In *Proc. 9th SIAM Conf. on Parallel Processing* (1999).

Load Balancing 2-Phased Geometrically Based Problems

Andrew A. Poe[*] Quentin F. Stout[*]

Abstract

A k-weighted graph is a directed graph where each vertex has k associated weights. Partitioning such a graph so that each weight is roughly evenly divided into each piece is useful for balancing a parallel task with k distinct components. Such a goal arises in crash simulation, in which there there are 2 components. We present a 2-weighted graph partitioning algorithm, based on the Ham Sandwich Theorem, for graphs embedded in 2-space. We demonstrate that this algorithm runs in linear expected time and yields a good partition in practice.

1 Introduction

When one endeavors to make a serious study of parallel algorithms, the problem of load balancing inevitably presents itself. Load balancing can be described as the process of dividing a problem or subproblem into a number of smaller problems equal to the number of processors at hand such that the running time of this problem or subproblem when executed in parallel on these processors is minimized.

A problem can seldom be load balanced by simply dividing the labor equally among all the processors. Other factors must also be considered. It is likely that the processors will have to communicate with each other during the run. Since communication takes time, communication should be minimized in order to achieve ideal load balancing.

A related concept to communication is synchronization. A synchronization is a requirement that all processors must simultaneously be at specific points in their execution for each process to continue beyond its respective point. The normal manifestation of synchronization is an idling by one or more processors as they wait for the other processors to reach these specific points in their executions. Synchronizations also should be minimized in an ideal load balancing.

An inadequacy of most existing load balancing algorithms is that they only balance a single aspect of a problem. There exist certain practical applications with more than one aspect requiring balancing, and these aspects are not always independent; they cannot be balanced separately. An ideal balancing of one portion of a problem could result in a hideous division of labor for another phase. If, however, these phases are separated by a synchronization boundary, they must each be well-balanced. The phases cannot simply be balanced in summation; each unbalanced phase results in a waste of execution time due to idling.

One example of a two-phased balancing problem is impact analysis. Consider a non-empty finite set of 2-manifolds in 3-space. These manifolds not only move through space but contort as well because the force that is acting on them is not necessarily uniform.

As these manifolds move and contort, one may very well intersect another at a certain time, or one may intersect another part of itself. We constrain this problem so that this cannot happen by applying a recoil force to points lying in such intersections.

[*]Department of Electrical Engineering and Computer Science, The University of Michigan, Ann Arbor, MI.

The first phase of the impact analysis algorithm is called the **force calculation phase**. This phase iteratively applies the given force and moves and contorts the manifold accordingly.

The second phase of the algorithm checks to see if any elements intersect each other at the given time step. If so, the algorithm applies a recoil force to those elements in contact in an attempt to rectify the physical impossibility of two objects occupying the same space at the same time. This is called the **contact phase**. (These phases are detailed in [1].)

We encountered this example when we were working under a grant from the Ford Motor Company. The impact analysis algorithms we coded were simulations of automobile collisions. The load balancing algorithm we initially used for this analysis was designed to balance the force calculation phase of the program. It was woefully inadequate for the contact phase. This problem was the inspiration for our interest in the problem of k-weighted graph balancing. Far from being a mere theoretical curiosity, our study of this problem will help improve currently-existing parallel code.

In this paper, we define a geometric graph and describe several algorithms for partitioning them. We compare and contrast these algorithms with respect to different criteria.

2 k-Weighted n-Dimensional Geometric Graph Definition

DEFINITION 2.1. *A k-**weighted** n-**dimensional geometric graph** is a pair: (W_V, W_E), such that:*

1. *W_V is a function: $\Re^n \to \Re^k_{\geq 0}$ such that $W_V(v) = \vec{0}$ for all but a finite number of points v in $\Re^n$*
2. *W_E is a function: $(\Re^n)^2 \to \Re_{\geq 0}$ such that $W_E(v_1, v_2) = 0$ if $W_V(v_1) = \vec{0}$ or $W_V(v_2) = \vec{0}$.*

We define the the set V to be $\{v \in \Re^n \mid W_V(v) \neq \vec{0}\}$ and the set E to be $\{e \in V^2 \mid W_E(e) > 0\}$

Essentially, this definition is the same as a vertex-weighted edge-weighted directed graph: V is the set of vertices; E is the set of edges; W_E is the edge-weighting. The only real differences are that W_V represents k distinct weights on the vertices rather than a unique weight, and that each vertex has a physical location in $\Re^n$.

The vertices in this structure represent the items to be apportioned among processors. The k vertex weights represent computation times for that item for each of the k phases.

The edges represent a possible transfer of information. The incoming edges to a vertex indicate that a processor computing information for this vertex would require knowledge of the information of the source vertices in order to correctly compute its own information.

DEFINITION 2.2. *An m-**partition**, M, of a k-weighted n-dimensional geometric graph $G = (W_V, W_E)$ is a finite set $\{G_1, \ldots, G_m\}$ where each $G_i = (W_V^i, W_E^i)$ is a k-weighted n-dimensional geometric graph, (with V^i and E^i also defined appropriately) such that:*

1. *$\{V^1, \ldots, V^m\}$ is a partition of V; i.e. $\bigcup_{i=1}^m V^i = V$; $V^i \neq \phi$, for all i; $V^i \cap V^j = \phi$, for all $i \neq j$.*
2. *$W_V^i(v) = W_V(v)$ for all $v \in V^i$.*
3. *$E^i = \{(v, w) | v, w \in V^i$ and $(v, w) \in E\}$*
4. *$W_E^i(e) = W_E(e)$ for all $e \in E$.*

The set $E_S = E - \bigcup_{i=1}^m E^i$ is the called the set of **severed edges** of an m-partition of a k-weighted graph.

In an ideal situation, we would like, for each k, the sum of the kth weight on the vertices, $\sum_{v \in V^i} (W_V^i(v))|_k$ to be as close as possible to equal for all V^i. In addition, we would also like the maximum weight of the severed edges associated with each partition, $\max_{G_i \in M} \sum_{\{e \in E_S \cap (v,w), v \in V^i\}} W_E(e)$, to be as small as possible.

3 The Ham Sandwich Theorem and Geometric Graph Partitioning

DEFINITION 3.1. *A* ***weighted point in*** d ***dimensions*** *is an ordered pair* (p, w) *with* $p \in \Re^d$ *and* $w \in \Re_{\geq 0}$. w *is called the* ***associated weight****, and* p *is called the* ***associated point*** *of* (p, w).

THEOREM 3.1. ***(Ham Sandwich Theorem)*** *Given* d *finite sets* $P_1, \ldots, P_d$ *of weighted points, there exists a* $(d-1)$*-dimensional hyperplane in* $\Re^d$ *dividing* $\Re^d$ *into two (open) regions (*R_1 *and* R_2*) so that the sum of the associated weights of each* $(P_i^p \cap R_j)$ *is at most half of the sum of the associated weights of* P_i*, where* P_i^p *is the set of associated points of the elements of* P_i. □
[5]

Historically, this theorem generally refers to manifolds rather than to discrete point sets. Given any d compact d-manifolds in $\Re^d$ with a smooth mass density function defined on each, there exists a $(d-1)$-dimensional hyperplane in $\Re^d$ bisecting each of the d manifolds exactly in half with respect to mass, even if these manifolds overlap. In 3-space, any three solid objects can be split in half with a single planar cut. These three objects might be two slices of bread and a slice of ham. The name of this theorem derives from this example. [5]

The Ham Sandwich Theorem shows immediate application for the k-weighted balancing problem. If we take k to be 2, and the vertices of our graph have explicit coordinates in the Cartesian plane, we can consider the vertices to be two sets of discrete weighted points. The associated points in each set are identical; one set has the first of the two weights as its associated weight; the other has the second. The Ham Sandwich Theorem states that there is a single line that will divide both these sets (even though the associated points are identical!) such that neither half-plane resulting contains more than half of each sum of associated weights.

Once a point set has been bisected, such an algorithm can be applied recursively to divide the point set further. In the sections below, we describe several algorithms that can be used to locate a Ham Sandwich cut of a planar set of points in linear expected time. If there are n points to be partitioned into p sets, the algorithm will run in $O(n \log p)$ expected time.

4 The BHS Algorithm

4.1 Introduction

The Biweighted Ham Sandwich (BHS) algorithm we implemented for the purpose of 2-weighted balancing was based upon a randomized ham sandwich algorithm by Matousek [4] that runs in linear time on average on the number of points in the discrete sets. Matousek's original algorithm had several defects which we corrected in BHS, the most serious of which were its requirement that the two point sets be disjoint and its inability to handle weighted points. BHS is designed specifically to handle two sets of weighted points where the sets of associated points are identical.

4.2 Description

Briefly, BHS works as follows:

1. All points (x_1, y_1) are mapped to a dual line $y = x_1 x - y_1$. Two sets of dual lines are made. One set associates with each line the first of the two weights on the points. The other set uses the second weight.

2. Median levels of the dual sets are found. The median level of a dual is defined as follows:

   DEFINITION 4.1. *The c-**intersection set of** $\mathcal{L}$ is the intersection of the vertical line $x = c$ with the union $\mathcal{L}$ of all the dual lines of a point set.*

   DEFINITION 4.2. *The c-**median value of** $\mathcal{L}$ is the weighted median value of all the ordinates of the points in the c-intersection set of $\mathcal{L}$.*

   DEFINITION 4.3. *The **median level of** $\mathcal{L}$ is a function $m : \Re \to \Re$ with $m(c)$ equal to the c-median value of $\mathcal{L}$ for all $x \in \Re$.*

   In other words, a median level is a function of x that, for any given x-value, lies above at most half the weight in the dual set and below at most half the weight in the dual set. It is easy to see that any point on the median curve is mapped to a dual line that bisects the original point set with respect to weight.

3. The intersection of the two median levels is found. This point translates to a dual line that bisects both point sets with respect to weight.

   This intersection point is found in linear expected time by a prune-and-search technique. At most iterations a fixed fraction of the dual lines are discarded as being too far above or below the median level at the endpoints to contain a intersection point of the median levels. (Occasionally, this doesn't happen, and if it fails to happen too frequently, the algorithm runs in quadratic time, rather than linear.)

   Also, at each iteration the width of the region containing an intersection point is narrowed so that each iteration checks a smaller region for the intersection of the level curves.

   Each prune phase runs in linear time on the number of lines not yet discarded, and the number of lines decreases geometrically with each iteration, so the whole thing runs in linear expected time.

5 Other Geometric Graph Partitioning Strategies

5.1 Coarsening, refining, and local correction.

The BHS algorithm is unusual among graph partitioning strategies in that the basic algorithm runs in linear expected time. Most basic partitioning algorithms run in quadratic expected time—a time considered unacceptably slow by most users of such algorithms.

These slower algorithms then employ a strategy called coarsening and refining to improve their total running times, although usually at the expense of the accuracy of the resulting partition.

5.2 The BHS+ Algorithm

Discrete point sets have no connecting edges. Thus, a pure Ham Sandwich Algorithm does not specifically address the problem of minimizing severed edges. We were able to modify the algorithm to consider edge cuts as well.

Prior to applying BHS to the graph, the graph was contracted via a coarsening strategy suggested in Karen Tomko's Ph.D. thesis [6]:

A vertex is chosen randomly from the graph and is paired with the adjacent vertex (i.e. a vertex sharing an edge with the chosen vertex) that minimizes the absolute difference between the sums over both vertices of the two weights. Subsequent vertices are chosen randomly (without coinciding with vertices previously chosen or their pairings) until no more pairings remain. The graph is contracted with respect to these pairings. The new graph will contain roughly half the number of vertices of the old graph.

The same process is applied to the new graph, contracting it further. This continues until the number of vertices falls below a fixed **coarsening constant**. At this point, BHS is applied to the smaller graph. The graph is then refined by undoing the previous contractions and applying the following local correction strategy, also from [6].

With this strategy, boundary vertices are relocated to the other piece if doing so will decrease the weight of the severed edges and if the relative difference of the vertex weights of the partitions would not rise above a user-specified **partitioning constant** should such a move be made.

The coarsening and refining process was designed to accomplish two aims: to shrink the number of severed edges in a ham sandwich cut, and to shrink the number of vertices in the graph to be divided by BHS.

However, since BHS is a linear (expected) time algorithm, and coarsening and refining are (at best) linear processes, the need for the second purpose is not immediately obvious, and this lack of need inspired another variant of BHS.

5.3 The BHS—L Algorithm

BHS—L does no graph contraction. It runs BHS on the entire graph. Following this, however, it does the local correction on the partitioned graph, just as BHS+ does at each stage of refinement. (The difference, of course, is that BHS—L performs local correction only once.) Because it performs local correction, it makes use of the partitioning constant described above; however, since the graph is never coarsened, no coarsening constant is necessary.

5.4 Standard Metis

Metis is a graph-partioning software package available to the public. Metis uses a standard coarsening and refining technique to partition graphs: It successively combines adjacent vertices into single vertices until the graph is sufficiently small. The smaller graph is then partitioned using any of a number of strategies that can be chosen by the user or by Metis itself. The combined nodes are separated again in reverse order, making local corrections to the partition when necessary. Metis 2.0 was the version used; however, as described below, Metis 4.0 is now available and is a more sophisticated version. [2]

5.5 Dual Weight Greedy Bisection

This is a modified version of Metis using Karen Tomko's dual weight partition strategy which generalized Metis to partition graphs with two-vertex weights. [6]

Tomko called her algorithm Dual Weight Greedy Bisection. The basic algorithm chooses a vertex at random from the graph and makes it the starting point for a subset of the graph called subdomain 0. At each step in the algorithm all the vertices not in subdomain 0 that share an edge with a vertex in subdomain 0 are examined, and the vertex that would cause the least increase in the weight of the severed edges if it were moved to subdomain 0 is moved to subdomain 0 if moving it does not unbalance either weight by a predetermined **partitioning constant**.

If there is no eligible vertex adjacent to subdomain 0 for reassignment to subdomain 0, another random vertex is chosen. The algorithm terminates when the weights of the subdomains are within a predetermined constant factor of each other, or when no appropriate vertex can be found at all to add to subdomain 0. If the algorithm terminates for the latter reason, the algorithm is said to have failed to find a partition that balances the weights within the bounds set by the partitioning constant.

TABLE 1

n = 2000, s = 3, HC = 1000, LC = 100, HP = 0.14, LP = 0.07

Algorithm	Ave. run time	% edge cut	%W1 ratio	%W2 ratio
Metis	0.0s	6.95	17.2	18.1
BHS	0.6s	36.20	5.1	5.0
DWGB/LP/LC	0.0s	8.79	70.1	66.0
DWGB/LP/HC	0.0s	9.01	77.7	75.8
DWGB/HP/LC	0.0s	7.40	240.9	229.5
DWGB/HP/HC	0.0s	7.89	260.5	258.6
BHS+/LP/LC	0.0s	9.86	49.9	50.3
BHS+/LP/HC	0.0s	9.99	36.4	39.7
BHS+/HP/LC	0.0s	7.56	88.5	89.0
BHS+/HP/HC	0.0s	8.27	72.4	70.9
BHS—L/LP	0.0s	9.99	36.4	39.7
BHS—L/HP	0.0s	8.27	72.4	70.9

TABLE 2

n = 10000, s = 1.5, HC = 1000, LC = 100, HP = 0.0016, LP = 0.008

Algorithm	Ave. run time	% edge cut	%W1 ratio	%W2 ratio
Metis	3.0s	3.51	7.5	6.8
BHS	5.0s	19.90	1.5	1.8
DWGB/LP/LC	3.0s	9.76	6.6	6.7
DWGB/LP/HC	3.0s	9.48	7.3	7.1
DWGB/HP/LC	3.0s	7.00	13.1	13.5
DWGB/HP/HC	3.1s	7.72	15.4	15.4
BHS+/LP/LC	3.9s	14.4	7.5	7.5
BHS+/LP/HC	4.0s	11.0	6.1	5.8
BHS+/HP/LC	3.9s	8.52	14.3	13.5
BHS+/HP/HC	4.0s	8.19	11.1	11.5
BHS—L/LP	4.0s	11.3	5.9	6.0
BHS—L/HP	4.0s	8.24	9.7	10.4

For the purposes of testing, when this occurred, we just made the constant a little higher and tried again.

Before the graph is partitioned using this strategy, it is contracted using the same coarsening and refining process later applied to BHS+.

While Tomko has not made her software available, she described her algorithms in her thesis, and we were able to duplicate them from this information.

6 Run-time testing

Data sets were created by generating n random points in the square ([0,100),[0,100)) amd assigning to each two random integer weights between 1 and 10. For some tests, n was 2000. For the others, n was 10000.

Once the points were generated, edges were generated in one of two ways:

For some tests, edges were added between all pairs of vertices separated by a distance no greater

TABLE 3
n = 10000, s = 3, HC = 1000, LC = 100, HP = 0.14, LP = 0.07

Algorithm	Ave. run time	% edge cut	%W1 ratio	%W2 ratio
Metis	4.7s	14.0	6.8	7.2
BHS	5.1s	36.5	1.5	1.8
DWGB/LP/LC	5.4s	25.7	9.7	10.1
DWGB/LP/HC	5.8s	23.4	10.5	10.2
DWGB/HP/LC	5.7s	18.7	17.6	19.0
DWGB/HP/HC	6.0s	19.1	21.2	21.8
BHS+/LP/LC	6.5s	32.8	9.8	10.2
BHS+/LP/HC	7.1s	24.9	10.6	10.4
BHS+/HP/LC	6.4s	21.7	19.2	19.6
BHS+/HP/HC	7.0s	20.4	18.6	18.6
BHS—L/LP	7.1s	26.6	9.8	9.4
BHS—L/HP	7.4s	21.4	16.5	16.5

TABLE 4
n = 2000, triangulated, HC = 1000, LC = 100, HP = 0.02, LP = 0.01

Algorithm	Ave. run time	% edge cut	%W1 ratio	%W2 ratio
Metis	0.0s	20.3	21.1	20.7
BHS	0.7s	37.5	9.6	10.5
DWGB/LP/LC	0.0s	21.4	63.5	66.6
DWGB/LP/HC	0.0s	21.5	78.5	83.8
DWGB/HP/LC	0.0s	20.4	237.2	230.8
DWGB/HP/HC	0.0s	20.4	257.6	249.9
BHS+/LP/LC	0.0s	23.1	54.5	53.9
BHS+/LP/HC	0.0s	23.4	46.8	46.1
BHS+/HP/LC	0.0s	21.8	85.1	87.7
BHS+/HP/HC	0.0s	21.8	73.3	74.1
BHS—L/LP	0.0s	23.4	46.8	46.1
BHS—L/HP	0.0s	21.8	73.3	74.1

than s. The value of s was 1.5 for some tests, 3 for others. Clearly, the graphs generated with the higher value of s were denser than the graphs generated with the lower value.

For other tests, we triangulated the graph: Starting with the pair of vertices closest together and continuing among the pairs in order of increasing separation, we added an edge to connect the pair, provided that the edge crossed no other edge already introduced.

Finally, we used a non-random data set: we used the dn5 data set used by Ford Motor's FCRASH program. The weights on the graphs were approximately the weights used by Tomko to test her Dual Weight Greedy Bisection Algorithm described above.

These were partitioned several times, each time with a different strategy using the five strategies described above.

We ran each algorithm ten times, partitioning random data sets of n points into 64 pieces. We executed these algorithms on a Sun Microsystems Ultra II with dual processor running Solaris 2.5.1.

We catalogued the following data for each algorithm: the average running time, the percentage

TABLE 5
n = 21896, FCRASH dn5 model, HC = 1000, LC = 100, HP = 0.04, LP = 0.02

Algorithm	Ave. run time	% edge cut	%W1 ratio	%W2 ratio
Metis	13.0s	5.6	42.7	56.1
BHS	20.0s	48.2	0.5	1.0
DWGB/LP/LC	15.0s	12.7	19.9	17.1
DWGB/LP/HC	15.0s	11.9	18.7	16.1
DWGB/HP/LC	15.0s	10.4	29.3	44.7
DWGB/HP/HC	15.0s	11.0	23.0	25.8
BHS+/LP/LC	16.0s	13.9	17.6	21.0
BHS+/LP/HC	22.0s	20.5	17.2	19.6
BHS+/HP/LC	15.0s	15.3	37.5	50.9
BHS+/HP/HC	22.0s	17.4	27.2	30.3
BHS—L/LC	23.0s	21.6	18.2	16.3
BHS—L/HC	23.0s	19.6	33.5	28.6

of edges that were cut, and the ratio of the excess weight of the largest partition (in other words, largest weight minus average weight) to the average partition weight for each vertex weight.

These data are listed in the following tables. DWGB stands for Dual Weight Greedy Bisection. LP and HP indicate the lower and higher of the partitioning constants listed in a specific table. LC and HC represent the lower and higher of the coarsening constants.

Here is an evaluation of these four algorithms.

- Time complexity. All the tested algorithms run in linear expected time. In terms of worst-case running time, Standard Metis and Dual Weight Greedy Bisection, which is based on Metis, and BHS+ and BHS—L, which use a coarsening and refining technique, run in $O(n^2)$ worst-case time; they run in expected linear time on the (reasonable) assumption that the number of edges of each vertex of the graph is bounded. However, Dual Weight Greedy Bisection will abort without reporting an answer if the partitioning constant is set too small. BHS and BHS—L use a randomized process in such a way that if the random number generator is extremely poor (or extremely unlucky), the algorithm will give up (to avoid a potential infinite loop) and transfer control to a brute force $O(n^2)$ algorithm. The same thing happens in BHS+; however, in this case, the graph has been contracted to a constant size; thus, the asymptotic time complexity does not depend on the strategy used to partition the contracted graph.

- Actual running time. The Metis software, as to be expected, performed the best. Since Metis didn't consider the dual weighting when it did the partition, it had less to compute. BHS performed the worst in this regard when n was 2000, but even so, the running time was not prohibitively high. For the tests when n was at least 10000, the Dual Weight Greedy Bisection and BHS+ algorithms performed similarly, although the Dual Weight Greedy Bisection algorithms were still a little quicker. For the dense graph ($n = 10000, s = 3$), BHS outperformed all of the BHS+ and Dual Weight Greedy Bisection algorithms. This would lead one to believe that in dense graphs, the coarsening and refining steps serve only to slow the Ham Sandwich algorithm down.

- Weight balancing. In this area, the Ham Sandwich algorithms really shone. Ham Sandwich is an ideal tool for partitioning sets with respect to two weights. Even adding a coarsening and refining phase didn't largely hinder the weight balancing. As described above, if Dual Weight Greedy Bisection is run with the partitioning constant set too low, the algorithm will abort without finding a partition at all; however, lower constants that are not too low yield better weight balancing. I gave Dual Weight Greedy Bisection a leg up in the competition by running it with the lowest partitioning constant that would allow it to work (a heuristic which is data-dependent), and even so, BHS, BHS+, and BHS—L cleanly beat it in this regard at $n = 2000$. At $n \geq 10000$, Dual Weight Greedy Bisection, BHS+, and BHS—L performed similarly; however, BHS had a much better weight balancing than all. Note that BHS—L did as well or better than BHS+, indicating that coarsening and refining is not improving weight balance significantly.

The Ham Sandwich algorithm is so good at weight balancing that a coarsening and refining phase only gets in the way as far as this category goes. This is only to be expected, though, as this phase was introduced to improve edge cut at the expense of weight balance. Generally speaking, the more Ham Sandwich coarsens and refines (the smaller the coarsening heuristic) the worse the weight balance will be.

For Dual Weight Greedy Bisection, BHS+, and BHS—L, lowering the partitioning constant gave better performance with respect to weight balance. This heuristic establishes a strict maximum weight imbalance for Dual Weight Greedy Bisection. Although BHS+ and BHS—L use this heuristic only as a guideline rather than as a strict limit, shrinking the heuristic also improved the performance of BHS+ and BHS—L.

- Severed edge minimization. BHS decidedly lost this category on all values of n and s. The problem with a straight Ham-Sandwich cut is that, by failing to consider edge-cuts at all, this algorithm partitions the region into discrete convex geometric subregions. This is wonderful for weight balancing, but not so good for minimizing severed edges. Adding a coarsening and refining stage seemed to help, but actually considering that the BHS—L algorithm did about as well as the BHS+ algorithm, it would seem the edge-cut improvement is derived primarily from the local correction phase, rather than from coarsening and refining per se.

  Varying the coarsening heuristic did not change the edge cut of Dual Weight Greedy Bisection in any major way.

  For Dual Weight Greedy Bisection, BHS+, and BHS—L, edge cut was improved by a high partitioning constant. During the refining phase, both algorithms improve the edge cut at the expense of the weight balance. Higher partitioning constants give these algorithms more maneuvering room to minimize the severed edges.

7 Recent Developments

Karypis and Kumar have recently released Metis 4.0 which handles partitions on graphs with more than one vertex weight. This gives us a new benchmark with which to compare our work. Metis 4.0 uses methods that are based on principles similar to the ones on which Tomko's Dual Weight Greedy Bisection is based, but appear to be more sophisticated [3]. At the time of this writing, we have not yet compared our code extensively against Metis 4.0; however, we plan to do this in the immediate future. We are also continuing to improve and refine our code.

References

[1] D. J. Benson and J. O. Hallquist, *A single surface contact algorithm for the postbuckling analysis of shell structures*, Report to the University of California at San Diego, CA, 1987

[2] G. Karypis and V. Kumar, *METIS unstructured graph partitioning and sparse matrix ordering system version 2.0*, Instruction manual

[3] G. Karypis and V. Kumar, *Multilevel algorithms for multi-constraint graph partitioning*, Technical Report #98-019, University of Minnesota, Department of Computer Science/Army HPC Research Center

[4] J. Matousek, *Ham-sandwich cut in the plane*, section of unpublished textbook

[5] J. V. Peters, *The ham sandwich theorem and some related results*, The Rocky Mountain Journal of Mathematics, 11 (1981), pp. 473–482

[6] K. A. Tomko, *Domain decomposition, irregular applications, and parallel computers*, Ph.D. thesis, The University of Michigan (1995)