

# GRiTS: Global Re-indexing for Triangular (Tetrahedral) Simplices

Grant Boquet

May 03, 2004

## Abstract

In this paper, we describe and analyze a number of index filters associated with graph partitioning. A heuristic filter based on two level adjacency and random sampling demonstrated the best performance. A numerical study using a parallel finite elements flow solver (ViTLES) and the parallel graph partitioning package ParMETIS is included.

## Introduction

Finite Elements is a popular method of finding numerical solutions to PDEs (Partial Differential Equations). By projecting an infinite dimensional solution onto a finite dimensional solution space, one is able to reduce the problem to one of solving nonlinear algebraic equations (which can be solved by Newton's method). Increasing CPU clock speeds and more efficient algorithms reduce the required cycles for an instruction, thus network latency is remaining the bottleneck in parallel FEM. Load balancing and network latency are problems that one must overcome in order to make parallel computation efficient. One way of doing this is to partition the Finite Element Mesh into  $k$ -zones so that each processor has an "equal" number of tasks to perform and the network use is minimized. By using the graph partitioning software ParMETIS and forming a set of mappings between a hypergraph and a Finite Elements Mesh, optimal load balancing can be achieved. The purpose of this paper is present how ParMETIS can be used to partition a Finite Element Mesh. This requires the study of how to project the resulting graph onto a FE mesh. Numerical results show the performance gain from using our software implementation, GRiTS (Global Re-Indexing for Triangular (Tetrahedral) Simplices) will be shown along with several pictures

demonstrating the various algorithms. Computational time, MPI messages, and graphical illustrations of the partitioned meshes are presented.

## Background of Finite Elements

Much of nature can be interpreted mathematically in continuous models. Of such models, differential equations prove to be quite prominent. From describing fluid flow, stress in a beam, force in an electric field, PDEs enable mathematicians to understand nature. However, unless the PDE is simple and the domain is simple, analytic solutions do not exist or are hard to come by.

Once computers started becoming available, numerical methods could be easily employed by researchers to find approximations that were quite accurate to these differential equations. Starting in the 1960s, the theory Finite Elements had set its roots in being understood mathematically. The method allows finding an approximation of a PDE to eventually be reduced to finding a solution to a linear system. This is done by projecting an infinite-dimensional solution onto a finite set of “test functions.” Some two dimensional information is demonstrated here to give the origin of the problem. For more information about FEM and it’s origin, see [6].

Consider the heat equation below,

$$\begin{aligned} u_t &= \nabla \cdot (p \nabla u) + f, & p : \mathbb{R}^2 &\rightarrow \mathbb{R} \\ u(0, y, t) &= 0, & u(1, y, t) &= 0 \\ u_y(x, 0, t) &= 0, & u_y(x, 1, t) &= 0 \\ u(x, y, 0) &= b(x, y). \end{aligned}$$

For a simple domain, an analytic solution exists using, e.g. separation of variables. However, for a more complicated case, this may not be true. Therefore, we turn to numerical solutions such as FEM that partitions the domain into triangular elements. In light of such, we must now have a discrete domain. In other words, the domain has to be broken into pieces. An example is in Figure 1

On such a small domain, parallel computation isn’t required. However, as an example its quite illustrative. In Figure 2, we present two cases for partitioning the mesh. Both partitions have the same number of elements, but the partition on the right minimizes shared edges. While load balancing is the same, communication is better with the one on the right.

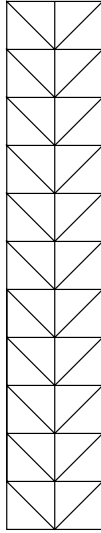


Figure 1: Discretized rod using triangular elements.

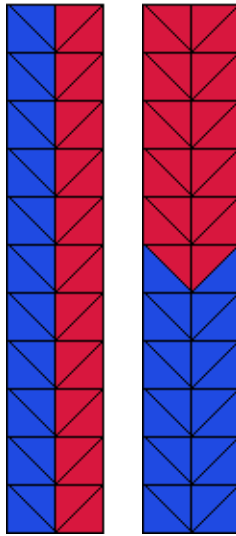


Figure 2: The left picture is a bad partition for two processors, the right is the optimal case.

Thus, the problem discussed in this paper is to find the optimal (or close) partition in a reasonable amount of computational resources.

## Parallel Finite Elements

Starting in the mid 1990s, as network speeds became more reasonable, clustered computing starting achieving popularity. As recently demonstrated by Virginia Tech, for a small sum of money, a powerful computer could be built. So began clustered computing. In the study of Finite Elements, it is one of the key features that computation and analysis is done on the element. Known as the Finite Element Assembling Procedure, it allows for a systematic way of building a large linear system piece by piece. This, by nature, is easily done in parallel. Moreover, iterative methods in finding solutions to linear systems can be done in parallel with such methods as GMRES. Thus began the start of parallel finite elements.

Finite elements requires a discrete domain, which involves breaking the domain into pieces (which may not be of the same type). In doing so, one could also determine which element should go to a specific processor. Such action is called producing a *Distributed Mesh*. This concept of evenly distributing work has been studied extensively in discrete situations which involves optimizing processing ordering, bin packing, and other forms. However, this was normally only done in terms of optimizing with respect to minimization of idle time. The problem posed is optimizing with respect to both idle time and the topology of the mesh. This problem though is too complicated, even on the discrete scale, to optimize completely in an efficient manner.

One common pattern in mathematics is employing the tools of a different field to find a solution by relating a problem to a different problem. By following this concept and relating a problem in computational geometry to graph theory, an approximate solution can be found that produces efficient results. This is done by breaking a continuous domain into a graph, optimizing the graph to have an even cardinality of vertex sets for each of the disjoint subgraphs it produces. Moreover, by minimizing the intersected edge set of each graph, network communication can be minimized. The following sections explain how to start with a finite element mesh, and in return have an optimized distributed mesh.

## Producing an Adjacency Graph

ParMETIS, a package for graph partitioning, is designed on the requirement of a structure called an adjacency graph. This section introduces the reader to the concept, and presents a simple algorithm for producing one. The first step is to form one; which will be covered in some detail starting with what a graph is. More details about graphs can be found in [3], [1].

**Definition 1** *A graph  $G = (V, E)$  is defined as a set of vertices  $V$  and a set of edges  $E$ , where each edge is a subset of the vertex set  $V$ .*

Although ParMETIS is designed to work with hypergraphs, for the context of this paper, regular graphs are more than sufficient. In application, such as in the numerical simulation, hypergraphs were used to include several vertices in one edge (for quadratic elements). The methods in this paper remain the same in this context, but for simplification are left out. Of these graphs, we work in particular with a special graph called a zoned-graph.

**Definition 2** *A zoned-graph  $Z = (V, E, N)$  is a graph in which  $N \subset T$  is the zone set. An element of  $N$  is called an index.*

The only difference is that each vertex is marked with a Turing machine in which is it destined to be sent to.

**Definition 3** *Define a cluster as a set of Turing machines  $T_i = (Q, \Sigma, \Gamma, q_0, \delta)$ , denoted by  $T$  in this paper.*

Concepts of computability are not relevant in this paper, just the notion that a cluster is composed of objects that can perform a set of calculations. The first step is to relate a finite element mesh to a graph, so that a graph can be formed out of the mesh. This involves, initially, working with a continuous domain. In finite elements, the domain is discretized in both space and time. However, in terms of what one would distribute, objects that are not in two or more dimensions are trivial. It is important to relate the concept of a mesh to a graph, which have similar properties. Therefore, the following definition is required.

**Definition 4** *A finite element graph is a graph  $F = (V, E)$  where every element of  $V$  corresponds to an element  $e_i$ . Moreover,  $\cup_i e_i = \Omega$  where  $\Omega$  is the weak form's domain. (See Figure 3).*

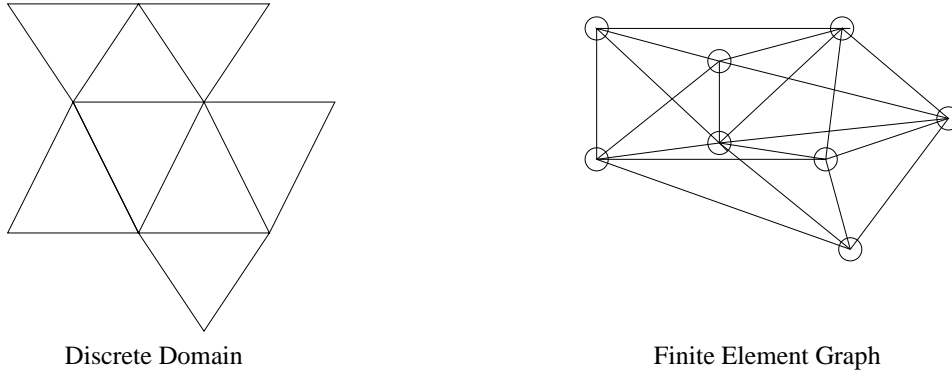


Figure 3: An example of a finite element graph and its discrete domain.

A weak form is the variational form of the posed PDE. Its domain is composed of the domains that the mesh is generated on, be it space, or space-time. This allows us to finally reach the next definition, which is required to use the ParMETIS package.

**Definition 5** *An adjacency graph  $A$  is a set of subgraphs of a graph  $X$  in which each subgraph,  $A_i$ , is induced by  $V = \{\cup_i E_i \cap v\}$  for  $E_i \in E$ .  $v$  is the center of  $A_i$ .*

GRITS is designed to work with, currently, the ViTLES (Virginia Tech Large Eddy Simulation) file format which incorporates within it a connectivity graph. This corresponds to the connectivity of each element, and thus each node of a finite element graph. This is important in the algorithm below.

Given a FE mesh,  $\Omega_h$ , there corresponds a set of “complete” subgraphs to each vertex. Each subgraph may not be complete because for the case the elements are interpolated with quadratic polynomials, there are vertices in between the main vertices of the triangle. However, because of their location, it may as well be complete because the structure is maintained if they are discarded. This is important in forming an algorithm for constructing an adjacency graph because some assumptions about the connectivity can then be made.

**Lemma 1** *Let  $Y(V_v, E_v)$  be the graph induced by  $v \in V$  for a given graph  $X(V, E)$ . If the valency of  $v > 1$ , then for any distinct  $a, b \in V_v \setminus v$ ,  $d(a, b) = 2$ .*

Proof. By definition of an adjacency graph, given two vertexes  $a$  and  $b$ , there is a path from  $a$  to  $b$ . Take  $(a, h_1), \dots, (h_n, b)$  as path. By construction of  $Y$ , all elements in the edge set contain  $v$ . Therefore, the shortest path from  $a$  to  $b$  is  $(a, v), (v, b)$ , which is length 2.  $\square$

The previous result is used to justify the following algorithm for producing an adjacency graph from a graph. It requires a graph and a list of all the vertices which correspond to nodes in a finite element mesh. Note that this is done on each processor in parallel.

### Algorithm 1

Input:  $X(V, E), K = \{K_1 = (a_1, b_1, c_1), K_2 = (a_2, b_2, c_2), \dots, K_n = (a_n, b_n, c_n)\}$   
Output:  $Y = \{X_1(V_1, E_1), \dots, X_n(V_n, E_n)\}, |Y| = |V|$

```

V1 = V2 = ... = Vn = ∅
for i = 1 ... |K|
    Vai = Vai ∪ {(ai, bi), (ai, ci)}
    Vbi = Vbi ∪ {(bi, ai), (bi, ci)}
    Vci = Vci ∪ {(ci, ai), (ci, bi)}
end for

```

By Lemma 1, this algorithm produces an adjacency graph. It relies on the fact that a particular node is not connected to something that is more than a distance of 1 away. Hyperedges are generalized to several regular edges to follow with this algorithm. Thus for each vertex, find the set of elements that contain it, then form the union of all nodes in these elements. This will produce the adjacency graph.

## ParMETIS

Once we have this adjacency graph, ParMETIS's  $k$ -way partitioning algorithm can be called to produce a zoned graph. A more detailed reference to the algorithm can be found in [4]. It is a multi-level algorithm, which means that it coarsens the graph several times, and then fills in the pieces several times. There is a good relationship to assembling a puzzle with two people, John(Processor 1) and Jane(Processor 2). Keep in mind that ParMETIS

is designed to partition graphs in parallel, which means that each processor has associated with it a number of vertices, which may have edges that contain vertices on other processors.

Consider there is a domain which is labeled as in figure 4. Then if there are two people (processors), John would have the following pieces {0, 1, 2, 3, 4, 5, 6, 7} and Jane would have the other pieces, {8, 9, 10, 11, 12, 13, 14}.

1	2	7	14	11	10	6
5	12	9	13	4	8	3

Figure 4: A numbered domain.

The algorithm would, initially, coarsen the graph to where some edges are now represented by vertices. ParMETIS requires an adjacency graph, which was discussed earlier. For our example, it would be:

Piece	Connected To
1	2, 5, 12
2	1, 5, 12, 9, 7
3	8, 10, 6
4	13, 14, 11, 10, 8
5	1, 2, 12
6	10, 8, 3
7	2, 12, 9, 13, 14
8	4, 11, 10, 6, 3
9	12, 2, 7, 14, 13
10	4, 11, 10, 6, 3
11	13, 14, 4, 8, 10
12	5, 1, 2, 7, 9
13	9, 7, 14, 11, 4
14	7, 9, 13, 4, 11

ParMETIS now has to look at where each piece is in the global scope, which uses the above table. Each person has a copy of what they are hooked to globally. This means that John would know that piece 1 is connected to piece 12, even though piece 12 is not in his pile. This requires communication before calling ParMETIS in constructing these adjacency graphs.

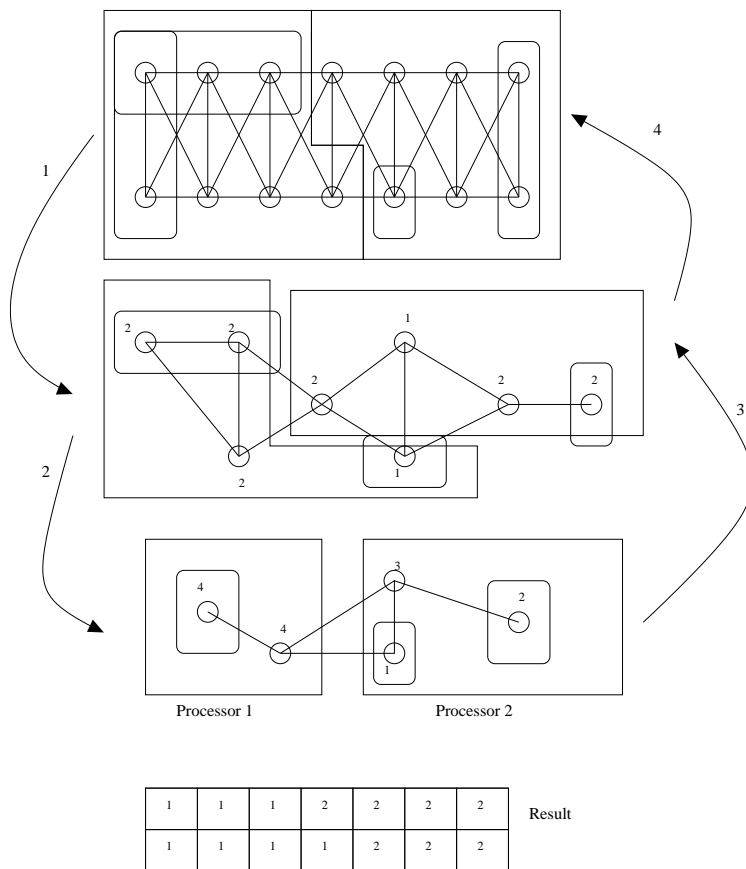


Figure 5: Coarsening phase in ParMETIS. boxed portions with rounded edges are for processor 1, the remaining are for processor 2. The sharp edge boxes denote the partitioned parts.

ParMETIS then takes this information and uses it to first coarsen the graph. The coarsened graph sequence can be seen in figure 5. In the initial graph, each vertex has a “weight” of one. Then after step one, some of the edges are replaced by vertices, and their new weights are the sum of the vertices they encompass. This is repeated again, leaving 5 vertices, each with different weights.

John and Jane now randomly pick a “piece” which is no longer a real puzzle piece, but a marker for several pieces. Say John picks the piece on the far left, and Jane picks the one on the far right. Then they look at an

adjacency graph to see what pieces their piece is connected to. They look through the pile to get it. The piece may be in another person's pile, in which case they may exchange if the sizes of their piles is not equal. Once each piece is taken, they try to assemble their parts of the puzzle by finding the pieces that the marker piece points to. Then after this is done, they realize they have an unequal distribution, John has one more piece than Jane. So John switches with Jane, he gives her two pieces for one. This is seen in step three. Finally, all the puzzle is assembled, and the finished product is made after step four.

In the end, a much smaller graph is found. This is then partitioned. This information is returned to the previous graph, optimized, and then returned to the previous graph. The method allows vertices for each processor to be connected, yet still balance them. Their algorithm is more complex, allowing for weighted vertices, and for a certain degree of flexibility in how it distributes the nodes. However, such an example easily shows the power and simplicity of the  $k$ -way partitioning algorithm.

## Creating Distributed FE Mesh

ParMETIS returns a zoned-graph, which is an overdetermined system. In many cases, multiple optimal partitions may exist (See Figure 6). Because we are trying to form an index for an element and there are associated with each element several nodes, there has to be a way of filtering the data to make a wise choice. Creating a systematic way of making a decision as to which processor an element should be sent to had to be studied. In order to perform this, the bulk contribution of this research follows in what are called a set of index filters. These were created to allow one to map this graph back to a mesh, yet still have control over what gain is desired, be it reduction network use or reduction in overall time. More over, it allows this process of optimizing a mesh to be implemented in a systematic way so that it can be carried out over more complex domains than studied here.

**Definition 6** *An index-filter performs the mapping  $\phi : T^n \rightarrow T$ .*

In our situation, Turing machines are denoted by numbers. Therefore, the following algorithms work as mappings from a set of natural numbers to a particular number. Before progressing to each index-filter, we cover what a function named MCI (Most Common Index) is, see the attached code in Figure 7.

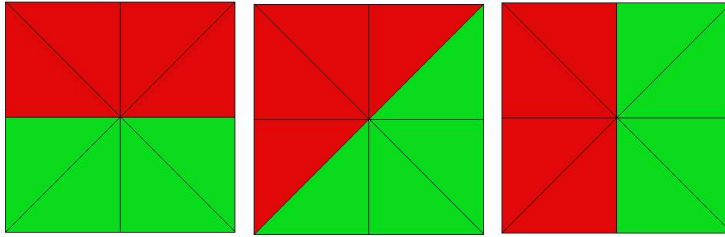


Figure 6: Multiple optimal solution example.

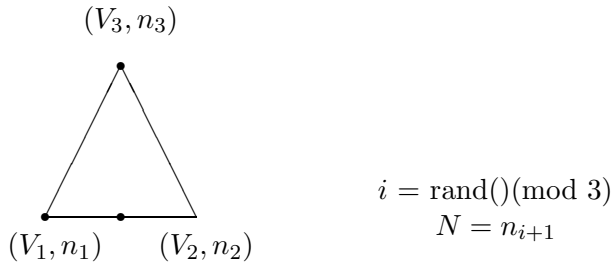
This function takes a list of natural numbers, performs a sort on them, and picks the number that appears the most. In the situation that the function is not able to make a decision, when there are equal number of particular indices, then it returns false. In this case another algorithm has to be used, or the set of indices has to be changed. Each index filter below keeps this in mind so that a result will always follow.

### Five Index-Filters

GRiTS has five index-filter implementations, each unique in outcome. For each element, for the case that they are triangles, there are three pieces of data associated with each element. This makes the decision process somewhat complicated. What follows are each of the methods for making a decision as to what is the optimal index to choose. For some of the index-filters there are accompanying illustrations of their performance on producing the corresponding mesh.

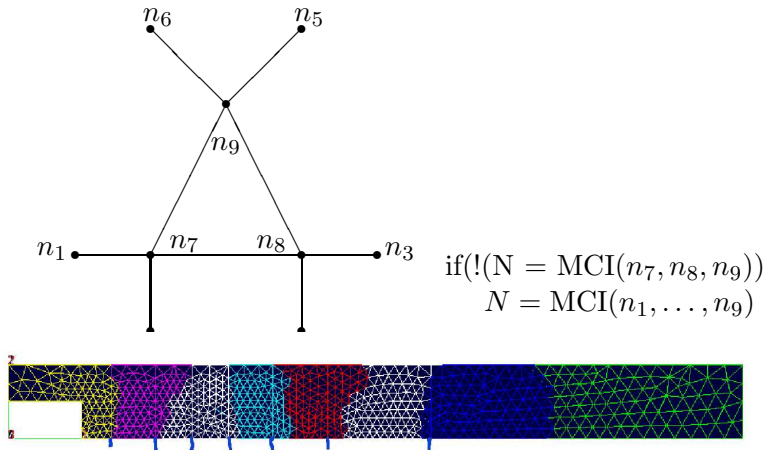
#### Index-Filter 1, Random

The random index-filter just blindly picks an index of the three and sets the element index to this value. Although this seems rather crude, it does work in favor of the index that appears the most.



**Index-Filter 2, 1-Element MCI (Most Common Index) with Adjacent Smoothing Split Support**

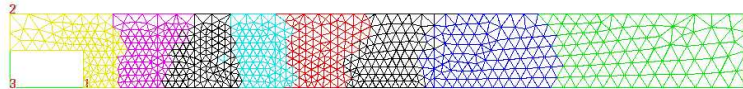
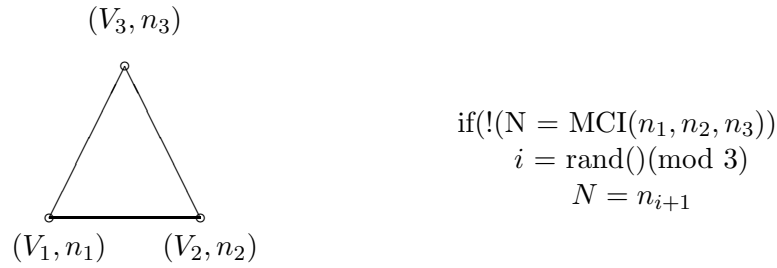
This index-filter finds the index within the element that appears the most. In the case that the element is split (equal number of indicies for two or more indicies) then it uses the adjacent elements to make a choice. It does this by finding the most common index of the vertices in the element, and the neighboring elements. The illustration below shows that some intrusion into a neighboring zone is occurring. The lines below the mesh show where the initial or “eyeball” distribution were.



**Index-Filter 3, 1-Element MCI with Random Split Support**

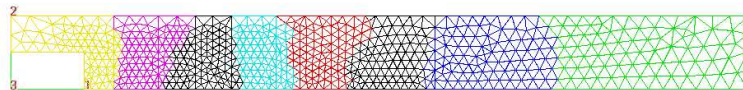
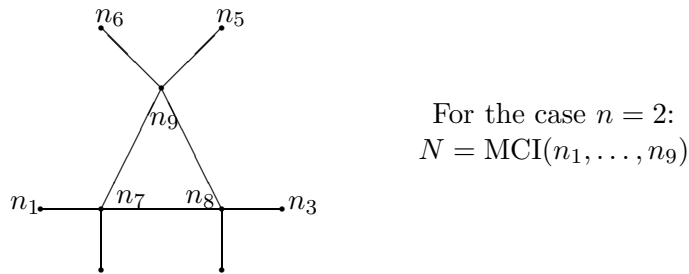
Similar to the above index filter, this one instead randomly makes a choice in the case that it is not able to the first time. This may result in elements that are disconnected from the “main” part for a particular processor. However, it allows for better load balancing. Based on the topology of the graph, a particular processor may win more elements because of how the elements are

distributed. The illustration below shows that the intrusion into neighboring territory is still taking place.



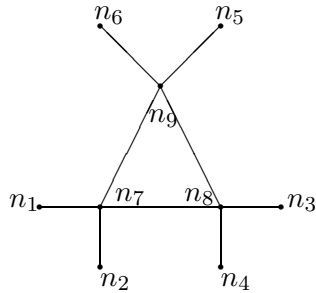
### Index-Filter 4, $n$ -Adjacent-Element MCI

In order to produce a mesh with nice borders and reduce messaging, considering  $n$  adjacent elements appears to be an obvious solution. This would significantly reduce jagged edges, elements penetrating neighboring zones, and other problems that would arise in messaging. However, this may result in unequal distributions which could increase processor idle time. This illustration shows that the edges are much smoother in comparison to the previous edges and that there are not elements pushing their way into neighboring areas.



### Index-Filter 5, 1-Element MCI with Adjacent Smoothing Split Support and Random Samples

This index filter is like index-filter two, however, it will perform a check to see if a random internal vertex has the same index as the one determined by MCI. Two element adjacent smoothing is then used on both split elements and these randomly marked “bad” elements. This prevents elements from moving deep into neighboring zones, yet, performs at such a depth, that it prevents stealing elements based on the topology of the graph. This, as shown later, proved to perform best in numerical tests.



$$i = \text{rand}()(\bmod 3) + 1$$

$$\text{if}(! (N = \text{MCI}(n_1, n_2, n_3) \parallel N \neq n_i))$$

$$N = \text{MCI}(n_1, \dots, n_9)$$

Computational cost bounds for each algorithm are given in Table 1. Big-oh  $O(\cdot)$  notation is an upper bound in computational cost up to some constant, and big-omega  $\Omega(\cdot)$  is the lower bound in computational cost up to some constant. For more information in algorithm analysis, see [5].

Index-Filter	$O$	$\Omega$
1	$N$	$N$
2	$N^2$	$N$
3	$N$	$N$
4	$N^n$	$N^n$
5	$N^2$	$N$

Table 1: Computational cost bounds for each index-filter.

The following Lemma demonstrates that after proceeding with this algorithm, no modifications to the connectivity and the the domain are made; only additional information is created.

**Lemma 2** *A zoned-graph  $Z(V, E, N)$  is isomorphic to the graph  $Y(V, E)$ .*

Proof. Select two adjacent vertices,  $a, b \in V$ . Define  $\phi : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N} \times \mathbb{N}$  as  $\phi(a, b) = (a, b)$ . Isomorphism follows since adjacency is not affected by the zone set.

## Results

In order to test each algorithm for performance, a simulation was performed using each mesh with ViTLES. The results, which contain time, flops/s (floating point operations per second), and number of MPI messages, can be found in Table 2. It can be seen that although message reduction is not significant, that GRiTS was able to produce a 35% speed up in the computations. Most of the reduction is due to better load balancing. Differences among the different index filters (1-5) are due to combinations of load balancing, communication and iterative linear algebra (preconditioning).

Algorithm	Time(s)	flops/s	Messages
1	3902.45	2.250e8	1.095e+07
2	4063.27	2.271e8	1.151e+07
3	3880.48	2.266e8	1.097e+07
4	3894.18	2.264e8	1.100e07
5	3020.970	3.299e8	1.243e+07
“Eyeball”	4961.421	1.790e8	1.107e+07

Table 2: Computational Information

Table 3 shows the number of elements for each zone (of eight zones) followed by the standard deviation of the distribution. The standard deviation shows that index-filter 2 produced the most equal distribution.

Index Filter	P0	P1	P2	P3	P4	P5	P6	P7	$\sigma$
1	461	436	449	436	426	444	443	464	12.88
2	460	436	456	437	427	447	445	451	11.05
3	460	435	455	443	424	451	442	449	11.53
4	464	429	453	442	427	440	451	453	12.76
5	461	434	455	438	428	445	445	453	11.20
Initial	430	481	402	411	463	464	485	423	32.28

Table 3: Number of elements for each zone for eight zones followed by the standard deviation of the distribution.

Table 2 contains values in which one could determine which is the “best” algorithm, they only distinguish with respect to one variable. In the situation that one must choose a method over a set of variables, a more complex fitness function should be employed. Thus the following functions were introduced.

Fitness:

$$\beta(t, m) = \frac{C_1}{(1+t)} + \frac{C_2}{(1+m)}$$

$$C_1 + C_2 = 1 \quad C_1, C_2 > 0$$

Used Fitness:

$$\beta(t, m) = \frac{1}{2(1+t)} + \frac{1}{2(1+m)} \quad (1)$$

Minimization Function:

$$f(t, m) = \frac{1}{\beta(t, m)} - 1 \quad (2)$$

This fitness function allows for a flexibility in ranking a method with respect to both computational time and network communication. In equation (1) the one used to test the fitness of each algorithm has equally weighted time and message parts. In the case that network communication was a big issue, this could be changed to reflect that. Table 4 gives the results of each algorithm. In the case of the simulation, network communication and computational time were both equally weighted for importance. Adding further information such as flops into the fitness function would also improve one’s search for the optimal algorithm.

## Conclusions

In the search to reduce stalling and network communication it was found that of the problems presented, messaging was not as big an issue as stalling (idle hands). As seen in Table 2, index-filter 5 had the greatest reduction in speed, however, had the largest number of messages. Therefore, optimization with respect to both load balancing and messaging appears to be the most obvious way of optimizing one’s mesh design. As far as goals presented in this research, they have been accomplished in producing a systematic way of

Algorithm	Comp. Fitness by (2)
1	7803.118
2	8124.671
3	7759.214
4	7786.602
5	6039.532
“Eyeball”	9919.395

Table 4: Computational fitness of each index-filter.

producing efficient meshes in a short period of time. Thus, for more complex domains, when just looking at the mesh will no longer work, there is a way of performing optimized parallel computation on the mesh. We intend to extend this to tetrahedral meshes and possibly mixed element meshes. All of the previous discussion would hold for this case.

## Acknowledgments

We would like to thank Jeff Borggaard and Traian Iliescu for both of their contributions in answer questions and posing interesting questions. Jeff was also quite useful for his typesetting and Unix knowlege. Moreover, we would like to thank Peter Haskell for arranging the times and answer questions in regards to the Layman talk. The developers of Xfig for allowing easy generation of many of the figures included in this paper. Web sources such as [2] were a tremendous help in production of the software.

## References

- [1] *Encyclopedic Dictionary of Mathematics*, second edition.
- [2] Message passing interface (mpi). <http://www.llnl.gov/computing/tutorials/mpi>.
- [3] Chris Godsil and Gordon Royle. *Algebraic Graph Theory*. GTM. Springer-Verlag, 2000.
- [4] George Karypis and Vipin Kumar. Multi-level *k*way hyper-graph partitioning. <http://www-users.cs.umn.edu/karypis/publications/partitioning.html>.
- [5] Donald E. Knuth. *Fundamental Algorithms*. The Art of Computer Programming. Addison-Wesley, third edition, 1996.

- [6] J.T. Oden and J.N Reddy. *An Introduction to the Mathematical Theory of Finite Elements*. John Wiley & Sons, 1976.

```

bool most_common_index(vector<int> input, int& out){

// Sort the index list
sort(input.begin(), input.end());
int index1 = 0;
int storen = 0;
int count1 = 1;
int storec = 0;

for(int j = 1; j < (input.size()); j++){
    if(input[index1] == input[j]){ // Has the index changed
        count1++; // If not there is another common index
    }
    else{
        if(count1 > storen){ // Should I change the stored index
            storec = count1;
            storen = index1;
            index1 = j;
            count1 = 1;
        }
    }
}
if(count1 > storec){
    out = input[index1];
    return true;
}
else if(count1 < storec){
    out = input[storen];
    return true;
}
else{ // Not able to make a choice
    out = 0;
    return false;
}
return false;
}

```

Figure 7: Most Common Index (MCI) Algorithm