8th SIAM Conference on Parallel Processing for Scientific Computing, Minneapolis, MN, March 1997.

A Data–Parallel Implementation of the Geometric Partitioning Algorithm

Yu Charlie Hu* Shang-Hua Teng† S. Lennart Johnsson‡

Abstract

We present a data-parallel, High Performance Fortran (HPF) implementation of the geometric partitioning algorithm. The geometric partitioning algorithm has provably good partitioning quality. To our knowledge, our implementation is the first data–parallel implementation of the algorithm. Our data–parallel formulation makes extensive use of segmented prefix sums and parallel selections, and provide a data-parallel procedure for geometric sampling. Experiments in partitioning particles for load–balance and data interactions as required in hierarchical N-body algorithms and iterative algorithms for the solution of equilibrium equations on unstructured meshes by the finite element method have shown that the geometric partitioning algorithm has an efficient data–parallel formulation. Moreover, the quality of the generated partitions is competitive with that offered by the spectral bisection technique and better than the quality offered by other partitioning heuristics.

1 Introduction

The solution of many large–scale scientific and engineering problems are based on domain discretization in the form of unstructured meshes in two or three dimensions. The computational problem is defined by the numerical formulation used to solve the physical problem on the discretized domain. Large–scale problems can only be solved in reasonable time on scalable parallel computers which typically have the memory physically distributed among the processors. Efficiency in processor (and memory) utilization requires that the data for the problem be partitioned and distributed among the processors and their memories. The extent of the interaction between the data in different memory modules affect the interprocessor communication need. In most scalable architectures, the interprocessor data motion capacity is considerably less than the capacity between a processor and its local memory. The quality of the partition measured in terms of evenness of workload among partitions and need for interpartition references affects the efficiency in using the system resources.

Mesh partitioning is an important case of general graph partitioning. General graph partitioning has been an active field of research, both theoretically [1, 4, 12, 13, 14, 16], and experimentally [5, 18, 8, 11]. Finding an optimal partitioning is an NP–hard problem.

*Aiken Computation Laboratory, Harvard University, Cambridge, MA 02138. Supported by the Air Force Office of Scientific Research through grants F49620-93-1-0480 and F49620-96-1-0289.

†Department of Computer Science, University of Minnesota, Minneapolis, MN 55455. Supported by an NSF CAREER award (CCR-9502540), an Alfred P. Sloan Research Fellowship, and an Intel research grant. Part of the work was done while the author was at MIT and Xerox Corporation (PARC).

‡Aiken Computation Laboratory, Harvard University, Cambridge, MA 02138 and University of Houston, Houston, TX 77204. Supported by the Air Force Office of Scientific Research through grants F49620-93-1-0480 and F49620-96-1-0289.

Heuristic partitioning algorithms that provide strong guarantees for the quality of the partitioning have been developed for certain classes of graphs, such as planar graphs [13], bounded genus graphs [4], bounded forbidden minor graphs [1], nearest neighbor graphs, well–shaped meshes [16, 20], and hierarchical N–body simulation graphs [23]. The main objective of this work is to show that the geometric partitioning algorithm of Miller–Teng–Thurston–Vavasis [16] has a practical data–parallel formulation/implementation.

The geometric partitioning algorithm, as its name suggests, is based on the geometric structure of a mesh. The geometric information is used not only for proving the guaranteed quality of the partitioning of the algorithm, but also, as shown by Gilbert, Miller, and Teng [5], for efficient algorithm design and implementation. This paper shows that the geometric structure can be used also for a data–parallel formulation that yields efficient implementations.

2 The Geometric Partitioning Algorithm: A Review

We assume that a mesh M is given by its geometric structure xyz together with its combinatorial structure A, where xyz is an array of coordinates of the mesh vertices and A is an array of vertex–pairs that represent the edges among mesh vertices in M. If M has n vertices in $\mathbb{R}^d$ and e edges, then xyz is an $n \times d$ array and A is an $e \times 2$ array.

A *(2–way) partitioning* of a mesh M is a division of M into two submeshes M_1 and M_2 of roughly the same size. The *cut size* of the partitioning is equal to the number of edges that bridge M_1 and M_2.

To describe the algorithm we need two concepts. Let Π denote the *stereographic projection* mapping from $\mathbb{R}^d$ to S^d, where S^d is the unit d–sphere embedded in $\mathbb{R}^{d+1}$. A unit d–sphere is defined as $\sum_{i=0}^{d} x_i = 1$. For each $p \in \mathbb{R}^d$, $\Pi(p)$ is given as follows. Append '0' to p as coordinate $d+1$ yielding $p' \in \mathbb{R}^{d+1}$. Then, compute the intersection of S^d with the line in $\mathbb{R}^{d+1}$ passing through p' and $(0, 0, \ldots, 0, 1)^T$. This intersection point is $\Pi(p)$. The formula for Π and the inverse of Π are very simple and can be found in [16].

The second concept is that of a centerpoint. A *centerpoint* of a given set of points is a point (not necessarily one of the given points) such that every (hyper)plane through the centerpoint divides the given points approximately evenly (in the ratio d:1 or better, in $\mathbb{R}^d$). Every finite point set in $\mathbb{R}^d$ has a centerpoint [3, Section 4]. We now describe the algorithm of Miller–Teng–Thurston–Vavasis [15, 16].

Algorithm *Geometric Partitioning*
Input $M = (A, xyz)$;

1. **Project Up.** Let $xyzw = \Pi(xyz)$.
2. **Find Centerpoint.** Find a centerpoint c of $xyzw$.
3. **Conformal Map: Rotate and Dilate.** In principle, move the projected points in $\mathbb{R}^{d+1}$ on the surface of the unit sphere in two steps. First, rotate the projected points about the origin in $\mathbb{R}^{d+1}$ so that the centerpoint becomes a point $(0, \ldots, 0, r)$ on the $(d+1)$–st axis. Second, dilate the points on the surface of the sphere so that the center point becomes the origin. The dilation can be described as a scaling in $\mathbb{R}^d$: project the rotated points stereographically down to $\mathbb{R}^d$; scale the points in $\mathbb{R}^d$ by a factor of $\sqrt{(1-r)/(1+r)}$; and project the scaled points up to the unit sphere in $\mathbb{R}^{d+1}$ again.
4. **Find Great Circle.** Choose a random great circle GC (i.e., d–dimensional unit sphere) on the unit sphere in $\mathbb{R}^{d+1}$.
5. **Unmap and Project Down.** Transform the great circle to a sphere S in $\mathbb{R}^d$ by undoing the dilation, rotation, and stereographic projection.
6. **Induce a Partition from the Separating Sphere.** The sphere S in $\mathbb{R}^d$ divides the vertices of M into two subsets xyz_I and xyz_E, the set of vertices that are in the interior of S and in

the exterior of S, respectively. We return $M_I = (A_I, xyz_I)$ and $M_E = (A_E, xyz_E)$, where A_I and A_E are induced meshes by xyz_I and xyz_E, respectively.

After the projection and conformal mapping, the origin of $\mathbb{R}^{d+1}$ is a centerpoint for the mesh vertices. Therefore the mapped vertices are divided approximately evenly by every plane through the origin—that is, by every great circle on the unit sphere in $\mathbb{R}^{d+1}$.

Miller, Teng, Thurston, and Vavasis [16] proved the following mathematical result on the performance of the algorithm given above for the class of *well–shaped meshes.* We refer the reader to [16] for the detailed definition of well–shaped meshes. For this paper, well–shaped meshes include all structured and unstructured finite element meshes.

THEOREM 2.1 (GEOMETRIC PARTITIONING [16]). *Let $M = (A, xyz)$ be a well–shaped mesh in $\mathbb{R}^d$ of n vertices and e edges. Then $|xyz_I|, |xyz_E| \leq (d+1)/(d+2) \cdot n$, and with probability at least $1/2$, the number of edges between M_I and M_E is $O(n^{1-1/d})$.*

3 Data–Parallel Formulations

A Matlab [5] implementation of the geometric partitioning algorithm has shown that the algorithm generates partitionings competitive with those rendered by other partitioning algorithms. In the next two sections, we present our data–parallel formulation and HPF implementation of this algorithm.

3.1 Data–Parallel Primitives

Data–parallel computations are expressed by a set of primitives over array aggregates. The simplest primitives are elementwise array operations; elementwise array operations are embarrassingly parallel and does not involve data movement if the operand arrays are properly aligned. Two important classes of primitives that induce data movements are

- prefix sums (over an associative operator), segmented prefix sums, broadcast, and array reduction. We often refer to prefix sums as scans and segmented prefix sums as segmented scans.
- array permutations, gather, and scatter operations.

On most parallel machines, the first class is much more efficient than the second class, in part because the data movement for the commonly used algorithms is regular and can be implemented using binary tree structures. Therefore, we have attempted to avoid operations of the second class to the extent possible. Moreover, important operations such as parallel sorting, ranking, and selection can be efficiently expressed in turn by parallel segmented scans and some small number of gather and scatter operations.

3.2 A Data–Parallel Formulation

3.2.1 2–Way Partitioning We now analyze the computation steps of the geometric partitioning algorithm (see Section 2) in the context of a data–parallel formulation. Step 1 (Project Up) involves only elementwise operations on array xyz. We will discuss Step 2 (Find Centerpoint) shortly. In Step 3, we only need to compute the conformal map which can be expressed by a single rotation matrix and a dilation factor. In d dimensions, the rotation matrix is of shape $d \times d$. We first broadcast the centerpoint, then each processor computes the rotation matrix and the dilation factor in $O(d^2)$ time. In Step 4 (Find Great Circle), we generate the random great circle on only one processor (in $O(d)$ time), then this processor broadcasts the great circle to all other processors. In Step 5 (Unmap and Project Down) each processor can independently undo the conformal map and transform the great circle to a sphere S in $\mathbb{R}^d$. In Step 6, to induce the partitioning of the mesh M from the circle S, we need to determine which mesh vertices are in the interior of S and which are in the exterior. We make use of parallel prefix sums for this computation. Let u be the center of S and r be its radius. A mesh vertex p_i is in the interior of S iff $||p_i - u|| \leq r$. We create an auxiliary array where entry i is equal to 1 if the corresponding mesh vertex p_i is in the interior of S, otherwise, the entry is 0. We can construct the auxiliary array using elementwise operations. A prefix sum on the auxiliary array generates the indices of the mesh vertices in array xyz_I. Similarly, we can construct indices for mesh vertices in xyz_E by a prefix sum. Once xyz_I and xyz_E are determined, we can determine the number of edges cut by the partitioning induced by S by a parallel array reduction. We use an auxiliary array of e elements and assign an entry 1 if the edge has one endpoint in xyz_I and another endpoint in xyz_E; otherwise, we assign it 0. The sum–reduction of the auxiliary array gives the cut size. By assigning the auxiliary array proper 0–1 values, we can compute the indices of edges in arrays A_I and A_E. If we want to try another random great circle (to improve the quality of the partition) we can repeat this process. Once we decide the final partition, we can apply gather and scatter to construct arrays xyz_I, xyz_E, A_I, and A_E.

We now discuss Step 2. As suggested in [15] and implemented in [5], an efficient way to find a centerpoint of a point set P is to use geometric sampling. To find an approximate centerpoint, we first choose a uniform sample W of P. For practical applications, the size of W is about 1000. It has been shown [21] that with high probability the centerpoint of W is a point whose worst hyperplane separation ratio for P is $1 : d + \epsilon$ for a very small ϵ, $0 < \epsilon < 1$. Therefore sampling can be used to drastically reduce the amount of calculations for centerpoint computation.

However, finding the centerpoint for 1000 points is still expensive. We, as in [5], use an additional idea from [15] for finding an approximate centerpoint. This idea is based on a concept called *Radon point*. A point q is a *Radon point* of a point set P in $\mathbb{R}^d$ if P can be partitioned into two disjoint subsets P_1 and P_2 such that q lies in the intersection of the convex hull of P_1 and the convex hull of P_2. Every set of $d + 2$ points has a Radon point and can be found by solving a linear system on $d+2$ variables. The basic strategy from [16] is to repeatedly replace randomly chosen groups of $d + 2$ points with their Radon points. We can first randomly permute the sample array and then divide the sample into groups of $d + 2$ points and apply the Radon reduction to each group. We repeat the grouping and reduction. Eventually the set is reduced to a single point which is the approximate centerpoint. The above reduction process forms a complete $d + 2$–way tree and can be naturally expressed in data–parallel paradigm as a tree reduction. We refer readers to [2] for a proof of the quality of the above reduction process for centerpoint approximation. For 2–way partitioning, we only need to compute a single approximate centerpoint from 1000 sample points. We perform this on a single processor and broadcast the result to all other processors. Parallelism is needed when we recursively apply the 2–way partitioning procedure to generate a multi–way partitionings, as described next.

3.2.2 Multi–way Partition For parallel processing, we often need a multi–way partitioning, where a *k–way partitioning* of a mesh is a division of the mesh into k submeshes of roughly equal size. The *partition number* of a mesh vertex is the label of the submesh that contains the mesh vertex. In our formulation, we will recursively apply the 2-way partitioning method. For simplicity, we assume that k is a power of 2.

THEOREM 3.1 (MULTI–WAY GEOMETRY PARTITIONING [16, 19]). *Let $M = (A, xyz)$ be a well–shaped mesh in $\mathbb{R}^d$ of n vertices and e edges. For any positive integer k, the recursive application of the geometry partitioning algorithm finds a k-way partitioning which cuts $O(k^{1/d} n^{1-1/d})$ edges.*

A data–parallel formulation for 2–way partitioning can not be directly translated into a data–parallel formulation for multi–way partitioning (unlike in the message–passing programming model). One of the main reasons is that we need to use global array structures to express concurrent partitioning of submeshes. We use segmented parallel scans in accomplishing this task. Our formulation for Steps 1, 3, 4, and 5 can be extended directly to multi–way partitioning. We now focus on Steps 2 and 6. The recursive bisection procedure first finds one approximate centerpoint of all mesh elements. After the top level partitioning, the centerpoint procedure needs to find two centerpoints, one for each submesh; after that it needs to find four centerpoints, and then eight, and so on. At level i, 2^i centerpoints need to be computed. For this computation we use an array of $1000 * 2^i$ sample mesh vertices, 1000 for each submesh. We then run segmented tree–based Radon reduction for all segments of 1000 points in parallel. Submeshes may have different sizes, but we use the same number of sampling points for each submesh. This simplifies the HPF implementation. HPF currently does not support so–called ragged arrays.

3.3 Sampling for a More Efficient Formulation

On a distributed memory parallel machine, array permutations implying extensive data motion (and gather and scatter operations) are more expensive than parallel prefix sums. Therefore, for efficient data–parallel formulations it is desirable to attempt to minimize the number of array permutations as well as the size of arrays that are permuted. The data–parallel formulation above permutes the arrays of mesh vertices and edges at each level of the recursive partitioning procedure so that the vertices and edges of each submesh is stored in consecutive sub–array locations. These permutations may require extensive data motion. Below, we give a data–parallel formulation that computes the partition number of each mesh vertex based on permutation of (small) subsets of mesh data. A single permutation of all mesh data at the end of the partitioning procedure suffice to order submeshes into consecutive sub–array locations.

The basic idea is sampling. It is based on the following simple probabilistic fact.

LEMMA 3.2 (CHERNOFF-HOEFFDING[6]). *There is a constant $c > 1$ such that the following is true: Suppose there are L red balls in a set of n balls. Then, for any sample of $s(n)$ random balls from the set containing r red balls,*

$$\mathbf{Prob}[r/(2s(n)) \leq L/n \leq 2r/s(n)] \geq 1 - e^{-cs(n)L/n}.$$

This lemma can be applied to estimate the number of edges cut by a separating sphere. In our case, red balls correspond to edges cut by the sphere. Theorem 2.1 implies that the bound L that we would like to estimate is $L = O(n^{1-1/d})$. Thus as long as we sample more than $\Theta(n^{1/d} \log n)$ edges, we can approximate the size of the cut–size to within a small multiplicative factor, with very high probability (e.g., $1 - 1/n^2$).

The strategy now is to estimate the cut size of separating spheres by sampling edges rather than the entire mesh. Notice that the computation of centerpoints is also performed on samples. Therefore, the first step of our formulation is to form a sample array of edges and a sample array of mesh vertices. We then use these two sample arrays to support the recursive geometric partitioning procedure. At each level of the partitioning process, we only permute these samples rather than the entire mesh, hence reduce the complexity of the computation and communication. For a k–way partitioning, we need to sample $c \cdot 1000 * (k/2)$ mesh vertices (for some small constant c), which is in general much smaller than the number of input mesh vertices (in practical applications). If fact, if the number of processors in a system is p and if $p < k$, then we can first find a p–way partitioning using this idea. Then we permute the input mesh. By now, each submesh will be stored in a single processor, and we can apply an embarrassingly data–parallel formulation to complete the generation of the k–way partitioning; each processor can work independently on its own submesh without the need of communicating with other processors. It follows from Theorem 3.1 and Lemma 3.2 that we only need to sample $\Theta(k^{1-1/d}n^{1/d}\log n)$ edges, which is much smaller number than the total number of mesh edges. After we compute the final level of a k–way partitioning using the samples, we generate a complete binary tree (of $\log k$ levels) of the separating spheres. To determine the partition number of a mesh vertex v, we can perform a binary search against this tree in $\log k$ steps. On a parallel machine, we broadcast this tree structure to all processors, and all processors determine concurrently the partition numbers of the mesh vertices it stores in its memory according to the original mapping of vertices to processors. If only the partition numbers are needed, we do not need to permute the mesh. If we need to output the mesh according to the k–way partitioning, we can use parallel scans as described next.

Let $(sample_A, sample_{xyz})$ be the edge and mesh sample arrays respectively, $sample_{xyzw}$ be the stereographic image of $sample_{xyz}$, $centerpoint_i$ be the array of 2^i centerpoints at level i of the partitioning process, $sphere_i$ be the array of 2^i spheres at level i, T be the data structure for the complete binary tree of the separating spheres, *partition* be the array which will store the partition number of each mesh vertex, and B be a $p \times k$ auxiliary array whose (i,j)–th entry will store the number of mesh vertices on processor j whose partition number is equal to i.

Formulation *Sampling–Based Data–Parallel Geometric Partitioning*
Input (A, xyz)

1. Create $sample_A$ and $sample_{xyz}$ and randomly permute $sample_{xyz}$.
2. for $i = 0$ to $\log k - 1$ do
   (a) Let $sample_{xyzw} = \Pi(sample_{xyz})$. Let S_i be an array of $1000 * 2^i$ sample vertices of $sample_{xyzw}$, where we choose 1000 sample vertices from each submesh.
   (b) Find 2^i approximate centerpoints and store them in array $centerpoint_i$.
   (c) From $centerpoint_i$, find the proper conformal maps for all centerpoints; generate random great circles for all centerpoints, unmap them, and store these spheres in array $sphere_i$.
   (d) Test the quality of these spheres using $sample_A$ (we may repeat the random sphere step a few times for finding better spheres.)
   (e) Use segmented prefix scan to help permute $sample_{xyz}$ and $sample_A$.
   (f) Form the partial complete binary tree structure T.
3. Broadcast T to all processors. Determine the values of array *partition* in parallel by having all processors concurrently determine the partition number of the mesh vertices in xyz that are initially assigned to their local memory. For all j, processor j fills the entries of $B[:,j]$.
4. If only partition numbers are needed, then output array *partition*. Otherwise, using prefix scan on the rows of B, find the number of mesh vertices on all processors whose processor numbers are no more than j for all j in the range $1 \leq j \leq p$.
5. From the scan information and the local indices on each processor, determine the indices of all mesh vertices and edges in the final rearranged array for the k–way partitioned mesh. Permute the mesh according to these indices.

4 HPF Implementation and Experiments

High Performance Fortran[9] consists of Fortran 90 with extensions mainly for data management. The main extensions are: data distribution directives, which describe data aggregation such as cyclic and block aggregation, and the partitioning of data among memory regions; `FORALL` statements and constructs, which allow fairly general array sectioning and specifications of parallel computations; extrinsic procedures (local procedures), which defines interfaces to procedures written in other programming paradigms, such as explicit message–passing; a set of extended intrinsic and library procedures, including mapping inquiry subroutines and prefix scan and sorting functions.

An HPF implementation of the geometric partitioning algorithm is straight–forward from our data–parallel formulation above. We have chosen *pghpf* [17], the PGI HPF compiler, for our implementation mainly because it supports the complete set of HPF prefix functions which are heavily used in our data–parallel formulation.

The HPF implementation of the geometric partitioning (GEO) algorithm is incorporated into a data–parallel adaptive $O(N)$ N–body code (also in HPF) [10]. Table 1 compares the partitioning results of GEO and various other partitioning algorithms in simulations of one million particles having a 2–D Plummer distribution and at most 64 particles per leaf–level box. Two separate arrays representing active boxes in List–1 and List–2 interactions of the adaptive Anderson's method [10] are partitioned. The number of remote references and floating–point operations per partition are shown. The recursive spectral bisection (RSB) [18, 22] results are based on the RSB routine in the Connection Machine Scientific Software Library, CMSSL [24], which does not perform weighted partitioning. The other partitioning algorithms are heuristic, including orthogonal recursive bisection (ORB), partitioning based on the Morton and Peano–Hilbert ordering [25], rotational recursive bisection (RRB), and the level–by–level ordering (LBL) [10]. From Table 1, GEO with ten trials of great circles gives slightly more balanced computation than with two trials, but the edge cut is actually worse. Compared with Morton, GEO gives better partitions for List–1, and almost the same ones for List–2, but is much more expensive. The expense needs to be justified by the potentially increased efficiency of the more balanced computation. Methods involving gather/scatter or prefix operations are 5 – 10 times slower on SP2 than on CM–5E mostly because of the poor performance of the unoptimized run–time system subroutines generated by *pghpf* 2.1.

5 Conclusion

We have described a data–parallel formulation/implementation of the geometric partitioning algorithm. This work positively answers the question posed by Gilbert, Miller, and Teng [5]

"A chief application of graph partitioning is to distribute a computational mesh across a distributed–memory parallel machine. Can the partition itself be found in parallel? This is challenging because most partitioners make heavy use of the edges of the graph, and therefore require a lot of communication unless most adjacent vertices share the same processor—that is, unless a good partition is already known. We expect the geometric partitioner to be reasonably efficient in parallel, because almost none of the data manipulation involves the edges. (Coordinate bisection shares this desirable property, as Heath and Raghavan's parallel implementation shows [7].)".

References

[1] N. Alon, P. Seymour, and R. Thomas. A separator theorem for graphs with an excluded minor and applications. In *Proc. of the 22th Annual Symp. on Theory of Computing*. ACM, 1990.

[2] K. Clarkson, D. Eppstein, G. L. Miller, C. Sturtivant, and S.-H. Teng. Approximating center points with and without linear programming. In *Proc. of 9th ACM Symp. on Computational Geometry*, pages 91-98, 1993.

[3] H. Edelsbrunner. *Algorithms in Combinatorial Geometry*. Springer-Verlag, NY, 1987.

[4] J. R. Gilbert, J. P. Hutchinson, and R. E. Tarjan. A separator theorem for graphs of bounded genus. *J. Algorithms*, 5 pp391-407, 1984.

[5] J. R. Gilbert, G. L. Miller, and S.-H. Teng. Geometric mesh partitioning: Implementation and experiments. In *SIAM J. Sci. Comput.*, to appear 1997.

[6] T. Hagerup and C. Rüb. A guided tour of Chernoff bounds. *Info. Proc. Let.*, 33:305-308, 1990.

[7] M. Heath and P. Raghavan. A Cartesian parallel nested dissection algorithm, 1994. To appear in *SIAM Journal on Matrix Analysis and Applications*.

TABLE 1

Comparison of various partitioning algorithms for the 2-D Plummer model with one million particles. 128 partitions are generated. Arrays are initially in the LBL ordering.

Method	Remote references (Bytes/partition)		FLOPS / partition		Running time (sec.)	
	avg.	max.	avg.	max.	32-node CM-5E	16-wide-node SP2
List-1 (44144 nodes, 1118614 edges)						
LBL	8.05e+04	3.07e+05	5.42e+07	5.46e+07	N/A	N/A
Morton	6.32e+04	1.54e+05	5.42e+07	1.23e+08	0.06	0.03
Peano	6.00e+04	1.28e+05	5.42e+07	5.46e+07	0.13	0.08
ORB	6.20e+04	1.95e+05	5.42e+07	5.47e+07	0.57	3.84
RRB (10 trials)	4.86e+04	8.99e+04	5.42e+07	5.46e+07	14.7	98.3
RSB (no weights)	5.01e+04	8.70e+04	5.42e+07	9.61e+07	104.	—
GEO (10 trials)	4.95e+04	9.83e+04	5.42e+07	5.48e+07	19.6	202.
GEO (2 trials)	5.06e+04	9.19e+04	5.42e+07	5.49e+07	9.58	110.
List-2 (33103 nodes, 295181 edges)						
LBL	1.02e+05	1.62e+05	9.06e+07	9.09e+07	N/A	N/A
Morton	1.62e+05	2.52e+05	9.06e+07	9.08e+07	0.05	0.02
Peano	1.53e+05	2.78e+05	9.06e+07	9.09e+07	0.10	0.06
ORB	1.70e+05	3.74e+05	9.06e+07	9.10e+07	0.44	3.66
RRB (10 trials)	1.48e+05	2.50e+05	9.06e+07	9.11e+07	8.88	64.7
RSB (no weights)	1.80e+05	2.79e+05	9.06e+07	9.66e+07	31.1	—
GEO (10 trials)	1.48e+05	2.60e+05	9.06e+07	9.10e+07	13.0	169.
GEO (2 trials)	1.52+05	2.55e+05	9.06e+07	9.11e+07	6.95	95.5

[8] B. Hendrickson and R. Leland. The Chaco user's guide, Version 2.0. Technical Report SAND93-2339, Sandia National Laboratories, Albuquerque, NM, 1993.

[9] HPF Forum. HPF Language Specification, version 1.0. *Sci. Prog.*, 2(1-2):1-170, 1993.

[10] Y. C. Hu and S. L. Johnsson and S.-H. Teng. A data-parallel adaptive N-body method. In *Proc. of the 8th SIAM Conf. on Parallel Processing for Scientific Computing*, March 1997.

[11] G. Karypis and V. Kumar. METIS: Unstructured Graph Partitioning and Sparse Matrix Ordering System, Version 2.0 (1995), Dept. of Computer Science, University of Minnesota.

[12] F. T. Leighton and S. Rao. An approximate max-flow min-cut theorem for uniform multicommodity flow problems with applications to approximation algorithms. In *29th Annual Symp. on Foundations of Computer Science*, pp 422-431, 1988.

[13] R. J. Lipton and R. E. Tarjan. A separator theorem for planar graphs. *SIAM J. of Appl. Math.*, 36:177-189, April 1979.

[14] G. L. Miller. Finding small simple cycle separators for 2-connected planar graphs. *Journal of Computer and System Sciences*, 32(3):265-279, June 1986.

[15] G. L. Miller, S.-H. Teng, W. Thurston, and S. A. Vavasis. Automatic mesh partitioning. In A. George, J. Gilbert, and J. Liu, editors, *Sparse Matrix Computations: Graph Theory Issues and Algorithms*, IMA Volumes in Mathematics and its Applications. Springer-Verlag, 1993.

[16] G. L. Miller, S.-H. Teng, W. Thurston, and S. A. Vavasis. Finite element meshes and geometric separators. *SIAM J. Scientific Computing*, to appear, 1997.

[17] The Portland Group, Inc. *pghpf Reference Manual, Version 2.1*, 1996.

[18] A. Pothen, H. D. Simon, K.-P. Liou. Partitioning sparse matrices with eigenvectors of graphs. *SIAM J. Matrix Anal. Appl.* 11 (3): 430-452, July, 1990.

[19] H. D. Simon and S.-H. Teng. How good is recursive bisection? *SIAM J. Sci. Comput.*, 1997.

[20] D. A. Spielman and S.-H. Teng. Spectral partitioning works: planar graphs and finite element meshes. In *Proc. of the 37th Annual Symp. on Foundation of Computer Science*, 96-107, IEEE.

[21] S.-H. Teng. *Points, Spheres, and Separators: a unified geometric approach to graph partitioning.* PhD thesis, Carnegie-Mellon University, 1991. CMU-CS-91-184.

[22] D. A. Spielman and S.-H. Teng. Spectral partitioning works: planar graphs and finite element meshes. In *37th Annual Symp. on Foundation of Computer Science*, pages 96-107, 1996.

[23] S.-H. Teng. Provably good partitioning and load balancing algorithms for parallel adaptive N-body simulation. to appear *SIAM J. Scientific Computing*, 1997.

[24] Thinking Machines Corp. *CMSSL for CM Fortran, Version 3.1*, 1993.

[25] M. Warren and J. Salmon. A parallel hashed oct-tree N-body algorithm. In *Supercomputing'93*, pages 12 - 21. IEEE Computer Society, Los Alamitos, 1993.