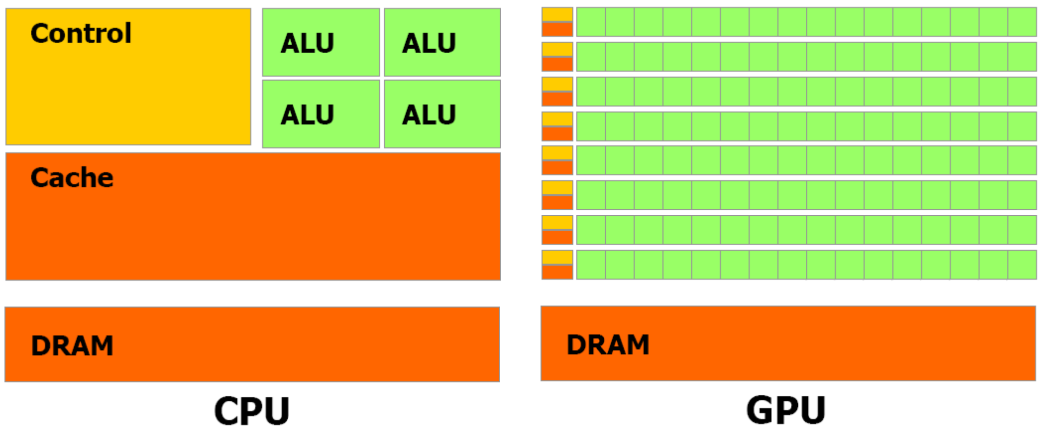

Figure 1.6. Comparison of CPU and GPU chip layout. Reproduced from NVIDIA CUDA Programming Guide, Version 2.2.1 [7].

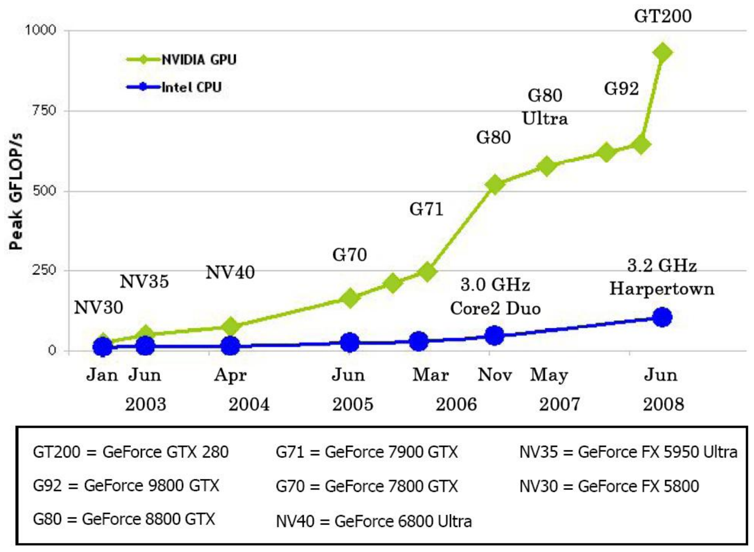

Figure 1.7. Comparison of peak GFLOPS for serial processors and NVIDIA graphics cards. Reproduced from NVIDIA CUDA Programming Guide, Version 2.2.1 [7].

Using graphics hardware for CFD purposes is not a new idea. In 2003, three significant papers were written on the benefits of using graphics cards in this area. Bolz *et al.* [38] implemented both conjugate gradient and multigrid solvers on NVIDIA GeForce graphics cards. At the same time, Goodnight *et al.* [39] also published their work on GPU multigrid algorithms. Krüger and Westermann [40] were able to implement several basic linear algebra subroutines as well as a CG solver, and apply them to solving the incompressible Navier-Stokes equations. It is important to realize that all these demonstrations were conducted on 2D, regular Cartesian meshes, with power-of-two sizes. However, since these initial studies, GPGPU has become more popular within the CFD community, and the scope of research has broadened considerably.

CHAPTER 2

ALGORITHMS

2.1 Smoothing Algorithms

A main goal of this research project is to implement a parallel mesh smoothing algorithm on multiple GPUs. For a given number of smoothing iterations, the code runs a loop over three major steps. These steps consist of calculating a cellular deformation value, determining the amount of movement – or residual – for each node, and updating the existing node positions with these movement values. The iteration loop remains on the CPU, while each of the three sub-steps occurs on the GPU.

2.1.1 Calculation of Cell Deformation Measures

While this first step is not necessary for the smoothing algorithm, it is used to track the code's progress. It calculates a mesh deformation value based on current node positions. The statistic is calculated individually for each cell in the mesh, and cellular values are then summed to find the total value for the mesh as a whole. The *L2V* deformation measure comes from taking the average of the cell's six edge magnitudes (edge lengths, squared), and dividing by cell volume. The formula for this deformation measure is shown below in equation 1.

$$L2V = \frac{1}{6V}\sum_{edges} L^2 \qquad (1)$$

To calculate this value, the algorithm begins by retrieving the positions of the cell's four nodes. The first values to be determined are edge lengths, found by performing a vector subtraction. These edge lengths are then squared to obtain edge magnitudes and a loop over edges sums these, and divides by six.

Face normal vectors are used in finding cell volumes, and while they are not important for this kernel, they *are* necessary for the calculation of nodal residuals in the next step. Cross products of specific edge pairs give these outward face normals, which have a magnitude of twice the face area, shown in the first image in Figure 2.1. Cell volume is found by taking the dot product of one of these face normals with an edge intersecting that face, seen in the second image of Figure 2.1. This dot product gives the volume of the corresponding hexahedra, which is divided by six to find the true volume of the tetrahedral cell. Using these geometric quantities, cellular deformation values are determined and added to the mesh's running totals.

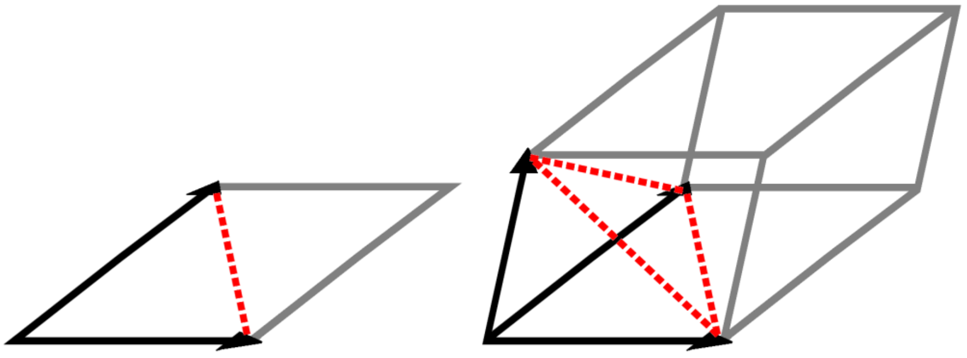

Figure 2.1. Illustration of face area and cell volume calculations. The cell face and volume are defined by the black and red lines. The cross-product of two edges gives twice the face area. Dotting this area with a third edge gives a value that is six times the cell volume.

2.1.2 Calculation of Nodal Residuals

The second step in the smoothing algorithm is the most complex, and finds the amount of movement, or residual, for each node. The goal of the smoothing code is to find the optimal location for each node. In other words, the algorithm attempts to minimize the amount of potential mesh deformation. The following calculations use the *L2V* function, taking its first derivative and letting the system relax to the minimum amount of potential deformation. The initial function is shown in equation 2 below.

$$F = \sum_{cells} \frac{\left(\frac{1}{6} \sum_{edges} L^2 \right)}{V_{cell}} = \sum_{cells} \left(\frac{1}{6} \sum_{edges} L^2 \right) \cdot \left(V_{cell}^{-1} \right) \tag{2}$$

For a tetrahedral cell, both the edge lengths and cell volume are affected by the movement of a particular node, so the product rule is used on these terms to obtain equation 3. The outer summation is over all cells which touch node *n*.

$$\frac{\partial F}{\partial \vec{x}_n} = \sum_{cells}^{node} \frac{1}{6} \left[\left(\frac{\partial \sum L^2}{\partial \vec{x}_n} \right) \left(V^{-1} \right) + \left(\sum L^2 \right) \left(\frac{\partial V^{-1}}{\partial \vec{x}_n} \right) \right] \tag{3}$$

When examining the edge lengths, it is apparent that the position of node *n* ($\vec{x}_n$) contributes to only three of the six edges in equation 4, as shown in Figure 2.2. As a result, the other three edge terms drop out when taking the partial derivative with respect to this node. The partial derivative of the sum of edge magnitudes is shown in equation 5.

$$\sum L^2 = \left(\vec{x}_1 - \vec{x}_4 \right)^2 + \left(\vec{x}_2 - \vec{x}_4 \right)^2 + \left(\vec{x}_3 - \vec{x}_4 \right)^2 + \left(\vec{x}_1 - \vec{x}_3 \right)^2 + \left(\vec{x}_3 - \vec{x}_2 \right)^2 + \left(\vec{x}_2 - \vec{x}_1 \right)^2 \tag{4}$$

$$\frac{\partial \sum L^2}{\partial \vec{x}_n} = 2\left(\vec{x}_n - \vec{x}_a \right) + 2\left(\vec{x}_n - \vec{x}_b \right) + 2\left(\vec{x}_n - \vec{x}_c \right) \tag{5}$$

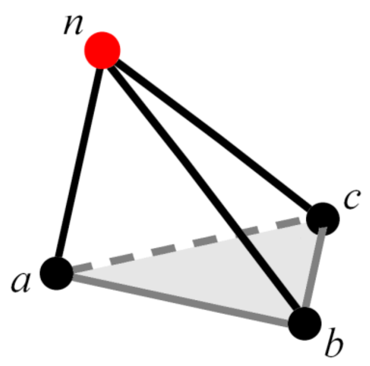

Figure 2.2. Effects of node movement on cell edges. Moving node *n* only affects the three black edges connecting it to nodes *a*, *b* and *c*. The remaining edges and opposing face (gray) remain unchanged.

Since each of the three terms in parentheses in equation 5 represent an edge length, we substitute L_a, L_b and L_c, and simplify to get equation 6. In this equation, these terms are all positive, but depending on the reference node, some of these lengths may be negative due to the orientation of the edge's nodes in equation 4.

$$\frac{\partial \sum L^2}{\partial \vec{x}_n} = 2\left(\vec{L}_a + \vec{L}_b + \vec{L}_c\right) \tag{6}$$

The current implementation calculates cell volume using equation 7. The area term is represented by the opposing face normal, $\vec{n}$, and cell height is calculated from the distance between node *n* and point *m*, an arbitrary point on the opposing face.

$$V = \frac{1}{3} \cdot \vec{n} \cdot \left(\vec{x}_m - \vec{x}_n\right) \tag{7}$$

When taking the derivative of cell volume, because the movement of node *n* has no effect on the geometry of the opposing face, the normal vector can be treated as a constant along with the factor of one third. Thus, the only term affected by the derivative is the one involving $\vec{x}_n$. The derivative is shown in equation 8.

$$\frac{\partial V}{\partial \vec{x}_n} = -\frac{1}{3} \cdot \vec{n} \tag{8}$$

Recalling from the first step that edge cross-products give an area two times larger than the true value, to accurately use these values in the calculations, they must either be divided by two, or the normal term must be doubled in the equations. It is simpler to leave this term undivided, so, correspondingly, the entire right-hand side must be halved to keep the equation balanced. This step is shown below, and the final derivative is presented in equation 9. Applying the chain rule, produces the partial derivative of the *inverse* of cell volume, seen in equation 10.

$$\frac{\partial V}{\partial \vec{x}_n} = -\frac{1}{3}\cdot\left(\frac{1}{2}\cdot 2\vec{n}\right) = -\frac{2\vec{n}}{6} \tag{9}$$

$$\frac{\partial V^{-1}}{\partial \vec{x}_n} = \left(\frac{1}{V^2}\right)\left(\frac{2\vec{n}}{6}\right) = \frac{2\vec{n}}{6V^2} \tag{10}$$

Substituting equations 6 and 10 into equation 3 gives the derivative of the deformation function with respect to the movement of node *n*. Factoring out a $\frac{1}{V}$ term yields equation 11, which is the formula used to calculate a cell's contribution to the residual of node *n*.

$$\frac{\partial F}{\partial \vec{x}_n} = \sum_{cells}^{node} \frac{1}{6V}\left[2\left(\vec{L}_a + \vec{L}_b + \vec{L}_c\right) + \sum L^2\left(\frac{2\vec{n}}{6V}\right)\right] \tag{11}$$

To make use of equation 11, values such as edge vectors, outward face normal vectors and cell volume must be determined. Many of these values (L_a, L_b and L_c; $\sum L^2$) are found using the same methods as those in the deformation measure step, but there are some important distinctions. As stated in the previous section, "oversize" face normals and cell volumes are easy to calculate. In this kernel, these values are *not* divided like they were during the deformation measure step. The oversize values are inserted directly into the $\frac{1}{6V}$ and $\frac{2\vec{n}}{6V}$ terms in equation 11.

As a final step in the residual calculation phase, we find the preconditioner contribution at each node. The preconditioner is used to scale nodal residuals. One method for finding a highly effective preconditioner involves using the second derivative of the deformation function. This can be obtained in the same manner as the first derivative, and gives equation 12. The bracketed term represents a 3 × 3 matrix at each node.

$$\frac{\partial^2 F}{\partial \vec{x}_n^{\,2}} = \sum_{cells}^{nodes}\left[\frac{2\left(\vec{L}_a+\vec{L}_b+\vec{L}_c\right)}{6}\left(\frac{\partial V^{-1}}{\partial \vec{x}_n}\right)+\frac{2}{6V}\frac{\partial}{\partial \vec{x}_n}\left(\vec{L}_a+\vec{L}_b+\vec{L}_c\right)+\right.$$
$$\left.\frac{2\vec{n}}{(6V)^2}\frac{\partial}{\partial \vec{x}_n}\left(\sum L^2\right)+\left(\sum L^2\right)\left(\frac{2\vec{n}}{36}\right)\left(\frac{\partial V^{-2}}{\partial \vec{x}_n}\right)\right]$$

$$\frac{\partial^2 F}{\partial \vec{x}_n^{\,2}} = \sum_{cells}^{nodes}\frac{1}{6V}\left[6+2\left(\vec{L}_a+\vec{L}_b+\vec{L}_c\right)_i\left(\frac{2\vec{n}}{6V}\right)_j+2\left(\vec{L}_a+\vec{L}_b+\vec{L}_c\right)_j\left(\frac{2\vec{n}}{6V}\right)_i+\right.$$
$$\left.2\cdot\left(\sum L^2\right)\left(\frac{2\vec{n}}{6V}\right)_i\left(\frac{2\vec{n}}{6V}\right)_j\right] \qquad (12)$$

In an effort to reduce the complexity and computational workload even further, the GPU code uses an approximation of this preconditioner. This approximation, which is half of the trace of the matrix in equation 12, can be seen in equation 13. This formula produces a scalar quantity at each node.

$$\text{Prec.} = \sum_{cells}^{nodes}\frac{1}{6V}\left(\sum L^2\right)\left(\frac{2\vec{n}}{6V}\right)^2 \qquad (13)$$

2.1.3 Updating Node Positions

The third and final step in the smoothing algorithm is to update the existing node positions with the preconditioned residuals calculated in the second step. The residuals are scaled by the preconditioner matrix, or an approximation, and then added to the current node location. This formula is shown in equation 14.

$$\vec{x}_n^{k+1} = \vec{x}_n^{k} + \left(\frac{\partial^2 F}{\partial \vec{x}_n^2} \right)^{-1} \vec{r}_n^{\,k} \tag{14}$$

2.1.4 Smoothing GPU and MPI Parallelization

Two issues have been encountered in the parallelization of the smoothing algorithm, both stemming from the same problem. The first is easily solved, while the second is more challenging. The first issue is found in the first kernel, calculating the cellular distortion value. Because individual cell values need to be summed to a single value for the entire mesh, at any given time, many threads – if not all – are trying to read, increment and write their contribution to the grand total. When these conflicting threads attempt to access the same memory location simultaneously, overwriting and other negative consequences can result. There are a few ways around this problem, and one of the simplest solutions involves the use of an atomic function, which effectively serializes all operations for conflicting threads. Another solution is to store thread-contributions separately, and sum them carefully later on. Since there are typically a few thousand threads active at any given time, serializing these reads and writes would be quite expensive. On the other hand, allocating a floating point array of a few thousand elements is low-cost in terms of memory, and each thread can freely write its contributions to its own unique location. After the kernel is complete, reducing a few thousand of these thread-sums to a grand total on the CPU is simple and inexpensive. As a result, for this situation, the second option is easily the better choice.

The second problem also involves conflicting threads, and arises in the second kernel when calculating nodal residuals. In this kernel, values are calculated by cell but stored by node. This operation over two different and random data sets is the source of the problem. In parallel, it is possible for two or more threads to be working on cells which share the same node. These threads may attempt to read or write to this node's memory location simultaneously, causing a conflict. Compared to the situation in the first kernel, because the same threads are fighting over more memory locations ('number of nodes' locations, compared to a single location for the distortion sum), using an atomic function here would be less costly. However, testing has shown that even for large meshes with high node totals, requiring considerable amounts of individual storage space, a solution with atomics is still quite inefficient compared to one that stores data individually and retrieves it later on.

This means the second kernel must actually be divided into two GPU loops. The first loop runs over cells, calculating and storing cellular contributions in groups of 16. There are four values per node – a preconditioner, and *x*-, *y*- and *z*-components of the residual – and four nodes per cell, for a total of 16 cellular values. The second loop requires a node-to-cell (N2C) data structure, which is built from the standard cell-to-node (C2N) connectivity at the start of the program. This second loop runs over nodes, with each thread using N2C information to find contributing cells, then extracting and summing the correct set of contributing values.

When breaking up the workload across multiple GPUs, again, the presence of loops over both cells and nodes makes this task difficult. Since cell and node labeling is random, any cell may touch any node. Because of this, there is no way to guarantee a process working on a specific range of cells is also working exclusively on a specific range of nodes, and vice versa. The loops over cells in the first and second kernels require node position data, and the second kernel's loop over nodes requires cell residual data. Even if these loops only operate over a portion of the cell or node list, they need access to the entire set of node or cell data, respectively. This means all data must be communicated across all processes after any step where the workload is distributed. As a result, the decision has been made to break up only the second loop of the residual calculation step, since this step is the most computationally taxing. For this loop, each GPU sums residuals for its own portion of nodes, communicating its final sums once all processes have finished. Again, it is possible to break up other steps as well, but this would require considerably more MPI communication, drastically reducing the overall efficiency of the code.

2.2 Partitioning Algorithms

A secondary goal of the mesh adaptation research is the parallelization of a basic mesh partitioning algorithm across multiple GPUs. The algorithm runs a loop over two main steps which assign nodes to the partitions, and then adjust partition starting points to produce better load-balancing. The main loop remains on the CPU, while the two sub-steps operate on the GPU.

2.2.1 Partition Assignment

The basic algorithm builds partitions in a Voronoi-like manner. Initially, a starting point and weight value are assigned for each desired partition. A loop over nodes finds the distance between the node's coordinates, and those of each starting point. These distances are scaled based on partition weights and when the shortest scaled distance is found, the node is assigned to that partition. Essentially, each of these partitions begins moving out "collecting" nodes that are nearest to it. Nodes that are equidistant from multiple starting points are taken by the partition with the fewest nodes. The second step of the algorithm requires the average position of a partition's constituent nodes. When a node is assigned to a partition in this first step, a partition counter is incremented, and the node's coordinates are added to running totals for the partition. The same read/write conflicts are seen here as in the first kernel of the smoothing code, since multiple threads may attempt to access data for the same partition simultaneously. As with the smoothing code, these sums are stored per thread, and reduced to unique partition values following the kernel's completion.

2.2.2 Partition Adjustment

Just like the smoothing code, the partitioning process is iterative, and adjustments are made until partitions are well-balanced and the solution is close to optimal. The second step in this algorithm loops over partitions, modifying starting points and weights. First, the starting point for each partition is reset to represent the average location of all constituent nodes. Next, partition weights are adjusted, shortening or lengthening their "reach." Those with higher numbers of nodes are given larger weights, effectively lowering their maximum radius. Similarly, partitions with fewer nodes have their weights reduced, extending their reach. Eventually, partition locations and weights adjust until each contains roughly the same number of nodes, and the iteration loop exits.

2.2.3 Partitioning GPU and MPI Parallelization

There is some obvious fine-grained parallelism in the first step of the partitioning algorithm. Each GPU thread performs the calculations for a single node, determining the closest partition. Better yet, unlike the smoothing algorithm, the basic partitioning code maps well across multiple processors. While the algorithm’s two main steps still loop over different data sets, because the number of partitions is always relatively small, the entire second step can be completed individually by each process. Since the first loop is the only significant work being done, nodes are distributed over processors, and each does a share of the work, finding the closest partitions for its set of nodes. There is a need for some non-trivial MPI work in this algorithm, however. Since each process must update all partition starting points and weights in the second kernel, each process must have access to all partition data – namely, node position sums and node counts. As stated earlier, each thread keeps totals for the nodes it has worked on, and these thread-based sums are reduced to GPU-based sums after the first step. When using multiple processors, these process-sums are then added and communicated globally using MPI, so that each process has all necessary information for partition position and weight adjustment.

2.3 Graph Analysis Algorithms

The High Productivity Computer Systems program (HPCS) is a collection of benchmarks established by the Defense Advanced Research Projects Agency (DARPA) and was designed to test the viability of various high-performance systems. Within HPCS, there is a set of benchmarks known as the Scalable Synthetic Compact Applications (SSCA) which are intended to benchmark parallel machines. The SSCA #2 benchmark involves graph analysis. Some basic guidelines have been provided for this code [41], as well as a serial implementation written in C [42]. The following is a brief overview of the different sections of the SSCA2 code and the tasks performed by each.

In general, SSCA2 is a program designed to analyze very large graphs. As previously stated, a graph consists of a set of nodes, which are connected by a set of edges. These graphs are directed and weighted, meaning an edge has a clearly defined start and end node, and it also carries a randomly assigned cost or weight value. The program has four timed sections, as well as an un-timed section that creates the initial graph data from scratch.

Two distinct versions of the code have been created, with the most significant difference arising when multiple processors are used. These two algorithms differ based on the amount of the graph's connectivity structure that is stored locally in the memory of each GPU. Initially, a *full-graph* version of the code was created, where a copy of the graph's entire data structure is owned by each process. Later, a *distributed-graph* version was developed, where each process has direct access to only a small portion of the graph's connectivity information. In each algorithm, the Scalable Data

Generator and Kernels 1 and 2 are basically the same, with the most significant changes coming in Kernels 3 and 4.

2.3.1 Scalable Data Generator

The first portion of the SSCA2 code is the Scalable Data Generator, or SDG. The purpose of the SDG is to produce a tuple list, with each tuple representing an edge. This process is un-timed. Specifically, the user inputs a SCALE value, on which the size of the graph is based. The benchmark states that the number of nodes in the graph should be 2^{SCALE}, and the number of edges should be eight times the number of nodes. For each edge, the tuple will contain three values; start node, end node, and edge weight. Building the edge list is an iterative process, and the SDG performs two main tasks concerning; edge creation, and edge testing. Building edge weights is trivial, and this list is constructed after the start and end lists have been completed.

The SDG begins by creating edges to fill the tuple list. Edges are built using a recursive process which subdivides and "zooms in" on a location in an adjacency matrix. This matrix is 'nodes × nodes' big, with the coordinates of each entry representing the start and end nodes for a unique edge. The matrix is never actually constructed or stored, but the location in the matrix is monitored and updated at each subdivision step. Initially, the entire matrix is treated as the domain. At each step, the domain is divided into quadrants, and one is randomly selected to become the new domain. Because the number of nodes is a power of two, this process eventually focuses in on a single location in the matrix, and the coordinates for that location are stored as the start and end nodes for the next edge.

Once the edge list has been filled, the edges are tested for validity. The SSCA2 benchmark states that multiple edges and self-loops may be ignored by the code [41]. These types of edges are shown in Figure 2.3. While testing for self-loops is simple, the process to find and remove multiple edges – particularly when using multiple processors – is much more involved. Multiple edges have no impact on the operation or solution for Kernel 4, which is by far the most important part of the algorithm. As such, self-loops are tested for and marked for replacement, while multiple edges are left alone. The SDG iterates until no self-loops exist in the tuple list.

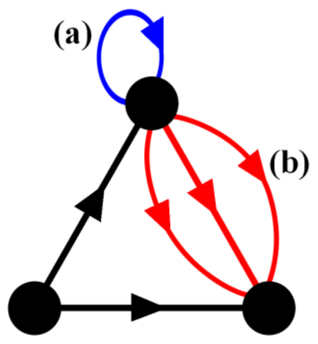

Figure 2.3. Graph of three nodes showing **(a)** a self-loop, and **(b)** multiple edges.

2.3.2 Kernel 1 – Graph Construction

The purpose of the first timed kernel is to convert the tuple list data structure into a format that all subsequent kernels will use. The benchmark states that the graph may be represented in any format, but cannot be altered by or between the remaining kernels [41].

To better understand the new data structure built by Kernel 1, a comparison to the old structure of the tuple list is considered. The tuple list is presented in an edge-to-node (E2N) format. Ignoring the list of edge weights, the tuple list consists of two arrays (start and end nodes) that are both ‘number of edges’ long. If you know a particular edge, you can easily find the two nodes it connects by looking in the arrays at that edge’s location. The new structure used in the parallel code, is a node-to-node (N2N) layout. Here, given a specific node as an input (parent), this structure makes it easy to find the other nodes it connects to (children). The N2N structure also uses two arrays, but this time there is a shorter list of pointers, and a longer list of children. Looking at location *p* in the pointer list gives a location *c* in the child array. This location *c* is the place where the children of parent node *p* are stored. The pointer list is ‘number of nodes’ long, since each node in the graph can be viewed as a parent, and the child list is ‘number of edges’ long, because it stores all the end nodes from the original edges. A simple way to think of building the N2N structure is by sorting the E2N lists according to start node. In this case, the start node array now consists of large groups of 1’s, 2’s, 3’s, etc. Next, this array is condensed into a list of *where* the 1’s, 2’s and 3’s – and hence, their children – begin. Figure 2.4 shows this simple conversion from E2N to N2N.

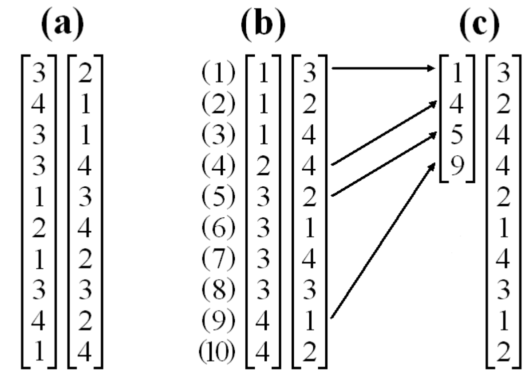

Figure 2.4. Illustration of conversion from **(a)** E2N format, to **(b)** sorted by start node, and finally to **(c)** N2N format. As an example, the third entry in the N2N pointer list is a five, meaning the children of node three begin at the fifth spot in the child array.

The GPU algorithm performs this conversion using three steps. The first is to count the number of children for each node in the graph, which is referred to as a node's *degree*. This uses a simple loop over edges which reads a start node from the E2N structure, and atomically increments that node's degree counter. The loop is easy enough to parallelize, but again, the problem of conflicting threads arises. If two threads are working on edges with the same start node, they will both attempt to increment the degree simultaneously. Unlike the smoothing and partitioning codes, where threads were competing for only a few thousand available memory locations, large graphs for this application contain several million nodes, drastically spreading out thread conflicts, making the use of an atomic operation much more affordable.

With the degree list filled with child counts for each node, the second step is to convert that list into the pointer list mentioned earlier. To find the pointer value for a particular node, the degrees of all preceding nodes must be summed. The simplest way to perform this operation is called a *scan*, which loops and sums the previous node's pointer and degree. An illustration of this process is shown in Figure 2.5. Because of the dependence on the preceding pointer, it is not immediately obvious that a parallel solution exists for this operation. However, it is possible to scan in parallel, and NVIDIA has included several algorithms with their SDK examples in the CUDA package [45]. One of these code samples has been modified to work for this project.

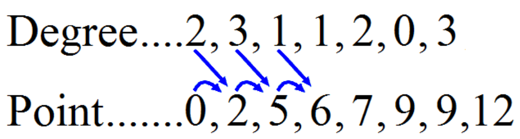
Degree....2, 3, 1, 1, 2, 0, 3
Point.......0, 2, 5, 6, 7, 9, 9,12

Figure 2.5. Illustration of simple scan algorithm. The first entry in the point array is set to zero, and all subsequent entries are the sum of the preceding point and degree values.

The final step of Kernel 1 is to build the child list. With the pointer list available, the code reads each edge's start node from the E2N structure, and finds that node's corresponding pointer. The edge's end node from the E2N list is inserted at that pointer. One complication here is that each end node must somehow be offset from its parent's pointer to ensure a unique location in the child list. To accomplish this, the end node is inserted at a location obtained by atomically referencing a second copy of the pointer list. These atomic additions return the original value in memory, and increment the pointer by one. Because this occurs atomically, there is no concern over thread conflicts, and each end node is guaranteed to be inserted into the next available spot for the parent node. The edge's weight from the E2N structure is inserted at the same location as the end node in a new weight array specific to the new N2N structure.

It is worth mentioning that as the Kernel 1 algorithm builds the standard N2N structure, which follows the true direction of the edges, it also builds a second N2N structure that goes "against the grain." These arrays are labeled $N2N_{out}$ going from parent to children, and $N2N_{in}$ for the version pointing from a child to its parents. Creating $N2N_{in}$ is as simple as sending the start and end node arrays to the GPU kernels in reverse order. This second version was designed with the Kernel 4 implementation in mind, but is also useful in Kernel 2 as well.

2.3.3 Kernel 2 – Classify Large Sets

Although the task of Kernel 2 is a simple one, the underlying algorithm is more complex than one would expect. The purpose of this kernel is to search through the edge weights and pick out edges with the largest possible weight value, which is determined prior to the execution of this kernel. The desired output is a list of the start and end nodes of these maximum-weight edges. The reason this algorithm has been difficult to develop is because of the N2N data structure used by the code. In an E2N structure, edge weights are paired with data for both nodes. In the N2N structure, weights are tied only to one node, making finding the second non-trivial.

This solution has proven to be slightly complex, and relies on the $N2N_{in}$ structure created by the first kernel. Initially, threads search the weight list of the $N2N_{in}$ structure for max-weight values. When one is found, the corresponding node from the child list (the edge's start node) is saved. Using this start node and the $N2N_{out}$ structure, the pointer list is used to pinpoint the location of that node's children, and quickly search them for a max-weight edge. When the weight is found, the corresponding node from the child list (the edge's end node) is paired with the start node. An illustration of this process is shown in Figure 2.6.

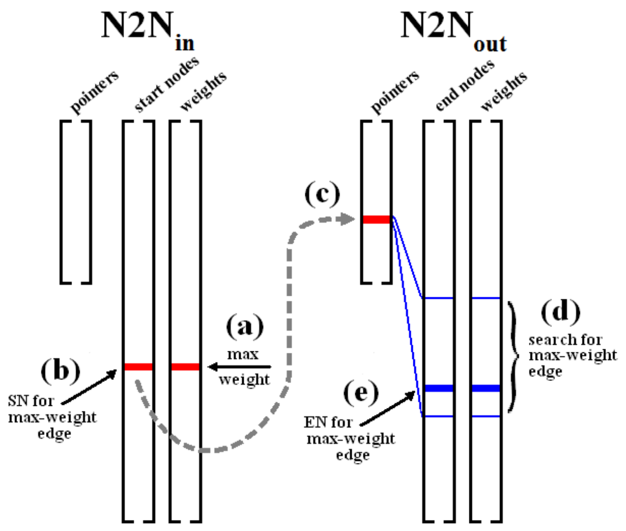

Figure 2.6. Kernel 2 algorithm steps. **(a)** Search $N2N_{in}$ for max-weights, **(b)** save edge's start node, **(c)** use pointer list for $N2N_{out}$ to find start node's children, **(d)** search children for max-weight, **(e)** locate end node of max-weight edge.

2.3.4 Kernel 3 – Graph Extraction

The third kernel of SSCA2 is designed to construct subsets – or subgraphs – of the original graph, using the edges found in Kernel 2 as starting points. Kernel 3 starts at a max-weight edge, and moves out a user-specified number of generations from it. While the full-graph and distributed-graph algorithms for Kernel 3 have the same general architecture, they contain significant differences. These differences are explained in greater detail in the section on SSCA2 GPU and MPI parallelization, while this section provides a brief overview of the broad scope of both versions.

Subgraph construction relies on the idea of using a queue, which serves two important purposes; it is a record of the nodes belonging to the subgraph, and it provides a roadmap of where to explore next. This queue is filled in sections, as the code steps out to each new generation or depth level of the subgraph. The code reads parent nodes from the current generation of the queue, and fills in their children in the next. On the subsequent iteration the children are treated as parents, and using the $N2N_{out}$ structure, any of their children that are not already in the queue are added. Working in this manner allows the work done at each level to be parallelized on the GPU. The algorithm begins by seeding the first generation with the end node of the initial max-weight edge, and loops over the GPU subgraph code until the desired number of generations has been reached. In both the full-graph and distributed-graph versions of the code, the basic algorithm for Kernel 3 plays a major role in the coding of Kernel 4.

2.3.5 Kernel 4 – Graph Analysis Algorithm (Betweenness Centrality)

The intent of the fourth and final kernel is to assign betweenness centrality (BC) scores to each node, with these values reflecting how "connected" a node is to the rest of the graph. As with the full and distributed versions of Kernel 3, the two algorithms for Kernel 4 share a similar basic structure, but are significantly different in the details of their execution. The subtleties of the algorithms are explored further in the section on SSCA2 GPU and MPI parallelization, while this section gives a general overview of the common aspects.

BC scores are calculated by selecting a start node, and assigning two values for all other nodes in the graph. A node's shortest path value, or σ, is the number of routes that exist from it back to the start node which require the fewest possible number of steps. This value is the sum of the shortest path values for each of the node's parents. A node's delta value, or δ, is described by the relationship shown in equation 15, where ω represents the child node, and v represents a parent node for ω lying on a shortest path. A node's delta value represents its partial BC score when beginning from the given starting node. This process is repeated and delta values are calculated using all nodes in the graph (or at least some subset) as the starting point. The final BC score is the sum of all these intermediate scores.

$$\delta[v] = \delta[v] + \frac{\sigma[v]}{\sigma[\omega]} \cdot (1 + \delta[\omega]) \tag{15}$$

Shortest path and delta values are assigned during two basic steps, as proposed by Brandes [43], and Bader and Madduri [44]. Starting from the initial BC node, the algorithm steps out by generation, just like in Kernel 3. This process, known as the outsweep, is where shortest path values are determined. The algorithm tests each child node to ensure it hasn't yet been visited, and adds its parents' σ values to its own. In both the full- and distributed-graph versions, the Kernel 4 outsweep moves through the graph in the same manner as Kernel 3, and this code is able to be reused with some minor modifications. First, while the Kernel 3 algorithm only keeps a record of nodes it has visited, the outsweep also needs code to actually update the appropriate σ values. More importantly, parent nodes along shortest paths must be "marked," since the code works its way back in along these paths during the second part of the algorithm. A loop searches for these shortest path parent entries in a child's $N2N_{in}$ data, and subtracts a predetermined value from the parent node, making the entry negative.

The second step calculates delta values for all nodes, which are initialized to zero, and is referred to as the insweep. This portion of the Kernel 4 code begins at the farthest generation from the start node, and works its way back in towards it. This is where the $N2N_{in}$ data structure is truly indispensible. For a given child, its $N2N_{in}$ data is searched for negative entries, indicating a shortest path parent. When one is located, the previously subtracted value is added back on, giving the parent's true node number. With the σ and δ values for the child and appropriate parent, the δ value for the parent is updated accordingly, using the formula in equation 15. Once the insweep has finished the current generation and is ready to move on, children's delta values are added to their nodal BC scores.

However, safely updating a parent's delta value is a major problem encountered in both implementations of Kernel 4. Because several children may share the same parent, several threads may try to update that parent's delta value concurrently. This is a relatively low-cost atomic addition, but floating-point based atomics are not supported on current graphics hardware. This problem has been circumvented by multiplying these values by large powers of ten, and typecasting them as integers. The code atomically adds the integer values, and later divides the final sum by the same power of ten to return to floating point form. Although this allows the code to work around this issue, it is not a perfect solution. For one, performing the multiplications and divisions to change decimal location is time consuming. Also, the answer is not as accurate as the CPU solution, which does not require this technique. With standard integer precision, the code is only able to achieve three or four digits of accuracy. Multiplying by a larger power of ten helps this, but puts final BC sums close to the upper limit of machine precision. Typecasting as "unsigned long long integers" for the delta values has improved precision to five or six digits, but this new data type requires twice as much space in memory.

2.3.6 SSCA2 GPU and MPI Parallelization

As stated earlier, the full-graph and distributed-graph versions of the SSCA2 code are very similar for the SDG and Kernels 1 and 2. The most significant difference in these sections is the amount of work done by each processor. In the full-graph code, every GPUs does *all* work associated with these parts of the code. In the distributed-graph version, each GPU creates, sorts, and searches only a fraction of the graph's edges. The important algorithmic differences are seen in Kernels 3 and 4.

2.3.6.1 Full-Graph Algorithm

With each processor owning a local copy of the entire N2N lists, MPI communication for the final two kernels becomes trivial. To distribute work across multiple processors, each GPU simply builds its own unique subgraph or calculates delta values for one of the BC start nodes. No MPI is required at all for Kernel 3, and only a simple reduction is performed in Kernel 4 to sum partial BC values across all processors. For this version of the code, the main focus has been on the algorithm for inserting nodes at the proper location in the queue.

The most challenging part of the parallelization the queueing code is determining where to insert children. In serial, because the processor can only work on a single parent at once, a pointer to the next empty location in the queue is incremented every time a node is added. In parallel, there are multiple threads filling in children for multiple parents at the same time. Without each thread knowing specifically where to insert its children, a thread would need to wait for the preceding one to finish, effectively serializing the operation. Each thread must have its own dedicated space in the queue, and it must know where that space begins and ends. This is achieved is by keeping a *count* array which corresponds to the queue. When a child is added into the queue, its number of children is determined from the $N2N_{out}$ pointer list, and recorded in the count array. After each generation has been filled, this count list is scanned into a *point* list, which indicates the queue location where that node’s children will go. This procedure guarantees unique spaces in the queue for all children. A simple example of this process is illustrated in Figure 2.7.

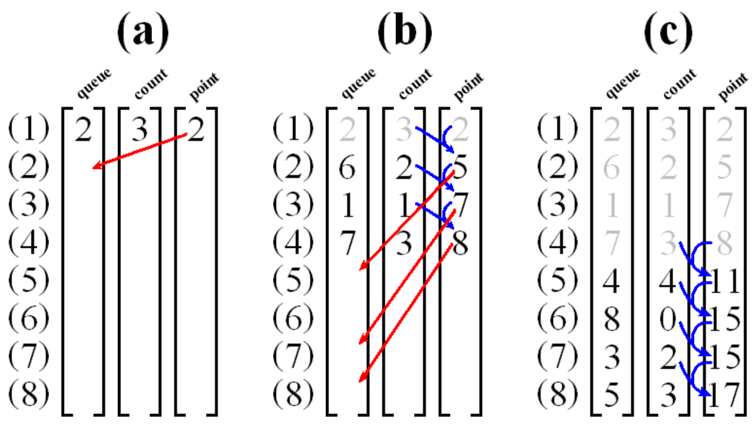

Figure 2.7. Construction of the queue used by Kernel 3. Part **(a)** shows the first node in the queue, along with its number of children, and the location they will be inserted. Part **(b)** shows the second generation, and how their children scanned. Part **(c)** shows a portion of the third generation.

Another important consideration for Kernel 3 is how to deal with nodes that are visited multiple times, since child node almost always has several parents. Once a child has been added to the queue, the code needs to avoid re-recording that same node later on. By filling the queue with unnecessary duplicate entries, the queue requires more space in memory, and a considerable amount of additional work is created. The solution to this problem involves two important control structures. First, an 'if' statement at the beginning of the outsweep code tests if the parent node from the queue is valid. The queue array is initialized to all –1's (node values begin at zero) and threads only proceed if their queue entry is non-negative. Second, the 'number of hits' value for each node is maintained and incremented atomically every time that node is encountered. If this atomic test returns a zero, it means the child has not been hit, and can be added to its space in the queue, along with its number of children. If the entry is non-zero, the child has already been hit and added somewhere else by another one of its parents. The corresponding queue entry is left as a –1, and its number of children is recorded as zero, since its true number of children has already been recorded elsewhere.

As mentioned earlier, the full-graph version of Kernel 4 uses this same queuing structure. Again, code must be incorporated to atomically add parents' σ values to their children's, and the aforementioned $N2N_{in}$ marking code is also included. Because the queueing algorithm does not allow for duplicate entries, the outsweep code continues until it reaches a generation whose children have all previously been hit. Without any nodes added to the subsequent level, the algorithm has a built-in stopping point.

2.3.6.2 Distributed-Graph Algorithm

With each process owning only a portion of the graph's connectivity structure, but still needing access to all data, the focus for the distributed-graph algorithm has shifted from the queuing algorithm, to simple and effective communication. Since node numbering is random, as the number of processors increases, it becomes less and less likely that connectivity data for a parent and its children will exist on the same process. As a result, at each generation of the graph, large amounts of MPI communication are required between all processes. Even when using only two processors, the time required to copy and communicate data via MPI dwarfs GPU computational time, by a factor of roughly 50-60 times.

A simpler queuing system has been developed for the distributed graph code. The original system requires extra memory for the child and point arrays, and takes extra time to perform scans. While the simplest insertion method would use an atomically incremented pointer, thousands of threads fighting for a single value would be inappropriate. Even though the efficiency of the GPU code is not as much of a concern due to the dominance of MPI communication, this is still not an option. So, instead of using a single, very large queue, that space in memory has been split into many smaller queues. The reasoning is that with many queues – and therefore, many pointers – atomic conflicts are greatly reduced, making their use much more efficient. Each process owns a share of these queues, and its range of nodes is distributed evenly over its queue space. As the next generation of nodes is inserted, their node numbers are used to determine which queue they belong to, and their threads fight atomically for that queue's pointer. Each process has two sets of queue and pointer arrays; one for parents and one for children. The parent arrays hold node data from the current generation, while the child arrays are filled with data for the next generation. After the completion of each level, queues and pointers are communicated from the child arrays of the sending process to the parent arrays of the recipient. Since children are loaded into queues based on their node number (and therefore, their owner process), at the start of each new generation, nodes in the parent queues are guaranteed to be locally owned by the recipient process. The queuing and MPI structure is shown in Figure 2.8.

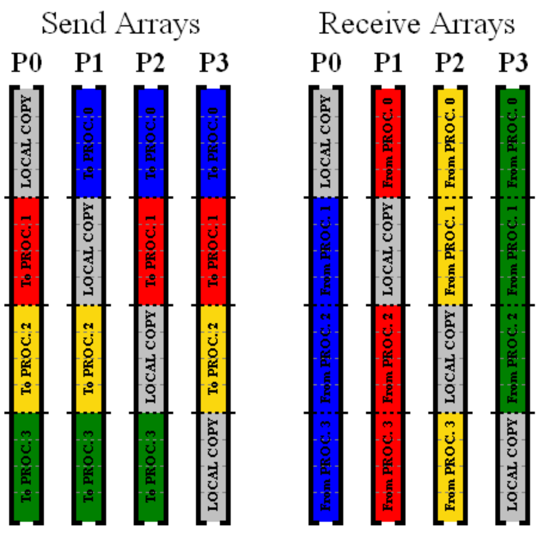

Figure 2.8. Illustration of distributed-graph Kernel 4 queuing and MPI system. This example shows four processors, with four queues each. The MPI function essentially performs a basic transpose operation for the queues belonging to each process.

While this is the general structure for the distributed-graph version of Kernel 3, the modifications required for Kernel 4 are significant. Like connectivity data, a node's hits, depth, shortest path and delta values are also stored locally, which presents a new challenge during Kernel 4's outsweep and insweep steps. For the outsweep, a parent's shortest path value is read from memory and added onto its children's values. In the full-graph version of the code, this is simple, because both the parent and child values are stored locally. In the distributed-graph code, the read and add steps must be split, since parent and child data are almost always on different processors. Here, a parent's σ value is stored and sent along with the child node in an array corresponding to the queue. When the data arrives at the recipient process, the σ value is added onto the child's, which is available locally. In addition to the sending the child array and the array containing parents' σ values, a third array containing parent node numbers is also sent, which is used in marking the child's $N2N_{in}$ data structure. In this version of the algorithm, the outsweep consists of three steps. First, a GPU kernel updates shortest path values locally with the values received from the previous generation. Also in this first step, shortest path edges in the $N2N_{in}$ array are marked for use during the insweep. Second, a GPU kernel loads children, σ values and parents into queues (this step occurs second, because the parent's σ values must be up to date before being saved for their children). Finally, a MPI function communicates the necessary arrays between processes. A CPU loop over these three steps continues until all nodes have been visited.

For the insweep, a similar approach must be taken, as a child's sigma and delta values are necessary when calculating updates to their parent's delta value. Again, three steps are used. A GPU kernel loops over locally-owned nodes residing in the target generation, and checks their $N2N_{in}$ data for shortest path parents. When one is found, the kernel saves the parent node number to the queues, as well as the child's sigma and delta values. In the second step, this data is communicated via MPI to all processes. Finally, a second GPU kernel reads the parent nodes from the queues, as well as the associated child data, and atomically updates local parent delta values using equation 15. A CPU loop repeats these steps until the code works its way back to the start node.

CHAPTER 3

RESULTS

3.1 Smoothing Results

The performance of the smoothing code has been examined using two different criteria. The main focus of this study has been on timings, and the relative speeds at which the parallel code operates. However, some attention has also been devoted to the improvements in mesh quality that the smoothing code generates. The following two sections outline these results. For the smoothing code, all trials were run on Orion, a machine with four NVIDIA GeForce GTX 295 GPUs, each with 1 GB of RAM.

3.1.1 Smoothing Timing Results

The plots in Figure 3.1 show GPU computational timings for the mesh smoothing code. Tests were run on four different test meshes with varying node counts (roughly 9K, 46K, 69K and 72K), using up to four GPUs. Since the emphasis of this study is the evaluation of the parallel performance of the code, arguably the most important analysis comes from comparisons between test cases. Figure 3.2 compares smoothing times for multi-GPU tests to the times of the single-GPU cases. The expectation is that for the same size problem, doubling the number of processors should halve the required time, and the simulation should run twice as fast. This “halving” relationship is illustrated by the dashed gray line.

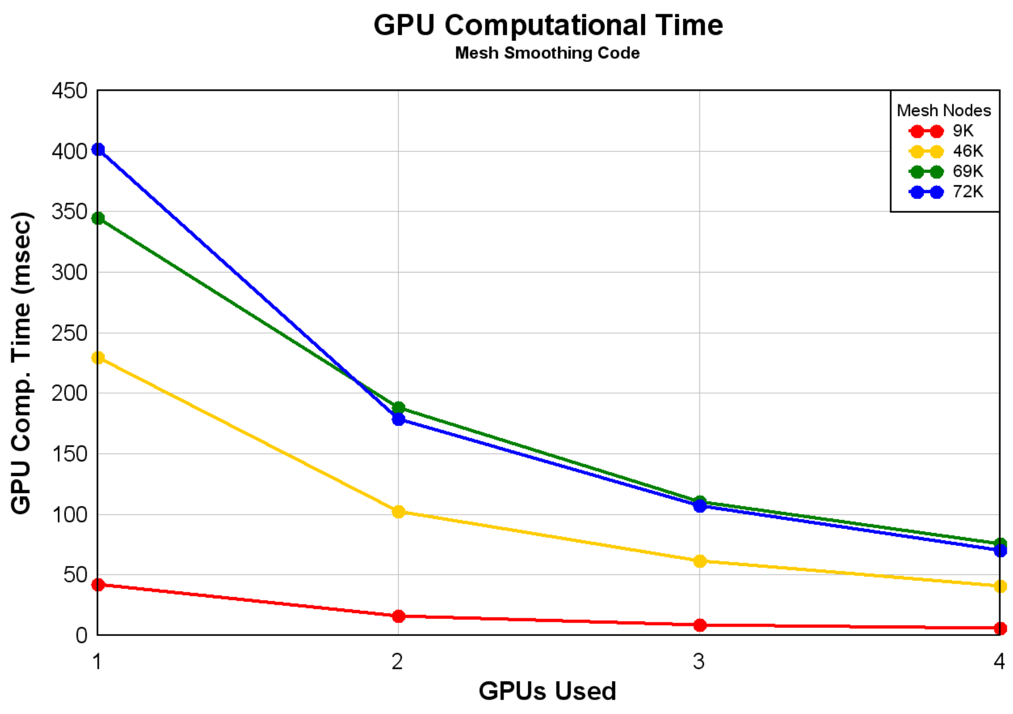

Figure 3.1. GPU computational timings for the mesh smoothing code (milliseconds).

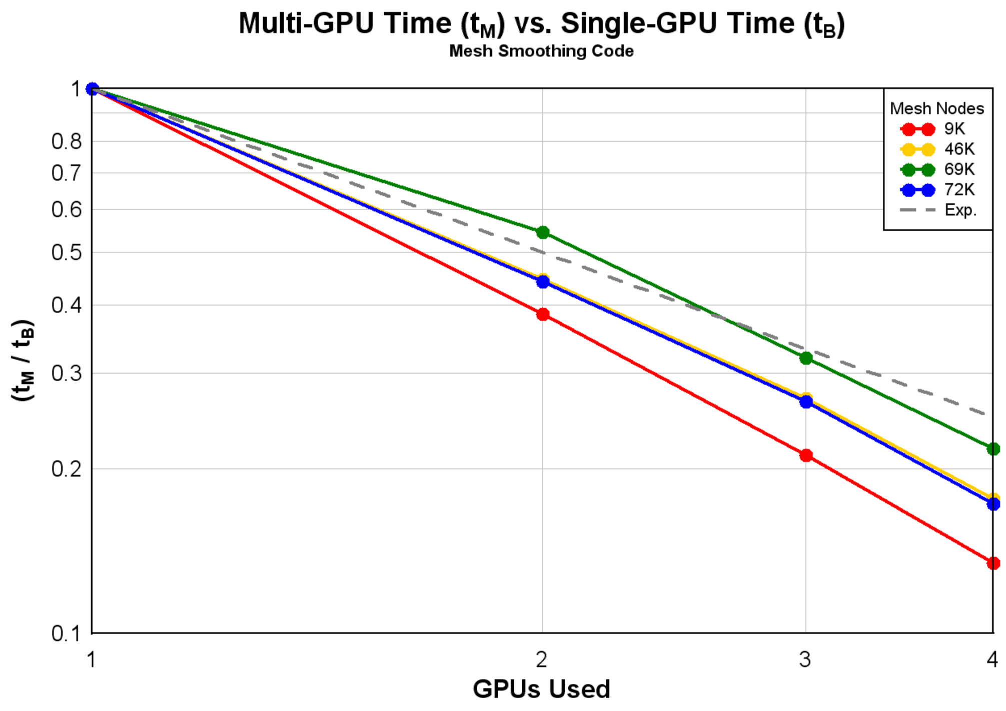

Figure 3.2. Smoothing code Multi-GPU test times compared to single-GPU test times. The expected doubling relationship is illustrated by the dashed gray line.

3.1.2 Smoothing Quality Results

To evaluate the code's improvements to mesh quality, several "bad" meshes were created for comparison. In these meshes, nodes were "perturbed" to intentionally create poor-quality elements, with particular emphasis on high-aspect ratio cells. These meshes were then run through the smoothing code, and the results were compared with the initial meshes. First, a coarse mesh of a 1×1×1 cube was used to visually confirm that node movement was taking place. Nodes were pushed away from the cube's centroid, producing high-volume cells at the center of the mesh, and slivers at the mesh's faces. Figure 3.3 shows both the perturbed and smoothed meshes, with views of the meshes' exteriors, along with thin layers of cells from the cubes' bottom faces.

While the cube case showed that the smoothing code was capable of significant and meaningful node movement, it did not provide any insight as to how that node movement would affect the quality of CFD or FEA simulations. To test this, four meshes of a 1×1×10 rectangular bar were perturbed and smoothed, and then all meshes were evaluated using the FEA solver from the OpenFOAM CFD software suite [46]. Nodes were perturbed using several different schemes, however, all variations converged to essentially the same smoothed mesh, with less than 1% variation in nodal positions. The final smoothed mesh of the bar geometry is shown in Figure 3.4.

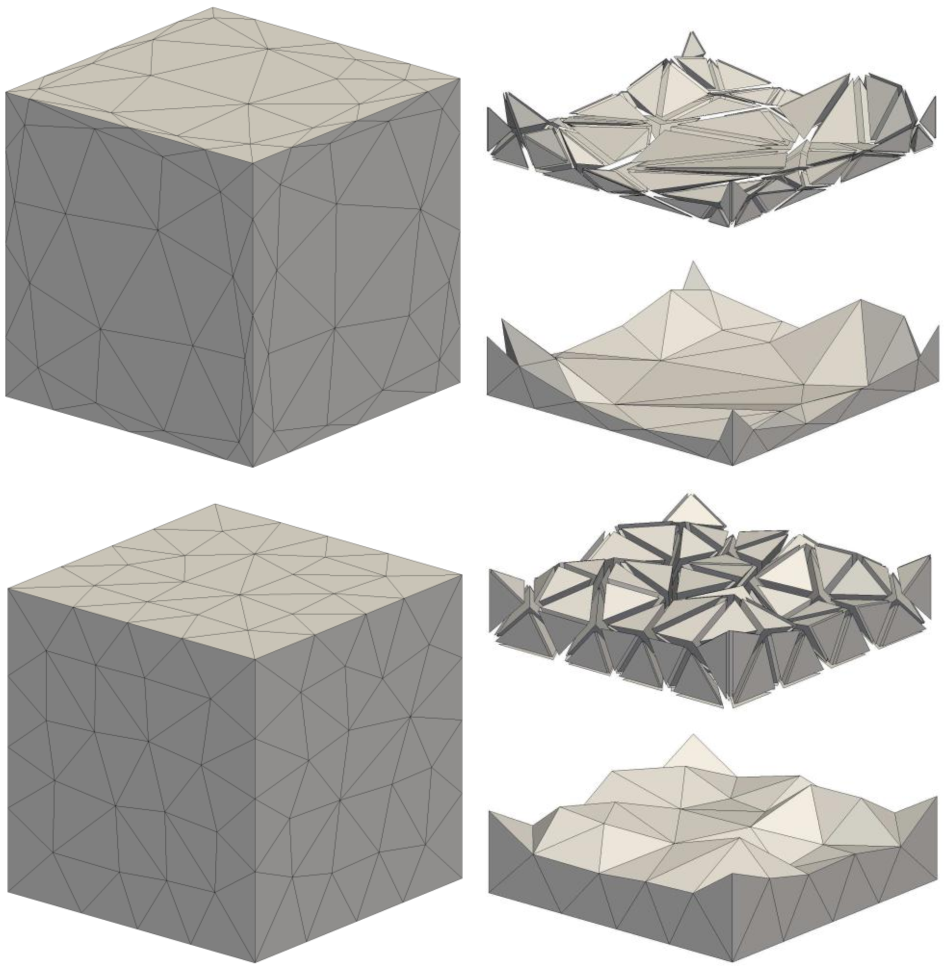

Figure 3.3. Comparison of the perturbed cube mesh (top) and the smoothed result (bottom).

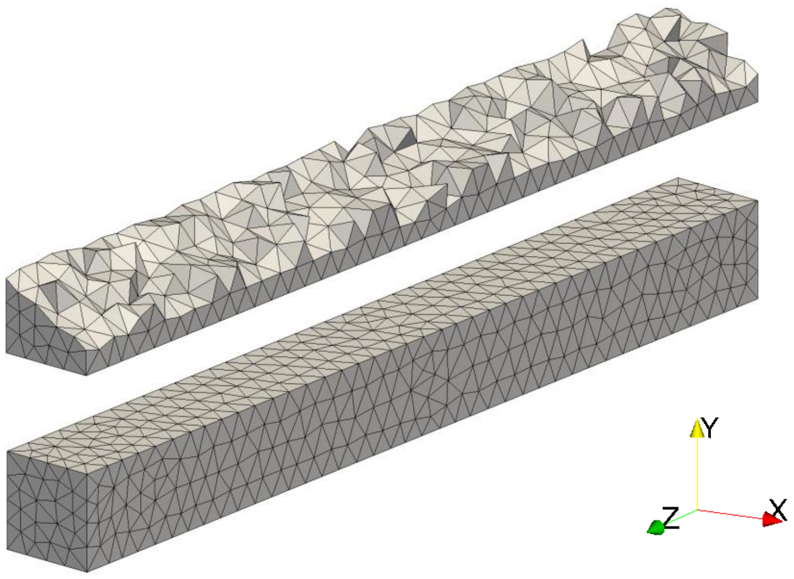

Figure 3.4. Smoothed bar mesh. The top image shows a cut-away displaying internal cells.

For the first perturbed mesh, nodes were pulled in toward the bar's major axis in the *y*-direction, and pushed away from the major axis in the *x*-direction. This created areas of high resolution horizontally along the major axis, and vertically at the bar's left and right sides. The second perturbed mesh reversed nodal movement in the *x*-direction, pulling nodes in toward the major axis in both the *x* and *y* directions. Now, areas of highest resolution occurred along the major axis, aligned with both of the bar's minor (*x* and *y*) axes.

Neither of the first two perturbed meshes moved nodes in the *z*-direction, along the beam's major axis. The third and fourth test meshes used the same nodal displacements as the second, but also pushed nodes away from the centroid in the *z*-direction as well. The third mesh used a moderate push, while the fourth mesh imposed a larger displacement. Both of these meshes resulted in severely skewed cells, including a great deal of slivers. The fourth mesh contained a particularly high number of these cells, as well as several very coarse cells near the center of the beam. Images of the four perturbed meshes can be seen in Figure 3.5.

Table 1 shows some common measurements associated with mesh quality. An ideal mesh would contain cells with low aspect ratios and comparable cell volumes. All beams were subjected to a 10 kN force in the negative *y*-direction on one end, with the other remaining fixed. Displacements and stress values for each mesh, along with the appropriate beam theory values, are shown in table 2. Volume measurements are reported in units of ($10^{-3}m^3$), and deformation measurements are in units of (m), and stresses are in units of (Pa). The simulation on the fourth perturbed mesh did not converge. While the smoothed mesh is still not ideal, it results show a dramatic improvement over the perturbed versions, and are close to the theoretical values. Because the focus of this research is not the optimization of the smoothing algorithm, but rather the viability of using GPUs for these types of CFD applications, these results are satisfactory.

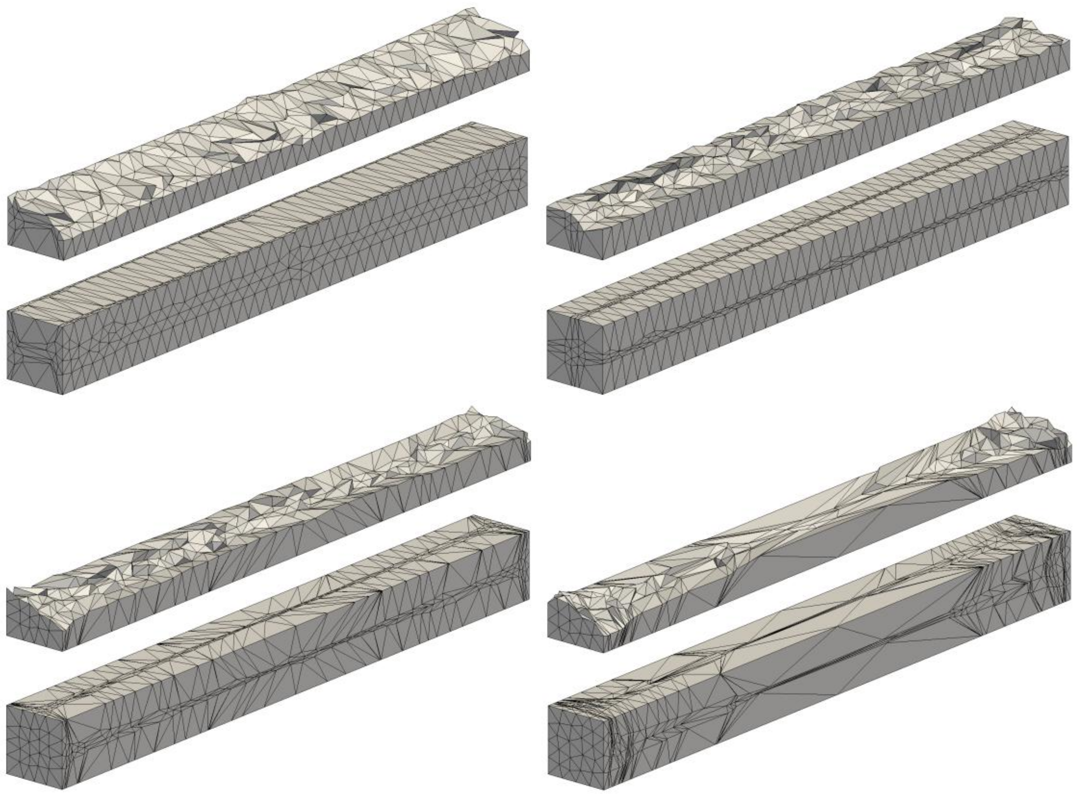

Figure 3.5. The four perturbed bar meshes.

Mesh	Max. Aspect	Max. Cell Vol.	Min. Cell Vol.
Smooth	5.8	5.4	0.68
Pert. 1	32.1	2.25	0.36
Pert. 2	17.3	9.65	0.49
Pert. 3	29.8	32.4	0.36
Pert. 4	414.8	108.8	0.02

Table 1. Bar mesh quality statistics.

Mesh	Max. Def.	Max. Stress	Min. Stress
Theory	2.00E-04	37,500	-37,500
Smooth	2.04E-05	41,708	-44,141
Pert. 1	2.20E-03	7.45E+10	-9.50E+09
Pert. 2	0.018	3.81E+11	-1.60E+12
Pert. 3	992,903	2.53E+18	-7.90E+17

Table 2. Bar mesh deformation and stress.

3.2 Partitioning Results

The plots in Figure 3.6 show GPU computational timings for the mesh partitioning code. Again, tests were run on four different test meshes (14K, 26K, 49K and 68K nodes), using up to four GPUs. Figure 3.7 compares partitioning times for multi-GPU tests to the times of the single-GPU cases. The expected halving relationship is illustrated by the dashed gray line.

In terms of the quality of mesh partitions, given a mesh with enough nodes, the code is capable of distributing them such that final partition node counts differ by as little as 2%. Additionally, the code to adjust partition starting points has proven quite robust, and is capable of evenly distributing partitions throughout the mesh – even when partitions are seeded extremely close to one another, or outside of the mesh geometry altogether.

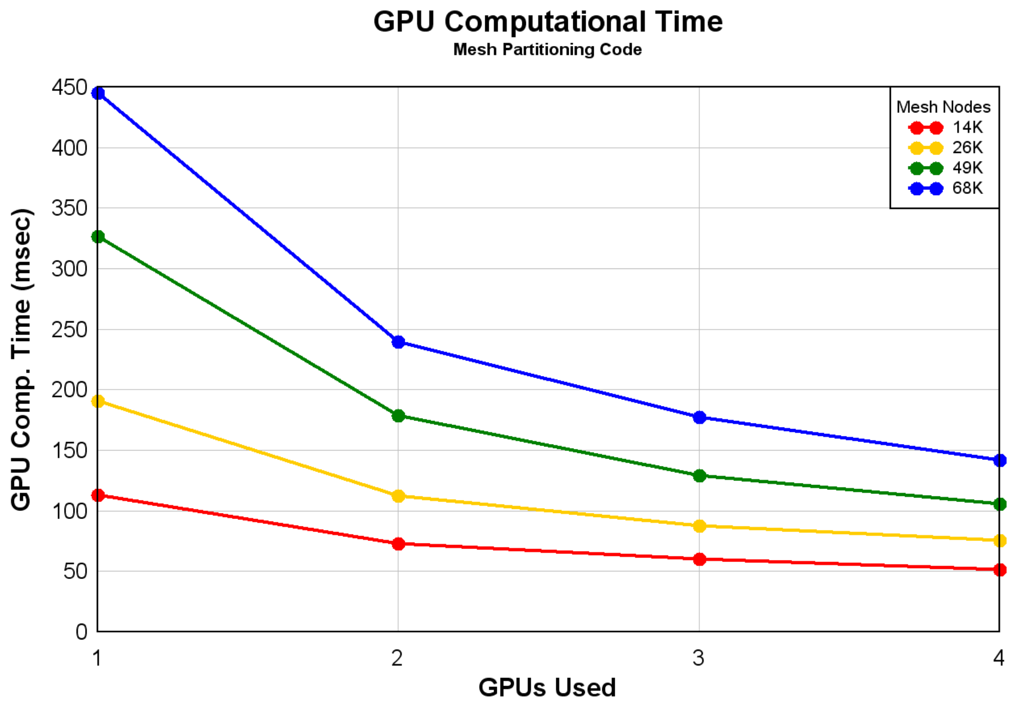

Figure 3.6. GPU computational timings for the mesh partitioning code (milliseconds).

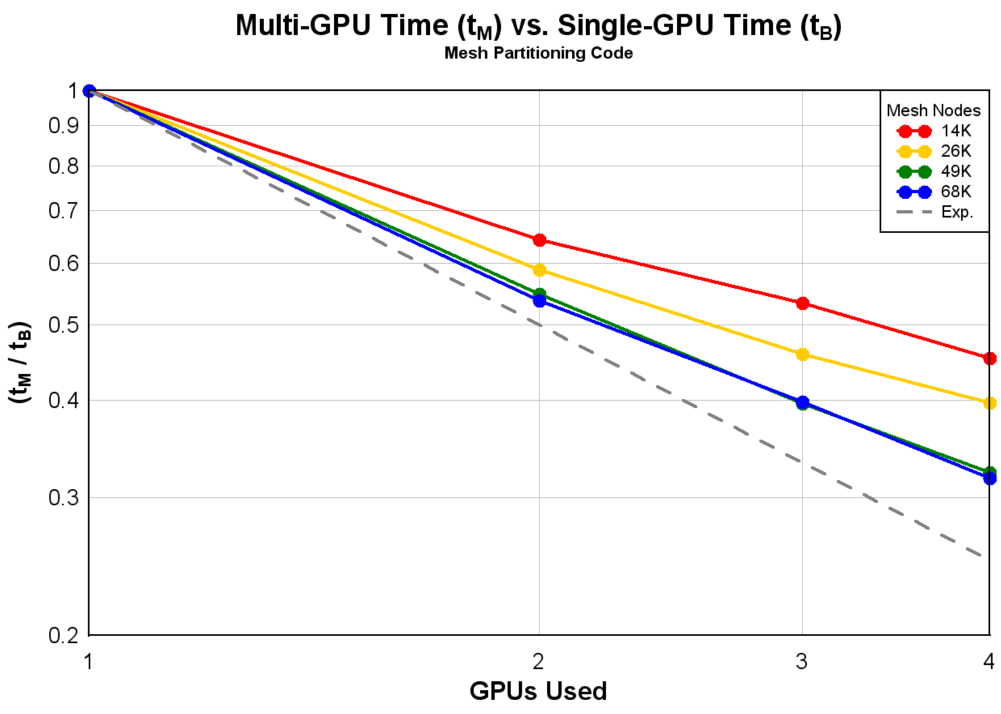

Figure 3.7. Partitioning code multi-GPU test times compared to single-GPU test times. The expected doubling relationship is illustrated by the dashed gray line.

3.3 SSCA2 Results

Both versions of the SSCA2 code have been tested using NCSA's GPU-based 'Lincoln' supercomputer. Lincoln utilizes 96 NVIDIA Tesla S1070 GPUs, (each housing four separate GTX 295 processors which share 16 GB of memory), paired with Intel 64 2.33 GHz quad core CPUs, connected via InfiniBand MPI interconnects [47]. The full-graph code was tested on SCALEs 16 to 21, using up to 128 GPUs. The distributed-graph code tested SCALE sizes 21 to 27, and used up to 32 GPUs. Additionally, some limited CPU tests were conducted on the distributed-graph algorithm as well. Results for Kernels 1, 3 and 4 are shown, since these are the most computationally taxing, and therefore the most relevant. Execution times for Kernel 2 are on the order of fractions of milliseconds – even for very large SCALE sizes. These times are simply too short to be either meaningful or even accurate. Also, while it may seem obvious to want to compare results for the two versions of the code, this data is not presented. Not only did testing only overlap for a single SCALE size, but the comparison is invalid due to the significant differences in the two algorithms. Finally, all SSCA2 timings are plotted on log scales, as this makes the exponential relationships between SCALE sizes and numbers of GPUs used appear linear, and easier to interpret.

3.3.1 Full-Graph Version

The plots in Figure 3.8 show GPU computational timings for the full-graph versions of Kernels 1, 3 and 4. These reflect only the time spent during GPU computation, and do not include any MPI, data copy or input/output operations. As previously stated, the full-graph version of the SSCA2 code stores an entire copy of the N2N connectivity data on each process which has two important consequences. First, MPI is used in only the most trivial manner, which means these timings not only reflect GPU computation, but also the bulk of the total time required for the entire kernel. Second, because the memory requirements are not scalable, there is an upper limit on problem size, based on available GPU memory.

Like with the mesh adaptation codes, comparisons between test runs are significant in evaluating parallel performance. Figure 3.9 shows the multi-GPU times divided by single-GPU times for each of the six tested SCALE sizes. Again, it is expected that for a constant problem size, as the number of processors is doubled, the code should run twice as fast, and the required time should be cut in half. The expected "halving" relationship is shown by the dashed gray line. Next, Figure 3.10 shows how large-SCALE timings relate to the smallest SCALE-16 test times when using various numbers of GPUs. As problem size doubles, using the same number of processors should take twice as long. Again, the "doubling" relationship is illustrated by the dashed gray line.

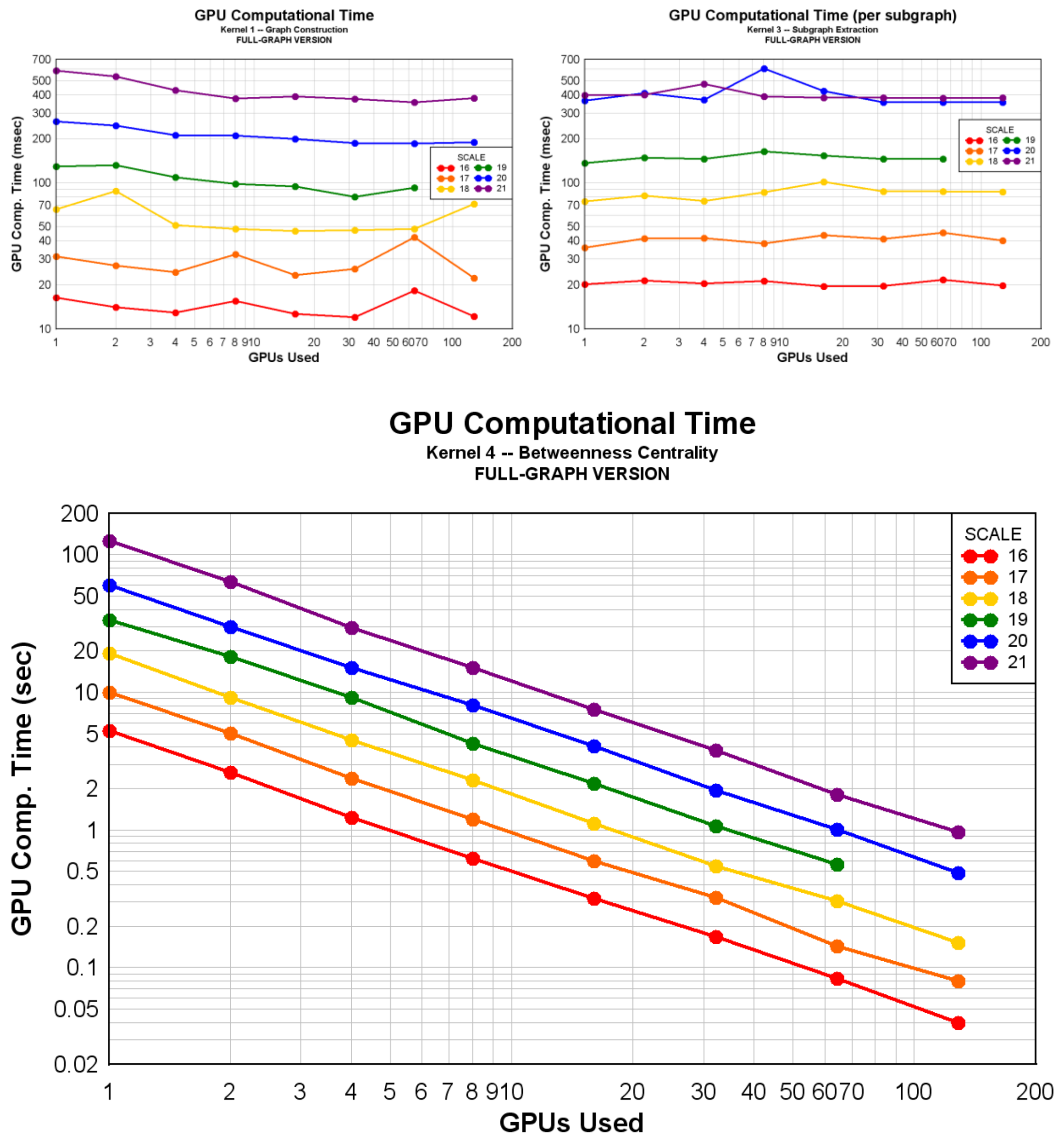

Figure 3.8. GPU computational timings for full-graph versions of Kernel 1 (milliseconds; top left), Kernel 3 (milliseconds; top right) and Kernel 4 (seconds; center).

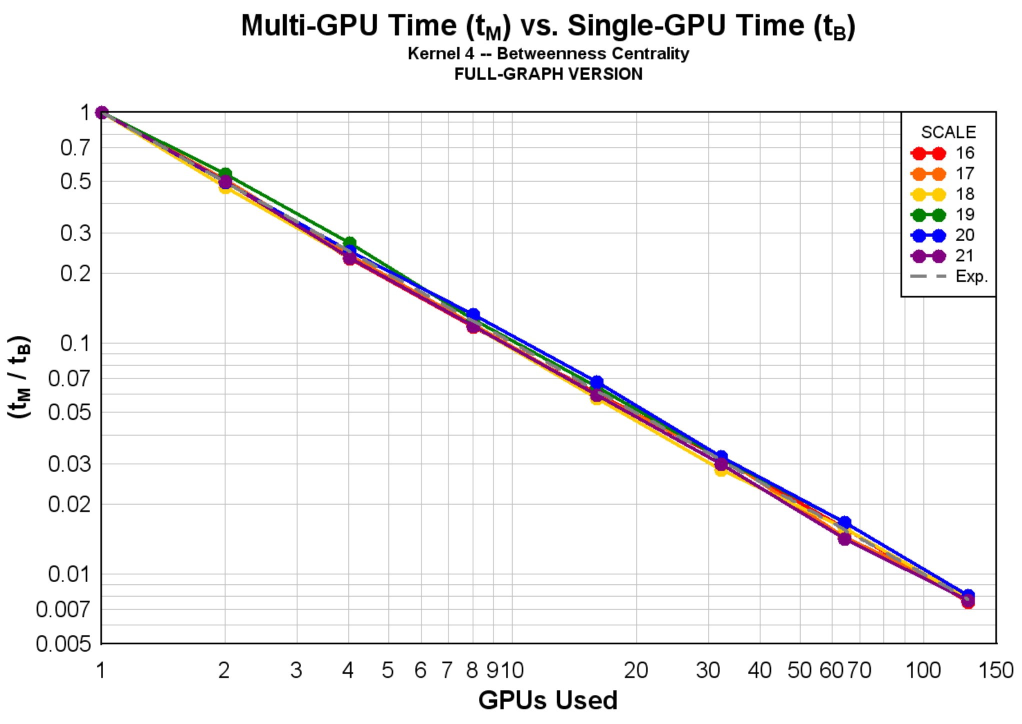

Figure 3.9. Full-graph Kernel 4 multi-GPU test times compared to single-GPU test times. The expected halving relationship is illustrated by the dashed gray line.

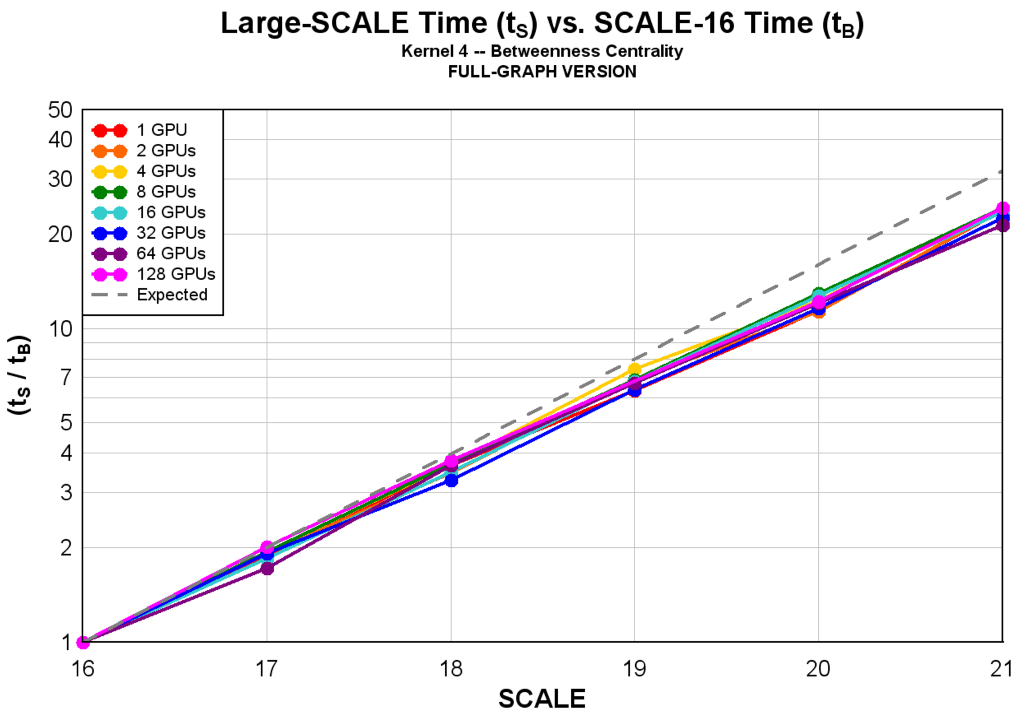

Figure 3.10. Full-graph Kernel 4 large-SCALE test times compared to SCALE-16 test times. The expected doubling relationship is illustrated by the dashed gray line.

3.3.2 Distributed-Graph Version

The plots in Figures 3.11, 3.12 and 3.13 analyze GPU computational timings for the distributed-graph code. Figure 3.11 shows GPU timings for the three major kernels, and 3.12 and 3.13 compare Kernel 4's multi-GPU and large-SCALE test times to those of baseline test cases. Similarly, Figures 3.14, 3.15 and 3.16 show these same plots, only these examine *total* computational timings, since a large portion of the total time is dedicated to non-GPU operations. For Figures 3.12 and 3.15, because SCALEs 25, 26 and 27 were too large to run on smaller numbers of processors (2- and 4-GPU runs), the baseline-GPU tests used for their comparisons are the 4-, 8- and 16-GPU test cases, respectively. All smaller SCALE sizes (21-24) used their 2-GPU test as the baseline for comparison. Plots in Figures 3.17 and 3.18 show percentages of the total times spent on GPU computation and on MPI and data transfer times (the bulk of the non-GPU operations). Figure 3.19 shows a comparison of the MPI and transfer time sums relative to GPU computation times. Lastly, the plots in Figures 3.20 and 3.21 deal with comparisons between timings for the GPU-based distributed-graph code, and comparable computation times from an all-CPU code. Here, the CPU-based algorithm is identical to the GPU code, only all computation has been moved to the CPU. Figure 3.20 shows MPI timings for the CPU and GPU versions, to confirm that the two algorithms were the same other than the computational areas. Figure 3.21 compares computational times for the CPU and GPU, showing how many times faster the GPU-based code is. It is worth noting that for the distributed-graph code, single-GPU tests were not conducted, since they do not require the use of any significant MPI communication, and would essentially be testing a largely different algorithm.

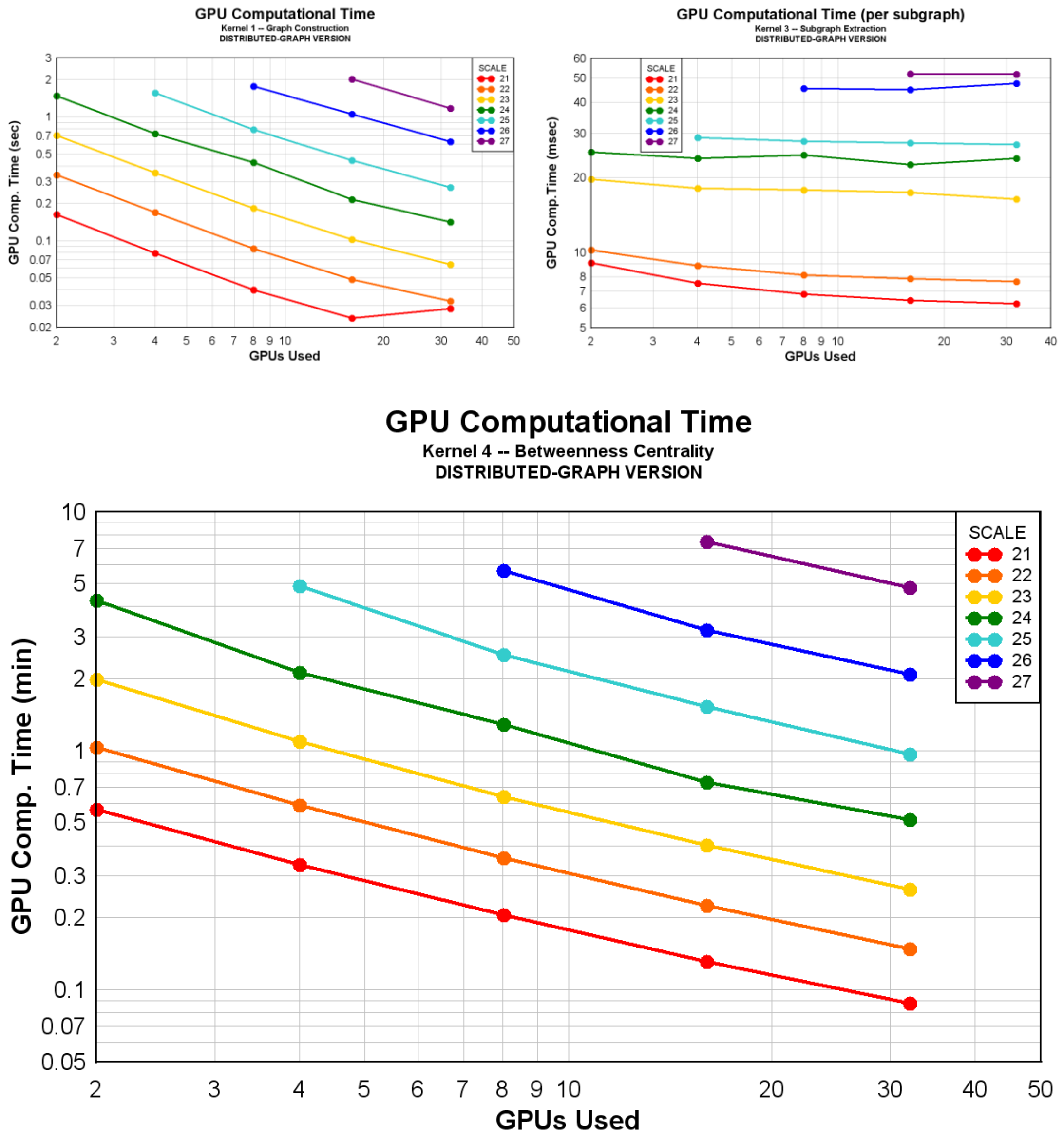

Figure 3.11. GPU computational timings for distributed-graph versions of Kernel 1 (seconds; top left), Kernel 3 (milliseconds; top right) and Kernel 4 (minutes; center).

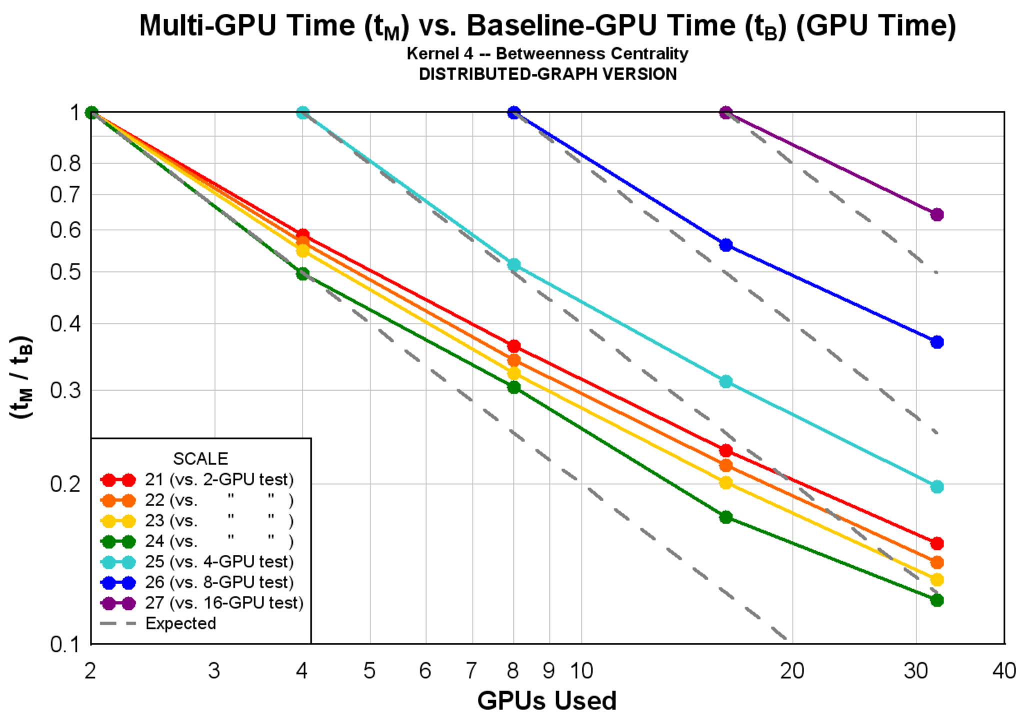

Figure 3.12. Distributed-graph Kernel 4 multi-GPU test times compared to baseline-GPU test times. The expected halving relationship is illustrated by the dashed gray line.

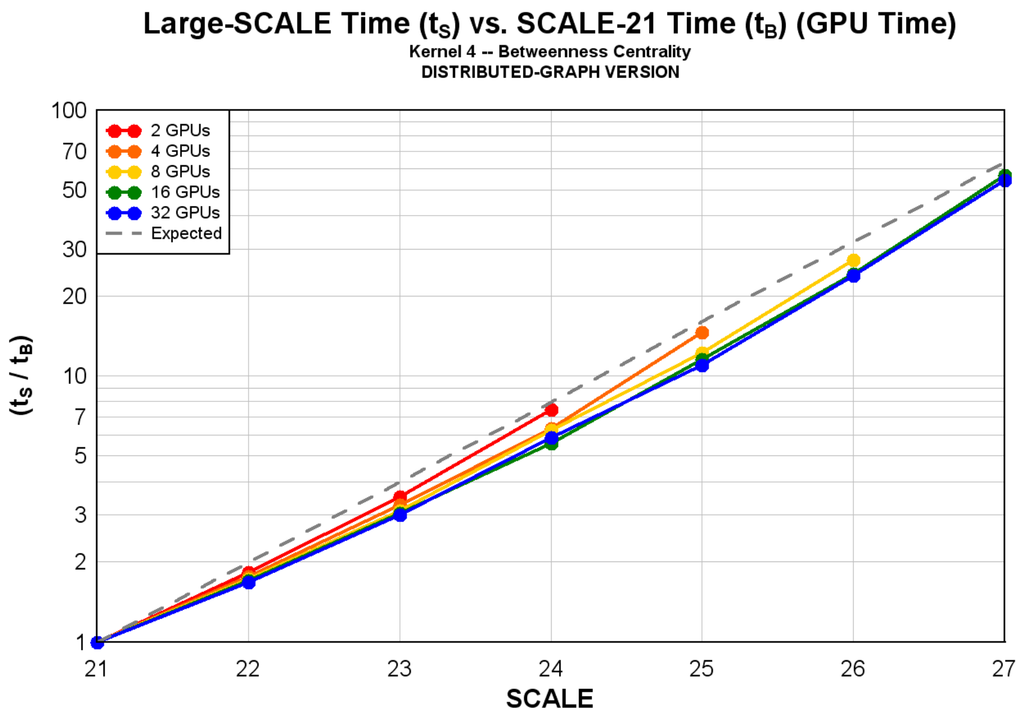

Figure 3.13. Distributed-graph Kernel 4 large-SCALE test times compared to SCALE-21 test times. The expected doubling relationship is illustrated by the dashed gray line.

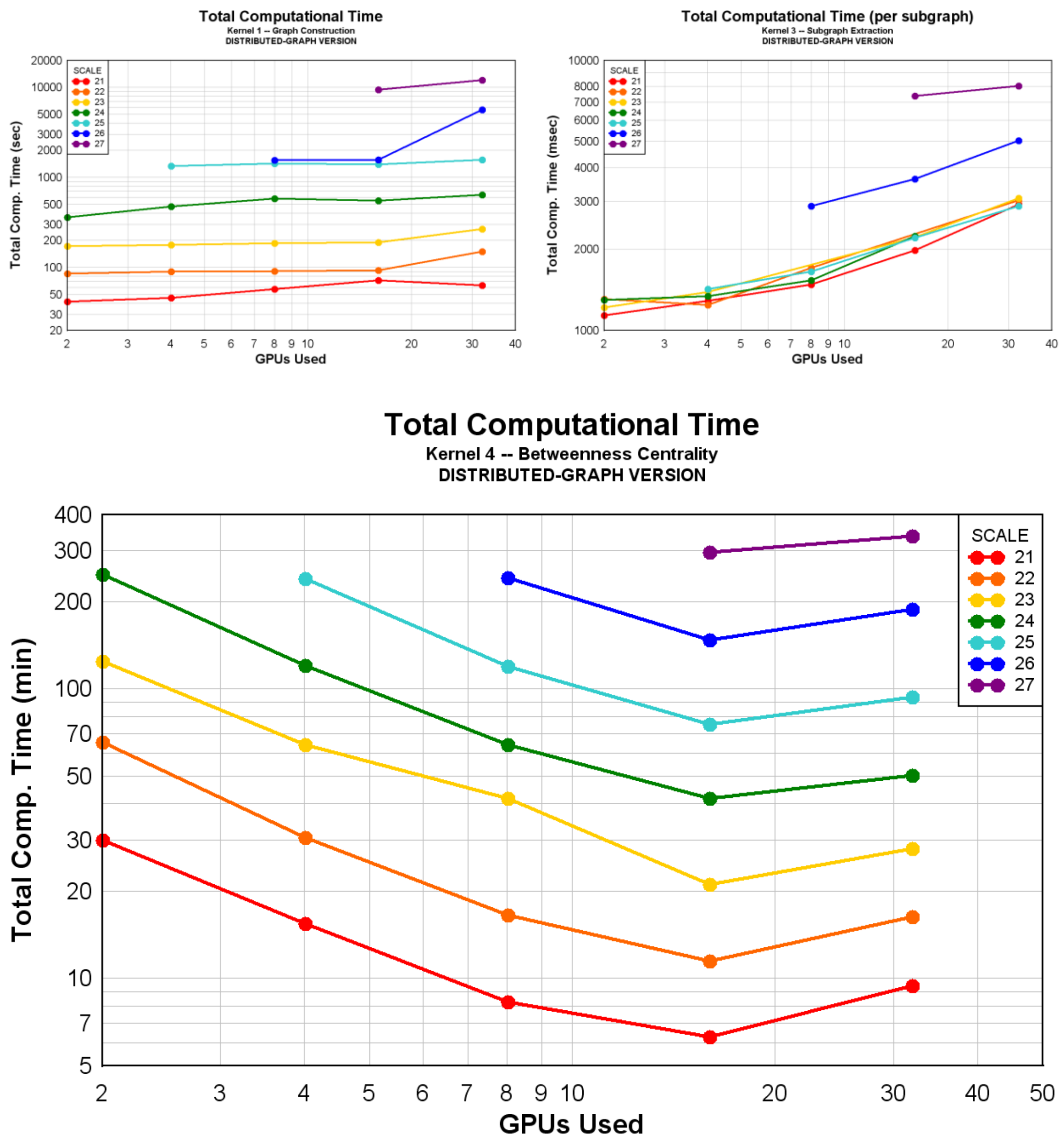

Figure 3.14. Total computational timings for distributed-graph versions of Kernel 1 (seconds; top left), Kernel 3 (milliseconds; top right) and Kernel 4 (minutes; center).

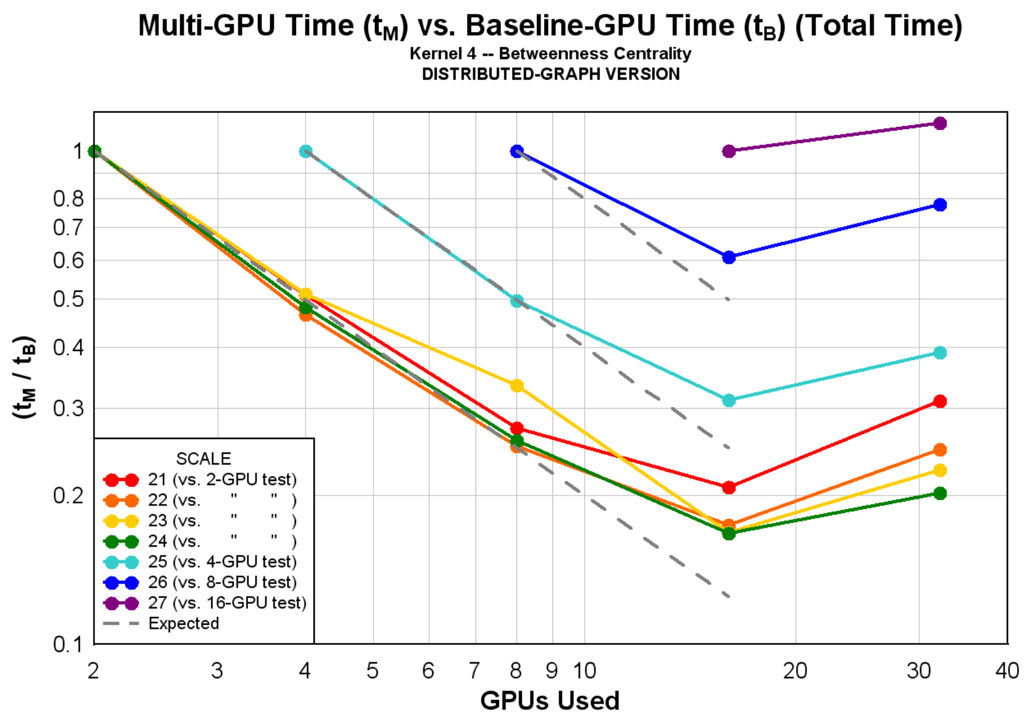

Figure 3.15. Distributed-graph Kernel 4 multi-GPU test times compared to baseline-GPU test times. The expected halving relationship is illustrated by the dashed gray line.

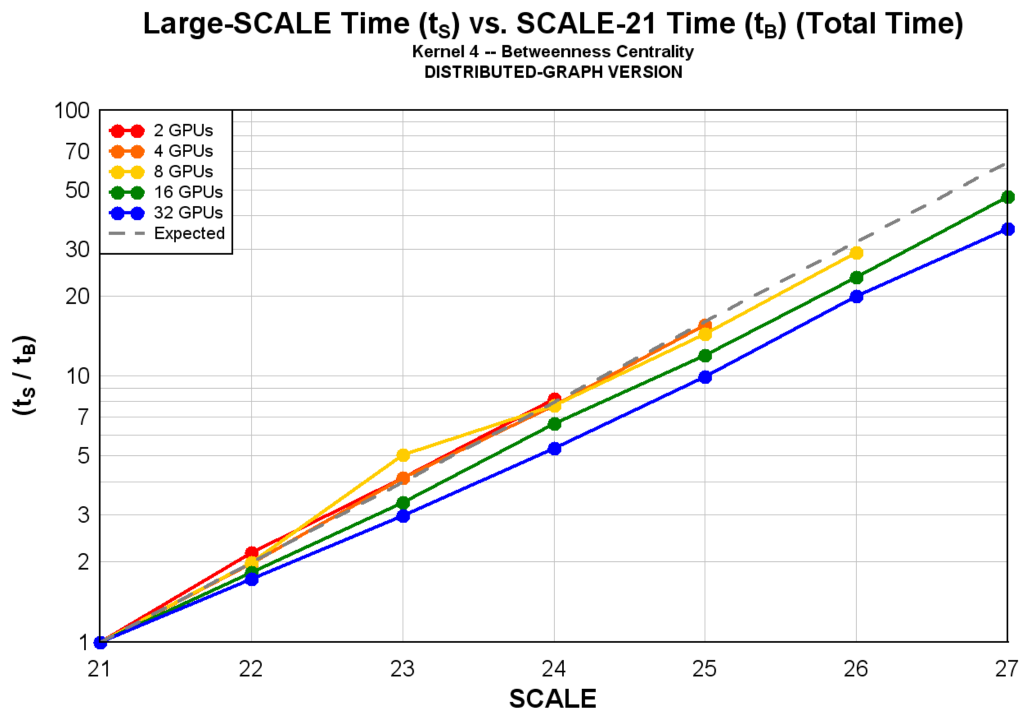

Figure 3.16. Distributed-graph Kernel 4 large-SCALE test times compared to SCALE-21 test times. The expected doubling relationship is illustrated by the dashed gray line.

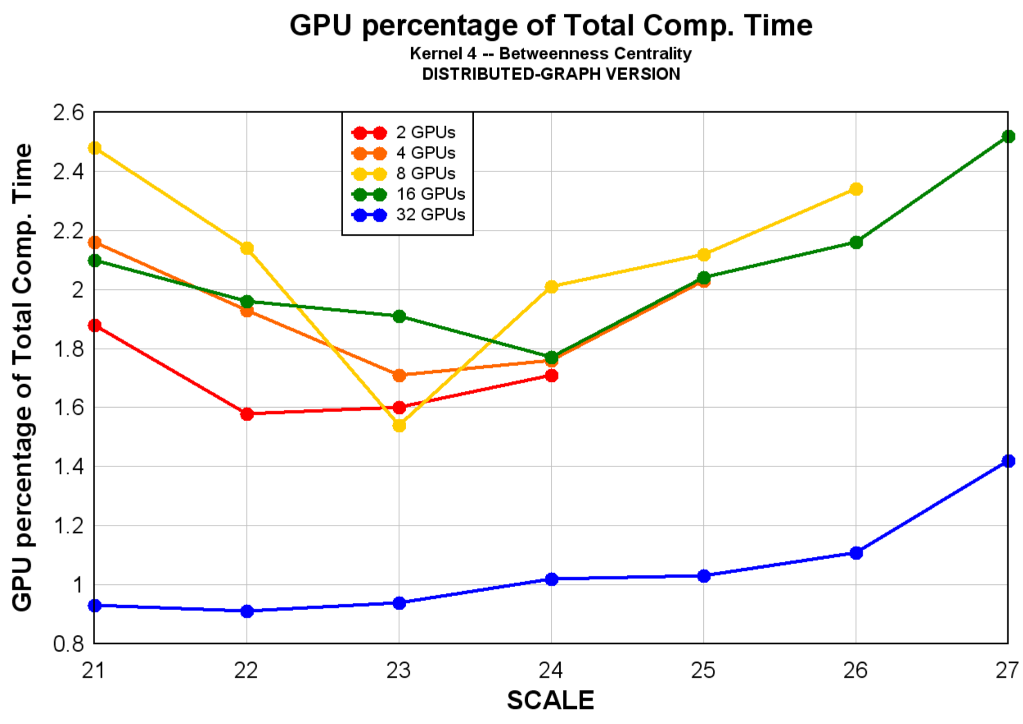

Figure 3.17. Distributed-graph Kernel 4 GPU timings as percentages of total computational time.

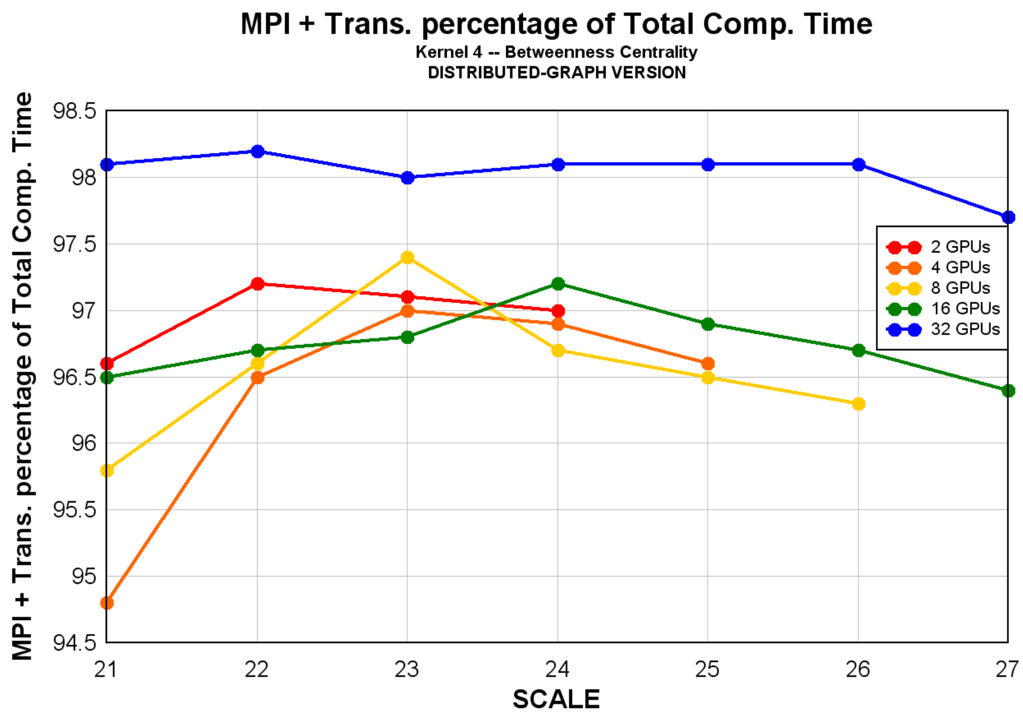

Figure 3.18. Distributed-graph Kernel 4 MPI and data transfer timing sums as percentages of total computational time.

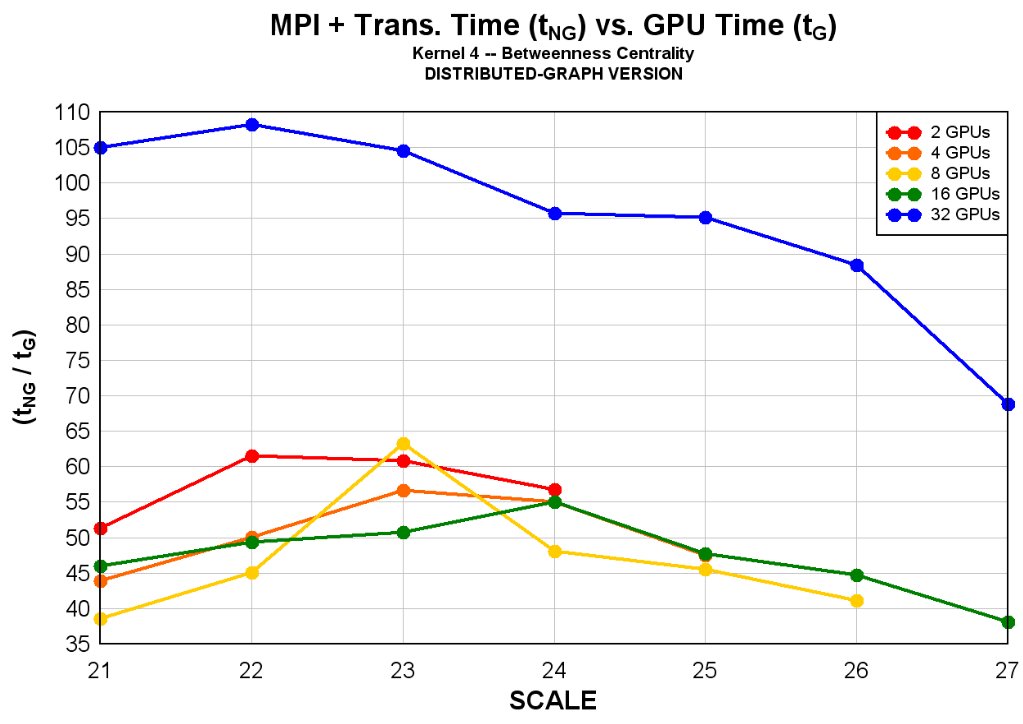

Figure 3.19. Distributed-graph Kernel 4 MPI and data transfer timing sums compared to GPU timings.

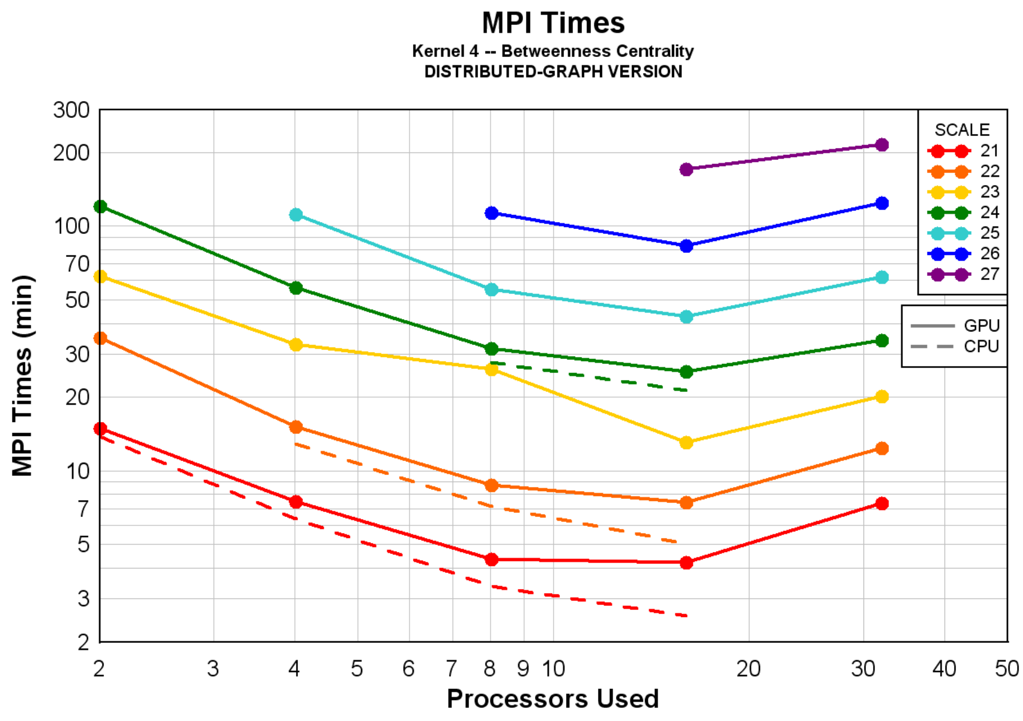

Figure 3.20. Distributed-graph Kernel 4 MPI timings for the GPU- and CPU-based codes.

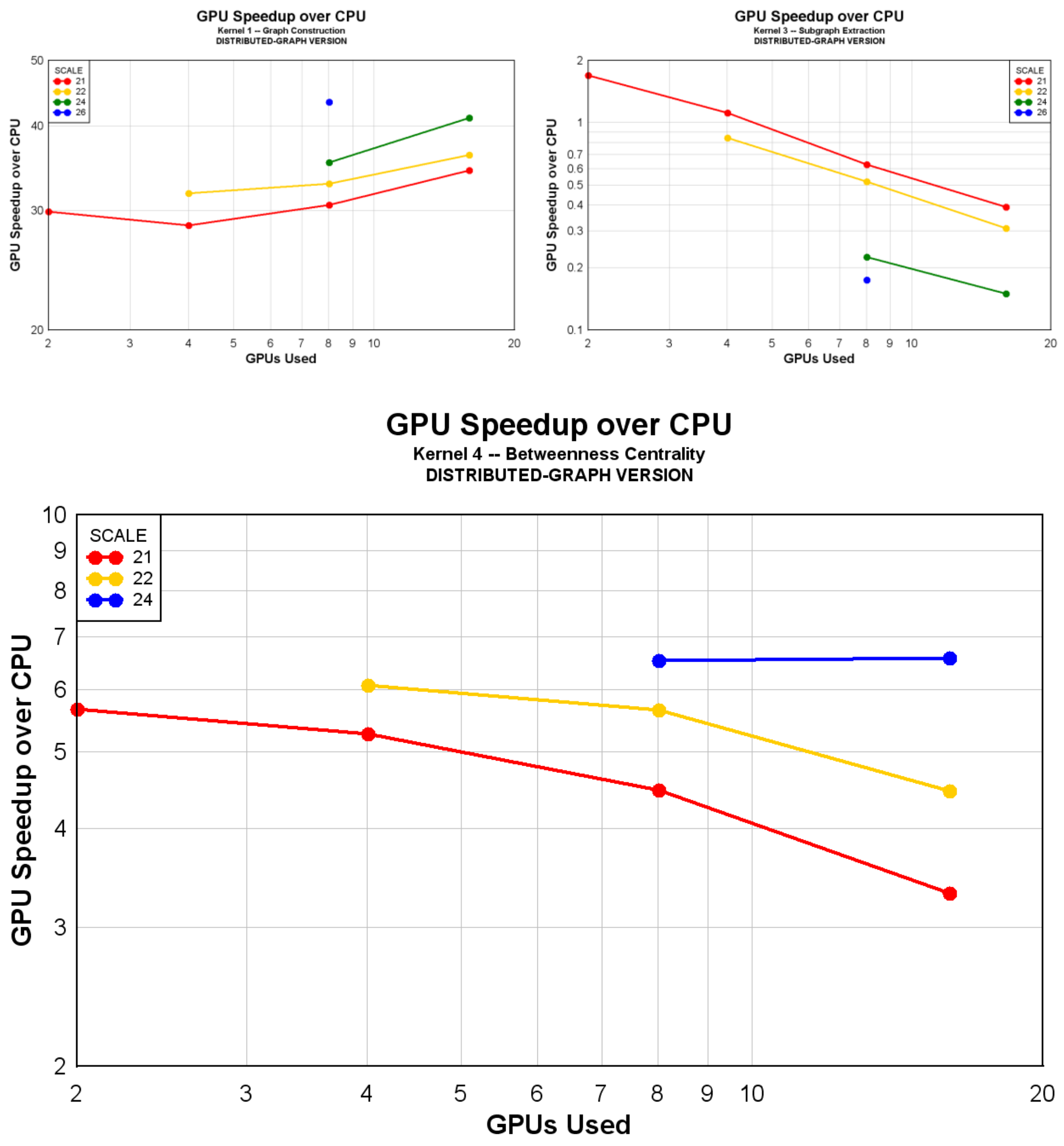

Figure 3.21. GPU speedups over CPU timings for distributed-graph versions of Kernel 1, Kernel 3 and Kernel 4.

CHAPTER 4

CONCLUSIONS

4.1 Performance and Algorithm Design

Perhaps the most important conclusion drawn from this research is that while the GPU typically outperforms the CPU in terms of pure computation, *it is the remaining parts of the algorithm that truly dictate the overall performance of the code*. For the distributed-graph Kernel 4 code, regardless of SCALE or number of processors, 98-99% of the total computational time comes from three sources: mathematical computational time, data transfer time (copying between GPU and CPU memory), and MPI communication time. Although not shown, the breakdown for the mesh adaptation codes is essentially identical. Figure 3.21 clearly shows that for mathematical computation, the fine-grained parallelism of the GPU allows for faster computation than a serial, CPU-based code. However, as shown in Figures 3.17, 3.18 and 3.19, this computation only takes a fraction of the total time relative to MPI and data transfer operations. With MPI times being essentially equal, as seen in Figure 3.20, the most significant difference in the serial and parallel codes is the need for data transfers when working on the GPU. Since this transfer time is roughly equivalent to the time required for MPI communication, the total time needed for the SSCA2 GPU code is roughly double that of the CPU code. Now, if this application only required a pair of transfers to and from the GPU at the start and completion of the code, the time savings of GPU-based computation may be competitive enough to neutralize these one-time costs. However, in iterative applications such as SSCA2's Kernel 4, and both of the mesh adaptation codes, the communication after each incremental step simply requires too many of these copies, and there is no way the GPU can compete in terms of overall performance.

Still, despite this fact, the trends in GPU and CPU computational timings do show some promise. The results in Figures 3.17 and 3.18 show that with increasing problem size, a greater percentage of time is spent on mathematical computations, while the amount of time spent copying and communicating data is shrinking. Furthermore, as stated, the GPU-based code is faster than comparable CPU code, and the relative performance does improve for larger problems. With the focus on large-scale supercomputing and increasing problem sizes, these trends in the performance of the distributed-graph code are encouraging, and indicate that at some point, for a large enough problem, the use of GPU-based supercomputers may become effective relative to CPU-based machines.

4.2 Performance and Amount of Work

One of the most obvious conclusions which can be drawn from the results of this research is that *performance scales with the amount of work to be done (problem size)*. Whether comparing to CPU code, or other GPU runs, since this research began, it has been very clear that results are better when testing on larger problems. While there are several reasons for this, the most obvious is simple leverage. Regardless of how small the advantage is, as GPU code runs for longer periods of time, the benefits are multiplied and grow exponentially. For GPU code that is slightly faster in the short-term, that same advantage will be compounded and enhanced in the long-term.

The most important piece of evidence in support of this claim comes from the distributed-graph results for SSCA2. From the relative timescales of the three significant kernels (Figures 3.11 and 3.14), it is apparent that Kernel 3 is the least taxing (milliseconds), Kernel 1 is moderately expensive (seconds), and Kernel 4 involves the most work (minutes). With this in mind, Kernel 3's GPU timing trends remain roughly constant and the total timings actually increase, indicating that this kernel with the smallest workload is relatively inefficient. Meanwhile, Kernel 1 and its increased workload perform better, showing the expected halving behavior for its GPU timings but only constant behavior in the total timings. Finally, Kernel 4, with the most work to do, shows the expected behavior for both sets of data. This evidence suggests that for kernels with increased workloads, the GPU code does in fact become more efficient. This same trend is also evident to a lesser extent in the full-graph timings shown in Figure 3.8. Here, Kernels 1 and 3 require less work (milliseconds), and show constant behavior in their timing trends. Kernel 4 is computationally harder (seconds), and shows linear speedup as the number of processors is increased.

While the SSCA2 results may be the most significant, maybe the most obvious example of this is seen in the comparison of the multi-GPU timings to the single-GPU timings for the mesh partitioning code. While these data points never reach the expected trend, Figure 3.7 shows that as mesh size increases, they become closer and closer to the ideal values. Even though the results for the two largest cases is similar, these meshes are still relatively small, and it is believed that for appropriately larger meshes, relative performance would continue to approach the projected relationship. Again, while this may be the most obvious piece of evidence, the relatively small sizes of the test cases make this less significant than the SSCA2 results, which tested graphs of several million edges and nodes.

Figures 3.10, 3.13 and 3.16, comparing large- and small-SCALE results for the full-graph and distributed-graph codes, also indicate that performance improves for larger problems. In these plots, as SCALE size increases, timings increase as well, producing the upward-sloping trends. While the large-SCALE timings should take two (or four, eight, etc.) times as long as the baseline case, if they run faster than this, data points will appear below the expected value. In all figures, this is the case, with the majority of data points – and particularly those for higher SCALE sizes – appearing below the dashed line illustrating the ideal trend. Figure 3.10 shows that results for the largest problem sizes took roughly half the expected amount of time. Particularly for this plot, and for the total timings for the distributed-graph results, as the size of the graph continues to grow, the plots continue to drop farther and farther below the expected values, indicating that performance improves with increasing problem size.

Perhaps the most subtle example reinforcing the connection between performance and problem size comes from Figure 3.15 comparing total timings from the multi-GPU and baseline-GPU tests of the distributed-graph SSCA2 code. In these results, the slopes of the final segments for each SCALE size appear to decrease, as the problem size becomes larger. Although it may not be appropriate to compare the plots of SCALE sizes which used different baseline tests, for SCALEs 21-24, which all use their 2-GPU test as the baseline, the results are intriguing. The figure clearly shows that in the final segments of these plots, the slopes decrease for larger SCALE sizes, indicating better performance. It seems logical that with enough memory, this trend would continue, indicating an increase in efficiency for larger problem sizes.

4.3 Performance and Number of Processors

The second significant conclusion which can be drawn from these results is that *performance scales with the number of processors used*. While this may seem intuitive, it is not always the case, and depends heavily on the design of the algorithm and amount and type of MPI communication. For the most part, more processors means less work for each, resulting in improved performance. The codes tested by this research exhibited this quality in several different scenarios.

The best example of this relationship comes from the full-graph results of the SSCA2 code. As stated in chapter two, Kernel 4 uses MPI in an extremely limited fashion, splitting up BC nodes evenly between processors. The plots for Kernel 4 in Figure 3.8 show the ideal scaling as the number of GPUs is increased. This fact is reinforced in the multi-GPU vs. single-GPU comparison shown in Figure 3.9, as all plots lie almost exactly on top of the expected halving trend. For this example, where results reflect only time spent during GPU computation, the results show an excellent connection between additional processors and improved performance.

Although perhaps not as impressive as the full-graph Kernel 4 results, the data for the distributed-graph code supports this conclusion as well. Again, the most significant timings are those of the GPU computation, and Figure 3.11 shows the results for the first and fourth kernels steadily and consistently decreasing as more processors are used. Even in terms of total time, the Kernel 4 data continues to drop when as many as 16 GPUs are used (Figure 3.14). An interesting trend takes place when in the transition from 16 to 32 processors, as these timings actually begin to increase. While this would seem to contradict the notion that more processors improve performance, because this trend is not seen in the GPU computational times, it is believed that this is caused by other parts of the code which contribute to total time – most likely, MPI communication.

Even in Figures 3.13 and 3.16, showing the large- vs. small-SCALE plots for the distributed-graph Kernel 4 code, the results strengthen the argument that more processors are beneficial. As stated in the preceding section, for these plots, the farther a data point lies *below* the expected trend line, the better it performs in relation to the baseline test. These plots – meant to highlight performance improvements in relation to problem size – clearly show the 32-GPU trials as being farthest below the expected trend. The next best results belong to the 16-GPU cases, then the 8-GPU cases.

Finally, the results for the mesh smoothing and mesh partitioning codes also reflect this same relationship. In Figures 3.1 and 3.6, as the number of GPUs increases, timings are reduced considerably. Figures 3.2 and 3.7 divide multi-GPU timings by the single-GPU results. Again, if the multi-GPU tests run faster than expected, the data points will appear below the expected values. The first figure shows smoothing plots actually fall far below the expected values. While the partitioning results are not as impressive, the fact remains that adding more processors still steadily improves performance.

4.4 Performance and MPI Communication

While MPI is a necessary part of parallel computing, and makes working on distributed-memory machines possible, there is no question that *performance scales* *inversely with the amount of MPI communication*. Just like the realization that performance is heavily dependent on problem size, it became clear early on that MPI is quite costly, even when used sparingly. For true distributed-memory applications, like the mesh adaptation codes and distributed-graph version of SSCA2, the difference in *total* times between single-GPU and 2-GPU tests is staggering. With the addition of even just a second processor, data that was previously read or copied locally now must be retrieved from the GPU, communicated via MPI, and copied back into the GPU's memory. Operations that cost little or no time at all, now require huge amounts of time.

While the best evidence of this fact is a simple comparison of single- and 2-GPU test times, the differences in Kernel 4's GPU and total timings for the distributed-graph code (Figures 3.11 and 3.14) are also significant. As stated in the previous section, it is believed that the reason for the uptick in the 32-GPU total timings is due to the inclusion of the MPI communication. When MPI and data transfer times are excluded, and only GPU computation is considered, the results display the expected behavior, roughly halving each time the number of processors is doubled. It is only when the non-GPU times are included that the undesired behavior is seen. The fact that the problem only starts to occur as the number of processors becomes *very* large would indicate that it is likely related to the exponential growth of required MPI communication. Simply put, it seems clear that the increase in total timings for highly-distributed tests is due to the slowdown incurred by the corresponding jump in MPI.

Along this same vein, a closer inspection of the GPU and non-GPU percentage plots in Figures 3.17 and 3.18 suggests that the non-GPU operations are in fact responsible for the decrease in performance for highly-distributed test runs. The amount of time required for non-GPU operations for the 32-GPU case is appreciably larger than the amount for cases using fewer processors. The latter figure shows that the 32-GPU case spends more than 98% of the computational time performing MPI and data transfer operations, while other cases average only around 96-97%. While this may not seem like a significant difference, Figure 3.17 shows just how important this is when looked at from the perspective of GPU computational time. Here, the results show that the GPU percentage of total time is roughly halved when using more than 16 GPUs. Furthermore, the comparison of non-GPU to GPU timings in Figure 3.19 also indicates that roughly twice as much time is spent on MPI and data transfer when using 32 GPUs. In short, using a large number of processors increases MPI and data transfer timings considerably.

Even though a direct comparison between timings is inappropriate, the different trends shown in the Kernel 3 full-graph and distributed-graph results support the idea that MPI is quite costly. In the full-graph code, which includes no MPI or data transfers, the performance shown in Figure 3.8 remains roughly constant even for large numbers of processors. On the other hand, Figures 3.14 and 3.15 show the distributed-graph version of the code exhibiting poorer results in terms of total time as more processors, and therefore more MPI, are added. Like the Kernel 4 code, the distributed-graph Kernel 3 MPI and data transfer operations again take roughly 70-90% of the total time. It is difficult to ignore this when looking for factors that could contribute to the Kernel 3 behavior seen in Figure 3.14. It seems quite likely that MPI is a large reason for the reduced performance of highly-distributed test cases.

As one final example showing that MPI is detrimental to performance, Figure 3.15 shows performance for Kernel 4 become considerably worse when using 16 and 32 processors. Again, although the direct comparison of timings would be invalid, the trends shown by the full-graph code (including no significant MPI or data transfers) in Figure 3.9 follow the expected values perfectly. With the most significant difference being the inclusion of a large amount of non-GPU operations, this strongly supports the idea that the poor performance of the distributed-graph code for large numbers of processors is due to large amounts of MPI and data transfer.

4.5 Mesh Smoothing Conclusions

In reviewing the final overall performance, it is the author's opinion that *the smoothing code is not fully optimized and could be improved upon*. The good news is that the loop to add cellular contributions to nodal sums was successfully parallelized on multiple GPUs, and the results for this kernel (Figures 3.1 and 3.2) are appropriate. Still, the costs of the MPI associated with this parallelization are similar to those seen in the SSCA2 code. Even for only two processors, and even in this computationally-heavy algorithm, the time spent on MPI dwarfs the time for GPU calculations. Furthermore, the remainder of the code – and particularly the loop to calculate cellular residual contributions – remains unparallelized. As stated in chapter two, it is possible to modify other parts of the code for use on multiple processors, but because of the fact that all GPUs need access to all data, even more MPI would have to be included. Lastly, and perhaps most importantly, although the current algorithm proved to be the most efficient option during initial testing, those tests were only conducted on a single processor. The considerable memory requirements and avoidance of atomics may not be the best options when considering the additional cost of MPI. It is because of the need for atomics, and the hesitancy to use them, that the second step of the algorithm was split into two loops. Relative to the expense of MPI, atomics are not nearly as intimidating. Combining these steps back into one, and parallelizing the entire second step might be more efficient than if both of the split loops are parallelized separately.

4.6 Mesh Partitioning Conclusions

Relative to the smoothing algorithm, the partitioning code is much simpler. It is because of this fact that *the parallelization of the partitioning algorithm is efficient, and likely cannot be improved upon significantly*. As with the smoothing code, there are multiple loops operating over different indices, and this means a large amount of MPI communication after any distributed step. However, unlike the smoothing code, there seems to be no other obvious solution to this problem. The most significant amount of work is the loop assigning nodes to partitions, so this is loop is distributed. All processors need access to updated partition data from the second kernel, making communication between these steps unavoidable.

4.7 SSCA2 Conclusions

The obvious question about the two SSCA2 algorithms is to ask which version performs better. While both have their merits, in the opinion of the author, *the distributed-graph version of the SSCA2 code is the more appropriate and insightful of the two algorithms*. The purpose of this research was to investigate the viability of GPU-based supercomputing. While the full-graph code shows impressive performance, and follows the expected trends almost exactly, these results come as no surprise.

Because MPI is used in this algorithm in such a minor way, the code is still basically all GPU-based. Parallelizing this one-dimensional task should certainly produce the expected result. On the other hand, the distributed-graph code is much more complex, and really displays the true relationships between the GPU code, MPI communication and data transfers. This algorithm paints a much more accurate picture of what supercomputing on GPU-based machines really looks like.

Not only is the distributed-graph code a more truthful representation in terms of relative timings, it also is more accurate in terms of capability. For the distributed-graph code, *problem size is limited only by the number of GPUs being used*. The full-graph code has a firm upper limit on problem size, tied directly to the amount of memory on each GPU. Although using a different processor can change this limit, even those with large amounts of memory can’t go much higher than SCALE 20. For the trials run on Lincoln, SCALE 27 was the largest graph tested, but that was only for 32 GPUs. Increasing to 64- or 128-GPU tests would extend the upper limit to SCALE 28 or 29, respectively. This true scalability is a crucial factor when evaluating massively-parallel architectures, and is an essential trait that is not shared by the full-graph code.

REFERENCES

1. Ferziger, J.H. and M. Peric, *Computational Methods for Fluid Dynamics*. 3rd ed. 2002, Berlin: Springer. 22.

2. Moore, G.E., *Cramming more components onto integrated circuits.* Electronics, 1965. **38**(8).

3. Thompson, S.E. and S. Parthasarathy, *Moore's law: the future of Si microelectronics* Materials Today, 2006. **9**(6): p. 20-25.

4. Kish, L.B., *End of Moore's law: thermal (noise) death of integration in micro and nano electronics.* Physics Letters A, 2002. **305**(3-4): p. 144-149.

5. Kim, N.S., et al., *Leakage current: Moore's law meets static power.* Computer, 2003. **36**(12): p. 68-75.

6. Wulf, W.A. and S.A. McKee, *Hitting the memory wall: implications of the obvious.* Computer Architecture News, 1995. **23**(1): p. 20-24.

7. *NVIDIA CUDA Programming Guide*. Version 2.2.1. 2009.

8. Reddy, J.N., *An Introduction to the Finite Element Method*. 2nd ed. 1993, New York: McGraw-Hill. 7-8.

9. Dompierre, J., et al., *Anisotropic mesh adaptation: towards user-independent, mesh-independent and solver-independent CFD. Part III. Unstructured meshes.* International Journal for Numerical Methods in Fluids, 2002. **39**(8): p. 675–702.

10. Habashi, W.G., et al., *Anisotropic mesh adaptation: towards user-independent, mesh-independent and solver-independent CFD. Part I: general principles.* International Journal for Numerical Methods in Fluids, 2000. **32**(6): p. 725-744.

11. Blom, F.J., *Considerations on the spring analogy*. International Journal for Numerical Methods in Fluids, 2000. **32**(6): p. 647-668.

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

13. Bottasso, C.L., 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, 2005. **194**(39-41): p. 4244-4264

14. Farhat, C., et al., *Torsional springs for two-dimensional dynamic unstructured fluid meshes.* Computer Methods in Applied Mechanics and Engineering, 1998. **163**(1-4): p. 231-245.

15. Degand, C. and C. Farhat, *A three-dimensional torsional spring analogy method for unstructured dynamic meshes.* Computers & Structures, 2002. **80**(3-4): p. 305-316.

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

17. Acikgoz, N., *Adaptive and Dynamic Meshing Methods for Numerical Simulations*, in *School of Aerospace Engineering*. 2007, Georgia Institute of Technology: Atlanta, Georgia. p. 146.

18. Bern, M., D. Eppstein, and J. Gilbert. *Provably Good Mesh Generation*. in *Foundations of Computer Science*. 1990. St. Louis, Missouri.

19. Edelsbrunner, H., et al. *Smoothing and cleaning up slivers*. in *ACM Symposium on Theory of Computing*. 2000. Portland, Oregon: Association for Computing Machinery.

20. Talmor, D., *Well-Spaced Points for Numerical Methods*, in *School of Computer Science*. 1997, Carnegie Mellon University: Pittsburgh, Pennsylvania.

21. Freitag, L.A. and C. Ollivier-Gooch, *Tetrahedral mesh improvement using swapping and smoothing.* International Journal for Numerical Methods in Engineering, 1998. **40**(21): p. 3979 - 4002.

22. Jones, M.T. and P.E. Plassmann, *Adaptive refinement of unstructured finite-element meshes.* Finite Elements in Analysis and Design, 1997. **25**(1-2): p. 41-60.

23. Carey, G.F., *A mesh-refinement scheme for finite element computations.* Computer Methods in Applied Mechanics and Engineering, 1976. **7**: p. 93-105.

24. Basu, P.K. and A. Peano, *Adaptivity in P-Version Finite Element Analysis.* Journal of Structural Engineering, 1983. **109**(10): p. 2310-2324.

25. Babuska, I., B.A. Szabo, and I.N. Katz, *The p-Version of the Finite Element Method.* Society for Industrial and Applied Mathematics, 1981. **18**(3): p. 515-545.

26. Ghosh, D.K. and P.K. Basu, *A parallel programming environment for adaptive p-version finite element analysis* Advances in Engineering Software, 1998. **29**(3-6): p. 227-240

27. Löhner, R., *Mesh adaptation in fluid mechanics* Engineering Fracture Mechanics, 1995. **50**(5-6): p. 819-831.

28. Farhat, C., *A simple and efficient automatic FEM domain decomposer* Computers and Structures, 1988. **28**(5): p. 579-602.

29. Farhat, C., S. Lanteri, and H.D. Simon, *TOP/DOMDEC—A software tool for mesh partitioning and parallel processing.* Computing Systems in Engineering, 1995. **6**(1): p. 13-26.

30. Walshaw, C. and M. Cross, *Parallel optimisation algorithms for multilevel mesh partitioning* Parallel Computing, 2000. **26**(12): p. 1635-1660

31. Walshaw, C., et al., *Multilevel Mesh Partitioning for Optimizing Domain Shape.* International Journal of High Performance Computing Applications, 1999. **13**(4): p. 334-353.

32. Walshaw, C., M. Cross, and K. McManus, *Multiphase mesh partitioning* Applied Mathematical Modelling, 2000. **25**(2): p. 123-140

33. Karypis, G. and V. Kumar, *METIS Manual, Version 4.0*. 1998: University of Minnesota, Department of Computer Science.

34. Karypis, G., K. Schloegel, and V. Kumar, *PARMETIS Manual, Version 3.1*. 2003: University of Minnesota, Department of Computer Science and Engineering.

35. Owens, J.D., et al., *A Survey of General-Purpose Computation on Graphics Hardware.* Computer Graphics Forum, 2007. **26**(1): p. 80-113.

36. Mark, W.R., et al. *Cg: a system for programming graphics hardware in a C-like language*. in *International Conference on Computer Graphics and Interactive Techniques*. 2003. San Diego, California: Association for Computing Machinery.

37. Kessenich, J., *The OpenGL Shading Language*. 1.50.09 ed, ed. D. Baldwin and R. Rost. 2009.

38. Bolz, J., et al. *Sparse Matrix Solvers on the GPU: Conjugate Gradients and Multigrid*. in *International Conference on Computer Graphics and Interactive Techniques*. 2003. San Diego, California: Association for Computing Machinery.

39. Goodnight, N., et al. *A multigrid solver for boundary value problems using programmable graphics hardware*. in *SIGGRAPH/EUROGRAPHICS Conf. On Graphics Hardware*. 2003. San Diego, CA, Assoc. for Computing Machinery.

40. Krüger, J. and R. Westermann. *Linear algebra operators for GPU implementation of numerical algorithms*. in *International Conference on Computer Graphics and Interactive Techniques*. 2005. Los Angeles, California: Association for Computing Machinery.

41. HPCS, *HPC Scalable Graph Analysis Benchmark, Version 1.0*, <www.graphanalysis.org/benchmark/GraphAnalysisBenchmark-v1.0.pdf>, D.A. Bader, et al., Editors. 2009.

42. HPCS, *Scalable Synthetic Compact Application (SSCA) Benchmarks*. <www.highproductivity.org/SSCABmks.htm>, 2007, HPCS.

43. Brandes, U., *A faster algorithm for betweenness centrality*. Journal of Mathematical Sociology, 2001. **25**(2): p. 163-177.

44. Bader, D.A. and K. Madduri. *Parallel Algorithms for Evaluating Centrality Indices in Real-World Networks*. in *Parallel Processing*. 2006.

45. NVIDIA, *Developer Zone – CUDA Toolkit 2.2*, <http://developer.nvidia.com/object/cuda_2_2_downloads.html>, 2009.

46. Weller, H., et al. *OpenFOAM – The Open Source Computational Fluid Dynamics (CFD) Toolbox*, < http://www.openfoam.com/ >, 2010.

47. NCSA, *NCSA Scientific Computing: Intel 64 Tesla Linux Cluster Lincoln*, <www.ncsa.illinois.edu/UserInfo/Resources/Hardware/Intel64TeslaCluster/>, 2009.