Mob - A Parallel Heuristic For Graph-Embedding

John E. Savage[1]
Markus G. Wloka[2]

Technical Report No. CS-93-01

January 1993

[1]Brown University Box 1910, Providence, RI 02912
[2]Motorola, Inc. SSDT MD EL, 510, 2100 E. Elliot Road, Tempe, AZ 85284

Mob – A Parallel Heuristic for Graph-Embedding [†] [‡] [§]

John E. Savage
Brown University
Department of Computer Science, Box 1910
Providence, Rhode Island 02912
Phone: (401) 863-7600, Fax: (401) 863-7657
Email: jes@cs.brown.edu

Markus G. Wloka
Motorola, Inc., SSDT
MD EL 510, 2100 E. Elliot Road
Tempe, AZ 85284
Phone (602) 897-4452, Fax: (602) 897-3518
Email: wloka@ssdt-tempe.sps.mot.com

January 7, 1992

[†] This work was supported in part by the National Science Foundation under Grant CDA 87-22809 and the Office of Naval Research under contract N00014-83-K-0146, ARPA Order Nos. 4786, 6320, and contract N00014-91-J-4052, ARPA Order 8225.

[‡] A short report on this work appears in the *5th SIAM Conference on Parallel Processing for Scientific Computing*, (Mar. 1991), pp. 472-477 [20].

[§] This report is a revision of the experimental portion of Report CS-91-07

Abstract

We have extended the *Mob* heuristic for graph partitioning [21] to grid and hypercube embedding and have efficiently implemented our new heuristic on the CM-2 Connection Machine. We have conducted an extensive series of experiments to show that it exploits parallelism, is fast, and gives very low embedding costs. For example, on the 32K-processor CM-2 it runs in less than 30 minutes on random graphs of 1 million edges and shows impressive reductions in edge costs. Due to excessive run times, other heuristics reported in the literature can only construct graph embeddings for graphs that are 100 to 1000 times smaller than those used in our experiments.

1 Introduction

We introduce and report on the *Mob* heuristic for the embedding of graphs into grids and hypercubes. This heuristic is an extension of our *Mob* heuristic for graph partitioning [21]. These heuristics exploit parallelism and give results at least as good as their most effective serial counterparts for the problems for which they are designed, namely, the partitioning and embedding of low-degree random graphs, yet they handle problems hundreds to thousands of times larger.

Graph embedding (GE) and graph partitioning (GP) are NP-complete problems. Heuristics for GE map vertices of a graph onto the vertices of a grid and hypercube so as to minimize the sum of the lengths of the embedded edges. Heuristics for GP separate vertices in a graph into two sets such that the number of edges connecting the two sets is small.

Graph embedding finds application in VLSI placement and the minimization of data movement in parallel computers. The *VLSI-placement problem* is to minimize the area of a chip occupied by wires and cells; it can be modeled by embedding a graph in a grid. The *task assignment problem* is to assign tasks to parallel processors connected by a network so as to minimize the cost of communication; communication between tasks is represented by weighted edges in a graph. When the communication networks are hypercubes and grids, the task assignment problem can be modeled by grid and hypercube embeddings.

The *Mob* heuristic for graph partitioning [19,21] is the basis for the heuristics reported in this paper. It identifies and swaps equal-sized collections ("mobs") of "promising" vertices between the two sets of a partition. The heuristic begins with a large mob size of about 10% of the vertices, and decreases the mob size every time a swap causes the number of edges joining the two sets to increase. The mob size is decreased monotonically to 1 and then recycles, perhaps increasing the number of steps on each cycle. We have generalized this technique to grid and hypercube embedding by defining a large number of partitions, selecting between them at random, and running a variant of the graph-partitioning *Mob* heuristic on them. We implemented this algorithm on the CM-2 Connection Machine. It gives very good partitions of large graphs in very little time.

Because graph partitioning is an important NP-complete problem [11], much effort has been devoted to developing good heuristics for it. (See for example, the paper by Johnson et al [13].) The *Mob* heuristic is as effective for graph partitioning as simulated annealing (SA) [15] and the Kernighan-Lin (KL) heuristic [14], the best serial heuristics for GP, but exhibits a high degree of parallelism and gives bisection widths that are at least as good as these heuristics [19,21]. As we show later, the new *Mob* heuristics are as effective for hypercube and grid embedding as the most effective serial heuristics for these problems.

Below we define the graph embedding problems and give an overview of related work in which simulated annealing plays an important role. In Section 2 we define the *Mob* heuristic and give an intuitive explanation of its operation. In Sections 3 and 4 we present our experimental results for the embedding of random graphs and random geometric graphs into the grid and hypercube, respectively. Finally, in Section 5 we draw conclusions and discuss future research.

1.1 The Graph Embedding Problem

Definition 1 Given two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, an *embedding* of G_1 into G_2 is a pair of mappings $S = \nu, \epsilon$ where $\nu : V_1 \mapsto V_2$ maps vertices of V_1 to those in V_2 and ϵ maps edges in G_1 to paths in G_2. The numbers of source vertices mapping to each target vertex is constrained to be the same.[2] Let $\mathcal{S}$ be the set of all graph embeddings S of the *source graph* G_1 into the *target graph* G_2. The *cost function* $f : \mathcal{S} \mapsto \mathcal{R}$ assigns a cost to each embedding in $\mathcal{S}$. The *graph-embedding problem* is to find a graph embedding S with small cost $f(S)$.

Graph partitioning is a special case of graph embedding in which the target graph, G_2, consists of just two nodes and half the source nodes are mapped to each of them. The goal of the graph-partitioning problem is to embed an undirected graph G_1 so that the number of edges between the targe nodes (the *bisection width*) is minimized.

Grid embedding is graph embedding into grids, regular two-dimensional arrays of elements with edges between adjacent vertices. Figures 1 and 2 show a 1000-node random graph of average degree $d = 5$ embedded into 8×8 and 32×32 grids before and after application of our new *Mob* heuristics, respectively. In Figure 1 sixteen vertices of the source graph are embedded into one grid vertex whereas in Figure 2 one source vertex is embedded into each grid vertex.

Hypercube embedding is graph embedding into hypercubes. The vertices of a hypercube can be represented by binary n-tuples, one tuple per vertex, and vertices whose binary tuples differ in one component are adjacent.

The goal of grid and hypercube embedding is to minimize the sum of the lengths of edges in the target graph. This is captured by the $PERIMETER$ and $HCUBE$ cost functions defined below. While these measures do not capture the congestion in embeddings, namely, the number of wires or messages passing through one edge in the target graph, they do provide good heuristics for initial assignment of tasks to processors and components to sites.

Definition 2 Let $S : V \mapsto \{0..k-1\} \times \{0..l-1\}$ be an embedding of G into a $k \times l$-vertex grid. The cost of the embedded edges under the $PERIMETER$ cost function is defined as $\sum_{(v,w)\in E} PERIMETER(S(v), S(w))$, where $PERIMETER(a,b) := |a_x - b_x| + |a_y - b_y|$ is the half-perimeter of a box enclosing two grid nodes a and b with x and y coordinates a_x, a_y and b_x, b_y, respectively.

[2]If necessary, the number of source vertices $|V|$ is increased by adding unconnected vertices to V to insure this condition is satisfied.

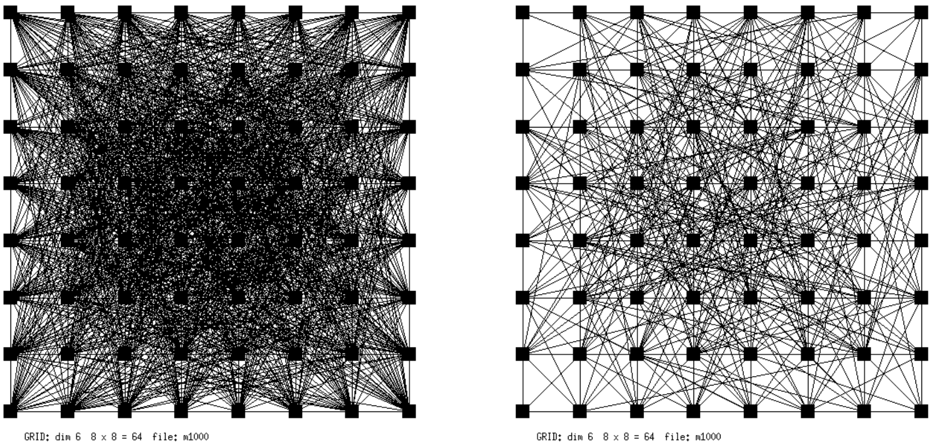


Figure 1: 16-to-1 grid embedding of a random graph with $|V_1| = 1000$, $d = 5$, and $|V_2| = 64$: (a) A randomly generated solution. (b) A solution generated by the *Mob* heuristic.

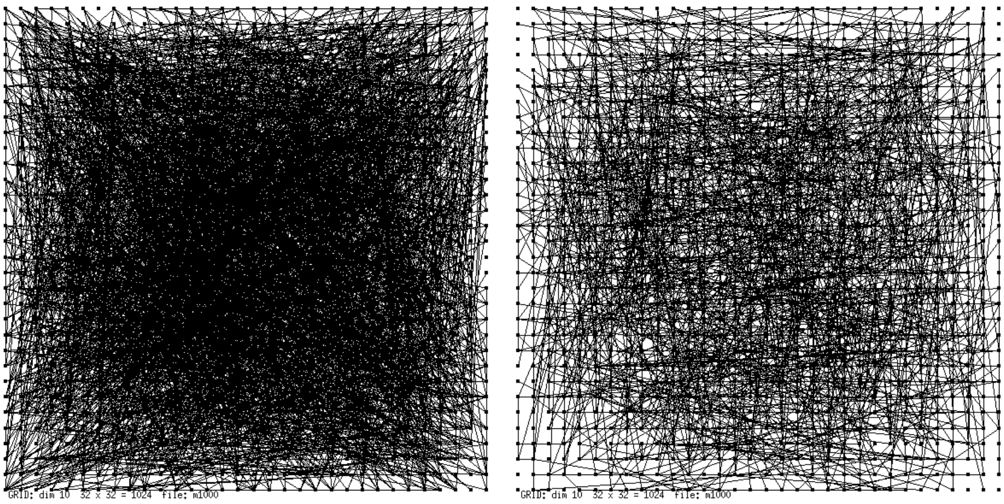


Figure 2: 1-to-1 grid embedding of a random graph with $|V_1| = 1000$, $d = 5$, and $|V_2| = 1024$: (a) A randomly generated solution. (b) A solution generated by the *Mob* heuristic.

Definition 3 Let $S : V \mapsto \{0..2^k-1\}$ be an embedding of G into a 2^k-vertex hypercube. The cost of the embedded edges under the $HCUBE$ cost function is $\sum_{(v,w)\in E} HCUBE(S(v), S(w))$, where $HCUBE(a,b)$ is the Hamming distance between a and b, i.e. the number of bit positions in which the vertices a and b differ when represented as binary k-tuples.

1.2 Previous Experimental Work

In an early paper, Bokhari [4] gives a heuristic to embed communication graphs into an $n \times n$ 2-D grid with connections between horizontal, vertical, and diagonal neighbors. His cost function is the number of adjacent source vertices mapped to adjacent target vertices. The goal is to maximize this cost function. Given an initial embedding, the heuristic swaps each vertex with another vertex that produces the largest change in cost. The cost of the new embedding is computed, and if it increases, it is accepted. If not, n random pairs of vertices are swapped and the process repeated. Experiments were conducted with thirteen graphs of up to 80 edges embedded into square grids of up to seven vertices on a side. Respectable increases in the value of the cost function were obtained.

Kravitz and Rutenbar [16] give a parallel version of SA for VLSI placement. Because non-interacting moves are considered for acceptance, the number of possible parallel moves tends to be small. Also, the strategy of only accepting certain moves introduces artifacts which degrade convergence behavior.

Parallel SA heuristics for VLSI circuit placement have been studied by Casotto and Sangiovanni-Vincentelli [5] for the CM-1, Wong and Fiebrich [22] for the CM-2 Connection Machine, and by Darema *et al* [9] for grid-based multiprocessors. Casotto *et al* used a 16-K processor CM-1 on a circuit with 800 cells and 843 nets (hypergraph edges) and 2935 pins. One iteration of their SA algorithm took 2 seconds. Wong and Fiebrich used several circuits with between 800 and 1,200 cells and others with 6,500 cells, and on a 32K CM-2 achieved a factor 200 speedup over a VAX 11/780. Darema *et al* demonstrated a speedup of 14 in their parallel SA heuristic to embed a 9×9 grid onto itself when simulating up to 32 virtual processors.

Chen, Stallmann, and Gehringer [6] examine the running time and performance of twelve hypercube-embedding algorithms. They compare these by embedding random graphs, geometrical random graphs, trees, hypercubes and hypercubes with randomly selected missing edges into a 128-node hypercube. The algorithms considered include SA [9], SA restricted to moves across hyperplanes (SAC) [6], Kernighan-Lin applied to the all partitions of the cube along hyperplanes (RMB) [10], greedy heuristics such as steepest descent, and local search heuristics using as neighborhoods all possible pairs of vertices (LS) and pairs of vertices adjacent across a hyperplane (LSC). They conclude that SA "performs better as the communication (source) graph becomes more random, and greedy heuristics perform better as the communication graph approaches a hypercube." SA is also found to be the most flexible, giving the best embeddings on random, random geometric graphs and trees at running times that are about 2, 7 and 40 times that of the next best heuristic for the three problems, respectively.

Roussel-Ragout and Dreyfus [18] propose a parallel implementation of SA on a MIMD multiprocessor. Every processor evaluates one move and at most one of the accepted moves is chosen at random by a master processor. All processors then update their data structures to incorporate the executed move. They show equivalence with serial SA but the degree of parallelism is very small and their technique has a serial bottleneck at high temperatures. Banerjee *et al* [2] have implemented SA for an Intel Hypercube and their version is also consistent with serial SA. As a consequence, no large benefits in solution quality are obtained and the time they spend to maintain serial consistency is very great.

Parallel simulated annealing has been applied by Dahl [8] to reduce communication costs (measured in the HCUBE metric) on the CM-2 Connection Machine hypercube network. Multiple source vertices are mapped to a target hypercube vertex and assigned a hypercube dimension. A dimension is chosen at random. SA is applied independently to each edge along this dimension which have source vertices assigned to them and it determines whether pairs of sources vertices swap or not. The energy function is computed ignoring the possibility that other vertices may swap. This approach works well for mappings of $G_1 = (V_1, E_1)$ into $G_2 = (V_2, E_2)$, where $|V_2| \approx |V_1|/\log|V_1|$ but it serializes for 1-to-1 mappings ($|V_1| = |V_2|$), and for many-to-1 mappings where $|V_1| \gg |V_2|$, such as graph partitioning ($|V_2| = 2$).

2 The *Mob* Heuristic for Graph Partitioning

The *Mob* heuristic for graph partitioning described in [19,21] is the basis for our two new *Mob*-based heuristics for grid and hypercube embedding. We begin by describing it in detail.

The *Mob* heuristic is given an initial partition $K = (A, B)$ of the vertices V of a graph $G = (V, E)$ into two equal-sized sets A and B. It identifies two equal-sized subsets of A and B of "promising" vertices ("mobs") and swaps them. These promising vertices have high "gain," that is, they are amongst those that might cause the largest reduction in bisection width when swapped. After mobs are swapped, *Mob* computes the bisection width of the new partition. If the bisection width decreases, two new mobs of the same size are formed and swapped. If it increases, the mob size is reduced according to a predetermined "schedule" and two mobs of the new size are formed and swapped. After mobs are swapped, the bisection width is again computed, appropriate action taken, and the process repeated until the mob size reaches a minimum value of 1. This cycle may be repeated several times.

A mob is chosen from each of the sets A and B of a partition K by computing for each vertex v in each set a $gain(v)$, the decrease in the bisection width caused by moving v to the other side of the partition. An array MS called the *mob schedule* is given in advance; the sth mob size is $MS[s]$. The initial mob size, $MS[1]$, is about 10% of $|V|$. A mob of set A of size $MS[s]$ is formed from a *premob* PM_a, the set of vertices in A of gain of at least g

```
Let K = (A,B) be a partition of V in G = (V,E);
Let Ra[1..q(n)], Rb[1..q(n)] be random variables over (0,1);
Let MS[1..q(n)] an integer-valued mob schedule;

Mob(K)
    t = 1; s = 1;
    while( t ≤ q(n) )
    {
      L = MOB-NR(K, MS[s], Ra[t], Rb[t]);
      D = bw(L) - bw(K);
      if( D > 0 )
        s = s + 1;
      K = L;
      t = t + 1;
    }
    return K ;
```

Figure 3: The generic *Mob* heuristic

where g is the largest gain such that PM_a has at least $MS[s]$ vertices. A premob PM_b of the set B is formed in the same way.

Premobs can be formed quickly on parallel machines by broadcasting a gain threshold g_s to all processors holding a vertex and its gain, and counting the number of vertices with gain g_s or larger, and then using bisection until the correct gain value has been found. Binary search on the value of g_s is used until a value is assigned to g_s that selects a smallest premob of size at least $MS[s]$.

The tth mob of A (B) of size $MS[s]$ is formed from PM_a (PM_b) by choosing $MS[s]$ vertices at random using a uniform random variable $Ra[t]$ ($Rb[t]$) over the interval $(0,1)$. The vertices in a premob are ordered and counted. The number pm_a (pm_b) of vertices is multiplied by $Ra[t]$ ($Rb[t]$), rounded down, and added modulo $MS[s]$ to the index of each vertex. Those with index in the set $(0..MS[s]-1)$ are made members of a mob. These steps can be implemented efficiently on a parallel machine.

The mob selection algorithm is implemented by the procedure *MOB-NR(K, MS[s], Ra[t], Rb[t])* shown in Figure 3 as part of the complete algorithm for the generic *Mob* heuristic. This procedure returns a new partition L in the neighborhood of the starting partition K. A time limit $q(n)$ is placed on the procedure, where $q()$ is a polynomial in the size $n = |V|$ of the problem. If the bisection width $bw(L)$ of L is less than that of K, the mob size $M[s]$

remains unchanged. Otherwise the next mob size is selected.

The *Mob* heuristic has elements in common with simulated annealing (SA) [15] and the Kernighan-Lin Heuristic (KL) [14]. SA explores neighborhoods by picking a pair of vertices of highest gain in each half of a partition and computing the effect on the bisection width if they are swapped. If the bisection width would decrease, the swap is made. If not, the swap is made with a probability that decreases exponentially in $D/Temp[t]$, where D is the change in the bisection width and $Temp[t]$ is a temperature parameter that is a non-increasing function of the time t. KL starts from a given partition and examines all partitions obtained by swapping and freezing pairs of vertices in order of decreasing gain until all vertices have been swapped. It moves from the given partition to the partition in this sequence which has the smallest bisection width. This process is repeated until no improvement is possible. The *Mob* heuristic has a randomizing element, as does SA, and explores a large neighborhood, as does KL.

2.1 Intuitive Explanation

The *Mob* heuristic is effective because in initial random partitions there are typically many vertices which, when swapped, cause a large reduction in the bisection width. This is especially true when the graphs in question have low degree because few candidates for a swap have edges in common, and swapping them in a group has about the same effect on the bisection width as swapping them individually. As the bisection width of the graph decreases, there are fewer vertices that can cause a large reduction in the bisection width, and swapping a large collection would cause the bisection width to increase rather than decrease. For this reason the mob size is adaptively decreased.

An analogy can be drawn between the movement of mobs and the rolling of a ball in a solution space where the goal is to find a global minimum. If the space is structured properly, a large ball rolling downhill will quickly find the region containing the global minimum but, due to its large size, will not find the global minimum itself. Replacing the ball with a ball of smaller size allows the algorithm to get closer to the global minimum.

2.2 The Mob Heuristic for Graph Embedding

When vertices in source graphs G are embedded into 2-D grid and hypercube target graphs H, there exist many ways to choose vertex sets for swapping. If we don't restrict the types of swap, the computation time for one iteration of our embedding heuristics could be very large. On the other hand, if our swapping regime does not permit enough movement (or *mixing*) of embedded vertices, it may be difficult to find good embeddings. Thus, our embedding heuristics need to balance the cost of one iteration with the number of iterations. The heuristics also need to insure good mixing so that over time it is possible for most pairs of vertices to be swapped. This argues for a *probabilistic embedding heuristic*.

```
E is an embedding of the source graph G into the target graph H;
I is an initial embedding;
p is a partition of the vertices of E;
P(G) is a set of partitions of the target graph H;
Mob(E,p) is the Mob heuristic on partition p in P(G) of embedding E;
T[n] is a polynomial in n, the number of vertices in G;

E = I;
for ( 1 ≤ t ≤ T[n])
{
    Choose p randomly from P(G);
    E= Mob(E,p);
}
```

Figure 4: The *Mob* heuristic for grid and hypercube embedding

The *Mob* heuristic for graph partitioning has a randomizing element. Vertices in a mob are chosen by selecting randomly among vertices in a premob. For graphs embedded in the grid and hypercube, more mixing than this is needed. Since the *Mob* heuristic for graph embedding is a proven parallel heuristic, our goal was to combine it with an appropriate partition selection mechanism to provide good mixing, yet have the heuristic run efficiently on parallel machines.

Shown in Figure 4 is pseudo-code for the *Mob* heuristic for grid and hypercube embedding. The heuristic $Mob(E, p)$ is given an initial embedding E of the source graph G into the target graph H and a partition p of the vertices chosen at random from a set of partitions. It produces a new embedding and loops for a polynomial number of steps. $Mob(E, p)$ is almost identical to *Mob* for graph partitioning except that vertices are paired for swapping, as described below.

2.3 Partition Selection for Graph Embedding

For efficiency reasons, we pair vertices for swapping. Defined below are "natural" partitions of grid and hypercube vertices chosen to insure good mixing. In these definitions we assume that only one source graph vertex is mapped to a grid or hypercube vertex. The case in which more than one source vertex is mapped to a target vertex is discussed later. The $k \times k$ grid consists of vertices with coordinates $\{(x, y) | 0 \le x, y \le k-1\}$.

Definition 4 The ith partition, P_i, $0 \leq i \leq k-1$, on the 2^k-vertex hypercube is defined by $P_i = (S_{i,0}, S_{i,1})$ where $S_{i,j} = \{v | v_i = j\}$ and v_i is the ith bit in the binary representation of v. Candidates for swapping under partition P_i are vertices v and w that differ in the ith bit.

Partitions of the $k \times k$ grid, $k = 2^l$, are defined for the x and y directions for $d = 2^r$, $0 \leq r \leq l-1$. The partition $P_{x,d}$ is defined by $P_{x,d} = (S_{x,d,0}, S_{x,d,1})$ where $S_{x,d,i} = \{(u,v) | \lfloor u/d \rfloor (mod\ 2) = i\}$. Candidates for swapping have indices $\{(u_1, v)\,(u_2, v) | u_1 = (2i)d + j, u_2 = (2i+1)d + j, 0 \leq j \leq d-1, 0 \leq i \leq (2^l/2d) - 1\}$ or have indices $\{(u_1, v)\,(u_2, v) | u_1 = (2i+1)d + j, u_2 = [(2i+2)d + j] mod\ 2^l, 0 \leq j \leq d-1, 0 \leq i \leq (2^l/2d) - 1\}$. $P_{y,d}$ and candidates for swapping in the y direction are defined similarly.

Swap gains are defined as the sum of the gains of vertex pairs that are candidates for swapping. Vertex gains are computed using the $PERIMETER$ and $HCUBE$ cost functions for grids and hypercubes, respectively.

When more than one source vertex is mapped to a target vertex, there is more than one way to select source vertex pairs for swapping. We have found that it is effective to match up source vertices that have the largest gain and to match the remaining vertices arbitrarily. This procedure saves time and provides good convergence behavior.

In the next section we discuss some of the practical details associated with the implementation of these algorithms on the Connection Machine.

2.4 Implementation of *Mob* on the Connection Machine

We now describe the key aspects of our implementation of the *Mob*-based heuristics on the CM-2. The CM-2 Connection Machine is a massively parallel computer consisting of up to 64K moderately slow one-bit processors [1,12] organized as a 12-dimensional hypercube with sixteen nodes at each corner of the hypercube.

We use edge and vertex data structures to support the exchange of vertices and the computation of gains and costs with minimal communication overhead. Our edge data structure is based on that described by Blelloch [3] and implemented for the graph partitioning *Mob* heuristic [21].

The Connection Machine supports the concept of virtual processor. Our implementation assigns one virtual processor per record in each vertex and edge data structure.

Edge Data Structure The edge data structure is constructed of records representing edges of an undirected source graph G. Each edge (a, b) appears twice, as the pairs (a, b) and (b, a). The records are sorted and pointers introduced between the two instances of an edge. The set of pairs with a given first component, say, a, constitute the edges adjacent to vertex a. The first pair in such a sublist does double duty; it represents the first edge as well as the vertex a. Fields are introduced in each record so that the leading record in a sublist can record data about source vertices.

Vertex Data Structure Our grid and hypercube heuristics map the same number of source vertices to each target vertex. Unfortunately, the edge data structure does not lend itself to the rapid identification of those vertices mapped to a given target vertex nor to the rapid pairing and swapping of vertices. By experimentation we have found that a most effective way to handle these issues is to add a vertex data structure, a linear array of records, one per source vertex. Each record contains a pointer to the edge record in the edge data structure representing the source vertex. In the vertex data structure source vertices mapped to the same target node are adjacent and they are laid out in memory so that it is easy to determine the location of the target vertices to which they map and to swap them. The swapping of records is efficient; the vertex data structure allows efficiently supported communication patterns on the CM-2 to be used.

The vertex data structure is not needed in the *Mob* heuristic for graph partitioning because the set into which a vertex falls can be identified with a single bit and source records themselves need not move.

Cost and Gain Computations The cost of each embedded edge is computed using the pointers between edges in the edge data structure. "Scan" operations[3] on this data structure sum the edge costs; they are divided by two to determine the cost of an embedding for a given partition since every edge has a twin.

Given a partition, the gain of each vertex is computed by summing the contributions of each edge incident on that vertex using a segmented additive scan in reverse order on the edge data structure, summing edge gains into a vertex gain. Segments are defined by the edges incident on a given vertex.

Selecting and Exchanging Mobs The procedure for selecting premobs and mobs is described above. The vertex data structure is used to identify vertices in each half of a partition. This information is transmitted to the edge data structure which is then used to rank the vertices in each half of a partition. This information is used to select a mob from a premob.

After every vertex has been told whether or not it will move, all edge records associated with vertices become active and communicate their new positions to all adjacent edges by a segmented Copy-Scan. Edges then notify their twin edges of their new positions using Send operations and records in the vertex data structure move to their new positions.

[3] A *scan* operation on a vector $x = (x_1, x_2, \ldots, x_p)$ returns a vector $y = (y_1, y_2, \ldots, y_p)$ such that $y_1 = x_1$ and $y_i = y_{i-1} \oplus x_i$ where $\oplus$ is an associative operation. Segmented scan operations are scan operations performed over contiguous segments of the full vector. If a segment begins at the ith position, $y_i = x_i$; otherwise, $y_i = y_{i-1} \oplus x_i$. Scans and segmented scans can be implemented efficiently on most parallel machines [3].

3 Experimental Results for Random Graphs

The performance of the *Mob* hypercube and grid-embedding algorithms were evaluated by conducting experiments on the CM-2. 1-to-1 and 16-to-1 mappings of source to target graphs were studied. The random source graphs used in these experiments are more than 1000 times larger than those previously studied in the literature. Serial algorithms such as SA and KL have prohibitive running times for such large graphs. We measured the solution quality, convergence as a function of time, and running times with different numbers of processors, and compared *Mob* to other algorithms. The *Mob*-based embedding heuristics on source graphs $G = (V, E)$ produce excellent solutions, converge quickly, run in time empirically found to be $O(\log |E|)$ with $2|E|$ processors, and are well matched to the CM-2 hardware.

We conducted experiments on randomly generated graphs with up to 512K vertices and 1M edges. Our random graphs were generated by selecting pairs of vertices at random, connecting them with an edge, and removing multiple edges and self-loops. The probability of selecting a vertex was chosen to provide graphs with integral average degree d. We did not use the standard approach of generating edges by flipping a coin with the appropriate probability for all potential edges in a graph because the number of trials would have been far too large. The degrees considered range from 3 to 16, degrees typical in VLSI and processor mapping problems.

The number of edges, vertices and average degree of the random graphs used in our experiments are given in Table 1. We used six graphs of average degree 4 and six graphs of average degree 8 to study the effect of increasing graph size on solution quality and running time. We performed experiments on nine graphs with 16K vertices and average degrees ranging from 3 to 16 to examine the effect of graph degree on the behavior of the *Mob* algorithms. The data shown in the tables was averaged over at least five runs.

3.1 Mob Schedule for Graph Embedding

The mob schedule specifies how many element pairs can swap in one iteration. It also determines the rate of convergence of the heuristics and the quality of the results. Let $|E|$ be the number of edges in a graph. Through experimentation the following characteristics of mob schedules were found to be effective for our random source graphs and embeddings into both grids and hypercubes:

- A maximum mob size of about $|E|/8$.
- Decrementing the mob size in 16 uniform steps in the first cycle.
- Doubling the number of steps on each subsequent cycle.
- Terminating the computation after 8000 steps.

The above maximum mob size corresponds to two vertex pairs per hypercube node in the 16-to-1 mappings and thus shows a preference for the specially selected vertices of maximum gain at each hypercube node but also moves other vertices. We have found that this schedule combines initial rapid convergence and very good results at the later stages of the algorithm when the possible improvements get smaller. We found that this schedule works well with 16-to-1 and 1-to-1 mappings and both grid and hypercube embeddings.

3.2 Solution Quality of *Mob*

The quality of the solutions produced by *Mob* is shown in Tables 2 and 3 for 1-to-1 and 16-to-1 embeddings into hypercubes and grids. The columns labeled *Dim* give the dimension of the hypercube into which the graph is embedded. For the grid embeddings, *Dim* is the dimension of the hypercube containing a $2^{\lfloor Dim/2 \rfloor} \times 2^{\lceil Dim/2 \rceil}$ grid. The columns labeled R give the average edge length produced by a random embedding, and those labeled Mob give the average edge length in an embedding produced by *Mob*, each normalized by the number of edges $|E| = nd/2$ in a graph, where d is the average degree of a graph. The columns labeled Mob/R give the ratio of improvement produced by *Mob* over a random solution. The *Mob* heuristic offers impressive reductions in embedding costs. Our experiments show:

- For fixed degree d, Mob/R is largely independent of graph size.
- 16-to-1 mappings give slightly better Mob/R ratios than 1-to-1 mappings.
- The grid-embedding *Mob* heuristic achieves lower Mob/R ratios than the hypercube *Mob* heuristic.
- The ratio Mob/R rises with increasing average graph degree, as shown in Figure 5 (a). The differences between grid and hypercube embeddings and between 1-to-1 and 16-to-1 embeddings become smaller with increasing degree.

3.3 Rates of Convergence of *Mob*

Tables 2 and 3 also report the convergence of *Mob* for an increasing number of iterations. The columns labeled Iterations shows the average embedding cost as percentages above the best-ever embedding cost (the best solution found by any run of the mob), produced after 100, 1000 and 4000 iterations of *Mob*, respectively. As shown, the number of iterations needed to reach a fixed percentage above the best-ever embedding cost is approximately independent of graph size for graphs of fixed average degree. This important observation holds for hypercube and grid embeddings and for 1-to-1 and 16-to-1 mappings. The fact that *Mob*'s rate of convergence on random graphs is independent of graph size was also observed for graph partitioning (see [19,21]).

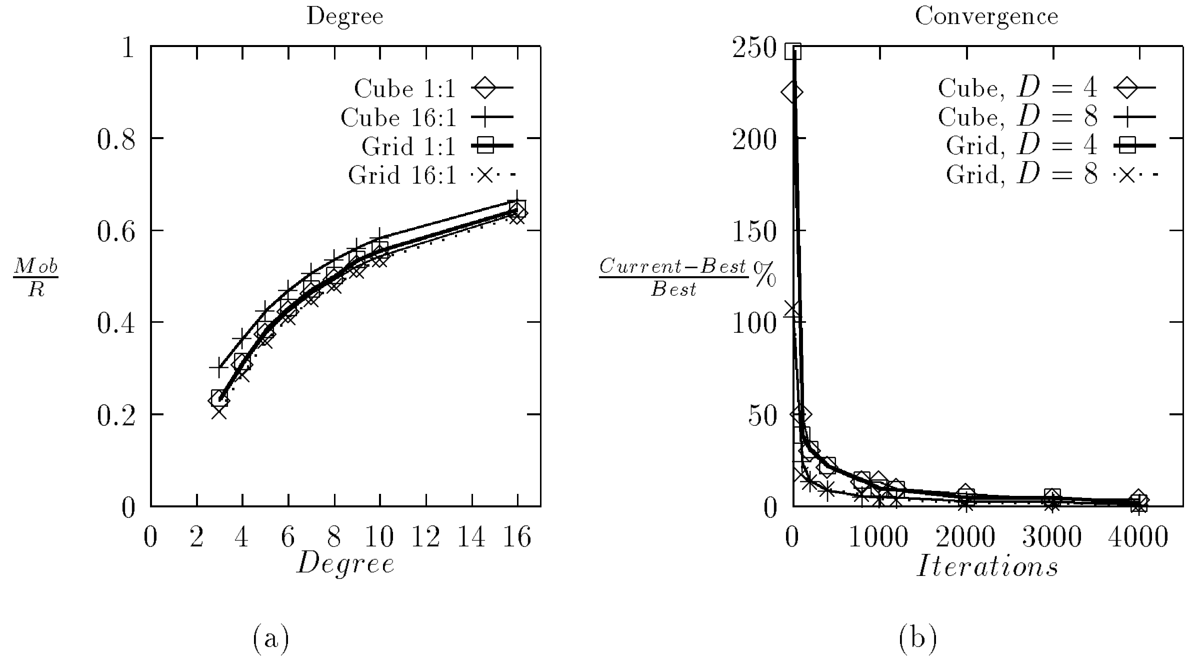


Figure 5: (a) Ratios of best *Mob* embedding cost to random embedding cost for random graphs plotted as a function of graph degree. (b) *Mob* convergence behavior after a number of iterations expressed as *Mob*'s bisection width as a percentage over the best solution obtained.

The Iterations columns in Tables 2 and 3 show that the reductions in embedding costs decrease rapidly as the total number of iterations increases. The rate of convergence is shown in Figure 5 (b) for 16-to-1 mappings of the *Mob* grid and hypercube-embedding algorithms. Thus, a good solution is produced rapidly; further improvements can be obtained if enough computation time is available. Also, the number of iterations to achieve a given percentage decreases as the degree d increases. Thus, fewer iterations are needed on high-degree graphs.

3.4 Computation Time

The edge data structure, which is larger than the vertex data structure, requires $2|E|$ virtual processors. The scan operations are the most complex operations in an iteration of *Mob*. Thus, one iteration of *Mob* is estimated to run in time $O(\log |E|)$. This was tested by experiments on the CM-2 with graphs of different sizes in which the number of real processors varied between 8K and 32K and the number of virtual (simulated) processors ranged between 16K and 2M.

The results of these experiments are reported in Table 4 for hypercube embeddings and in Table 5 for grid embeddings. We give evidence that the time per iteration normalized by the number of virtual processors grows logarithmically on the CM-2. Since the graph size that can be handled on the CM-2 is bounded by the memory required for each virtual processor, as more real processors and memory are added, execution times should keep decreasing and embeddings of increasingly larger graphs should be computable.

As observed above, the total number of iterations required by *Mob* to reach a fixed percentage above the best-ever embedding cost appears to be approximately constant. It follows that the empirical parallel running time of the *Mob* heuristic on sparse random graphs is $O(\log |E|)$ with $2|E|$ processors.

Tables 4 and 5 also indicate that one iteration of the *Mob* grid-embedding heuristic took approximately 40-50% more time than the corresponding iteration of the *Mob* hypercube-embedding heuristic. This is explained by the fact that the PERIMETER cost and gain functions require more arithmetic operations than their HCUBE equivalents. The 16-to-1 embeddings execute slightly slower than the 1-to-1 mappings, since *Mob* must select a vertex with maximum gain for every target vertex and move it to the head of this vertex list, as described in Section 2.4.

The absolute speed at which an embedding is produced by *Mob* is remarkable and shows that *Mob* can be implemented very efficiently on a SIMD-style machine. On a 32K CM-2 it takes approximately 1720 seconds ($\approx$ 29 minutes) to find an embedding of a 500K-vertex, 1M-edge graph into a 15-dimensional hypercube, and approximately 1264 seconds ($\approx$ 21 minutes) to embed that graph into a 19-dimensional hypercube. It takes approximately 2370 seconds ($\approx$ 40 minutes) to find an embedding of a 500K-vertex, 1M-edge graph into a 256×64 grid, and approximately 2157 seconds ($\approx$ 36 minutes) to embed that graph into a 1024×512 grid. All these embeddings are within 5 percent of best-ever.

3.5 Comparison to Graph-partitioning Algorithms

To calibrate our results against those obtained for graph partitioning, we computed the bisection width of the graphs embedded into grids and hypercubes. We found that the bisection widths for hypercube embeddings was about the same for all hyperplanes whereas for grid embeddings, the two partitions dividing the grid in half vertically and horizontally gave the best partitions.

Table 6 shows how the *Mob* hypercube and grid embedding algorithms perform as graph-partitioning algorithms. The data for random graphs on the performance of the *Mob* graph-partitioning algorithm and the KL graph-partitioning algorithm is taken from our study of local search graph-partitioning heuristics in [19,21]. The cost of the graph embeddings $P = (A, B)$ is given in Table 6 by the percentage of all edges that cross the cut between A and B. We found that 16-to-1 grid and hypercube embeddings with our *Mob*-based heuristics produced bisection widths comparable to those for the *Mob* heuristic for graph-partitioning. The KL graph-partitioning algorithm, which we ran on graphs with only 32K (or fewer) vertices due to running time considerations, was significantly outperformed by both 16-to-1 *Mob* embedding algorithms. KL gave slightly worse embeddings than the 1-to-1 *Mob* grid-embedding algorithm and slightly better embeddings than the 1-to-1 *Mob* hypercube-embedding algorithm.

The performance of the *Mob* embedding algorithms interpreted as graph-partitioning algorithms is remarkable, considering that *Mob* is optimizing the "wrong" cost function. While the data in Table 6 cannot show conclusively how good the *Mob* embedding algorithms are, the existence of a better graph-embedding algorithm would also imply the existence of a better graph-partitioning algorithm.

3.6 Comparison to Simulated Annealing

Chen *et al* [6,7] evaluated the performance of 1-to-1 hypercube-embedding heuristics. The graphs examined were random graphs, random geometric graphs, random trees, hypercubes, and hypercubes with randomly added and deleted edges. All graphs had 128 vertices and were embedded in a 7-dimensional hypercube. Among the algorithms tested, simulated annealing with a move set limited to swapping vertex pairs along hypercube edges produced the best solutions.

We obtained the ten random graphs used by Chen *et al* and generated ten random graphs ourselves. For each graph five runs were performed, and results were averaged over these runs. Each run of *Mob* was limited to 8000 iterations, and the schedule described above was used. Chen *et al* report that on ten random graphs of average degree 7, a reduction of 58.1% over the cost of a random embedding was achieved. We found that the average solution produced by *Mob* is 58.28% for Chen *et al*'s graphs and 58.17% for our own graphs. Thus *Mob*'s performance was about equal to the performance of a tuned version of SA.

4 Experimental Results for Geometric Graphs

We performed experiments on *random geometric graphs*, which have more structure than random graphs. The cost ratio of minimum embedding to random embedding tends to be much smaller than for random graphs. This makes it more desirable to find a good embedding, but also suggests that a good embedding is harder to find. A *random geometric graph* $G_{n,d}$ with n vertices and average degree d is generated by randomly selecting n points in the unit square $[0,1) \times [0,1)$. The geometric graph $G_{n,d}$ contains an edge if its endpoints are a distance r or less apart, as measured by a distance metric. To obtain graphs of degree d, the distance parameter r is set to the value r_d. Figure 6(a) shows a random geometric graph, and Figure 6(b) shows a grid embedding of this graph computed by placing each vertex on a neighboring grid point. The following three metrics have been used in the literature:

- *Manhattan metric* $r = |x_1 - x_2| + |y_1 - y_2|$
  $r_d = \sqrt{d/2n}$

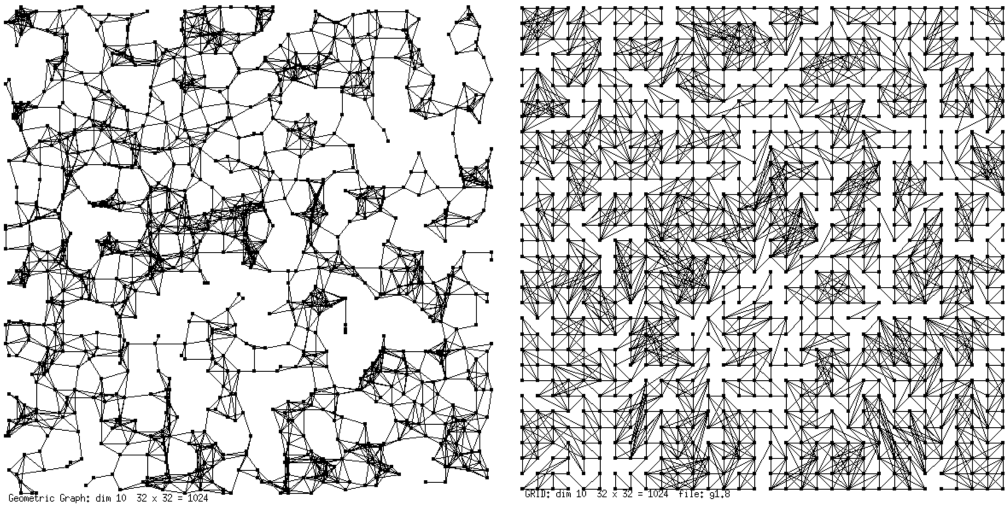


Figure 6: (a) A random geometric graph with $|V_1| = 1024$, $d = 8$ on the unit plane. (b) Grid embedding of the geometric graph with $|V_2| = 1024$, obtained by placing each vertex on a neighboring grid node.

- *Euclidean metric* $r = \sqrt{|x_1 - x_2|^2 + |y_1 - y_2|^2}$
  $r_d = \sqrt{d/\pi n}$
- *Infinity metric* $r = \max(|x_1 - x_2|, |y_1 - y_2|)$
  $r_d = \sqrt{d/4n}$

For our experiments we used the Manhattan metric, since it matches the cost function used for grid embeddings. Graph-partitioning experiments by Johnson *et al* [13] were performed on random geometric graphs generated with the Euclidean metric. Chen *et al* [6,7] used random geometric graphs generated with the infinity metric to test hypercube-embedding algorithms.

4.1 Efficient Construction of Random Geometric Graphs

We now address the problem of efficiently generating geometric graphs. The naive method of computing the distance of every vertex pair on the unit square leads to an $O(n^2)$ time algorithm. This approach is perfectly adequate for small graphs. However, a more sophisticated method is required for graphs with $1,000,000$ vertices or more.

Our approach is to divide the unit square into $1/r_d \times 1/r_d$ cells. It follows that all vertices a distance r_d or less apart must be located in the same cell or in one of the eight neighboring cells. (This holds for any of the above metrics.) Every vertex computes the cell it belongs to, and the vertices are sorted by their cell value into a linear vertex array. Vertices in the same cell are now adjacent in the vertex array. An indirection table is constructed that contains for every cell i the address in the vertex array of the first vertex in cell i, or a value of -1 if cell i is empty; this facilitates finding the contents of the eight neighboring cells.

Since n vertices were distributed randomly over the unit square, the number of cells on the unit square is $1/r_d \times 1/r_d = 2n/d$ for the Manhattan metric. Thus cells contain an average of $d/2$ vertices. An average total of $9nd/2$ distance computations is done to generate $nd/2$ edges, and the total computational work is $O(n \log n + nd)$. Under the realistic assumption that sorting n vertices by their cell value takes time $O(log^2 n)$ with $O(n)$ processors, the above algorithm is easily parallelized to run in time $O(\log^2 n + d)$ with $O(n)$ parallel processors. Our experiments show that the constants are very small.

Note that the above search structure works only for points that are distributed randomly in the plane. More sophisticated algorithms, such as quad-trees, Voronoi diagrams, and trapezoidal decompositions, have been developed in the field of computational geometry to deal with nearest-neighbor problems. The considerable implementation complexity and (usually) $O(n \log n)$ storage-space requirements make these algorithms inappropriate for the special case of generating geometric graphs, since the much simpler algorithm described above exists.

4.2 Analysis of the Cost of Embedding Geometric Graphs

We assume that our graphs will be embedded into the $\lfloor\sqrt{n}\rfloor \times \lceil\sqrt{n}\rceil$ grid. We estimate g_{min}, the minimum cost grid embedding, by assuming that the vertices of $G_{n,d}$ in the unit square are grid vertices, computing the sum of the Manhattan lengths of its edges, and multiplying the sum by $\sqrt{n}$. The average distance between two adjacent vertices in the unit square is easily shown to be $r_d/\sqrt{2}$, giving the following estimated total edge cost:[4]

$$g_{min} \approx \frac{nd}{2}\frac{r_d\sqrt{n}}{\sqrt{2}} = \frac{n}{\sqrt{2}}\left(\frac{d}{2}\right)^{3/2} \tag{1}$$

On the other hand the average cost of a random embedding of $G_{n,d}$ into the $\lfloor\sqrt{n}\rfloor \times \lceil\sqrt{n}\rceil$ grid embedded into the unit square, g_r, can be estimated by

$$g_r = \frac{nd(\lfloor\sqrt{n}\rfloor + \lceil\sqrt{n}\rceil)}{6} \tag{2}$$

since the average length of an edge whose endpoints are randomly chosen from the $\lfloor\sqrt{n}\rfloor \times \lceil\sqrt{n}\rceil$ grid is $(\lfloor\sqrt{n}\rfloor + \lceil\sqrt{n}\rceil)/3 - 1/3\lfloor\sqrt{n}\rfloor - 1/3\lceil\sqrt{n}\rceil$. The column labeled R in Table 9

[4]An exact analysis of the total cost edge averaged over the ensemble of all geometric graphs on n vertices of average degree d shows it to be $(2/3)n(d/2)^{3/2}$ when $d \ll n$.

is the computed average total length of a random embedding normalized by the number of edges; the match is almost perfect.

Because we can fold an $\lfloor\sqrt{n}\rfloor \times \lceil\sqrt{n}\rceil$ grid into a hypercube of dimension $\log n$, the same estimate as given above for g_{min} holds for h_{min}, the minimum cost hypercube embedding, namely,

$$h_{min} \approx \frac{nd}{2}\frac{r_d\sqrt{n}}{\sqrt{2}} = \frac{n}{\sqrt{2}}\left(\frac{d}{2}\right)^{3/2} \tag{3}$$

whereas h_r, the average cost of a random hypercube embedding of $G_{n,d}$ can be estimated by

$$h_r = \frac{nd}{4}\log n \tag{4}$$

since the average length of an edge with arbitrary endpoints is $(\log n)/2$. As seen in Table 8, there is an exact match with the column labeled R.

The above estimates indicate that, for geometric graphs with increasing size, the ratio of minimum-cost embeddings to random embeddings decreases asymptotically to 0.

The bisection width b of a random geometric graph $G_{n,d}$ of degree d can be estimated by considering the average number of edges which cross a vertical line l cutting the grid into two approximately equal sections. With the Manhattan metric the points adjacent to a given point a_i fall into a diamond-shaped region $D_{r_d}(a_i)$ of side $\sqrt{2}r_d$. The edge between two points a_i and a_j cuts the line l if it intersects $D_{r_d}(a_i)$. It follows from a straightforward analysis that on average the probability that a pair of vertices are adjacent and their incident edge crosses the line is $2r_d^3/3$. Since there are $n(n-1)/2$ such pairs, it follows that b_{min}, the average number of edges crossing the vertical line, satisfies

$$b_{min} \leq \frac{(n-1)d}{2}\frac{\sqrt{d/2}}{3\sqrt{n}} \tag{5}$$

In a random embedding of a random graph $G_{n,d}$ half the edges cross the cut, so the average bisection width b_r of a random graph is

$$b_r = \frac{nd}{2} \tag{6}$$

These estimates, normalized by the number of edges, are used below to evaluate the quality of the experimental data.

The columns labeled *Slice* in Tables 8 and 9 show results for the Slice heuristic, introduced below, that are close to the above estimates. (The results presented in the tables are expressed as average edge lengths, and must be multiplied by the number of edges $nd/2$ to give total embedding costs.)

4.3 The Slice Heuristic

Intuitively, if every randomly generated vertex of $G_{n,d}$ on the unit square were shifted by a small distance so that the point occupied a unique grid location, the resulting grid embedding should be quite good, and certainly better than a randomly generated mapping of vertices to the grid. Such a heuristic can serve both as a starting solution for the *Mob* heuristic and as a reference point to observe *Mob*'s convergence from a random solution.

The *Slice* heuristic presented here is a divide-and-conquer algorithm to find unique vertex to grid mappings by slightly displacing the vertices on the unit square. The Slice heuristic is closely related to the slicing structure tree introduced by Otten [17] for VLSI floorplan design. The vertices are sorted along the x- or the y-dimension. The sorted vertices are divided into two sets, which are mapped to different halves of the grid. Each set is now sorted along the other dimension. The procedure of sorting along alternating dimensions and halving the sets is repeated until the sets are of size one. At this point every vertex has a unique grid node assigned to it.

Johnson *et al* [13] used a similar approach in designing their LINE heuristic for partitioning geometric graphs: the unit square is cut into half to obtain a graph partition. They report that local search algorithms do not converge quickly on geometric graphs, local search algorithms need considerable running time to equal the performance of LINE, and LINE followed by local search produced the best results.

Since x, y-coordinates are usually not part of the input to a graph-embedding problem, *Slice* is definitely not a practical graph-embedding heuristic. We present its results here since we believe it produces solutions very close to the optimal embedding, thus allowing us to evaluate the performance of the *Mob* heuristic.

By itself, the knowledge that a graph G is a random geometric graph seems not to be very helpful. A heuristic is required to construct approximate x- and y-coordinates of G's vertices in the unit square. Unfortunately, the best candidate for doing so is a grid-embedding heuristic.

4.4 Experiments on Large Geometric Graphs

We evaluated the performance of the *Mob* hypercube and grid-embedding algorithms for geometric graphs with up to 256K vertices and 512K edges by conducting experiments on the CM-2. Again, both 1-to-1 and 16-to-1 mappings were studied. The exact number of edges, vertices and average degree of the random graphs used in our experiments are given in Table 7. We generated five graphs of average degree 4 and five graphs of average degree 8 to study the effect of increasing graph size on solution quality and running time. We performed experiments on nine graphs with 16K vertices and average degrees ranging from 3 to 16 to examine the effect of graph degree on the behavior of the *Mob* algorithms. At least five runs were performed for each embedding. The mob schedule used was the same as for random

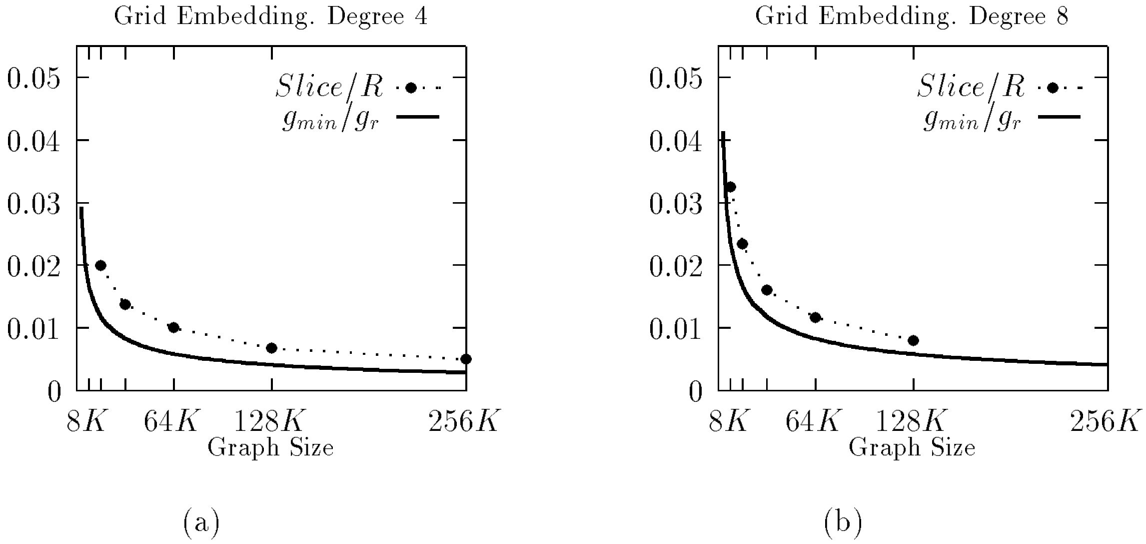


Figure 7: Comparison of g_{min}/g_r with $Slice/R$ for (a) degree 4 and degree 8 geometric graphs embedded in grids.

graphs, and is given in Section 3. *Mob* was always stopped after 8000 iterations.

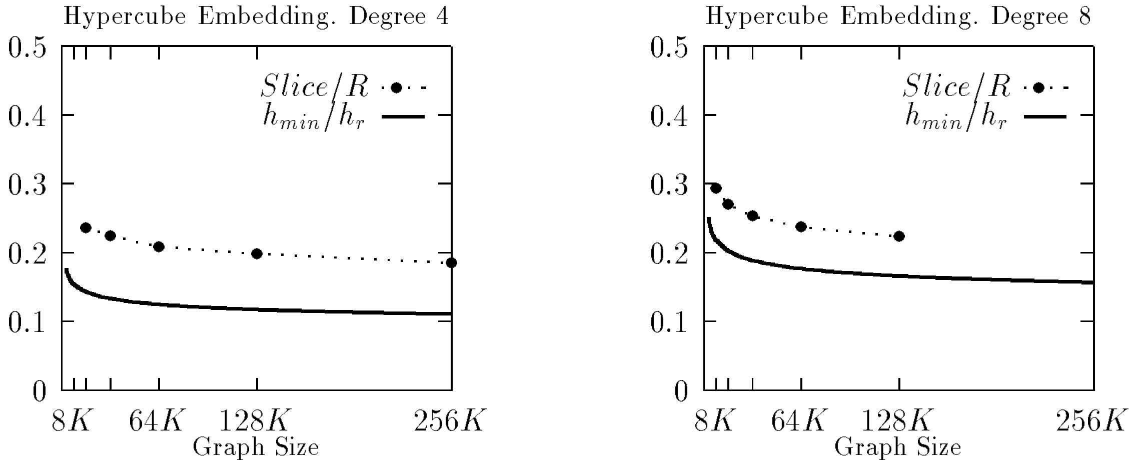
The computation time needed for one iteration of Mob is the same as for random graphs, so Tables 4 and 5 also apply to geometric graphs. We shall see that while *Mob* with a constant number of iterations does not produce embeddings that are close to optimal, the reduction of average edge lengths is larger than for random graphs.

4.5 Solution Quality of *Mob*

The quality of the solutions produced by *Mob* is shown in Tables 8 and 9 for 1-to-1 and 16-to-1 embeddings. Table 8 gives results for hypercube embeddings, Table 9 gives results for grid embeddings. The columns labeled *Dim* give the dimension of the hypercube in which the graph is embedded. For the grid embeddings, *Dim* is the dimension of the hypercube containing a $2^{\lfloor Dim/2 \rfloor} \times 2^{\lceil Dim/2 \rceil}$ grid. The columns labeled R give the average edge length produced by a random embedding, those labeled Slice give the average edge length produced by the Slice heuristic, and those labeled *Mob* give the average edge length in an embedding produced by *Mob* from an initial random embedding, each normalized by the number of edges $nd/2$ in a graph, where d is the average degree of a graph. The columns labeled $Slice/R$ and Mob/R give the ratio of improvement produced by Slice and *Mob* over a random solution.

Shown in Figures 7 and 8 are the ratios g_{min}/g_r and h_{min}/h_r for degree 4 and 8 random geometric graphs compared to the ratio of $Slice/R$.

Our experiments show that:



(a) (b)

Figure 8: Comparison of h_{min}/h_r with $Slice/R$ for (a) degree 4 and degree 8 geometric graphs embedded in hypercubes.

- The Slice heuristic produces comparable or slightly better results than *Mob* for hypercube embeddings, and considerably better results than *Mob* for grid embeddings.
- For fixed degree d, Mob/R changes slowly with graph size; $Slice/R$ decreases mildly with increasing graph size for hypercubes but strongly for grids. Thus, the gap between *Slice* and *Mob* widens as graphs become larger.
- 16-to-1 mappings give better Mob/R ratios than 1-to-1 mappings.
- The grid-embedding *Mob* heuristic achieves significantly lower Mob/R ratios than the hypercube *Mob* heuristic.
- The ratio Mob/R rises with increasing average graph degree. The differences between grid and hypercube embeddings and between 1-to-1 and 16-to-1 embeddings become smaller with increasing graph degree.
- The ratio Mob/R is smaller for geometric graphs than for random graphs. (Compare Tables 2, 3 to Tables 8, 9). For 16-to-1 and 1-to-1 grid embeddings and for 1-to-1 hypercube embeddings, Mob/R is about four to eight times smaller. For 1-to-1 hypercube embeddings, the ratio Mob/R for geometric graphs is about 0.5 to 0.6 of the ratio Mob/R for random graphs. Although *Mob* with a constant number of iterations does not produce embeddings that are close to optimal, the reduction of average edge lengths is larger than for random graphs.

4.6 Rates of Convergence of *Mob*

Tables 8 and 9 also report the convergence of *Mob* for increasing number of iterations. The columns labeled Iterations show the average embedding cost as percentages above best-ever embedding cost, produced after 100, 1000, and 4000 iterations of *Mob*, respectively. Our experiments for geometric graphs indicate that *Mob* still converges rapidly towards a solution that is good compared to the best-ever solution produced by *Mob*. However, as can be inferred from the Iterations columns, the cost ratio of a *Mob* solution divided by the solution produced by Slice generally increases with increasing graph size.

4.7 Comparison to Graph-partitioning Algorithms

To calibrate our results, we again measured the graph partitions produced by the hypercube and grid embeddings, as described in Section 3. Table 10 shows how the *Mob* hypercube and grid embedding algorithms behaved as graph-partitioning algorithms, compared to the *Mob* and KL graph-partitioning algorithms.

The cost of the graph embeddings $P = (A, B)$ is given in Table 10 as the bisection width normalized by the number of edges $nd/2$. The *Mob* graph-partitioning heuristic produced better results than the hypercube- and grid-embedding algorithms, but it does not approach the partitions produced by Slice. At least with a constant number of iterations, the *Mob* heuristics produce embeddings in which the average bisection width as a function of graph size is constant or decreases slowly, whereas the normalized bisection width produced by Slice as a function of graph size are approximately proportional to the normalized value of the estimate b_{min} given above (see equation (5)), namely, $\sqrt{d/2}/3\sqrt{n}$.

We found that both 16-to-1 embedding algorithms and the grid 1-to-1 embedding algorithm produced good graph partitions that were larger by a factor of roughly 1.5 to 3 than the *Mob* graph-partitioning algorithm. The KL graph-partitioning algorithm, which we ran only on graphs with 32K or fewer vertices due to running-time considerations, produced results noticeably worse than these three graph embeddings. The 1-to-1 hypercube embeddings are larger by a factor of roughly 4 to 9.

4.8 Comparison to Simulated Annealing

Analogous to the experiments with random graphs, we compared the performance of the *Mob* cube-embedding heuristic on geometric graphs to the results reported for simulated annealing by Chen *et al* [6,7]. To duplicate their experiments, we generated 10 random geometric graphs with our own generator. Each graph had 128 vertices. The distance parameter for the infinity metric was set to $r_d = \sqrt{d/4n} = 0.117386$ to obtain graphs of degree 7.

Five runs were performed for each graph with the infinity metric and the results were averaged over these runs. Each run of *Mob* was limited by 8000 iterations, and a schedule as described above was used. The results, expressed as percent reduction in edge lengths of a random embedding, are given in Table 11. Chen *et al* report that on ten random geometric graphs of average degree 7, a reduction of 48.0% was achieved by SAC, a version of SA with the move set limited to hypercube edges. We found that the average reduction produced by *Mob* is 49.5% for our own graphs. The Slice heuristic produced solutions with 52.1% reductions. The best solutions were obtained when the solutions produced by Slice were further improved by *Mob*.

5 Conclusions

We have developed a new *Mob*-based heuristic that exploits parallelism and gives high-quality embeddings of low degree graphs in grids and hypercubes. It is closely related to both the Kernighan-Lin and simulated annealing heuristics. We have implemented our *Mob* heuristic on the CM-2 Connection Machine to demonstrate that it is fast and can handle very large graphs. The speed of the *Mob* heuristic should be adequate for the optimized placement of large (100,000 to 1,000,000 gates) VLSI circuits. It can also be applied to other optimization problems in which local search heuristics have been successful. It would be interesting to see the *Mob* heuristic used in an industrial production system for VLSI gate-array placement or full custom logic placement. *Mob*'s speed and its ability to handle unusually large problem sizes could reduce design time by several orders of magnitude and would allow the creation of new tools to handle larger problem instances and return higher-quality solutions.

Acknowledgements The *Mob* heuristic was run on the Internet CM-2 facility supported by DARPA. Many thanks to the staff of Thinking Machines Corporation and especially to Denny Dahl, David Ray and Jim Reardon for their fast and knowledgeable help. We would like to thank Woei-Kae Chen and Matthias Stallmann for supplying graphs against which the performance of our hypercube-embedding heuristic was compared.

Bibliography

[1] "Connection Machine Model CM-2 Technical Summary," Thinking Machines Corporation TMC Technical Report HA 87-4, 1987.

[2] P. Banerjee, M. H. Jones, and J. S. Sargent, "Parallel Simulated Annealing Algorithms for Cell Placement on Hypercube Multiprocessors," *IEEE Transactions on Parallel and Distributed Systems* PDS-1 (January 1990), 91–106.

[3] G. E. Blelloch, "Scans as Primitive Parallel Operations," *IEEE Transactions on Computers* 38 (1989), 1526–1538.

[4] S. H. Bokhari, "On the Mapping Problem," *IEEE Transactions on Computers* C-30 (Mar. 1981), 207–214.

[5] A. Casotto and A. Sangiovanni-Vincentelli, "Placement of Standard Cells Using Simulated Annealing on the Connection Machine," *ICCAD* (Nov. 1987), 350–453.

[6] W. -K. Chen, E. F. Gehringer, and M. F. M. Stallmann, "Hypercube Embedding Heuristics: An Evaluation," *International Journal of Parallel Programming* 18 (1989), 505–549.

[7] W. -K. Chen and M. F. M. Stallmann, "Local Search Variants for Hypercube Embedding," *Proceedings 5th Distributed Memory Computer Conference* (1990), 1375–1383.

[8] E. D. Dahl, "Mapping and Compiled Communication on the Connection Machine," *Proceedings 5th Distributed Memory Computer Conference* (1990), 756–766.

[9] F. Darema, S. Kirkpatrick, and V. A. Norton, "Parallel Algorithms for Chip Placement by Simulated Annealing," *IBM Journal of Research and Development* 31 (May 1987), 391–401.

[10] F. Ercal, J. Ramanujam, and P. Sadayappan, "Task Allocation onto a Hypercube by Recursive Mincut Bipartitioning," *Journal of Parallel and Distributed Computing* 10 (1990), 35–44.

[11] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, Freeman, New York, 1979.

[12] W. D. Hillis, *The Connection Machine*, MIT Press, 1985.

[13] D. S. Johnson, C. A. Aragon, L. A. McGeoch, and C. Schevon, "Optimization by Simulated Annealing: An Experimental Evaluation (Part 1)," *Operations Research* 37 (Nov.-Dec. 1989), 865–892.

[14] B. W. Kernighan and S. Lin, "An Efficient Heuristic Procedure for Partitioning Graphs," *AT&T Bell Labs. Tech. J.* 49 (Feb. 1970), 291–307.

[15] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by Simulated Annealing," *Science* 220 (May 1983), 671–680.

[16] S. Kravitz and R. Rutenbar, "Multiprocessor-based Placement by Simulated Annealing," *23rd IEEE Design Automation Conf.* (1986), 567–573.

[17] R. H. J. M. Otten, "Automatic Floorplan Design," *Proc. 19th IEEE Design Automation Conf.* (1982), 261–267.

[18] P. Roussel-Ragot and G. Dreyfus, "A Problem Independent Parallel Implementation of Simulated Annealing: Models and Experiments," *IEEE Trans. Computer-Aided Design* 9 (Aug. 1990), 827–835.

[19] J. E. Savage and M. G. Wloka, "On Parallelizing Graph-Partitioning Heuristics," *Proceedings of the ICALP'90* (July 1990), 476–489.

[20] J. E. Savage and M. G. Wloka, "Parallel Graph-Embedding Heuristics," *5th SIAM Conference on Parallel Processing for Scientific Computing* (Mar. 1991), 472–47.

[21] J. E. Savage and M. G. Wloka, "Parallelism in Graph-Partitioning," *Journal of Parallel and Distributed Computing* 13 (Nov. 1991), 257–272.

[22] C. Wong and R. Fiebrich, "Simulated Annealing-Based Circuit Placement on the Connection Machine," *Proceedings of the International Conference on Computer Design* (Oct. 1987), 78–82.

Table 1: Large random graphs of small degree.

Graph	Vertices	Edges	Degree
Degree 4			
4.16K	16,384	32,699	3.94
4.32K	32,768	12,996	3.97
4.64K	65,536	129,996	3.97
4.128K	131,072	259,898	3.97
4.256K	262,144	519,996	3.97
4.512K	524,288	1,039,999	3.97
Degree 8			
8.8K	8,192	31,997	7.81
8.16K	16,384	64,995	7.93
8.32K	32,786	129,994	7.93
8.64K	65,536	259,997	7.93
8.128K	131,072	519,996	7.93
8.256K	262,144	1,039,998	7.93
Variable Degree			
3.16K	16,384	24,574	2.99
4.16K	16,384	32,699	3.94
5.16K	16,384	40,923	4.99
6.16K	16,384	49,093	5.99
7.16K	16,384	57,199	6.98
8.16K	16,384	64,995	7.93
9.16K	16,384	73,693	8.99
10.16K	16,384	81,918	9.99
16.16K	16,384	130,996	15.99

Table 2: Hypercube-embedding results for large random graphs. The costs of the *Mob* hypercube-embedding algorithm, expressed as average edge length, are compared to a hypercube embedding chosen at random. Convergence is measured by expressing *Mob*'s cost after a number of iterations as a percentage over the best solution obtained.

	16-to-1 mappings							Degree 4						
					Iterations							Iterations		
Graph	Dim	R	Mob	Mob/R	100	1000	4000	Dim	R	Mob	Mob/R	100	1000	4000
Degree 4														
4.16K	10	5.0	1.538	.3076	50.0	13.1	3.6	14	7.0	2.559	.3655	62.7	16.1	6.6
4.32K	11	5.5	1.720	.3127	51.3	10.6	2.2	15	7.5	2.783	.3711	62.5	16.0	3.2
4.64K	12	6.0	1.861	.3102	55.8	13.9	3.0	16	8.0	2.924	.3655	66.2	16.0	2.7
4.128K	13	6.5	2.022	.3110	68.7	12.3	4.0	17	8.5	3.111	.3660	90.8	16.2	5.0
4.256K	14	7.0	2.175	.3107	72.7	13.3	3.9	18	9.0	3.251	.3612	97.7	17.3	4.5
4.512K	15	7.5	2.369	.3159	76.6	12.6	1.8	19	9.5	3.398	.3577	98.9	19.8	4.2
Degree 8														
8.8K	9	4.5	2.207	.4904	20.6	4.9	1.0	13	6.5	3.497	.5380	28.5	6.9	1.2
8.16K	10	5.0	2.460	.4920	24.6	5.3	1.2	14	7.0	3.759	.5370	30.3	6.6	1.4
8.32K	11	5.5	2.712	.4931	25.4	4.7	0.9	15	7.5	4.007	.5343	31.8	7.1	1.6
8.64K	12	6.0	2.955	.4925	31.4	5.7	1.8	16	8.0	4.261	.5326	32.9	7.6	2.5
8.128K	13	6.5	3.201	.4925	29.6	6.6	1.7	17	8.5	4.517	.5314	33.8	8.5	2.7
8.256K	14	7.0	3.473	.4961	31.7	5.0	0.8	18	9.0	4.781	.5312	35.1	8.3	2.4
Variable Degree														
3.16K	10	5.0	1.148	.2296	79.8	19.4	5.1	14	7.0	2.123	.3033	80.3	24.2	4.5
4.16K	10	5.0	1.538	.3076	50.0	13.1	3.6	14	7.0	2.559	.3655	62.7	16.1	6.6
5.16K	10	5.0	1.874	.3748	34.9	9.4	1.7	14	7.0	2.975	.4250	47.9	11.6	4.5
6.16K	10	5.0	2.111	.4222	28.7	7.6	1.3	14	7.0	3.291	.4701	40.0	9.1	2.2
7.16K	10	5.0	2.312	.4624	25.4	6.0	1.1	14	7.0	3.553	.5075	34.5	7.6	1.5
8.16K	10	5.0	2.460	.4920	24.6	5.3	1.2	14	7.0	3.759	.5370	30.3	6.6	1.4
9.16K	10	5.0	2.607	.5214	23.8	4.6	1.1	14	7.0	3.933	.5619	27.4	6.4	1.7
10.16K	10	5.0	2.719	.5434	21.3	4.3	1.1	14	7.0	4.085	.5836	24.3	5.7	1.6
16.16K	10	5.0	3.186	.6372	13.5	2.7	0.5	14	7.0	4.666	.6666	17.0	4.5	1.0

Table 3: Grid-embedding results for large random graphs. The costs of the *Mob* grid-embedding algorithm, expressed as average edge length, are compared to a grid embedding chosen at random. Convergence is measured by expressing *Mob*'s cost after a number of iterations as a percentage over the best solution obtained.

	16-to-1 mappings							1-to-1 mappings						
					Iterations							Iterations		
Graph	Dim	R	Mob	Mob/R	100	1000	4000	Dim	R	Mob	Mob/R	100	1000	4000
Degree 4														
4.16K	10	21.3	6.142	.2884	38.7	9.6	1.7	14	85.1	26.702	.3138	54.1	16.9	4.6
4.32K	11	32.0	9.312	.2910	40.2	11.8	2.	15	128.3	40.387	.3148	55.9	16.4	4.3
4.64K	12	42.6	12.486	.2931	41.5	12.9	2.	16	170.7	55.386	.3245	59.9	17.2	4.5
4.128K	13	64.0	18.917	.2956	43.4	13.0	2.	17	256.1	81.779	.3193	59.9	18.1	3.9
4.256K	14	85.3	25.186	.2953	49.6	12.4	1.	18	341.1	108.484	.3180	65.6	17.4	4.0
4.512K	15	128.0	37.787	.2952	52.8	12.2	3.3	19	512.2	163.316	.3189	68.4	18.7	4.2
Degree 8														
8.8K	9	15.9	7.694	.4839	16.8	3.4	0.	13	64.1	31.987	.4990	24.9	7.9	2.2
8.16K	10	21.3	10.226	.4801	17.7	4.1	0.9	14	85.5	42.779	.5003	24.8	7.8	1.9
8.32K	11	32.1	15.525	.4836	18.1	4.4	0.7	15	127.9	64.102	.5012	28.4	7.9	1.7
8.64K	12	42.7	20.737	.4856	18.5	4.8	1.1	16	170.7	85.097	.4985	27.8	8.1	2.0
8.128K	13	64.0	31.063	.4854	18.5	5.2	1.3	17	256.5	127.866	.4985	29.7	9.5	4.9
8.256K	14	85.3	41.672	.4885	18.7	5.3	1.2	18	341.4	170.619	.4998	29.7	8.0	1.7
Variable Degree														
3.16K	10	21.2	4.417	.2074	61.8	15.7	3.4	14	85.1	19.946	.2344	77.6	23.7	6.3
4.16K	10	21.3	6.142	.2884	38.7	9.6	1.7	14	85.1	26.702	.3138	54.1	16.9	4.6
5.16K	10	21.3	7.688	.3609	29.2	6.8	1.0	14	85.2	32.750	.3844	39.8	12.3	3.1
6.16K	10	21.2	8.727	.4117	24.2	6.0	1.3	14	85.8	36.978	.4310	33.7	10.1	2.3
7.16K	10	21.3	9.597	.4506	20.3	4.7	1.0	14	85.3	40.232	.4717	28.4	8.6	2.3
8.16K	10	21.3	10.226	.4801	17.7	4.1	0.9	14	85.5	42.779	.5003	24.8	7.8	1.9
9.16K	10	21.3	10.934	.5133	15.7	3.6	0.7	14	85.1	45.511	.5348	22.4	6.2	1.5
10.16K	10	21.3	11.450	.5376	14.7	3.3	0.7	14	85.3	47.416	.5559	19.8	6.1	1.4
16.16K	10	21.3	13.449	.6314	9.8	2.2	0.5	14	85.3	55.061	.6455	14.8	4.2	1.0

Table 4: Hypercube timing results. Execution times were measured for 1 *Mob* iteration on an 8K, 16K, and 32K CM-2.

	16-to-1 mappings				1-to-1 mappings			
Degree 4								
Graph	Dim	8K	16K	32K	Dim	8K	16K	32K
4.16K	10	.0532	.0306	-	14	.0395	.0222	-
4.32K	11	.1089	.0547	.0314	15	.0740	.0405	.0223
4.64K	12	.2012	.1050	.0564	16	.1650	.0760	.0416
4.128K	13	.4060	.2113	.1139	17	.3073	.1572	.0799
4.256K	14	-	.4165	.2190	18	-	.3111	.1630
4.512K	15	-	-	.4302	19	-	-	.3160
Degree 8								
8.8K	9	.0417	-	-	13	.0323	-	-
8.16K	10	.0734	.0446	-	14	.0590	.0329	-
8.32K	11	.1434	.0754	.0475	15	.1133	.0602	.0353
8.64K	12	.2841	.1618	.0803	16	.2305	.1169	.0664
8.128K	13	-	.3163	.1544	17	-	.2360	.1210
8.256K	14	-	-	.3031	18	-	-	.2441
Variable Degree								
3.16K	10	.0555	.0310	-	14	.0406	.0225	-
4.16K	10	.0532	.0306	-	14	.0395	.0222	-
5.16K	10	.0712	.0462	-	14	.0610	.0346	-
6.16K	10	.0751	.0448	-	14	.0595	.0337	-
7.16K	10	.0737	.0418	-	14	.0592	.0328	-
8.16K	10	.0734	.0446	-	14	.0590	.0329	-
9.16K	10	.1157	.0648	-	14	.0974	.0540	-
10.16K	10	.1140	.0633	-	14	.0963	.0529	-
16.16K	10	.1106	.0617	-	14	.0988	.0525	-

Table 5: Grid embedding timing results. Execution times were measured for 1 *Mob* iteration on an 8K, 16K, and 32K CM-2.

	16-to-1 mappings				1-to-1 mappings			
Degree 4								
Graph	Dim	8K	16K	32K	Dim	8K	16K	32K
4.16K	10	.0726	.0428	-	14	.0640	.0355	-
4.32K	11	.1405	.0768	.0441	15	.1161	.0640	.0395
4.64K	12	.2682	.1422	.0784	16	.2252	.1152	.0691
4.128K	13	.5502	.2824	.1514	17	.4600	.2342	.1277
4.256K	14	-	.5630	.2940	18	-	.4762	.2500
4.512K	15	-	-	.5932	19	-	-	.5394
Degree 8								
8.8K	9	.0583	-	-	13	.0532	-	-
8.16K	10	.1094	.0615	-	14	.0974	.0566	-
8.32K	11	.2092	.1224	.0643	15	.1886	.1134	.0626
8.64K	12	.4127	.2232	.1151	16	.4015	.2229	.1091
8.128K	13	-	.4241	.2273	17	-	.3950	.2156
8.256K	14	-	-	.4495	18	-	-	.4246
Variable Degree								
3.16K	10	.0730	.0417	-	14	.0593	.0448	-
4.16K	10	.0726	.0428	-	14	.0640	.0355	-
5.16K	10	.1095	.0624	-	14	.0974	.0554	-
6.16K	10	.1101	.0632	-	14	.0976	.0563	-
7.16K	10	.1089	.0623	-	14	.0984	.0569	-
8.16K	10	.1094	.0615	-	14	.0974	.0566	-
9.16K	10	.1758	.1056	-	14	.1581	.0951	-
10.16K	10	.1734	.1060	-	14	.1607	.0962	-
16.16K	10	.1771	.0956	-	14	.1648	.0916	-

Table 6: Graph partitions of random graphs generated by cutting the hypercube and grid embeddings across a hyperplane. Bisection widths are normalized by the number of edges. The *Mob* hypercube and grid heuristic produce bisection widths that are better than those of the KL heuristic.

Graph	R	*Mob* Partition	KL Partition	Cube 16:1	Cube 1:1	Grid 16:1	Grid 1:1
Degree 4							
4.16K	.5	.1480	.1739	.1520	.1808	.1503	.1608
4.32K	.5	.1512	.1765	.1551	.1835	.1510	.1616
4.64K	.5	.1500	-	.1541	.1804	.1521	.1619
4.128K	.5	.1500	-	.1545	.1813	.1525	.1636
4.256K	.5	.1552	-	.1547	.1797	.1527	.1629
4.512K	.5	.1503	-	.1567	.1770	.1524	.1633
Degree 8							
8.8K	.5	.2411	.2531	.2442	.2660	.2456	.2539
8.16K	.5	.2442	.2581	.2453	.2671	.2449	.2539
8.32K	.5	.2436	.2610	.2462	.2658	.2468	.2539
8.64K	.5	.2444	-	.2458	.2652	.2484	.2539
8.128K	.5	.2442	-	.2459	.2648	.2473	.2543
8.256K	.5	.2440	-	.2478	.2650	.2476	.2543
Variable Degree							
3.16K	.5	.1080	.1384	.1135	.1485	.1096	.1200
4.16K	.5	.1480	.1739	.1520	.1808	.1503	.1608
5.16K	.5	.1838	.2073	.1863	.2103	.1858	.1951
6.16K	.5	.2085	.2290	.2100	.2330	.2095	.2220
7.16K	.5	.2278	.2485	.2303	.2521	.2295	.2408
8.16K	.5	.2442	.2581	.2453	.2671	.2449	.2539
9.16K	.5	.2585	.2730	.2596	.2794	.2611	.2701
10.16K	.5	.2694	.2847	.2710	.2910	.2729	.2814
16.16K	.5	.3155	.3261	.3181	.3323	.3187	.3258

Table 7: Large random geometric graphs of small degree.

Graph	Vertices	Edges	Degree
Degree 4			
4.16K	16,384	31,946	3.90
4.32K	32,768	64,222	3.92
4.64K	65,536	129,649	3.96
4.128K	131,072	258,561	3.95
4.256K	262,144	517,080	3.95
Degree 8			
8.8K	8,192	32,233	7.87
8.16K	16,384	64,515	7.88
8.32K	32,786	129,661	7.91
8.64K	65,536	259,982	7.93
8.128K	131,072	520,719	7.95
Variable Degree			
3.16K	16,384	24,327	2.97
4.16K	16,384	31,946	3.90
5.16K	16,384	40,542	4.95
6.16K	16,384	49,868	6.09
7.16K	16,384	56,557	6.90
8.16K	16,384	64,515	7.88
9.16K	16,384	72,397	8.84
10.16K	16,384	80,549	9.83
16.16K	16,384	130,514	15.93

Table 8: Hypercube-embedding results for large geometric graphs. The cost of the Slice and *Mob* hypercube-embedding algorithms, expressed as average edge length, are compared to a hypercube embedding chosen at random. Convergence is measured by expressing *Mob*'s cost after a number of iterations as a percentage over the best solution obtained.

	16-to-1 mappings								
							Iterations		
Graph	Dim	R	Slice	Slice/R	Mob	Mob/R	100	1000	4000
Degree 4									
4.16K	10	5.0	0.298	0.0596	0.249	0.0498	283.1	60.2	12.7
4.32K	11	5.5	0.318	0.0578	0.274	0.0498	304.6	52.6	10.7
4.64K	12	6.0	0.305	0.0508	0.298	0.0497	314.6	53.2	12.1
4.128K	13	6.5	0.320	0.0492	0.317	0.0488	349.4	60.7	22.1
4.256K	14	7.0	0.306	0.0437	0.338	0.0483	422.5	90.5	33.8
Degree 8									
8.8K	9	4.5	0.402	0.0893	0.419	0.0931	147.7	29.2	12.3
8.16K	10	5.0	0.404	0.0808	0.465	0.0930	197.8	47.3	22.9
8.32K	11	5.5	0.426	0.0774	0.499	0.0907	230.6	64.3	25.1
8.64K	12	6.0	0.404	0.0673	0.536	0.0893	306.7	84.1	41.7
8.128K	13	6.5	0.422	0.0649	0.573	0.0881	339.3	91.5	47.8
Variable Degree									
3.16K	10	5.0	0.273	0.0546	0.170	0.0340	356.0	70.3	12.7
4.16K	10	5.0	0.298	0.0596	0.249	0.0498	283.1	60.2	12.7
5.16K	10	5.0	0.322	0.0644	0.304	0.0608	242.0	53.9	11.1
6.16K	10	5.0	0.364	0.0728	0.389	0.0778	203.8	44.2	15.3
7.16K	10	5.0	0.382	0.0764	0.434	0.0868	205.6	47.7	21.0
8.16K	10	5.0	0.404	0.0808	0.465	0.0930	197.8	47.3	22.9
9.16K	10	5.0	0.417	0.0834	0.507	0.1014	208.1	52.4	29.5
10.16K	10	5.0	0.440	0.0880	0.529	0.1058	195.3	53.7	29.7
16.16K	10	5.0	0.535	1.1000	0.668	0.1336	167.7	54.3	32.3

	1-to-1 mappings								
							Iterations		
Graph	Dim	R	Slice	Slice/R	*Mob*	*Mob*/*R*	100	1000	4000
Degree 4									
4.16K	14	7.0	1.661	0.2372	1.719	0.2456	68.4	17.1	6.7
4.32K	15	7.5	1.684	0.2245	1.747	0.2329	79.5	22.4	7.3
4.64K	16	8.0	1.676	0.2095	1.783	0.2229	90.8	27.1	10.3
4.128K	17	8.5	1.689	0.1986	1.808	0.2207	104.2	28.1	13.7
4.256K	18	9.0	1.676	0.1863	1.838	0.2042	120.1	32.1	17.4
Degree 8									
8.8K	13	6.5	1.910	0.2938	2.069	0.3183	56.1	18.9	11.7
8.16K	14	7.0	1.893	0.2704	2.106	0.3009	72.3	24.3	14.7
8.32K	15	7.5	1.904	0.2539	2.140	0.2853	79.4	30.6	15.8
8.64K	16	8.0	1.904	0.2380	2.174	0.2718	94.3	35.7	17.9
8.128K	17	8.5	1.905	0.2242	2.213	0.2603	106.6	39.1	22.3
Variable Degree									
3.16K	14	7.0	1.600	0.2286	1.565	0.2236	70.7	11.7	3.3
4.16K	14	7.0	1.661	0.2373	1.719	0.2456	68.4	17.1	6.7
5.16K	14	7.0	1.737	0.2481	1.867	0.2667	66.4	20.2	10.7
6.16K	14	7.0	1.811	0.2587	1.979	0.2827	69.4	23.1	12.6
7.16K	14	7.0	1.850	0.2643	2.045	0.2921	71.4	23.5	13.8
8.16K	14	7.0	1.893	0.2704	2.106	0.3009	72.3	24.3	14.7
9.16K	14	7.0	1.944	0.2777	2.171	0.3101	68.5	24.4	14.8
10.16K	14	7.0	1.987	0.2839	2.219	0.3170	65.7	23.8	14.8
16.16K	14	7.0	2.201	0.3144	2.468	0.3526	60.3	22.1	14.8

Table 9: Grid-embedding results for large geometric graphs. The costs of the Slice and *Mob* grid-embedding algorithms, expressed as average edge length, are compared to a grid embedding chosen at random. Convergence is measured by expressing *Mob*'s cost after a number of iterations as a percentage over the best solution obtained.

	16-to-1 mappings								
							Iterations		
Graph	Dim	R	Slice	Slice/R	Mob	Mob/R	100	1000	4000
Degree 4									
4.16K	10	21.280	0.298	0.0140	0.797	0.0375	958.8	347.5	210.9
4.32K	11	31.955	0.318	0.0099	1.142	0.0357	1575.3	580.1	324.3
4.64K	12	42.676	0.305	0.0071	1.780	0.0417	2149.3	874.3	580.9
4.128K	13	64.009	0.320	0.0050	2.627	0.0410	3325.8	1299.4	857.7
4.256K	14	85.342	0.306	0.0036	3.479	0.0408	5273.8	1847.6	1228.0
Degree 8									
8.8K	9	15.998	0.402	0.0251	1.142	0.0714	708.9	282.1	202.5
8.16K	10	21.296	0.404	0.0190	1.435	0.0674	1029.4	455.7	301.7
8.32K	11	31.998	0.426	0.0133	2.301	0.0720	1624.5	746.5	527.0
8.64K	12	42.652	0.404	0.0095	3.089	0.0724	2334.1	1105.5	771.3
8.128K	13	63.970	0.422	0.0066	4.651	0.0727	3509.2	1604.7	1142.4
Variable Degree									
3.16K	10	21.281	0.273	0.0128	0.489	0.0230	966.5	239.1	121.0
4.16K	10	21.280	0.298	0.0140	0.797	0.0375	958.8	347.5	210.9
5.16K	10	21.311	0.322	0.0151	1.011	0.0474	1049.5	391.3	260.2
6.16K	10	21.231	0.364	0.0171	1.238	0.0583	1040.0	419.2	280.1
7.16K	10	21.293	0.382	0.0180	1.373	0.0645	1048.1	442.0	299.6
8.16K	10	21.296	0.404	0.0190	1.435	0.0674	1029.4	455.7	301.7
9.16K	10	21.312	0.417	0.0200	1.616	0.0758	967.1	466.1	329.5
10.16K	10	21.311	0.440	0.0206	1.595	0.0749	1017.7	440.5	310.8
16.16K	10	21.319	0.535	0.0251	1.894	0.0889	910.4	431.8	311.4

	1-to-1 mappings								
							Iterations		
Graph	Dim	R	Slice	Slice/R	Mob	Mob/R	100	1000	4000
Degree 4									
4.16K	14	85.374	1.708	0.0200	6.510	0.0763	1288.6	510.1	344.4
4.32K	15	127.980	1.765	0.0138	9.054	0.0707	2021.3	771.4	500.8
4.64K	16	170.618	1.728	0.0101	11.236	0.0659	3043.7	1084.0	674.2
4.128K	17	256.180	1.771	0.0069	15.903	0.0621	4321.8	1476.5	977.0
4.256K	18	341.428	1.728	0.0051	20.504	0.0601	6435.3	2572.7	1321.4
Degree 8									
8.8K	13	64.153	2.087	0.0326	7.169	0.1117	936.5	433.0	285.7
8.16K	14	85.288	2.008	0.0235	9.057	0.1062	1450.0	637.5	415.1
8.32K	15	128.052	2.079	0.0162	12.744	0.0995	2173.8	938.4	609.7
8.64K	16	170.595	2.021	0.0118	16.852	0.0990	3019.9	1246.4	859.9
8.128K	17	255.985	2.080	0.0081	25.198	0.0984	4576.2	1936.5	1296.7
Variable Degree									
3.16K	14	85.296	1.639	0.0192	5.011	0.0587	1170.5	426.1	265.7
4.16K	14	85.374	1.708	0.0200	6.510	0.0763	1288.6	510.1	344.4
5.16K	14	85.324	1.803	0.0211	7.402	0.0868	1422.9	578.3	381.6
6.16K	14	85.251	1.897	0.0220	8.196	0.0961	1434.9	604.2	402.8
7.16K	14	85.365	1.947	0.0228	8.449	0.0990	1464.1	618.5	413.0
8.16K	14	85.288	2.008	0.0235	9.057	0.1062	1450.0	637.5	415.1
9.16K	14	85.460	2.072	0.0242	9.453	0.1106	1431.7	624.5	425.2
10.16K	14	85.469	2.137	0.0250	9.259	0.1083	1417.5	611.7	407.0
16.16K	14	85.376	2.475	0.0290	10.199	0.1195	1219.6	593.0	372.0

Table 10: Graph partitions of geometric graphs generated by cutting the hypercube and grid embeddings across a hyperplane. Bisection widths are normalized by the number of edges. The *Mob* hypercube and grid heuristic produce bisection widths comparable to those of the *Mob* graph-partitioning heuristic and better than those of the KL heuristic.

Graph	R	Slice	Mob Partition	KL Partition	Cube 16:1	Cube 1:1	Grid 16:1	Grid 1:1
Degree 4								
4.16K	.5	0.0031	0.0093	0.0376	0.0230	0.1174	0.0166	0.0291
4.32K	.5	0.0027	0.0130	0.0421	0.0238	0.1081	0.0156	0.0284
4.64K	.5	0.0014	0.0143	0.0409	0.0235	0.1053	0.0186	0.0266
4.128K	.5	0.0015	0.0146	-	0.0238	0.1004	0.0185	0.0257
4.256K	.5	0.0009	0.0116	-	0.0236	0.0974	0.0188	0.0243
Degree 8								
8.8K	.5	0.0066	0.0204	0.0438	0.0422	0.1472	0.0300	0.0420
8.16K	.5	0.0047	0.0177	0.0476	0.0433	0.1423	0.0284	0.0422
8.32K	.5	0.0037	0.0171	0.0463	0.0431	0.1382	0.0300	0.0390
8.64K	.5	0.0026	0.0228	-	0.0431	0.1296	0.0297	0.0401
8.128K	.5	0.0016	0.0196	-	0.0428	0.1250	0.0322	0.0413
Variable Degree								
3.16K	.5	0.0022	0.0077	0.0293	0.0146	0.1028	0.0091	0.0225
4.16K	.5	0.0031	0.0093	0.0376	0.0230	0.1174	0.0166	0.0291
5.16K	.5	0.0033	0.0162	0.0464	0.0288	0.1253	0.0203	0.0339
6.16K	.5	0.0050	0.0139	0.0496	0.0365	0.1304	0.0257	0.0386
7.16K	.5	0.0049	0.0154	0.0565	0.0414	0.1360	0.0248	0.0423
8.16K	.5	0.0047	0.0177	0.0476	0.0433	0.1423	0.0275	0.0407
9.16K	.5	0.0042	0.0103	0.0517	0.0459	0.1441	0.0284	0.0422
10.16K	.5	0.0059	0.0213	0.0477	0.0483	0.1502	0.0320	0.0445
16.16K	.5	0.0072	0.0175	0.0542	0.0579	0.1597	0.0335	0.0480

Table 11: Hypercube embeddings of 128-vertex, degree-7 geometric graphs. Comparison of *Mob* to SA.

Heuristic	Min	Average Cost	Average. Edge Length	% of Random
Mob + Slice	620	676.0	1.694	47.7
Mob	649	702.7	1.758	49.5
Slice	664	737.6	1.849	52.1
Random	1320	1410.1	3.552	100.0
SAC (pushed)	694	742.8	1.691	48.0